
The scikit-image Documentation

Release 0.21.0

scikit-image development team

Jul 22, 2023

CONTENTS

Date: Jul 22, 2023, **Version:** 0.21.0

Welcome! `scikit-image` is an image processing toolbox which builds on `numpy`, `scipy.ndimage` and other libraries to provide a versatile set of image processing routines in `Python`. Our project and community is guided by the *scikit-image Code of Conduct*.

QUICK LINKS

Get started New to scikit-image? Start with our installation guide and scikit-image’s key concepts.

Examples Browse our gallery of entry-level and domain-specific examples.

Get help Need help with a particular image analysis task? Ask us and a large image analysis community on our user forum *image.sc*.

API reference A detailed description of scikit-image’s public Python API. Assumes an understanding of the key concepts.

Contribute Saw a typo? Found a bug? Want to improve a function? Learn how to contribute to scikit-image!

Release notes Upgrading from a previous version? See what’s new and changed between each release of scikit-image.

About us Get to know the project and the community. Learn where we are going and how we work together.

SKIPs scikit-image proposals, documents describing major changes to the library.

See also our site-wide genindex our search this documentation.

1.1 User guide

Here you can find our narrative documentation, learn about scikit-image’s key concepts and more advanced topics.

1.1.1 Installing scikit-image

How you should install scikit-image depends on your needs and skills:

- Simplest solution: *scientific Python distribution*.
- If you can install Python packages and work in virtual environments:
 - *pip*
 - *conda*
- Easy solution but with pitfalls: *system package manager* (yum, apt, …).
- *You’re looking to contribute to scikit-image.*

Supported platforms

- Windows 64-bit on x86 processors
- macOS on x86 and M (ARM) processors
- Linux 64-bit on x86 processors

While we do not officially support other platforms, you could still try *building from source*.

Version check

To see whether scikit-image is already installed or to check if an install has worked, run the following in a Python shell or Jupyter notebook:

```
import skimage
print(skimage.__version__)
```

or, from the command line:

```
python -c "import skimage; print(skimage.__version__)"
```

(Try python3 if python is unsuccessful.)

You'll see the version number if scikit-image is installed and an error message otherwise.

Installation via pip and conda

These install only scikit-image and its dependencies; pip has an option to include related packages.

pip

Prerequisites to a pip install: You're able to use your system's command line to install packages and are using a [virtual environment](#) (any of [several](#)).

While it is possible to use pip without a virtual environment, it is not advised: virtual environments create a clean Python environment that does not interfere with any existing system installation, can be easily removed, and contain only the package versions your application needs. They help avoid a common challenge known as [dependency hell](#).

To install the current scikit-image you'll need at least Python 3.6. If your Python is older, pip will find the most recent compatible version.

```
# Update pip
python -m pip install -U pip
# Install scikit-image
python -m pip install -U scikit-image
```

To access the full selection of demo datasets, use `scikit-image[data]`. To include a selection of other scientific Python packages that expand scikit-image's capabilities to include, e.g., parallel processing, you can install the package `scikit-image[optional]`:

```
python -m pip install -U scikit-image[optional]
```

Warning: Please do not use the command sudo and pip together as pip may overwrite critical system libraries which may require you to reinstall your operating system.

conda

Miniconda is a bare-essentials version of the Anaconda package; you'll need to install packages like `scikit-image` yourself. Like Anaconda, it installs Python and provides virtual environments.

- [conda documentation](#)
- [Miniconda](#)
- [conda-forge](#), a conda channel maintained with the latest `scikit-image` package

Once you have your conda environment set up, you can install `scikit-image` with the command:

```
conda install scikit-image
```

System package managers

Using a package manager (`yum`, `apt-get`, etc.) to install `scikit-image` or other Python packages is not your best option:

- You're likely to get an older version.
- You'll probably want to make updates and add new packages outside of the package manager, leaving you with the same kind of dependency conflicts you see when using pip without a virtual environment.
- There's an added risk because operating systems use Python, so if you make system-wide Python changes (installing as root or using sudo), you can break the operating system.

Downloading all demo datasets

Some of the data used in our examples is hosted online and is not installed by default by the procedures explained above. Data are downloaded once, at the first call, but this requires an internet connection. If you prefer downloading all the demo datasets to be able to work offline, ensure that package `pooch` is installed and then run this command:

```
python -c 'from skimage.data import download_all; download_all()'
```

or call `download_all()` in your favourite interactive Python environment (IPython, Jupyter notebook, ...).

Additional help

If you still have questions, reach out through

- our [user forum](#)
- our [developer forum](#)
- our [chat channel](#)
- [Stack Overflow](#)

To suggest a change in these instructions, [please open an issue on GitHub](#).

1.1.2 Installing scikit-image for contributors

We are assuming that you have a default Python environment already configured on your computer and that you intend to install `scikit-image` inside of it.

We also make a few more assumptions about your system:

- You have a C compiler set up.
- You have a C++ compiler set up.
- You are running a version of Python compatible with our system as listed in our [pyproject.toml](#).
- You've cloned the git repository into a directory called `scikit-image`. You have set up the *upstream* remote to point to our repository and *origin* to point to your fork.

This directory contains the following files:

```
scikit-image
├── asv.conf.json
├── azure-pipelines.yml
├── benchmarks/
├── CITATION.bib
├── CODE_OF_CONDUCT.md
├── CONTRIBUTING.rst
├── CONTRIBUTORS.txt
├── doc/
├── INSTALL.rst
├── LICENSE.txt
├── MANIFEST.in
├── meson.build
└── meson.md
├── pyproject.toml
├── README.md
├── RELEASE.txt
└── requirements/
    └── requirements.txt
└── skimage/
└── TODO.txt
└── tools/
```

All commands below are assumed to be running from the `scikit-image` directory containing the files above.

Build environment setup

Once you've cloned your fork of the scikit-image repository, you should set up a Python development environment tailored for scikit-image. You may choose the environment manager of your choice. Here we provide instructions for two popular environment managers: `venv` (pip based) and `conda` (Anaconda or Miniconda).

venv

```
# Create a virtualenv named ``skimage-dev`` that lives outside of the repository.
# One common convention is to place it inside an ``envs`` directory under your home_
→directory:
mkdir ~/envs
python -m venv ~/envs/skimage-dev
# Activate it
# (On Windows, please use ``skimage-dev\Scripts\activate``)
source ~/envs/skimage-dev/bin/activate
# Install main development and runtime dependencies
pip install -r requirements.txt
# Install build dependencies of scikit-image
pip install -r requirements/build.txt
# Build scikit-image from source
spin build
# Test your installation
spin test
# Build docs
spin docs
# Try the new version in IPython
spin ipython
```

conda

When using conda for development, we recommend adding the conda-forge channel for the most up-to-date version of many dependencies. Some dependencies we use (for testing and documentation) are not available from the default Anaconda channel. Please follow the official [conda-forge installation instructions](#) before you get started.

```
# Create a conda environment named ``skimage-dev``
conda create --name skimage-dev
# Activate it
conda activate skimage-dev
# Install main development and runtime dependencies
conda install -c conda-forge --file requirements/default.txt
conda install -c conda-forge --file requirements/test.txt
conda install -c conda-forge pre-commit
# Install build dependencies of scikit-image
pip install -r requirements/build.txt
# Build scikit-image from source
spin build
# Test your installation
spin test
# Build docs
spin docs
# Try the new version
spin python
```

For more information about building and using the `spin` package, see `meson.md`.

Updating the installation

When updating your installation, it is often necessary to recompile submodules that have changed. Do so with the following commands:

```
# Grab the latest source
git checkout main
git pull upstream main
# Update the installation
pip install -e . -vv
```

Testing

scikit-image has an extensive test suite that ensures correct execution on your system. The test suite must pass before a pull request can be merged, and tests should be added to cover any modifications to the code base.

We use the [pytest](#) testing framework, with tests located in the various `skimage/submodule/tests` folders.

Our testing requirements are listed below:

```
asv
matplotlib>=3.5
pooch>=1.6.0
pytest>=7.0
pytest-cov>=2.11.0
pytest-localserver
pytest-faulthandler
```

Run all tests using:

```
pytest skimage
```

Or the tests for a specific submodule:

```
pytest skimage/morphology
```

Or tests from a specific file:

```
pytest skimage/morphology/tests/test_gray.py
```

Or a single test within that file:

```
pytest skimage/morphology/tests/test_gray.py::test_3d_fallback_black_tophat
```

Use `--doctest-modules` to run doctests. For example, run all tests and all doctests using:

```
pytest --doctest-modules skimage
```

Warnings during testing phase

Scikit-image tries to catch all warnings in its development builds to ensure that crucial warnings from dependencies are not missed. This might cause certain tests to fail if you are building scikit-image with versions of dependencies that were not tested at the time of the release. To disable failures on warnings, export the environment variable `SKIMAGE_TEST_STRICT_WARNINGS` with a value of `0` or `False` and run the tests:

```
export SKIMAGE_TEST_STRICT_WARNINGS=False  
pytest --pyargs skimage
```

Platform-specific notes

Windows

A run-through of the compilation process for Windows is included in our setup of [Azure Pipelines](#) (a continuous integration service).

Debian and Ubuntu

Install suitable compilers:

```
sudo apt-get install build-essential
```

Full requirements list

Build Requirements

```
# Also update `tools/pyproject.toml.in`, [build-system] -> requires  
meson>=0.13  
wheel  
setuptools>=67  
packaging>=21  
ninja  
Cython>=0.29.32  
pythran  
numpy>=1.21.1  
  
# Developer UI  
spin==0.3  
build
```

Runtime Requirements

```
numpy>=1.21.1  
scipy>=1.8  
networkx>=2.8  
pillow>=9.0.1  
imageio>=2.27  
tiff>=2022.8.12  
PyWavelets>=1.1.1  
packaging>=21  
lazy_loader>=0.2
```

Test Requirements

```
asv
matplotlib>=3.5
pooch>=1.6.0
pytest>=7.0
pytest-cov>=2.11.0
pytest-localserver
pytest-faulthandler
```

Documentation Requirements

```
sphinx>=5.0
sphinx-gallery>=0.11
numpydoc>=1.5
sphinx-copybutton
pytest-runner
matplotlib>=3.5
dask[array]>=2022.9.2
pandas>=1.5
seaborn>=0.11
pooch>=1.6
tifffile>=2022.8.12
myst-parser
ipywidgets
# Needed until https://github.com/jupyter-widgets/ipywidgets/issues/3731 is resolved
ipykernel
plotly>=5.10
kaleido
scikit-learn>=0.24.0
sphinx_design>=0.3
pydata-sphinx-theme>=0.13
```

Developer Requirements

```
pre-commit
rtoml
```

Data Requirements

The full selection of demo datasets is only available with the following installed:

```
pooch>=1.6.0
```

Optional Requirements

You can use `scikit-image` with the basic requirements listed above, but some functionality is only available with the following installed:

- **SimpleITK**

Optional I/O plugin providing a wide variety of formats. including specialized formats using in medical imaging.

- **Astropy**

Provides FITS I/O capability.

- **PyAMG**

The `pyamg` module is used for the fast `cg_mg` mode of random walker segmentation.

- **Dask**

The `dask` module is used to speed up certain functions.

```
SimpleITK
astropy>=5.0
# cloudpickle is necessary to provide the 'processes' scheduler for dask
cloudpickle>=0.2.1
dask[array]>=2021.1.0
matplotlib>=3.5
pooch>=1.6.0
pyamg
scikit-learn>=0.24.0
```

Help with contributor installation

See *Additional help* above.

1.1.3 Getting started

`scikit-image` is an image processing Python package that works with `numpy` arrays. The package is imported as `skimage`:

```
>>> import skimage
```

Most functions of `skimage` are found within submodules:

```
>>> from skimage import data
>>> camera = data.camera()
```

A list of submodules and functions is found on the [API reference](#) webpage.

Within `scikit-image`, images are represented as NumPy arrays, for example 2-D arrays for grayscale 2-D images

```
>>> type(camera)
<type 'numpy.ndarray'>
>>> # An image with 512 rows and 512 columns
>>> camera.shape
(512, 512)
```

The `skimage.data` submodule provides a set of functions returning example images, that can be used to get started quickly on using `scikit-image`'s functions:

```
>>> coins = data.coins()
>>> from skimage import filters
>>> threshold_value = filters.threshold_otsu(coins)
>>> threshold_value
107
```

Of course, it is also possible to load your own images as NumPy arrays from image files, using `skimage.io.imread()`:

```
>>> import os
>>> filename = os.path.join(skimage.data_dir, 'moon.png')
>>> from skimage import io
>>> moon = io.imread(filename)
```

Use `natsort` to load multiple images

```
>>> import os
>>> from natsort import natsorted, ns
>>> from skimage import io
>>> list_files = os.listdir('.')
>>> list_files
['01.png', '010.png', '0101.png', '0190.png', '02.png']
>>> list_files = natsorted(list_files)
>>> list_files
['01.png', '02.png', '010.png', '0101.png', '0190.png']
>>> image_list = []
>>> for filename in list_files:
...     image_list.append(io.imread(filename))
```

1.1.4 A crash course on NumPy for images

Images in scikit-image are represented by NumPy ndarrays. Hence, many common operations can be achieved using standard NumPy methods for manipulating arrays:

```
>>> from skimage import data
>>> camera = data.camera()
>>> type(camera)
<type 'numpy.ndarray'>
```

Retrieving the geometry of the image and the number of pixels:

```
>>> camera.shape
(512, 512)
>>> camera.size
262144
```

Retrieving statistical information about image intensity values:

```
>>> camera.min(), camera.max()
(0, 255)
>>> camera.mean()
118.31400299072266
```

NumPy arrays representing images can be of different integer or float numerical types. See *Image data types and what they mean* for more information about these types and how scikit-image treats them.

NumPy indexing

NumPy indexing can be used both for looking at the pixel values and to modify them:

```
>>> # Get the value of the pixel at the 10th row and 20th column
>>> camera[10, 20]
153
>>> # Set to black the pixel at the 3rd row and 10th column
>>> camera[3, 10] = 0
```

Be careful! In NumPy indexing, the first dimension (`camera.shape[0]`) corresponds to rows, while the second (`camera.shape[1]`) corresponds to columns, with the origin (`camera[0, 0]`) at the top-left corner. This matches matrix/linear algebra notation, but is in contrast to Cartesian (x, y) coordinates. See *Coordinate conventions* below for more details.

Beyond individual pixels, it is possible to access/modify values of whole sets of pixels using the different indexing capabilities of NumPy.

Slicing:

```
>>> # Set the first ten lines to "black" (0)
>>> camera[:10] = 0
```

Masking (indexing with masks of booleans):

```
>>> mask = camera < 87
>>> # Set to "white" (255) the pixels where mask is True
>>> camera[mask] = 255
```

Fancy indexing (indexing with sets of indices):

```
>>> inds_r = np.arange(len(camera))
>>> inds_c = 4 * inds_r % len(camera)
>>> camera[inds_r, inds_c] = 0
```

Masks are very useful when you need to select a set of pixels on which to perform the manipulations. The mask can be any boolean array of the same shape as the image (or a shape broadcastable to the image shape). This can be used to define a region of interest, for example, a disk:

```
>>> nrows, ncols = camera.shape
>>> row, col = np.ogrid[:nrows, :ncols]
>>> cnt_row, cnt_col = nrows / 2, ncols / 2
>>> outer_disk_mask = ((row - cnt_row)**2 + (col - cnt_col)**2 >
...                      (nrows / 2)**2)
>>> camera[outer_disk_mask] = 0
```



Boolean operations from NumPy can be used to define even more complex masks:

```
>>> lower_half = row > cnt_row
>>> lower_half_disk = np.logical_and(lower_half, outer_disk_mask)
>>> camera = data.camera()
>>> camera[lower_half_disk] = 0
```

Color images

All of the above remains true for color images. A color image is a NumPy array with an additional trailing dimension for the channels:

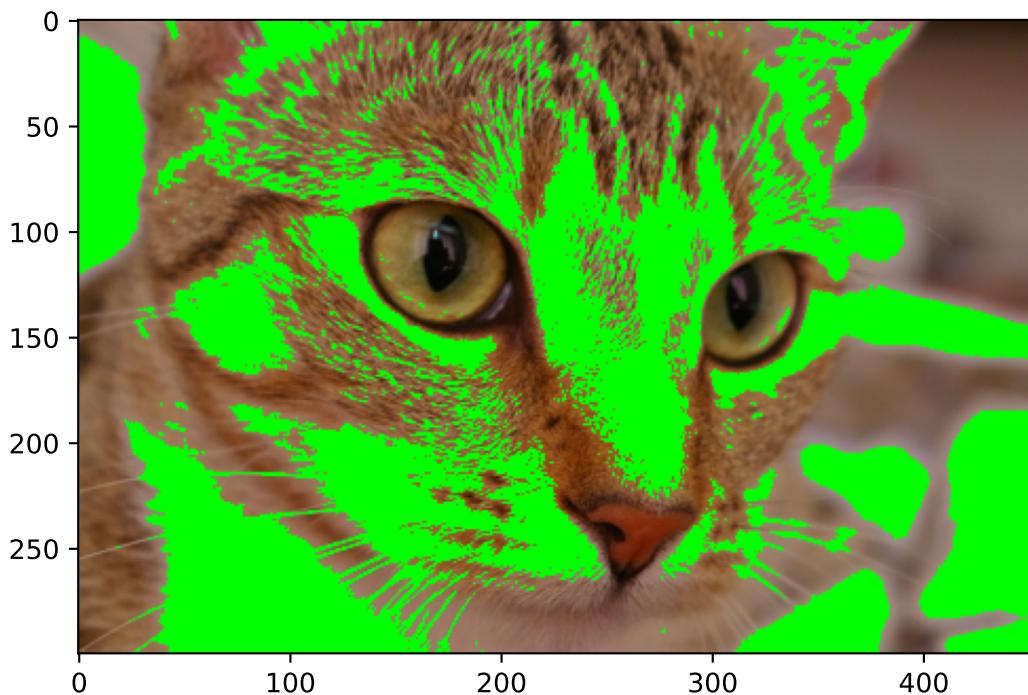
```
>>> cat = data.chelsea()
>>> type(cat)
<type 'numpy.ndarray'>
>>> cat.shape
(300, 451, 3)
```

This shows that `cat` is a 300-by-451 pixel image with three channels (red, green, and blue). As before, we can get and set the pixel values:

```
>>> cat[10, 20]
array([151, 129, 115], dtype=uint8)
>>> # Set the pixel at (50th row, 60th column) to "black"
>>> cat[50, 60] = 0
>>> # set the pixel at (50th row, 61st column) to "green"
>>> cat[50, 61] = [0, 255, 0] # [red, green, blue]
```

We can also use 2D boolean masks for 2D multichannel images, as we did with the grayscale image above:

The example color images included in `skimage.data` have channels stored along the last axis, although other software may follow different conventions. The scikit-image library functions supporting color images have a `channel_axis` argument that can be used to specify which axis of an array corresponds to channels.



Coordinate conventions

Because scikit-image represents images using NumPy arrays, the coordinate conventions must match. Two-dimensional (2D) grayscale images (such as `camera` above) are indexed by rows and columns (abbreviated to either `(row, col)` or `(r, c)`), with the lowest element `(0, 0)` at the top-left corner. In various parts of the library, you will also see `rr` and `cc` refer to lists of row and column coordinates. We distinguish this convention from `(x, y)`, which commonly denote standard Cartesian coordinates, where `x` is the horizontal coordinate, `y` - the vertical one, and the origin is at the bottom left (Matplotlib axes, for example, use this convention).

In the case of multichannel images, any dimension (array axis) can be used for color channels, and is denoted by `channel` or `ch`. Prior to scikit-image 0.19, this channel dimension was always last, but in the current release the channel dimension can be specified by a `channel_axis` argument. Functions that require multichannel data default to `channel_axis=-1`. Otherwise, functions default to `channel_axis=None`, indicating that no axis is assumed to correspond to channels.

Finally, for volumetric (3D) images, such as videos, magnetic resonance imaging (MRI) scans, confocal microscopy, etc., we refer to the leading dimension as `plane`, abbreviated as `pln` or `p`.

These conventions are summarized below:

Table 1: Dimension name and order conventions in scikit-image

Image type	Coordinates
2D grayscale	<code>(row, col)</code>
2D multichannel (eg. RGB)	<code>(row, col, ch)</code>
3D grayscale	<code>(pln, row, col)</code>
3D multichannel	<code>(pln, row, col, ch)</code>

Note that the position of `ch` is controlled by the `channel_axis` argument.

Many functions in scikit-image can operate on 3D images directly:

```
>>> rng = np.random.default_rng()
>>> im3d = rng.random((100, 1000, 1000))
>>> from skimage import morphology
>>> from scipy import ndimage as ndi
>>> seeds = ndi.label(im3d < 0.1)[0]
>>> ws = morphology.watershed(im3d, seeds)
```

In many cases, however, the third spatial dimension has lower resolution than the other two. Some scikit-image functions provide a `spacing` keyword argument to help handle this kind of data:

```
>>> from skimage import segmentation
>>> slices = segmentation.slic(im3d, spacing=[5, 1, 1], channel_axis=None)
```

Other times, the processing must be done plane-wise. When planes are stacked along the leading dimension (in agreement with our convention), the following syntax can be used:

```
>>> from skimage import filters
>>> edges = np.empty_like(im3d)
```

(continues on next page)

(continued from previous page)

```
>>> for pln, image in enumerate(im3d):
...     # Iterate over the leading dimension
...     edges[pln] = filters.sobel(image)
```

Notes on the order of array dimensions

Although the labeling of the axes might seem arbitrary, it can have a significant effect on the speed of operations. This is because modern processors never retrieve just one item from memory, but rather a whole chunk of adjacent items (an operation called prefetching). Therefore, processing of elements that are next to each other in memory is faster than processing them when they are scattered, even if the number of operations is the same:

```
>>> def in_order_multiply(arr, scalar):
...     for plane in list(range(arr.shape[0])):
...         arr[plane, :, :] *= scalar
...
>>> def out_of_order_multiply(arr, scalar):
...     for plane in list(range(arr.shape[2])):
...         arr[:, :, plane] *= scalar
...
>>> import time
>>> rng = np.random.default_rng()
>>> im3d = rng.random((100, 1024, 1024))
>>> t0 = time.time(); x = in_order_multiply(im3d, 5); t1 = time.time()
>>> print("%.2f seconds" % (t1 - t0))
0.14 seconds
>>> s0 = time.time(); x = out_of_order_multiply(im3d, 5); s1 = time.time()
>>> print("%.2f seconds" % (s1 - s0))
1.18 seconds
>>> print("Speedup: %.1fx" % ((s1 - s0) / (t1 - t0)))
Speedup: 8.6x
```

When the last/rightmost dimension becomes even larger the speedup is even more dramatic. It is worth thinking about *data locality* when developing algorithms. In particular, scikit-image uses C-contiguous arrays by default. When using nested loops, the last/rightmost dimension of the array should be in the innermost loop of the computation. In the example above, the *= numpy operator iterates over all remaining dimensions.

A note on the time dimension

Although scikit-image does not currently provide functions to work specifically with time-varying 3D data, its compatibility with NumPy arrays allows us to work quite naturally with a 5D array of the shape (t, pln, row, col, ch):

```
>>> for timepoint in image5d:
...     # Each timepoint is a 3D multichannel image
...     do_something_with(timepoint)
```

We can then supplement the above table as follows:

Table 2: Addendum to dimension names and orders in scikit-image

Image type	coordinates
2D color video	(t, row, col, ch)
3D color video	(t, pln, row, col, ch)

1.1.5 Image data types and what they mean

In skimage, images are simply numpy arrays, which support a variety of data types¹, i.e. “dtypes”. To avoid distorting image intensities (see *Rescaling intensity values*), we assume that images use the following dtype ranges:

Data type	Range
uint8	0 to 255
uint16	0 to 65535
uint32	0 to $2^{32} - 1$
float	-1 to 1 or 0 to 1
int8	-128 to 127
int16	-32768 to 32767
int32	-2^{31} to $2^{31} - 1$

Note that float images should be restricted to the range -1 to 1 even though the data type itself can exceed this range; all integer dtypes, on the other hand, have pixel intensities that can span the entire data type range. With a few exceptions, *64-bit (u)int images are not supported*.

Functions in skimage are designed so that they accept any of these dtypes, but, for efficiency, *may return an image of a different dtype* (see *Output types*). If you need a particular dtype, skimage provides utility functions that convert dtypes and properly rescale image intensities (see *Input types*). You should **never use astype** on an image, because it violates these assumptions about the dtype range:

```
>>> from skimage.util import img_as_float
>>> image = np.arange(0, 50, 10, dtype=np.uint8)
>>> print(image.astype(float)) # These float values are out of range.
[ 0.  10.  20.  30.  40.]
>>> print(img_as_float(image))
[ 0.          0.03921569  0.07843137  0.11764706  0.15686275]
```

Input types

Although we aim to preserve the data range and type of input images, functions may support only a subset of these data-types. In such a case, the input will be converted to the required type (if possible), and a warning message printed to the log if a memory copy is needed. Type requirements should be noted in the docstrings.

The following utility functions in the main package are available to developers and users:

¹ <https://docs.scipy.org/doc/numpy/user/basics.types.html>

Function name	Description
img_as_float	Convert to floating point (integer types become 64-bit floats)
img_as_ubyte	Convert to 8-bit uint.
img_as_uint	Convert to 16-bit uint.
img_as_int	Convert to 16-bit int.

These functions convert images to the desired dtype and *properly rescale their values*:

```
>>> from skimage.util import img_as_ubyte
>>> image = np.array([0, 0.5, 1], dtype=float)
>>> img_as_ubyte(image)
array([ 0, 128, 255], dtype=uint8)
```

Be careful! These conversions can result in a loss of precision, since 8 bits cannot hold the same amount of information as 64 bits:

```
>>> image = np.array([0, 0.5, 0.503, 1], dtype=float)
>>> image_as_ubyte(image)
array([ 0, 128, 128, 255], dtype=uint8)
```

Note that `img_as_float` will preserve the precision of floating point types and does not automatically rescale the range of floating point inputs.

Additionally, some functions take a `preserve_range` argument where a range conversion is convenient but not necessary. For example, interpolation in `transform.warp` requires an image of type float, which should have a range in [0, 1]. So, by default, input images will be rescaled to this range. However, in some cases, the image values represent physical measurements, such as temperature or rainfall values, that the user does not want rescaled. With `preserve_range=True`, the original range of the data will be preserved, even though the output is a float image. Users must then ensure this non-standard image is properly processed by downstream functions, which may expect an image in [0, 1]. In general, unless a function has a `preserve_range=False` keyword argument, floating point inputs will not be automatically rescaled.

```
>>> from skimage import data
>>> from skimage.transform import rescale
>>> image = data.coins()
>>> image.dtype, image.min(), image.max(), image.shape
(dtype('uint8'), 1, 252, (303, 384))
>>> rescaled = rescale(image, 0.5)
>>> (rescaled.dtype, np.round(rescaled.min(), 4),
... np.round(rescaled.max(), 4), rescaled.shape)
(dtype('float64'), 0.0147, 0.9456, (152, 192))
>>> rescaled = rescale(image, 0.5, preserve_range=True)
>>> (rescaled.dtype, np.round(rescaled.min()),
... np.round(rescaled.max()), rescaled.shape)
(dtype('float64'), 4.0, 241.0, (152, 192))
```

Output types

The output type of a function is determined by the function author and is documented for the benefit of the user. While this requires the user to explicitly convert the output to whichever format is needed, it ensures that no unnecessary data copies take place.

A user that requires a specific type of output (e.g., for display purposes), may write:

```
>>> from skimage.util import img_as_uint
>>> out = img_as_uint(sobel(image))
>>> plt.imshow(out)
```

Working with OpenCV

It is possible that you may need to use an image created using `skimage` with OpenCV or vice versa. OpenCV image data can be accessed (without copying) in NumPy (and, thus, in scikit-image). OpenCV uses BGR (instead of scikit-image's RGB) for color images, and its dtype is uint8 by default (See *Image data types and what they mean*). BGR stands for Blue Green Red.

Converting BGR to RGB or vice versa

The color images in `skimage` and OpenCV have 3 dimensions: width, height and color. RGB and BGR use the same color space, except the order of colors is reversed.

Note that in `scikit-image` we usually refer to `rows` and `columns` instead of width and height (see *Coordinate conventions*).

For an image with colors along the last axis, the following instruction effectively reverses the order of the colors, leaving the rows and columns unaffected.

```
>>> image = image[:, :, ::-1]
```

Using an image from OpenCV with `skimage`

If `cv_image` is an array of unsigned bytes, `skimage` will understand it by default. If you prefer working with floating point images, `img_as_float()` can be used to convert the image:

```
>>> from skimage.util import img_as_float
>>> image = img_as_float(any_opencv_image)
```

Using an image from `skimage` with OpenCV

The reverse can be achieved with `img_as_ubyte()`:

```
>>> from skimage.util import img_as_ubyte
>>> cv_image = img_as_ubyte(any_skimage_image)
```

Image processing pipeline

This dtype behavior allows you to string together any `skimage` function without worrying about the image dtype. On the other hand, if you want to use a custom function that requires a particular dtype, you should call one of the dtype conversion functions (here, `func1` and `func2` are `skimage` functions):

```
>>> from skimage.util import img_as_float
>>> image = img_as_float(func1(func2(image)))
>>> processed_image = custom_func(image)
```

Better yet, you can convert the image internally and use a simplified processing pipeline:

```
>>> def custom_func(image):
...     image = img_as_float(image)
...     # do something
...
>>> processed_image = custom_func(func1(func2(image)))
```

Rescaling intensity values

When possible, functions should avoid blindly stretching image intensities (e.g. rescaling a float image so that the min and max intensities are 0 and 1), since this can heavily distort an image. For example, if you're looking for bright markers in dark images, there may be an image where no markers are present; stretching its input intensity to span the full range would make background noise look like markers.

Sometimes, however, you have images that should span the entire intensity range but do not. For example, some cameras store images with 10-, 12-, or 14-bit depth per pixel. If these images are stored in an array with dtype `uint16`, then the image won't extend over the full intensity range, and thus, would appear dimmer than it should. To correct for this, you can use the `rescale_intensity` function to rescale the image so that it uses the full dtype range:

```
>>> from skimage import exposure
>>> image = exposure.rescale_intensity(img10bit, in_range=(0, 2**10 - 1))
```

Here, the `in_range` argument is set to the maximum range for a 10-bit image. By default, `rescale_intensity` stretches the values of `in_range` to match the range of the dtype. `rescale_intensity` also accepts strings as inputs to `in_range` and `out_range`, so the example above could also be written as:

```
>>> image = exposure.rescale_intensity(img10bit, in_range='uint10')
```

Note about negative values

People very often represent images in signed dtypes, even though they only manipulate the positive values of the image (e.g., using only 0-127 in an `int8` image). For this reason, conversion functions *only spread the positive values* of a signed dtype over the entire range of an unsigned dtype. In other words, negative values are clipped to 0 when converting from signed to unsigned dtypes. (Negative values are preserved when converting between signed dtypes.) To prevent this clipping behavior, you should rescale your image beforehand:

```
>>> image = exposure.rescale_intensity(img_int32, out_range=(0, 2**31 - 1))
>>> img_uint8 = img_as_ubyte(image)
```

This behavior is symmetric: The values in an unsigned dtype are spread over just the positive range of a signed dtype.

References

1.1.6 I/O Plugin Infrastructure

A plugin consists of two files, the source and the descriptor `.ini`. Let's say we'd like to provide a plugin for `imshow` using `matplotlib`. We'll call our plugin `mpl`:

```
skimage/io/_plugins/mpl.py  
skimage/io/_plugins/mpl.ini
```

The name of the `.py` and `.ini` files must correspond. Inside the `.ini` file, we give the plugin meta-data:

```
[mpl] <-- name of the plugin, may be anything  
description = Matplotlib image I/O plugin  
provides = imshow <-- a comma-separated list, one or more of  
            imshow, imsave, imread, _app_show
```

The “provides”-line lists all the functions provided by the plugin. Since our plugin provides `imshow`, we have to define it inside `mpl.py`:

```
# This is mpl.py  
  
import matplotlib.pyplot as plt  
  
def imshow(img):  
    plt.imshow(img)
```

Note that, by default, `imshow` is non-blocking, so a special function `_app_show` must be provided to block the GUI. We can modify our plugin to provide it as follows:

```
[mpl]  
provides = imshow, _app_show
```

```
# This is mpl.py  
  
import matplotlib.pyplot as plt  
  
def imshow(img):  
    plt.imshow(img)  
  
def _app_show():  
    plt.show()
```

Any plugin in the `_plugins` directory is automatically examined by `skimage.io` upon import. You may list all the plugins on your system:

```
>>> import skimage.io as io  
>>> io.find_available_plugins()  
{'gtk': ['imshow'],  
 'matplotlib': ['imshow', 'imread', 'imread_collection'],  
 'pil': ['imread', 'imsave', 'imread_collection'],  
 'test': ['imsave', 'imshow', 'imread', 'imread_collection'],}
```

or only those already loaded:

```
>>> io.find_available_plugins(loader=True)
{'matplotlib': ['imshow', 'imread', 'imread_collection'],
 'pil': ['imread', 'imsave', 'imread_collection']}
```

A plugin is loaded using the `use_plugin` command:

```
>>> import skimage.io as io
>>> io.use_plugin('pil') # Use all capabilities provided by PIL
```

or

```
>>> io.use_plugin('pil', 'imread') # Use only the imread capability of PIL
```

Note that, if more than one plugin provides certain functionality, the last plugin loaded is used.

To query a plugin's capabilities, use `plugin_info`:

```
>>> io.plugin_info('pil')
>>>
{'description': 'Image reading via the Python Imaging Library',
 'provides': 'imread, imsave'}
```

1.1.7 Handling Video Files

Sometimes it is necessary to read a sequence of images from a standard video file, such as .avi and .mov files.

In a scientific context, it is usually better to avoid these formats in favor of a simple directory of images or a multi-dimensional TIF. Video formats are more difficult to read piecemeal, typically do not support random frame access or research-minded meta data, and use lossy compression if not carefully configured. But video files are in widespread use, and they are easy to share, so it is convenient to be equipped to read and write them when necessary.

Tools for reading video files vary in their ease of installation and use, their disk and memory usage, and their cross-platform compatibility. This is a practical guide.

A Workaround: Convert the Video to an Image Sequence

For a one-off solution, the simplest, surest route is to convert the video to a collection of sequentially-numbered image files, often called an image sequence. Then the images files can be read into an `ImageCollection` by `skimage.io.imread_collection`. Converting the video to frames can be done easily in `ImageJ`, a cross-platform, GUI-based program from the bio-imaging community, or `FFmpeg`, a powerful command-line utility for manipulating video files.

In FFmpeg, the following command generates an image file from each frame in a video. The files are numbered with five digits, padded on the left with zeros.

```
ffmpeg -i "video.mov" -f image2 "video-frame%05d.png"
```

More information is available in an FFmpeg tutorial on image sequences.

Generating an image sequence has disadvantages: they can be large and unwieldy, and generating them can take some time. It is generally preferable to work directly with the original video file. For a more direct solution, we need to execute FFmpeg or LibAV from Python to read frames from the video. FFmpeg and LibAV are two large open-source projects that decode video from the sprawling variety of formats used in the wild. There are several ways to use them from Python. Each, unfortunately, has some disadvantages.

PyAV

PyAV uses FFmpeg's (or LibAV's) libraries to read image data directly from the video file. It invokes them using Cython bindings, so it is very fast.

```
import av
v = av.open('path/to/video.mov')
```

PyAV's API reflects the way frames are stored in a video file.

```
for packet in container.demux():
    for frame in packet.decode():
        if frame.type == 'video':
            img = frame.to_image() # PIL/Pillow image
            arr = np.asarray(img) # numpy array
            # Do something!
```

Adding Random Access to PyAV

The *Video* class in PIMS invokes PyAV and adds additional functionality to solve a common problem in scientific applications, accessing a video by frame number. Video file formats are designed to be searched in an approximate way, by time, and they do not support an efficient means of seeking a specific frame number. PIMS adds this missing functionality by decoding (but not reading) the entire video at and producing an internal table of contents that supports indexing by frame.

```
import pims
v = pims.Video('path/to/video.mov')
v[-1] # a 2D numpy array representing the last frame
```

MoviePy

Moviepy invokes FFmpeg through a subprocess, pipes the decoded video from FFmpeg into RAM, and reads it out. This approach is straightforward, but it can be brittle, and it's not workable for large videos that exceed available RAM. It works on all platforms if FFmpeg is installed.

Since it does not link to FFmpeg's underlying libraries, it is easier to install but about half as fast.

```
from moviepy.editor import VideoFileClip
myclip = VideoFileClip("some_video.avi")
```

Imageio

Imageio takes the same approach as MoviePy. It supports a wide range of other image file formats as well.

```
import imageio
filename = '/tmp/file.mp4'
vid = imageio.get_reader(filename, 'ffmpeg')

for num, image in vid.iter_data():
    print(image.mean())
```

(continues on next page)

(continued from previous page)

```
metadata = vid.get_meta_data()
```

OpenCV

Finally, another solution is the [VideoReader](#) class in OpenCV, which has bindings to FFmpeg. If you need OpenCV for other reasons, then this may be the best approach.

1.1.8 Data visualization

Data visualization takes an important place in image processing. Data can be a single 2D grayscale image or a more complex one with multidimensional aspects: 3D in space, timelapse, multiple channels.

Therefore, the visualization strategy will depend on the data complexity and a range of tools external to scikit-image can be used for this purpose. Historically, scikit-image provided viewer tools but powerful packages are now available and must be preferred.

Matplotlib

[Matplotlib](#) is a library able to generate static plots, which includes image visualization.

Plotly

[Plotly](#) is a plotting library relying on web technologies with interaction capabilities.

Mayavi

[Mayavi](#) can be used to visualize 3D images.

Napari

[Napari](#) is a multi-dimensional image viewer. It's designed for browsing, annotating, and analyzing large multi-dimensional images.

1.1.9 Image adjustment: transforming image content

Color manipulation

Most functions for manipulating color channels are found in the submodule `skimage.color`.

Conversion between color models

Color images can be represented using different color spaces. One of the most common color spaces is the RGB space, where an image has red, green and blue channels. However, other color models are widely used, such as the HSV color model, where hue, saturation and value are independent channels, or the CMYK model used for printing.

`skimage.color` provides utility functions to convert images to and from different color spaces. Integer-type arrays can be transformed to floating-point type by the conversion operation:

```
>>> # bright saturated red
>>> red_pixel_rgb = np.array([[[255, 0, 0]]], dtype=np.uint8)
>>> color.rgb2hsv(red_pixel_rgb)
array([[[ 0.,  1.,  1.]]])

>>> # darker saturated blue
>>> dark_blue_pixel_rgb = np.array([[[0, 0, 100]]], dtype=np.uint8)
>>> color.rgb2hsv(dark_blue_pixel_rgb)
array([[[ 0.66666667,  1.        ,  0.39215686]]])

>>> # less saturated pink
>>> pink_pixel_rgb = np.array([[[255, 100, 255]]], dtype=np.uint8)
>>> color.rgb2hsv(pink_pixel_rgb)
array([[[ 0.83333333,  0.60784314,  1.        ]]])
```

Conversion from RGBA to RGB - Removing alpha channel through alpha blending

Converting an RGBA image to an RGB image by alpha blending it with a background is realized with `rgba2rgb()`

```
>>> from skimage.color import rgba2rgb
>>> from skimage import data
>>> img_rgba = data.logo()
>>> img_rgb = rgba2rgb(img_rgba)
```

Conversion between color and gray values

Converting an RGB image to a grayscale image is realized with `rgb2gray()`

```
>>> from skimage.color import rgb2gray
>>> from skimage import data
>>> img = data.astronaut()
>>> img_gray = rgb2gray(img)
```

`rgb2gray()` uses a non-uniform weighting of color channels, because of the different sensitivity of the human eye to different colors. Therefore, such a weighting ensures luminance preservation from RGB to grayscale:

```
>>> red_pixel = np.array([[[255, 0, 0]]], dtype=np.uint8)
>>> color.rgb2gray(red_pixel)
array([[ 0.2125]])

>>> green_pixel = np.array([[[0, 255, 0]]], dtype=np.uint8)
>>> color.rgb2gray(green_pixel)
array([[ 0.7154]])
```

Converting a grayscale image to RGB with `gray2rgb()` simply duplicates the gray values over the three color channels.

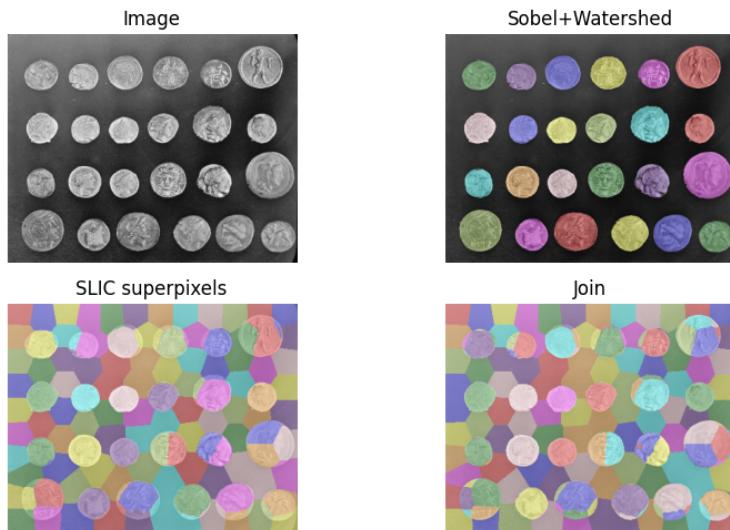
Image inversion

An inverted image is also called complementary image. For binary images, True values become False and conversely. For grayscale images, pixel values are replaced by the difference of the maximum value of the data type and the actual value. For RGB images, the same operation is done for each channel. This operation can be achieved with `skimage.util.invert()`:

```
>>> from skimage import util
>>> img = data.camera()
>>> inverted_img = util.invert(img)
```

Painting images with labels

`label2rgb()` can be used to superimpose colors on a grayscale image using an array of labels to encode the regions to be represented with the same color.



Examples:

- *Tinting gray-scale images*
- *Find the intersection of two segmentations*
- *RAG Thresholding*

Contrast and exposure

Image pixels can take values determined by the `dtype` of the image (see *Image data types and what they mean*), such as 0 to 255 for `uint8` images or `[0, 1]` for floating-point images. However, most images either have a narrower range of values (because of poor contrast), or have most pixel values concentrated in a subrange of the accessible values. `skimage.exposure` provides functions that spread the intensity values over a larger range.

A first class of methods compute a nonlinear function of the intensity, that is independent of the pixel values of a specific image. Such methods are often used for correcting a known non-linearity of sensors, or receptors such as the human eye. A well-known example is [Gamma correction](#), implemented in `adjust_gamma()`.

Other methods re-distribute pixel values according to the *histogram* of the image. The histogram of pixel values is computed with `skimage.exposure.histogram()`:

```
>>> image = np.array([[1, 3], [1, 1]])
>>> exposure.histogram(image)
(array([3, 0, 1]), array([1, 2, 3]))
```

`histogram()` returns the number of pixels for each value bin, and the centers of the bins. The behavior of `histogram()` is therefore slightly different from the one of `numpy.histogram()`, which returns the boundaries of the bins.

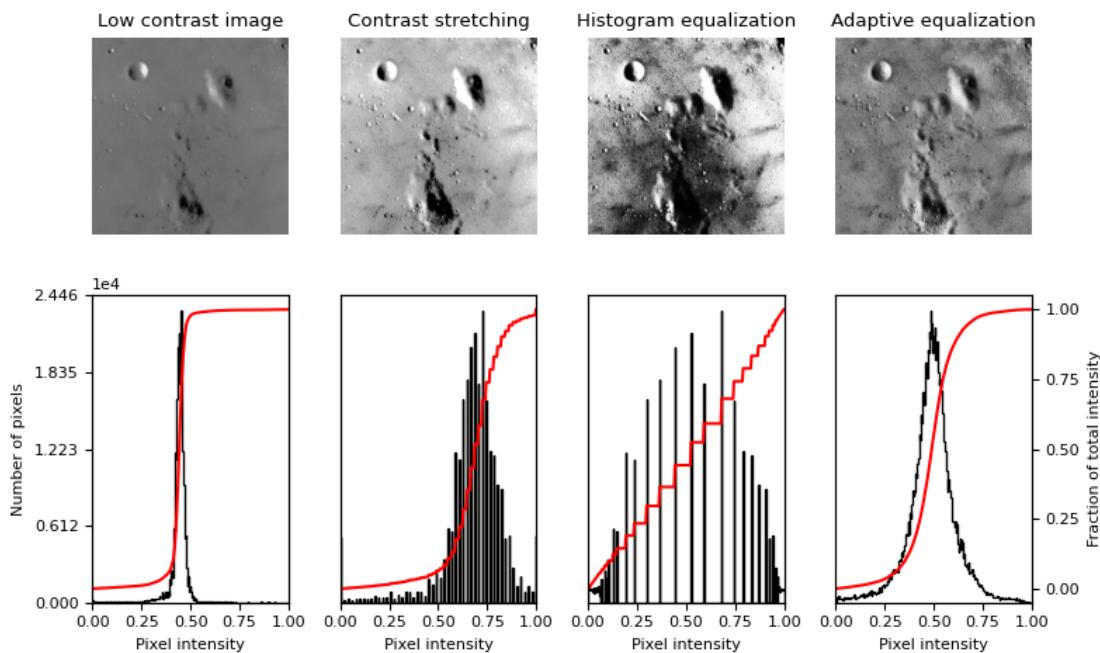
The simplest contrast enhancement `rescale_intensity()` consists in stretching pixel values to the whole allowed range, using a linear transformation:

```
>>> from skimage import exposure
>>> text = data.text()
>>> text.min(), text.max()
(10, 197)
>>> better_contrast = exposure.rescale_intensity(text)
>>> better_contrast.min(), better_contrast.max()
(0, 255)
```

Even if an image uses the whole value range, sometimes there is very little weight at the ends of the value range. In such a case, clipping pixel values using percentiles of the image improves the contrast (at the expense of some loss of information, because some pixels are saturated by this operation):

```
>>> moon = data.moon()
>>> v_min, v_max = np.percentile(moon, (0.2, 99.8))
>>> v_min, v_max
(10.0, 186.0)
>>> better_contrast = exposure.rescale_intensity(
...                         moon, in_range=(v_min, v_max))
```

The function `equalize_hist()` maps the cumulative distribution function (cdf) of pixel values onto a linear cdf, ensuring that all parts of the value range are equally represented in the image. As a result, details are enhanced in large regions with poor contrast. As a further refinement, histogram equalization can be performed in subregions of the image with `equalize_adapthist()`, in order to correct for exposure gradients across the image. See the example *Histogram Equalization*.

**Examples:**

- *Histogram Equalization*

1.1.10 Geometrical transformations of images

Cropping, resizing and rescaling images

Images being NumPy arrays (as described in the *A crash course on NumPy for images* section), cropping an image can be done with simple slicing operations. Below we crop a 100x100 square corresponding to the top-left corner of the astronaut image. Note that this operation is done for all color channels (the color dimension is the last, third dimension):

```
>>> from skimage import data
>>> img = data.astronaut()
>>> top_left = img[:100, :100]
```

In order to change the shape of the image, `skimage.color` provides several functions described in *Rescale, resize, and downscale*.

```
from skimage import data, color
from skimage.transform import rescale, resize, downscale_local_mean

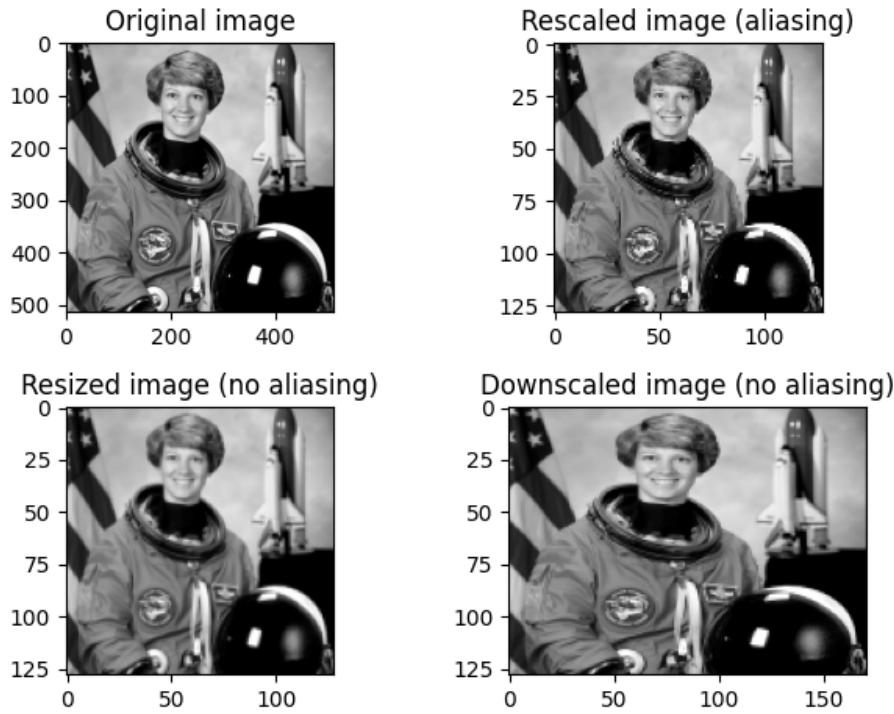
image = color.rgb2gray(data.astronaut())

image_rescaled = rescale(image, 0.25, anti_aliasing=False)
image_resized = resize(image, (image.shape[0] // 4, image.shape[1] // 4),
                      anti_aliasing=True)
```

(continues on next page)

(continued from previous page)

```
image_downscaled = downscale_local_mean(image, (4, 3))
```



Projective transforms (homographies)

Homographies are transformations of a Euclidean space that preserve the alignment of points. Specific cases of homographies correspond to the conservation of more properties, such as parallelism (affine transformation), shape (similar transformation) or distances (Euclidean transformation). The different types of homographies available in scikit-image are presented in *Types of homographies*.

Projective transformations can either be created using the explicit parameters (e.g. scale, shear, rotation and translation):

```
from skimage import data
from skimage import transform
from skimage import img_as_float

tform = transform.EuclideanTransform(
    rotation=np.pi / 12.,
    translation = (100, -20)
)
```

or the full transformation matrix:

```
from skimage import data
from skimage import transform
from skimage import img_as_float
```

(continues on next page)

(continued from previous page)

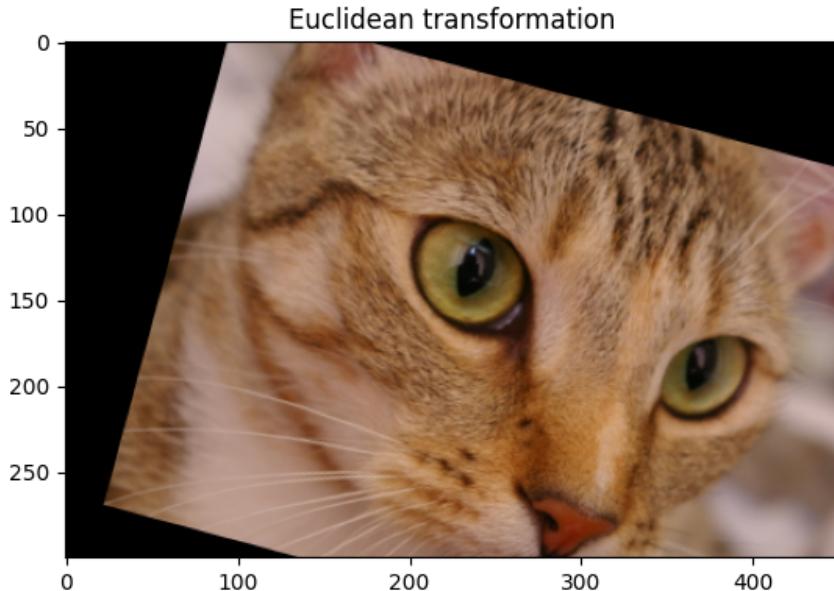
```
matrix = np.array([[np.cos(np.pi/12), -np.sin(np.pi/12), 100],
                  [np.sin(np.pi/12), np.cos(np.pi/12), -20],
                  [0, 0, 1]])
tform = transform.EuclideanTransform(matrix)
```

The transformation matrix of a transform is available as its `tform.params` attribute. Transformations can be composed by multiplying matrices with the @ matrix multiplication operator.

Transformation matrices use [Homogeneous coordinates](#), which are the extension of Cartesian coordinates used in Euclidean geometry to the more general projective geometry. In particular, points at infinity can be represented with finite coordinates.

Transformations can be applied to images using `skimage.transform.warp()`:

```
img = img_as_float(data.chelsea())
tf_img = transform.warp(img, tform.inverse)
```

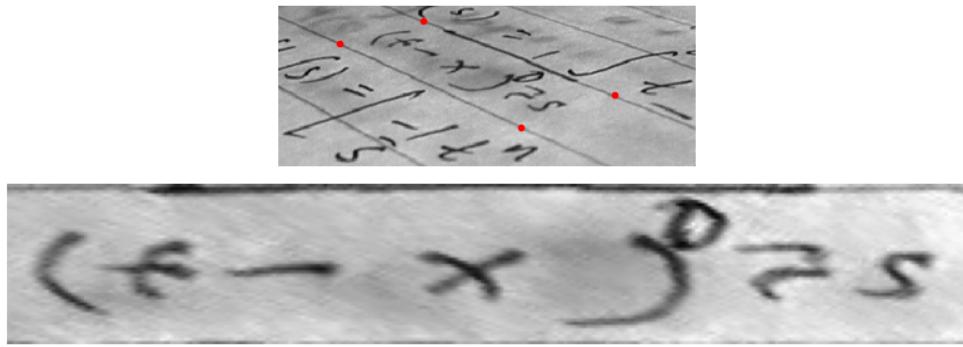


The different transformations in `skimage.transform` have a `estimate` method in order to estimate the parameters of the transformation from two sets of points (the source and the destination), as explained in the [Using geometric transformations](#) tutorial:

```
text = data.text()

src = np.array([[0, 0], [0, 50], [300, 50], [300, 0]])
dst = np.array([[155, 15], [65, 40], [260, 130], [360, 95]])

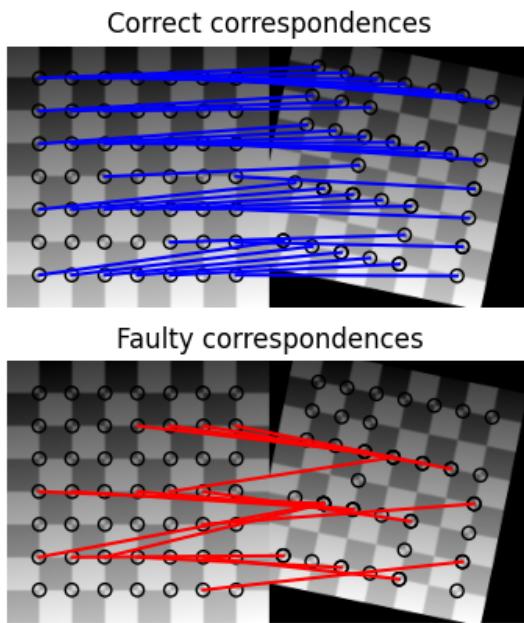
tform3 = transform.ProjectiveTransform()
tform3.estimate(src, dst)
warped = transform.warp(text, tform3, output_shape=(50, 300))
```



The `estimate` method uses least-squares optimization to minimize the distance between source and destination points. Source and destination points can be determined manually, or using the different methods for feature detection available in `skimage.feature`, such as

- *Corner detection,*
- *ORB feature detector and binary descriptor,*
- *BRIEF binary descriptor,*
- etc.

and matching points using `skimage.feature.match_descriptors()` before estimating transformation parameters. However, spurious matches are often made, and it is advisable to use the RANSAC algorithm (instead of simple least-squares optimization) to improve the robustness to outliers, as explained in *Robust matching using RANSAC*.



Examples showing applications of transformation estimation are

- stereo matching *Fundamental matrix estimation* and
- image rectification *Using geometric transformations*

The `estimate` method is point-based, that is, it uses only a set of points from the source and destination images. For estimating translations (shifts), it is also possible to use a *full-field* method using all pixels, based on Fourier-space cross-correlation. This method is implemented by `skimage.registration.register_translation()` and explained in the *Image Registration* tutorial.



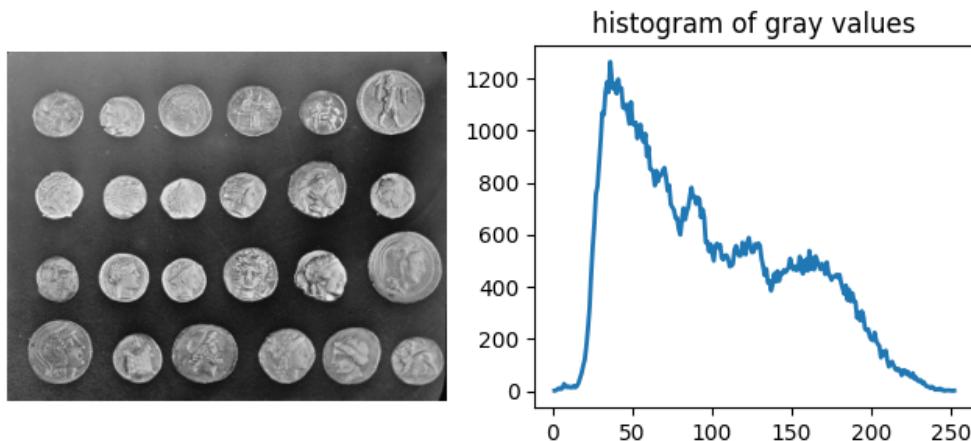
The *Using Polar and Log-Polar Transformations for Registration* tutorial explains a variant of this full-field method for estimating a rotation, by using first a log-polar transformation.

1.1.11 Tutorials

Image Segmentation

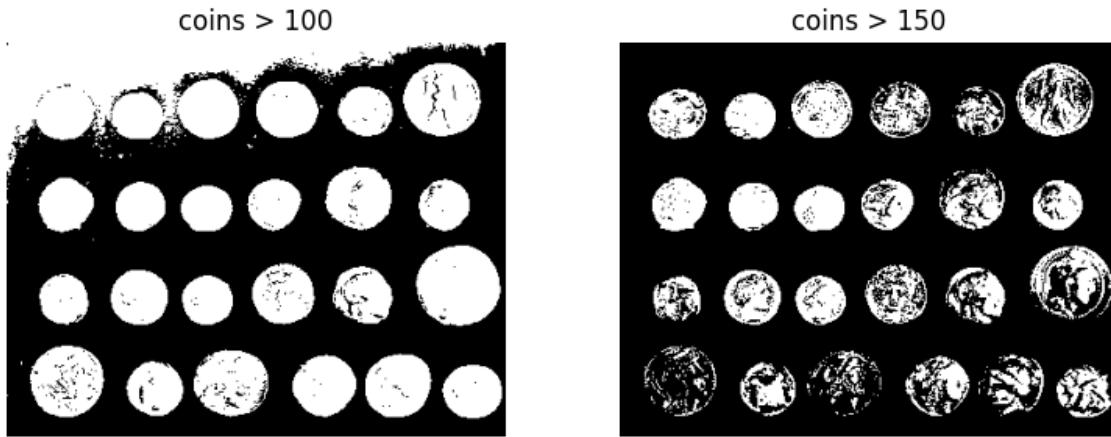
Image segmentation is the task of labeling the pixels of objects of interest in an image.

In this tutorial, we will see how to segment objects from a background. We use the `coins` image from `skimage.data`. This image shows several coins outlined against a darker background. The segmentation of the coins cannot be done directly from the histogram of gray values, because the background shares enough gray levels with the coins that a thresholding segmentation is not sufficient.



```
>>> from skimage import data
>>> from skimage.exposure import histogram
>>> coins = data.coins()
>>> hist, hist_centers = histogram(coins)
```

Simply thresholding the image leads either to missing significant parts of the coins, or to merging parts of the background with the coins. This is due to the inhomogeneous lighting of the image.



A first idea is to take advantage of the local contrast, that is, to use the gradients rather than the gray values.

Edge-based segmentation

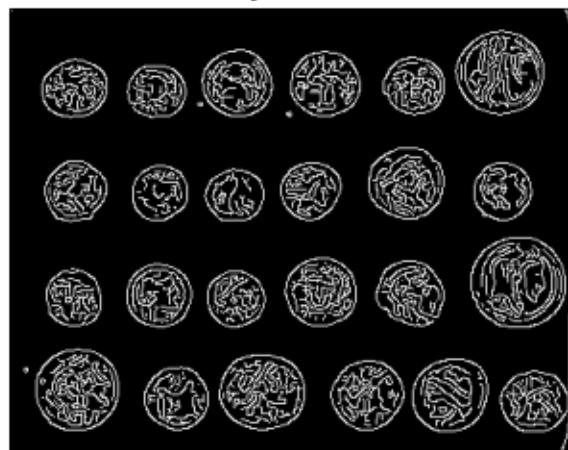
Let us first try to detect edges that enclose the coins. For edge detection, we use the Canny detector of `skimage.feature.canny`

```
>>> from skimage.feature import canny
>>> edges = canny(coins/255.)
```

As the background is very smooth, almost all edges are found at the boundary of the coins, or inside the coins.

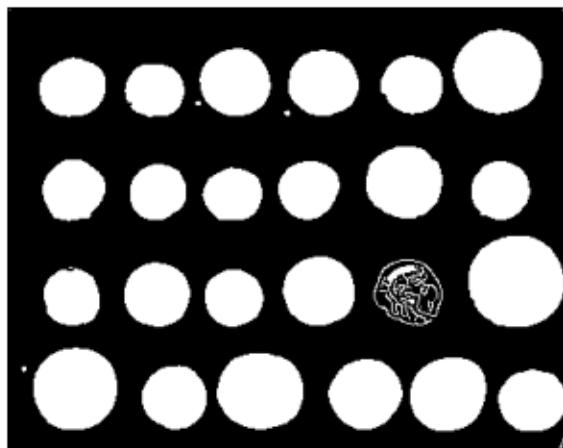
```
>>> from scipy import ndimage as ndi
>>> fill_coins = ndi.binary_fill_holes(edges)
```

Canny detector



Now that we have contours that delineate the outer boundary of the coins, we fill the inner part of the coins using the `ndi.binary_fill_holes` function, which uses mathematical morphology to fill the holes.

filling the holes

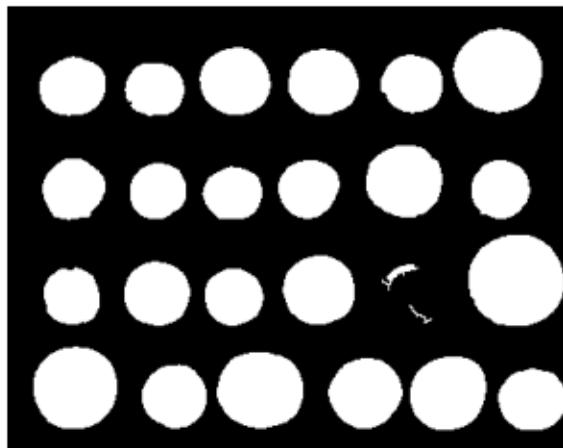


Most coins are well segmented out of the background. Small objects from the background can be easily removed using the `ndi.label` function to remove objects smaller than a small threshold.

```
>>> label_objects, nb_labels = ndi.label(fill_coins)
>>> sizes = np.bincount(label_objects.ravel())
>>> mask_sizes = sizes > 20
>>> mask_sizes[0] = 0
>>> coins_cleaned = mask_sizes[label_objects]
```

However, the segmentation is not very satisfying, since one of the coins has not been segmented correctly at all. The reason is that the contour that we got from the Canny detector was not completely closed, therefore the filling function did not fill the inner part of the coin.

removing small objects



Therefore, this segmentation method is not very robust: if we miss a single pixel of the contour of the object, we will not be able to fill it. Of course, we could try to dilate the contours in order to close them. However, it is preferable to try a more robust method.

Region-based segmentation

Let us first determine markers of the coins and the background. These markers are pixels that we can label unambiguously as either object or background. Here, the markers are found at the two extreme parts of the histogram of gray values:

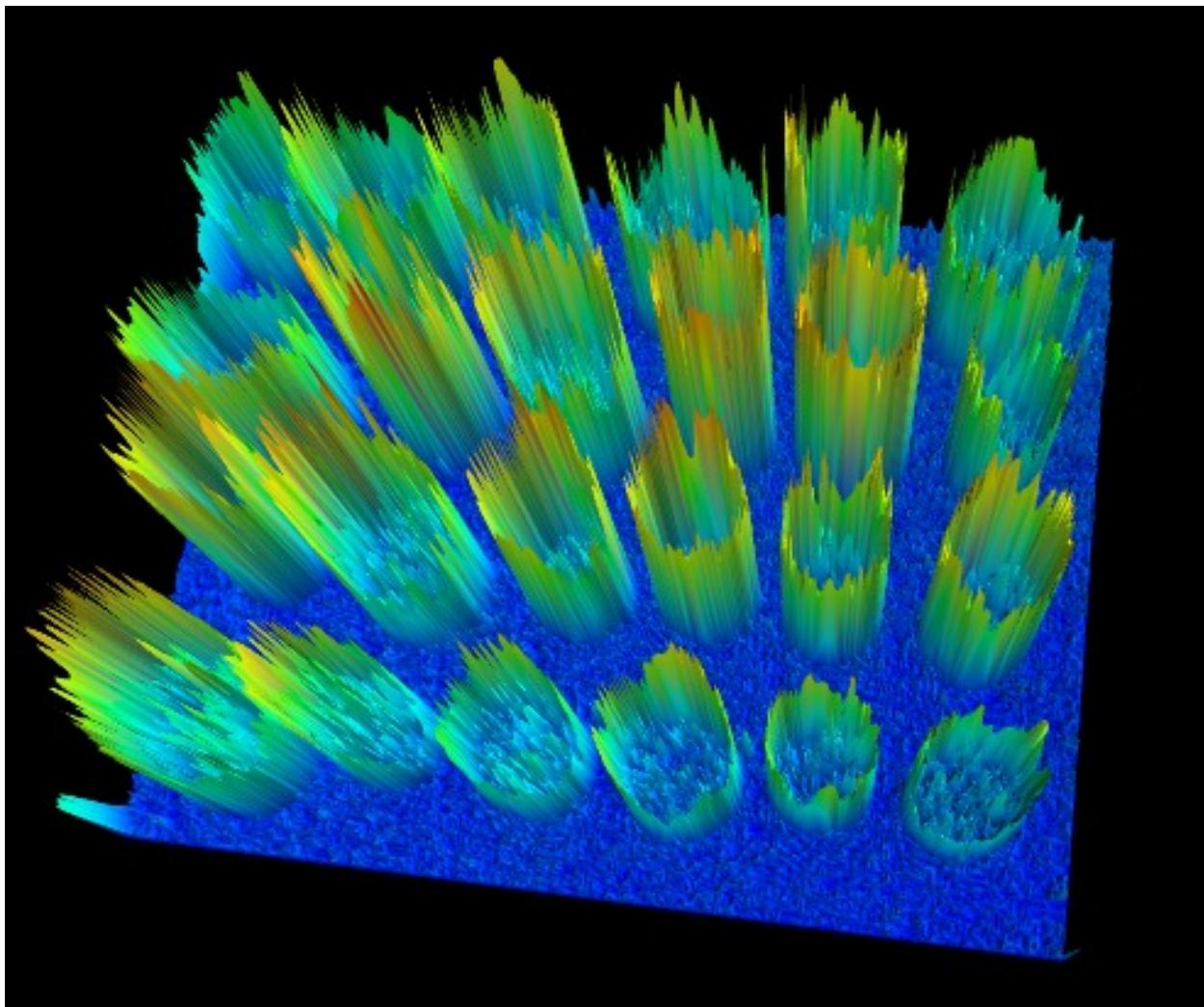
```
>>> markers = np.zeros_like(coins)
>>> markers[coins < 30] = 1
>>> markers[coins > 150] = 2
```

We will use these markers in a watershed segmentation. The name watershed comes from an analogy with hydrology. The [watershed transform](#) floods an image of elevation starting from markers, in order to determine the catchment basins of these markers. Watershed lines separate these catchment basins, and correspond to the desired segmentation.

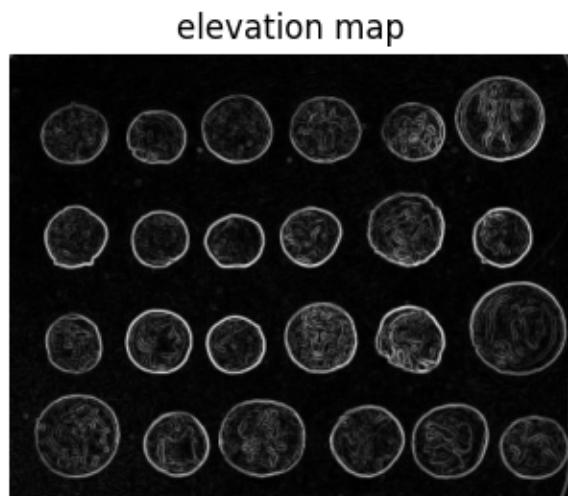
The choice of the elevation map is critical for good segmentation. Here, the amplitude of the gradient provides a good elevation map. We use the Sobel operator for computing the amplitude of the gradient:

```
>>> from skimage.filters import sobel
>>> elevation_map = sobel(coins)
```

From the 3-D surface plot shown below, we see that high barriers effectively separate the coins from the background.

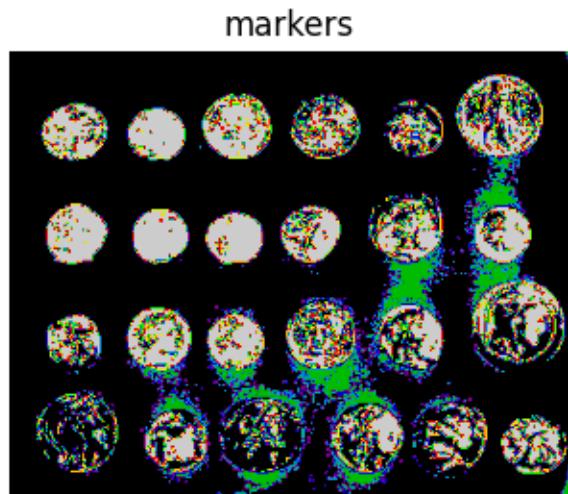


and here is the corresponding 2-D plot:



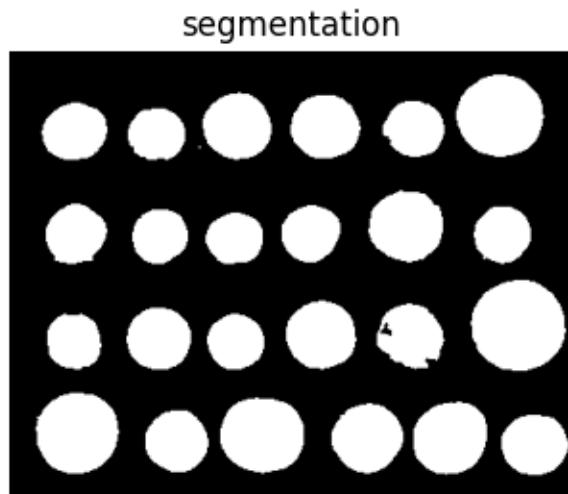
The next step is to find markers of the background and the coins based on the extreme parts of the histogram of gray values:

```
>>> markers = np.zeros_like(coins)
>>> markers[coins < 30] = 1
>>> markers[coins > 150] = 2
```



Let us now compute the watershed transform:

```
>>> from skimage.segmentation import watershed
>>> segmentation = watershed(elevation_map, markers)
```



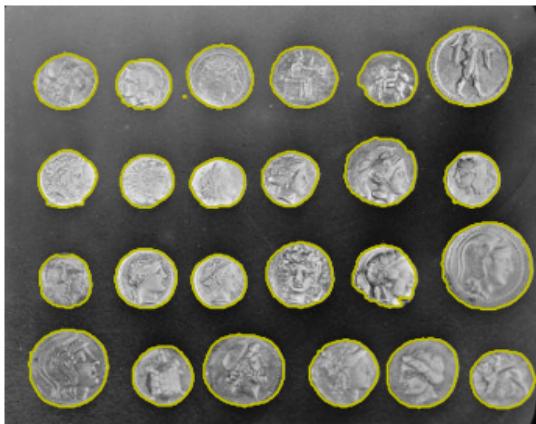
With this method, the result is satisfying for all coins. Even if the markers for the background were not well distributed, the barriers in the elevation map were high enough for these markers to flood the entire background.

We remove a few small holes with mathematical morphology:

```
>>> segmentation = ndi.binary_fill_holes(segmentation - 1)
```

We can now label all the coins one by one using `ndi.label`:

```
>>> labeled_coins, _ = ndi.label(segmentation)
```



How to parallelize loops

In image processing, we frequently apply the same algorithm on a large batch of images. In this paragraph, we propose to use `joblib` to parallelize loops. Here is an example of such repetitive tasks:

```
from skimage import data, color, util
from skimage.restoration import denoise_tv_chambolle
from skimage.feature import hog

def task(image):
    """
    Apply some functions and return an image.
    """
    image = denoise_tv_chambolle(image[0][0], weight=0.1, channel_axis=-1)
    fd, hog_image = hog(color.rgb2gray(image), orientations=8,
                        pixels_per_cell=(16, 16), cells_per_block=(1, 1),
                        visualize=True)
    return hog_image

# Prepare images
hubble = data.hubble_deep_field()
width = 10
pics = util.view_as_windows(hubble, (width, hubble.shape[1], hubble.shape[2]), step=width)
```

To call the function `task` on each element of the list `pics`, it is usual to write a for loop. To measure the execution time of this loop, you can use ipython and measure the execution time with `%timeit`.

```
def classic_loop():
    for image in pics:
        task(image)

%timeit classic_loop()
```

Another equivalent way to code this loop is to use a comprehension list which has the same efficiency.

```
def comprehension_loop():
    [task(image) for image in pics]

%timeit comprehension_loop()
```

`joblib` is a library providing an easy way to parallelize for loops once we have a comprehension list. The number of jobs can be specified.

```
from joblib import Parallel, delayed
def joblib_loop():
    Parallel(n_jobs=4)(delayed(task)(i) for i in pics)

%timeit joblib_loop()
```

1.1.12 Getting help on using skimage

API Reference

Keep the [reference guide](#) handy while programming with scikit-image. Select the docs that match the version of skimage you are using.

Examples gallery

The *Examples* gallery provides graphical examples and code snippets of typical image processing tasks. There, you may find an example that is close to your use case.

Feel free to suggest new gallery examples on our [developer forum](#).

Search field

Use the `quick` search field in the navigation bar of the online documentation to find mentions of keywords (segmentation, rescaling, denoising, etc.) in the documentation.

API Discovery

We provide a `lookfor` function to search API functions:

```
import skimage as ski
ski.lookfor('eigenvector')
```

Also see NumPy's `lookfor`.

Ask for help

Still stuck? We are here to help! Reach out through:

- our [user forum](#) for image processing and usage questions;
- our [developer forum](#) for technical questions and suggestions;
- our [chat channel](#) for real-time interaction; or
- [Stack Overflow](#) for coding questions.

1.1.13 Glossary

Work in progress

array

Numerical array, provided by the `numpy.ndarray` object. In scikit-image, images are NumPy arrays with dimensions that correspond to spatial dimensions of the image, and color channels for color images. See [A crash course on NumPy for images](#).

channel

Typically used to refer to a single color channel in a color image. RGBA images have an additional alpha (transparency) channel. Functions use a `channel_axis` argument to specify which axis of an array corresponds

to channels. Images without channels are indicated via `channel_axis=None`. Aside from the functions in `skimage.color`, most functions with a `channel_axis` argument just apply the same operation across each channel. In this case, the “channels” do not strictly need to represent color or alpha information, but may be any generic batch dimension over which to operate.

circle

The perimeter of a *disk*.

contour

Curve along which a 2-D image has a constant value. The interior (resp. exterior) of the contour has values greater (resp. smaller) than the contour value.

contrast

Differences of intensity or color in an image, which make objects distinguishable. Several functions to manipulate the contrast of an image are available in `skimage.exposure`. See *Contrast and exposure*.

disk

A filled-in *circle*.

float

Representation of real numbers, for example as `np.float32` or `np.float64`. See *Image data types and what they mean*. Some operations on images need a float datatype (such as multiplying image values with exponential prefactors in `filters.gaussian()`), so that images of integer type are often converted to float type internally. Also see `int` values.

float values

See `float`.

histogram

For an image, histogram of intensity values, where the range of intensity values is divided into bins and the histogram counts how many pixel values fall in each bin. See `exposure.histogram()`.

int

Representation of integer numbers, which can be signed or not, and encoded on one, two, four or eight bytes according to the maximum value which needs to be represented. In `scikit-image`, the most common integer types are `np.int64` (for large integer values) and `np.uint8` (for small integer values, typically images of labels with less than 255 labels). See *Image data types and what they mean*.

int values

See `int`.

iso-valued contour

See `contour`.

labels

An image of labels is of integer type, where pixels with the same integer value belong to the same object. For example, the result of a segmentation is an image of labels. `measure.label()` labels connected components of a binary image and returns an image of labels. Labels are usually contiguous integers, and

`segmentation.relabel_sequential()` can be used to relabel arbitrary labels to sequential (contiguous) ones.

label image

See *labels*.

pixel

Smallest element of an image. An image is a grid of pixels, and the intensity of each pixel is variable. A pixel can have a single intensity value in grayscale images, or several channels for color images. In scikit-image, pixels are the individual elements of `numpy arrays` (see *A crash course on NumPy for images*). Also see `voxel`.

segmentation

Partitioning an image into multiple objects (segments), for example an object of interest and its background. The output of a segmentation is typically an image of *labels*, where the pixels of different objects have been attributed different integer labels. Several segmentation algorithms are available in `skimage.segmentation`.

voxel

pixel (smallest element of an image) of a three-dimensional image.

1.2 Examples

A gallery of examples and that showcase how scikit-image can be used. Some examples demonstrate the use of the API in general and some demonstrate specific applications in tutorial form.

Hint: Check out our *User guide* for a narrative introduction to key library conventions and basic image manipulation.

1.2.1 Data

1.2.2 Operations on NumPy arrays

1.2.3 Manipulating exposure and color channels

1.2.4 Edges and lines

1.2.5 Geometrical transformations and registration

1.2.6 Image registration

1.2.7 Filtering and restoration

1.2.8 Detection of features and objects

1.2.9 Segmentation of objects

1.2.10 Longer examples and demonstrations

1.2.11 Examples for developers

In this folder, we have examples for advanced topics, including detailed explanations of the inner workings of certain algorithms.

These examples require some basic knowledge of image processing. They are targeted at existing or would-be scikit-image developers wishing to develop their knowledge of image processing algorithms.

Data

Datasets with 3 or more spatial dimensions

Most scikit-image functions are compatible with 3D datasets, i.e., images with 3 spatial dimensions (to be distinguished from 2D multichannel images, which are also arrays with three axes). `skimage.data.cells3d()` returns a 3D fluorescence microscopy image of cells. The returned dataset is a 3D multichannel image with dimensions provided in (z, c, y, x) order. Channel 0 contains cell membranes, while channel 1 contains nuclei.

The example below shows how to explore this dataset. This 3D image can be used to test the various functions of scikit-image.

```
from skimage import data
import plotly
import plotly.express as px
import numpy as np

img = data.cells3d()[20:]

# omit some slices that are partially empty
img = img[5:26]

upper_limit = 1.5 * np.percentile(img, q=99)
img = np.clip(img, 0, upper_limit)
```

(continues on next page)

(continued from previous page)

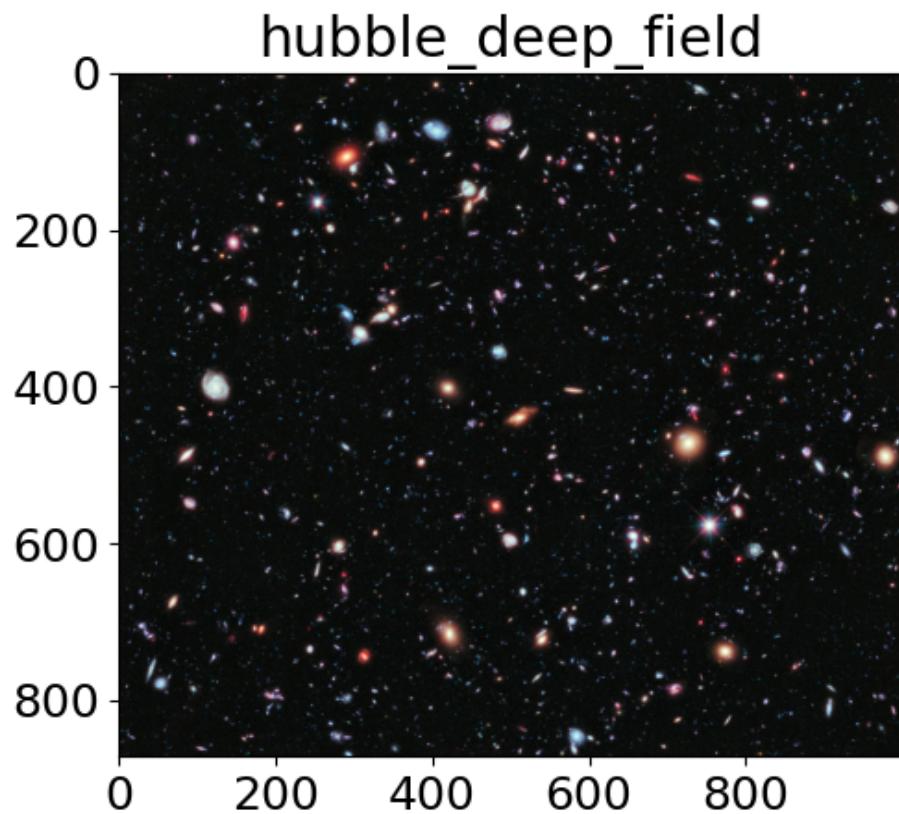
```
fig = px.imshow(  
    img,  
    facet_col=1,  
    animation_frame=0,  
    binary_string=True,  
    binary_format="jpg",  
)  
fig.layout.annotations[0]["text"] = "Cell membranes"  
fig.layout.annotations[1]["text"] = "Nuclei"  
plotly.io.show(fig)
```

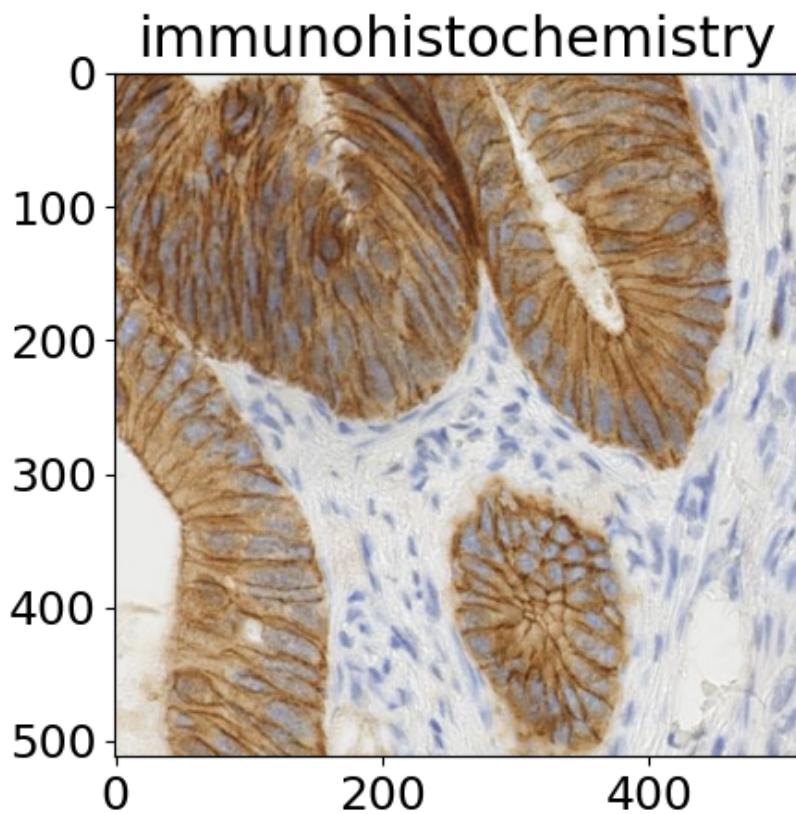
Total running time of the script: (0 minutes 2.650 seconds)

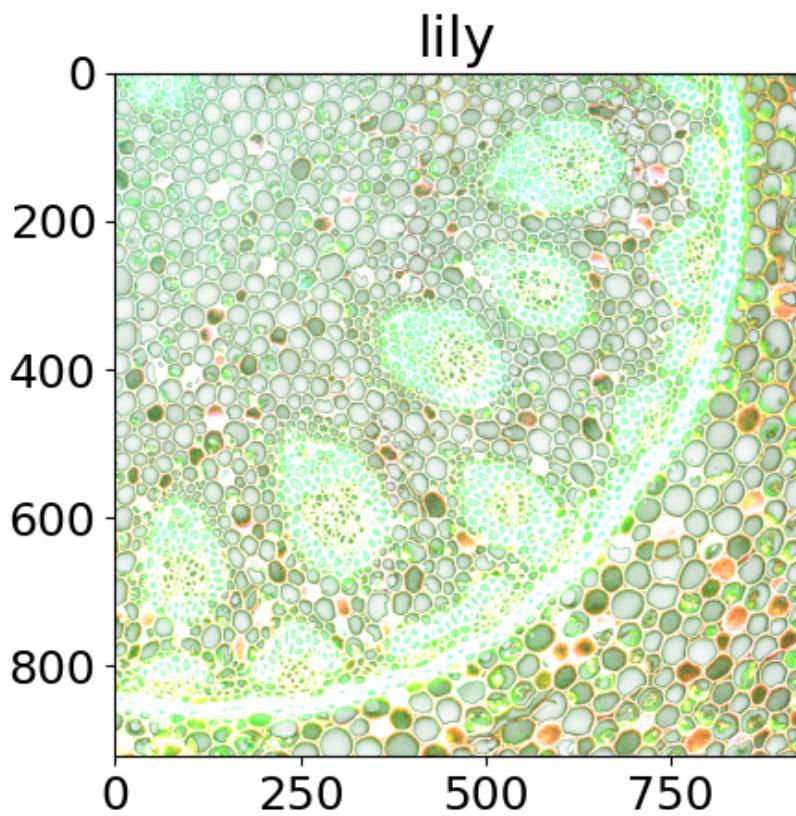
Scientific images

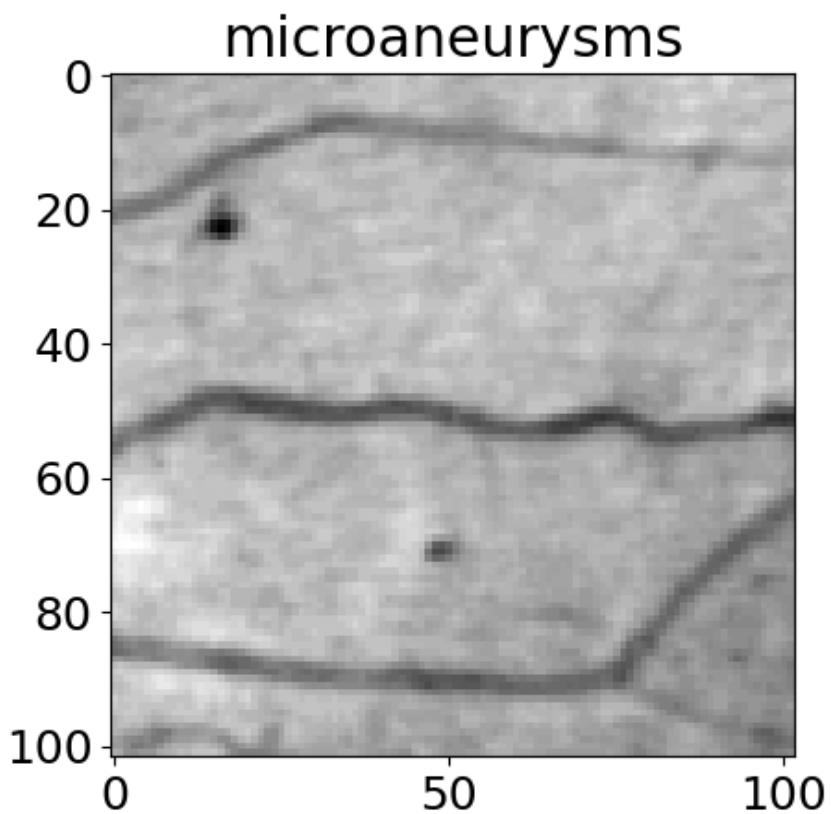
The title of each image indicates the name of the function.

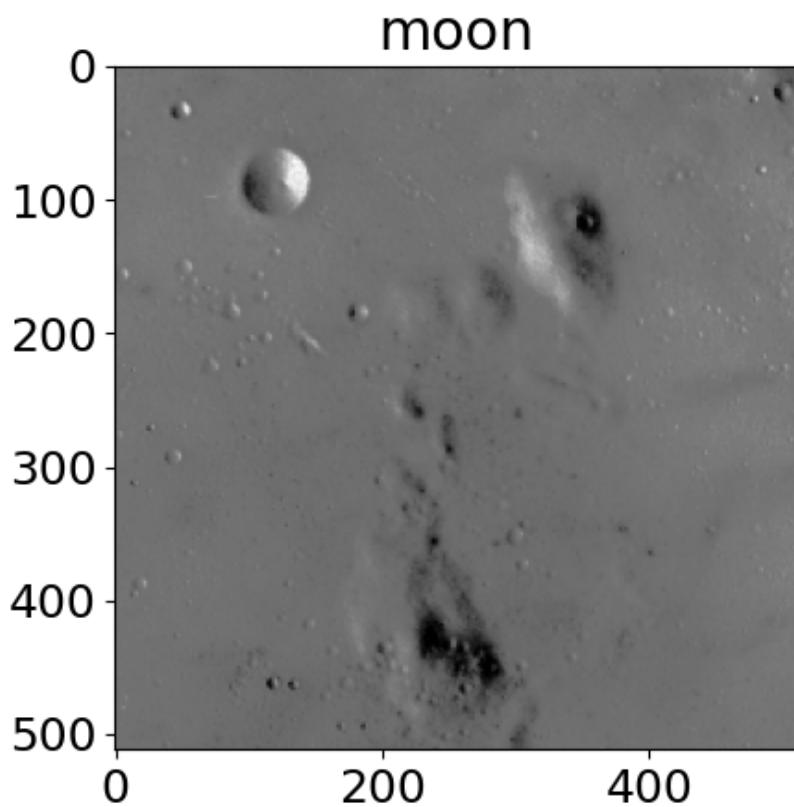
```
import matplotlib.pyplot as plt  
import matplotlib  
import numpy as np  
  
from skimage import data  
  
matplotlib.rcParams['font.size'] = 18  
  
images = ('hubble_deep_field',  
          'immunohistochemistry',  
          'lily',  
          'microaneurysms',  
          'moon',  
          'retina',  
          'shepp_logan_phantom',  
          'skin',  
          'cell',  
          'human_mitosis',  
          )  
  
  
for name in images:  
    caller = getattr(data, name)  
    image = caller()  
    plt.figure()  
    plt.title(name)  
    if image.ndim == 2:  
        plt.imshow(image, cmap=plt.cm.gray)  
    else:  
        plt.imshow(image)  
  
plt.show()
```

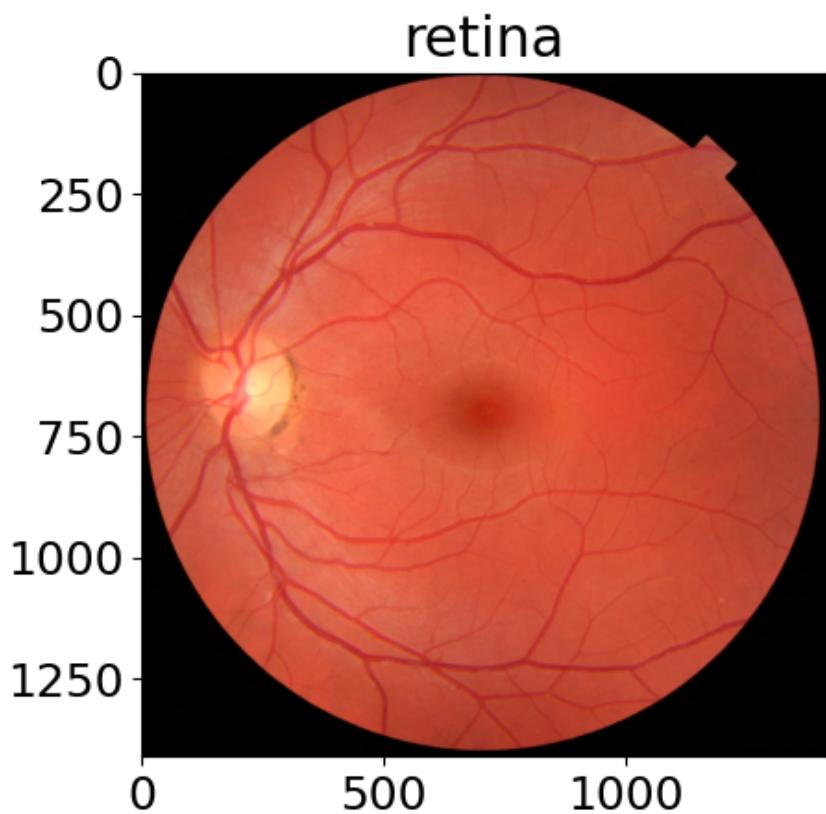


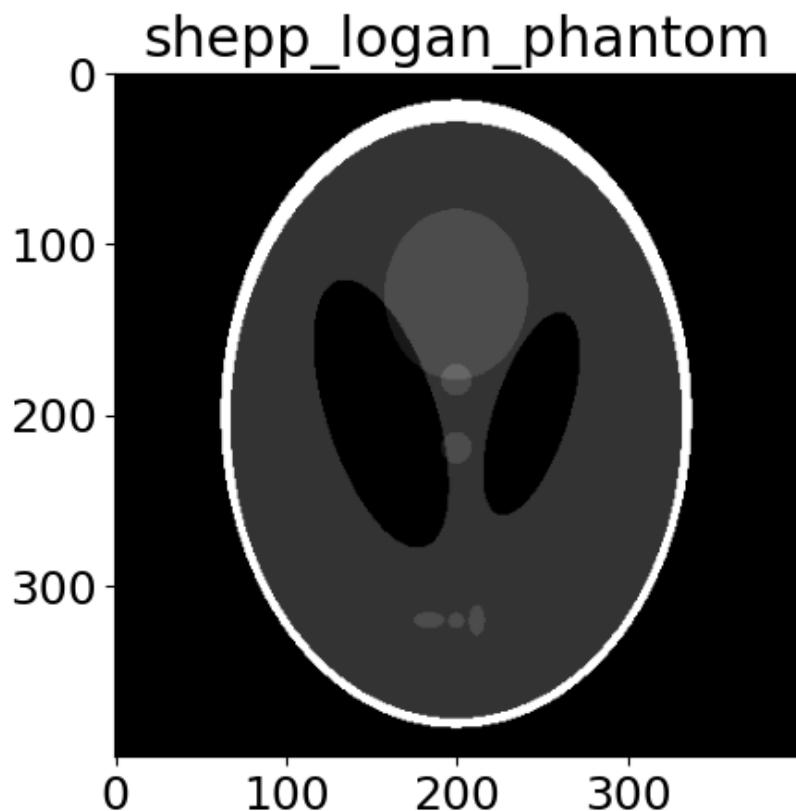


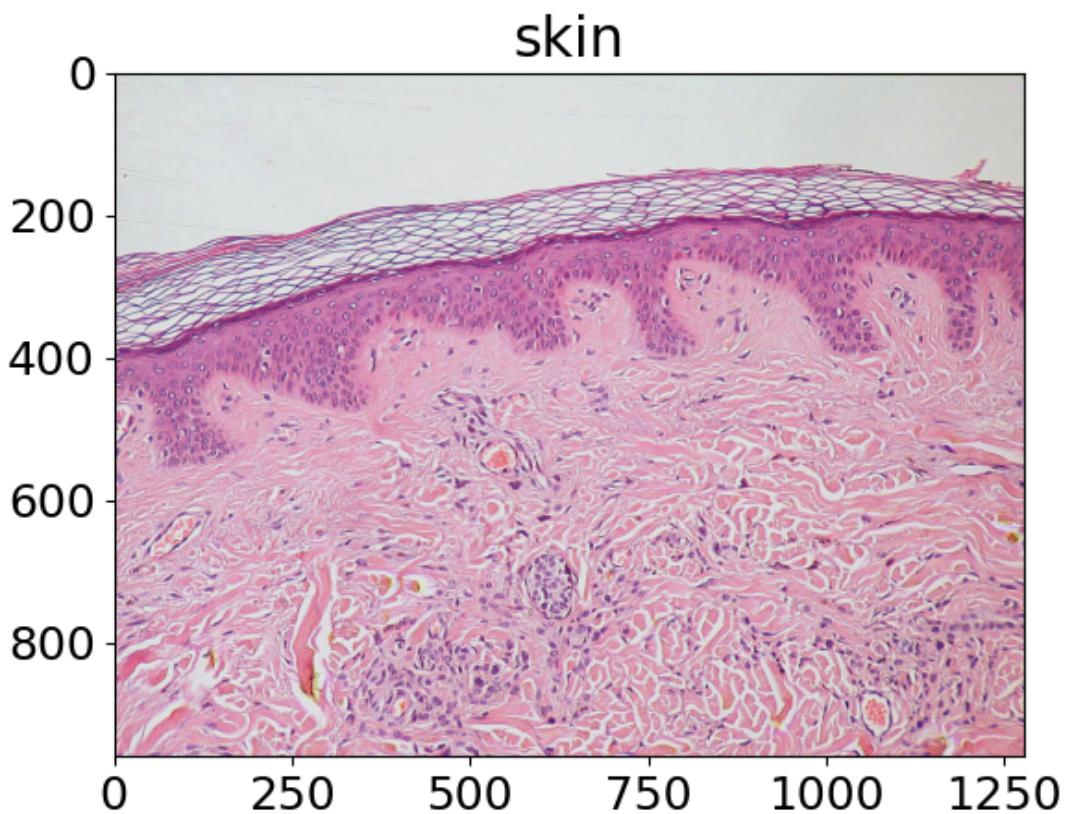


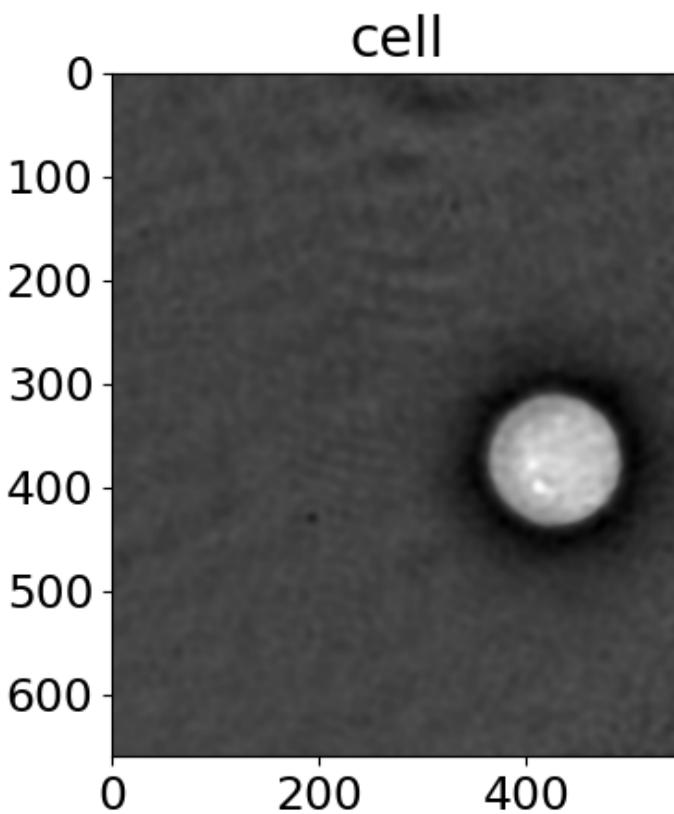


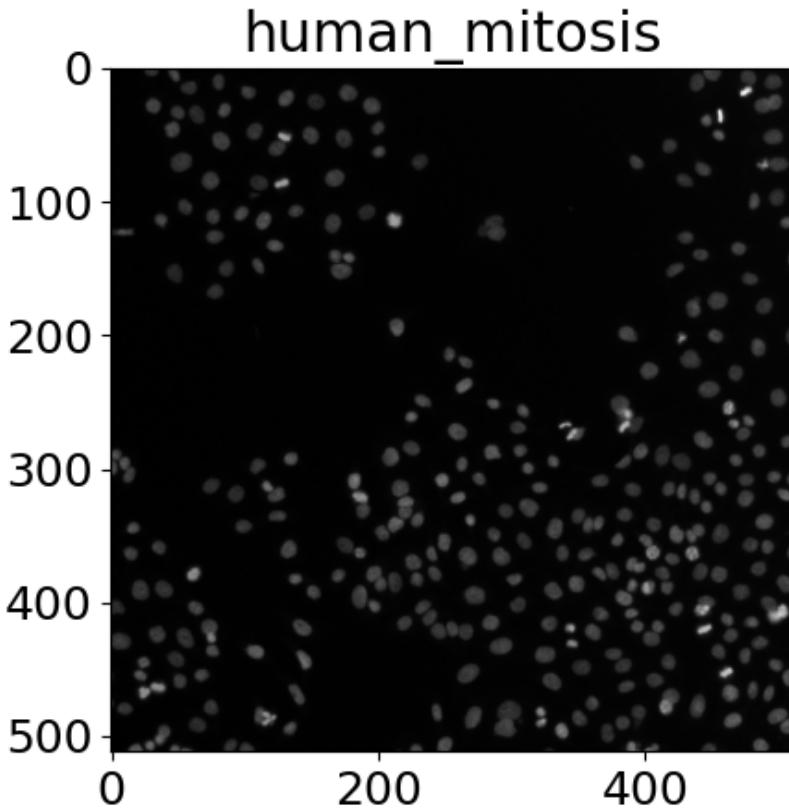








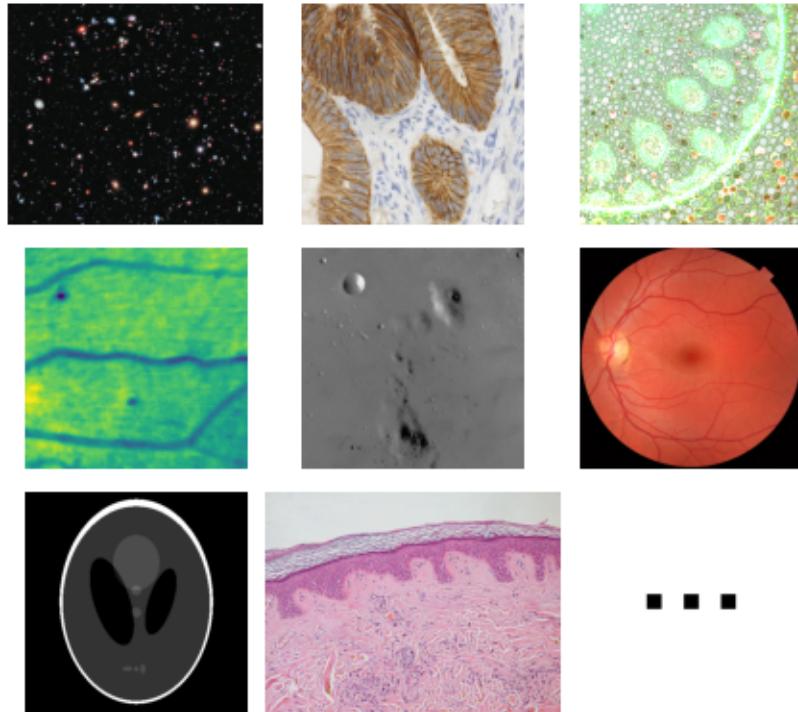




Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).

Thumbnail image for the gallery

```
fig, axs = plt.subplots(nrows=3, ncols=3)
for ax in axs.flat:
    ax.axis("off")
axs[0, 0].imshow(data.hubble_deep_field())
axs[0, 1].imshow(data.immunohistochemistry())
axs[0, 2].imshow(data.lily())
axs[1, 0].imshow(data.microaneurysms())
axs[1, 1].imshow(data.moon(), cmap=plt.cm.gray)
axs[1, 2].imshow(data.retina())
axs[2, 0].imshow(data.shepp_logan_phantom(), cmap=plt.cm.gray)
axs[2, 1].imshow(data.skin())
further_img = np.full((300, 300), 255)
for xpos in [100, 150, 200]:
    further_img[150 - 10 : 150 + 10, xpos - 10 : xpos + 10] = 0
axs[2, 2].imshow(further_img, cmap=plt.cm.gray)
plt.subplots_adjust(wspace=-0.3, hspace=0.1)
```



Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0.. \leftrightarrow 255] for integers).

Total running time of the script: (0 minutes 2.784 seconds)

General-purpose images

The title of each image indicates the name of the function.

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

from skimage import data

matplotlib.rcParams['font.size'] = 18

images = ('astronaut',
          'binary_blobs',
          'brick',
          'colorwheel',
          'camera',
          'cat',
          'checkerboard',
```

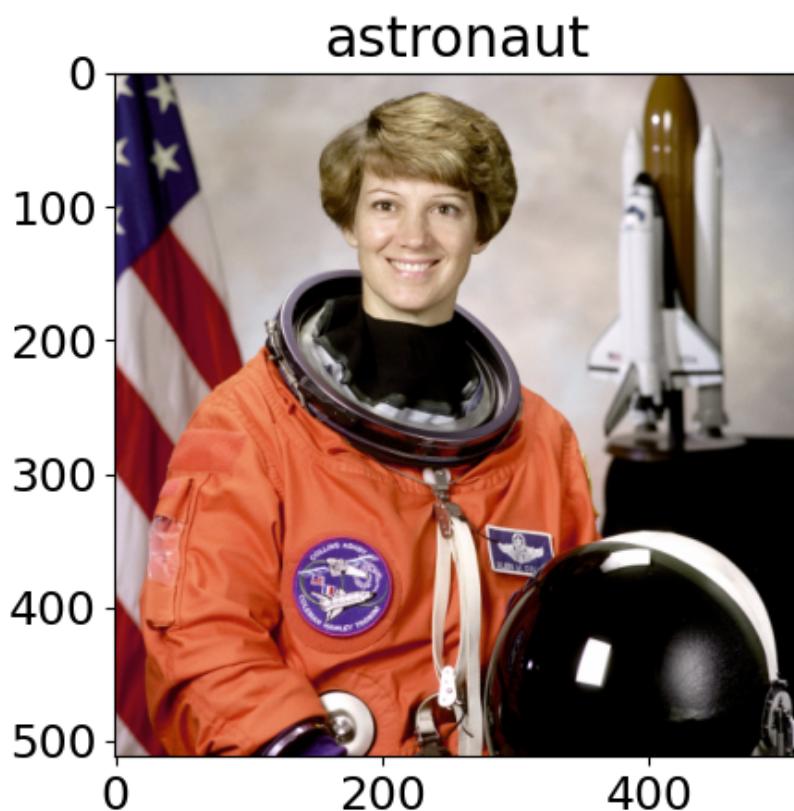
(continues on next page)

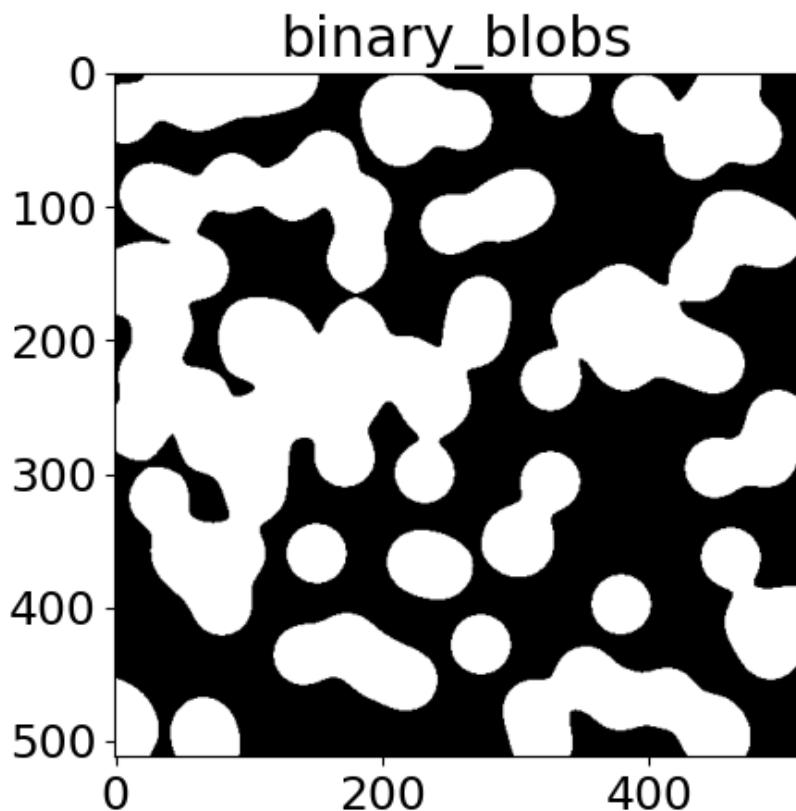
(continued from previous page)

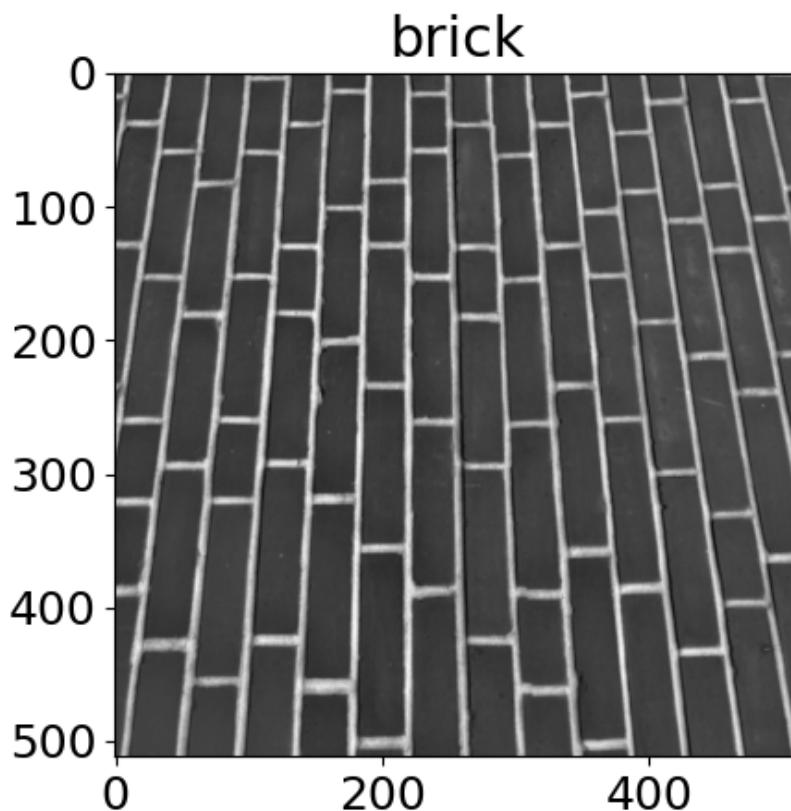
```
'clock',
'coffee',
'coins',
'eagle',
'grass',
'gravel',
'horse',
'logo',
'page',
'text',
'rocket',
)

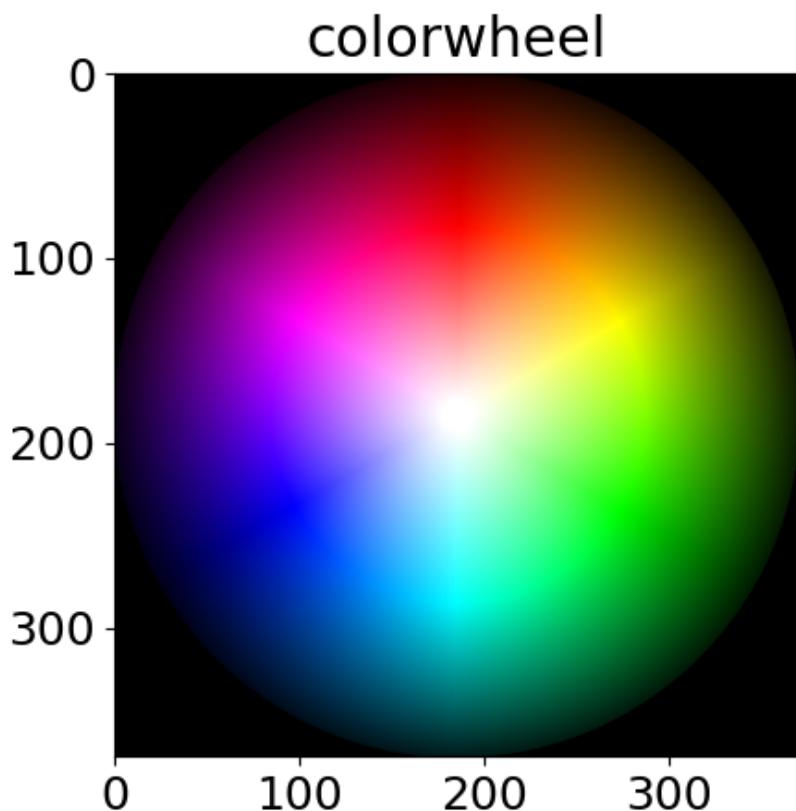
for name in images:
    caller = getattr(data, name)
    image = caller()
    plt.figure()
    plt.title(name)
    if image.ndim == 2:
        plt.imshow(image, cmap=plt.cm.gray)
    else:
        plt.imshow(image)

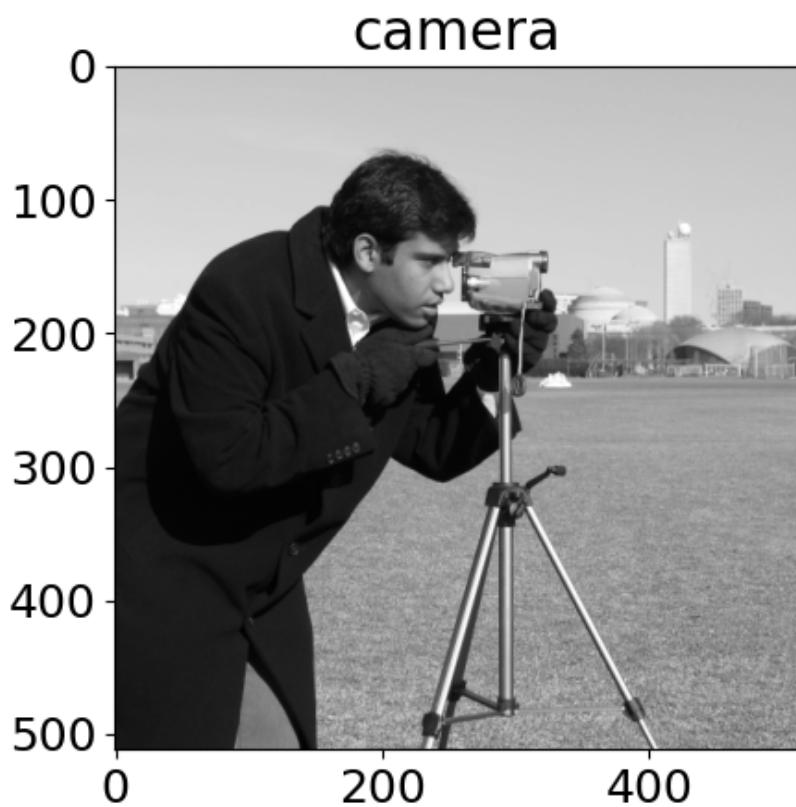
plt.show()
```

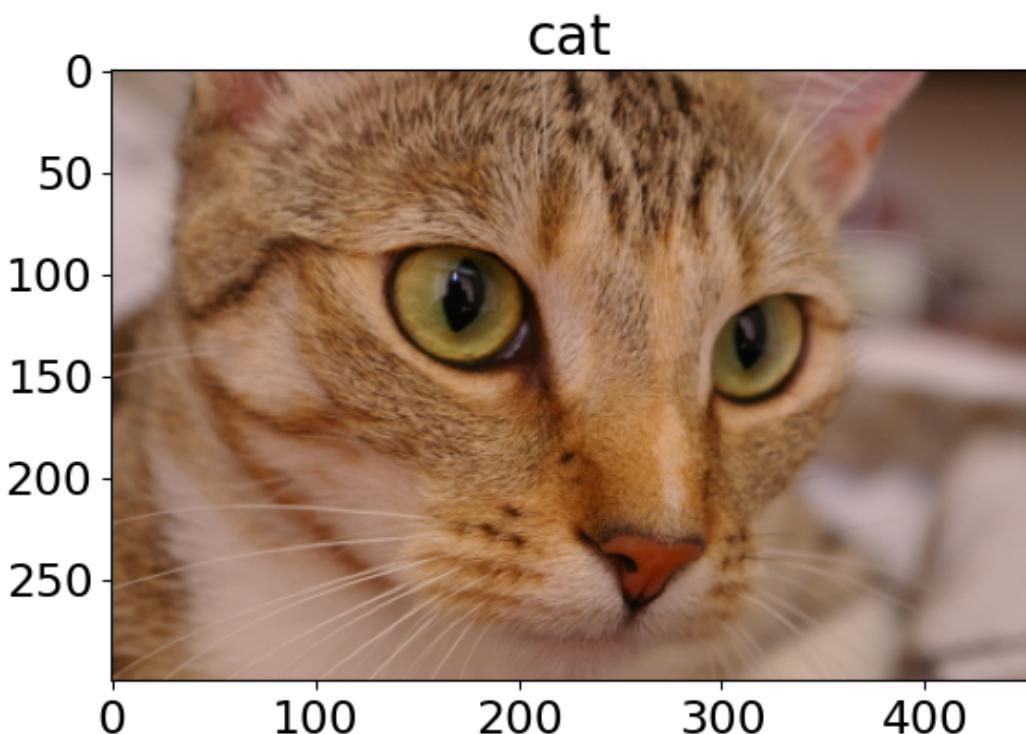


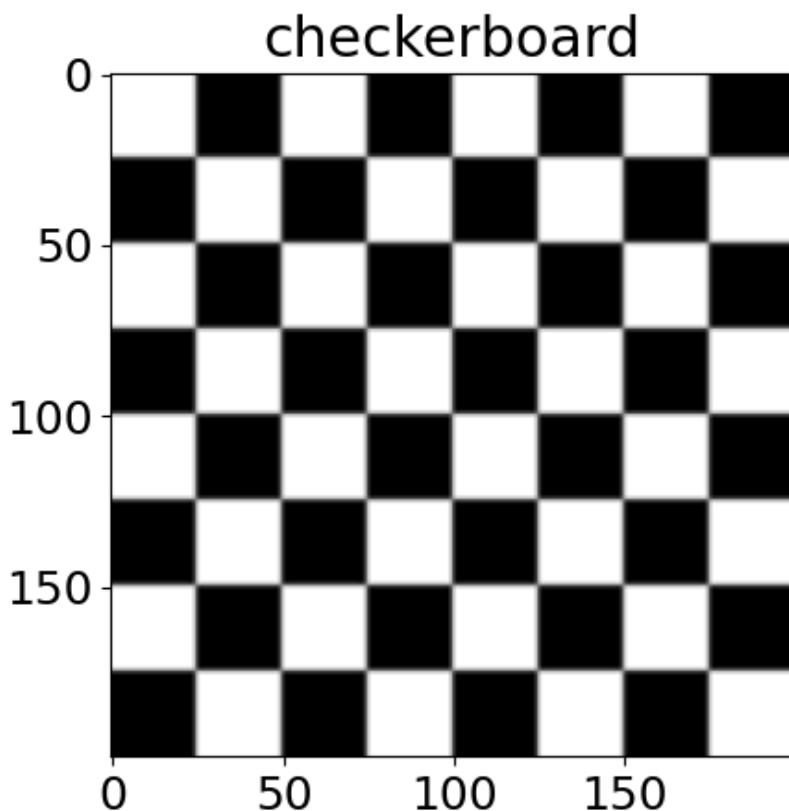


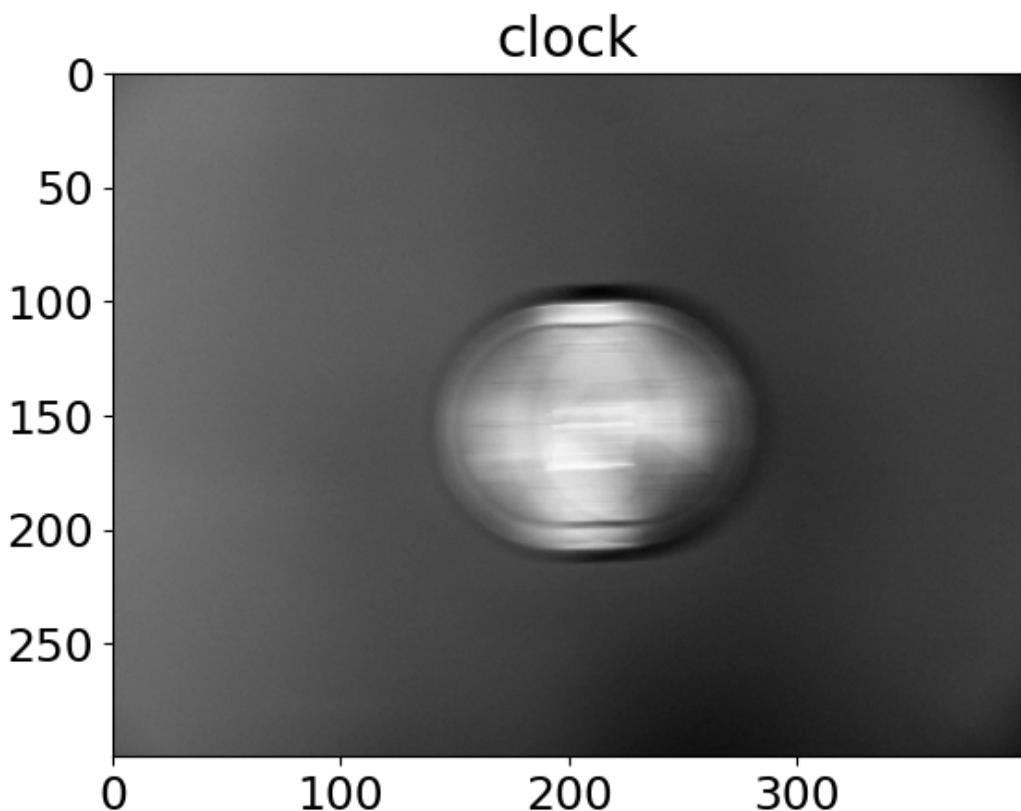


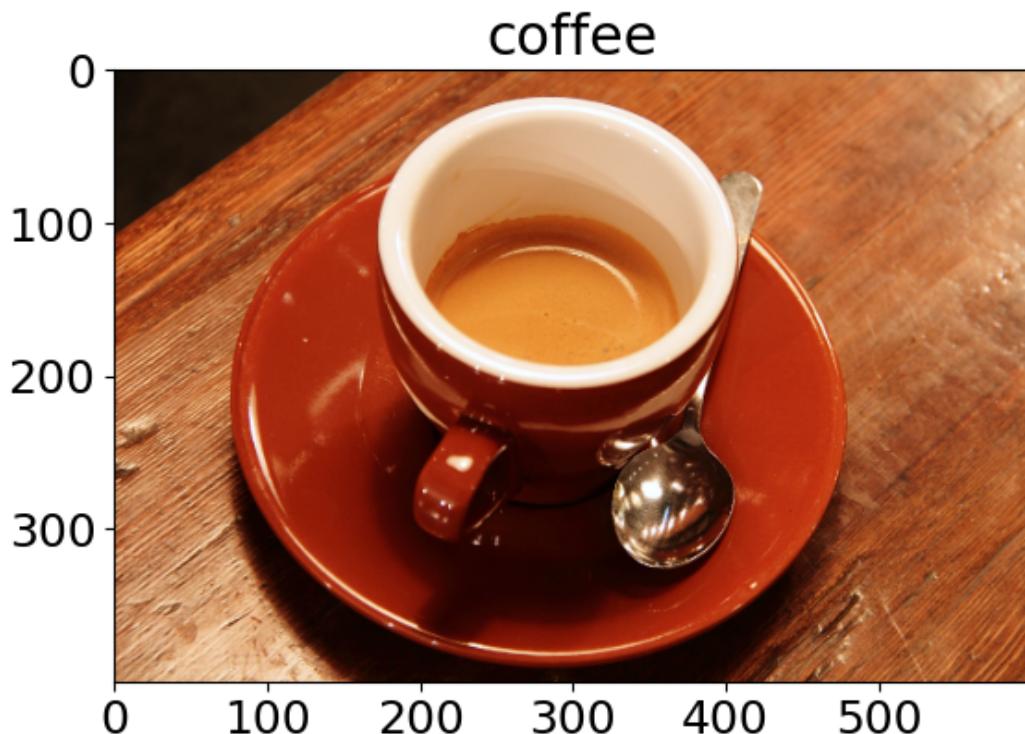


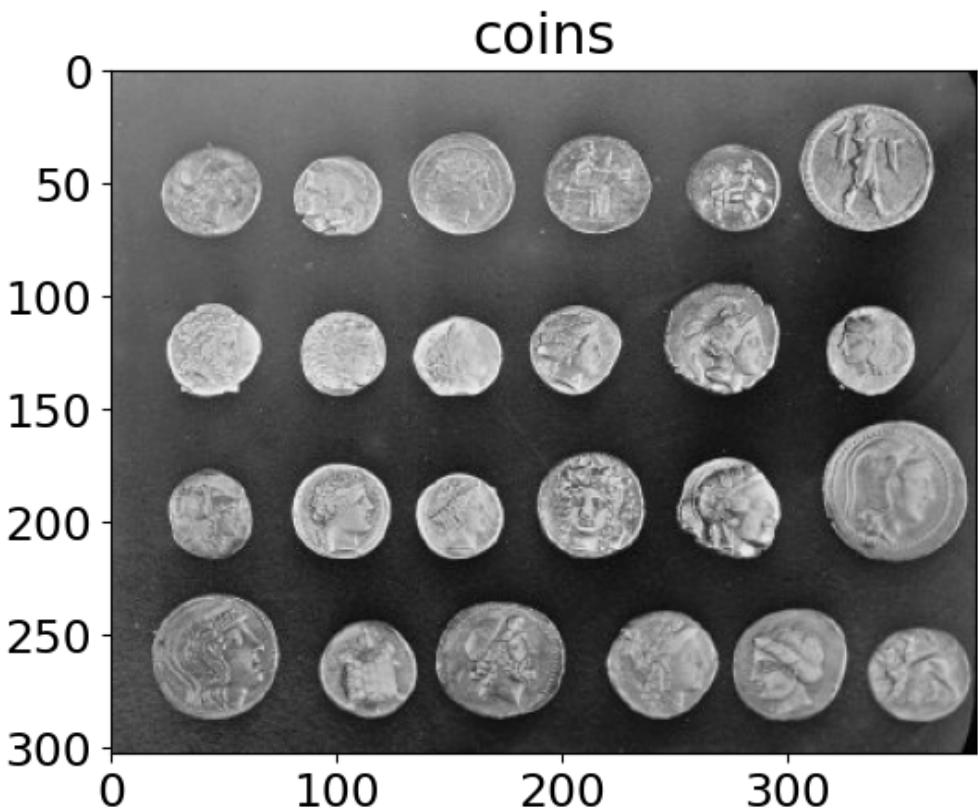


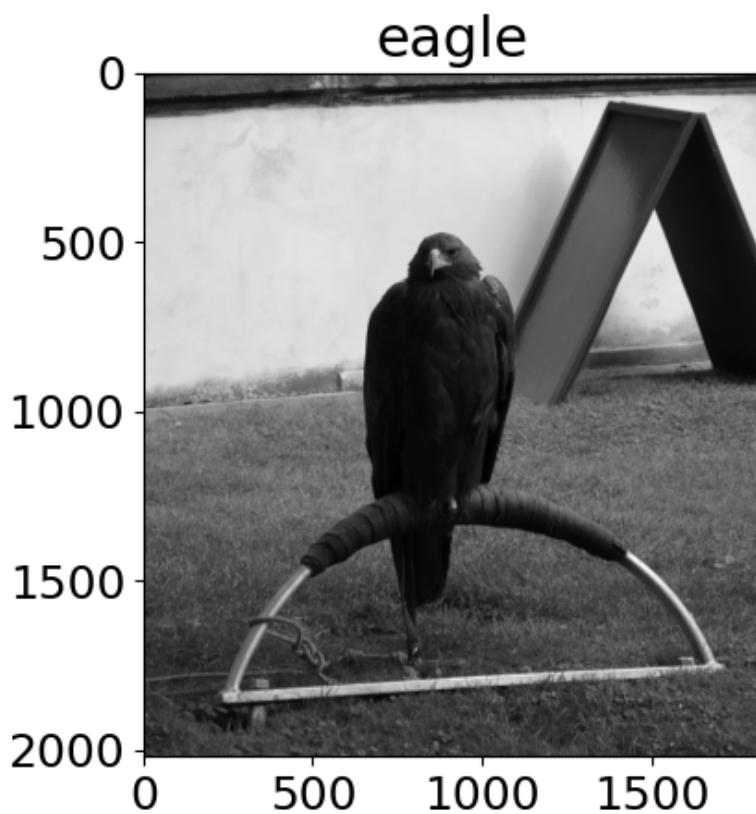


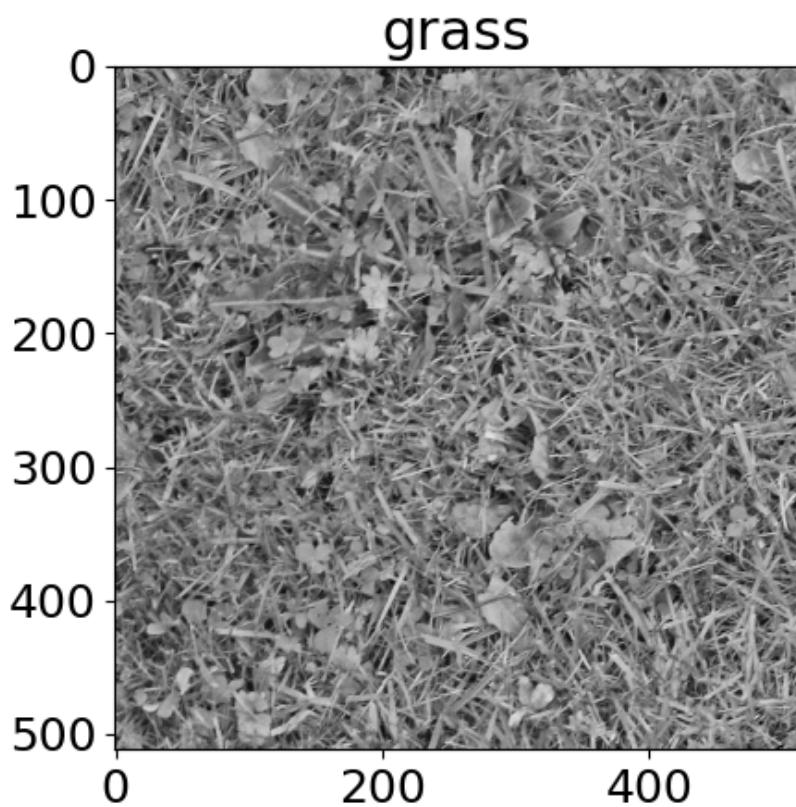


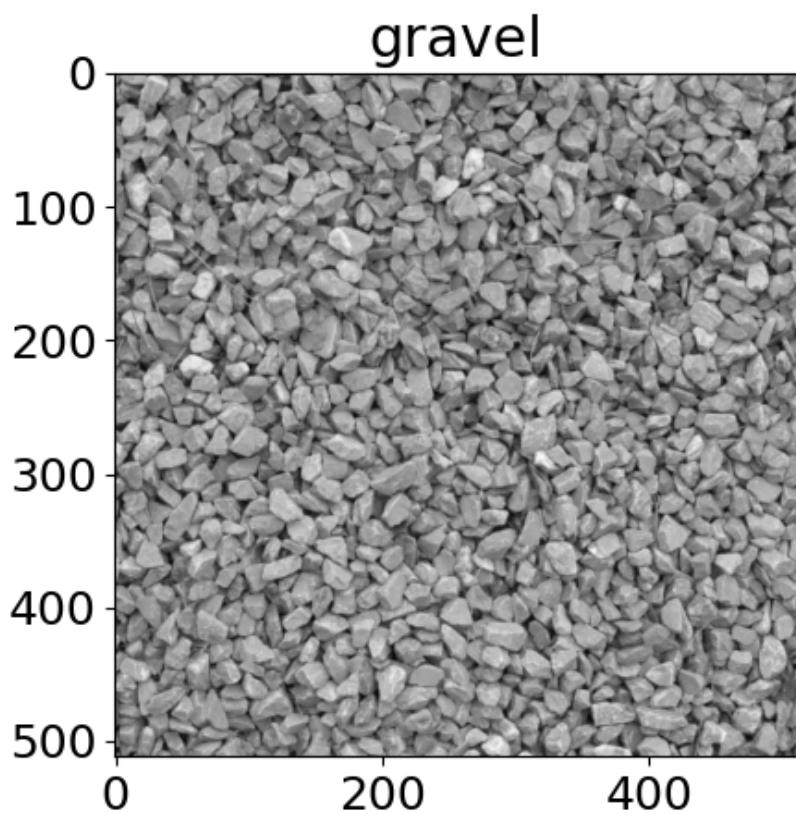


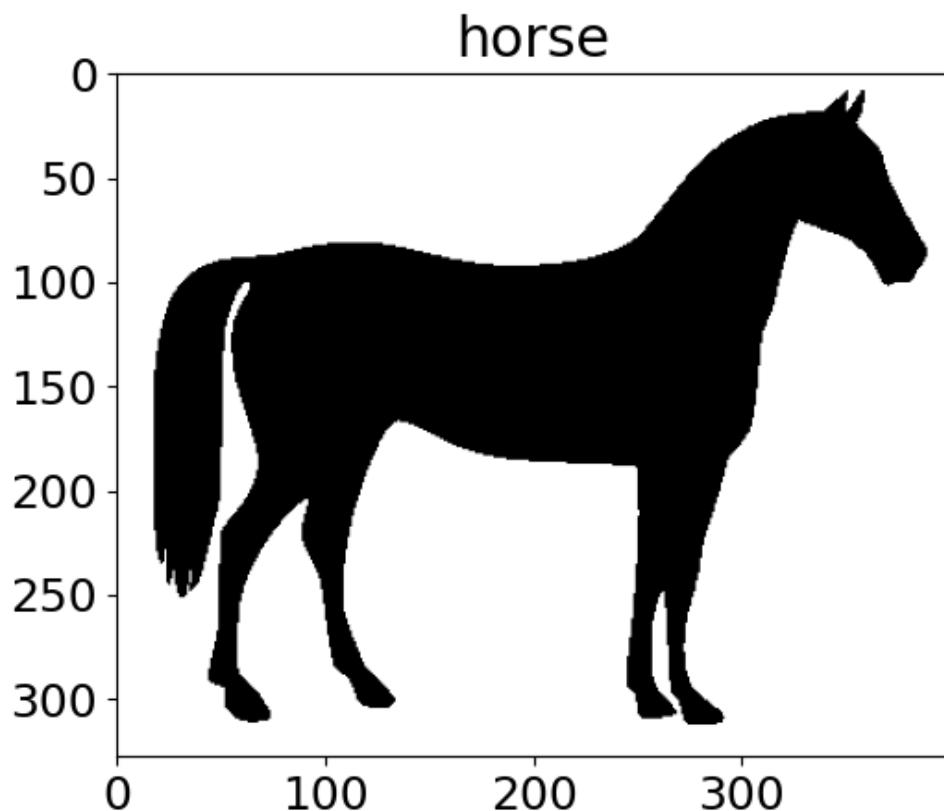


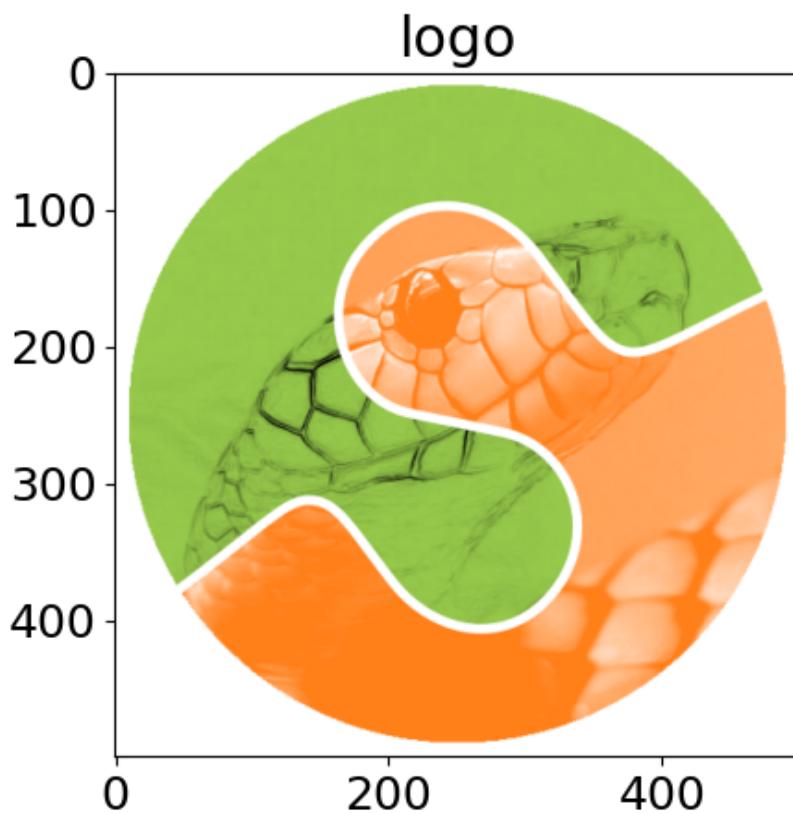


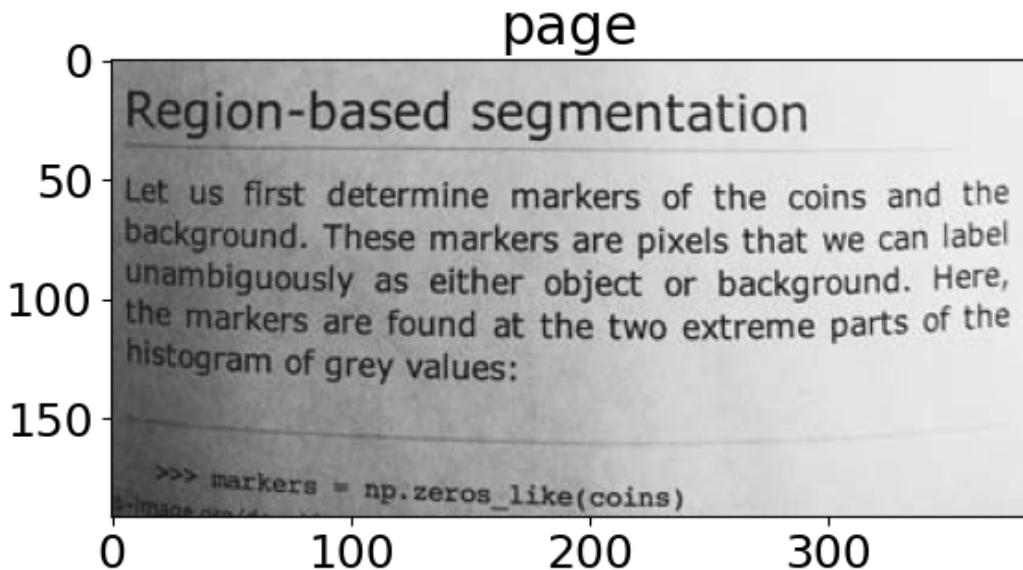


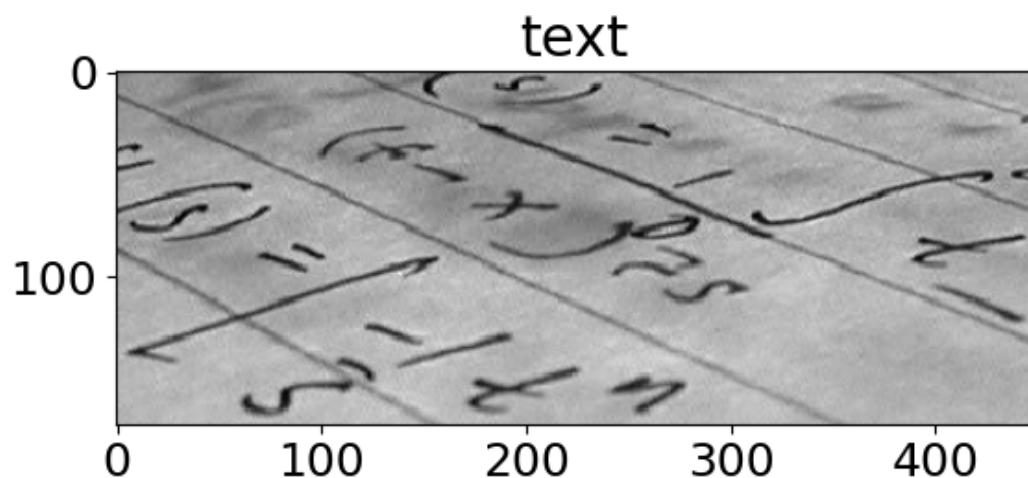


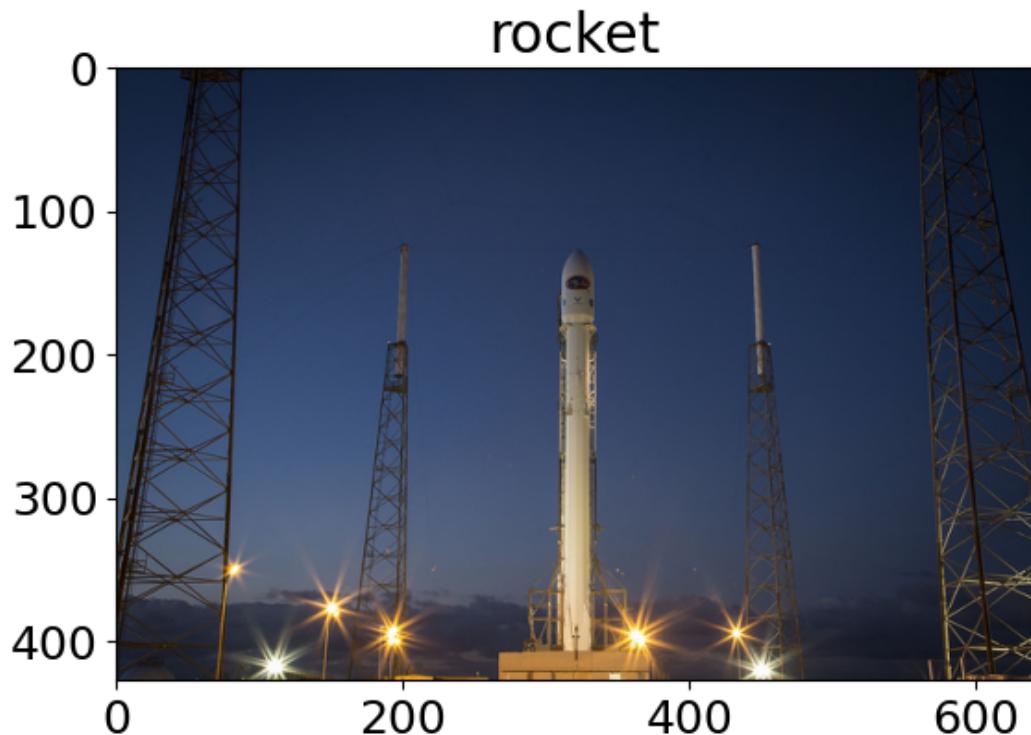






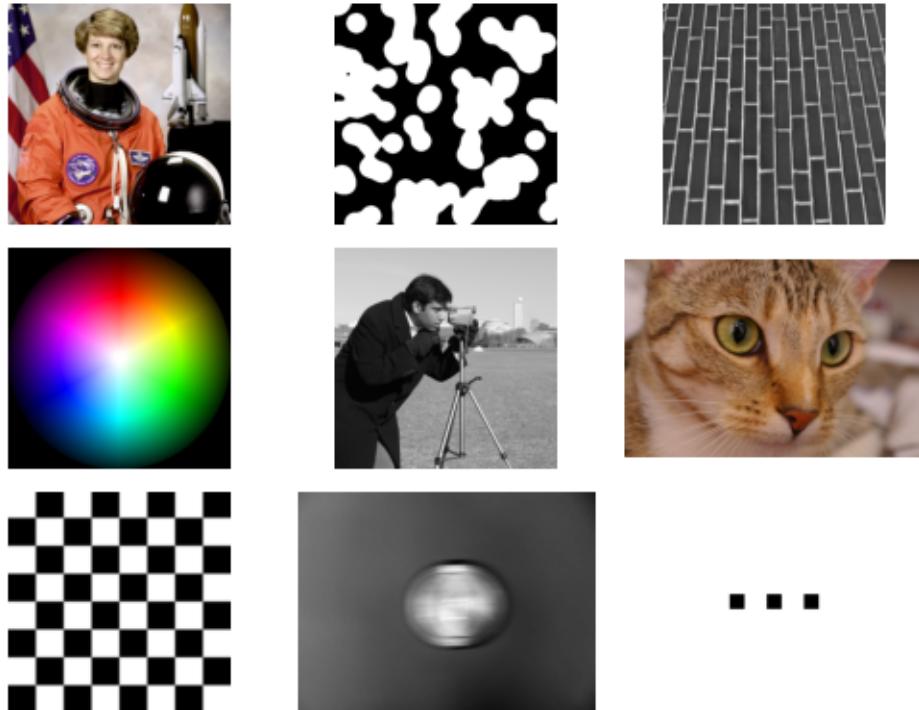






Thumbnail image for the gallery

```
fig, axs = plt.subplots(nrows=3, ncols=3)
for ax in axs.flat:
    ax.axis("off")
axs[0, 0].imshow(data.astronaut())
axs[0, 1].imshow(data.binary_blobs(), cmap=plt.cm.gray)
axs[0, 2].imshow(data.brick(), cmap=plt.cm.gray)
axs[1, 0].imshow(data.colorwheel())
axs[1, 1].imshow(data.camera(), cmap=plt.cm.gray)
axs[1, 2].imshow(data.cat())
axs[2, 0].imshow(data.checkerboard(), cmap=plt.cm.gray)
axs[2, 1].imshow(data.clock(), cmap=plt.cm.gray)
further_img = np.full((300, 300), 255)
for xpos in [100, 150, 200]:
    further_img[150 - 10 : 150 + 10, xpos - 10 : xpos + 10] = 0
axs[2, 2].imshow(further_img, cmap=plt.cm.gray)
plt.subplots_adjust(wspace=0.1, hspace=0.1)
```



Total running time of the script: (0 minutes 2.896 seconds)

Specific images

```
import matplotlib.pyplot as plt
import matplotlib

from skimage import data

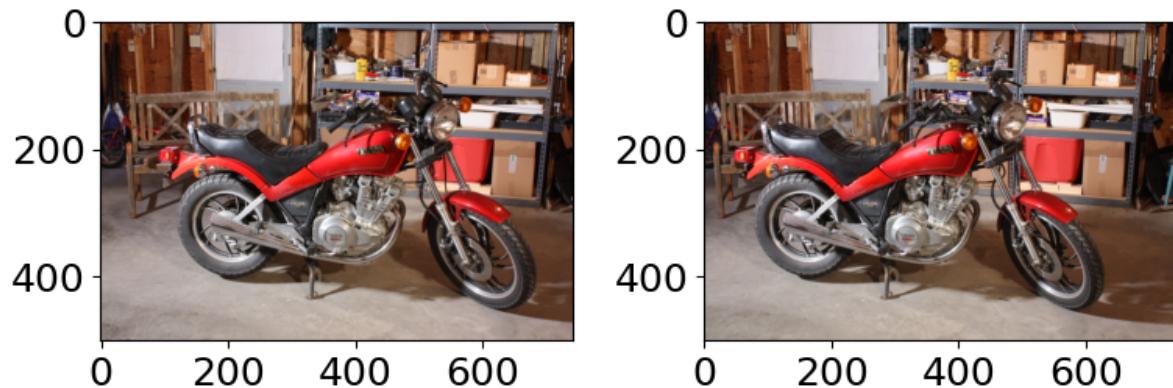
matplotlib.rcParams["font.size"] = 18
```

Stereo images

```
fig, axes = plt.subplots(1, 2, figsize=(8, 4))
ax = axes.ravel()

cycle_images = data.stereo_motorcycle()
ax[0].imshow(cycle_images[0])
ax[1].imshow(cycle_images[1])

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

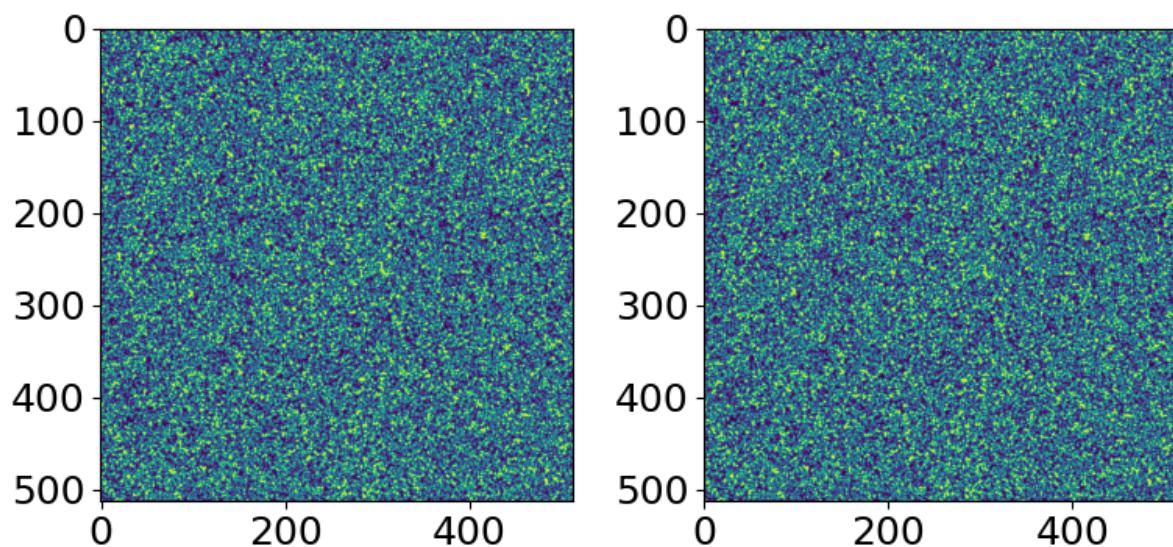


PIV images

```
fig, axes = plt.subplots(1, 2, figsize=(8, 4))
ax = axes.ravel()

vortex_images = data.vortex()
ax[0].imshow(vortex_images[0])
ax[1].imshow(vortex_images[1])

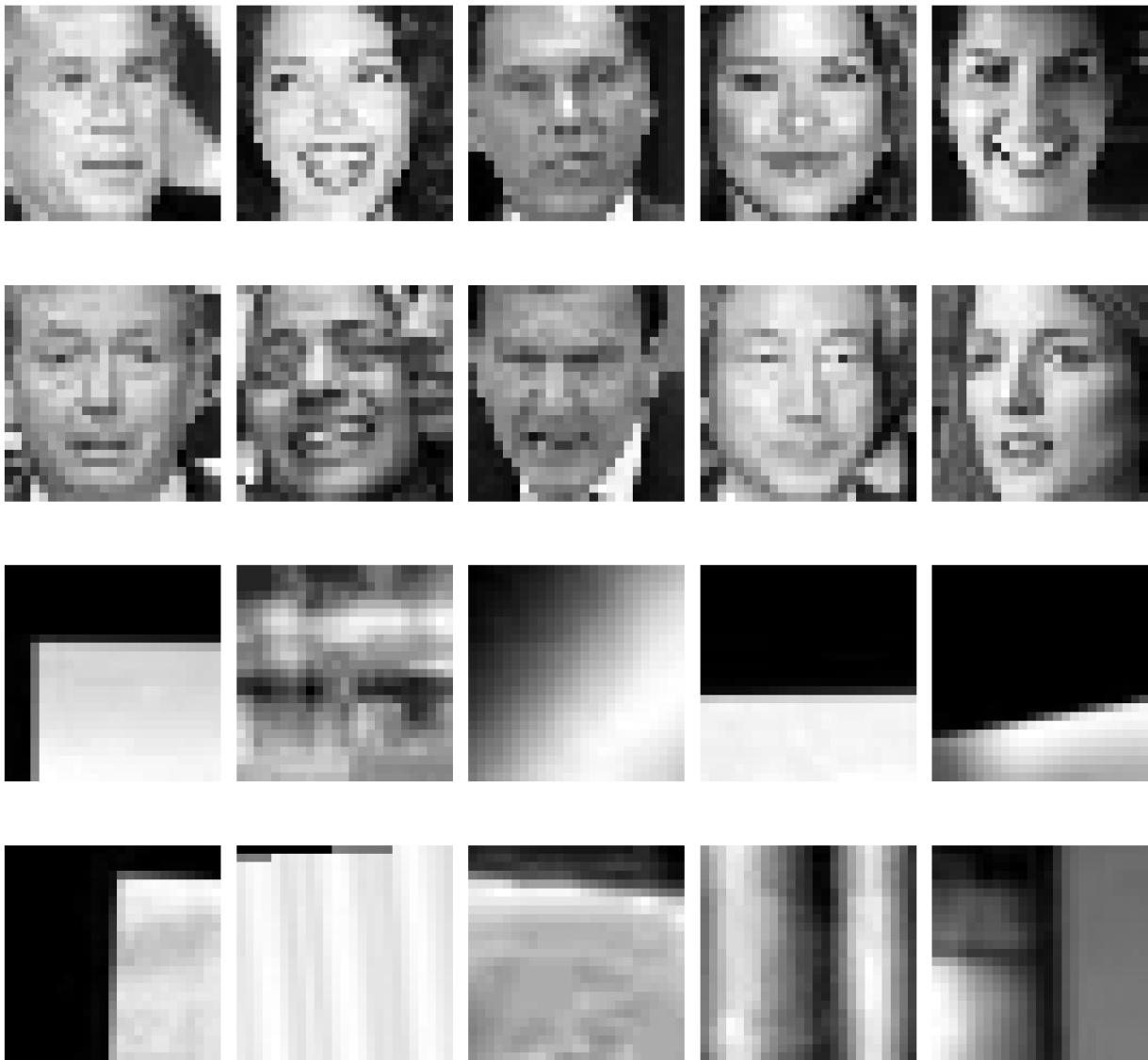
fig.tight_layout()
plt.show()
```



Faces and non-faces dataset

A sample of 20 over 200 images is displayed.

```
fig, axes = plt.subplots(4, 5, figsize=(20, 20))
ax = axes.ravel()
lfw_images = data.lfw_subset()
for i in range(20):
    ax[i].imshow(lfw_images[90 + i], cmap=plt.cm.gray)
    ax[i].axis("off")
fig.tight_layout()
plt.show()
```

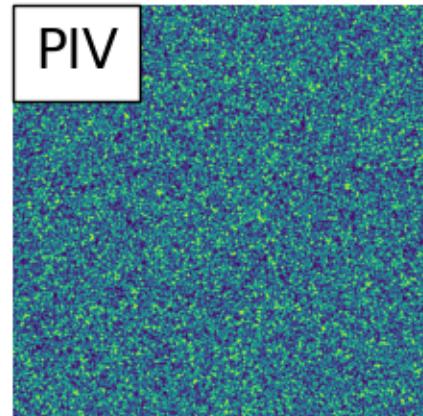


Thumbnail image for the gallery

```
from matplotlib.offsetbox import AnchoredText

# Create a gridspec with two images in the first and 4 in the second row
fig, axd = plt.subplot_mosaic(
    [["stereo", "stereo", "piv", "piv"], ["lfw0", "lfw1", "lfw2", "lfw3"]], 
)
axd["stereo"].imshow(cycle_images[0])
axd["stereo"].add_artist(
    AnchoredText(
        "Stereo",
        prop=dict(size=20),
        frameon=True,
        borderpad=0,
        loc="upper left",
    )
)
axd["piv"].imshow(vortex_images[0])
axd["piv"].add_artist(
    AnchoredText(
        "PIV",
        prop=dict(size=20),
        frameon=True,
        borderpad=0,
        loc="upper left",
    )
)
axd["lfw0"].imshow(lfw_images[91], cmap="gray")
axd["lfw1"].imshow(lfw_images[92], cmap="gray")
axd["lfw2"].imshow(lfw_images[93], cmap="gray")
axd["lfw3"].imshow(lfw_images[94], cmap="gray")

for ax in axd.values():
    ax.axis("off")
fig.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 1.852 seconds)

Operations on NumPy arrays

Using simple NumPy operations for manipulating images

This script illustrates how to use basic NumPy operations, such as slicing, masking and fancy indexing, in order to modify the pixel values of an image.



```
import numpy as np
from skimage import data
import matplotlib.pyplot as plt

camera = data.camera()
camera[:10] = 0
mask = camera < 87
camera[mask] = 255
inds_x = np.arange(len(camera))
inds_y = (4 * inds_x) % len(camera)
camera[inds_x, inds_y] = 0

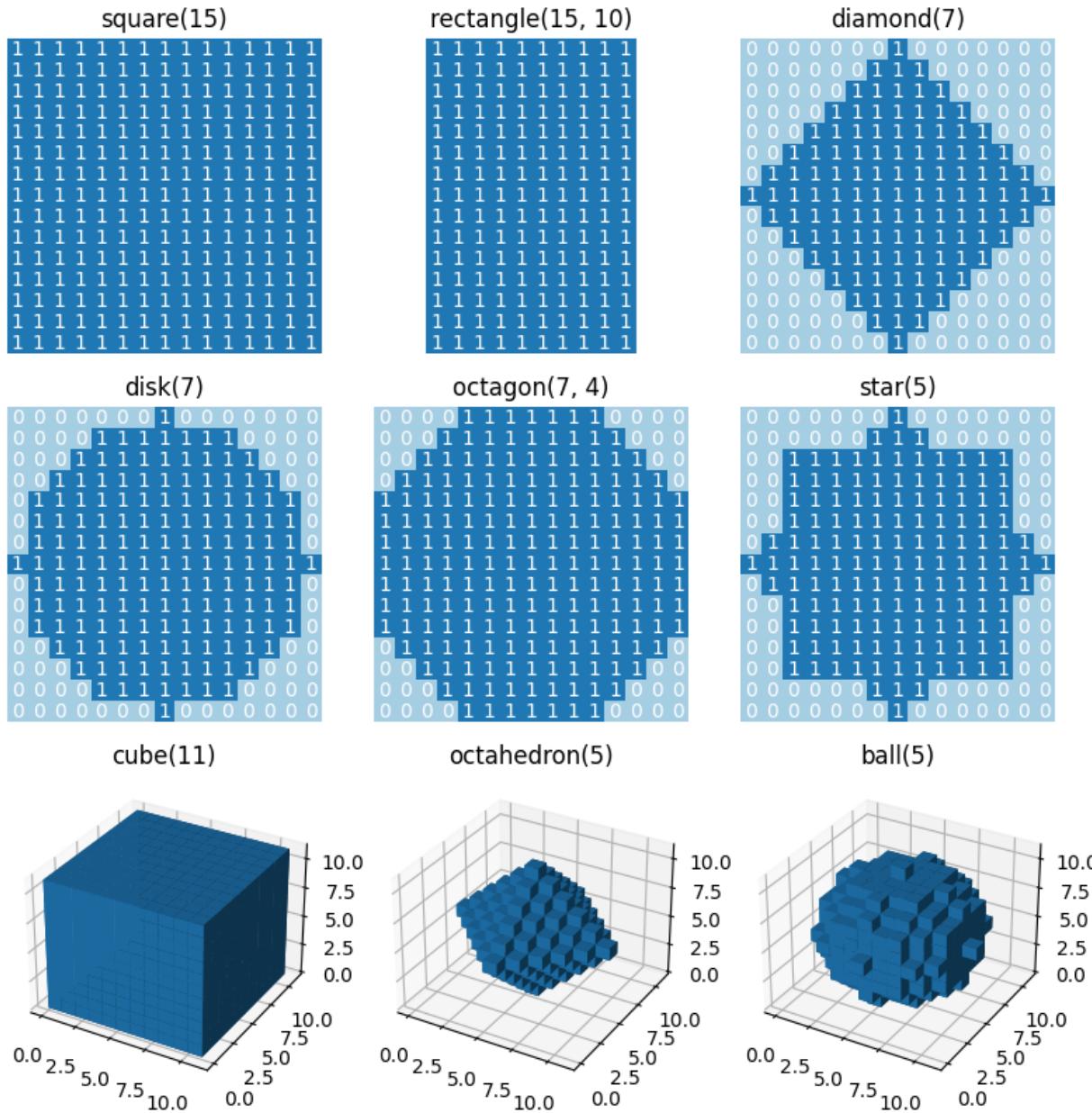
l_x, l_y = camera.shape[0], camera.shape[1]
X, Y = np.ogrid[:l_x, :l_y]
outer_disk_mask = (X - l_x / 2)**2 + (Y - l_y / 2)**2 > (l_x / 2)**2
camera[outer_disk_mask] = 0

plt.figure(figsize=(4, 4))
plt.imshow(camera, cmap='gray')
plt.axis('off')
plt.show()
```

Total running time of the script: (0 minutes 0.060 seconds)

Generate footprints (structuring elements)

This example shows how to use functions in `skimage.morphology` to generate footprints (structuring elements) for use in morphology operations. The title of each plot indicates the call of the function.



```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

from skimage.morphology import (square, rectangle, diamond, disk, cube,
                                octahedron, ball, octagon, star)

# Generate 2D and 3D structuring elements.
struc_2d = {
    "square(15)": square(15),
```

(continues on next page)

(continued from previous page)

```
"rectangle(15, 10)": rectangle(15, 10),
"diamond(7)": diamond(7),
"disk(7)": disk(7),
"octagon(7, 4)": octagon(7, 4),
"star(5)": star(5)
}

struc_3d = {
    "cube(11)": cube(11),
    "octahedron(5)": octahedron(5),
    "ball(5)": ball(5)
}

# Visualize the elements.
fig = plt.figure(figsize=(8, 8))

idx = 1
for title, struc in struc_2d.items():
    ax = fig.add_subplot(3, 3, idx)
    ax.imshow(struc, cmap="Paired", vmin=0, vmax=12)
    for i in range(struc.shape[0]):
        for j in range(struc.shape[1]):
            ax.text(j, i, struc[i, j], ha="center", va="center", color="w")
    ax.set_axis_off()
    ax.set_title(title)
    idx += 1

for title, struc in struc_3d.items():
    ax = fig.add_subplot(3, 3, idx, projection=Axes3D.name)
    ax.voxels(struc)
    ax.set_title(title)
    idx += 1

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

Total running time of the script: (0 minutes 3.053 seconds)

Block views on images/arrays

This example illustrates the use of `view_as_blocks` from `skimage.util()`. Block views can be incredibly useful when one wants to perform local operations on non-overlapping image patches.

We use `astronaut` from `skimage.data` and virtually ‘slice’ it into square blocks. Then, on each block, we either pool the mean, the max or the median value of that block. The results are displayed altogether, along with a spline interpolation of order 3 rescaling of the original `astronaut` image.

Original rescaled with
spline interpolation (order=3)



Block view with
local mean pooling



Block view with
local max pooling



Block view with
local median pooling



```

import numpy as np
from scipy import ndimage as ndi
from matplotlib import pyplot as plt
import matplotlib.cm as cm

from skimage import data
from skimage import color
from skimage.util import view_as_blocks

# get astronaut from skimage.data in grayscale
l = color.rgb2gray(data.astronaut())

```

(continues on next page)

(continued from previous page)

```

# size of blocks
block_shape = (4, 4)

# see astronaut as a matrix of blocks (of shape block_shape)
view = view_as_blocks(l, block_shape)

# collapse the last two dimensions in one
flatten_view = view.reshape(view.shape[0], view.shape[1], -1)

# resampling the image by taking either the `mean`,
# the `max` or the `median` value of each blocks.
mean_view = np.mean(flatten_view, axis=2)
max_view = np.max(flatten_view, axis=2)
median_view = np.median(flatten_view, axis=2)

# display resampled images
fig, axes = plt.subplots(2, 2, figsize=(8, 8), sharex=True, sharey=True)
ax = axes.ravel()

l_resized = ndi.zoom(l, 2, order=3)
ax[0].set_title("Original rescaled with\n spline interpolation (order=3)")
ax[0].imshow(l_resized, extent=(-0.5, 128.5, 128.5, -0.5),
             cmap=cm.Greys_r)

ax[1].set_title("Block view with\n local mean pooling")
ax[1].imshow(mean_view, cmap=cm.Greys_r)

ax[2].set_title("Block view with\n local max pooling")
ax[2].imshow(max_view, cmap=cm.Greys_r)

ax[3].set_title("Block view with\n local median pooling")
ax[3].imshow(median_view, cmap=cm.Greys_r)

for a in ax:
    a.set_axis_off()

fig.tight_layout()
plt.show()

```

Total running time of the script: (0 minutes 0.614 seconds)

Decompose flat footprints (structuring elements)

Many footprints (structuring elements) can be decomposed into an equivalent series of smaller structuring elements. The term “flat” refers to footprints that only contain values of 0 or 1 (i.e., all methods in `skimage.morphology.footprints`). Binary dilation operations have an associative and distributive property that often allows decomposition into an equivalent series of smaller footprints. Most often this is done to provide a performance benefit.

As a concrete example, dilation with a square footprint of shape (15, 15) is equivalent to dilation with a rectangle of shape (15, 1) followed by another dilation with a rectangle of shape (1, 15). It is also equivalent to 7 consecutive dilations with a square footprint of shape (3, 3).

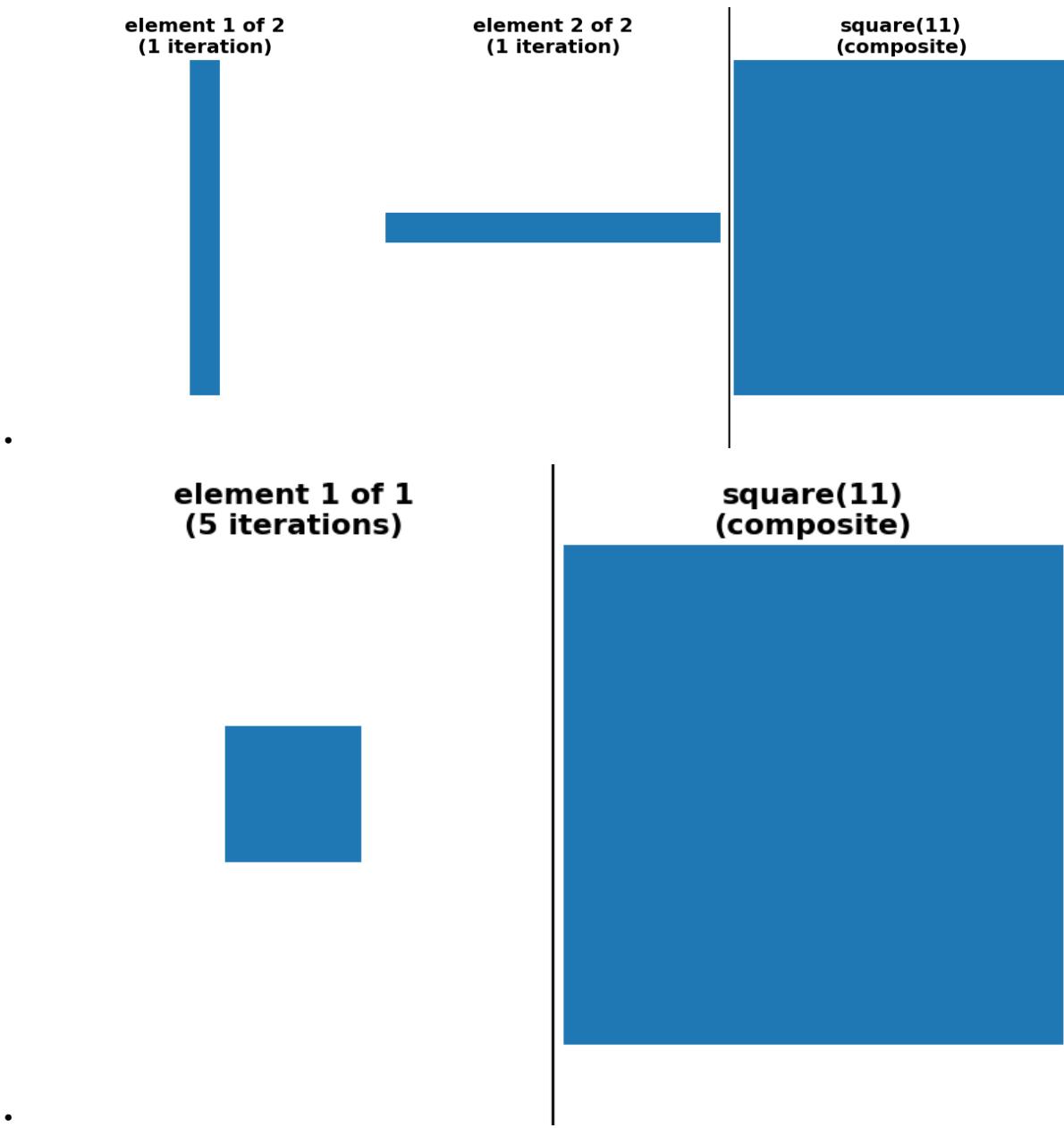
There are many possible decompositions and which one performs best may be architecture-dependent.

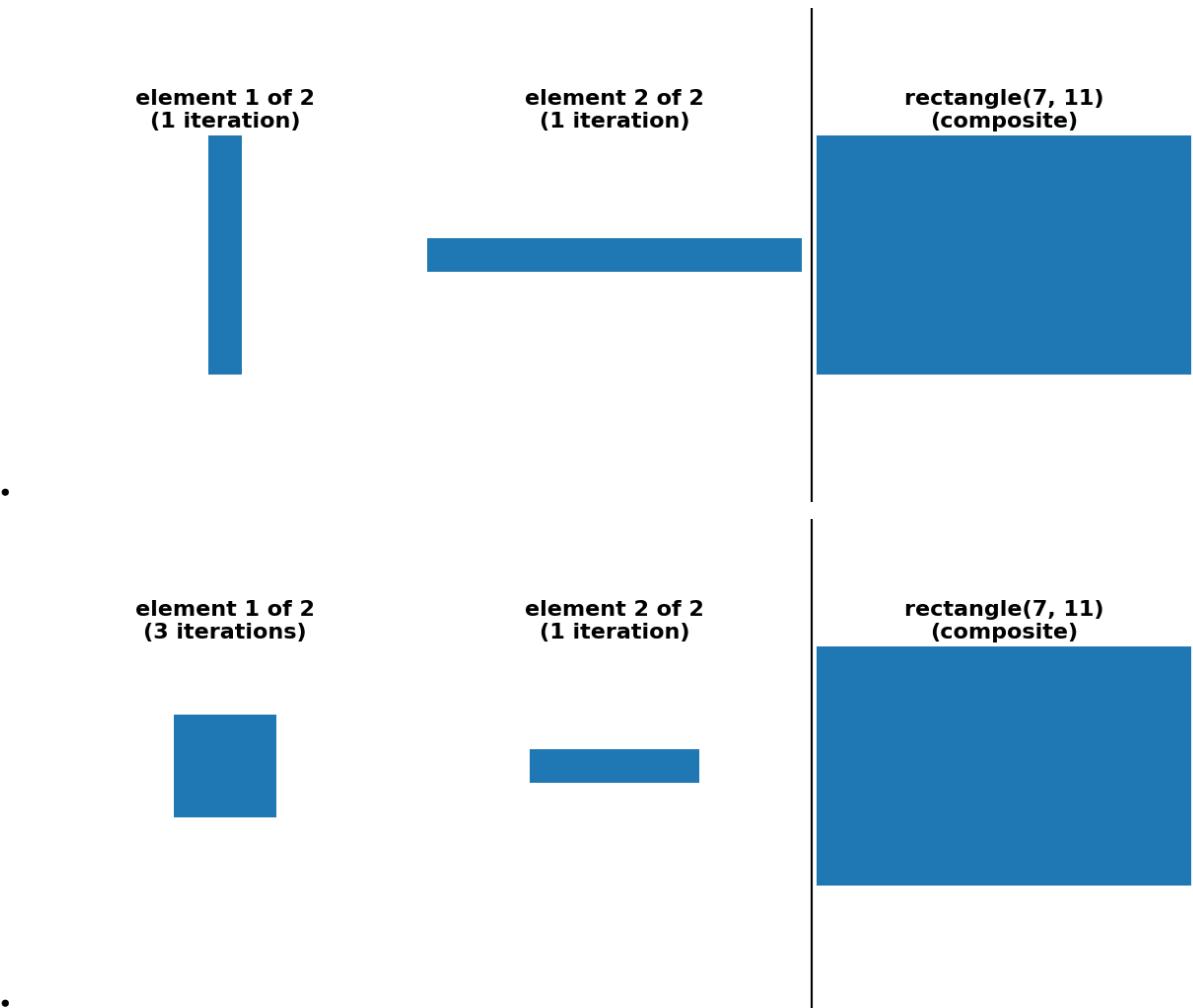
scikit-image currently provides two forms of automated decomposition. For the cases of `square`, `rectangle` and

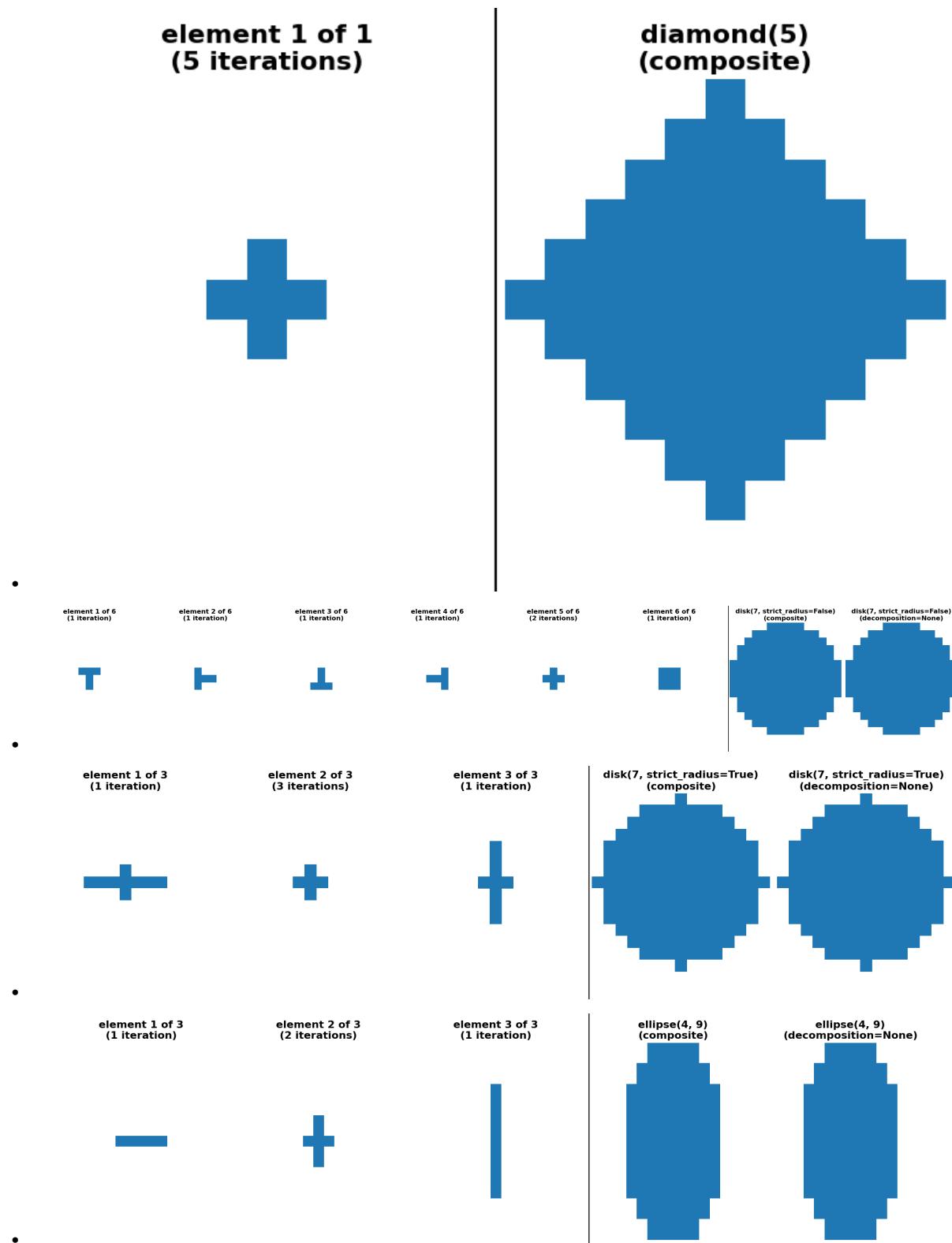
cube footprints, there is an option for a “separable” decomposition (size > 1 along only one axis at a time).

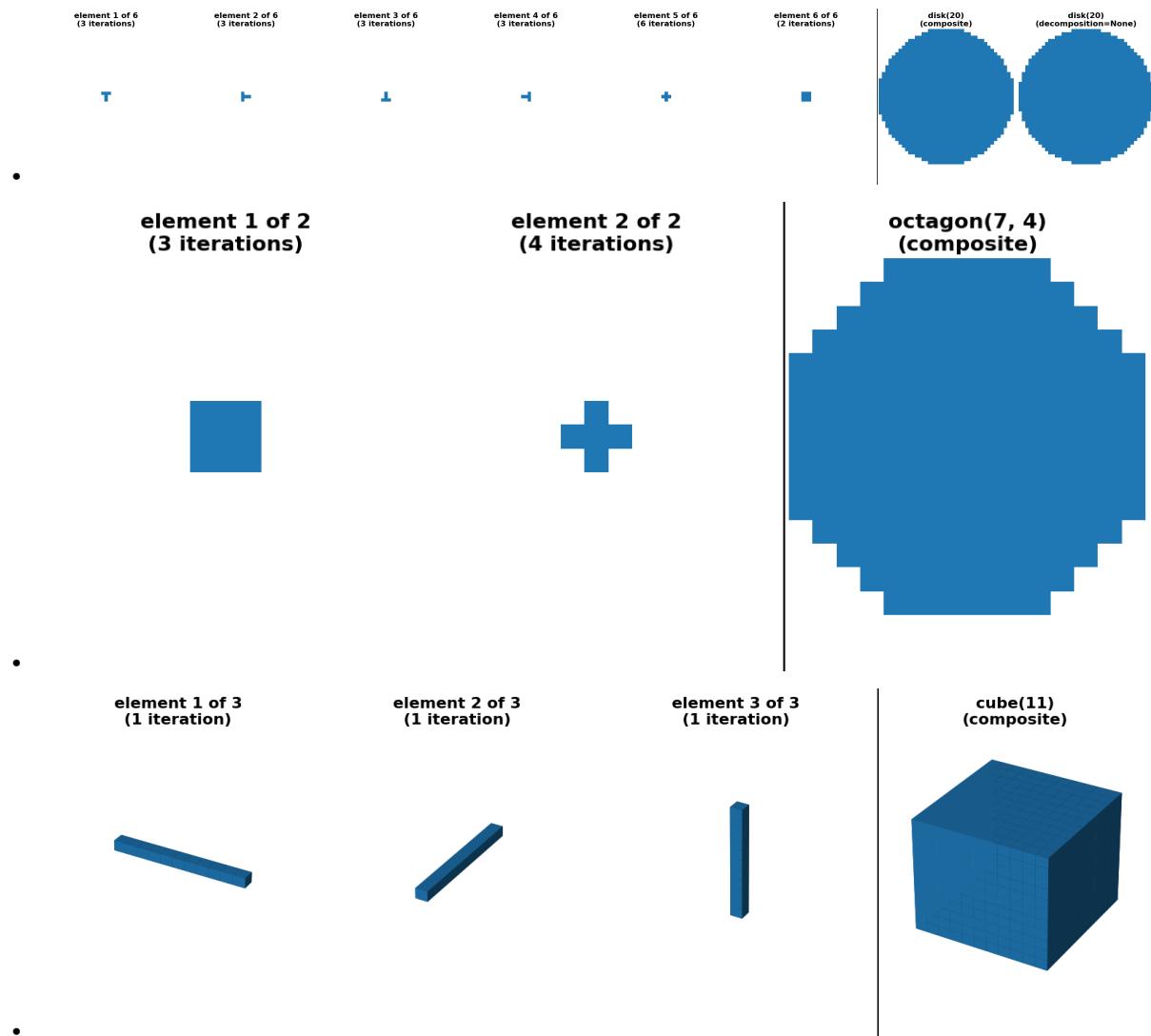
There is no separable decomposition into 1D operations for some other symmetric convex shapes, e.g., diamond, octahedron and octagon. However, it is possible to provide a “sequence” decomposition based on a series of small footprints of shape $(3,) * \text{ndim}$.

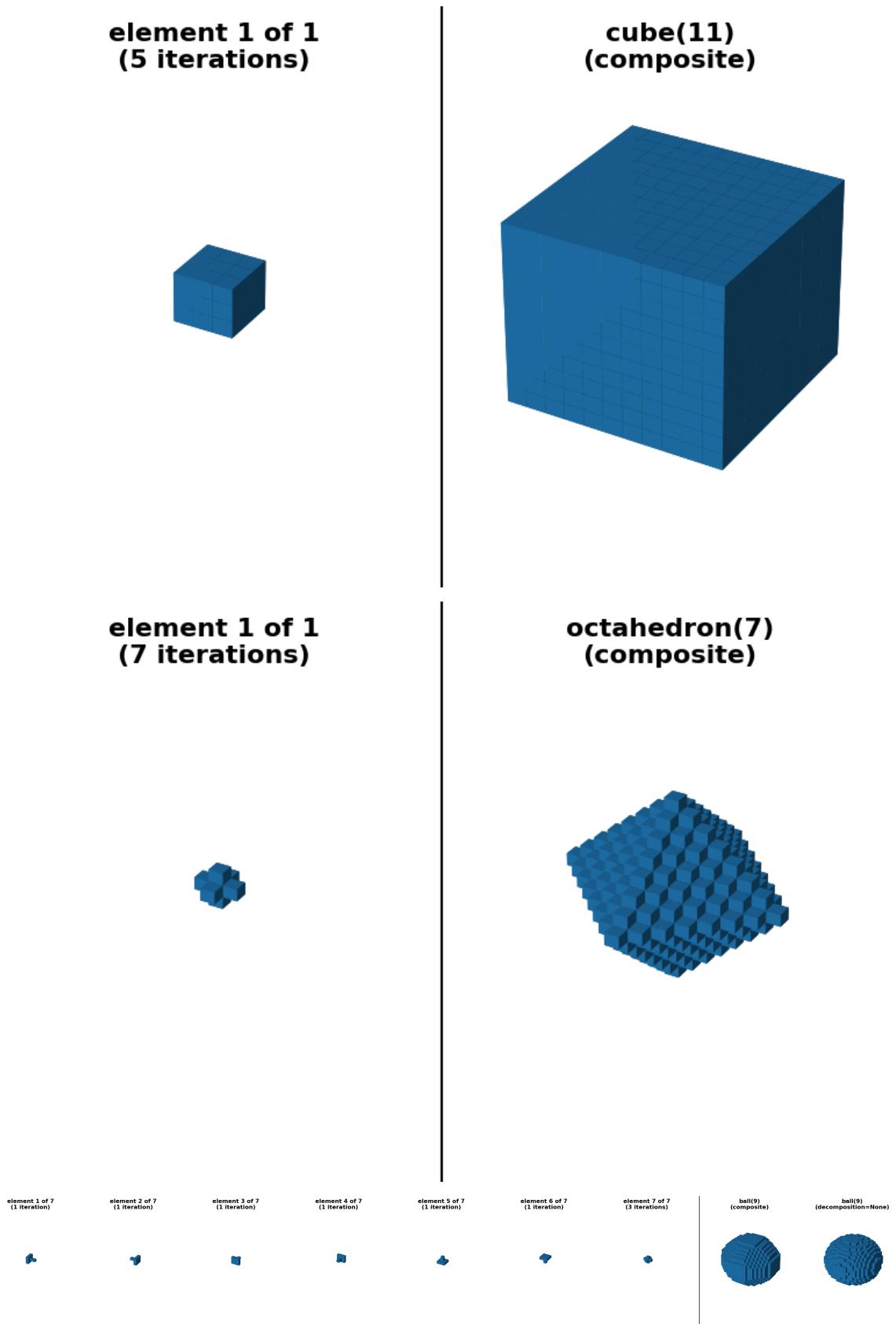
For simplicity of implementation, all decompositions shown below use only odd-sized footprints with their origin located at the center of the footprint.











```

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import colors
from mpl_toolkits.mplot3d import Axes3D

from skimage.morphology import (ball, cube, diamond, disk, ellipse, octagon,
                                 octahedron, rectangle, square)
from skimage.morphology.footprints import footprint_from_sequence

# Generate 2D and 3D structuring elements.
footprint_dict = {
    "square(11) (separable)": (square(11, decomposition=None),
                                square(11, decomposition="separable")),
    "square(11) (sequence)": (square(11, decomposition=None),
                              square(11, decomposition="sequence")),
    "rectangle(7, 11) (separable)": (rectangle(7, 11, decomposition=None),
                                     rectangle(7, 11,
                                               decomposition="separable")),
    "rectangle(7, 11) (sequence)": (rectangle(7, 11, decomposition=None),
                                    rectangle(7, 11,
                                              decomposition="sequence")),
    "diamond(5) (sequence)": (diamond(5, decomposition=None),
                             diamond(5, decomposition="sequence")),
    "disk(7, strict_radius=False) (sequence)": (
        disk(7, strict_radius=False, decomposition=None),
        disk(7, strict_radius=False, decomposition="sequence"))
    ),
    "disk(7, strict_radius=True) (crosses)": (
        disk(7, strict_radius=True, decomposition=None),
        disk(7, strict_radius=True, decomposition="crosses"))
    ),
    "ellipse(4, 9) (crosses)": (
        ellipse(4, 9, decomposition=None),
        ellipse(4, 9, decomposition="crosses"))
    ),
    "disk(20) (sequence)": (disk(20, strict_radius=False, decomposition=None),
                           disk(20, strict_radius=False,
                                 decomposition="sequence")),
    "octagon(7, 4) (sequence)": (octagon(7, 4, decomposition=None),
                                 octagon(7, 4, decomposition="sequence")),
    "cube(11) (separable)": (cube(11, decomposition=None),
                             cube(11, decomposition="separable")),
    "cube(11) (sequence)": (cube(11, decomposition=None),
                           cube(11, decomposition="sequence")),
    "octahedron(7) (sequence)": (octahedron(7, decomposition=None),
                                 octahedron(7, decomposition="sequence")),
    "ball(9) (sequence)": (ball(9, strict_radius=False, decomposition=None),
                          ball(9, strict_radius=False,
                               decomposition="sequence")),
}

```

(continues on next page)

(continued from previous page)

```
# Visualize the elements

# binary white / blue colormap
cmap = colors.ListedColormap(['white', (0.1216, 0.4706, 0.70588)])

fontdict = dict(fontsize=16, fontweight='bold')
for title, (footprint, footprint_sequence) in footprint_dict.items():
    ndim = footprint.ndim
    num_seq = len(footprint_sequence)
    approximate_decomposition = (
        'ball' in title or 'disk' in title or 'ellipse' in title
    )
    if approximate_decomposition:
        # Two extra plot in approximate cases to show both:
        # 1.) decomposition=None idea footprint
        # 2.) actual composite footprint corresponding to the sequence
        num_subplots = num_seq + 2
    else:
        # composite and decomposition=None are identical so only 1 extra plot
        num_subplots = num_seq + 1
    fig = plt.figure(figsize=(4 * num_subplots, 5))
    if ndim == 2:
        ax = fig.add_subplot(1, num_subplots, num_subplots)
        ax.imshow(footprint, cmap=cmap, vmin=0, vmax=1)
        if approximate_decomposition:
            ax2 = fig.add_subplot(1, num_subplots, num_subplots - 1)
            footprint_composite = footprint_from_sequence(footprint_sequence)
            ax2.imshow(footprint_composite, cmap=cmap, vmin=0, vmax=1)

    else:
        ax = fig.add_subplot(1, num_subplots, num_subplots,
                            projection=Axes3D.name)
        ax.voxels(footprint, cmap=cmap)
        if approximate_decomposition:
            ax2 = fig.add_subplot(1, num_subplots, num_subplots - 1,
                                 projection=Axes3D.name)
            footprint_composite = footprint_from_sequence(footprint_sequence)
            ax2.voxels(footprint_composite, cmap=cmap)

    title1 = title.split(' ')[0]
    if approximate_decomposition:
        # plot decomposition=None on a separate axis from the composite
        title = title1 + '\n(decomposition=None)'
    else:
        # for exact cases composite and decomposition=None are identical
        title = title1 + '\n(composite)'
    ax.set_title(title, fontdict=fontdict)
    ax.set_axis_off()
    if approximate_decomposition:
        ax2.set_title(title1 + '\n(composite)', fontdict=fontdict)
        ax2.set_axis_off()
```

(continues on next page)

(continued from previous page)

```

for n, (fp, num_reps) in enumerate(footprint_sequence):
    npad = [((footprint.shape[d] - fp.shape[d]) // 2, ) * 2
            for d in range(ndim)]
    fp = np.pad(fp, npad, mode='constant')
    if ndim == 2:
        ax = fig.add_subplot(1, num_subplots, n + 1)
        ax.imshow(fp, cmap=cmap, vmin=0, vmax=1)
    else:
        ax = fig.add_subplot(1, num_subplots, n + 1,
                            projection=Axes3D.name)
        ax.voxels(fp, cmap=cmap)
    title = f"element {n + 1} of {num_seq}\n({num_reps} iteration"
    title += "s)" if num_reps > 1 else ")"
    ax.set_title(title, fontdict=fontdict)
    ax.set_axis_off()
    ax.set_xlabel(f'num_reps = {num_reps}')
fig.tight_layout()

# draw a line separating the sequence elements from the composite
line_pos = num_seq / num_subplots
line = plt.Line2D([line_pos, line_pos], [0, 1], color="black")
fig.add_artist(line)

plt.show()

```

Total running time of the script: (0 minutes 9.459 seconds)

Manipulating exposure and color channels

RGB to grayscale

This example converts an image with RGB channels into an image with a single grayscale channel.

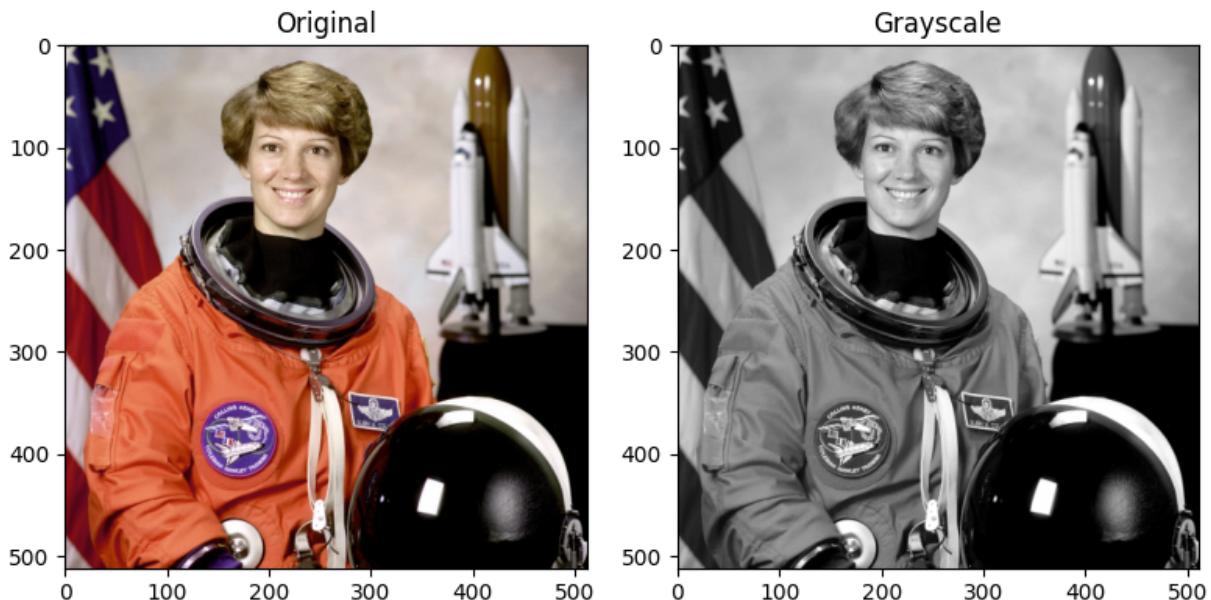
The value of each grayscale pixel is calculated as the weighted sum of the corresponding red, green and blue pixels as:

$$Y = 0.2125 \text{ R} + 0.7154 \text{ G} + 0.0721 \text{ B}$$

These weights are used by CRT phosphors as they better represent human perception of red, green and blue than equal weights.¹

¹ <http://poynton.ca/PDFs/ColorFAQ.pdf>

References



```
import matplotlib.pyplot as plt

from skimage import data
from skimage.color import rgb2gray

original = data.astronaut()
grayscale = rgb2gray(original)

fig, axes = plt.subplots(1, 2, figsize=(8, 4))
ax = axes.ravel()

ax[0].imshow(original)
ax[0].set_title("Original")
ax[1].imshow(grayscale, cmap=plt.cm.gray)
ax[1].set_title("Grayscale")

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

Total running time of the script: (0 minutes 0.350 seconds)

RGB to HSV

This example illustrates how RGB to HSV (Hue, Saturation, Value) conversion¹ can be used to facilitate segmentation processes.

Usually, objects in images have distinct colors (hues) and luminosities, so that these features can be used to separate different areas of the image. In the RGB representation the hue and the luminosity are expressed as a linear combination of the R,G,B channels, whereas they correspond to single channels of the HSV image (the Hue and the Value channels). A simple segmentation of the image can then be effectively performed by a mere thresholding of the HSV channels.

```
import matplotlib.pyplot as plt

from skimage import data
from skimage.color import rgb2hsv
```

We first load the RGB image and extract the Hue and Value channels:

```
rgb_img = data.coffee()
hsv_img = rgb2hsv(rgb_img)
hue_img = hsv_img[:, :, 0]
value_img = hsv_img[:, :, 2]

fig, (ax0, ax1, ax2) = plt.subplots(ncols=3, figsize=(8, 2))

ax0.imshow(rgb_img)
ax0.set_title("RGB image")
ax0.axis('off')
ax1.imshow(hue_img, cmap='hsv')
ax1.set_title("Hue channel")
ax1.axis('off')
ax2.imshow(value_img)
ax2.set_title("Value channel")
ax2.axis('off')

fig.tight_layout()
```



We then set a threshold on the Hue channel to separate the cup from the background:

```
hue_threshold = 0.04
binary_img = hue_img > hue_threshold

fig, (ax0, ax1) = plt.subplots(ncols=2, figsize=(8, 3))
```

(continues on next page)

¹ https://en.wikipedia.org/wiki/HSL_and_HSV

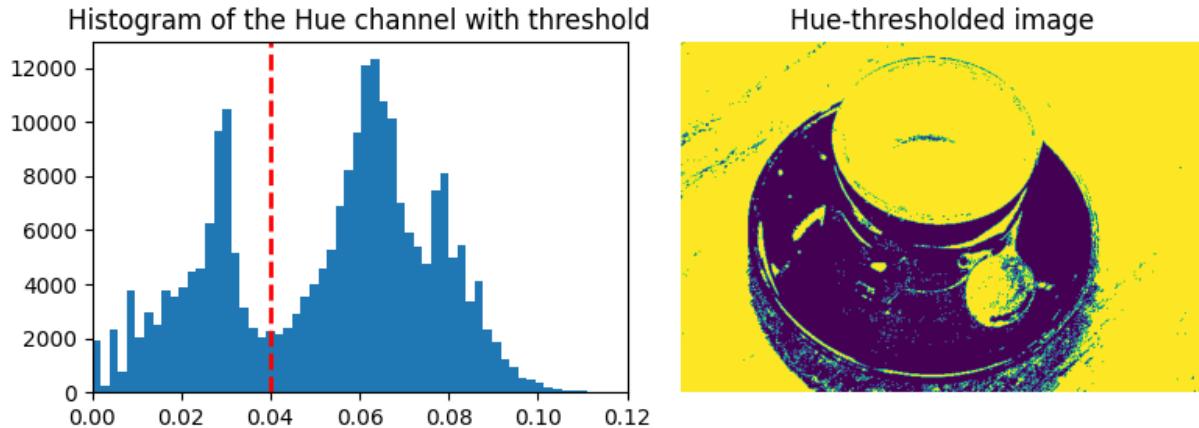
(continued from previous page)

```

ax0.hist(hue_img.ravel(), 512)
ax0.set_title("Histogram of the Hue channel with threshold")
ax0.axvline(x=hue_threshold, color='r', linestyle='dashed', linewidth=2)
ax0.set_xbound(0, 0.12)
ax1.imshow(binary_img)
ax1.set_title("Hue-thresholded image")
ax1.axis('off')

fig.tight_layout()

```



We finally perform an additional thresholding on the Value channel to partly remove the shadow of the cup:

```

fig, ax0 = plt.subplots(figsize=(4, 3))

value_threshold = 0.10
binary_img = (hue_img > hue_threshold) | (value_img < value_threshold)

ax0.imshow(binary_img)
ax0.set_title("Hue and value thresholded image")
ax0.axis('off')

fig.tight_layout()
plt.show()

```

Hue and value thresholded image



Total running time of the script: (0 minutes 1.033 seconds)

Histogram matching

This example demonstrates the feature of histogram matching. It manipulates the pixels of an input image so that its histogram matches the histogram of the reference image. If the images have multiple channels, the matching is done independently for each channel, as long as the number of channels is equal in the input image and the reference.

Histogram matching can be used as a lightweight normalisation for image processing, such as feature matching, especially in circumstances where the images have been taken from different sources or in different conditions (i.e. lighting).

```
import matplotlib.pyplot as plt

from skimage import data
from skimage import exposure
from skimage.exposure import match_histograms

reference = data.coffee()
image = data.chelsea()

matched = match_histograms(image, reference, channel_axis=-1)

fig, (ax1, ax2, ax3) = plt.subplots(nrows=1, ncols=3, figsize=(8, 3),
                                    sharex=True, sharey=True)
for aa in (ax1, ax2, ax3):
    aa.set_axis_off()

ax1.imshow(image)
ax1.set_title('Source')
ax2.imshow(reference)
ax2.set_title('Reference')
ax3.imshow(matched)
ax3.set_title('Matched')

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



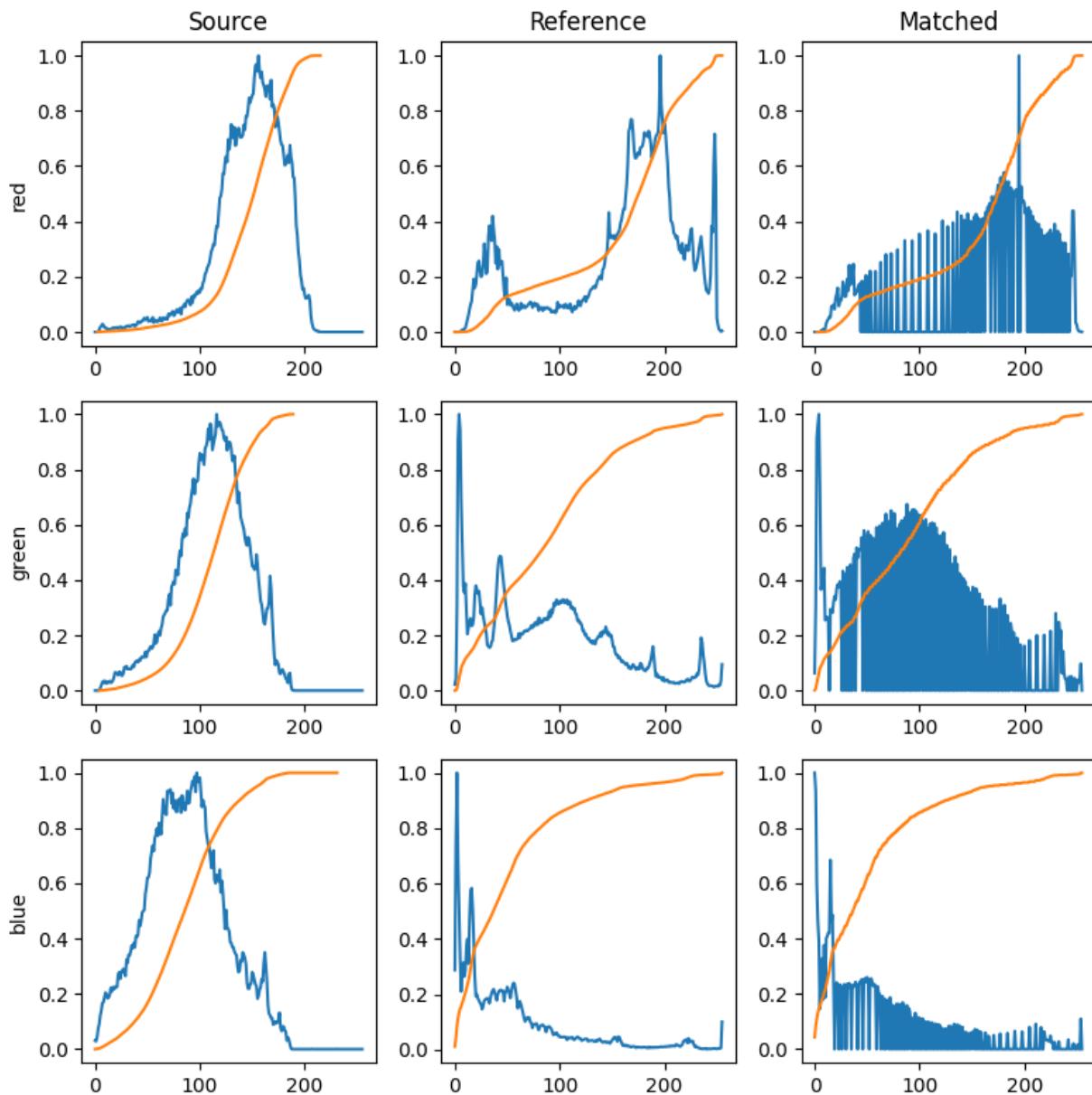
To illustrate the effect of the histogram matching, we plot for each RGB channel, the histogram and the cumulative histogram. Clearly, the matched image has the same cumulative histogram as the reference image for each channel.

```
fig, axes = plt.subplots(nrows=3, ncols=3, figsize=(8, 8))

for i, img in enumerate((image, reference, matched)):
    for c, c_color in enumerate(('red', 'green', 'blue')):
        img_hist, bins = exposure.histogram(img[..., c], source_range='dtype')
        axes[c, i].plot(bins, img_hist / img_hist.max())
        img_cdf, bins = exposure.cumulative_distribution(img[..., c])
        axes[c, i].plot(bins, img_cdf)
        axes[c, 0].set_ylabel(c_color)

axes[0, 0].set_title('Source')
axes[0, 1].set_title('Reference')
axes[0, 2].set_title('Matched')

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



Total running time of the script: (0 minutes 0.946 seconds)

Adapting gray-scale filters to RGB images

There are many filters that are designed to work with gray-scale images but not with color images. To simplify the process of creating functions that can adapt to RGB images, scikit-image provides the `adapt_rgb` decorator.

To actually use the `adapt_rgb` decorator, you have to decide how you want to adapt the RGB image for use with the gray-scale filter. There are two pre-defined handlers:

`each_channel`

Pass each of the RGB channels to the filter one-by-one, and stitch the results back into an RGB image.

`hsv_value`

Convert the RGB image to HSV and pass the value channel to the filter. The filtered result is inserted back into the HSV image and converted back to RGB.

Below, we demonstrate the use of `adapt_rgb` on a couple of gray-scale filters:

```
from skimage.color.adapt_rgb import adapt_rgb, each_channel, hsv_value
from skimage import filters

@adapt_rgb(each_channel)
def sobel_each(image):
    return filters.sobel(image)

@adapt_rgb(hsv_value)
def sobel_hsv(image):
    return filters.sobel(image)
```

We can use these functions as we would normally use them, but now they work with both gray-scale and color images. Let's plot the results with a color image:

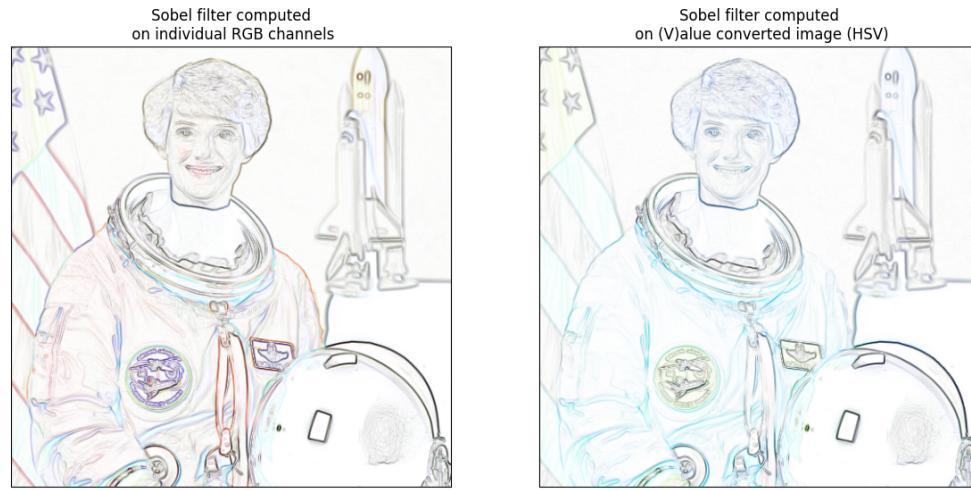
```
from skimage import data
from skimage.exposure import rescale_intensity
import matplotlib.pyplot as plt

image = data.astronaut()

fig, (ax_each, ax_hsv) = plt.subplots(ncols=2, figsize=(14, 7))

# We use 1 - sobel_each(image) but this won't work if image is not normalized
ax_each.imshow(rescale_intensity(1 - sobel_each(image)))
ax_each.set_xticks([]), ax_each.set_yticks([])
ax_each.set_title("Sobel filter computed\non individual RGB channels")

# We use 1 - sobel_hsv(image) but this won't work if image is not normalized
ax_hsv.imshow(rescale_intensity(1 - sobel_hsv(image)))
ax_hsv.set_xticks([]), ax_hsv.set_yticks([])
ax_hsv.set_title("Sobel filter computed\non (V)alue converted image (HSV)")
```



```
Text(0.5, 1.0, 'Sobel filter computed\n on (V)alue converted image (HSV)')
```

Notice that the result for the value-filtered image preserves the color of the original image, but channel filtered image combines in a more surprising way. In other common cases, smoothing for example, the channel filtered image will produce a better result than the value-filtered image.

You can also create your own handler functions for `adapt_rgb`. To do so, just create a function with the following signature:

```
def handler(image_filter, image, *args, **kwargs):
    # Manipulate RGB image here...
    image = image_filter(image, *args, **kwargs)
    # Manipulate filtered image here...
    return image
```

Note that `adapt_rgb` handlers are written for filters where the image is the first argument.

As a very simple example, we can just convert any RGB image to grayscale and then return the filtered result:

```
from skimage.color import rgb2gray

def as_gray(image_filter, image, *args, **kwargs):
    gray_image = rgb2gray(image)
    return image_filter(gray_image, *args, **kwargs)
```

It's important to create a signature that uses `*args` and `**kwargs` to pass arguments along to the filter so that the decorated function is allowed to have any number of positional and keyword arguments.

Finally, we can use this handler with `adapt_rgb` just as before:

```
@adapt_rgb(as_gray)
def sobel_gray(image):
    return filters.sobel(image)
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots(ncols=1, nrows=1, figsize=(7, 7))

# We use 1 - sobel_gray(image) but this won't work if image is not normalized
ax.imshow(rescale_intensity(1 - sobel_gray(image)), cmap=plt.cm.gray)
ax.set_xticks([]), ax.set_yticks([])
ax.set_title("Sobel filter computed\non the converted grayscale image")

plt.show()
```

Sobel filter computed
on the converted grayscale image



Note: A very simple check of the array shape is used for detecting RGB images, so `adapt_rgb` is not recommended

for functions that support 3D volumes or color images in non-RGB spaces.

Total running time of the script: (0 minutes 0.657 seconds)

Separate colors in immunohistochemical staining

Color deconvolution consists in the separation of features by their colors.

In this example we separate the immunohistochemical (IHC) staining from the hematoxylin counterstaining. The separation is achieved with the method described in¹ and known as “color deconvolution”.

The IHC staining expression of the FHL2 protein is here revealed with diaminobenzidine (DAB) which gives a brown color.

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

from skimage import data
from skimage.color import rgb2hed, hed2rgb

# Example IHC image
ihc_rgb = data.immunohistochemistry()

# Separate the stains from the IHC image
ihc_hed = rgb2hed(ihc_rgb)

# Create an RGB image for each of the stains
null = np.zeros_like(ihc_hed[:, :, 0])
ihc_h = hed2rgb(np.stack((ihc_hed[:, :, 0], null, null), axis=-1))
ihc_e = hed2rgb(np.stack((null, ihc_hed[:, :, 1], null), axis=-1))
ihc_d = hed2rgb(np.stack((null, null, ihc_hed[:, :, 2]), axis=-1))

# Display
fig, axes = plt.subplots(2, 2, figsize=(7, 6), sharex=True, sharey=True)
ax = axes.ravel()

ax[0].imshow(ihc_rgb)
ax[0].set_title("Original image")

ax[1].imshow(ihc_h)
ax[1].set_title("Hematoxylin")

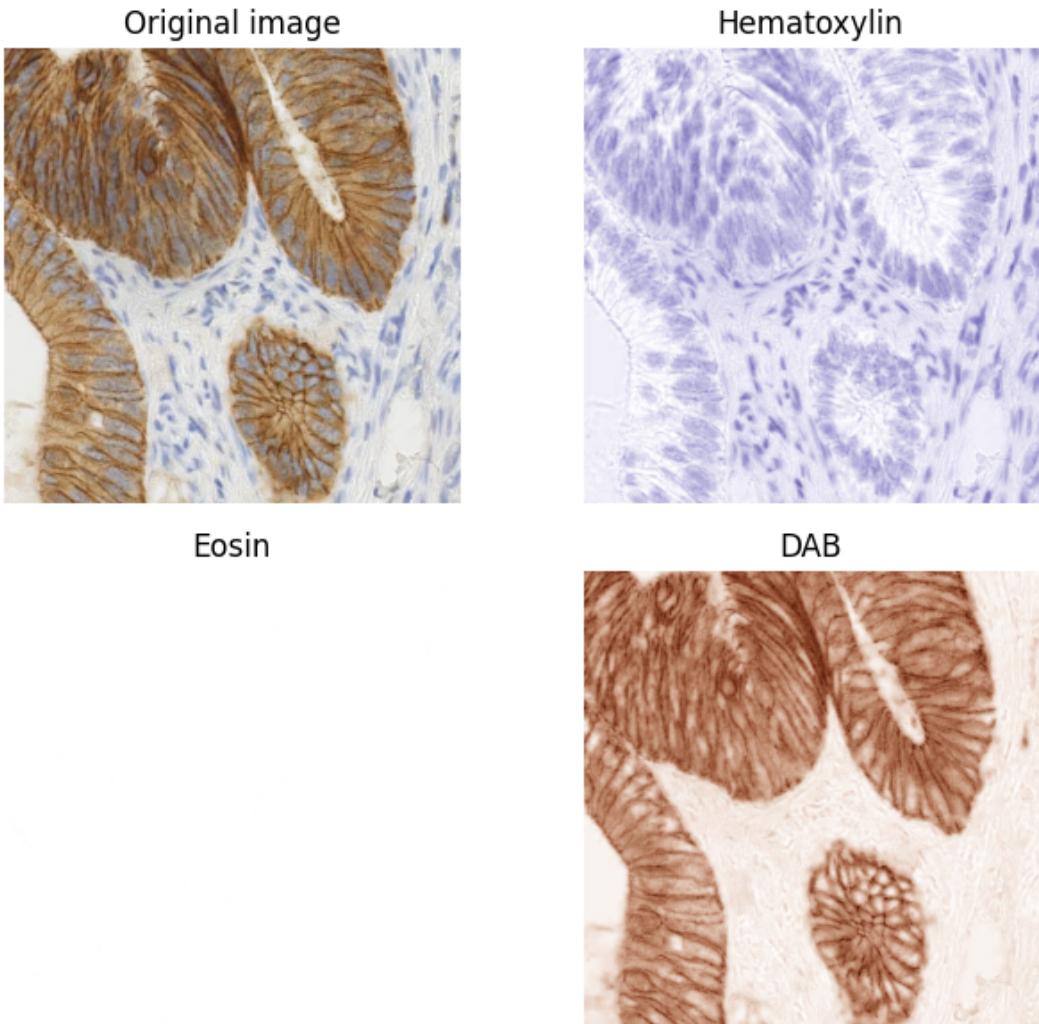
ax[2].imshow(ihc_e)
ax[2].set_title("Eosin") # Note that there is no Eosin stain in this image

ax[3].imshow(ihc_d)
ax[3].set_title("DAB")

for a in ax.ravel():
    a.axis('off')

fig.tight_layout()
```

¹ A. C. Ruifrok and D. A. Johnston, “Quantification of histochemical staining by color deconvolution,” Analytical and quantitative cytology and histology / the International Academy of Cytology [and] American Society of Cytology, vol. 23, no. 4, pp. 291-9, Aug. 2001. PMID: 11531144



Now we can easily manipulate the hematoxylin and DAB channels:

```
from skimage.exposure import rescale_intensity

# Rescale hematoxylin and DAB channels and give them a fluorescence look
h = rescale_intensity(ihc_hed[:, :, 0], out_range=(0, 1),
                      in_range=(0, np.percentile(ihc_hed[:, :, 0], 99)))
d = rescale_intensity(ihc_hed[:, :, 2], out_range=(0, 1),
                      in_range=(0, np.percentile(ihc_hed[:, :, 2], 99)))

# Cast the two channels into an RGB image, as the blue and green channels
# respectively
zdh = np.dstack((h, d, h))

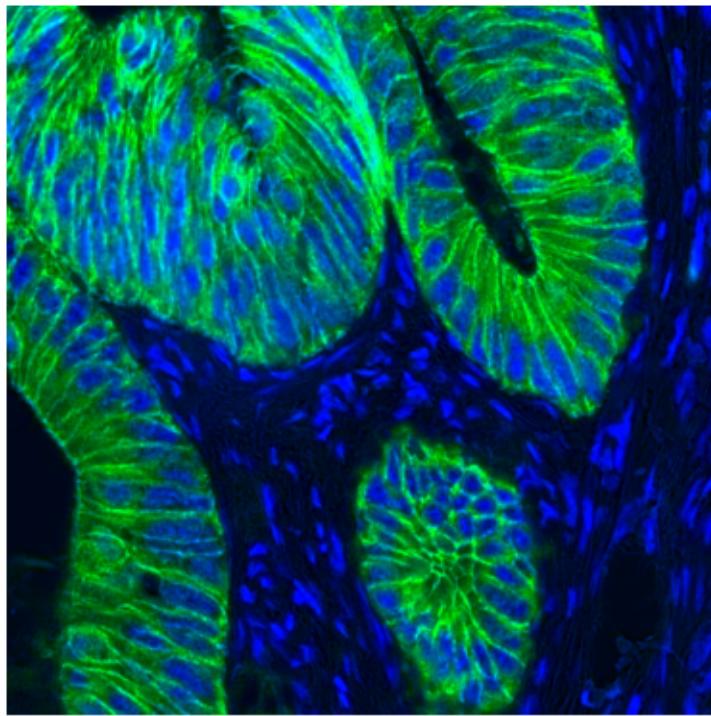
fig = plt.figure()
axis = plt.subplot(1, 1, 1, sharex=ax[0], sharey=ax[0])
axis.imshow(zdh)
axis.set_title('Stain-separated image (rescaled)')
axis.axis('off')
```

(continues on next page)

(continued from previous page)

plt.show()

Stain-separated image (rescaled)



Total running time of the script: (0 minutes 1.456 seconds)

Filtering regional maxima

Here, we use morphological reconstruction to create a background image, which we can subtract from the original image to isolate bright features (regional maxima).

First we try reconstruction by dilation starting at the edges of the image. We initialize a seed image to the minimum intensity of the image, and set its border to be the pixel values in the original image. These maximal pixels will get dilated in order to reconstruct the background image.

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

from scipy.ndimage import gaussian_filter
from skimage import data
from skimage import img_as_float
from skimage.morphology import reconstruction

# Convert to float: Important for subtraction later which won't work with uint8
image = img_as_float(data.coins())
image = gaussian_filter(image, 1)
```

(continues on next page)

(continued from previous page)

```
seed = np.copy(image)
seed[1:-1, 1:-1] = image.min()
mask = image

dilated = reconstruction(seed, mask, method='dilation')
```

Subtracting the dilated image leaves an image with just the coins and a flat, black background, as shown below.

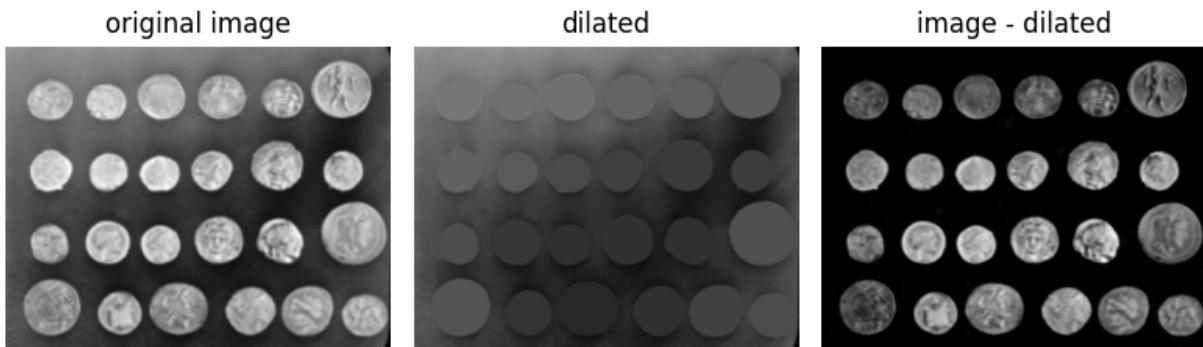
```
fig, (ax0, ax1, ax2) = plt.subplots(nrows=1,
                                    ncols=3,
                                    figsize=(8, 2.5),
                                    sharex=True,
                                    sharey=True)

ax0.imshow(image, cmap='gray')
ax0.set_title('original image')
ax0.axis('off')

ax1.imshow(dilated, vmin=image.min(), vmax=image.max(), cmap='gray')
ax1.set_title('dilated')
ax1.axis('off')

ax2.imshow(image - dilated, cmap='gray')
ax2.set_title('image - dilated')
ax2.axis('off')

fig.tight_layout()
```



Although the features (i.e. the coins) are clearly isolated, the coins surrounded by a bright background in the original image are dimmer in the subtracted image. We can attempt to correct this using a different seed image.

Instead of creating a seed image with maxima along the image border, we can use the features of the image itself to seed the reconstruction process. Here, the seed image is the original image minus a fixed value, h.

```
h = 0.4
seed = image - h
dilated = reconstruction(seed, mask, method='dilation')
hdome = image - dilated
```

To get a feel for the reconstruction process, we plot the intensity of the mask, seed, and dilated images along a slice of

the image (indicated by red line).

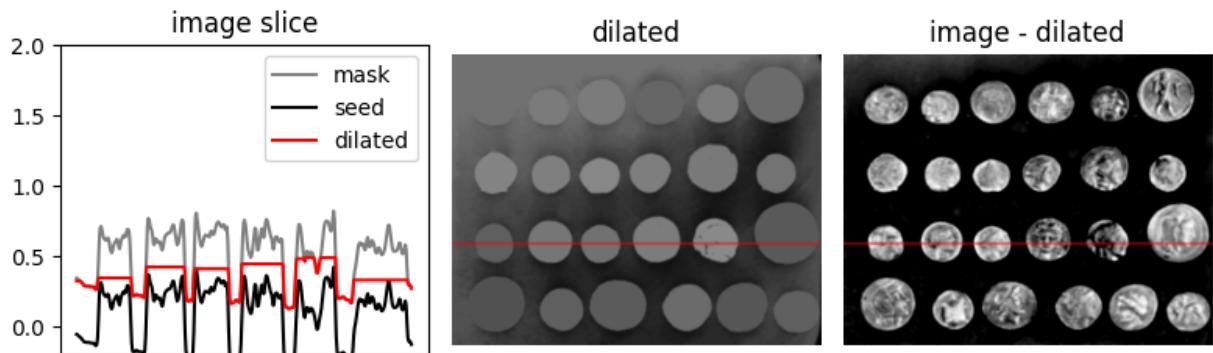
```
fig, (ax0, ax1, ax2) = plt.subplots(nrows=1, ncols=3, figsize=(8, 2.5))
yslice = 197

ax0.plot(mask[yslice], '0.5', label='mask')
ax0.plot(seed[yslice], 'k', label='seed')
ax0.plot(dilated[yslice], 'r', label='dilated')
ax0.set_xlim(-0.2, 2)
ax0.set_title('image slice')
ax0.set_xticks([])
ax0.legend()

ax1.imshow(dilated, vmin=image.min(), vmax=image.max(), cmap='gray')
ax1.axhline(yslice, color='r', alpha=0.4)
ax1.set_title('dilated')
ax1.axis('off')

ax2.imshow(hdome, cmap='gray')
ax2.axhline(yslice, color='r', alpha=0.4)
ax2.set_title('image - dilated')
ax2.axis('off')

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

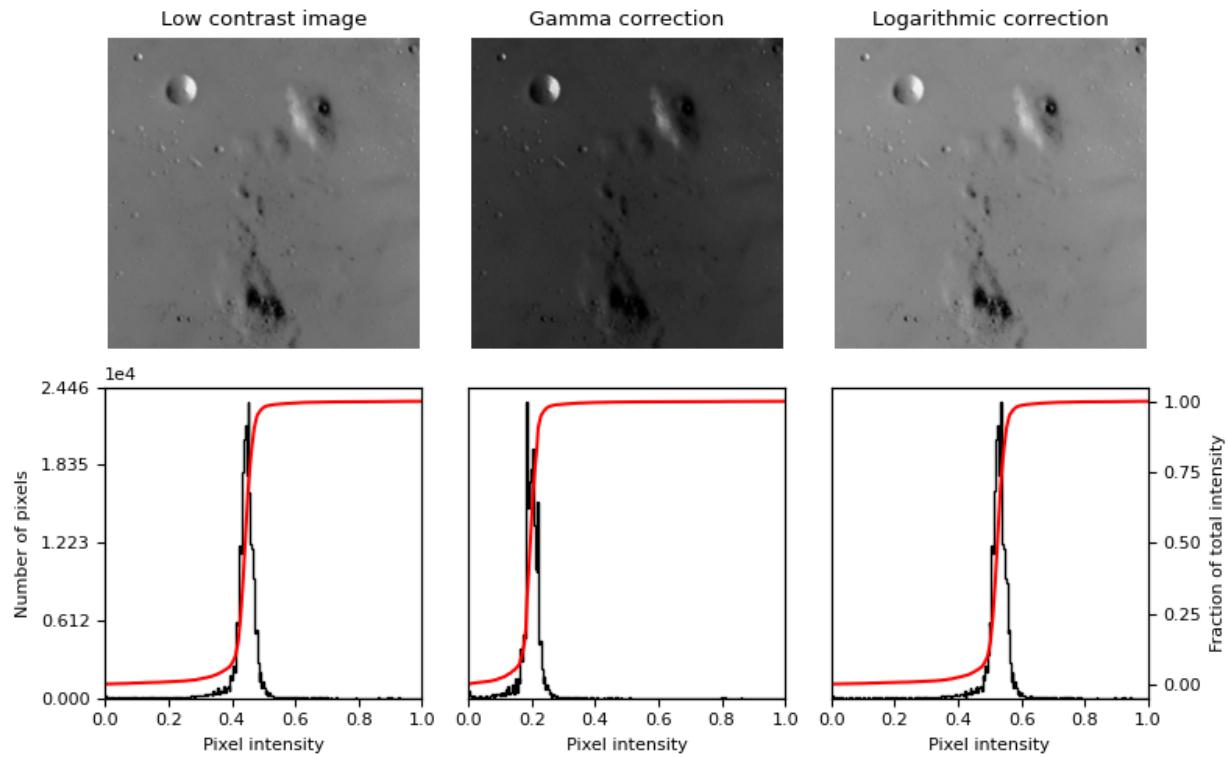


As you can see in the image slice, each coin is given a different baseline intensity in the reconstructed image; this is because we used the local intensity (shifted by h) as a seed value. As a result, the coins in the subtracted image have similar pixel intensities. The final result is known as the h-dome of an image since this tends to isolate regional maxima of height h . This operation is particularly useful when your images are unevenly illuminated.

Total running time of the script: (0 minutes 0.512 seconds)

Gamma and log contrast adjustment

This example adjusts image contrast by performing a Gamma and a Logarithmic correction on the input image.



```

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

from skimage import data, img_as_float
from skimage import exposure

matplotlib.rcParams['font.size'] = 8

def plot_img_and_hist(image, axes, bins=256):
    """Plot an image along with its histogram and cumulative histogram.

    """
    image = img_as_float(image)
    ax_img, ax_hist = axes
    ax_cdf = ax_hist.twinx()

    # Display image
    ax_img.imshow(image, cmap=plt.cm.gray)
    ax_img.set_axis_off()

    # Display histogram
    ax_hist.hist(image.ravel(), bins=bins, histtype='step', color='black')
    ax_hist.ticklabel_format(axis='y', style='scientific', scilimits=(0, 0))
    ax_cdf.plot(image.ravel(), np.arange(bins+1)/float(bins), color='red')

    return ax_img, ax_hist, ax_cdf

```

(continues on next page)

(continued from previous page)

```
ax_hist.set_xlabel('Pixel intensity')
ax_hist.set_xlim(0, 1)
ax_hist.set_yticks([])

# Display cumulative distribution
img_cdf, bins = exposure.cumulative_distribution(image, bins)
ax_cdf.plot(bins, img_cdf, 'r')
ax_cdf.set_yticks([])

return ax_img, ax_hist, ax_cdf

# Load an example image
img = data.moon()

# Gamma
gamma_corrected = exposure.adjust_gamma(img, 2)

# Logarithmic
logarithmic_corrected = exposure.adjust_log(img, 1)

# Display results
fig = plt.figure(figsize=(8, 5))
axes = np.zeros((2, 3), dtype=object)
axes[0, 0] = plt.subplot(2, 3, 1)
axes[0, 1] = plt.subplot(2, 3, 2, sharex=axes[0, 0], sharey=axes[0, 0])
axes[0, 2] = plt.subplot(2, 3, 3, sharex=axes[0, 0], sharey=axes[0, 0])
axes[1, 0] = plt.subplot(2, 3, 4)
axes[1, 1] = plt.subplot(2, 3, 5)
axes[1, 2] = plt.subplot(2, 3, 6)

ax_img, ax_hist, ax_cdf = plot_img_and_hist(img, axes[:, 0])
ax_img.set_title('Low contrast image')

y_min, y_max = ax_hist.get_ylimits()
ax_hist.set_ylabel('Number of pixels')
ax_hist.set_yticks(np.linspace(0, y_max, 5))

ax_img, ax_hist, ax_cdf = plot_img_and_hist(gamma_corrected, axes[:, 1])
ax_img.set_title('Gamma correction')

ax_img, ax_hist, ax_cdf = plot_img_and_hist(logarithmic_corrected, axes[:, 2])
ax_img.set_title('Logarithmic correction')

ax_cdf.set_ylabel('Fraction of total intensity')
ax_cdf.set_yticks(np.linspace(0, 1, 5))

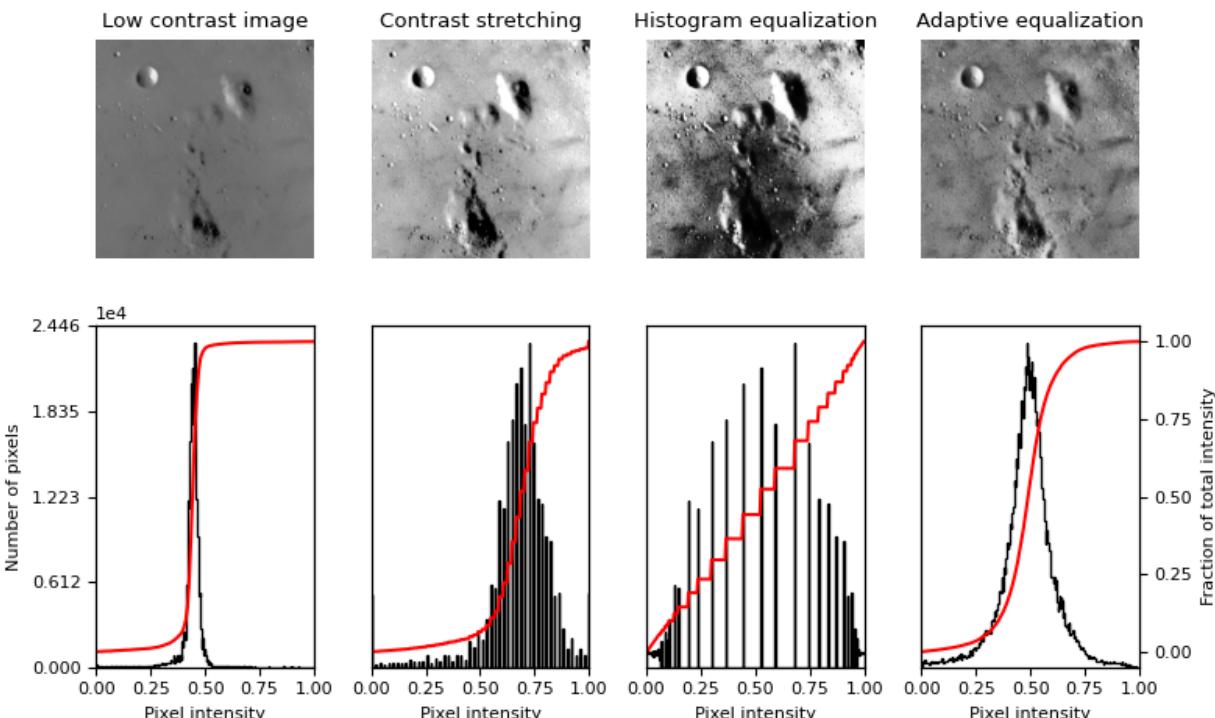
# prevent overlap of y-axis labels
fig.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.541 seconds)

Histogram Equalization

This examples enhances an image with low contrast, using a method called *histogram equalization*, which “spreads out the most frequent intensity values” in an image¹. The equalized image has a roughly linear cumulative distribution function.

While histogram equalization has the advantage that it requires no parameters, it sometimes yields unnatural looking images. An alternative method is *contrast stretching*, where the image is rescaled to include all intensities that fall within the 2nd and 98th percentiles².



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

from skimage import data, img_as_float
from skimage import exposure

matplotlib.rcParams['font.size'] = 8

def plot_img_and_hist(image, axes, bins=256):
    """Plot an image along with its histogram and cumulative histogram.

    """
    image = img_as_float(image)
    ax_img, ax_hist = axes
    ax_hist.hist(image.ravel(), bins=bins)
    ax_hist.set_yticks([0, 612, 1223, 1835, 2446])
    ax_hist.set_ylabel('Number of pixels')
    ax_hist.set_xlim(0, 1)
    ax_hist.set_ylabel('Fraction of total intensity')
    ax_hist.set_xlabel('Pixel intensity')

    cdf = np.zeros(bins + 1)
    for b in range(1, bins + 1):
        cdf[b - 1] = np.sum(image < b)
    cdf /= np.sum(image)
    ax_cdf = ax_hist.twinx()
    ax_cdf.plot(cdf, color='red')
    ax_cdf.set_yticks([0.00, 0.25, 0.50, 0.75, 1.00])
    ax_cdf.set_ylabel('Fraction of total intensity')
    ax_cdf.set_xlabel('Pixel intensity')
```

(continues on next page)

¹ https://en.wikipedia.org/wiki/Histogram_equalization

² <http://homepages.inf.ed.ac.uk/rbf/HIPR2/stretch.htm>

(continued from previous page)

```
ax_cdf = ax_hist.twinx()

# Display image
ax_img.imshow(image, cmap=plt.cm.gray)
ax_img.set_axis_off()

# Display histogram
ax_hist.hist(image.ravel(), bins=bins, histtype='step', color='black')
ax_hist.ticklabel_format(axis='y', style='scientific', scilimits=(0, 0))
ax_hist.set_xlabel('Pixel intensity')
ax_hist.set_xlim(0, 1)
ax_hist.set_yticks([])

# Display cumulative distribution
img_cdf, bins = exposure.cumulative_distribution(image, bins)
ax_cdf.plot(bins, img_cdf, 'r')
ax_cdf.set_yticks([])

return ax_img, ax_hist, ax_cdf

# Load an example image
img = data.moon()

# Contrast stretching
p2, p98 = np.percentile(img, (2, 98))
img_rescale = exposure.rescale_intensity(img, in_range=(p2, p98))

# Equalization
img_eq = exposure.equalize_hist(img)

# Adaptive Equalization
img_adapteq = exposure.equalize_adapthist(img, clip_limit=0.03)

# Display results
fig = plt.figure(figsize=(8, 5))
axes = np.zeros((2, 4), dtype=object)
axes[0, 0] = fig.add_subplot(2, 4, 1)
for i in range(1, 4):
    axes[0, i] = fig.add_subplot(2, 4, 1+i, sharex=axes[0, 0], sharey=axes[0, 0])
for i in range(0, 4):
    axes[1, i] = fig.add_subplot(2, 4, 5+i)

ax_img, ax_hist, ax_cdf = plot_img_and_hist(img, axes[:, 0])
ax_img.set_title('Low contrast image')

y_min, y_max = ax_hist.get_ylim()
ax_hist.set_ylabel('Number of pixels')
ax_hist.set_yticks(np.linspace(0, y_max, 5))

ax_img, ax_hist, ax_cdf = plot_img_and_hist(img_rescale, axes[:, 1])
ax_img.set_title('Contrast stretching')
```

(continues on next page)

(continued from previous page)

```

ax_img, ax_hist, ax_cdf = plot_img_and_hist(img_eq, axes[:, 2])
ax_img.set_title('Histogram equalization')

ax_img, ax_hist, ax_cdf = plot_img_and_hist(img_adapteq, axes[:, 3])
ax_img.set_title('Adaptive equalization')

ax_cdf.set_ylabel('Fraction of total intensity')
ax_cdf.set_yticks(np.linspace(0, 1, 5))

# prevent overlap of y-axis labels
fig.tight_layout()
plt.show()

```

Total running time of the script: (0 minutes 0.682 seconds)

Tinting gray-scale images

It can be useful to artificially tint an image with some color, either to highlight particular regions of an image or maybe just to liven up a grayscale image. This example demonstrates image-tinting by scaling RGB values and by adjusting colors in the HSV color-space.

In 2D, color images are often represented in RGB—3 layers of 2D arrays, where the 3 layers represent (R)ed, (G)reen and (B)lue channels of the image. The simplest way of getting a tinted image is to set each RGB channel to the grayscale image scaled by a different multiplier for each channel. For example, multiplying the green and blue channels by 0 leaves only the red channel and produces a bright red image. Similarly, zeroing-out the blue channel leaves only the red and green channels, which combine to form yellow.

```

import matplotlib.pyplot as plt
from skimage import data
from skimage import color
from skimage import img_as_float, img_as_ubyte

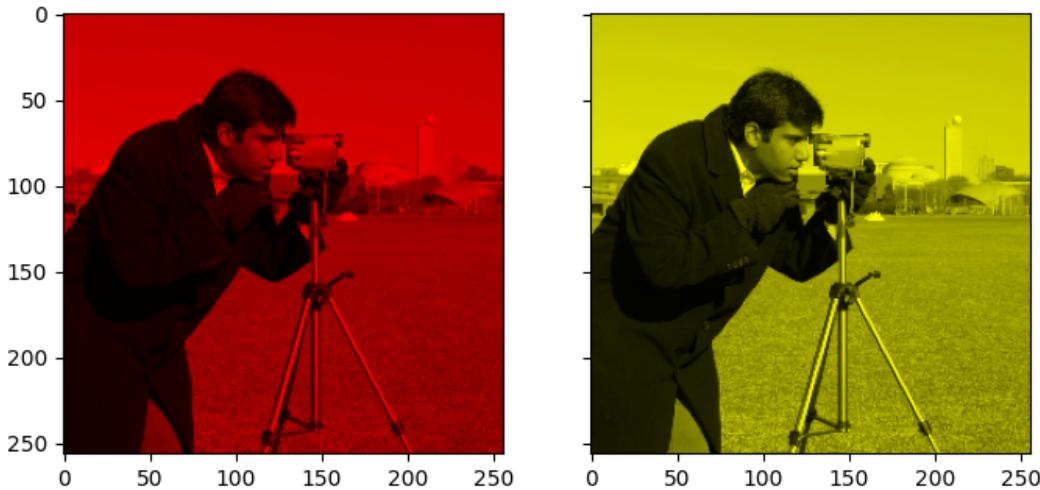
grayscale_image = img_as_float(data.camera()[:, :, ::2])
image = color.gray2rgb(grayscale_image)

red_multiplier = [1, 0, 0]
yellow_multiplier = [1, 1, 0]

fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(8, 4),
                             sharex=True, sharey=True)
ax1.imshow(red_multiplier * image)
ax2.imshow(yellow_multiplier * image)

plt.show()

```



In many cases, dealing with RGB values may not be ideal. Because of that, there are many other color spaces in which you can represent a color image. One popular color space is called HSV, which represents hue (~the color), saturation (~colorfulness), and value (~brightness). For example, a color (hue) might be green, but its saturation is how intense that green is —where olive is on the low end and neon on the high end.

In some implementations, the hue in HSV goes from 0 to 360, since hues wrap around in a circle. In scikit-image, however, hues are float values from 0 to 1, so that hue, saturation, and value all share the same scale.

Below, we plot a linear gradient in the hue, with the saturation and value turned all the way up:

```
import numpy as np

hue_gradient = np.linspace(0, 1)
hsv = np.ones(shape=(1, len(hue_gradient), 3), dtype=float)
hsv[:, :, 0] = hue_gradient

all_hues = color.hsv2rgb(hsv)

fig, ax = plt.subplots(figsize=(5, 2))
# Set image extent so hues go from 0 to 1 and the image is a nice aspect ratio.
ax.imshow(all_hues, extent=(0 - 0.5 / len(hue_gradient),
                           1 + 0.5 / len(hue_gradient), 0, 0.2))
ax.set_axis_off()
```



Notice how the colors at the far left and far right are the same. That reflects the fact that the hues wrap around like the color wheel (see [HSV](#) for more info).

Now, let's create a little utility function to take an RGB image and:

1. Transform the RGB image to HSV 2. Set the hue and saturation 3. Transform the HSV image back to RGB

```
def colorize(image, hue, saturation=1):
    """ Add color of the given hue to an RGB image.

    By default, set the saturation to 1 so that the colors pop!
    """
    hsv = color.rgb2hsv(image)
    hsv[:, :, 1] = saturation
    hsv[:, :, 0] = hue
    return color.hsv2rgb(hsv)
```

Notice that we need to bump up the saturation; images with zero saturation are grayscale, so we need to a non-zero value to actually see the color we've set.

Using the function above, we plot six images with a linear gradient in the hue and a non-zero saturation:

```
hue_rotations = np.linspace(0, 1, 6)

fig, axes = plt.subplots(nrows=2, ncols=3, sharex=True, sharey=True)

for ax, hue in zip(axes.flat, hue_rotations):
    # Turn down the saturation to give it that vintage look.
    tinted_image = colorize(image, hue, saturation=0.3)
    ax.imshow(tinted_image, vmin=0, vmax=1)
    ax.set_axis_off()
fig.tight_layout()
```



You can combine this tinting effect with numpy slicing and fancy-indexing to selectively tint your images. In the example below, we set the hue of some rectangles using slicing and scale the RGB values of some pixels found by thresholding. In practice, you might want to define a region for tinting based on segmentation results or blob detection methods.

```
from skimage.filters import rank

# Square regions defined as slices over the first two dimensions.
top_left = (slice(25),) * 2
bottom_right = (slice(-25, None),) * 2

sliced_image = image.copy()
sliced_image[top_left] = colorize(image[top_left], 0.82, saturation=0.5)
sliced_image[bottom_right] = colorize(image[bottom_right], 0.5, saturation=0.5)

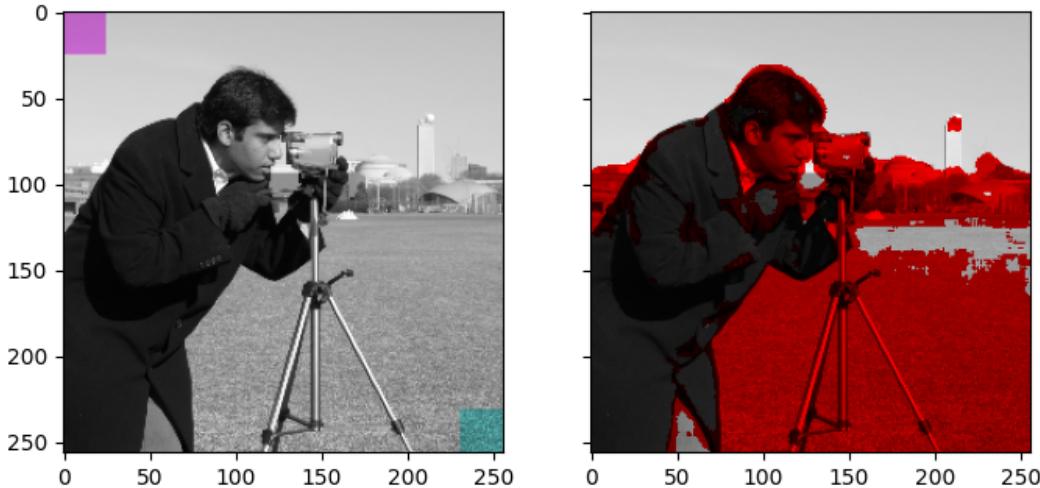
# Create a mask selecting regions with interesting texture.
noisy = rank.entropy(img_as_ubyte(grayscale_image), np.ones((9, 9)))
textured_regions = noisy > 4.25
# Note that using `colorize` here is a bit more difficult, since `rgb2hsv`
# expects an RGB image (height x width x channel), but fancy-indexing returns
# a set of RGB pixels (# pixels x channel).
masked_image = image.copy()
masked_image[textured_regions, :] *= red_multiplier
```

(continues on next page)

(continued from previous page)

```
fig, (ax1, ax2) = plt.subplots(ncols=2, nrows=1, figsize=(8, 4),
                             sharex=True, sharey=True)
ax1.imshow(sliced_image)
ax2.imshow(masked_image)

plt.show()
```



For coloring multiple regions, you may also be interested in `skimage.color.label2rgb`.

Total running time of the script: (0 minutes 0.946 seconds)

Local Histogram Equalization

This example enhances an image with low contrast, using a method called *local histogram equalization*, which spreads out the most frequent intensity values in an image.

The equalized image¹ has a roughly linear cumulative distribution function for each pixel neighborhood.

The local version² of the histogram equalization emphasized every local graylevel variations.

These algorithms can be used on both 2D and 3D images.

References

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

from skimage import data
from skimage.util.dtype import dtype_range
from skimage.util import img_as_ubyte
```

(continues on next page)

¹ https://en.wikipedia.org/wiki/Histogram_equalization

² https://en.wikipedia.org/wiki/Adaptive_histogram_equalization

(continued from previous page)

```
from skimage import exposure
from skimage.morphology import disk
from skimage.morphology import ball
from skimage.filters import rank

matplotlib.rcParams['font.size'] = 9

def plot_img_and_hist(image, axes, bins=256):
    """Plot an image along with its histogram and cumulative histogram.

    """
    ax_img, ax_hist = axes
    ax_cdf = ax_hist.twinx()

    # Display image
    ax_img.imshow(image, cmap=plt.cm.gray)
    ax_img.set_axis_off()

    # Display histogram
    ax_hist.hist(image.ravel(), bins=bins)
    ax_hist.ticklabel_format(axis='y', style='scientific', scilimits=(0, 0))
    ax_hist.set_xlabel('Pixel intensity')

    xmin, xmax = dtype_range[image.dtype.type]
    ax_hist.set_xlim(xmin, xmax)

    # Display cumulative distribution
    img_cdf, bins = exposure.cumulative_distribution(image, bins)
    ax_cdf.plot(bins, img_cdf, 'r')

    return ax_img, ax_hist, ax_cdf

# Load an example image
img = img_as_ubyte(data.moon())

# Global equalize
img_rescale = exposure.equalize_hist(img)

# Equalization
footprint = disk(30)
img_eq = rank.equalize(img, footprint=footprint)

# Display results
fig = plt.figure(figsize=(8, 5))
axes = np.zeros((2, 3), dtype=object)
axes[0, 0] = plt.subplot(2, 3, 1)
axes[0, 1] = plt.subplot(2, 3, 2, sharex=axes[0, 0], sharey=axes[0, 0])
axes[0, 2] = plt.subplot(2, 3, 3, sharex=axes[0, 0], sharey=axes[0, 0])
```

(continues on next page)

(continued from previous page)

```

axes[1, 0] = plt.subplot(2, 3, 4)
axes[1, 1] = plt.subplot(2, 3, 5)
axes[1, 2] = plt.subplot(2, 3, 6)

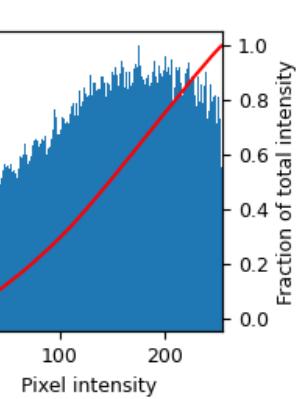
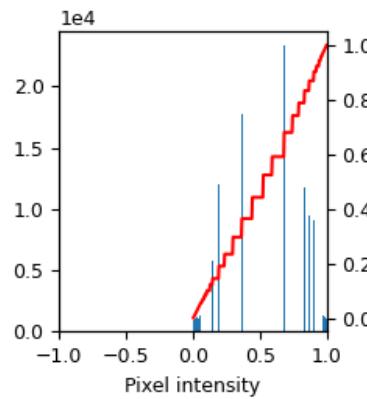
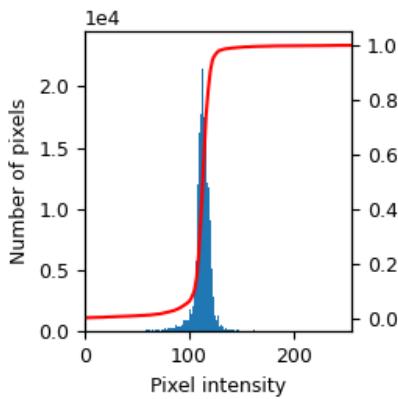
ax_img, ax_hist, ax_cdf = plot_img_and_hist(img, axes[:, 0])
ax_img.set_title('Low contrast image')
ax_hist.set_ylabel('Number of pixels')

ax_img, ax_hist, ax_cdf = plot_img_and_hist(img_rescale, axes[:, 1])
ax_img.set_title('Global equalise')

ax_img, ax_hist, ax_cdf = plot_img_and_hist(img_eq, axes[:, 2])
ax_img.set_title('Local equalize')
ax_cdf.set_ylabel('Fraction of total intensity')

# prevent overlap of y-axis labels
fig.tight_layout()

```



3D Equalization

3D Volumes can also be equalized in a similar fashion. Here the histograms are collected from the entire 3D image, but only a single slice is shown for visual inspection.

```
matplotlib.rcParams['font.size'] = 9

def plot_img_and_hist(image, axes, bins=256):
    """Plot an image along with its histogram and cumulative histogram.

    """
    ax_img, ax_hist = axes
    ax_cdf = ax_hist.twinx()

    # Display Slice of Image
    ax_img.imshow(image[0], cmap=plt.cm.gray)
    ax_img.set_axis_off()

    # Display histogram
    ax_hist.hist(image.ravel(), bins=bins)
    ax_hist.ticklabel_format(axis='y', style='scientific', scilimits=(0, 0))
    ax_hist.set_xlabel('Pixel intensity')

    xmin, xmax = dtype_range[image.dtype.type]
    ax_hist.set_xlim(xmin, xmax)

    # Display cumulative distribution
    img_cdf, bins = exposure.cumulative_distribution(image, bins)
    ax_cdf.plot(bins, img_cdf, 'r')

    return ax_img, ax_hist, ax_cdf

# Load an example image
img = img_as_ubyte(data.brain())

# Global equalization
img_rescale = exposure.equalize_hist(img)

# Local equalization
neighborhood = ball(3)
img_eq = rank.equalize(img, footprint=neighborhood)

# Display results
fig, axes = plt.subplots(2, 3, figsize=(8, 5))
axes[0, 1] = plt.subplot(2, 3, 2, sharex=axes[0, 0], sharey=axes[0, 0])
axes[0, 2] = plt.subplot(2, 3, 3, sharex=axes[0, 0], sharey=axes[0, 0])

ax_img, ax_hist, ax_cdf = plot_img_and_hist(img, axes[:, 0])
ax_img.set_title('Low contrast image')
ax_hist.set_ylabel('Number of pixels')
```

(continues on next page)

(continued from previous page)

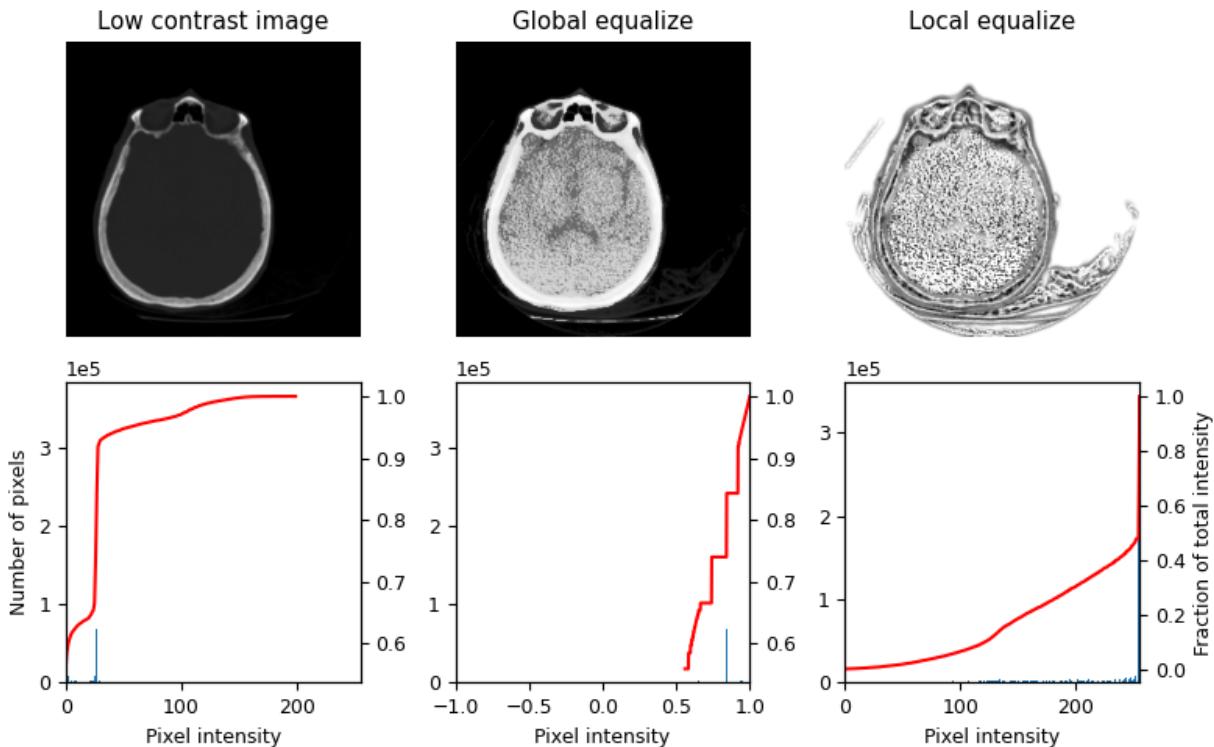
```

ax_img, ax_hist, ax_cdf = plot_img_and_hist(img_rescale, axes[:, 1])
ax_img.set_title('Global equalize')

ax_img, ax_hist, ax_cdf = plot_img_and_hist(img_eq, axes[:, 2])
ax_img.set_title('Local equalize')
ax_cdf.set_ylabel('Fraction of total intensity')

# prevent overlap of y-axis labels
fig.tight_layout()
plt.show()

```



```

/github/workspace/build/scikit-image/doc/examples/color_exposure/plot_local_equalize.py:154: MatplotlibDeprecationWarning:
Auto-removal of overlapping axes is deprecated since 3.6 and will be removed two minor releases later; explicitly call ax.remove() as needed.

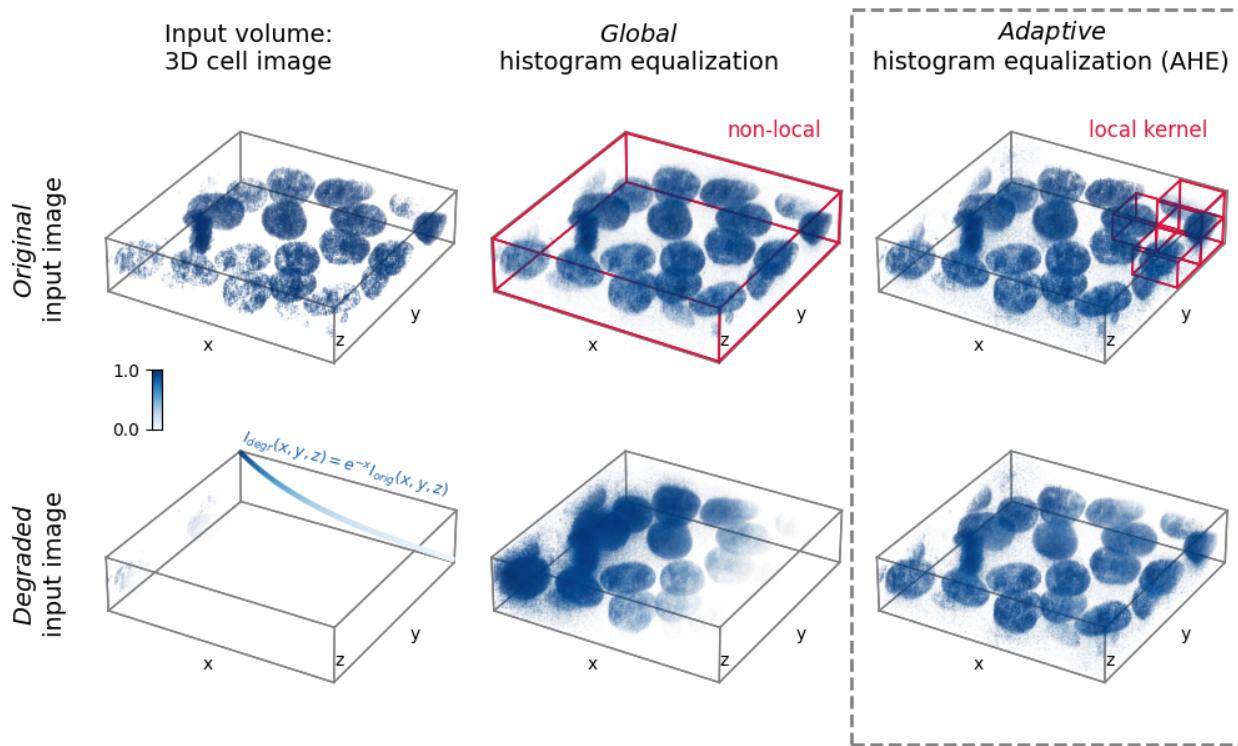
/github/workspace/build/scikit-image/doc/examples/color_exposure/plot_local_equalize.py:155: MatplotlibDeprecationWarning:
Auto-removal of overlapping axes is deprecated since 3.6 and will be removed two minor releases later; explicitly call ax.remove() as needed.

```

Total running time of the script: (0 minutes 3.256 seconds)

3D adaptive histogram equalization

Adaptive histogram equalization (AHE) can be used to improve the local contrast of an image¹. Specifically, AHE can be useful for normalizing intensities across images. This example compares the results of applying global histogram equalization and AHE to a 3D image and a synthetically degraded version of it.



```

import matplotlib.pyplot as plt
import matplotlib.patches as patches
from matplotlib import cm, colors
from mpl_toolkits.mplot3d import Axes3D

import numpy as np
from skimage import exposure, util

# Prepare data and apply histogram equalization
from skimage.data import cells3d

im_orig = util.img_as_float(cells3d()[:, 1, :, :]) # grab just the nuclei

# Reorder axis order from (z, y, x) to (x, y, z)
im_orig = im_orig.transpose()

# Rescale image data to range [0, 1]
im_orig = np.clip(im_orig,
                  np.percentile(im_orig, 5),
                  np.percentile(im_orig, 95))

```

(continues on next page)

¹ https://en.wikipedia.org/wiki/Histogram_equalization

(continued from previous page)

```

im_orig = (im_orig - im_orig.min()) / (im_orig.max() - im_orig.min())

# Degrade image by applying exponential intensity decay along x
sigmoid = np.exp(-3 * np.linspace(0, 1, im_orig.shape[0]))
im_degraded = (im_orig.T * sigmoid).T

# Set parameters for AHE
# Determine kernel sizes in each dim relative to image shape
kernel_size = (im_orig.shape[0] // 5,
               im_orig.shape[1] // 5,
               im_orig.shape[2] // 2)
kernel_size = np.array(kernel_size)
clip_limit = 0.9

# Perform histogram equalization
im_orig_he, im_degraded_he = \
    (exposure.equalize_hist(im)
     for im in [im_orig, im_degraded])

im_orig_ahe, im_degraded_ahe = \
    (exposure.equalize_adapthist(im,
                                 kernel_size=kernel_size,
                                 clip_limit=clip_limit)
     for im in [im_orig, im_degraded])

# Define functions to help plot the data

def scalars_to_rgba(scalars, cmap, vmin=0., vmax=1., alpha=0.2):
    """
    Convert array of scalars into array of corresponding RGBA values.
    """
    norm = colors.Normalize(vmin=vmin, vmax=vmax)
    scalar_map = cm.ScalarMappable(norm=norm, cmap=cmap)
    rgbs = scalar_map.to_rgba(scalars)
    rgbs[:, 3] = alpha
    return rgbs

def plt_render_volume(vol, fig_ax, cmap,
                     vmin=0, vmax=1,
                     bin_widths=None, n_levels=20):
    """
    Render a volume in a 3D matplotlib scatter plot.
    Better would be to use napari.
    """
    vol = np.clip(vol, vmin, vmax)

    xs, ys, zs = np.mgrid[0:vol.shape[0]:bin_widths[0],
                          0:vol.shape[1]:bin_widths[1],
                          0:vol.shape[2]:bin_widths[2]]
    vol_scaled = vol[:, :, :, ::bin_widths[0],

```

(continues on next page)

(continued from previous page)

```

        ::bin_widths[1],
        ::bin_widths[2]).flatten()

# Define alpha transfer function
levels = np.linspace(vmin, vmax, n_levels)
alphas = np.linspace(0, .7, n_levels)
alphas = alphas ** 11
alphas = (alphas - alphas.min()) / (alphas.max() - alphas.min())
alphas *= 0.8

# Group pixels by intensity and plot separately,
# as 3D scatter does not accept arrays of alpha values
for il in range(1, len(levels)):
    sel = (vol_scaled >= levels[il - 1])
    sel *= (vol_scaled <= levels[il])
    if not np.max(sel):
        continue
    c = scalars_to_rgba(vol_scaled[sel], cmap,
                         vmin=vmin, vmax=vmax, alpha=alphas[il - 1])
    fig_ax.scatter(xs.flatten()[sel],
                   ys.flatten()[sel],
                   zs.flatten()[sel],
                   c=c, s=0.5 * np.mean(bin_widths),
                   marker='o', linewidth=0)

# Create figure with subplots

cmap = 'Blues'

fig = plt.figure(figsize=(10, 6))
axs = [fig.add_subplot(2, 3, i + 1,
                      projection=Axes3D.name, facecolor="none")
       for i in range(6)]
ims = [im_orig, im_orig_he, im_orig_ahe,
       im_degraded, im_degraded_he, im_degraded_ahe]

# Prepare lines for the various boxes to be plotted
verts = np.array([[i, j, k] for i in [0, 1]
                  for j in [0, 1] for k in [0, 1]]).astype(np.float32)
lines = [np.array([i, j]) for i in verts
         for j in verts if np.allclose(np.linalg.norm(i - j), 1)]

# "render" volumetric data
for iax, ax in enumerate(axs[:]):
    plt_render_volume(ims[iax], ax, cmap, 0, 1, [2, 2, 2], 20)

# plot 3D box
rect_shape = np.array(im_orig.shape) + 2
for line in lines:
    ax.plot((line * rect_shape)[:, 0] - 1,
            (line * rect_shape)[:, 1] - 1,
            (line * rect_shape)[:, 2] - 1,
            color='red')

```

(continues on next page)

(continued from previous page)

```

        (line * rect_shape)[:, 2] - 1,
        linewidth=1, color='gray')

# Add boxes illustrating the kernels
ns = np.array(im_orig.shape) // kernel_size - 1
for axis_ind, vertex_ind, box_shape in zip([1] + [2] * 4,
                                            [[0, 0, 0],
                                             [ns[0] - 1, ns[1], ns[2] - 1],
                                             [ns[0], ns[1] - 1, ns[2] - 1],
                                             [ns[0], ns[1], ns[2] - 1],
                                             [ns[0], ns[1], ns[2]]],
                                             [np.array(im_orig.shape)] +
                                             [kernel_size] * 4):

    for line in lines:
        axs[axis_ind].plot((line + vertex_ind) * box_shape)[:, 0],
                           ((line + vertex_ind) * box_shape)[:, 1],
                           ((line + vertex_ind) * box_shape)[:, 2],
                           linewidth=1.2, color='crimson')

# Plot degradation function
axs[3].scatter(xs=np.arange(len(sigmoid)),
               ys=np.zeros(len(sigmoid)) + im_orig.shape[1],
               zs=sigmoid * im_orig.shape[2],
               s=5,
               c=scalars_to_rgba(sigmoid,
                                  cmap=cmap, vmin=0, vmax=1, alpha=1.)[:, :3])

# Subplot aesthetics
for iax, ax in enumerate(axs[:]):

    # Get rid of panes and axis lines
    for dim_ax in [ax.xaxis, ax.yaxis, ax.zaxis]:
        dim_ax.set_pane_color((1., 1., 1., 0.))
        dim_ax.line.set_color((1., 1., 1., 0.))

    # Define 3D axes limits, see https://github.com/
    # matplotlib/matplotlib/issues/17172#issuecomment-617546105
    xyzlim = np.array([ax.get_xlim3d(),
                      ax.get_ylim3d(),
                      ax.get_zlim3d()]).T
    XYZlim = np.asarray([min(xyzlim[0]), max(xyzlim[1])])
    ax.set_xlim3d(XYZlim)
    ax.set_ylim3d(XYZlim)
    ax.set_zlim3d(XYZlim * 0.5)

    try:
        ax.set_aspect('equal')
    except NotImplementedError:
        pass

    ax.set_xlabel('x', labelpad=-20)

```

(continues on next page)

(continued from previous page)

```
ax.set_ylabel('y', labelpad=-20)
ax.text2D(0.63, 0.2, "z", transform=ax.transAxes)
ax.set_xticks([])
ax.set_yticks([])
ax.set_zticks([])
ax.grid(False)
ax.elev = 30

plt.subplots_adjust(left=0.05,
                    bottom=-0.1,
                    right=1.01,
                    top=1.1,
                    wspace=-0.1,
                    hspace=-0.45)

# Highlight AHE
rect_ax = fig.add_axes([0, 0, 1, 1], facecolor='none')
rect_ax.set_axis_off()
rect = patches.Rectangle((0.68, 0.01), 0.315, 0.98,
                        edgecolor='gray', facecolor='none',
                        linewidth=2, linestyle='--')
rect_ax.add_patch(rect)

# Add text
rect_ax.text(0.19, 0.34, '$I_{degr}(x,y,z) = e^{-x}I_{orig}(x,y,z)$',
             fontsize=9, rotation=-15,
             color=scalars_to_rgba([0.8], cmap='Blues', alpha=1.0)[0])

fc = {'size': 14}
rect_ax.text(0.03, 0.58, r'$\it{Original}$' + '\ninput image',
             rotation=90, fontdict=fc, horizontalalignment='center')
rect_ax.text(0.03, 0.16, r'$\it{Degraded}$' + '\ninput image',
             rotation=90, fontdict=fc, horizontalalignment='center')
rect_ax.text(0.13, 0.91, 'Input volume:\n3D cell image', fontdict=fc)
rect_ax.text(0.51, 0.91, r'$\it{Global}$' + '\nhistogram equalization',
             fontdict=fc, horizontalalignment='center')
rect_ax.text(0.84, 0.91,
             r'$\it{Adaptive}$' + '\nhistogram equalization (AHE)',
             fontdict=fc, horizontalalignment='center')
rect_ax.text(0.58, 0.82, 'non-local', fontsize=12, color='crimson')
rect_ax.text(0.87, 0.82, 'local kernel', fontsize=12, color='crimson')

# Add colorbar
cbar_ax = fig.add_axes([0.12, 0.43, 0.008, 0.08])
cbar_ax.imshow(np.arange(256).reshape(256, 1)[:, :-1],
               cmap=cmap, aspect="auto")
cbar_ax.set_xticks([])
cbar_ax.set_yticks([0, 255])
cbar_ax.set_xticklabels([])
cbar_ax.set_yticklabels([1., 0.])

plt.show()
```

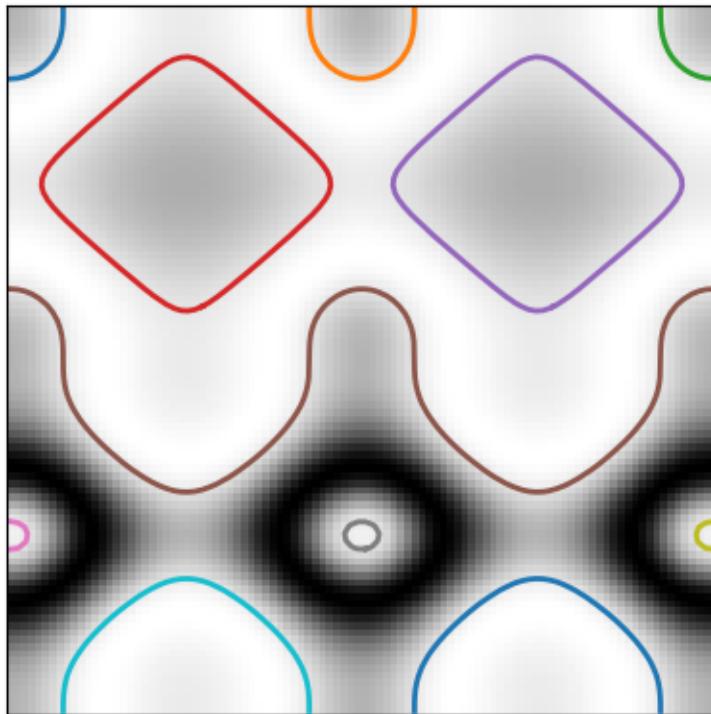
Total running time of the script: (0 minutes 17.492 seconds)

Edges and lines

Contour finding

We use a marching squares method to find constant valued contours in an image. In `skimage.measure.find_contours`, array values are linearly interpolated to provide better precision of the output contours. Contours which intersect the image edge are open; all others are closed.

The [marching squares algorithm](#) is a special case of the marching cubes algorithm (Lorensen, William and Harvey E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. Computer Graphics SIGGRAPH 87 Proceedings) 21(4) July 1987, p. 163-170).



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

from skimage import measure

# Construct some test data
x, y = np.ogrid[-np.pi:np.pi:100j, -np.pi:np.pi:100j]
r = np.sin(np.exp(np.sin(x)**3 + np.cos(y)**2))
```

(continues on next page)

(continued from previous page)

```
# Find contours at a constant value of 0.8
contours = measure.find_contours(r, 0.8)

# Display the image and plot all contours found
fig, ax = plt.subplots()
ax.imshow(r, cmap=plt.cm.gray)

for contour in contours:
    ax.plot(contour[:, 1], contour[:, 0], linewidth=2)

ax.axis('image')
ax.set_xticks([])
ax.set_yticks([])
plt.show()
```

Total running time of the script: (0 minutes 0.053 seconds)

Convex Hull

The convex hull of a binary image is the set of pixels included in the smallest convex polygon that surround all white pixels in the input.

A good overview of the algorithm is given on Steve Eddin's blog.

```
import matplotlib.pyplot as plt

from skimage.morphology import convex_hull_image
from skimage import data, img_as_float
from skimage.util import invert

# The original image is inverted as the object must be white.
image = invert(data.horse())

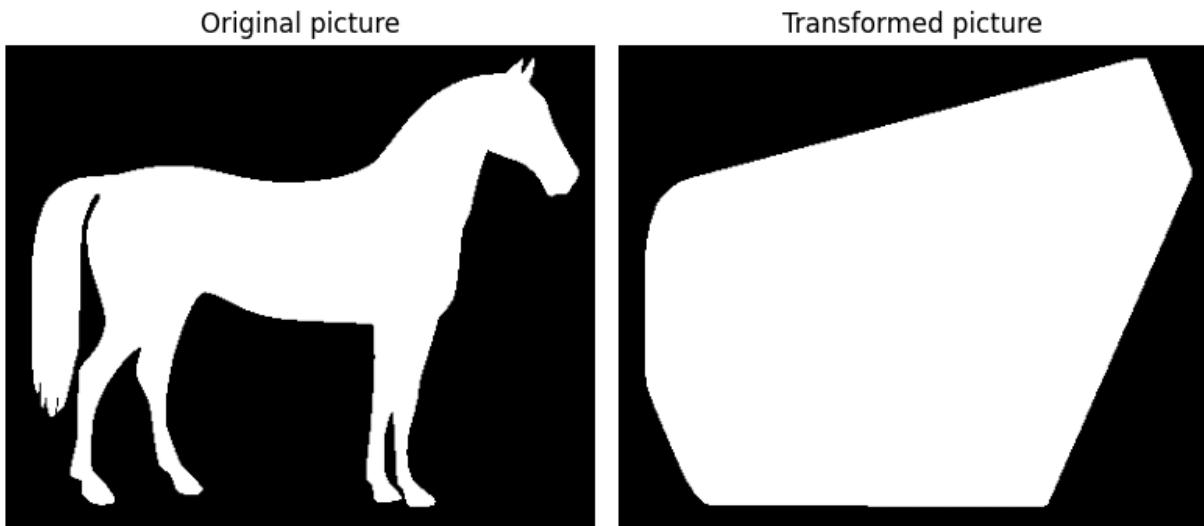
chull = convex_hull_image(image)

fig, axes = plt.subplots(1, 2, figsize=(8, 4))
ax = axes.ravel()

ax[0].set_title('Original picture')
ax[0].imshow(image, cmap=plt.cm.gray)
ax[0].set_axis_off()

ax[1].set_title('Transformed picture')
ax[1].imshow(chull, cmap=plt.cm.gray)
ax[1].set_axis_off()

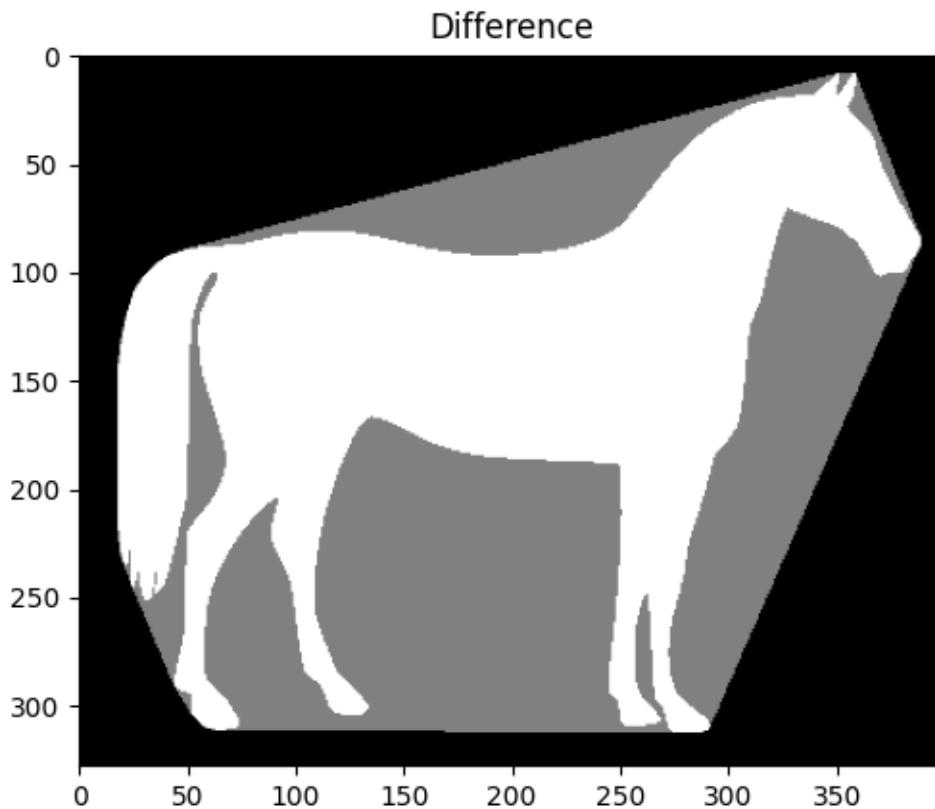
plt.tight_layout()
plt.show()
```



We prepare a second plot to show the difference.

```
chull_diff = img_as_float(chull.copy())
chull_diff[image] = 2

fig, ax = plt.subplots()
ax.imshow(chull_diff, cmap=plt.cm.gray)
ax.set_title('Difference')
plt.show()
```



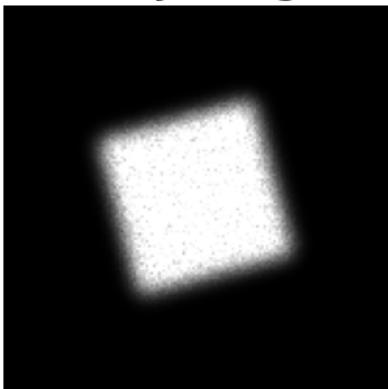
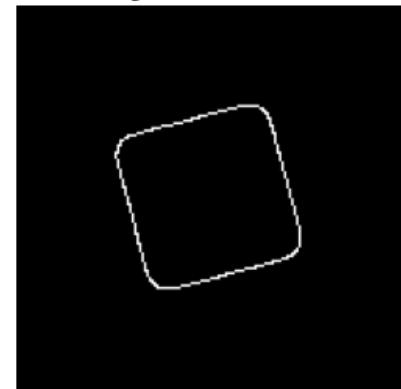
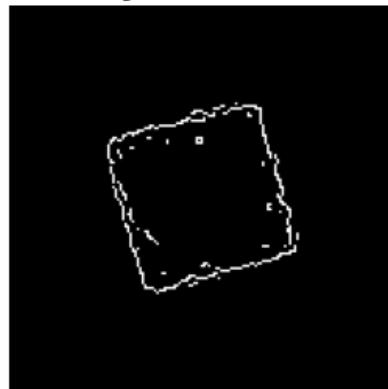
Total running time of the script: (0 minutes 0.261 seconds)

Canny edge detector

The Canny filter is a multi-stage edge detector. It uses a filter based on the derivative of a Gaussian in order to compute the intensity of the gradients. The Gaussian reduces the effect of noise present in the image. Then, potential edges are thinned down to 1-pixel curves by removing non-maximum pixels of the gradient magnitude. Finally, edge pixels are kept or removed using hysteresis thresholding on the gradient magnitude.

The Canny has three adjustable parameters: the width of the Gaussian (the noisier the image, the greater the width), and the low and high threshold for the hysteresis thresholding.

noisy image

Canny filter, $\sigma = 1$ 

```

import numpy as np
import matplotlib.pyplot as plt
from scipy import ndimage as ndi
from skimage.util import random_noise
from skimage import feature

# Generate noisy image of a square
image = np.zeros((128, 128), dtype=float)
image[32:-32, 32:-32] = 1

image = ndi.rotate(image, 15, mode='constant')
image = ndi.gaussian_filter(image, 4)
image = random_noise(image, mode='speckle', mean=0.1)

# Compute the Canny filter for two values of sigma
edges1 = feature.canny(image)
edges2 = feature.canny(image, sigma=3)

# display results
fig, ax = plt.subplots(nrows=1, ncols=3, figsize=(8, 3))

ax[0].imshow(image, cmap='gray')
ax[0].set_title('noisy image', fontsize=20)

ax[1].imshow(edges1, cmap='gray')
ax[1].set_title(r'Canny filter, $\sigma=1$', fontsize=20)

ax[2].imshow(edges2, cmap='gray')
ax[2].set_title(r'Canny filter, $\sigma=3$', fontsize=20)

for a in ax:
    a.axis('off')

fig.tight_layout()
plt.show()

```

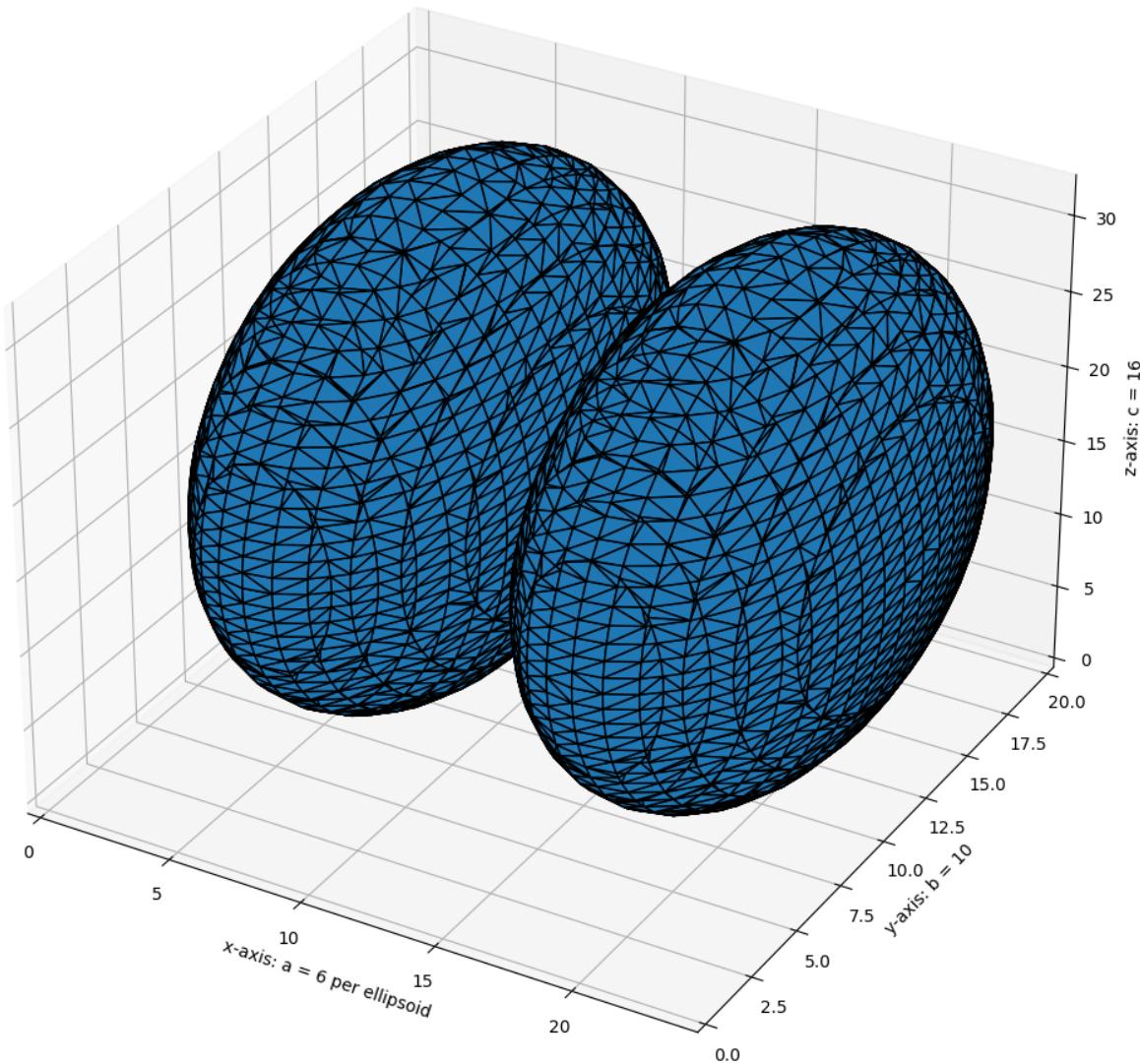
Total running time of the script: (0 minutes 0.194 seconds)

Marching Cubes

Marching cubes is an algorithm to extract a 2D surface mesh from a 3D volume. This can be conceptualized as a 3D generalization of isolines on topographical or weather maps. It works by iterating across the volume, looking for regions which cross the level of interest. If such regions are found, triangulations are generated and added to an output mesh. The final result is a set of vertices and a set of triangular faces.

The algorithm requires a data volume and an isosurface value. For example, in CT imaging Hounsfield units of +700 to +3000 represent bone. So, one potential input would be a reconstructed CT set of data and the value +700, to extract a mesh for regions of bone or bone-like density.

This implementation also works correctly on anisotropic datasets, where the voxel spacing is not equal for every spatial dimension, through use of the `spacing` kwarg.



```
import numpy as np
```

(continues on next page)

(continued from previous page)

```

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.art3d import Poly3DCollection

from skimage import measure
from skimage.draw import ellipsoid


# Generate a level set about zero of two identical ellipsoids in 3D
ellip_base = ellipsoid(6, 10, 16, levelset=True)
ellip_double = np.concatenate((ellip_base[:-1, ...],
                             ellip_base[2:, ...]), axis=0)

# Use marching cubes to obtain the surface mesh of these ellipsoids
verts, faces, normals, values = measure.marching_cubes(ellip_double, 0)

# Display resulting triangular mesh using Matplotlib. This can also be done
# with mayavi (see skimage.measure.marching_cubes docstring).
fig = plt.figure(figsize=(10, 10))
ax = fig.add_subplot(111, projection='3d')

# Fancy indexing: `verts[faces]` to generate a collection of triangles
mesh = Poly3DCollection(verts[faces])
mesh.set_edgecolor('k')
ax.add_collection3d(mesh)

ax.set_xlabel("x-axis: a = 6 per ellipsoid")
ax.set_ylabel("y-axis: b = 10")
ax.set_zlabel("z-axis: c = 16")

ax.set_xlim(0, 24) # a = 6 (times two for 2nd ellipsoid)
ax.set_ylim(0, 20) # b = 10
ax.set_zlim(0, 32) # c = 16

plt.tight_layout()
plt.show()

```

Total running time of the script: (0 minutes 0.696 seconds)

Ridge operators

Ridge filters can be used to detect ridge-like structures, such as neurites¹, tubes², vessels³, wrinkles⁴ or rivers.

Different ridge filters may be suited for detecting different structures, e.g., depending on contrast or noise level.

The present class of ridge filters relies on the eigenvalues of the Hessian matrix of image intensities to detect ridge structures where the intensity changes perpendicular but not along the structure.

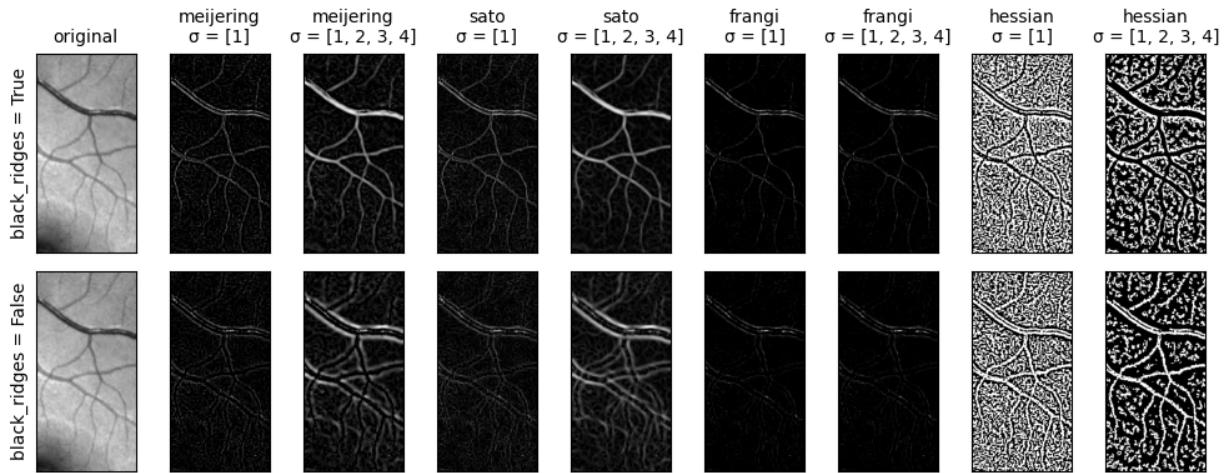
¹ Meijering, E., Jacob, M., Sarria, J. C., Steiner, P., Hirling, H., Unser, M. (2004). Design and validation of a tool for neurite tracing and analysis in fluorescence microscopy images. *Cytometry Part A*, 58(2), 167-176. DOI:10.1002/cyto.a.20022

² Sato, Y., Nakajima, S., Shiraga, N., Atsumi, H., Yoshida, S., Koller, T., ..., Kikinis, R. (1998). Three-dimensional multi-scale line filter for segmentation and visualization of curvilinear structures in medical images. *Medical image analysis*, 2(2), 143-168. DOI:10.1016/S1361-8415(98)80009-1

³ Frangi, A. F., Niessen, W. J., Vincken, K. L., & Viergever, M. A. (1998, October). Multiscale vessel enhancement filtering. In International Conference on Medical Image Computing and Computer-Assisted Intervention (pp. 130-137). Springer Berlin Heidelberg. DOI:10.1007/BFb0056195

⁴ Ng, C. C., Yap, M. H., Costen, N., & Li, B. (2014, November). Automatic wrinkle detection using hybrid Hessian filter. In Asian Conference on Computer Vision (pp. 609-622). Springer International Publishing. DOI:10.1007/978-3-319-16811-1_40

References



```

from skimage import data
from skimage import color
from skimage.filters import meijering, sato, frangi, hessian
import matplotlib.pyplot as plt

def original(image, **kwargs):
    """Return the original image, ignoring any kwargs."""
    return image

image = color.rgb2gray(data.retina())[300:700, 700:900]
cmap = plt.cm.gray

plt.rcParams["axes.titlesize"] = "medium"
axes = plt.figure(figsize=(10, 4)).subplots(2, 9)
for i, black_ridges in enumerate([True, False]):
    for j, (func, sigmas) in enumerate([
        (original, None),
        (meijering, [1]),
        (meijering, range(1, 5)),
        (sato, [1]),
        (sato, range(1, 5)),
        (frangi, [1]),
        (frangi, range(1, 5)),
        (hessian, [1]),
        (hessian, range(1, 5)),
    ]):
        result = func(image, black_ridges=black_ridges, sigmas=sigmas)
        axes[i, j].imshow(result, cmap=cmap)
        if i == 0:
            title = func.__name__
        if sigmas:
            title += f"\n\N{GREEK SMALL LETTER SIGMA} = {list(sigmas)}"


```

(continues on next page)

(continued from previous page)

```

        axes[i, j].set_title(title)
if j == 0:
    axes[i, j].set_ylabel(f' {black_ridges = }')
axes[i, j].set_xticks([])
axes[i, j].set_yticks([])

plt.tight_layout()
plt.show()

```

Total running time of the script: (0 minutes 3.149 seconds)

Active Contour Model

The active contour model is a method to fit open or closed splines to lines or edges in an image¹. It works by minimising an energy that is in part defined by the image and part by the spline's shape: length and smoothness. The minimization is done implicitly in the shape energy and explicitly in the image energy.

In the following two examples the active contour model is used (1) to segment the face of a person from the rest of an image by fitting a closed curve to the edges of the face and (2) to find the darkest curve between two fixed points while obeying smoothness considerations. Typically it is a good idea to smooth images a bit before analyzing, as done in the following examples.

We initialize a circle around the astronaut's face and use the default boundary condition `boundary_condition='periodic'` to fit a closed curve. The default parameters `w_line=0`, `w_edge=1` will make the curve search towards edges, such as the boundaries of the face.

```

import numpy as np
import matplotlib.pyplot as plt
from skimage.color import rgb2gray
from skimage import data
from skimage.filters import gaussian
from skimage.segmentation import active_contour

img = data.astronaut()
img = rgb2gray(img)

s = np.linspace(0, 2*np.pi, 400)
r = 100 + 100*np.sin(s)
c = 220 + 100*np.cos(s)
init = np.array([r, c]).T

snake = active_contour(gaussian(img, 3, preserve_range=False),
                       init, alpha=0.015, beta=10, gamma=0.001)

fig, ax = plt.subplots(figsize=(7, 7))
ax.imshow(img, cmap=plt.cm.gray)
ax.plot(init[:, 1], init[:, 0], '--r', lw=3)
ax.plot(snake[:, 1], snake[:, 0], '-b', lw=3)
ax.set_xticks([]), ax.set_yticks([])
ax.axis([0, img.shape[1], img.shape[0], 0])

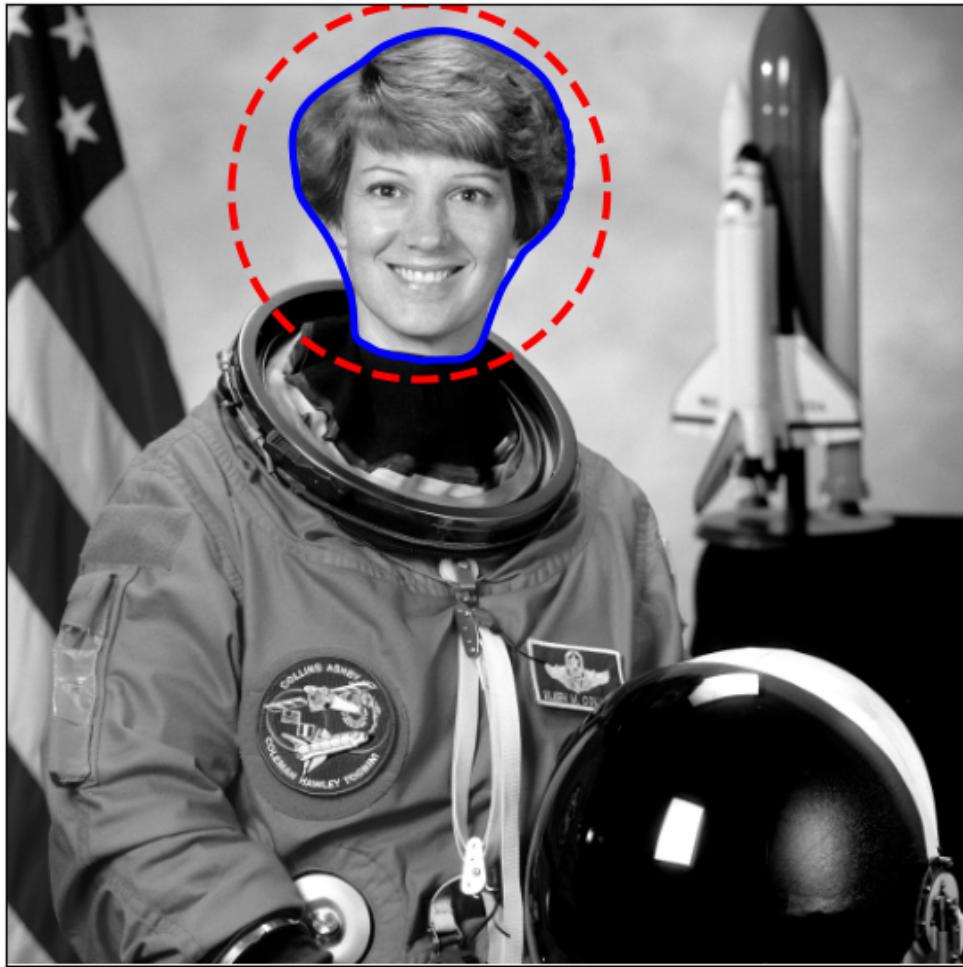
```

(continues on next page)

¹ *Snakes: Active contour models.* Kass, M.; Witkin, A.; Terzopoulos, D. International Journal of Computer Vision 1 (4): 321 (1988). DOI:10.1007/BF00133570

(continued from previous page)

```
plt.show()
```



Here we initialize a straight line between two points, $(5, 136)$ and $(424, 50)$, and require that the spline has its end points there by giving the boundary condition `boundary_condition='fixed'`. We furthermore make the algorithm search for dark lines by giving a negative `w_line` value.

```
img = data.text()  
  
r = np.linspace(136, 50, 100)  
c = np.linspace(5, 424, 100)  
init = np.array([r, c]).T
```

(continues on next page)

(continued from previous page)

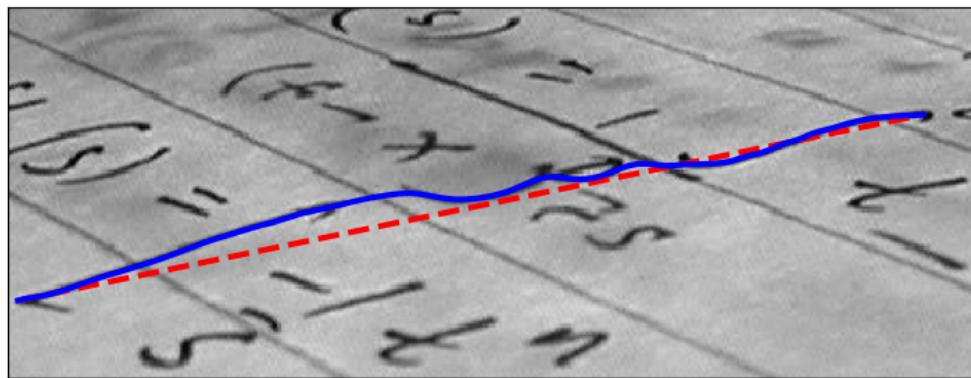
```

snake = active_contour(gaussian(img, 1, preserve_range=False),
                       init, boundary_condition='fixed',
                       alpha=0.1, beta=1.0, w_line=-5, w_edge=0, gamma=0.1)

fig, ax = plt.subplots(figsize=(9, 5))
ax.imshow(img, cmap=plt.cm.gray)
ax.plot(init[:, 1], init[:, 0], '--r', lw=3)
ax.plot(snake[:, 1], snake[:, 0], '-b', lw=3)
ax.set_xticks([]), ax.set_yticks([])
ax.axis([0, img.shape[1], img.shape[0], 0])

plt.show()

```



Total running time of the script: (0 minutes 0.589 seconds)

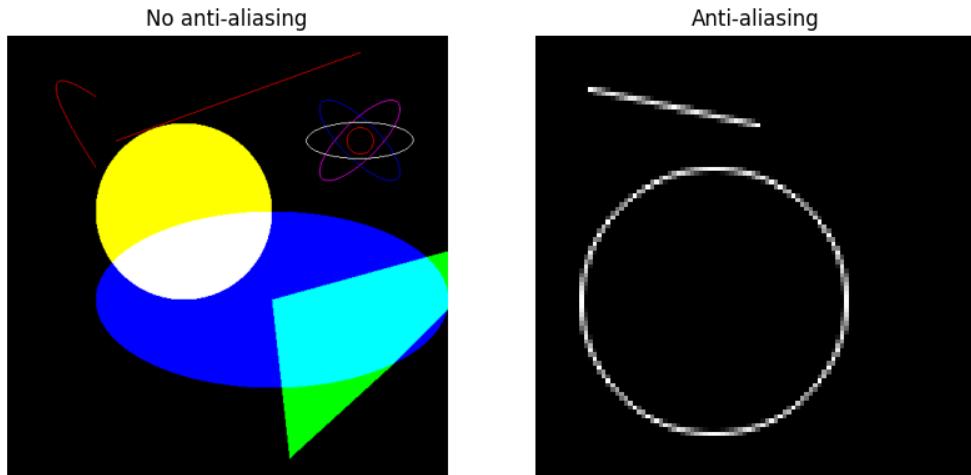
Shapes

This example shows how to draw several different shapes:

- line
- Bezier curve
- polygon
- disk
- ellipse

Anti-aliased drawing for:

- line
- circle



Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).

```

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

from skimage.draw import line, polygon, disk,
                        circle_perimeter,
                        ellipse, ellipse_perimeter,
                        bezier_curve

fig, (ax1, ax2) = plt.subplots(ncols=2, nrows=1, figsize=(10, 6))

img = np.zeros((500, 500, 3), dtype=np.double)

# draw line
rr, cc = line(120, 123, 20, 400)
img[rr, cc, 0] = 255

# fill polygon
poly = np.array([
    [100, 100], [200, 100], [200, 200], [100, 200], [100, 100]
])

```

(continues on next page)

(continued from previous page)

```

(300, 300),
(480, 320),
(380, 430),
(220, 590),
(300, 300),
))
rr, cc = polygon(poly[:, 0], poly[:, 1], img.shape)
img[rr, cc, 1] = 1

# fill circle
rr, cc = disk((200, 200), 100, shape=img.shape)
img[rr, cc, :] = (1, 1, 0)

# fill ellipse
rr, cc = ellipse(300, 300, 100, 200, img.shape)
img[rr, cc, 2] = 1

# circle
rr, cc = circle_perimeter(120, 400, 15)
img[rr, cc, :] = (1, 0, 0)

# Bezier curve
rr, cc = bezier_curve(70, 100, 10, 10, 150, 100, 1)
img[rr, cc, :] = (1, 0, 0)

# ellipses
rr, cc = ellipse_perimeter(120, 400, 60, 20, orientation=math.pi / 4.)
img[rr, cc, :] = (1, 0, 1)
rr, cc = ellipse_perimeter(120, 400, 60, 20, orientation=-math.pi / 4.)
img[rr, cc, :] = (0, 0, 1)
rr, cc = ellipse_perimeter(120, 400, 60, 20, orientation=math.pi / 2.)
img[rr, cc, :] = (1, 1, 1)

ax1.imshow(img)
ax1.set_title('No anti-aliasing')
ax1.axis('off')

from skimage.draw import line_aa, circle_perimeter_aa

img = np.zeros((100, 100), dtype=np.double)

# anti-aliased line
rr, cc, val = line_aa(12, 12, 20, 50)
img[rr, cc] = val

# anti-aliased circle
rr, cc, val = circle_perimeter_aa(60, 40, 30)
img[rr, cc] = val

```

(continues on next page)

(continued from previous page)

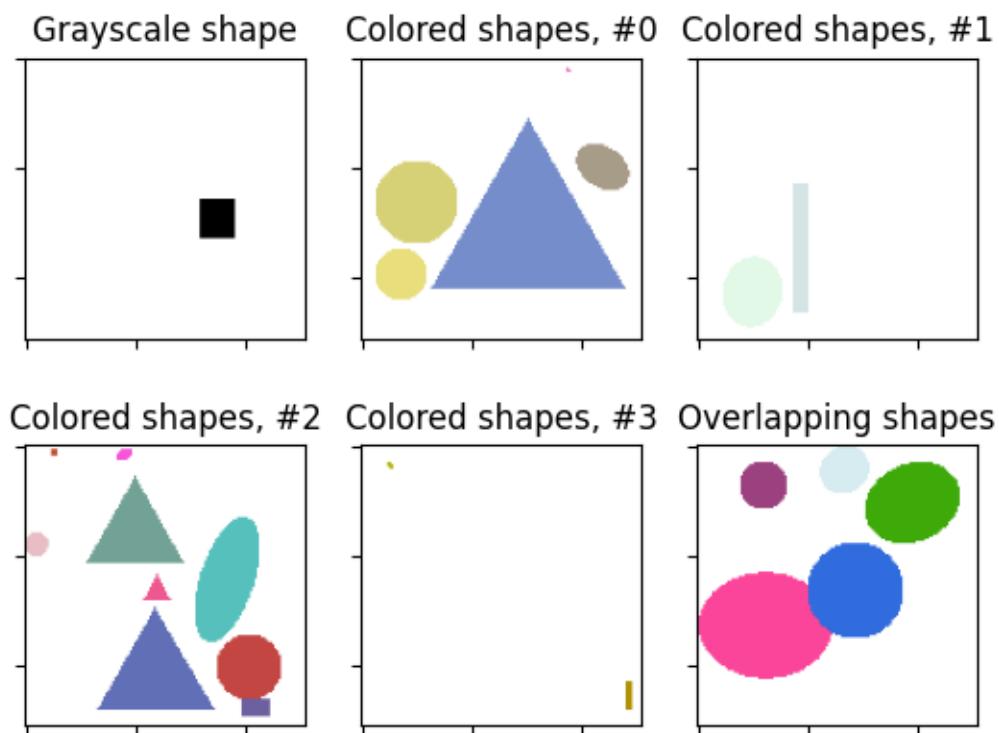
```
ax2.imshow(img, cmap=plt.cm.gray)
ax2.set_title('Anti-aliasing')
ax2.axis('off')

plt.show()
```

Total running time of the script: (0 minutes 0.129 seconds)

Random Shapes

Example of generating random shapes with particular properties.



```
Image shape: (128, 128)
Labels: [('rectangle', ((64, 82), (80, 96)))]
```

```
import matplotlib.pyplot as plt
from skimage.draw import random_shapes
```

(continues on next page)

(continued from previous page)

```

# Let's start simple and generate a 128x128 image
# with a single grayscale rectangle.
result = random_shapes((128, 128), max_shapes=1, shape='rectangle',
                        channel_axis=None, rng=0)

# We get back a tuple consisting of (1) the image with the generated shapes
# and (2) a list of label tuples with the kind of shape (e.g. circle,
# rectangle) and ((r0, r1), (c0, c1)) coordinates.
image, labels = result
print(f'Image shape: {image.shape}\nLabels: {labels}')

# We can visualize the images.
fig, axes = plt.subplots(nrows=2, ncols=3)
ax = axes.ravel()
ax[0].imshow(image, cmap='gray')
ax[0].set_title('Grayscale shape')

# The generated images can be much more complex. For example, let's try many
# shapes of any color. If we want the colors to be particularly light, we can
# set the `intensity_range` to an upper subrange of (0, 255).
image1, _ = random_shapes((128, 128), max_shapes=10,
                           intensity_range=((100, 255),))

# Moar :)
image2, _ = random_shapes((128, 128), max_shapes=10,
                           intensity_range=((200, 255),))
image3, _ = random_shapes((128, 128), max_shapes=10,
                           intensity_range=((50, 255),))
image4, _ = random_shapes((128, 128), max_shapes=10,
                           intensity_range=((0, 255),))

for i, image in enumerate([image1, image2, image3, image4], 1):
    ax[i].imshow(image)
    ax[i].set_title(f'Colored shapes, #{i-1}')

# These shapes are well suited to test segmentation algorithms. Often, we
# want shapes to overlap to test the algorithm. This is also possible:
image, _ = random_shapes((128, 128), min_shapes=5, max_shapes=10,
                           min_size=20, allow_overlap=True)
ax[5].imshow(image)
ax[5].set_title('Overlapping shapes')

for a in ax:
    a.set_xticklabels([])
    a.set_yticklabels([])

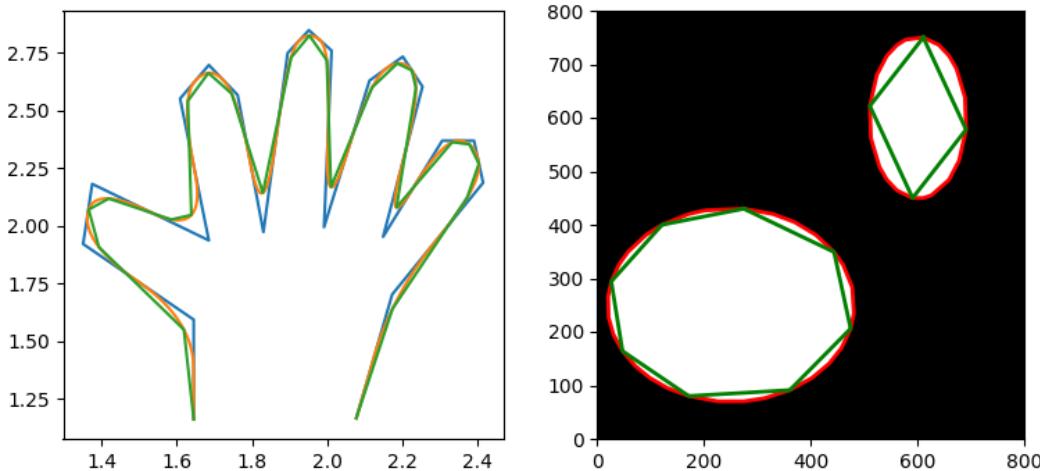
plt.show()

```

Total running time of the script: (0 minutes 0.249 seconds)

Approximate and subdivide polygons

This example shows how to approximate (Douglas-Peucker algorithm) and subdivide (B-Splines) polygonal chains.



```
Number of coordinates: 22 704 26
Number of coordinates: 1173 32 9
Number of coordinates: 701 21 5
```

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

from skimage.draw import ellipse
from skimage.measure import find_contours, approximate_polygon, \
    subdivide_polygon

hand = np.array([[1.64516129, 1.16145833],
                [1.64516129, 1.59375],
                [1.35080645, 1.921875],
                [1.375, 2.18229167],
                [1.68548387, 1.9375],
                [1.60887097, 2.55208333],
                [1.68548387, 2.69791667],
                [1.76209677, 2.56770833],
                [1.83064516, 1.97395833],
                [1.89516129, 2.75],
                [1.9516129, 2.84895833],
                [2.01209677, 2.76041667],
                [1.99193548, 1.99479167],
                [2.11290323, 2.63020833],
                [2.2016129, 2.734375],
```

(continues on next page)

(continued from previous page)

```
[2.25403226, 2.60416667],
[2.14919355, 1.953125],
[2.30645161, 2.36979167],
[2.39112903, 2.36979167],
[2.41532258, 2.1875],
[2.1733871, 1.703125],
[2.07782258, 1.16666667]])

# subdivide polygon using 2nd degree B-Splines
new_hand = hand.copy()
for _ in range(5):
    new_hand = subdivide_polygon(new_hand, degree=2, preserve_ends=True)

# approximate subdivided polygon with Douglas-Peucker algorithm
appr_hand = approximate_polygon(new_hand, tolerance=0.02)

print("Number of coordinates:", len(hand), len(new_hand), len(appr_hand))

fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(9, 4))

ax1.plot(hand[:, 0], hand[:, 1])
ax1.plot(new_hand[:, 0], new_hand[:, 1])
ax1.plot(appr_hand[:, 0], appr_hand[:, 1])

# create two ellipses in image
img = np.zeros((800, 800), 'int32')
rr, cc = ellipse(250, 250, 180, 230, img.shape)
img[rr, cc] = 1
rr, cc = ellipse(600, 600, 150, 90, img.shape)
img[rr, cc] = 1

plt.gray()
ax2.imshow(img)

# approximate / simplify coordinates of the two ellipses
for contour in find_contours(img, 0):
    coords = approximate_polygon(contour, tolerance=2.5)
    ax2.plot(coords[:, 1], coords[:, 0], '-r', linewidth=2)
    coords2 = approximate_polygon(contour, tolerance=39.5)
    ax2.plot(coords2[:, 1], coords2[:, 0], '-g', linewidth=2)
    print("Number of coordinates:", len(contour), len(coords), len(coords2))

ax2.axis((0, 800, 0, 800))

plt.show()
```

Total running time of the script: (0 minutes 0.173 seconds)

Straight line Hough transform

The Hough transform in its simplest form is a method to detect straight lines¹.

In the following example, we construct an image with a line intersection. We then use the Hough transform. to explore a parameter space for straight lines that may run through the image.

Algorithm overview

Usually, lines are parameterised as $y = mx + c$, with a gradient m and y-intercept c . However, this would mean that m goes to infinity for vertical lines. Instead, we therefore construct a segment perpendicular to the line, leading to the origin. The line is represented by the length of that segment, r , and the angle it makes with the x-axis, θ .

The Hough transform constructs a histogram array representing the parameter space (i.e., an $M \times N$ matrix, for M different values of the radius and N different values of θ). For each parameter combination, r and θ , we then find the number of non-zero pixels in the input image that would fall close to the corresponding line, and increment the array at position (r, θ) appropriately.

We can think of each non-zero pixel “voting” for potential line candidates. The local maxima in the resulting histogram indicates the parameters of the most probable lines. In our example, the maxima occur at 45 and 135 degrees, corresponding to the normal vector angles of each line.

Another approach is the Progressive Probabilistic Hough Transform². It is based on the assumption that using a random subset of voting points give a good approximation to the actual result, and that lines can be extracted during the voting process by walking along connected components. This returns the beginning and end of each line segment, which is useful.

The function `probabilistic_hough` has three parameters: a general threshold that is applied to the Hough accumulator, a minimum line length and the line gap that influences line merging. In the example below, we find lines longer than 10 with a gap less than 3 pixels.

References

Line Hough Transform

```
import numpy as np

from skimage.transform import hough_line, hough_line_peaks
from skimage.feature import canny
from skimage.draw import line as draw_line
from skimage import data

import matplotlib.pyplot as plt
from matplotlib import cm

# Constructing test image
image = np.zeros((200, 200))
idx = np.arange(25, 175)
```

(continues on next page)

¹ Duda, R. O. and P. E. Hart, “Use of the Hough Transformation to Detect Lines and Curves in Pictures,” Comm. ACM, Vol. 15, pp. 11-15 (January, 1972)

² 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.

(continued from previous page)

```
image[idx, idx] = 255
image[draw_line(45, 25, 25, 175)] = 255
image[draw_line(25, 135, 175, 155)] = 255

# Classic straight-line Hough transform
# Set a precision of 0.5 degree.
tested_angles = np.linspace(-np.pi / 2, np.pi / 2, 360, endpoint=False)
h, theta, d = hough_line(image, theta=tested_angles)

# Generating figure 1
fig, axes = plt.subplots(1, 3, figsize=(15, 6))
ax = axes.ravel()

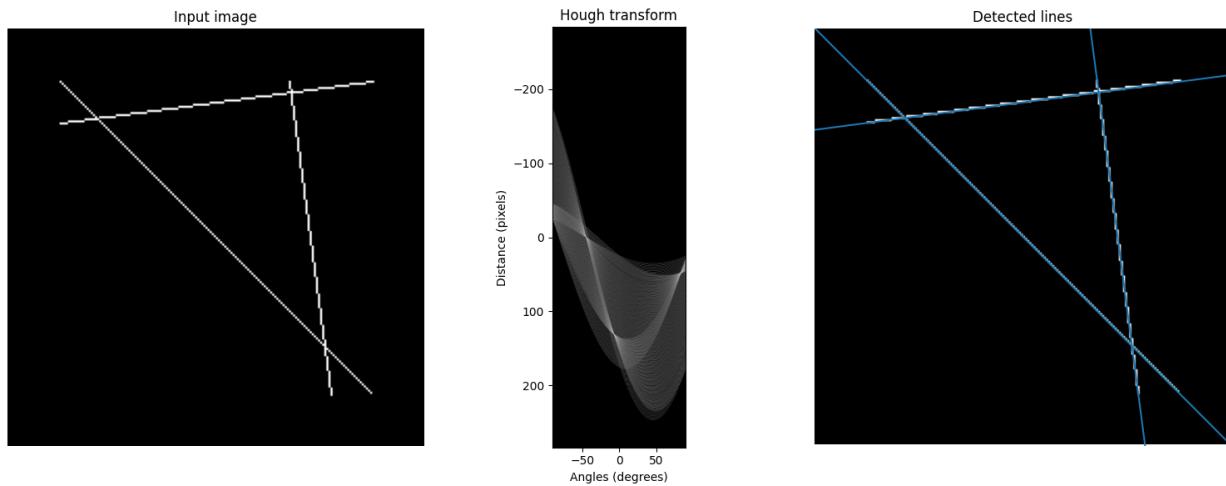
ax[0].imshow(image, cmap=cm.gray)
ax[0].set_title('Input image')
ax[0].set_axis_off()

angle_step = 0.5 * np.diff(theta).mean()
d_step = 0.5 * np.diff(d).mean()
bounds = [np.rad2deg(theta[0] - angle_step),
          np.rad2deg(theta[-1] + angle_step),
          d[-1] + d_step, d[0] - d_step]
ax[1].imshow(np.log(1 + h), extent=bounds, cmap=cm.gray, aspect=1 / 1.5)
ax[1].set_title('Hough transform')
ax[1].set_xlabel('Angles (degrees)')
ax[1].set_ylabel('Distance (pixels)')
ax[1].axis('image')

ax[2].imshow(image, cmap=cm.gray)
ax[2].set_ylim((image.shape[0], 0))
ax[2].set_axis_off()
ax[2].set_title('Detected lines')

for _, angle, dist in zip(*hough_line_peaks(h, theta, d)):
    (x0, y0) = dist * np.array([np.cos(angle), np.sin(angle)])
    ax[2].axline((x0, y0), slope=np.tan(angle + np.pi/2))

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



Probabilistic Hough Transform

```
from skimage.transform import probabilistic_hough_line

# Line finding using the Probabilistic Hough Transform
image = data.camera()
edges = canny(image, 2, 1, 25)
lines = probabilistic_hough_line(edges, threshold=10, line_length=5,
                                 line_gap=3)

# Generating figure 2
fig, axes = plt.subplots(1, 3, figsize=(15, 5), sharex=True, sharey=True)
ax = axes.ravel()

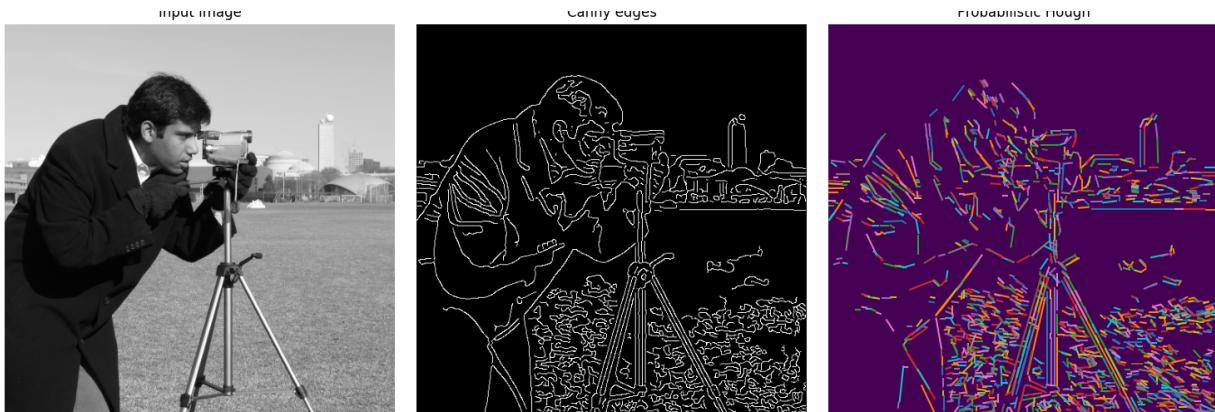
ax[0].imshow(image, cmap=cm.gray)
ax[0].set_title('Input image')

ax[1].imshow(edges, cmap=cm.gray)
ax[1].set_title('Canny edges')

ax[2].imshow(edges * 0)
for line in lines:
    p0, p1 = line
    ax[2].plot((p0[0], p1[0]), (p0[1], p1[1]))
ax[2].set_xlim((0, image.shape[1]))
ax[2].set_ylim((image.shape[0], 0))
ax[2].set_title('Probabilistic Hough')

for a in ax:
    a.set_axis_off()

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



Total running time of the script: (0 minutes 1.703 seconds)

Circular and Elliptical Hough Transforms

The Hough transform in its simplest form is a method to detect straight lines but it can also be used to detect circles or ellipses. The algorithm assumes that the edge is detected and it is robust against noise or missing points.

Circle detection

In the following example, the Hough transform is used to detect coin positions and match their edges. We provide a range of plausible radii. For each radius, two circles are extracted and we finally keep the five most prominent candidates. The result shows that coin positions are well-detected.

Algorithm overview

Given a black circle on a white background, we first guess its radius (or a range of radii) to construct a new circle. This circle is applied on each black pixel of the original picture and the coordinates of this circle are voting in an accumulator. From this geometrical construction, the original circle center position receives the highest score.

Note that the accumulator size is built to be larger than the original picture in order to detect centers outside the frame. Its size is extended by two times the larger radius.

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

from skimage import data, color
from skimage.transform import hough_circle, hough_circle_peaks
from skimage.feature import canny
from skimage.draw import circle_perimeter
from skimage.util import img_as_ubyte

# Load picture and detect edges
image = img_as_ubyte(data.coins()[160:230, 70:270])
edges = canny(image, sigma=3, low_threshold=10, high_threshold=50)

# Detect two radii
hough_radii = np.arange(20, 35, 2)
```

(continues on next page)

(continued from previous page)

```

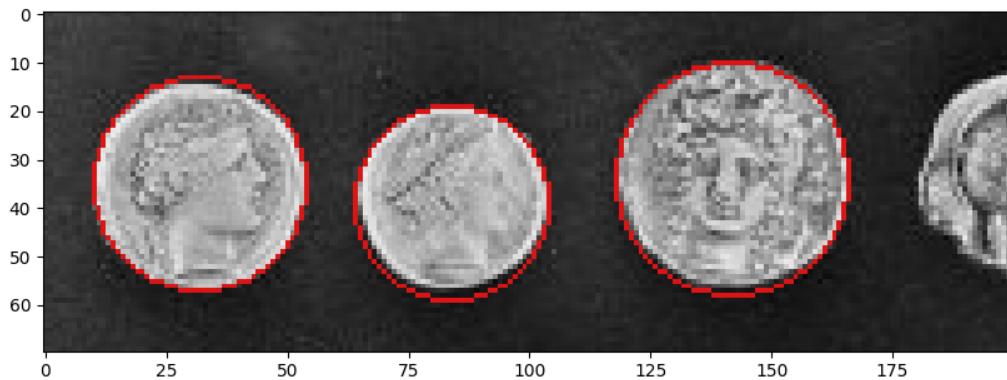
hough_res = hough_circle(edges, hough_radii)

# Select the most prominent 3 circles
accums, cx, cy, radii = hough_circle_peaks(hough_res, hough_radii,
                                             total_num_peaks=3)

# Draw them
fig, ax = plt.subplots(ncols=1, nrows=1, figsize=(10, 4))
image = color.gray2rgb(image)
for center_y, center_x, radius in zip(cy, cx, radii):
    circy, circx = circle_perimeter(center_y, center_x, radius,
                                     shape=image.shape)
    image[circy, circx] = (220, 20, 20)

ax.imshow(image, cmap=plt.cm.gray)
plt.show()

```



Ellipse detection

In this second example, the aim is to detect the edge of a coffee cup. Basically, this is a projection of a circle, i.e. an ellipse. The problem to solve is much more difficult because five parameters have to be determined, instead of three for circles.

Algorithm overview

The algorithm takes two different points belonging to the ellipse. It assumes that it is the main axis. A loop on all the other points determines how much an ellipse passes to them. A good match corresponds to high accumulator values.

A full description of the algorithm can be found in reference¹.

¹ Xie, Yonghong, and Qiang Ji. "A new efficient ellipse detection method." Pattern Recognition, 2002. Proceedings. 16th International Conference on. Vol. 2. IEEE, 2002

References

```

import matplotlib.pyplot as plt

from skimage import data, color, img_as_ubyte
from skimage.feature import canny
from skimage.transform import hough_ellipse
from skimage.draw import ellipse_perimeter

# Load picture, convert to grayscale and detect edges
image_rgb = data.coffee()[0:220, 160:420]
image_gray = color.rgb2gray(image_rgb)
edges = canny(image_gray, sigma=2.0,
              low_threshold=0.55, high_threshold=0.8)

# Perform a Hough Transform
# The accuracy corresponds to the bin size of a major axis.
# The value is chosen in order to get a single high accumulator.
# The threshold eliminates low accumulators
result = hough_ellipse(edges, accuracy=20, threshold=250,
                       min_size=100, max_size=120)
result.sort(order='accumulator')

# Estimated parameters for the ellipse
best = list(result[-1])
yc, xc, a, b = (int(round(x)) for x in best[1:5])
orientation = best[5]

# Draw the ellipse on the original image
cy, cx = ellipse_perimeter(yc, xc, a, b, orientation)
image_rgb[cy, cx] = (0, 0, 255)
# Draw the edge (white) and the resulting ellipse (red)
edges = color.gray2rgb(img_as_ubyte(edges))
edges[cy, cx] = (250, 0, 0)

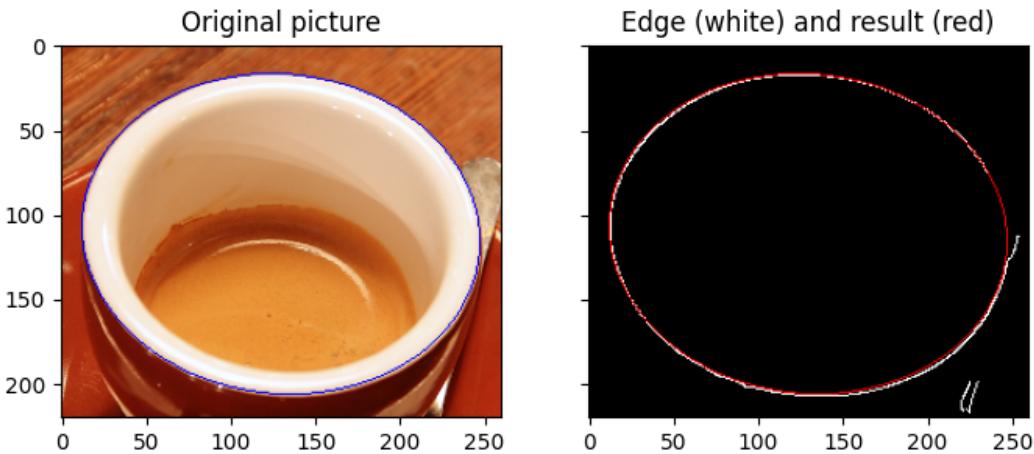
fig2, (ax1, ax2) = plt.subplots(ncols=2, nrows=1, figsize=(8, 4),
                               sharex=True, sharey=True)

ax1.set_title('Original picture')
ax1.imshow(image_rgb)

ax2.set_title('Edge (white) and result (red)')
ax2.imshow(edges)

plt.show()

```



Total running time of the script: (0 minutes 6.132 seconds)

Skeletonize

Skeletonization reduces binary objects to 1 pixel wide representations. This can be useful for feature extraction, and/or representing an object's topology.

`skeletonize` works by making successive passes of the image. On each pass, border pixels are identified and removed on the condition that they do not break the connectivity of the corresponding object.

```
from skimage.morphology import skeletonize
from skimage import data
import matplotlib.pyplot as plt
from skimage.util import invert

# Invert the horse image
image = invert(data.horse())

# perform skeletonization
skeleton = skeletonize(image)

# display results
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(8, 4),
                        sharex=True, sharey=True)

ax = axes.ravel()

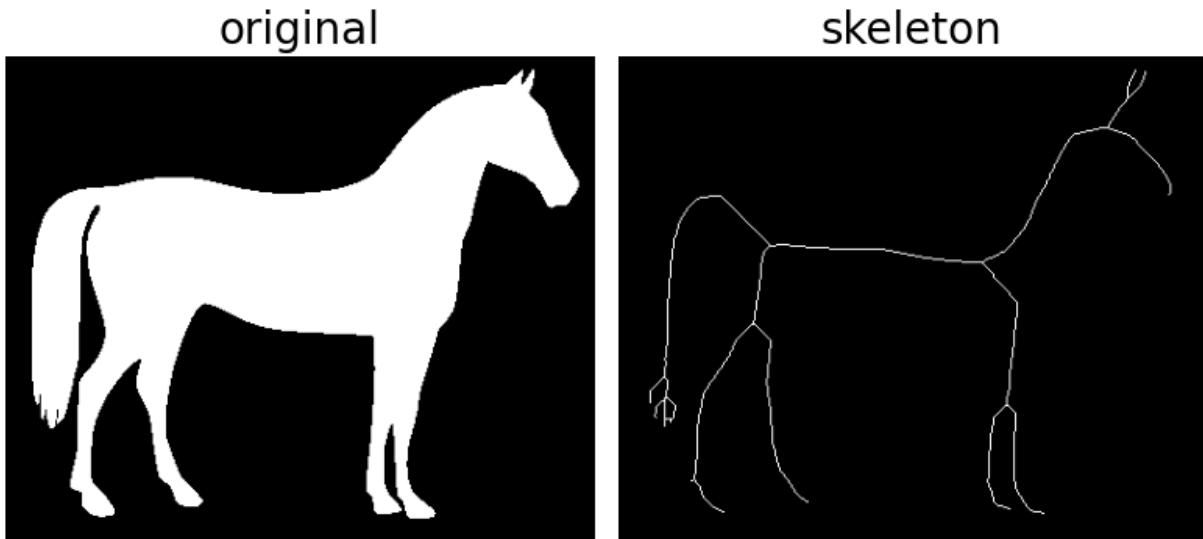
ax[0].imshow(image, cmap=plt.cm.gray)
ax[0].axis('off')
ax[0].set_title('original', fontsize=20)

ax[1].imshow(skeleton, cmap=plt.cm.gray)
ax[1].axis('off')
ax[1].set_title('skeleton', fontsize=20)
```

(continues on next page)

(continued from previous page)

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



Zhang's method vs Lee's method

`skeletonize` [?] works by making successive passes of the image, removing pixels on object borders. This continues until no more pixels can be removed. The image is correlated with a mask that assigns each pixel a number in the range [0...255] corresponding to each possible pattern of its 8 neighboring pixels. A look up table is then used to assign the pixels a value of 0, 1, 2 or 3, which are selectively removed during the iterations.

`skeletonize(..., method='lee')` [?] uses an octree data structure to examine a 3x3x3 neighborhood of a pixel. The algorithm proceeds by iteratively sweeping over the image, and removing pixels at each iteration until the image stops changing. Each iteration consists of two steps: first, a list of candidates for removal is assembled; then pixels from this list are rechecked sequentially, to better preserve connectivity of the image.

Note that Lee's method [?] is designed to be used on 3-D images, and is selected automatically for those. For illustrative purposes, we apply this algorithm to a 2-D image.

```
import matplotlib.pyplot as plt
from skimage.morphology import skeletonize

blobs = data.binary_blobs(200, blob_size_fraction=.2,
                         volume_fraction=.35, rng=1)

skeleton = skeletonize(blobs)
skeleton_lee = skeletonize(blobs, method='lee')

fig, axes = plt.subplots(1, 3, figsize=(8, 4), sharex=True, sharey=True)
ax = axes.ravel()

ax[0].imshow(blobs, cmap=plt.cm.gray)
ax[0].set_title('original')
```

(continues on next page)

(continued from previous page)

```

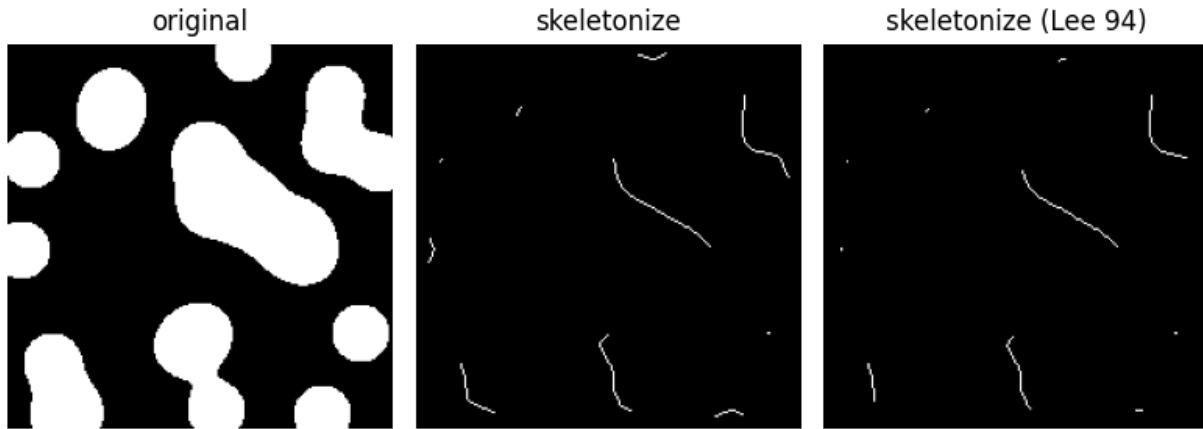
ax[0].axis('off')

ax[1].imshow(skeleton, cmap=plt.cm.gray)
ax[1].set_title('skeletonize')
ax[1].axis('off')

ax[2].imshow(skeleton_lee, cmap=plt.cm.gray)
ax[2].set_title('skeletonize (Lee 94)')
ax[2].axis('off')

fig.tight_layout()
plt.show()

```



Medial axis skeletonization

The medial axis of an object is the set of all points having more than one closest point on the object's boundary. It is often called the *topological skeleton*, because it is a 1-pixel wide skeleton of the object, with the same connectivity as the original object.

Here, we use the medial axis transform to compute the width of the foreground objects. As the function `medial_axis` returns the distance transform in addition to the medial axis (with the keyword argument `return_distance=True`), it is possible to compute the distance to the background for all points of the medial axis with this function. This gives an estimate of the local width of the objects.

For a skeleton with fewer branches, `skeletonize` should be preferred.

```

from skimage.morphology import medial_axis, skeletonize

# Generate the data
blobs = data.binary_blobs(200, blob_size_fraction=.2,
                          volume_fraction=.35, rng=1)

# Compute the medial axis (skeleton) and the distance transform

```

(continues on next page)

(continued from previous page)

```
skel, distance = medial_axis(blobs, return_distance=True)

# Compare with other skeletonization algorithms
skeleton = skeletonize(blobs)
skeleton_lee = skeletonize(blobs, method='lee')

# Distance to the background for pixels of the skeleton
dist_on_skel = distance * skel

fig, axes = plt.subplots(2, 2, figsize=(8, 8), sharex=True, sharey=True)
ax = axes.ravel()

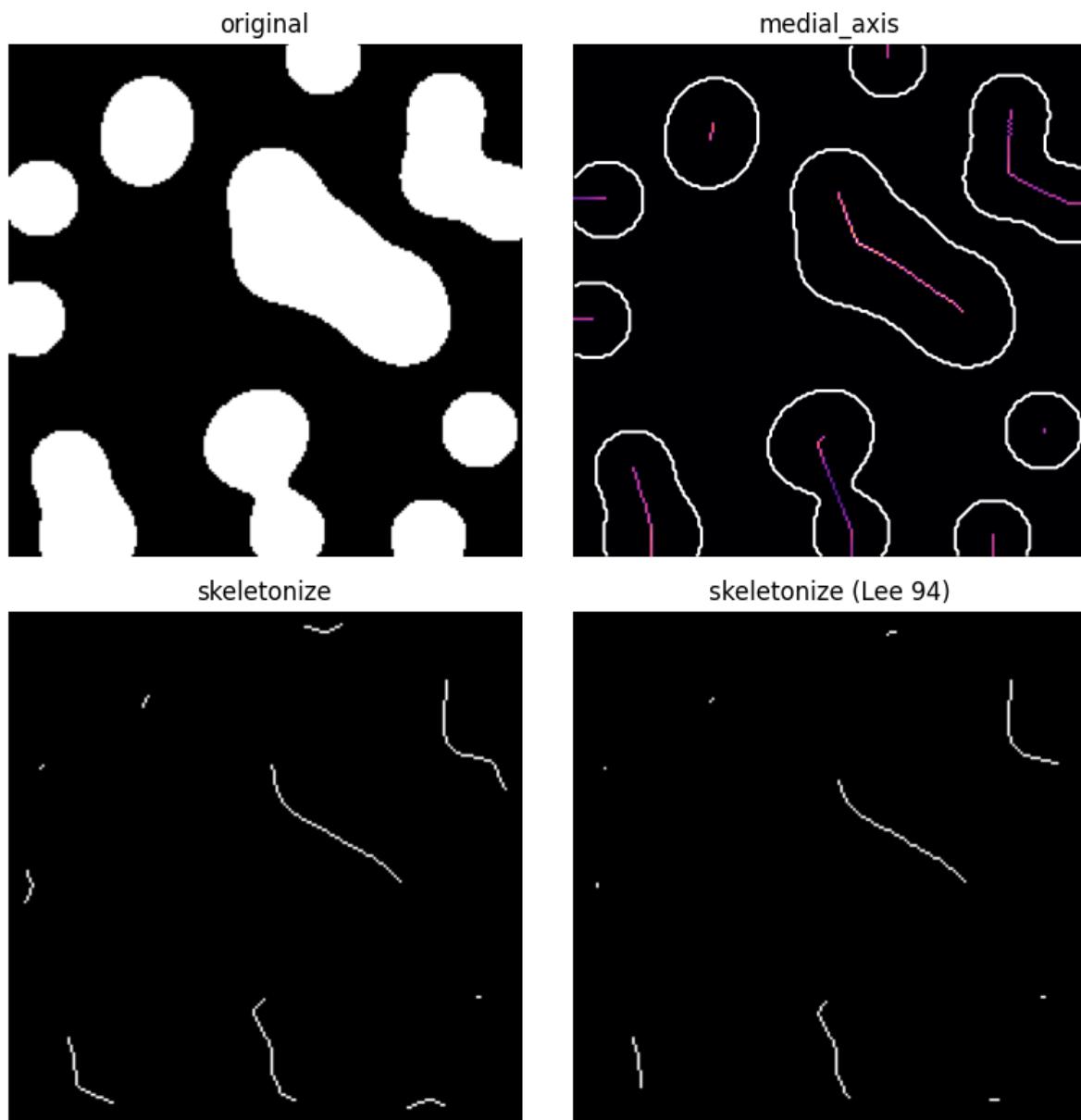
ax[0].imshow(blobs, cmap=plt.cm.gray)
ax[0].set_title('original')
ax[0].axis('off')

ax[1].imshow(dist_on_skel, cmap='magma')
ax[1].contour(blobs, [0.5], colors='w')
ax[1].set_title('medial_axis')
ax[1].axis('off')

ax[2].imshow(skeleton, cmap=plt.cm.gray)
ax[2].set_title('skeletonize')
ax[2].axis('off')

ax[3].imshow(skeleton_lee, cmap=plt.cm.gray)
ax[3].set_title("skeletonize (Lee 94)")
ax[3].axis('off')

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



Morphological thinning

Morphological thinning, implemented in the `thin` function, works on the same principle as `skeletonize`: remove pixels from the borders at each iteration until none can be removed without altering the connectivity. The different rules of removal can speed up skeletonization and result in different final skeletons.

The `thin` function also takes an optional `max_num_iter` keyword argument to limit the number of thinning iterations, and thus produce a relatively thicker skeleton.

```
from skimage.morphology import skeletonize, thin

skeleton = skeletonize(image)
thinned = thin(image)
thinned_partial = thin(image, max_num_iter=25)
```

(continues on next page)

(continued from previous page)

```
fig, axes = plt.subplots(2, 2, figsize=(8, 8), sharex=True, sharey=True)
ax = axes.ravel()

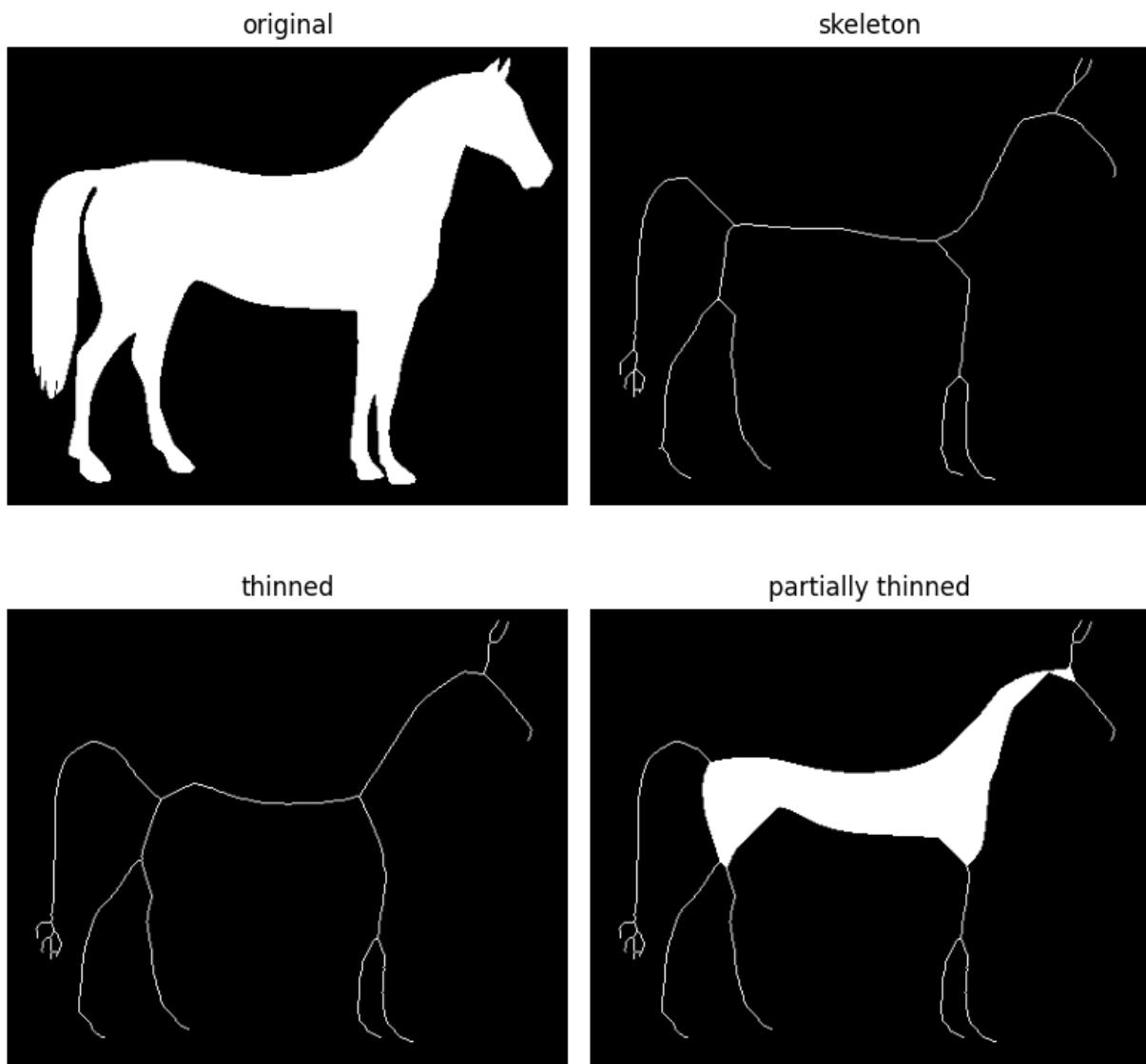
ax[0].imshow(image, cmap=plt.cm.gray)
ax[0].set_title('original')
ax[0].axis('off')

ax[1].imshow(skeleton, cmap=plt.cm.gray)
ax[1].set_title('skeleton')
ax[1].axis('off')

ax[2].imshow(thinned, cmap=plt.cm.gray)
ax[2].set_title('thinned')
ax[2].axis('off')

ax[3].imshow(thinned_partial, cmap=plt.cm.gray)
ax[3].set_title('partially thinned')
ax[3].axis('off')

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



Total running time of the script: (0 minutes 1.505 seconds)

Edge operators

Edge operators are used in image processing within edge detection algorithms. They are discrete differentiation operators, computing an approximation of the gradient of the image intensity function.

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

from skimage import filters
from skimage.data import camera
from skimage.util import compare_images
```

(continues on next page)

(continued from previous page)

```

image = camera()
edge_roberts = filters.roberts(image)
edge_sobel = filters.sobel(image)

fig, axes = plt.subplots(ncols=2, sharex=True, sharey=True,
                       figsize=(8, 4))

axes[0].imshow(edge_roberts, cmap=plt.cm.gray)
axes[0].set_title('Roberts Edge Detection')

axes[1].imshow(edge_sobel, cmap=plt.cm.gray)
axes[1].set_title('Sobel Edge Detection')

for ax in axes:
    ax.axis('off')

plt.tight_layout()
plt.show()

```

Roberts Edge Detection



Sobel Edge Detection



Different operators compute different finite-difference approximations of the gradient. For example, the Scharr filter results in a less rotational variance than the Sobel filter that is in turn better than the Prewitt filter¹²³. The difference between the Prewitt and Sobel filters and the Scharr filter is illustrated below with an image that is the discretization of a rotation- invariant continuous function. The discrepancy between the Prewitt and Sobel filters, and the Scharr filter is stronger for regions of the image where the direction of the gradient is close to diagonal, and for regions with high spatial frequencies. For the example image the differences between the filter results are very small and the filter results are visually almost indistinguishable.

¹ https://en.wikipedia.org/wiki/Sobel_operator#Alternative_operators

² B. Jaehne, H. Scharr, and S. Koerkel. Principles of filter design. In Handbook of Computer Vision and Applications. Academic Press, 1999.

³ https://en.wikipedia.org/wiki/Prewitt_operator

```
x, y = np.ogrid[:100, :100]

# Creating a rotation-invariant image with different spatial frequencies.
image_rot = np.exp(1j * np.hypot(x, y) ** 1.3 / 20.).real

edge_sobel = filters.sobel(image_rot)
edge_scharr = filters.scharr(image_rot)
edge_prewitt = filters.prewitt(image_rot)

diff_scharr_prewitt = compare_images(edge_scharr, edge_prewitt)
diff_scharr_sobel = compare_images(edge_scharr, edge_sobel)
max_diff = np.max(np.maximum(diff_scharr_prewitt, diff_scharr_sobel))

fig, axes = plt.subplots(nrows=2, ncols=2, sharex=True, sharey=True,
                        figsize=(8, 8))
axes = axes.ravel()

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

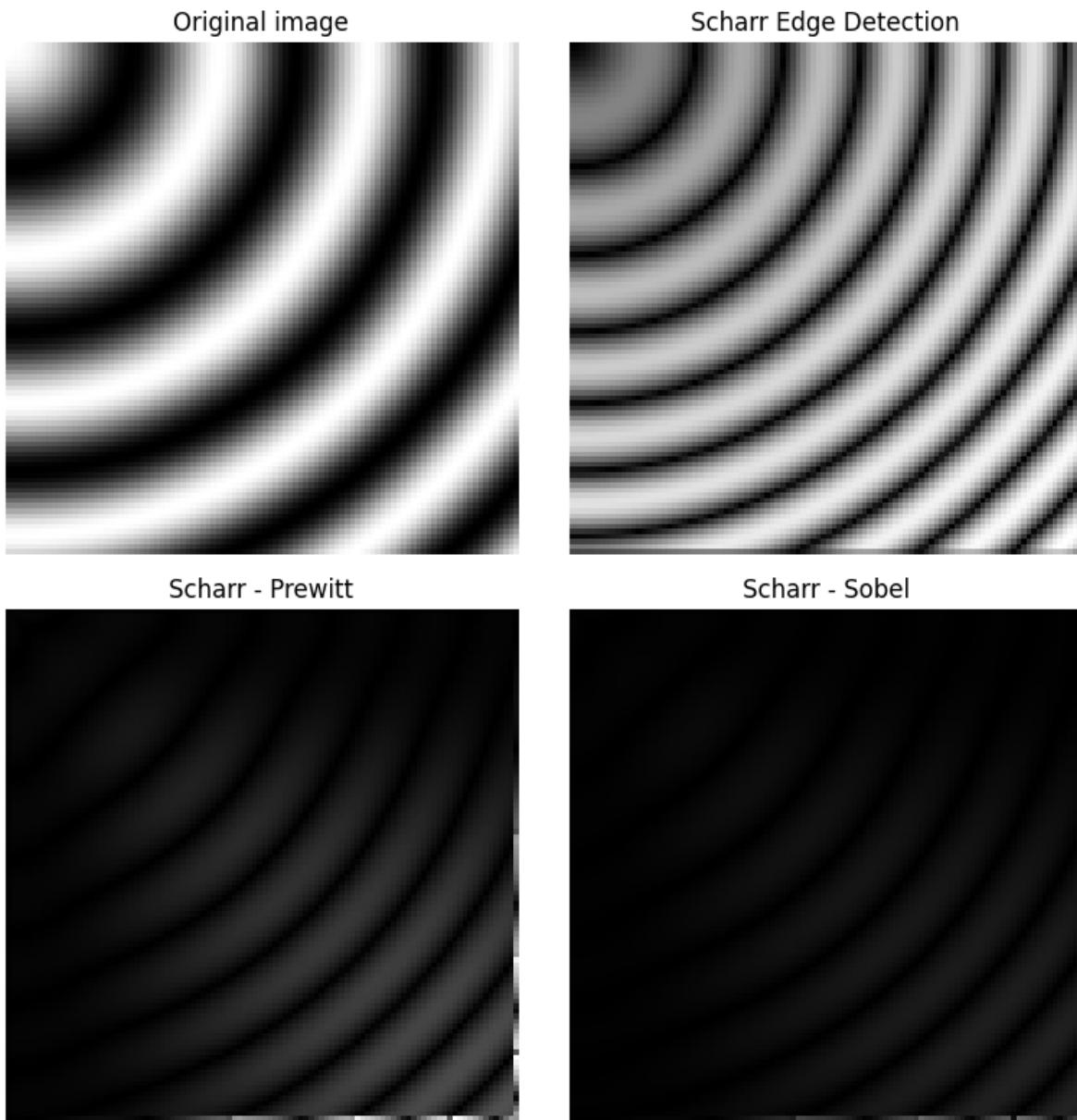
axes[1].imshow(edge_scharr, cmap=plt.cm.gray)
axes[1].set_title('Scharr Edge Detection')

axes[2].imshow(diff_scharr_prewitt, cmap=plt.cm.gray, vmax=max_diff)
axes[2].set_title('Scharr - Prewitt')

axes[3].imshow(diff_scharr_sobel, cmap=plt.cm.gray, vmax=max_diff)
axes[3].set_title('Scharr - Sobel')

for ax in axes:
    ax.axis('off')

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



As in the previous example, here we illustrate the rotational invariance of the filters. The top row shows a rotationally invariant image along with the angle of its analytical gradient. The other two rows contain the difference between the different gradient approximations (Sobel, Prewitt, Scharr & Farid) and analytical gradient.

The Farid & Simoncelli derivative filters⁴⁵ are the most rotationally invariant, but require a 5x5 kernel, which is computationally more intensive than a 3x3 kernel.

```
x, y = np.mgrid[-10:10:255j, -10:10:255j]
image_rotinv = np.sin(x ** 2 + y ** 2)
```

(continues on next page)

⁴ Farid, H. and Simoncelli, E. P., “Differentiation of discrete multidimensional signals”, IEEE Transactions on Image Processing 13(4): 496-508, 2004. DOI:10.1109/TIP.2004.823819

⁵ Wikipedia, “Farid and Simoncelli Derivatives.” Available at: <https://en.wikipedia.org/wiki/Image_derivatives#Farid_and_Simoncelli_Derivatives>

(continued from previous page)

```
image_x = 2 * x * np.cos(x ** 2 + y ** 2)
image_y = 2 * y * np.cos(x ** 2 + y ** 2)

def angle(dx, dy):
    """Calculate the angles between horizontal and vertical operators."""
    return np.mod(np.arctan2(dy, dx), np.pi)

true_angle = angle(image_x, image_y)

angle_farid = angle(filters.farid_h(image_rotinv),
                    filters.farid_v(image_rotinv))
angle_sobel = angle(filters.sobel_h(image_rotinv),
                    filters.sobel_v(image_rotinv))
angle_scharr = angle(filters.scharr_h(image_rotinv),
                     filters.scharr_v(image_rotinv))
angle_prewitt = angle(filters.prewitt_h(image_rotinv),
                      filters.prewitt_v(image_rotinv))

def diff_angle(angle_1, angle_2):
    """Calculate the differences between two angles."""
    return np.minimum(np.pi - np.abs(angle_1 - angle_2),
                     np.abs(angle_1 - angle_2))

diff_farid = diff_angle(true_angle, angle_farid)
diff_sobel = diff_angle(true_angle, angle_sobel)
diff_scharr = diff_angle(true_angle, angle_scharr)
diff_prewitt = diff_angle(true_angle, angle_prewitt)

fig, axes = plt.subplots(nrows=3, ncols=2, sharex=True, sharey=True,
                        figsize=(8, 8))
axes = axes.ravel()

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

axes[1].imshow(true_angle, cmap=plt.cm.hsv)
axes[1].set_title('Analytical gradient angle')

axes[2].imshow(diff_sobel, cmap=plt.cm.inferno, vmin=0, vmax=0.02)
axes[2].set_title('Sobel error')

axes[3].imshow(diff_prewitt, cmap=plt.cm.inferno, vmin=0, vmax=0.02)
axes[3].set_title('Prewitt error')

axes[4].imshow(diff_scharr, cmap=plt.cm.inferno, vmin=0, vmax=0.02)
axes[4].set_title('Scharr error')

color_ax = axes[5].imshow(diff_farid, cmap=plt.cm.inferno, vmin=0, vmax=0.02)
axes[5].set_title('Farid error')
```

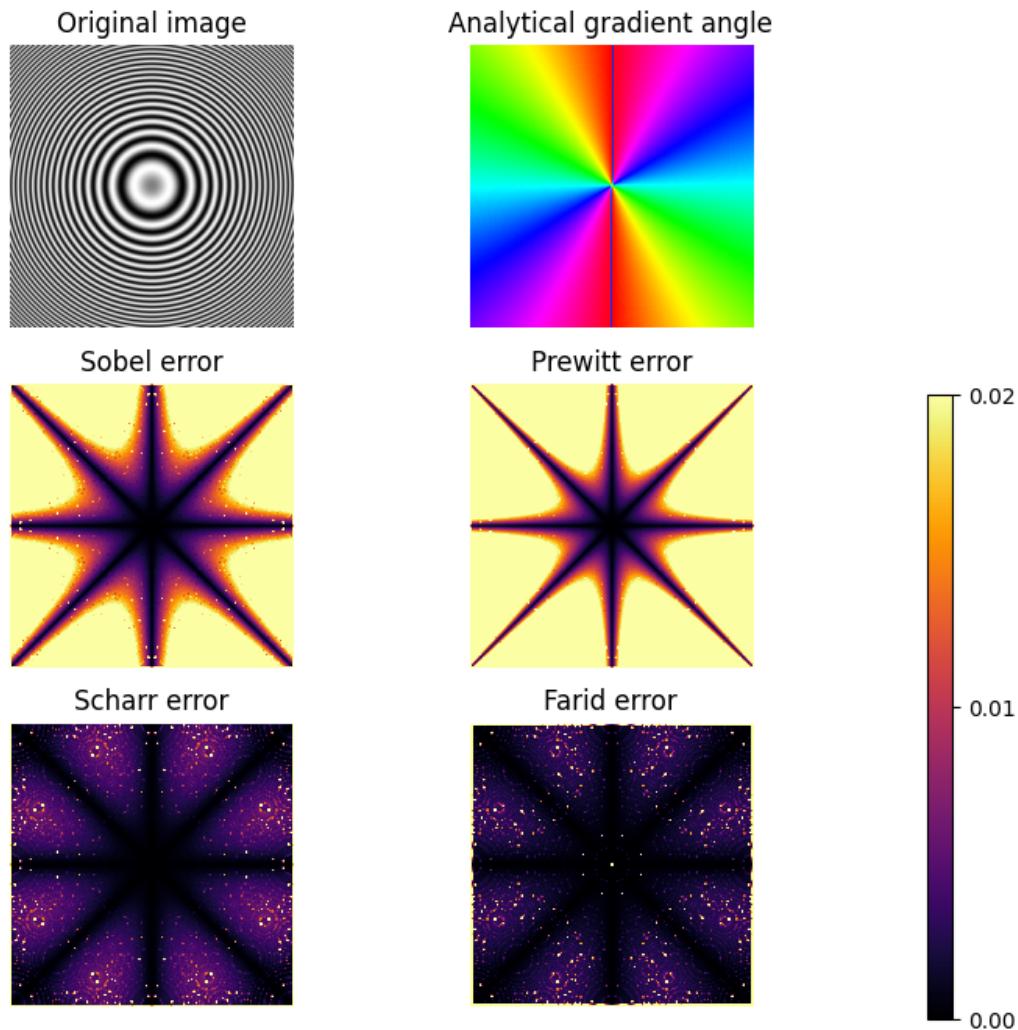
(continues on next page)

(continued from previous page)

```
fig.subplots_adjust(right=0.8)
colorbar_ax = fig.add_axes([0.90, 0.10, 0.02, 0.50])
fig.colorbar(colorbar_ax, cax=colorbar_ax, ticks=[0, 0.01, 0.02])

for ax in axes:
    ax.axis('off')

plt.show()
```



Total running time of the script: (0 minutes 0.864 seconds)

Geometrical transformations and registration

Swirl

Image swirling is a non-linear image deformation that creates a whirlpool effect. This example describes the implementation of this transform in `skimage`, as well as the underlying warp mechanism.

Image warping

When applying a geometric transformation on an image, we typically make use of a reverse mapping, i.e., for each pixel in the output image, we compute its corresponding position in the input. The reason is that, if we were to do it the other way around (map each input pixel to its new output position), some pixels in the output may be left empty. On the other hand, each output coordinate has exactly one corresponding location in (or outside) the input image, and even if that position is non-integer, we may use interpolation to compute the corresponding image value.

Performing a reverse mapping

To perform a geometric warp in `skimage`, you simply need to provide the reverse mapping to the `skimage.transform.warp()` function. E.g., consider the case where we would like to shift an image 50 pixels to the left. The reverse mapping for such a shift would be:

```
def shift_left(xy):
    xy[:, 0] += 50
    return xy
```

The corresponding call to warp is:

```
from skimage.transform import warp
warp(image, shift_left)
```

The swirl transformation

Consider the coordinate (x, y) in the output image. The reverse mapping for the swirl transformation first computes, relative to a center (x_0, y_0) , its polar coordinates,

$$\begin{aligned}\theta &= \arctan((y - y_0)/(x - x_0)) \\ \rho &= \sqrt{(x - x_0)^2 + (y - y_0)^2},\end{aligned}$$

and then transforms them according to

$$\begin{aligned}r &= \ln(2) \text{radius}/5 \\ \phi &= \text{rotation} \\ s &= \text{strength} \\ \theta' &= \phi + s e^{-\rho/r} + \theta\end{aligned}$$

where `radius` indicates the swirl extent in pixels, `rotation` adds a rotation angle, and `strength` is a parameter for the amount of swirl. The transformation of `radius` into `r` is to ensure that the transformation decays to $\approx 1/1000^{\text{th}}$ within the specified radius.



```
import matplotlib.pyplot as plt

from skimage import data
from skimage.transform import swirl

image = data.checkerboard()
swirled = swirl(image, rotation=0, strength=10, radius=120)

fig, (ax0, ax1) = plt.subplots(nrows=1, ncols=2, figsize=(8, 3),
                             sharex=True, sharey=True)

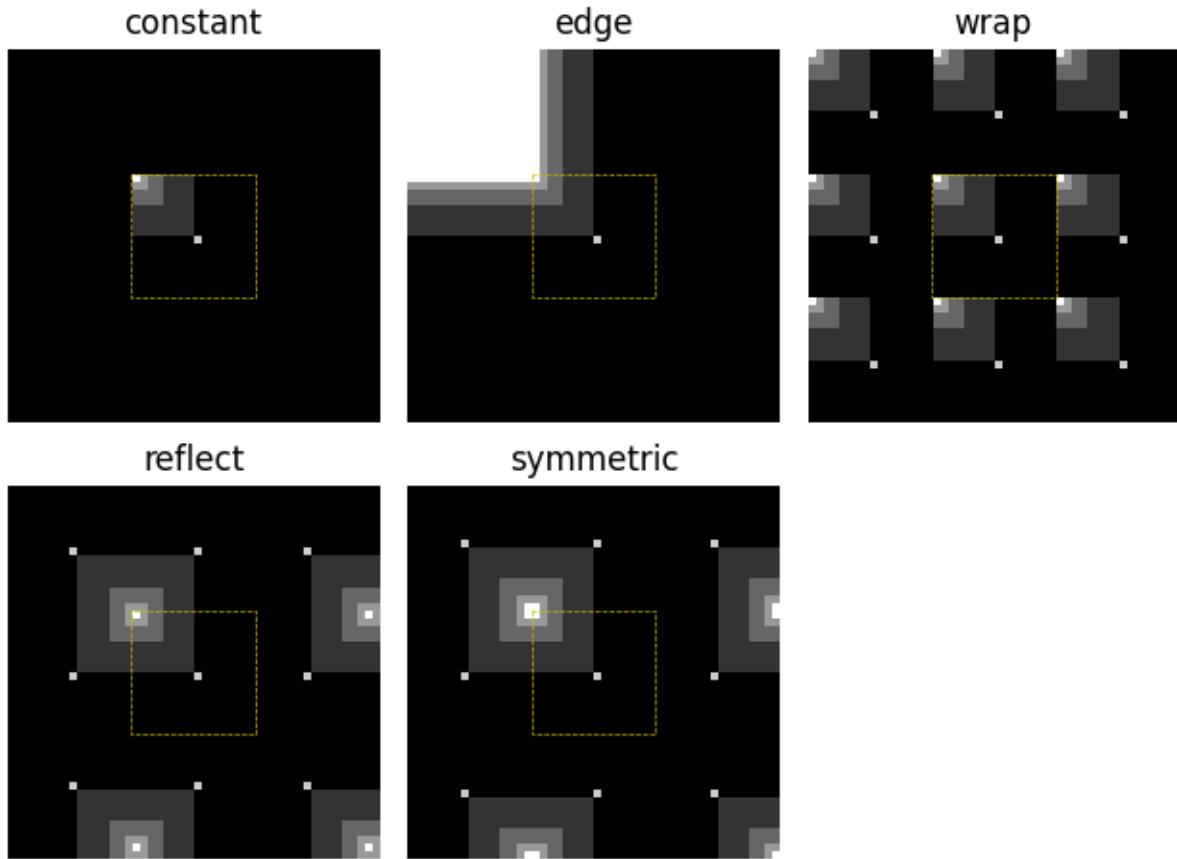
ax0.imshow(image, cmap=plt.cm.gray)
ax0.axis('off')
ax1.imshow(swirled, cmap=plt.cm.gray)
ax1.axis('off')

plt.show()
```

Total running time of the script: (0 minutes 0.061 seconds)

Interpolation: Edge Modes

This example illustrates the different edge modes available during interpolation in routines such as `skimage.transform.rescale()` and `skimage.transform.resize()`.



```

import numpy as np
import matplotlib.pyplot as plt

img = np.zeros((16, 16))
img[:8, :8] += 1
img[:4, :4] += 1
img[:2, :2] += 1
img[:1, :1] += 2
img[8, 8] = 4

modes = ['constant', 'edge', 'wrap', 'reflect', 'symmetric']
fig, axes = plt.subplots(2, 3)
ax = axes.flatten()

for n, mode in enumerate(modes):
    img_padded = np.pad(img, pad_width=img.shape[0], mode=mode)
    ax[n].imshow(img_padded, cmap=plt.cm.gray)
    ax[n].plot([15.5, 15.5, 31.5, 31.5, 15.5],
               [15.5, 31.5, 31.5, 15.5, 15.5], 'y--', linewidth=0.5)
    ax[n].set_title(mode)

for a in ax:

```

(continues on next page)

(continued from previous page)

```
a.set_axis_off()
a.set_aspect('equal')

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

Total running time of the script: (0 minutes 0.241 seconds)

Rescale, resize, and downscale

Rescale operation resizes an image by a given scaling factor. The scaling factor can either be a single floating point value, or multiple values - one along each axis.

Resize serves the same purpose, but allows to specify an output image shape instead of a scaling factor.

Note that when down-sampling an image, *resize* and *rescale* should perform Gaussian smoothing to avoid aliasing artifacts. See the *anti_aliasing* and *anti_aliasing_sigma* arguments to these functions.

Downscale serves the purpose of down-sampling an n-dimensional image by integer factors using the local mean on the elements of each block of the size factors given as a parameter to the function.



```
import matplotlib.pyplot as plt

from skimage import data, color
```

(continues on next page)

(continued from previous page)

```
from skimage.transform import rescale, resize, downscale_local_mean

image = color.rgb2gray(data.astronaut())

image_rescaled = rescale(image, 0.25, anti_aliasing=False)
image_resized = resize(image, (image.shape[0] // 4, image.shape[1] // 4),
                      anti_aliasing=True)
image_downscaled = downscale_local_mean(image, (4, 3))

fig, axes = plt.subplots(nrows=2, ncols=2)

ax = axes.ravel()

ax[0].imshow(image, cmap='gray')
ax[0].set_title("Original image")

ax[1].imshow(image_rescaled, cmap='gray')
ax[1].set_title("Rescaled image (aliasing)")

ax[2].imshow(image_resized, cmap='gray')
ax[2].set_title("Resized image (no aliasing)")

ax[3].imshow(image_downscaled, cmap='gray')
ax[3].set_title("Downscaled image (no aliasing)")

ax[0].set_xlim(0, 512)
ax[0].set_ylim(512, 0)
plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.435 seconds)

Build image pyramids

The `pyramid_gaussian` function takes an image and yields successive images shrunk by a constant scale factor. Image pyramids are often used, e.g., to implement algorithms for denoising, texture discrimination, and scale-invariant detection.

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

from skimage import data
from skimage.transform import pyramid_gaussian

image = data.astronaut()
rows, cols, dim = image.shape
pyramid = tuple(pyramid_gaussian(image, downscale=2, channel_axis=-1))
```

Generate a composite image for visualization

For visualization, we generate a composite image with the same number of rows as the source image but with `cols + pyramid[1].shape[1]` columns. We then have space to stack all of the downsampled images to the right of the original.

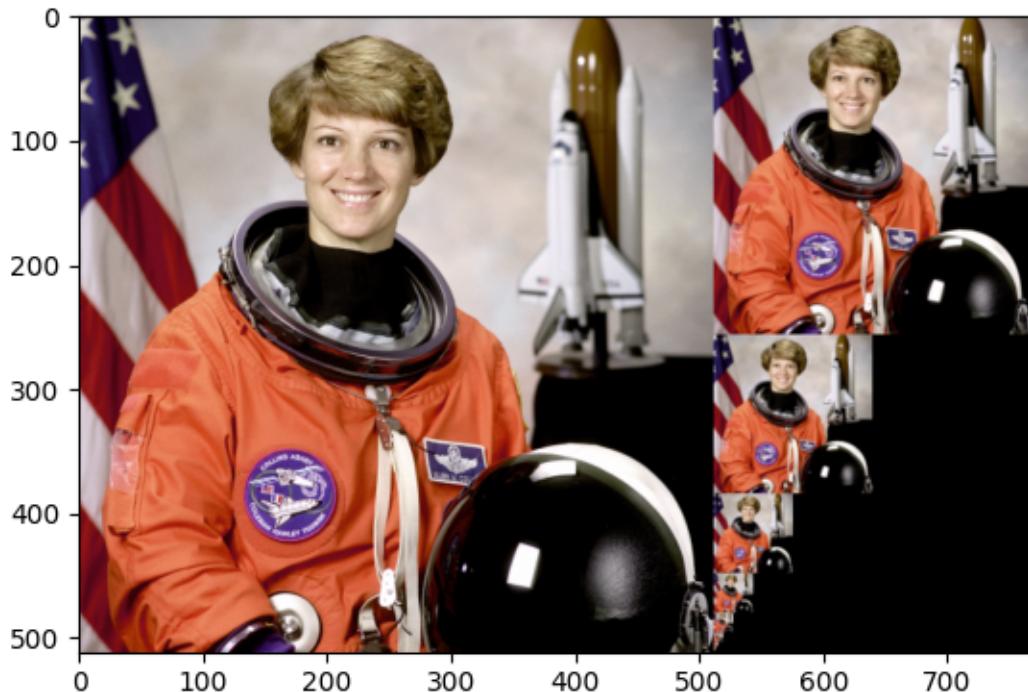
Note: The sum of the number of rows in all downsampled images in the pyramid may sometimes exceed the original image size in cases when `image.shape[0]` is not a power of two. We expand the number of rows in the composite slightly as necessary to account for this. Expansion beyond the number of rows in the original will also be necessary to cover cases where `downscale < 2`.

```
# determine the total number of rows and columns for the composite
composite_rows = max(rows, sum(p.shape[0] for p in pyramid[1:]))
composite_cols = cols + pyramid[1].shape[1]
composite_image = np.zeros((composite_rows, composite_cols, 3),
                           dtype=np.double)

# store the original to the left
composite_image[:rows, :cols, :] = pyramid[0]

# stack all downsampled images in a column to the right of the original
i_row = 0
for p in pyramid[1:]:
    n_rows, n_cols = p.shape[:2]
    composite_image[i_row:i_row + n_rows, cols:cols + n_cols] = p
    i_row += n_rows

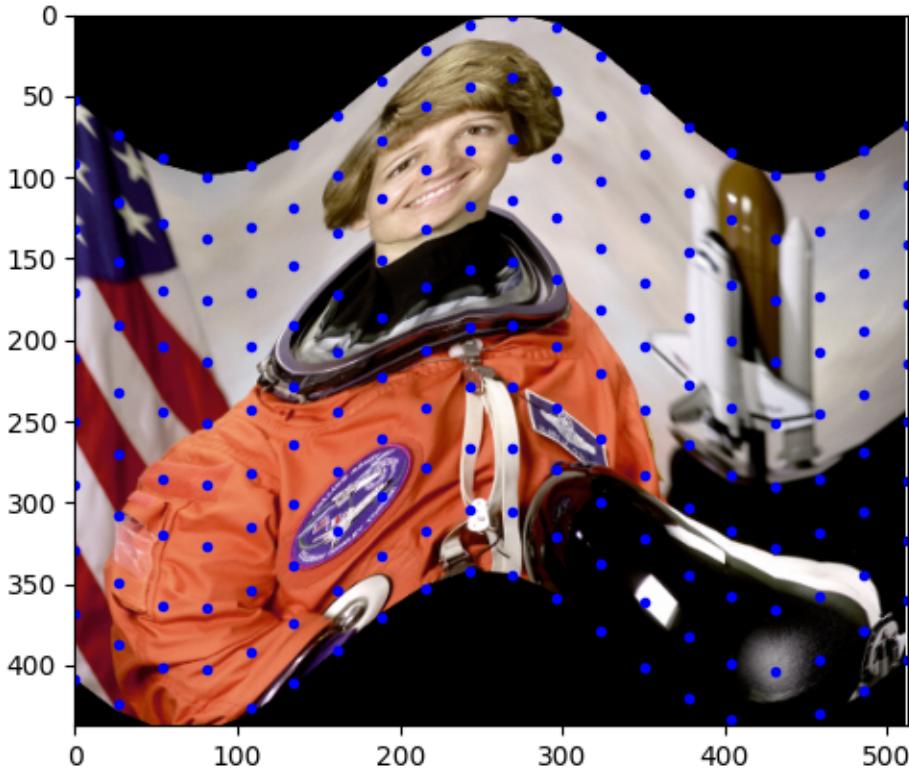
fig, ax = plt.subplots()
ax.imshow(composite_image)
plt.show()
```



Total running time of the script: (0 minutes 0.213 seconds)

Piecewise Affine Transformation

This example shows how to use the Piecewise Affine Transformation.



```

import numpy as np
import matplotlib.pyplot as plt
from skimage.transform import PiecewiseAffineTransform, warp
from skimage import data

image = data.astronaut()
rows, cols = image.shape[0], image.shape[1]

src_cols = np.linspace(0, cols, 20)
src_rows = np.linspace(0, rows, 10)
src_rows, src_cols = np.meshgrid(src_rows, src_cols)
src = np.dstack([src_cols.flat, src_rows.flat])[0]

# add sinusoidal oscillation to row coordinates
dst_rows = src[:, 1] - np.sin(np.linspace(0, 3 * np.pi, src.shape[0])) * 50
dst_cols = src[:, 0]
dst_rows *= 1.5
dst_rows -= 1.5 * 50
dst = np.vstack([dst_cols, dst_rows]).T

tform = PiecewiseAffineTransform()

```

(continues on next page)

(continued from previous page)

```
tform.estimate(src, dst)

out_rows = image.shape[0] - 1.5 * 50
out_cols = cols
out = warp(image, tform, output_shape=(out_rows, out_cols))

fig, ax = plt.subplots()
ax.imshow(out)
ax.plot(tform.inverse(src)[:, 0], tform.inverse(src)[:, 1], '.b')
ax.axis((0, out_cols, out_rows, 0))
plt.show()
```

Total running time of the script: (0 minutes 0.974 seconds)

Using geometric transformations

In this example, we will see how to use geometric transformations in the context of image processing.

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

from skimage import data
from skimage import transform
```

Basics

Several different geometric transformation types are supported: similarity, affine, projective and polynomial. For a tutorial on the available types of transformations, see *Types of homographies*.

Geometric transformations can either be created using the explicit parameters (e.g. scale, shear, rotation and translation) or the transformation matrix.

First we create a transformation using explicit parameters:

```
tform = transform.SimilarityTransform(scale=1, rotation=math.pi/2,
                                      translation=(0, 1))
print(tform.params)
```

```
[[ 6.123234e-17 -1.000000e+00  0.000000e+00]
 [ 1.000000e+00  6.123234e-17  1.000000e+00]
 [ 0.000000e+00  0.000000e+00  1.000000e+00]]
```

Alternatively you can define a transformation by the transformation matrix itself:

```
matrix = tform.params.copy()
matrix[1, 2] = 2
tform2 = transform.SimilarityTransform(matrix)
```

These transformation objects can then be used to apply forward and inverse coordinate transformations between the source and destination coordinate systems:

```
coord = [1, 0]
print(tform2(coord))
print(tform2.inverse(tform(coord)))
```

```
[[6.123234e-17 3.000000e+00]]
[[ 0.000000e+00 -6.123234e-17]]
```

Image warping

Geometric transformations can also be used to warp images:

```
text = data.text()

tform = transform.SimilarityTransform(scale=1, rotation=math.pi/4,
                                      translation=(text.shape[0]/2, -100))

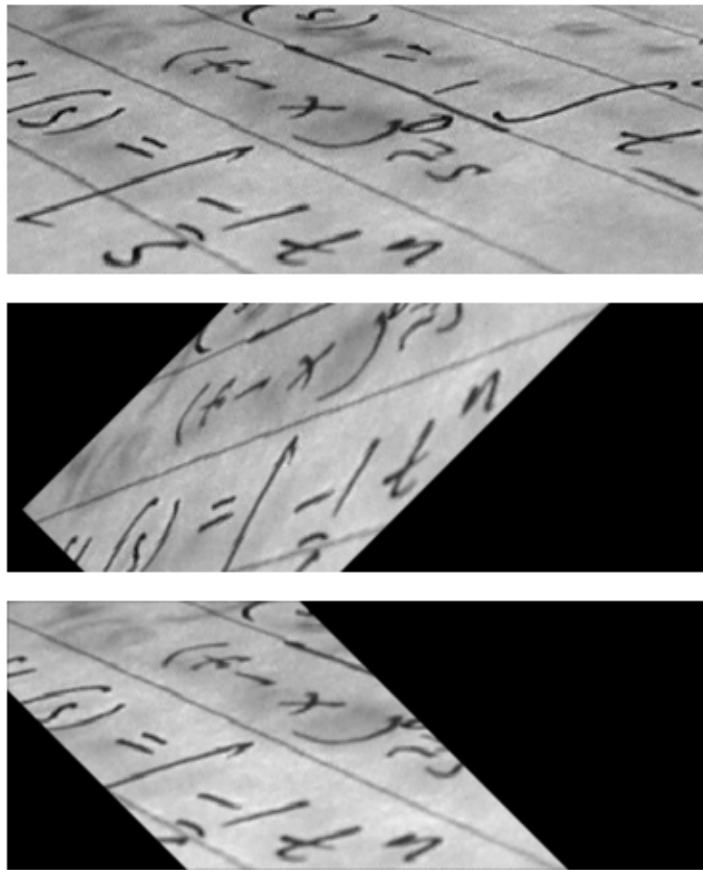
rotated = transform.warp(text, tform)
back_rotated = transform.warp(rotated, tform.inverse)

fig, ax = plt.subplots(nrows=3)

ax[0].imshow(text, cmap=plt.cm.gray)
ax[1].imshow(rotated, cmap=plt.cm.gray)
ax[2].imshow(back_rotated, cmap=plt.cm.gray)

for a in ax:
    a.axis('off')

plt.tight_layout()
```



Parameter estimation

In addition to the basic functionality mentioned above you can also estimate the parameters of a geometric transformation using the least-squares method.

This can amongst other things be used for image registration or rectification, where you have a set of control points or homologous/corresponding points in two images.

Let's assume we want to recognize letters on a photograph which was not taken from the front but at a certain angle. In the simplest case of a plane paper surface the letters are projectively distorted. Simple matching algorithms would not be able to match such symbols. One solution to this problem would be to warp the image so that the distortion is removed and then apply a matching algorithm:

```
text = data.text()

src = np.array([[0, 0], [0, 50], [300, 50], [300, 0]])
dst = np.array([[155, 15], [65, 40], [260, 130], [360, 95]])

tform3 = transform.ProjectiveTransform()
tform3.estimate(src, dst)
warped = transform.warp(text, tform3, output_shape=(50, 300))

fig, ax = plt.subplots(nrows=2, figsize=(8, 3))
```

(continues on next page)

(continued from previous page)

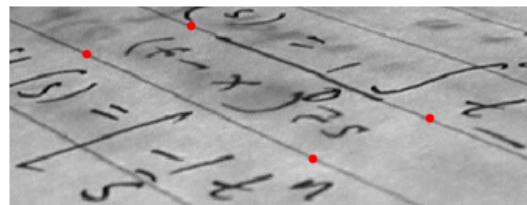
```

ax[0].imshow(text, cmap=plt.cm.gray)
ax[0].plot(dst[:, 0], dst[:, 1], '.r')
ax[1].imshow(warped, cmap=plt.cm.gray)

for a in ax:
    a.axis('off')

plt.tight_layout()
plt.show()

```



The above estimation relies on accurate knowledge of the location of points and an accurate selection of their correspondence. If point locations have an uncertainty associated with them, then weighting can be provided so that the resulting transform prioritises an accurate fit to those points with the highest weighting. An alternative approach called the [RANSAC algorithm](#) is useful when the correspondence points are not perfectly accurate. See the *Robust matching using RANSAC* tutorial for an in-depth description of how to use this approach in scikit-image.

Total running time of the script: (0 minutes 0.251 seconds)

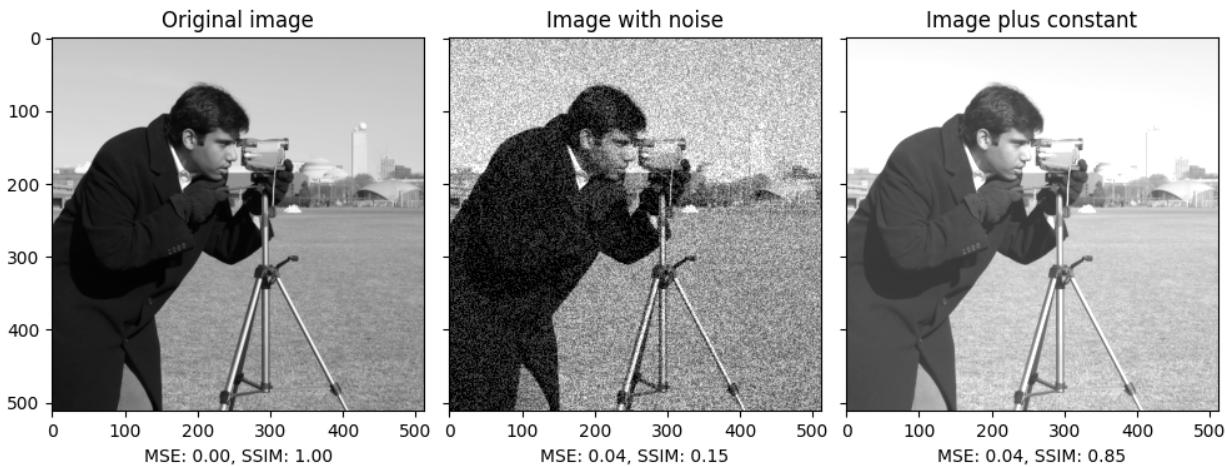
Structural similarity index

When comparing images, the mean squared error (MSE)—while simple to implement—is not highly indicative of perceived similarity. Structural similarity aims to address this shortcoming by taking texture into account^{1,2}.

The example shows two modifications of the input image, each with the same MSE, but with very different mean structural similarity indices.

¹ Zhou Wang; Bovik, A.C.; "Mean squared error: Love it or leave it? A new look at Signal Fidelity Measures," Signal Processing Magazine, IEEE, vol. 26, no. 1, pp. 98-117, Jan. 2009.

² Z. Wang, A. C. Bovik, H. R. Sheikh and E. P. Simoncelli, "Image quality assessment: From error visibility to structural similarity," IEEE Transactions on Image Processing, vol. 13, no. 4, pp. 600-612, Apr. 2004.



```

import numpy as np
import matplotlib.pyplot as plt

from skimage import data, img_as_float
from skimage.metrics import structural_similarity as ssim
from skimage.metrics import mean_squared_error

img = img_as_float(data.camera())
rows, cols = img.shape

noise = np.ones_like(img) * 0.2 * (img.max() - img.min())
rng = np.random.default_rng()
noise[rng.random(size=noise.shape) > 0.5] *= -1

img_noise = img + noise
img_const = img + abs(noise)

fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(10, 4),
                        sharex=True, sharey=True)
ax = axes.ravel()

mse_none = mean_squared_error(img, img)
ssim_none = ssim(img, img, data_range=img.max() - img.min())

mse_noise = mean_squared_error(img, img_noise)
ssim_noise = ssim(img, img_noise,
                  data_range=img_noise.max() - img_noise.min())

mse_const = mean_squared_error(img, img_const)
ssim_const = ssim(img, img_const,
                  data_range=img_const.max() - img_const.min())

ax[0].imshow(img, cmap=plt.cm.gray, vmin=0, vmax=1)
ax[0].set_xlabel(f'MSE: {mse_none:.2f}, SSIM: {ssim_none:.2f}')
ax[0].set_title('Original image')

```

(continues on next page)

(continued from previous page)

```

ax[1].imshow(img_noise, cmap=plt.cm.gray, vmin=0, vmax=1)
ax[1].set_xlabel(f'MSE: {mse_noise:.2f}, SSIM: {ssim_noise:.2f}')
ax[1].set_title('Image with noise')

ax[2].imshow(img_const, cmap=plt.cm.gray, vmin=0, vmax=1)
ax[2].set_xlabel(f'MSE: {mse_const:.2f}, SSIM: {ssim_const:.2f}')
ax[2].set_title('Image plus constant')

plt.tight_layout()
plt.show()

```

Total running time of the script: (0 minutes 0.507 seconds)

Types of homographies

Homographies are transformations of a Euclidean space that preserve the alignment of points. Specific cases of homographies correspond to the conservation of more properties, such as parallelism (affine transformation), shape (similar transformation) or distances (Euclidean transformation).

Homographies on a 2D Euclidean space (i.e., for 2D grayscale or multichannel images) are defined by a 3x3 matrix. All types of homographies can be defined by passing either the transformation matrix, or the parameters of the simpler transformations (rotation, scaling, ...) which compose the full transformation.

The different types of homographies available in scikit-image are shown here, by increasing order of complexity (i.e. by reducing the number of constraints). While we focus here on the mathematical properties of transformations, tutorial *Using geometric transformations* explains how to use such transformations for various tasks such as image warping or parameter estimation.

```

import numpy as np
import matplotlib.pyplot as plt

from skimage import data
from skimage import transform
from skimage import img_as_float

```

Euclidean (rigid) transformation

A Euclidean transformation, also called rigid transformation, preserves the Euclidean distance between pairs of points. It can be described as a rotation about the origin followed by a translation.

```

tform = transform.EuclideanTransform(
    rotation=np.pi / 12.,
    translation = (100, -20)
)
print(tform.params)

```

```

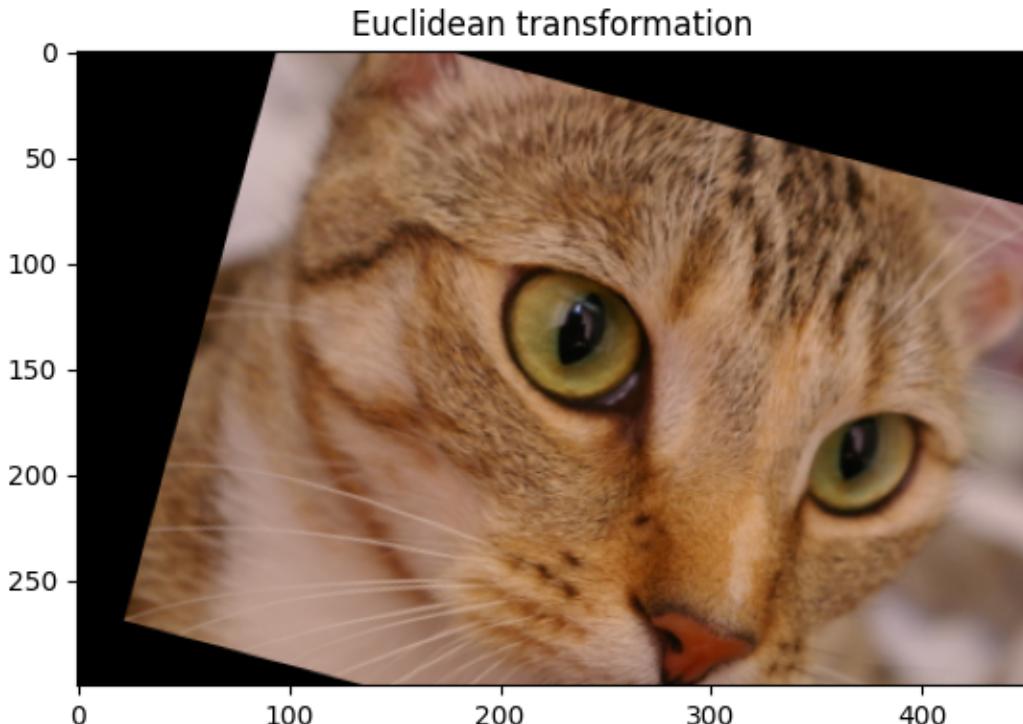
[[ 0.96592583 -0.25881905 100.      ]
 [ 0.25881905  0.96592583 -20.      ]
 [ 0.          0.          1.        ]]

```

Now let's apply this transformation to an image. Because we are trying to reconstruct the *image* after transformation, it is not useful to see where a *coordinate* from the input image ends up in the output, which is what the transform gives

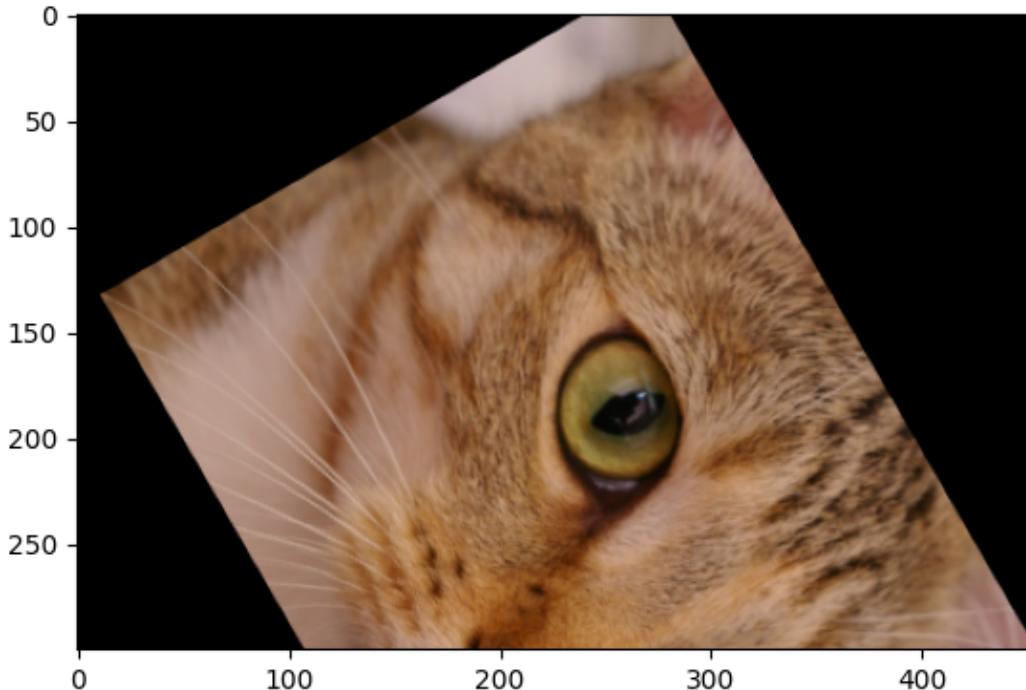
us. Instead, for every pixel (coordinate) in the output image, we want to figure out where in the input image it comes from. Therefore, we need to use the inverse of `tform`, rather than `tform` directly.

```
img = img_as_float(data.chelsea())
tf_img = transform.warp(img, tform.inverse)
fig, ax = plt.subplots()
ax.imshow(tf_img)
_ = ax.set_title('Euclidean transformation')
```



For a rotation around the center of the image, one can compose a translation to change the origin, a rotation, and finally the inverse of the first translation.

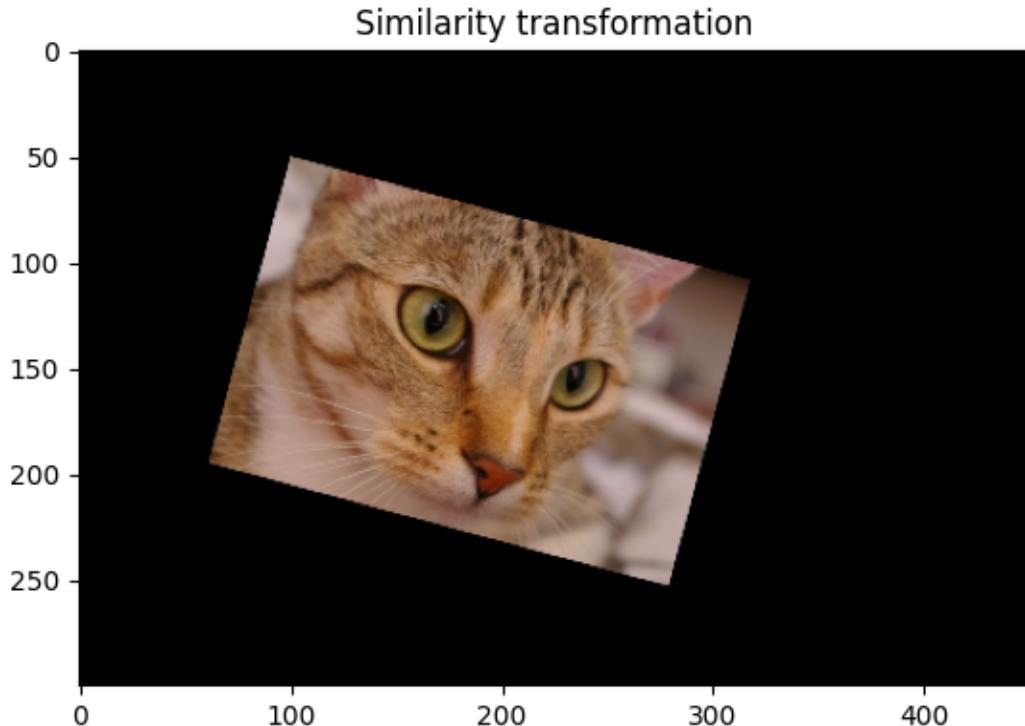
```
rotation = transform.EuclideanTransform(rotation=np.pi/3)
shift = transform.EuclideanTransform(translation=-np.array(img.shape[:2]) / 2)
# Compose transforms by multiplying their matrices
matrix = np.linalg.inv(shift.params) @ rotation.params @ shift.params
tform = transform.EuclideanTransform(matrix)
tf_img = transform.warp(img, tform.inverse)
fig, ax = plt.subplots()
_ = ax.imshow(tf_img)
```



Similarity transformation

A [similarity transformation](#) preserves the shape of objects. It combines scaling, translation and rotation.

```
tform = transform.SimilarityTransform(  
    scale=0.5,  
    rotation=np.pi/12,  
    translation=(100, 50))  
print(tform.params)  
tf_img = transform.warp(img, tform.inverse)  
fig, ax = plt.subplots()  
ax.imshow(tf_img)  
_ = ax.set_title('Similarity transformation')
```

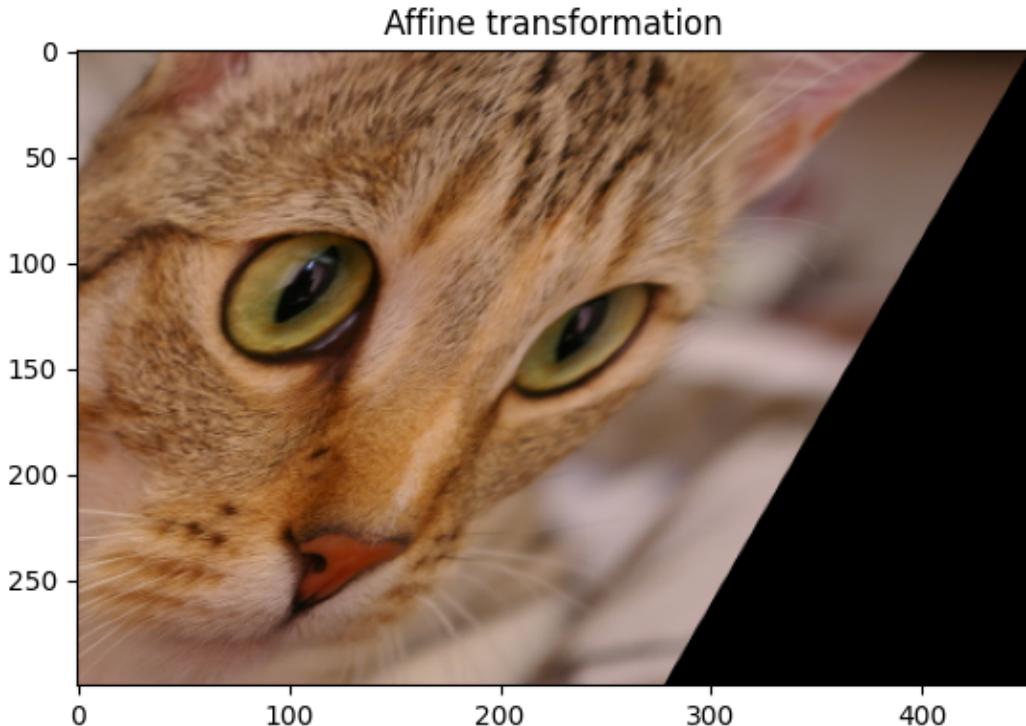


```
[[ 0.48296291 -0.12940952 100.      ]
 [ 0.12940952  0.48296291  50.      ]
 [ 0.          0.          1.        ]]
```

Affine transformation

An affine transformation preserves lines (hence the alignment of objects), as well as parallelism between lines. It can be decomposed into a similarity transform and a [shear transformation](#).

```
tform = transform.AffineTransform(
    shear=np.pi/6,
)
print(tform.params)
tf_img = transform.warp(img, tform.inverse)
fig, ax = plt.subplots()
ax.imshow(tf_img)
_ = ax.set_title('Affine transformation')
```



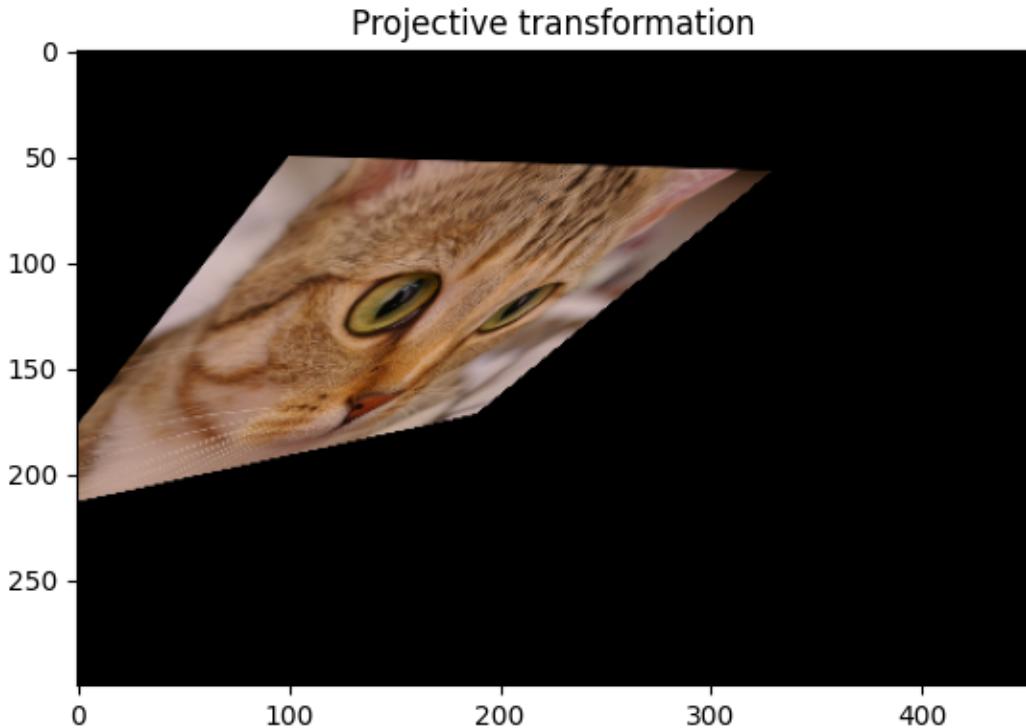
```
[[ 1.        -0.57735027  0.        ],
 [ 0.         1.          0.        ],
 [ 0.         0.          1.        ]]
```

Projective transformation (homographies)

A [homography](#), also called projective transformation, preserves lines but not necessarily parallelism.

```
matrix = np.array([[1, -0.5, 100],
                  [0.1, 0.9, 50],
                  [0.0015, 0.0015, 1]])
tform = transform.ProjectiveTransform(matrix=matrix)
tf_img = transform.warp(img, tform.inverse)
fig, ax = plt.subplots()
ax.imshow(tf_img)
ax.set_title('Projective transformation')

plt.show()
```



See also

- [Using geometric transformations](#) for composing transformations or estimating their parameters
- [Rescale, resize, and downscale](#) for simple rescaling and resizing operations
- [skimage.transform.rotate\(\)](#) for rotating an image around its center

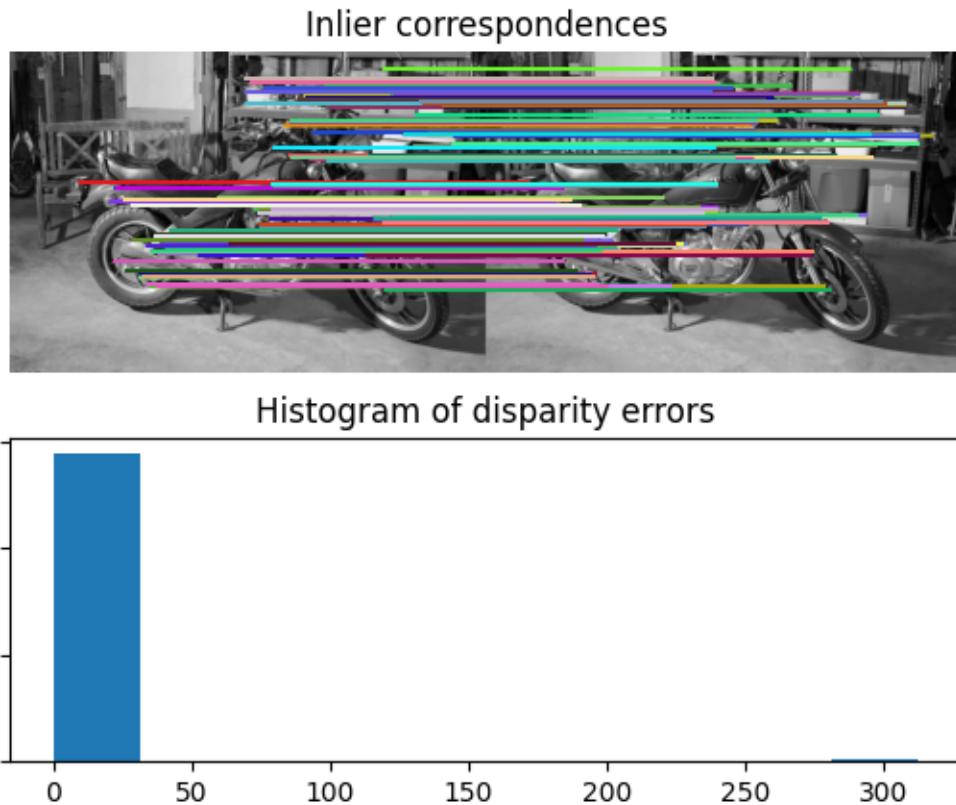
Total running time of the script: (0 minutes 0.676 seconds)

Fundamental matrix estimation

This example demonstrates how to robustly estimate *epipolar geometry* (the geometry of stereo vision) between two views using sparse ORB feature correspondences.

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.

Uncalibrated means that the intrinsic calibration (focal lengths, pixel skew, principal point) of the two cameras is not known. The fundamental matrix thus enables projective 3D reconstruction of the captured scene. If the calibration is known, estimating the essential matrix enables metric 3D reconstruction of the captured scene.



```
Number of matches: 223
/github/workspace/build/scikit-image/doc/examples/transform/plot_fundamental_matrix.
→ by:54: FutureWarning:
`random_state` is a deprecated argument name for `ransac`. It will be removed in version →
0.23. Please use `rng` instead.

Number of inliers: 162
```

```
import numpy as np
from skimage import data
from skimage.color import rgb2gray
from skimage.feature import match_descriptors, ORB, plot_matches
from skimage.measure import ransac
from skimage.transform import FundamentalMatrixTransform
import matplotlib.pyplot as plt

img_left, img_right, groundtruth_disp = data.stereo_motorcycle()
```

(continues on next page)

(continued from previous page)

```
img_left, img_right = map(rgb2gray, (img_left, img_right))

# Find sparse feature correspondences between left and right image.

descriptor_extractor = ORB()

descriptor_extractor.detect_and_extract(img_left)
keypoints_left = descriptor_extractor.keypoints
descriptors_left = descriptor_extractor.descriptors

descriptor_extractor.detect_and_extract(img_right)
keypoints_right = descriptor_extractor.keypoints
descriptors_right = descriptor_extractor.descriptors

matches = match_descriptors(descriptors_left, descriptors_right,
                             cross_check=True)

print(f'Number of matches: {matches.shape[0]}')

# Estimate the epipolar geometry between the left and right image.
random_seed = 9
rng = np.random.default_rng(random_seed)

model, inliers = ransac((keypoints_left[matches[:, 0]],
                         keypoints_right[matches[:, 1]]),
                         FundamentalMatrixTransform, min_samples=8,
                         residual_threshold=1, max_trials=5000,
                         random_state=rng)

inlier_keypoints_left = keypoints_left[matches[inliers, 0]]
inlier_keypoints_right = keypoints_right[matches[inliers, 1]]

print(f'Number of inliers: {inliers.sum()}')

# Compare estimated sparse disparities to the dense ground-truth disparities.

disp = inlier_keypoints_left[:, 1] - inlier_keypoints_right[:, 1]
disp_coords = np.round(inlier_keypoints_left).astype(np.int64)
disp_idxs = np.ravel_multi_index(disp_coords.T, groundtruth_disp.shape)
disp_error = np.abs(groundtruth_disp.ravel()[disp_idxs] - disp)
disp_error = disp_error[np.isfinite(disp_error)]

# Visualize the results.

fig, ax = plt.subplots(nrows=2, ncols=1)

plt.gray()

plot_matches(ax[0], img_left, img_right, keypoints_left, keypoints_right,
             matches[inliers], only_matches=True)
ax[0].axis("off")
ax[0].set_title("Inlier correspondences")
```

(continues on next page)

(continued from previous page)

```
ax[1].hist(disp_error)
ax[1].set_title("Histogram of disparity errors")

plt.show()
```

Total running time of the script: (0 minutes 2.446 seconds)

Robust line model estimation using RANSAC

In this example we see how to robustly fit a line model to faulty data using the RANSAC (random sample consensus) algorithm.

Firstly the data are generated by adding a gaussian noise to a linear function. Then, the outlier points are added to the data set.

RANSAC iteratively estimates the parameters from the data set. At each iteration the following steps are performed:

1. Select `min_samples` random samples from the original data and check whether the set of data is valid (see `is_data_valid` option).
2. Estimate a model on the random subset (`model_cls.estimate(*data[random_subset])`) and check whether the estimated model is valid (see `is_model_valid` option).
3. Classify all the data points as either inliers or outliers by calculating the residuals using the estimated model (`model_cls.residuals(*data)`) - all data samples with residuals smaller than the `residual_threshold` are considered as inliers.
4. If the number of the inlier samples is greater than ever before, save the estimated model as the best model. In case the current estimated model has the same number of inliers, it is considered as the best model only if the sum of residuals is lower.

These steps are performed either a maximum number of times or until one of the special stop criteria are met. The final model is estimated using all the inlier samples of the previously determined best model.

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

from skimage.measure import LineModelND, ransac

rng = np.random.default_rng()

# generate coordinates of line
x = np.arange(-200, 200)
y = 0.2 * x + 20
data = np.column_stack([x, y])

# add gaussian noise to coordinates
noise = rng.normal(size=data.shape)
data += 0.5 * noise
data[::2] += 5 * noise[::2]
data[::4] += 20 * noise[::4]

# add faulty data
faulty = np.array(30 * [(180., -100)])
faulty += 10 * rng.normal(size=faulty.shape)
```

(continues on next page)

(continued from previous page)

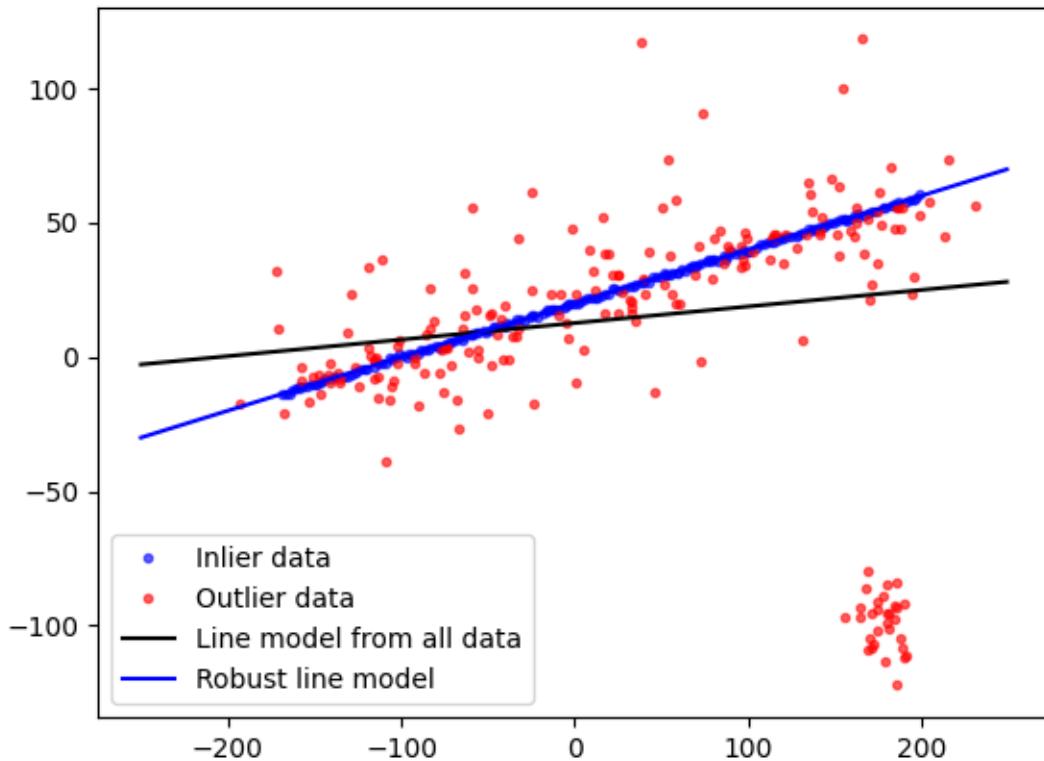
```
data[:faulty.shape[0]] = faulty

# fit line using all data
model = LineModelND()
model.estimate(data)

# robustly fit line only using inlier data with RANSAC algorithm
model_robust, inliers = ransac(data, LineModelND, min_samples=2,
                               residual_threshold=1, max_trials=1000)
outliers = (inliers == False)

# generate coordinates of estimated models
line_x = np.arange(-250, 250)
line_y = model.predict_y(line_x)
line_y_robust = model_robust.predict_y(line_x)

fig, ax = plt.subplots()
ax.plot(data[inliers, 0], data[inliers, 1], '.b', alpha=0.6,
        label='Inlier data')
ax.plot(data[outliers, 0], data[outliers, 1], '.r', alpha=0.6,
        label='Outlier data')
ax.plot(line_x, line_y, '-k', label='Line model from all data')
ax.plot(line_x, line_y_robust, '-b', label='Robust line model')
ax.legend(loc='lower left')
plt.show()
```



Now, we generalize this example to 3D points.

```

import numpy as np
from matplotlib import pyplot as plt
from skimage.measure import LineModelND, ransac

# generate coordinates of line
point = np.array([0, 0, 0], dtype='float')
direction = np.array([1, 1, 1], dtype='float') / np.sqrt(3)
xyz = point + 10 * np.arange(-100, 100)[..., np.newaxis] * direction

# add gaussian noise to coordinates
noise = rng.normal(size=xyz.shape)
xyz += 0.5 * noise
xyz[::2] += 20 * noise[::2]
xyz[::4] += 100 * noise[::4]

# robustly fit line only using inlier data with RANSAC algorithm
model_robust, inliers = ransac(xyz, LineModelND, min_samples=2,
                                residual_threshold=1, max_trials=1000)
outliers = inliers == False

fig = plt.figure()

```

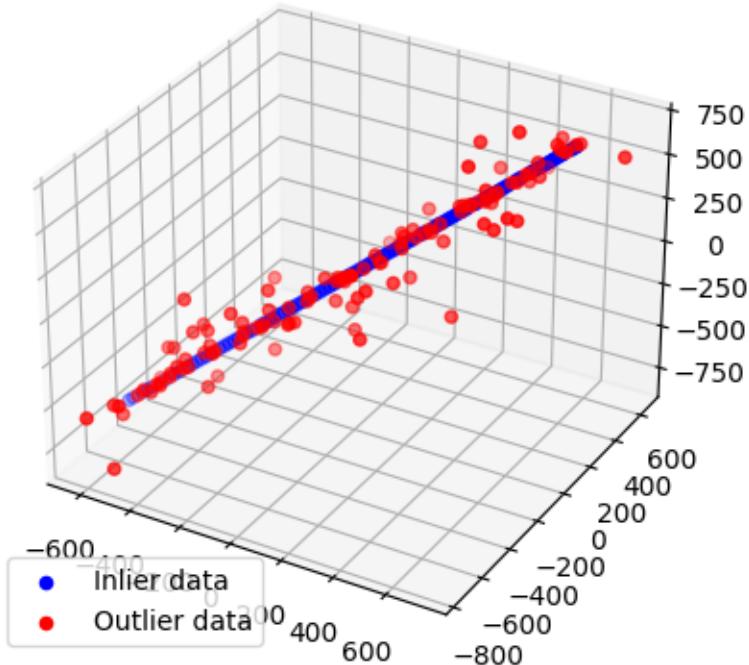
(continues on next page)

(continued from previous page)

```

ax = fig.add_subplot(111, projection='3d')
ax.scatter(xyz[inliers][:, 0], xyz[inliers][:, 1], xyz[inliers][:, 2], c='b',
           marker='o', label='Inlier data')
ax.scatter(xyz[outliers][:, 0], xyz[outliers][:, 1], xyz[outliers][:, 2], c='r',
           marker='o', label='Outlier data')
ax.legend(loc='lower left')
plt.show()

```



Total running time of the script: (0 minutes 0.222 seconds)

Radon transform

In computed tomography, the tomography reconstruction problem is to obtain a tomographic slice image from a set of projections¹. A projection is formed by drawing a set of parallel rays through the 2D object of interest, assigning the integral of the object's contrast along each ray to a single pixel in the projection. A single projection of a 2D object is one dimensional. To enable computed tomography reconstruction of the object, several projections must be acquired, each of them corresponding to a different angle between the rays with respect to the object. A collection of projections at several angles is called a sinogram, which is a linear transform of the original image.

The inverse Radon transform is used in computed tomography to reconstruct a 2D image from the measured projections (the sinogram). A practical, exact implementation of the inverse Radon transform does not exist, but there are several good approximate algorithms available.

¹ AC Kak, M Slaney, “Principles of Computerized Tomographic Imaging”, IEEE Press 1988. <http://www.slaney.org/pct/pct-toc.html>

As the inverse Radon transform reconstructs the object from a set of projections, the (forward) Radon transform can be used to simulate a tomography experiment.

This script performs the Radon transform to simulate a tomography experiment and reconstructs the input image based on the resulting sinogram formed by the simulation. Two methods for performing the inverse Radon transform and reconstructing the original image are compared: The Filtered Back Projection (FBP) and the Simultaneous Algebraic Reconstruction Technique (SART).

For further information on tomographic reconstruction, see:

The forward transform

As our original image, we will use the Shepp-Logan phantom. When calculating the Radon transform, we need to decide how many projection angles we wish to use. As a rule of thumb, the number of projections should be about the same as the number of pixels there are across the object (to see why this is so, consider how many unknown pixel values must be determined in the reconstruction process and compare this to the number of measurements provided by the projections), and we follow that rule here. Below is the original image and its Radon transform, often known as its *sinogram*:

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

from skimage.data import shepp_logan_phantom
from skimage.transform import radon, rescale

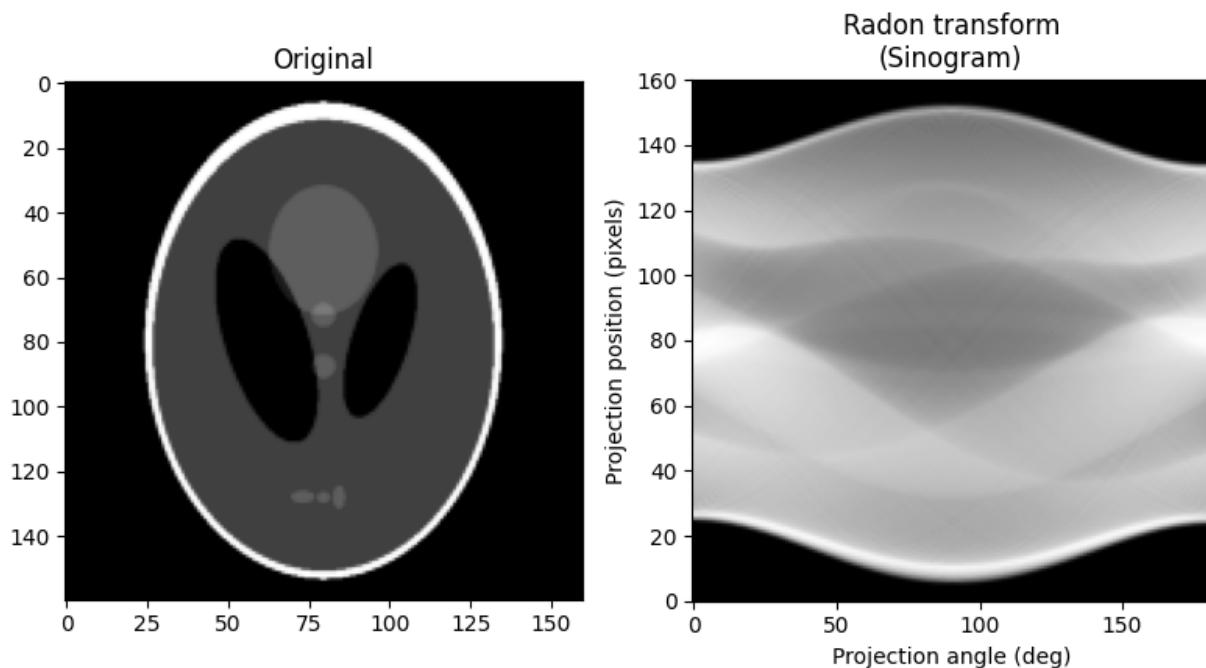
image = shepp_logan_phantom()
image = rescale(image, scale=0.4, mode='reflect', channel_axis=None)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 4.5))

ax1.set_title("Original")
ax1.imshow(image, cmap=plt.cm.Greys_r)

theta = np.linspace(0., 180., max(image.shape), endpoint=False)
sinogram = radon(image, theta=theta)
dx, dy = 0.5 * 180.0 / max(image.shape), 0.5 / sinogram.shape[0]
ax2.set_title("Radon transform\n(Sinogram)")
ax2.set_xlabel("Projection angle (deg)")
ax2.set_ylabel("Projection position (pixels)")
ax2.imshow(sinogram, cmap=plt.cm.Greys_r,
           extent=(-dx, 180.0 + dx, -dy, sinogram.shape[0] + dy),
           aspect='auto')

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



Reconstruction with the Filtered Back Projection (FBP)

The mathematical foundation of the filtered back projection is the Fourier slice theorem². It uses Fourier transform of the projection and interpolation in Fourier space to obtain the 2D Fourier transform of the image, which is then inverted to form the reconstructed image. The filtered back projection is among the fastest methods of performing the inverse Radon transform. The only tunable parameter for the FBP is the filter, which is applied to the Fourier transformed projections. It may be used to suppress high frequency noise in the reconstruction. skimage provides the filters ‘ramp’, ‘shepp-logan’, ‘cosine’, ‘hamming’, and ‘hann’:

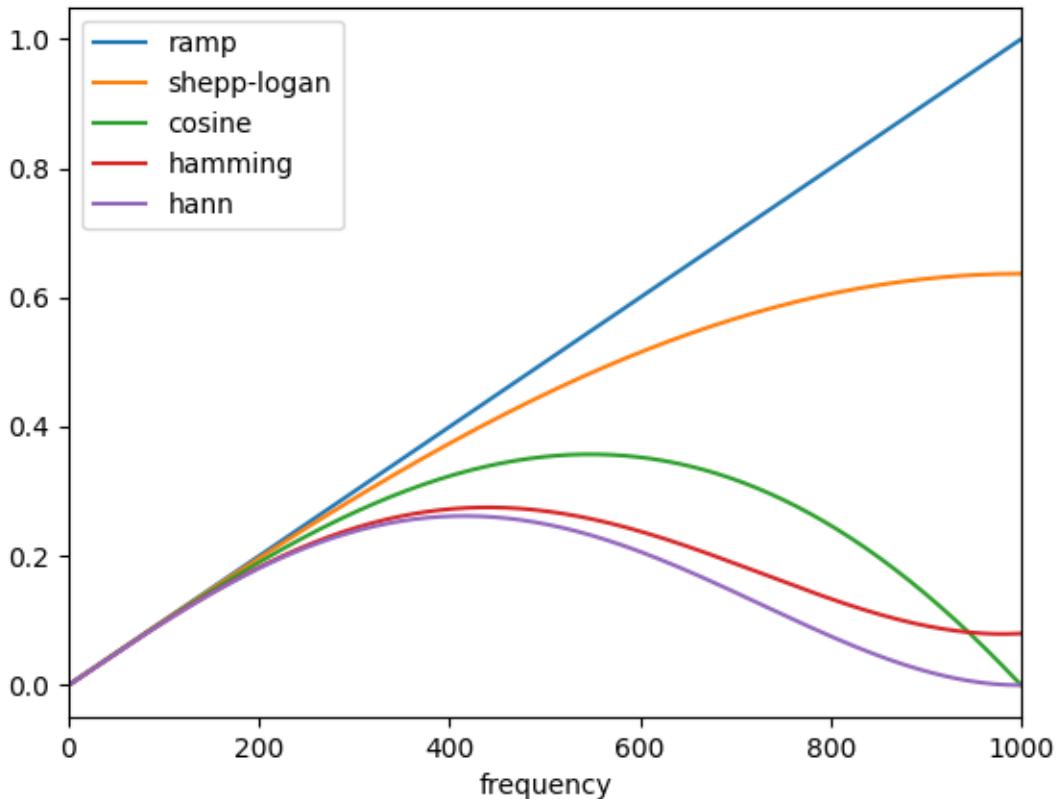
```
import matplotlib.pyplot as plt
from skimage.transform.radon_transform import _get_fourier_filter

filters = ['ramp', 'shepp-logan', 'cosine', 'hamming', 'hann']

for ix, f in enumerate(filters):
    response = _get_fourier_filter(2000, f)
    plt.plot(response, label=f)

plt.xlim([0, 1000])
plt.xlabel('frequency')
plt.legend()
plt.show()
```

² Wikipedia, Radon transform, https://en.wikipedia.org/wiki/Radon_transform#Relationship_with_the_Fourier_transform

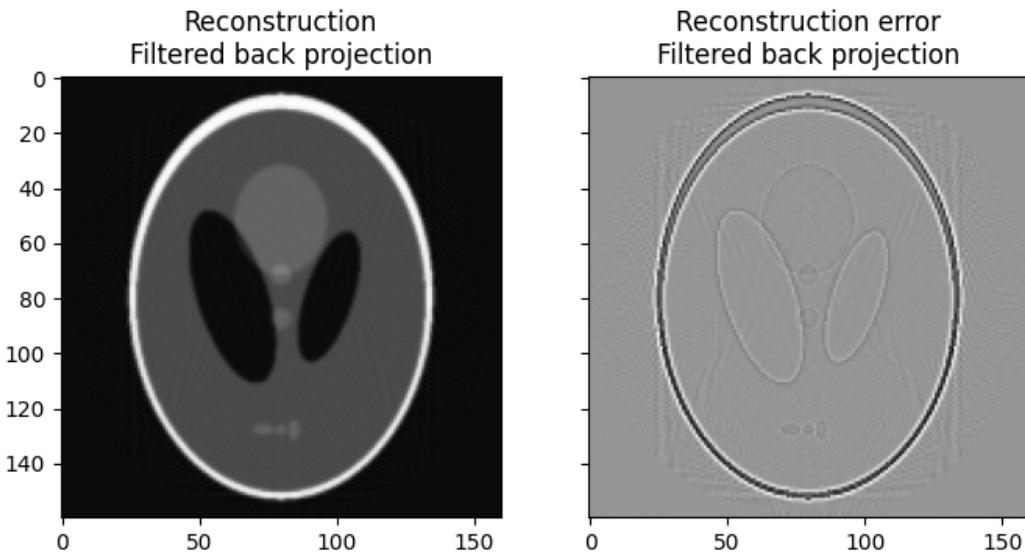


Applying the inverse radon transformation with the ‘ramp’ filter, we get:

```
from skimage.transform import iradon

reconstruction_fbp = iradon(sinogram, theta=theta, filter_name='ramp')
error = reconstruction_fbp - image
print(f'FBP rms reconstruction error: {np.sqrt(np.mean(error**2)):.3g}')

imkwargs = dict(vmin=-0.2, vmax=0.2)
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 4.5),
                             sharex=True, sharey=True)
ax1.set_title("Reconstruction\nFiltered back projection")
ax1.imshow(reconstruction_fbp, cmap=plt.cm.Greys_r)
ax2.set_title("Reconstruction error\nFiltered back projection")
ax2.imshow(reconstruction_fbp - image, cmap=plt.cm.Greys_r, **imkwargs)
plt.show()
```



FBP rms reconstruction error: 0.0283

Reconstruction with the Simultaneous Algebraic Reconstruction Technique

Algebraic reconstruction techniques for tomography are based on a straightforward idea: for a pixelated image the value of a single ray in a particular projection is simply a sum of all the pixels the ray passes through on its way through the object. This is a way of expressing the forward Radon transform. The inverse Radon transform can then be formulated as a (large) set of linear equations. As each ray passes through a small fraction of the pixels in the image, this set of equations is sparse, allowing iterative solvers for sparse linear systems to tackle the system of equations. One iterative method has been particularly popular, namely Kaczmarz' method³, which has the property that the solution will approach a least-squares solution of the equation set.

The combination of the formulation of the reconstruction problem as a set of linear equations and an iterative solver makes algebraic techniques relatively flexible, hence some forms of prior knowledge can be incorporated with relative ease.

skimage provides one of the more popular variations of the algebraic reconstruction techniques: the Simultaneous Algebraic Reconstruction Technique (SART)⁴. It uses Kaczmarz' method as the iterative solver. A good reconstruction is normally obtained in a single iteration, making the method computationally effective. Running one or more extra iterations will normally improve the reconstruction of sharp, high frequency features and reduce the mean squared error at the expense of increased high frequency noise (the user will need to decide on what number of iterations is best suited to the problem at hand). The implementation in skimage allows prior information of the form of a lower and upper threshold on the reconstructed values to be supplied to the reconstruction.

```
from skimage.transform import iradon_sart
```

(continues on next page)

³ S Kaczmarz, "Angenäherte Auflösung von Systemen linearer Gleichungen", Bulletin International de l'Academie Polonaise des Sciences et des Lettres, 35 pp 355–357 (1937)

⁴ AH Andersen, AC Kak, "Simultaneous algebraic reconstruction technique (SART): a superior implementation of the ART algorithm", Ultrasonic Imaging 6 pp 81–94 (1984)

(continued from previous page)

```
reconstruction_sart = iradon_sart(sinogram, theta=theta)
error = reconstruction_sart - image
print(f'SART (1 iteration) rms reconstruction error: '
      f'{np.sqrt(np.mean(error**2)):.3g}')

fig, axes = plt.subplots(2, 2, figsize=(8, 8.5), sharex=True, sharey=True)
ax = axes.ravel()

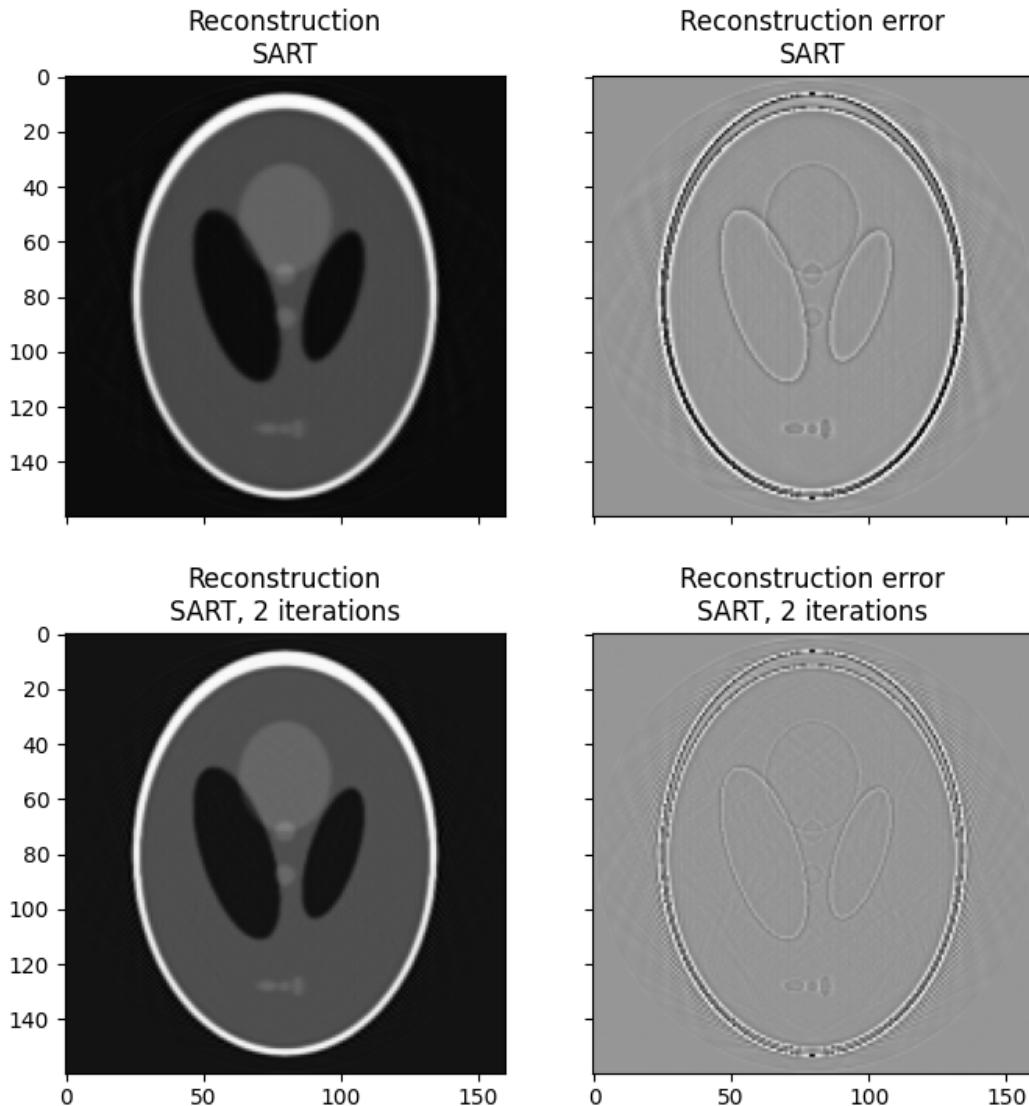
ax[0].set_title("Reconstruction\nSART")
ax[0].imshow(reconstruction_sart, cmap=plt.cm.Greys_r)

ax[1].set_title("Reconstruction error\nSART")
ax[1].imshow(reconstruction_sart - image, cmap=plt.cm.Greys_r, **imkwargs)

# Run a second iteration of SART by supplying the reconstruction
# from the first iteration as an initial estimate
reconstruction_sart2 = iradon_sart(sinogram, theta=theta,
                                    image=reconstruction_sart)
error = reconstruction_sart2 - image
print(f'SART (2 iterations) rms reconstruction error: '
      f'{np.sqrt(np.mean(error**2)):.3g}')

ax[2].set_title("Reconstruction\nSART, 2 iterations")
ax[2].imshow(reconstruction_sart2, cmap=plt.cm.Greys_r)

ax[3].set_title("Reconstruction error\nSART, 2 iterations")
ax[3].imshow(reconstruction_sart2 - image, cmap=plt.cm.Greys_r, **imkwargs)
plt.show()
```



```
SART (1 iteration) rms reconstruction error: 0.0329
SART (2 iterations) rms reconstruction error: 0.0214
```

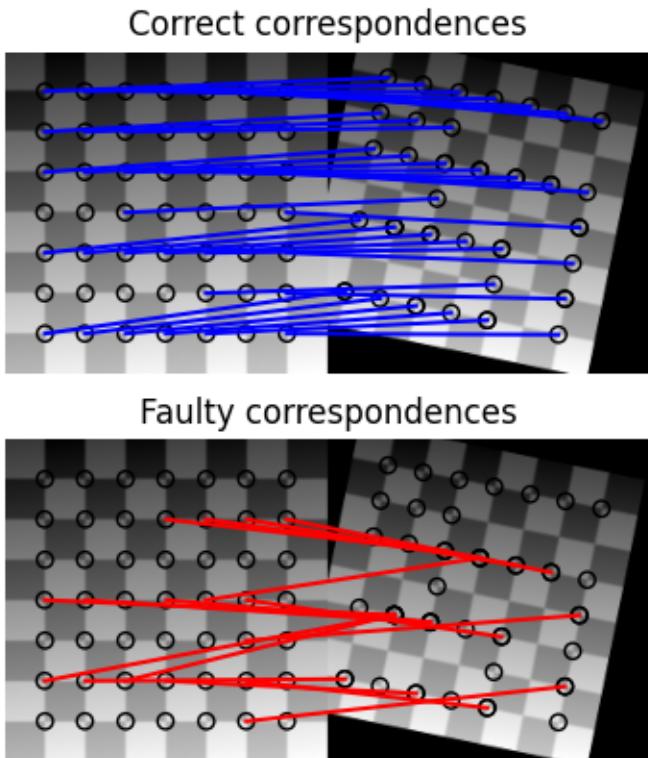
Total running time of the script: (0 minutes 1.639 seconds)

Robust matching using RANSAC

In this simplified example we first generate two synthetic images as if they were taken from different view points.

In the next step we find interest points in both images and find correspondences based on a weighted sum of squared differences of a small neighborhood around them. Note, that this measure is only robust towards linear radiometric and not geometric distortions and is thus only usable with slight view point changes.

After finding the correspondences we end up having a set of source and destination coordinates which can be used to estimate the geometric transformation between both images. However, many of the correspondences are faulty and simply estimating the parameter set with all coordinates is not sufficient. Therefore, the RANSAC algorithm is used on top of the normal model to robustly estimate the parameter set by detecting outliers.



Ground truth:

Scale: (0.9000, 0.9000), Translation: (-10.0000, 20.0000), Rotation: -0.2000

Affine transform:

Scale: (0.9015, 0.8904), Translation: (-9.3136, 14.9768), Rotation: -0.1678

RANSAC:

Scale: (0.8999, 0.9001), Translation: (-10.0005, 19.9744), Rotation: -0.1999

```

import numpy as np
from matplotlib import pyplot as plt

from skimage import data
from skimage.util import img_as_float
from skimage.feature import (corner_harris, corner_subpix, corner_peaks,
                             plot_matches)
from skimage.transform import warp, AffineTransform
from skimage.exposure import rescale_intensity
from skimage.color import rgb2gray
from skimage.measure import ransac

# generate synthetic checkerboard image and add gradient for the later matching
checkerboard = img_as_float(data.checkerboard())
img_orig = np.zeros(list(checkerboard.shape) + [3])
img_orig[..., 0] = checkerboard
gradient_r, gradient_c = (np.mgrid[0:img_orig.shape[0],
                                     0:img_orig.shape[1]] /
                           float(img_orig.shape[0]))
img_orig[..., 1] = gradient_r
img_orig[..., 2] = gradient_c
img_orig = rescale_intensity(img_orig)
img_orig_gray = rgb2gray(img_orig)

# warp synthetic image
tform = AffineTransform(scale=(0.9, 0.9), rotation=0.2, translation=(20, -10))
img_warped = warp(img_orig, tform.inverse, output_shape=(200, 200))
img_warped_gray = rgb2gray(img_warped)

# extract corners using Harris' corner measure
coords_orig = corner_peaks(corner_harris(img_orig_gray), threshold_rel=0.001,
                            min_distance=5)
coords_warped = corner_peaks(corner_harris(img_warped_gray),
                             threshold_rel=0.001, min_distance=5)

# determine sub-pixel corner position
coords_orig_subpix = corner_subpix(img_orig_gray, coords_orig, window_size=9)
coords_warped_subpix = corner_subpix(img_warped_gray, coords_warped,
                                      window_size=9)

def gaussian_weights(window_ext, sigma=1):
    y, x = np.mgrid[-window_ext:window_ext+1, -window_ext:window_ext+1]
    g = np.zeros(y.shape, dtype=np.double)
    g[:] = np.exp(-0.5 * (x**2 / sigma**2 + y**2 / sigma**2))
    g /= 2 * np.pi * sigma * sigma
    return g

def match_corner(coord, window_ext=5):
    r, c = np.round(coord).astype(np.intp)

```

(continues on next page)

(continued from previous page)

```

window_orig = img_orig[r-window_ext:r+window_ext+1,
                      c-window_ext:c+window_ext+1, :]

# weight pixels depending on distance to center pixel
weights = gaussian_weights(window_ext, 3)
weights = np.dstack((weights, weights, weights))

# compute sum of squared differences to all corners in warped image
SSDs = []
for cr, cc in coords_warped:
    window_warped = img_warped[cr-window_ext:cr+window_ext+1,
                                cc-window_ext:cc+window_ext+1, :]
    SSD = np.sum(weights * (window_orig - window_warped)**2)
    SSDs.append(SSD)

# use corner with minimum SSD as correspondence
min_idx = np.argmin(SSDs)
return coords_warped_subpix[min_idx]

# find correspondences using simple weighted sum of squared differences
src = []
dst = []
for coord in coords_orig_subpix:
    src.append(coord)
    dst.append(match_corner(coord))
src = np.array(src)
dst = np.array(dst)

# estimate affine transform model using all coordinates
model = AffineTransform()
model.estimate(src, dst)

# robustly estimate affine transform model with RANSAC
model_robust, inliers = ransac((src, dst), AffineTransform, min_samples=3,
                               residual_threshold=2, max_trials=100)
outliers = (inliers == False)

# compare "true" and estimated transform parameters
print("Ground truth:")
print(f'Scale: ({tform.scale[1]:.4f}, {tform.scale[0]:.4f}), '
      f'Translation: ({tform.translation[1]:.4f}, '
      f'{tform.translation[0]:.4f}), '
      f'Rotation: {-tform.rotation:.4f}')
print("Affine transform:")
print(f'Scale: ({model.scale[0]:.4f}, {model.scale[1]:.4f}), '
      f'Translation: ({model.translation[0]:.4f}, '
      f'{model.translation[1]:.4f}), '
      f'Rotation: {model.rotation:.4f}')
print("RANSAC:")

```

(continues on next page)

(continued from previous page)

```

print(f'Scale: ({model_robust.scale[0]:.4f}, {model_robust.scale[1]:.4f}), '
      f'Translation: ({model_robust.translation[0]:.4f}, '
      f'{model_robust.translation[1]:.4f}), '
      f'Rotation: {model_robust.rotation:.4f}')
```

visualize correspondence

```

fig, ax = plt.subplots(nrows=2, ncols=1)

plt.gray()

inlier_idxs = np.nonzero(inliers)[0]
plot_matches(ax[0], img_orig_gray, img_warped_gray, src, dst,
             np.column_stack((inlier_idxs, inlier_idxs)), matches_color='b')
ax[0].axis('off')
ax[0].set_title('Correct correspondences')

outlier_idxs = np.nonzero(outliers)[0]
plot_matches(ax[1], img_orig_gray, img_warped_gray, src, dst,
             np.column_stack((outlier_idxs, outlier_idxs)), matches_color='r')
ax[1].axis('off')
ax[1].set_title('Faulty correspondences')

plt.show()

```

Total running time of the script: (0 minutes 0.253 seconds)

Image registration

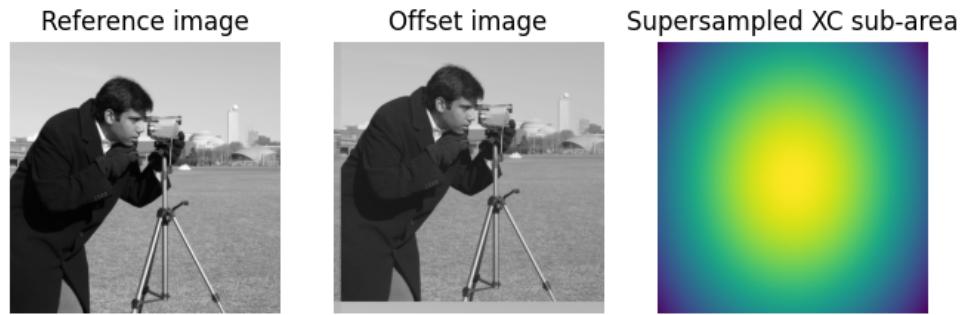
Image Registration

In this example, we use phase cross-correlation to identify the relative shift between two similar-sized images.

The `phase_cross_correlation` function uses cross-correlation in Fourier space, optionally employing an upsampled matrix-multiplication DFT to achieve arbitrary subpixel precision¹.



¹ Manuel Guizar-Sicairos, Samuel T. Thurman, and James R. Fienup, “Efficient subpixel image registration algorithms,” Optics Letters 33, 156-158 (2008). DOI:10.1364/OL.33.000156



•

```
Known offset (y, x): (-22.4, 13.32)
Detected pixel offset (y, x): [ 22. -13.]
Detected subpixel offset (y, x): [ 22.4 -13.32]
```

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

from skimage import data
from skimage.registration import phase_cross_correlation
from skimage.registration._phase_cross_correlation import _upsampled_dft
from scipy.ndimage import fourier_shift

image = data.camera()
shift = (-22.4, 13.32)
# The shift corresponds to the pixel offset relative to the reference image
offset_image = fourier_shift(np.fft.fftn(image), shift)
offset_image = np.fft.ifftn(offset_image)
print(f'Known offset (y, x): {shift}')

# pixel precision first
shift, error, diffphase = phase_cross_correlation(image, offset_image)

fig = plt.figure(figsize=(8, 3))
ax1 = plt.subplot(1, 3, 1)
ax2 = plt.subplot(1, 3, 2, sharex=ax1, sharey=ax1)
ax3 = plt.subplot(1, 3, 3)

ax1.imshow(image, cmap='gray')
ax1.set_axis_off()
ax1.set_title('Reference image')

ax2.imshow(offset_image.real, cmap='gray')
ax2.set_axis_off()
```

(continues on next page)

(continued from previous page)

```
ax2.set_title('Offset image')

# Show the output of a cross-correlation to show what the algorithm is
# doing behind the scenes
image_product = np.fft.fft2(image) * np.fft.fft2(offset_image).conj()
cc_image = np.fft.fftshift(np.fft.ifft2(image_product))
ax3.imshow(cc_image.real)
ax3.set_axis_off()
ax3.set_title("Cross-correlation")

plt.show()

print(f'Detected pixel offset (y, x): {shift}')

# subpixel precision
shift, error, diffphase = phase_cross_correlation(image, offset_image,
                                                   upsample_factor=100)

fig = plt.figure(figsize=(8, 3))
ax1 = plt.subplot(1, 3, 1)
ax2 = plt.subplot(1, 3, 2, sharex=ax1, sharey=ax1)
ax3 = plt.subplot(1, 3, 3)

ax1.imshow(image, cmap='gray')
ax1.set_axis_off()
ax1.set_title('Reference image')

ax2.imshow(offset_image.real, cmap='gray')
ax2.set_axis_off()
ax2.set_title('Offset image')

# Calculate the upsampled DFT, again to show what the algorithm is doing
# behind the scenes. Constants correspond to calculated values in routine.
# See source code for details.
cc_image = _upsampled_dft(image_product, 150, 100, (shift*100)+75).conj()
ax3.imshow(cc_image.real)
ax3.set_axis_off()
ax3.set_title("Supersampled XC sub-area")

plt.show()

print(f'Detected subpixel offset (y, x): {shift}')
```

Total running time of the script: (0 minutes 0.417 seconds)

Masked Normalized Cross-Correlation

In this example, we use the masked normalized cross-correlation to identify the relative shift between two similar images containing invalid data.

In this case, the images cannot simply be masked before computing the cross-correlation, as the masks will influence the computation. The influence of the masks must be removed from the cross-correlation, as is described in¹.

In this example, we register the translation between two images. However, one of the images has about 25% of the pixels which are corrupted.

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

from skimage import data, draw
from skimage.registration import phase_cross_correlation
from scipy import ndimage as ndi
```

Define areas of the image which are invalid. Probability of an invalid pixel is 25%. This could be due to a faulty detector, or edges that are not affected by translation (e.g. moving object in a window). See reference paper for more examples

```
image = data.camera()
shift = (-22, 13)

rng = np.random.default_rng()
corrupted_pixels = rng.choice([False, True], size=image.shape, p=[0.25, 0.75])

# The shift corresponds to the pixel offset relative to the reference image
offset_image = ndi.shift(image, shift)
offset_image *= corrupted_pixels
print(f'Known offset (row, col): {shift}')

# Determine what the mask is based on which pixels are invalid
# In this case, we know what the mask should be since we corrupted
# the pixels ourselves
mask = corrupted_pixels

detected_shift = phase_cross_correlation(image, offset_image,
                                           reference_mask=mask)

print(f'Detected pixel offset (row, col): {-detected_shift}')

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, sharex=True, sharey=True,
                                    figsize=(8, 3))

ax1.imshow(image, cmap='gray')
ax1.set_axis_off()
ax1.set_title('Reference image')

ax2.imshow(offset_image.real, cmap='gray')
ax2.set_axis_off()
ax2.set_title('Corrupted, offset image')
```

(continues on next page)

¹ D. Padfield, “Masked object registration in the Fourier domain” IEEE Transactions on Image Processing (2012). DOI:10.1109/TIP.2011.2181402

(continued from previous page)

```

ax3.imshow(mask, cmap='gray')
ax3.set_axis_off()
ax3.set_title('Masked pixels')

plt.show()

```

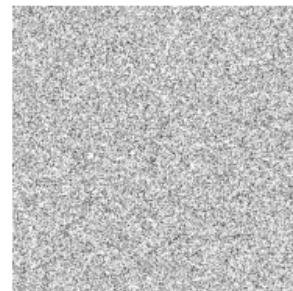
Reference image



Corrupted, offset image



Masked pixels



```

Known offset (row, col): (-22, 13)
/gITHUB/workspace/build/scikit-image/doc/examples/registration/plot_masked_register_
→translation.py:53: FutureWarning:

```

In scikit-image 0.22, `phase_cross_correlation` will start returning a tuple or 3 items →`(shift, error, phasediff)` always. To enable the new return behavior and silence this →warning, use `return_error='always'`.

```
Detected pixel offset (row, col): [-22. 13.]
```

Solid masks are another illustrating example. In this case, we have a limited view of an image and an offset image. The masks for these images need not be the same. The `phase_cross_correlation` function will correctly identify which part of the images should be compared.

```

image = data.camera()
shift = (-22, 13)

rr1, cc1 = draw.ellipse(259, 248, r_radius=125, c_radius=100,
                       shape=image.shape)

rr2, cc2 = draw.ellipse(300, 200, r_radius=110, c_radius=180,
                       shape=image.shape)

mask1 = np.zeros_like(image, dtype=bool)
mask2 = np.zeros_like(image, dtype=bool)
mask1[rr1, cc1] = True
mask2[rr2, cc2] = True

```

(continues on next page)

(continued from previous page)

```

offset_image = ndi.shift(image, shift)
image *= mask1
offset_image *= mask2

print(f'Known offset (row, col): {shift}')

detected_shift = phase_cross_correlation(image, offset_image,
                                           reference_mask=mask1,
                                           moving_mask=mask2)

print(f'Detected pixel offset (row, col): {-detected_shift}')

fig = plt.figure(figsize=(8,3))
ax1 = plt.subplot(1, 2, 1)
ax2 = plt.subplot(1, 2, 2, sharex=ax1, sharey=ax1)

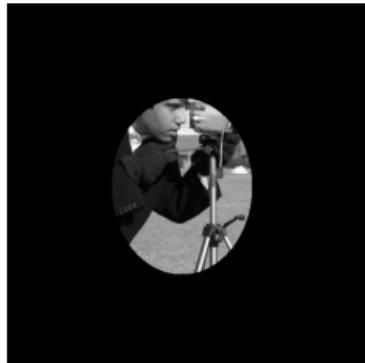
ax1.imshow(image, cmap='gray')
ax1.set_axis_off()
ax1.set_title('Reference image')

ax2.imshow(offset_image.real, cmap='gray')
ax2.set_axis_off()
ax2.set_title('Masked, offset image')

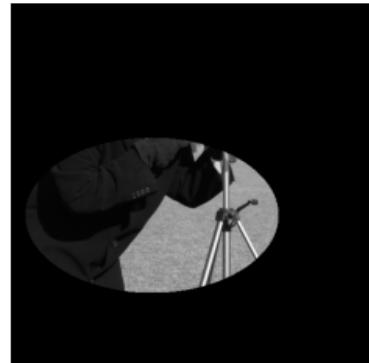
plt.show()

```

Reference image



Masked, offset image



```

Known offset (row, col): (-22, 13)
/github/workspace/build/scikit-image/doc/examples/registration/plot_masked_register_
translation.py:102: FutureWarning:

```

In scikit-image 0.22, `phase_cross_correlation` will start returning a tuple or 3 items
`(shift, error, phasediff)` always. To enable the new return behavior and silence this
warning, use `return_error='always'`.

```
Detected pixel offset (row, col): [-22.  13.]
```

Total running time of the script: (0 minutes 0.917 seconds)

Registration using optical flow

Demonstration of image registration using optical flow.

By definition, the optical flow is the vector field (u, v) verifying $image1(x+u, y+v) = image0(x, y)$, where $(image0, image1)$ is a couple of consecutive 2D frames from a sequence. This vector field can then be used for registration by image warping.

To display registration results, an RGB image is constructed by assigning the result of the registration to the red channel and the target image to the green and blue channels. A perfect registration results in a gray level image while misregistered pixels appear colored in the constructed RGB image.

```
import numpy as np
from matplotlib import pyplot as plt
from skimage.color import rgb2gray
from skimage.data import stereo_motorcycle, vortex
from skimage.transform import warp
from skimage.registration import optical_flow_tvl1, optical_flow_ilk

# --- Load the sequence
image0, image1, disp = stereo_motorcycle()

# --- Convert the images to gray level: color is not supported.
image0 = rgb2gray(image0)
image1 = rgb2gray(image1)

# --- Compute the optical flow
v, u = optical_flow_tvl1(image0, image1)

# --- Use the estimated optical flow for registration

nr, nc = image0.shape

row_coords, col_coords = np.meshgrid(np.arange(nr), np.arange(nc),
                                      indexing='ij')

image1_warp = warp(image1, np.array([row_coords + v, col_coords + u]),
                   mode='edge')

# build an RGB image with the unregistered sequence
seq_im = np.zeros((nr, nc, 3))
seq_im[..., 0] = image1
seq_im[..., 1] = image0
seq_im[..., 2] = image0

# build an RGB image with the registered sequence
reg_im = np.zeros((nr, nc, 3))
reg_im[..., 0] = image1_warp
reg_im[..., 1] = image0
reg_im[..., 2] = image0

# build an RGB image with the registered sequence
target_im = np.zeros((nr, nc, 3))
target_im[..., 0] = image0
target_im[..., 1] = image0
```

(continues on next page)

(continued from previous page)

```
target_im[..., 2] = image0

# --- Show the result

fig, (ax0, ax1, ax2) = plt.subplots(3, 1, figsize=(5, 10))

ax0.imshow(seq_im)
ax0.set_title("Unregistered sequence")
ax0.set_axis_off()

ax1.imshow(reg_im)
ax1.set_title("Registered sequence")
ax1.set_axis_off()

ax2.imshow(target_im)
ax2.set_title("Target")
ax2.set_axis_off()

fig.tight_layout()
```

Unregistered sequence



Registered sequence



Target



The estimated vector field (u, v) can also be displayed with a quiver plot.

In the following example, Iterative Lukas-Kanade algorithm (iLK) is applied to images of particles in the context of particle image velocimetry (PIV). The sequence is the Case B from the [PIV challenge 2001](#)

```
image0, image1 = vortex()

# --- Compute the optical flow
v, u = optical_flow_ilk(image0, image1, radius=15)

# --- Compute flow magnitude
norm = np.sqrt(u ** 2 + v ** 2)

# --- Display
fig, (ax0, ax1) = plt.subplots(1, 2, figsize=(8, 4))

# --- Sequence image sample

ax0.imshow(image0, cmap='gray')
ax0.set_title("Sequence image sample")
ax0.set_axis_off()

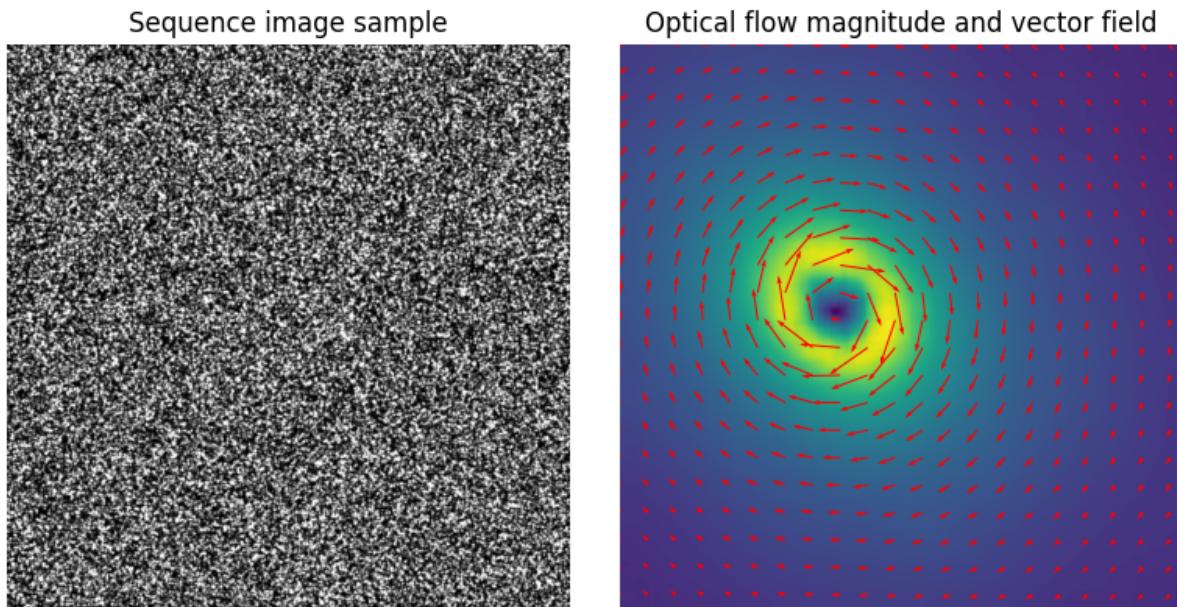
# --- Quiver plot arguments

nvec = 20 # Number of vectors to be displayed along each image dimension
nl, nc = image0.shape
step = max(nl//nvec, nc//nvec)

y, x = np.mgrid[:nl:step, :nc:step]
u_ = u[::step, ::step]
v_ = v[::step, ::step]

ax1.imshow(norm)
ax1.quiver(x, y, u_, v_, color='r', units='dots',
            angles='xy', scale_units='xy', lw=3)
ax1.set_title("Optical flow magnitude and vector field")
ax1.set_axis_off()
fig.tight_layout()

plt.show()
```



Total running time of the script: (0 minutes 6.340 seconds)

Assemble images with simple image stitching

This example demonstrates how a set of images can be assembled under the hypothesis of rigid body motions.

```
from matplotlib import pyplot as plt
import numpy as np
from skimage import data, util, transform, feature, measure, filters, metrics

def match_locations(img0, img1, coords0, coords1, radius=5, sigma=3):
    """Match image locations using SSD minimization.

    Areas from `img0` are matched with areas from `img1`. These areas
    are defined as patches located around pixels with Gaussian
    weights.

    Parameters:
    -----
    img0, img1 : 2D array
        Input images.
    coords0 : (2, m) array_like
        Centers of the reference patches in `img0`.
    coords1 : (2, n) array_like
        Centers of the candidate patches in `img1`.
    radius : int
        Radius of the considered patches.
    sigma : float
        Standard deviation of the Gaussian kernel centered over the patches.

    Returns:
    -----
    match_coords: (2, m) array
        Indices of the matched patches in `img1` for each patch in `img0`.
```

(continues on next page)

(continued from previous page)

The points in `coords1` that are the closest corresponding matches to those in `coords0` as determined by the (Gaussian weighted) sum of squared differences between patches surrounding each point.

```
"""
y, x = np.mgrid[-radius:radius + 1, -radius:radius + 1]
weights = np.exp(-0.5 * (x ** 2 + y ** 2) / sigma ** 2)
weights /= 2 * np.pi * sigma * sigma

match_list = []
for r0, c0 in coords0:
    roi0 = img0[r0 - radius:r0 + radius + 1, c0 - radius:c0 + radius + 1]
    roi1_list = [img1[r1 - radius:r1 + radius + 1,
                      c1 - radius:c1 + radius + 1] for r1, c1 in coords1]
    # sum of squared differences
    ssd_list = [np.sum(weights * (roi0 - roi1) ** 2) for roi1 in roi1_list]
    match_list.append(coords1[np.argmin(ssd_list)])

return np.array(match_list)
```

Data generation

For this example, we generate a list of slightly tilted noisy images.

```
img = data.moon()

angle_list = [0, 5, 6, -2, 3, -4]
center_list = [(0, 0), (10, 10), (5, 12), (11, 21), (21, 17), (43, 15)]

img_list = [transform.rotate(img, angle=a, center=c)[40:240, 50:350]
            for a, c in zip(angle_list, center_list)]
ref_img = img_list[0].copy()

img_list = [util.random_noise(filters.gaussian(im, 1.1), var=5e-4, rng=seed)
            for seed, im in enumerate(img_list)]

psnr_ref = metrics.peak_signal_noise_ratio(ref_img, img_list[0])
```

Image registration

Note: This step is performed using the approach described in *Robust matching using RANSAC*, but any other method from the *Image registration* section can be applied, depending on your problem.

Reference points are detected over all images in the list.

```
min_dist = 5
corner_list = [feature.corner_peaks(
    feature.corner_harris(img), threshold_rel=0.001, min_distance=min_dist)
               for img in img_list]
```

The Harris corners detected in the first image are chosen as references. Then the detected points on the other images are matched to the reference points.

```
img0 = img_list[0]
coords0 = corner_list[0]
matching_corners = [match_locations(img0, img1, coords0, coords1, min_dist)
                    for img1, coords1 in zip(img_list, corner_list)]
```

Once all the points are registered to the reference points, robust relative affine transformations can be estimated using the RANSAC method.

```
src = np.array(coords0)
trfm_list = [measure.ransac((dst, src),
                             transform.EuclideanTransform, min_samples=3,
                             residual_threshold=2, max_trials=100)[0].params
             for dst in matching_corners]

fig, ax_list = plt.subplots(6, 2, figsize=(6, 9), sharex=True, sharey=True)
for idx, (im, trfm, (ax0, ax1)) in enumerate(zip(img_list, trfm_list,
                                                   ax_list)):
    ax0.imshow(im, cmap="gray", vmin=0, vmax=1)
    ax1.imshow(transform.warp(im, trfm), cmap="gray", vmin=0, vmax=1)

    if idx == 0:
        ax0.set_title("Tilted images")
        ax0.set_ylabel(f"Reference Image\nPSNR={psnr_ref:.2f}")
        ax1.set_title("Registered images")

    ax0.set(xticklabels=[], yticklabels=[], xticks=[], yticks[])
    ax1.set_axis_off()

fig.tight_layout()
```

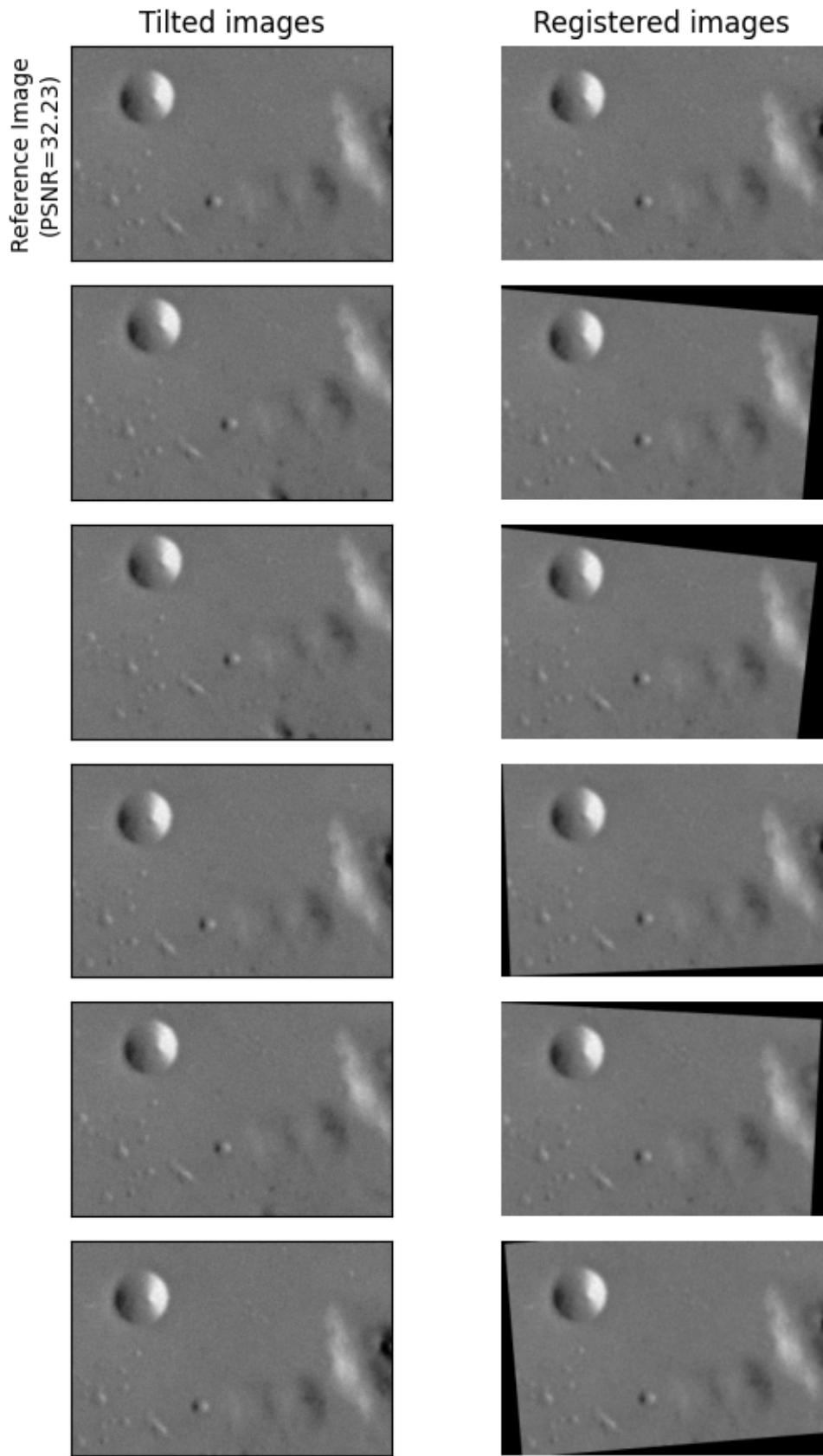


Image assembling

A composite image can be obtained using the positions of the registered images relative to the reference one. To do so, we define a global domain around the reference image and position the other images in this domain.

A global transformation is defined to move the reference image in the global domain image via a simple translation:

```
margin = 50
height, width = img_list[0].shape
out_shape = height + 2 * margin, width + 2 * margin
glob_trfm = np.eye(3)
glob_trfm[:2, 2] = -margin, -margin
```

Finally, the relative position of the other images in the global domain are obtained by composing the global transformation with the relative transformations:

```
global_img_list = [transform.warp(img, trfm.dot(glob_trfm),
                                  output_shape=out_shape,
                                  mode="constant", cval=np.nan)
                    for img, trfm in zip(img_list, trfm_list)]

all_nan_mask = np.all([np.isnan(img) for img in global_img_list], axis=0)
global_img_list[0][all_nan_mask] = 1.

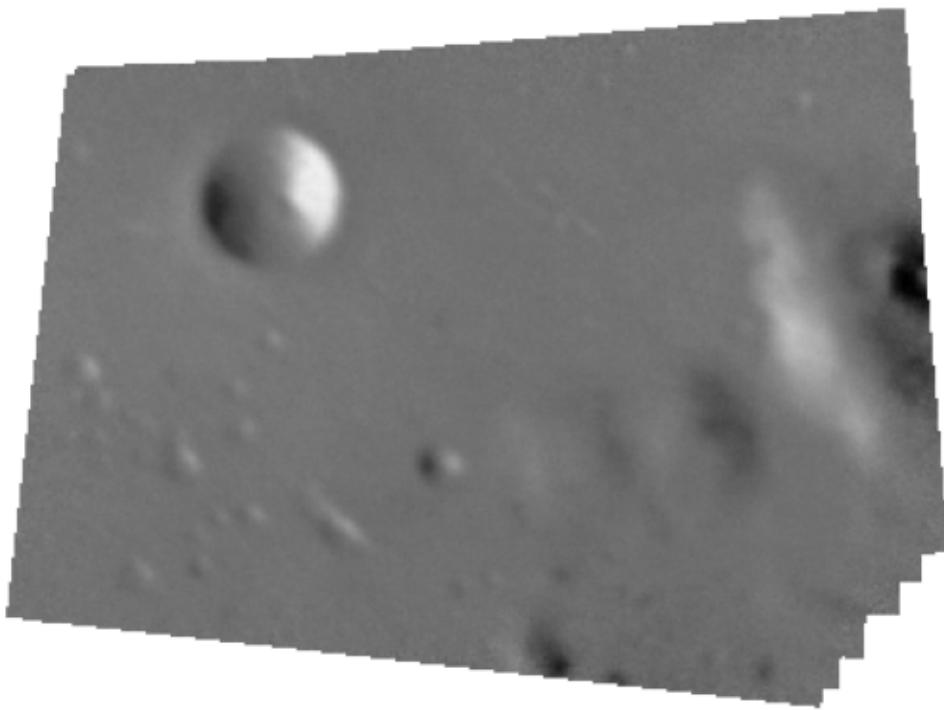
composite_img = np.nanmean(global_img_list, 0)
psnr_composite = metrics.peak_signal_noise_ratio(
    ref_img,
    composite_img[margin:margin + height,
                  margin:margin + width])

fig, ax = plt.subplots(1, 1)

ax.imshow(composite_img, cmap="gray", vmin=0, vmax=1)
ax.set_axis_off()
ax.set_title(f'Reconstructed image (PSNR={psnr_composite:.2f})')
fig.tight_layout()

plt.show()
```

Reconstructed image (PSNR=36.68)



Total running time of the script: (0 minutes 0.976 seconds)

Using Polar and Log-Polar Transformations for Registration

Phase correlation (`registration.phase_cross_correlation`) is an efficient method for determining translation offset between pairs of similar images. However this approach relies on a near absence of rotation/scaling differences between the images, which are typical in real-world examples.

To recover rotation and scaling differences between two images, we can take advantage of two geometric properties of the log-polar transform and the translation invariance of the frequency domain. First, rotation in Cartesian space becomes translation along the angular coordinate (θ) axis of log-polar space. Second, scaling in Cartesian space becomes translation along the radial coordinate ($\rho = \ln \sqrt{x^2 + y^2}$) of log-polar space. Finally, differences in translation in the spatial domain do not impact magnitude spectrum in the frequency domain.

In this series of examples, we build on these concepts to show how the log-polar transform `transform.warp_polar` can be used in conjunction with phase correlation to recover rotation and scaling differences between two images that also have a translation offset.

Recover rotation difference with a polar transform

In this first example, we consider the simple case of two images that only differ with respect to rotation around a common center point. By remapping these images into polar space, the rotation difference becomes a simple translation difference that can be recovered by phase correlation.

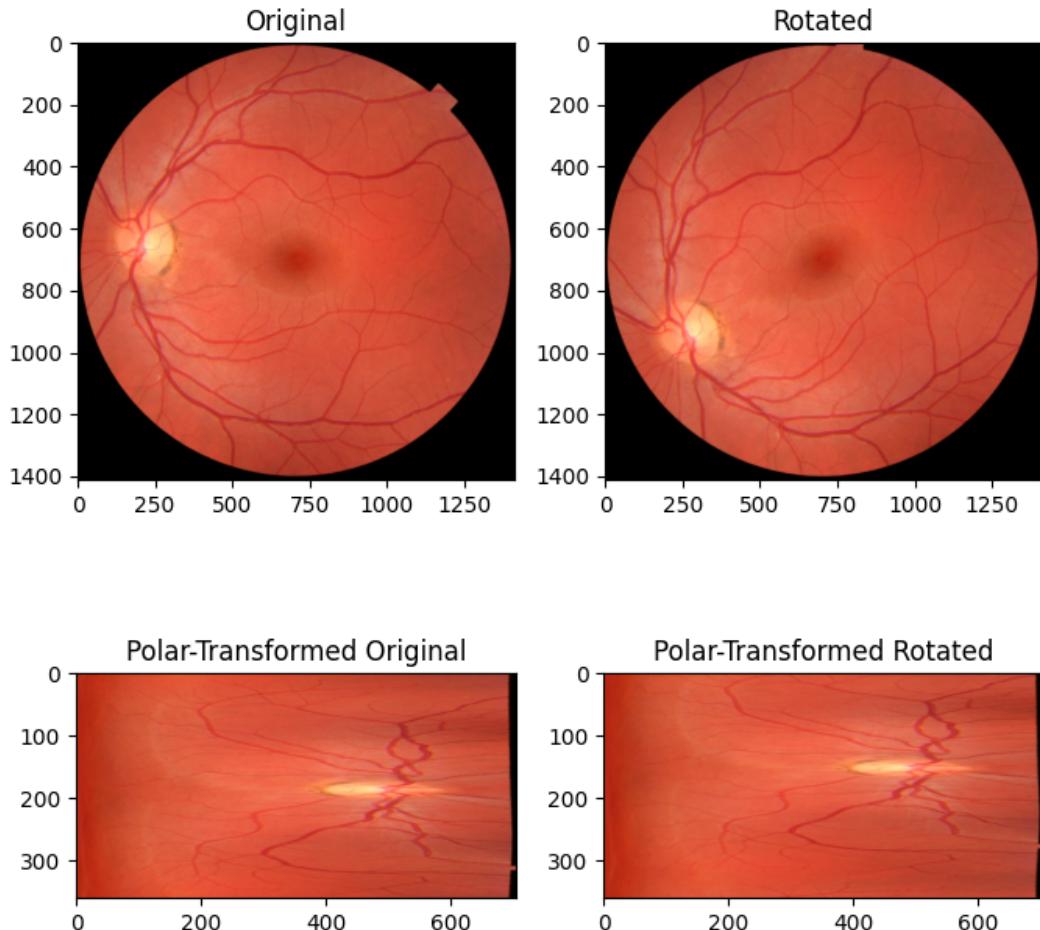
```
import numpy as np
import matplotlib.pyplot as plt

from skimage import data
from skimage.registration import phase_cross_correlation
from skimage.transform import warp_polar, rotate, rescale
from skimage.util import img_as_float

radius = 705
angle = 35
image = data.retina()
image = img_as_float(image)
rotated = rotate(image, angle)
image_polar = warp_polar(image, radius=radius, channel_axis=-1)
rotated_polar = warp_polar(rotated, radius=radius, channel_axis=-1)

fig, axes = plt.subplots(2, 2, figsize=(8, 8))
ax = axes.ravel()
ax[0].set_title("Original")
ax[0].imshow(image)
ax[1].set_title("Rotated")
ax[1].imshow(rotated)
ax[2].set_title("Polar-Transformed Original")
ax[2].imshow(image_polar)
ax[3].set_title("Polar-Transformed Rotated")
ax[3].imshow(rotated_polar)
plt.show()

shifts, error, phasediff = phase_cross_correlation(image_polar,
                                                    rotated_polar,
                                                    normalization=None)
print(f'Expected value for counterclockwise rotation in degrees: '
      f'{angle}')
print(f'Recovered value for counterclockwise rotation: '
      f'{shifts[0]}')
```



Expected value for counterclockwise rotation in degrees: 35
Recovered value for counterclockwise rotation: 35.0

Recover rotation and scaling differences with log-polar transform

In this second example, the images differ by both rotation and scaling (note the axis tick values). By remapping these images into log-polar space, we can recover rotation as before, and now also scaling, by phase correlation.

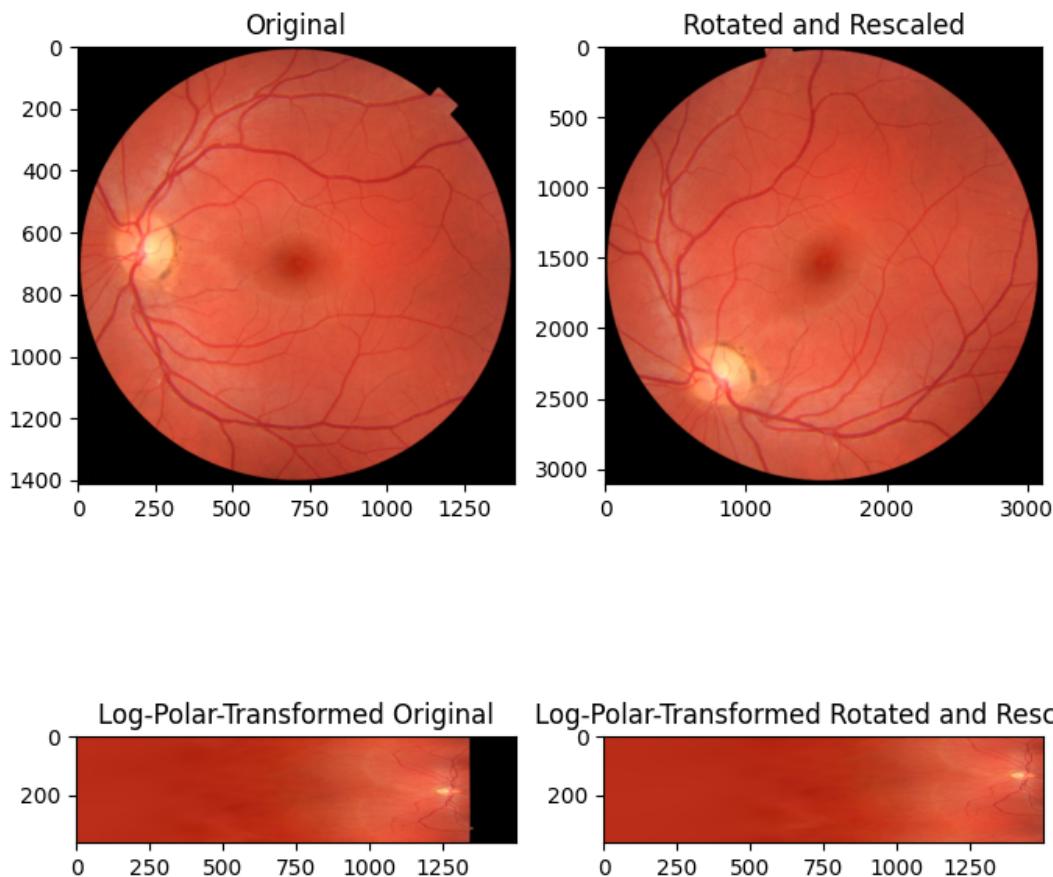
```
# radius must be large enough to capture useful info in larger image
radius = 1500
angle = 53.7
scale = 2.2
image = data.retina()
image = img_as_float(image)
rotated = rotate(image, angle)
rescaled = rescale(rotated, scale, channel_axis=-1)
image_polar = warp_polar(image, radius=radius,
                         scaling='log', channel_axis=-1)
rescaled_polar = warp_polar(rescaled, radius=radius,
                           scaling='log', channel_axis=-1)

fig, axes = plt.subplots(2, 2, figsize=(8, 8))
ax = axes.ravel()
ax[0].set_title("Original")
ax[0].imshow(image)
ax[1].set_title("Rotated and Rescaled")
ax[1].imshow(rescaled)
ax[2].set_title("Log-Polar-Transformed Original")
ax[2].imshow(image_polar)
ax[3].set_title("Log-Polar-Transformed Rotated and Rescaled")
ax[3].imshow(rescaled_polar)
plt.show()

# setting `upsample_factor` can increase precision
shifts, error, phasediff = phase_cross_correlation(image_polar, rescaled_polar,
                                                    upsample_factor=20,
                                                    normalization=None)
shiftr, shiftc = shifts[:2]

# Calculate scale factor from translation
klog = radius / np.log(radius)
shift_scale = 1 / (np.exp(shiftc / klog))

print(f'Expected value for cc rotation in degrees: {angle}')
print(f'Recovered value for cc rotation: {shiftr}')
print()
print(f'Expected value for scaling difference: {scale}')
print(f'Recovered value for scaling difference: {shift_scale}'')
```



```
Expected value for cc rotation in degrees: 53.7
Recovered value for cc rotation: 53.75
```

```
Expected value for scaling difference: 2.2
Recovered value for scaling difference: 2.1981889915232165
```

Register rotation and scaling on a translated image - Part 1

The above examples only work when the images to be registered share a center. However, it is more often the case that there is also a translation component to the difference between two images to be registered. One approach to register rotation, scaling and translation is to first correct for rotation and scaling, then solve for translation. It is possible to resolve rotation and scaling differences for translated images by working on the magnitude spectra of the Fourier-transformed images.

In this next example, we first show how the above approaches fail when two images differ by rotation, scaling, and translation.

```
from skimage.color import rgb2gray
from skimage.filters import window, difference_of_gaussians
from scipy.fft import fft2, fftshift

angle = 24
scale = 1.4
shiftx = 30
shifty = 15

image = rgb2gray(data.retina())
translated = image[shiftx:, shifty:]
rotated = rotate(translated, angle)
rescaled = rescale(rotated, scale)
sizer, sizec = image.shape
rts_image = rescaled[:sizer, :sizec]

# When center is not shared, log-polar transform is not helpful!
radius = 705
warped_image = warp_polar(image, radius=radius, scaling="log")
warped_rts = warp_polar(rts_image, radius=radius, scaling="log")
shifts, error, phasediff = phase_cross_correlation(warped_image, warped_rts,
                                                    upsample_factor=20,
                                                    normalization=None)

shiftx, shifty = shifts[:, 0]
klog = radius / np.log(radius)
shift_scale = 1 / (np.exp(shifty / klog))

fig, axes = plt.subplots(2, 2, figsize=(8, 8))
ax = axes.ravel()
ax[0].set_title("Original Image")
ax[0].imshow(image, cmap='gray')
ax[1].set_title("Modified Image")
ax[1].imshow(rts_image, cmap='gray')
ax[2].set_title("Log-Polar-Transformed Original")
ax[2].imshow(warped_image)
ax[3].set_title("Log-Polar-Transformed Modified")
ax[3].imshow(warped_rts)
fig.suptitle('log-polar-based registration fails when no shared center')
plt.show()

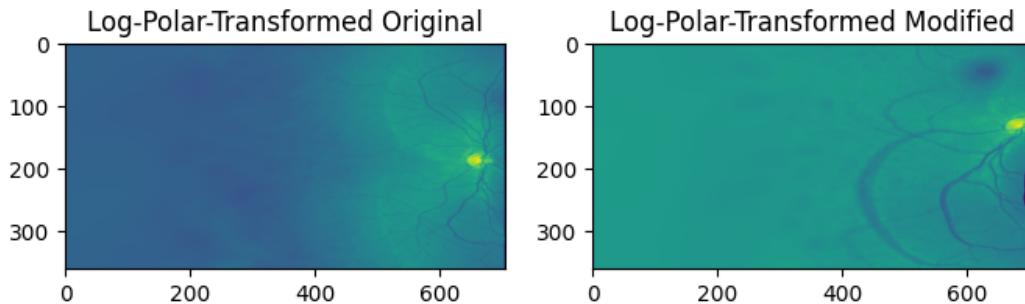
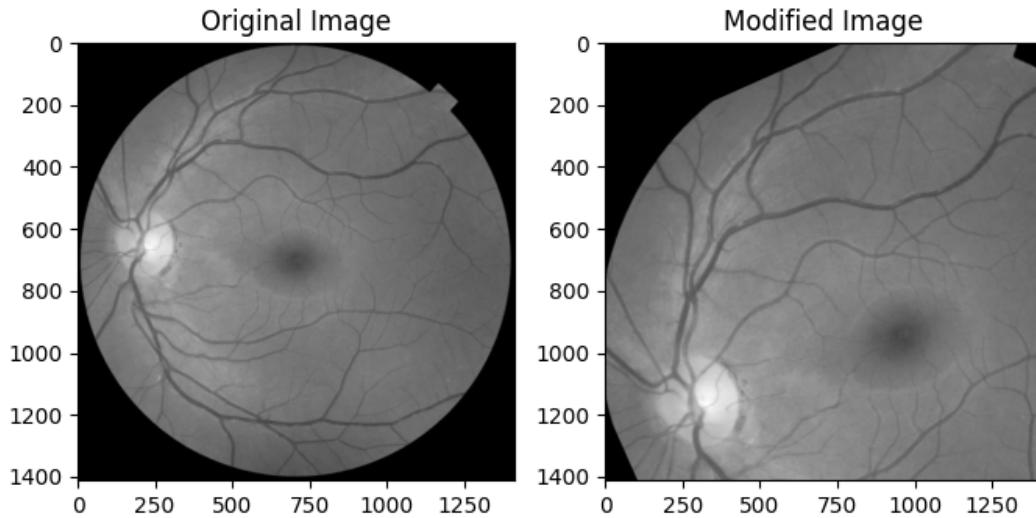
print(f'Expected value for cc rotation in degrees: {angle}')
print(f'Recovered value for cc rotation: {shiftx}'')
```

(continues on next page)

(continued from previous page)

```
print()  
print(f'Expected value for scaling difference: {scale}')  
print(f'Recovered value for scaling difference: {shift_scale}')
```

log-polar-based registration fails when no shared center



```
Expected value for cc rotation in degrees: 24  
Recovered value for cc rotation: -167.55
```

```
Expected value for scaling difference: 1.4  
Recovered value for scaling difference: 25.110458986143566
```

Register rotation and scaling on a translated image - Part 2

We next show how rotation and scaling differences, but not translation differences, are apparent in the frequency magnitude spectra of the images. These differences can be recovered by treating the magnitude spectra as images themselves, and applying the same log-polar + phase correlation approach taken above.

```
# First, band-pass filter both images
image = difference_of_gaussians(image, 5, 20)
rts_image = difference_of_gaussians(rts_image, 5, 20)

# window images
wimage = image * window('hann', image.shape)
rts_wimage = rts_image * window('hann', image.shape)

# work with shifted FFT magnitudes
image_fs = np.abs(fftshift(fft2(wimage)))
rts_fs = np.abs(fftshift(fft2(rts_wimage)))

# Create log-polar transformed FFT mag images and register
shape = image_fs.shape
radius = shape[0] // 8 # only take lower frequencies
warped_image_fs = warp_polar(image_fs, radius=radius, output_shape=shape,
                             scaling='log', order=0)
warped_rts_fs = warp_polar(rts_fs, radius=radius, output_shape=shape,
                           scaling='log', order=0)

warped_image_fs = warped_image_fs[:shape[0] // 2, :] # only use half of FFT
warped_rts_fs = warped_rts_fs[:shape[0] // 2, :]
shifts, error, phasediff = phase_cross_correlation(warped_image_fs,
                                                    warped_rts_fs,
                                                    upsample_factor=10,
                                                    normalization=None)

# Use translation parameters to calculate rotation and scaling parameters
shift_r, shift_c = shifts[:, 2]
recovered_angle = (360 / shape[0]) * shift_r
klog = shape[1] / np.log(radius)
shift_scale = np.exp(shift_c / klog)

fig, axes = plt.subplots(2, 2, figsize=(8, 8))
ax = axes.ravel()
ax[0].set_title("Original Image FFT\n(magnitude; zoomed)")
center = np.array(shape) // 2
ax[0].imshow(image_fs[center[0] - radius:center[0] + radius,
                     center[1] - radius:center[1] + radius],
              cmap='magma')
ax[1].set_title("Modified Image FFT\n(magnitude; zoomed)")
ax[1].imshow(rts_fs[center[0] - radius:center[0] + radius,
                     center[1] - radius:center[1] + radius],
              cmap='magma')
ax[2].set_title("Log-Polar-Transformed\nOriginal FFT")
ax[2].imshow(warped_image_fs, cmap='magma')
ax[3].set_title("Log-Polar-Transformed\nModified FFT")
```

(continues on next page)

(continued from previous page)

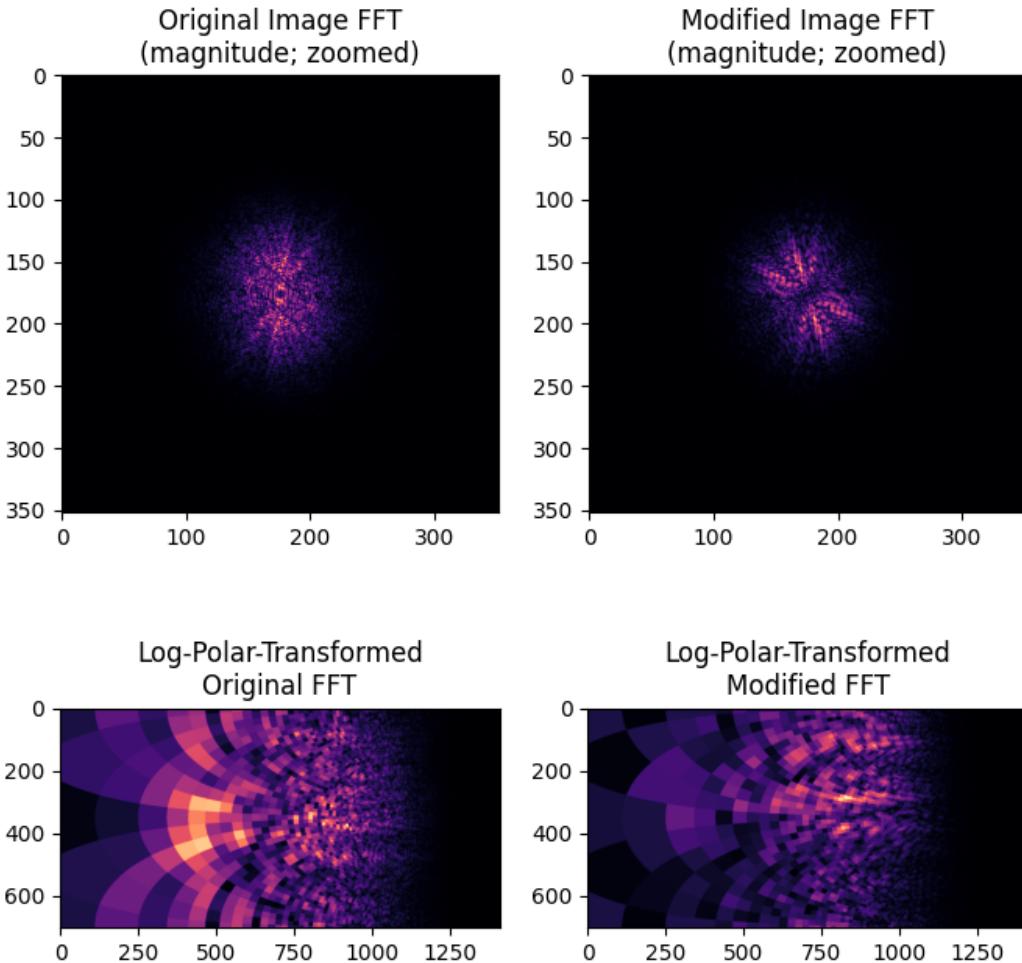
```

ax[3].imshow(warped_rts_fs, cmap='magma')
fig.suptitle('Working in frequency domain can recover rotation and scaling')
plt.show()

print(f'Expected value for cc rotation in degrees: {angle}')
print(f'Recovered value for cc rotation: {recovered_angle}')
print()
print(f'Expected value for scaling difference: {scale}')
print(f'Recovered value for scaling difference: {shift_scale}')

```

Working in frequency domain can recover rotation and scaling



Expected value for cc rotation in degrees: 24
Recovered value for cc rotation: 23.753366406803682

(continues on next page)

(continued from previous page)

```
Expected value for scaling difference: 1.4
Recovered value for scaling difference: 1.3901762721757436
```

Some notes on this approach

It should be noted that this approach relies on a couple of parameters that have to be chosen ahead of time, and for which there are no clearly optimal choices:

1. The images should have some degree of bandpass filtering applied, particularly to remove high frequencies, and different choices here may impact outcome. The bandpass filter also complicates matters because, since the images to be registered will differ in scale and these scale differences are unknown, any bandpass filter will necessarily attenuate different features between the images. For example, the log-polar transformed magnitude spectra don't really look "alike" in the last example here. Yet if you look closely, there are some common patterns in those spectra, and they do end up aligning well by phase correlation as demonstrated.
2. Images must be windowed using windows with circular symmetry, to remove the spectral leakage coming from image borders. There is no clearly optimal choice of window.

Finally, we note that large changes in scale will dramatically alter the magnitude spectra, especially since a big change in scale will usually be accompanied by some cropping and loss of information content. The literature advises staying within 1.8-2x scale change¹². This is fine for most biological imaging applications.

References

Total running time of the script: (0 minutes 8.970 seconds)

Filtering and restoration

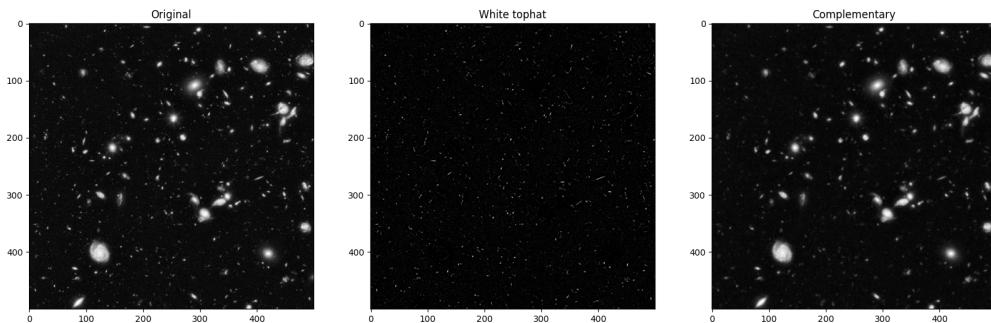
Removing small objects in grayscale images with a top hat filter

This example shows how to remove small objects from grayscale images. The top-hat transform¹ is an operation that extracts small elements and details from given images. Here we use a white top-hat transform, which is defined as the difference between the input image and its (mathematical morphology) opening.

¹ B.S. Reddy and B.N. Chatterji. An FFT-based technique for translation, rotation and scale- invariant image registration. *IEEE Trans. Image Processing*, 5(8):1266–1271, 1996. DOI:[10.1109/83.506761](https://doi.org/10.1109/83.506761)

² Tzimiropoulos, Georgios, and Tania Stathaki. “Robust FFT-based scale-invariant image registration.” In 4th SEAS DTC Technical Conference. 2009. DOI:[10.1109/TPAMI.2010.107](https://doi.org/10.1109/TPAMI.2010.107)

¹ https://en.wikipedia.org/wiki/Top-hat_transform



```

import matplotlib.pyplot as plt

from skimage import data
from skimage import color, morphology

image = color.rgb2gray(data.hubble_deep_field())[:500, :500]

footprint = morphology.disk(1)
res = morphology.white_tophat(image, footprint)

fig, ax = plt.subplots(ncols=3, figsize=(20, 8))
ax[0].set_title('Original')
ax[0].imshow(image, cmap='gray')
ax[1].set_title('White tophat')
ax[1].imshow(res, cmap='gray')
ax[2].set_title('Complementary')
ax[2].imshow(image - res, cmap='gray')

plt.show()

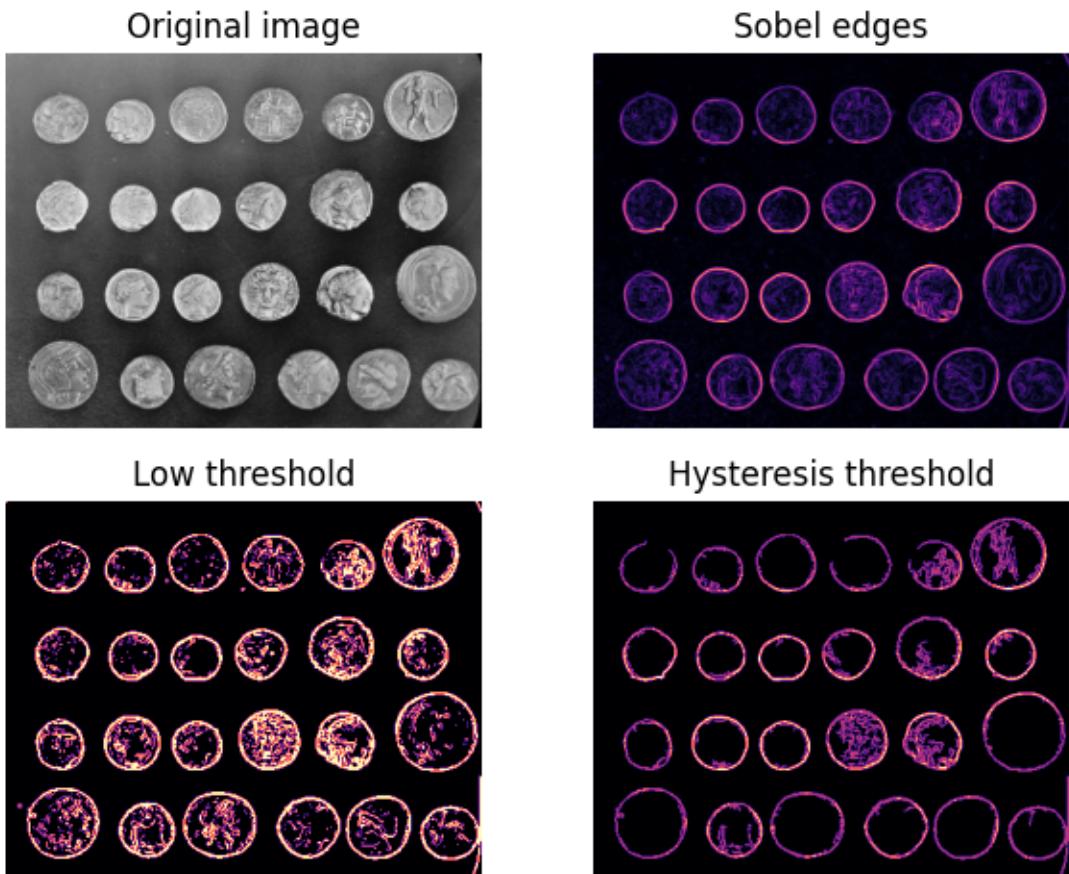
```

Total running time of the script: (0 minutes 0.478 seconds)

Hysteresis thresholding

Hysteresis is the lagging of an effect—a kind of inertia. In the context of thresholding, it means that areas above some *low* threshold are considered to be above the threshold *if* they are also connected to areas above a higher, more stringent, threshold. They can thus be seen as continuations of these high-confidence areas.

Below, we compare normal thresholding to hysteresis thresholding. Notice how hysteresis allows one to ignore “noise” outside of the coin edges.



```

import matplotlib.pyplot as plt
from skimage import data, filters

fig, ax = plt.subplots(nrows=2, ncols=2)

image = data.coins()
edges = filters.sobel(image)

low = 0.1
high = 0.35

lowt = (edges > low).astype(int)
hight = (edges > high).astype(int)
hyst = filters.apply_hysteresis_threshold(edges, low, high)

ax[0, 0].imshow(image, cmap='gray')
ax[0, 0].set_title('Original image')

ax[0, 1].imshow(edges, cmap='magma')
ax[0, 1].set_title('Sobel edges')

ax[1, 0].imshow(lowt, cmap='magma')
ax[1, 0].set_title('Low threshold')

ax[1, 1].imshow(hyst, cmap='magma')
ax[1, 1].set_title('Hysteresis threshold')

```

(continues on next page)

(continued from previous page)

```
ax[1, 1].imshow(hight + hyst, cmap='magma')
ax[1, 1].set_title('Hysteresis threshold')

for a in ax.ravel():
    a.axis('off')

plt.tight_layout()

plt.show()
```

Total running time of the script: (0 minutes 0.284 seconds)

Image Deconvolution

In this example, we deconvolve a noisy version of an image using Wiener and unsupervised Wiener algorithms. These algorithms are based on linear models that can't restore sharp edge as much as non-linear methods (like TV restoration) but are much faster.

Wiener filter

The inverse filter based on the PSF (Point Spread Function), the prior regularization (penalisation of high frequency) and the tradeoff between the data and prior adequacy. The regularization parameter must be hand tuned.

Unsupervised Wiener

This algorithm has a self-tuned regularization parameters based on data learning. This is not common and based on the following publication¹. The algorithm is based on an iterative Gibbs sampler that draw alternatively samples of posterior conditional law of the image, the noise power and the image frequency power.

¹ François Orieux, Jean-François Giovannelli, and Thomas Rodet, “Bayesian estimation of regularization and point spread function parameters for Wiener-Hunt deconvolution”, J. Opt. Soc. Am. A 27, 1593-1607 (2010) <https://www.osapublishing.org/josaa/abstract.cfm?URI=josaa-27-7-1593>
<https://hal.archives-ouvertes.fr/hal-00674508>



```

import numpy as np
import matplotlib.pyplot as plt

from skimage import color, data, restoration

rng = np.random.default_rng()

astro = color.rgb2gray(data.astronaut())
from scipy.signal import convolve2d as conv2
psf = np.ones((5, 5)) / 25
astro = conv2(astro, psf, 'same')
astro += 0.1 * astro.std() * rng.standard_normal(astro.shape)

deconvolved, _ = restoration.unsupervised_wiener(astro, psf)

fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(8, 5),
                      sharex=True, sharey=True)

plt.gray()

ax[0].imshow(astro, vmin=deconvolved.min(), vmax=deconvolved.max())
ax[0].axis('off')
ax[0].set_title('Data')

ax[1].imshow(deconvolved)
ax[1].axis('off')
ax[1].set_title('Self tuned restoration')

```

(continues on next page)

(continued from previous page)

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

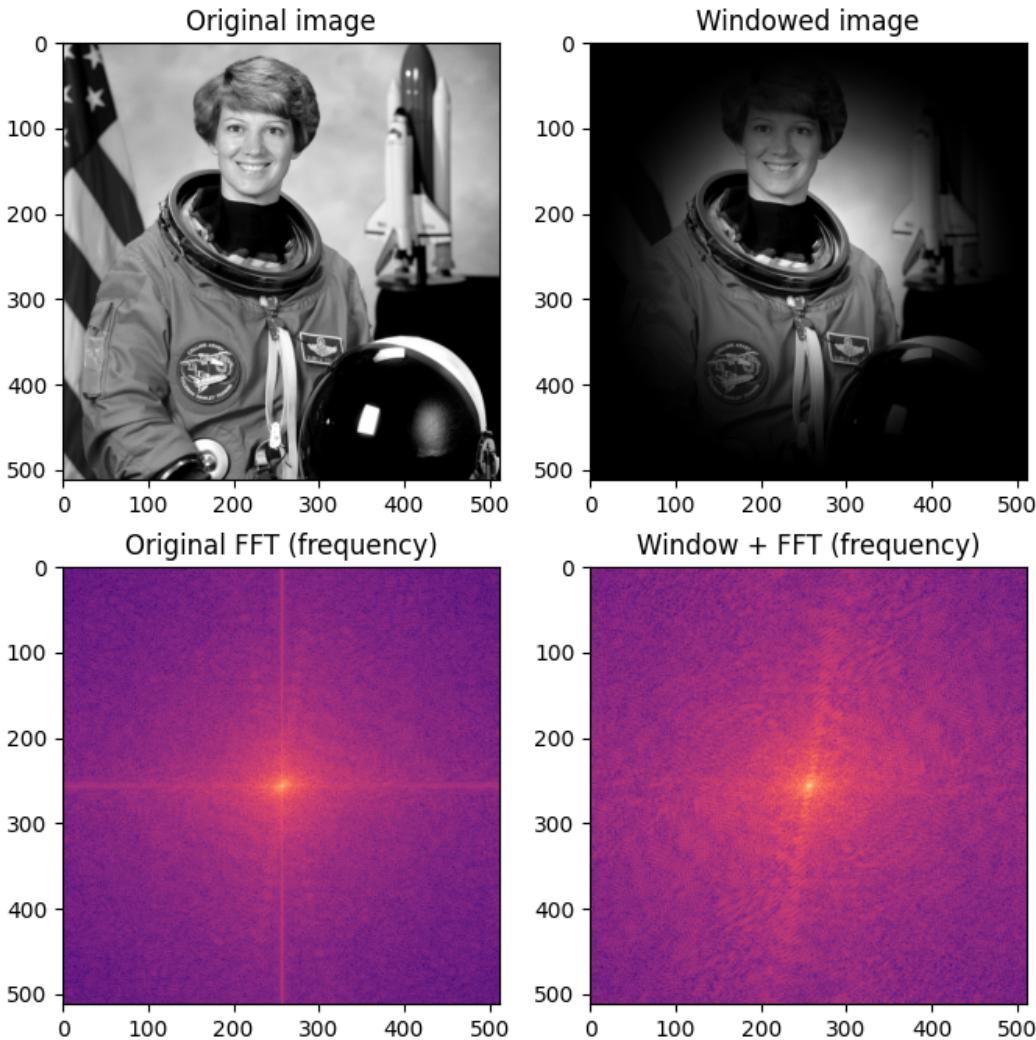
Total running time of the script: (0 minutes 0.661 seconds)

Using window functions with images

Fast Fourier transforms (FFTs) assume that the data being transformed represent one period of a periodic signal. Thus the endpoints of the signal to be transformed can behave as discontinuities in the context of the FFT. These discontinuities distort the output of the FFT, resulting in energy from “real” frequency components leaking into wider frequencies.

The effects of spectral leakage can be reduced by multiplying the signal with a window function. Windowing smoothly reduces the amplitude of the signal as it reaches the edges, removing the effect of the artificial discontinuity that results from the FFT.

In this example, we see that the FFT of a typical image can show strong spectral leakage along the x and y axes (see the vertical and horizontal lines in the figure). The application of a two-dimensional Hann window greatly reduces the spectral leakage, making the “real” frequency information more visible in the plot of the frequency component of the FFT.



```
import matplotlib.pyplot as plt
import numpy as np
from scipy.fft import fft2, fftshift
from skimage import img_as_float
from skimage.color import rgb2gray
from skimage.data import astronaut
from skimage.filters import window

image = img_as_float(rgb2gray(astronaut()))

wimage = image * window('hann', image.shape)

image_f = np.abs(fftshift(fft2(image)))
```

(continues on next page)

(continued from previous page)

```
wimage_f = np.abs(fftshift(fft2(wimage)))

fig, axes = plt.subplots(2, 2, figsize=(8, 8))
ax = axes.ravel()
ax[0].set_title("Original image")
ax[0].imshow(image, cmap='gray')
ax[1].set_title("Windowed image")
ax[1].imshow(wimage, cmap='gray')
ax[2].set_title("Original FFT (frequency)")
ax[2].imshow(np.log(image_f), cmap='magma')
ax[3].set_title("Window + FFT (frequency)")
ax[3].imshow(np.log(wimage_f), cmap='magma')
plt.show()
```

Total running time of the script: (0 minutes 0.427 seconds)

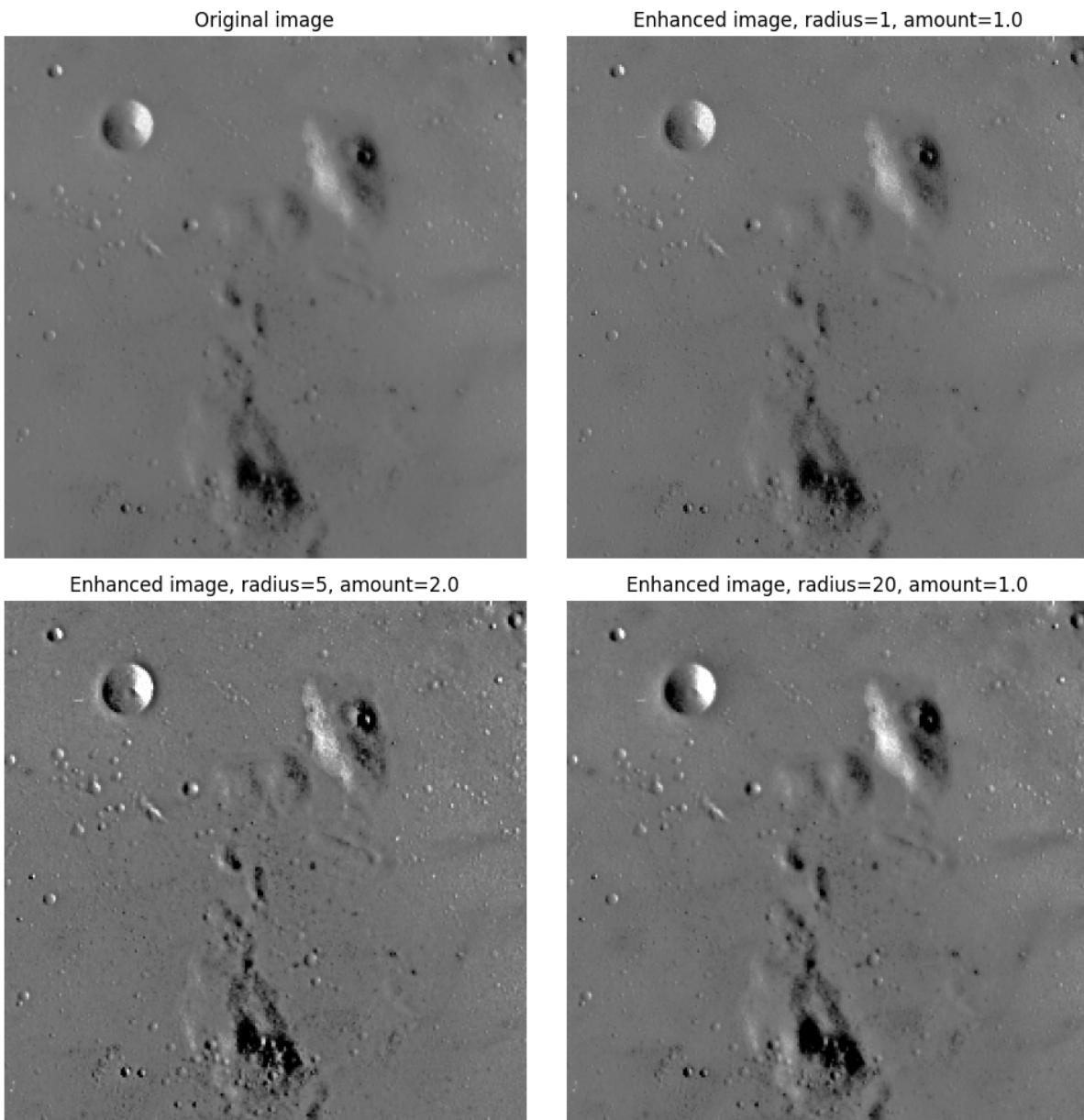
Unsharp masking

Unsharp masking is a linear image processing technique which sharpens the image. The sharp details are identified as a difference between the original image and its blurred version. These details are then scaled, and added back to the original image:

$$\text{enhanced image} = \text{original} + \text{amount} * (\text{original} - \text{blurred})$$

The blurring step could use any image filter method, e.g. median filter, but traditionally a gaussian filter is used. The radius parameter in the unsharp masking filter refers to the sigma parameter of the gaussian filter.

This example shows the effect of different radius and amount parameters.



```
from skimage import data
from skimage.filters import unsharp_mask
import matplotlib.pyplot as plt

image = data.moon()
result_1 = unsharp_mask(image, radius=1, amount=1)
result_2 = unsharp_mask(image, radius=5, amount=2)
result_3 = unsharp_mask(image, radius=20, amount=1)

fig, axes = plt.subplots(nrows=2, ncols=2,
                       sharex=True, sharey=True, figsize=(10, 10))
ax = axes.ravel()
```

(continues on next page)

(continued from previous page)

```
ax[0].imshow(image, cmap=plt.cm.gray)
ax[0].set_title('Original image')
ax[1].imshow(result_1, cmap=plt.cm.gray)
ax[1].set_title('Enhanced image, radius=1, amount=1.0')
ax[2].imshow(result_2, cmap=plt.cm.gray)
ax[2].set_title('Enhanced image, radius=5, amount=2.0')
ax[3].imshow(result_3, cmap=plt.cm.gray)
ax[3].set_title('Enhanced image, radius=20, amount=1.0')

for a in ax:
    a.axis('off')
fig.tight_layout()
plt.show()
```

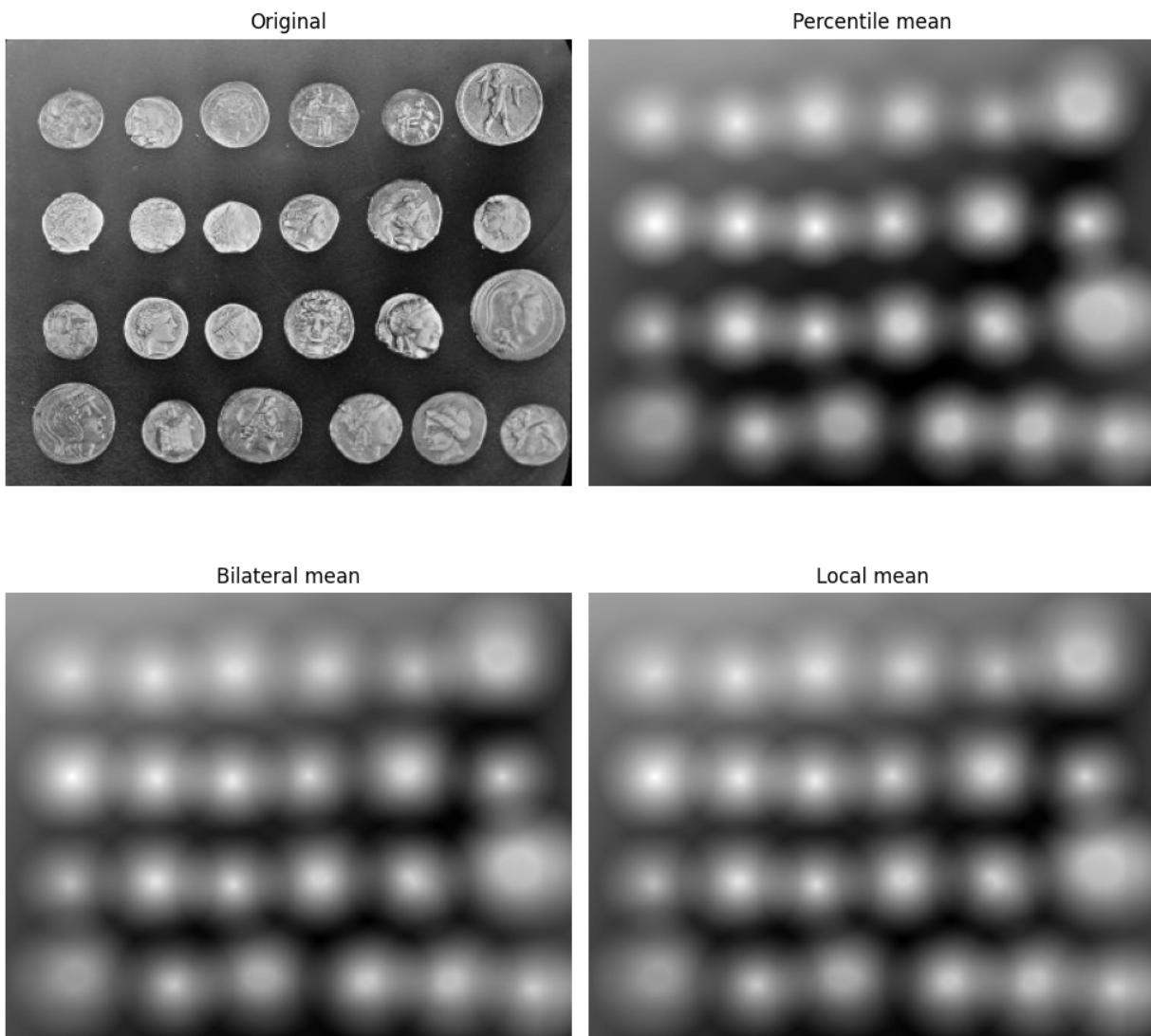
Total running time of the script: (0 minutes 0.686 seconds)

Mean filters

This example compares the following mean filters of the rank filter package:

- **local mean**: all pixels belonging to the structuring element to compute average gray level.
- **percentile mean**: only use values between percentiles p0 and p1 (here 10% and 90%).
- **bilateral mean**: only use pixels of the structuring element having a gray level situated inside g-s0 and g+s1 (here g-500 and g+500)

Percentile and usual mean give here similar results, these filters smooth the complete image (background and details). Bilateral mean exhibits a high filtering rate for continuous area (i.e. background) while higher image frequencies remain untouched.



```
import matplotlib.pyplot as plt

from skimage import data
from skimage.morphology import disk
from skimage.filters import rank

image = data.coins()
footprint = disk(20)

percentile_result = rank.mean_percentile(
    image, footprint=footprint, p0=.1, p1=.9
)
```

(continues on next page)

(continued from previous page)

```
bilateral_result = rank.mean_bilateral(  
    image, footprint=footprint, s0=500, s1=500  
)  
normal_result = rank.mean(image, footprint=footprint)  
  
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(10, 10),  
                       sharex=True, sharey=True)  
ax = axes.ravel()  
  
titles = ['Original', 'Percentile mean', 'Bilateral mean', 'Local mean']  
imgs = [image, percentile_result, bilateral_result, normal_result]  
for n in range(0, len(imgs)):  
    ax[n].imshow(imgs[n], cmap=plt.cm.gray)  
    ax[n].set_title(titles[n])  
    ax[n].axis('off')  
  
plt.tight_layout()  
plt.show()
```

Total running time of the script: (0 minutes 0.715 seconds)

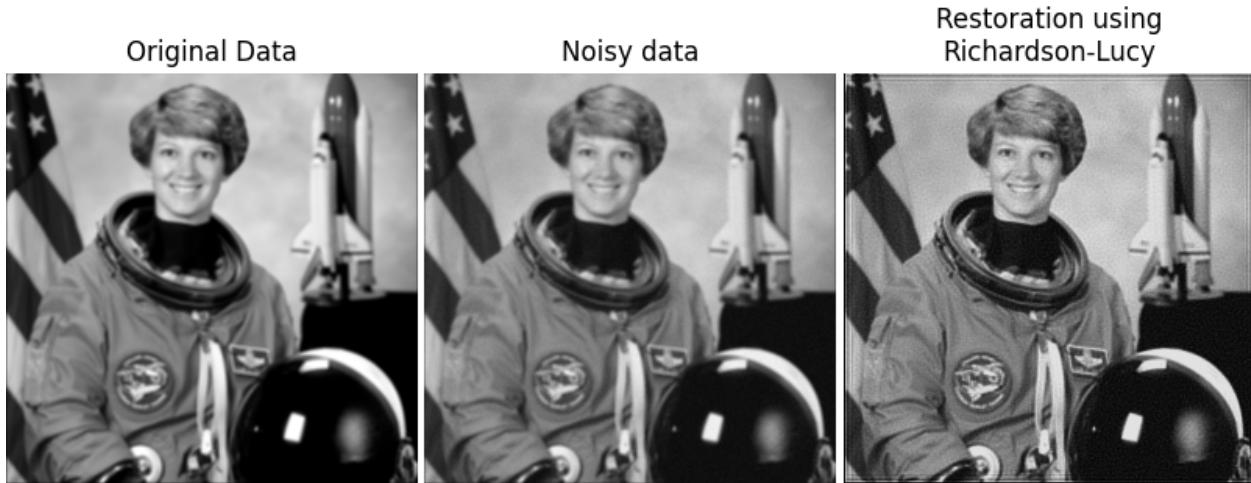
Image Deconvolution

In this example, we deconvolve an image using Richardson-Lucy deconvolution algorithm (¹,²).

The algorithm is based on a PSF (Point Spread Function), where PSF is described as the impulse response of the optical system. The blurred image is sharpened through a number of iterations, which needs to be hand-tuned.

¹ William Hadley Richardson, “Bayesian-Based Iterative Method of Image Restoration”, J. Opt. Soc. Am. A 27, 1593-1607 (1972), DOI:10.1364/JOSA.62.000055

² https://en.wikipedia.org/wiki/Richardson-Lucy_deconvolution



```

import numpy as np
import matplotlib.pyplot as plt

from scipy.signal import convolve2d as conv2

from skimage import color, data, restoration

rng = np.random.default_rng()

astro = color.rgb2gray(data.astronaut())

psf = np.ones((5, 5)) / 25
astro = conv2(astro, psf, 'same')
# Add Noise to Image
astro_noisy = astro.copy()
astro_noisy += (rng.poisson(lam=25, size=astro.shape) - 10) / 255.

# Restore Image using Richardson-Lucy algorithm
deconvolved_RL = restoration.richardson_lucy(astro_noisy, psf, num_iter=30)

fig, ax = plt.subplots(nrows=1, ncols=3, figsize=(8, 5))
plt.gray()

for a in (ax[0], ax[1], ax[2]):
    a.axis('off')

ax[0].imshow(astro)
ax[0].set_title('Original Data')

```

(continues on next page)

(continued from previous page)

```

ax[1].imshow(astro_noisy)
ax[1].set_title('Noisy data')

ax[2].imshow(deconvolved_RL, vmin=astro_noisy.min(), vmax=astro_noisy.max())
ax[2].set_title('Restoration using\nRichardson-Lucy')

fig.subplots_adjust(wspace=0.02, hspace=0.2,
                    top=0.9, bottom=0.05, left=0, right=1)
plt.show()

```

Total running time of the script: (0 minutes 0.759 seconds)

Estimate strength of blur

This example shows how the metric implemented in `measure.blur_effect` behaves, both as a function of the strength of blur and of the size of the re-blurring filter. This no-reference perceptual blur metric is described in¹.

```

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import scipy.ndimage as ndi

import plotly
import plotly.express as px
from skimage import (
    color, data, measure
)

```

Generate series of increasingly blurred images

Let us load an image available through scikit-image's data registry. The blur metric applies to single-channel images.

```

image = data.astronaut()
image = color.rgb2gray(image)

```

Let us blur this image with a series of uniform filters of increasing size.

```

blurred_images = [ndi.uniform_filter(image, size=k) for k in range(2, 32, 2)]
img_stack = np.stack(blurred_images)

fig = px.imshow(
    img_stack,
    animation_frame=0,
    binary_string=True,
    labels={'animation_frame': 'blur strength ~'}
)
plotly.io.show(fig)

```

¹ Frederique Crete, Thierry Dolmire, Patricia Ladret, and Marina Nicolas “The blur effect: perception and estimation with a new no-reference perceptual blur metric” Proc. SPIE 6492, Human Vision and Electronic Imaging XII, 64920I (2007) <https://hal.archives-ouvertes.fr/hal-00232709>
DOI:10.1117/12.702790

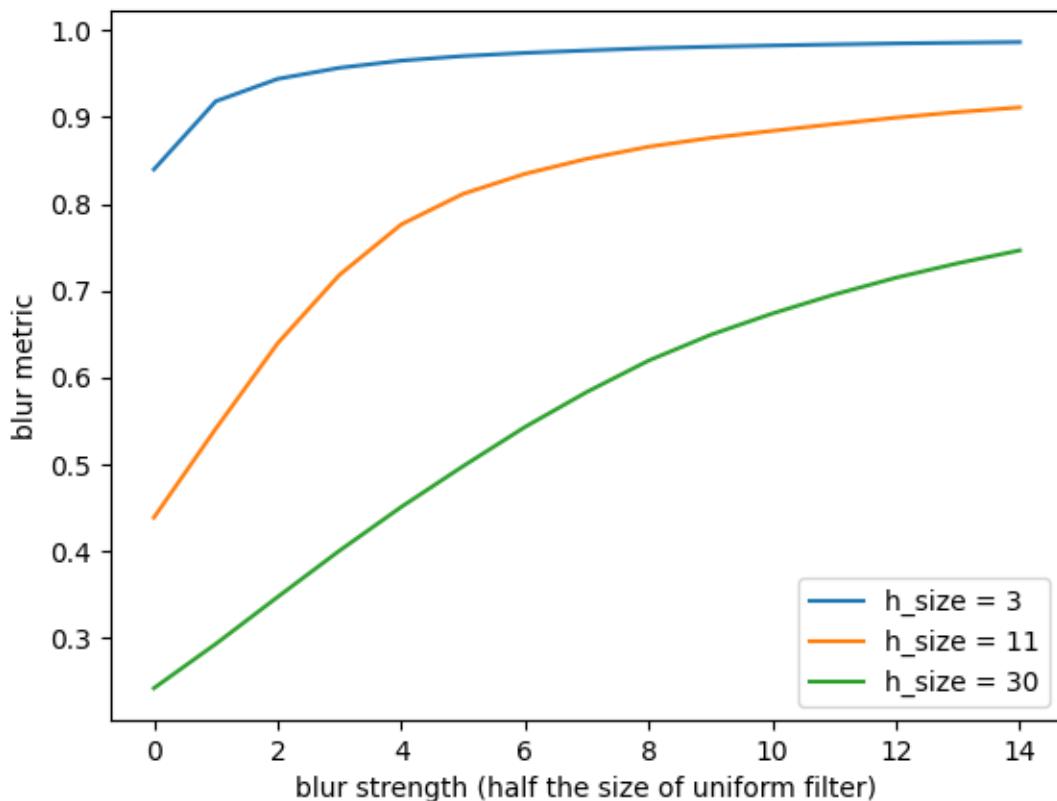
Plot blur metric

Let us compute the blur metric for all blurred images: We expect it to increase towards 1 with increasing blur strength. We compute it for three different values of re-blurring filter: 3, 11 (default), and 30.

```
B = pd.DataFrame(
    data=np.zeros(len(blurred_images), 3)),
    columns=['h_size = 3', 'h_size = 11', 'h_size = 30']
)
for ind, im in enumerate(blurred_images):
    B.loc[ind, 'h_size = 3'] = measure.blur_effect(im, h_size=3)
    B.loc[ind, 'h_size = 11'] = measure.blur_effect(im, h_size=11)
    B.loc[ind, 'h_size = 30'] = measure.blur_effect(im, h_size=30)

B.plot().set(xlabel='blur strength (half the size of uniform filter)',
             ylabel='blur metric')

plt.show()
```



We can see that as soon as the blur is stronger than (reaches the scale of) the size of the uniform filter, the metric gets close to 1 and, hence, tends asymptotically to 1 with increasing blur strength. The value of 11 pixels gives a blur metric which correlates best with human perception. That's why it's the default value in the implementation of the perceptual blur metric `measure.blur_effect`.

Total running time of the script: (0 minutes 1.944 seconds)

Entropy

In information theory, information entropy is the log-base-2 of the number of possible outcomes for a message.

For an image, local entropy is related to the complexity contained in a given neighborhood, typically defined by a structuring element. The entropy filter can detect subtle variations in the local gray level distribution.

In the first example, the image is composed of two surfaces with two slightly different distributions. The image has a uniform random distribution in the range [-15, +15] in the middle of the image and a uniform random distribution in the range [-14, 14] at the image borders, both centered at a gray value of 128. To detect the central square, we compute the local entropy measure using a circular structuring element of a radius big enough to capture the local gray level distribution. The second example shows how to detect texture in the camera image using a smaller structuring element.

Object detection

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

from skimage import data
from skimage.util import img_as_ubyte
from skimage.filters.rank import entropy
from skimage.morphology import disk

rng = np.random.default_rng()

noise_mask = np.full((128, 128), 28, dtype=np.uint8)
noise_mask[32:-32, 32:-32] = 30

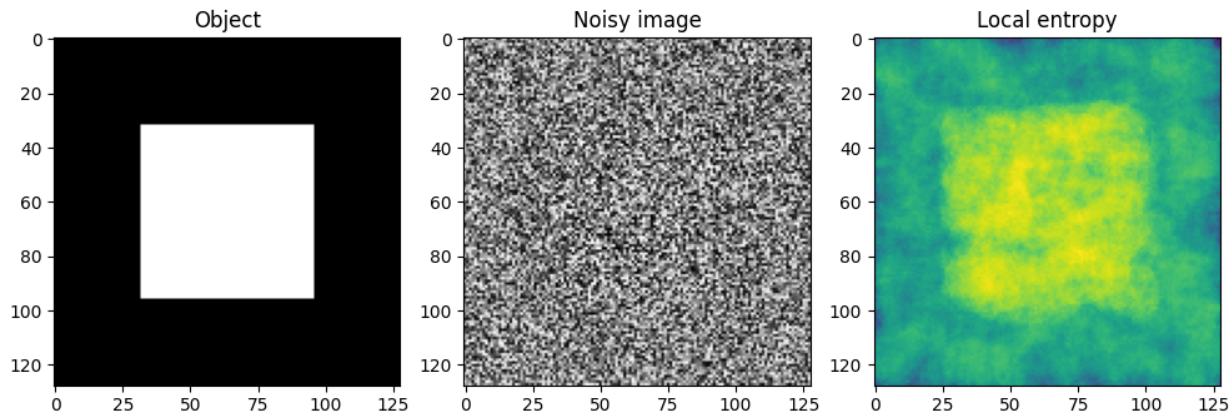
noise = (noise_mask * rng.random(noise_mask.shape) - 0.5
         * noise_mask).astype(np.uint8)
img = noise + 128

entr_img = entropy(img, disk(10))

fig, (ax0, ax1, ax2) = plt.subplots(nrows=1, ncols=3, figsize=(10, 4))

img0 = ax0.imshow(noise_mask, cmap='gray')
ax0.set_title("Object")
ax1.imshow(img, cmap='gray')
ax1.set_title("Noisy image")
ax2.imshow(entr_img, cmap='viridis')
ax2.set_title("Local entropy")

fig.tight_layout()
```



Texture detection

```
image = img_as_ubyte(data.camera())

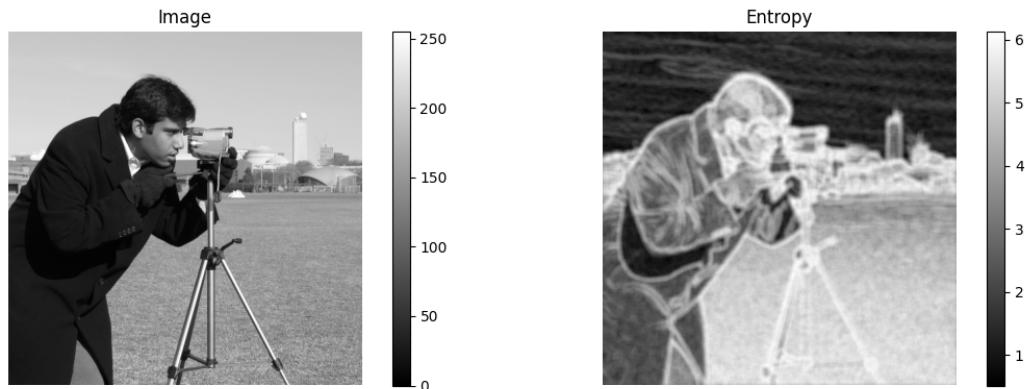
fig, (ax0, ax1) = plt.subplots(ncols=2, figsize=(12, 4),
                             sharex=True, sharey=True)

img0 = ax0.imshow(image, cmap=plt.cm.gray)
ax0.set_title("Image")
ax0.axis("off")
fig.colorbar(img0, ax=ax0)

img1 = ax1.imshow(entropy(image, disk(5)), cmap='gray')
ax1.set_title("Entropy")
ax1.axis("off")
fig.colorbar(img1, ax=ax1)

fig.tight_layout()

plt.show()
```



Total running time of the script: (0 minutes 1.040 seconds)

Calibrating Denoisers Using J-Invariance

In this example, we show how to find an optimally calibrated version of any denoising algorithm.

The calibration method is based on the *noise2self* algorithm of¹.

See also:

More details about the method are given in the full tutorial *Full tutorial on calibrating Denoisers Using J-Invariance*.

Calibrating a wavelet denoiser

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

from skimage.data import chelsea
from skimage.restoration import calibrate_denoiser, denoise_wavelet

from skimage.util import img_as_float, random_noise
from functools import partial

# rescale_sigma=True required to silence deprecation warnings
_denoise_wavelet = partial(denoise_wavelet, rescale_sigma=True)

image = img_as_float(chelsea())
sigma = 0.3
noisy = random_noise(image, var=sigma ** 2)

# Parameters to test when calibrating the denoising algorithm
parameter_ranges = {'sigma': np.arange(0.1, 0.3, 0.02),
                     'wavelet': ['db1', 'db2'],
                     'convert2ycbcr': [True, False],
                     'channel_axis': [-1]}

# Denoised image using default parameters of `denoise_wavelet`
default_output = denoise_wavelet(noisy, channel_axis=-1, rescale_sigma=True)

# Calibrate denoiser
calibrated_denoiser = calibrate_denoiser(noisy,
                                           _denoise_wavelet,
                                           denoise_parameters=parameter_ranges)

# Denoised image using calibrated denoiser
calibrated_output = calibrated_denoiser(noisy)

fig, axes = plt.subplots(1, 3, sharex=True, sharey=True, figsize=(15, 5))

for ax, img, title in zip(
    axes,
    [noisy, default_output, calibrated_output],
    ['Noisy Image', 'Denoised (Default)', 'Denoised (Calibrated)']):
    ax.imshow(img)
    ax.set_title(title)
```

(continues on next page)

¹ J. Batson & L. Royer. Noise2Self: Blind Denoising by Self-Supervision, International Conference on Machine Learning, p. 524-533 (2019).

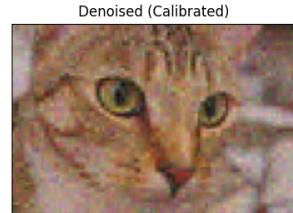
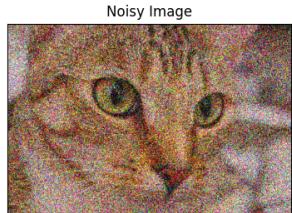
(continued from previous page)

```

ax.set_yticks([])
ax.set_xticks([])

plt.show()

```



Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).

Total running time of the script: (0 minutes 1.606 seconds)

Inpainting

Inpainting¹ is the process of reconstructing lost or deteriorated parts of images and videos.

The reconstruction is supposed to be performed in fully automatic way by exploiting the information presented in non-damaged regions.

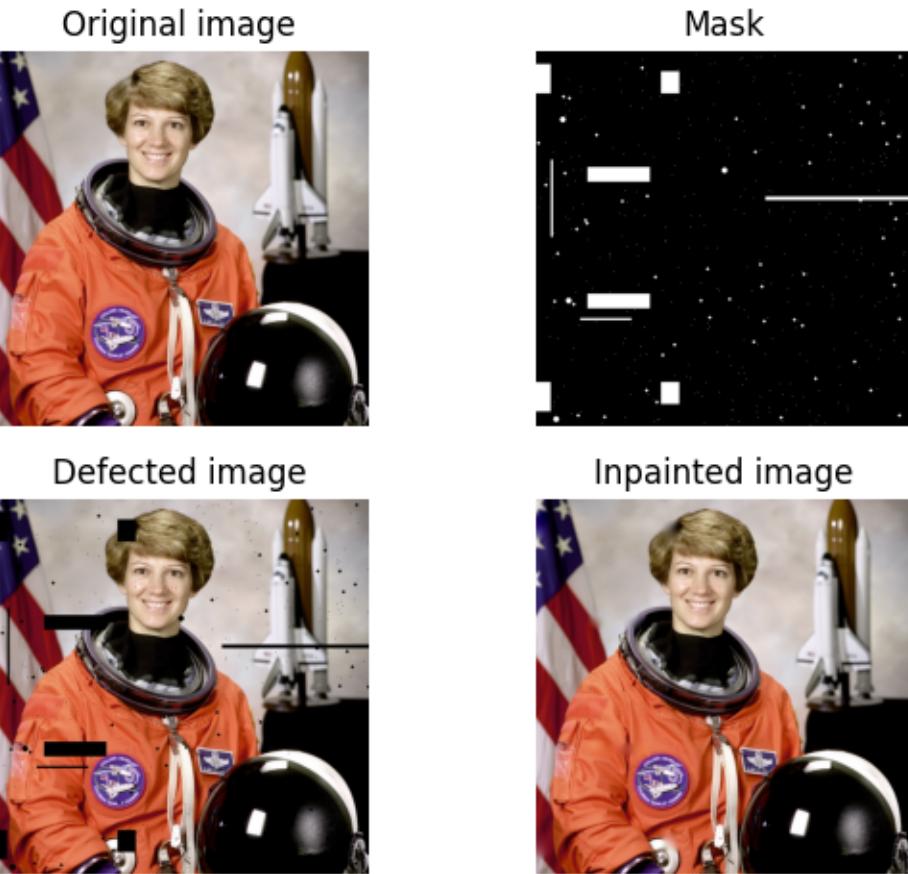
In this example, we show how the masked pixels get inpainted by inpainting algorithm based on ‘biharmonic equation’-assumption²³⁴.

¹ Wikipedia. Inpainting <https://en.wikipedia.org/wiki/Inpainting>

² Wikipedia. Biharmonic equation https://en.wikipedia.org/wiki/Biharmonic_equation

³ S.B.Damelin and N.S.Hoang. “On Surface Completion and Image Inpainting by Biharmonic Functions: Numerical Aspects”, International Journal of Mathematics and Mathematical Sciences, Vol. 2018, Article ID 3950312 DOI:10.1155/2018/3950312

⁴ C. K. Chui and H. N. Mhaskar, MRA Contextual-Recovery Extension of Smooth Functions on Manifolds, Appl. and Comp. Harmonic Anal., 28 (2010), 104-113, DOI:10.1016/j.acha.2009.04.004



```

import numpy as np
import matplotlib.pyplot as plt

from skimage import data
from skimage.morphology import disk, binary_dilation
from skimage.restoration import inpaint

image_orig = data.astronaut()

# Create mask with six block defect regions
mask = np.zeros(image_orig.shape[:-1], dtype=bool)
mask[20:60, 0:20] = 1
mask[160:180, 70:155] = 1
mask[30:60, 170:195] = 1
mask[-60:-30, 170:195] = 1
mask[-180:-160, 70:155] = 1
mask[-60:-20, 0:20] = 1

# add a few long, narrow defects
mask[200:205, -200:] = 1
mask[150:255, 20:23] = 1
mask[365:368, 60:130] = 1

```

(continues on next page)

(continued from previous page)

```
# add randomly positioned small point-like defects
rstate = np.random.default_rng(0)
for radius in [0, 2, 4]:
    # Larger defects are less common
    thresh = 3 + 0.25 * radius # make larger defects less common
    tmp_mask = rstate.standard_normal(image_orig.shape[:-1]) > thresh
    if radius > 0:
        tmp_mask = binary_dilation(tmp_mask, disk(radius, dtype=bool))
    mask[tmp_mask] = 1

# Apply defect mask to the image over the same region in each color channel
image_defect = image_orig * ~mask[..., np.newaxis]

image_result = inpaint.inpaint_biharmonic(image_defect, mask, channel_axis=-1)

fig, axes = plt.subplots(ncols=2, nrows=2)
ax = axes.ravel()

ax[0].set_title('Original image')
ax[0].imshow(image_orig)

ax[1].set_title('Mask')
ax[1].imshow(mask, cmap=plt.cm.gray)

ax[2].set_title('Defected image')
ax[2].imshow(image_defect)

ax[3].set_title('Inpainted image')
ax[3].imshow(image_result)

for a in ax:
    a.axis('off')

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

Total running time of the script: (0 minutes 0.443 seconds)

Band-pass filtering by Difference of Gaussians

Band-pass filters attenuate signal frequencies outside of a range (band) of interest. In image analysis, they can be used to denoise images while at the same time reducing low-frequency artifacts such as uneven illumination. Band-pass filters can be used to find image features such as blobs and edges.

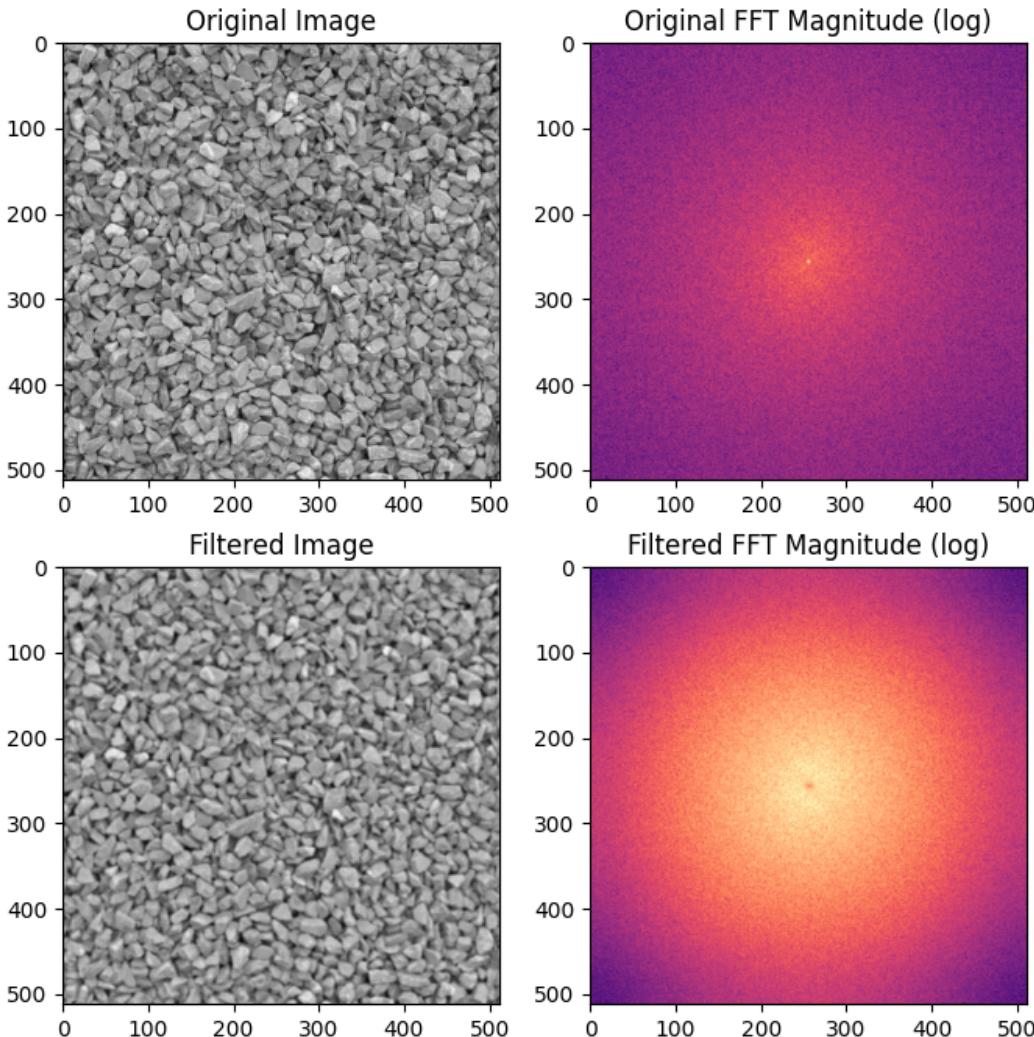
One method for applying band-pass filters to images is to subtract an image blurred with a Gaussian kernel from a less-blurred image. This example shows two applications of the Difference of Gaussians approach for band-pass filtering.

Denoise image and reduce shadows

```
import matplotlib.pyplot as plt
import numpy as np
from skimage.data import gravel
from skimage.filters import difference_of_gaussians, window
from scipy.fft import fftn, fftshift

image = gravel()
wimage = image * window('hann', image.shape) # window image to improve FFT
filtered_image = difference_of_gaussians(image, 1, 12)
filtered_wimage = filtered_image * window('hann', image.shape)
im_f_mag = fftshift(np.abs(fftn(wimage)))
fim_f_mag = fftshift(np.abs(fftn(filtered_wimage)))

fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(8, 8))
ax[0, 0].imshow(image, cmap='gray')
ax[0, 0].set_title('Original Image')
ax[0, 1].imshow(np.log(im_f_mag), cmap='magma')
ax[0, 1].set_title('Original FFT Magnitude (log)')
ax[1, 0].imshow(filtered_image, cmap='gray')
ax[1, 0].set_title('Filtered Image')
ax[1, 1].imshow(np.log(fim_f_mag), cmap='magma')
ax[1, 1].set_title('Filtered FFT Magnitude (log)')
plt.show()
```



Enhance edges in an image

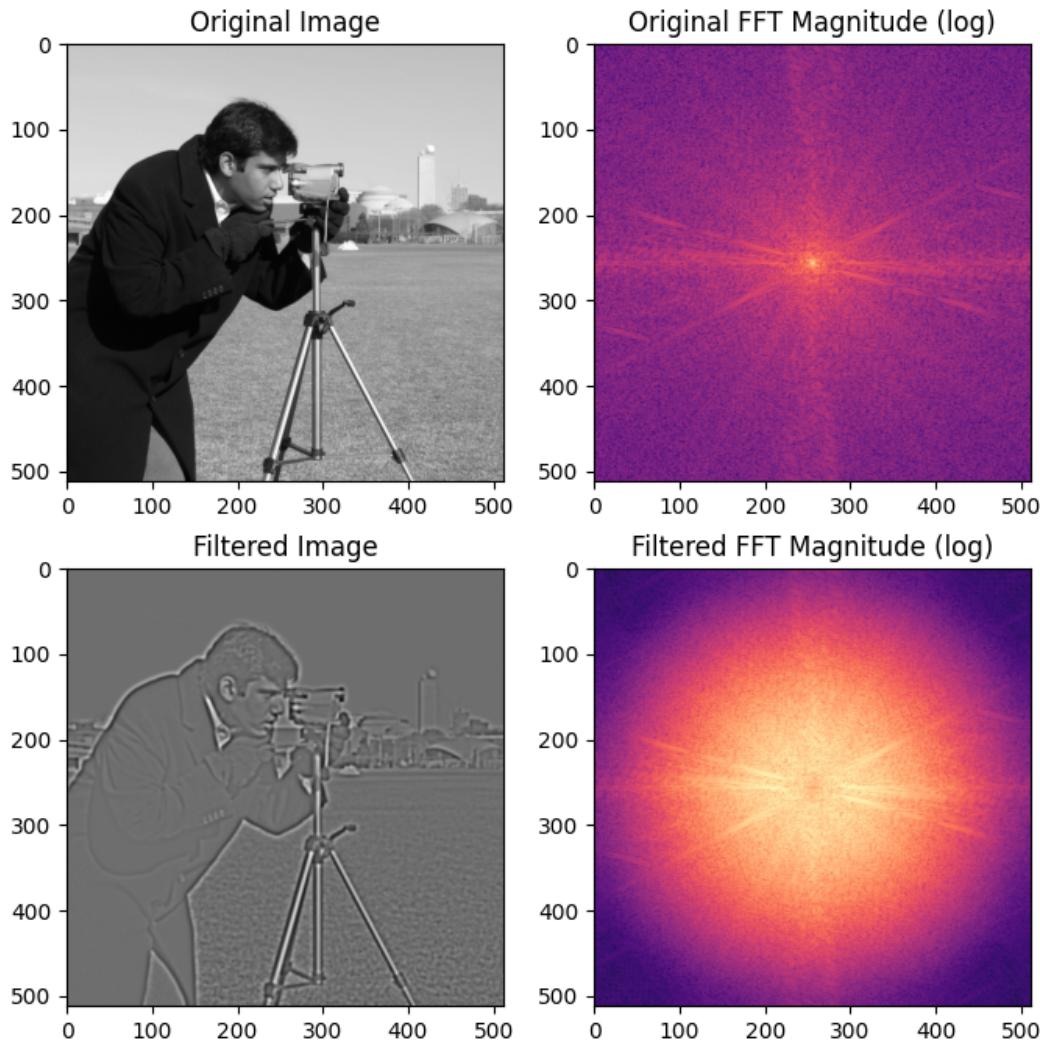
```
from skimage.data import camera

image = camera()
wimage = image * window('hann', image.shape) # window image to improve FFT
filtered_image = difference_of_gaussians(image, 1.5)
filtered_wimage = filtered_image * window('hann', image.shape)
im_f_mag = fftshift(np.abs(fft2(wimage)))
fim_f_mag = fftshift(np.abs(fft2(filtered_wimage)))
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(8, 8))
ax[0, 0].imshow(image, cmap='gray')
ax[0, 0].set_title('Original Image')
ax[0, 1].imshow(np.log(im_f_mag), cmap='magma')
ax[0, 1].set_title('Original FFT Magnitude (log)')
ax[1, 0].imshow(filtered_image, cmap='gray')
ax[1, 0].set_title('Filtered Image')
ax[1, 1].imshow(np.log(fim_f_mag), cmap='magma')
ax[1, 1].set_title('Filtered FFT Magnitude (log)')
plt.show()
```



Total running time of the script: (0 minutes 0.918 seconds)

Denoising a picture

In this example, we denoise a noisy version of a picture using the total variation, bilateral, and wavelet denoising filters.

Total variation and bilateral algorithms typically produce “posterized” images with flat domains separated by sharp edges. It is possible to change the degree of posterization by controlling the tradeoff between denoising and faithfulness to the original image.

Total variation filter

The result of this filter is an image that has a minimal total variation norm, while being as close to the initial image as possible. The total variation is the L1 norm of the gradient of the image.

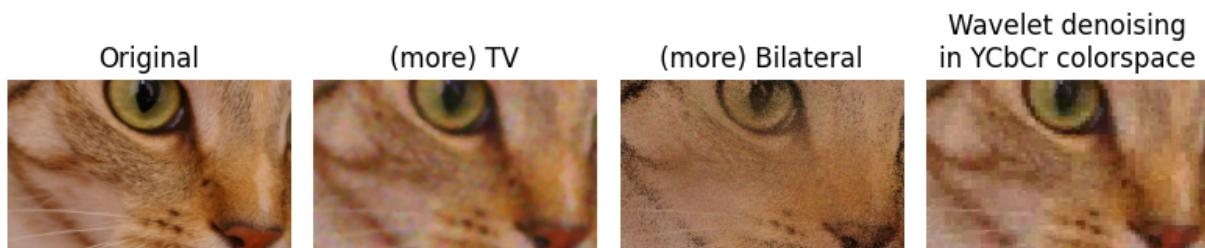
Bilateral filter

A bilateral filter is an edge-preserving and noise reducing filter. It averages pixels based on their spatial closeness and radiometric similarity.

Wavelet denoising filter

A wavelet denoising filter relies on the wavelet representation of the image. The noise is represented by small values in the wavelet domain which are set to 0.

In color images, wavelet denoising is typically done in the YCbCr color space as denoising in separate color channels may lead to more apparent noise.



Estimated Gaussian noise standard deviation = 0.14781146727224173
 Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0.. \rightarrow 255] for integers).

```
import matplotlib.pyplot as plt

from skimage.restoration import (denoise_tv_chambolle, denoise_bilateral,
                                  denoise_wavelet, estimate_sigma)
from skimage import data, img_as_float
from skimage.util import random_noise

original = img_as_float(data.chelsea()[100:250, 50:300])

sigma = 0.155
noisy = random_noise(original, var=sigma**2)

fig, ax = plt.subplots(nrows=2, ncols=4, figsize=(8, 5),
                      sharex=True, sharey=True)

plt.gray()

# Estimate the average noise standard deviation across color channels.
sigma_est = estimate_sigma(noisy, channel_axis=-1, average_sigmas=True)
# Due to clipping in random_noise, the estimate will be a bit smaller than the
# specified sigma.
print(f'Estimated Gaussian noise standard deviation = {sigma_est}')

ax[0, 0].imshow(noisy)
ax[0, 0].axis('off')
ax[0, 0].set_title('Noisy')
ax[0, 1].imshow(denoise_tv_chambolle(noisy, weight=0.1, channel_axis=-1))
ax[0, 1].axis('off')
ax[0, 1].set_title('TV')
ax[0, 2].imshow(denoise_bilateral(noisy, sigma_color=0.05, sigma_spatial=15,
                                   channel_axis=-1))
ax[0, 2].axis('off')
ax[0, 2].set_title('Bilateral')
ax[0, 3].imshow(denoise_wavelet(noisy, channel_axis=-1, rescale_sigma=True))
ax[0, 3].axis('off')
ax[0, 3].set_title('Wavelet denoising')

ax[1, 1].imshow(denoise_tv_chambolle(noisy, weight=0.2, channel_axis=-1))
ax[1, 1].axis('off')
ax[1, 1].set_title('(more) TV')
ax[1, 2].imshow(denoise_bilateral(noisy, sigma_color=0.1, sigma_spatial=15,
                                   channel_axis=-1))
ax[1, 2].axis('off')
```

(continues on next page)

(continued from previous page)

```

ax[1, 2].set_title('(more) Bilateral')
ax[1, 3].imshow(denoise_wavelet(noisy, channel_axis=-1, convert2ycbcr=True,
                                 rescale_sigma=True))
ax[1, 3].axis('off')
ax[1, 3].set_title('Wavelet denoising\nin YCbCr colorspace')
ax[1, 0].imshow(original)
ax[1, 0].axis('off')
ax[1, 0].set_title('Original')

fig.tight_layout()

plt.show()

```

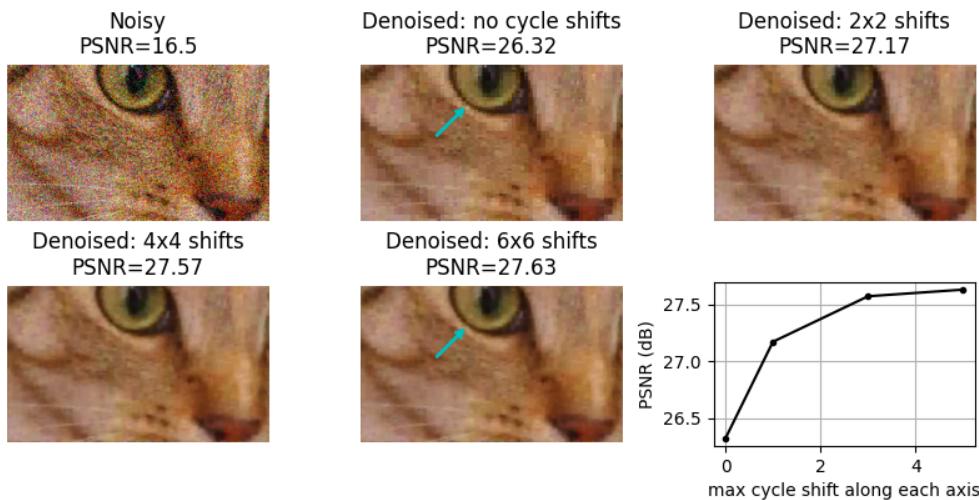
Total running time of the script: (0 minutes 9.812 seconds)

Shift-invariant wavelet denoising

The discrete wavelet transform is not [shift-invariant](#). Shift invariance can be achieved through an undecimated wavelet transform (also called stationary wavelet transform), at cost of increased redundancy (i.e. more wavelet coefficients than input image pixels). An alternative way to approximate shift-invariance in the context of image denoising with the discrete wavelet transform is to use the technique known as “cycle spinning”. This involves averaging the results of the following 3-step procedure for multiple spatial shifts, n:

1. (circularly) shift the signal by an amount, n
2. apply denoising
3. apply the inverse shift

For 2D image denoising, we demonstrate here that such cycle-spinning can provide a substantial increase in quality, with much of the gain being achieved simply by averaging shifts of only n=0 and n=1 on each axis.



Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0.. \sim 255] for integers).

Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0.. \sim 255] for integers).

Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0.. \sim 255] for integers).

(continues on next page)

(continued from previous page)

Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).

```

import matplotlib.pyplot as plt

from skimage.restoration import denoise_wavelet, cycle_spin
from skimage import data, img_as_float
from skimage.util import random_noise
from skimage.metrics import peak_signal_noise_ratio

original = img_as_float(data.chelsea()[100:250, 50:300])

sigma = 0.155
noisy = random_noise(original, var=sigma**2)

fig, ax = plt.subplots(nrows=2, ncols=3, figsize=(10, 4),
                      sharex=False, sharey=False)
ax = ax.ravel()

psnr_noisy = peak_signal_noise_ratio(original, noisy)
ax[0].imshow(noisy)
ax[0].axis('off')
ax[0].set_title(f'Noisy\nPSNR={psnr_noisy:.4g}')

# Repeat denosing with different amounts of cycle spinning. e.g.
# max_shift = 0 -> no cycle spinning
# max_shift = 1 -> shifts of (0, 1) along each axis
# max_shift = 3 -> shifts of (0, 1, 2, 3) along each axis
# etc...

denoise_kw_args = dict(channel_axis=-1, convert2ycbcr=True, wavelet='db1',
                       rescale_sigma=True)

all_psnr = []
max_shifts = [0, 1, 3, 5]
for n, s in enumerate(max_shifts):
    im_bayescs = cycle_spin(noisy, func=denoise_wavelet, max_shifts=s,
                            func_kw=denoise_kw_args, channel_axis=-1)
    ax[n+1].imshow(im_bayescs)
    ax[n+1].axis('off')
    psnr = peak_signal_noise_ratio(original, im_bayescs)
    if s == 0:
        ax[n+1].set_title(
            f'Denoised: no cycle shifts\nPSNR={psnr:.4g}')
    else:
```

(continues on next page)

(continued from previous page)

```

    ax[n+1].set_title(
        f'Denoised: {s+1}x{s+1} shifts\nPSNR={psnr:.0.4g}')
    all_psnr.append(psnr)

# plot PSNR as a function of the degree of cycle shifting
ax[5].plot(max_shifts, all_psnr, 'k.-')
ax[5].set_ylabel('PSNR (dB)')
ax[5].set_xlabel('max cycle shift along each axis')
ax[5].grid(True)
plt.subplots_adjust(wspace=0.35, hspace=0.35)

# Annotate with a cyan arrow on the 6x6 case vs. no cycle shift case to
# illustrate a region with reduced block-like artifact with cycle shifting
arrowprops = dict(arrowstyle="simple", tail_width=0.1, head_width=0.5",
                  connectionstyle="arc3",
                  color='c')
for i in [1, 4]:
    ax[i].annotate("", xy=(101, 39), xycoords='data',
                   xytext=(70, 70), textcoords='data',
                   arrowprops=arrowprops)

plt.show()

```

Total running time of the script: (0 minutes 0.768 seconds)

Phase Unwrapping

Some signals can only be observed modulo 2π , and this can also apply to two- and three dimensional images. In these cases phase unwrapping is needed to recover the underlying, unwrapped signal. In this example we will demonstrate an algorithm¹ implemented in skimage at work for such a problem. One-, two- and three dimensional images can all be unwrapped using skimage. Here we will demonstrate phase unwrapping in the two dimensional case.

```

import numpy as np
from matplotlib import pyplot as plt
from skimage import data, img_as_float, color, exposure
from skimage.restoration import unwrap_phase

# Load an image as a floating-point grayscale
image = color.rgb2gray(img_as_float(data.chelsea()))
# Scale the image to [0, 4*pi]
image = exposure.rescale_intensity(image, out_range=(0, 4 * np.pi))
# Create a phase-wrapped image in the interval [-pi, pi]
image_wrapped = np.angle(np.exp(1j * image))
# Perform phase unwrapping
image_unwrapped = unwrap_phase(image_wrapped)

fig, ax = plt.subplots(2, 2, sharex=True, sharey=True)
ax1, ax2, ax3, ax4 = ax.ravel()

```

(continues on next page)

¹ Miguel Arevalillo Herreza, David R. Burton, Michael J. Lalor, and Munther A. Gdeisat, “Fast two-dimensional phase-unwrapping algorithm based on sorting by reliability following a noncontinuous path”, Journal Applied Optics, Vol. 41, No. 35, pp. 7437, 2002

(continued from previous page)

```

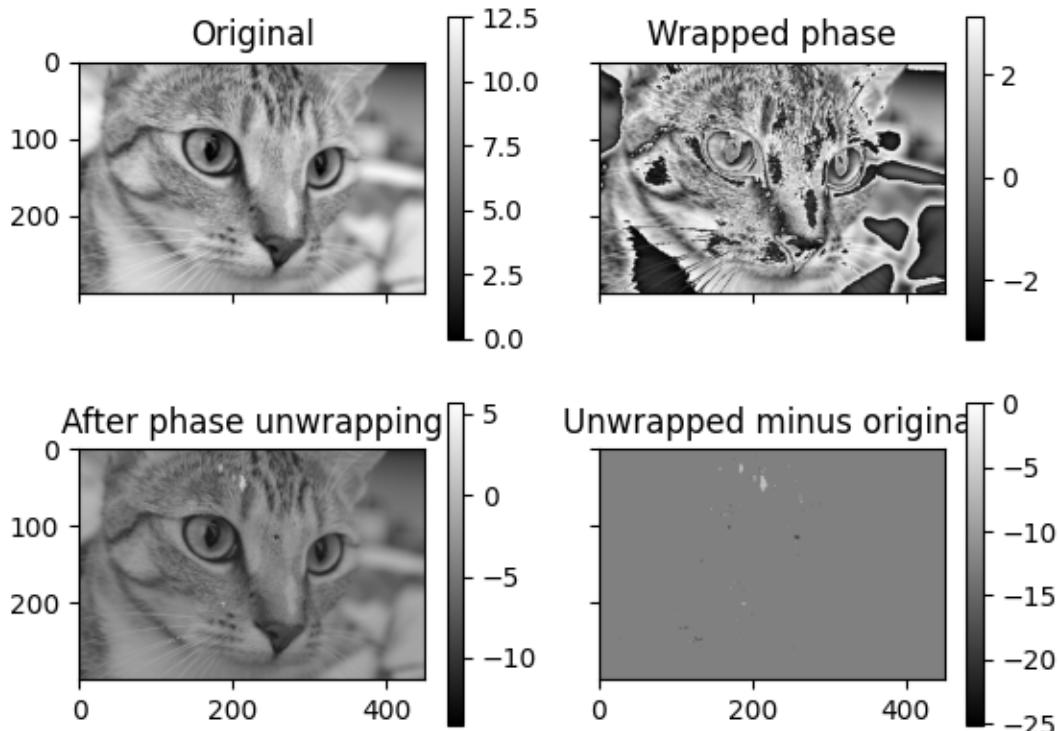
fig.colorbar(ax1.imshow(image, cmap='gray', vmin=0, vmax=4 * np.pi), ax=ax1)
ax1.set_title('Original')

fig.colorbar(ax2.imshow(image_wrapped, cmap='gray', vmin=-np.pi, vmax=np.pi),
            ax=ax2)
ax2.set_title('Wrapped phase')

fig.colorbar(ax3.imshow(image_unwrapped, cmap='gray'), ax=ax3)
ax3.set_title('After phase unwrapping')

fig.colorbar(ax4.imshow(image_unwrapped - image, cmap='gray'), ax=ax4)
ax4.set_title('Unwrapped minus original')

```



```
Text(0.5, 1.0, 'Unwrapped minus original')
```

The unwrapping procedure accepts masked arrays, and can also optionally assume cyclic boundaries to connect edges of an image. In the example below, we study a simple phase ramp which has been split in two by masking a row of the image.

```

# Create a simple ramp
image = np.ones((100, 100)) * np.linspace(0, 8 * np.pi, 100).reshape((-1, 1))
# Mask the image to split it in two horizontally

```

(continues on next page)

(continued from previous page)

```
mask = np.zeros_like(image, dtype=bool)
mask[image.shape[0] // 2, :] = True

image_wrapped = np.ma.array(np.angle(np.exp(1j * image)), mask=mask)
# Unwrap image without wrap around
image_unwrapped_no_wrap_around = unwrap_phase(image_wrapped,
                                                wrap_around=(False, False))
# Unwrap with wrap around enabled for the 0th dimension
image_unwrapped_wrap_around = unwrap_phase(image_wrapped,
                                             wrap_around=(True, False))

fig, ax = plt.subplots(2, 2)
ax1, ax2, ax3, ax4 = ax.ravel()

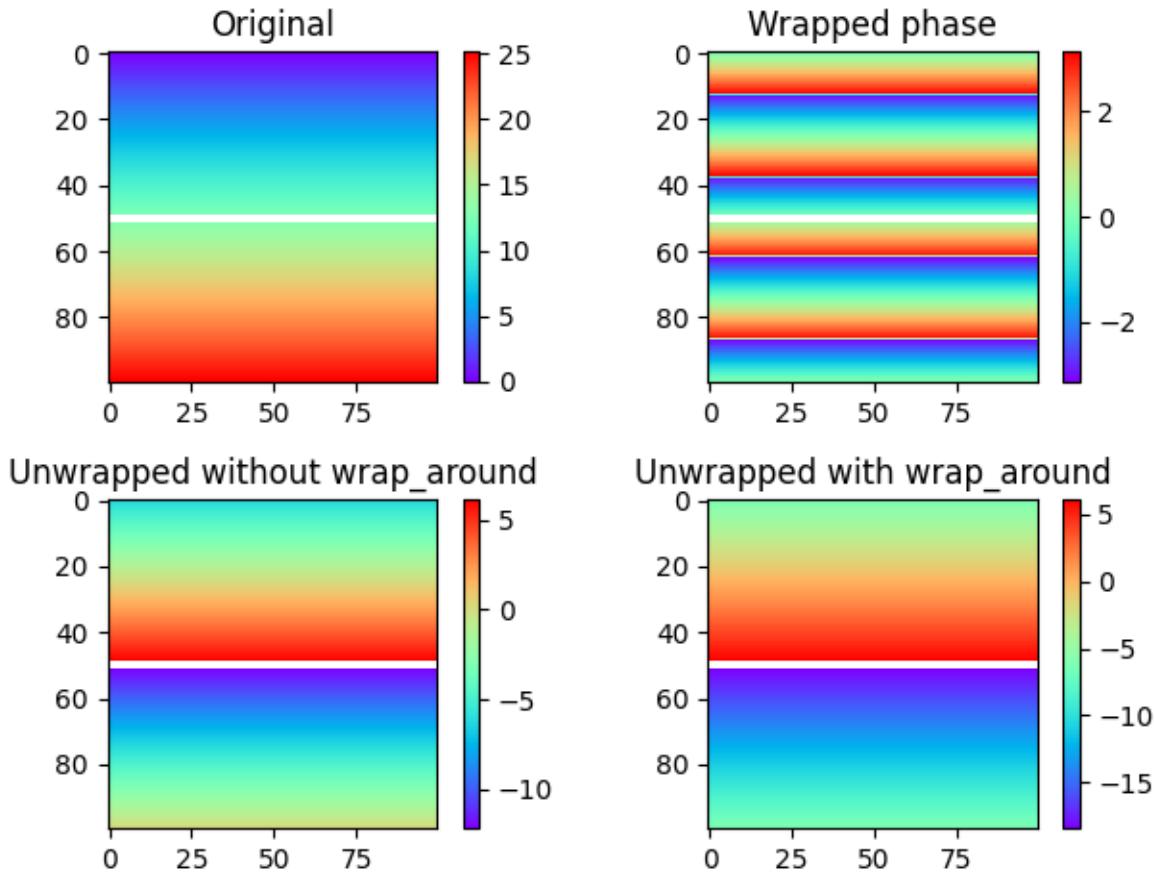
fig.colorbar(ax1.imshow(np.ma.array(image, mask=mask), cmap='rainbow'), ax=ax1)
ax1.set_title('Original')

fig.colorbar(ax2.imshow(image_wrapped, cmap='rainbow', vmin=-np.pi, vmax=np.pi),
            ax=ax2)
ax2.set_title('Wrapped phase')

fig.colorbar(ax3.imshow(image_unwrapped_no_wrap_around, cmap='rainbow'),
            ax=ax3)
ax3.set_title('Unwrapped without wrap_around')

fig.colorbar(ax4.imshow(image_unwrapped_wrap_around, cmap='rainbow'), ax=ax4)
ax4.set_title('Unwrapped with wrap_around')

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



In the figures above, the masked row can be seen as a white line across the image. The difference between the two unwrapped images in the bottom row is clear: Without unwrapping (lower left), the regions above and below the masked boundary do not interact at all, resulting in an offset between the two regions of an arbitrary integer times two π . We could just as well have unwrapped the regions as two separate images. With wrap around enabled for the vertical direction (lower right), the situation changes: Unwrapping paths are now allowed to pass from the bottom to the top of the image and vice versa, in effect providing a way to determine the offset between the two regions.

References

Total running time of the script: (0 minutes 0.976 seconds)

Non-local means denoising for preserving textures

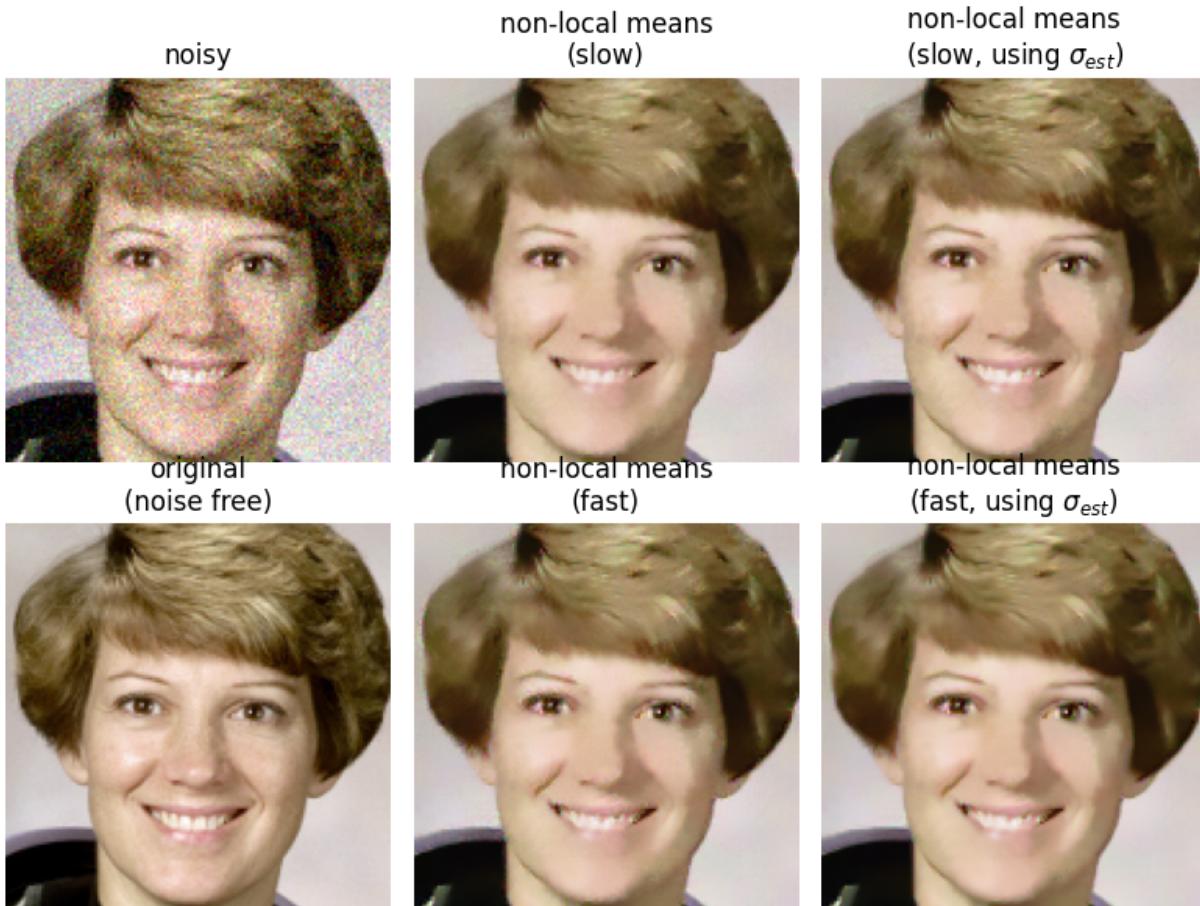
In this example, we denoise a detail of the astronaut image using the non-local means filter. The non-local means algorithm replaces the value of a pixel by an average of a selection of other pixels values: small patches centered on the other pixels are compared to the patch centered on the pixel of interest, and the average is performed only for pixels that have patches close to the current patch. As a result, this algorithm can restore well textures, that would be blurred by other denoising algorithm.

When the `fast_mode` argument is `False`, a spatial Gaussian weighting is applied to the patches when computing patch distances. When `fast_mode` is `True` a faster algorithm employing uniform spatial weighting on the patches is applied.

For either of these cases, if the noise standard deviation, `sigma`, is provided, the expected noise variance is subtracted out when computing patch distances. This can lead to a modest improvement in image quality.

The `estimate_sigma` function can provide a good starting point for setting the `h` (and optionally, `sigma`) parameters for the non-local means algorithm. `h` is a constant that controls the decay in patch weights as a function of the distance between patches. Larger `h` allows more smoothing between dissimilar patches.

In this demo, `h`, was hand-tuned to give the approximate best-case performance of each variant.



```
estimated noise standard deviation = 0.07775039152442557
PSNR (noisy) = 22.22
PSNR (slow) = 29.34
PSNR (slow, using sigma) = 29.73
PSNR (fast) = 28.94
PSNR (fast, using sigma) = 29.32
```

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

from skimage import data, img_as_float
from skimage.restoration import denoise_nl_means, estimate_sigma
from skimage.metrics import peak_signal_noise_ratio
```

(continues on next page)

(continued from previous page)

```

from skimage.util import random_noise

astro = img_as_float(data.astronaut())
astro = astro[30:180, 150:300]

sigma = 0.08
noisy = random_noise(astro, var=sigma**2)

# estimate the noise standard deviation from the noisy image
sigma_est = np.mean(estimate_sigma(noisy, channel_axis=-1))
print(f'estimated noise standard deviation = {sigma_est}')

patch_kw = dict(patch_size=5,      # 5x5 patches
                 patch_distance=6, # 13x13 search area
                 channel_axis=-1)

# slow algorithm
denoise = denoise_nl_means(noisy, h=1.15 * sigma_est, fast_mode=False,
                           **patch_kw)

# slow algorithm, sigma provided
denoise2 = denoise_nl_means(noisy, h=0.8 * sigma_est, sigma=sigma_est,
                            fast_mode=False, **patch_kw)

# fast algorithm
denoise_fast = denoise_nl_means(noisy, h=0.8 * sigma_est, fast_mode=True,
                                 **patch_kw)

# fast algorithm, sigma provided
denoise2_fast = denoise_nl_means(noisy, h=0.6 * sigma_est, sigma=sigma_est,
                                  fast_mode=True, **patch_kw)

fig, ax = plt.subplots(nrows=2, ncols=3, figsize=(8, 6),
                      sharex=True, sharey=True)

ax[0, 0].imshow(noisy)
ax[0, 0].axis('off')
ax[0, 0].set_title('noisy')

ax[0, 1].imshow(denoise)
ax[0, 1].axis('off')
ax[0, 1].set_title('non-local means\n(slow)')

ax[0, 2].imshow(denoise2)
ax[0, 2].axis('off')
ax[0, 2].set_title('non-local means\n(slow, using $\sigma_{est}$)')

ax[1, 0].imshow(astro)
ax[1, 0].axis('off')
ax[1, 0].set_title('original\n(noise free)')

ax[1, 1].imshow(denoise_fast)
ax[1, 1].axis('off')
ax[1, 1].set_title('non-local means\n(fast)')

ax[1, 2].imshow(denoise2_fast)

```

(continues on next page)

(continued from previous page)

```

ax[1, 2].axis('off')
ax[1, 2].set_title('non-local means\n(fast, using $\sigma_{est}$)')

fig.tight_layout()

# print PSNR metric for each case
psnr_noisy = peak_signal_noise_ratio(astro, noisy)
psnr = peak_signal_noise_ratio(astro, denoise)
psnr2 = peak_signal_noise_ratio(astro, denoise2)
psnr_fast = peak_signal_noise_ratio(astro, denoise_fast)
psnr2_fast = peak_signal_noise_ratio(astro, denoise2_fast)

print(f'PSNR (noisy) = {psnr_noisy:.2f}')
print(f'PSNR (slow) = {psnr:.2f}')
print(f'PSNR (slow, using sigma) = {psnr2:.2f}')
print(f'PSNR (fast) = {psnr_fast:.2f}')
print(f'PSNR (fast, using sigma) = {psnr2_fast:.2f}')

plt.show()

```

Total running time of the script: (0 minutes 1.962 seconds)

Attribute operators

Attribute operators (or connected operators)¹ is a family of contour preserving filtering operations in mathematical morphology. They can be implemented by max-trees², a compact hierarchical representation of the image.

Here, we show how to use diameter closing³⁴, which is compared to morphological closing. Comparing the two results, we observe that the difference between image and morphological closing also extracts the long line. A thin but long line cannot contain the structuring element. The diameter closing stops the filling as soon as a maximal extension is reached. The line is therefore not filled and therefore not extracted by the difference.

```

import matplotlib.pyplot as plt
from skimage.morphology import diameter_closing
from skimage import data
from skimage.morphology import closing
from skimage.morphology import square

datasets = {
    'retina': {'image': data.microaneurysms(),
               'figsize': (15, 9),
               'diameter': 10,
               'vis_factor': 3,
               'title': 'Detection of microaneurysm'},
    'page': {'image': data.page(),
}

```

(continues on next page)

¹ Salembier, P., Oliveras, A., & Garrido, L. (1998). Antextensive Connected Operators for Image and Sequence Processing. IEEE Transactions on Image Processing, 7(4), 555-570. DOI:10.1109/83.663500

² Carlinet, E., & Géraud, T. (2014). A Comparative Review of Component Tree Computation Algorithms. IEEE Transactions on Image Processing, 23(9), 3885-3895. DOI:10.1109/TIP.2014.2336551

³ Vincent L., Proc. "Grayscale area openings and closings, their efficient implementation and applications", EURASIP Workshop on Mathematical Morphology and its Applications to Signal Processing, Barcelona, Spain, pp.22-27, May 1993.

⁴ Walter, T., & Klein, J.-C. (2002). Automatic Detection of Microaneurysms in Color Fundus Images of the Human Retina by Means of the Bounding Box Closing. In A. Colosimo, P. Sirabella, A. Giuliani (Eds.), Medical Data Analysis. Lecture Notes in Computer Science, vol 2526, pp. 210-220. Springer Berlin Heidelberg. DOI:10.1007/3-540-36104-9_23

(continued from previous page)

```

'figsize': (15, 7),
'diameter': 23,
'veis_factor': 1,
'title': 'Text detection'}
}

for dataset in datasets.values():
    # image with printed letters
    image = dataset['image']
    figsize = dataset['figsize']
    diameter = dataset['diameter']

    fig, ax = plt.subplots(2, 3, figsize=figsize)
    # Original image
    ax[0, 0].imshow(image, cmap='gray', aspect='equal',
                    vmin=0, vmax=255)
    ax[0, 0].set_title('Original', fontsize=16)
    ax[0, 0].axis('off')

    ax[1, 0].imshow(image, cmap='gray', aspect='equal',
                    vmin=0, vmax=255)
    ax[1, 0].set_title('Original', fontsize=16)
    ax[1, 0].axis('off')

    # Diameter closing : we remove all dark structures with a maximal
    # extension of less than <diameter> (12 or 23). I.e. in closed_attr, all
    # local minima have at least a maximal extension of <diameter>.
    closed_attr = diameter_closing(image, diameter, connectivity=2)

    # We then calculate the difference to the original image.
    tophat_attr = closed_attr - image

    ax[0, 1].imshow(closed_attr, cmap='gray', aspect='equal',
                    vmin=0, vmax=255)
    ax[0, 1].set_title('Diameter Closing', fontsize=16)
    ax[0, 1].axis('off')

    ax[0, 2].imshow(dataset['vis_factor'] * tophat_attr, cmap='gray',
                    aspect='equal', vmin=0, vmax=255)
    ax[0, 2].set_title('Tophat (Difference)', fontsize=16)
    ax[0, 2].axis('off')

    # A morphological closing removes all dark structures that cannot
    # contain a structuring element of a certain size.
    closed = closing(image, square(diameter))

    # Again we calculate the difference to the original image.
    tophat = closed - image

    ax[1, 1].imshow(closed, cmap='gray', aspect='equal',
                    vmin=0, vmax=255)
    ax[1, 1].set_title('Morphological Closing', fontsize=16)

```

(continues on next page)

(continued from previous page)

```

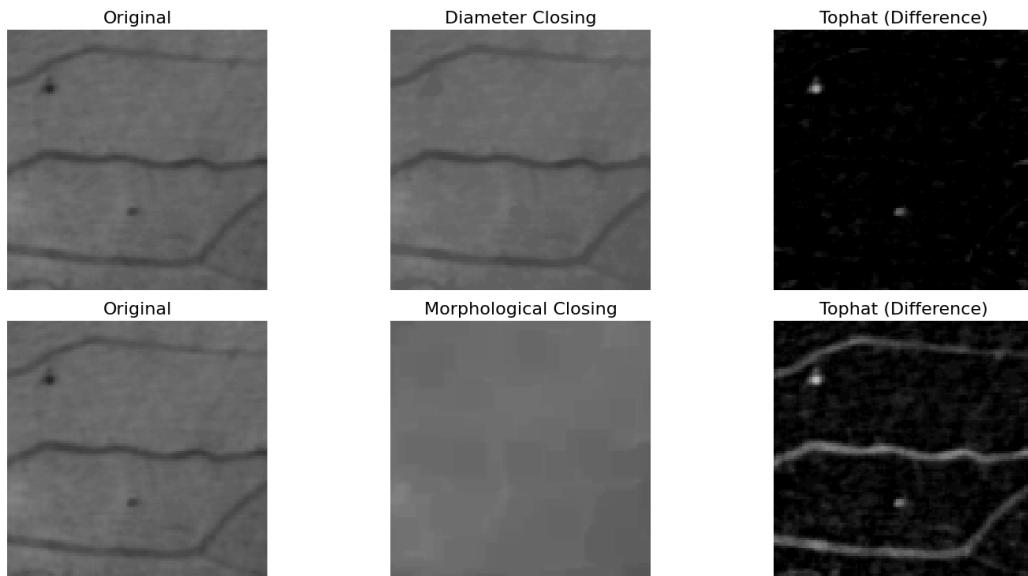
ax[1, 1].axis('off')

ax[1, 2].imshow(dataset['vis_factor'] * tophat, cmap='gray',
                aspect='equal', vmin=0, vmax=255)
ax[1, 2].set_title('Tophat (Difference)', fontsize=16)
ax[1, 2].axis('off')
fig.suptitle(dataset['title'], fontsize=18)
fig.tight_layout(rect=(0, 0, 1, 0.88))

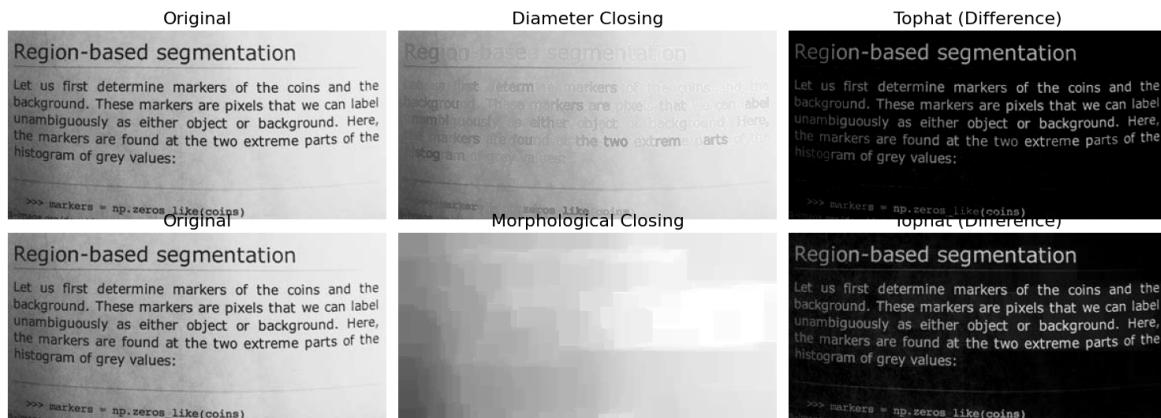
plt.show()

```

Detection of microaneurysm



Text detection



References

Total running time of the script: (0 minutes 1.829 seconds)

Wavelet denoising

Wavelet denoising relies on the wavelet representation of the image. Gaussian noise tends to be represented by small values in the wavelet domain and can be removed by setting coefficients below a given threshold to zero (hard thresholding) or shrinking all coefficients toward zero by a given amount (soft thresholding).

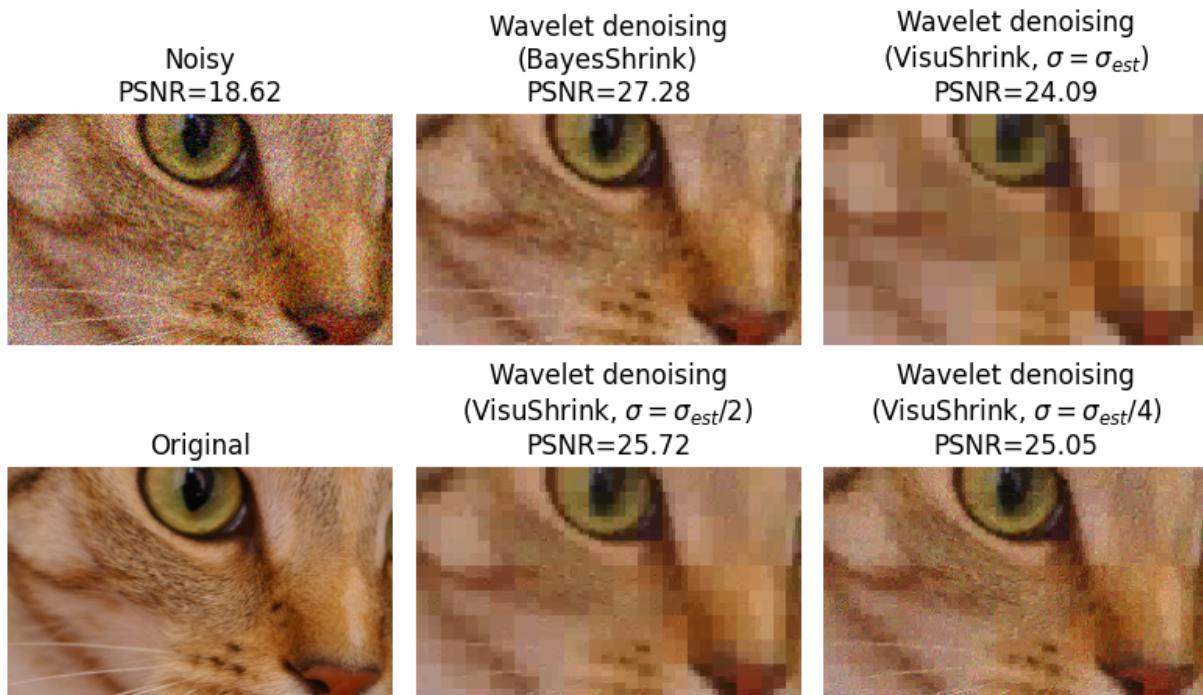
In this example, we illustrate two different methods for wavelet coefficient threshold selection: BayesShrink and VisuShrink.

VisuShrink

The VisuShrink approach employs a single, universal threshold to all wavelet detail coefficients. This threshold is designed to remove additive Gaussian noise with high probability, which tends to result in overly smooth image appearance. By specifying a sigma that is smaller than the true noise standard deviation, a more visually agreeable result can be obtained.

BayesShrink

The BayesShrink algorithm is an adaptive approach to wavelet soft thresholding where a unique threshold is estimated for each wavelet subband. This generally results in an improvement over what can be obtained with a single threshold.



Estimated Gaussian noise standard deviation = 0.11559377175191149
 Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..
 (continues on next page)

(continued from previous page)

```
↪.255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0.
↪.255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0.
↪.255] for integers).
```

```
import matplotlib.pyplot as plt

from skimage.restoration import denoise_wavelet, estimate_sigma
from skimage import data, img_as_float
from skimage.util import random_noise
from skimage.metrics import peak_signal_noise_ratio

original = img_as_float(data.chelsea()[100:250, 50:300])

sigma = 0.12
noisy = random_noise(original, var=sigma**2)

fig, ax = plt.subplots(nrows=2, ncols=3, figsize=(8, 5),
                      sharex=True, sharey=True)

plt.gray()

# Estimate the average noise standard deviation across color channels.
sigma_est = estimate_sigma(noisy, channel_axis=-1, average_sigmas=True)
# Due to clipping in random_noise, the estimate will be a bit smaller than the
# specified sigma.
print(f'Estimated Gaussian noise standard deviation = {sigma_est}')

im_bayes = denoise_wavelet(noisy, channel_axis=-1, convert2ycbcr=True,
                           method='BayesShrink', mode='soft',
                           rescale_sigma=True)
im_visushrink = denoise_wavelet(noisy, channel_axis=-1, convert2ycbcr=True,
                                 method='VisuShrink', mode='soft',
                                 sigma=sigma_est, rescale_sigma=True)

# VisuShrink is designed to eliminate noise with high probability, but this
# results in a visually over-smooth appearance. Repeat, specifying a reduction
# in the threshold by factors of 2 and 4.
im_visushrink2 = denoise_wavelet(noisy, channel_axis=-1, convert2ycbcr=True,
                                 method='VisuShrink', mode='soft',
                                 sigma=sigma_est/2, rescale_sigma=True)
im_visushrink4 = denoise_wavelet(noisy, channel_axis=-1, convert2ycbcr=True,
                                 method='VisuShrink', mode='soft',
                                 sigma=sigma_est/4, rescale_sigma=True)
```

(continues on next page)

(continued from previous page)

```

# Compute PSNR as an indication of image quality
psnr_noisy = peak_signal_noise_ratio(original, noisy)
psnr_bayes = peak_signal_noise_ratio(original, im_bayes)
psnr_visushrink = peak_signal_noise_ratio(original, im_visushrink)
psnr_visushrink2 = peak_signal_noise_ratio(original, im_visushrink2)
psnr_visushrink4 = peak_signal_noise_ratio(original, im_visushrink4)

ax[0, 0].imshow(noisy)
ax[0, 0].axis('off')
ax[0, 0].set_title(f'Noisy\nPSNR={psnr_noisy:.04g}')
ax[0, 1].imshow(im_bayes)
ax[0, 1].axis('off')
ax[0, 1].set_title(
    f'Wavelet denoising\n(BayesShrink)\nPSNR={psnr_bayes:.04g}')
ax[0, 2].imshow(im_visushrink)
ax[0, 2].axis('off')
ax[0, 2].set_title(
    f'Wavelet denoising\n(VisuShrink, $\sigma=\sigma_{est})\n'
    'PSNR=%0.4g' % psnr_visushrink)
ax[1, 0].imshow(original)
ax[1, 0].axis('off')
ax[1, 0].set_title('Original')
ax[1, 1].imshow(im_visushrink2)
ax[1, 1].axis('off')
ax[1, 1].set_title(
    f'Wavelet denoising\n(VisuShrink, $\sigma=\sigma_{est}/2$)\n'
    'PSNR=%0.4g' % psnr_visushrink2)
ax[1, 2].imshow(im_visushrink4)
ax[1, 2].axis('off')
ax[1, 2].set_title(
    f'Wavelet denoising\n(VisuShrink, $\sigma=\sigma_{est}/4$)\n'
    'PSNR=%0.4g' % psnr_visushrink4)
fig.tight_layout()

plt.show()

```

Total running time of the script: (0 minutes 0.504 seconds)

Butterworth Filters

The Butterworth filter is implemented in the frequency domain and is designed to have no passband or stopband ripple. It can be used in either a lowpass or highpass variant. The `cutoff_frequency_ratio` parameter is used to set the cutoff frequency as a fraction of the sampling frequency. Given that the Nyquist frequency is half the sampling frequency, this means that this parameter should be a positive floating point value < 0.5. The `order` of the filter can be adjusted to control the transition width, with higher values leading to a sharper transition between the passband and stopband.

Butterworth filtering example

Here we define a `get_filtered` helper function to repeat lowpass and highpass filtering at a specified series of cutoff frequencies.

```
import matplotlib.pyplot as plt

from skimage import data, filters

image = data.camera()

# cutoff frequencies as a fraction of the maximum frequency
cutoffs = [.02, .08, .16]

def get_filtered(image, cutoffs, squared_butterworth=True, order=3.0, npad=0):
    """Lowpass and highpass butterworth filtering at all specified cutoffs.

    Parameters
    -----
    image : ndarray
        The image to be filtered.
    cutoffs : sequence of int
        Both lowpass and highpass filtering will be performed for each cutoff
        frequency in `cutoffs`.
    squared_butterworth : bool, optional
        Whether the traditional Butterworth filter or its square is used.
    order : float, optional
        The order of the Butterworth filter

    Returns
    -----
    lowpass_filtered : list of ndarray
        List of images lowpass filtered at the frequencies in `cutoffs`.
    highpass_filtered : list of ndarray
        List of images highpass filtered at the frequencies in `cutoffs`.
    """

    lowpass_filtered = []
    highpass_filtered = []
    for cutoff in cutoffs:
        lowpass_filtered.append(
            filters.butterworth(
                image,
                cutoff_frequency_ratio=cutoff,
                order=order,
                high_pass=False,
                squared_butterworth=squared_butterworth,
                npad=npad,
            )
        )
        highpass_filtered.append(
            filters.butterworth(

```

(continues on next page)

(continued from previous page)

```

        image,
        cutoff_frequency_ratio=cutoff,
        order=order,
        high_pass=True,
        squared_butterworth=squared_butterworth,
        npad=npad,
    )
)
)
return lowpass_filtered, highpass_filtered

def plot_filtered(lowpass_filtered, highpass_filtered, cutoffs):
    """Generate plots for paired lists of lowpass and highpass images."""
    fig, axes = plt.subplots(2, 1 + len(cutoffs), figsize=(12, 8))
    fontdict = dict(fontsize=14, fontweight='bold')

    axes[0, 0].imshow(image, cmap='gray')
    axes[0, 0].set_title('original', fontdict=fontdict)
    axes[1, 0].set_axis_off()

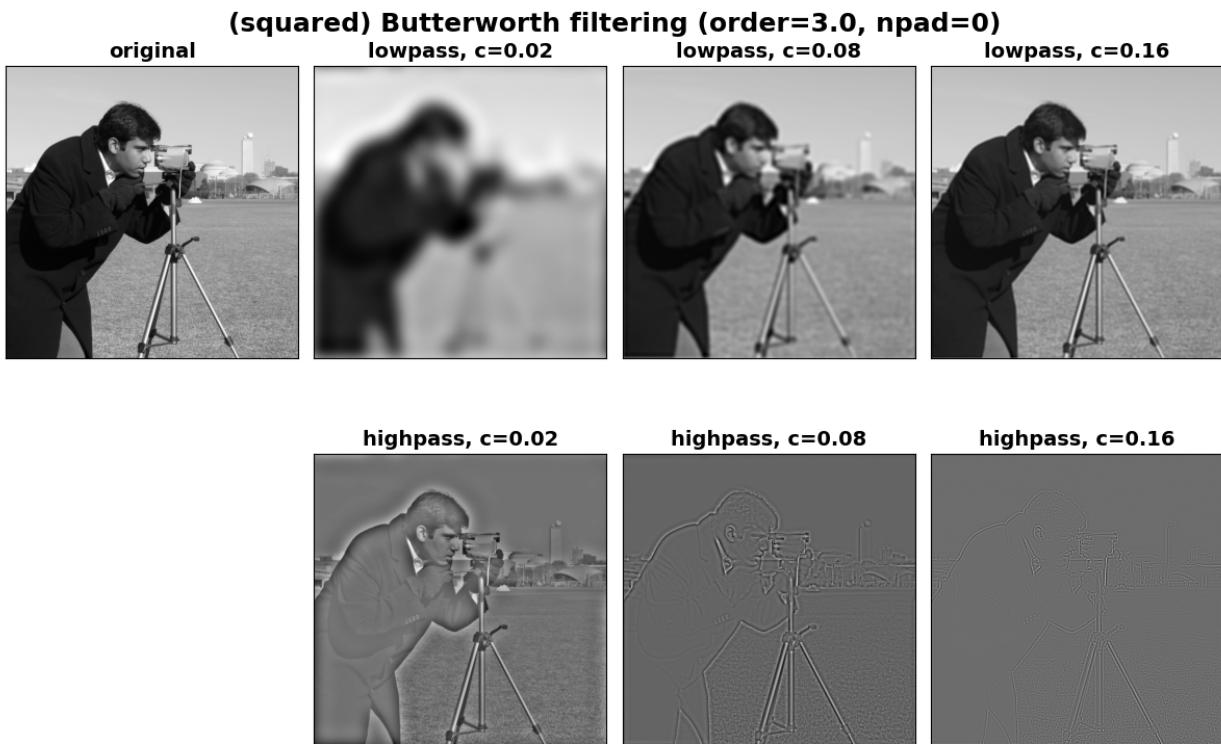
    for i, c in enumerate(cutoffs):
        axes[0, i + 1].imshow(lowpass_filtered[i], cmap='gray')
        axes[0, i + 1].set_title(f'lowpass, c={c}', fontdict=fontdict)
        axes[1, i + 1].imshow(highpass_filtered[i], cmap='gray')
        axes[1, i + 1].set_title(f'highpass, c={c}', fontdict=fontdict)

    for ax in axes.ravel():
        ax.set_xticks([])
        ax.set_yticks([])
    plt.tight_layout()
    return fig, axes

# Perform filtering with the (squared) Butterworth filter at a range of
# cutoffs.
lowpasses, highpasses = get_filtered(image, c cutoffs, squared_butterworth=True)

fig, axes = plot_filtered(lowpasses, highpasses, c cutoffs)
titledict = dict(fontsize=18, fontweight='bold')
fig.text(0.5, 0.95, '(squared) Butterworth filtering (order=3.0, npad=0)',
         fontdict=titledict, horizontalalignment='center')

```



```
Text(0.5, 0.95, '(squared) Butterworth filtering (order=3.0, npad=0)')
```

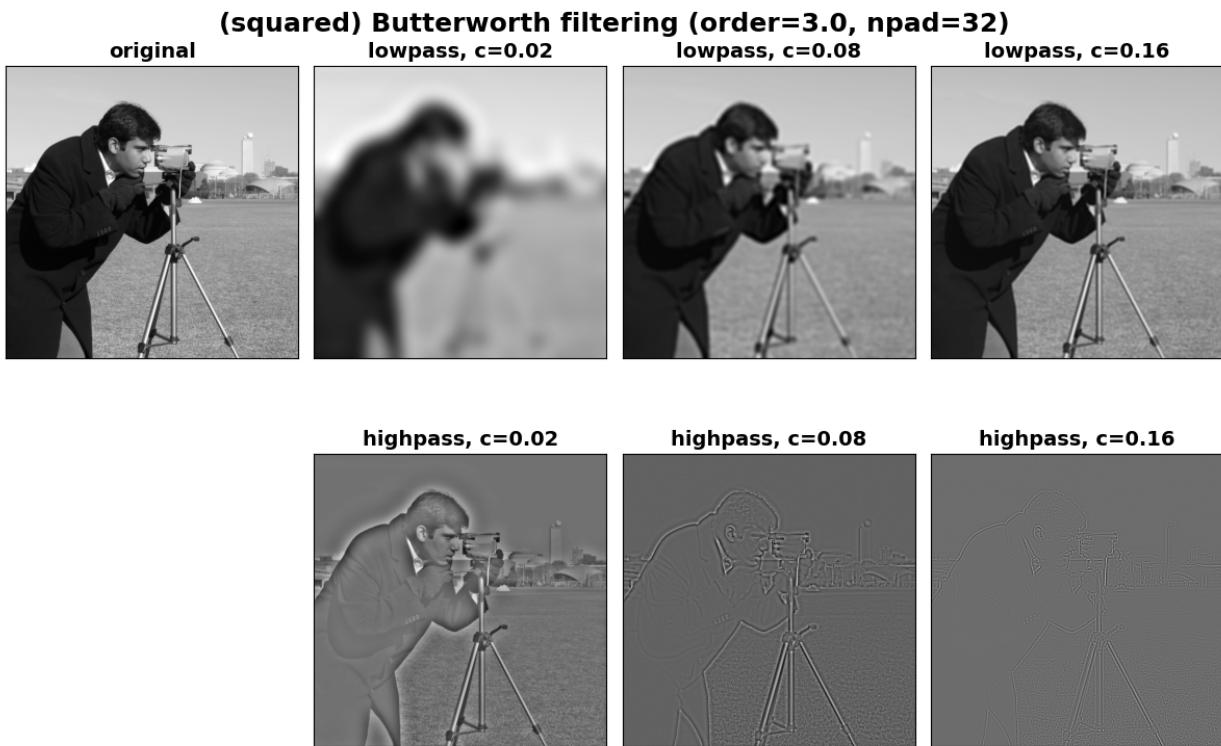
Avoiding boundary artifacts

It can be seen in the images above that there are artifacts near the edge of the images (particularly for the smaller cutoff values). This is due to the periodic nature of the DFT and can be reduced by applying some amount of padding to the edges prior to filtering so that there are not sharp edges in the periodic extension of the image. This can be done via the `npad` argument to `butterworth`.

Note that with padding, the undesired shading at the image edges is substantially reduced.

```
lowpasses, highpasses = get_filtered(image, cutoffs, squared_butterworth=True,
                                      npad=32)

fig, axes = plot_filtered(lowpasses, highpasses, cutoffs)
fig.text(0.5, 0.95, '(squared) Butterworth filtering (order=3.0, npad=32)',
         fontdict=titledict, horizontalalignment='center')
```



```
Text(0.5, 0.95, '(squared) Butterworth filtering (order=3.0, npad=32)')
```

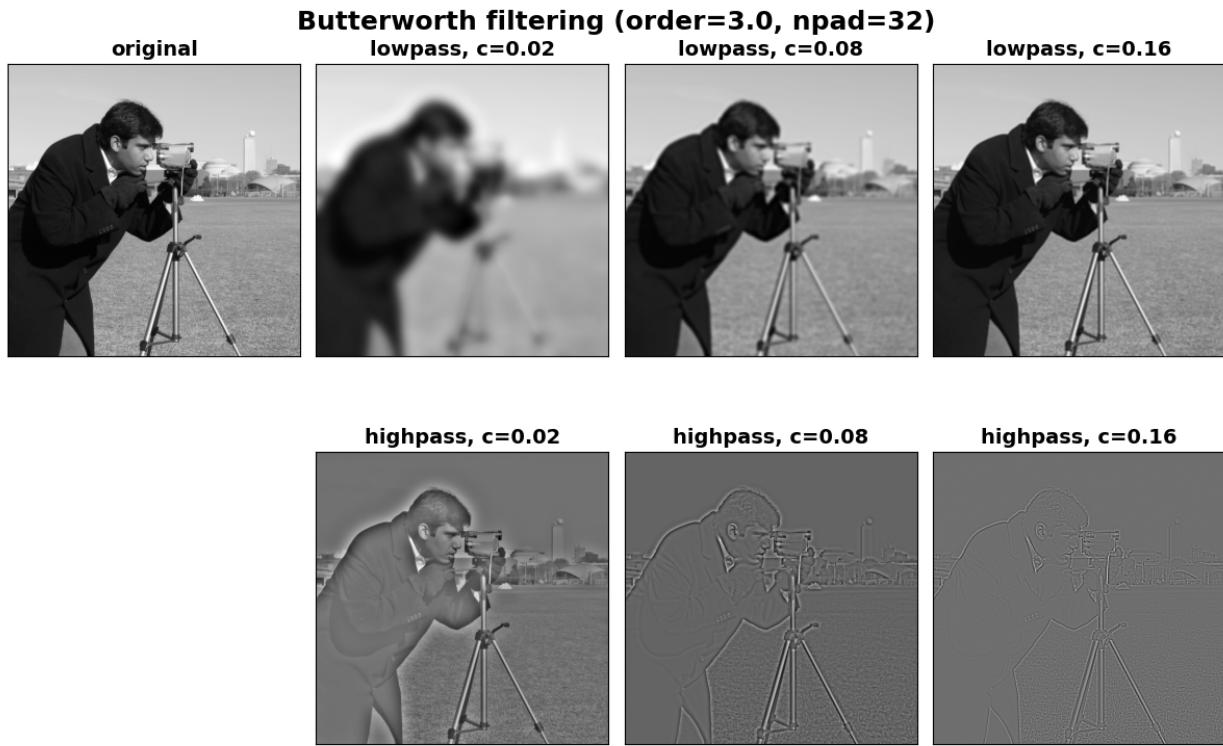
True Butterworth filter

To use the traditional signal processing definition of the Butterworth filter, set `squared_butterworth=False`. This variant has an amplitude profile in the frequency domain that is the square root of the default case. This causes the transition from the passband to the stopband to be more gradual at any given `order`. This can be seen in the following images which appear a bit sharper in the lowpass case than their squared Butterworth counterparts above.

```
lowpasses, highpasses = get_filtered(image, cutoffs, squared_butterworth=False,
                                      npad=32)

fig, axes = plot_filtered(lowpasses, highpasses, cutoffs)
fig.text(0.5, 0.95, 'Butterworth filtering (order=3.0, npad=32)',
         fontdict=titledict, horizontalalignment='center')

plt.show()
```



Total running time of the script: (0 minutes 1.753 seconds)

Full tutorial on calibrating Denoisers Using J-Invariance

In this example, we show how to find an optimally calibrated version of any denoising algorithm.

The calibration method is based on the *noise2self* algorithm of¹.

See also:

A simple example of the method is given in *Calibrating Denoisers Using J-Invariance*.

Calibrating a wavelet denoiser

```
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import gridspec

from skimage.data import chelsea, hubble_deep_field
from skimage.metrics import mean_squared_error as mse
from skimage.metrics import peak_signal_noise_ratio as psnr
from skimage.restoration import (calibrate_denoiser,
                                  denoise_wavelet,
                                  denoise_tv_chambolle, denoise_nl_means,
                                  estimate_sigma)
from skimage.util import img_as_float, random_noise
from skimage.color import rgb2gray
from functools import partial
```

(continues on next page)

¹ J. Batson & L. Royer. Noise2Self: Blind Denoising by Self-Supervision, International Conference on Machine Learning, p. 524-533 (2019).

(continued from previous page)

```

_denoise_wavelet = partial(denoise_wavelet, rescale_sigma=True)

image = img_as_float(chelsea())
sigma = 0.2
noisy = random_noise(image, var=sigma ** 2)

# Parameters to test when calibrating the denoising algorithm
parameter_ranges = {'sigma': np.arange(0.1, 0.3, 0.02),
                     'wavelet': ['db1', 'db2'],
                     'convert2ycbcr': [True, False],
                     'channel_axis': [-1]}

# Denoised image using default parameters of `denoise_wavelet`
default_output = denoise_wavelet(noisy, channel_axis=-1, rescale_sigma=True)

# Calibrate denoiser
calibrated_denoiser = calibrate_denoiser(noisy,
                                           _denoise_wavelet,
                                           denoise_parameters=parameter_ranges
                                           )

# Denoised image using calibrated denoiser
calibrated_output = calibrated_denoiser(noisy)

fig, axes = plt.subplots(1, 3, sharex=True, sharey=True, figsize=(15, 5))

for ax, img, title in zip(axes,
                           [noisy, default_output, calibrated_output],
                           ['Noisy Image', 'Denoised (Default)',
                            'Denoised (Calibrated)']):
    ax.imshow(img)
    ax.set_title(title)
    ax.set_yticks([])
    ax.set_xticks([])
```



Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0.. \rightarrow 255] for integers).

The Self-Supervised Loss and J-Invariance

The key to this calibration method is the notion of J-invariance. A denoising function is J-invariant if the prediction it makes for each pixel does not depend on the value of that pixel in the original image. The prediction for each pixel may instead use all the relevant information contained in the rest of the image, which is typically quite significant. Any function can be converted into a J-invariant one using a simple masking procedure, as described in [1].

The pixel-wise error of a J-invariant denoiser is uncorrelated to the noise, so long as the noise in each pixel is independent. Consequently, the average difference between the denoised image and the noisy image, the *self-supervised loss*, is the same as the difference between the denoised image and the original clean image, the *ground-truth loss* (up to a constant).

This means that the best J-invariant denoiser for a given image can be found using the noisy data alone, by selecting the denoiser minimizing the self-supervised loss. Below, we demonstrate this for a family of wavelet denoisers with varying *sigma* parameter. The self-supervised loss (solid blue line) and the ground-truth loss (dashed blue line) have the same shape and the same minimizer.

```
from skimage.restoration import denoise_invariant

sigma_range = np.arange(sigma/2, 1.5*sigma, 0.025)

parameters_tested = [{'sigma': sigma, 'convert2ycbcr': True, 'wavelet': 'db2',
                      'channel_axis': -1}
                      for sigma in sigma_range]

denoised_invariant = [denoise_invariant(noisy, _denoise_wavelet,
                                         denoiser_kwarg=params)
                      for params in parameters_tested]

self_supervised_loss = [mse(img, noisy) for img in denoised_invariant]
ground_truth_loss = [mse(img, image) for img in denoised_invariant]

opt_idx = np.argmin(self_supervised_loss)
plot_idx = [0, opt_idx, len(sigma_range) - 1]

def get_inset(x):
    return x[25:225, 100:300]

plt.figure(figsize=(10, 12))

gs = gridspec.GridSpec(3, 3)
ax1 = plt.subplot(gs[0, :])
ax2 = plt.subplot(gs[1, :])
ax_image = [plt.subplot(gs[2, i]) for i in range(3)]

ax1.plot(sigma_range, self_supervised_loss, color='C0',
          label='Self-Supervised Loss')
ax1.scatter(sigma_range[opt_idx], self_supervised_loss[opt_idx] + 0.0003,
            marker='v', color='red', label='optimal sigma')

ax1.set_ylabel('MSE')
ax1.set_xticks([])
ax1.legend()
ax1.set_title('Self-Supervised Loss')
```

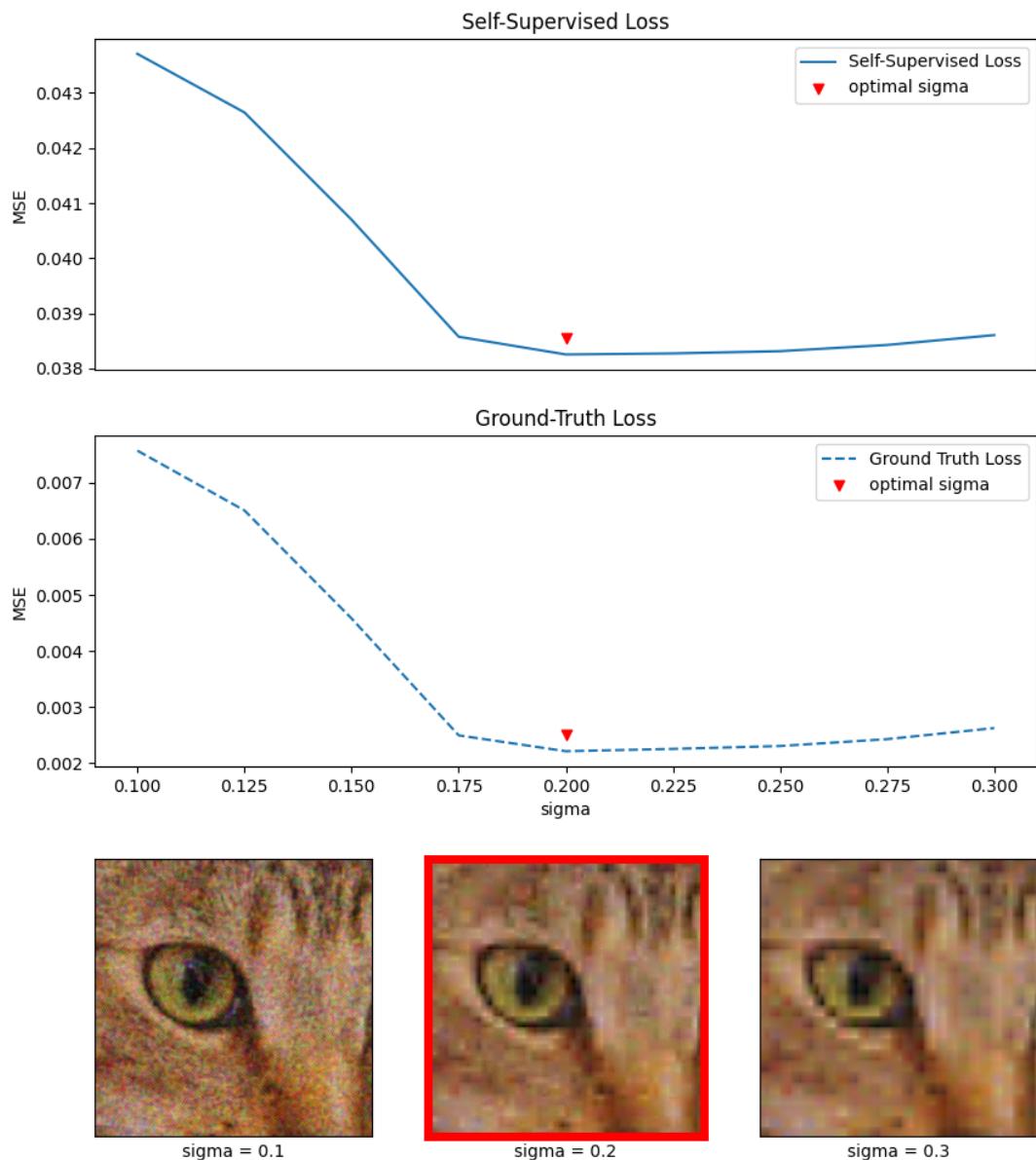
(continues on next page)

(continued from previous page)

```
ax2.plot(sigma_range, ground_truth_loss, color='C0', linestyle='--',
         label='Ground Truth Loss')
ax2.scatter(sigma_range[opt_idx], ground_truth_loss[opt_idx] + 0.0003,
            marker='v', color='red', label='optimal sigma')
ax2.set_ylabel('MSE')
ax2.legend()
ax2.set_xlabel('sigma')
ax2.set_title('Ground-Truth Loss')

for i in range(3):
    ax = ax_image[i]
    ax.set_xticks([])
    ax.set_yticks([])
    ax.imshow(get_inset(denoisedInvariant[plot_idx[i]]))
    ax.set_xlabel('sigma = ' + str(np.round(sigma_range[plot_idx[i]], 2)))

for spine in ax_image[1].spines.values():
    spine.set_edgecolor('red')
    spine.set_linewidth(5)
```



Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0.. \sim 255] for integers).

Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0.. \sim 255] for integers).

Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0.. \sim 255] for integers).

(continues on next page)

(continued from previous page)

`↪.255] for integers).`

Conversion to J-invariance

The function `_invariant_denoise` acts as a J-invariant version of a given denoiser. It works by masking a fraction of the pixels, interpolating them, running the original denoiser, and extracting the values returned in the masked pixels. Iterating over the image results in a fully J-invariant output.

For any given set of parameters, the J-invariant version of a denoiser is different from the original denoiser, but it is not necessarily better or worse. In the plot below, we see that, for the test image of a cat, the J-invariant version of a wavelet denoiser is significantly better than the original at small values of variance-reduction `sigma` and imperceptibly worse at larger values.

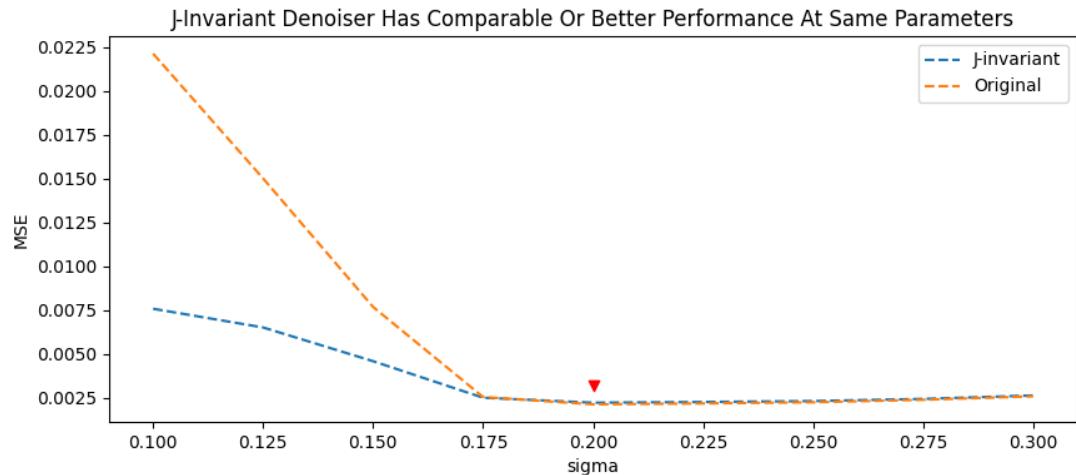
```
parameters_tested = [ {'sigma': sigma, 'convert2ycbcr': True,
                      'wavelet': 'db2', 'channel_axis': -1}
                      for sigma in sigma_range]

denoised_original = [_denoise_wavelet(noisy, **params)
                      for params in parameters_tested]

ground_truth_loss_invariant = [mse(img, image) for img in denoised_invariant]
ground_truth_loss_original = [mse(img, image) for img in denoised_original]

fig, ax = plt.subplots(figsize=(10, 4))

ax.plot(sigma_range, ground_truth_loss_invariant, color='C0', linestyle='--',
        label='J-invariant')
ax.plot(sigma_range, ground_truth_loss_original, color='C1', linestyle='--',
        label='Original')
ax.scatter(sigma_range[opt_idx], ground_truth_loss[opt_idx] + 0.001,
           marker='v', color='red')
ax.legend()
ax.set_title(
    'J-Invariant Denoiser Has Comparable Or '
    'Better Performance At Same Parameters')
)
ax.set_ylabel('MSE')
ax.set_xlabel('sigma')
```



```
Text(0.5, 14.72222222222216, 'sigma')
```

Comparing Different Classes of Denoiser

The self-supervised loss can be used to compare different classes of denoiser in addition to choosing parameters for a single class. This allows the user to, in an unbiased way, choose the best parameters for the best class of denoiser for a given image.

Below, we show this for an image of the hubble deep field with significant speckle noise added. In this case, the J-invariant calibrated denoiser is better than the default denoiser in each of three families of denoisers – Non-local means, wavelet, and TV norm. Additionally, the self-supervised loss shows that the TV norm denoiser is the best for this noisy image.

```
image = rgb2gray(img_as_float(hubble_deep_field()[100:250, 50:300]))

sigma = 0.4
noisy = random_noise(image, mode='speckle', var=sigma ** 2)

parameter_ranges_tv = {'weight': np.arange(0.01, 0.3, 0.02)}
_, (parameters_tested_tv, losses_tv) = calibrate_denoiser(
    noisy,
    denoise_tv_chambolle,
    denoise_parameters=parameter_ranges_tv,
    extra_output=True)
print(f'Minimum self-supervised loss TV: {np.min(losses_tv):.4f}')

best_parameters_tv = parameters_tested_tv[np.argmin(losses_tv)]
denoised_calibrated_tv = denoise_invariant(noisy, denoise_tv_chambolle,
                                             denoiser_kwarg=best_parameters_tv)
denoised_default_tv = denoise_tv_chambolle(noisy, **best_parameters_tv)

psnr_calibrated_tv = psnr(image, denoised_calibrated_tv)
psnr_default_tv = psnr(image, denoised_default_tv)

parameter_ranges_wavelet = {'sigma': np.arange(0.01, 0.3, 0.03)}
_, (parameters_tested_wavelet, losses_wavelet) = calibrate_denoiser(
```

(continues on next page)

(continued from previous page)

```

        noisy,
        _denoise_wavelet,
        parameter_ranges_wavelet,
        extra_output=True)
print(f'Minimum self-supervised loss wavelet: {np.min(losses_wavelet):.4f}')

best_parameters_wavelet = parameters_tested_wavelet[np.argmin(losses_wavelet)]
denoised_calibrated_wavelet = denoise_invariant(
    noisy, _denoise_wavelet,
    denoiser_kwags=best_parameters_wavelet)
denoised_default_wavelet = _denoise_wavelet(noisy, **best_parameters_wavelet)

psnr_calibrated_wavelet = psnr(image, denoised_calibrated_wavelet)
psnr_default_wavelet = psnr(image, denoised_default_wavelet)

sigma_est = estimate_sigma(noisy)

parameter_ranges_nl = {'sigma': np.arange(0.6, 1.4, 0.2) * sigma_est,
                      'h': np.arange(0.6, 1.2, 0.2) * sigma_est}

parameter_ranges_nl = {'sigma': np.arange(0.01, 0.3, 0.03)}
_, (parameters_tested_nl, losses_nl) = calibrate_denoiser(noisy,
                                                          denoise_nl_means,
                                                          parameter_ranges_nl,
                                                          extra_output=True)
print(f'Minimum self-supervised loss NL means: {np.min(losses_nl):.4f}')

best_parameters_nl = parameters_tested_nl[np.argmin(losses_nl)]
denoised_calibrated_nl = denoise_invariant(noisy, denoise_nl_means,
                                             denoiser_kwags=best_parameters_nl)
denoised_default_nl = denoise_nl_means(noisy, **best_parameters_nl)

psnr_calibrated_nl = psnr(image, denoised_calibrated_nl)
psnr_default_nl = psnr(image, denoised_default_nl)

print('          PSNR')
print(f'NL means (Default) : {psnr_default_nl:.1f}')
print(f'NL means (Calibrated): {psnr_calibrated_nl:.1f}')
print(f'Wavelet (Default) : {psnr_default_wavelet:.1f}')
print(f'Wavelet (Calibrated): {psnr_calibrated_wavelet:.1f}')
print(f'TV norm (Default) : {psnr_default_tv:.1f}')
print(f'TV norm (Calibrated): {psnr_calibrated_tv:.1f}')

plt.subplots(figsize=(10, 12))
plt.imshow(noisy, cmap='Greys_r')
plt.xticks([])
plt.yticks([])
plt.title('Noisy Image')

def get_inset(x):
    return x[0:100, -140:]

```

(continues on next page)

(continued from previous page)

```
fig, axes = plt.subplots(ncols=3, nrows=2, figsize=(15, 8))

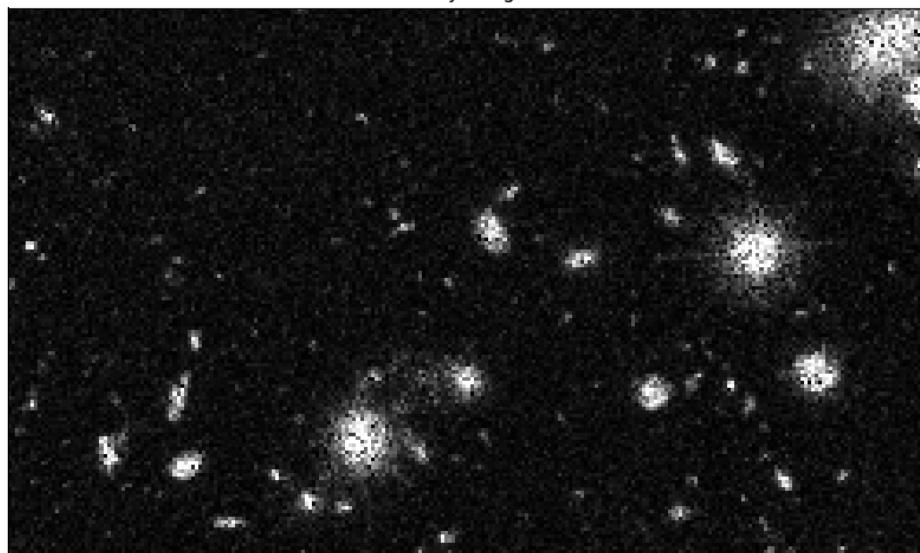
for ax in axes.ravel():
    ax.set_xticks([])
    ax.set_yticks([])

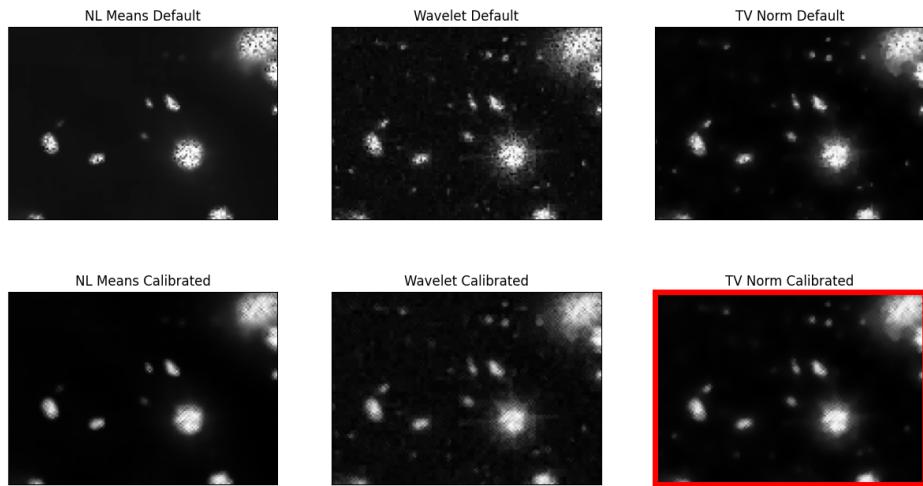
axes[0, 0].imshow(get_inset(denoised_default_nl), cmap='Greys_r')
axes[0, 0].set_title('NL Means Default')
axes[1, 0].imshow(get_inset(denoised_calibrated_nl), cmap='Greys_r')
axes[1, 0].set_title('NL Means Calibrated')
axes[0, 1].imshow(get_inset(denoised_default_wavelet), cmap='Greys_r')
axes[0, 1].set_title('Wavelet Default')
axes[1, 1].imshow(get_inset(denoised_calibrated_wavelet), cmap='Greys_r')
axes[1, 1].set_title('Wavelet Calibrated')
axes[0, 2].imshow(get_inset(denoised_default_tv), cmap='Greys_r')
axes[0, 2].set_title('TV Norm Default')
axes[1, 2].imshow(get_inset(denoised_calibrated_tv), cmap='Greys_r')
axes[1, 2].set_title('TV Norm Calibrated')

for spine in axes[1, 2].spines.values():
    spine.set_edgecolor('red')
    spine.set_linewidth(5)

plt.show()
```

Noisy Image





```
•
Minimum self-supervised loss TV: 0.0037
Minimum self-supervised loss wavelet: 0.0038
Minimum self-supervised loss NL means: 0.0043
    PSNR
NL means (Default) : 25.6
NL means (Calibrated): 27.2
Wavelet (Default) : 26.8
Wavelet (Calibrated): 28.8
TV norm (Default) : 28.8
TV norm (Calibrated): 29.1
```

Total running time of the script: (0 minutes 11.871 seconds)

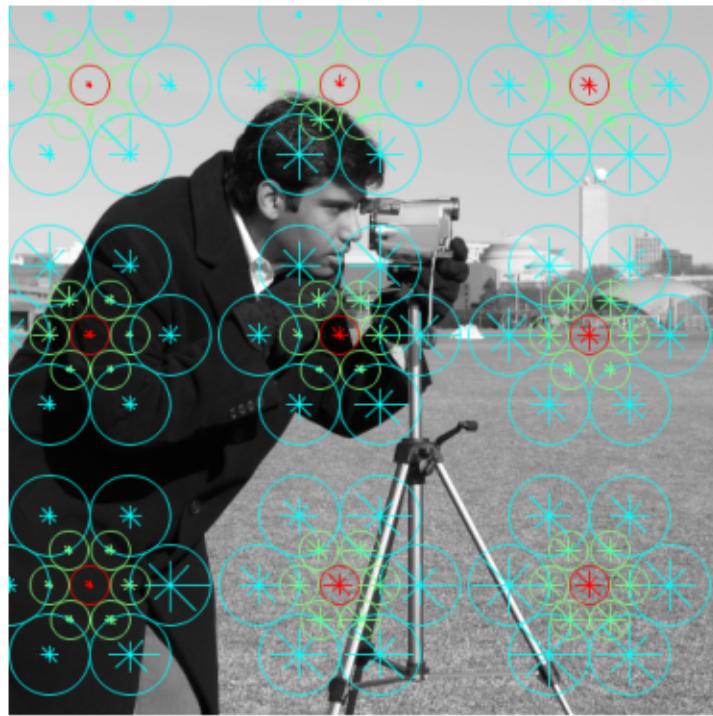
Detection of features and objects

Dense DAISY feature description

The DAISY local image descriptor is based on gradient orientation histograms similar to the SIFT descriptor. It is formulated in a way that allows for fast dense extraction which is useful for e.g. bag-of-features image representations.

In this example a limited number of DAISY descriptors are extracted at a large scale for illustrative purposes.

9 DAISY descriptors extracted:



```
from skimage.feature import daisy
from skimage import data
import matplotlib.pyplot as plt

img = data.camera()
descs, descs_img = daisy(img, step=180, radius=58, rings=2, histograms=6,
                         orientations=8, visualize=True)

fig, ax = plt.subplots()
ax.axis("off")
ax.imshow(descs_img)
descs_num = descs.shape[0] * descs.shape[1]
ax.set_title(f"{descs_num} DAISY descriptors extracted:")
plt.show()
```

Total running time of the script: (0 minutes 1.442 seconds)

Histogram of Oriented Gradients

The Histogram of Oriented Gradient (HOG) feature descriptor is popular for object detection¹.

In the following example, we compute the HOG descriptor and display a visualisation.

Algorithm overview

Compute a Histogram of Oriented Gradients (HOG) by

1. (optional) global image normalisation
2. computing the gradient image in x and y
3. computing gradient histograms
4. normalising across blocks
5. flattening into a feature vector

The first stage applies an optional global image normalisation equalisation that is designed to reduce the influence of illumination effects. In practice we use gamma (power law) compression, either computing the square root or the log of each color channel. Image texture strength is typically proportional to the local surface illumination so this compression helps to reduce the effects of local shadowing and illumination variations.

The second stage computes first order image gradients. These capture contour, silhouette and some texture information, while providing further resistance to illumination variations. The locally dominant color channel is used, which provides color invariance to a large extent. Variant methods may also include second order image derivatives, which act as primitive bar detectors - a useful feature for capturing, e.g. bar like structures in bicycles and limbs in humans.

The third stage aims to produce an encoding that is sensitive to local image content while remaining resistant to small changes in pose or appearance. The adopted method pools gradient orientation information locally in the same way as the SIFT² feature. The image window is divided into small spatial regions, called “cells”. For each cell we accumulate a local 1-D histogram of gradient or edge orientations over all the pixels in the cell. This combined cell-level 1-D histogram forms the basic “orientation histogram” representation. Each orientation histogram divides the gradient angle range into a fixed number of predetermined bins. The gradient magnitudes of the pixels in the cell are used to vote into the orientation histogram.

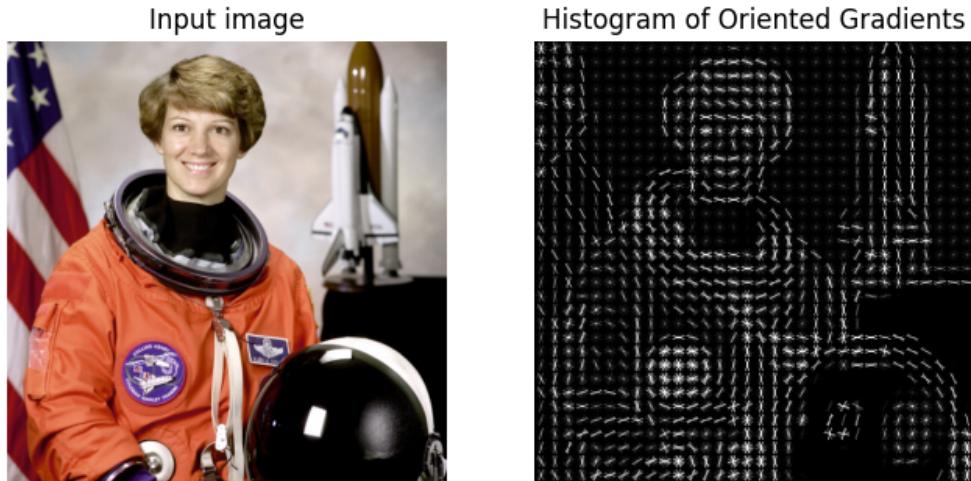
The fourth stage computes normalisation, which takes local groups of cells and contrast normalises their overall responses before passing to next stage. Normalisation introduces better invariance to illumination, shadowing, and edge contrast. It is performed by accumulating a measure of local histogram “energy” over local groups of cells that we call “blocks”. The result is used to normalise each cell in the block. Typically each individual cell is shared between several blocks, but its normalisations are block dependent and thus different. The cell thus appears several times in the final output vector with different normalisations. This may seem redundant but it improves the performance. We refer to the normalised block descriptors as Histogram of Oriented Gradient (HOG) descriptors.

The final step collects the HOG descriptors from all blocks of a dense overlapping grid of blocks covering the detection window into a combined feature vector for use in the window classifier.

¹ Dalal, N. and Triggs, B., “Histograms of Oriented Gradients for Human Detection,” IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2005, San Diego, CA, USA.

² David G. Lowe, “Distinctive image features from scale-invariant keypoints,” International Journal of Computer Vision, 60, 2 (2004), pp. 91-110.

References



```

import matplotlib.pyplot as plt

from skimage.feature import hog
from skimage import data, exposure

image = data.astronaut()

fd, hog_image = hog(image, orientations=8, pixels_per_cell=(16, 16),
                    cells_per_block=(1, 1), visualize=True, channel_axis=-1)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 4), sharex=True, sharey=True)

ax1.axis('off')
ax1.imshow(image, cmap=plt.cm.gray)
ax1.set_title('Input image')

# Rescale histogram for better display
hog_image_rescaled = exposure.rescale_intensity(hog_image, in_range=(0, 10))

ax2.axis('off')
ax2.imshow(hog_image_rescaled, cmap=plt.cm.gray)
ax2.set_title('Histogram of Oriented Gradients')
plt.show()

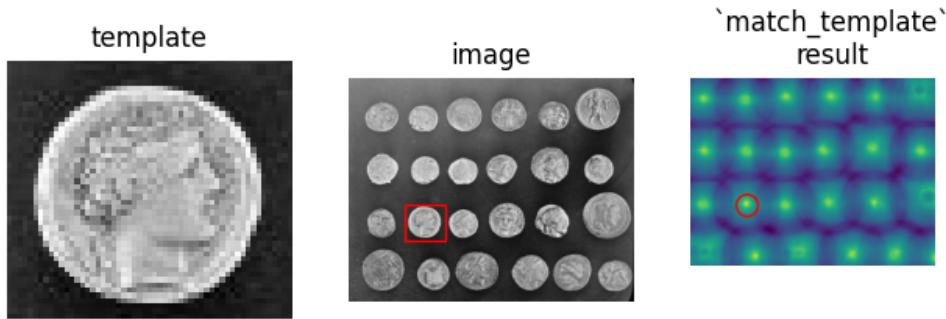
```

Total running time of the script: (0 minutes 0.432 seconds)

Template Matching

We use template matching to identify the occurrence of an image patch (in this case, a sub-image centered on a single coin). Here, we return a single match (the exact same coin), so the maximum value in the `match_template` result corresponds to the coin location. The other coins look similar, and thus have local maxima; if you expect multiple matches, you should use a proper peak-finding function.

The `match_template` function uses fast, normalized cross-correlation¹ to find instances of the template in the image. Note that the peaks in the output of `match_template` correspond to the origin (i.e. top-left corner) of the template.



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

from skimage import data
from skimage.feature import match_template

image = data.coins()
coin = image[170:220, 75:130]

result = match_template(image, coin)
ij = np.unravel_index(np.argmax(result), result.shape)
x, y = ij[::-1]

fig = plt.figure(figsize=(8, 3))
ax1 = plt.subplot(1, 3, 1)
ax2 = plt.subplot(1, 3, 2)
ax3 = plt.subplot(1, 3, 3, sharex=ax2, sharey=ax2)

ax1.imshow(coin, cmap=plt.cm.gray)
ax1.set_axis_off()
ax1.set_title('template')

ax2.imshow(image, cmap=plt.cm.gray)
ax2.set_axis_off()
ax2.set_title('image')
# highlight matched region
```

(continues on next page)

¹ J. P. Lewis, “Fast Normalized Cross-Correlation”, Industrial Light and Magic.

(continued from previous page)

```

hcoin, wcoin = coin.shape
rect = plt.Rectangle((x, y), wcoin, hcoin, edgecolor='r', facecolor='none')
ax2.add_patch(rect)

ax3.imshow(result)
ax3.set_axis_off()
ax3.set_title(`match_template`\nresult')
# highlight matched region
ax3.autoscale(False)
ax3.plot(x, y, 'o', markeredgecolor='r', markerfacecolor='none', markersize=10)

plt.show()

```

Total running time of the script: (0 minutes 0.137 seconds)

Filling holes and finding peaks

We fill holes (i.e. isolated, dark spots) in an image using morphological reconstruction by erosion. Erosion expands the minimal values of the seed image until it encounters a mask image. Thus, the seed image and mask image represent the maximum and minimum possible values of the reconstructed image.

We start with an image containing both peaks and holes:

```

import matplotlib.pyplot as plt

from skimage import data
from skimage.exposure import rescale_intensity

image = data.moon()
# Rescale image intensity so that we can see dim features.
image = rescale_intensity(image, in_range=(50, 200))

```

Now we need to create the seed image, where the minima represent the starting points for erosion. To fill holes, we initialize the seed image to the maximum value of the original image. Along the borders, however, we use the original values of the image. These border pixels will be the starting points for the erosion process. We then limit the erosion by setting the mask to the values of the original image.

```

import numpy as np
from skimage.morphology import reconstruction

seed = np.copy(image)
seed[1:-1, 1:-1] = image.max()
mask = image

filled = reconstruction(seed, mask, method='erosion')

```

As shown above, eroding inward from the edges removes holes, since (by definition) holes are surrounded by pixels of brighter value. Finally, we can isolate the dark regions by subtracting the reconstructed image from the original image.

Alternatively, we can find bright spots in an image using morphological reconstruction by dilation. Dilation is the inverse of erosion and expands the *maximal* values of the seed image until it encounters a mask image. Since this is an inverse operation, we initialize the seed image to the minimum image intensity instead of the maximum. The remainder of the process is the same.

```
seed = np.copy(image)
seed[1:-1, 1:-1] = image.min()
rec = reconstruction(seed, mask, method='dilation')

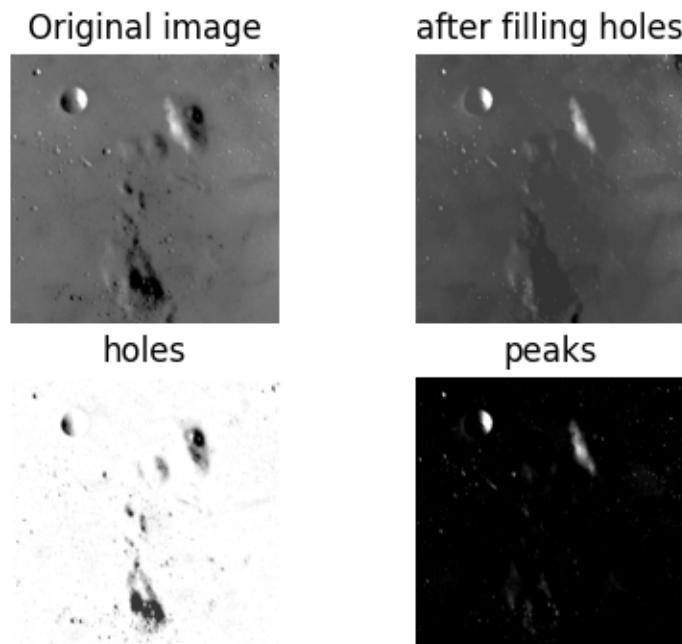
fig, ax = plt.subplots(2, 2, figsize=(5, 4), sharex=True, sharey=True)
ax = ax.ravel()

ax[0].imshow(image, cmap='gray')
ax[0].set_title('Original image')
ax[0].axis('off')

ax[1].imshow(filled, cmap='gray')
ax[1].set_title('after filling holes')
ax[1].axis('off')

ax[2].imshow(image-filled, cmap='gray')
ax[2].set_title('holes')
ax[2].axis('off')

ax[3].imshow(image-rec, cmap='gray')
ax[3].set_title('peaks')
ax[3].axis('off')
plt.show()
```



Total running time of the script: (0 minutes 0.310 seconds)

CENSURE feature detector

The CENSURE feature detector is a scale-invariant center-surround detector (CENSURE) that claims to outperform other detectors and is capable of real-time implementation.



```
from skimage import data
from skimage import transform
from skimage.feature import CENSURE
from skimage.color import rgb2gray

import matplotlib.pyplot as plt

img_orig = rgb2gray(data.astronaut())
tform = transform.AffineTransform(scale=(1.5, 1.5), rotation=0.5,
                                  translation=(150, -200))
img_warp = transform.warp(img_orig, tform)

detector = CENSURE()

fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(12, 6))

detector.detect(img_orig)

ax[0].imshow(img_orig, cmap=plt.cm.gray)
ax[0].scatter(detector.keypoints[:, 1], detector.keypoints[:, 0],
              2 ** detector.scales, facecolors='none', edgecolors='r')
ax[0].set_title("Original Image")

detector.detect(img_warp)

ax[1].imshow(img_warp, cmap=plt.cm.gray)
ax[1].scatter(detector.keypoints[:, 1], detector.keypoints[:, 0],
```

(continues on next page)

(continued from previous page)

```

2 ** detector.scales, facecolors='none', edgecolors='r')
ax[1].set_title('Transformed Image')

for a in ax:
    a.axis('off')

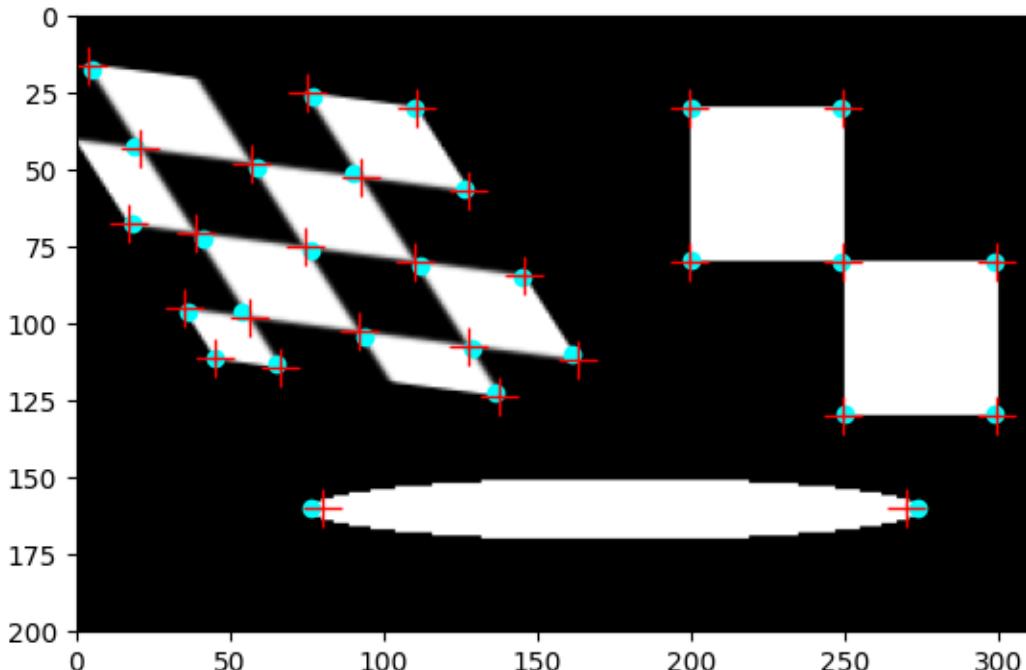
plt.tight_layout()
plt.show()

```

Total running time of the script: (0 minutes 0.956 seconds)

Corner detection

Detect corner points using the Harris corner detector and determine the subpixel position of corners (¹,²).



```

from matplotlib import pyplot as plt

from skimage import data
from skimage.feature import corner_harris, corner_subpix, corner_peaks
from skimage.transform import warp, AffineTransform
from skimage.draw import ellipse

```

(continues on next page)

¹ https://en.wikipedia.org/wiki/Corner_detection

² https://en.wikipedia.org/wiki/Interest_point_detection

(continued from previous page)

```
# Sheared checkerboard
tform = AffineTransform(scale=(1.3, 1.1), rotation=1, shear=0.7,
                      translation=(110, 30))
image = warp(data.checkerboard()[:90, :90], tform.inverse,
             output_shape=(200, 310))
# Ellipse
rr, cc = ellipse(160, 175, 10, 100)
image[rr, cc] = 1
# Two squares
image[30:80, 200:250] = 1
image[80:130, 250:300] = 1

coords = corner_peaks(corner_harris(image), min_distance=5, threshold_rel=0.02)
coords_subpix = corner_subpix(image, coords, window_size=13)

fig, ax = plt.subplots()
ax.imshow(image, cmap=plt.cm.gray)
ax.plot(coords[:, 1], coords[:, 0], color='cyan', marker='o',
        linestyle='None', markersize=6)
ax.plot(coords_subpix[:, 1], coords_subpix[:, 0], '+r', markersize=15)
ax.axis((0, 310, 200, 0))
plt.show()
```

Total running time of the script: (0 minutes 0.101 seconds)

Multi-Block Local Binary Pattern for texture classification

This example shows how to compute multi-block local binary pattern (MB-LBP) features as well as how to visualize them.

The features are calculated similarly to local binary patterns (LBPs), except that summed blocks are used instead of individual pixel values.

MB-LBP is an extension of LBP that can be computed on multiple scales in constant time using the integral image. 9 equally-sized rectangles are used to compute a feature. For each rectangle, the sum of the pixel intensities is computed. Comparisons of these sums to that of the central rectangle determine the feature, similarly to LBP (See [LBP](#)).

First, we generate an image to illustrate the functioning of MB-LBP: consider a (9, 9) rectangle and divide it into (3, 3) block, upon which we then apply MB-LBP.

```
from skimage.feature import multiblock_lbp
import numpy as np
from numpy.testing import assert_equal
from skimage.transform import integral_image

# Create test matrix where first and fifth rectangles starting
# from top left clockwise have greater value than the central one.
test_img = np.zeros((9, 9), dtype='uint8')
test_img[3:6, 3:6] = 1
test_img[:3, :3] = 50
test_img[6:, 6:] = 50

# First and fifth bits should be filled. This correct value will
```

(continues on next page)

(continued from previous page)

```
# be compared to the computed one.
correct_answer = 0b10001000

int_img = integral_image(test_img)

lbp_code = multiblock_lbp(int_img, 0, 0, 3, 3)

assert_equal(correct_answer, lbp_code)
```

Now let's apply the operator to a real image and see how the visualization works.

```
from skimage import data
from matplotlib import pyplot as plt
from skimage.feature import draw_multiblock_lbp

test_img = data.coins()

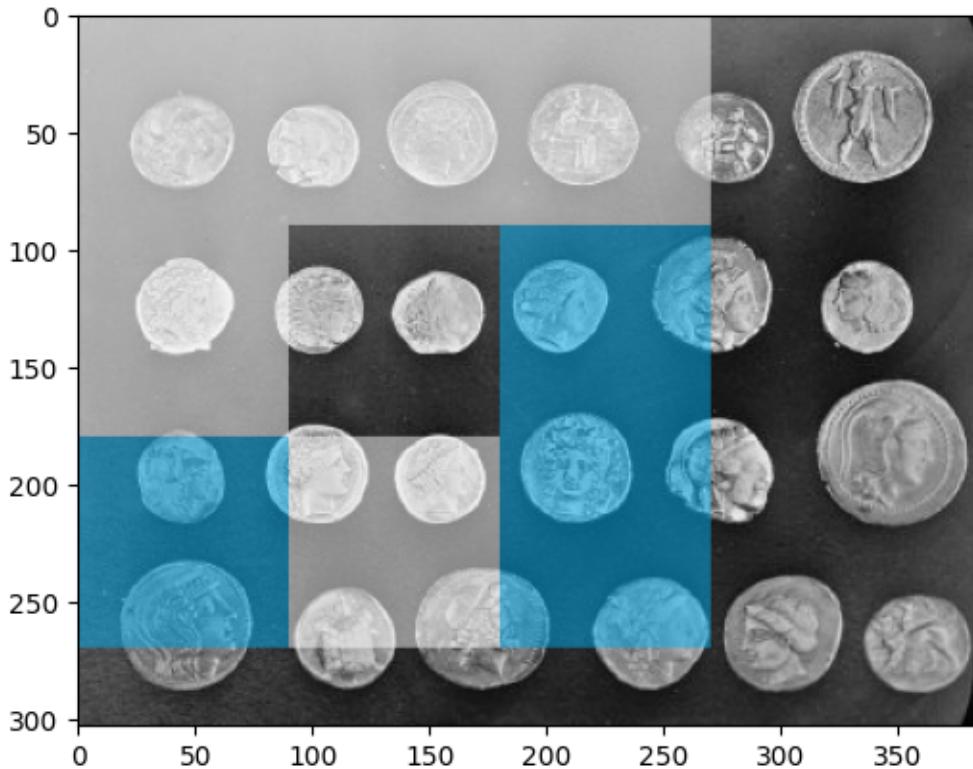
int_img = integral_image(test_img)

lbp_code = multiblock_lbp(int_img, 0, 0, 90, 90)

img = draw_multiblock_lbp(test_img, 0, 0, 90, 90,
                         lbp_code=lbp_code, alpha=0.5)

plt.imshow(img)

plt.show()
```



On the above plot we see the result of computing a MB-LBP and visualization of the computed feature. The rectangles that have less intensities' sum than the central rectangle are marked in cyan. The ones that have higher intensity values are marked in white. The central rectangle is left untouched.

Total running time of the script: (0 minutes 0.143 seconds)

Haar-like feature descriptor

Haar-like features are simple digital image features that were introduced in a real-time face detector¹. These features can be efficiently computed on any scale in constant time, using an integral image². After that, a small number of critical features is selected from this large set of potential features (e.g., using AdaBoost learning algorithm as in²). The following example will show the mechanism to build this family of descriptors.

¹ Viola, Paul, and Michael J. Jones. "Robust real-time face detection." International journal of computer vision 57.2 (2004): 137-154. <https://www.merl.com/publications/docs/TR2004-043.pdf> DOI:10.1109/CVPR.2001.990517

References

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

from skimage.feature import haar_like_feature_coord
from skimage.feature import draw_haar_like_feature
```

Different types of Haar-like feature descriptors

The Haar-like feature descriptors come into 5 different types as illustrated in the figure below. The value of the descriptor is equal to the difference between the sum of intensity values in the green and the red one.

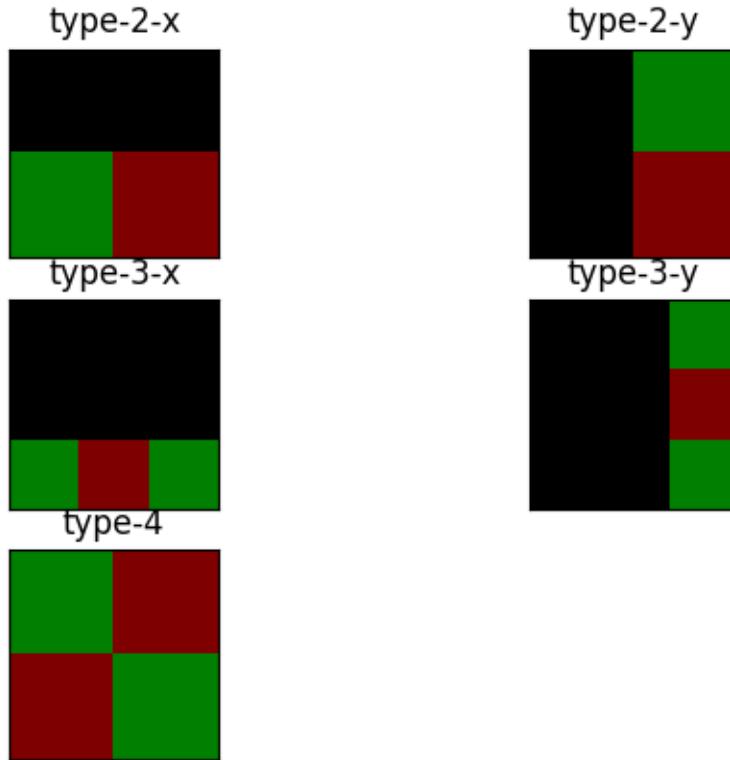
```
images = [np.zeros((2, 2)), np.zeros((2, 2)),
          np.zeros((3, 3)), np.zeros((3, 3)),
          np.zeros((2, 2))]

feature_types = ['type-2-x', 'type-2-y',
                  'type-3-x', 'type-3-y',
                  'type-4']

fig, axs = plt.subplots(3, 2)
for ax, img, feat_t in zip(np.ravel(axs), images, feature_types):
    coord, _ = haar_like_feature_coord(img.shape[0], img.shape[1], feat_t)
    haar_feature = draw_haar_like_feature(img, 0, 0,
                                           img.shape[0],
                                           img.shape[1],
                                           coord,
                                           max_n_features=1,
                                           random_state=0)
    ax.imshow(haar_feature)
    ax.set_title(feat_t)
    ax.set_xticks([])
    ax.set_yticks([])

fig.suptitle('The different Haar-like feature descriptors')
plt.axis('off')
plt.show()
```

The different Haar-like feature descriptors



```
/github/workspace/build/scikit-image/doc/examples/features_detection/plot_haar.py:48: FutureWarning:
```

```
`random_state` is a deprecated argument name for `draw_haar_like_feature`. It will be removed in version 0.23. Please use `rng` instead.
```

The value of the descriptor is equal to the difference between the sum of the intensity values in the green rectangle and the red one. The red area is subtracted to the sum of the pixel intensities of the green. In practice, the Haar-like features will be placed in all possible location of an image and a feature value will be computed for each of these locations.

Total running time of the script: (0 minutes 0.123 seconds)

Blob Detection

Blobs are bright on dark or dark on bright regions in an image. In this example, blobs are detected using 3 algorithms. The image used in this case is the Hubble eXtreme Deep Field. Each bright dot in the image is a star or a galaxy.

Laplacian of Gaussian (LoG)

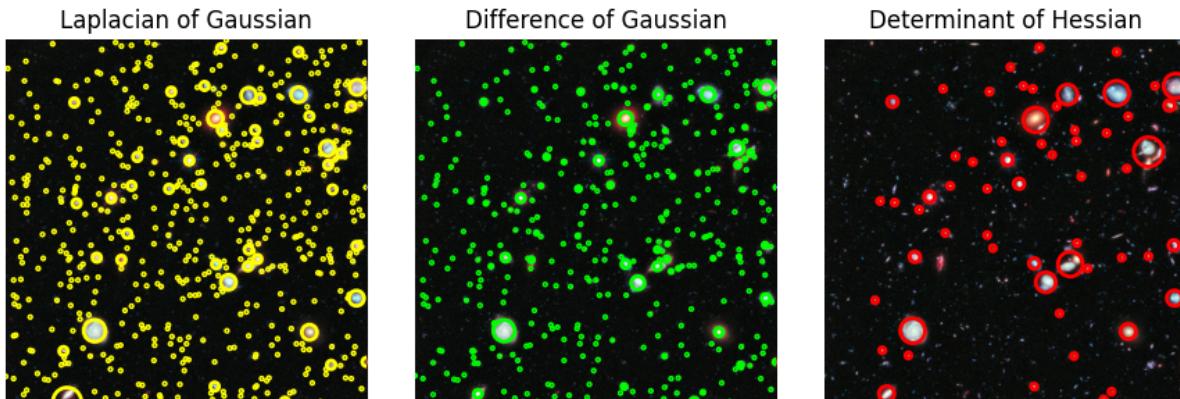
This is the most accurate and slowest approach. It computes the Laplacian of Gaussian images with successively increasing standard deviation and stacks them up in a cube. Blobs are local maximas in this cube. Detecting larger blobs is especially slower because of larger kernel sizes during convolution. Only bright blobs on dark backgrounds are detected. See `skimage.feature.blob_log()` for usage.

Difference of Gaussian (DoG)

This is a faster approximation of LoG approach. In this case the image is blurred with increasing standard deviations and the difference between two successively blurred images are stacked up in a cube. This method suffers from the same disadvantage as LoG approach for detecting larger blobs. Blobs are again assumed to be bright on dark. See `skimage.feature.blob_dog()` for usage.

Determinant of Hessian (DoH)

This is the fastest approach. It detects blobs by finding maximas in the matrix of the Determinant of Hessian of the image. The detection speed is independent of the size of blobs as internally the implementation uses box filters instead of convolutions. Bright on dark as well as dark on bright blobs are detected. The downside is that small blobs (<3px) are not detected accurately. See `skimage.feature.blob_doh()` for usage.



```
from math import sqrt
from skimage import data
from skimage.feature import blob_dog, blob_log, blob_doh
from skimage.color import rgb2gray

import matplotlib.pyplot as plt

image = data.hubble_deep_field()[0:500, 0:500]
image_gray = rgb2gray(image)

blobs_log = blob_log(image_gray, max_sigma=30, num_sigma=10, threshold=.1)

# Compute radii in the 3rd column.
blobs_log[:, 2] = blobs_log[:, 2] * sqrt(2)
```

(continues on next page)

(continued from previous page)

```
blobs_dog = blob_dog(image_gray, max_sigma=30, threshold=.1)
blobs_dog[:, 2] = blobs_dog[:, 2] * sqrt(2)

blobs_doh = blob_doh(image_gray, max_sigma=30, threshold=.01)

blobs_list = [blobs_log, blobs_dog, blobs_doh]
colors = ['yellow', 'lime', 'red']
titles = ['Laplacian of Gaussian', 'Difference of Gaussian',
          'Determinant of Hessian']
sequence = zip(blobs_list, colors, titles)

fig, axes = plt.subplots(1, 3, figsize=(9, 3), sharex=True, sharey=True)
ax = axes.ravel()

for idx, (blobs, color, title) in enumerate(sequence):
    ax[idx].set_title(title)
    ax[idx].imshow(image)
    for blob in blobs:
        y, x, r = blob
        c = plt.Circle((x, y), r, color=color, linewidth=2, fill=False)
        ax[idx].add_patch(c)
    ax[idx].set_axis_off()

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

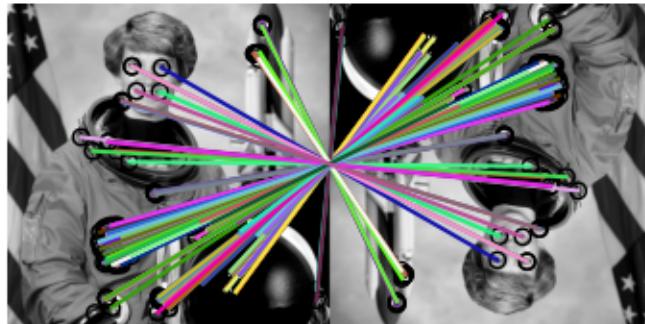
Total running time of the script: (0 minutes 4.391 seconds)

ORB feature detector and binary descriptor

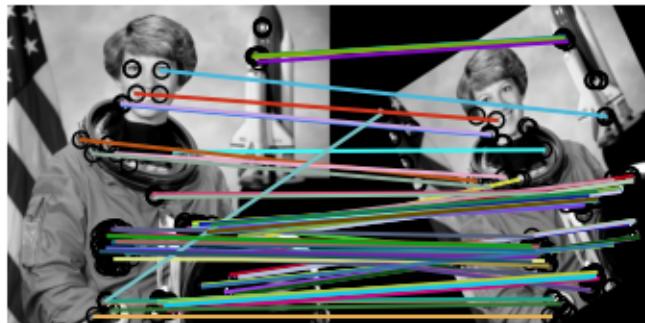
This example demonstrates the ORB feature detection and binary description algorithm. It uses an oriented FAST detection method and the rotated BRIEF descriptors.

Unlike BRIEF, ORB is comparatively scale and rotation invariant while still employing the very efficient Hamming distance metric for matching. As such, it is preferred for real-time applications.

Original Image vs. Transformed Image



Original Image vs. Transformed Image



```
from skimage import data
from skimage import transform
from skimage.feature import match_descriptors, ORB, plot_matches
from skimage.color import rgb2gray
import matplotlib.pyplot as plt

img1 = rgb2gray(data.astronaut())
img2 = transform.rotate(img1, 180)
tform = transform.AffineTransform(scale=(1.3, 1.1), rotation=0.5,
                                 translation=(0, -200))
img3 = transform.warp(img1, tform)

descriptor_extractor = ORB(n_keypoints=200)

descriptor_extractor.detect_and_extract(img1)
keypoints1 = descriptor_extractor.keypoints
descriptors1 = descriptor_extractor.descriptors

descriptor_extractor.detect_and_extract(img2)
keypoints2 = descriptor_extractor.keypoints
descriptors2 = descriptor_extractor.descriptors
```

(continues on next page)

(continued from previous page)

```

descriptor_extractor.detect_and_extract(img3)
keypoints3 = descriptor_extractor.keypoints
descriptors3 = descriptor_extractor.descriptors

matches12 = match_descriptors(descriptors1, descriptors2, cross_check=True)
matches13 = match_descriptors(descriptors1, descriptors3, cross_check=True)

fig, ax = plt.subplots(nrows=2, ncols=1)

plt.gray()

plot_matches(ax[0], img1, img2, keypoints1, keypoints2, matches12)
ax[0].axis('off')
ax[0].set_title("Original Image vs. Transformed Image")

plot_matches(ax[1], img1, img3, keypoints1, keypoints3, matches13)
ax[1].axis('off')
ax[1].set_title("Original Image vs. Transformed Image")

plt.show()

```

Total running time of the script: (0 minutes 1.602 seconds)

Gabors / Primary Visual Cortex “Simple Cells” from an Image

How to build a (bio-plausible) *sparse* dictionary (or ‘codebook’, or ‘filterbank’) for e.g. image classification without any fancy math and with just standard python scientific libraries?

Please find below a short answer ;-)

This simple example shows how to get Gabor-like filters¹ using just a simple image. In our example, we use a photograph of the astronaut Eileen Collins. Gabor filters are good approximations of the “Simple Cells”² receptive fields³ found in the mammalian primary visual cortex (V1) (for details, see e.g. the Nobel-prize winning work of Hubel & Wiesel done in the 60s⁴⁵).

Here we use McQueen’s ‘kmeans’ algorithm⁶, as a simple biologically plausible hebbian-like learning rule and we apply it (a) to patches of the original image (retinal projection), and (b) to patches of an LGN-like⁷ image using a simple difference of gaussians (DoG) approximation.

Enjoy ;-) And keep in mind that getting Gabors on natural image patches is not rocket science.

¹ https://en.wikipedia.org/wiki/Gabor_filter

² https://en.wikipedia.org/wiki/Simple_cell

³ https://en.wikipedia.org/wiki/Receptive_field

⁴ D. H. Hubel and T. N., Wiesel Receptive Fields of Single Neurones in the Cat's Striate Cortex, J. Physiol. pp. 574-591 (148) 1959

⁵ D. H. Hubel and T. N., Wiesel Receptive Fields, Binocular Interaction, and Functional Architecture in the Cat's Visual Cortex, J. Physiol. 160 pp. 106-154 1962

⁶ https://en.wikipedia.org/wiki/K-means_clustering

⁷ https://en.wikipedia.org/wiki/Lateral_geniculate_nucleus

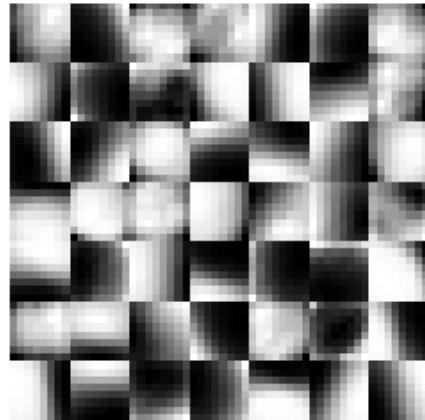
Image (original)



Image (LGN-like DoG)



K-means filterbank (codebook)
on original image



K-means filterbank (codebook)
on LGN-like DoG image



```
/usr/local/lib/python3.8/site-packages/scipy/cluster/vq.py:602: UserWarning:
```

```
One of the clusters is empty. Re-run kmeans with a different initialization.
```

```
from scipy.cluster.vq import kmeans2
from scipy import ndimage as ndi
import matplotlib.pyplot as plt

from skimage import data
from skimage import color
from skimage.util.shape import view_as_windows
from skimage.util import montage

patch_shape = 8, 8
```

(continues on next page)

(continued from previous page)

```

n_filters = 49

astro = color.rgb2gray(data.astronaut())

# -- filterbank1 on original image
patches1 = view_as_windows(astro, patch_shape)
patches1 = patches1.reshape(-1, patch_shape[0] * patch_shape[1])[::8]
fb1, _ = kmeans2(patches1, n_filters, minit='points')
fb1 = fb1.reshape((-1,) + patch_shape)
fb1_montage = montage(fb1, rescale_intensity=True)

# -- filterbank2 LGN-like image
astro_dog = ndi.gaussian_filter(astro, .5) - ndi.gaussian_filter(astro, 1)
patches2 = view_as_windows(astro_dog, patch_shape)
patches2 = patches2.reshape(-1, patch_shape[0] * patch_shape[1])[::8]
fb2, _ = kmeans2(patches2, n_filters, minit='points')
fb2 = fb2.reshape((-1,) + patch_shape)
fb2_montage = montage(fb2, rescale_intensity=True)

# -- plotting
fig, axes = plt.subplots(2, 2, figsize=(7, 6))
ax = axes.ravel()

ax[0].imshow(astro, cmap=plt.cm.gray)
ax[0].set_title("Image (original)")

ax[1].imshow(fb1_montage, cmap=plt.cm.gray)
ax[1].set_title("K-means filterbank (codebook)\non original image")

ax[2].imshow(astro_dog, cmap=plt.cm.gray)
ax[2].set_title("Image (LGN-like DoG)")

ax[3].imshow(fb2_montage, cmap=plt.cm.gray)
ax[3].set_title("K-means filterbank (codebook)\non LGN-like DoG image")

for a in ax.ravel():
    a.axis('off')

fig.tight_layout()
plt.show()

```

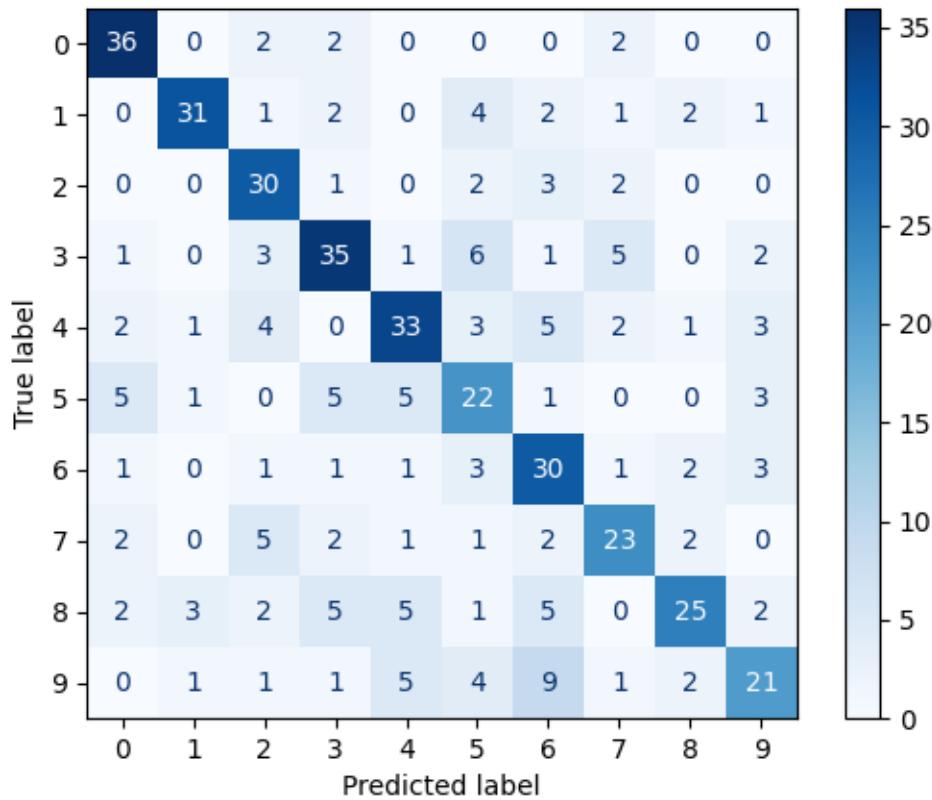
Total running time of the script: (0 minutes 0.793 seconds)

Fisher vector feature encoding

A Fisher vector is an image feature encoding and quantization technique that can be seen as a soft or probabilistic version of the popular bag-of-visual-words or VLAD algorithms. Images are modelled using a visual vocabulary which is estimated using a K-mode Gaussian mixture model trained on low-level image features such as SIFT or ORB descriptors. The Fisher vector itself is a concatenation of the gradients of the Gaussian mixture model (GMM) with respect to its parameters - mixture weights, means, and covariance matrices.

In this example, we compute Fisher vectors for the digits dataset in scikit-learn, and train a classifier on these representations.

Please note that scikit-learn is required to run this example.



```
/usr/local/lib/python3.8/site-packages/sklearn/svm/_classes.py:32: FutureWarning:
```

The default value of `dual` will change from `True` to ``auto`` in 1.5. Set the value of [`dual`](#) explicitly to suppress the warning.

	precision	recall	f1-score	support
0	0.73	0.86	0.79	42
1	0.84	0.70	0.77	44
2	0.61	0.79	0.69	38
3	0.65	0.65	0.65	54
4	0.65	0.61	0.63	54

(continues on next page)

(continued from previous page)

5	0.48	0.52	0.50	42
6	0.52	0.70	0.59	43
7	0.62	0.61	0.61	38
8	0.74	0.50	0.60	50
9	0.60	0.47	0.52	45
accuracy			0.64	450
macro avg	0.64	0.64	0.64	450
weighted avg	0.65	0.64	0.63	450

```

from matplotlib import pyplot as plt
import numpy as np
from sklearn.datasets import load_digits
from sklearn.metrics import classification_report, ConfusionMatrixDisplay
from sklearn.model_selection import train_test_split
from sklearn.svm import LinearSVC

from skimage.transform import resize
from skimage.feature import fisher_vector, ORB, learn_gmm

data = load_digits()
images = data.images
targets = data.target

# Resize images so that ORB detects interest points for all images
images = np.array([resize(image, (80, 80)) for image in images])

# Compute ORB descriptors for each image
descriptors = []
for image in images:
    detector_extractor = ORB(n_keypoints=5, harris_k=0.01)
    detector_extractor.detect_and_extract(image)
    descriptors.append(detector_extractor.descriptors.astype('float32'))

# Split the data into training and testing subsets
train_descriptors, test_descriptors, train_targets, test_targets = \
    train_test_split(descriptors, targets)

# Train a K-mode GMM
k = 16
gmm = learn_gmm(train_descriptors, n_modes=k)

# Compute the Fisher vectors
training_fvs = np.array([
    fisher_vector(descriptor_mat, gmm)
    for descriptor_mat in train_descriptors
])

```

(continues on next page)

(continued from previous page)

```
])

testing_fvs = np.array([
    fisher_vector(descriptor_mat, gmm)
    for descriptor_mat in test_descriptors
])

svm = LinearSVC().fit(training_fvs, train_targets)

predictions = svm.predict(testing_fvs)

print(classification_report(test_targets, predictions))

ConfusionMatrixDisplay.from_estimator(
    svm,
    testing_fvs,
    test_targets,
    cmap=plt.cm.Blues,
)

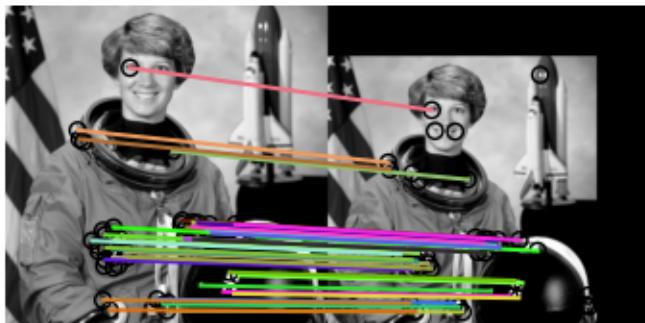
plt.show()
```

Total running time of the script: (0 minutes 55.462 seconds)

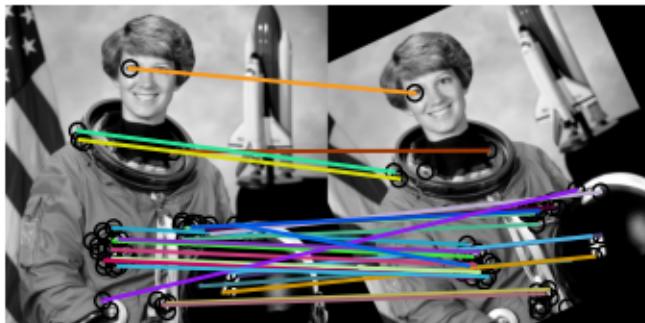
BRIEF binary descriptor

This example demonstrates the BRIEF binary description algorithm. The descriptor consists of relatively few bits and can be computed using a set of intensity difference tests. The short binary descriptor results in low memory footprint and very efficient matching based on the Hamming distance metric. BRIEF does not provide rotation-invariance. Scale-invariance can be achieved by detecting and extracting features at different scales.

Original Image vs. Transformed Image



Original Image vs. Transformed Image



```

from skimage import data
from skimage import transform
from skimage.feature import (match_descriptors, corner_peaks, corner_harris,
                             plot_matches, BRIEF)
from skimage.color import rgb2gray
import matplotlib.pyplot as plt

img1 = rgb2gray(data.astronaut())
tform = transform.AffineTransform(scale=(1.2, 1.2), translation=(0, -100))
img2 = transform.warp(img1, tform)
img3 = transform.rotate(img1, 25)

keypoints1 = corner_peaks(corner_harris(img1), min_distance=5,
                          threshold_rel=0.1)
keypoints2 = corner_peaks(corner_harris(img2), min_distance=5,
                          threshold_rel=0.1)
keypoints3 = corner_peaks(corner_harris(img3), min_distance=5,
                          threshold_rel=0.1)

extractor = BRIEF()

extractor.extract(img1, keypoints1)

```

(continues on next page)

(continued from previous page)

```
keypoints1 = keypoints1[extractor.mask]
descriptors1 = extractor.descriptors

extractor.extract(img2, keypoints2)
keypoints2 = keypoints2[extractor.mask]
descriptors2 = extractor.descriptors

extractor.extract(img3, keypoints3)
keypoints3 = keypoints3[extractor.mask]
descriptors3 = extractor.descriptors

matches12 = match_descriptors(descriptors1, descriptors2, cross_check=True)
matches13 = match_descriptors(descriptors1, descriptors3, cross_check=True)

fig, ax = plt.subplots(nrows=2, ncols=1)

plt.gray()

plot_matches(ax[0], img1, img2, keypoints1, keypoints2, matches12)
ax[0].axis('off')
ax[0].set_title("Original Image vs. Transformed Image")

plot_matches(ax[1], img1, img3, keypoints1, keypoints3, matches13)
ax[1].axis('off')
ax[1].set_title("Original Image vs. Transformed Image")

plt.show()
```

Total running time of the script: (0 minutes 0.344 seconds)

SIFT feature detector and descriptor extractor

This example demonstrates the SIFT feature detection and its description algorithm.

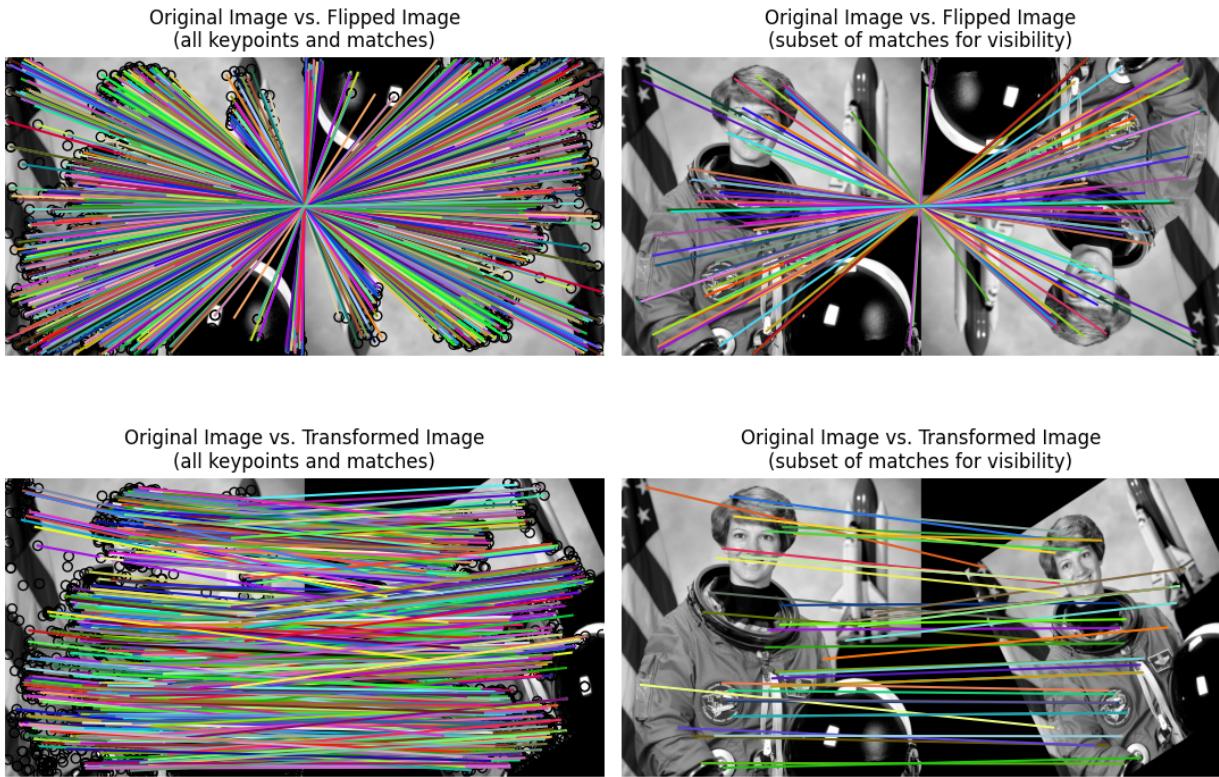
The scale-invariant feature transform (SIFT)¹ was published in 1999 and is still one of the most popular feature detectors available, as it promises to be “invariant to image scaling, translation, and rotation, and partially invariant to illumination changes and affine or 3D projection”². Its biggest drawback is its runtime, that’s said to be “at two orders of magnitude”³ slower than ORB, which makes it unsuitable for real-time applications.

¹ https://en.wikipedia.org/wiki/Scale-invariant_feature_transform

² D.G. Lowe. “Object recognition from local scale-invariant features”, Proceedings of the Seventh IEEE International Conference on Computer Vision, 1999, vol.2, pp. 1150-1157. DOI:10.1109/ICCV.1999.790410

³ Ethan Rublee, Vincent Rabaud, Kurt Konolige and Gary Bradski “ORB: An efficient alternative to SIFT and SURF” http://www.gwylab.com/download/ORB_2012.pdf

References



```

import matplotlib.pyplot as plt

from skimage import data
from skimage import transform
from skimage.color import rgb2gray
from skimage.feature import match_descriptors, plot_matches, SIFT

img1 = rgb2gray(data.astronaut())
img2 = transform.rotate(img1, 180)
tform = transform.AffineTransform(scale=(1.3, 1.1), rotation=0.5,
                                 translation=(0, -200))
img3 = transform.warp(img1, tform)

descriptor_extractor = SIFT()

descriptor_extractor.detect_and_extract(img1)
keypoints1 = descriptor_extractor.keypoints
descriptors1 = descriptor_extractor.descriptors

descriptor_extractor.detect_and_extract(img2)
keypoints2 = descriptor_extractor.keypoints

```

(continues on next page)

(continued from previous page)

```

descriptors2 = descriptor_extractor.descriptors

descriptor_extractor.detect_and_extract(img3)
keypoints3 = descriptor_extractor.keypoints
descriptors3 = descriptor_extractor.descriptors

matches12 = match_descriptors(descriptors1, descriptors2, max_ratio=0.6,
                               cross_check=True)
matches13 = match_descriptors(descriptors1, descriptors3, max_ratio=0.6,
                               cross_check=True)
fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(11, 8))

plt.gray()

plot_matches(ax[0, 0], img1, img2, keypoints1, keypoints2, matches12)
ax[0, 0].axis('off')
ax[0, 0].set_title("Original Image vs. Flipped Image\n"
                    "(all keypoints and matches)")

plot_matches(ax[1, 0], img1, img3, keypoints1, keypoints3, matches13)
ax[1, 0].axis('off')
ax[1, 0].set_title("Original Image vs. Transformed Image\n"
                    "(all keypoints and matches)")

plot_matches(ax[0, 1], img1, img2, keypoints1, keypoints2, matches12[::-15],
            only_matches=True)
ax[0, 1].axis('off')
ax[0, 1].set_title("Original Image vs. Flipped Image\n"
                    "(subset of matches for visibility)")

plot_matches(ax[1, 1], img1, img3, keypoints1, keypoints3, matches13[::-15],
            only_matches=True)
ax[1, 1].axis('off')
ax[1, 1].set_title("Original Image vs. Transformed Image\n"
                    "(subset of matches for visibility)")

plt.tight_layout()
plt.show()

```

Total running time of the script: (0 minutes 4.874 seconds)

GLCM Texture Features

This example illustrates texture classification using gray level co-occurrence matrices (GLCMs)¹. A GLCM is a histogram of co-occurring grayscale values at a given offset over an image.

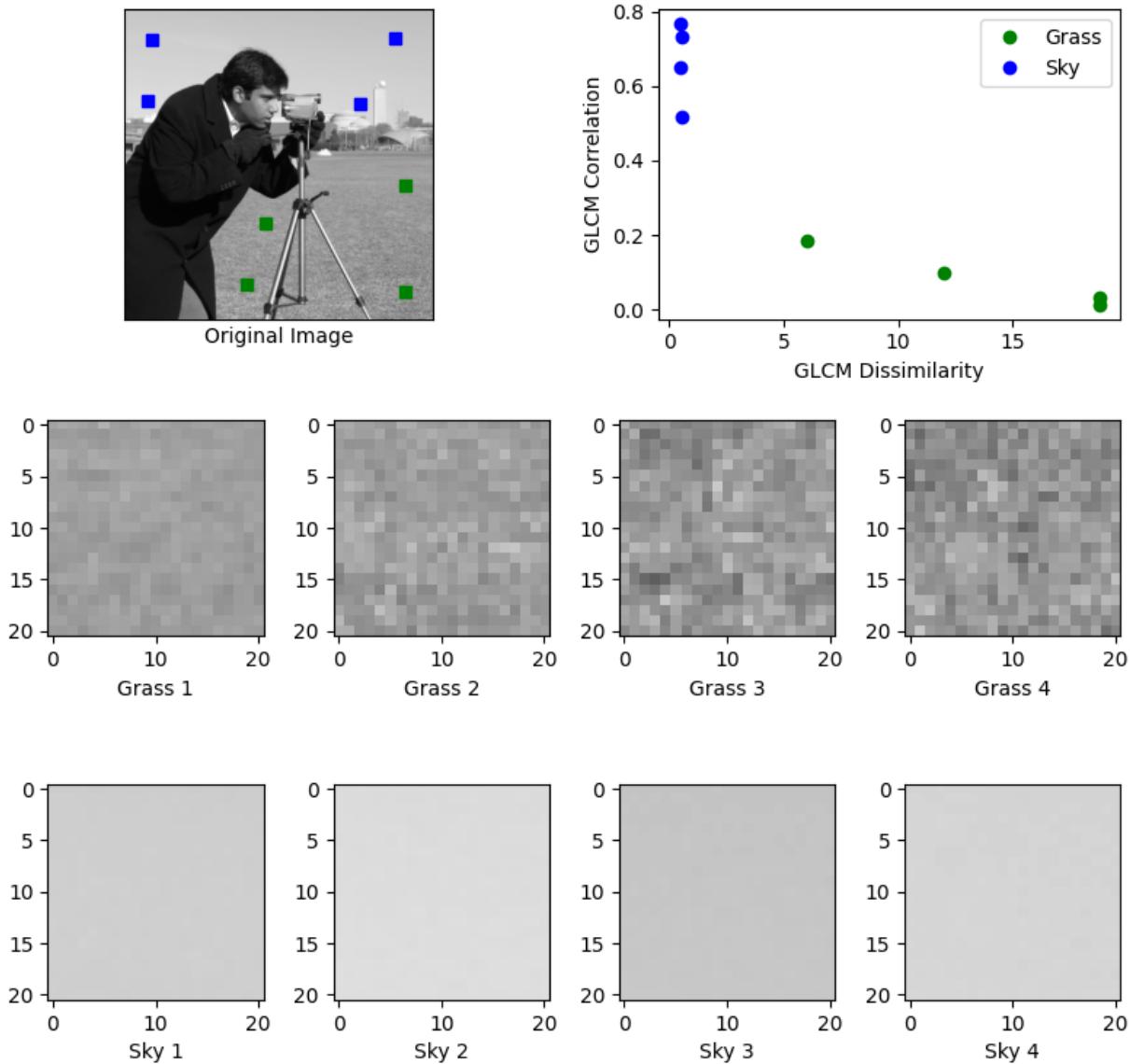
In this example, samples of two different textures are extracted from an image: grassy areas and sky areas. For each patch, a GLCM with a horizontal offset of 5 (*distance*=[5] and *angles*=[0]) is computed. Next, two features of the GLCM matrices are computed: dissimilarity and correlation. These are plotted to illustrate that the classes form clusters in feature space. In a typical classification problem, the final step (not included in this example) would be to train a classifier, such as logistic regression, to label image patches from new images.

¹ Haralick, RM.; Shanmugam, K., "Textural features for image classification" IEEE Transactions on systems, man, and cybernetics 6 (1973): 610-621. DOI:10.1109/TSMC.1973.4309314

Changed in version 0.19: `greymatrix` was renamed to `graymatrix` in 0.19.

Changed in version 0.19: `greycoprops` was renamed to `graycoprops` in 0.19.

References



```
import matplotlib.pyplot as plt

from skimage.feature import graycomatrix, graycoprops
from skimage import data
```

(continues on next page)

(continued from previous page)

```
PATCH_SIZE = 21

# open the camera image
image = data.camera()

# select some patches from grassy areas of the image
grass_locations = [(280, 454), (342, 223), (444, 192), (455, 455)]
grass_patches = []
for loc in grass_locations:
    grass_patches.append(image[loc[0]:loc[0] + PATCH_SIZE,
                               loc[1]:loc[1] + PATCH_SIZE])

# select some patches from sky areas of the image
sky_locations = [(38, 34), (139, 28), (37, 437), (145, 379)]
sky_patches = []
for loc in sky_locations:
    sky_patches.append(image[loc[0]:loc[0] + PATCH_SIZE,
                           loc[1]:loc[1] + PATCH_SIZE])

# compute some GLCM properties each patch
xs = []
ys = []
for patch in (grass_patches + sky_patches):
    glcm = graycomatrix(patch, distances=[5], angles=[0], levels=256,
                        symmetric=True, normed=True)
    xs.append(greycoprops(glcm, 'dissimilarity')[0, 0])
    ys.append(greycoprops(glcm, 'correlation')[0, 0])

# create the figure
fig = plt.figure(figsize=(8, 8))

# display original image with locations of patches
ax = fig.add_subplot(3, 2, 1)
ax.imshow(image, cmap=plt.cm.gray,
          vmin=0, vmax=255)
for (y, x) in grass_locations:
    ax.plot(x + PATCH_SIZE / 2, y + PATCH_SIZE / 2, 'gs')
for (y, x) in sky_locations:
    ax.plot(x + PATCH_SIZE / 2, y + PATCH_SIZE / 2, 'bs')
ax.set_xlabel('Original Image')
ax.set_xticks([])
ax.set_yticks([])
ax.axis('image')

# for each patch, plot (dissimilarity, correlation)
ax = fig.add_subplot(3, 2, 2)
ax.plot(xs[:len(grass_patches)], ys[:len(grass_patches)], 'go',
        label='Grass')
ax.plot(xs[len(grass_patches):], ys[len(grass_patches):], 'bo',
        label='Sky')
ax.set_xlabel('GLCM Dissimilarity')
ax.set_ylabel('GLCM Correlation')
```

(continues on next page)

(continued from previous page)

```

ax.legend()

# display the image patches
for i, patch in enumerate(grass_patches):
    ax = fig.add_subplot(3, len(grass_patches), len(grass_patches)*1 + i + 1)
    ax.imshow(patch, cmap=plt.cm.gray,
              vmin=0, vmax=255)
    ax.set_xlabel(f"Grass {i + 1}")

for i, patch in enumerate(sky_patches):
    ax = fig.add_subplot(3, len(sky_patches), len(sky_patches)*2 + i + 1)
    ax.imshow(patch, cmap=plt.cm.gray,
              vmin=0, vmax=255)
    ax.set_xlabel(f"Sky {i + 1}")

# display the patches and plot
fig.suptitle('Grey level co-occurrence matrix features', fontsize=14, y=1.05)
plt.tight_layout()
plt.show()

```

Total running time of the script: (0 minutes 0.714 seconds)

Shape Index

The shape index is a single valued measure of local curvature, derived from the eigen values of the Hessian, defined by Koenderink & van Doorn¹.

It can be used to find structures based on their apparent local shape.

The shape index maps to values from -1 to 1, representing different kind of shapes (see the documentation for details).

In this example, a random image with spots is generated, which should be detected.

A shape index of 1 represents ‘spherical caps’, the shape of the spots we want to detect.

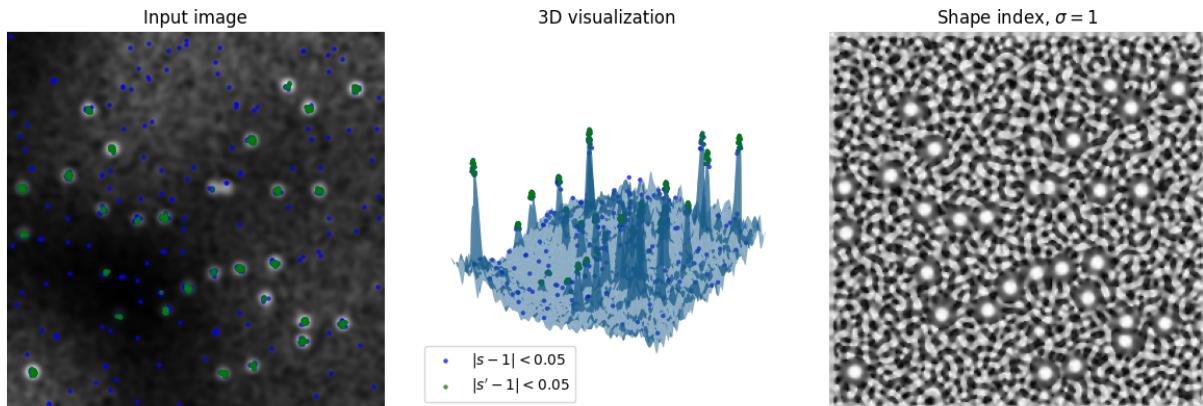
The leftmost plot shows the generated image, the center shows a 3D render of the image, taking intensity values as height of a 3D surface, and the right one shows the shape index (s).

As visible, the shape index readily amplifies the local shape of noise as well, but is insusceptible to global phenomena (e.g. uneven illumination).

The blue and green marks are points which deviate no more than 0.05 from the desired shape. To attenuate noise in the signal, the green marks are taken from the shape index (s) after another Gaussian blur pass (yielding s’).

Note how spots interjoined too closely are *not* detected, as they do not posses the desired shape.

¹ Koenderink, J. J. & van Doorn, A. J., “Surface shape and curvature scales”, Image and Vision Computing, 1992, 10, 557-564.
DOI:10.1016/0262-8856(92)90076-F



```

import numpy as np
import matplotlib.pyplot as plt
from scipy import ndimage as ndi
from skimage.feature import shape_index
from skimage.draw import disk

def create_test_image(
    image_size=256, spot_count=30, spot_radius=5, cloud_noise_size=4):
    """
    Generate a test image with random noise, uneven illumination and spots.
    """
    rng = np.random.default_rng()
    image = rng.normal(
        loc=0.25,
        scale=0.25,
        size=(image_size, image_size)
    )

    for _ in range(spot_count):
        rr, cc = disk(
            (rng.integers(image.shape[0]),
             rng.integers(image.shape[1])),
            spot_radius,
            shape=image.shape
        )
        image[rr, cc] = 1

    image *= rng.normal(loc=1.0, scale=0.1, size=image.shape)

    image *= ndi.zoom(
        rng.normal(
            loc=1.0,
            scale=0.5,
            size=(cloud_noise_size, cloud_noise_size)
        ),
        image_size / cloud_noise_size
    )

```

(continues on next page)

(continued from previous page)

```
    return ndi.gaussian_filter(image, sigma=2.0)

# First create the test image and its shape index

image = create_test_image()

s = shape_index(image)

# In this example we want to detect 'spherical caps',
# so we threshold the shape index map to
# find points which are 'spherical caps' (~1)

target = 1
delta = 0.05

point_y, point_x = np.where(np.abs(s - target) < delta)
point_z = image[point_y, point_x]

# The shape index map relentlessly produces the shape, even that of noise.
# In order to reduce the impact of noise, we apply a Gaussian filter to it,
# and show the results once in

s_smooth = ndi.gaussian_filter(s, sigma=0.5)

point_y_s, point_x_s = np.where(np.abs(s_smooth - target) < delta)
point_z_s = image[point_y_s, point_x_s]

fig = plt.figure(figsize=(12, 4))
ax1 = fig.add_subplot(1, 3, 1)

ax1.imshow(image, cmap=plt.cm.gray)
ax1.axis('off')
ax1.set_title('Input image')

scatter_settings = dict(alpha=0.75, s=10, linewidths=0)

ax1.scatter(point_x, point_y, color='blue', **scatter_settings)
ax1.scatter(point_x_s, point_y_s, color='green', **scatter_settings)

ax2 = fig.add_subplot(1, 3, 2, projection='3d', sharex=ax1, sharey=ax1)

x, y = np.meshgrid(
    np.arange(0, image.shape[0], 1),
    np.arange(0, image.shape[1], 1)
)

ax2.plot_surface(x, y, image, linewidth=0, alpha=0.5)

ax2.scatter(
    point_x,
    point_y,
```

(continues on next page)

(continued from previous page)

```

    point_z,
    color='blue',
    label='\$|s - 1|<0.05\$',
    **scatter_settings
)

ax2.scatter(
    point_x_s,
    point_y_s,
    point_z_s,
    color='green',
    label='\$|s\` - 1|<0.05\$',
    **scatter_settings
)

ax2.legend(loc='lower left')

ax2.axis('off')
ax2.set_title('3D visualization')

ax3 = fig.add_subplot(1, 3, 3, sharex=ax1, sharey=ax1)

ax3.imshow(s, cmap=plt.cm.gray)
ax3.axis('off')
ax3.set_title(r'Shape index, $\sigma=1$')

fig.tight_layout()

plt.show()

```

Total running time of the script: (0 minutes 0.504 seconds)

Sliding window histogram

Histogram matching can be used for object detection in images¹. This example extracts a single coin from the `skimage.data.coins` image and uses histogram matching to attempt to locate it within the original image.

First, a box-shaped region of the image containing the target coin is extracted and a histogram of its grayscale values is computed.

Next, for each pixel in the test image, a histogram of the grayscale values in a region of the image surrounding the pixel is computed. `skimage.filters.rank.windowed_histogram` is used for this task, as it employs an efficient sliding window based algorithm that is able to compute these histograms quickly². The local histogram for the region surrounding each pixel in the image is compared to that of the single coin, with a similarity measure being computed and displayed.

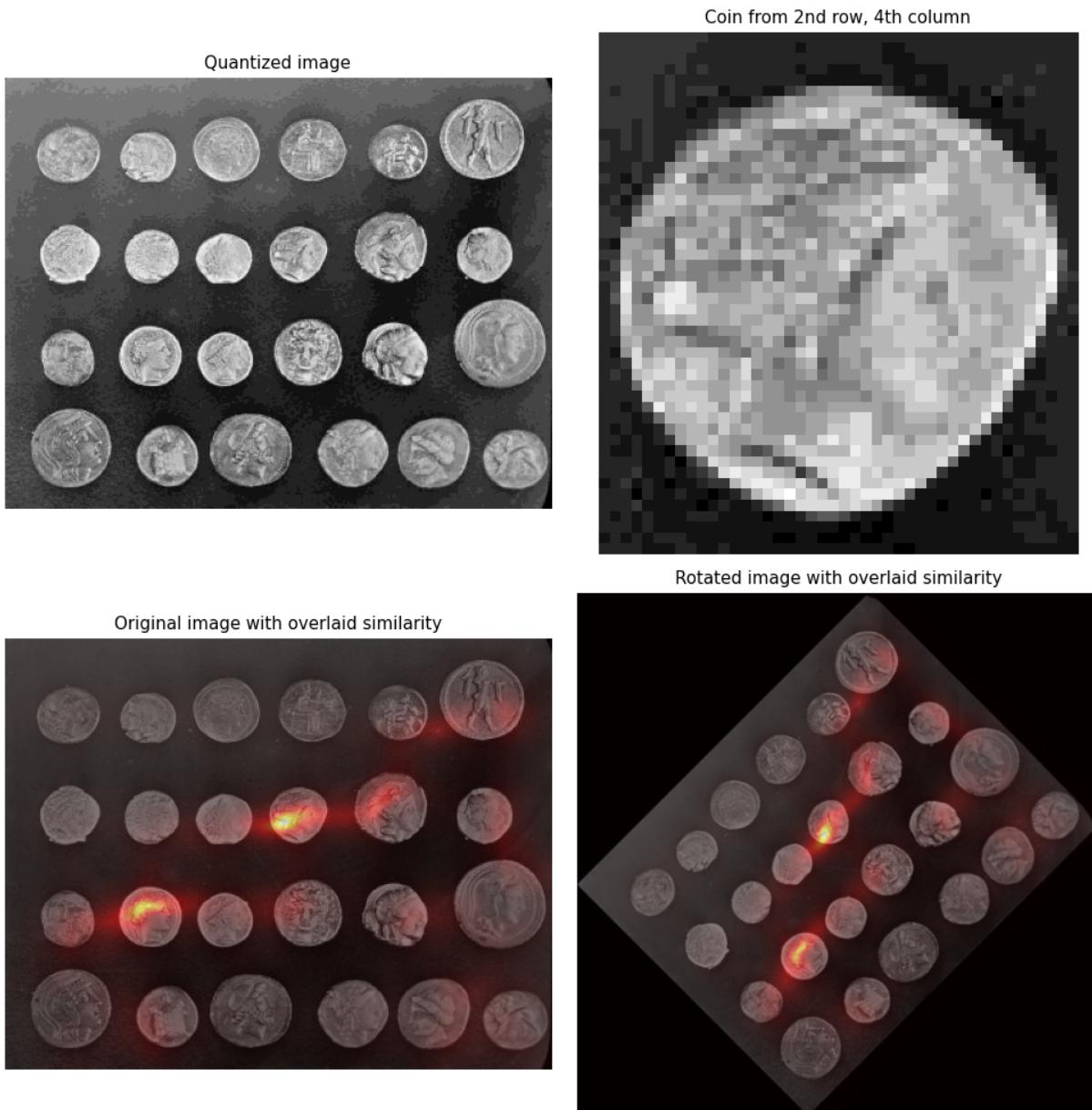
The histogram of the single coin is computed using `numpy.histogram` on a box shaped region surrounding the coin, while the sliding window histograms are computed using a disc shaped structural element of a slightly different size. This is done in aid of demonstrating that the technique still finds similarity in spite of these differences.

To demonstrate the rotational invariance of the technique, the same test is performed on a version of the coins image rotated by 45 degrees.

¹ Porikli, F. “Integral Histogram: A Fast Way to Extract Histograms in Cartesian Spaces” CVPR, 2005. Vol. 1. IEEE, 2005

² S.Perreault and P.Hebert. Median filtering in constant time. Trans. Image Processing, 16(9):2389-2394, 2007.

References



```

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

from skimage import data, transform
from skimage.util import img_as_ubyte
from skimage.morphology import disk
from skimage.filters import rank

```

(continues on next page)

(continued from previous page)

```
matplotlib.rcParams['font.size'] = 9

def windowed_histogram_similarity(image, footprint, reference_hist, n_bins):
    # Compute normalized windowed histogram feature vector for each pixel
    px_histograms = rank.windowed_histogram(image, footprint, n_bins=n_bins)

    # Reshape coin histogram to (1,1,N) for broadcast when we want to use it in
    # arithmetic operations with the windowed histograms from the image
    reference_hist = reference_hist.reshape((1, 1) + reference_hist.shape)

    # Compute Chi squared distance metric: sum((X-Y)^2 / (X+Y));
    # a measure of distance between histograms
    X = px_histograms
    Y = reference_hist

    num = (X - Y) ** 2
    denom = X + Y
    denom[denom == 0] = np.infty
    frac = num / denom

    chi_sqr = 0.5 * np.sum(frac, axis=2)

    # Generate a similarity measure. It needs to be low when distance is high
    # and high when distance is low; taking the reciprocal will do this.
    # Chi squared will always be >= 0, add small value to prevent divide by 0.
    similarity = 1 / (chi_sqr + 1.0e-4)

    return similarity

# Load the `skimage.data.coins` image
img = img_as_ubyte(data.coins())

# Quantize to 16 levels of grayscale; this way the output image will have a
# 16-dimensional feature vector per pixel
quantized_img = img // 16

# Select the coin from the 4th column, second row.
# Coordinate ordering: [x1,y1,x2,y2]
coin_coords = [184, 100, 228, 148]    # 44 x 44 region
coin = quantized_img[coin_coords[1]:coin_coords[3],
                     coin_coords[0]:coin_coords[2]]

# Compute coin histogram and normalize
coin_hist, _ = np.histogram(coin.flatten(), bins=16, range=(0, 16))
coin_hist = coin_hist.astype(float) / np.sum(coin_hist)

# Compute a disk shaped mask that will define the shape of our sliding window
# Example coin is ~44px across, so make a disk 61px wide (2 * rad + 1) to be
# big enough for other coins too.
footprint = disk(30)
```

(continues on next page)

(continued from previous page)

```

# Compute the similarity across the complete image
similarity = windowed_histogram_similarity(quantized_img, footprint, coin_hist,
                                            coin_hist.shape[0])

# Now try a rotated image
rotated_img = img_as_ubyte(transform.rotate(img, 45.0, resize=True))
# Quantize to 16 levels as before
quantized_rotated_image = rotated_img // 16
# Similarity on rotated image
rotated_similarity = windowed_histogram_similarity(quantized_rotated_image,
                                                    footprint, coin_hist,
                                                    coin_hist.shape[0])

fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(10, 10))

axes[0, 0].imshow(quantized_img, cmap='gray')
axes[0, 0].set_title('Quantized image')
axes[0, 0].axis('off')

axes[0, 1].imshow(coin, cmap='gray')
axes[0, 1].set_title('Coin from 2nd row, 4th column')
axes[0, 1].axis('off')

axes[1, 0].imshow(img, cmap='gray')
axes[1, 0].imshow(similarity, cmap='hot', alpha=0.5)
axes[1, 0].set_title('Original image with overlaid similarity')
axes[1, 0].axis('off')

axes[1, 1].imshow(rotated_img, cmap='gray')
axes[1, 1].imshow(rotated_similarity, cmap='hot', alpha=0.5)
axes[1, 1].set_title('Rotated image with overlaid similarity')
axes[1, 1].axis('off')

plt.tight_layout()
plt.show()

```

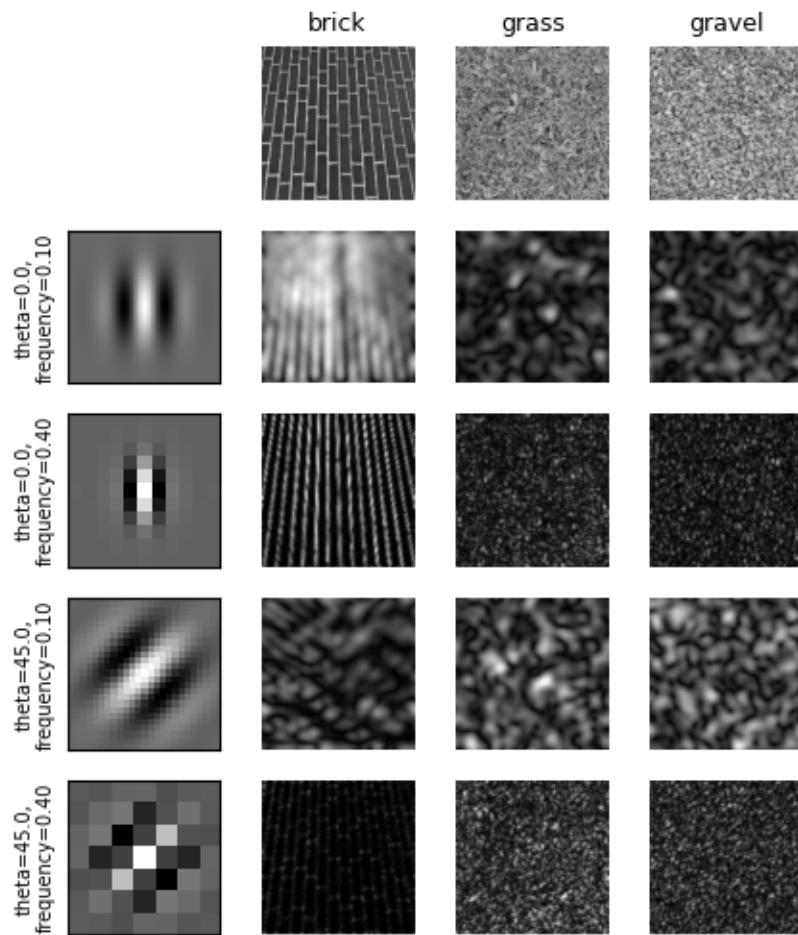
Total running time of the script: (0 minutes 0.757 seconds)

Gabor filter banks for texture classification

In this example, we will see how to classify textures based on Gabor filter banks. Frequency and orientation representations of the Gabor filter are similar to those of the human visual system.

The images are filtered using the real parts of various different Gabor filter kernels. The mean and variance of the filtered images are then used as features for classification, which is based on the least squared error for simplicity.

Image responses for Gabor filter kernels



Rotated images matched against references using Gabor filter banks:
original: brick, rotated: 30deg, match result: brick
original: brick, rotated: 70deg, match result: brick
original: grass, rotated: 145deg, match result: brick

```
import matplotlib.pyplot as plt
import numpy as np
from scipy import ndimage as ndi

from skimage import data
from skimage.util import img_as_float
```

(continues on next page)

(continued from previous page)

```

from skimage.filters import gabor_kernel

def compute_feats(image, kernels):
    feats = np.zeros((len(kernels), 2), dtype=np.double)
    for k, kernel in enumerate(kernels):
        filtered = ndi.convolve(image, kernel, mode='wrap')
        feats[k, 0] = filtered.mean()
        feats[k, 1] = filtered.var()
    return feats

def match(feats, ref_feats):
    min_error = np.inf
    min_i = None
    for i in range(ref_feats.shape[0]):
        error = np.sum((feats - ref_feats[i, :]) ** 2)
        if error < min_error:
            min_error = error
            min_i = i
    return min_i

# prepare filter bank kernels
kernels = []
for theta in range(4):
    theta = theta / 4. * np.pi
    for sigma in (1, 3):
        for frequency in (0.05, 0.25):
            kernel = np.real(gabor_kernel(frequency, theta=theta,
                                          sigma_x=sigma, sigma_y=sigma))
            kernels.append(kernel)

shrink = (slice(0, None, 3), slice(0, None, 3))
brick = img_as_float(data.brick())[shrink]
grass = img_as_float(data.grass())[shrink]
gravel = img_as_float(data.gravel())[shrink]
image_names = ('brick', 'grass', 'gravel')
images = (brick, grass, gravel)

# prepare reference features
ref_feats = np.zeros((3, len(kernels), 2), dtype=np.double)
ref_feats[0, :, :] = compute_feats(brick, kernels)
ref_feats[1, :, :] = compute_feats(grass, kernels)
ref_feats[2, :, :] = compute_feats(gravel, kernels)

print('Rotated images matched against references using Gabor filter banks:')

print('original: brick, rotated: 30deg, match result: ', end='')
feats = compute_feats(ndi.rotate(brick, angle=190, reshape=False), kernels)
print(image_names[match(feats, ref_feats)])

```

(continues on next page)

(continued from previous page)

```
print('original: brick, rotated: 70deg, match result: ', end=' ')
feats = compute_feats(ndi.rotate(brick, angle=70, reshape=False), kernels)
print(image_names[match(feats, ref_feats)])

print('original: grass, rotated: 145deg, match result: ', end=' ')
feats = compute_feats(ndi.rotate(grass, angle=145, reshape=False), kernels)
print(image_names[match(feats, ref_feats)])

def power(image, kernel):
    # Normalize images for better comparison.
    image = (image - image.mean()) / image.std()
    return np.sqrt(ndi.convolve(image, np.real(kernel), mode='wrap')**2 +
                  ndi.convolve(image, np.imag(kernel), mode='wrap')**2)

# Plot a selection of the filter bank kernels and their responses.
results = []
kernel_params = []
for theta in (0, 1):
    theta = theta / 4. * np.pi
    for frequency in (0.1, 0.4):
        kernel = gabor_kernel(frequency, theta=theta)
        params = f"theta={theta * 180 / np.pi},\nfrequency={frequency:.2f}"
        kernel_params.append(params)
        # Save kernel and the power image for each image
        results.append((kernel, [power(img, kernel) for img in images]))

fig, axes = plt.subplots(nrows=5, ncols=4, figsize=(5, 6))
plt.gray()

fig.suptitle('Image responses for Gabor filter kernels', fontsize=12)

axes[0][0].axis('off')

# Plot original images
for label, img, ax in zip(image_names, images, axes[0][1:]):
    ax.imshow(img)
    ax.set_title(label, fontsize=9)
    ax.axis('off')

for label, (kernel, powers), ax_row in zip(kernel_params, results, axes[1:]):
    # Plot Gabor kernel
    ax = ax_row[0]
    ax.imshow(np.real(kernel))
    ax.set_ylabel(label, fontsize=7)
    ax.set_xticks([])
    ax.set_yticks([])

    # Plot Gabor responses with the contrast normalized for each filter
    vmin = np.min(powers)
    vmax = np.max(powers)
```

(continues on next page)

(continued from previous page)

```

for patch, ax in zip(powers, ax_row[1:]):
    ax.imshow(patch, vmin=vmin, vmax=vmax)
    ax.axis('off')

plt.show()

```

Total running time of the script: (0 minutes 1.651 seconds)

Local Binary Pattern for texture classification

In this example, we will see how to classify textures based on LBP (Local Binary Pattern). LBP looks at points surrounding a central point and tests whether the surrounding points are greater than or less than the central point (i.e. gives a binary result).

Before trying out LBP on an image, it helps to look at a schematic of LBPs. The below code is just used to plot the schematic.

```

import numpy as np
import matplotlib.pyplot as plt

METHOD = 'uniform'
plt.rcParams['font.size'] = 9

def plot_circle(ax, center, radius, color):
    circle = plt.Circle(center, radius, facecolor=color, edgecolor='0.5')
    ax.add_patch(circle)

def plot_lbp_model(ax, binary_values):
    """Draw the schematic for a local binary pattern."""
    # Geometry spec
    theta = np.deg2rad(45)
    R = 1
    r = 0.15
    w = 1.5
    gray = '0.5'

    # Draw the central pixel.
    plot_circle(ax, (0, 0), radius=r, color=gray)
    # Draw the surrounding pixels.
    for i, facecolor in enumerate(binary_values):
        x = R * np.cos(i * theta)
        y = R * np.sin(i * theta)
        plot_circle(ax, (x, y), radius=r, color=str(facecolor))

    # Draw the pixel grid.
    for x in np.linspace(-w, w, 4):
        ax.axvline(x, color=gray)
        ax.axhline(x, color=gray)

    # Tweak the layout.

```

(continues on next page)

(continued from previous page)

```

ax.axis('image')
ax.axis('off')
size = w + 0.2
ax.set_xlim(-size, size)
ax.set_ylim(-size, size)

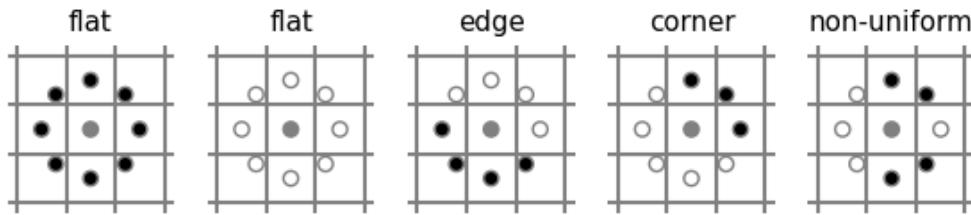
fig, axes = plt.subplots(ncols=5, figsize=(7, 2))

titles = ['flat', 'flat', 'edge', 'corner', 'non-uniform']

binary_patterns = [np.zeros(8),
                   np.ones(8),
                   np.hstack([np.ones(4), np.zeros(4)]),
                   np.hstack([np.zeros(3), np.ones(5)]),
                   [1, 0, 0, 1, 1, 1, 0, 0]]

for ax, values, name in zip(axes, binary_patterns, titles):
    plot_lbp_model(ax, values)
    ax.set_title(name)

```



The figure above shows example results with black (or white) representing pixels that are less (or more) intense than the central pixel. When surrounding pixels are all black or all white, then that image region is flat (i.e. featureless). Groups of continuous black or white pixels are considered “uniform” patterns that can be interpreted as corners or edges. If pixels switch back-and-forth between black and white pixels, the pattern is considered “non-uniform”.

When using LBP to detect texture, you measure a collection of LBPs over an image patch and look at the distribution of these LBPs. Lets apply LBP to a brick texture.

```

from skimage.transform import rotate
from skimage.feature import local_binary_pattern
from skimage import data
from skimage.color import label2rgb

# settings for LBP
radius = 3
n_points = 8 * radius

def overlay_labels(image, lbp, labels):
    mask = np.logical_or.reduce([lbp == each for each in labels])

```

(continues on next page)

(continued from previous page)

```

    return label2rgb(mask, image=image, bg_label=0, alpha=0.5)

def highlight_bars(bars, indexes):
    for i in indexes:
        bars[i].set_facecolor('r')

image = databrick()
lbp = local_binary_pattern(image, n_points, radius, METHOD)

def hist(ax, lbp):
    n_bins = int(lbp.max() + 1)
    return ax.hist(lbp.ravel(), density=True, bins=n_bins, range=(0, n_bins),
                   facecolor='0.5')

# plot histograms of LBP of textures
fig, (ax_img, ax_hist) = plt.subplots(nrows=2, ncols=3, figsize=(9, 6))
plt.gray()

titles = ('edge', 'flat', 'corner')
w = width = radius - 1
edge_labels = range(n_points // 2 - w, n_points // 2 + w + 1)
flat_labels = list(range(0, w + 1)) + list(range(n_points - w, n_points + 2))
i_14 = n_points // 4           # 1/4th of the histogram
i_34 = 3 * (n_points // 4)     # 3/4th of the histogram
corner_labels = (list(range(i_14 - w, i_14 + w + 1)) +
                 list(range(i_34 - w, i_34 + w + 1)))

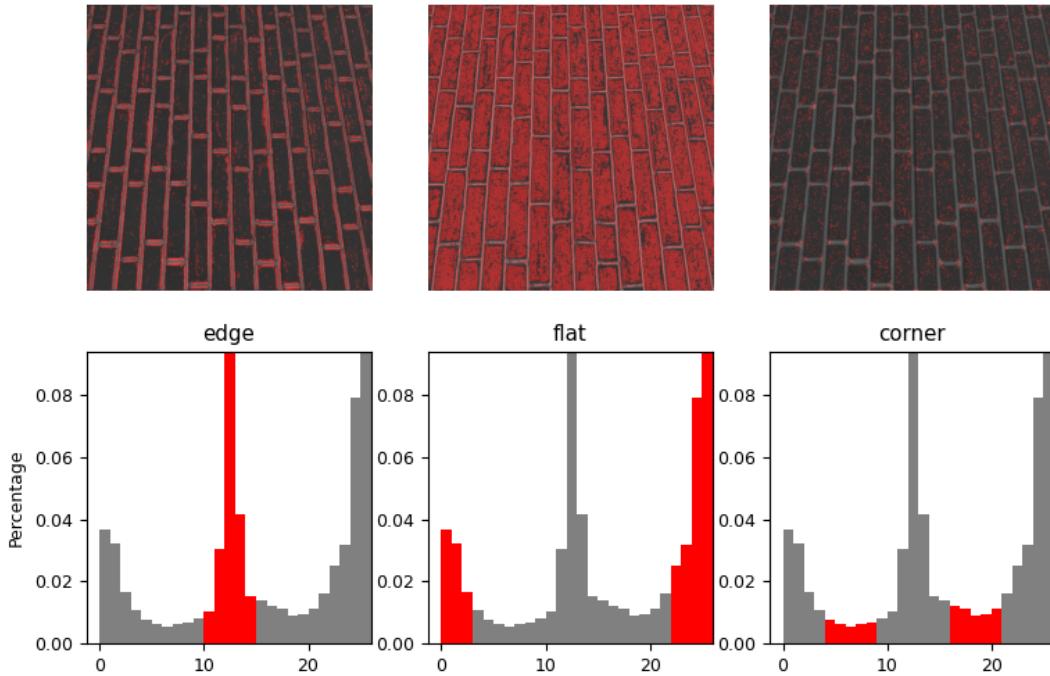
label_sets = (edge_labels, flat_labels, corner_labels)

for ax, labels in zip(ax_img, label_sets):
    ax.imshow(overlay_labels(image, lbp, labels))

for ax, labels, name in zip(ax_hist, label_sets, titles):
    counts, _, bars = hist(ax, lbp)
    highlight_bars(bars, labels)
    ax.set_xlim(right=n_points + 2)
    ax.set_title(name)

    ax_hist[0].set_ylabel('Percentage')
    for ax in ax_img:
        ax.axis('off')

```



The above plot highlights flat, edge-like, and corner-like regions of the image.

The histogram of the LBP result is a good measure to classify textures. Here, we test the histogram distributions against each other using the Kullback-Leibler-Divergence.

```
# settings for LBP
radius = 2
n_points = 8 * radius

def kullback_leibler_divergence(p, q):
    p = np.asarray(p)
    q = np.asarray(q)
    filt = np.logical_and(p != 0, q != 0)
    return np.sum(p[filt] * np.log2(p[filt] / q[filt]))

def match(refs, img):
    best_score = 10
    best_name = None
    lbp = local_binary_pattern(img, n_points, radius, METHOD)
    n_bins = int(lbp.max() + 1)
    hist, _ = np.histogram(lbp, density=True, bins=n_bins, range=(0, n_bins))
    for name, ref in refs.items():
        ref_hist, _ = np.histogram(ref, density=True, bins=n_bins,
                                  range=(0, n_bins))
        score = kullback_leibler_divergence(hist, ref_hist)
```

(continues on next page)

(continued from previous page)

```
if score < best_score:
    best_score = score
    best_name = name
return best_name

brick = data.brick()
grass = data.grass()
gravel = data.gravel()

refs = {
    'brick': local_binary_pattern(brick, n_points, radius, METHOD),
    'grass': local_binary_pattern(grass, n_points, radius, METHOD),
    'gravel': local_binary_pattern(gravel, n_points, radius, METHOD)
}

# classify rotated textures
print('Rotated images matched against references using LBP:')
print('original: brick, rotated: 30deg, match result: ',
      match(refs, rotate(brick, angle=30, resize=False)))
print('original: brick, rotated: 70deg, match result: ',
      match(refs, rotate(brick, angle=70, resize=False)))
print('original: grass, rotated: 145deg, match result: ',
      match(refs, rotate(grass, angle=145, resize=False)))

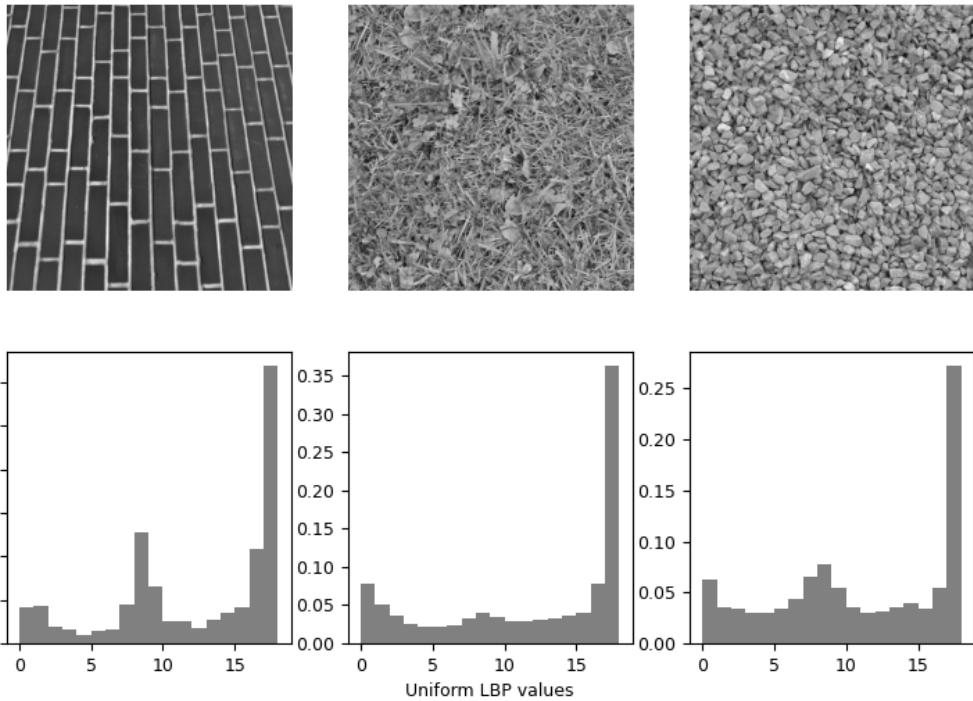
# plot histograms of LBP of textures
fig, ((ax1, ax2, ax3), (ax4, ax5, ax6)) = plt.subplots(nrows=2, ncols=3,
                                                       figsize=(9, 6))
plt.gray()

ax1.imshow(brick)
ax1.axis('off')
hist(ax4, refs['brick'])
ax4.set_ylabel('Percentage')

ax2.imshow(grass)
ax2.axis('off')
hist(ax5, refs['grass'])
ax5.set_xlabel('Uniform LBP values')

ax3.imshow(gravel)
ax3.axis('off')
hist(ax6, refs['gravel'])

plt.show()
```



```
Rotated images matched against references using LBP:  

/usr/local/lib/python3.8/site-packages/skimage/feature/text.py:353: UserWarning:
```

```
Applying `local_binary_pattern` to floating-point images may give unexpected results.  

when small numerical differences between adjacent pixels are present. It is  

recommended to use this function with images of integer dtype.
```

```
original: brick, rotated: 30deg, match result: brick  

original: brick, rotated: 70deg, match result: brick  

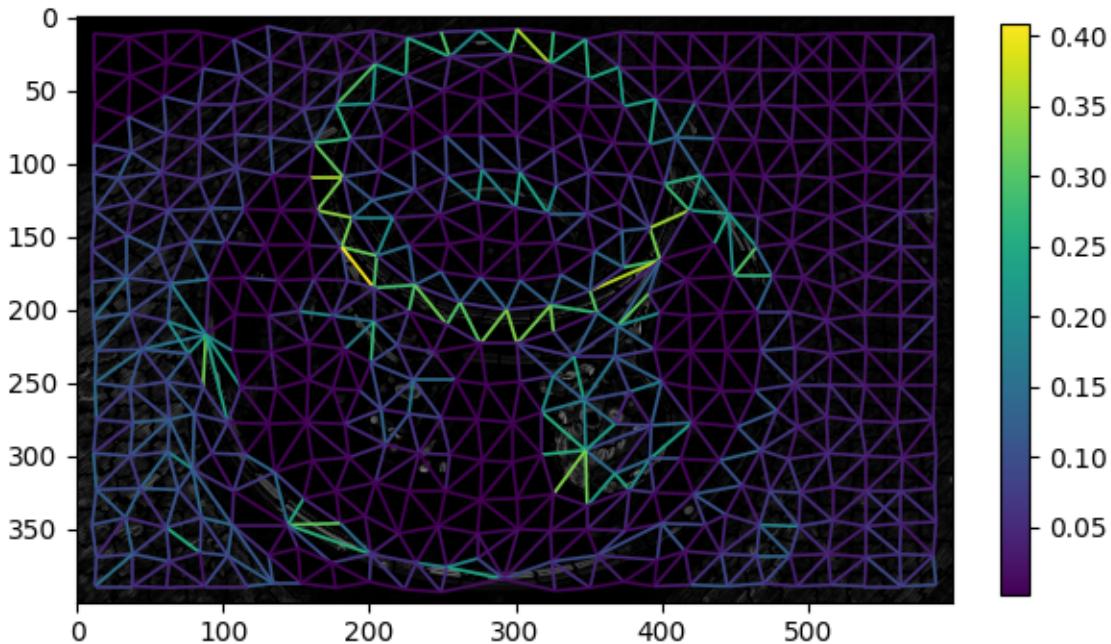
original: grass, rotated: 145deg, match result: grass
```

Total running time of the script: (0 minutes 1.584 seconds)

Segmentation of objects

Region Boundary based RAGs

Construct a region boundary RAG with the `rag_boundary` function. The function `skimage.graph.rag_boundary()` takes an `edge_map` argument, which gives the significance of a feature (such as edges) being present at each pixel. In a region boundary RAG, the edge weight between two regions is the average value of the corresponding pixels in `edge_map` along their shared boundary.



```

from skimage import graph
from skimage import data, segmentation, color, filters, io
from matplotlib import pyplot as plt

img = data.coffee()
gimg = color.rgb2gray(img)

labels = segmentation.slic(img, compactness=30, n_segments=400, start_label=1)
edges = filters.sobel(gimg)
edges_rgb = color.gray2rgb(edges)

g = graph.rag_boundary(labels, edges)
lc = graph.show_rag(labels, g, edges_rgb, img_cmap=None, edge_cmap='viridis',
                     edge_width=1.2)

plt.colorbar(lc, fraction=0.03)
io.show()

```

Total running time of the script: (0 minutes 0.512 seconds)

RAG Thresholding

This example constructs a Region Adjacency Graph (RAG) and merges regions which are similar in color. We construct a RAG and define edges as the difference in mean color. We then join regions with similar mean color.



```
from skimage import data, segmentation, color
from skimage import graph
from matplotlib import pyplot as plt

img = data.coffee()

labels1 = segmentation.slic(img, compactness=30, n_segments=400, start_label=1)
out1 = color.label2rgb(labels1, img, kind='avg', bg_label=0)

g = graph.rag_mean_color(img, labels1)
labels2 = graph.cut_threshold(labels1, g, 29)
out2 = color.label2rgb(labels2, img, kind='avg', bg_label=0)

fig, ax = plt.subplots(nrows=2, sharex=True, sharey=True,
                      figsize=(6, 8))

ax[0].imshow(out1)
ax[1].imshow(out2)

for a in ax:
    a.axis('off')

plt.tight_layout()
```

Total running time of the script: (0 minutes 2.131 seconds)

Normalized Cut

This example constructs a Region Adjacency Graph (RAG) and recursively performs a Normalized Cut on it¹.

¹ Shi, J.; Malik, J., “Normalized cuts and image segmentation”, Pattern Analysis and Machine Intelligence, IEEE Transactions on, vol. 22, no. 8, pp. 888-905, August 2000.

References



```
from skimage import data, segmentation, color
from skimage import graph
from matplotlib import pyplot as plt

img = data.coffee()

labels1 = segmentation.slic(img, compactness=30, n_segments=400,
                           start_label=1)
out1 = color.label2rgb(labels1, img, kind='avg', bg_label=0)

g = graph.rag_mean_color(img, labels1, mode='similarity')
labels2 = graph.cut_normalized(labels1, g)
out2 = color.label2rgb(labels2, img, kind='avg', bg_label=0)

fig, ax = plt.subplots(nrows=2, sharex=True, sharey=True, figsize=(6, 8))

ax[0].imshow(out1)
ax[1].imshow(out2)

for a in ax:
    a.axis('off')

plt.tight_layout()
```

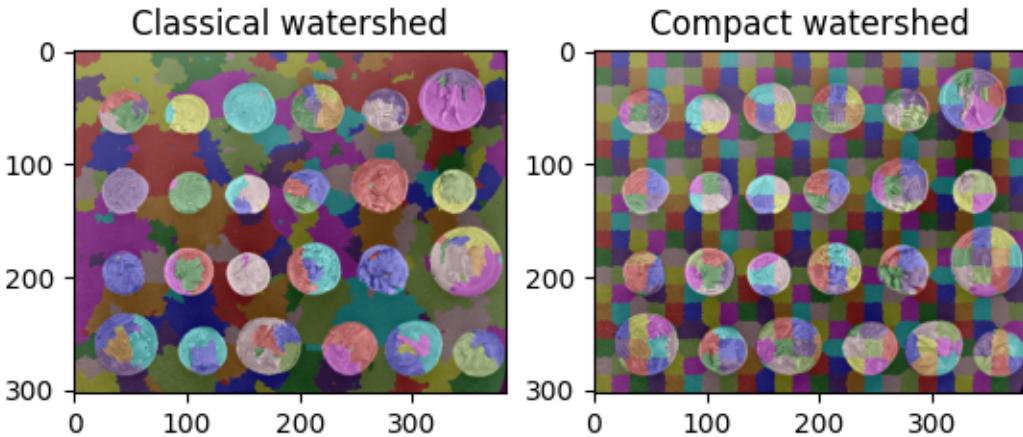
Total running time of the script: (0 minutes 3.333 seconds)

Find Regular Segments Using Compact Watershed

The watershed transform is commonly used as a starting point for many segmentation algorithms. However, without a judicious choice of seeds, it can produce very uneven fragment sizes, which can be difficult to deal with in downstream analyses.

The *compact* watershed transform remedies this by favoring seeds that are close to the pixel being considered.

Both algorithms are implemented in the `skimage.morphology.watershed()` function. To use the compact form, simply pass a `compactness` value greater than 0.



```
import numpy as np
from skimage import data, util, filters, color
from skimage.segmentation import watershed
import matplotlib.pyplot as plt

coins = data.coins()
edges = filters.sobel(coins)

grid = util.regular_grid(coins.shape, n_points=468)

seeds = np.zeros(coins.shape, dtype=int)
seeds[grid] = np.arange(seeds[grid].size).reshape(seeds[grid].shape) + 1

w0 = watershed(edges, seeds)
w1 = watershed(edges, seeds, compactness=0.01)

fig, (ax0, ax1) = plt.subplots(1, 2)

ax0.imshow(color.label2rgb(w0, coins, bg_label=-1))
ax0.set_title('Classical watershed')

ax1.imshow(color.label2rgb(w1, coins, bg_label=-1))
ax1.set_title('Compact watershed')
```

(continues on next page)

(continued from previous page)

```
plt.show()
```

Total running time of the script: (0 minutes 0.285 seconds)

Thresholding

Thresholding is used to create a binary image from a grayscale image¹.

See also:

A more comprehensive presentation on *Thresholding*

```
import matplotlib.pyplot as plt
from skimage import data
from skimage.filters import threshold_otsu
```

We illustrate how to apply one of these thresholding algorithms. Otsu's method² calculates an “optimal” threshold (marked by a red line in the histogram below) by maximizing the variance between two classes of pixels, which are separated by the threshold. Equivalently, this threshold minimizes the intra-class variance.

```
image = data.camera()
thresh = threshold_otsu(image)
binary = image > thresh

fig, axes = plt.subplots(ncols=3, figsize=(8, 2.5))
ax = axes.ravel()
ax[0] = plt.subplot(1, 3, 1)
ax[1] = plt.subplot(1, 3, 2)
ax[2] = plt.subplot(1, 3, 3, sharex=ax[0], sharey=ax[0])

ax[0].imshow(image, cmap=plt.cm.gray)
ax[0].set_title('Original')
ax[0].axis('off')

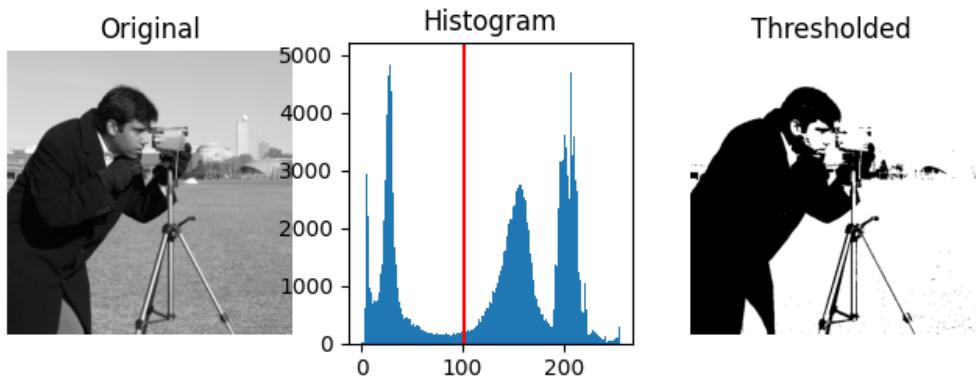
ax[1].hist(image.ravel(), bins=256)
ax[1].set_title('Histogram')
ax[1].axvline(thresh, color='r')

ax[2].imshow(binary, cmap=plt.cm.gray)
ax[2].set_title('Thresholded')
ax[2].axis('off')

plt.show()
```

¹ https://en.wikipedia.org/wiki/Thresholding_%28image_processing%29

² https://en.wikipedia.org/wiki/Otsu's_method

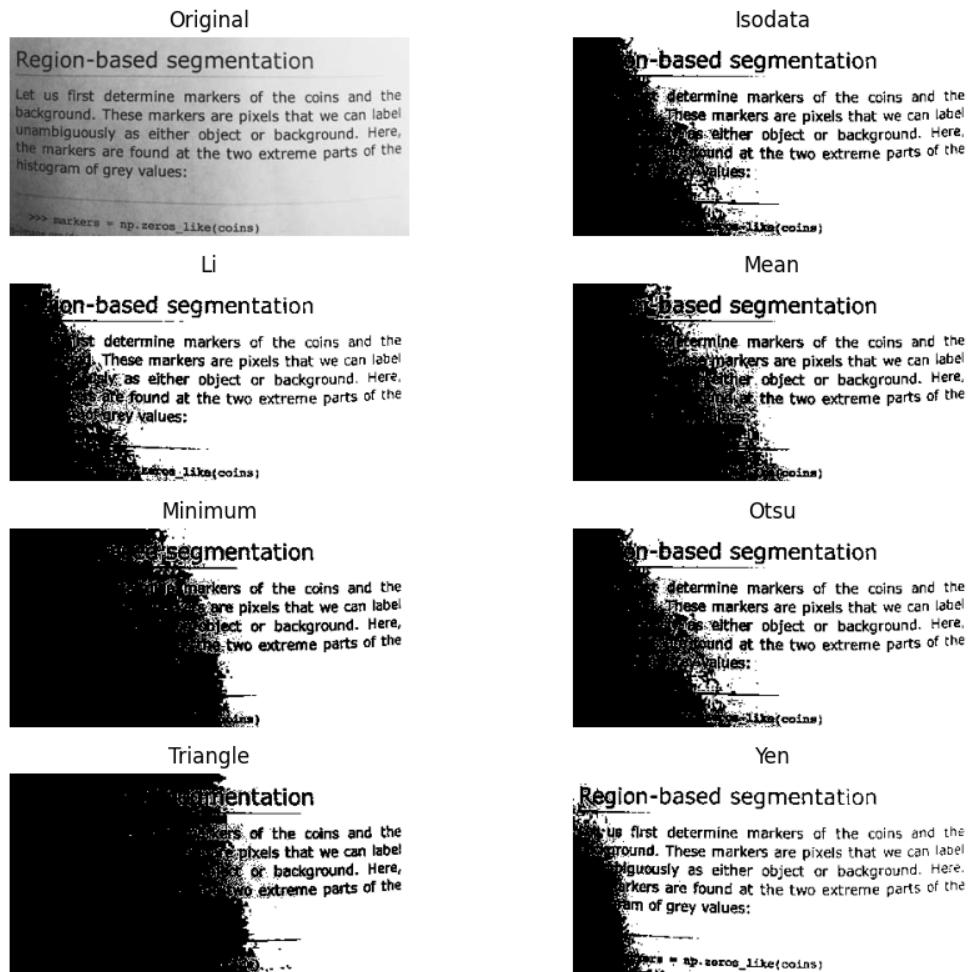


```
/github/workspace/build/scikit-image/doc/examples/segmentation/plot_thresholding.py:39:  
  ↪MatplotlibDeprecationWarning:
```

```
Auto-removal of overlapping axes is deprecated since 3.6 and will be removed two minor  
  ↪releases later; explicitly call ax.remove() as needed.
```

If you are not familiar with the details of the different algorithms and the underlying assumptions, it is often difficult to know which algorithm will give the best results. Therefore, Scikit-image includes a function to evaluate thresholding algorithms provided by the library. At a glance, you can select the best algorithm for your data without a deep understanding of their mechanisms.

```
from skimage.filters import try_all_threshold  
  
img = data.page()  
  
fig, ax = try_all_threshold(img, figsize=(10, 8), verbose=False)  
plt.show()
```

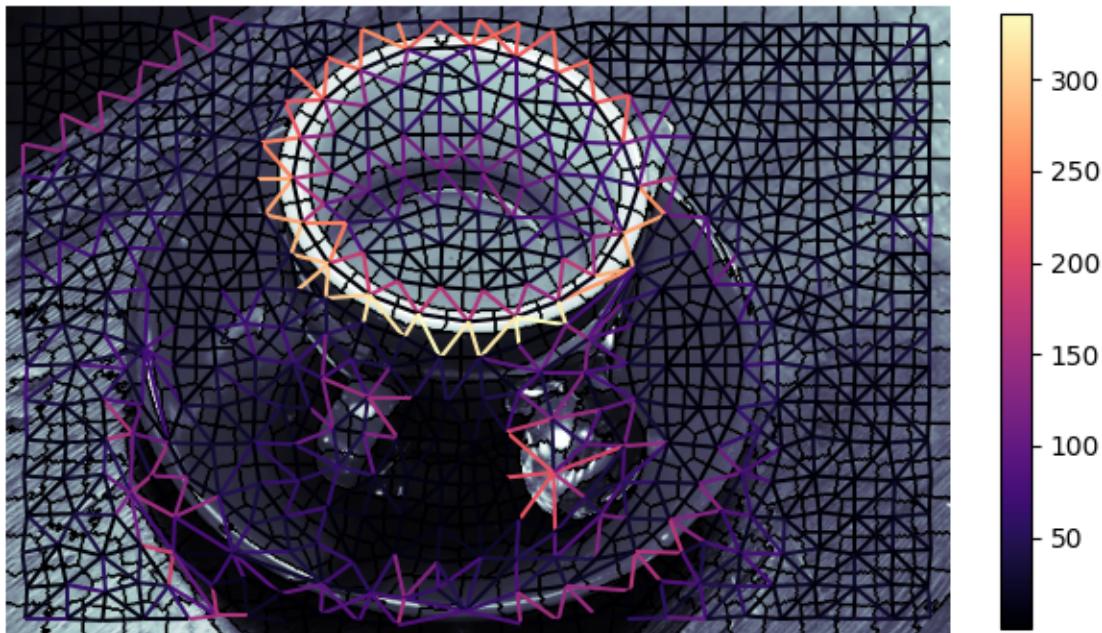


Total running time of the script: (0 minutes 0.982 seconds)

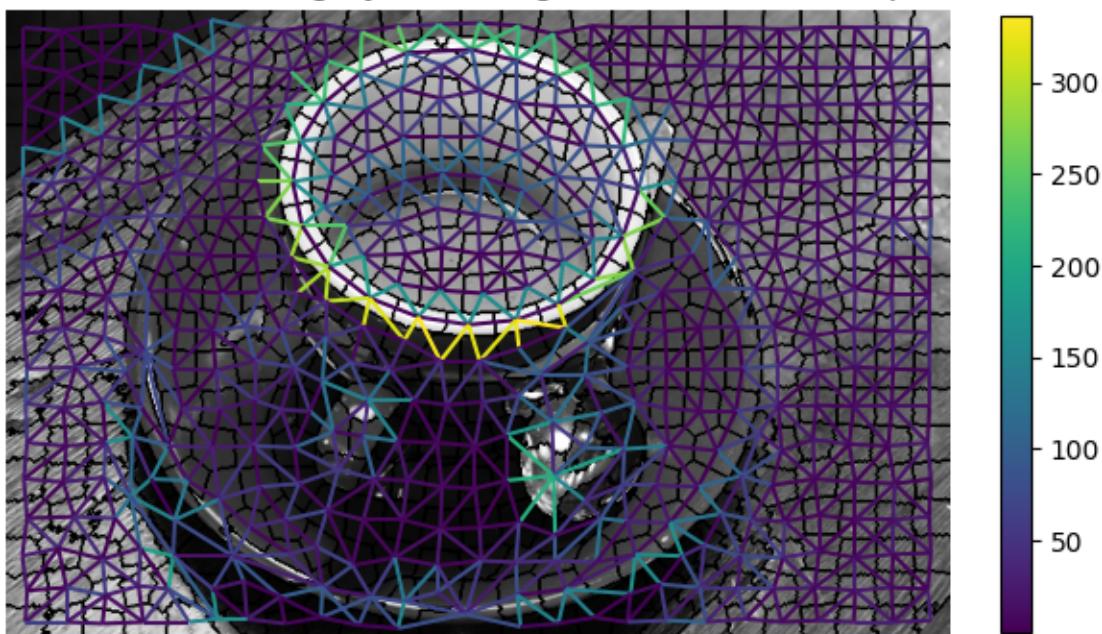
Drawing Region Adjacency Graphs (RAGs)

This example constructs a Region Adjacency Graph (RAG) and draws it with the `rag_draw` method.

RAG drawn with default settings



RAG drawn with grayscale image and viridis colormap



```
from skimage import data, segmentation
from skimage import graph
from matplotlib import pyplot as plt
```

(continues on next page)

(continued from previous page)

```



```

Total running time of the script: (0 minutes 2.176 seconds)

Chan-Vese Segmentation

The Chan-Vese segmentation algorithm is designed to segment objects without clearly defined boundaries. This algorithm is based on level sets that are evolved iteratively to minimize an energy, which is defined by weighted values corresponding to the sum of differences intensity from the average value outside the segmented region, the sum of differences from the average value inside the segmented region, and a term which is dependent on the length of the boundary of the segmented region.

This algorithm was first proposed by Tony Chan and Luminita Vese, in a publication entitled “An Active Contour Model Without Edges”¹. See also^{2 3}.

This implementation of the algorithm is somewhat simplified in the sense that the area factor ‘nu’ described in the original paper is not implemented, and is only suitable for grayscale images.

Typical values for `lambda1` and `lambda2` are 1. If the ‘background’ is very different from the segmented object in terms of distribution (for example, a uniform black image with figures of varying intensity), then these values should be different from each other.

Typical values for `mu` are between 0 and 1, though higher values can be used when dealing with shapes with very ill-defined contours.

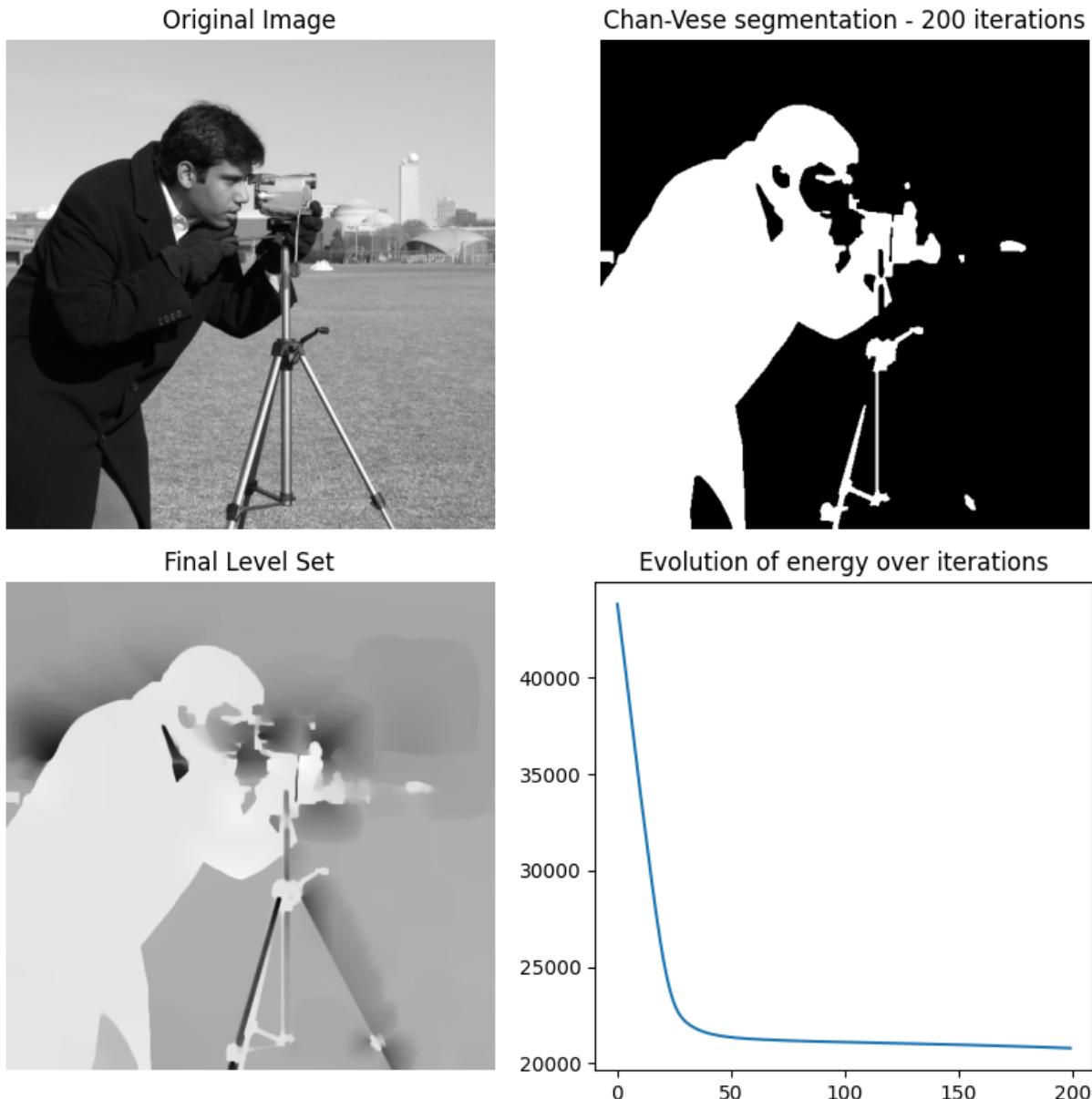
The algorithm also returns a list of values that corresponds to the energy at each iteration. This can be used to adjust the various parameters described above.

¹ An Active Contour Model without Edges, Tony Chan and Luminita Vese, Scale-Space Theories in Computer Vision, 1999, DOI:10.1007/3-540-48236-9_13

² Chan-Vese Segmentation, Pascal Getreuer, Image Processing On Line, 2 (2012), pp. 214-224, DOI:10.5201/ipol.2012.g-cv

³ The Chan-Vese Algorithm - Project Report, Rami Cohen, 2011 arXiv:1107.2782

References



```
import matplotlib.pyplot as plt
from skimage import data, img_as_float
from skimage.segmentation import chan_vese

image = img_as_float(data.camera())
# Feel free to play around with the parameters to see how they impact the result
cv = chan_vese(image, mu=0.25, lambda1=1, lambda2=1, tol=1e-3,
               max_num_iter=200, dt=0.5, init_level_set="checkerboard",
               extended_output=True)
```

(continues on next page)

(continued from previous page)

```

fig, axes = plt.subplots(2, 2, figsize=(8, 8))
ax = axes.flatten()

ax[0].imshow(image, cmap="gray")
ax[0].set_axis_off()
ax[0].set_title("Original Image", fontsize=12)

ax[1].imshow(cv[0], cmap="gray")
ax[1].set_axis_off()
title = f'Chan-Vese segmentation - {len(cv[2])} iterations'
ax[1].set_title(title, fontsize=12)

ax[2].imshow(cv[1], cmap="gray")
ax[2].set_axis_off()
ax[2].set_title("Final Level Set", fontsize=12)

ax[3].plot(cv[2])
ax[3].set_title("Evolution of energy over iterations", fontsize=12)

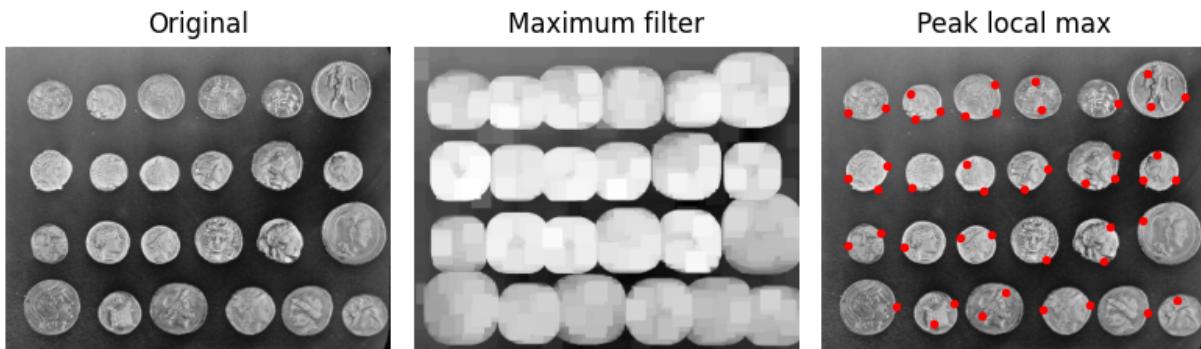
fig.tight_layout()
plt.show()

```

Total running time of the script: (0 minutes 4.958 seconds)

Finding local maxima

The `peak_local_max` function returns the coordinates of local peaks (maxima) in an image. Internally, a maximum filter is used for finding local maxima. This operation dilates the original image and merges neighboring local maxima closer than the size of the dilation. Locations where the original image is equal to the dilated image are returned as local maxima.



```

from scipy import ndimage as ndi
import matplotlib.pyplot as plt
from skimage.feature import peak_local_max
from skimage import data, img_as_float

im = img_as_float(data.coins())

```

(continues on next page)

(continued from previous page)

```
# image_max is the dilation of im with a 20*20 structuring element
# It is used within peak_local_max function
image_max = ndi.maximum_filter(im, size=20, mode='constant')

# Comparison between image_max and im to find the coordinates of local maxima
coordinates = peak_local_max(im, min_distance=20)

# display results
fig, axes = plt.subplots(1, 3, figsize=(8, 3), sharex=True, sharey=True)
ax = axes.ravel()
ax[0].imshow(im, cmap=plt.cm.gray)
ax[0].axis('off')
ax[0].set_title('Original')

ax[1].imshow(image_max, cmap=plt.cm.gray)
ax[1].axis('off')
ax[1].set_title('Maximum filter')

ax[2].imshow(im, cmap=plt.cm.gray)
ax[2].autoscale(False)
ax[2].plot(coordinates[:, 1], coordinates[:, 0], 'r.')
ax[2].axis('off')
ax[2].set_title('Peak local max')

fig.tight_layout()

plt.show()
```

Total running time of the script: (0 minutes 0.206 seconds)

Niblack and Sauvola Thresholding

Niblack and Sauvola thresholds are local thresholding techniques that are useful for images where the background is not uniform, especially for text recognition^{1,2}. Instead of calculating a single global threshold for the entire image, several thresholds are calculated for every pixel by using specific formulae that take into account the mean and standard deviation of the local neighborhood (defined by a window centered around the pixel).

Here, we binarize an image using these algorithms compare it to a common global thresholding technique. Parameter `window_size` determines the size of the window that contains the surrounding pixels.

¹ Niblack, W (1986), An introduction to Digital Image Processing, Prentice-Hall.

² J. Sauvola and M. Pietikainen, "Adaptive document image binarization," Pattern Recognition 33(2), pp. 225-236, 2000. DOI:10.1016/S0031-3203(99)00055-2

Original

Region-based segmentation

Let us first determine markers of the coins and the background. These markers are pixels that we can label unambiguously as either object or background. Here, the markers are found at the two extreme parts of the histogram of grey values:

```
>>> markers = np.zeros_like(coins)
```

Global Threshold

Region-based segmentation

determine markers of the coins and the background. These markers are pixels that we can label unambiguously as either object or background. Here, the markers are found at the two extreme parts of the histogram of grey values:

```
>>> markers = np.zeros_like(coins)
```

Niblack Threshold

Region-based segmentation

Let us first determine markers of the coins and the background. These markers are pixels that we can label unambiguously as either object or background. Here, the markers are found at the two extreme parts of the histogram of grey values:

```
>>> markers = np.zeros_like(coins)
```

Sauvola Threshold

Region-based segmentation

Let us first determine markers of the coins and the background. These markers are pixels that we can label unambiguously as either object or background. Here, the markers are found at the two extreme parts of the histogram of grey values:

```
>>> markers = np.zeros_like(coins)
```

```
import matplotlib
import matplotlib.pyplot as plt

from skimage.data import page
from skimage.filters import (threshold_otsu, threshold_niblack,
                             threshold_sauvola)

matplotlib.rcParams['font.size'] = 9

image = page()
binary_global = image > threshold_otsu(image)

window_size = 25
thresh_niblack = threshold_niblack(image, window_size=window_size, k=0.8)
thresh_sauvola = threshold_sauvola(image, window_size=window_size)
```

(continues on next page)

(continued from previous page)

```

binary_niblack = image > thresh_niblack
binary_sauvola = image > thresh_sauvola

plt.figure(figsize=(8, 7))
plt.subplot(2, 2, 1)
plt.imshow(image, cmap=plt.cm.gray)
plt.title('Original')
plt.axis('off')

plt.subplot(2, 2, 2)
plt.title('Global Threshold')
plt.imshow(binary_global, cmap=plt.cm.gray)
plt.axis('off')

plt.subplot(2, 2, 3)
plt.imshow(binary_niblack, cmap=plt.cm.gray)
plt.title('Niblack Threshold')
plt.axis('off')

plt.subplot(2, 2, 4)
plt.imshow(binary_sauvola, cmap=plt.cm.gray)
plt.title('Sauvola Threshold')
plt.axis('off')

plt.show()

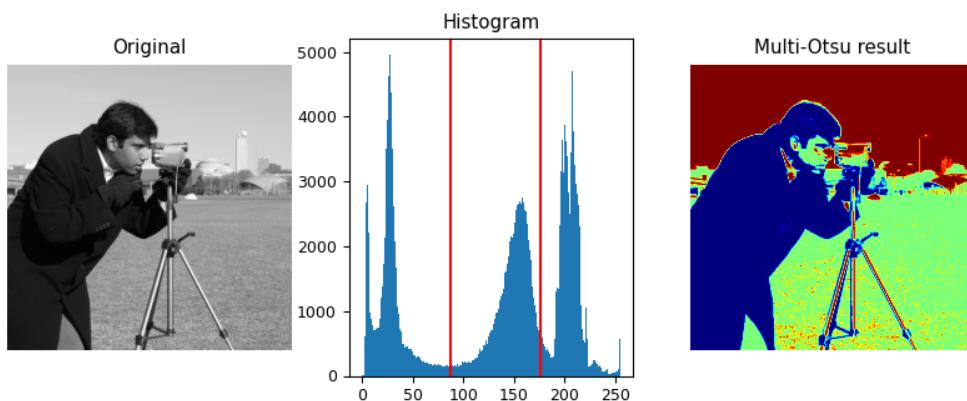
```

Total running time of the script: (0 minutes 0.170 seconds)

Multi-Otsu Thresholding

The multi-Otsu threshold¹ is a thresholding algorithm that is used to separate the pixels of an input image into several different classes, each one obtained according to the intensity of the gray levels within the image.

Multi-Otsu calculates several thresholds, determined by the number of desired classes. The default number of classes is 3: for obtaining three classes, the algorithm returns two threshold values. They are represented by a red line in the histogram below.



¹ Liao, P-S., Chen, T-S. and Chung, P-C., "A fast algorithm for multilevel thresholding", Journal of Information Science and Engineering 17 (5): 713-727, 2001. Available at: <https://ftp.iis.sinica.edu.tw/JISE/2001/200109_01.pdf>.

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

from skimage import data
from skimage.filters import threshold_multiotsu

# Setting the font size for all plots.
matplotlib.rcParams['font.size'] = 9

# The input image.
image = data.camera()

# Applying multi-Otsu threshold for the default value, generating
# three classes.
thresholds = threshold_multiotsu(image)

# Using the threshold values, we generate the three regions.
regions = np.digitize(image, bins=thresholds)

fig, ax = plt.subplots(nrows=1, ncols=3, figsize=(10, 3.5))

# Plotting the original image.
ax[0].imshow(image, cmap='gray')
ax[0].set_title('Original')
ax[0].axis('off')

# Plotting the histogram and the two thresholds obtained from
# multi-Otsu.
ax[1].hist(image.ravel(), bins=255)
ax[1].set_title('Histogram')
for thresh in thresholds:
    ax[1].axvline(thresh, color='r')

# Plotting the Multi Otsu result.
ax[2].imshow(regions, cmap='jet')
ax[2].set_title('Multi-Otsu result')
ax[2].axis('off')

plt.subplots_adjust()
plt.show()
```

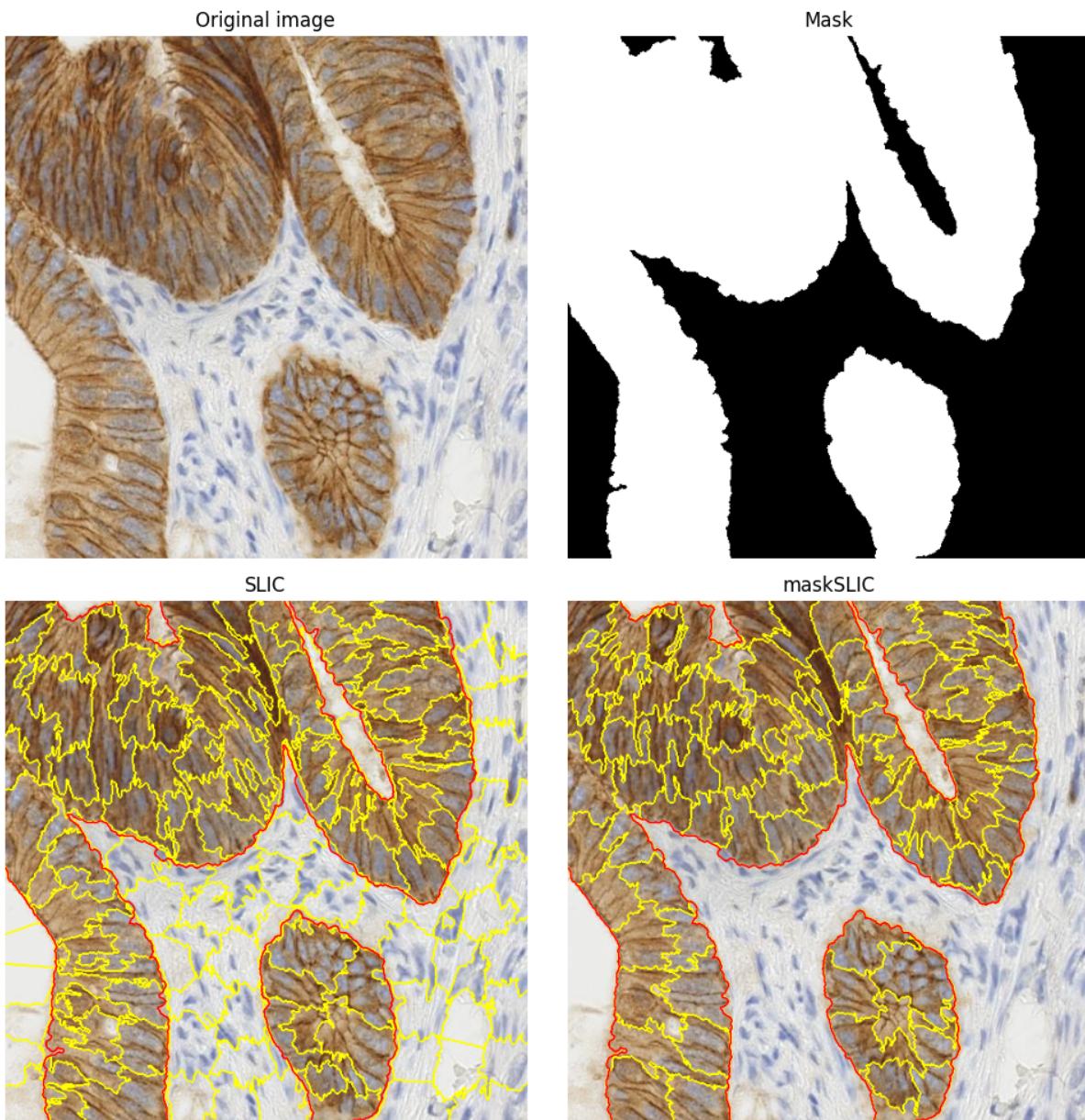
Total running time of the script: (0 minutes 0.379 seconds)

Apply maskSLIC vs SLIC

This example is about comparing the segmentations obtained using the plain SLIC method¹ and its masked version maskSLIC².

To illustrate these segmentation methods, we use an image of biological tissue with immunohistochemical (IHC) staining. The same biomedical image is used in the example on how to *Separate colors in immunohistochemical staining*.

The maskSLIC method is an extension of the SLIC method for the generation of superpixels in a region of interest. maskSLIC is able to overcome border problems that affects SLIC method, particularly in case of irregular mask.



¹ Radhakrishna Achanta, Appu Shaji, Kevin Smith, Aurelien Lucchi, Pascal Fua, and Sabine Süsstrunk, “SLIC Superpixels Compared to State-of-the-Art Superpixel Methods,” IEEE TPAMI, 2012, DOI:10.1109/TPAMI.2012.120

² Irving, Benjamin. “maskSLIC: regional superpixel generation with application to local pathology characterisation in medical images,” 2016, arXiv:1606.09518

```

import matplotlib.pyplot as plt

from skimage import data
from skimage import color
from skimage import morphology
from skimage import segmentation

# Input data
img = data.immunohistochemistry()

# Compute a mask
lum = color.rgb2gray(img)
mask = morphology.remove_small_holes(
    morphology.remove_small_objects(
        lum < 0.7, 500),
    500)

mask = morphology.opening(mask, morphology.disk(3))

# SLIC result
slic = segmentation.slic(img, n_segments=200, start_label=1)

# maskSLIC result
m_slic = segmentation.slic(img, n_segments=100, mask=mask, start_label=1)

# Display result
fig, ax_arr = plt.subplots(2, 2, sharex=True, sharey=True, figsize=(10, 10))
ax1, ax2, ax3, ax4 = ax_arr.ravel()

ax1.imshow(img)
ax1.set_title('Original image')

ax2.imshow(mask, cmap='gray')
ax2.set_title('Mask')

ax3.imshow(segmentation.mark_boundaries(img, slic))
ax3.contour(mask, colors='red', linewidths=1)
ax3.set_title('SLIC')

ax4.imshow(segmentation.mark_boundaries(img, m_slic))
ax4.contour(mask, colors='red', linewidths=1)
ax4.set_title('maskSLIC')

for ax in ax_arr.ravel():
    ax.set_axis_off()

plt.tight_layout()
plt.show()

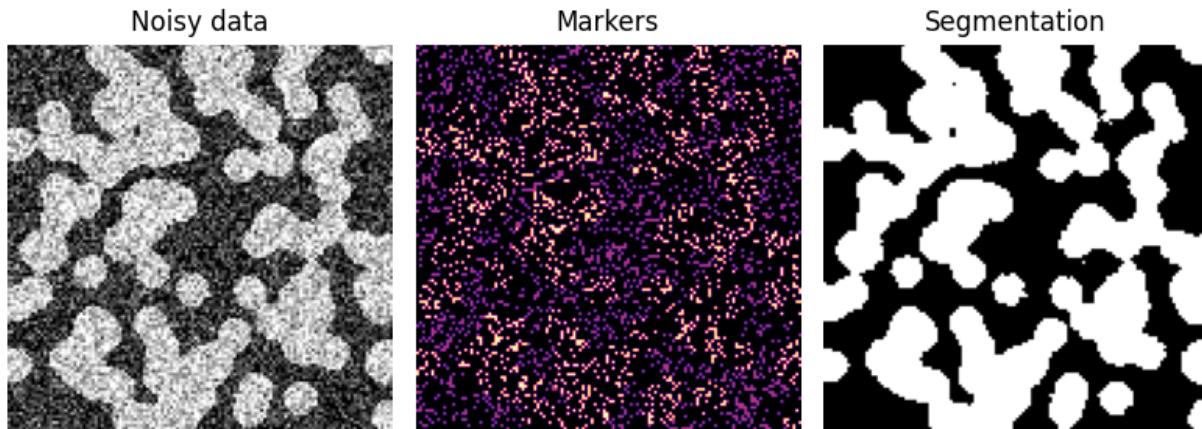
```

Total running time of the script: (0 minutes 1.335 seconds)

Random walker segmentation

The random walker algorithm¹ determines the segmentation of an image from a set of markers labeling several phases (2 or more). An anisotropic diffusion equation is solved with tracers initiated at the markers' position. The local diffusivity coefficient is greater if neighboring pixels have similar values, so that diffusion is difficult across high gradients. The label of each unknown pixel is attributed to the label of the known marker that has the highest probability to be reached first during this diffusion process.

In this example, two phases are clearly visible, but the data are too noisy to perform the segmentation from the histogram only. We determine markers of the two phases from the extreme tails of the histogram of gray values, and use the random walker for the segmentation.



```

import numpy as np
import matplotlib.pyplot as plt

from skimage.segmentation import random_walker
from skimage.data import binary_blobs
from skimage.exposure import rescale_intensity
import skimage

rng = np.random.default_rng()

# Generate noisy synthetic data
data = skimage.img_as_float(binary_blobs(length=128, rng=rng))
sigma = 0.35
data += rng.normal(loc=0, scale=sigma, size=data.shape)
data = rescale_intensity(data, in_range=(-sigma, 1 + sigma),
                         out_range=(-1, 1))

# The range of the binary image spans over (-1, 1).
# We choose the hottest and the coldest pixels as markers.
markers = np.zeros(data.shape, dtype=np.uint)
markers[data < -0.95] = 1
markers[data > 0.95] = 2

# Run random walker algorithm

```

(continues on next page)

¹ Random walks for image segmentation, Leo Grady, IEEE Trans. Pattern Anal. Mach. Intell. 2006 Nov; 28(11):1768-83
DOI:10.1109/TPAMI.2006.233

(continued from previous page)

```

labels = random_walker(data, markers, beta=10, mode='bf')

# Plot results
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(8, 3.2),
                                    sharex=True, sharey=True)
ax1.imshow(data, cmap='gray')
ax1.axis('off')
ax1.set_title('Noisy data')
ax2.imshow(markers, cmap='magma')
ax2.axis('off')
ax2.set_title('Markers')
ax3.imshow(labels, cmap='gray')
ax3.axis('off')
ax3.set_title('Segmentation')

fig.tight_layout()
plt.show()

```

Total running time of the script: (0 minutes 0.235 seconds)

Expand segmentation labels without overlap

Given several connected components represented by a label image, these connected components can be expanded into background regions using `skimage.segmentation.expand_labels()`. In contrast to `skimage.morphology.dilation()` this method will not let connected components expand into neighboring connected components with lower label number.



```

import matplotlib.pyplot as plt
import numpy as np
from skimage import data

```

(continues on next page)

(continued from previous page)

```
from skimage.color import label2rgb
from skimage.filters import sobel
from skimage.measure import label
from skimage.segmentation import expand_labels, watershed

coins = data.coins()

# Make segmentation using edge-detection and watershed.
edges = sobel(coins)

# Identify some background and foreground pixels from the intensity values.
# These pixels are used as seeds for watershed.
markers = np.zeros_like(coins)
foreground, background = 1, 2
markers[coins < 30.0] = background
markers[coins > 150.0] = foreground

ws = watershed(edges, markers)
seg1 = label(ws == foreground)

expanded = expand_labels(seg1, distance=10)

# Show the segmentations.
fig, axes = plt.subplots(
    nrows=1,
    ncols=3,
    figsize=(9, 5),
    sharex=True,
    sharey=True,
)

axes[0].imshow(coins, cmap="Greys_r")
axes[0].set_title("Original")

color1 = label2rgb(seg1, image=coins, bg_label=0)
axes[1].imshow(color1)
axes[1].set_title("Sobel+Watershed")

color2 = label2rgb(expanded, image=coins, bg_label=0)
axes[2].imshow(color2)
axes[2].set_title("Expanded labels")

for a in axes:
    a.axis("off")
fig.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.313 seconds)

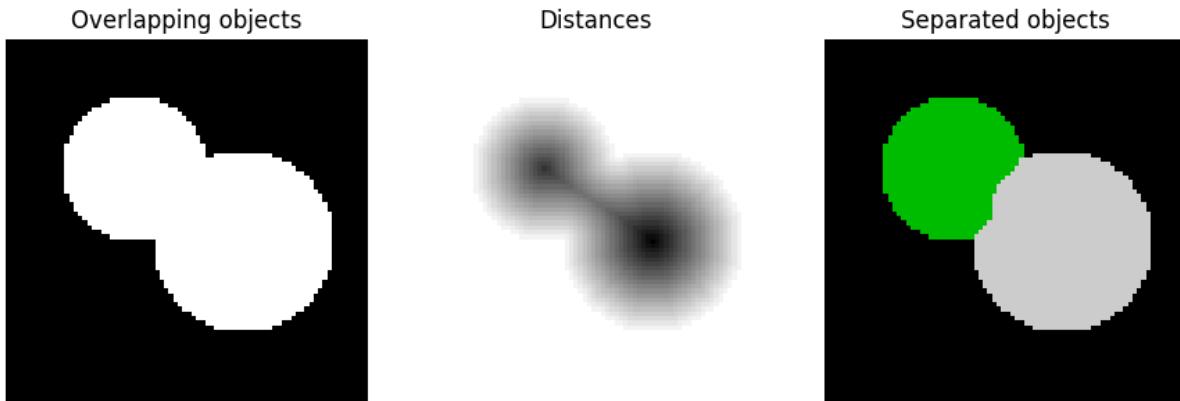
Watershed segmentation

The watershed is a classical algorithm used for **segmentation**, that is, for separating different objects in an image.

Starting from user-defined markers, the watershed algorithm treats pixels values as a local topography (elevation). The algorithm floods basins from the markers until basins attributed to different markers meet on watershed lines. In many cases, markers are chosen as local minima of the image, from which basins are flooded.

In the example below, two overlapping circles are to be separated. To do so, one computes an image that is the distance to the background. The maxima of this distance (i.e., the minima of the opposite of the distance) are chosen as markers and the flooding of basins from such markers separates the two circles along a watershed line.

See [Wikipedia](#) for more details on the algorithm.



```

import numpy as np
import matplotlib.pyplot as plt
from scipy import ndimage as ndi

from skimage.segmentation import watershed
from skimage.feature import peak_local_max


# Generate an initial image with two overlapping circles
x, y = np.indices((80, 80))
x1, y1, x2, y2 = 28, 28, 44, 52
r1, r2 = 16, 20
mask_circle1 = (x - x1)**2 + (y - y1)**2 < r1**2
mask_circle2 = (x - x2)**2 + (y - y2)**2 < r2**2
image = np.logical_or(mask_circle1, mask_circle2)

# Now we want to separate the two objects in image
# Generate the markers as local maxima of the distance to the background
distance = ndi.distance_transform_edt(image)
coords = peak_local_max(distance, footprint=np.ones((3, 3)), labels=image)
mask = np.zeros(distance.shape, dtype=bool)
mask[tuple(coords.T)] = True
markers, _ = ndi.label(mask)
labels = watershed(-distance, markers, mask=image)

fig, axes = plt.subplots(ncols=3, figsize=(9, 3), sharex=True, sharey=True)
ax = axes.ravel()

```

(continues on next page)

(continued from previous page)

```
ax[0].imshow(image, cmap=plt.cm.gray)
ax[0].set_title('Overlapping objects')
ax[1].imshow(-distance, cmap=plt.cm.gray)
ax[1].set_title('Distances')
ax[2].imshow(labels, cmap=plt.cm.nipy_spectral)
ax[2].set_title('Separated objects')

for a in ax:
    a.set_axis_off()

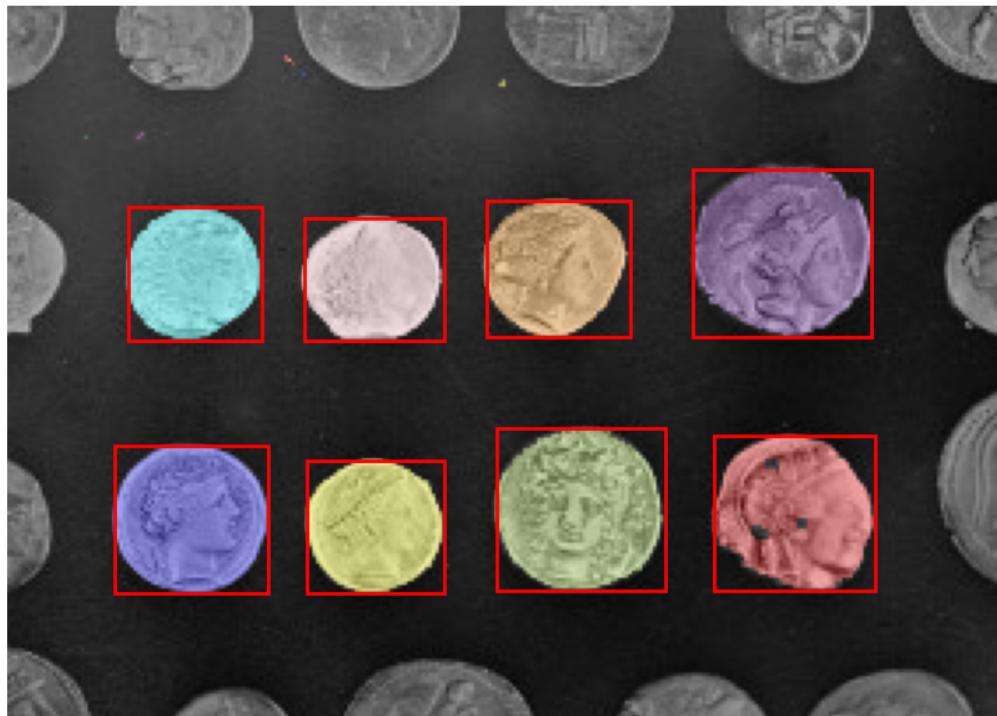
fig.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.151 seconds)

Label image regions

This example shows how to segment an image with image labelling. The following steps are applied:

1. Thresholding with automatic Otsu method
2. Close small holes with binary closing
3. Remove artifacts touching image border
4. Measure image regions to filter small objects



```
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
```

(continues on next page)

(continued from previous page)

```
from skimage import data
from skimage.filters import threshold_otsu
from skimage.segmentation import clear_border
from skimage.measure import label, regionprops
from skimage.morphology import closing, square
from skimage.color import label2rgb

image = data.coins()[50:-50, 50:-50]

# apply threshold
thresh = threshold_otsu(image)
bw = closing(image > thresh, square(3))

# remove artifacts connected to image border
cleared = clear_border(bw)

# label image regions
label_image = label(cleared)
# to make the background transparent, pass the value of `bg_label`,
# and leave `bg_color` as `None` and `kind` as `overlay`
image_label_overlay = label2rgb(label_image, image=image, bg_label=0)

fig, ax = plt.subplots(figsize=(10, 6))
ax.imshow(image_label_overlay)

for region in regionprops(label_image):
    # take regions with large enough areas
    if region.area >= 100:
        # draw rectangle around segmented coins
        minr, minc, maxr, maxc = region.bbox
        rect = mpatches.Rectangle((minc, minr), maxc - minc, maxr - minr,
                                  fill=False, edgecolor='red', linewidth=2)
        ax.add_patch(rect)

ax.set_axis_off()
plt.tight_layout()
plt.show()
```

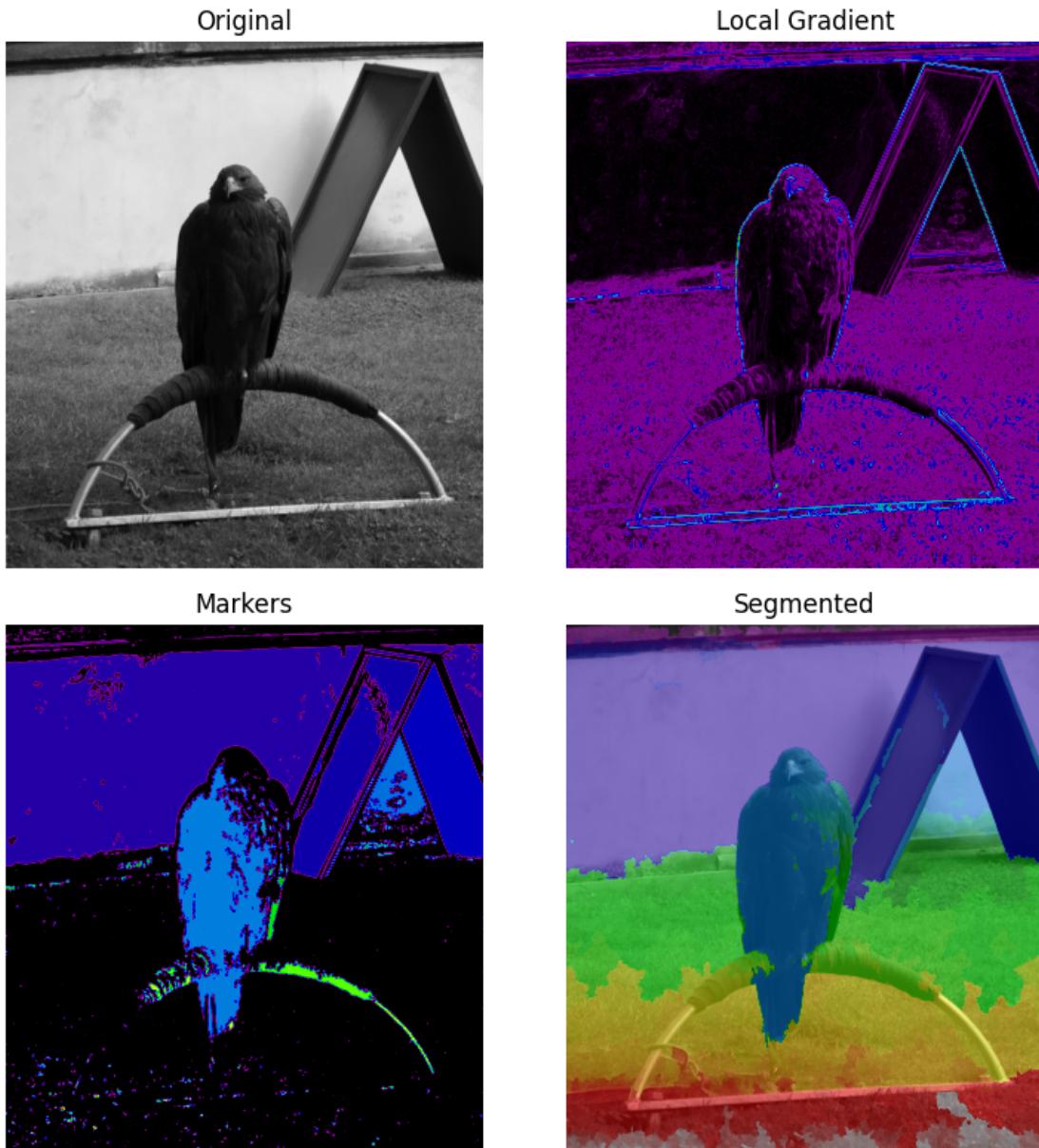
Total running time of the script: (0 minutes 0.247 seconds)

Markers for watershed transform

The watershed is a classical algorithm used for **segmentation**, that is, for separating different objects in an image.

Here a marker image is built from the region of low gradient inside the image. In a gradient image, the areas of high values provide barriers that help to segment the image. Using markers on the lower values will ensure that the segmented objects are found.

See [Wikipedia](#) for more details on the algorithm.



```
from scipy import ndimage as ndi
import matplotlib.pyplot as plt

from skimage.morphology import disk
from skimage.segmentation import watershed
```

(continues on next page)

(continued from previous page)

```
from skimage import data
from skimage.filters import rank
from skimage.util import img_as_ubyte

image = img_as_ubyte(data.eagle())

# denoise image
denoised = rank.median(image, disk(2))

# find continuous region (low gradient -
# where less than 10 for this image) --> markers
# disk(5) is used here to get a more smooth image
markers = rank.gradient(denoised, disk(5)) < 10
markers = ndi.label(markers)[0]

# local gradient (disk(2) is used to keep edges thin)
gradient = rank.gradient(denoised, disk(2))

# process the watershed
labels = watershed(gradient, markers)

# display results
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(8, 8),
                        sharex=True, sharey=True)
ax = axes.ravel()

ax[0].imshow(image, cmap=plt.cm.gray)
ax[0].set_title("Original")

ax[1].imshow(gradient, cmap=plt.cm.nipy_spectral)
ax[1].set_title("Local Gradient")

ax[2].imshow(markers, cmap=plt.cm.nipy_spectral)
ax[2].set_title("Markers")

ax[3].imshow(image, cmap=plt.cm.gray)
ax[3].imshow(labels, cmap=plt.cm.nipy_spectral, alpha=.5)
ax[3].set_title("Segmented")

for a in ax:
    a.axis('off')

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

Total running time of the script: (0 minutes 5.670 seconds)

Comparison of segmentation and superpixel algorithms

This example compares four popular low-level image segmentation methods. As it is difficult to obtain good segmentations, and the definition of “good” often depends on the application, these methods are usually used for obtaining an oversegmentation, also known as superpixels. These superpixels then serve as a basis for more sophisticated algorithms such as conditional random fields (CRF).

Felzenszwalb's efficient graph based segmentation

This fast 2D image segmentation algorithm, proposed in¹ is popular in the computer vision community. The algorithm has a single `scale` parameter that influences the segment size. The actual size and number of segments can vary greatly, depending on local contrast.

Quickshift image segmentation

Quickshift is a relatively recent 2D image segmentation algorithm, based on an approximation of kernelized mean-shift. Therefore it belongs to the family of local mode-seeking algorithms and is applied to the 5D space consisting of color information and image location².

One of the benefits of quickshift is that it actually computes a hierarchical segmentation on multiple scales simultaneously.

Quickshift has two main parameters: `sigma` controls the scale of the local density approximation, `max_dist` selects a level in the hierarchical segmentation that is produced. There is also a trade-off between distance in color-space and distance in image-space, given by `ratio`.

SLIC - K-Means based image segmentation

This algorithm simply performs K-means in the 5d space of color information and image location and is therefore closely related to quickshift. As the clustering method is simpler, it is very efficient. It is essential for this algorithm to work in Lab color space to obtain good results. The algorithm quickly gained momentum and is now widely used. See³ for details. The `compactness` parameter trades off color-similarity and proximity, as in the case of Quickshift, while `n_segments` chooses the number of centers for kmeans.

Compact watershed segmentation of gradient images

Instead of taking a color image as input, watershed requires a grayscale *gradient* image, where bright pixels denote a boundary between regions. The algorithm views the image as a landscape, with bright pixels forming high peaks. This landscape is then flooded from the given *markers*, until separate flood basins meet at the peaks. Each distinct basin then forms a different image segment.⁴

As with SLIC, there is an additional `compactness` argument that makes it harder for markers to flood faraway pixels. This makes the watershed regions more regularly shaped.⁵

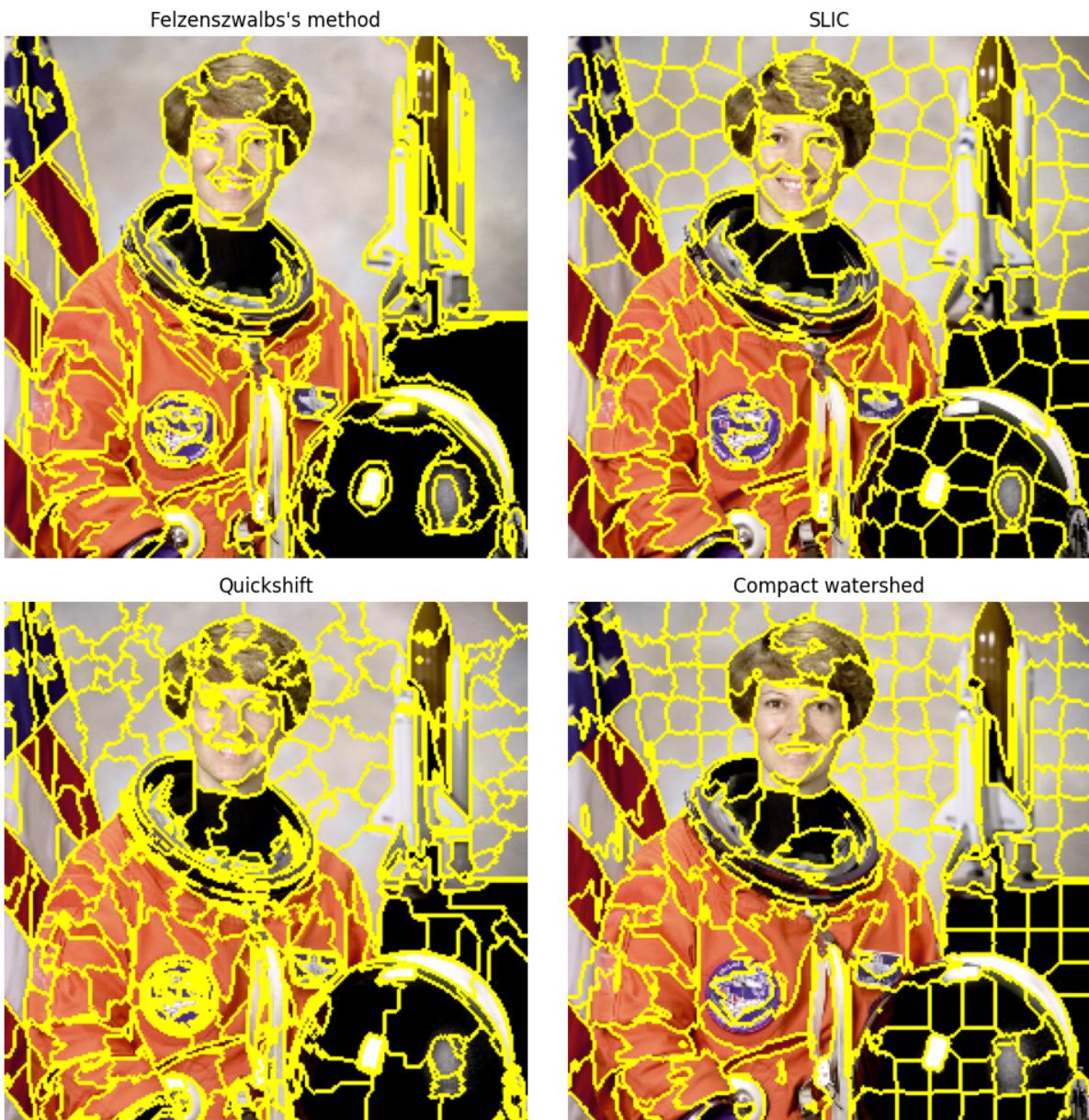
¹ Efficient graph-based image segmentation, Felzenszwalb, P.F. and Huttenlocher, D.P. International Journal of Computer Vision, 2004

² Quick shift and kernel methods for mode seeking, Vedaldi, A. and Soatto, S. European Conference on Computer Vision, 2008

³ Radhakrishna Achanta, Appu Shaji, Kevin Smith, Aurelien Lucchi, Pascal Fua, and Sabine Suesstrunk, SLIC Superpixels Compared to State-of-the-art Superpixel Methods, TPAMI, May 2012.

⁴ https://en.wikipedia.org/wiki/Watershed_%28image_processing%29

⁵ Peer Neubert & Peter Protzel (2014). Compact Watershed and Preemptive SLIC: On Improving Trade-offs of Superpixel Segmentation Algorithms. ICPR 2014, pp 996-1001. DOI:10.1109/ICPR.2014.181 https://www.tu-chemnitz.de/etit/proaut/publications/cws_pSLIC_ICPR.pdf



```
Felzenszwalb number of segments: 194
SLIC number of segments: 196
Quickshift number of segments: 695
Watershed number of segments: 256
```

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

(continues on next page)

(continued from previous page)

```
from skimage.data import astronaut
from skimage.color import rgb2gray
from skimage.filters import sobel
from skimage.segmentation import felzenszwalb, slic, quickshift, watershed
from skimage.segmentation import mark_boundaries
from skimage.util import img_as_float

img = img_as_float(astronaut()[:, ::2])

segments_fz = felzenszwalb(img, scale=100, sigma=0.5, min_size=50)
segments_slic = slic(img, n_segments=250, compactness=10, sigma=1,
                     start_label=1)
segments_quick = quickshift(img, kernel_size=3, max_dist=6, ratio=0.5)
gradient = sobel(rgb2gray(img))
segments_watershed = watershed(gradient, markers=250, compactness=0.001)

print(f'Felzenszwalb number of segments: {len(np.unique(segments_fz))}')
print(f'SLIC number of segments: {len(np.unique(segments_slic))}')
print(f'Quickshift number of segments: {len(np.unique(segments_quick))}')
print(f'Watershed number of segments: {len(np.unique(segments_watershed))}')

fig, ax = plt.subplots(2, 2, figsize=(10, 10), sharex=True, sharey=True)

ax[0, 0].imshow(mark_boundaries(img, segments_fz))
ax[0, 0].set_title("Felzenszwalb's method")
ax[0, 1].imshow(mark_boundaries(img, segments_slic))
ax[0, 1].set_title('SLIC')
ax[1, 0].imshow(mark_boundaries(img, segments_quick))
ax[1, 0].set_title('Quickshift')
ax[1, 1].imshow(mark_boundaries(img, segments_watershed))
ax[1, 1].set_title('Compact watershed')

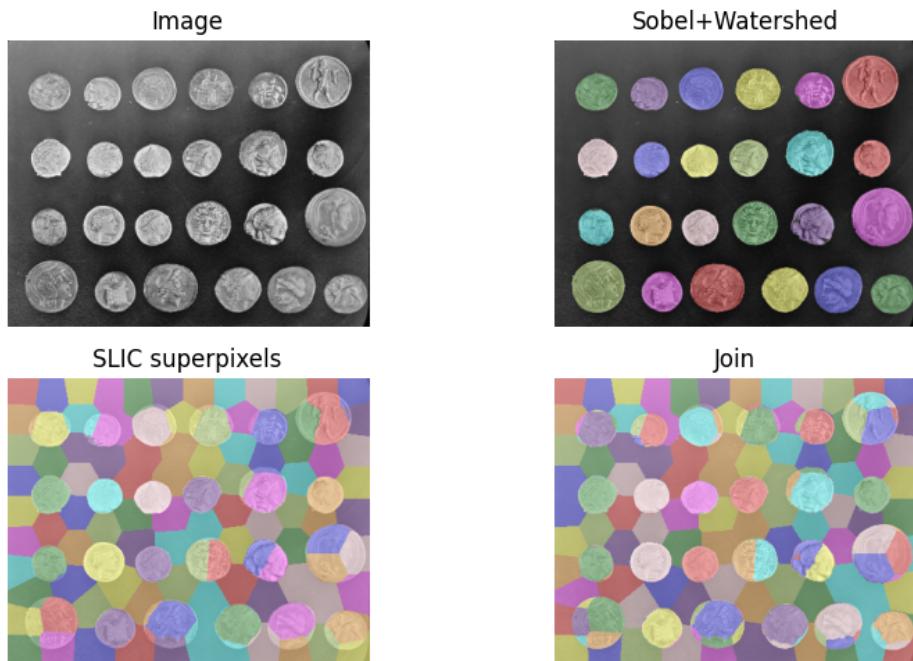
for a in ax.ravel():
    a.set_axis_off()

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

Total running time of the script: (0 minutes 1.471 seconds)

Find the intersection of two segmentations

When segmenting an image, you may want to combine multiple alternative segmentations. The `skimage.segmentation.join_segmentations()` function computes the join of two segmentations, in which a pixel is placed in the same segment if and only if it is in the same segment in *both* segmentations.



```

import numpy as np
import matplotlib.pyplot as plt

from skimage.filters import sobel
from skimage.measure import label
from skimage.segmentation import slic, join_segmentations, watershed
from skimage.color import label2rgb
from skimage import data

coins = data.coins()

# Make segmentation using edge-detection and watershed.
edges = sobel(coins)

# Identify some background and foreground pixels from the intensity values.
# These pixels are used as seeds for watershed.
markers = np.zeros_like(coins)
foreground, background = 1, 2
markers[coins < 30.0] = background
markers[coins > 150.0] = foreground

ws = watershed(edges, markers)
seg1 = label(ws == foreground)

# Make segmentation using SLIC superpixels.
seg2 = slic(coins, n_segments=117, max_num_iter=160, sigma=1, compactness=0.75,
            channel_axis=None, start_label=0)

# Combine the two.
segj = join_segmentations(seg1, seg2)

```

(continues on next page)

(continued from previous page)

```
# Show the segmentations.
fig, axes = plt.subplots(ncols=2, nrows=2, figsize=(9, 5),
                       sharex=True, sharey=True)
ax = axes.ravel()
ax[0].imshow(coins, cmap='gray')
ax[0].set_title('Image')

color1 = label2rgb(seg1, image=coins, bg_label=0)
ax[1].imshow(color1)
ax[1].set_title('Sobel+Watershed')

color2 = label2rgb(seg2, image=coins, image_alpha=0.5, bg_label=-1)
ax[2].imshow(color2)
ax[2].set_title('SLIC superpixels')

color3 = label2rgb(segj, image=coins, image_alpha=0.5, bg_label=-1)
ax[3].imshow(color3)
ax[3].set_title('Join')

for a in ax:
    a.axis('off')
fig.tight_layout()
plt.show()
```

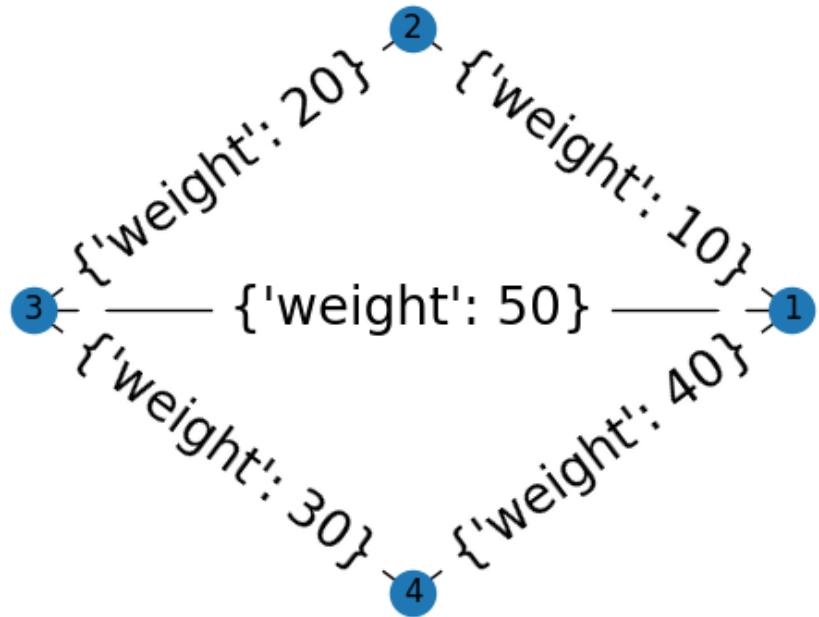
Total running time of the script: (0 minutes 1.559 seconds)

Region Adjacency Graphs

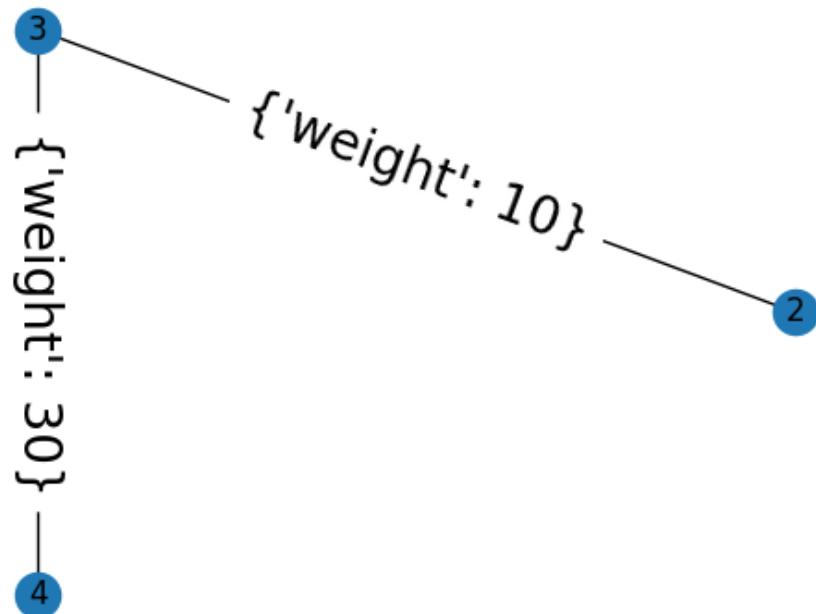
This example demonstrates the use of the `merge_nodes` function of a Region Adjacency Graph (RAG). The `RAG` class represents an undirected weighted graph which inherits from `networkx.Graph` class. When a new node is formed by merging two nodes, the edge weight of all the edges incident on the resulting node can be updated by a user defined function `weight_func`.

The default behaviour is to use the smaller edge weight in case of a conflict. The example below also shows how to use a custom function to select the larger weight instead.

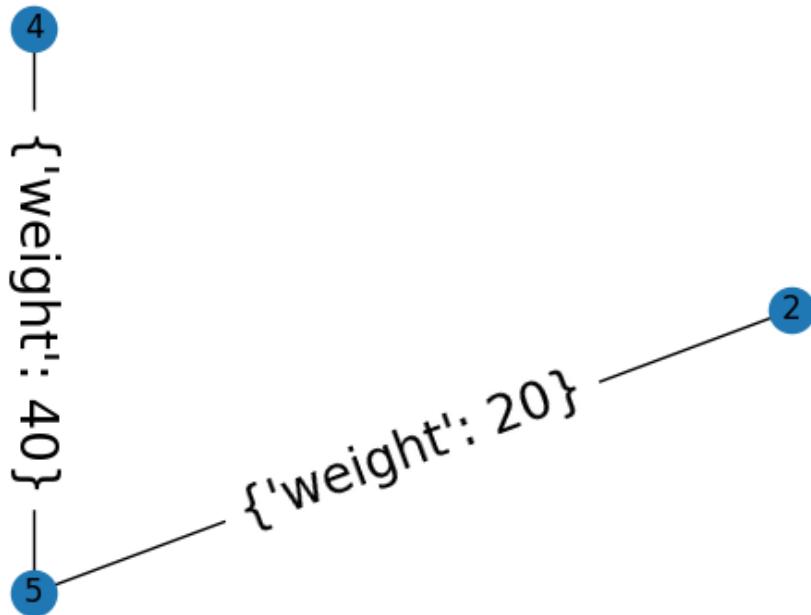
Original Graph



Merged with default (min)



Merged with max without `in_place`



```

import skimage as ski
import networkx as nx
from matplotlib import pyplot as plt
import numpy as np

def max_edge(g, src, dst, n):
    """Callback to handle merging nodes by choosing maximum weight.

    Returns a dictionary with ``"weight"`` set as either the weight between
    `(src, `n`)` or `(dst, `n`)` in `g` or the maximum of the two when
    both exist.

    Parameters
    -----
    g : RAG
        The graph under consideration.
    src, dst : int
        The vertices in `g` to be merged.
    n : int
        A neighbor of `src` or `dst` or both.

    Returns
    -----
  
```

(continues on next page)

(continued from previous page)

```
data : dict
    A dict with the "weight" attribute set the weight between
    (`src`, `n`) or (`dst`, `n`) in `g` or the maximum of the two when
    both exist.
"""

w1 = g[n].get(src, {'weight': -np.inf})['weight']
w2 = g[n].get(dst, {'weight': -np.inf})['weight']
return {'weight': max(w1, w2)}

def display(g, title):
    """Displays a graph with the given title."""
    pos = nx.circular_layout(g)
    plt.figure()
    plt.title(title)
    nx.draw(g, pos)
    nx.draw_networkx_labels(g, pos)
    nx.draw_networkx_edge_labels(g, pos, font_size=20)

g = ski.graph.RAG()
g.add_edge(1, 2, weight=10)
g.add_edge(2, 3, weight=20)
g.add_edge(3, 4, weight=30)
g.add_edge(4, 1, weight=40)
g.add_edge(1, 3, weight=50)

# Assigning dummy labels.
for n in g.nodes():
    g.nodes[n]['labels'] = [n]

gc = g.copy()

display(g, "Original Graph")

g.merge_nodes(1, 3)
display(g, "Merged with default (min)")

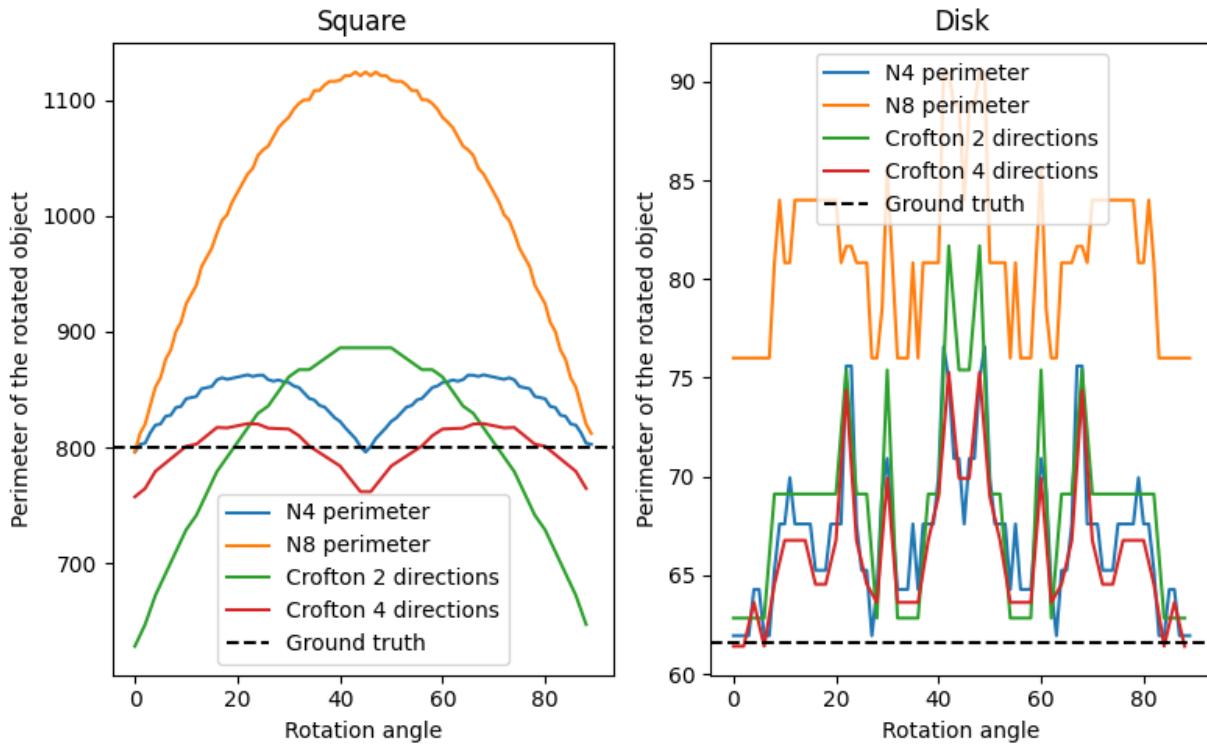
gc.merge_nodes(1, 3, weight_func=max_edge, in_place=False)
display(gc, "Merged with max without in_place")

plt.show()
```

Total running time of the script: (0 minutes 0.236 seconds)

Measure perimeters with different estimators

In this example, we show the error on measuring perimeters, comparing classic approximations and Crofton ones. For that, we estimate the perimeter of an object (either a square or a disk) and its rotated version, as we increase the rotation angle.



```

import matplotlib.pyplot as plt
import numpy as np

from skimage.measure import perimeter, perimeter_crofton
from skimage.transform import rotate

# scale parameter can be used to increase the grid size. The resulting curves
# should be smoothed with higher scales
scale = 10

# Construct two objects, a square and a disk
square = np.zeros((100*scale, 100*scale))
square[40*scale:60*scale, 40*scale:60*scale] = 1

[X, Y] = np.meshgrid(np.linspace(0, 100*scale), np.linspace(0, 100*scale))
R = 20 * scale
disk = (X-50*scale)**2+(Y-50*scale)**2 <= R**2

fig, axes = plt.subplots(1, 2, figsize=(8, 5))
ax = axes.flatten()

dX = X[0, 1] - X[0, 0]

```

(continues on next page)

(continued from previous page)

```
true_perimeters = [80 * scale, 2 * np.pi * R / dX]

# For each type of object, estimate its perimeter as the object is rotated,
# according to different approximations
for index, obj in enumerate([square, disk]):

    # `neighborhood` value can be 4 or 8 for the classic perimeter estimator
    for n in [4, 8]:
        p = []
        angles = range(90)
        for i in angles:
            rotated = rotate(obj, i, order=0)
            p.append(perimeter(rotated, n))
        ax[index].plot(angles, p)

    # `directions` value can be 2 or 4 for the Crofton estimator
    for d in [2, 4]:
        p = []
        angles = np.arange(0, 90, 2)
        for i in angles:
            rotated = rotate(obj, i, order=0)
            p.append(perimeter_crofton(rotated, d))
        ax[index].plot(angles, p)

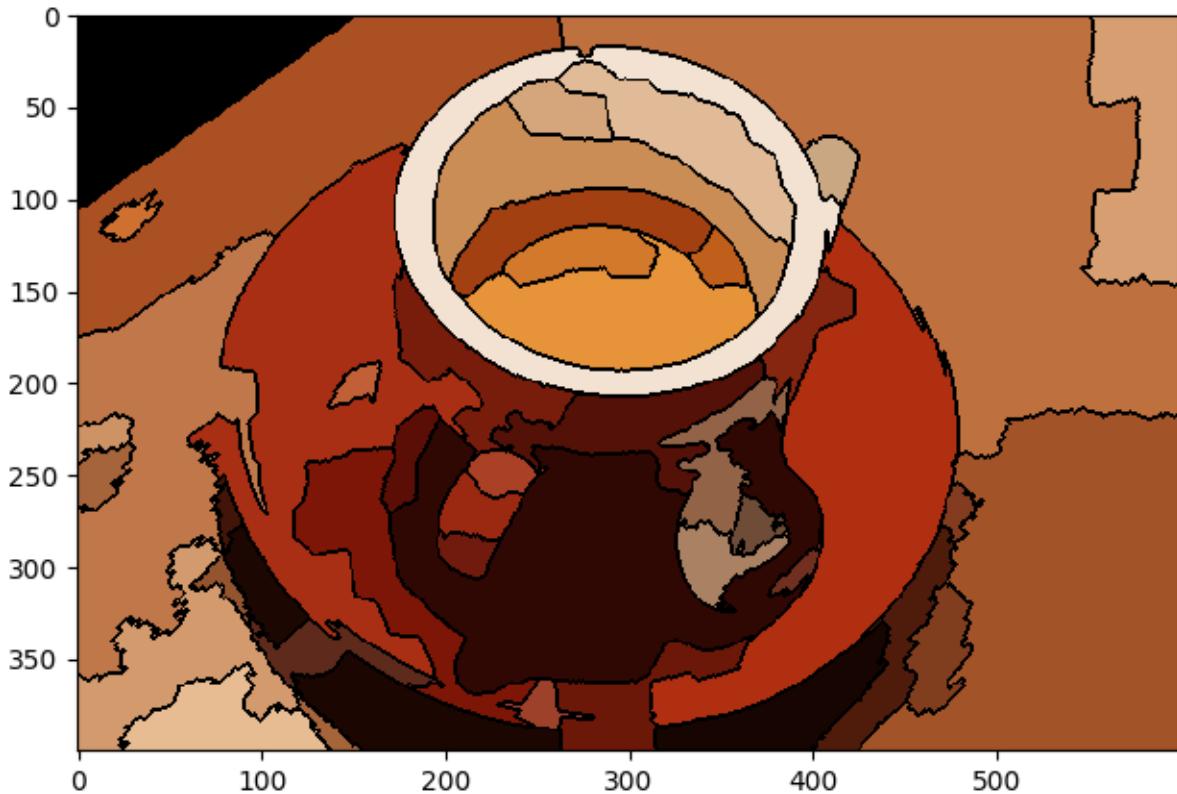
    ax[index].axhline(true_perimeters[index], linestyle='--', color='k')
    ax[index].set_xlabel('Rotation angle')
    ax[index].legend(['N4 perimeter', 'N8 perimeter',
                      'Crofton 2 directions', 'Crofton 4 directions',
                      'Ground truth'],
                    loc='best')
    ax[index].set_ylabel('Perimeter of the rotated object')

ax[0].set_title('Square')
ax[1].set_title('Disk')
plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 30.932 seconds)

RAG Merging

This example constructs a Region Adjacency Graph (RAG) and progressively merges regions that are similar in color. Merging two adjacent regions produces a new region with all the pixels from the merged regions. Regions are merged until no highly similar region pairs remain.



```

from skimage import data, io, segmentation, color
from skimage import graph
import numpy as np

def _weight_mean_color(graph, src, dst, n):
    """Callback to handle merging nodes by recomputing mean color.

    The method expects that the mean color of `dst` is already computed.

    Parameters
    -----
    graph : RAG
        The graph under consideration.
    src, dst : int
        The vertices in `graph` to be merged.
    n : int
        A neighbor of `src` or `dst` or both.

    Returns
    -----
    data : dict
        A dictionary with the ``"weight"`` attribute set as the absolute

```

(continues on next page)

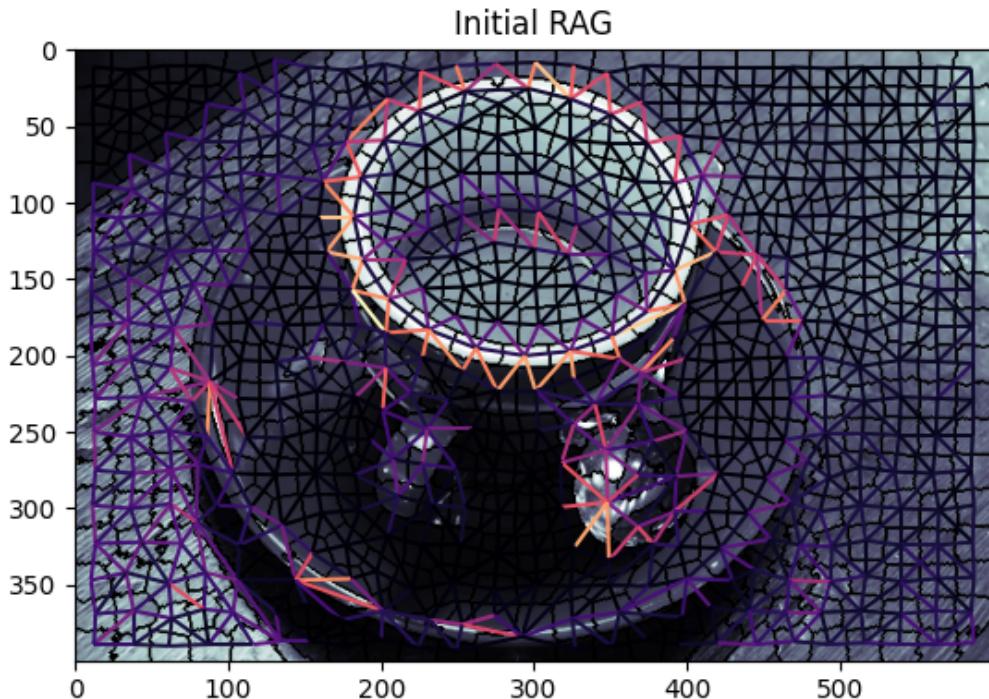
(continued from previous page)

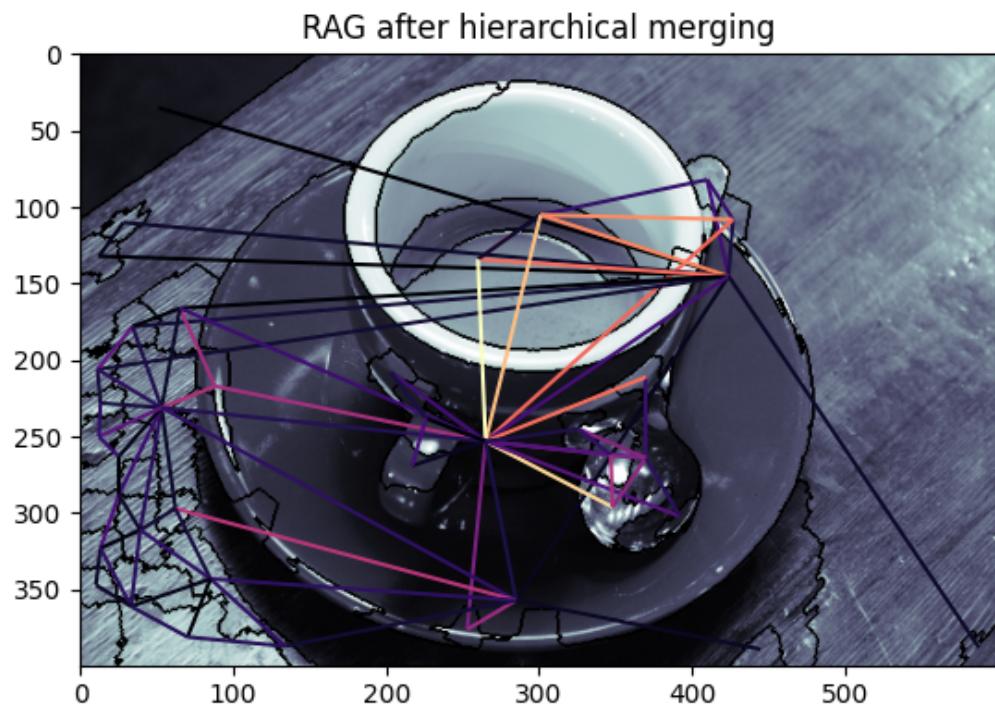
```
difference of the mean color between node `dst` and `n`.  
"""  
  
diff = graph.nodes[dst]['mean color'] - graph.nodes[n]['mean color']  
diff = np.linalg.norm(diff)  
return {'weight': diff}  
  
def merge_mean_color(graph, src, dst):  
    """Callback called before merging two nodes of a mean color distance graph.  
  
    This method computes the mean color of `dst`.  
  
    Parameters  
    -----  
    graph : RAG  
        The graph under consideration.  
    src, dst : int  
        The vertices in `graph` to be merged.  
    """  
  
    graph.nodes[dst]['total color'] += graph.nodes[src]['total color']  
    graph.nodes[dst]['pixel count'] += graph.nodes[src]['pixel count']  
    graph.nodes[dst]['mean color'] = (graph.nodes[dst]['total color'] /  
                                      graph.nodes[dst]['pixel count'])  
  
  
img = data.coffee()  
labels = segmentation.slic(img, compactness=30, n_segments=400, start_label=1)  
g = graph.rag_mean_color(img, labels)  
  
labels2 = graph.merge_hierarchical(labels, g, thresh=35, rag_copy=False,  
                                   in_place_merge=True,  
                                   merge_func=merge_mean_color,  
                                   weight_func=_weight_mean_color)  
  
out = color.label2rgb(labels2, img, kind='avg', bg_label=0)  
out = segmentation.mark_boundaries(out, labels2, (0, 0, 0))  
io.imshow(out)  
io.show()
```

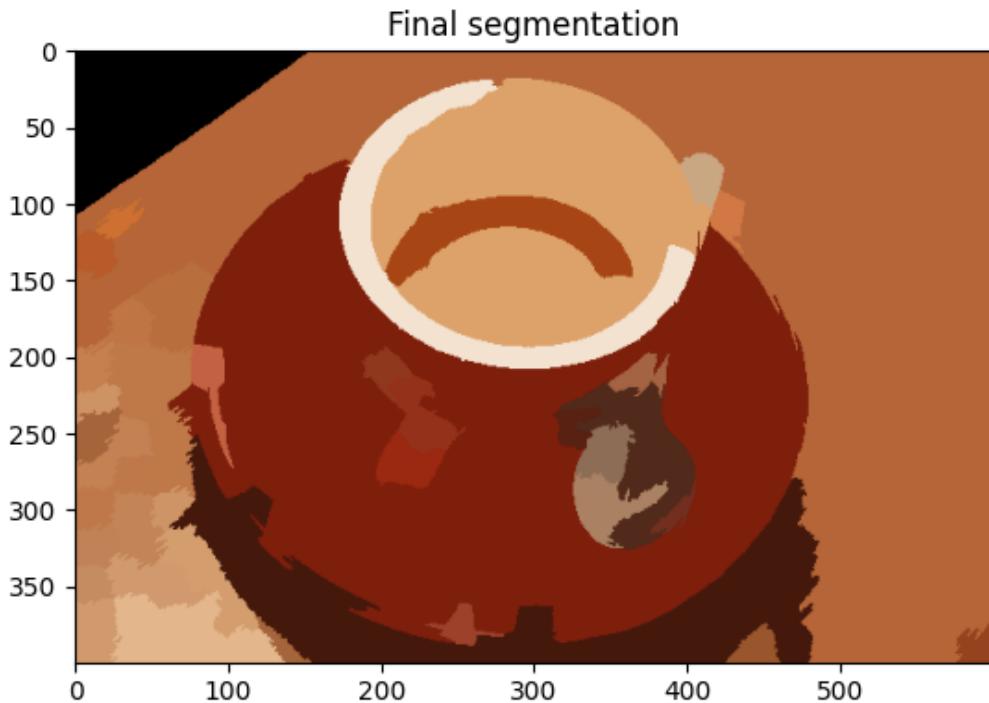
Total running time of the script: (0 minutes 1.828 seconds)

Hierarchical Merging of Region Boundary RAGs

This example demonstrates how to perform hierarchical merging on region boundary Region Adjacency Graphs (RAGs). Region boundary RAGs can be constructed with the `skimage.graph.rag_boundary()` function. The regions with the lowest edge weights are successively merged until there is no edge with weight less than `thresh`. The hierarchical merging is done through the `skimage.graph.merge_hierarchical()` function. For an example of how to construct region boundary based RAGs, see *Region Boundary based RAGs*.







```
from skimage import data, segmentation, filters, color
from skimage import graph
from matplotlib import pyplot as plt

def weight_boundary(graph, src, dst, n):
    """
    Handle merging of nodes of a region boundary region adjacency graph.

    This function computes the ``weight`` and the count ``count``
    attributes of the edge between `n` and the node formed after
    merging `src` and `dst`.
    """

    Parameters
    -----
    graph : RAG
        The graph under consideration.
    src, dst : int
        The vertices in `graph` to be merged.
    n : int
        A neighbor of `src` or `dst` or both.

    Returns
```

(continues on next page)

(continued from previous page)

```
-----  
data : dict  
    A dictionary with the "weight" and "count" attributes to be  
    assigned for the merged node.  
  
"""  
default = {'weight': 0.0, 'count': 0}  
  
count_src = graph[src].get(n, default)['count']  
count_dst = graph[dst].get(n, default)['count']  
  
weight_src = graph[src].get(n, default)['weight']  
weight_dst = graph[dst].get(n, default)['weight']  
  
count = count_src + count_dst  
return {  
    'count': count,  
    'weight': (count_src * weight_src + count_dst * weight_dst)/count  
}  
  
  
def merge_boundary(graph, src, dst):  
    """Call back called before merging 2 nodes.  
  
    In this case we don't need to do any computation here.  
    """  
    pass  
  
img = data.coffee()  
edges = filters.sobel(color.rgb2gray(img))  
labels = segmentation.slic(img, compactness=30, n_segments=400, start_label=1)  
g = graph.rag_boundary(labels, edges)  
  
graph.show_rag(labels, g, img)  
plt.title('Initial RAG')  
  
labels2 = graph.merge_hierarchical(labels, g, thresh=0.08, rag_copy=False,  
                                   in_place_merge=True,  
                                   merge_func=merge_boundary,  
                                   weight_func=weight_boundary)  
  
graph.show_rag(labels, g, img)  
plt.title('RAG after hierarchical merging')  
  
plt.figure()  
out = color.label2rgb(labels2, img, kind='avg', bg_label=0)  
plt.imshow(out)  
plt.title('Final segmentation')  
  
plt.show()
```

Total running time of the script: (0 minutes 0.961 seconds)

Extrema

We detect local maxima in a galaxy image. The image is corrupted by noise, generating many local maxima. To keep only those maxima with sufficient local contrast, we use h-maxima.

```
import matplotlib.pyplot as plt

from skimage.measure import label
from skimage import data
from skimage import color
from skimage.morphology import extrema
from skimage import exposure

color_image = data.hubble_deep_field()

# for illustration purposes, we work on a crop of the image.
x_0 = 70
y_0 = 354
width = 100
height = 100

img = color.rgb2gray(color_image)[y_0:(y_0 + height), x_0:(x_0 + width)]

# the rescaling is done only for visualization purpose.
# the algorithms would work identically in an unscaled version of the
# image. However, the parameter h needs to be adapted to the scale.
img = exposure.rescale_intensity(img)
```

MAXIMA DETECTION

```
# Maxima in the galaxy image are detected by mathematical morphology.
# There is no a priori constraint on the density.

# We find all local maxima
local_maxima = extrema.local_maxima(img)
label_maxima = label(local_maxima)
overlay = color.label2rgb(label_maxima, img, alpha=0.7, bg_label=0,
                           bg_color=None, colors=[(1, 0, 0)])

# We observed in the previous image, that there are many local maxima
# that are caused by the noise in the image.
# For this, we find all local maxima with a height of h.
# This height is the gray level value by which we need to descent
# in order to reach a higher maximum and it can be seen as a local
# contrast measurement.
# The value of h scales with the dynamic range of the image, i.e.
# if we multiply the image with a constant, we need to multiply
# the value of h with the same constant in order to achieve the same result.
h = 0.05
h_maxima = extrema.h_maxima(img, h)
label_h_maxima = label(h_maxima)
overlay_h = color.label2rgb(label_h_maxima, img, alpha=0.7, bg_label=0,
                           bg_color=None, colors=[(1, 0, 0)])
```

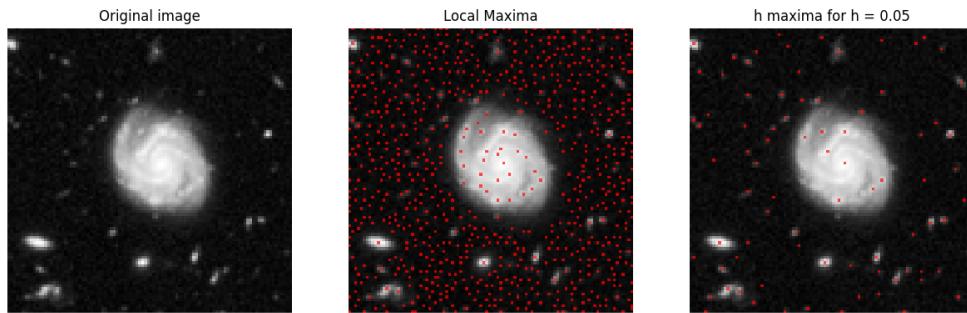
GRAPHICAL OUTPUT

```
# a new figure with 3 subplots
fig, ax = plt.subplots(1, 3, figsize=(15, 5))

ax[0].imshow(img, cmap='gray')
ax[0].set_title('Original image')
ax[0].axis('off')

ax[1].imshow(overlay)
ax[1].set_title('Local Maxima')
ax[1].axis('off')

ax[2].imshow(overlay_h)
ax[2].set_title(f'h maxima for h = {h:.2f}')
ax[2].axis('off')
plt.show()
```



Total running time of the script: (0 minutes 0.160 seconds)

Explore and visualize region properties with pandas

This toy example shows how to compute the size of every labelled region in a series of 10 images. We use 2D images and then 3D images. The blob-like regions are generated synthetically. As the volume fraction (i.e., ratio of pixels or voxels covered by the blobs) increases, the number of blobs (regions) decreases, and the size (area or volume) of a single region can get larger and larger. The area (size) values are available in a pandas-compatible format, which makes for convenient data analysis and visualization.

Besides area, many other region properties are available.

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns

from skimage import data, measure

fractions = np.linspace(0.05, 0.5, 10)
```

2D images

```
images = [data.binary_blobs(volume_fraction=f) for f in fractions]

labeled_images = [measure.label(image) for image in images]

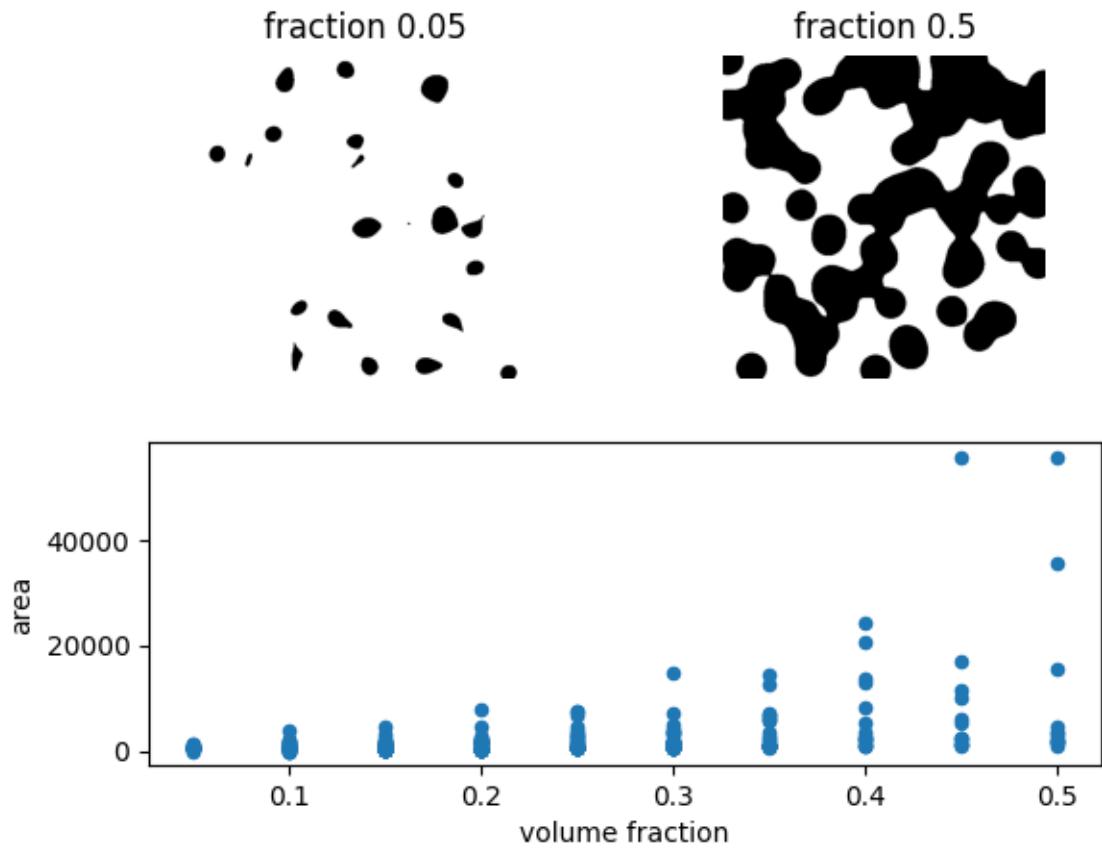
properties = ['label', 'area']

tables = [measure.regionprops_table(image, properties=properties)
          for image in labeled_images]
tables = [pd.DataFrame(table) for table in tables]

for fraction, table in zip(fractions, tables):
    table['volume fraction'] = fraction

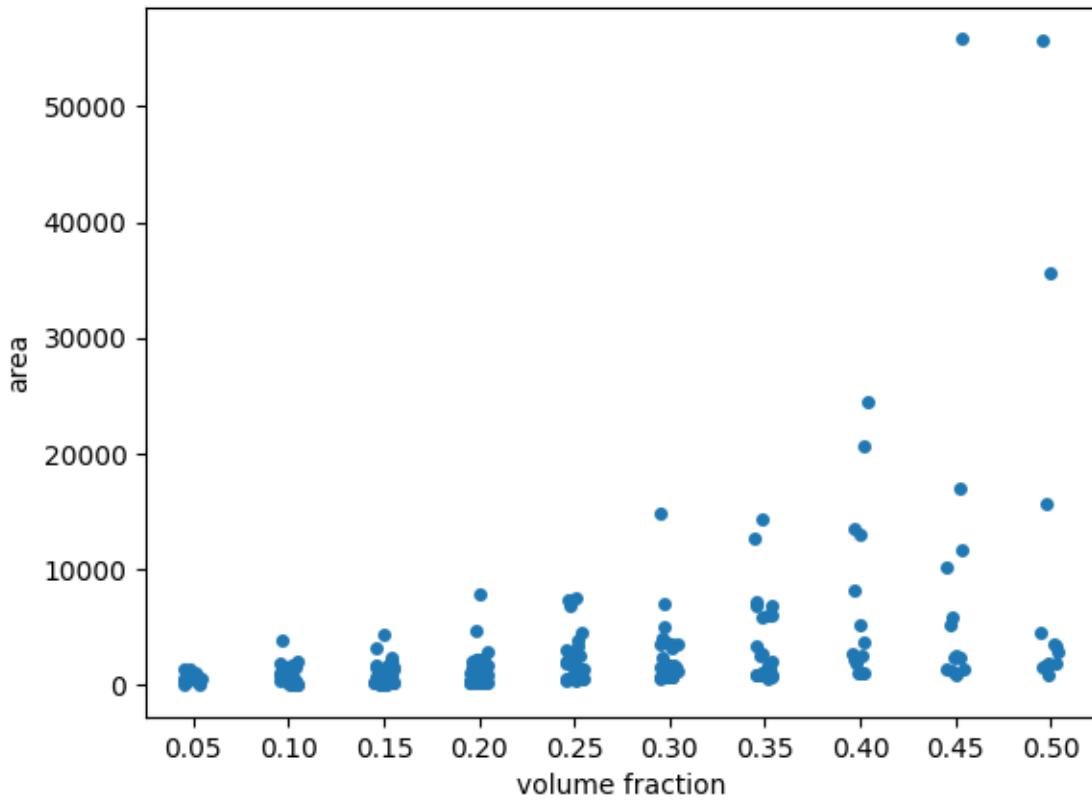
areas = pd.concat(tables, axis=0)

# Create custom grid of subplots
grid = plt.GridSpec(2, 2)
ax1 = plt.subplot(grid[0, 0])
ax2 = plt.subplot(grid[0, 1])
ax = plt.subplot(grid[1, :])
# Show image with lowest volume fraction
ax1.imshow(images[0], cmap='gray_r')
ax1.set_axis_off()
ax1.set_title(f'fraction {fractions[0]}')
# Show image with highest volume fraction
ax2.imshow(images[-1], cmap='gray_r')
ax2.set_axis_off()
ax2.set_title(f'fraction {fractions[-1]}')
# Plot area vs volume fraction
areas.plot(x='volume fraction', y='area', kind='scatter', ax=ax)
plt.show()
```



In the scatterplot, many points seem to be overlapping at low area values. To get a better sense of the distribution, we may want to add some ‘jitter’ to the visualization. To this end, we use `stripplot` (from `seaborn`, the Python library dedicated to statistical data visualization) with argument `jitter=True`.

```
fig, ax = plt.subplots()
sns.stripplot(x='volume fraction', y='area', data=areas, jitter=True,
               ax=ax)
# Fix floating point rendering
ax.set_xticklabels([f'{frac:.2f}' for frac in fractions])
plt.show()
```



```
/github/workspace/build/scikit-image/doc/examples/segmentation/plot_regionprops_table.py:74: UserWarning:
```

```
FixedFormatter should only be used together with FixedLocator
```

3D images

Doing the same analysis in 3D, we find a much more dramatic behaviour: blobs coalesce into a single, giant piece as the volume fraction crosses ~ 0.25 . This corresponds to the [percolation threshold](#) in statistical physics and graph theory.

```
images = [data.binary_blobs(length=128, n_dim=3, volume_fraction=f)
          for f in fractions]

labeled_images = [measure.label(image) for image in images]

properties = ['label', 'area']

tables = [measure.regionprops_table(image, properties=properties)
          for image in labeled_images]
tables = [pd.DataFrame(table) for table in tables]

for fraction, table in zip(fractions, tables):
```

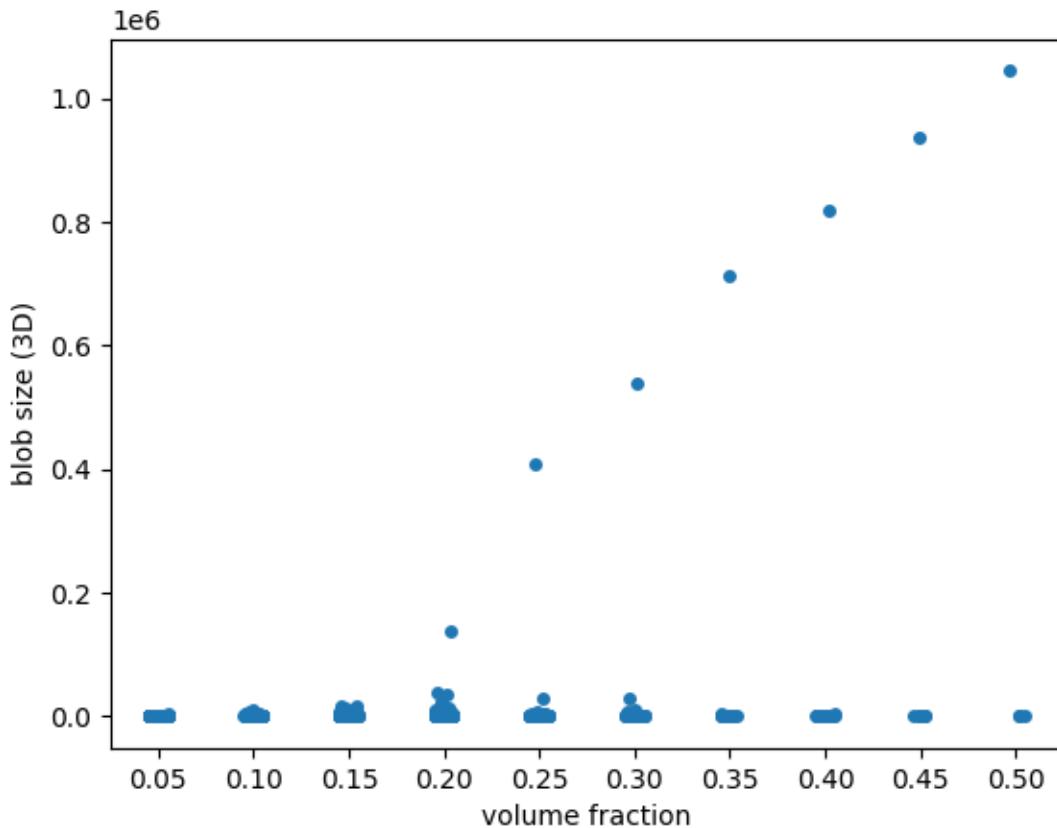
(continues on next page)

(continued from previous page)

```
table['volume fraction'] = fraction

blob_volumes = pd.concat(tables, axis=0)

fig, ax = plt.subplots()
sns.stripplot(x='volume fraction', y='area', data=blob_volumes, jitter=True,
               ax=ax)
ax.set_ylabel('blob size (3D)')
# Fix floating point rendering
ax.set_xticklabels([f'{frac:.2f}' for frac in fractions])
plt.show()
```



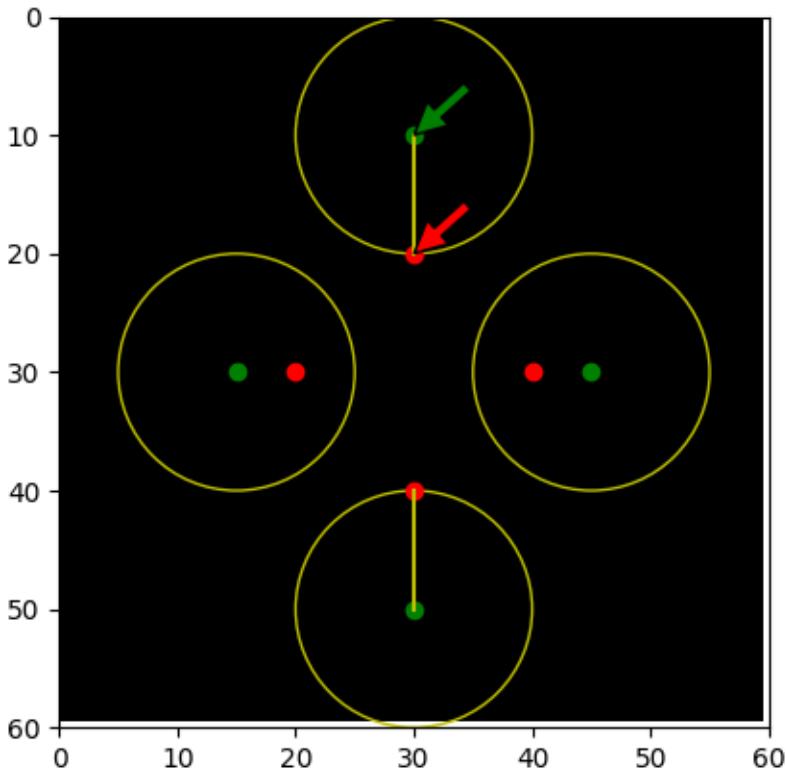
```
/github/workspace/build/scikit-image/doc/examples/segmentation/plot_regionprops_table.  
→py:107: UserWarning:
```

```
FixedFormatter should only be used together with FixedLocator
```

Total running time of the script: (0 minutes 2.762 seconds)

Hausdorff Distance

This example shows how to calculate the Hausdorff distance between two sets of points. The [Hausdorff distance](#) is the maximum distance between any point on the first set and its nearest point on the second set, and vice-versa.



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

from skimage import metrics

shape = (60, 60)
image = np.zeros(shape)

# Create a diamond-like shape where the four corners form the 1st set of points
x_diamond = 30
y_diamond = 30
r = 10

fig, ax = plt.subplots()
plt_x = [0, 1, 0, -1]
plt_y = [1, 0, -1, 0]

set_ax = [(x_diamond + r * x) for x in plt_x]
set_ay = [(y_diamond + r * y) for y in plt_y]
```

(continues on next page)

(continued from previous page)

```
plt.plot(set_ax, set_ay, 'or')

# Create a kite-like shape where the four corners form the 2nd set of points
x_kite = 30
y_kite = 30
x_r = 15
y_r = 20

set_bx = [(x_kite + x_r * x) for x in plt_x]
set_by = [(y_kite + y_r * y) for y in plt_y]
plt.plot(set_bx, set_by, 'og')

# Set up the data to compute the Hausdorff distance
coords_a = np.zeros(shape, dtype=bool)
coords_b = np.zeros(shape, dtype=bool)
for x, y in zip(set_ax, set_ay):
    coords_a[(x, y)] = True

for x, y in zip(set_bx, set_by):
    coords_b[(x, y)] = True

# Call the Hausdorff function on the coordinates
metrics.hausdorff_distance(coords_a, coords_b)
hausdorff_point_a, hausdorff_point_b = metrics.hausdorff_pair(coords_a,
                                                               coords_b)

# Plot the lines that shows the length of the Hausdorff distance
x_line = [30, 30]
y_line = [20, 10]
plt.plot(x_line, y_line, 'y')

x_line = [30, 30]
y_line = [40, 50]
plt.plot(x_line, y_line, 'y')

# Plot circles to show that at this distance, the Hausdorff distance can
# travel to its nearest neighbor (in this case, from the kite to diamond)
ax.add_artist(plt.Circle((30, 10), 10, color='y', fill=None))
ax.add_artist(plt.Circle((30, 50), 10, color='y', fill=None))
ax.add_artist(plt.Circle((15, 30), 10, color='y', fill=None))
ax.add_artist(plt.Circle((45, 30), 10, color='y', fill=None))

# Annotate the returned pair of points that are Hausdorff distance apart
ax.annotate('a', xy=hausdorff_point_a, xytext=(35, 15),
            arrowprops=dict(facecolor='red', shrink=0.005))
ax.annotate('b', xy=hausdorff_point_b, xytext=(35, 5),
            arrowprops=dict(facecolor='green', shrink=0.005))

ax.imshow(image, cmap=plt.cm.gray)
ax.axis([0, 60, 60, 0])
plt.show()
```

Total running time of the script: (0 minutes 0.086 seconds)

Measure region properties

This example shows how to measure properties of labelled image regions. We first analyze an image with two ellipses. Below we show how to explore interactively the properties of labelled objects.

```
import math
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

from skimage.draw import ellipse
from skimage.measure import label, regionprops, regionprops_table
from skimage.transform import rotate

image = np.zeros((600, 600))

rr, cc = ellipse(300, 350, 100, 220)
image[rr, cc] = 1

image = rotate(image, angle=15, order=0)

rr, cc = ellipse(100, 100, 60, 50)
image[rr, cc] = 1

label_img = label(image)
regions = regionprops(label_img)
```

We use the `skimage.measure.regionprops()` result to draw certain properties on each region. For example, in red, we plot the major and minor axes of each ellipse.

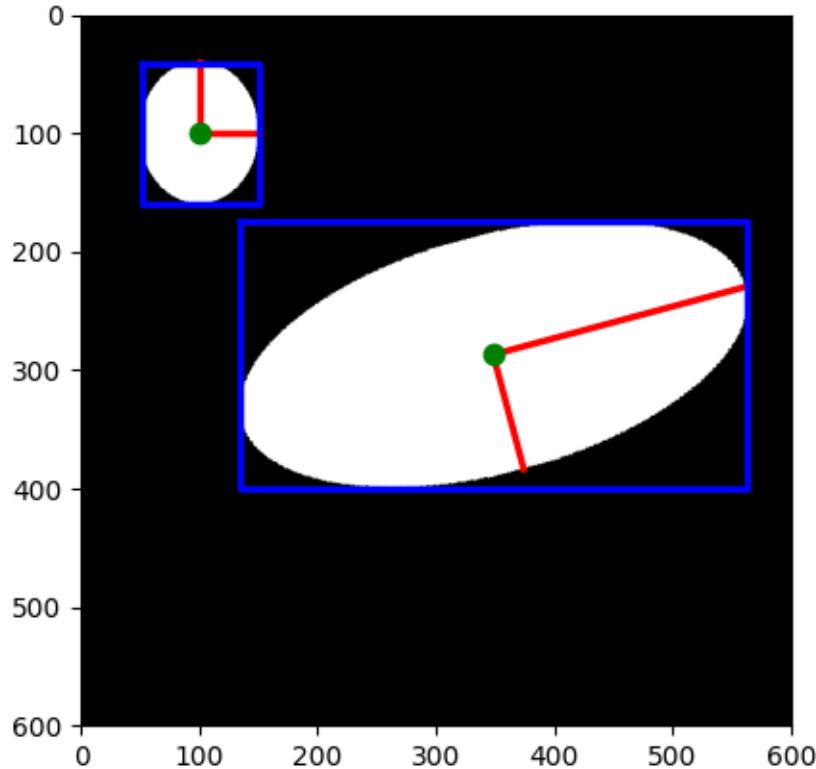
```
fig, ax = plt.subplots()
ax.imshow(image, cmap=plt.cm.gray)

for props in regions:
    y0, x0 = props.centroid
    orientation = props.orientation
    x1 = x0 + math.cos(orientation) * 0.5 * props.axis_minor_length
    y1 = y0 - math.sin(orientation) * 0.5 * props.axis_minor_length
    x2 = x0 - math.sin(orientation) * 0.5 * props.axis_major_length
    y2 = y0 - math.cos(orientation) * 0.5 * props.axis_major_length

    ax.plot((x0, x1), (y0, y1), '-r', linewidth=2.5)
    ax.plot((x0, x2), (y0, y2), '-r', linewidth=2.5)
    ax.plot(x0, y0, '.g', markersize=15)

    minr, minc, maxr, maxc = props.bbox
    bx = (minc, maxc, maxc, minc, minc)
    by = (minr, minr, maxr, maxr, minr)
    ax.plot(bx, by, '-b', linewidth=2.5)

ax.axis((0, 600, 600, 0))
plt.show()
```



We use the `skimage.measure.regionprops_table()` function to compute (selected) properties for each region. Note that `skimage.measure.regionprops_table` actually computes the properties, whereas `skimage.measure.regionprops` computes them when they come in use (lazy evaluation).

```
props = regionprops_table(label_img, properties=['centroid',
                                                'orientation',
                                                'axis_major_length',
                                                'axis_minor_length'])
```

We now display a table of these selected properties (one region per row), the `skimage.measure.regionprops_table` result being a pandas-compatible dict.

```
pd.DataFrame(props)
```

It is also possible to explore interactively the properties of labelled objects by visualizing them in the hover information of the labels. This example uses `plotly` in order to display properties when hovering over the objects.

```
import plotly
import plotly.express as px
import plotly.graph_objects as go
from skimage import data, filters, measure, morphology

img = data.coins()
# Binary image, post-process the binary mask and compute labels
```

(continues on next page)

(continued from previous page)

```

threshold = filters.threshold_otsu(img)
mask = img > threshold
mask = morphology.remove_small_objects(mask, 50)
mask = morphology.remove_small_holes(mask, 50)
labels = measure.label(mask)

fig = px.imshow(img, binary_string=True)
fig.update_traces(hoverinfo='skip') # hover is only for label info

props = measure.regionprops(labels, img)
properties = ['area', 'eccentricity', 'perimeter', 'intensity_mean']

# For each label, add a filled scatter trace for its contour,
# and display the properties of the label in the hover of this trace.
for index in range(1, labels.max()):
    label_i = props[index].label
    contour = measure.find_contours(labels == label_i, 0.5)[0]
    y, x = contour.T
    hoverinfo = ''
    for prop_name in properties:
        hoverinfo += f'{prop_name}: {getattr(props[index], prop_name):.2f}<br>'
    fig.add_trace(go.Scatter(
        x=x, y=y, name=label_i,
        mode='lines', fill='toself', showlegend=False,
        hovertemplate=hoverinfo, hoveron='points+fills'))

plotly.io.show(fig)

```

Total running time of the script: (0 minutes 0.893 seconds)

Trainable segmentation using local features and random forests

A pixel-based segmentation is computed here using local features based on local intensity, edges and textures at different scales. A user-provided mask is used to identify different regions. The pixels of the mask are used to train a random-forest classifier¹ from scikit-learn. Unlabeled pixels are then labeled from the prediction of the classifier.

This segmentation algorithm is called trainable segmentation in other software such as ilastik² or ImageJ³ (where it is also called “weka segmentation”).

```

import numpy as np
import matplotlib.pyplot as plt
from skimage import data, segmentation, feature, future
from sklearn.ensemble import RandomForestClassifier
from functools import partial

full_img = data.skin()

img = full_img[:900, :900]

# Build an array of labels for training the segmentation.

```

(continues on next page)

¹ <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html>

² <https://www.ilastik.org/documentation/pixelclassification/pixelclassification>

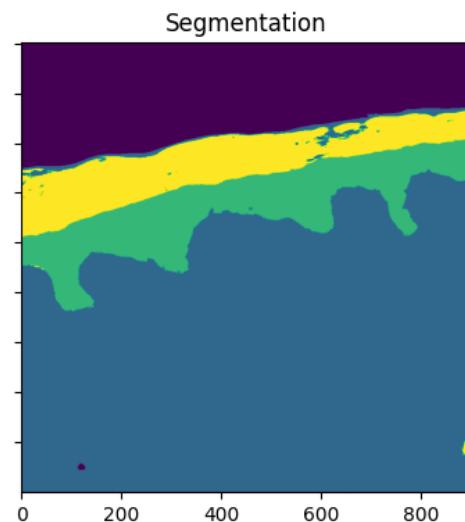
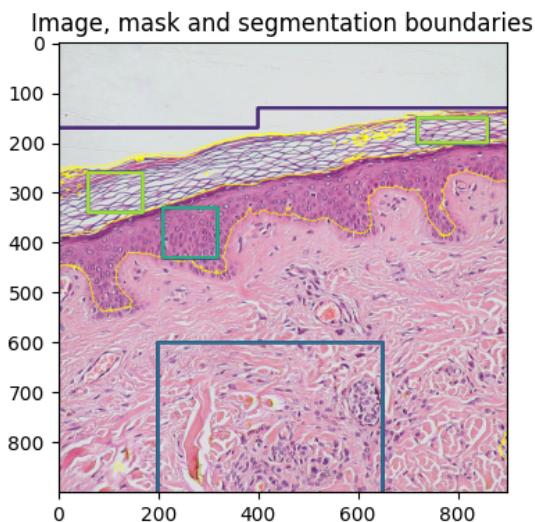
³ https://imagej.net/Trainable_Weka_Segmentation#Training_features_282D.29

(continued from previous page)

```
# Here we use rectangles but visualization libraries such as plotly
# (and napari?) can be used to draw a mask on the image.
training_labels = np.zeros(img.shape[:2], dtype=np.uint8)
training_labels[:130] = 1
training_labels[:170, :400] = 1
training_labels[600:900, 200:650] = 2
training_labels[330:430, 210:320] = 3
training_labels[260:340, 60:170] = 4
training_labels[150:200, 720:860] = 4

sigma_min = 1
sigma_max = 16
features_func = partial(feature.multiscale_basic_features,
                       intensity=True, edges=False, texture=True,
                       sigma_min=sigma_min, sigma_max=sigma_max,
                       channel_axis=-1)
features = features_func(img)
clf = RandomForestClassifier(n_estimators=50, n_jobs=-1,
                            max_depth=10, max_samples=0.05)
clf = future.fit_segmenter(training_labels, features, clf)
result = future.predict_segmenter(features, clf)

fig, ax = plt.subplots(1, 2, sharex=True, sharey=True, figsize=(9, 4))
ax[0].imshow(segmentation.mark_boundaries(img, result, mode='thick'))
ax[0].contour(training_labels)
ax[0].set_title('Image, mask and segmentation boundaries')
ax[1].imshow(result)
ax[1].set_title('Segmentation')
fig.tight_layout()
```

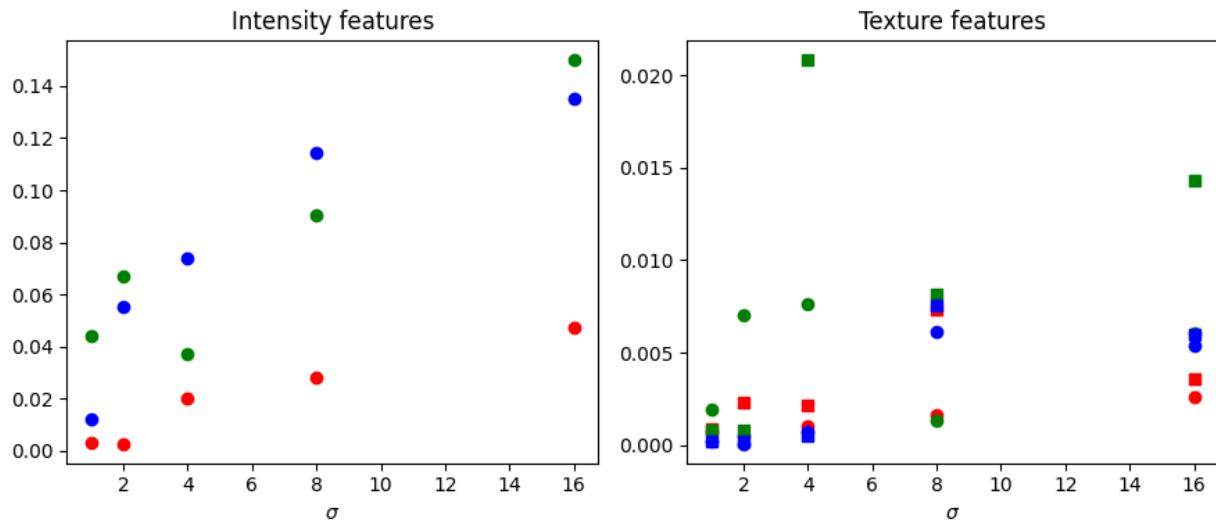


Feature importance

We inspect below the importance of the different features, as computed by scikit-learn. Intensity features have a much higher importance than texture features. It can be tempting to use this information to reduce the number of features given to the classifier, in order to reduce the computing time. However, this can lead to overfitting and a degraded result at the boundary between regions.

```
fig, ax = plt.subplots(1, 2, figsize=(9, 4))
l = len(clf.feature_importances_)
feature_importance = (
    clf.feature_importances_[:l//3],
    clf.feature_importances_[l//3:2*l//3],
    clf.feature_importances_[2*l//3:])
sigmas = np.logspace(
    np.log2(sigma_min), np.log2(sigma_max),
    num=int(np.log2(sigma_max) - np.log2(sigma_min) + 1),
    base=2, endpoint=True)
for ch, color in zip(range(3), ['r', 'g', 'b']):
    ax[0].plot(sigmas, feature_importance[ch][::3], 'o', color=color)
    ax[0].set_title("Intensity features")
    ax[0].set_xlabel("$\sigma$")
for ch, color in zip(range(3), ['r', 'g', 'b']):
    ax[1].plot(sigmas, feature_importance[ch][1::3], 'o', color=color)
    ax[1].plot(sigmas, feature_importance[ch][2::3], 's', color=color)
    ax[1].set_title("Texture features")
    ax[1].set_xlabel("$\sigma$")

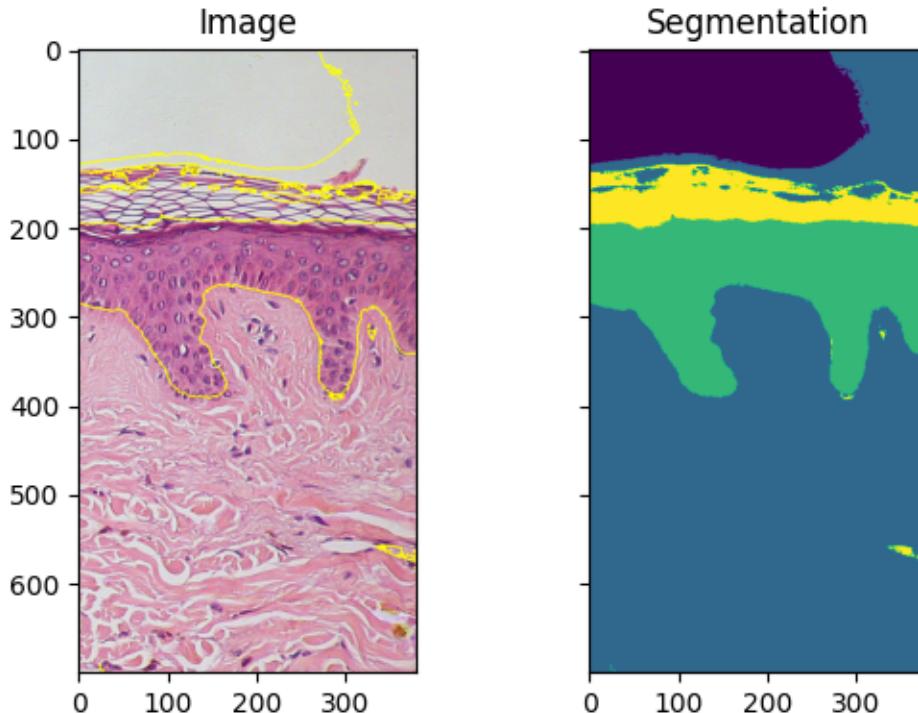
fig.tight_layout()
```



Fitting new images

If you have several images of similar objects acquired in similar conditions, you can use the classifier trained with `fit_segmenter` to segment other images. In the example below we just use a different part of the image.

```
img_new = full_img[:700, 900:]  
  
features_new = features_func(img_new)  
result_new = future.predict_segmenter(features_new, clf)  
fig, ax = plt.subplots(1, 2, sharex=True, sharey=True, figsize=(6, 4))  
ax[0].imshow(segmentation.mark_boundaries(img_new, result_new, mode='thick'))  
ax[0].set_title('Image')  
ax[1].imshow(result_new)  
ax[1].set_title('Segmentation')  
fig.tight_layout()  
  
plt.show()
```



Total running time of the script: (0 minutes 6.808 seconds)

Morphological Snakes

*Morphological Snakes*¹ are a family of methods for image segmentation. Their behavior is similar to that of active contours (for example, *Geodesic Active Contours*² or *Active Contours without Edges*³). However, *Morphological Snakes* use morphological operators (such as dilation or erosion) over a binary array instead of solving PDEs over a floating point array, which is the standard approach for active contours. This makes *Morphological Snakes* faster and numerically more stable than their traditional counterpart.

There are two *Morphological Snakes* methods available in this implementation: *Morphological Geodesic Active Contours* (**MorphGAC**, implemented in the function `morphological_geodesic_active_contour`) and *Morphological Active Contours without Edges* (**MorphACWE**, implemented in the function `morphological_chan_vese`).

MorphGAC is suitable for images with visible contours, even when these contours might be noisy, cluttered, or partially unclear. It requires, however, that the image is preprocessed to highlight the contours. This can be done using the function `inverse_gaussian_gradient`, although the user might want to define their own version. The quality of the **MorphGAC** segmentation depends greatly on this preprocessing step.

On the contrary, **MorphACWE** works well when the pixel values of the inside and the outside regions of the object to segment have different averages. Unlike **MorphGAC**, **MorphACWE** does not require that the contours of the object are well defined, and it works over the original image without any preceding processing. This makes **MorphACWE** easier to use and tune than **MorphGAC**.

¹ A Morphological Approach to Curvature-based Evolution of Curves and Surfaces, Pablo Márquez-Neila, Luis Baumela and Luis Álvarez. In IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI), 2014, DOI:10.1109/TPAMI.2013.106

² Geodesic Active Contours, Vicent Caselles, Ron Kimmel and Guillermo Sapiro. In International Journal of Computer Vision (IJCV), 1997, DOI:10.1023/A:1007979827043

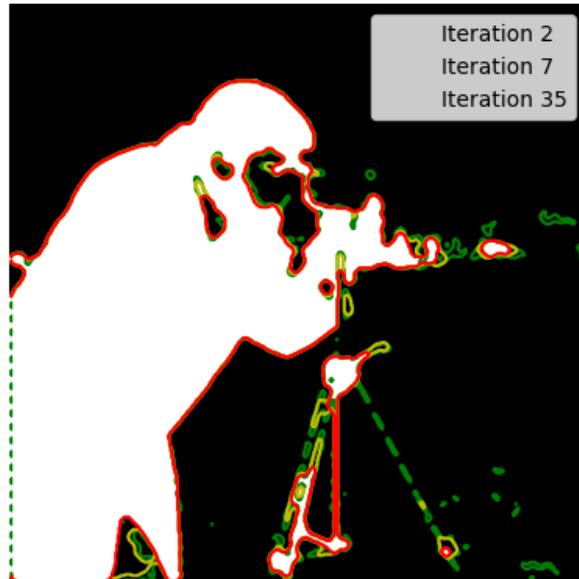
³ Active Contours without Edges, Tony Chan and Luminita Vese. In IEEE Transactions on Image Processing, 2001, DOI:10.1109/83.902291

References

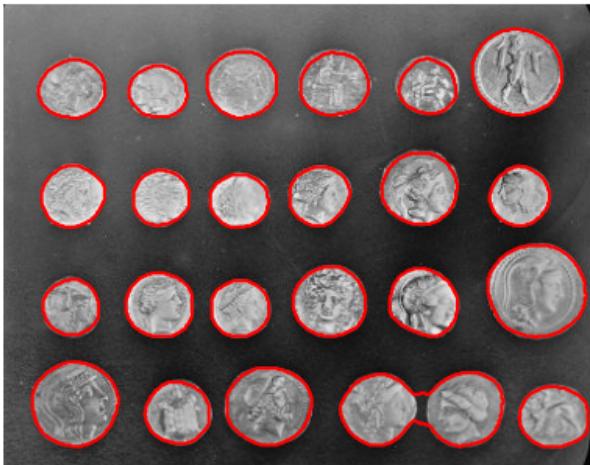
Morphological ACWE segmentation



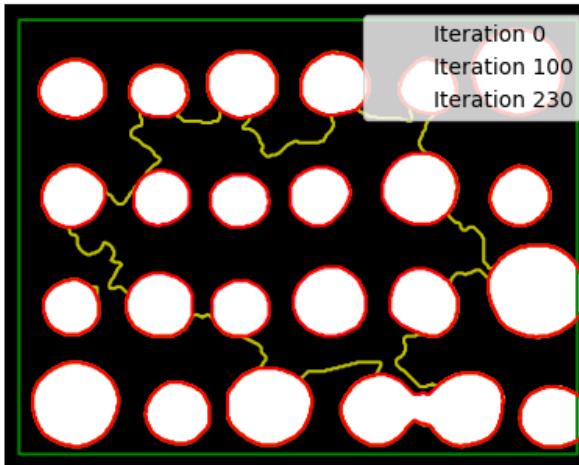
Morphological ACWE evolution



Morphological GAC segmentation



Morphological GAC evolution



```
import numpy as np
import matplotlib.pyplot as plt
from skimage import data, img_as_float
from skimage.segmentation import (morphological_chan_vese,
                                  morphological_geodesic_active_contour,
                                  inverse_gaussian_gradient,
                                  checkerboard_level_set)

def store_evolution_in(lst):
```

(continues on next page)

(continued from previous page)

```

"""Returns a callback function to store the evolution of the level sets in
the given list.
"""

def _store(x):
    lst.append(np.copy(x))

return _store

# Morphological ACWE
image = img_as_float(data.camera())

# Initial level set
init_ls = checkerboard_level_set(image.shape, 6)
# List with intermediate results for plotting the evolution
evolution = []
callback = store_evolution_in(evolution)
ls = morphological_chan_vese(image, num_iter=35, init_level_set=init_ls,
                             smoothing=3, iter_callback=callback)

fig, axes = plt.subplots(2, 2, figsize=(8, 8))
ax = axes.flatten()

ax[0].imshow(image, cmap="gray")
ax[0].set_axis_off()
ax[0].contour(ls, [0.5], colors='r')
ax[0].set_title("Morphological ACWE segmentation", fontsize=12)

ax[1].imshow(ls, cmap="gray")
ax[1].set_axis_off()
contour = ax[1].contour(evolution[2], [0.5], colors='g')
contour.collections[0].set_label("Iteration 2")
contour = ax[1].contour(evolution[7], [0.5], colors='y')
contour.collections[0].set_label("Iteration 7")
contour = ax[1].contour(evolution[-1], [0.5], colors='r')
contour.collections[0].set_label("Iteration 35")
ax[1].legend(loc="upper right")
title = "Morphological ACWE evolution"
ax[1].set_title(title, fontsize=12)

# Morphological GAC
image = img_as_float(data.coins())
gimage = inverse_gaussian_gradient(image)

# Initial level set
init_ls = np.zeros(image.shape, dtype=np.int8)
init_ls[10:-10, 10:-10] = 1
# List with intermediate results for plotting the evolution
evolution = []
callback = store_evolution_in(evolution)

```

(continues on next page)

(continued from previous page)

```

ls = morphological_geodesic_active_contour(gimage, num_iter=230,
                                             init_level_set=init_ls,
                                             smoothing=1, balloon=-1,
                                             threshold=0.69,
                                             iter_callback=callback)

ax[2].imshow(image, cmap="gray")
ax[2].set_axis_off()
ax[2].contour(ls, [0.5], colors='r')
ax[2].set_title("Morphological GAC segmentation", fontsize=12)

ax[3].imshow(ls, cmap="gray")
ax[3].set_axis_off()
contour = ax[3].contour(evolution[0], [0.5], colors='g')
contour.collections[0].set_label("Iteration 0")
contour = ax[3].contour(evolution[100], [0.5], colors='y')
contour.collections[0].set_label("Iteration 100")
contour = ax[3].contour(evolution[-1], [0.5], colors='r')
contour.collections[0].set_label("Iteration 230")
ax[3].legend(loc="upper right")
title = "Morphological GAC evolution"
ax[3].set_title(title, fontsize=12)

fig.tight_layout()
plt.show()

```

Total running time of the script: (0 minutes 5.678 seconds)

Flood Fill

Flood fill is an algorithm to identify and/or change adjacent values in an image based on their similarity to an initial seed point¹. The conceptual analogy is the ‘paint bucket’ tool in many graphic editors.

Basic example

First, a basic example where we will change a checkerboard square from white to mid-gray.

```

import numpy as np
import matplotlib.pyplot as plt
from skimage import data, filters, color, morphology
from skimage.segmentation import flood, flood_fill

checkers = data.checkerboard()

# Fill a square near the middle with value 127, starting at index (76, 76)
filled_checkers = flood_fill(checkers, (76, 76), 127)

fig, ax = plt.subplots(ncols=2, figsize=(10, 5))

```

(continues on next page)

¹ https://en.wikipedia.org/wiki/Flood_fill

(continued from previous page)

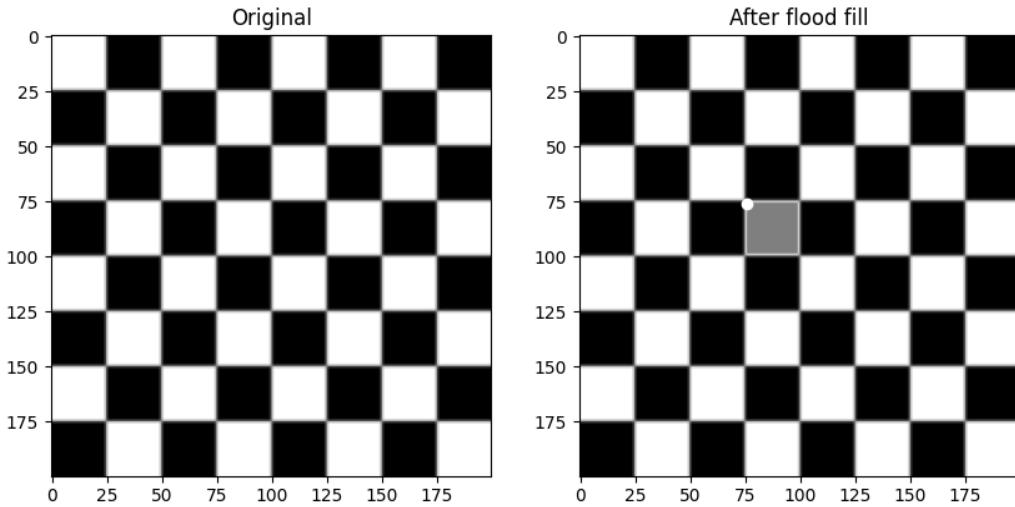
```

ax[0].imshow(checkers, cmap=plt.cm.gray)
ax[0].set_title('Original')

ax[1].imshow(filled_checkers, cmap=plt.cm.gray)
ax[1].plot(76, 76, 'wo') # seed point
ax[1].set_title('After flood fill')

plt.show()

```



Advanced example

Because standard flood filling requires the neighbors to be strictly equal, its use is limited on real-world images with color gradients and noise. The `tolerance` keyword argument widens the permitted range about the initial value, allowing use on real-world images.

Here we will experiment a bit on the cameraman. First, turning his coat from dark to light.

```

cameraman = data.camera()

# Change the cameraman's coat from dark to light (255). The seed point is
# chosen as (155, 150)
light_coat = flood_fill(cameraman, (155, 150), 255, tolerance=10)
fig, ax = plt.subplots(ncols=2, figsize=(10, 5))

ax[0].imshow(cameraman, cmap=plt.cm.gray)
ax[0].set_title('Original')
ax[0].axis('off')

ax[1].imshow(light_coat, cmap=plt.cm.gray)
ax[1].plot(150, 155, 'ro') # seed point
ax[1].set_title('After flood fill')

```

(continues on next page)

(continued from previous page)

```
ax[1].axis('off')

plt.show()
```



The cameraman's coat is in varying shades of gray. Only the parts of the coat matching the shade near the seed value is changed.

Experimentation with tolerance

To get a better intuitive understanding of how the tolerance parameter works, here is a set of images progressively increasing the parameter with seed point in the upper left corner.

```
output = []

for i in range(8):
    tol = 5 + 20 * i
    output.append(flood_fill(cameraman, (0, 0), 255, tolerance=tol))

# Initialize plot and place original image
fig, ax = plt.subplots(nrows=3, ncols=3, figsize=(12, 12))
ax[0, 0].imshow(cameraman, cmap=plt.cm.gray)
ax[0, 0].set_title('Original')
ax[0, 0].axis('off')

# Plot all eight different tolerances for comparison.
for i in range(8):
    m, n = np.unravel_index(i + 1, (3, 3))
    ax[m, n].imshow(output[i], cmap=plt.cm.gray)
    ax[m, n].set_title(f'Tolerance {5 + 20 * i}')
    ax[m, n].axis('off')
    ax[m, n].plot(0, 0, 'bo') # seed point
```

(continues on next page)

(continued from previous page)

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



Flood as mask

A sister function, *flood*, is available which returns a mask identifying the flood rather than modifying the image itself. This is useful for segmentation purposes and more advanced analysis pipelines.

Here we segment the nose of a cat. However, multi-channel images are not supported by *flood[_fill]*. Instead we Sobel filter the red channel to enhance edges, then flood the nose with a tolerance.

```
cat = data.chelsea()
cat_sobel = filters.sobel(cat[..., 0])
cat_nose = flood(cat_sobel, (240, 265), tolerance=0.03)

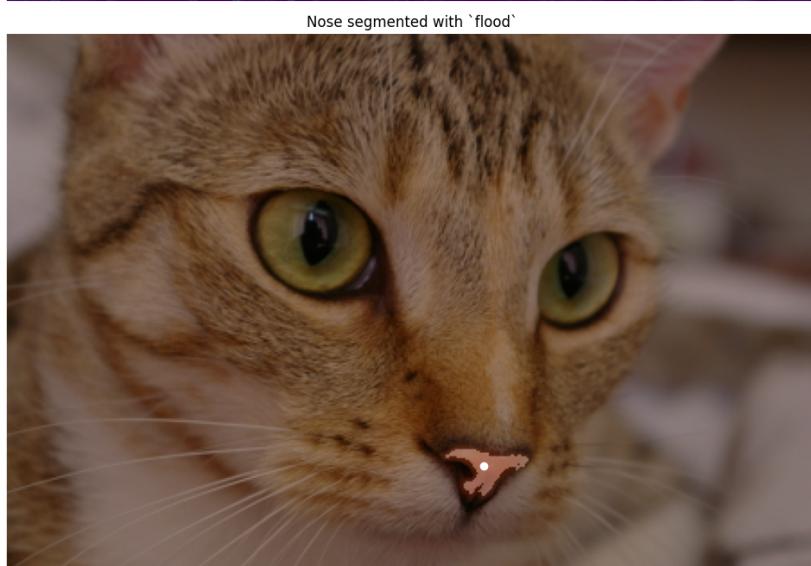
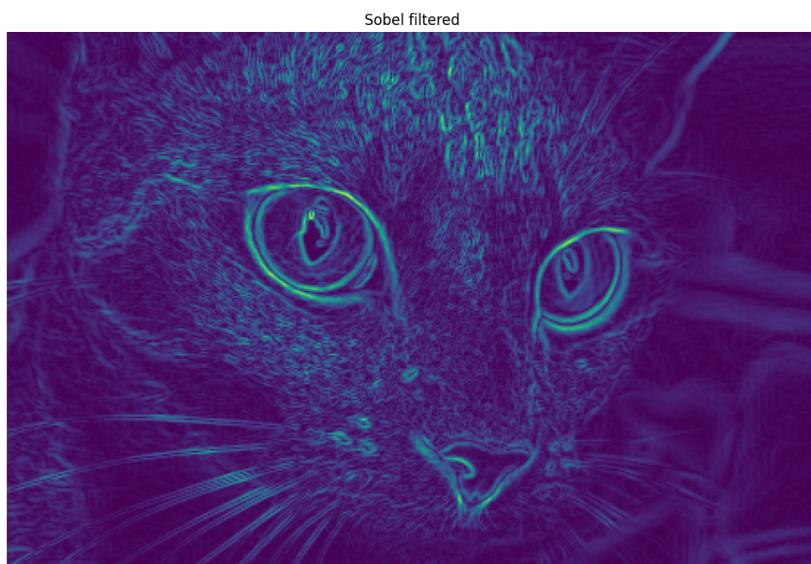
fig, ax = plt.subplots(nrows=3, figsize=(10, 20))

ax[0].imshow(cat)
ax[0].set_title('Original')
ax[0].axis('off')

ax[1].imshow(cat_sobel)
ax[1].set_title('Sobel filtered')
ax[1].axis('off')

ax[2].imshow(cat)
ax[2].imshow(cat_nose, cmap=plt.cm.gray, alpha=0.3)
ax[2].plot(265, 240, 'wo') # seed point
ax[2].set_title('Nose segmented with `flood`')
ax[2].axis('off')

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



Flood-fill in HSV space and mask post-processing

Since flood fill operates on single-channel images, we transform here the image to the HSV (Hue Saturation Value) space in order to flood pixels of similar hue.

In this example we also show that it is possible to post-process the binary mask returned by `skimage.segmentation.flood()` thanks to the functions of `skimage.morphology`.

```
img = data.astronaut()
img_hsv = color.rgb2hsv(img)
img_hsv_copy = np.copy(img_hsv)

# flood function returns a mask of flooded pixels
mask = flood(img_hsv[..., 0], (313, 160), tolerance=0.016)
# Set pixels of mask to new value for hue channel
img_hsv[mask, 0] = 0.5
# Post-processing in order to improve the result
# Remove white pixels from flag, using saturation channel
mask_postprocessed = np.logical_and(mask,
                                    img_hsv_copy[..., 1] > 0.4)
# Remove thin structures with binary opening
mask_postprocessed = morphology.binary_opening(mask_postprocessed,
                                                np.ones((3, 3)))
# Fill small holes with binary closing
mask_postprocessed = morphology.binary_closing(
    mask_postprocessed, morphology.disk(20))
img_hsv_copy[mask_postprocessed, 0] = 0.5

fig, ax = plt.subplots(1, 2, figsize=(8, 4))
ax[0].imshow(color.hsv2rgb(img_hsv))
ax[0].axis('off')
ax[0].set_title('After flood fill')
ax[1].imshow(color.hsv2rgb(img_hsv_copy))
ax[1].axis('off')
ax[1].set_title('After flood fill and post-processing')

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



Total running time of the script: (0 minutes 3.286 seconds)

Euler number

This example shows an illustration of the computation of the Euler number¹ in 2D and 3D objects.

For 2D objects, the Euler number is the number of objects minus the number of holes. Notice that if a neighborhood of 8 connected pixels (2-connectivity) is considered for objects, then this amounts to considering a neighborhood of 4 connected pixels (1-connectivity) for the complementary set (holes, background), and conversely. It is also possible to compute the number of objects using `skimage.measure.label()`, and to deduce the number of holes from the difference between the two numbers.

For 3D objects, the Euler number is obtained as the number of objects plus the number of holes, minus the number of tunnels, or loops. If one uses 3-connectivity for an object (considering the 26 surrounding voxels as its neighborhood), this corresponds to using 1-connectivity for the complementary set (holes, background), that is considering only 6 neighbors for a given voxel. The voxels are represented here with blue transparent surfaces. Inner porosities are represented in red.

```
from skimage.measure import euler_number, label
import matplotlib.pyplot as plt
import numpy as np

# Sample image.
SAMPLE = np.array(
    [[[0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0],
      [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0],
      [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0],
      [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0],
      [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0],
      [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0],
      [0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
      [0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
      [1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0]]]
)
```

(continues on next page)

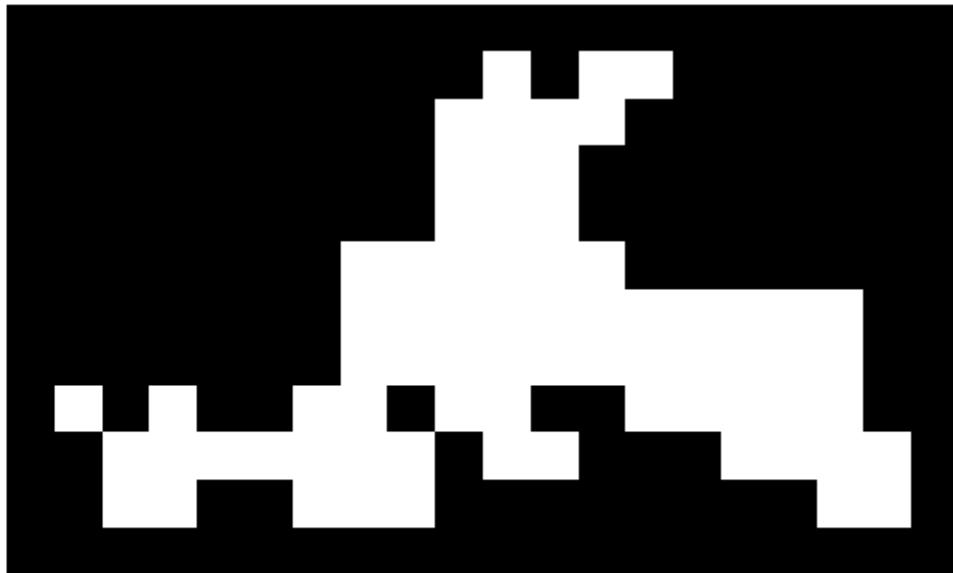
¹ https://en.wikipedia.org/wiki/Euler_characteristic

(continued from previous page)

```
[0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1],
 [0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1]
)
SAMPLE = np.pad(SAMPLE, 1, mode='constant')

fig, ax = plt.subplots()
ax.imshow(SAMPLE, cmap=plt.cm.gray)
ax.axis('off')
e4 = euler_number(SAMPLE, connectivity=1)
object_nb_4 = label(SAMPLE, connectivity=1).max()
holes_nb_4 = object_nb_4 - e4
e8 = euler_number(SAMPLE, connectivity=2)
object_nb_8 = label(SAMPLE, connectivity=2).max()
holes_nb_8 = object_nb_8 - e8
ax.set_title(f'Euler number for N4: {e4} ({object_nb_4} objects, {holes_nb_4} '
             f'holes), \n for N8: {e8} ({object_nb_8} objects, '
             f'{holes_nb_8} holes)')
plt.show()
```

Euler number for N4: 2 (2 objects, 0 holes),
for N8: 0 (1 objects, 1 holes)



3-D objects

In this example, a 3-D cube is generated, then holes and tunnels are added. Euler number is evaluated with 6 and 26 neighborhood configuration. This code is inspired by https://matplotlib.org/devdocs/gallery/mplot3d/voxels_numpy_logo.html

```
def make_ax(grid=False):
    ax = plt.figure().add_subplot(projection='3d')
    ax.grid(grid)
    ax.set_axis_off()
    return ax

def explode(data):
    """visualization to separate voxels

    Data voxels are separated by 0-valued ones so that they appear
    separated in the matplotlib figure.
    """
    size = np.array(data.shape) * 2
    data_e = np.zeros(size - 1, dtype=data.dtype)
    data_e[::2, ::2, ::2] = data
    return data_e

# shrink the gaps between voxels

def expand_coordinates(indices):
    """
    This collapses together pairs of indices, so that
    the gaps in the volume array will have a zero width.
    """
    x, y, z = indices
    x[1::2, :, :] += 1
    y[:, 1::2, :] += 1
    z[:, :, 1::2] += 1
    return x, y, z

def display_voxels(volume):
    """
    volume: (N,M,P) array
        Represents a binary set of pixels: objects are marked with 1,
        complementary (porosities) with 0.

    The voxels are actually represented with blue transparent surfaces.
    Inner porosities are represented in red.
    """
    # define colors
    red = '#ff0000ff'
    blue = '#1f77b410'
```

(continues on next page)

(continued from previous page)

```
# upscale the above voxel image, leaving gaps
filled = explode(np.ones(volume.shape))

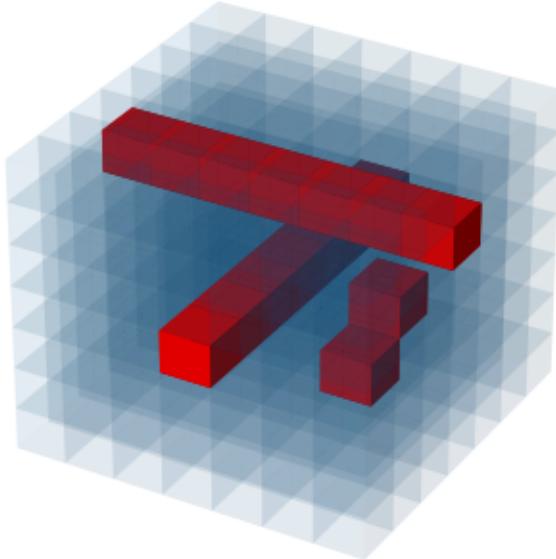
fcolors = explode(np.where(volume, blue, red))

# Shrink the gaps
x, y, z = expand_coordinates(np.indices(np.array(filled.shape) + 1))

# Define 3D figure and place voxels
ax = make_ax()
ax.voxels(x, y, z, filled, facecolors=fcolors)
# Compute Euler number in 6 and 26 neighborhood configuration, that
# correspond to 1 and 3 connectivity, respectively
e26 = euler_number(volume, connectivity=3)
e6 = euler_number(volume, connectivity=1)
plt.title(f'Euler number for N26: {e26}, for N6: {e6}')
plt.show()

# Define a volume of 7x7x7 voxels
n = 7
cube = np.ones((n, n, n), dtype=bool)
# Add a tunnel
c = int(n/2)
cube[c, :, c] = False
# Add a new hole
cube[int(3*n/4), c-1, c-1] = False
# Add a hole in neighborhood of previous one
cube[int(3*n/4), c, c] = False
# Add a second tunnel
cube[:, c, int(3*n/4)] = False
display_voxels(cube)
```

Euler number for N26: 1, for N6: 0



Total running time of the script: (0 minutes 0.613 seconds)

Evaluating segmentation metrics

When trying out different segmentation methods, how do you know which one is best? If you have a *ground truth* or *gold standard* segmentation, you can use various metrics to check how close each automated method comes to the truth. In this example we use an easy-to-segment image as an example of how to interpret various segmentation metrics. We will use the adapted Rand error and the variation of information as example metrics, and see how *oversegmentation* (splitting of true segments into too many sub-segments) and *undersegmentation* (merging of different true segments into a single segment) affect the different scores.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import ndimage as ndi

from skimage import data
from skimage.metrics import (adapted_rand_error,
                             variation_of_information)
from skimage.filters import sobel
from skimage.measure import label
from skimage.util import img_as_float
from skimage.feature import canny
from skimage.morphology import remove_small_objects
from skimage.segmentation import (morphological_geodesic_active_contour,
```

(continues on next page)

(continued from previous page)

```

        inverse_gaussian_gradient,
        watershed,
        mark_boundaries)

image = data.coins()

```

First, we generate the true segmentation. For this simple image, we know exact functions and parameters that will produce a perfect segmentation. In a real scenario, typically you would generate ground truth by manual annotation or “painting” of a segmentation.

```

elevation_map = sobel(image)
markers = np.zeros_like(image)
markers[image < 30] = 1
markers[image > 150] = 2
im_true = watershed(elevation_map, markers)
im_true = ndi.label(ndi.binary_fill_holes(im_true - 1))[0]

```

Next, we create three different segmentations with different characteristics. The first one uses `skimage.segmentation.watershed()` with `compactness`, which is a useful initial segmentation but too fine as a final result. We will see how this causes the oversegmentation metrics to shoot up.

```

edges = sobel(image)
im_test1 = watershed(edges, markers=468, compactness=0.001)

```

The next approach uses the Canny edge filter, `skimage.filters.canny()`. This is a very good edge finder, and gives balanced results.

```

edges = canny(image)
fill_coins = ndi.binary_fill_holes(edges)
im_test2 = ndi.label(remove_small_objects(fill_coins, 21))[0]

```

Finally, we use morphological geodesic active contours, `skimage.segmentation.morphological_geodesic_active_contour()`, a method that generally produces good results, but requires a long time to converge on a good answer. We purposefully cut short the procedure at 100 iterations, so that the final result is *undersegmented*, meaning that many regions are merged into one segment. We will see the corresponding effect on the segmentation metrics.

```

image = img_as_float(image)
gradient = inverse_gaussian_gradient(image)
init_ls = np.zeros(image.shape, dtype=np.int8)
init_ls[10:-10, 10:-10] = 1
im_test3 = morphological_geodesic_active_contour(gradient, num_iter=100,
                                                init_level_set=init_ls,
                                                smoothing=1, balloon=-1,
                                                threshold=0.69)
im_test3 = label(im_test3)

method_names = ['Compact watershed', 'Canny filter',
                'Morphological Geodesic Active Contours']
short_method_names = ['Compact WS', 'Canny', 'GAC']

precision_list = []

```

(continues on next page)

(continued from previous page)

```

recall_list = []
split_list = []
merge_list = []
for name, im_test in zip(method_names, [im_true, im_test1, im_test2, im_test3]):
    error, precision, recall = adapted_rand_error(im_true, im_test)
    splits, merges = variation_of_information(im_true, im_test)
    split_list.append(splits)
    merge_list.append(merges)
    precision_list.append(precision)
    recall_list.append(recall)
    print(f'\n## Method: {name}')
    print(f'Adapted Rand error: {error}')
    print(f'Adapted Rand precision: {precision}')
    print(f'Adapted Rand recall: {recall}')
    print(f'False Splits: {splits}')
    print(f'False Merges: {merges}')

fig, axes = plt.subplots(2, 3, figsize=(9, 6), constrained_layout=True)
ax = axes.ravel()

ax[0].scatter(merge_list, split_list)
for i, txt in enumerate(short_method_names):
    ax[0].annotate(txt, (merge_list[i], split_list[i]),
                   verticalalignment='center')
ax[0].set_xlabel('False Merges (bits)')
ax[0].set_ylabel('False Splits (bits)')
ax[0].set_title('Split Variation of Information')

ax[1].scatter(precision_list, recall_list)
for i, txt in enumerate(short_method_names):
    ax[1].annotate(txt, (precision_list[i], recall_list[i]),
                   verticalalignment='center')
ax[1].set_xlabel('Precision')
ax[1].set_ylabel('Recall')
ax[1].set_title('Adapted Rand precision vs. recall')
ax[1].set_xlim(0, 1)
ax[1].set_ylim(0, 1)

ax[2].imshow(mark_boundaries(image, im_true))
ax[2].set_title('True Segmentation')
ax[2].set_axis_off()

ax[3].imshow(mark_boundaries(image, im_test1))
ax[3].set_title('Compact Watershed')
ax[3].set_axis_off()

ax[4].imshow(mark_boundaries(image, im_test2))
ax[4].set_title('Edge Detection')
ax[4].set_axis_off()

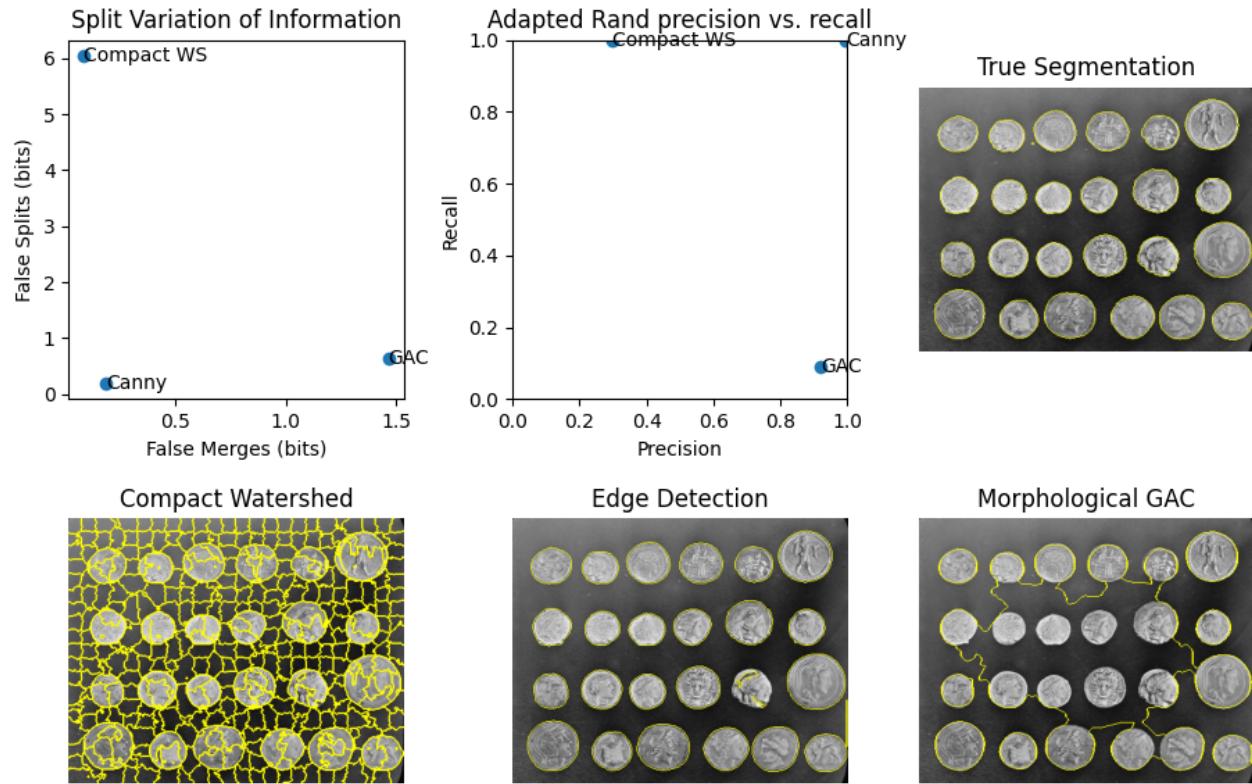
ax[5].imshow(mark_boundaries(image, im_test3))
ax[5].set_title('Morphological GAC')

```

(continues on next page)

(continued from previous page)

```
ax[5].set_axis_off()
plt.show()
```



```
## Method: Compact watershed
Adapted Rand error: 0.5421684624091794
Adapted Rand precision: 0.2968781380256405
Adapted Rand recall: 0.9999664222191392
False Splits: 6.036024332525563
False Merges: 0.0825883711820654

## Method: Canny filter
Adapted Rand error: 0.0027247598212836177
Adapted Rand precision: 0.9946425605360896
Adapted Rand recall: 0.9999218934767155
False Splits: 0.20042002116129515
False Merges: 0.18076872508600775

## Method: Morphological Geodesic Active Contours
Adapted Rand error: 0.8346015951433162
Adapted Rand precision: 0.9191321393095933
Adapted Rand recall: 0.09087577915161697
False Splits: 0.6466330168716372
False Merges: 1.4656270133195097
```

Total running time of the script: (0 minutes 2.056 seconds)

Use rolling-ball algorithm for estimating background intensity

The rolling-ball algorithm estimates the background intensity of a grayscale image in case of uneven exposure. It is frequently used in biomedical image processing and was first proposed by Stanley R. Sternberg in 1983¹.

The algorithm works as a filter and is quite intuitive. We think of the image as a surface that has unit-sized blocks stacked on top of each other in place of each pixel. The number of blocks, and hence surface height, is determined by the intensity of the pixel. To get the intensity of the background at a desired (pixel) position, we imagine submerging a ball under the surface at the desired position. Once it is completely covered by the blocks, the apex of the ball determines the intensity of the background at that position. We can then *roll* this ball around below the surface to get the background values for the entire image.

Scikit-image implements a general version of this rolling-ball algorithm, which allows you to not just use balls, but arbitrary shapes as kernel and works on n-dimensional ndimages. This allows you to directly filter RGB images or filter image stacks along any (or all) spacial dimensions.

Classic rolling ball

In scikit-image, the rolling ball algorithm assumes that your background has low intensity (black), whereas the features have high intensity (white). If this is the case for your image, you can directly use the filter like so:

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

from skimage import (
    data, restoration, util
)

def plot_result(image, background):
    fig, ax = plt.subplots(nrows=1, ncols=3)

    ax[0].imshow(image, cmap='gray')
    ax[0].set_title('Original image')
    ax[0].axis('off')

    ax[1].imshow(background, cmap='gray')
    ax[1].set_title('Background')
    ax[1].axis('off')

    ax[2].imshow(image - background, cmap='gray')
    ax[2].set_title('Result')
    ax[2].axis('off')

    fig.tight_layout()

image = data.coins()

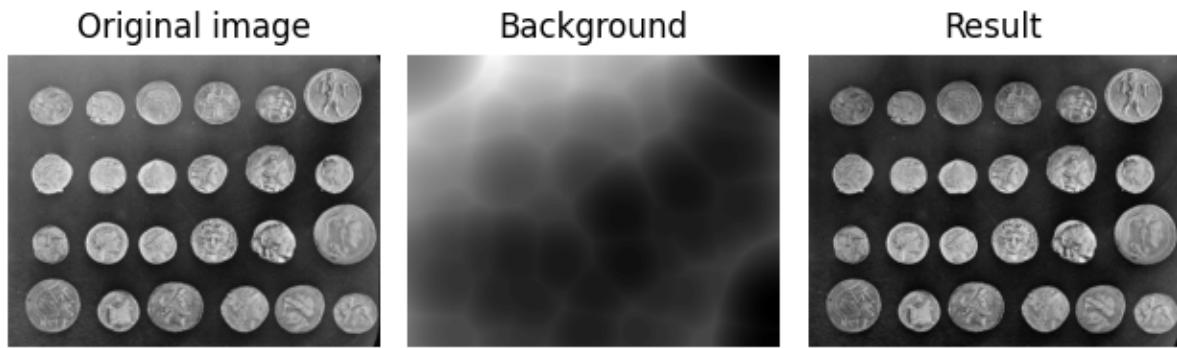
background = restoration.rolling_ball(image)
```

(continues on next page)

¹ Sternberg, Stanley R. "Biomedical image processing." Computer 1 (1983): 22-34. DOI:10.1109/MC.1983.1654163

(continued from previous page)

```
plot_result(image, background)
plt.show()
```



White background

If you have dark features on a bright background, you need to invert the image before you pass it into the algorithm, and then invert the result. This can be accomplished via:

```
image = data.page()
image_inverted = util.invert(image)

background_inverted = restoration.rolling_ball(image_inverted, radius=45)
filtered_image_inverted = image_inverted - background_inverted
filtered_image = util.invert(filtered_image_inverted)
background = util.invert(background_inverted)

fig, ax = plt.subplots(nrows=1, ncols=3)

ax[0].imshow(image, cmap='gray')
ax[0].set_title('Original image')
ax[0].axis('off')
```

(continues on next page)

(continued from previous page)

```

ax[1].imshow(background, cmap='gray')
ax[1].set_title('Background')
ax[1].axis('off')

ax[2].imshow(filtered_image, cmap='gray')
ax[2].set_title('Result')
ax[2].axis('off')

fig.tight_layout()

plt.show()

```

Original image

Region-based segmentation

Let us first determine markers of the coins and the background. These markers are pixels that we can label unambiguously as either object or background. Here, the markers are found at the two extreme parts of the histogram of grey values:

```
>>> markers = np.zeros_like(coins)
```

Background



Result

Region-based segmentation

Let us first determine markers of the coins and the background. These markers are pixels that we can label unambiguously as either object or background. Here, the markers are found at the two extreme parts of the histogram of grey values:

```
>>> markers = np.zeros_like(coins)
```

Be careful not to fall victim to an integer underflow when subtracting a bright background. For example, this code looks correct, but may suffer from an underflow leading to unwanted artifacts. You can see this in the top right corner of the visualization.

```

image = data.page()
image_inverted = util.invert(image)

background_inverted = restoration.rolling_ball(image_inverted, radius=45)
background = util.invert(background_inverted)

```

(continues on next page)

(continued from previous page)

```
underflow_image = image - background # integer underflow occurs here

# correct subtraction
correct_image = util.invert(image_inverted - background_inverted)

fig, ax = plt.subplots(nrows=1, ncols=2)

ax[0].imshow(underflow_image, cmap='gray')
ax[0].set_title('Background Removal with Underflow')
ax[0].axis('off')

ax[1].imshow(correct_image, cmap='gray')
ax[1].set_title('Correct Background Removal')
ax[1].axis('off')

fig.tight_layout()

plt.show()
```

Background Removal with Underflow

Region-based segmentation

Let us first determine markers of the coins and the background. These markers are pixels that we can label unambiguously as either object or background. Here, the markers are found at the two extreme parts of the histogram of grey values:

```
>>> markers = np.zeros_like(coins)
```

Correct Background Removal

Region-based segmentation

Let us first determine markers of the coins and the background. These markers are pixels that we can label unambiguously as either object or background. Here, the markers are found at the two extreme parts of the histogram of grey values:

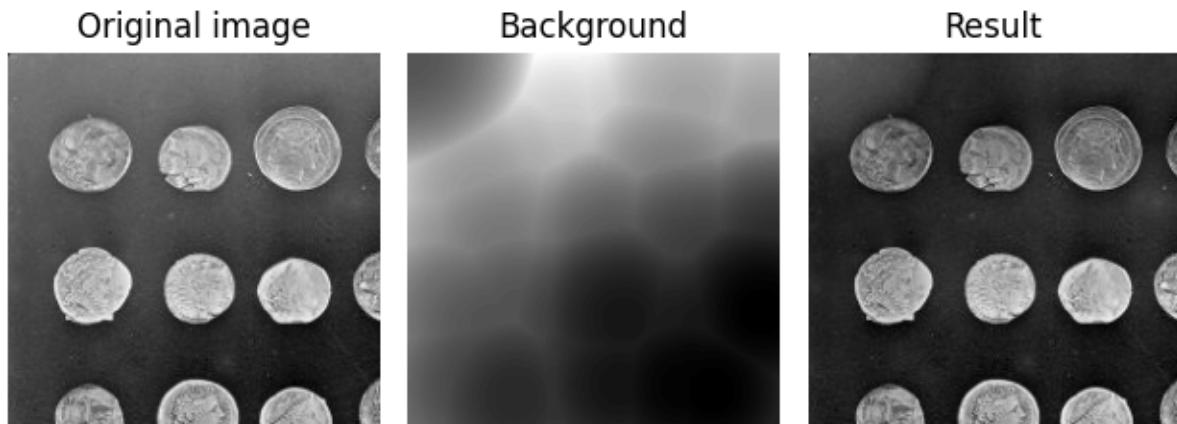
```
>>> markers = np.zeros_like(coins)
```

Image Datatypes

`rolling_ball` can handle datatypes other than `np.uint8`. You can pass them into the function in the same way.

```
image = data.coins()[:200, :200].astype(np.uint16)

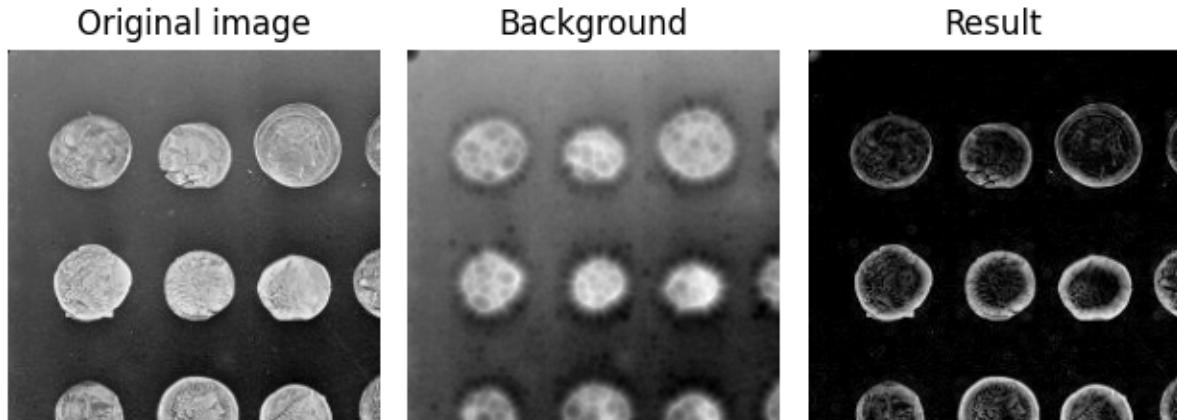
background = restoration.rolling_ball(image, radius=70.5)
plot_result(image, background)
plt.show()
```



However, you need to be careful if you use floating point images that have been normalized to [0, 1]. In this case the ball will be much larger than the image intensity, which can lead to unexpected results.

```
image = util.img_as_float(data.coins()[:200, :200])

background = restoration.rolling_ball(image, radius=70.5)
plot_result(image, background)
plt.show()
```



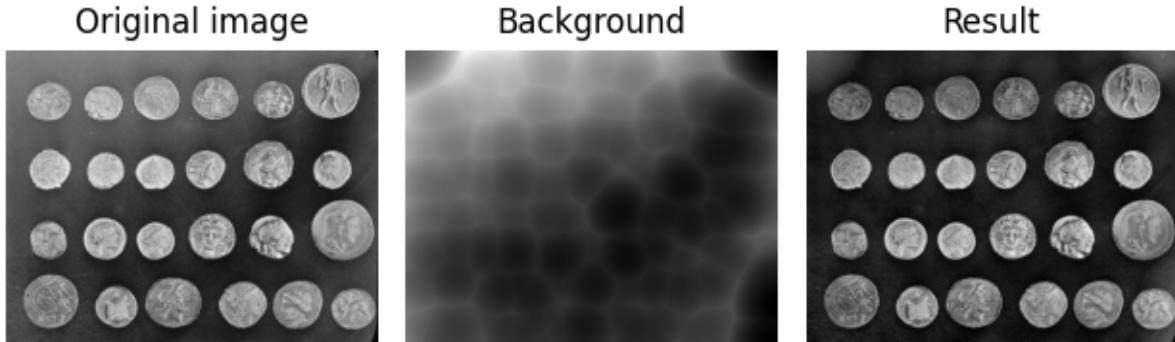
Because `radius=70.5` is much larger than the maximum intensity of the image, the effective kernel size is reduced significantly, i.e., only a small cap (approximately `radius=10`) of the ball is rolled around in the image. You can find a reproduction of this strange effect in the Advanced Shapes section below.

To get the expected result, you need to reduce the intensity of the kernel. This is done by specifying the kernel manually using the `kernel` argument.

Note: The radius is equal to the length of a semi-axis of an ellipsis, which is *half* a full axis. Hence, the kernel shape is multiplied by two.

```
normalized_radius = 70.5 / 255
image = util.img_as_float(data.coins())
kernel = restoration.ellipsoid_kernel(
    (70.5 * 2, 70.5 * 2),
    normalized_radius * 2
)

background = restoration.rolling_ball(
    image,
    kernel=kernel
)
plot_result(image, background)
plt.show()
```



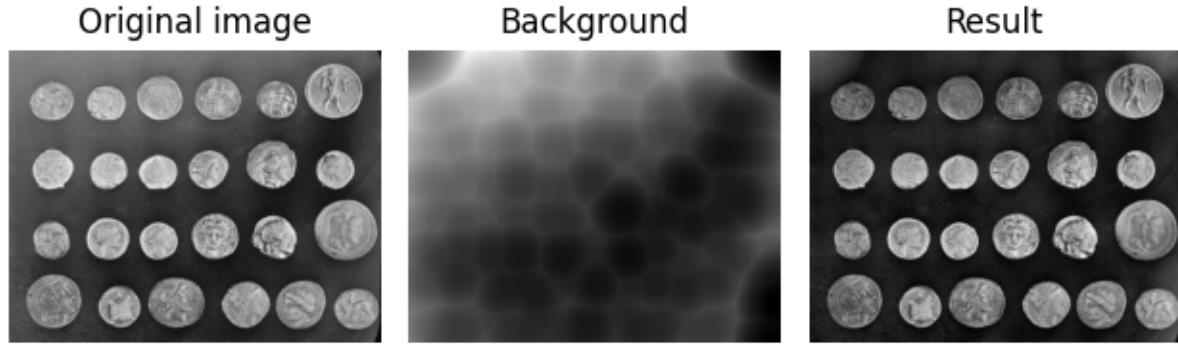
Advanced Shapes

By default, `rolling_ball` uses a ball shaped kernel (surprise). Sometimes, this can be too limiting - as in the example above -, because the intensity dimension has a different scale compared to the spatial dimensions, or because the image dimensions may have different meanings - one could be a stack counter in an image stack.

To account for this, `rolling_ball` has a `kernel` argument which allows you to specify the kernel to be used. A kernel must have the same dimensionality as the image (Note: dimensionality, not shape). To help with its creation, two default kernels are provided by `skimage.ball_kernel` specifies a ball shaped kernel and is used as the default kernel. `ellipsoid_kernel` specifies an ellipsoid shaped kernel.

```
image = data.coins()
kernel = restoration.ellipsoid_kernel(
    (70.5 * 2, 70.5 * 2),
    70.5 * 2
)

background = restoration.rolling_ball(
    image,
    kernel=kernel
)
plot_result(image, background)
plt.show()
```

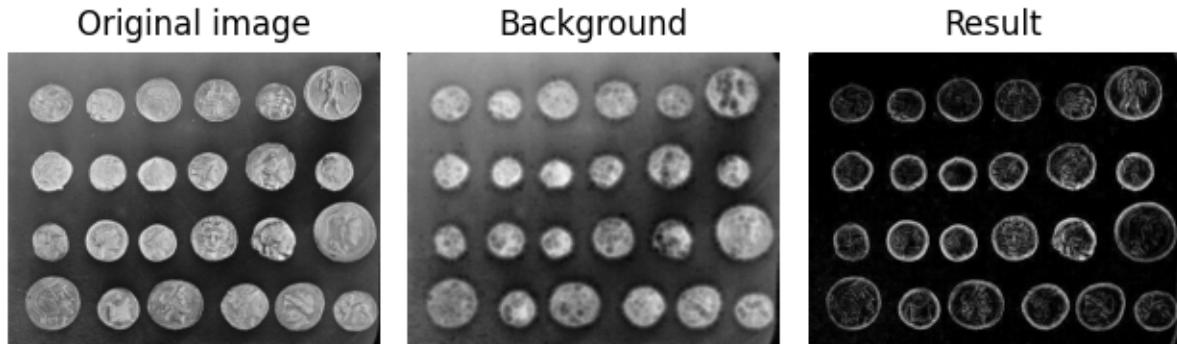


You can also use `ellipsoid_kernel` to recreate the previous, unexpected result and see that the effective (spatial) filter size was reduced.

```
image = data.coins()

kernel = restoration.ellipsoid_kernel(
    (10 * 2, 10 * 2),
    255 * 2
)

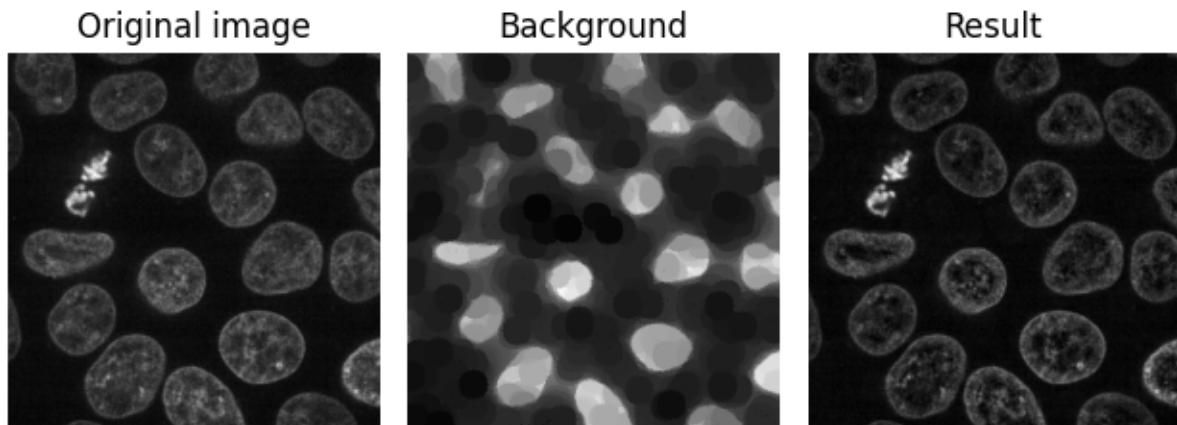
background = restoration.rolling_ball(
    image,
    kernel=kernel
)
plot_result(image, background)
plt.show()
```



Higher Dimensions

Another feature of `rolling_ball` is that you can directly apply it to higher dimensional images, e.g., a z-stack of images obtained during confocal microscopy. The number of kernel dimensions must match the image dimensions, hence the kernel shape is now 3 dimensional.

```
image = data.cells3d()[:, 1, ...]
background = restoration.rolling_ball(
    image,
    kernel=restoration.ellipsoid_kernel(
        (1, 21, 21),
        0.1
    )
)
plot_result(image[30, ...], background[30, ...])
plt.show()
```

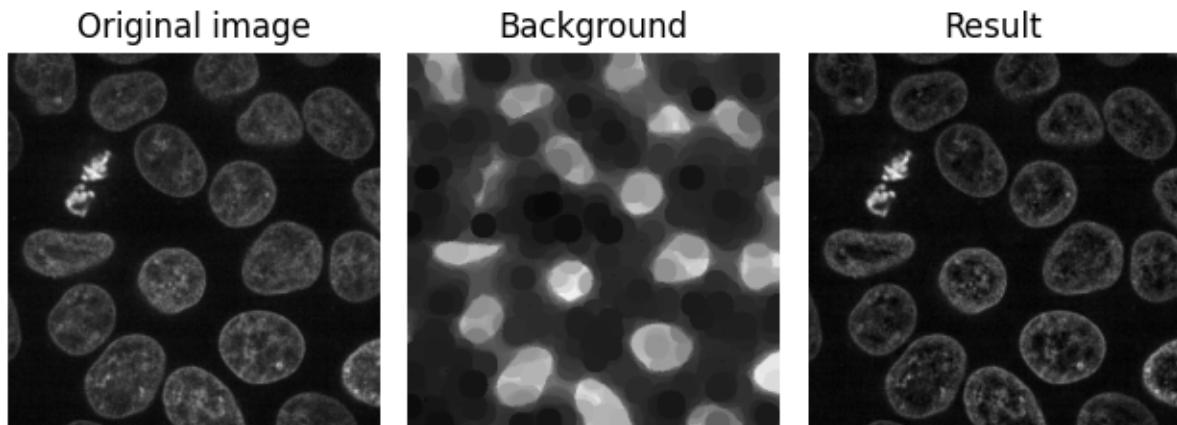


A kernel size of 1 does not filter along this axis. In other words, above filter is applied to each image in the stack individually.

However, you can also filter along all 3 dimensions at the same time by specifying a value other than 1.

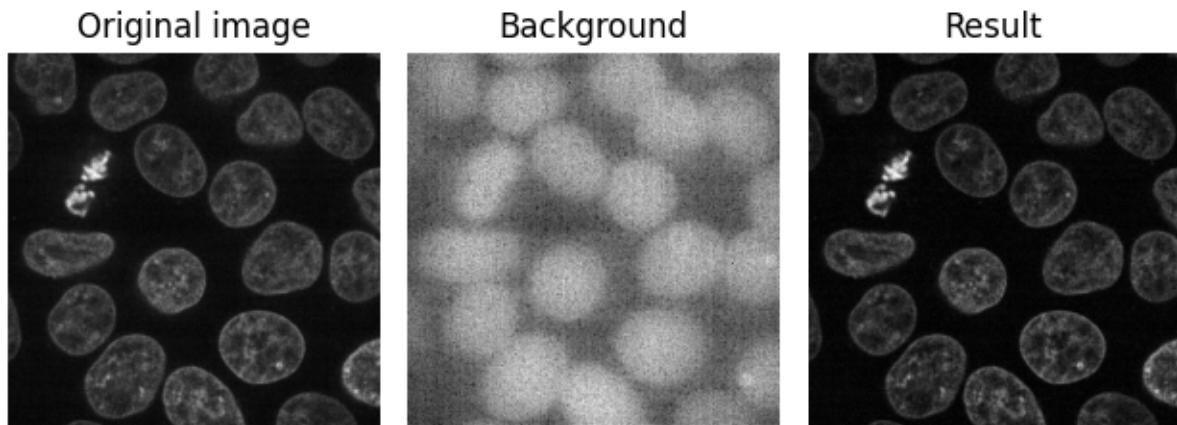
```
image = data.cells3d()[:, 1, ...]
background = restoration.rolling_ball(
    image,
    kernel=restoration.ellipsoid_kernel(
        (5, 21, 21),
        0.1
    )
)

plot_result(image[30, ...], background[30, ...])
plt.show()
```



Another possibility is to filter individual pixels along the planar axis (z-stack axis).

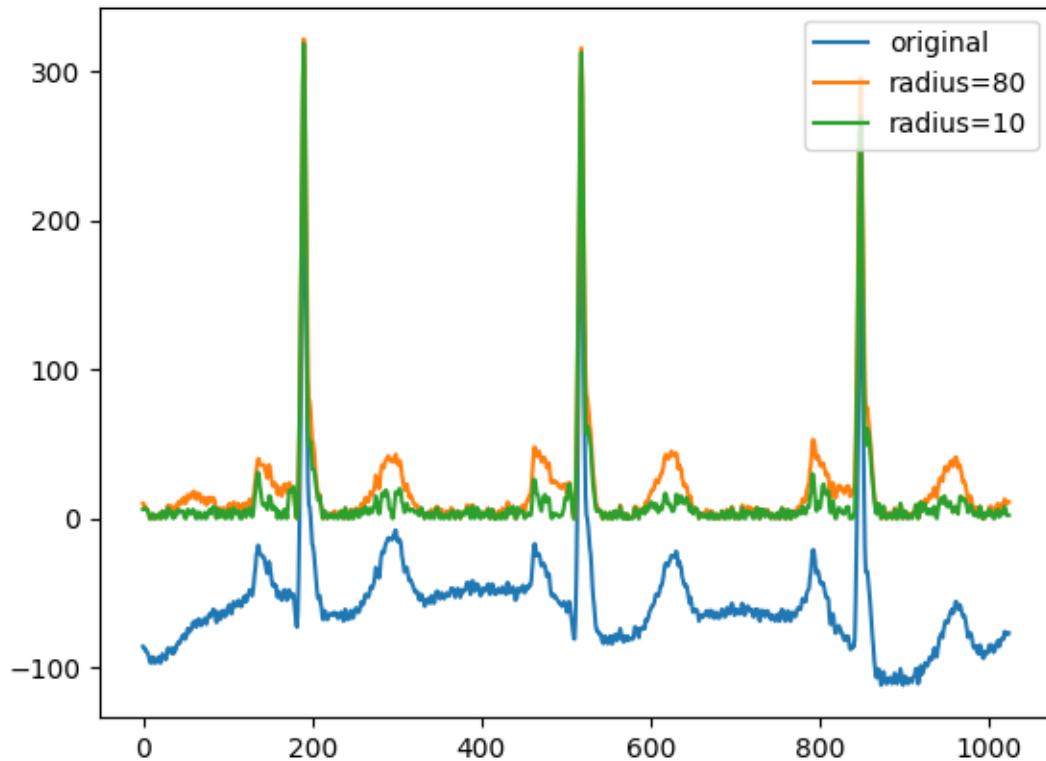
```
image = data.cells3d()[:, 1, ...]
background = restoration.rolling_ball(
    image,
    kernel=restoration.ellipsoid_kernel(
        (100, 1, 1),
        0.1
    )
)
plot_result(image[30, ...], background[30, ...])
plt.show()
```



1D Signal Filtering

As another example of the n-dimensional feature of `rolling_ball`, we show an implementation for 1D data. Here, we are interested in removing the background signal of an ECG waveform to detect prominent peaks (higher values than the local baseline). Smoother peaks can be removed with smaller values of the radius.

```
x = pywt.data.ecg()
background = restoration.rolling_ball(x, radius=80)
background2 = restoration.rolling_ball(x, radius=10)
plt.figure()
plt.plot(x, label='original')
plt.plot(x - background, label='radius=80')
plt.plot(x - background2, label='radius=10')
plt.legend()
plt.show()
```



Total running time of the script: (1 minutes 18.235 seconds)

Longer examples and demonstrations

Render text onto an image

Scikit-image currently doesn't feature a function that allows you to write text onto an image. However, there is a fairly easy workaround using scikit-image's optional dependency `matplotlib`.

```
import matplotlib.pyplot as plt
import numpy as np
from skimage import data

img = data.cat()

fig = plt.figure()
fig.figimage(img, resize=True)
fig.text(0, 0.99, "I am stefan's cat.", fontsize=32, va="top")
fig.canvas.draw()
annotated_img = np.asarray(fig.canvas.renderer.buffer_rgba())
plt.close(fig)
```

For the purpose of this example, we can also show the image; however, if one just wants to write onto the image, this step is not necessary.

```
fig, ax = plt.subplots()
ax.imshow(annotated_img)
ax.set_axis_off()
ax.set_position([0, 0, 1, 1])
plt.show()
```



Total running time of the script: (0 minutes 0.214 seconds)

Face detection using a cascade classifier

This computer vision example shows how to detect faces on an image using object detection framework based on machine learning.

First, you will need an xml file, from which the trained data can be read. The framework works with files, trained using Multi-block Local Binary Patterns Features (See [MB-LBP](#)) and Gentle Adaboost with attentional cascade. So, the detection framework will also work with [xml files from OpenCV](#). There you can find files that were trained to detect cat faces, profile faces and other things. But if you want to detect frontal faces, the respective file is already included in scikit-image.

Next you will have to specify the parameters for the `detect_multi_scale` function. Here you can find the meaning of each of them.

First one is `scale_ratio`. To find all faces, the algorithm does the search on multiple scales. This is done by changing the size of searching window. The smallest window size is the size of window that was used in training. This size is specified in the xml file with trained parameters. The `scale_ratio` parameter specifies by which ratio the search window is increased on each step. If you increase this parameter, the search time decreases and the accuracy decreases.

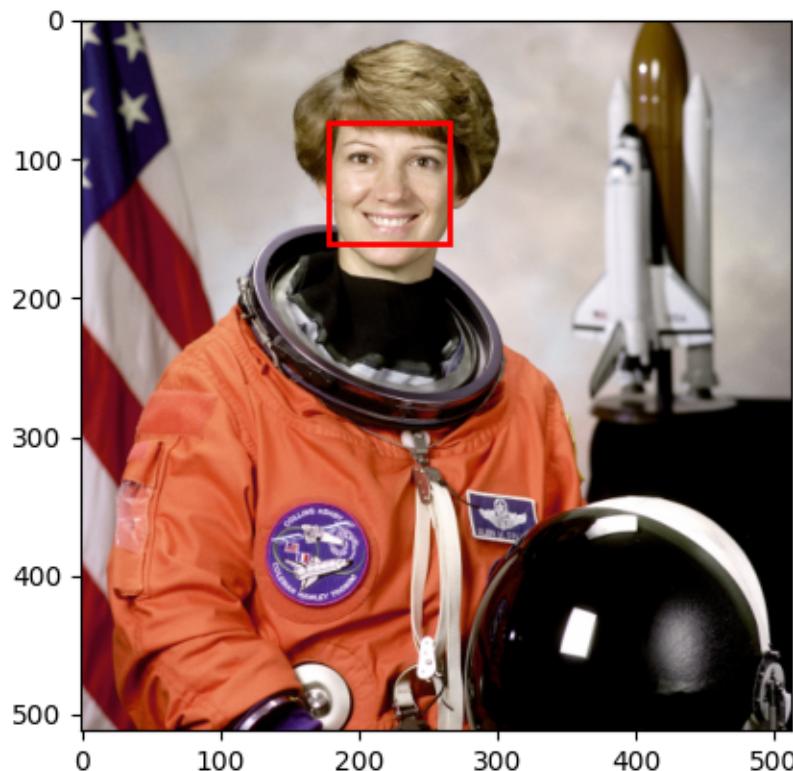
So, faces on some scales can be not detected.

`step_ratio` specifies the step of sliding window that is used to search for faces on each scale of the image. If this parameter is equal to one, then all the possible locations are searched. If the parameter is greater than one, for example, two, the window will be moved by two pixels and not all of the possible locations will be searched for faces. By increasing this parameter we can reduce the working time of the algorithm, but the accuracy will also be decreased.

`min_size` is the minimum size of search window during the scale search. `max_size` specifies the maximum size of the window. If you know the size of faces on the images that you want to search, you should specify these parameters as precisely as possible, because you can avoid doing expensive computations and possibly decrease the amount of false detections. You can save a lot of time by increasing the `min_size` parameter, because the majority of time is spent on searching on the smallest scales.

`min_neighbor_number` and `intersection_score_threshold` parameters are made to cluster the excessive detections of the same face and to filter out false detections. True faces usually has a lot of detections around them and false ones usually have single detection. First algorithm searches for clusters: two rectangle detections are placed in the same cluster if the intersection score between them is larger then `intersection_score_threshold`. The intersection score is computed using the equation $(\text{intersection area}) / (\text{small rectangle ratio})$. The described intersection criteria was chosen over intersection over union to avoid a corner case when small rectangle inside of a big one have small intersection score. Then each cluster is thresholded using `min_neighbor_number` parameter which leaves the clusters that have a same or bigger number of detections in them.

You should also take into account that false detections are inevitable and if you want to have a really precise detector, you will have to train it yourself using [OpenCV train cascade utility](#).



```

from skimage import data
from skimage.feature import Cascade

import matplotlib.pyplot as plt
from matplotlib import patches

# Load the trained file from the module root.
trained_file = data.lbp_frontal_face_cascade_filename()

# Initialize the detector cascade.
detector = Cascade(trained_file)

img = data.astronaut()

detected = detector.detect_multi_scale(img=img,
                                         scale_factor=1.2,
                                         step_ratio=1,
                                         min_size=(60, 60),
                                         max_size=(123, 123))

plt.imshow(img)
img_desc = plt.gca()
plt.set_cmap('gray')

for patch in detected:

    img_desc.add_patch(
        patches.Rectangle(
            (patch['c'], patch['r']),
            patch['width'],
            patch['height'],
            fill=False,
            color='r',
            linewidth=2
        )
    )

plt.show()

```

Total running time of the script: (0 minutes 0.272 seconds)

Interact with 3D images (of kidney tissue)

In this tutorial, we explore interactively a biomedical image which has three spatial dimensions and three color dimensions (channels). For a general introduction to 3D image processing, please refer to *Explore 3D images (of cells)*. The data we use here correspond to kidney tissue which was imaged with confocal fluorescence microscopy (more details at¹ under `kidney-tissue-fluorescence.tif`).

```

import matplotlib.pyplot as plt
import numpy as np

import plotly

```

(continues on next page)

¹ <https://gitlab.com/scikit-image/data/#data>

(continued from previous page)

```
import plotly.express as px
from skimage import data
```

Load image

This biomedical image is available through *scikit-image*'s data registry.

```
data = data.kidney()
```

The returned dataset is a 3D multichannel image:

```
print(f'number of dimensions: {data.ndim}')
print(f'shape: {data.shape}')
print(f'dtype: {data.dtype}')
```

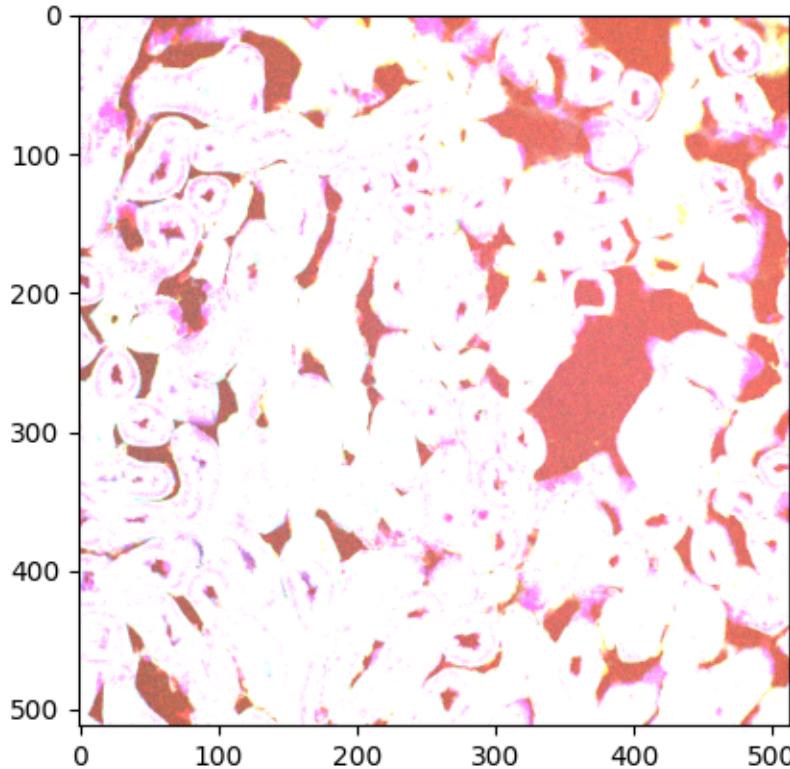
```
number of dimensions: 4
shape: (16, 512, 512, 3)
dtype: uint16
```

Dimensions are provided in the following order: (z, y, x, c), i.e., [plane, row, column, channel].

```
n_plane, n_row, n_col, n_chan = data.shape
```

Let us consider only a slice (2D plane) of the data for now. More specifically, let us consider the slice located halfway in the stack. The *imshow* function can display both grayscale and RGB(A) 2D images.

```
_, ax = plt.subplots()
ax.imshow(data[n_plane // 2])
```



```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..  
→255] for integers).
```

```
<matplotlib.image.AxesImage object at 0x7ff48536e9a0>
```

According to the warning message, the range of values is unexpected. The image rendering is clearly not satisfactory color-wise.

```
vmin, vmax = data.min(), data.max()  
print(f'range: ({vmin}, {vmax})')
```

```
range: (10, 4095)
```

We turn to plotly's implementation of the `imshow` function, for it supports value ranges beyond `(0.0, 1.0)` for floats and `(0, 255)` for integers.

```
fig = px.imshow(data[n_plane // 2], zmax=vmax)  
plotly.io.show(fig)
```

Here you go, *fluorescence* microscopy!

Normalize range for each channel

Generally speaking, we may want to normalize the value range on a per-channel basis. Let us facet our data (slice) along the channel axis. This way, we get three single-channel images, where the max value of each image is used:

```
fig = px.imshow(
    data[n_plane // 2],
    facet_col=2,
    binary_string=True,
    labels={'facet_col': 'channel'}
)
plotly.io.show(fig)
```

What is the range of values for each color channel? We check by taking the min and max across all non-channel axes.

```
vmin_0, vmin_1, vmin_2 = np.min(data, axis=(0, 1, 2))
vmax_0, vmax_1, vmax_2 = np.max(data, axis=(0, 1, 2))
print(f'range for channel 0: ({vmin_0}, {vmax_0})')
print(f'range for channel 1: ({vmin_1}, {vmax_1})')
print(f'range for channel 2: ({vmin_2}, {vmax_2})')
```

```
range for channel 0: (10, 4095)
range for channel 1: (68, 4095)
range for channel 2: (35, 4095)
```

Let us be very specific and pass value ranges on a per-channel basis:

```
fig = px.imshow(
    data[n_plane // 2],
    zmin=[vmin_0, vmin_1, vmin_2],
    zmax=[vmax_0, vmax_1, vmax_2]
)
plotly.io.show(fig)
```

Plotly lets you interact with this visualization by panning, zooming in and out, and exporting the desired figure as a static image in PNG format.

Explore slices as animation frames

Click the play button to move along the z axis, through the stack of all 16 slices.

```
fig = px.imshow(
    data,
    zmin=[vmin_0, vmin_1, vmin_2],
    zmax=[vmax_0, vmax_1, vmax_2],
    animation_frame=0,
    binary_string=True,
    labels={'animation_frame': 'plane'}
)
plotly.io.show(fig)
```

Combine channel facetting and slice animation

```
fig = px.imshow(
    data,
    animation_frame=0,
    facet_col=3,
    binary_string=True,
    labels={'facet_col': 'channel', 'animation_frame': 'plane'}
)
plotly.io.show(fig)
```

The biologist's eye can spot that the two bright blobs (best seen in `channel=2`) are kidney glomeruli².

```
plt.show()
```

Total running time of the script: (0 minutes 9.081 seconds)

Use pixel graphs to find an object's geodesic center

In various image analysis situations, it is useful to think of the pixels of an image, or of a region of an image, as a network or graph, in which each pixel is connected to its neighbors (with or without diagonals). One such situation is finding the *geodesic center* of an object, which is the point closest to all other points *if you are only allowed to travel on the pixels of the object*, rather than in a straight line. This point is the one with maximal *closeness centrality*¹ in the network.

In this example, we create such a *pixel graph* of a skeleton and find the central pixel of that skeleton. This demonstrates its utility in contrast with the centroid (also known as the center of mass) which may actually fall outside the object.

References

```
import matplotlib.pyplot as plt
import numpy as np
from scipy import ndimage as ndi
from skimage import color, data, filters, graph, measure, morphology
```

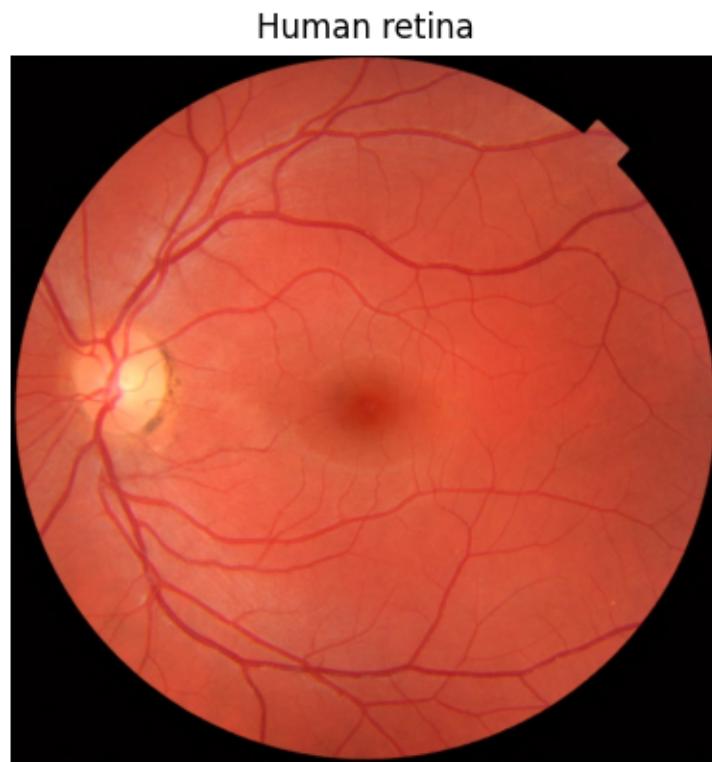
We start by loading the data: an image of a human retina.

```
retina_source = data.retina()

_, ax = plt.subplots()
ax.imshow(retina_source)
ax.set_axis_off()
_ = ax.set_title('Human retina')
```

² [https://en.wikipedia.org/wiki/Glomerulus_\(kidney\)](https://en.wikipedia.org/wiki/Glomerulus_(kidney))

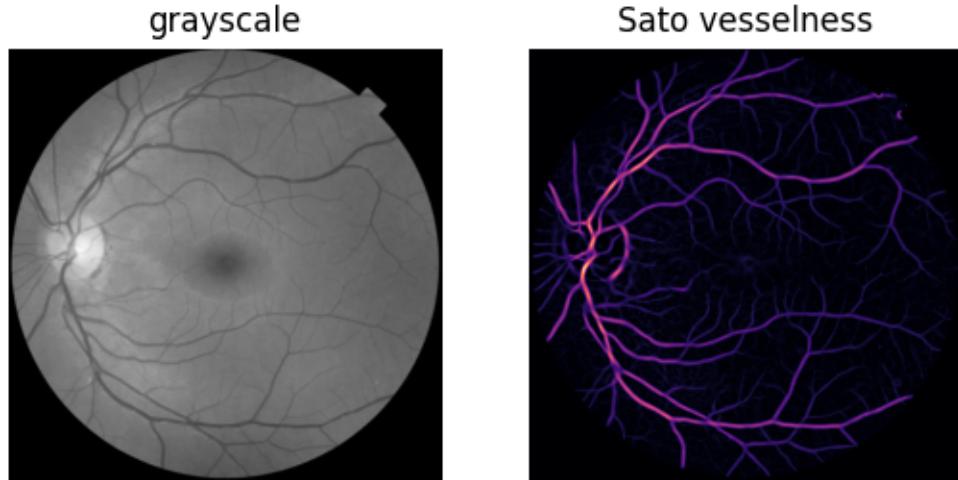
¹ Linton C. Freeman: Centrality in networks: I. Conceptual clarification. Social Networks 1:215-239, 1979. DOI:10.1016/0378-8733(78)90021-



We convert the image to grayscale, then use the *Sato vesselness filter* to better distinguish the main vessels in the image.

```
retina = color.rgb2gray(retina_source)
t0, t1 = filters.threshold_multiotsu(retina, classes=3)
mask = (retina > t0)
vessels = filters.sato(retina, sigmas=range(1, 10)) * mask

_, axes = plt.subplots(nrows=1, ncols=2)
axes[0].imshow(retina, cmap='gray')
axes[0].set_axis_off()
axes[0].set_title('grayscale')
axes[1].imshow(vessels, cmap='magma')
axes[1].set_axis_off()
_ = axes[1].set_title('Sato vesselness')
```

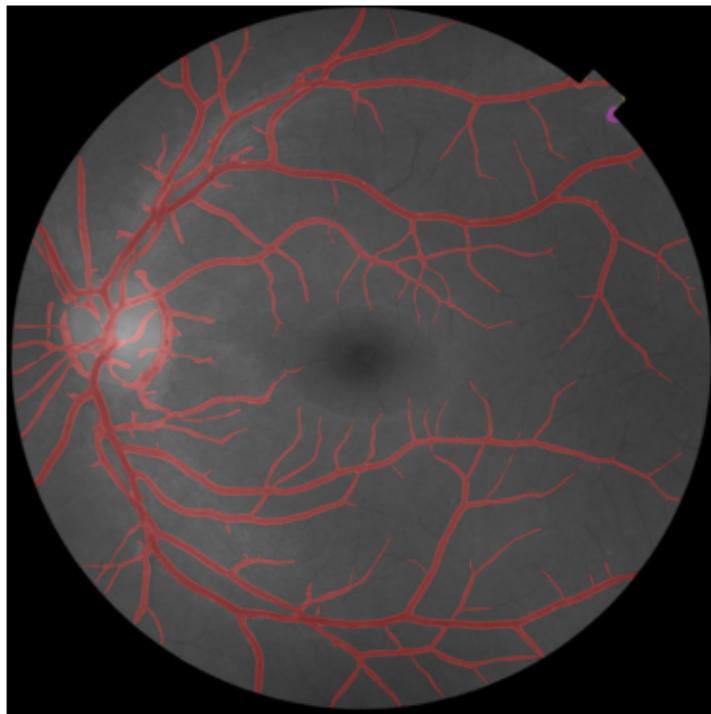


Based on the observed vesselness values, we use *hysteresis thresholding* to define the main vessels.

```
thresholded = filters.apply_hysteresis_threshold(vessels, 0.01, 0.03)
labeled = ndi.label(thresholded)[0]

_, ax = plt.subplots()
ax.imshow(color.label2rgb(labeled, retina))
ax.set_axis_off()
_ = ax.set_title('thresholded vesselness')
```

thresholded vesselness



Finally, we can *skeletonize* this label image and use that as the basis to find the *central pixel* in that skeleton. Compare that to the position of the centroid!

```

largest_nonzero_label = np.argmax(np.bincount(labeled[labeled > 0]))
binary = labeled == largest_nonzero_label
skeleton = morphology.skeletonize(binary)
g, nodes = graph.pixel_graph(skeleton, connectivity=2)
px, distances = graph.central_pixel(
    g, nodes=nodes, shape=skeleton.shape, partition_size=100
)

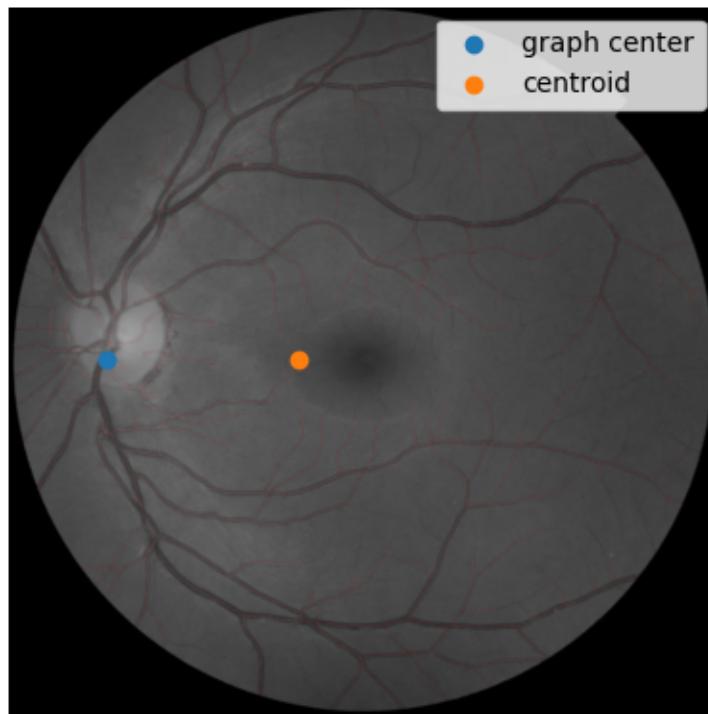
centroid = measure.centroid(labeled > 0)

_, ax = plt.subplots()
ax.imshow(color.label2rgb(skeleton, retina))
ax.scatter(px[1], px[0], label='graph center')
ax.scatter(centroid[1], centroid[0], label='centroid')
ax.legend()
ax.set_axis_off()
ax.set_title('vessel graph center vs centroid')

plt.show()

```

vessel graph center vs centroid



Total running time of the script: (1 minutes 13.998 seconds)

Visual image comparison

Image comparison is particularly useful when performing image processing tasks such as exposure manipulations, filtering, and restoration.

This example shows how to easily compare two images with various approaches.

```
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec

from skimage import data, transform, exposure
from skimage.util import compare_images


img1 = data.coins()
img1_equalized = exposure.equalize_hist(img1)
img2 = transform.rotate(img1, 2)

comp_equalized = compare_images(img1, img1_equalized, method='checkerboard')
diff_rotated = compare_images(img1, img2, method='diff')
blend_rotated = compare_images(img1, img2, method='blend')
```

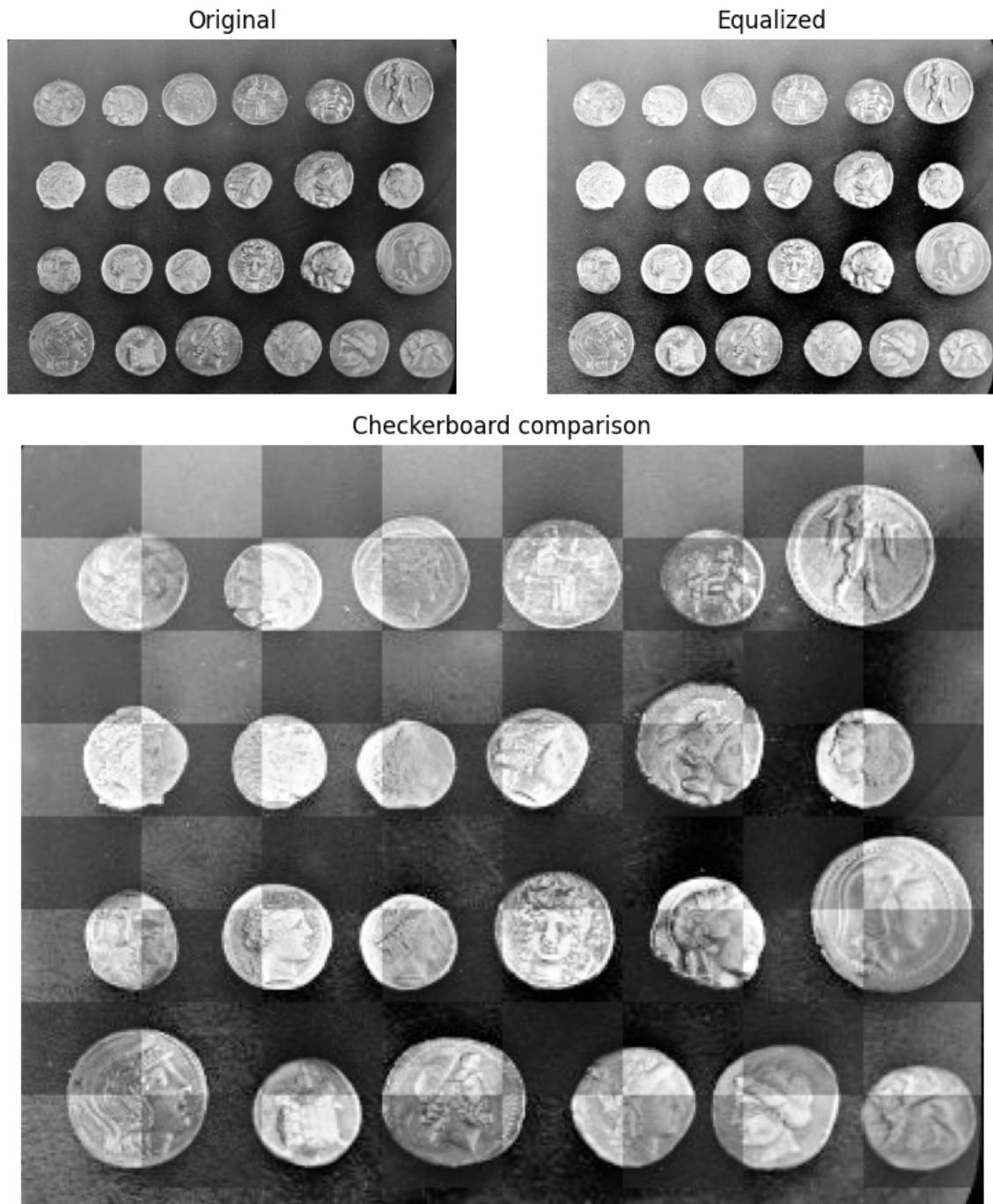
Checkerboard

The *checkerboard* method alternates tiles from the first and the second images.

```
fig = plt.figure(figsize=(8, 9))

gs = GridSpec(3, 2)
ax0 = fig.add_subplot(gs[0, 0])
ax1 = fig.add_subplot(gs[0, 1])
ax2 = fig.add_subplot(gs[1:, :])

ax0.imshow(img1, cmap='gray')
ax0.set_title('Original')
ax1.imshow(img1_equalized, cmap='gray')
ax1.set_title('Equalized')
ax2.imshow(comp_equalized, cmap='gray')
ax2.set_title('Checkerboard comparison')
for a in (ax0, ax1, ax2):
    a.axis('off')
plt.tight_layout()
plt.plot()
```



[]

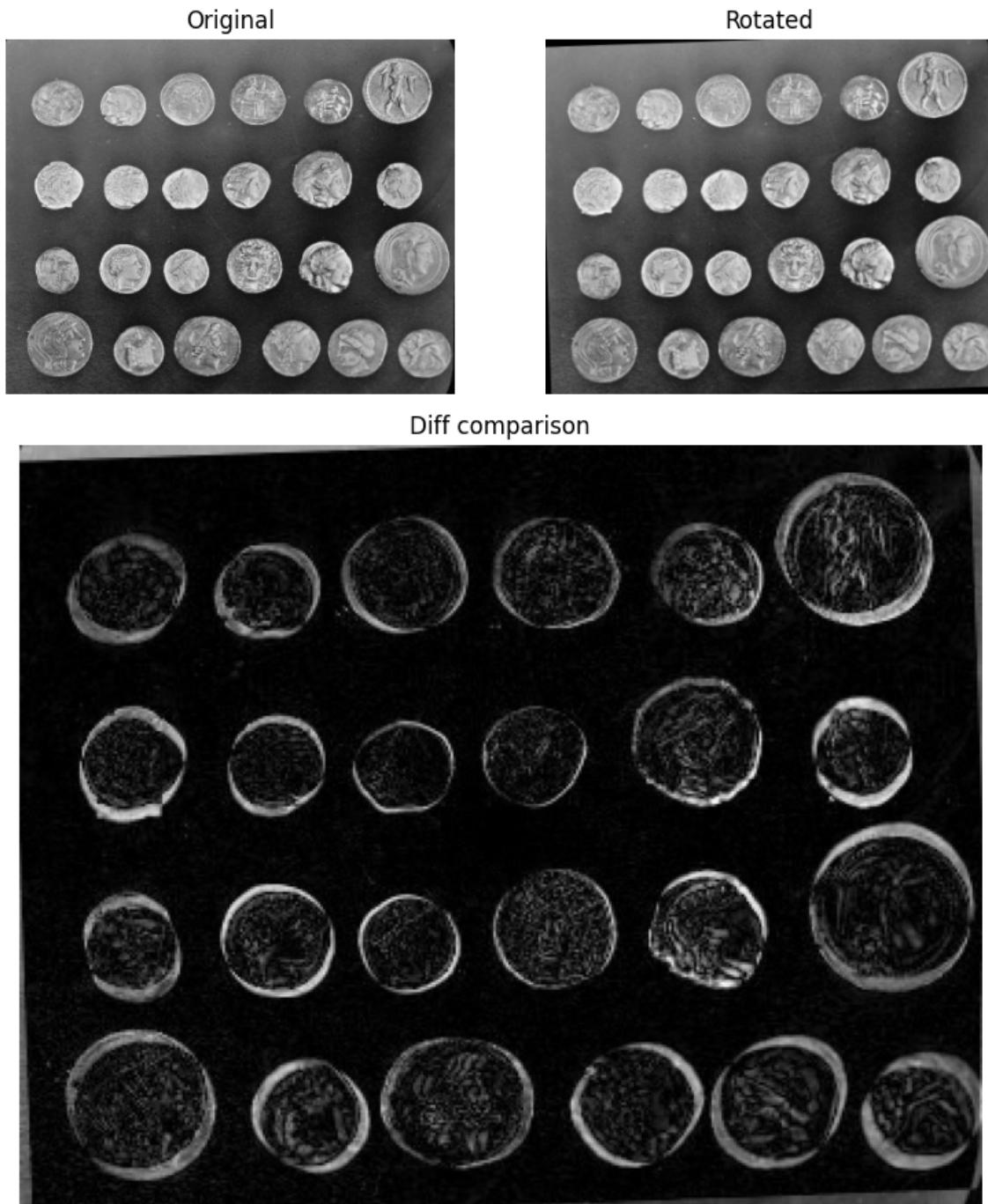
Diff

The `diff` method computes the absolute difference between the two images.

```
fig = plt.figure(figsize=(8, 9))

gs = GridSpec(3, 2)
ax0 = fig.add_subplot(gs[0, 0])
ax1 = fig.add_subplot(gs[0, 1])
ax2 = fig.add_subplot(gs[1:, :])

ax0.imshow(img1, cmap='gray')
ax0.set_title('Original')
ax1.imshow(img2, cmap='gray')
ax1.set_title('Rotated')
ax2.imshow(diff_rotated, cmap='gray')
ax2.set_title('Diff comparison')
for a in (ax0, ax1, ax2):
    a.axis('off')
plt.tight_layout()
plt.plot()
```



[

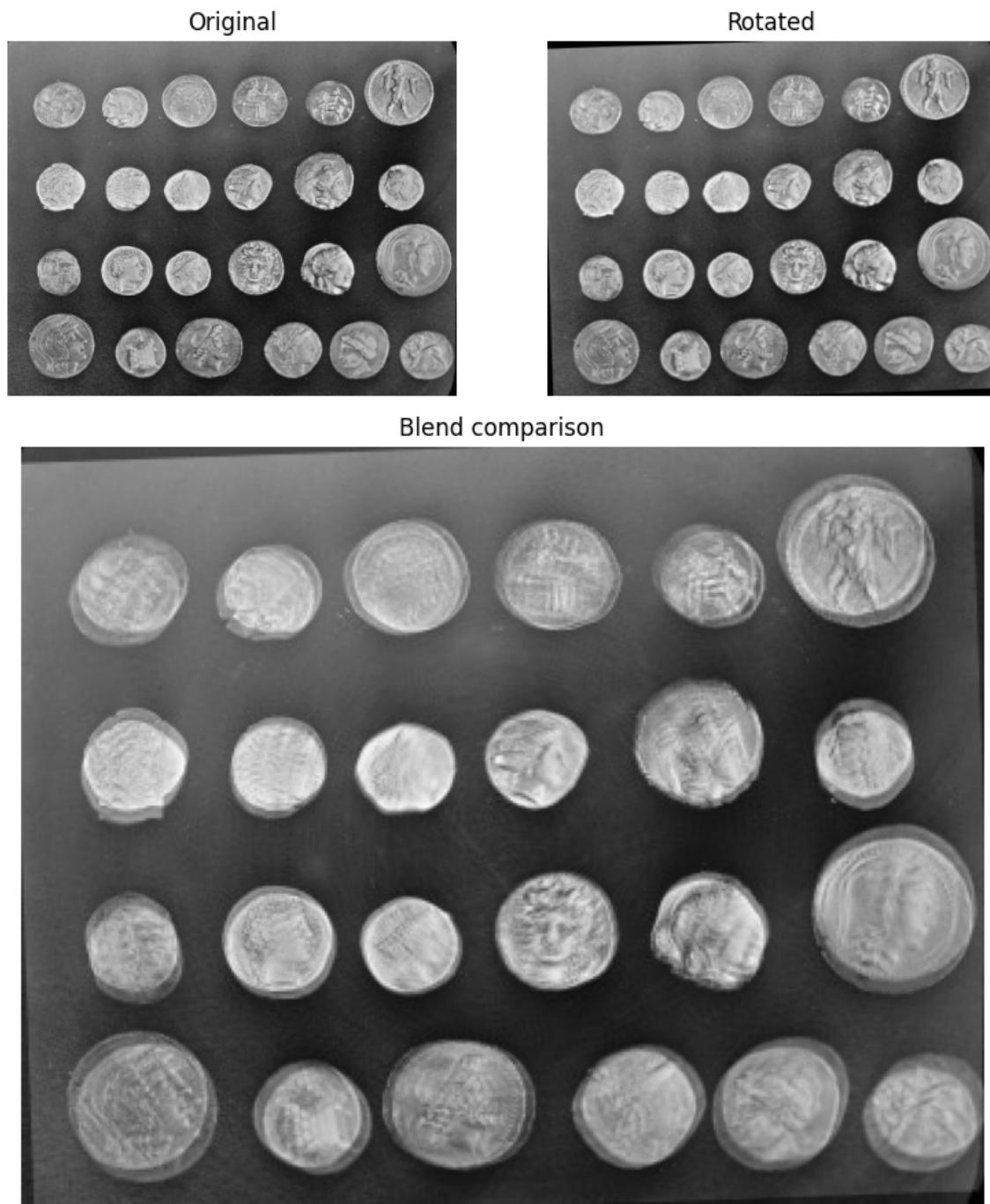
Blend

blend is the result of the average of the two images.

```
fig = plt.figure(figsize=(8, 9))

gs = GridSpec(3, 2)
ax0 = fig.add_subplot(gs[0, 0])
ax1 = fig.add_subplot(gs[0, 1])
ax2 = fig.add_subplot(gs[1:, :])

ax0.imshow(img1, cmap='gray')
ax0.set_title('Original')
ax1.imshow(img2, cmap='gray')
ax1.set_title('Rotated')
ax2.imshow(blend_rotated, cmap='gray')
ax2.set_title('Blend comparison')
for a in (ax0, ax1, ax2):
    a.axis('off')
plt.tight_layout()
plt.plot()
```



[]

Total running time of the script: (0 minutes 1.234 seconds)

Morphological Filtering

Morphological image processing is a collection of non-linear operations related to the shape or morphology of features in an image, such as boundaries, skeletons, etc. In any given technique, we probe an image with a small shape or template called a structuring element, which defines the region of interest or neighborhood around a pixel.

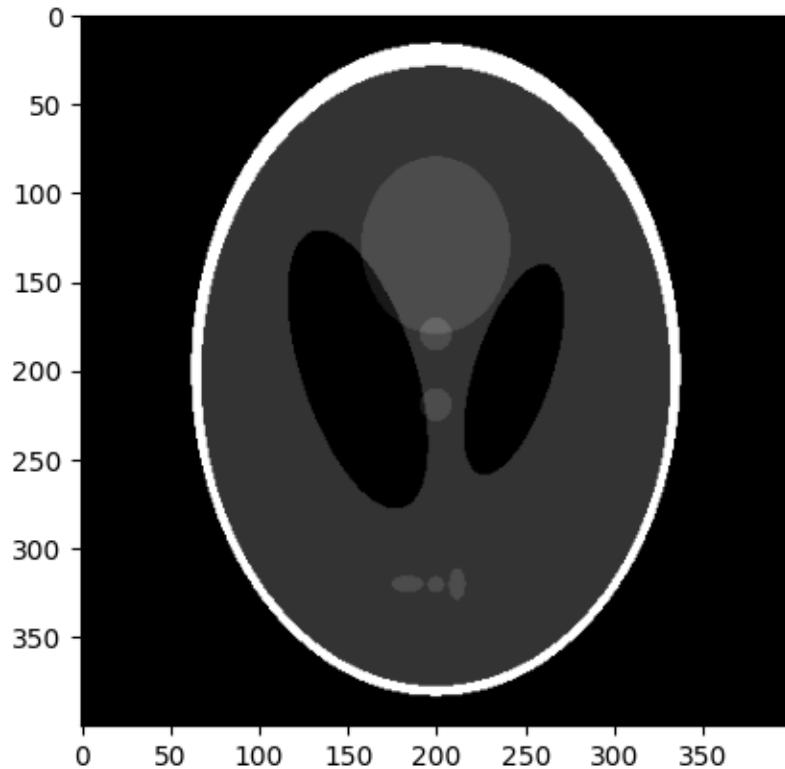
In this document we outline the following basic morphological operations:

1. Erosion
2. Dilation
3. Opening
4. Closing
5. White Tophat
6. Black Tophat
7. Skeletonize
8. Convex Hull

To get started, let's load an image using `io.imread`. Note that morphology functions only work on gray-scale or binary images, so we set `as_gray=True`.

```
import matplotlib.pyplot as plt
from skimage import data
from skimage.util import img_as_ubyte

orig_phantom = img_as_ubyte(data.shepp_logan_phantom())
fig, ax = plt.subplots()
ax.imshow(orig_phantom, cmap=plt.cm.gray)
```



```
<matplotlib.image.AxesImage object at 0x7ff4868a6760>
```

Let's also define a convenience function for plotting comparisons:

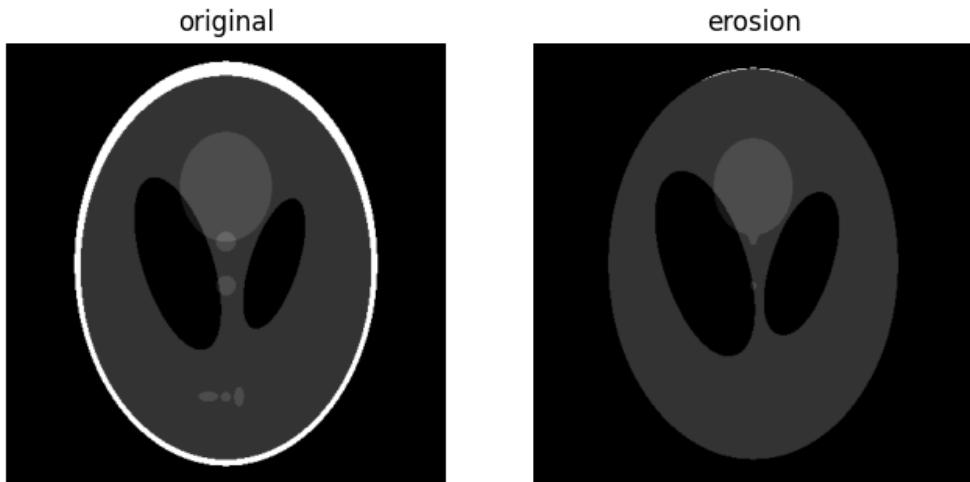
```
def plot_comparison(original, filtered, filter_name):  
    fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(8, 4), sharex=True,  
                                 sharey=True)  
    ax1.imshow(original, cmap=plt.cm.gray)  
    ax1.set_title('original')  
    ax1.axis('off')  
    ax2.imshow(filtered, cmap=plt.cm.gray)  
    ax2.set_title(filter_name)  
    ax2.axis('off')
```

Erosion

Morphological erosion sets a pixel at (i, j) to the *minimum over all pixels in the neighborhood centered at (i, j)* . The structuring element, `footprint`, passed to `erosion` is a boolean array that describes this neighborhood. Below, we use `disk` to create a circular structuring element, which we use for most of the following examples.

```
from skimage.morphology import erosion, dilation, opening, closing, # noqa
                               white_tophat)
from skimage.morphology import black_tophat, skeletonize, convex_hull_image # noqa
from skimage.morphology import disk # noqa

footprint = disk(6)
eroded = erosion(orig_phantom, footprint)
plot_comparison(orig_phantom, eroded, 'erosion')
```

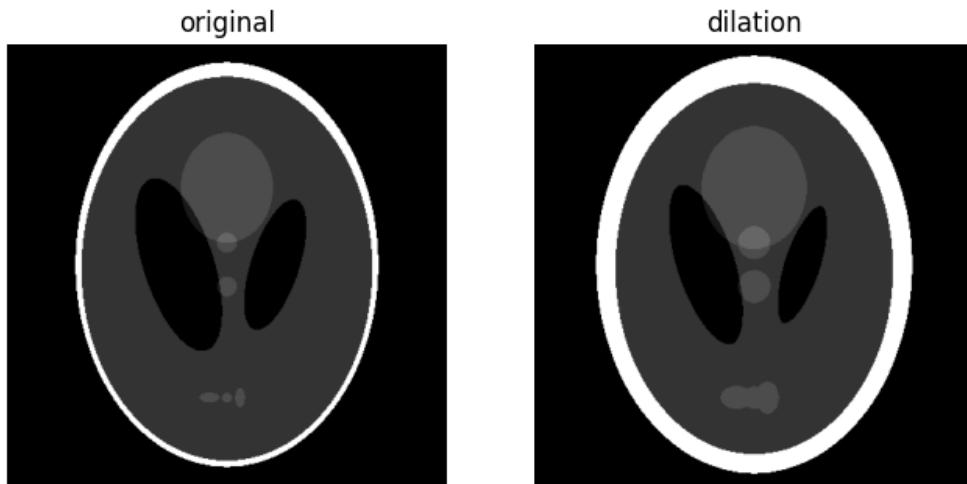


Notice how the white boundary of the image disappears or gets eroded as we increase the size of the disk. Also notice the increase in size of the two black ellipses in the center and the disappearance of the 3 light gray patches in the lower part of the image.

Dilation

Morphological dilation sets a pixel at (i, j) to the *maximum over all pixels in the neighborhood centered at (i, j)* . Dilation enlarges bright regions and shrinks dark regions.

```
dilated = dilation(orig_phantom, footprint)
plot_comparison(orig_phantom, dilated, 'dilation')
```

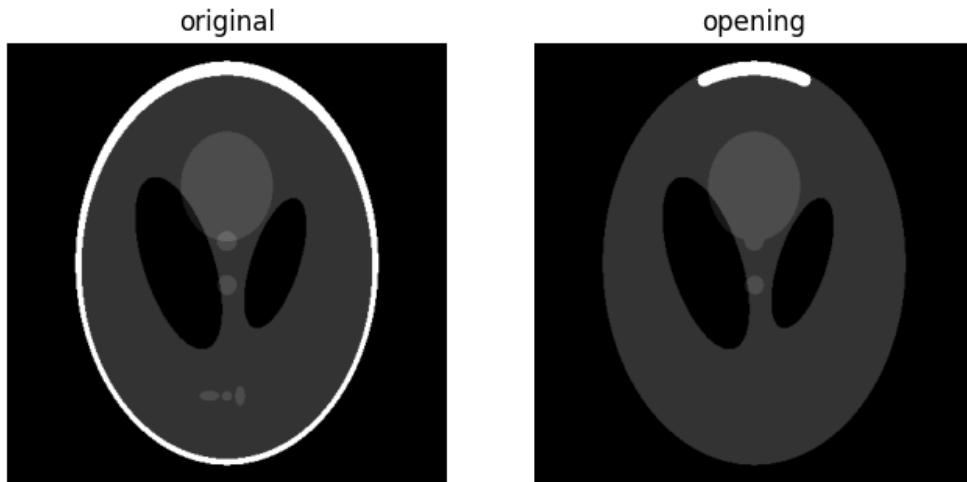


Notice how the white boundary of the image thickens, or gets dilated, as we increase the size of the disk. Also notice the decrease in size of the two black ellipses in the center, and the thickening of the light gray circle in the center and the 3 patches in the lower part of the image.

Opening

Morphological opening on an image is defined as an *erosion followed by a dilation*. Opening can remove small bright spots (i.e. “salt”) and connect small dark cracks.

```
opened = opening(orig_phantom, footprint)
plot_comparison(orig_phantom, opened, 'opening')
```



Since opening an image starts with an erosion operation, light regions that are *smaller* than the structuring element are removed. The dilation operation that follows ensures that light regions that are *larger* than the structuring element retain their original size. Notice how the light and dark shapes in the center their original thickness but the 3 lighter patches in the bottom get completely eroded. The size dependence is highlighted by the outer white ring: The parts of the ring thinner than the structuring element were completely erased, while the thicker region at the top retains its original thickness.

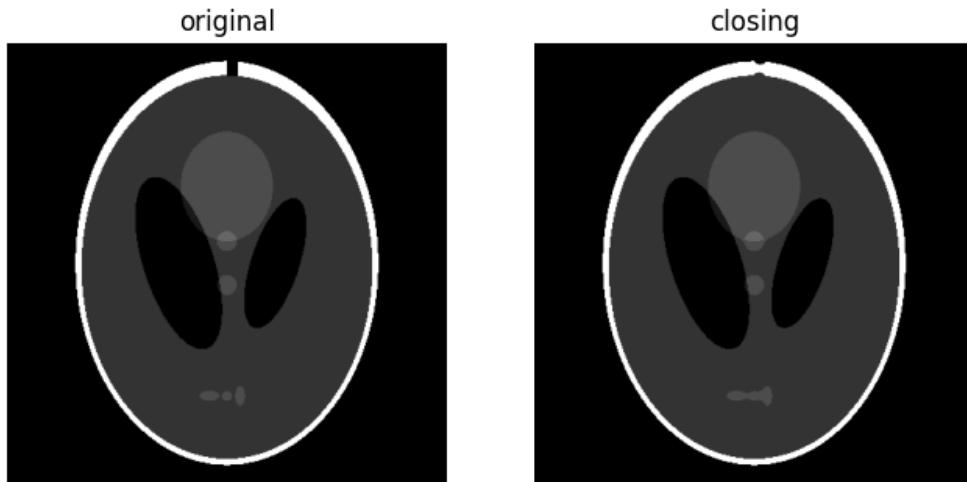
Closing

Morphological closing on an image is defined as a *dilation followed by an erosion*. Closing can remove small dark spots (i.e. “pepper”) and connect small bright cracks.

To illustrate this more clearly, let’s add a small crack to the white border:

```
phantom = orig_phantom.copy()
phantom[10:30, 200:210] = 0

closed = closing(phantom, footprint)
plot_comparison(phantom, closed, 'closing')
```



Since `closing` an image starts with a dilation operation, dark regions that are *smaller* than the structuring element are removed. The dilation operation that follows ensures that dark regions that are *larger* than the structuring element retain their original size. Notice how the white ellipses at the bottom get connected because of dilation, but other dark region retain their original sizes. Also notice how the crack we added is mostly removed.

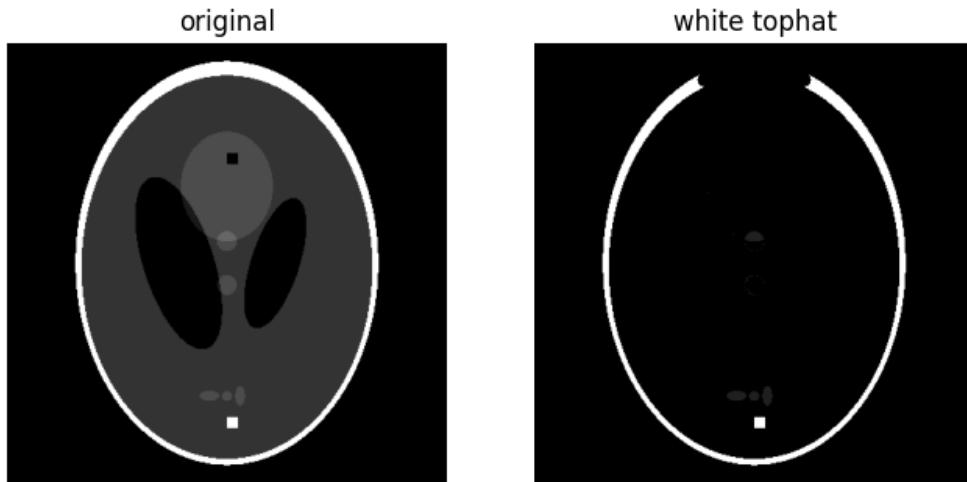
White tophat

The `white_tophat` of an image is defined as the *image minus its morphological opening*. This operation returns the bright spots of the image that are smaller than the structuring element.

To make things interesting, we'll add bright and dark spots to the image:

```
phantom = orig_phantom.copy()
phantom[340:350, 200:210] = 255
phantom[100:110, 200:210] = 0

w_tophat = white_tophat(phantom, footprint)
plot_comparison(phantom, w_tophat, 'white tophat')
```

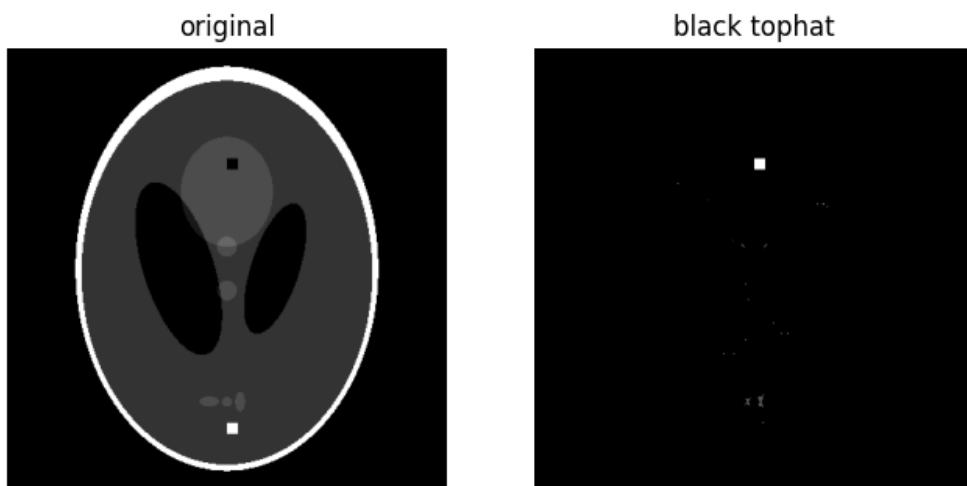


As you can see, the 10-pixel wide white square is highlighted since it is smaller than the structuring element. Also, the thin, white edges around most of the ellipse are retained because they're smaller than the structuring element, but the thicker region at the top disappears.

Black tophat

The `black_tophat` of an image is defined as its morphological **closing minus the original image**. This operation returns the *dark spots of the image that are smaller than the structuring element*.

```
b_tophat = black_tophat(phantom, footprint)
plot_comparison(phantom, b_tophat, 'black tophat')
```



As you can see, the 10-pixel wide black square is highlighted since it is smaller than the structuring element.

Duality

As you should have noticed, many of these operations are simply the reverse of another operation. This duality can be summarized as follows:

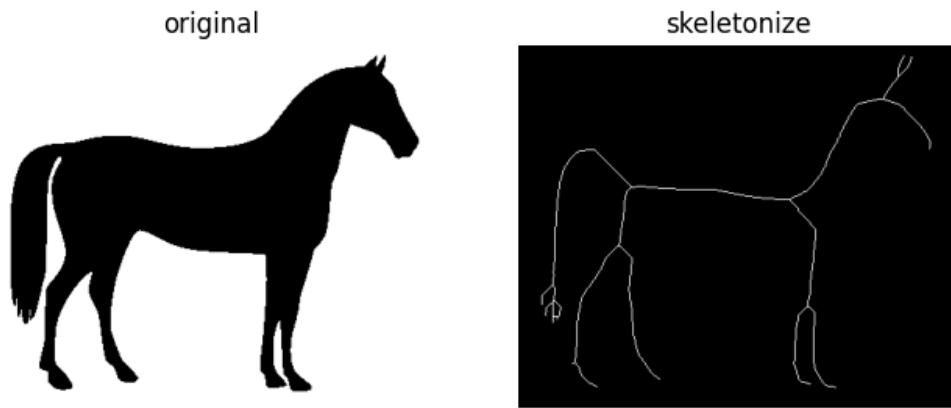
1. Erosion <-> Dilation
2. Opening <-> Closing
3. White tophat <-> Black tophat

Skeletonize

Thinning is used to reduce each connected component in a binary image to a *single-pixel wide skeleton*. It is important to note that this is performed on binary images only.

```
horse = data.horse()

sk = skeletonize(horse == 0)
plot_comparison(horse, sk, 'skeletonize')
```

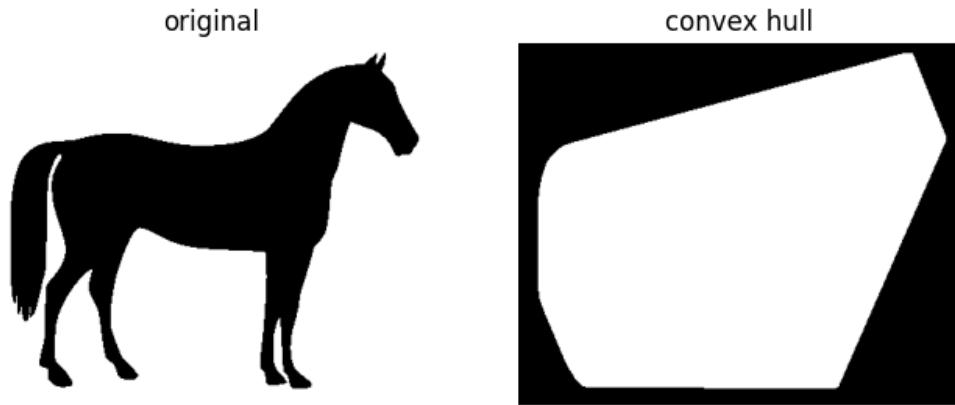


As the name suggests, this technique is used to thin the image to 1-pixel wide skeleton by applying thinning successively.

Convex hull

The `convex_hull_image` is the *set of pixels included in the smallest convex polygon that surround all white pixels in the input image*. Again note that this is also performed on binary images.

```
hull1 = convex_hull_image(horse == 0)
plot_comparison(horse, hull1, 'convex hull')
```

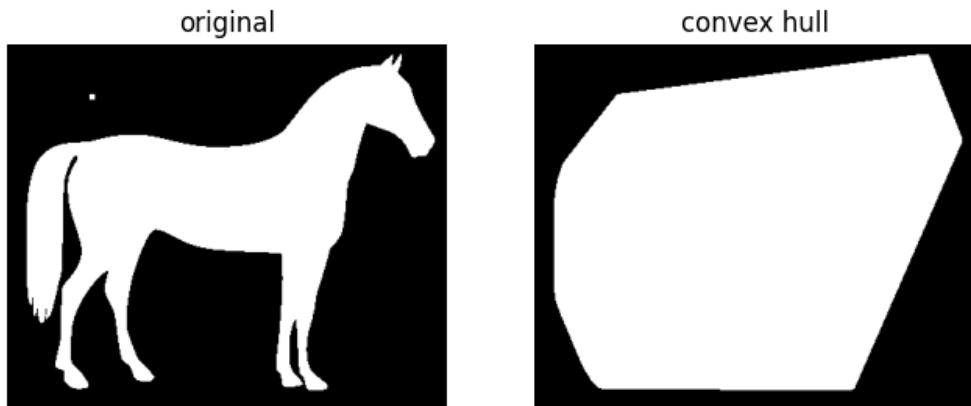


As the figure illustrates, `convex_hull_image` gives the smallest polygon which covers the white or True completely in the image.

If we add a small grain to the image, we can see how the convex hull adapts to enclose that grain:

```
horse_mask = horse == 0
horse_mask[45:50, 75:80] = 1

hull2 = convex_hull_image(horse_mask)
plot_comparison(horse_mask, hull2, 'convex hull')
```



Additional Resources

1. MathWorks tutorial on morphological processing
2. Auckland university's tutorial on Morphological Image Processing
3. https://en.wikipedia.org/wiki/Mathematical_morphology

```
plt.show()
```

Total running time of the script: (0 minutes 1.286 seconds)

Estimate anisotropy in a 3D microscopy image

In this tutorial, we compute the structure tensor of a 3D image. For a general introduction to 3D image processing, please refer to *Explore 3D images (of cells)*. The data we use here are sampled from an image of kidney tissue obtained by confocal fluorescence microscopy (more details at¹ under `kidney-tissue-fluorescence.tif`).

```
import matplotlib.pyplot as plt
import numpy as np
import plotly.express as px
import plotly.io

from skimage import (
    data, feature
)
```

Load image

This biomedical image is available through *scikit-image*'s data registry.

```
data = data.kidney()
```

What exactly are the shape and size of our 3D multichannel image?

```
print(f'number of dimensions: {data.ndim}')
print(f'shape: {data.shape}')
print(f'dtype: {data.dtype}')
```

```
number of dimensions: 4
shape: (16, 512, 512, 3)
dtype: uint16
```

For the purposes of this tutorial, we shall consider only the second color channel, which leaves us with a 3D single-channel image. What is the range of values?

```
n_plane, n_row, n_col, n_chan = data.shape
v_min, v_max = data[:, :, :, 1].min(), data[:, :, :, 1].max()
print(f'range: ({v_min}, {v_max})')
```

```
range: (68, 4095)
```

¹ <https://gitlab.com/scikit-image/data/#data>

Let us visualize the middle slice of our 3D image.

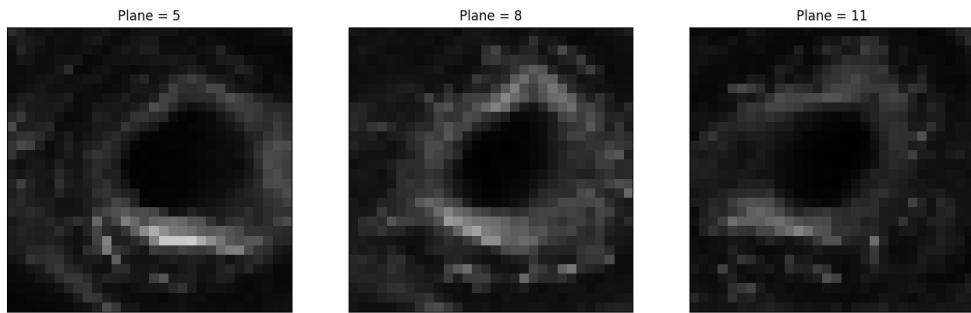
```
fig1 = px.imshow(
    data[n_plane // 2, :, :, 1],
    zmin=v_min,
    zmax=v_max,
    labels={'x': 'col', 'y': 'row', 'color': 'intensity'}
)

plotly.io.show(fig1)
```

Let us pick a specific region, which shows relative X-Y isotropy. In contrast, the gradient is quite different (and, for that matter, weak) along Z.

```
sample = data[5:13, 380:410, 370:400, 1]
step = 3
cols = sample.shape[0] // step + 1
_, axes = plt.subplots(nrows=1, ncols=cols, figsize=(16, 8))

for it, (ax, image) in enumerate(zip(axes.flatten(), sample[:, ::step])):
    ax.imshow(image, cmap='gray', vmin=v_min, vmax=v_max)
    ax.set_title(f'Plane = {5 + it * step}')
    ax.set_xticks([])
    ax.set_yticks([])
```



To view the sample data in 3D, run the following code:

```
import plotly.graph_objects as go

(n_Z, n_Y, n_X) = sample.shape
Z, Y, X = np.mgrid[:n_Z, :n_Y, :n_X]

fig = go.Figure(
```

(continues on next page)

(continued from previous page)

```

data=go.Volume(
    x=X.flatten(),
    y=Y.flatten(),
    z=Z.flatten(),
    value=sample.flatten(),
    opacity=0.5,
    slices_z=dict(show=True, locations=[4])
)
fig.show()

```

Compute structure tensor

Let us visualize the bottom slice of our sample data and determine the typical size for strong variations. We shall use this size as the ‘width’ of the window function.

```

fig2 = px.imshow(
    sample[0, :, :],
    zmin=v_min,
    zmax=v_max,
    labels={'x': 'col', 'y': 'row', 'color': 'intensity'},
    title='Interactive view of bottom slice of sample data.'
)

plotly.io.show(fig2)

```

About the brightest region (i.e., at row ~ 22 and column ~ 17), we can see variations (and, hence, strong gradients) over 2 or 3 (resp. 1 or 2) pixels across columns (resp. rows). We may thus choose, say, `sigma = 1.5` for the window function. Alternatively, we can pass sigma on a per-axis basis, e.g., `sigma = (1, 2, 3)`. Note that size 1 sounds reasonable along the first (Z, plane) axis, since the latter is of size 8 (13 - 5). Viewing slices in the X-Z or Y-Z planes confirms it is reasonable.

```

sigma = (1, 1.5, 2.5)
A_elems = feature.structure_tensor(sample, sigma=sigma)

```

We can then compute the eigenvalues of the structure tensor.

```

eigen = feature.structure_tensor_eigenvalues(A_elems)
eigen.shape

```

```
(3, 8, 30, 30)
```

Where is the largest eigenvalue?

```

coords = np.unravel_index(eigen.argmax(), eigen.shape)
assert coords[0] == 0 # by definition
coords

```

```
(0, 1, 22, 16)
```

Note: The reader may check how robust this result (coordinates (plane, row, column) = coords[1:]) is to varying sigma.

Let us view the spatial distribution of the eigenvalues in the X-Y plane where the maximum eigenvalue is found (i.e., Z = coords[1]).

```
fig3 = px.imshow(
    eigen[:, coords[1], :, :],
    facet_col=0,
    labels={'x': 'col', 'y': 'row', 'facet_col': 'rank'},
    title=f'Eigenvalues for plane Z = {coords[1]}.')
plotly.io.show(fig3)
```

We are looking at a local property. Let us consider a tiny neighborhood around the maximum eigenvalue in the above X-Y plane.

```
eigen[0, coords[1], coords[2] - 2:coords[2] + 1, coords[3] - 2:coords[3] + 1]
```

```
array([[0.05530323, 0.05929082, 0.06043806],
       [0.05922725, 0.06268274, 0.06354238],
       [0.06190861, 0.06685075, 0.06840962]])
```

If we examine the second-largest eigenvalues in this neighborhood, we can see that they have the same order of magnitude as the largest ones.

```
eigen[1, coords[1], coords[2] - 2:coords[2] + 1, coords[3] - 2:coords[3] + 1]
```

```
array([[0.03577746, 0.03577334, 0.03447714],
       [0.03819524, 0.04172036, 0.04323701],
       [0.03139592, 0.03587025, 0.03913327]])
```

In contrast, the third-largest eigenvalues are one order of magnitude smaller.

```
eigen[2, coords[1], coords[2] - 2:coords[2] + 1, coords[3] - 2:coords[3] + 1]
```

```
array([[0.00337661, 0.00306529, 0.00276288],
       [0.0041869, 0.00397519, 0.00375595],
       [0.00479742, 0.00462116, 0.00442455]])
```

Let us visualize the slice of sample data in the X-Y plane where the maximum eigenvalue is found.

```
fig4 = px.imshow(
    sample[coords[1], :, :],
    zmin=v_min,
    zmax=v_max,
    labels={'x': 'col', 'y': 'row', 'color': 'intensity'},
    title=f'Interactive view of plane Z = {coords[1]}.')
plotly.io.show(fig4)
```

Let us visualize the slices of sample data in the X-Z (left) and Y-Z (right) planes where the maximum eigenvalue is found. The Z axis is the vertical axis in the subplots below. We can see the expected relative invariance along the Z axis (corresponding to longitudinal structures in the kidney tissue), especially in the Y-Z plane (longitudinal=1).

```
subplots = np.dstack((sample[:, coords[2], :], sample[:, :, coords[3]]))
fig5 = px.imshow(
    subplots,
    zmin=v_min,
    zmax=v_max,
    facet_col=2,
    labels={'facet_col': 'longitudinal'}
)
plotly.io.show(fig5)
```

As a conclusion, the region about voxel (plane, row, column) = coords[1:] is anisotropic in 3D: There is an order of magnitude between the third-largest eigenvalues on one hand, and the largest and second-largest eigenvalues on the other hand. We could see this at first glance in figure *Eigenvalues for plane Z = 1*.

The neighborhood in question is ‘somewhat isotropic’ in a plane (which, here, would be relatively close to the X-Y plane): There is a factor of less than 2 between the second-largest and largest eigenvalues. This description is compatible with what we are seeing in the image, i.e., a stronger gradient across a direction (which, here, would be relatively close to the row axis) and a weaker gradient perpendicular to it.

In an ellipsoidal representation of the 3D structure tensor², we would get the pancake situation.

Total running time of the script: (0 minutes 1.550 seconds)

Comparing edge-based and region-based segmentation

In this example, we will see how to segment objects from a background. We use the coins image from skimage.data, which shows several coins outlined against a darker background.

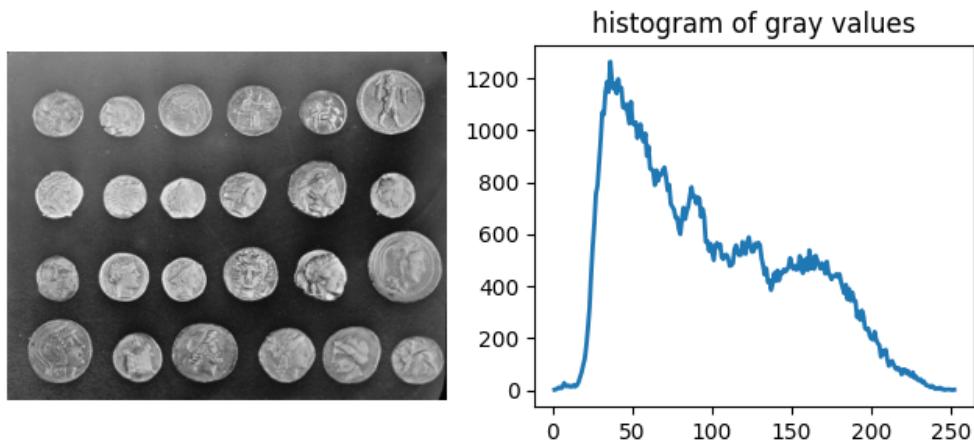
```
import numpy as np
import matplotlib.pyplot as plt

from skimage import data
from skimage.exposure import histogram

coins = data.coins()
hist, hist_centers = histogram(coins)

fig, axes = plt.subplots(1, 2, figsize=(8, 3))
axes[0].imshow(coins, cmap=plt.cm.gray)
axes[0].axis('off')
axes[1].plot(hist_centers, hist, lw=2)
axes[1].set_title('histogram of gray values')
```

² https://en.wikipedia.org/wiki/Structure_tensor#Interpretation_2



```
Text(0.5, 1.0, 'histogram of gray values')
```

Thresholding

A simple way to segment the coins is to choose a threshold based on the histogram of gray values. Unfortunately, thresholding this image gives a binary image that either misses significant parts of the coins or merges parts of the background with the coins:

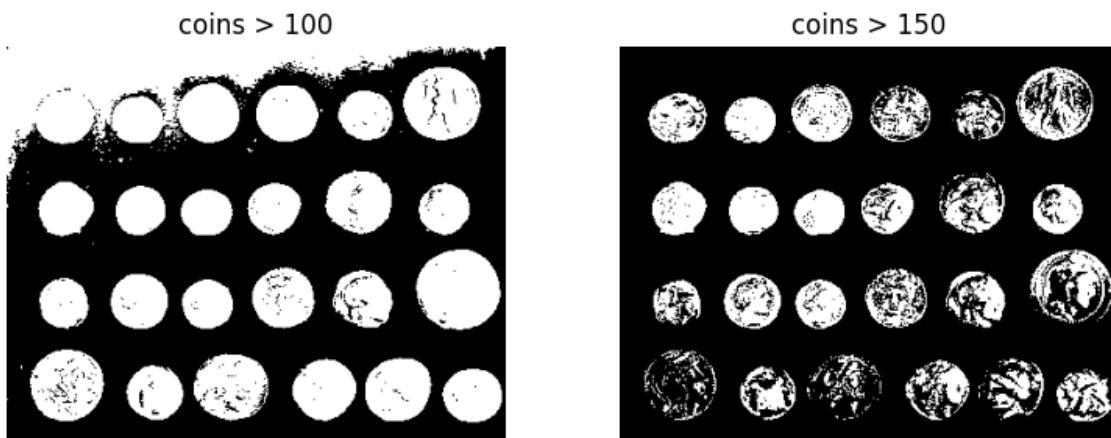
```
fig, axes = plt.subplots(1, 2, figsize=(8, 3), sharey=True)

axes[0].imshow(coins > 100, cmap=plt.cm.gray)
axes[0].set_title('coins > 100')

axes[1].imshow(coins > 150, cmap=plt.cm.gray)
axes[1].set_title('coins > 150')

for a in axes:
    a.axis('off')

plt.tight_layout()
```



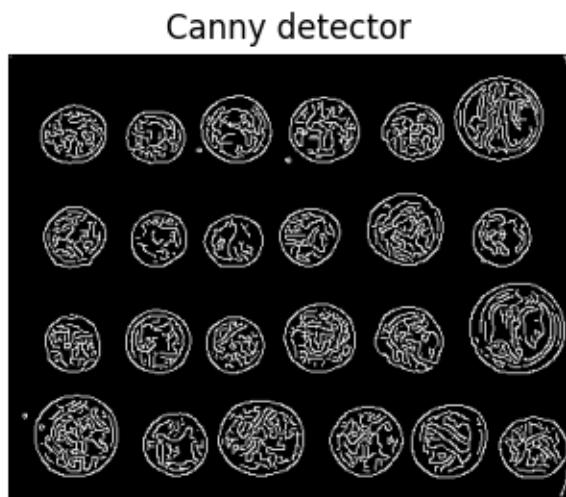
Edge-based segmentation

Next, we try to delineate the contours of the coins using edge-based segmentation. To do this, we first get the edges of features using the Canny edge-detector.

```
from skimage.feature import canny

edges = canny(coins)

fig, ax = plt.subplots(figsize=(4, 3))
ax.imshow(edges, cmap=plt.cm.gray)
ax.set_title('Canny detector')
ax.axis('off')
```



```
(-0.5, 383.5, 302.5, -0.5)
```

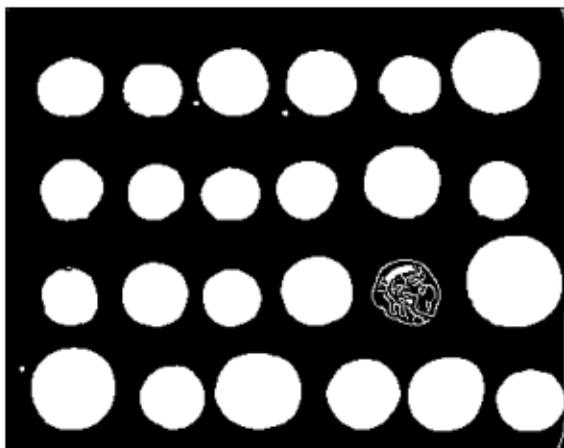
These contours are then filled using mathematical morphology.

```
from scipy import ndimage as ndi

fill_coins = ndi.binary_fill_holes(edges)

fig, ax = plt.subplots(figsize=(4, 3))
ax.imshow(fill_coins, cmap=plt.cm.gray)
ax.set_title('filling the holes')
ax.axis('off')
```

filling the holes



(-0.5, 383.5, 302.5, -0.5)

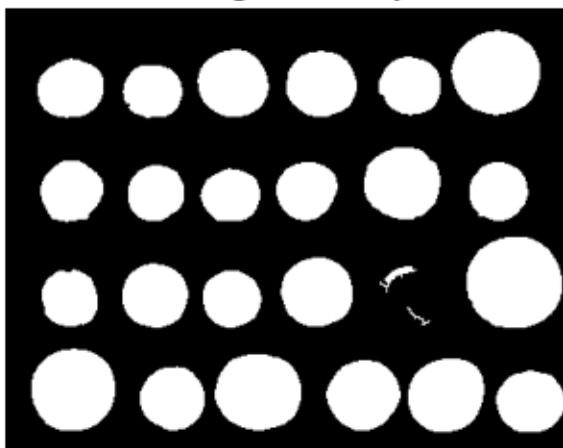
Small spurious objects are easily removed by setting a minimum size for valid objects.

```
from skimage import morphology

coins_cleaned = morphology.remove_small_objects(fill_coins, 21)

fig, ax = plt.subplots(figsize=(4, 3))
ax.imshow(coins_cleaned, cmap=plt.cm.gray)
ax.set_title('removing small objects')
ax.axis('off')
```

removing small objects



(-0.5, 383.5, 302.5, -0.5)

However, this method is not very robust, since contours that are not perfectly closed are not filled correctly, as is the

case for one unfilled coin above.

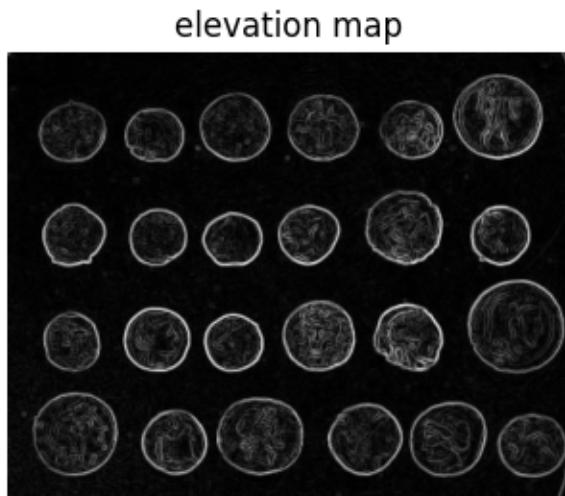
Region-based segmentation

We therefore try a region-based method using the watershed transform. First, we find an elevation map using the Sobel gradient of the image.

```
from skimage.filters import sobel

elevation_map = sobel(coins)

fig, ax = plt.subplots(figsize=(4, 3))
ax.imshow(elevation_map, cmap=plt.cm.gray)
ax.set_title('elevation map')
ax.axis('off')
```

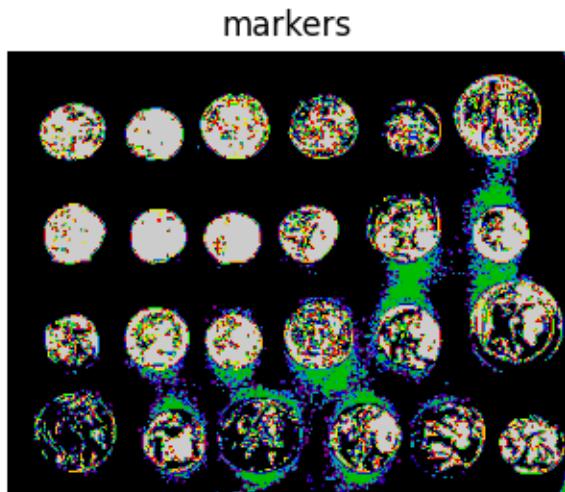


(-0.5, 383.5, 302.5, -0.5)

Next we find markers of the background and the coins based on the extreme parts of the histogram of gray values.

```
markers = np.zeros_like(coins)
markers[coins < 30] = 1
markers[coins > 150] = 2

fig, ax = plt.subplots(figsize=(4, 3))
ax.imshow(markers, cmap=plt.cm.nipy_spectral)
ax.set_title('markers')
ax.axis('off')
```



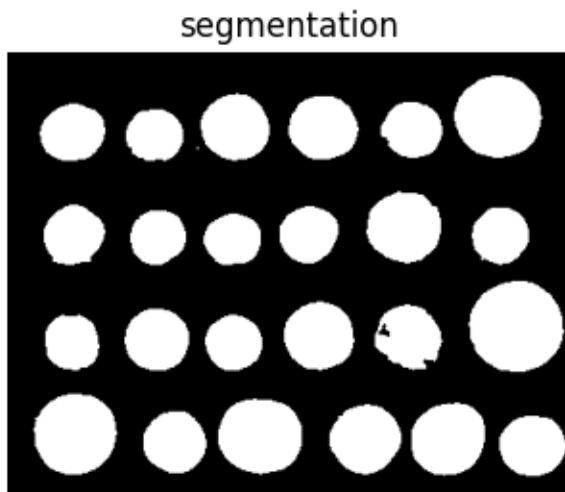
```
(-0.5, 383.5, 302.5, -0.5)
```

Finally, we use the watershed transform to fill regions of the elevation map starting from the markers determined above:

```
from skimage import segmentation

segmentation_coins = segmentation.watershed(elevation_map, markers)

fig, ax = plt.subplots(figsize=(4, 3))
ax.imshow(segmentation_coins, cmap=plt.cm.gray)
ax.set_title('segmentation')
ax.axis('off')
```



```
(-0.5, 383.5, 302.5, -0.5)
```

This last method works even better, and the coins can be segmented and labeled individually.

```

from skimage.color import label2rgb

segmentation_coins = ndi.binary_fill_holes(segmentation_coins - 1)
labeled_coins, _ = ndi.label(segmentation_coins)
image_label_overlay = label2rgb(labeled_coins, image=coins, bg_label=0)

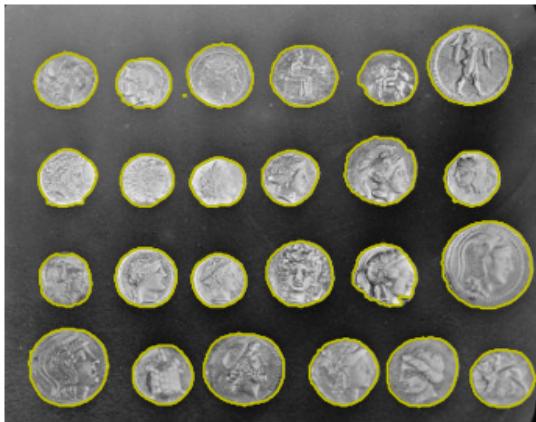
fig, axes = plt.subplots(1, 2, figsize=(8, 3), sharey=True)
axes[0].imshow(coins, cmap=plt.cm.gray)
axes[0].contour(segmentation_coins, [0.5], linewidths=1.2, colors='y')
axes[1].imshow(image_label_overlay)

for a in axes:
    a.axis('off')

plt.tight_layout()

plt.show()

```



Total running time of the script: (0 minutes 0.784 seconds)

Colocalization metrics

In this example, we demonstrate the use of different metrics to assess the colocalization of two different image channels.

Colocalization can be split into two different concepts: 1. Co-occurrence: What proportion of a substance is localized to a particular area? 2. Correlation: What is the relationship in intensity between two substances?

Co-occurrence: subcellular localization

Imagine that we are trying to determine the subcellular localization of a protein - is it located more in the nucleus or cytoplasm compared to a control?

We begin by segmenting the nucleus of a sample image as described in another [example](#) and assume that whatever is not in the nucleus is in the cytoplasm. The protein, “protein A”, will be simulated as blobs and segmented.

```

import matplotlib.pyplot as plt
import numpy as np
from matplotlib.colors import LinearSegmentedColormap

```

(continues on next page)

(continued from previous page)

```
from scipy import ndimage as ndi
from skimage import data, filters, measure, segmentation

rng = np.random.default_rng()

# segment nucleus
nucleus = data.protein_transport()[0, 0, :, :180]
smooth = filters.gaussian(nucleus, sigma=1.5)
thresh = smooth > filters.threshold_otsu(smooth)
fill = ndi.binary_fill_holes(thresh)
nucleus_seg = segmentation.clear_border(fill)

# protein blobs of varying intensity
proteinA = np.zeros_like(nucleus, dtype="float64")
proteinA_seg = np.zeros_like(nucleus, dtype="float64")

for blob_seed in range(10):
    blobs = data.binary_blobs(180,
                              blob_size_fraction=0.5,
                              volume_fraction=(50/(180**2)),
                              rng=blob_seed)
    blobs_image = filters.gaussian(blobs, sigma=1.5) * rng.integers(50, 256)
    proteinA += blobs_image
    proteinA_seg += blobs

# plot data
fig, ax = plt.subplots(3, 2, figsize=(8, 12), sharey=True)
ax[0, 0].imshow(nucleus, cmap=plt.cm.gray)
ax[0, 0].set_title('Nucleus')

ax[0, 1].imshow(nucleus_seg, cmap=plt.cm.gray)
ax[0, 1].set_title('Nucleus segmentation')

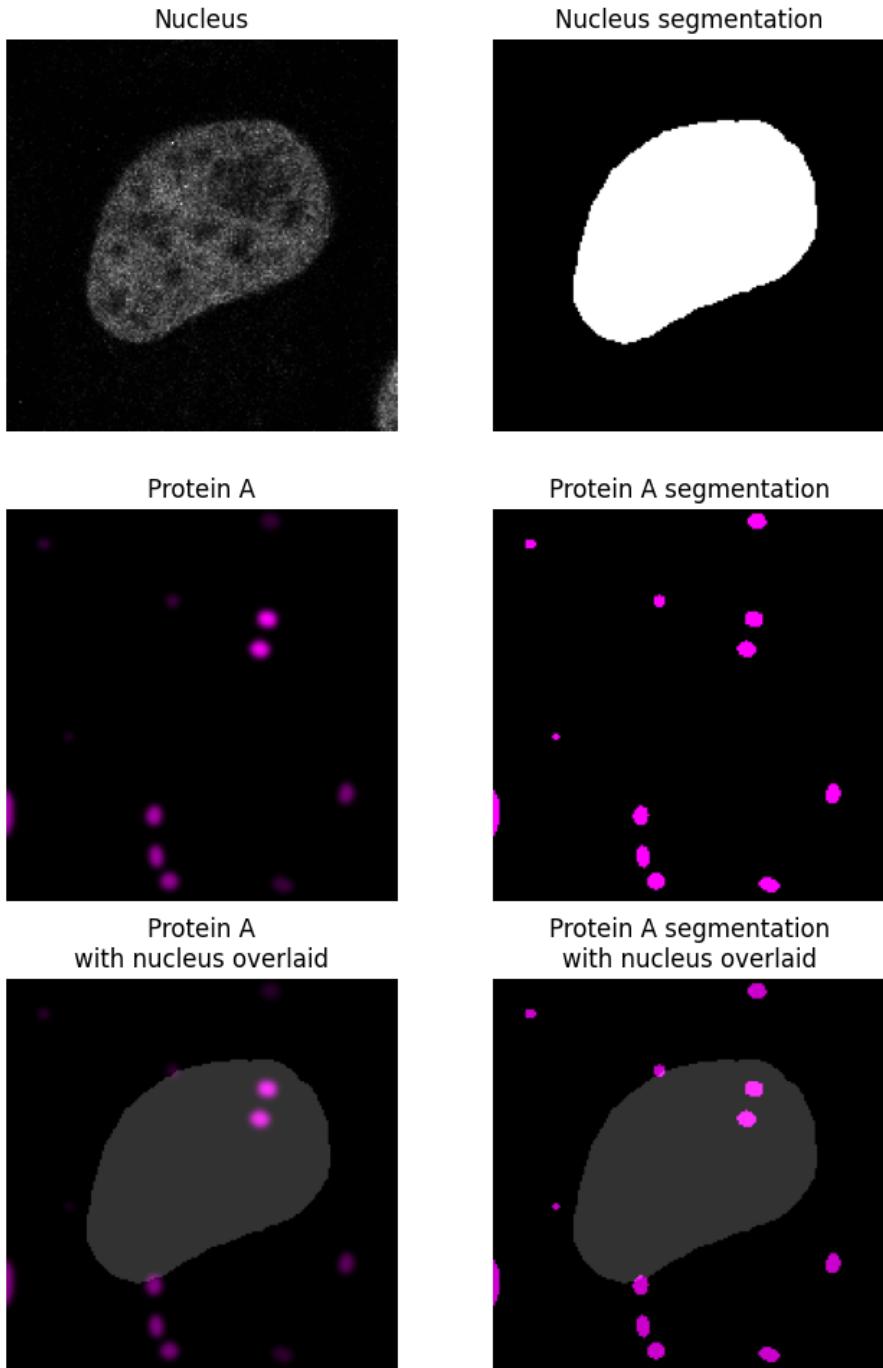
black_magenta = LinearSegmentedColormap.from_list("", ["black", "magenta"])
ax[1, 0].imshow(proteinA, cmap=black_magenta)
ax[1, 0].set_title('Protein A')

ax[1, 1].imshow(proteinA_seg, cmap=black_magenta)
ax[1, 1].set_title('Protein A segmentation')

ax[2, 0].imshow(proteinA, cmap=black_magenta)
ax[2, 0].imshow(nucleus_seg, cmap=plt.cm.gray, alpha=0.2)
ax[2, 0].set_title('Protein A\nwith nucleus overlaid')

ax[2, 1].imshow(proteinA_seg, cmap=black_magenta)
ax[2, 1].imshow(nucleus_seg, cmap=plt.cm.gray, alpha=0.2)
ax[2, 1].set_title('Protein A segmentation\nwith nucleus overlaid')

for a in ax.ravel():
    a.set_axis_off()
```



Intersection coefficient

After segmenting both the nucleus and the protein of interest, we can determine what fraction of the protein A segmentation overlaps with the nucleus segmentation.

```
measure.intersection_coeff(proteinA_seg, nucleus_seg)
```

```
0.22
```

Manders' Colocalization Coefficient (MCC)

The overlap coefficient assumes that the area of protein segmentation corresponds to the concentration of that protein - with larger areas indicating more protein. As the resolution of images are usually too small to make out individual proteins, they can clump together within one pixel, making the intensity of that pixel brighter. So, to better capture the protein concentration, we may choose to determine what proportion of the *intensity* of the protein channel is inside the nucleus. This metric is known as Manders' Colocalization Coefficient.

In this image, while there are a lot of protein A spots within the nucleus they are dim compared to some of the spots outside the nucleus, so the MCC is much lower than the overlap coefficient.

```
measure.manders_coloc_coeff(proteinA, nucleus_seg)
```

```
0.36770193540732216
```

After choosing a co-occurrence metric, we can apply the same process to control images. If no control images are available, the Costes method could be used to compare the MCC value of the original image with that of the randomly scrambled image. Information about this method is given in¹.

Correlation: association of two proteins

Now, imagine that we want to know how closely related two proteins are.

First, we will generate protein B and plot intensities of the two proteins in every pixel to see the relationship between them.

```
# generating protein B data that is correlated to protein A for demo
proteinB = proteinA + rng.normal(loc=100, scale=10, size=proteinA.shape)

# plot images
fig, ax = plt.subplots(1, 2, figsize=(8, 8), sharey=True)

ax[0].imshow(proteinA, cmap=black_magenta)
ax[0].set_title('Protein A')

black_cyan = LinearSegmentedColormap.from_list("", ["black", "cyan"])
ax[1].imshow(proteinB, cmap=black_cyan)
ax[1].set_title('Protein B')

for a in ax.ravel():
```

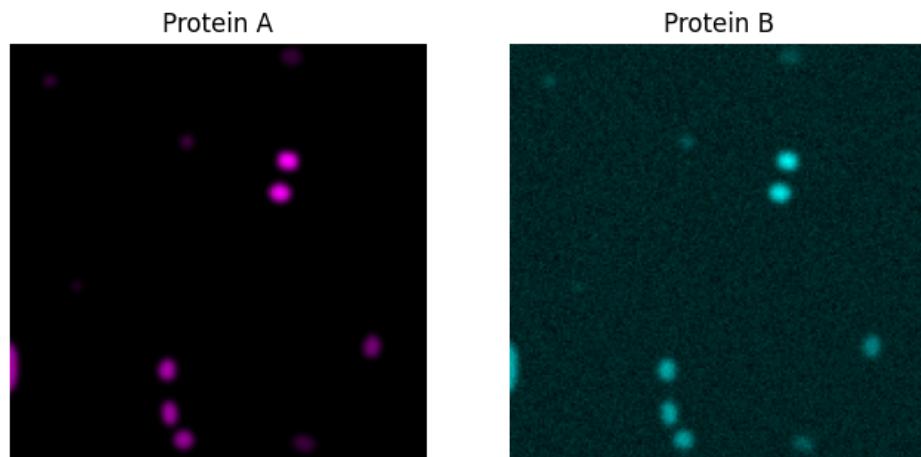
(continues on next page)

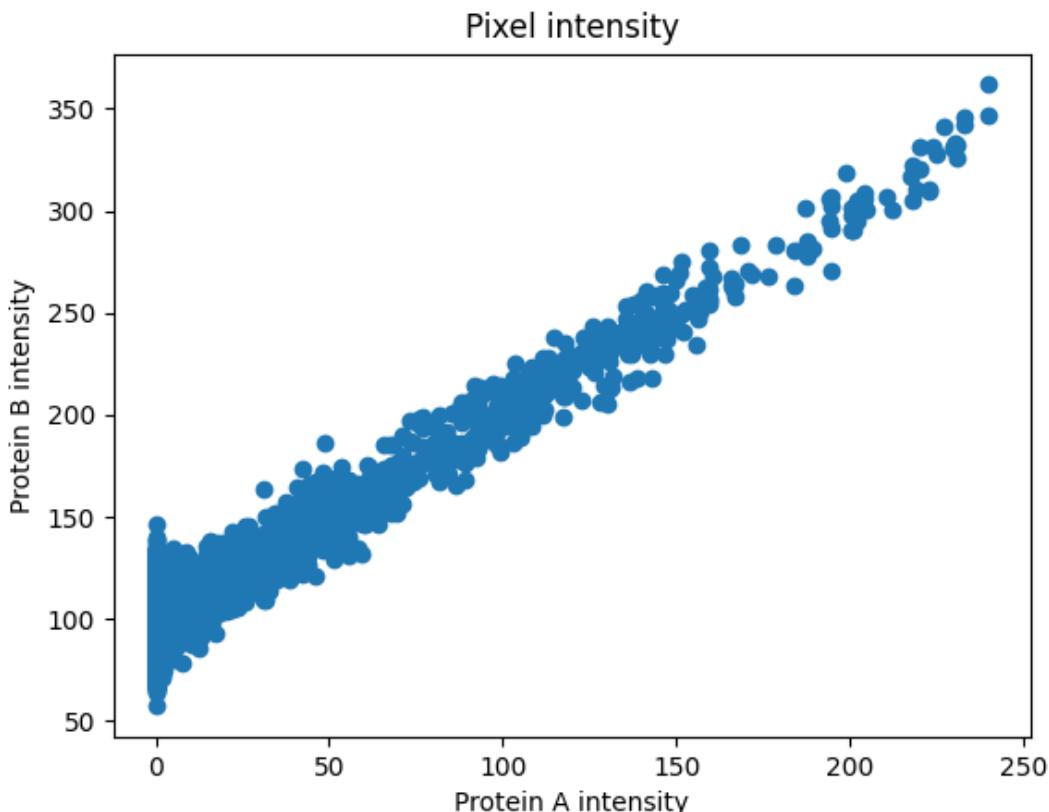
¹ J. S. Aaron, A. B. Taylor and T.-L. Chew, Image co-localization – co-occurrence versus correlation. J Cell Sci 1 February 2018 131 (3): jcs211847. doi: <https://doi.org/10.1242/jcs.211847>

(continued from previous page)

```
a.set_axis_off()

# plot pixel intensity scatter
plt.figure()
plt.scatter(proteinA, proteinB)
plt.title('Pixel intensity')
plt.xlabel('Protein A intensity')
plt.ylabel('Protein B intensity')
```





```
Text(38.347222222222214, 0.5, 'Protein B intensity')
```

The intensities look linearly correlated so Pearson's Correlation Coefficient would give us a good measure of how strong the association is.

```
pcc, pval = measure.pearson_corr_coeff(proteinA, proteinB)
print(f"PCC: {pcc:.3g}, p-val: {pval:.3g}")
```

```
PCC: 0.829, p-val: 0
```

Sometimes the intensities are correlated but not in a linear way. A rank-based correlation coefficient like Spearman's might give a more accurate measure of the non-linear relationship in that case.

```
plt.show()
```

Total running time of the script: (0 minutes 1.115 seconds)

Segment human cells (in mitosis)

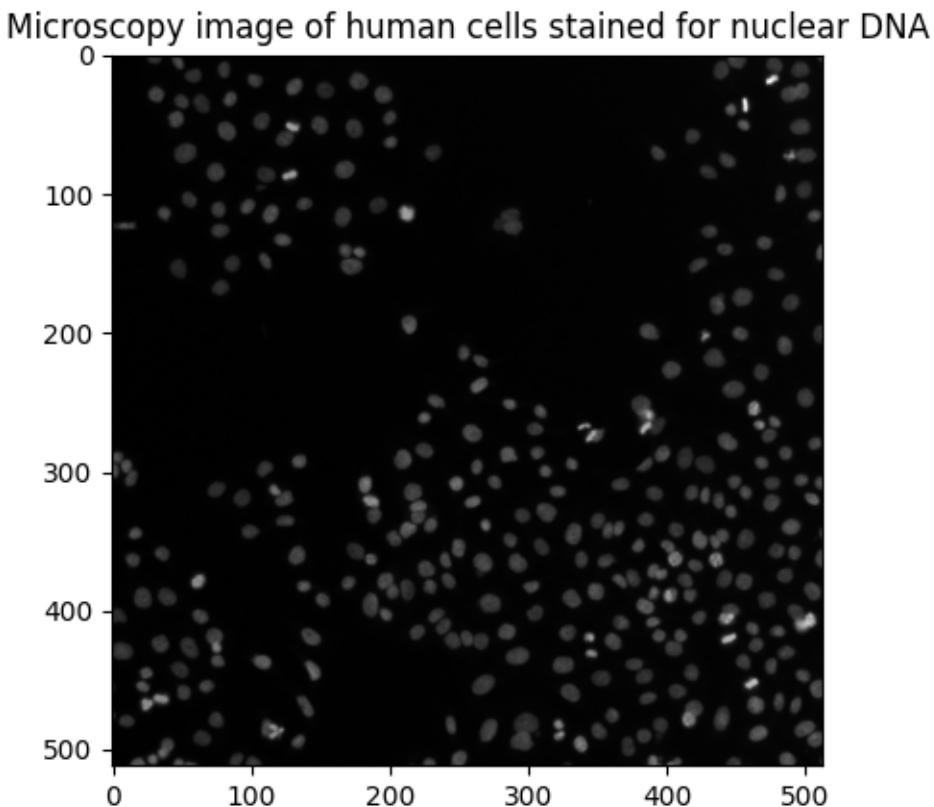
In this example, we analyze a microscopy image of human cells. We use data provided by Jason Moffat¹ through CellProfiler.

```
import matplotlib.pyplot as plt
import numpy as np
from scipy import ndimage as ndi

from skimage import (
    color, feature, filters, measure, morphology, segmentation, util
)
from skimage.data import human_mitosis

image = human_mitosis()

fig, ax = plt.subplots()
ax.imshow(image, cmap='gray')
ax.set_title('Microscopy image of human cells stained for nuclear DNA')
plt.show()
```



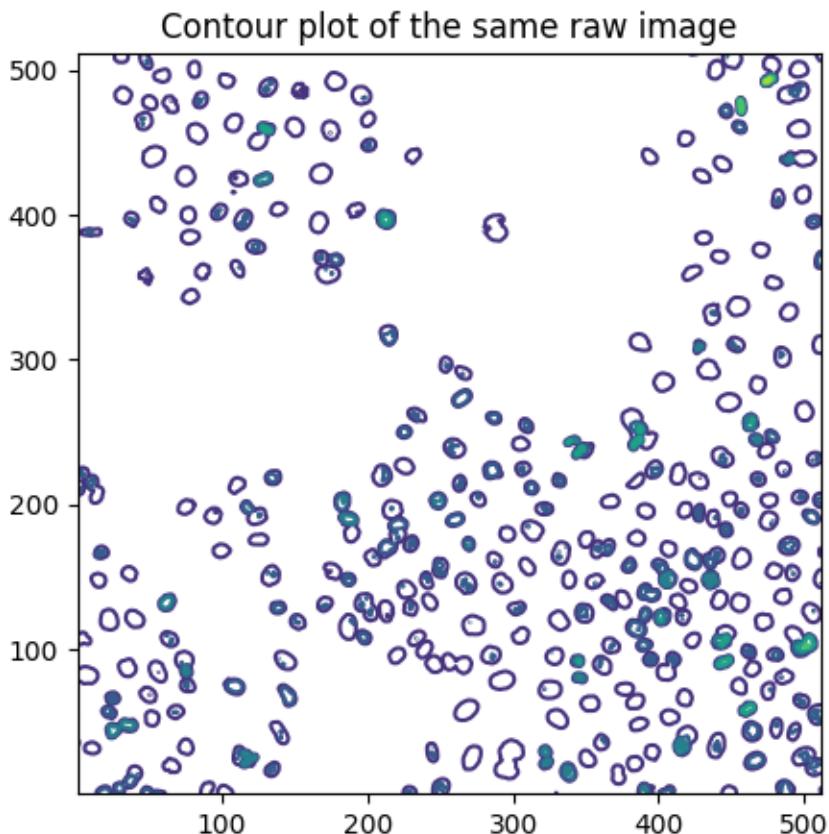
We can see many cell nuclei on a dark background. Most of them are smooth and have an elliptical shape. However,

¹ Moffat J, Grueneberg DA, Yang X, Kim SY, Kloepfer AM, Hinkle G, Piqani B, Eisenhaure TM, Luo B, Grenier JK, Carpenter AE, Foo SY, Stewart SA, Stockwell BR, Hacohen N, Hahn WC, Lander ES, Sabatini DM, Root DE (2006) "A lentiviral RNAi library for human and mouse genes applied to an arrayed viral high-content screen" *Cell*, 124(6):1283-98. PMID: 16564017 DOI:10.1016/j.cell.2006.01.040

we can distinguish some brighter spots corresponding to nuclei undergoing mitosis (cell division).

Another way of visualizing a grayscale image is contour plotting:

```
fig, ax = plt.subplots(figsize=(5, 5))
qcs = ax.contour(image, origin='image')
ax.set_title('Contour plot of the same raw image')
plt.show()
```



The contour lines are drawn at these levels:

```
qcs.levels
```

```
array([  0.,   40.,   80.,  120.,  160.,  200.,  240.,  280.])
```

Each level has, respectively, the following number of segments:

```
[len(seg) for seg in qcs.allsegs]
```

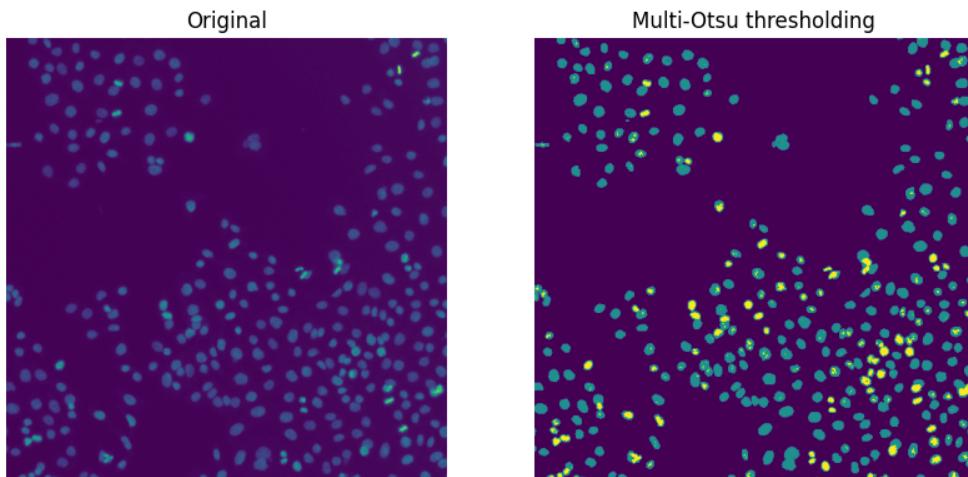
```
[0, 320, 270, 48, 19, 3, 1, 0]
```

Estimate the mitotic index

Cell biology uses the [mitotic index](#) to quantify cell division and, hence, cell proliferation. By definition, it is the ratio of cells in mitosis over the total number of cells. To analyze the above image, we are thus interested in two thresholds: one separating the nuclei from the background, the other separating the dividing nuclei (brighter spots) from the non-dividing nuclei. To separate these three different classes of pixels, we resort to *Multi-Otsu Thresholding*.

```
thresholds = filters.threshold_multiotsu(image, classes=3)
regions = np.digitize(image, bins=thresholds)

fig, ax = plt.subplots(ncols=2, figsize=(10, 5))
ax[0].imshow(image)
ax[0].set_title('Original')
ax[0].axis('off')
ax[1].imshow(regions)
ax[1].set_title('Multi-Otsu thresholding')
ax[1].axis('off')
plt.show()
```



Since there are overlapping nuclei, thresholding is not enough to segment all the nuclei. If it were, we could readily compute a mitotic index for this sample:

```
cells = image > thresholds[0]
dividing = image > thresholds[1]
labeled_cells = measure.label(cells)
labeled_dividing = measure.label(dividing)
naive_mi = labeled_dividing.max() / labeled_cells.max()
print(naive_mi)
```

```
0.7847222222222222
```

Whoa, this can't be! The number of dividing nuclei

```
print(labeled_dividing.max())
```

226

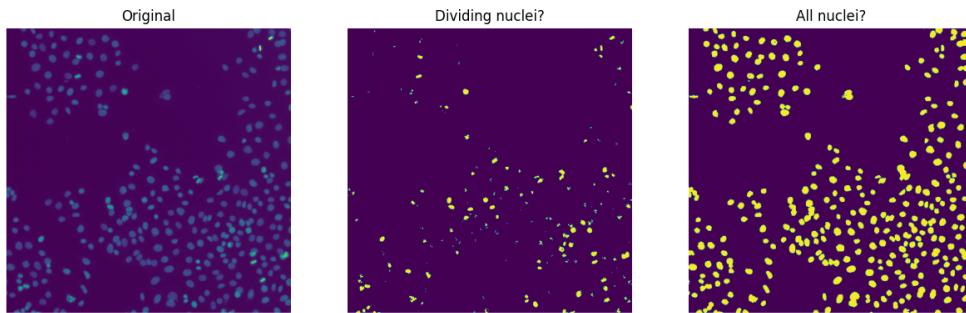
is overestimated, while the total number of cells

```
print(labeled_cells.max())
```

288

is underestimated.

```
fig, ax = plt.subplots(ncols=3, figsize=(15, 5))
ax[0].imshow(image)
ax[0].set_title('Original')
ax[0].axis('off')
ax[2].imshow(cells)
ax[2].set_title('All nuclei?')
ax[2].axis('off')
ax[1].imshow(dividing)
ax[1].set_title('Dividing nuclei?')
ax[1].axis('off')
plt.show()
```



Count dividing nuclei

Clearly, not all connected regions in the middle plot are dividing nuclei. On one hand, the second threshold (value of `thresholds[1]`) appears to be too low to separate those very bright areas corresponding to dividing nuclei from relatively bright pixels otherwise present in many nuclei. On the other hand, we want a smoother image, removing small spurious objects and, possibly, merging clusters of neighboring objects (some could correspond to two nuclei emerging from one cell division). In a way, the segmentation challenge we are facing with dividing nuclei is the opposite of that with (touching) cells.

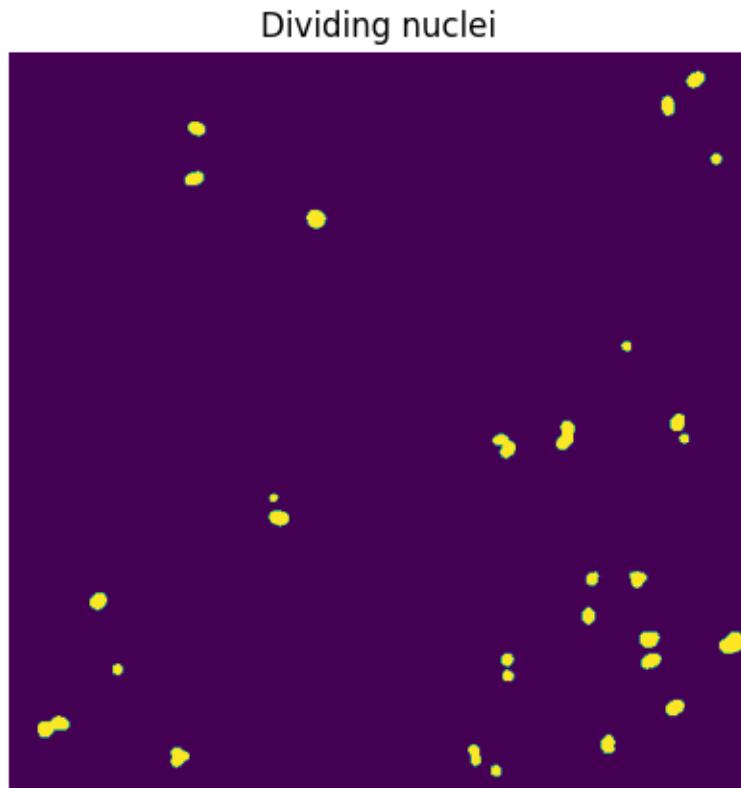
To find suitable values for thresholds and filtering parameters, we proceed by dichotomy, visually and manually.

```
higher_threshold = 125
dividing = image > higher_threshold
```

(continues on next page)

(continued from previous page)

```
smoother_dividing = filters.rank.mean(util.img_as_ubyte(dividing),  
                                     morphology.disk(4))  
  
binary_smoothen_dividing = smoother_dividing > 20  
  
fig, ax = plt.subplots(figsize=(5, 5))  
ax.imshow(binary_smoothen_dividing)  
ax.set_title('Dividing nuclei')  
ax.axis('off')  
plt.show()
```



We are left with

```
cleaned_dividing = measure.label(binary_smoothen_dividing)  
print(cleaned_dividing.max())
```

29

dividing nuclei in this sample.

Segment nuclei

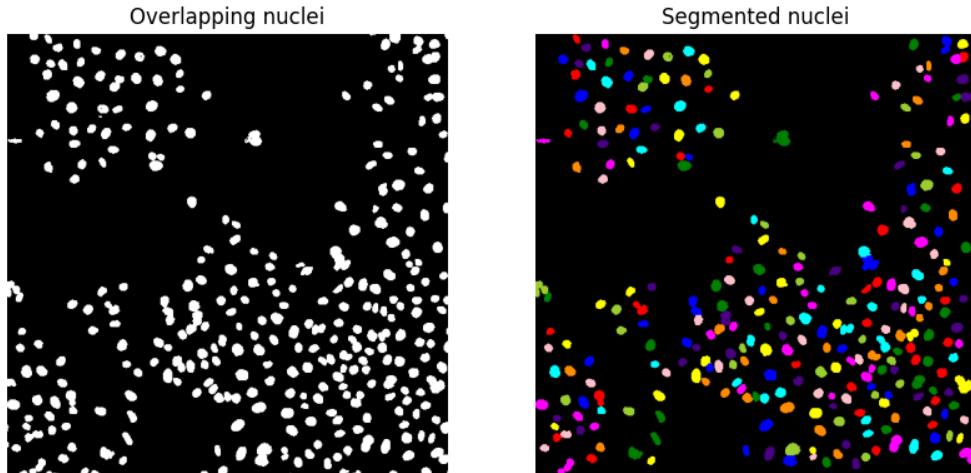
To separate overlapping nuclei, we resort to *Watershed segmentation*. To visualize the segmentation conveniently, we colour-code the labelled regions using the `color.label2rgb` function, specifying the background label with argument `bg_label=0`.

```
distance = ndi.distance_transform_edt(cells)

local_max_coords = feature.peak_local_max(distance, min_distance=7)
local_max_mask = np.zeros(distance.shape, dtype=bool)
local_max_mask[tuple(local_max_coords.T)] = True
markers = measure.label(local_max_mask)

segmented_cells = segmentation.watershed(-distance, markers, mask=cells)

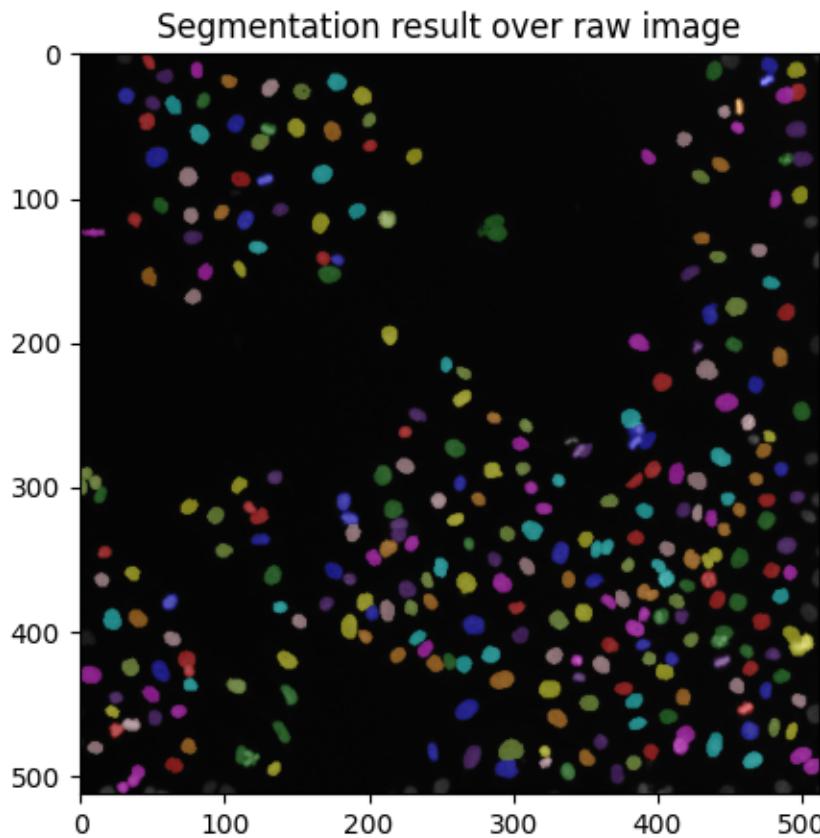
fig, ax = plt.subplots(ncols=2, figsize=(10, 5))
ax[0].imshow(cells, cmap='gray')
ax[0].set_title('Overlapping nuclei')
ax[0].axis('off')
ax[1].imshow(color.label2rgb(segmented_cells, bg_label=0))
ax[1].set_title('Segmented nuclei')
ax[1].axis('off')
plt.show()
```



Additionally, we may use function `color.label2rgb` to overlay the original image with the segmentation result, using transparency (alpha parameter).

```
color_labels = color.label2rgb(segmented_cells, image, alpha=0.4, bg_label=0)

fig, ax = plt.subplots(figsize=(5, 5))
ax.imshow(color_labels)
ax.set_title('Segmentation result over raw image')
plt.show()
```



Finally, we find a total number of

```
print(segmented_cells.max())
```

```
286
```

cells in this sample. Therefore, we estimate the mitotic index to be:

```
print(cleaned_dividing.max() / segmented_cells.max())
```

```
0.10139860139860139
```

Total running time of the script: (0 minutes 1.072 seconds)

Thresholding

Thresholding is used to create a binary image from a grayscale image¹. It is the simplest way to segment objects from a background.

Thresholding algorithms implemented in scikit-image can be separated in two categories:

- Histogram-based. The histogram of the pixels' intensity is used and certain assumptions are made on the properties of this histogram (e.g. bimodal).
- Local. To process a pixel, only the neighboring pixels are used. These algorithms often require more computation time.

If you are not familiar with the details of the different algorithms and the underlying assumptions, it is often difficult to know which algorithm will give the best results. Therefore, Scikit-image includes a function to evaluate thresholding algorithms provided by the library. At a glance, you can select the best algorithm for your data without a deep understanding of their mechanisms.

See also:

Presentation on *Rank filters*.

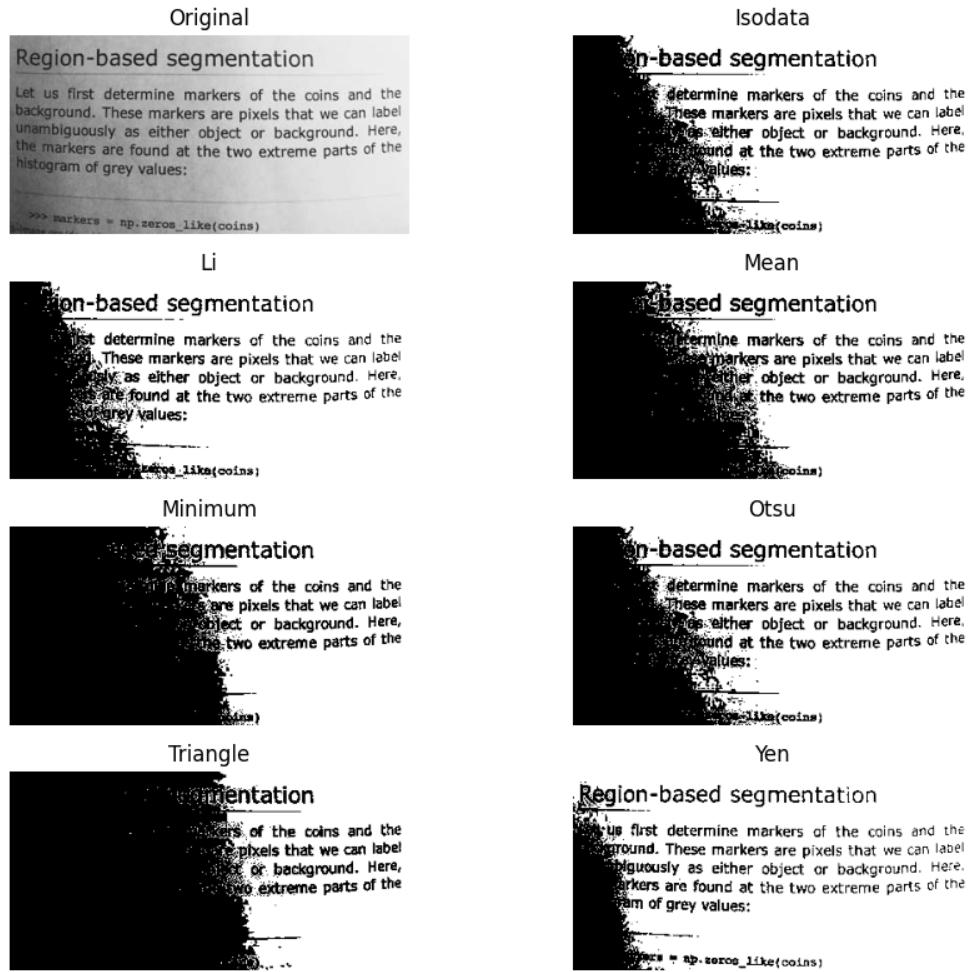
```
import matplotlib.pyplot as plt

from skimage import data
from skimage.filters import try_all_threshold

img = data.page()

fig, ax = try_all_threshold(img, figsize=(10, 8), verbose=False)
plt.show()
```

¹ https://en.wikipedia.org/wiki/Thresholding_%28image_processing%29



How to apply a threshold?

Now, we illustrate how to apply one of these thresholding algorithms. This example uses the mean value of pixel intensities. It is a simple and naive threshold value, which is sometimes used as a guess value.

```
from skimage.filters import threshold_mean

image = data.camera()
thresh = threshold_mean(image)
binary = image > thresh

fig, axes = plt.subplots(ncols=2, figsize=(8, 3))
ax = axes.ravel()

ax[0].imshow(image, cmap=plt.cm.gray)
ax[0].set_title('Original image')

ax[1].imshow(binary, cmap=plt.cm.gray)
ax[1].set_title('Result')
```

(continues on next page)

(continued from previous page)

```
for a in ax:
    a.axis('off')

plt.show()
```



Bimodal histogram

For pictures with a bimodal histogram, more specific algorithms can be used. For instance, the minimum algorithm takes a histogram of the image and smooths it repeatedly until there are only two peaks in the histogram.

```
from skimage.filters import threshold_minimum

image = data.camera()

thresh_min = threshold_minimum(image)
binary_min = image > thresh_min

fig, ax = plt.subplots(2, 2, figsize=(10, 10))

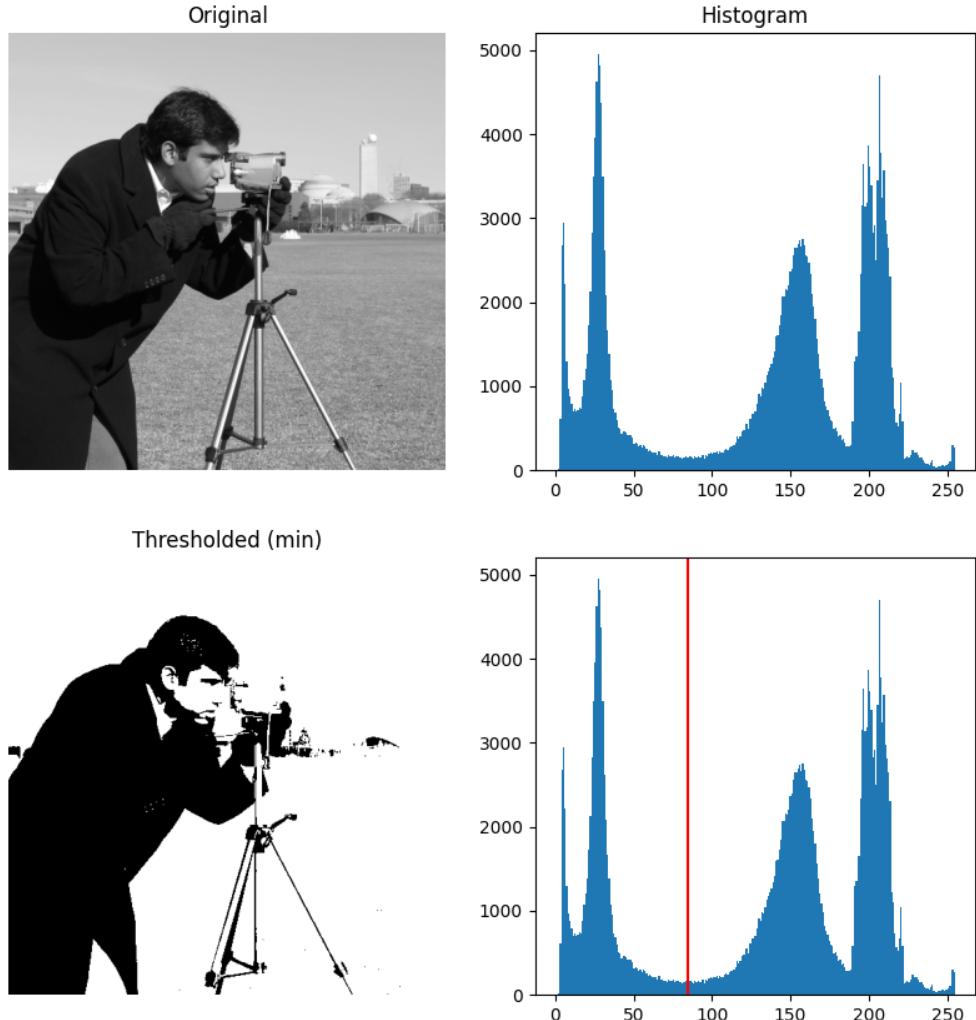
ax[0, 0].imshow(image, cmap=plt.cm.gray)
ax[0, 0].set_title('Original')

ax[0, 1].hist(image.ravel(), bins=256)
ax[0, 1].set_title('Histogram')

ax[1, 0].imshow(binary_min, cmap=plt.cm.gray)
ax[1, 0].set_title('Thresholded (min)')

ax[1, 1].hist(image.ravel(), bins=256)
ax[1, 1].axvline(thresh_min, color='r')

for a in ax[:, 0]:
    a.axis('off')
plt.show()
```



Otsu’s method² calculates an “optimal” threshold (marked by a red line in the histogram below) by maximizing the variance between two classes of pixels, which are separated by the threshold. Equivalently, this threshold minimizes the intra-class variance.

```
from skimage.filters import threshold_otsu

image = data.camera()
thresh = threshold_otsu(image)
binary = image > thresh

fig, axes = plt.subplots(ncols=3, figsize=(8, 2.5))
```

(continues on next page)

² https://en.wikipedia.org/wiki/Otsu's_method

(continued from previous page)

```

ax = axes.ravel()
ax[0] = plt.subplot(1, 3, 1)
ax[1] = plt.subplot(1, 3, 2)
ax[2] = plt.subplot(1, 3, 3, sharex=ax[0], sharey=ax[0])

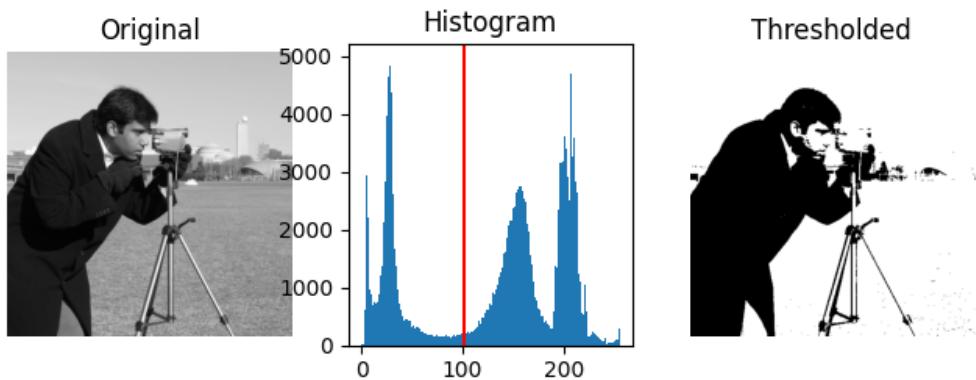
ax[0].imshow(image, cmap=plt.cm.gray)
ax[0].set_title('Original')
ax[0].axis('off')

ax[1].hist(image.ravel(), bins=256)
ax[1].set_title('Histogram')
ax[1].axvline(thresh, color='r')

ax[2].imshow(binary, cmap=plt.cm.gray)
ax[2].set_title('Thresholded')
ax[2].axis('off')

plt.show()

```



```
/github/workspace/build/scikit-image/doc/examples/applications/plot_thresholding_guide.py:125: MatplotlibDeprecationWarning:
```

```
Auto-removal of overlapping axes is deprecated since 3.6 and will be removed two minor releases later; explicitly call ax.remove() as needed.
```

Local thresholding

If the image background is relatively uniform, then you can use a global threshold value as presented above. However, if there is large variation in the background intensity, adaptive thresholding (a.k.a. local or dynamic thresholding) may produce better results. Note that local is much slower than global thresholding.

Here, we binarize an image using the `threshold_local` function, which calculates thresholds in regions with a characteristic size `block_size` surrounding each pixel (i.e. local neighborhoods). Each threshold value is the weighted mean of the local neighborhood minus an offset value.

```
from skimage.filters import threshold_otsu, threshold_local
```

(continues on next page)

(continued from previous page)

```
image = data.page()

global_thresh = threshold_otsu(image)
binary_global = image > global_thresh

block_size = 35
local_thresh = threshold_local(image, block_size, offset=10)
binary_local = image > local_thresh

fig, axes = plt.subplots(nrows=3, figsize=(7, 8))
ax = axes.ravel()
plt.gray()

ax[0].imshow(image)
ax[0].set_title('Original')

ax[1].imshow(binary_global)
ax[1].set_title('Global thresholding')

ax[2].imshow(binary_local)
ax[2].set_title('Local thresholding')

for a in ax:
    a.axis('off')

plt.show()
```

Original

Region-based segmentation

Let us first determine markers of the coins and the background. These markers are pixels that we can label unambiguously as either object or background. Here, the markers are found at the two extreme parts of the histogram of grey values:

```
>>> markers = np.zeros_like(coins)
```

Global thresholding

Region-based segmentation

Let us first determine markers of the coins and the background. These markers are pixels that we can label unambiguously as either object or background. Here, the markers are found at the two extreme parts of the histogram of grey values:

```
>>> markers = np.zeros_like(coins)
```

Local thresholding

Region-based segmentation

Let us first determine markers of the coins and the background. These markers are pixels that we can label unambiguously as either object or background. Here, the markers are found at the two extreme parts of the histogram of grey values:

```
>>> markers = np.zeros_like(coins)
```

Now, we show how Otsu's threshold⁷ method can be applied locally. For each pixel, an “optimal” threshold is determined by maximizing the variance between two classes of pixels of the local neighborhood defined by a structuring element.

The example compares the local threshold with the global threshold.

```
from skimage.morphology import disk
from skimage.filters import threshold_otsu, rank
```

(continues on next page)

(continued from previous page)

```
from skimage.util import img_as_ubyte

img = img_as_ubyte(data.page())

radius = 15
footprint = disk(radius)

local_otsu = rank.otsu(img, footprint)
threshold_global_otsu = threshold_otsu(img)
global_otsu = img >= threshold_global_otsu

fig, axes = plt.subplots(2, 2, figsize=(8, 5), sharex=True, sharey=True)
ax = axes.ravel()
plt.tight_layout()

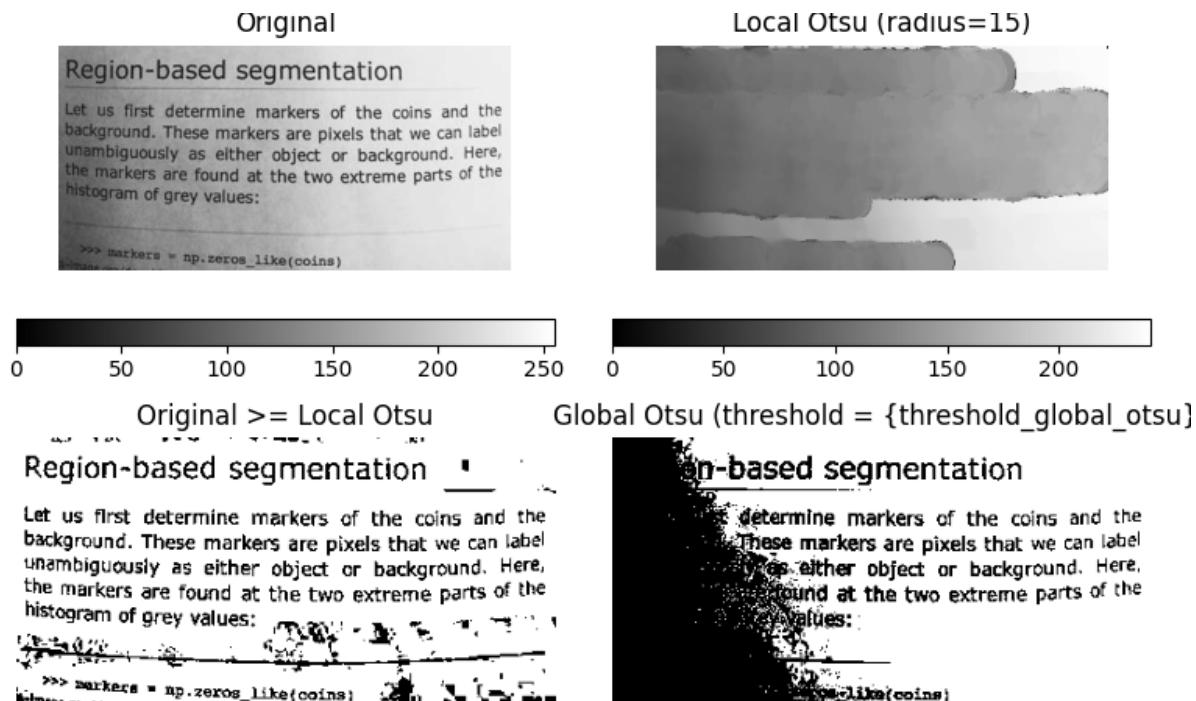
fig.colorbar(ax[0].imshow(img, cmap=plt.cm.gray),
             ax=ax[0], orientation='horizontal')
ax[0].set_title('Original')
ax[0].axis('off')

fig.colorbar(ax[1].imshow(local_otsu, cmap=plt.cm.gray),
             ax=ax[1], orientation='horizontal')
ax[1].set_title(f'Local Otsu (radius={radius})')
ax[1].axis('off')

ax[2].imshow(img >= local_otsu, cmap=plt.cm.gray)
ax[2].set_title('Original >= Local Otsu')
ax[2].axis('off')

ax[3].imshow(global_otsu, cmap=plt.cm.gray)
ax[3].set_title('Global Otsu (threshold = {threshold_global_otsu})')
ax[3].axis('off')

plt.show()
```



Total running time of the script: (0 minutes 2.578 seconds)

Track solidification of a metallic alloy

In this example, we identify and track the solid-liquid (S-L) interface in a nickel-based alloy undergoing solidification. Tracking the solidification over time enables the calculation of the solidification velocity. This is important to characterize the solidified structure of the sample and will be used to inform research into additive manufacturing of metals. The image sequence was obtained by the Center for Advanced Non-Ferrous Structural Alloys (CANFSA) using synchrotron x-radiography at the Advanced Photon Source (APS) at Argonne National Laboratory (ANL). This analysis was first presented at a conference¹.

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import plotly.express as px
import plotly.io

from skimage import filters, measure, restoration
from skimage.data import nickel_solidification

image_sequence = nickel_solidification()

y0, y1, x0, x1 = 0, 180, 100, 330

image_sequence = image_sequence[:, y0:y1, x0:x1]

print(f'shape: {image_sequence.shape}')
```

¹ Corvellec M. and Becker C. G. (2021, May 17-18) "Quantifying solidification of metallic alloys with scikit-image" [Conference presentation]. BIDS ImageXD 2021 (Image Analysis Across Domains). Virtual participation. <https://www.youtube.com/watch?v=cB1HTgmWTd8>

```
shape: (11, 180, 230)
```

The dataset is a 2D image stack with 11 frames (time points). We visualize and analyze it in a workflow where the first image processing steps are performed on the entire three-dimensional dataset (i.e., across space and time), such that the removal of localized, transient noise is favored as opposed to that of physical features (e.g., bubbles, splatters, etc.), which exist in roughly the same position from one frame to the next.

```
fig = px.imshow(
    image_sequence,
    animation_frame=0,
    binary_string=True,
    labels={'animation_frame': 'time point'}
)
plotly.io.show(fig)
```

Compute image deltas

Let us first apply a Gaussian low-pass filter in order to smooth the images and reduce noise. Next, we compute the image deltas, i.e., the sequence of differences between two consecutive frames. To do this, we subtract the image sequence ending at the second-to-last frame from the image sequence starting at the second frame.

```
smoothed = filters.gaussian(image_sequence)
image_deltas = smoothed[1:, :, :] - smoothed[:-1, :, :]

fig = px.imshow(
    image_deltas,
    animation_frame=0,
    binary_string=True,
    labels={'animation_frame': 'time point'}
)
plotly.io.show(fig)
```

Clip lowest and highest intensities

We now calculate the 5th and 95th percentile intensities of `image_deltas`: We want to clip the intensities which lie below the 5th percentile intensity and above the 95th percentile intensity, while also rescaling the intensity values to [0, 1].

```
p_low, p_high = np.percentile(image_deltas, [5, 95])
clipped = image_deltas - p_low
clipped[clipped < 0.0] = 0.0
clipped = clipped / p_high
clipped[clipped > 1.0] = 1.0

fig = px.imshow(
    clipped,
    animation_frame=0,
    binary_string=True,
    labels={'animation_frame': 'time point'}
```

(continues on next page)

(continued from previous page)

```
)  
plotly.io.show(fig)
```

Invert and denoise

We invert the clipped images so the regions of highest intensity will include the region we are interested in tracking (i.e., the S-L interface). Then, we apply a total variation denoising filter to reduce noise beyond the interface.

```
inverted = 1 - clipped  
denoised = restoration.denoise_tv_chambolle(inverted, weight=0.15)  
  
fig = px.imshow(  
    denoised,  
    animation_frame=0,  
    binary_string=True,  
    labels={'animation_frame': 'time point'}  
)  
plotly.io.show(fig)
```

Binarize

Our next step is to create binary images, splitting the images into foreground and background: We want the S-L interface to be the most prominent feature in the foreground of each binary image, so that it can eventually be separated from the rest of the image.

We need a threshold value `thresh_val` to create our binary images, `binarized`. This can be set manually, but we shall use an automated minimum threshold method from the `filters` submodule of scikit-image (there are other methods that may work better for different applications).

```
thresh_val = filters.threshold_minimum(denoised)  
binarized = denoised > thresh_val  
  
fig = px.imshow(  
    binarized,  
    animation_frame=0,  
    binary_string=True,  
    labels={'animation_frame': 'time point'}  
)  
plotly.io.show(fig)
```

Select largest region

In our binary images, the S-L interface appears as the largest region of connected pixels. For this step of the workflow, we will operate on each 2D image separately, as opposed to the entire 3D dataset, because we are interested in a single moment in time for each region.

We apply `skimage.measure.label()` on the binary images so that each region has its own label. Then, we select the largest region in each image by computing region properties, including the `area` property, and sorting by `area` values. Function `skimage.measure.regionprops_table()` returns a table of region properties which can be read into a Pandas DataFrame. To begin with, let us consider the first image delta at this stage of the workflow, `binarized[0, :, :]`.

```
labeled_0 = measure.label(binarized[0, :, :])
props_0 = measure.regionprops_table(
    labeled_0, properties=['label', 'area', 'bbox'])
props_0_df = pd.DataFrame(props_0)
props_0_df = props_0_df.sort_values('area', ascending=False)
# Show top five rows
props_0_df.head()
```

We can thus select the largest region by matching its label with the one found in the first row of the above (sorted) table. Let us visualize it, along with its bounding box (bbox) in red.

```
largest_region_0 = labeled_0 == props_0_df.iloc[0]['label']
minr, minc, maxr, maxc = (props_0_df.iloc[0][f'bbox-{i}'] for i in range(4))
fig = px.imshow(largest_region_0, binary_string=True)
fig.add_shape(
    type='rect', x0=minc, y0=minr, x1=maxc, y1=maxr, line=dict(color='Red'))
plotly.io.show(fig)
```

We can see how the lower bounds of the box align with the bottom of the S-L interface by overlaying the same bounding box onto the 0th raw image. This bounding box was calculated from the image delta between the 0th and 1st images, but the bottom-most region of the box corresponds to the location of the interface earlier in time (0th image) because the interface is moving upward.

```
fig = px.imshow(image_sequence[0, :, :], binary_string=True)
fig.add_shape(
    type='rect', x0=minc, y0=minr, x1=maxc, y1=maxr, line=dict(color='Red'))
plotly.io.show(fig)
```

We are now ready to perform this selection for all image deltas in the sequence. We shall also store the bbox information, which will be used to track the position of the S-L interface.

```
largest_region = np.empty_like(binarized)
bboxes = []

for i in range(binarized.shape[0]):
    labeled = measure.label(binarized[i, :, :])
    props = measure.regionprops_table(
        labeled, properties=['label', 'area', 'bbox'])
    props_df = pd.DataFrame(props)
    props_df = props_df.sort_values('area', ascending=False)
    largest_region[i, :, :] = (labeled == props_df.iloc[0]['label'])
    bboxes.append([props_df.iloc[0][f'bbox-{i}'] for i in range(4)])
```

(continues on next page)

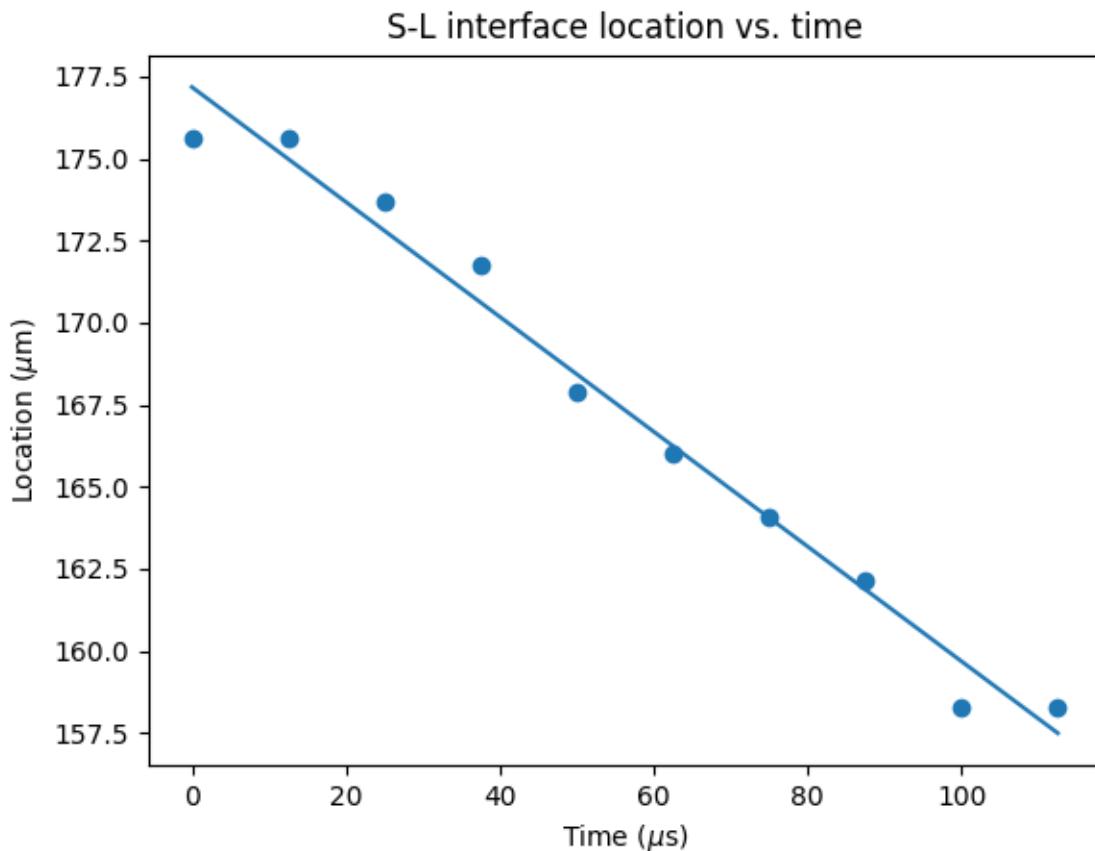
(continued from previous page)

```
fig = px.imshow(
    largest_region,
    animation_frame=0,
    binary_string=True,
    labels={'animation_frame': 'time point'}
)
plotly.io.show(fig)
```

Plot interface location over time

The final step in this analysis is to plot the location of the S-L interfaces over time. This can be achieved by plotting `maxr` (third element in `bbox`) over time since this value shows the y location of the bottom of the interface. The pixel size in this experiment was 1.93 microns and the framerate was 80,000 frames per second, so these values are used to convert pixels and image number to physical units. We calculate the average solidification velocity by fitting a linear polynomial to the scatter plot. The velocity is the first-order coefficient.

```
ums_per_pixel = 1.93
fps = 80000
interface_y_um = [ums_per_pixel * bbox[2] for bbox in bboxes]
time_us = 1e6 / fps * np.arange(len(interface_y_um))
fig, ax = plt.subplots(dpi=100)
ax.scatter(time_us, interface_y_um)
c0, c1 = np.polynomial.polynomial.polyfit(time_us, interface_y_um, 1)
ax.plot(time_us, c1 * time_us + c0, label=f'Velocity: {abs(round(c1, 3))} m/s')
ax.set_title('S-L interface location vs. time')
ax.set_ylabel(r'Location ($\mu\text{m}$)')
ax.set_xlabel(r'Time ($\mu\text{s}$)')
plt.show()
```



Total running time of the script: (0 minutes 3.282 seconds)

Measure fluorescence intensity at the nuclear envelope

This example reproduces a well-established workflow in bioimage data analysis for measuring the fluorescence intensity localized to the nuclear envelope, in a time sequence of cell images (each with two channels and two spatial dimensions) which shows a process of protein re-localization from the cytoplasmic area to the nuclear envelope. This biological application was first presented by Andrea Boni and collaborators in¹; it was used in a textbook by Kota Miura² as well as in other works (^{3,4}). In other words, we port this workflow from ImageJ Macro to Python with scikit-image.

```
import matplotlib.pyplot as plt
import numpy as np
import plotly.io
import plotly.express as px
from scipy import ndimage as ndi

from skimage import (
    filters, measure, morphology, segmentation
)
```

(continues on next page)

¹ Boni A, Politi AZ, Strnad P, Xiang W, Hossain MJ, Ellenberg J (2015) “Live imaging and modeling of inner nuclear membrane targeting reveals its molecular requirements in mammalian cells” *J Cell Biol* 209(5):705–720. ISSN: 0021-9525. DOI:10.1083/jcb.201409133

² Miura K (2020) “Measurements of Intensity Dynamics at the Periphery of the Nucleus” in: Miura K, Sladoje N (eds) Bioimage Data Analysis Workflows. Learning Materials in Biosciences. Springer, Cham. DOI:10.1007/978-3-030-22386-1_2

³ Klemm A (2020) “ImageJ/Fiji Macro Language” NEUBIAS Academy Online Course: <https://www.youtube.com/watch?v=o8tfkcd3DA>

⁴ Vorkel D and Haase R (2020) “GPU-accelerating ImageJ Macro image processing workflows using CLIJ” <https://arxiv.org/abs/2008.11799>

(continued from previous page)

```
from skimage.data import protein_transport
```

We start with a single cell/nucleus to construct the workflow.

```
image_sequence = protein_transport()

print(f'shape: {image_sequence.shape}')
```

```
shape: (15, 2, 180, 183)
```

The dataset is a 2D image stack with 15 frames (time points) and 2 channels.

```
vmin, vmax = 0, image_sequence.max()

fig = px.imshow(
    image_sequence,
    facet_col=1,
    animation_frame=0,
    zmin=vmin,
    zmax=vmax,
    binary_string=True,
    labels={'animation_frame': 'time point', 'facet_col': 'channel'}
)
plotly.io.show(fig)
```

To begin with, let us consider the first channel of the first image (step a) in the figure below).

```
image_t_0_channel_0 = image_sequence[0, 0, :, :]
```

Segment the nucleus rim

Let us apply a Gaussian low-pass filter to this image in order to smooth it (step b)). Next, we segment the nuclei, finding the threshold between the background and foreground with Otsu's method: We get a binary image (step c)). We then fill the holes in the objects (step c-1)).

```
smooth = filters.gaussian(image_t_0_channel_0, sigma=1.5)

thresh_value = filters.threshold_otsu(smooth)
thresh = smooth > thresh_value

fill = ndi.binary_fill_holes(thresh)
```

Following the original workflow, let us remove objects which touch the image border (step c-2)). Here, we can see that part of another nucleus was touching the bottom right-hand corner.

```
clear = segmentation.clear_border(fill)
clear.dtype
```

```
dtype('bool')
```

We compute both the morphological dilation of this binary image (step d)) and its morphological erosion (step e)).

```
dilate = morphology.binary_dilation(clear)

erode = morphology.binary_erosion(clear)
```

Finally, we subtract the eroded from the dilated to get the nucleus rim (step f)). This is equivalent to selecting the pixels which are in dilate, but not in erode:

```
mask = np.logical_and(dilate, ~erode)
```

Let us visualize these processing steps in a sequence of subplots.

```
fig, ax = plt.subplots(2, 4, figsize=(12, 6), sharey=True)

ax[0, 0].imshow(image_t_0_channel_0, cmap=plt.cm.gray)
ax[0, 0].set_title('a) Raw')

ax[0, 1].imshow(smooth, cmap=plt.cm.gray)
ax[0, 1].set_title('b) Blur')

ax[0, 2].imshow(thresh, cmap=plt.cm.gray)
ax[0, 2].set_title('c) Threshold')

ax[0, 3].imshow(fill, cmap=plt.cm.gray)
ax[0, 3].set_title('c-1) Fill in')

ax[1, 0].imshow(clear, cmap=plt.cm.gray)
ax[1, 0].set_title('c-2) Keep one nucleus')

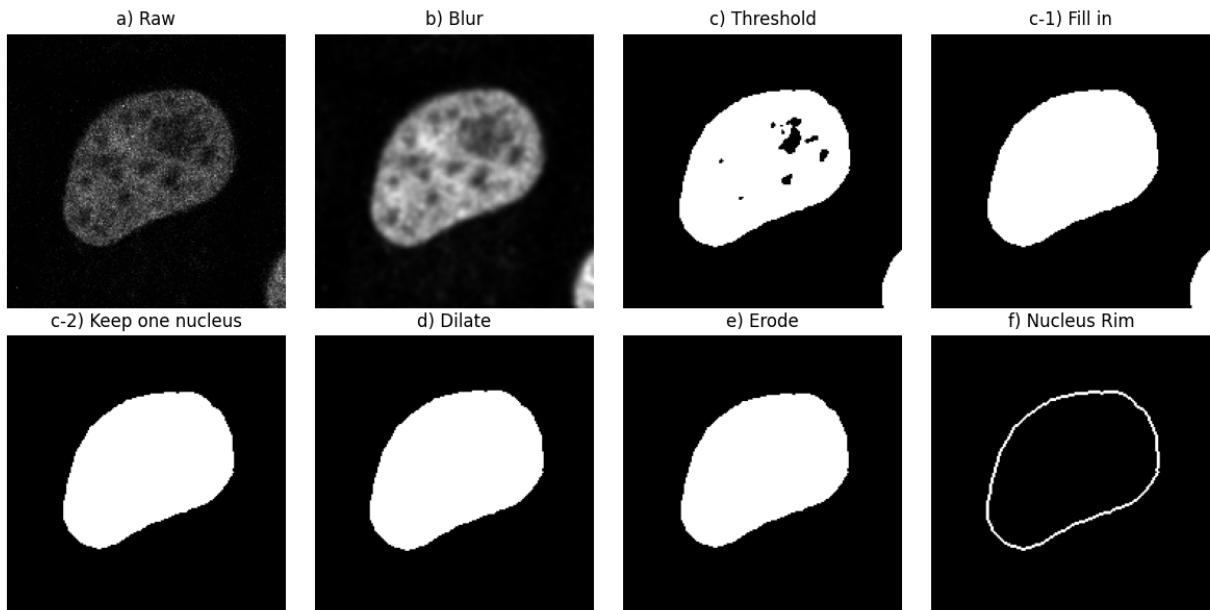
ax[1, 1].imshow(dilate, cmap=plt.cm.gray)
ax[1, 1].set_title('d) Dilate')

ax[1, 2].imshow(erode, cmap=plt.cm.gray)
ax[1, 2].set_title('e) Erode')

ax[1, 3].imshow(mask, cmap=plt.cm.gray)
ax[1, 3].set_title('f) Nucleus Rim')

for a in ax.ravel():
    a.set_axis_off()

fig.tight_layout()
```



Apply the segmented rim as a mask

Now that we have segmented the nuclear membrane in the first channel, we use it as a mask to measure the intensity in the second channel.

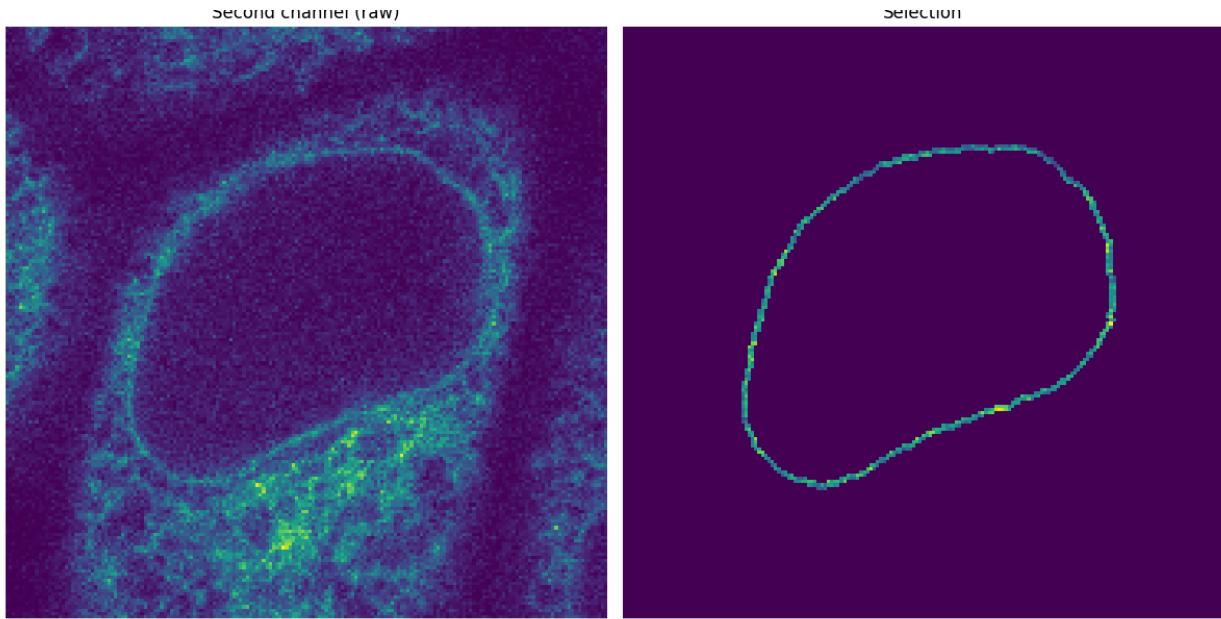
```
image_t_0_channel_1 = image_sequence[0, 1, :, :]
selection = np.where(mask, image_t_0_channel_1, 0)

fig, (ax0, ax1) = plt.subplots(1, 2, figsize=(12, 6), sharey=True)

ax0.imshow(image_t_0_channel_1)
ax0.set_title('Second channel (raw)')
ax0.set_axis_off()

ax1.imshow(selection)
ax1.set_title('Selection')
ax1.set_axis_off()

fig.tight_layout()
```



Measure the total intensity

The mean intensity is readily available as a region property in a labeled image.

```
props = measure.regionprops_table(  
    mask.astype(np.uint8),  
    intensity_image=image_t_0_channel_1,  
    properties=['label', 'area', 'intensity_mean'])  
)
```

We may want to check that the value for the total intensity

```
selection.sum()
```

```
28350
```

can be recovered from:

```
props['area'] * props['intensity_mean']
```

```
array([28350.])
```

Process the entire image sequence

Instead of iterating the workflow for each time point, we process the multidimensional dataset directly (except for the thresholding step). Indeed, most scikit-image functions support nD images.

```
n_z = image_sequence.shape[0] # number of frames

smooth_seq = filters.gaussian(image_sequence[:, 0, :, :], sigma=(0, 1.5, 1.5))
thresh_values = [filters.threshold_otsu(s) for s in smooth_seq[:, :]]
thresh_seq = [smooth_seq[k, ...] > val for k, val in enumerate(thresh_values)]
```

Alternatively, we could compute `thresh_values` without using a list comprehension, by reshaping `smooth_seq` to make it 2D (where the first dimension still corresponds to time points, but the second and last dimension now contains all pixel values), and applying the thresholding function on the image sequence along its second axis:

```
thresh_values = np.apply_along_axis(filters.threshold_otsu,
                                    axis=1,
                                    arr=smooth_seq.reshape(n_z, -1))
```

We use the following flat structuring element for morphological computations (`np.newaxis` is used to prepend an axis of size 1 for time):

```
footprint = ndi.generate_binary_structure(2, 1)[np.newaxis, ...]
footprint
```

```
array([[[False,  True,  False],
       [ True,  True,  True],
       [False,  True,  False]]])
```

This way, each frame is processed independently (pixels from consecutive frames are never spatial neighbors).

```
fill_seq = ndi.binary_fill_holes(thresh_seq, structure=footprint)
```

When clearing objects which touch the image border, we want to make sure that the bottom (first) and top (last) frames are not considered as borders. In this case, the only relevant border is the edge at the greatest (x, y) values. This can be seen in 3D by running the following code:

```
import plotly.graph_objects as go

sample = fill_seq
(n_Z, n_Y, n_X) = sample.shape
Z, Y, X = np.mgrid[:n_Z, :n_Y, :n_X]

fig = go.Figure(
    data=go.Volume(
        x=X.flatten(),
        y=Y.flatten(),
        z=Z.flatten(),
        value=sample.flatten(),
        opacity=0.5,
        slices_z=dict(show=True, locations=[n_z // 2])
    )
)
fig.show()
```

```

border_mask = np.ones_like(fill_seq)
border_mask[n_z // 2, -1, -1] = False
clear_seq = segmentation.clear_border(fill_seq, mask=border_mask)

dilate_seq = morphology.binary_dilation(clear_seq, footprint=footprint)
erode_seq = morphology.binary_erosion(clear_seq, footprint=footprint)
mask_sequence = np.logical_and(dilate_seq, ~erode_seq)

```

Let us give each mask (corresponding to each time point) a different label, running from 1 to 15. We use `np.min_scalar_type` to determine the minimum-size integer dtype needed to represent the number of time points:

```

label_dtype = np.min_scalar_type(n_z)
mask_sequence = mask_sequence.astype(label_dtype)
labels = np.arange(1, n_z + 1, dtype=label_dtype)
mask_sequence *= labels[:, np.newaxis, np.newaxis]

```

Let us compute the region properties of interest for all these labeled regions.

```

props = measure.regionprops_table(
    mask_sequence,
    intensity_image=image_sequence[:, 1, :, :],
    properties=['label', 'area', 'intensity_mean']
)
np.testing.assert_array_equal(props['label'], np.arange(n_z) + 1)

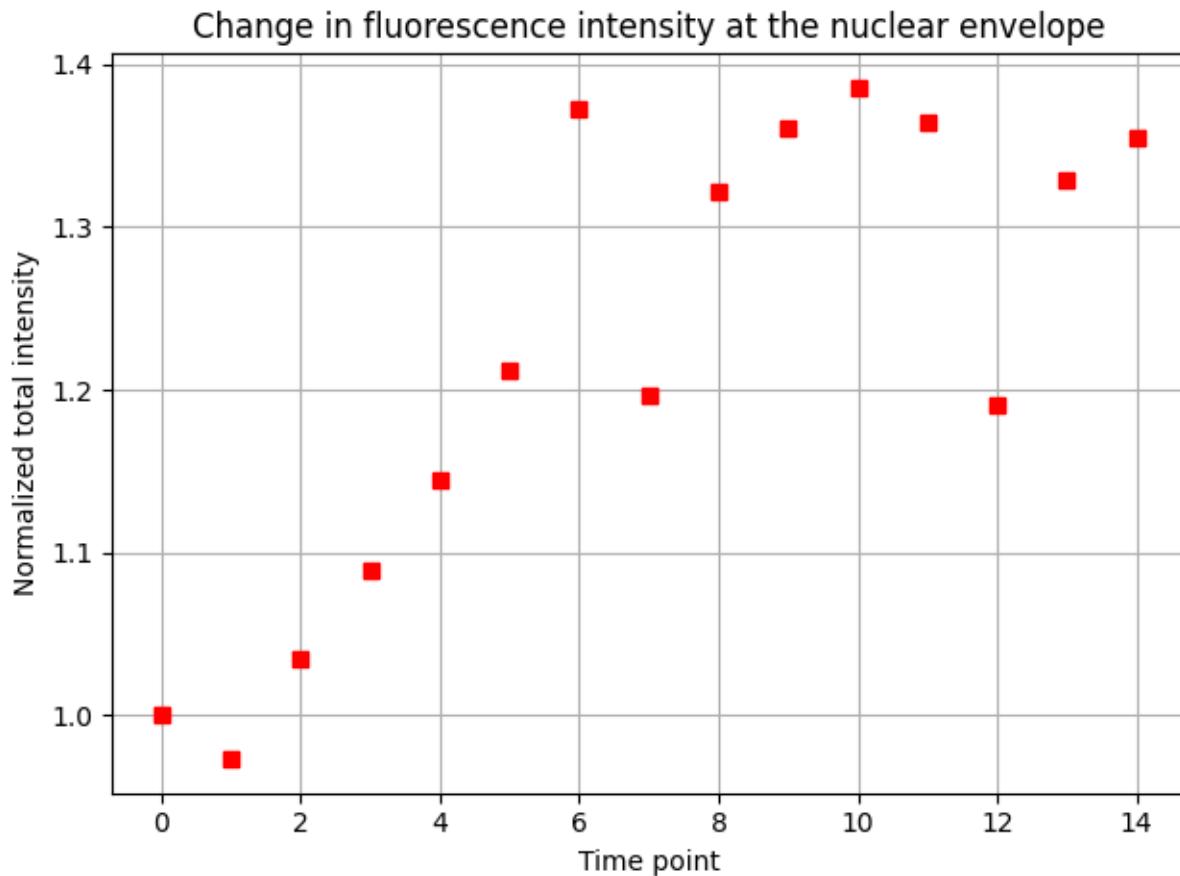
fluorescence_change = [props['area'][i] * props['intensity_mean'][i]
                      for i in range(n_z)]

fluorescence_change /= fluorescence_change[0] # normalization

fig, ax = plt.subplots()
ax.plot(fluorescence_change, 'rs')
ax.grid()
ax.set_xlabel('Time point')
ax.set_ylabel('Normalized total intensity')
ax.set_title('Change in fluorescence intensity at the nuclear envelope')
fig.tight_layout()

plt.show()

```



Reassuringly, we find the expected result: The total fluorescence intensity at the nuclear envelope increases 1.3-fold in the initial five time points, and then becomes roughly constant.

Total running time of the script: (0 minutes 1.409 seconds)

Face classification using Haar-like feature descriptor

Haar-like feature descriptors were successfully used to implement the first real-time face detector¹. Inspired by this application, we propose an example illustrating the extraction, selection, and classification of Haar-like features to detect faces vs. non-faces.

¹ Viola, Paul, and Michael J. Jones. "Robust real-time face detection." International journal of computer vision 57.2 (2004): 137-154. <https://www.merl.com/publications/docs/TR2004-043.pdf> DOI:10.1109/CVPR.2001.990517

Notes

This example relies on scikit-learn for feature selection and classification.

References

```
from time import time

import numpy as np
import matplotlib.pyplot as plt

from dask import delayed

from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score

from skimage.data import lfw_subset
from skimage.transform import integral_image
from skimage.feature import haar_like_feature
from skimage.feature import haar_like_feature_coord
from skimage.feature import draw_haar_like_feature
```

The procedure to extract the Haar-like features from an image is relatively simple. Firstly, a region of interest (ROI) is defined. Secondly, the integral image within this ROI is computed. Finally, the integral image is used to extract the features.

```
@delayed
def extract_feature_image(img, feature_type, feature_coord=None):
    """Extract the haar feature for the current image"""
    ii = integral_image(img)
    return haar_like_feature(ii, 0, 0, ii.shape[0], ii.shape[1],
                            feature_type=feature_type,
                            feature_coord=feature_coord)
```

We use a subset of CBCL dataset which is composed of 100 face images and 100 non-face images. Each image has been resized to a ROI of 19 by 19 pixels. We select 75 images from each group to train a classifier and determine the most salient features. The remaining 25 images from each class are used to assess the performance of the classifier.

```
images = lfw_subset()
# To speed up the example, extract the two types of features only
feature_types = ['type-2-x', 'type-2-y']

# Build a computation graph using Dask. This allows the use of multiple
# CPU cores later during the actual computation
X = delayed(extract_feature_image(img, feature_types) for img in images)
# Compute the result
t_start = time()
X = np.array(X.compute(scheduler='single-threaded'))
```

(continues on next page)

(continued from previous page)

```

time_full_feature_comp = time() - t_start

# Label images (100 faces and 100 non-faces)
y = np.array([1] * 100 + [0] * 100)

X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=150,
                                                    random_state=0,
                                                    stratify=y)

# Extract all possible features
feature_coord, feature_type = \
    haar_like_feature_coord(width=images.shape[2], height=images.shape[1],
                            feature_type=feature_types)

```

A random forest classifier can be trained in order to select the most salient features, specifically for face classification. The idea is to determine which features are most often used by the ensemble of trees. By using only the most salient features in subsequent steps, we can drastically speed up the computation while retaining accuracy.

```

# Train a random forest classifier and assess its performance
clf = RandomForestClassifier(n_estimators=1000, max_depth=None,
                            max_features=100, n_jobs=-1, random_state=0)
t_start = time()
clf.fit(X_train, y_train)
time_full_train = time() - t_start
auc_full_features = roc_auc_score(y_test, clf.predict_proba(X_test)[:, 1])

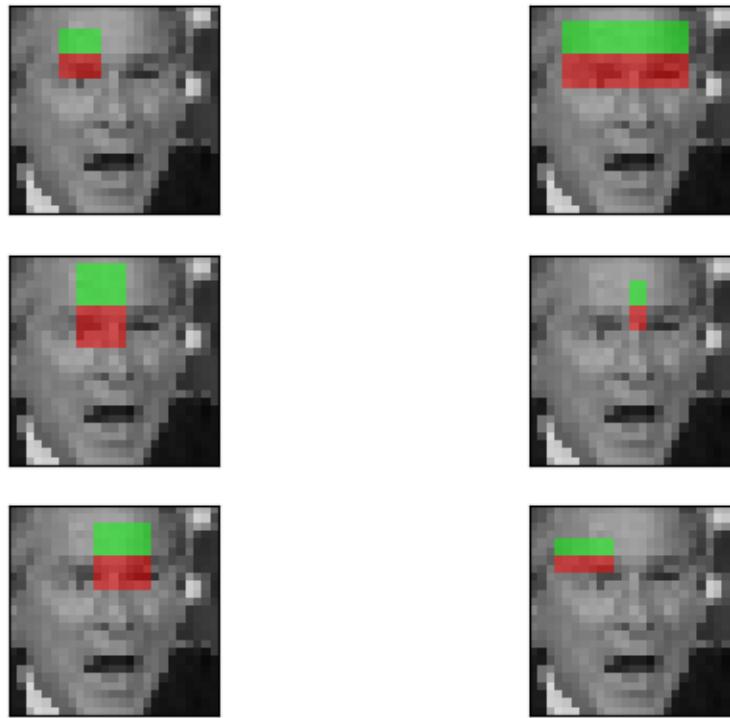
# Sort features in order of importance and plot the six most significant
idx_sorted = np.argsort(clf.feature_importances_)[-1:]

fig, axes = plt.subplots(3, 2)
for idx, ax in enumerate(axes.ravel()):
    image = images[0]
    image = draw_haar_like_feature(image, 0, 0,
                                    images.shape[2],
                                    images.shape[1],
                                    [feature_coord[idx_sorted[idx]]])
    ax.imshow(image)
    ax.set_xticks([])
    ax.set_yticks([])

_= fig.suptitle('The most important features')

```

The most important features



We can select the most important features by checking the cumulative sum of the feature importance. In this example, we keep the features representing 70% of the cumulative value (which corresponds to using only 3% of the total number of features).

```

cdf_feature_importances = np.cumsum(clf.feature_importances_[idx_sorted])
cdf_feature_importances /= cdf_feature_importances[-1] # divide by max value
sig_feature_count = np.count_nonzero(cdf_feature_importances < 0.7)
sig_feature_percent = round(sig_feature_count /
                             len(cdf_feature_importances) * 100, 1)
print(f'{sig_feature_count} features, or {sig_feature_percent}%, '
      f'account for 70% of branch points in the random forest.')

# Select the determined number of most informative features
feature_coord_sel = feature_coord[idx_sorted[:sig_feature_count]]
feature_type_sel = feature_type[idx_sorted[:sig_feature_count]]
# Note: it is also possible to select the features directly from the matrix X,
# but we would like to emphasize the usage of `feature_coord` and `feature_type`
# to recompute a subset of desired features.

# Build the computational graph using Dask
X = delayed(extract_feature_image(img, feature_type_sel, feature_coord_sel))
    for img in images)
# Compute the result
t_start = time()

```

(continues on next page)

(continued from previous page)

```
X = np.array(X.compute(scheduler='single-threaded'))
time_subs_feature_comp = time() - t_start

y = np.array([1] * 100 + [0] * 100)
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=150,
                                                    random_state=0,
                                                    stratify=y)
```

712 features, or 0.7%, account for 70% of branch points in the random forest.

Once the features are extracted, we can train and test a new classifier.

```
t_start = time()
clf.fit(X_train, y_train)
time_subs_train = time() - t_start

auc_subs_features = roc_auc_score(y_test, clf.predict_proba(X_test)[:, 1])

summary = (f'Computing the full feature set took '
           f'{time_full_feature_comp:.3f}s, '
           f'plus {time_full_train:.3f}s training, '
           f'for an AUC of {auc_full_features:.2f}. '
           f'Computing the restricted feature set took '
           f'{time_subs_feature_comp:.3f}s, plus {time_subs_train:.3f}s '
           f'training, for an AUC of {auc_subs_features:.2f}.')
print(summary)
plt.show()
```

Computing the full feature set took 36.246s, plus 2.604s training, for an AUC of 1.00.
 ↵ Computing the restricted feature set took 0.104s, plus 2.020s training, for an AUC of
 ↵ 1.00.

Total running time of the script: (0 minutes 46.048 seconds)

Explore 3D images (of cells)

This tutorial is an introduction to three-dimensional image processing. For a quick intro to 3D datasets, please refer to *Datasets with 3 or more spatial dimensions*. Images are represented as `numpy` arrays. A single-channel, or grayscale, image is a 2D matrix of pixel intensities of shape `(n_row, n_col)`, where `n_row` (resp. `n_col`) denotes the number of rows (resp. columns). We can construct a 3D volume as a series of 2D *planes*, giving 3D images the shape `(n_plane, n_row, n_col)`, where `n_plane` is the number of planes. A multichannel, or RGB(A), image has an additional *channel* dimension in the final position containing color information.

These conventions are summarized in the table below:

Image type	Coordinates
2D grayscale	[row, column]
2D multichannel	[row, column, channel]
3D grayscale	[plane, row, column]
3D multichannel	[plane, row, column, channel]

Some 3D images are constructed with equal resolution in each dimension (e.g., synchrotron tomography or computer-generated rendering of a sphere). But most experimental data are captured with a lower resolution in one of the three dimensions, e.g., photographing thin slices to approximate a 3D structure as a stack of 2D images. The distance between pixels in each dimension, called spacing, is encoded as a tuple and is accepted as a parameter by some `skimage` functions and can be used to adjust contributions to filters.

The data used in this tutorial were provided by the Allen Institute for Cell Science. They were downsampled by a factor of 4 in the `row` and `column` dimensions to reduce their size and, hence, computational time. The spacing information was reported by the microscope used to image the cells.

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.art3d import Poly3DCollection
import numpy as np

import plotly
import plotly.express as px
from skimage import (
    exposure, util
)
from skimage.data import cells3d
```

Load and display 3D images

```
data = util.img_as_float(cells3d()[:, :, :, :]) # grab just the nuclei

print(f'shape: {data.shape}')
print(f'dtype: {data.dtype}')
print(f'range: ({data.min()}, {data.max()})')

# Report spacing from microscope
original_spacing = np.array([0.2900000, 0.0650000, 0.0650000])

# Account for downsampling of slices by 4
rescaled_spacing = original_spacing * [1, 4, 4]

# Normalize spacing so that pixels are a distance of 1 apart
spacing = rescaled_spacing / rescaled_spacing[2]

print(f'microscope spacing: {original_spacing}\n')
print(f'rescaled spacing: {rescaled_spacing} (after downsampling)\n')
print(f'normalized spacing: {spacing}\n')
```

```
shape: (60, 256, 256)
dtype: float64
range: (0.0, 1.0)
microscope spacing: [0.29  0.065  0.065]

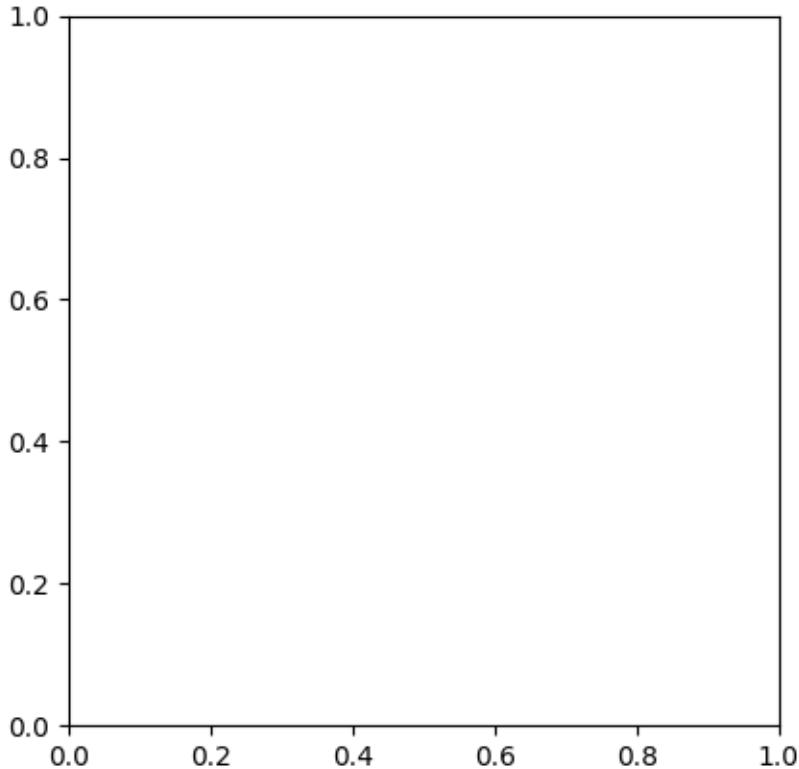
rescaled spacing: [0.29  0.26  0.26] (after downsampling)

normalized spacing: [1.11538462 1.           1.           ]
```

Let us try and visualize our 3D image. Unfortunately, many image viewers, such as matplotlib's `imshow`, are only

capable of displaying 2D data. We can see that they raise an error when we try to view 3D data:

```
try:
    fig, ax = plt.subplots()
    ax.imshow(data, cmap='gray')
except TypeError as e:
    print(str(e))
```



```
Invalid shape (60, 256, 256) for image data
```

The `imshow` function can only display grayscale and RGB(A) 2D images. We can thus use it to visualize 2D planes. By fixing one axis, we can observe three different views of the image.

```
def show_plane(ax, plane, cmap="gray", title=None):
    ax.imshow(plane, cmap=cmap)
    ax.axis("off")

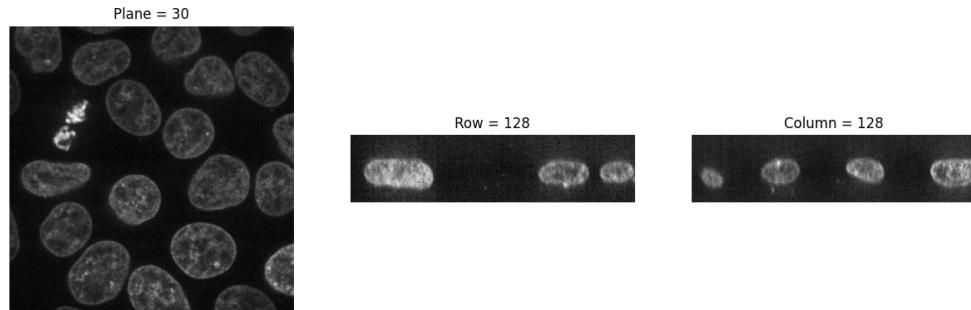
    if title:
        ax.set_title(title)

(n_plane, n_row, n_col) = data.shape
_, (a, b, c) = plt.subplots(ncols=3, figsize=(15, 5))
```

(continues on next page)

(continued from previous page)

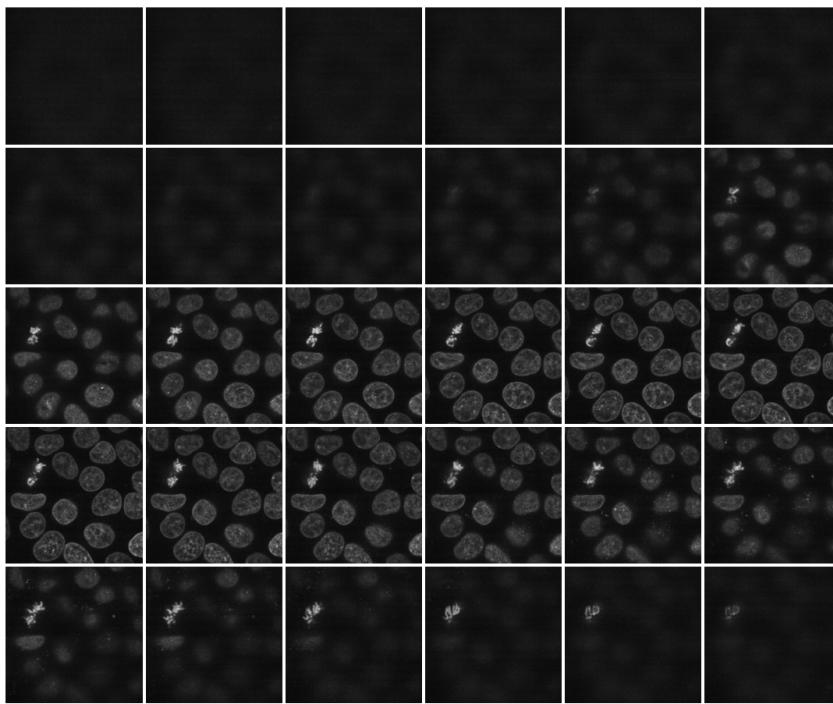
```
show_plane(a, data[n_plane // 2], title=f'Plane = {n_plane // 2}')
show_plane(b, data[:, n_row // 2, :], title=f'Row = {n_row // 2}')
show_plane(c, data[:, :, n_col // 2], title=f'Column = {n_col // 2}')
```



As hinted before, a three-dimensional image can be viewed as a series of two-dimensional planes. Let us write a helper function, *display*, to create a montage of several planes. By default, every other plane is displayed.

```
def display(im3d, cmap='gray', step=2):
    data_montage = util.montage(im3d[::step], padding_width=4, fill=np.nan)
    _, ax = plt.subplots(figsize=(16, 14))
    ax.imshow(data_montage, cmap=cmap)
    ax.set_axis_off()

display(data)
```



Alternatively, we can explore these planes (slices) interactively using Jupyter widgets. Let the user select which slice to display and show the position of this slice in the 3D dataset. Note that you cannot see the Jupyter widget at work in a static HTML page, as is the case in the online version of this example. For the following piece of code to work, you need a Jupyter kernel running either locally or in the cloud: see the bottom of this page to either download the Jupyter notebook and run it on your computer, or open it directly in Binder. On top of an active kernel, you need a web browser: running the code in pure Python will not work either.

```
def slice_in_3D(ax, i):
    # From https://stackoverflow.com/questions/44881885/python-draw-3d-cube
    Z = np.array([[0, 0, 0],
                  [1, 0, 0],
                  [1, 1, 0],
                  [0, 1, 0],
                  [0, 0, 1],
                  [1, 0, 1],
                  [1, 1, 1],
                  [0, 1, 1]])
```

(continues on next page)

(continued from previous page)

```

Z = Z * data.shape
r = [-1, 1]
X, Y = np.meshgrid(r, r)

# Plot vertices
ax.scatter3D(Z[:, 0], Z[:, 1], Z[:, 2])

# List sides' polygons of figure
verts = [[Z[0], Z[1], Z[2], Z[3]],
          [Z[4], Z[5], Z[6], Z[7]],
          [Z[0], Z[1], Z[5], Z[4]],
          [Z[2], Z[3], Z[7], Z[6]],
          [Z[1], Z[2], Z[6], Z[5]],
          [Z[4], Z[7], Z[3], Z[0]],
          [Z[2], Z[3], Z[7], Z[6]]]

# Plot sides
ax.add_collection3d(
    Poly3DCollection(
        verts,
        facecolors=(0, 1, 1, 0.25),
        linewidths=1,
        edgecolors="darkblue"
    )
)

verts = np.array([[[0, 0, 0],
                  [0, 0, 1],
                  [0, 1, 1],
                  [0, 1, 0]]])
verts = verts * (60, 256, 256)
verts += [i, 0, 0]

ax.add_collection3d(
    Poly3DCollection(
        verts,
        facecolors="magenta",
        linewidths=1,
        edgecolors="black"
    )
)

ax.set_xlabel("plane")
ax.set_xlim(0, 100)
ax.set_ylabel("row")
ax.set_zlabel("col")

# Autoscale plot axes
scaling = np.array([getattr(ax,
                           f'get_{dim}lim')() for dim in "xyz"])
ax.auto_scale_xyz(* [[np.min(scaling), np.max(scaling)]] * 3)

```

(continues on next page)

(continued from previous page)

```
def explore_slices(data, cmap="gray"):
    from ipywidgets import interact
    N = len(data)

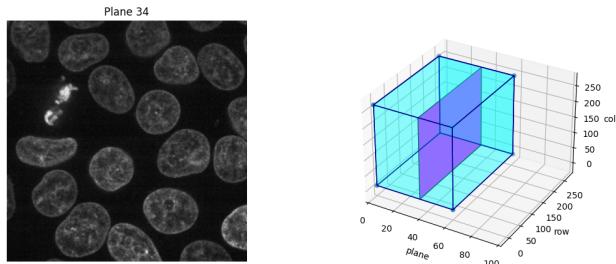
    @interact(plane=(0, N - 1))
    def display_slice(plane=34):
        fig, ax = plt.subplots(figsize=(20, 5))

        ax_3D = fig.add_subplot(133, projection="3d")
        show_plane(ax, data[plane], title=f'Plane {plane}', cmap=cmap)
        slice_in_3D(ax_3D, plane)

        plt.show()

    return display_slice

explore_slices(data)
```



```
interactive(children=(IntSlider(value=34, description='plane', max=59), Output()), _dom_
           ↴classes=('widget-interact',))
<function explore_slices.<locals>.display_slice at 0x7ff4879dd8b0>
```

Adjust exposure

Scikit-image's *exposure* module contains a number of functions for adjusting image contrast. These functions operate on pixel values. Generally, image dimensionality or pixel spacing doesn't need to be considered. In local exposure correction, though, one might want to adjust the window size to ensure equal size in *real* coordinates along each axis.

Gamma correction brightens or darkens an image. A power-law transform, where *gamma* denotes the power-law exponent, is applied to each pixel in the image: *gamma* < 1 will brighten an image, while *gamma* > 1 will darken an image.

```
def plot_hist(ax, data, title=None):
    # Helper function for plotting histograms
    ax.hist(data.ravel(), bins=256)
    ax.ticklabel_format(axis="y", style="scientific", scilimits=(0, 0))
```

(continues on next page)

(continued from previous page)

```
if title:
    ax.set_title(title)

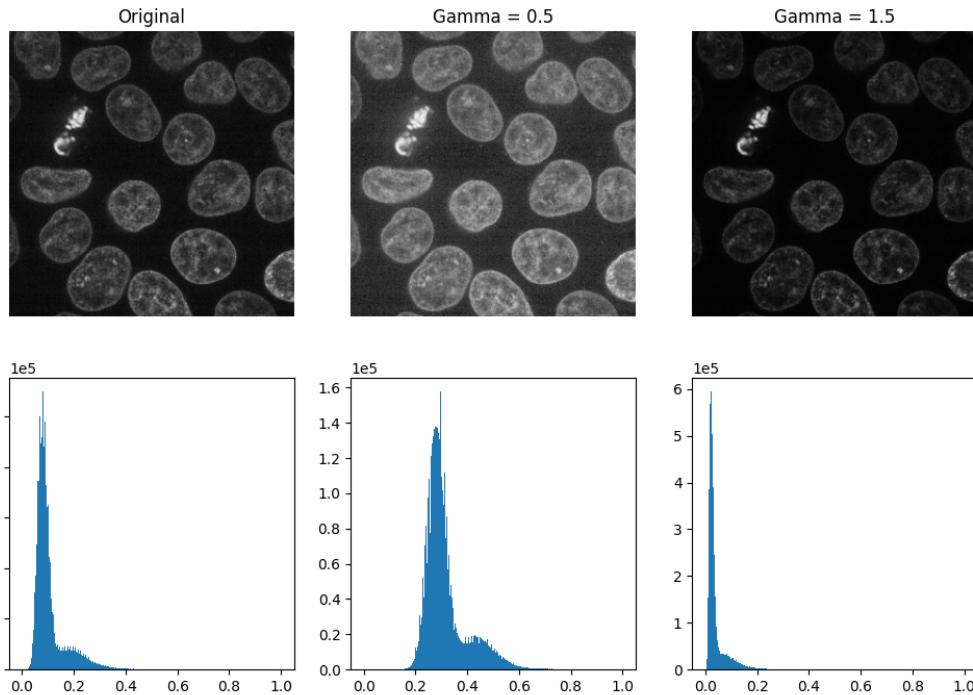
gamma_low_val = 0.5
gamma_low = exposure.adjust_gamma(data, gamma=gamma_low_val)

gamma_high_val = 1.5
gamma_high = exposure.adjust_gamma(data, gamma=gamma_high_val)

_, ((a, b, c), (d, e, f)) = plt.subplots(nrows=2, ncols=3, figsize=(12, 8))

show_plane(a, data[32], title='Original')
show_plane(b, gamma_low[32], title=f'Gamma = {gamma_low_val}')
show_plane(c, gamma_high[32], title=f'Gamma = {gamma_high_val}')

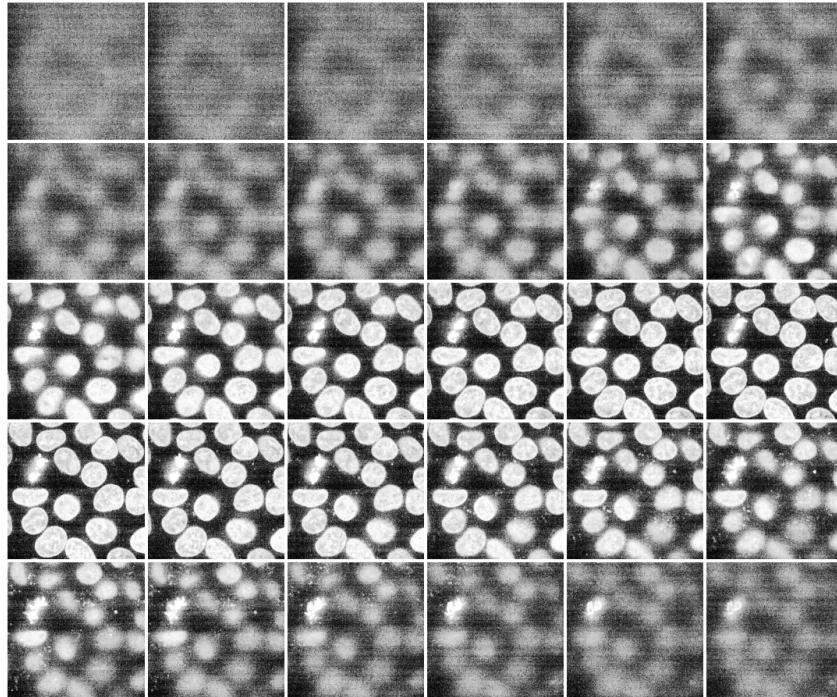
plot_hist(d, data)
plot_hist(e, gamma_low)
plot_hist(f, gamma_high)
```



Histogram equalization improves contrast in an image by redistributing pixel intensities. The most common pixel intensities get spread out, increasing contrast in low-contrast areas. One downside of this approach is that it may enhance background noise.

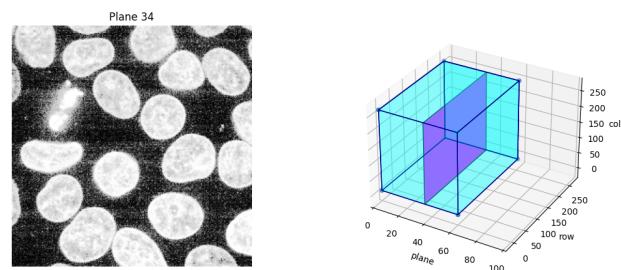
```
equalized_data = exposure.equalize_hist(data)

display(equalized_data)
```



As before, if we have a Jupyter kernel running, we can explore the above slices interactively.

```
explore_slices(equalized_data)
```



```
interactive(children=(IntSlider(value=34, description='plane', max=59), Output()), _dom_
˓→classes=['widget-interact',))

<function explore_slices.<locals>.display_slice at 0x7ff484a71040>
```

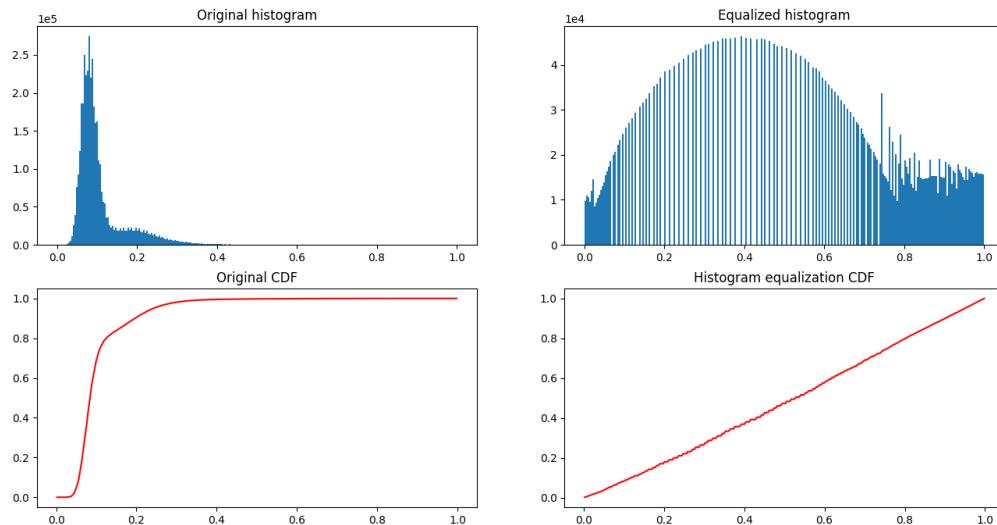
Let us now plot the image histogram before and after histogram equalization. Below, we plot the respective cumulative distribution functions (CDF).

```
_, ((a, b), (c, d)) = plt.subplots(nrows=2, ncols=2, figsize=(16, 8))

plot_hist(a, data, title="Original histogram")
plot_hist(b, equalized_data, title="Equalized histogram")

cdf, bins = exposure.cumulative_distribution(data.ravel())
c.plot(bins, cdf, "r")
c.set_title("Original CDF")

cdf, bins = exposure.cumulative_distribution(equalized_data.ravel())
d.plot(bins, cdf, "r")
d.set_title("Histogram equalization CDF")
```



```
Text(0.5, 1.0, 'Histogram equalization CDF')
```

Most experimental images are affected by salt and pepper noise. A few bright artifacts can decrease the relative intensity of the pixels of interest. A simple way to improve contrast is to clip the pixel values on the lowest and highest extremes. Clipping the darkest and brightest 0.5% of pixels will increase the overall contrast of the image.

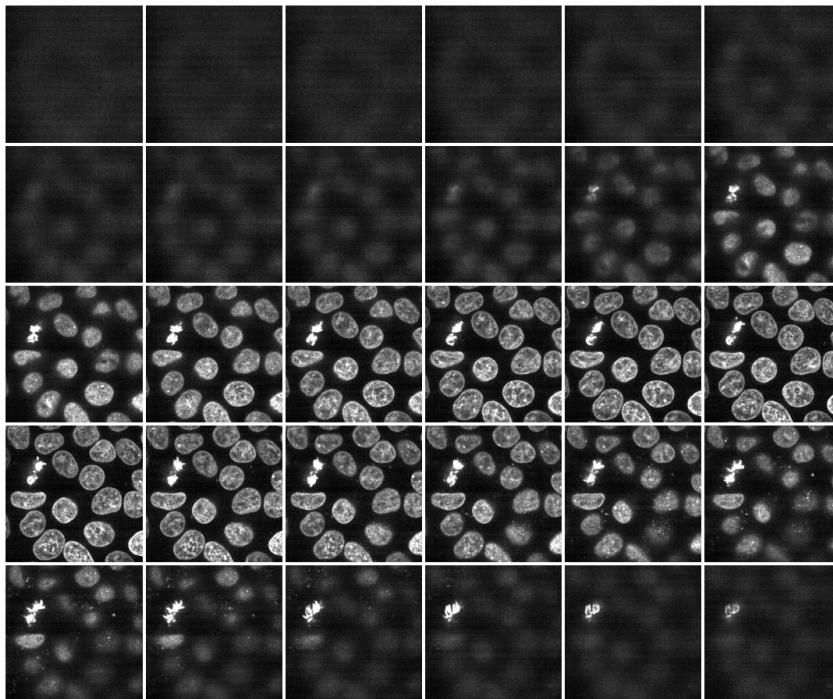
```
vmin, vmax = np.percentile(data, q=(0.5, 99.5))

clipped_data = exposure.rescale_intensity(
    data,
    in_range=(vmin, vmax),
```

(continues on next page)

(continued from previous page)

```
    out_range=np.float32  
)  
  
display(clipped_data)
```



Alternatively, we can explore these planes (slices) interactively using [Plotly Express](#). Note that this works in a static HTML page!

```
fig = px.imshow(data, animation_frame=0, binary_string=True)  
fig.update_xaxes(showticklabels=False)  
fig.update_yaxes(showticklabels=False)  
fig.update_layout(  
    autosize=False,  
    width=500,  
    height=500,  
    coloraxis.showscale=False  
)
```

(continues on next page)

(continued from previous page)

```
# Drop animation buttons
fig['layout'].pop('updatemenus')
plotly.io.show(fig)

plt.show()
```

Total running time of the script: (0 minutes 6.109 seconds)

Rank filters

Rank filters are non-linear filters using local gray-level ordering to compute the filtered value. This ensemble of filters share a common base: the local gray-level histogram is computed on the neighborhood of a pixel (defined by a 2D structuring element). If the filtered value is taken as the middle value of the histogram, we get the classical median filter.

Rank filters can be used for several purposes, such as:

- image quality enhancement, e.g., image smoothing, sharpening
- image pre-processing, e.g., noise reduction, contrast enhancement
- feature extraction, e.g., border detection, isolated point detection
- image post-processing, e.g., small object removal, object grouping, contour smoothing

Some well-known filters (e.g., morphological dilation and morphological erosion) are specific cases of rank filters¹.

In this example, we will see how to filter a gray-level image using some of the linear and non-linear filters available in skimage. We use the camera image from `skimage.data` for all comparisons.

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

from skimage.util import img_as_ubyte
from skimage import data
from skimage.exposure import histogram

noisy_image = img_as_ubyte(data.camera())
hist, hist_centers = histogram(noisy_image)

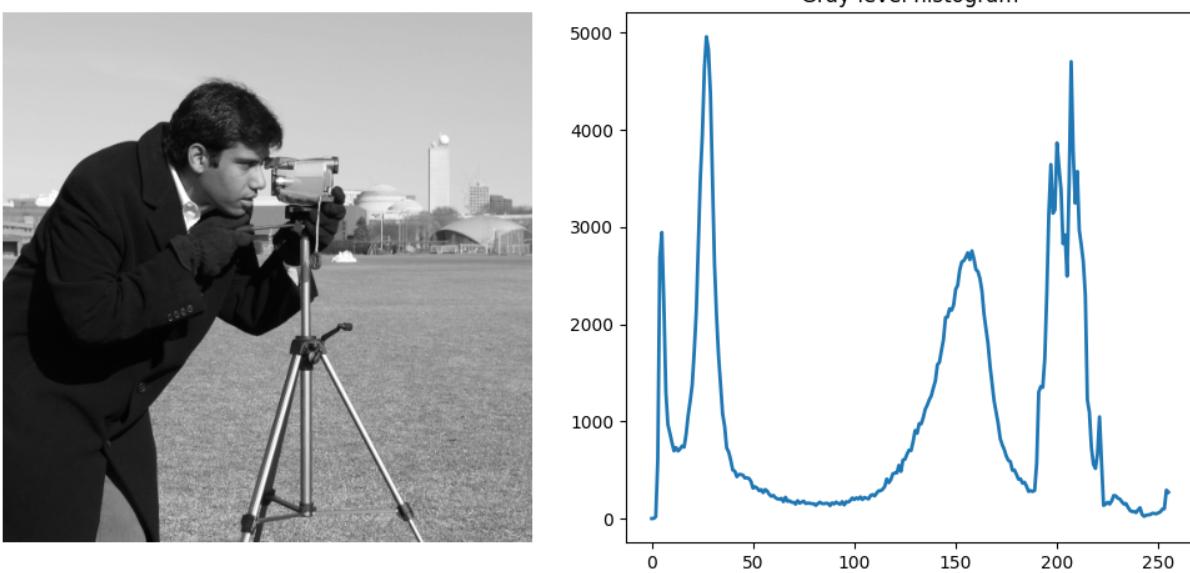
fig, ax = plt.subplots(ncols=2, figsize=(10, 5))

ax[0].imshow(noisy_image, cmap=plt.cm.gray)
ax[0].axis('off')

ax[1].plot(hist_centers, hist, lw=2)
ax[1].set_title('Gray-level histogram')

plt.tight_layout()
```

¹ Pierre Soille, On morphological operators based on rank filters, Pattern Recognition 35 (2002) 527-535, DOI:10.1016/S0031-3203(01)00047-4



Noise removal

Some noise is added to the image: 1% of pixels are randomly set to 255, 1% are randomly set to 0. The **median** filter is applied to remove the noise.

```
from skimage.filters.rank import median
from skimage.morphology import disk, ball

rng = np.random.default_rng()
noise = rng.random(noisy_image.shape)
noisy_image = img_as_ubyte(data.camera())
noisy_image[noise > 0.99] = 255
noisy_image[noise < 0.01] = 0

fig, axes = plt.subplots(2, 2, figsize=(10, 10), sharex=True, sharey=True)
ax = axes.ravel()

ax[0].imshow(noisy_image, vmin=0, vmax=255, cmap=plt.cm.gray)
ax[0].set_title('Noisy image')

ax[1].imshow(median(noisy_image, disk(1)), vmin=0, vmax=255, cmap=plt.cm.gray)
ax[1].set_title('Median $r=1$')

ax[2].imshow(median(noisy_image, disk(5)), vmin=0, vmax=255, cmap=plt.cm.gray)
ax[2].set_title('Median $r=5$')

ax[3].imshow(median(noisy_image, disk(20)), vmin=0, vmax=255, cmap=plt.cm.gray)
ax[3].set_title('Median $r=20$')

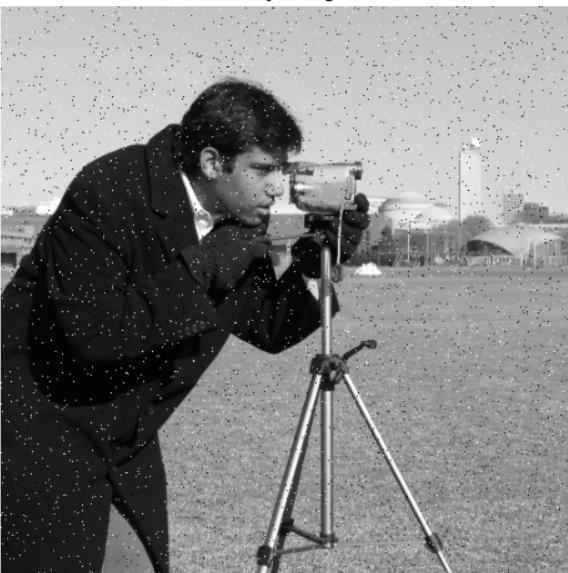
for a in ax:
    a.axis('off')
```

(continues on next page)

(continued from previous page)

```
plt.tight_layout()
```

Noisy image

Median $r = 1$ Median $r = 5$ Median $r = 20$ 

The added noise is efficiently removed, as the image defaults are small (1-pixel wide), a small filter radius is sufficient. As the radius increases, objects with bigger sizes get filtered as well, such as the camera tripod. The median filter is often used for noise removal because it preserves borders. For example, consider noise which is located only on a few pixels in the entire image, as is the case with salt-and-pepper noise²: the median filter will ignore the noisy pixels, for they will appear as outliers; thus, it will not change significantly the median of a group of local pixels, in contrast to what a moving average filter would do.

² https://en.wikipedia.org/wiki/Salt-and-pepper_noise

Image smoothing

The example hereunder shows how a local **mean** filter smooths the camera man image.

```
from skimage.filters.rank import mean

loc_mean = mean(noisy_image, disk(10))

fig, ax = plt.subplots(ncols=2, figsize=(10, 5), sharex=True, sharey=True)

ax[0].imshow(noisy_image, vmin=0, vmax=255, cmap=plt.cm.gray)
ax[0].set_title('Original')

ax[1].imshow(loc_mean, vmin=0, vmax=255, cmap=plt.cm.gray)
ax[1].set_title('Local mean $r=10$')

for a in ax:
    a.axis('off')

plt.tight_layout()
```



One may be interested in smoothing an image while preserving important borders (median filters already achieved this). Here, we use the **bilateral** filter that restricts the local neighborhood to pixels with gray levels similar to the central one.

Note: A different implementation is available for color images in `skimage.restoration.denoise_bilateral()`.

```
from skimage.filters.rank import mean_bilateral

noisy_image = img_as_ubyte(data.camera())
```

(continues on next page)

(continued from previous page)

```
bilat = mean_bilateral(noisy_image.astype(np.uint16), disk(20), s0=10, s1=10)

fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(10, 10),
                        sharex='row', sharey='row')
ax = axes.ravel()

ax[0].imshow(noisy_image, cmap=plt.cm.gray)
ax[0].set_title('Original')

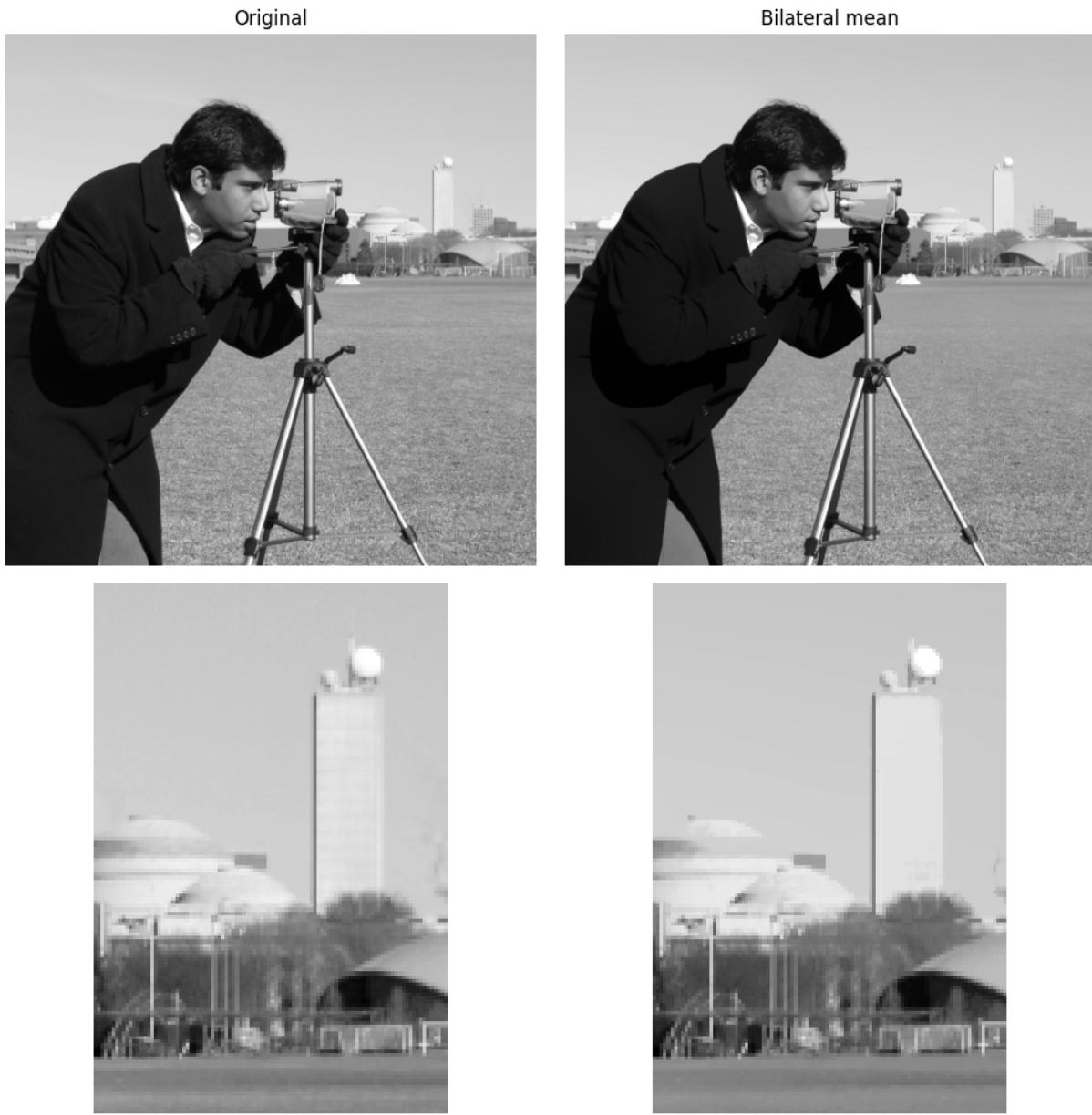
ax[1].imshow(bilat, cmap=plt.cm.gray)
ax[1].set_title('Bilateral mean')

ax[2].imshow(noisy_image[100:250, 350:450], cmap=plt.cm.gray)

ax[3].imshow(bilat[100:250, 350:450], cmap=plt.cm.gray)

for a in ax:
    a.axis('off')

plt.tight_layout()
```



One can see that the large continuous part of the image (e.g. sky) is smoothed whereas other details are preserved.

Contrast enhancement

We compare here how the global histogram equalization is applied locally.

The equalized image³ has a roughly linear cumulative distribution function for each pixel neighborhood. The local version⁴ of histogram equalization emphasizes every local gray-level variation.

```
from skimage import exposure
from skimage.filters import rank
```

(continues on next page)

³ https://en.wikipedia.org/wiki/Histogram_equalization

⁴ https://en.wikipedia.org/wiki/Adaptive_histogram_equalization

(continued from previous page)

```
noisy_image = img_as_ubyte(data.camera())

# equalize globally and locally
glob = exposure.equalize_hist(noisy_image) * 255
loc = rank.equalize(noisy_image, disk(20))

# extract histogram for each image
hist = np.histogram(noisy_image, bins=np.arange(0, 256))
glob_hist = np.histogram(glob, bins=np.arange(0, 256))
loc_hist = np.histogram(loc, bins=np.arange(0, 256))

fig, axes = plt.subplots(nrows=3, ncols=2, figsize=(12, 12))
ax = axes.ravel()

ax[0].imshow(noisy_image, cmap=plt.cm.gray)
ax[0].axis('off')

ax[1].plot(hist[1][:-1], hist[0], lw=2)
ax[1].set_title('Histogram of gray values')

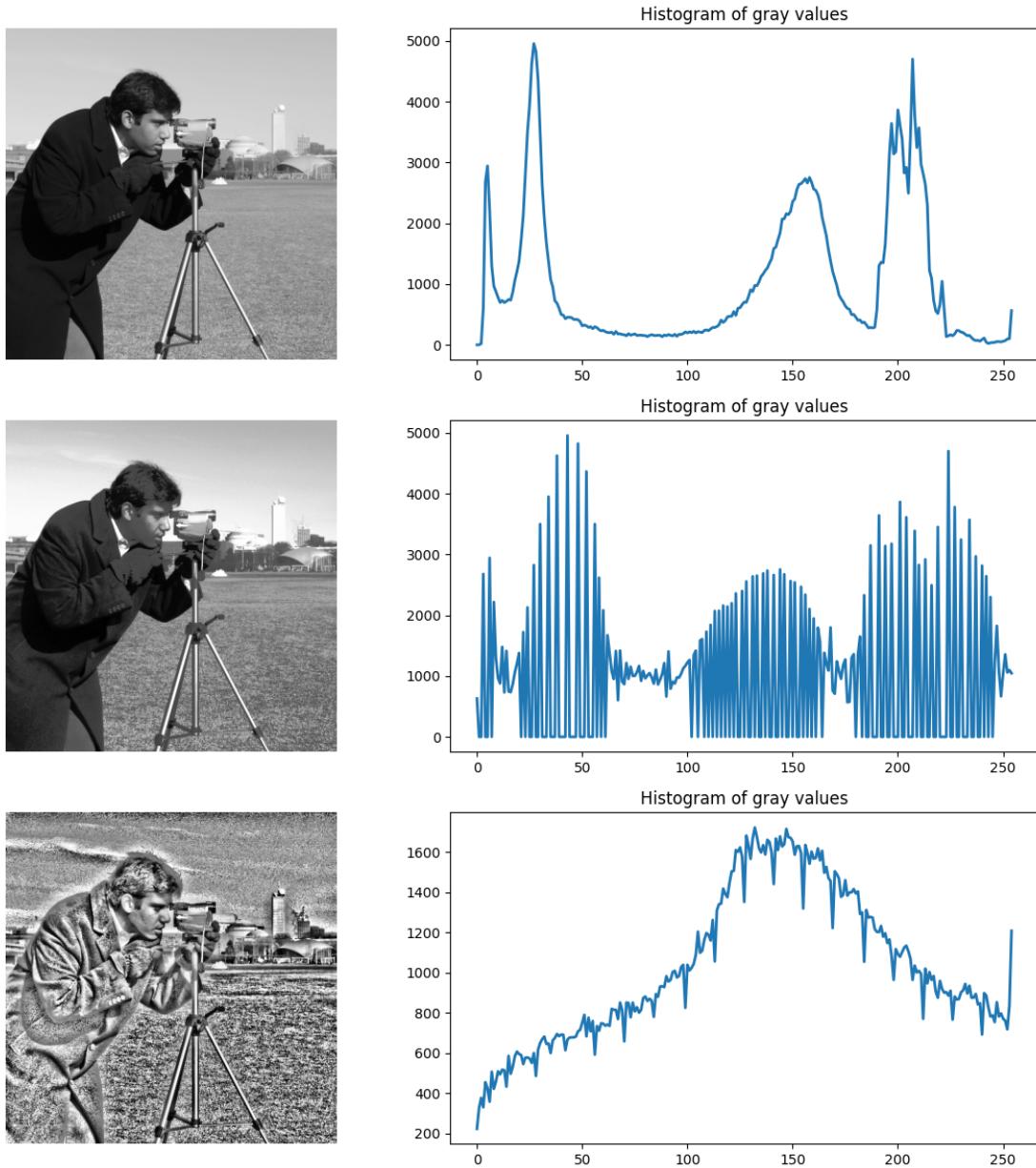
ax[2].imshow(glob, cmap=plt.cm.gray)
ax[2].axis('off')

ax[3].plot(glob_hist[1][:-1], glob_hist[0], lw=2)
ax[3].set_title('Histogram of gray values')

ax[4].imshow(loc, cmap=plt.cm.gray)
ax[4].axis('off')

ax[5].plot(loc_hist[1][:-1], loc_hist[0], lw=2)
ax[5].set_title('Histogram of gray values')

plt.tight_layout()
```



Another way to maximize the number of gray-levels used for an image is to apply a local auto-leveling, i.e. the gray-value of a pixel is proportionally remapped between local minimum and local maximum.

The following example shows how local auto-level enhances the camara man picture.

```
from skimage.filters.rank import autolevel

noisy_image = img_as_ubyte(data.camera())

auto = autolevel(noisy_image.astype(np.uint16), disk(20))

fig, ax = plt.subplots(ncols=2, figsize=(10, 5), sharex=True, sharey=True)
ax[0].imshow(noisy_image, cmap=plt.cm.gray)
```

(continues on next page)

(continued from previous page)

```

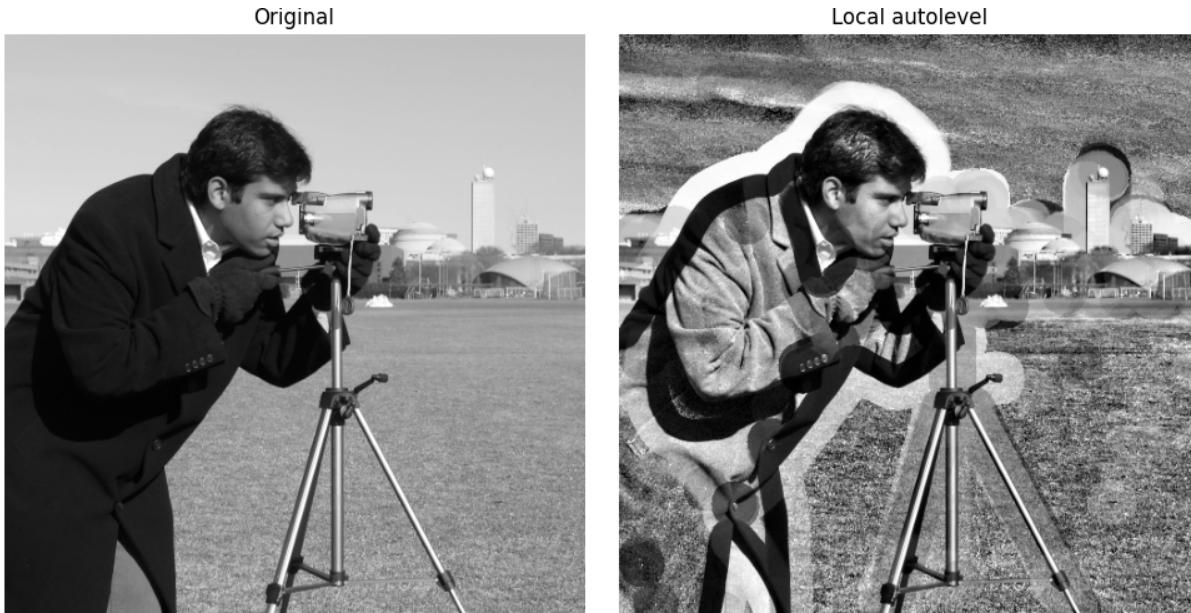
ax[0].set_title('Original')

ax[1].imshow(auto, cmap=plt.cm.gray)
ax[1].set_title('Local autolevel')

for a in ax:
    a.axis('off')

plt.tight_layout()

```



This filter is very sensitive to local outliers. One can moderate this using the percentile version of the auto-level filter which uses given percentiles (one inferior, one superior) in place of local minimum and maximum. The example below illustrates how the percentile parameters influence the local auto-level result.

```

from skimage.filters.rank import autolevel_percentile

image = data.camera()

footprint = disk(20)
loc_autolevel = autolevel(image, footprint=footprint)
loc_perc_autolevel0 = autolevel_percentile(
    image, footprint=footprint, p0=.01, p1=.99
)
loc_perc_autolevel1 = autolevel_percentile(
    image, footprint=footprint, p0=.05, p1=.95
)
loc_perc_autolevel2 = autolevel_percentile(
    image, footprint=footprint, p0=.1, p1=.9
)
loc_perc_autolevel3 = autolevel_percentile(
    image, footprint=footprint, p0=.15, p1=.85
)

```

(continues on next page)

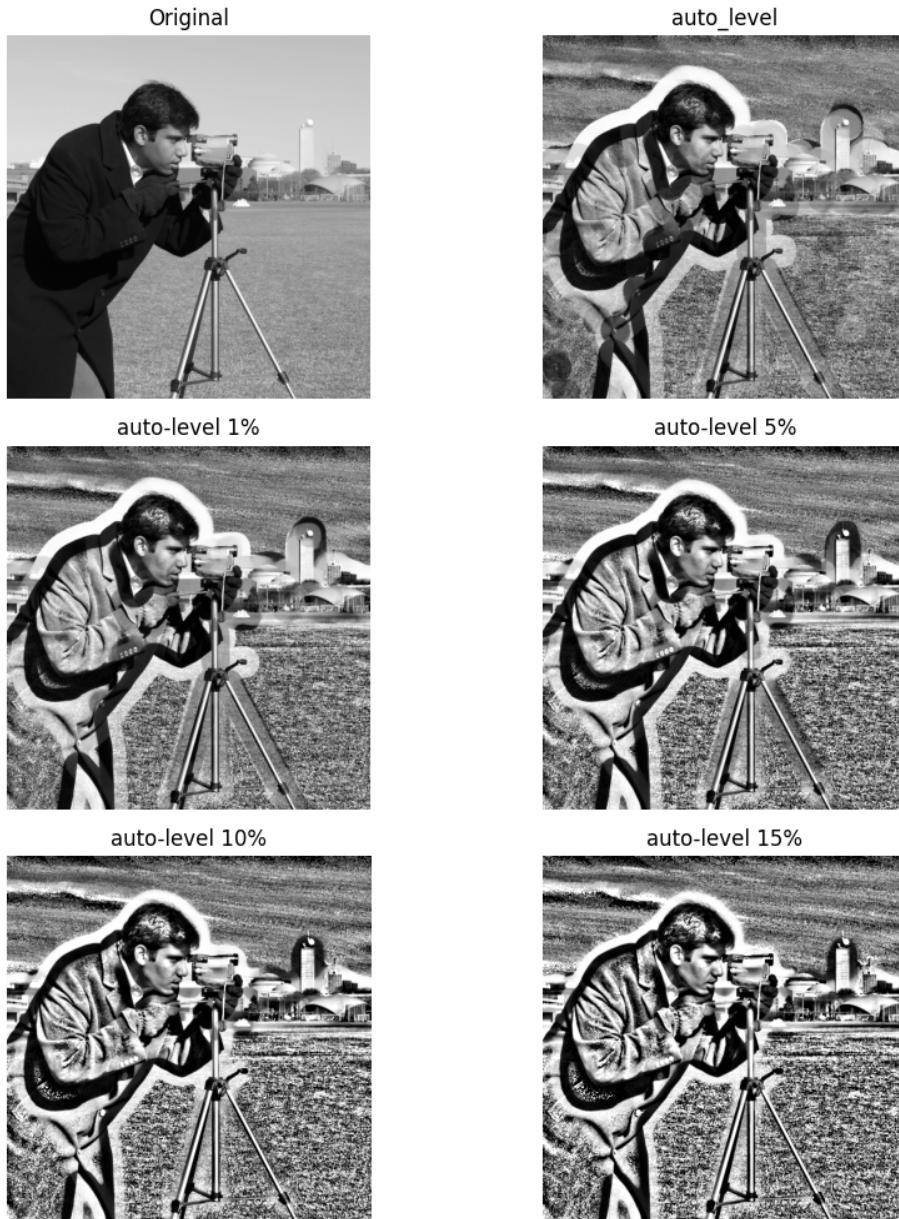
(continued from previous page)

```
fig, axes = plt.subplots(nrows=3, ncols=2, figsize=(10, 10),
                       sharex=True, sharey=True)
ax = axes.ravel()

title_list = ['Original',
              'auto_level',
              'auto-level 1%',
              'auto-level 5%',
              'auto-level 10%',
              'auto-level 15%']
image_list = [image,
              loc_autolevel,
              loc_perc_autolevel0,
              loc_perc_autolevel1,
              loc_perc_autolevel2,
              loc_perc_autolevel3]

for i in range(0, len(image_list)):
    ax[i].imshow(image_list[i], cmap=plt.cm.gray, vmin=0, vmax=255)
    ax[i].set_title(title_list[i])
    ax[i].axis('off')

plt.tight_layout()
```



The morphological contrast enhancement filter replaces the central pixel by the local maximum if the original pixel value is closest to local maximum, otherwise by the minimum local.

```
from skimage.filters.rank import enhance_contrast

noisy_image = img_as_ubyte(data.camera())

enh = enhance_contrast(noisy_image, disk(5))

fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(10, 10),
                       sharex='row', sharey='row')
ax = axes.ravel()
```

(continues on next page)

(continued from previous page)

```
ax[0].imshow(noisy_image, cmap=plt.cm.gray)
ax[0].set_title('Original')

ax[1].imshow(enh, cmap=plt.cm.gray)
ax[1].set_title('Local morphological contrast enhancement')

ax[2].imshow(noisy_image[100:250, 350:450], cmap=plt.cm.gray)

ax[3].imshow(enh[100:250, 350:450], cmap=plt.cm.gray)

for a in ax:
    a.axis('off')

plt.tight_layout()
```



The percentile version of the local morphological contrast enhancement uses percentile $p0$ and $p1$ instead of the local minimum and maximum.

```
from skimage.filters.rank import enhance_contrast_percentile

noisy_image = img_as_ubyte(data.camera())

penh = enhance_contrast_percentile(noisy_image, disk(5), p0=.1, p1=.9)

fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(10, 10),
                       sharex='row', sharey='row')
ax = axes.ravel()
```

(continues on next page)

(continued from previous page)

```
ax[0].imshow(noisy_image, cmap=plt.cm.gray)
ax[0].set_title('Original')

ax[1].imshow(penh, cmap=plt.cm.gray)
ax[1].set_title('Local percentile morphological\n contrast enhancement')

ax[2].imshow(noisy_image[100:250, 350:450], cmap=plt.cm.gray)

ax[3].imshow(penh[100:250, 350:450], cmap=plt.cm.gray)

for a in ax:
    a.axis('off')

plt.tight_layout()
```

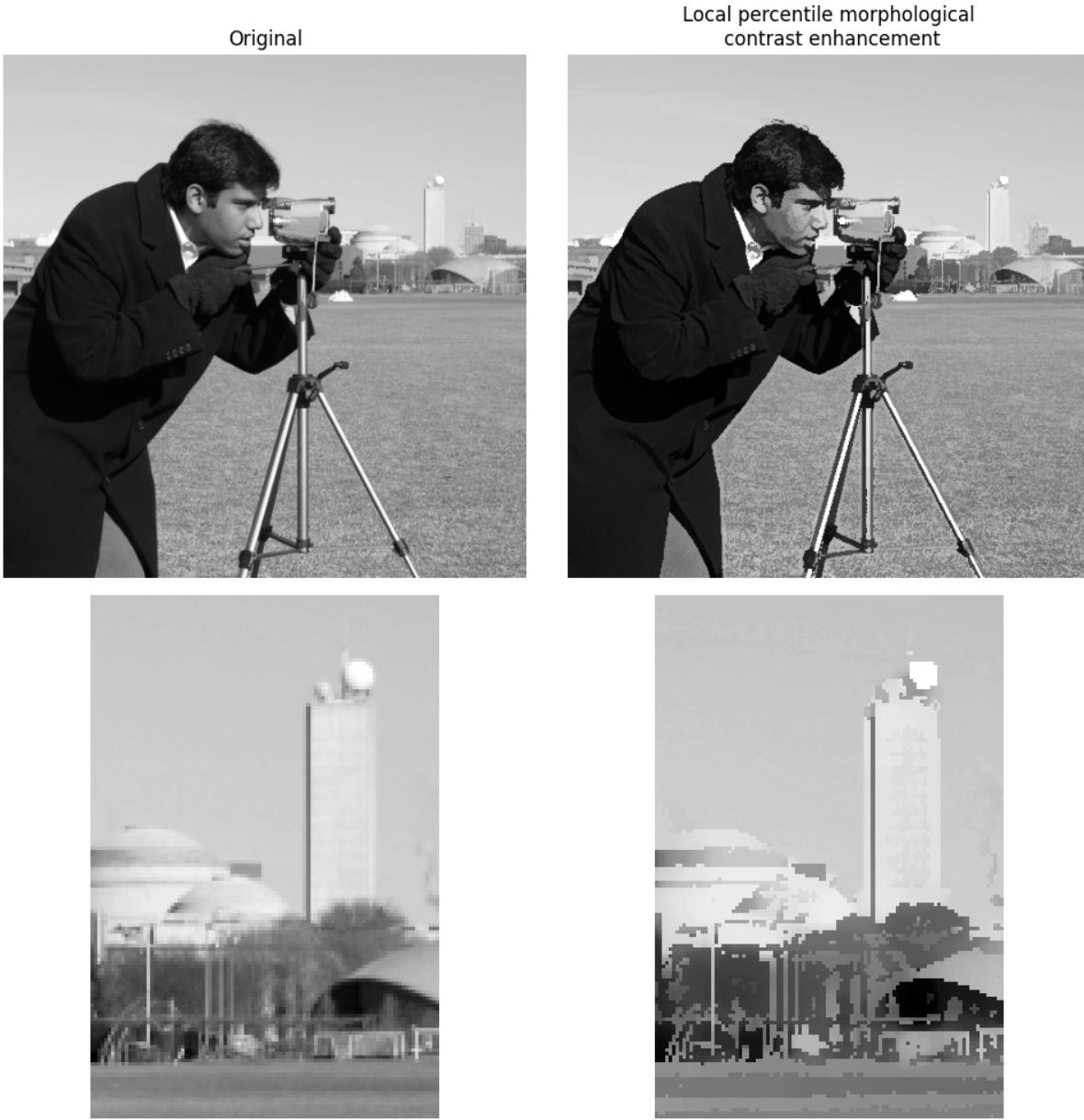


Image threshold

The Otsu threshold method⁵ can be applied locally using the local gray-level distribution. In the example below, for each pixel, an “optimal” threshold is determined by maximizing the variance between two classes of pixels of the local neighborhood defined by a structuring element.

These algorithms can be used on both 2D and 3D images.

The example compares local thresholding with global thresholding, which is provided by `skimage.filters.threshold_otsu()`. Note that the former is much slower than the latter.

⁵ https://en.wikipedia.org/wiki/Otsu's_method

```
from skimage.filters.rank import otsu
from skimage.filters import threshold_otsu
from skimage import exposure

p8 = data.page()

radius = 10
footprint = disk(radius)

# t_loc_otsu is an image
t_loc_otsu = otsu(p8, footprint)
loc_otsu = p8 >= t_loc_otsu

# t_glob_otsu is a scalar
t_glob_otsu = threshold_otsu(p8)
glob_otsu = p8 >= t_glob_otsu

fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(12, 12),
                        sharex=True, sharey=True)
ax = axes.ravel()

fig.colorbar(ax[0].imshow(p8, cmap=plt.cm.gray), ax=ax[0])
ax[0].set_title('Original')

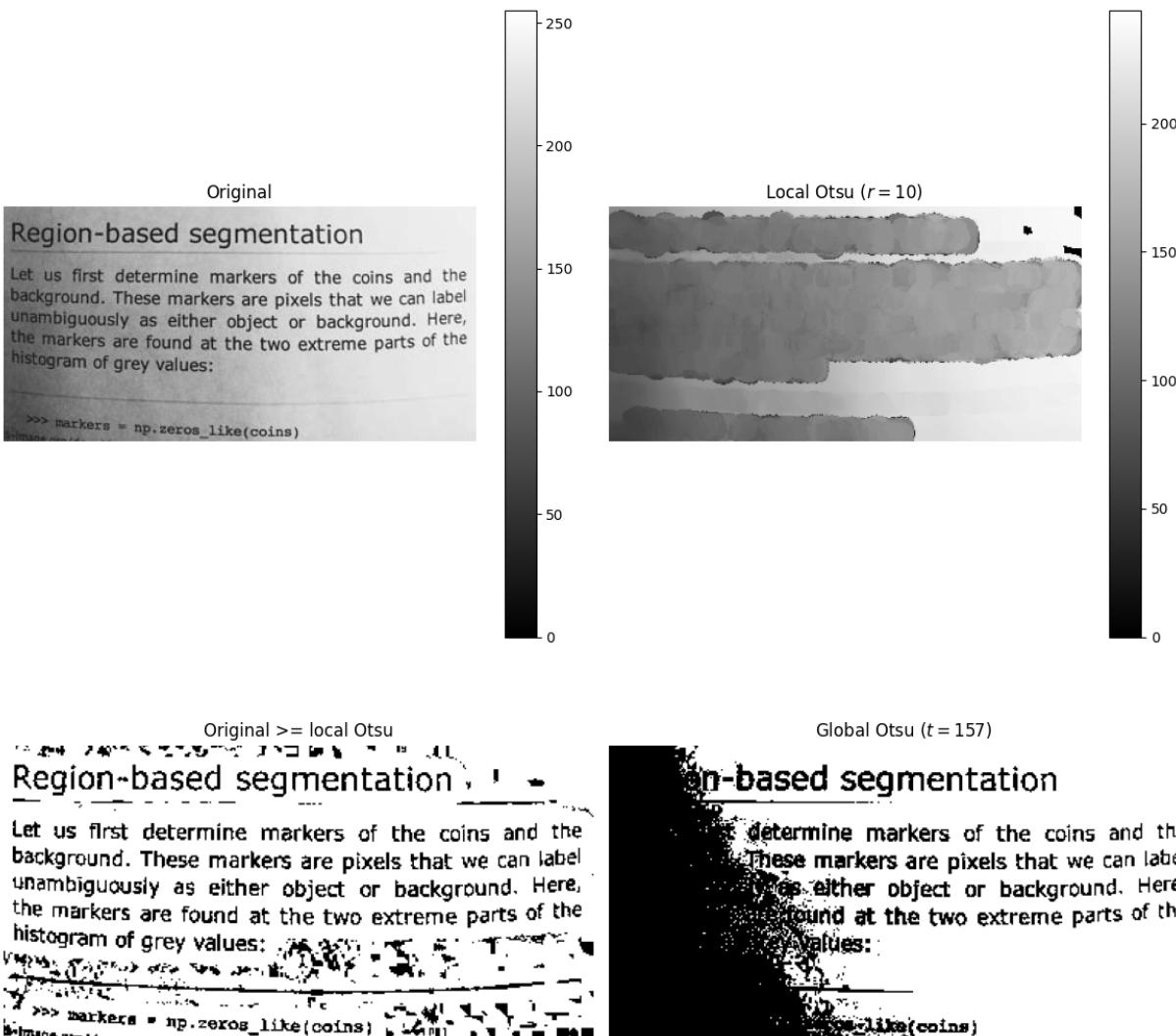
fig.colorbar(ax[1].imshow(t_loc_otsu, cmap=plt.cm.gray), ax=ax[1])
ax[1].set_title(f'Local Otsu ($r={radius}$)')

ax[2].imshow(p8 >= t_loc_otsu, cmap=plt.cm.gray)
ax[2].set_title('Original >= local Otsu')

ax[3].imshow(glob_otsu, cmap=plt.cm.gray)
ax[3].set_title(f'Global Otsu ($t={t_glob_otsu}$)')

for a in ax:
    a.axis('off')

plt.tight_layout()
```



The example below performs the same comparison, using a 3D image this time.

```

brain = exposure.rescale_intensity(data.brain().astype(float))

radius = 5
neighborhood = ball(radius)

# t_loc_otsu is an image
t_loc_otsu = rank.otsu(brain, neighborhood)
loc_otsu = brain >= t_loc_otsu

# t_glob_otsu is a scalar
t_glob_otsu = threshold_otsu(brain)

```

(continues on next page)

(continued from previous page)

```
glob_otsu = brain >= t_glob_otsu

fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(12, 12),
                        sharex=True, sharey=True)
ax = axes.ravel()

slice_index = 3

fig.colorbar(ax[0].imshow(brain[slice_index], cmap=plt.cm.gray), ax=ax[0])
ax[0].set_title('Original')

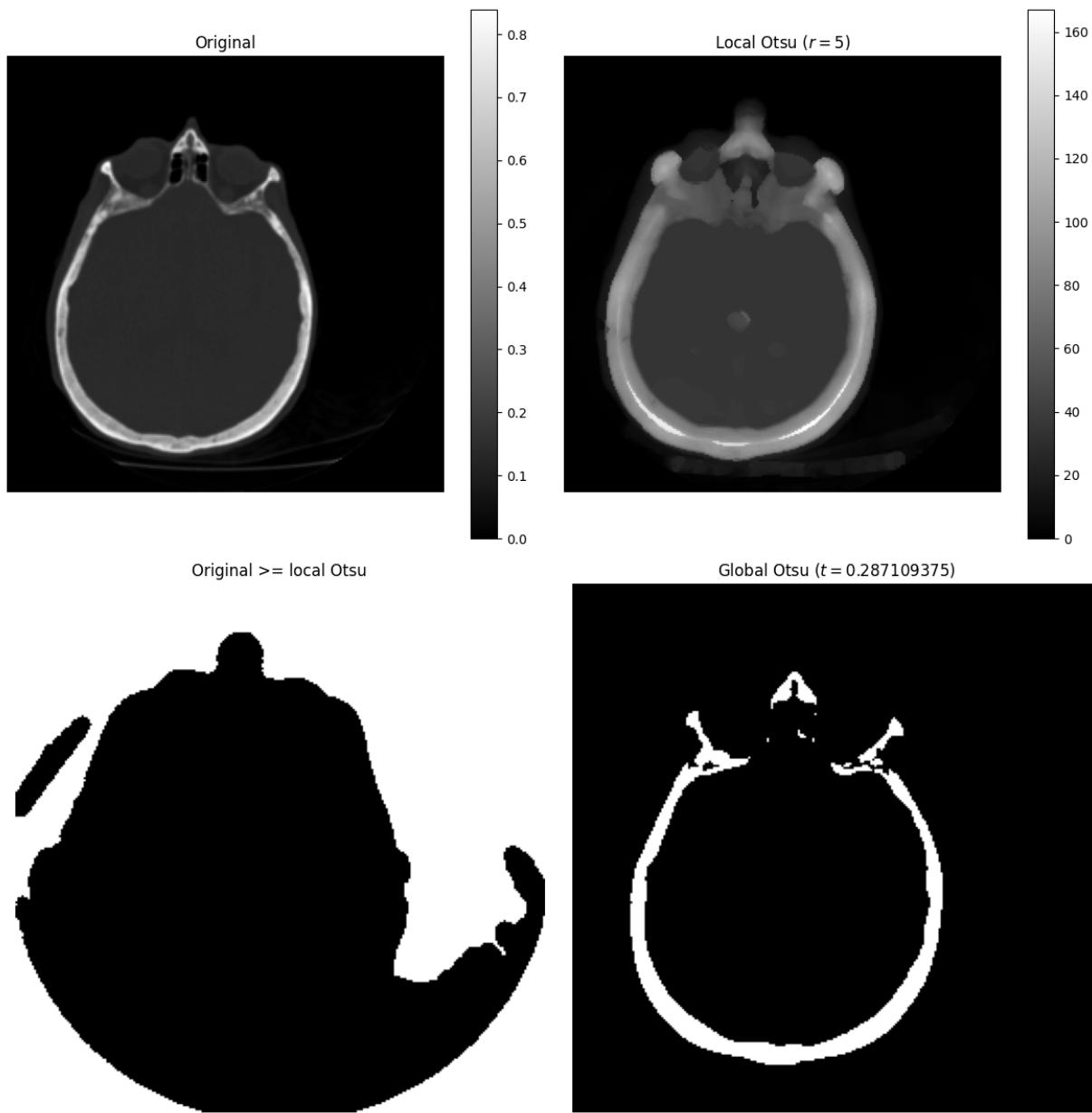
fig.colorbar(ax[1].imshow(t_loc_otsu[slice_index], cmap=plt.cm.gray), ax=ax[1])
ax[1].set_title(f'Local Otsu (r={radius})')

ax[2].imshow(brain[slice_index] >= t_loc_otsu[slice_index], cmap=plt.cm.gray)
ax[2].set_title('Original >= local Otsu')

ax[3].imshow(glob_otsu[slice_index], cmap=plt.cm.gray)
ax[3].set_title(f'Global Otsu (t={t_glob_otsu})')

for a in ax:
    a.axis('off')

fig.tight_layout()
```



```
/usr/local/lib/python3.8/site-packages/skimage/filters/rank/generic.py:282: UserWarning:
Possible precision loss converting image of type float64 to uint8 as required by rank_
filters. Convert manually using skimage.util.img_as_ubyte to silence this warning.
```

The following example shows how local Otsu thresholding handles a global level shift applied to a synthetic image.

```
n = 100
theta = np.linspace(0, 10 * np.pi, n)
x = np.sin(theta)
m = (np.tile(x, (n, 1)) * np.linspace(0.1, 1, n) * 128 + 128).astype(np.uint8)

radius = 10
```

(continues on next page)

(continued from previous page)

```
t = rank.otsu(m, disk(radius))

fig, ax = plt.subplots(ncols=2, figsize=(10, 5),
                      sharex=True, sharey=True)

ax[0].imshow(m, cmap=plt.cm.gray)
ax[0].set_title('Original')

ax[1].imshow(m >= t, cmap=plt.cm.gray)
ax[1].set_title(f'Local Otsu ($r={radius}$)')

for a in ax:
    a.axis('off')

plt.tight_layout()
```

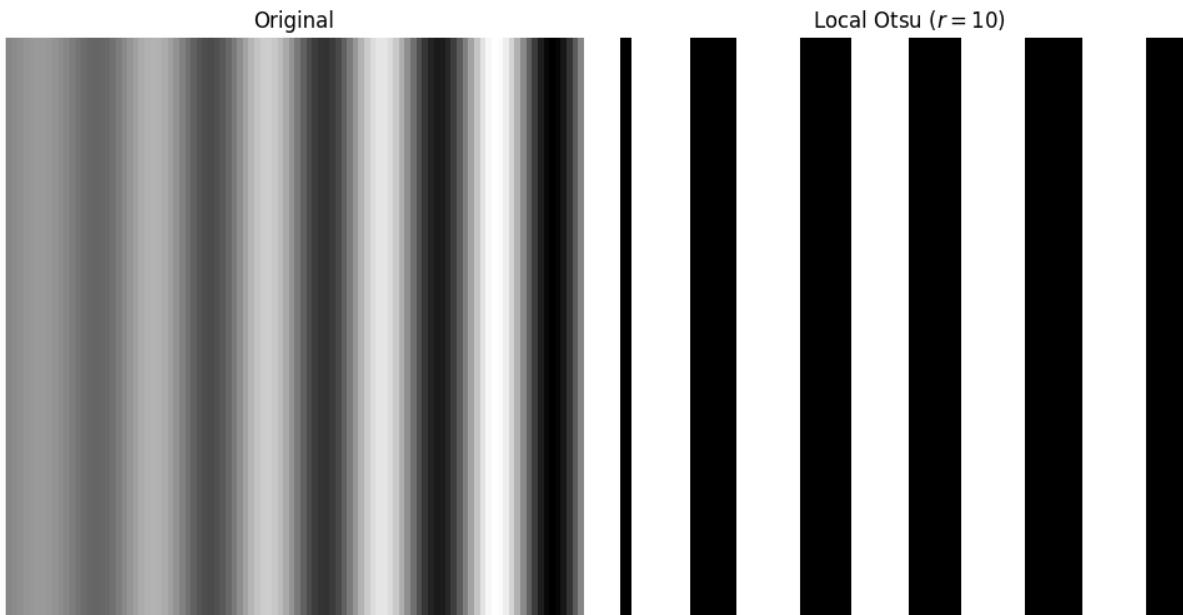


Image morphology

Local maximum and local minimum are the base operators for gray-level morphology.

Here is an example of the classical morphological gray-level filters: opening, closing and morphological gradient.

```
from skimage.filters.rank import maximum, minimum, gradient

noisy_image = img_as_ubyte(data.camera())

opening = maximum(minimum(noisy_image, disk(5)), disk(5))
closing = minimum(maximum(noisy_image, disk(5)), disk(5))
grad = gradient(noisy_image, disk(5))

# display results
```

(continues on next page)

(continued from previous page)

```
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(10, 10),
                        sharex=True, sharey=True)
ax = axes.ravel()

ax[0].imshow(noisy_image, cmap=plt.cm.gray)
ax[0].set_title('Original')

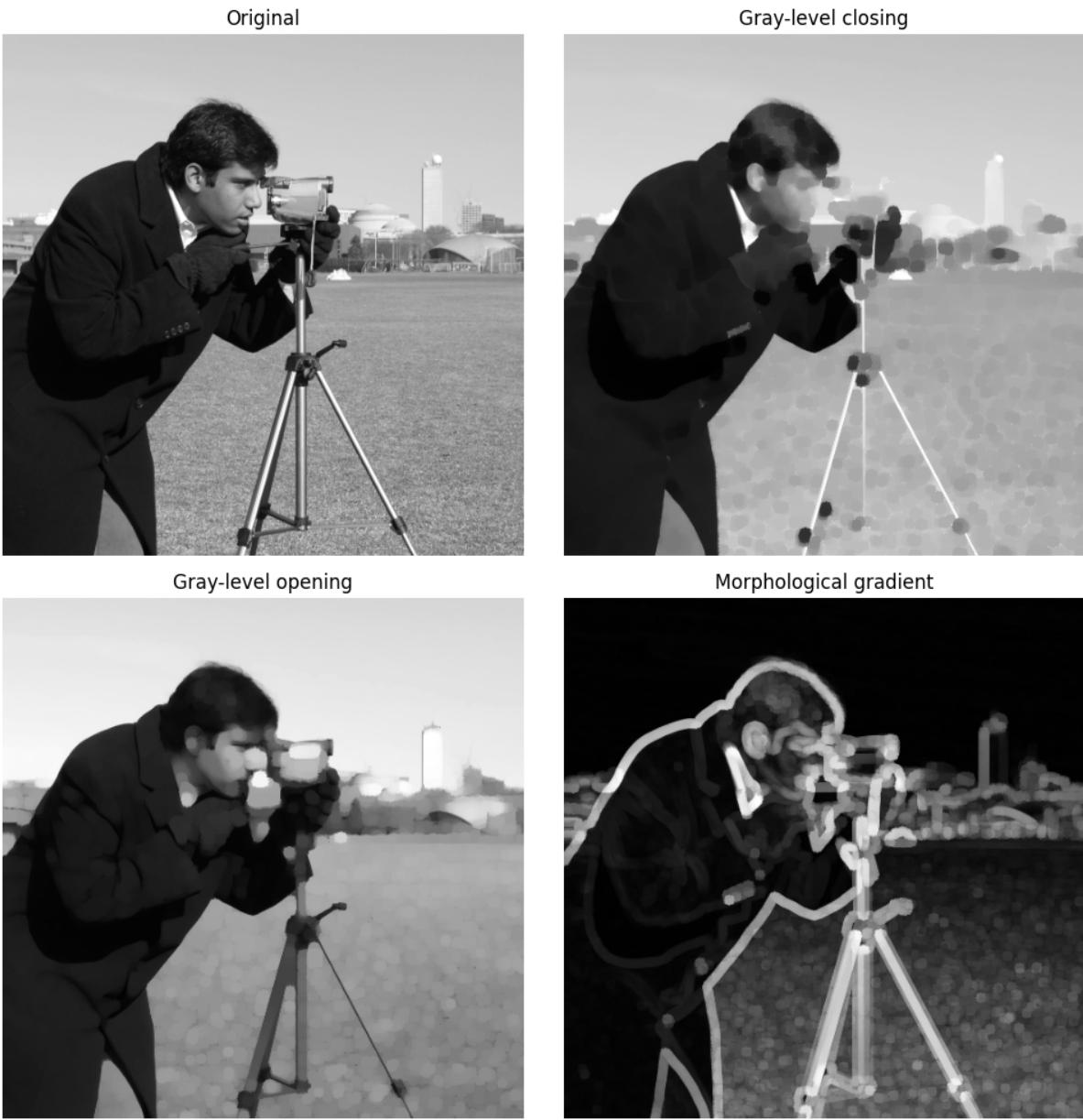
ax[1].imshow(closing, cmap=plt.cm.gray)
ax[1].set_title('Gray-level closing')

ax[2].imshow(opening, cmap=plt.cm.gray)
ax[2].set_title('Gray-level opening')

ax[3].imshow(grad, cmap=plt.cm.gray)
ax[3].set_title('Morphological gradient')

for a in ax:
    a.axis('off')

plt.tight_layout()
```



Feature extraction

Local histograms can be exploited to compute local entropy, which is related to the local image complexity. Entropy is computed using base 2 logarithm, i.e., the filter returns the minimum number of bits needed to encode local gray-level distribution.

`skimage.filters.rank.entropy()` returns the local entropy on a given structuring element. The following example applies this filter on 8- and 16-bit images.

Note: To better use the available image bit, the function returns 10x entropy for 8-bit images and 1000x entropy for 16-bit images.

```

from skimage import data
from skimage.filters.rank import entropy
from skimage.morphology import disk
import numpy as np
import matplotlib.pyplot as plt

image = data.camera()

fig, ax = plt.subplots(ncols=2, figsize=(12, 6), sharex=True, sharey=True)

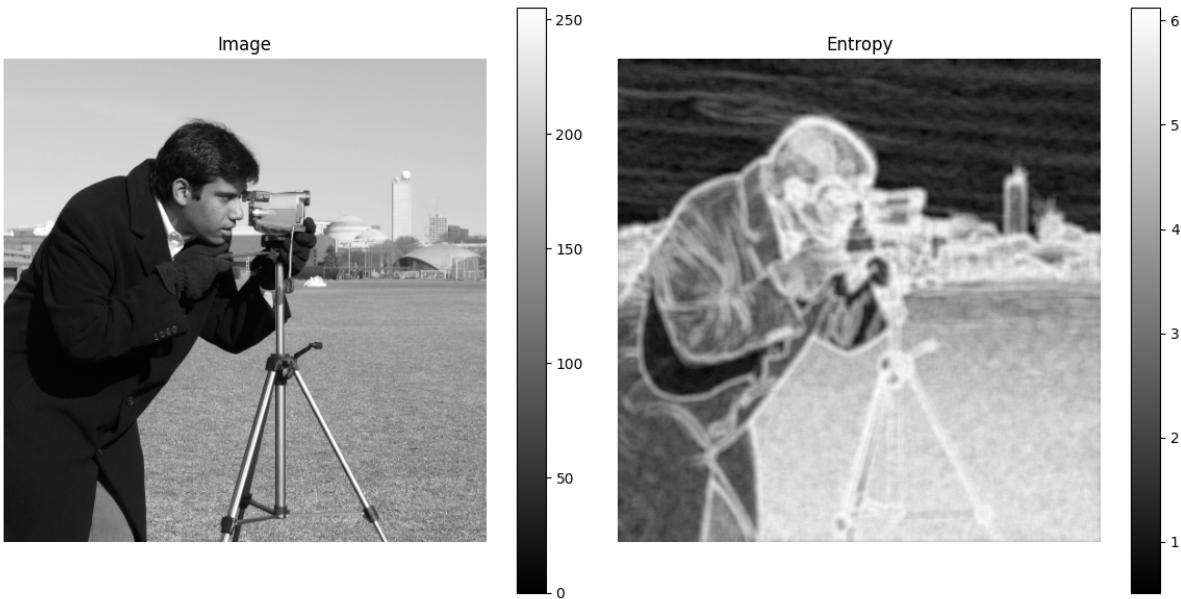
fig.colorbar(ax[0].imshow(image, cmap=plt.cm.gray), ax=ax[0])
ax[0].set_title('Image')

fig.colorbar(ax[1].imshow(entropy(image, disk(5)), cmap=plt.cm.gray), ax=ax[1])
ax[1].set_title('Entropy')

for a in ax:
    a.axis('off')

plt.tight_layout()

```



Implementation

The central part of the `skimage.filters.rank` filters is built on a sliding window that updates the local gray-level histogram. This approach limits the algorithm complexity to $O(n)$ where n is the number of image pixels. The complexity is also limited with respect to the structuring element size.

In the following, we compare the performance of different implementations available in `skimage`.

```
from time import time
```

(continues on next page)

(continued from previous page)

```

from scipy.ndimage import percentile_filter
from skimage.morphology import dilation
from skimage.filters.rank import median, maximum

def exec_and_timeit(func):
    """Decorator that returns both function results and execution time."""
    def wrapper(*arg):
        t1 = time()
        res = func(*arg)
        t2 = time()
        ms = (t2 - t1) * 1000.0
        return (res, ms)
    return wrapper

@exec_and_timeit
def cr_med(image, footprint):
    return median(image=image, footprint=footprint)

@exec_and_timeit
def cr_max(image, footprint):
    return maximum(image=image, footprint=footprint)

@exec_and_timeit
def cm_dil(image, footprint):
    return dilation(image=image, footprint=footprint)

@exec_and_timeit
def ndi_med(image, n):
    return percentile_filter(image, 50, size=n * 2 - 1)

```

Comparison between

- `skimage.filters.rank.maximum`
- `skimage.morphology.dilation`

on increasing structuring element size:

```

a = data.camera()

rec = []
e_range = range(1, 20, 2)
for r in e_range:
    elem = disk(r + 1)
    rc, ms_rc = cr_max(a, elem)
    rcm, ms_rcm = cm_dil(a, elem)
    rec.append((ms_rc, ms_rcm))

```

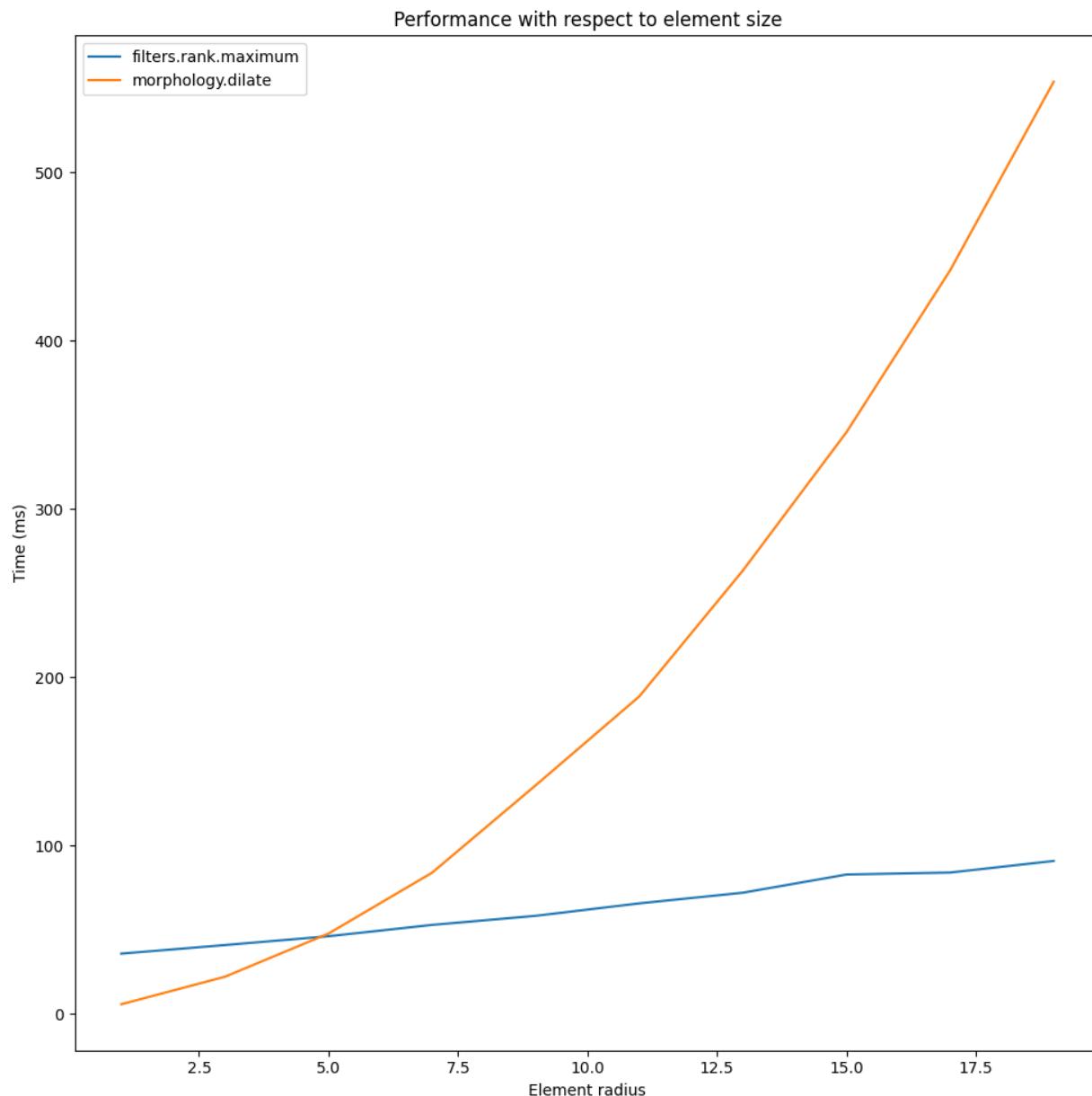
(continues on next page)

(continued from previous page)

```
rec = np.asarray(rec)

fig, ax = plt.subplots(figsize=(10, 10), sharey=True)
ax.set_title('Performance with respect to element size')
ax.set_ylabel('Time (ms)')
ax.set_xlabel('Element radius')
ax.plot(e_range, rec)
ax.legend(['filters.rank.maximum', 'morphology.dilate'])

plt.tight_layout()
```



and increasing image size:

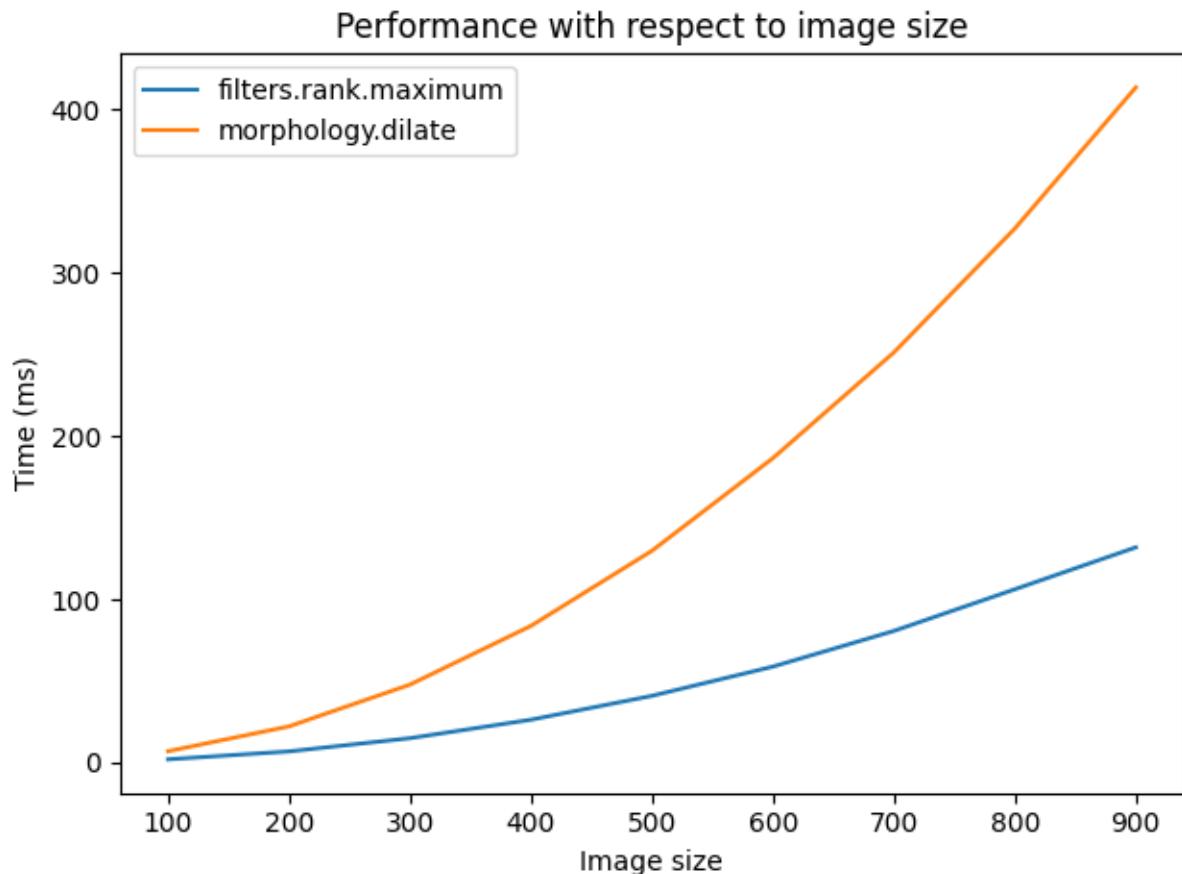
```
r = 9
elem = disk(r + 1)

rec = []
s_range = range(100, 1000, 100)
for s in s_range:
    a = (rng.random((s, s)) * 256).astype(np.uint8)
    (rc, ms_rc) = cr_max(a, elem)
    (rcm, ms_rcm) = cm_dil(a, elem)
    rec.append((ms_rc, ms_rcm))

rec = np.asarray(rec)

fig, ax = plt.subplots()
ax.set_title('Performance with respect to image size')
ax.set_ylabel('Time (ms)')
ax.set_xlabel('Image size')
ax.plot(s_range, rec)
ax.legend(['filters.rank.maximum', 'morphology.dilate'])

plt.tight_layout()
```



Comparison between:

- `skimage.filters.rank.median`
- `scipy.ndimage.percentile_filter`

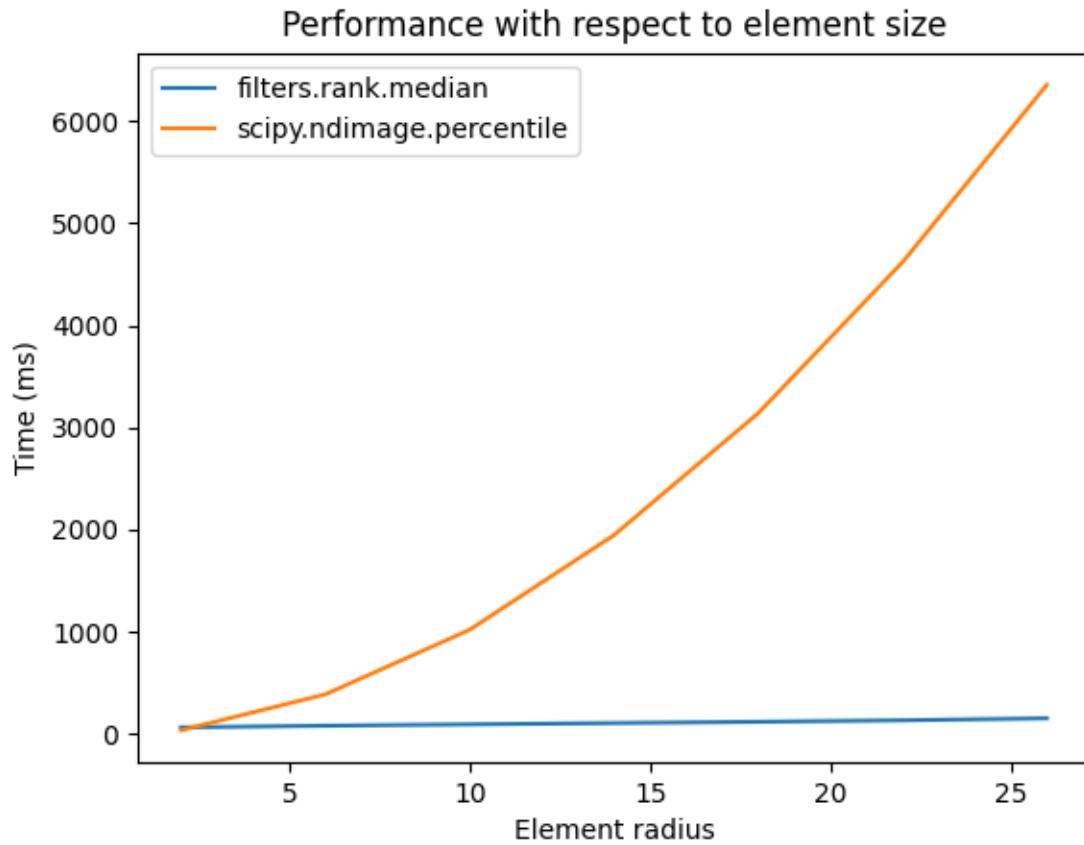
on increasing structuring element size:

```
a = data.camera()

rec = []
e_range = range(2, 30, 4)
for r in e_range:
    elem = disk(r + 1)
    rc, ms_rc = cr_med(a, elem)
    rndi, ms_ndi = ndi_med(a, r)
    rec.append((ms_rc, ms_ndi))

rec = np.asarray(rec)

fig, ax = plt.subplots()
ax.set_title('Performance with respect to element size')
ax.plot(e_range, rec)
ax.legend(['filters.rank.median', 'scipy.ndimage.percentile'])
ax.set_ylabel('Time (ms)')
ax.set_xlabel('Element radius')
```



```
Text(0.5, 23.52222222222222, 'Element radius')
```

Comparison of outcome of the two methods:

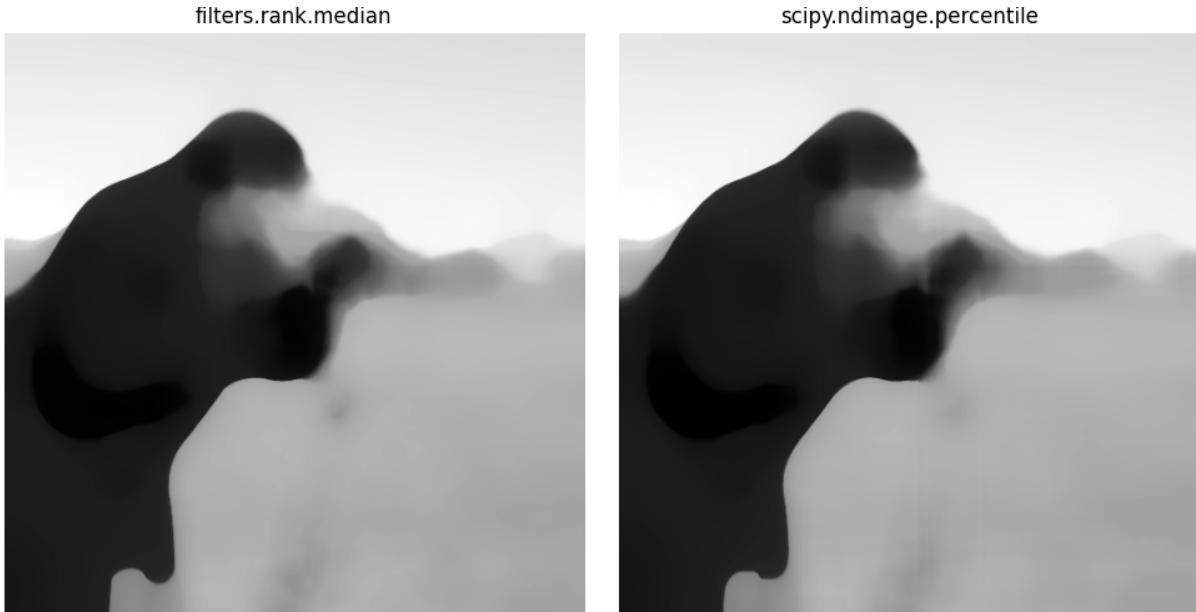
```
fig, ax = plt.subplots(ncols=2, figsize=(10, 5), sharex=True, sharey=True)

ax[0].set_title('filters.rank.median')
ax[0].imshow(rc, cmap=plt.cm.gray)

ax[1].set_title('scipy.ndimage.percentile')
ax[1].imshow(ndi, cmap=plt.cm.gray)

for a in ax:
    a.axis('off')

plt.tight_layout()
```



on increasing image size:

```
r = 9
elem = disk(r + 1)

rec = []
s_range = [100, 200, 500, 1000]
for s in s_range:
    a = (rng.random((s, s)) * 256).astype(np.uint8)
    (rc, ms_rc) = cr_med(a, elem)
    ndi, ms_ndi = ndi_med(a, r)
    rec.append((ms_rc, ms_ndi))

rec = np.asarray(rec)

fig, ax = plt.subplots()
```

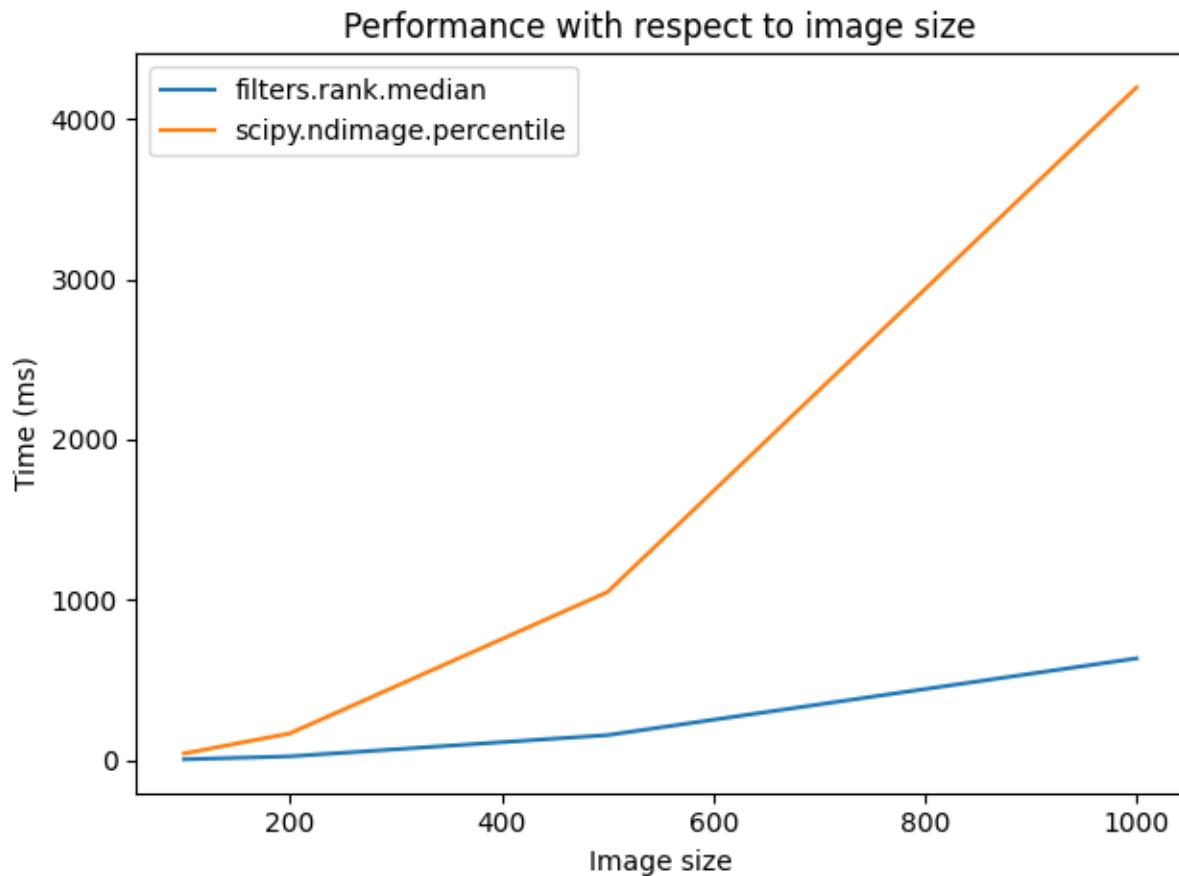
(continues on next page)

(continued from previous page)

```
ax.set_title('Performance with respect to image size')
ax.plot(s_range, rec)
ax.legend(['filters.rank.median', 'scipy.ndimage.percentile'])
ax.set_ylabel('Time (ms)')
ax.set_xlabel('Image size')

plt.tight_layout()

plt.show()
```



Total running time of the script: (0 minutes 39.002 seconds)

Examples for developers

In this folder, we have examples for advanced topics, including detailed explanations of the inner workings of certain algorithms.

These examples require some basic knowledge of image processing. They are targeted at existing or would-be scikit-image developers wishing to develop their knowledge of image processing algorithms.

Li thresholding

In 1993, Li and Lee proposed a new criterion for finding the “optimal” threshold to distinguish between the background and foreground of an image¹. They proposed that minimizing the *cross-entropy* between the foreground and the foreground mean, and the background and the background mean, would give the best threshold in most situations.

Until 1998, though, the way to find this threshold was by trying all possible thresholds and then choosing the one with the smallest cross-entropy. At that point, Li and Tam implemented a new, iterative method to more quickly find the optimum point by using the slope of the cross-entropy². This is the method implemented in scikit-image’s `skimage.filters.threshold_li()`.

Here, we demonstrate the cross-entropy and its optimization by Li’s iterative method. Note that we are using the private function `_cross_entropy`, which should not be used in production code, as it could change.

```
import numpy as np
import matplotlib.pyplot as plt
from skimage import data
from skimage import filters
from skimage.filters.thresholding import _cross_entropy

cell = data.cell()
camera = data.camera()
```

First, we let’s plot the cross entropy for the `skimage.data.camera()` image at all possible thresholds.

```
thresholds = np.arange(np.min(camera) + 1.5, np.max(camera) - 1.5)
entropies = [_cross_entropy(camera, t) for t in thresholds]

optimal_camera_threshold = thresholds[np.argmin(entropies)]

fig, ax = plt.subplots(1, 3, figsize=(8, 3))

ax[0].imshow(camera, cmap='gray')
ax[0].set_title('image')
ax[0].set_axis_off()

ax[1].imshow(camera > optimal_camera_threshold, cmap='gray')
ax[1].set_title('thresholded')
ax[1].set_axis_off()

ax[2].plot(thresholds, entropies)
ax[2].set_xlabel('thresholds')
ax[2].set_ylabel('cross-entropy')
ax[2].vlines(optimal_camera_threshold,
             ymin=np.min(entropies) - 0.05 * np.ptp(entropies),
             ymax=np.max(entropies) - 0.05 * np.ptp(entropies))
ax[2].set_title('optimal threshold')

fig.tight_layout()

print('The brute force optimal threshold is:', optimal_camera_threshold)
```

(continues on next page)

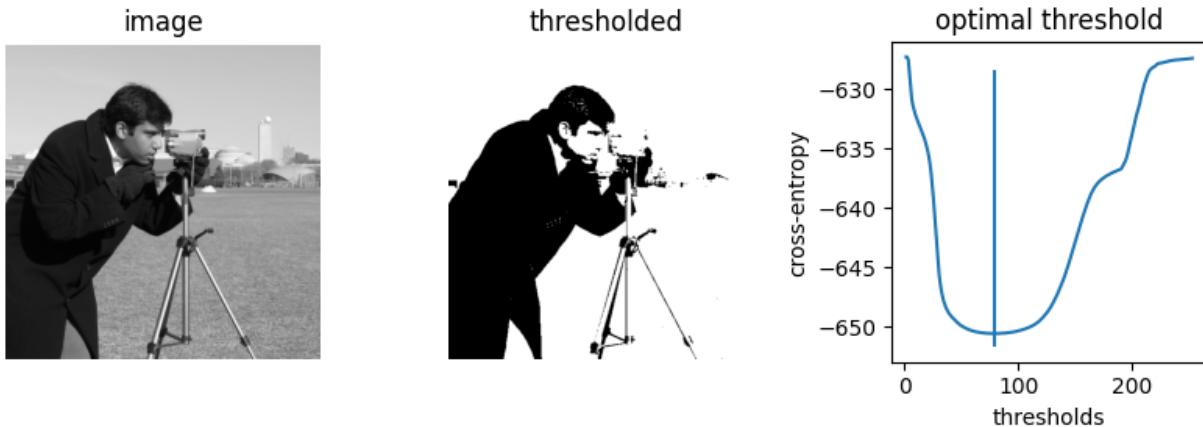
¹ Li C.H. and Lee C.K. (1993) “Minimum Cross Entropy Thresholding” Pattern Recognition, 26(4): 617-625 DOI:10.1016/0031-3203(93)90115-D

² Li C.H. and Tam P.K.S. (1998) “An Iterative Algorithm for Minimum Cross Entropy Thresholding” Pattern Recognition Letters, 18(8): 771-776 DOI:10.1016/S0167-8655(98)00057-9

(continued from previous page)

```
print('The computed optimal threshold is:', filters.threshold_li(camera))

plt.show()
```



```
The brute force optimal threshold is: 78.5
The computed optimal threshold is: 78.91288426606151
```

Next, let's use the `iter_callback` feature of `threshold_li` to examine the optimization process as it happens.

```
iter_thresholds = []

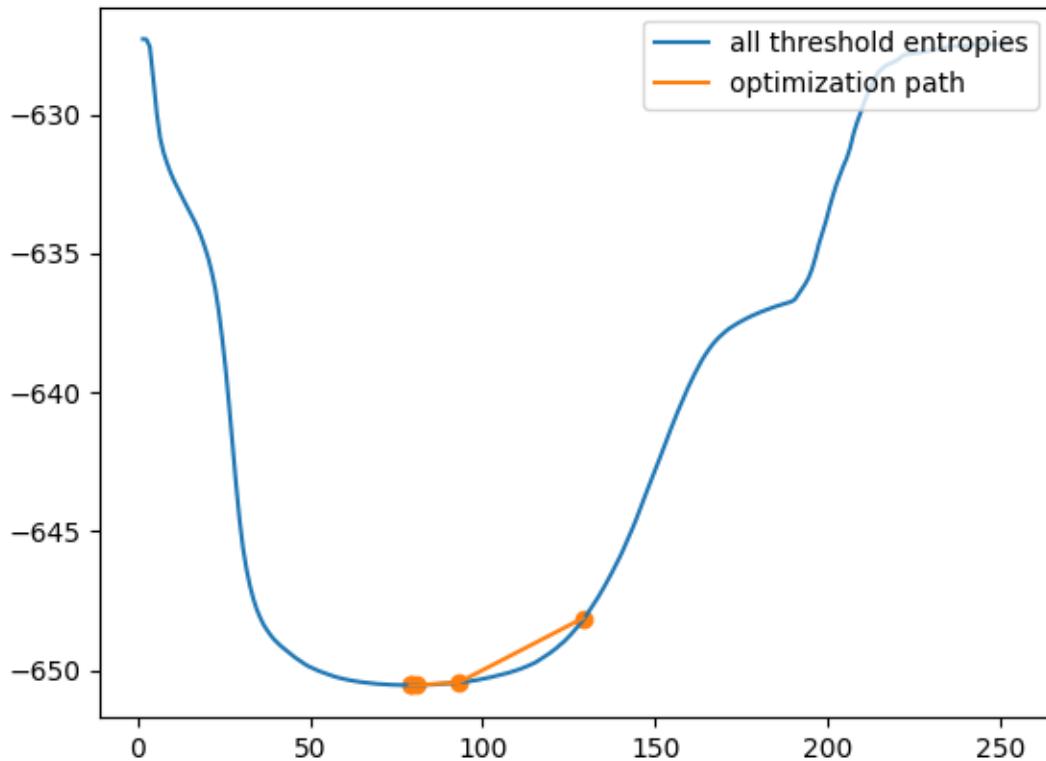
optimal_threshold = filters.threshold_li(camera,
                                         iter_callback=iter_thresholds.append)
iter_entropies = [_cross_entropy(camera, t) for t in iter_thresholds]

print('Only', len(iter_thresholds), 'thresholds examined.')

fig, ax = plt.subplots()

ax.plot(thresholds, entropies, label='all threshold entropies')
ax.plot(iter_thresholds, iter_entropies, label='optimization path')
ax.scatter(iter_thresholds, iter_entropies, c='C1')
ax.legend(loc='upper right')

plt.show()
```



Only 5 thresholds examined.

This is clearly much more efficient than the brute force approach. However, in some images, the cross-entropy is not *convex*, meaning having a single optimum. In this case, gradient descent could yield a threshold that is not optimal. In this example, we see how a bad initial guess for the optimization results in a poor threshold selection.

```
iter_thresholds2 = []

opt_threshold2 = filters.threshold_li(cell, initial_guess=64,
                                       iter_callback=iter_thresholds2.append)

thresholds2 = np.arange(np.min(cell) + 1.5, np.max(cell) - 1.5)
entropies2 = [_cross_entropy(cell, t) for t in thresholds]
iter_entropies2 = [_cross_entropy(cell, t) for t in iter_thresholds2]

fig, ax = plt.subplots(1, 3, figsize=(8, 3))

ax[0].imshow(cell, cmap='magma')
ax[0].set_title('image')
ax[0].set_axis_off()

ax[1].imshow(cell > opt_threshold2, cmap='gray')
ax[1].set_title('thresholded')
```

(continues on next page)

(continued from previous page)

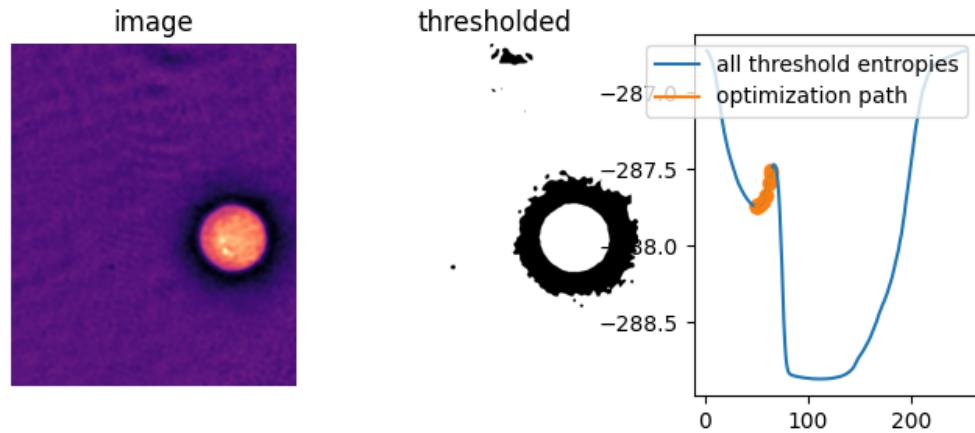
```

ax[1].set_axis_off()

ax[2].plot(thresholds2, entropies2, label='all threshold entropies')
ax[2].plot(iter_thresholds2, iter_entropies2, label='optimization path')
ax[2].scatter(iter_thresholds2, iter_entropies2, c='C1')
ax[2].legend(loc='upper right')

plt.show()

```



In this image, amazingly, the *default* initial guess, the mean image value, actually lies *right* on top of the peak between the two “valleys” of the objective function. Without supplying an initial guess, Li’s thresholding method does nothing at all!

```

iter_thresholds3 = []

opt_threshold3 = filters.threshold_li(cell,
                                       iter_callback=iter_thresholds3.append)

iter_entropies3 = [_cross_entropy(cell, t) for t in iter_thresholds3]

fig, ax = plt.subplots(1, 3, figsize=(8, 3))

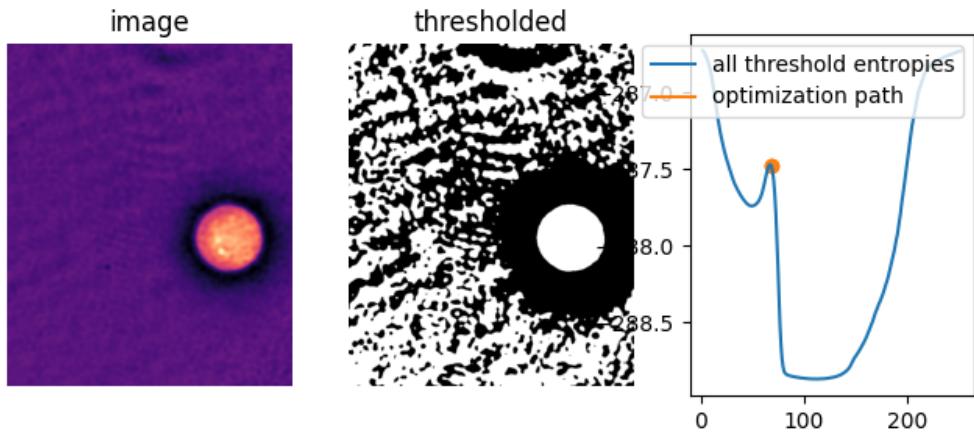
ax[0].imshow(cell, cmap='magma')
ax[0].set_title('image')
ax[0].set_axis_off()

ax[1].imshow(cell > opt_threshold3, cmap='gray')
ax[1].set_title('thresholded')
ax[1].set_axis_off()

ax[2].plot(thresholds2, entropies2, label='all threshold entropies')
ax[2].plot(iter_thresholds3, iter_entropies3, label='optimization path')
ax[2].scatter(iter_thresholds3, iter_entropies3, c='C1')
ax[2].legend(loc='upper right')

plt.show()

```



To see what is going on, let's define a function, `li_gradient`, that replicates the inner loop of the Li method and returns the *change* from the current threshold value to the next one. When this gradient is 0, we are at a so-called *stationary point* and Li returns this value. When it is negative, the next threshold guess will be lower, and when it is positive, the next guess will be higher.

In the plot below, we show the cross-entropy and the Li update path when the initial guess is on the *right* side of that entropy peak. We overlay the threshold update gradient, marking the 0 gradient line and the default initial guess by `threshold_li`.

```
def li_gradient(image, t):
    """Find the threshold update at a given threshold."""
    foreground = image > t
    mean_fore = np.mean(image[foreground])
    mean_back = np.mean(image[~foreground])
    t_next = ((mean_back - mean_fore) /
               (np.log(mean_back) - np.log(mean_fore)))
    dt = t_next - t
    return dt

iter_thresholds4 = []
opt_threshold4 = filters.threshold_li(cell, initial_guess=68,
                                       iter_callback=iter_thresholds4.append)
iter_entropies4 = [_cross_entropy(cell, t) for t in iter_thresholds4]
print(len(iter_thresholds4), 'examined, optimum:', opt_threshold4)

gradients = [li_gradient(cell, t) for t in thresholds2]

fig, ax1 = plt.subplots()
ax1.plot(thresholds2, entropies2)
ax1.plot(iter_thresholds4, iter_entropies4)
ax1.scatter(iter_thresholds4, iter_entropies4, c='C1')
ax1.set_xlabel('threshold')
ax1.set_ylabel('cross entropy', color='C0')
ax1.tick_params(axis='y', labelcolor='C0')

ax2 = ax1.twinx()
ax2.plot(thresholds2, gradients, c='C3')
```

(continues on next page)

(continued from previous page)

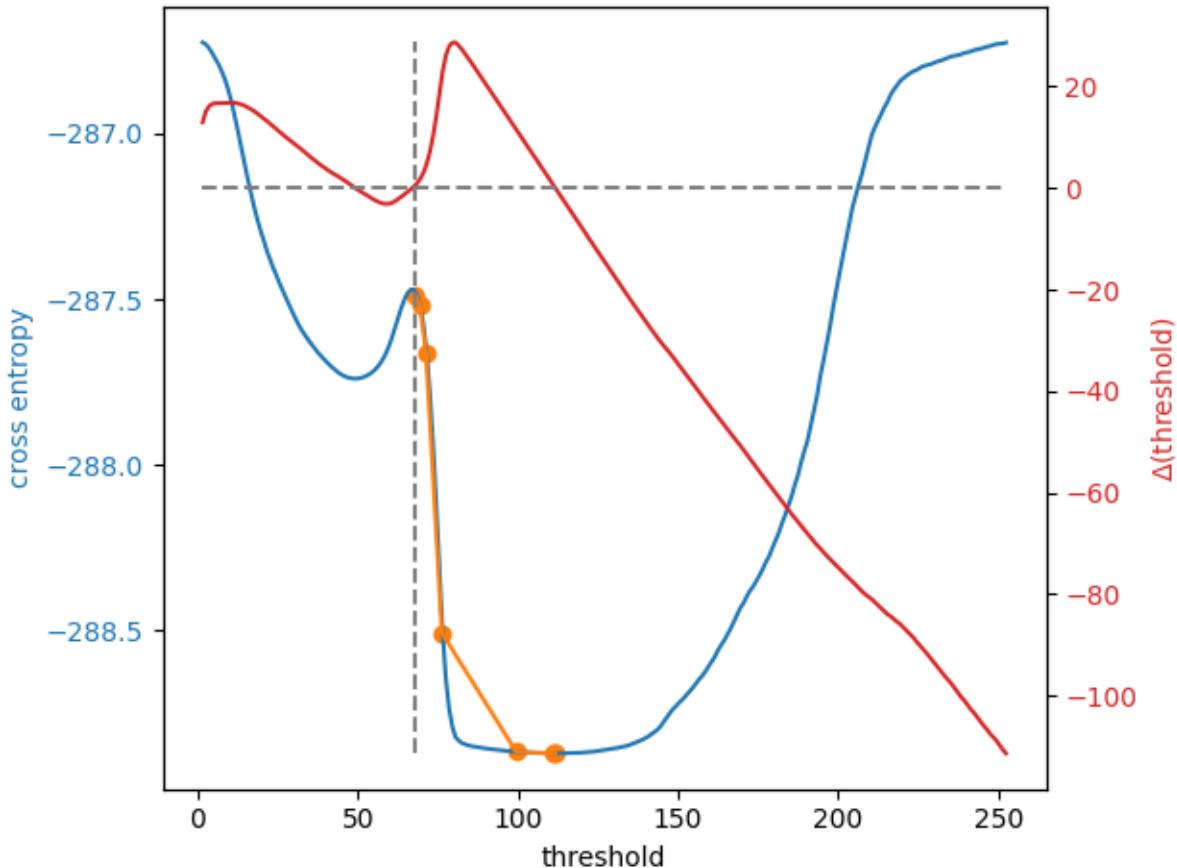
```

ax2.hlines([0], xmin=thresholds2[0], xmax=thresholds2[-1],
           colors='gray', linestyles='dashed')
ax2.vlines(np.mean(cell), ymin=np.min(gradients), ymax=np.max(gradients),
           colors='gray', linestyles='dashed')
ax2.set_ylabel(r'$\Delta(\text{threshold})$', color='C3')
ax2.tick_params(axis='y', labelcolor='C3')

fig.tight_layout()

plt.show()

```



```
8 examined, optimum: 111.68876119648344
```

In addition to allowing users to provide a number as an initial guess, `skimage.filters.threshold_li()` can receive a function that makes a guess from the image intensities, just like `numpy.mean()` does by default. This might be a good option when many images with different ranges need to be processed.

```

def quantile_95(image):
    # you can use np.quantile(image, 0.95) if you have NumPy>=1.15
    return np.percentile(image, 95)

iter_thresholds5 = []

```

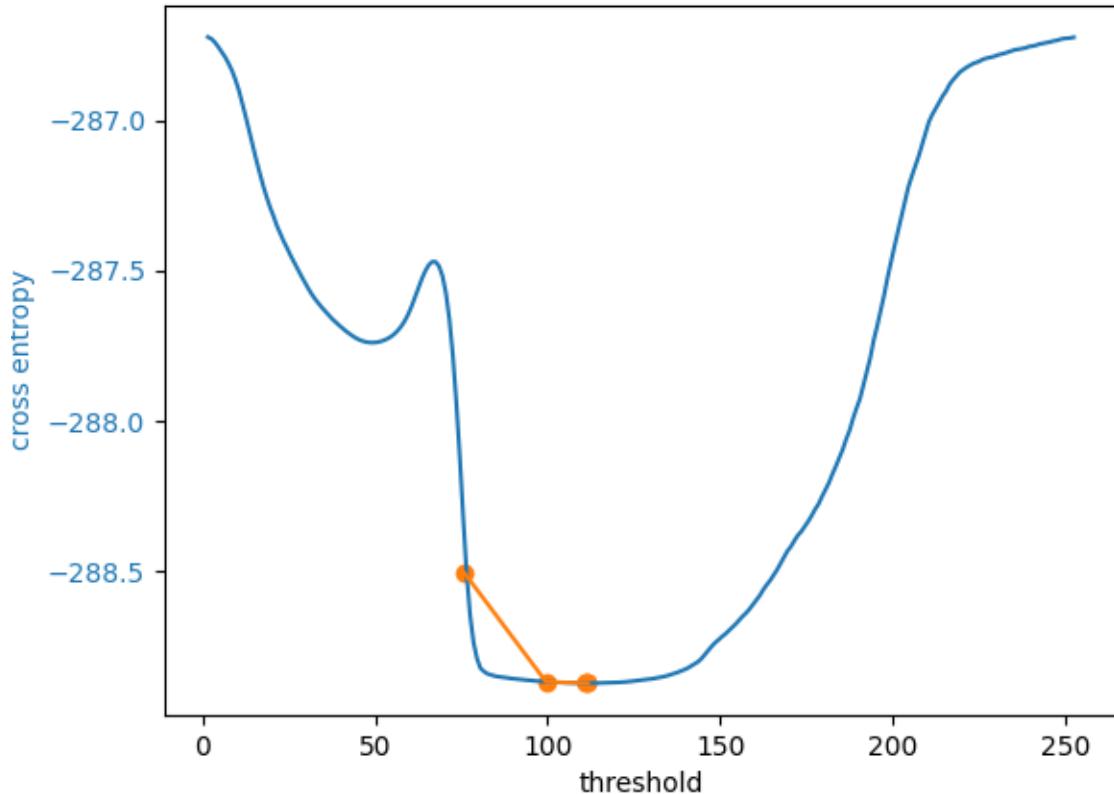
(continues on next page)

(continued from previous page)

```
opt_threshold5 = filters.threshold_li(cell, initial_guess=quantile_95,
                                      iter_callback=iter_thresholds5.append)
iter_entropies5 = [_cross_entropy(cell, t) for t in iter_thresholds5]
print(len(iter_thresholds5), 'examined, optimum:', opt_threshold5)

fig, ax1 = plt.subplots()
ax1.plot(thresholds2, entropies2)
ax1.plot(iter_thresholds5, iter_entropies5)
ax1.scatter(iter_thresholds5, iter_entropies5, c='C1')
ax1.set_xlabel('threshold')
ax1.set_ylabel('cross entropy', color='C0')
ax1.tick_params(axis='y', labelcolor='C0')

plt.show()
```



```
5 examined, optimum: 111.68876119648344
```

Total running time of the script: (0 minutes 6.017 seconds)

Max-tree

The max-tree is a hierarchical representation of an image that is the basis for a large family of morphological filters.

If we apply a threshold operation to an image, we obtain a binary image containing one or several connected components. If we apply a lower threshold, all the connected components from the higher threshold are contained in the connected components from the lower threshold. This naturally defines a hierarchy of nested components that can be represented by a tree. whenever a connected component A obtained by thresholding with threshold t1 is contained in a component B obtained by thresholding with threshold $t_1 < t_2$, we say that B is the parent of A. The resulting tree structure is called a component tree. The max-tree is a compact representation of such a component tree.^{1 2 3 4},

In this example we give an intuition of what a max-tree is.

References

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
from skimage.morphology import max_tree
import networkx as nx
```

Before we start : a few helper functions

```
def plot_img(ax, image, title, plot_text, image_values):
    """Plot an image, overlaying image values or indices."""
    ax.imshow(image, cmap='gray', aspect='equal', vmin=0, vmax=np.max(image))
    ax.set_title(title)
    ax.set_yticks([])
    ax.set_xticks([])

    for x in np.arange(-0.5, image.shape[0], 1.0):
        ax.add_artist(Line2D((x, x), (-0.5, image.shape[0] - 0.5),
                             color='blue', linewidth=2))

    for y in np.arange(-0.5, image.shape[1], 1.0):
        ax.add_artist(Line2D((-0.5, image.shape[1]), (y, y),
                             color='blue', linewidth=2))

    if plot_text:
        for i, j in np.ndindex(*image_values.shape):
            ax.text(j, i, image_values[i, j], fontsize=8,
                    horizontalalignment='center',
                    verticalalignment='center',
                    color='red')

    return
```

(continues on next page)

¹ Salembier, P., Oliveras, A., & Garrido, L. (1998). Antiextensive Connected Operators for Image and Sequence Processing. *IEEE Transactions on Image Processing*, 7(4), 555-570. DOI:10.1109/83.663500

² Berger, C., Géraud, T., Levillain, R., Widynski, N., Baillard, A., Bertin, E. (2007). Effective Component Tree Computation with Application to Pattern Recognition in Astronomical Imaging. In International Conference on Image Processing (ICIP) (pp. 41-44). DOI:10.1109/ICIP.2007.4379949

³ Najman, L., & Couperie, M. (2006). Building the component tree in quasi-linear time. *IEEE Transactions on Image Processing*, 15(11), 3531-3539. DOI:10.1109/TIP.2006.877518

⁴ Carlinet, E., & Géraud, T. (2014). A Comparative Review of Component Tree Computation Algorithms. *IEEE Transactions on Image Processing*, 23(9), 3885-3895. DOI:10.1109/TIP.2014.2336551

(continued from previous page)

```

def prune(G, node, res):
    """Transform a canonical max tree to a max tree."""
    value = G.nodes[node]['value']
    res[node] = str(node)
    preds = [p for p in G.predecessors(node)]
    for p in preds:
        if (G.nodes[p]['value'] == value):
            res[node] += f", {p}"
            G.remove_node(p)
        else:
            prune(G, p, res)
    G.nodes[node]['label'] = res[node]
    return

def accumulate(G, node, res):
    """Transform a max tree to a component tree."""
    total = G.nodes[node]['label']
    parents = G.predecessors(node)
    for p in parents:
        total += ', ' + accumulate(G, p, res)
    res[node] = total
    return total

def position_nodes_for_max_tree(G, image_rav, root_x=4, delta_x=1.2):
    """Set the position of nodes of a max-tree.

    This function helps to visually distinguish between nodes at the same
    level of the hierarchy and nodes at different levels.
    """

    pos = {}
    for node in reversed(list(nx.topological_sort(canonical_max_tree))):
        value = G.nodes[node]['value']
        if canonical_max_tree.out_degree(node) == 0:
            # root
            pos[node] = (root_x, value)

        in_nodes = [y for y in canonical_max_tree.predecessors(node)]

        # place the nodes at the same level
        level_nodes = [y for y in
                      filter(lambda x: image_rav[x] == value, in_nodes)]
        nb_level_nodes = len(level_nodes) + 1

        c = nb_level_nodes // 2
        i = -c
        if (len(level_nodes) < 3):
            hy = 0
            m = 0

```

(continues on next page)

(continued from previous page)

```
else:
    hy = 0.25
    m = hy / (c - 1)

    for level_node in level_nodes:
        if(i == 0):
            i += 1
        if (len(level_nodes) < 3):
            pos[level_node] = (pos[node][0] + i * 0.6 * delta_x, value)
        else:
            pos[level_node] = (pos[node][0] + i * 0.6 * delta_x,
                                value + m * (2 * np.abs(i) - c - 1))
        i += 1

    # place the nodes at different levels
    other_level_nodes = [y for y in
                          filter(lambda x: image_rav[x] > value, in_nodes)]
    if (len(other_level_nodes) == 1):
        i = 0
    else:
        i = - len(other_level_nodes) // 2
    for other_level_node in other_level_nodes:
        if((len(other_level_nodes) % 2 == 0) and (i == 0)):
            i += 1
        pos[other_level_node] = (pos[node][0] + i * delta_x,
                                  image_rav[other_level_node])
        i += 1

return pos

def plot_tree(graph, positions, ax, *, title='', labels=None,
              font_size=8, text_size=8):
    """Plot max and component trees."""
    nx.draw_networkx(graph, pos=positions, ax=ax,
                     node_size=40, node_shape='s', node_color='white',
                     font_size=font_size, labels=labels)
    for v in range(image_rav.min(), image_rav.max() + 1):
        ax.hlines(v - 0.5, -3, 10, linestyles='dotted')
        ax.text(-3, v - 0.15, f"val: {v}", fontsize=text_size)
    ax.hlines(v + 0.5, -3, 10, linestyles='dotted')
    ax.set_xlim(-3, 10)
    ax.set_title(title)
    ax.set_axis_off()
```

Image Definition

We define a small test image. For clarity, we choose an example image, where image values cannot be confounded with indices (different range).

```
image = np.array([[40, 40, 39, 39, 38],
                 [40, 41, 39, 39, 39],
                 [30, 30, 30, 32, 32],
                 [33, 33, 30, 32, 35],
                 [30, 30, 30, 33, 36]], dtype=np.uint8)
```

Max-tree

Next, we calculate the max-tree of this image. max-tree of the image

```
P, S = max_tree(image)

P_rav = P.ravel()
```

Image plots

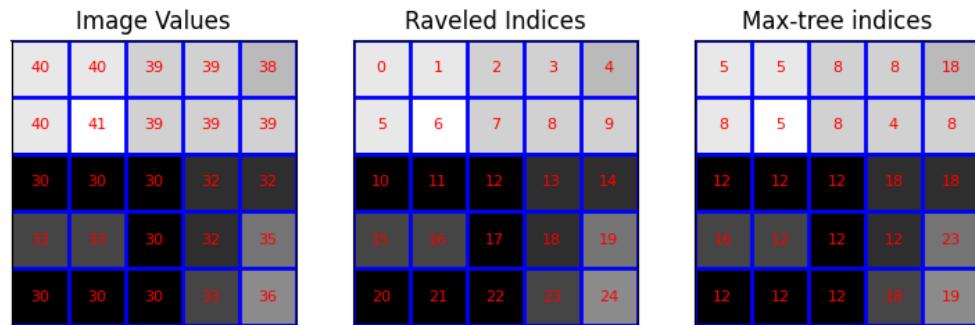
Then, we visualize the image and its raveled indices. Concretely, we plot the image with the following overlays: - the image values - the raveled indices (serve as pixel identifiers) - the output of the max_tree function

```
# raveled image
image_rav = image.ravel()

# raveled indices of the example image (for display purpose)
raveled_indices = np.arange(image.size).reshape(image.shape)

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, sharey=True, figsize=(9, 3))

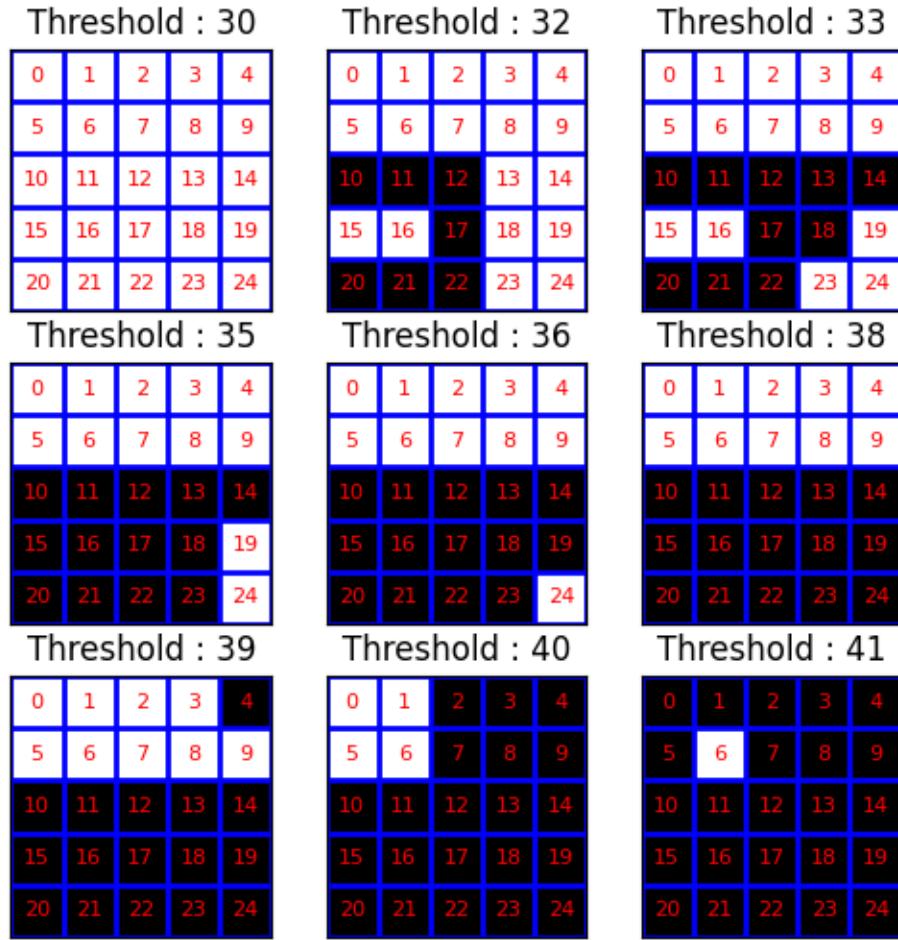
plot_img(ax1, image - image.min(), 'Image Values',
         plot_text=True, image_values=image)
plot_img(ax2, image - image.min(), 'Raveled Indices',
         plot_text=True, image_values=raveled_indices)
plot_img(ax3, image - image.min(), 'Max-tree indices',
         plot_text=True, image_values=P)
```



Visualizing threshold operations

Now, we investigate the results of a series of threshold operations. The component tree (and max-tree) provide representations of the inclusion relationships between connected components at different levels.

```
fig, axes = plt.subplots(3, 3, sharey=True, sharex=True, figsize=(6, 6))
thresholds = np.unique(image)
for k, threshold in enumerate(thresholds):
    bin_img = image >= threshold
    plot_img(axes[(k // 3), (k % 3)], bin_img, f"Threshold : {threshold}",
             plot_text=True, image_values=raveled_indices)
```



Max-tree plots

Now, we plot the component and max-trees. A component tree relates the different pixel sets resulting from all possible threshold operations to each other. There is an arrow in the graph, if a component at one level is included in the component of a lower level. The max-tree is just a different encoding of the pixel sets.

1. the component tree: pixel sets are explicitly written out. We see for instance that {6} (result of applying a threshold at 41) is the parent of {0, 1, 5, 6} (threshold at 40).
2. the max-tree: only pixels that come into the set at this level are explicitly written out. We therefore will write {6} -> {0,1,5} instead of {6} -> {0, 1, 5, 6}
3. the canonical max-tree: this is the representation which is given by our implementation. Here, every pixel is a node. Connected components of several pixels are represented by one of the pixels. We thus replace {6} -> {0,1,5} by {6} -> {5}, {1} -> {5}, {0} -> {5}. This allows us to represent the graph by an image (top row, third column).

```

# the canonical max-tree graph
canonical_max_tree = nx.DiGraph()
canonical_max_tree.add_nodes_from(S)
for node in canonical_max_tree.nodes():
    canonical_max_tree.nodes[node]['value'] = image_rav[node]
canonical_max_tree.add_edges_from([(n, P_rav[n]) for n in S[1:]])

# max-tree from the canonical max-tree
nx_max_tree = nx.DiGraph(canonical_max_tree)
labels = {}
prune(nx_max_tree, S[0], labels)

# component tree from the max-tree
labels_ct = {}
total = accumulate(nx_max_tree, S[0], labels_ct)

# positions of nodes : canonical max-tree (CMT)
pos_cmt = position_nodes_for_max_tree(canonical_max_tree, image_rav)

# positions of nodes : max-tree (MT)
pos_mt = dict(zip(nx_max_tree.nodes, [pos_cmt[node] for node in nx_max_tree.nodes]))

# plot the trees with networkx and matplotlib
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, sharey=True, figsize=(20, 8))

plot_tree(nx_max_tree, pos_mt, ax1, title='Component tree',
          labels=labels_ct, font_size=6, text_size=8)

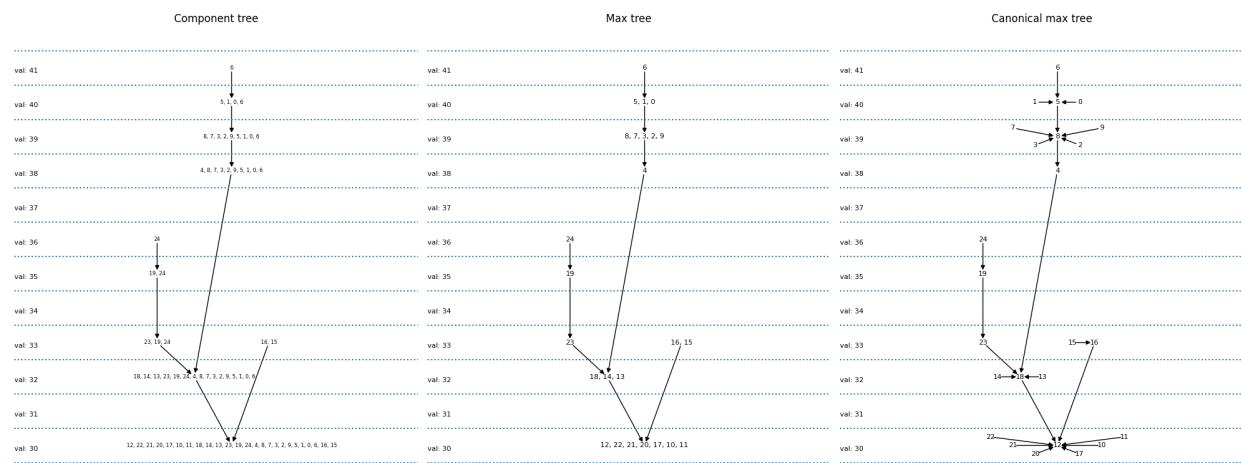
plot_tree(nx_max_tree, pos_mt, ax2, title='Max tree', labels=labels)

plot_tree(canonical_max_tree, pos_cmt, ax3, title='Canonical max tree')

fig.tight_layout()

plt.show()

```



Total running time of the script: (0 minutes 1.095 seconds)

1.3 API reference

1.3.1 skimage

Image Processing for Python

`scikit-image` (a.k.a. `skimage`) is a collection of algorithms for image processing and computer vision.

The main package of `skimage` only provides a few utilities for converting between image data types; for most features, you need to import one of the following subpackages:

Subpackages

color

Color space conversion.

data

Test images and example data.

draw

Drawing primitives (lines, text, etc.) that operate on NumPy arrays.

exposure

Image intensity adjustment, e.g., histogram equalization, etc.

feature

Feature detection and extraction, e.g., texture analysis corners, etc.

filters

Sharpening, edge finding, rank filters, thresholding, etc.

graph

Graph-theoretic operations, e.g., shortest paths.

io

Reading, saving, and displaying images and video.

measure

Measurement of image properties, e.g., region properties and contours.

metrics

Metrics corresponding to images, e.g. distance metrics, similarity, etc.

morphology

Morphological operations, e.g., opening or skeletonization.

restoration

Restoration algorithms, e.g., deconvolution algorithms, denoising, etc.

segmentation

Partitioning an image into multiple regions.

transform

Geometric and other transforms, e.g., rotation or the Radon transform.

util

Generic utilities.

Utility Functions

`img_as_float`

Convert an image to floating point format, with values in [0, 1]. Is similar to `img_as_float64`, but will not convert lower-precision floating point arrays to `float64`.

`img_as_float32`

Convert an image to single-precision (32-bit) floating point format, with values in [0, 1].

`img_as_float64`

Convert an image to double-precision (64-bit) floating point format, with values in [0, 1].

`img_as_uint`

Convert an image to unsigned integer format, with values in [0, 65535].

`img_as_int`

Convert an image to signed integer format, with values in [-32768, 32767].

`img_as_ubyte`

Convert an image to unsigned byte format, with values in [0, 255].

`img_as_bool`

Convert an image to boolean format, with values either True or False.

`dtype_limits`

Return intensity limits, i.e. (min, max) tuple, of the image's dtype.

<code>skimage.color</code>	
<code>skimage.data</code>	Test images and datasets.
<code>skimage.draw</code>	
<code>skimage.exposure</code>	
<code>skimage.feature</code>	
<code>skimage.filters</code>	
<code>skimage.future</code>	Functionality with an experimental API.
<code>skimage.graph</code>	
<code>skimage.io</code>	Utilities to read and write images in various formats.
<code>skimage.measure</code>	
<code>skimage.metrics</code>	
<code>skimage.morphology</code>	
<code>skimage.registration</code>	
<code>skimage.restoration</code>	Image restoration module.
<code>skimage.segmentation</code>	
<code>skimage.transform</code>	This module includes tools to transform images and volumetric data.
<code>skimage.util</code>	

1.3.2 `skimage.color`

<code>skimage.color.combine_stains</code>	Stain to RGB color space conversion.
<code>skimage.color.convert_colorspace</code>	Convert an image array to a new color space.
<code>skimage.color.deltaE_cie76</code>	Euclidean distance between two points in Lab color space
<code>skimage.color.deltaE_ciede2000</code>	Color difference as given by the CIEDE 2000 standard.
<code>skimage.color.deltaE_ciede94</code>	Color difference according to CIEDE 94 standard
<code>skimage.color.deltaE_cmc</code>	Color difference from the CMC l:c standard.
<code>skimage.color.gray2rgb</code>	Create an RGB representation of a gray-level image.
<code>skimage.color.gray2rgba</code>	Create a RGBA representation of a gray-level image.
<code>skimage.color.hed2rgb</code>	Haematoxylin-Eosin-DAB (HED) to RGB color space conversion.
<code>skimage.color.hsv2rgb</code>	HSV to RGB color space conversion.
<code>skimage.color.lab2lch</code>	Convert image in CIE-LAB to CIE-LCh color space.
<code>skimage.color.lab2rgb</code>	Convert image in CIE-LAB to sRGB color space.
<code>skimage.color.lab2xyz</code>	Convert image in CIE-LAB to XYZ color space.
<code>skimage.color.label2rgb</code>	Return an RGB image where color-coded labels are painted over the image.

continues on next page

Table 3 – continued from previous page

<code>skimage.color.lch2lab</code>	Convert image in CIE-LCh to CIE-LAB color space.
<code>skimage.color.luv2rgb</code>	Luv to RGB color space conversion.
<code>skimage.color.luv2xyz</code>	CIE-Luv to XYZ color space conversion.
<code>skimage.color.rgb2gray</code>	Compute luminance of an RGB image.
<code>skimage.color.rgb2hed</code>	RGB to Haematoxylin-Eosin-DAB (HED) color space conversion.
<code>skimage.color.rgb2hsv</code>	RGB to HSV color space conversion.
<code>skimage.color.rgb2lab</code>	Conversion from the sRGB color space (IEC 61966-2-1:1999) to the CIE Lab colorspace under the given illuminant and observer.
<code>skimage.color.rgb2luv</code>	RGB to CIE-Luv color space conversion.
<code>skimage.color.rgb2rgbcie</code>	RGB to RGB CIE color space conversion.
<code>skimage.color.rgb2xyz</code>	RGB to XYZ color space conversion.
<code>skimage.color.rgb2ycbcr</code>	RGB to YCbCr color space conversion.
<code>skimage.color.rgb2ydbdr</code>	RGB to YDbDr color space conversion.
<code>skimage.color.rgb2yiq</code>	RGB to YIQ color space conversion.
<code>skimage.color.rgb2ypbpr</code>	RGB to YPbPr color space conversion.
<code>skimage.color.rgb2yuv</code>	RGB to YUV color space conversion.
<code>skimage.color.rgba2rgb</code>	RGBA to RGB conversion using alpha blending [?].
<code>skimage.color.rgbcie2rgb</code>	RGB CIE to RGB color space conversion.
<code>skimage.color.separate_stains</code>	RGB to stain color space conversion.
<code>skimage.color.xyz2lab</code>	XYZ to CIE-LAB color space conversion.
<code>skimage.color.xyz2luv</code>	XYZ to CIE-Luv color space conversion.
<code>skimage.color.xyz2rgb</code>	XYZ to RGB color space conversion.
<code>skimage.color.xyz_tristimulus_values</code>	Get the CIE XYZ tristimulus values.
<code>skimage.color.ycbcr2rgb</code>	YCbCr to RGB color space conversion.
<code>skimage.color.ydbdr2rgb</code>	YDbDr to RGB color space conversion.
<code>skimage.color.yiq2rgb</code>	YIQ to RGB color space conversion.
<code>skimage.color.ypbpr2rgb</code>	YPbPr to RGB color space conversion.
<code>skimage.color.yuv2rgb</code>	YUV to RGB color space conversion.

`skimage.color.combine_stains(stains, conv_matrix, *, channel_axis=-1)`

Stain to RGB color space conversion.

Parameters

`stains`

[(..., 3, ...) array_like] The image in stain color space. By default, the final dimension denotes channels.

`conv_matrix: ndarray`

The stain separation matrix as described by G. Landini [?].

`channel_axis`

[int, optional] This parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

out

[(..., 3, ...) ndarray] The image in RGB format. Same dimensions as input.

Raises

ValueError

If *stains* is not at least 2-D with shape (... , 3, ...).

Notes

Stain combination matrices available in the color module and their respective colorspace:

- `rgb_from_hed`: Hematoxylin + Eosin + DAB
- `rgb_from_hdx`: Hematoxylin + DAB
- `rgb_from_fgx`: Feulgen + Light Green
- `rgb_from_bex`: Giemsa stain : Methyl Blue + Eosin
- `rgb_from_rbd`: FastRed + FastBlue + DAB
- `rgb_from_gdx`: Methyl Green + DAB
- `rgb_from_hax`: Hematoxylin + AEC
- `rgb_from_bro`: Blue matrix Aniline Blue + Red matrix Azocarmine + Orange matrix Orange-G
- `rgb_from_bpx`: Methyl Blue + Ponceau Fuchsin
- `rgb_from_ahx`: Alcian Blue + Hematoxylin
- `rgb_from_hpx`: Hematoxylin + PAS

References

[?], [?]

Examples

```
>>> from skimage import data
>>> from skimage.color import (separate_stains, combine_stains,
...                             hdx_from_rgb, rgb_from_hdx)
>>> ihc = data.immunohistochemistry()
>>> ihc_hdx = separate_stains(ihc, hdx_from_rgb)
>>> ihc_rgb = combine_stains(ihc_hdx, rgb_from_hdx)
```

`skimage.color.convert_colorspace(arr, fromspace, tospace, *, channel_axis=-1)`

Convert an image array to a new color space.

Valid color spaces are:

‘RGB’, ‘HSV’, ‘RGB CIE’, ‘XYZ’, ‘YUV’, ‘YIQ’, ‘YPbPr’, ‘YCbCr’, ‘YDbDr’

Parameters

arr

[(..., 3, ...) array_like] The image to convert. By default, the final dimension denotes channels.

fromspace

[str] The color space to convert from. Can be specified in lower case.

tospace

[str] The color space to convert to. Can be specified in lower case.

channel_axis

[int, optional] This parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

out

[(..., 3, ...) ndarray] The converted image. Same dimensions as input.

Raises

ValueError

If `fromspace` is not a valid color space

ValueError

If `tospace` is not a valid color space

Notes

Conversion is performed through the “central” RGB color space, i.e. conversion from XYZ to HSV is implemented as XYZ -> RGB -> HSV instead of directly.

Examples

```
>>> from skimage import data
>>> img = data.astronaut()
>>> img_hsv = convert_colorspace(img, 'RGB', 'HSV')
```

```
skimage.color.deltaE_cie76(lab1, lab2, channel_axis=-1)
```

Euclidean distance between two points in Lab color space

Parameters

lab1

[array_like] reference color (Lab colorspace)

lab2

[array_like] comparison color (Lab colorspace)

channel_axis

[int, optional] This parameter indicates which axis of the arrays corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

dE

[array_like] distance between colors `lab1` and `lab2`

References

[?], [?]

```
skimage.color.deltaE_ciede2000(lab1, lab2, kL=1, kC=1, kH=1, *, channel_axis=-1)
```

Color difference as given by the CIEDE 2000 standard.

CIEDE 2000 is a major revision of CIEDE94. The perceptual calibration is largely based on experience with automotive paint on smooth surfaces.

Parameters

lab1

[array_like] reference color (Lab colorspace)

lab2

[array_like] comparison color (Lab colorspace)

kL

[float (range), optional] lightness scale factor, 1 for “acceptably close”; 2 for “imperceptible”
see deltaE_cmc

kC

[float (range), optional] chroma scale factor, usually 1

kH

[float (range), optional] hue scale factor, usually 1

channel_axis

[int, optional] This parameter indicates which axis of the arrays corresponds to channels.

New in version 0.19: channel_axis was added in 0.19.

Returns

deltaE

[array_like] The distance between lab1 and lab2

Notes

CIEDE 2000 assumes parametric weighting factors for the lightness, chroma, and hue (kL , kC , kH respectively). These default to 1.

References

[?], [?], [?]

```
skimage.color.deltaE_ciede94(lab1, lab2, kH=1, kC=1, kL=1, k1=0.045, k2=0.015, *, channel_axis=-1)
```

Color difference according to CIEDE 94 standard

Accommodates perceptual non-uniformities through the use of application specific scale factors (kH , kC , kL , $k1$, and $k2$).

Parameters**lab1**

[array_like] reference color (Lab colorspace)

lab2

[array_like] comparison color (Lab colorspace)

kH

[float, optional] Hue scale

kC

[float, optional] Chroma scale

kL

[float, optional] Lightness scale

k1

[float, optional] first scale parameter

k2

[float, optional] second scale parameter

channel_axis

[int, optional] This parameter indicates which axis of the arrays corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns**dE**

[array_like] color difference between *lab1* and *lab2*

Notes

`deltaE_ciede94` is not symmetric with respect to *lab1* and *lab2*. CIEDE94 defines the scales for the lightness, hue, and chroma in terms of the first color. Consequently, the first color should be regarded as the “reference” color.

kL, *k1*, *k2* depend on the application and default to the values suggested for graphic arts

Parameter	Graphic Arts	Textiles
<i>kL</i>	1.000	2.000
<i>k1</i>	0.045	0.048
<i>k2</i>	0.015	0.014

References

[?], [?]

`skimage.color.deltaE_cmc(lab1, lab2, kL=1, kC=1, *, channel_axis=-1)`

Color difference from the CMC l:c standard.

This color difference was developed by the Colour Measurement Committee (CMC) of the Society of Dyers and Colourists (United Kingdom). It is intended for use in the textile industry.

The scale factors kL , kC set the weight given to differences in lightness and chroma relative to differences in hue. The usual values are $kL=2$, $kC=1$ for “acceptability” and $kL=1$, $kC=1$ for “imperceptibility”. Colors with $dE > 1$ are “different” for the given scale factors.

Parameters

lab1

[array_like] reference color (Lab colorspace)

lab2

[array_like] comparison color (Lab colorspace)

channel_axis

[int, optional] This parameter indicates which axis of the arrays corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

dE

[array_like] distance between colors $lab1$ and $lab2$

Notes

`deltaE_cmc` defines the scales for the lightness, hue, and chroma in terms of the first color. Consequently $\text{deltaE_cmc}(\text{lab1}, \text{lab2}) \neq \text{deltaE_cmc}(\text{lab2}, \text{lab1})$

References

[?], [?], [?]

`skimage.color.gray2rgb(image, *, channel_axis=-1)`

Create an RGB representation of a gray-level image.

Parameters

image

[array_like] Input image.

channel_axis

[int, optional] This parameter indicates which axis of the output array will correspond to channels.

Returns

rgb

[(..., 3, ...) ndarray] RGB image. A new dimension of length 3 is added to input image.

Notes

If the input is a 1-dimensional image of shape (M,), the output will be shape (M, 3).

- *Tinting gray-scale images*
- *Circular and Elliptical Hough Transforms*
- *Region Boundary based RAGs*

skimage.color.gray2rgba(image, alpha=None, *, channel_axis=-1)

Create a RGBA representation of a gray-level image.

Parameters

image

[array_like] Input image.

alpha

[array_like, optional] Alpha channel of the output image. It may be a scalar or an array that can be broadcast to `image`. If not specified it is set to the maximum limit corresponding to the `image` dtype.

channel_axis

[int, optional] This parameter indicates which axis of the output array will correspond to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

rgba

[ndarray] RGBA image. A new dimension of length 4 is added to input image shape.

`skimage.color.hed2rgb(hed, *, channel_axis=-1)`

Haematoxylin-Eosin-DAB (HED) to RGB color space conversion.

Parameters

hed

[(..., 3, ...) array_like] The image in the HED color space. By default, the final dimension denotes channels.

channel_axis

[int, optional] This parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

out

[(..., 3, ...) ndarray] The image in RGB. Same dimensions as input.

Raises

ValueError

If `hed` is not at least 2-D with shape (... , 3, ...).

References

[?]

Examples

```
>>> from skimage import data
>>> from skimage.color import rgb2hed, hed2rgb
>>> ihc = data.immunohistochemistry()
>>> ihc_hed = rgb2hed(ihc)
>>> ihc_rgb = hed2rgb(ihc_hed)
```

- Separate colors in immunohistochemical staining

`skimage.color.hsv2rgb(hsv, *, channel_axis=-1)`

HSV to RGB color space conversion.

Parameters

`hsv`

[(..., 3, ...) array_like] The image in HSV format. By default, the final dimension denotes channels.

`channel_axis`

[int, optional] This parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

`out`

[(..., 3, ...) ndarray] The image in RGB format. Same dimensions as input.

Raises

`ValueError`

If `hsv` is not at least 2-D with shape (... , 3, ...).

Notes

Conversion between RGB and HSV color spaces results in some loss of precision, due to integer arithmetic and rounding [?].

References

[?]

Examples

```
>>> from skimage import data
>>> img = data.astronaut()
>>> img_hsv = rgb2hsv(img)
>>> img_rgb = hsv2rgb(img_hsv)
```

- *Tinting gray-scale images*
- *Flood Fill*

`skimage.color.lab2lch(lab, *, channel_axis=-1)`

Convert image in CIE-LAB to CIE-LCh color space.

CIE-LCh is the cylindrical representation of the CIE-LAB (Cartesian) color space.

Parameters

lab

[(..., 3, ...) array_like] The input image in CIE-LAB color space. Unless `channel_axis` is set, the final dimension denotes the CIE-LAB channels. The L* values range from 0 to 100; the a* and b* values range from -128 to 127.

channel_axis

[int, optional] This parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

out

[(..., 3, ...) ndarray] The image in CIE-LCh color space, of same shape as input.

Raises

ValueError

If *lab* does not have at least 3 channels (i.e., L*, a*, and b*).

See also:

[lch2lab](#)

Notes

The h channel (i.e., hue) is expressed as an angle in range (0, 2*pi).

References

[?], [?], [?]

Examples

```
>>> from skimage import data
>>> from skimage.color import rgb2lab, lab2lch
>>> img = data.astronaut()
>>> img_lab = rgb2lab(img)
>>> img_lch = lab2lch(img_lab)
```

`skimage.color.lab2rgb(lab, illuminant='D65', observer='2', *, channel_axis=-1)`

Convert image in CIE-LAB to sRGB color space.

Parameters**lab**

[..., 3, ...] array_like] The input image in CIE-LAB color space. Unless *channel_axis* is set, the final dimension denotes the CIE-LAB channels. The L* values range from 0 to 100; the a* and b* values range from -128 to 127.

illuminant

[{"A", "B", "C", "D50", "D55", "D65", "D75", "E"}, optional] The name of the illuminant (the function is NOT case sensitive).

observer

[{"2", "10", "R"}, optional] The aperture angle of the observer.

channel_axis

[int, optional] This parameter indicates which axis of the array corresponds to channels.

New in version 0.19: *channel_axis* was added in 0.19.

Returns

out

[(..., 3, ...) ndarray] The image in sRGB color space, of same shape as input.

Raises

ValueError

If *lab* is not at least 2-D with shape (... , 3, ...).

See also:

`rgb2lab`

Notes

This function uses `lab2xyz()` and `xyz2rgb()`. The CIE XYZ tristimulus values are $x_{\text{ref}} = 95.047$, $y_{\text{ref}} = 100.$, and $z_{\text{ref}} = 108.883$. See function `xyz_tristimulus_values()` for a list of supported illuminants.

References

[?], [?]

`skimage.color.lab2xyz(lab, illuminant='D65', observer='2', *, channel_axis=-1)`

Convert image in CIE-LAB to XYZ color space.

Parameters

lab

[(..., 3, ...) array_like] The input image in CIE-LAB color space. Unless `channel_axis` is set, the final dimension denotes the CIE-LAB channels. The L* values range from 0 to 100; the a* and b* values range from -128 to 127.

illuminant

[{"A", "B", "C", "D50", "D55", "D65", "D75", "E"}, optional] The name of the illuminant (the function is NOT case sensitive).

observer

[{"2", "10", "R"}, optional] The aperture angle of the observer.

channel_axis

[int, optional] This parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns**out**

[(..., 3, ...) ndarray] The image in XYZ color space, of same shape as input.

Raises**ValueError**

If `lab` is not at least 2-D with shape (... , 3, ...).

ValueError

If either the illuminant or the observer angle are not supported or unknown.

UserWarning

If any of the pixels are invalid ($Z < 0$).

See also:**xyz2lab****Notes**

The CIE XYZ tristimulus values are `x_ref` = 95.047, `y_ref` = 100., and `z_ref` = 108.883. See function `xyz_tristimulus_values()` for a list of supported illuminants.

References

[?], [?]

```
skimage.color.label2rgb(label, image=None, colors=None, alpha=0.3, bg_label=0, bg_color=(0, 0, 0),
                        image_alpha=1, kind='overlay', *, saturation=0, channel_axis=-1)
```

Return an RGB image where color-coded labels are painted over the image.

Parameters**label**

[ndarray] Integer array of labels with the same shape as `image`.

image

[ndarray, optional] Image used as underlay for labels. It should have the same shape as *labels*, optionally with an additional RGB (channels) axis. If *image* is an RGB image, it is converted to grayscale before coloring.

colors

[list, optional] List of colors. If the number of labels exceeds the number of colors, then the colors are cycled.

alpha

[float [0, 1], optional] Opacity of colorized labels. Ignored if image is *None*.

bg_label

[int, optional] Label that's treated as the background. If *bg_label* is specified, *bg_color* is *None*, and *kind* is *overlay*, background is not painted by any colors.

bg_color

[str or array, optional] Background color. Must be a name in *color_dict* or RGB float values between [0, 1].

image_alpha

[float [0, 1], optional] Opacity of the image.

kind

[string, one of {‘overlay’, ‘avg’}] The kind of color image desired. ‘overlay’ cycles over defined colors and overlays the colored labels over the original image. ‘avg’ replaces each labeled segment with its average color, for a stained-class or pastel painting appearance.

saturation

[float [0, 1], optional] Parameter to control the saturation applied to the original image between fully saturated (original RGB, *saturation=1*) and fully unsaturated (grayscale, *saturation=0*). Only applies when *kind*=‘overlay’.

channel_axis

[int, optional] This parameter indicates which axis of the output array will correspond to channels. If *image* is provided, this must also match the axis of *image* that corresponds to channels.

New in version 0.19: *channel_axis* was added in 0.19.

Returns

result

[ndarray of float, same shape as *image*] The result of blending a cycling colormap (*colors*) for each distinct value in *label* with the image, at a certain alpha value.

- Local Binary Pattern for texture classification

- *RAG Thresholding*
 - *Normalized Cut*
 - *Find Regular Segments Using Compact Watershed*
 - *Expand segmentation labels without overlap*
 - *Label image regions*
 - *Find the intersection of two segmentations*
 - *RAG Merging*
 - *Hierarchical Merging of Region Boundary RAGs*
 - *Extrema*
 - *Use pixel graphs to find an object's geodesic center*
 - *Comparing edge-based and region-based segmentation*
 - *Segment human cells (in mitosis)*
-

```
skimage.color.lch2lab(lch, *, channel_axis=-1)
```

Convert image in CIE-LCh to CIE-LAB color space.

CIE-LCh is the cylindrical representation of the CIE-LAB (Cartesian) color space.

Parameters

lch

[(..., 3, ...) array_like] The input image in CIE-LCh color space. Unless *channel_axis* is set, the final dimension denotes the CIE-LAB channels. The L* values range from 0 to 100; the C values range from 0 to 100; the h values range from 0 to 2π .

channel_axis

[int, optional] This parameter indicates which axis of the array corresponds to channels.

New in version 0.19: *channel_axis* was added in 0.19.

Returns

out

[(..., 3, ...) ndarray] The image in CIE-LAB format, of same shape as input.

Raises

ValueError

If *lch* does not have at least 3 channels (i.e., L*, C, and h).

See also:

`lab2lch`

Notes

The h channel (i.e., hue) is expressed as an angle in range (0, 2*pi).

References

[?], [?], [?]

Examples

```
>>> from skimage import data
>>> from skimage.color import rgb2lab, lch2lab, lab2lch
>>> img = data.astronaut()
>>> img_lab = rgb2lab(img)
>>> img_lch = lab2lch(img_lab)
>>> img_lab2 = lch2lab(img_lch)
```

`skimage.color.luv2rgb(luv, *, channel_axis=-1)`

Luv to RGB color space conversion.

Parameters

luv

[(..., 3, ...) array_like] The image in CIE Luv format. By default, the final dimension denotes channels.

Returns

out

[(..., 3, ...) ndarray] The image in RGB format. Same dimensions as input.

Raises**ValueError**

If *luv* is not at least 2-D with shape (... , 3 , ...).

Notes

This function uses luv2xyz and xyz2rgb.

```
skimage.color.luv2xyz(luv, illuminant='D65', observer='2', *, channel_axis=-1)
```

CIE-Luv to XYZ color space conversion.

Parameters**luv**

[(... , 3 , ...) array_like] The image in CIE-Luv format. By default, the final dimension denotes channels.

illuminant

[{"A", "B", "C", "D50", "D55", "D65", "D75", "E"}, optional] The name of the illuminant (the function is NOT case sensitive).

observer

[{"2", "10", "R"}, optional] The aperture angle of the observer.

channel_axis

[int, optional] This parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns**out**

[(... , 3 , ...) ndarray] The image in XYZ format. Same dimensions as input.

Raises**ValueError**

If *luv* is not at least 2-D with shape (... , 3 , ...).

ValueError

If either the illuminant or the observer angle are not supported or unknown.

Notes

XYZ conversion weights use observer=2A. Reference whitepoint for D65 Illuminant, with XYZ tristimulus values of (95.047, 100., 108.883). See function `xyz_tristimulus_values()` for a list of supported illuminants.

References

[?], [?]

`skimage.color.rgb2gray(rgb, *, channel_axis=-1)`

Compute luminance of an RGB image.

Parameters**rgb**

[(..., 3, ...) array_like] The image in RGB format. By default, the final dimension denotes channels.

Returns**out**

[ndarray] The luminance image - an array which is the same size as the input array, but with the channel dimension removed.

Raises**ValueError**

If `rgb` is not at least 2-D with shape (... , 3, ...).

Notes

The weights used in this conversion are calibrated for contemporary CRT phosphors:

$$Y = 0.2125 R + 0.7154 G + 0.0721 B$$

If there is an alpha channel present, it is ignored.

References

[?]

Examples

```
>>> from skimage.color import rgb2gray
>>> from skimage import data
>>> img = data.astronaut()
>>> img_gray = rgb2gray(img)
```

- *Block views on images/arrays*
- *RGB to grayscale*
- *Adapting gray-scale filters to RGB images*
- *Ridge operators*
- *Active Contour Model*
- *Circular and Elliptical Hough Transforms*
- *Rescale, resize, and downscale*
- *Fundamental matrix estimation*
- *Robust matching using RANSAC*
- *Registration using optical flow*
- *Using Polar and Log-Polar Transformations for Registration*
- *Removing small objects in grayscale images with a top hat filter*
- *Image Deconvolution*
- *Using window functions with images*
- *Image Deconvolution*
- *Estimate strength of blur*
- *Phase Unwrapping*
- *Full tutorial on calibrating Denoisers Using J-Invariance*
- *CENSURE feature detector*
- *Blob Detection*
- *ORB feature detector and binary descriptor*

- *Gabors / Primary Visual Cortex “Simple Cells” from an Image*
 - *BRIEF binary descriptor*
 - *SIFT feature detector and descriptor extractor*
 - *Region Boundary based RAGs*
 - *Apply maskSLIC vs SLIC*
 - *Comparison of segmentation and superpixel algorithms*
 - *Hierarchical Merging of Region Boundary RAGs*
 - *Extrema*
 - *Use pixel graphs to find an object’s geodesic center*
-

`skimage.color.rgb2hed(rgb, *, channel_axis=-1)`

RGB to Haematoxylin-Eosin-DAB (HED) color space conversion.

Parameters

rgb

[(..., 3, ...) array_like] The image in RGB format. By default, the final dimension denotes channels.

channel_axis

[int, optional] This parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

out

[(..., 3, ...) ndarray] The image in HED format. Same dimensions as input.

Raises

ValueError

If `rgb` is not at least 2-D with shape (... , 3, ...).

References

[?]

Examples

```
>>> from skimage import data
>>> from skimage.color import rgb2hed
>>> ihc = data.immunohistochemistry()
>>> ihc_hed = rgb2hed(ihc)
```

- Separate colors in immunohistochemical staining

`skimage.color.rgb2hsv(rgb, *, channel_axis=-1)`

RGB to HSV color space conversion.

Parameters

`rgb`

[(..., 3, ...) array_like] The image in RGB format. By default, the final dimension denotes channels.

`channel_axis`

[int, optional] This parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

`out`

[(..., 3, ...) ndarray] The image in HSV format. Same dimensions as input.

Raises

`ValueError`

If `rgb` is not at least 2-D with shape (... , 3, ...).

Notes

Conversion between RGB and HSV color spaces results in some loss of precision, due to integer arithmetic and rounding [?].

References

[?]

Examples

```
>>> from skimage import color
>>> from skimage import data
>>> img = data.astronaut()
>>> img_hsv = color.rgb2hsv(img)
```

- *RGB to HSV*
- *Tinting gray-scale images*
- *Flood Fill*

```
skimage.color.rgb2lab(rgb, illuminant='D65', observer='2', *, channel_axis=-1)
```

Conversion from the sRGB color space (IEC 61966-2-1:1999) to the CIE Lab colorspace under the given illuminant and observer.

Parameters

rgb

[(..., 3, ...) array_like] The image in RGB format. By default, the final dimension denotes channels.

illuminant

[{"A", "B", "C", "D50", "D55", "D65", "D75", "E"}, optional] The name of the illuminant (the function is NOT case sensitive).

observer

[{"2", "10", "R"}, optional] The aperture angle of the observer.

channel_axis

[int, optional] This parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

out

[(..., 3, ...) ndarray] The image in Lab format. Same dimensions as input.

Raises**ValueError**

If *rgb* is not at least 2-D with shape (... , 3, ...).

Notes

RGB is a device-dependent color space so, if you use this function, be sure that the image you are analyzing has been mapped to the sRGB color space.

This function uses `rgb2xyz` and `xyz2lab`. By default `Observer="2"`, `Illuminant="D65"`. CIE XYZ tristimulus values `x_ref=95.047`, `y_ref=100.`, `z_ref=108.883`. See function `xyz_tristimulus_values()` for a list of supported illuminants.

References

[?]

`skimage.color.rgb2luv(rgb, *, channel_axis=-1)`

RGB to CIE-Luv color space conversion.

Parameters**rgb**

[(..., 3, ...) array_like] The image in RGB format. By default, the final dimension denotes channels.

channel_axis

[int, optional] This parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns**out**

[(..., 3, ...) ndarray] The image in CIE Luv format. Same dimensions as input.

Raises

ValueError

If *rgb* is not at least 2-D with shape (... , 3, ...).

Notes

This function uses `rgb2xyz` and `xyz2luv`.

References

[?], [?]

`skimage.color.rgb2rgbcie(rgb, *, channel_axis=-1)`

RGB to RGB CIE color space conversion.

Parameters

rgb

[(... , 3, ...) array_like] The image in RGB format. By default, the final dimension denotes channels.

channel_axis

[int, optional] This parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

out

[(... , 3, ...) ndarray] The image in RGB CIE format. Same dimensions as input.

Raises

ValueError

If *rgb* is not at least 2-D with shape (... , 3, ...).

References

[?]

Examples

```
>>> from skimage import data
>>> from skimage.color import rgb2rgbcie
>>> img = data.astronaut()
>>> img_rgbcie = rgb2rgbcie(img)
```

`skimage.color.rgb2xyz(rgb, *, channel_axis=-1)`

RGB to XYZ color space conversion.

Parameters

`rgb`

[..., 3, ...] The image in RGB format. By default, the final dimension denotes channels.

`channel_axis`

[int, optional] This parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

`out`

[..., 3, ...] The image in XYZ format. Same dimensions as input.

Raises

`ValueError`

If `rgb` is not at least 2-D with shape (... , 3, ...).

Notes

The CIE XYZ color space is derived from the CIE RGB color space. Note however that this function converts from sRGB.

References

[?]

Examples

```
>>> from skimage import data
>>> img = data.astronaut()
>>> img_xyz = rgb2xyz(img)
```

`skimage.color.rgb2ycbcr(rgb, *, channel_axis=-1)`

RGB to YCbCr color space conversion.

Parameters

`rgb`

[..., 3, ...] The image in RGB format. By default, the final dimension denotes channels.

`channel_axis`

[int, optional] This parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

`out`

[..., 3, ...] The image in YCbCr format. Same dimensions as input.

Raises

`ValueError`

If `rgb` is not at least 2-D with shape (... , 3, ...).

Notes

Y is between 16 and 235. This is the color space commonly used by video codecs; it is sometimes incorrectly called “YUV”.

References

[?]

`skimage.color.rgb2ydbdr(rgb, *, channel_axis=-1)`

RGB to YDbDr color space conversion.

Parameters

`rgb`

[(..., 3, ...) array_like] The image in RGB format. By default, the final dimension denotes channels.

`channel_axis`

[int, optional] This parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

`out`

[(..., 3, ...) ndarray] The image in YDbDr format. Same dimensions as input.

Raises

`ValueError`

If `rgb` is not at least 2-D with shape (... , 3, ...).

Notes

This is the color space commonly used by video codecs. It is also the reversible color transform in JPEG2000.

References

[?]

`skimage.color.rgb2yiq(rgb, *, channel_axis=-1)`

RGB to YIQ color space conversion.

Parameters

`rgb`

[(..., 3, ...) array_like] The image in RGB format. By default, the final dimension denotes channels.

`channel_axis`

[int, optional] This parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

`out`

[(..., 3, ...) ndarray] The image in YIQ format. Same dimensions as input.

Raises

`ValueError`

If `rgb` is not at least 2-D with shape (... , 3, ...).

`skimage.color.rgb2ypbpr(rgb, *, channel_axis=-1)`

RGB to YPbPr color space conversion.

Parameters

`rgb`

[(..., 3, ...) array_like] The image in RGB format. By default, the final dimension denotes channels.

`channel_axis`

[int, optional] This parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns**out**

[(..., 3, ...) ndarray] The image in YPbPr format. Same dimensions as input.

Raises**ValueError**

If *rgb* is not at least 2-D with shape (... , 3, ...).

References

[?]

`skimage.color.rgb2yuv(rgb, *, channel_axis=-1)`

RGB to YUV color space conversion.

Parameters**rgb**

[(..., 3, ...) array_like] The image in RGB format. By default, the final dimension denotes channels.

channel_axis

[int, optional] This parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns**out**

[(..., 3, ...) ndarray] The image in YUV format. Same dimensions as input.

Raises**ValueError**

If *rgb* is not at least 2-D with shape (... , 3, ...).

Notes

Y is between 0 and 1. Use YCbCr instead of YUV for the color space commonly used by video codecs, where Y ranges from 16 to 235.

References

[?]

`skimage.color.rgb2rgb(rgba, background=(1, 1, 1), *, channel_axis=-1)`

RGBA to RGB conversion using alpha blending [?].

Parameters

`rgba`

[(..., 4, ...) array_like] The image in RGBA format. By default, the final dimension denotes channels.

`background`

[array_like] The color of the background to blend the image with (3 floats between 0 to 1 - the RGB value of the background).

`channel_axis`

[int, optional] This parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

`out`

[(..., 3, ...) ndarray] The image in RGB format. Same dimensions as input.

Raises

`ValueError`

If `rgba` is not at least 2D with shape (... , 4, ...).

References

[?]

Examples

```
>>> from skimage import color
>>> from skimage import data
>>> img_rgba = data.logo()
>>> img_rgb = color.rgb2rgb(img_rgba)
```

```
skimage.color.rgb2rgb(rgbcie, *, channel_axis=-1)
```

RGB CIE to RGB color space conversion.

Parameters

rgbcie

[(..., 3, ...) array_like] The image in RGB CIE format. By default, the final dimension denotes channels.

channel_axis

[int, optional] This parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

out

[(..., 3, ...) ndarray] The image in RGB format. Same dimensions as input.

Raises

ValueError

If `rgbcie` is not at least 2-D with shape (... , 3, ...).

References

[?]

Examples

```
>>> from skimage import data
>>> from skimage.color import rgb2rgbcie, rgbcie2rgb
>>> img = data.astronaut()
>>> img_rgbcie = rgbcie2rgb(img)
>>> img_rgb = rgbcie2rgbcie(img_rgbcie)
```

`skimage.color.separate_stains`(*rgb*, *conv_matrix*, *, *channel_axis*=-1)

RGB to stain color space conversion.

Parameters

rgb

[..., 3, ...) array_like] The image in RGB format. By default, the final dimension denotes channels.

conv_matrix: ndarray

The stain separation matrix as described by G. Landini [?].

channel_axis

[int, optional] This parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

out

[..., 3, ...) ndarray] The image in stain color space. Same dimensions as input.

Raises

ValueError

If *rgb* is not at least 2-D with shape (... , 3, ...).

Notes

Stain separation matrices available in the `color` module and their respective colorspace:

- `hed_from_rgb`: Hematoxylin + Eosin + DAB
- `hdx_from_rgb`: Hematoxylin + DAB
- `fgx_from_rgb`: Feulgen + Light Green
- `bex_from_rgb`: Giemsa stain : Methyl Blue + Eosin
- `rbd_from_rgb`: FastRed + FastBlue + DAB
- `gdx_from_rgb`: Methyl Green + DAB
- `hax_from_rgb`: Hematoxylin + AEC
- `bro_from_rgb`: Blue matrix Aniline Blue + Red matrix Azocarmine + Orange matrix Orange-G
- `bpx_from_rgb`: Methyl Blue + Ponceau Fuchsin
- `ahx_from_rgb`: Alcian Blue + Hematoxylin
- `hpx_from_rgb`: Hematoxylin + PAS

This implementation borrows some ideas from DIPlib [?], e.g. the compensation using a small value to avoid log artifacts when calculating the Beer-Lambert law.

References

[?], [?], [?]

Examples

```
>>> from skimage import data
>>> from skimage.color import separate_stains, hdx_from_rgb
>>> ihc = data.immunohistochemistry()
>>> ihc_hdx = separate_stains(ihc, hdx_from_rgb)
```

`skimage.color.xyz2lab(xyz, illuminant='D65', observer='2', *, channel_axis=-1)`

XYZ to CIE-LAB color space conversion.

Parameters

xyz

[(..., 3, ...) array_like] The image in XYZ format. By default, the final dimension denotes channels.

illuminant

[{"A", "B", "C", "D50", "D55", "D65", "D75", "E"}, optional] The name of the illuminant (the function is NOT case sensitive).

observer

[{“2”, “10”, “R”}, optional] One of: 2-degree observer, 10-degree observer, or ‘R’ observer as in R function grDevices::convertColor.

channel_axis

[int, optional] This parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

out

[(..., 3, ...) ndarray] The image in CIE-LAB format. Same dimensions as input.

Raises

ValueError

If `xyz` is not at least 2-D with shape (... , 3, ...).

ValueError

If either the illuminant or the observer angle is unsupported or unknown.

Notes

By default `Observer=“2”`, `Illuminant=“D65”`. CIE XYZ tristimulus values `x_ref=95.047`, `y_ref=100.`, `z_ref=108.883`. See function `xyz_tristimulus_values()` for a list of supported illuminants.

References

[?], [?]

Examples

```
>>> from skimage import data
>>> from skimage.color import rgb2xyz, xyz2lab
>>> img = data.astronaut()
>>> img_xyz = rgb2xyz(img)
>>> img_lab = xyz2lab(img_xyz)
```

`skimage.color.xyz2luv(xyz, illuminant='D65', observer='2', *, channel_axis=-1)`

XYZ to CIE-Luv color space conversion.

Parameters

xyz

[(..., 3, ...) array_like] The image in XYZ format. By default, the final dimension denotes channels.

illuminant

[{"A", "B", "C", "D50", "D55", "D65", "D75", "E"}, optional] The name of the illuminant (the function is NOT case sensitive).

observer

[{"2", "10", "R"}, optional] The aperture angle of the observer.

channel_axis

[int, optional] This parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

out

[(..., 3, ...) ndarray] The image in CIE-Luv format. Same dimensions as input.

Raises

ValueError

If `xyz` is not at least 2-D with shape (... , 3, ...).

ValueError

If either the illuminant or the observer angle are not supported or unknown.

Notes

By default XYZ conversion weights use observer=2A. Reference whitepoint for D65 Illuminant, with XYZ tristimulus values of (95.047, 100., 108.883). See function `xyz_tristimulus_values()` for a list of supported illuminants.

References

[?], [?]

Examples

```
>>> from skimage import data
>>> from skimage.color import rgb2xyz, xyz2luv
>>> img = data.astronaut()
>>> img_xyz = rgb2xyz(img)
>>> img_luv = xyz2luv(img_xyz)
```

`skimage.color.xyz2rgb(xyz, *, channel_axis=-1)`

XYZ to RGB color space conversion.

Parameters

`xyz`

[..., 3, ...) array_like] The image in XYZ format. By default, the final dimension denotes channels.

`channel_axis`

[int, optional] This parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

`out`

[..., 3, ...) ndarray] The image in RGB format. Same dimensions as input.

Raises

`ValueError`

If `xyz` is not at least 2-D with shape (... , 3, ...).

Notes

The CIE XYZ color space is derived from the CIE RGB color space. Note however that this function converts to sRGB.

References

[?]

Examples

```
>>> from skimage import data
>>> from skimage.color import rgb2xyz, xyz2rgb
>>> img = data.astronaut()
>>> img_xyz = rgb2xyz(img)
>>> img_rgb = xyz2rgb(img_xyz)
```

`skimage.color.xyz_tristimulus_values(*, illuminant, observer, dtype=<class 'float'>)`

Get the CIE XYZ tristimulus values.

Given an illuminant and observer, this function returns the CIE XYZ tristimulus values [?] scaled such that $Y = 1$.

Parameters

illuminant

[{"A", "B", "C", "D50", "D55", "D65", "D75", "E"}] The name of the illuminant (the function is NOT case sensitive).

observer

[{"2", "10", "R"}] One of: 2-degree observer, 10-degree observer, or 'R' observer as in R function grDevices::convertColor [?].

dtype: dtype, optional

Output data type.

Returns

values

[array] Array with 3 elements X, Y, Z containing the CIE XYZ tristimulus values of the given illuminant.

Raises

ValueError

If either the illuminant or the observer angle are not supported or unknown.

Notes

The CIE XYZ tristimulus values are calculated from x, y [?], using the formula

$$X = x/y$$

$$Y = 1$$

$$Z = (1 - x - y)/y$$

The only exception is the illuminant “D65” with aperture angle 2° for backward-compatibility reasons.

References

[?], [?], [?]

Examples

Get the CIE XYZ tristimulus values for a “D65” illuminant for a 10 degree field of view

```
>>> xyz_tristimulus_values(illuminant="D65", observer="10")
array([0.94809668, 1. , 1.07305136])
```

```
skimage.color.ycbcr2rgb(ycbcr, *, channel_axis=-1)
```

YCbCr to RGB color space conversion.

Parameters

ycbcr

[..., 3, ...] array_like] The image in YCbCr format. By default, the final dimension denotes channels.

channel_axis

[int, optional] This parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

out

[..., 3, ...] ndarray] The image in RGB format. Same dimensions as input.

Raises**ValueError**

If $ycbcr$ is not at least 2-D with shape $(\dots, 3, \dots)$.

Notes

Y is between 16 and 235. This is the color space commonly used by video codecs; it is sometimes incorrectly called “YUV”.

References

[?]

`skimage.color.ydbdr2rgb(ydbdr, *, channel_axis=-1)`

YDbDr to RGB color space conversion.

Parameters**ydbdr**

[$(\dots, 3, \dots)$ array_like] The image in YDbDr format. By default, the final dimension denotes channels.

channel_axis

[int, optional] This parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns**out**

[$(\dots, 3, \dots)$ ndarray] The image in RGB format. Same dimensions as input.

Raises**ValueError**

If $ydbdr$ is not at least 2-D with shape $(\dots, 3, \dots)$.

Notes

This is the color space commonly used by video codecs, also called the reversible color transform in JPEG2000.

References

[?]

`skimage.color.yiq2rgb(yiq, *, channel_axis=-1)`

YIQ to RGB color space conversion.

Parameters

yiq

[(..., 3, ...) array_like] The image in YIQ format. By default, the final dimension denotes channels.

channel_axis

[int, optional] This parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

out

[(..., 3, ...) ndarray] The image in RGB format. Same dimensions as input.

Raises

ValueError

If `yiq` is not at least 2-D with shape (... , 3, ...).

`skimage.color.ypbpr2rgb(ypbpr, *, channel_axis=-1)`

YPbPr to RGB color space conversion.

Parameters

ypbpr

[(..., 3, ...) array_like] The image in YPbPr format. By default, the final dimension denotes channels.

channel_axis

[int, optional] This parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns**out**

[(..., 3, ...) ndarray] The image in RGB format. Same dimensions as input.

Raises**ValueError**

If `ypbpr` is not at least 2-D with shape (... , 3, ...).

References

[?]

skimage.color.yuv2rgb(yuv, *, channel_axis=-1)

YUV to RGB color space conversion.

Parameters**yuv**

[(..., 3, ...) array_like] The image in YUV format. By default, the final dimension denotes channels.

Returns**out**

[(..., 3, ...) ndarray] The image in RGB format. Same dimensions as input.

Raises**ValueError**

If `yuv` is not at least 2-D with shape (... , 3, ...).

References

[?]

1.3.3 `skimage.data`

Test images and datasets.

A curated set of general purpose and scientific images used in tests, examples, and documentation.

Newer datasets are no longer included as part of the package, but are downloaded on demand. To make data available offline, use `download_all()`.

<code>skimage.data.astronaut</code>	Color image of the astronaut Eileen Collins.
<code>skimage.data.binary_blobs</code>	Generate synthetic binary image with several rounded blob-like objects.
<code>skimage.data.brain</code>	Subset of data from the University of North Carolina Volume Rendering Test Data Set.
<code>skimage.data.brick</code>	Brick wall.
<code>skimage.data.camera</code>	Gray-level "camera" image.
<code>skimage.data.cat</code>	Chelsea the cat.
<code>skimage.data.cell</code>	Cell floating in saline.
<code>skimage.data.cells3d</code>	3D fluorescence microscopy image of cells.
<code>skimage.data.checkerboard</code>	Checkerboard image.
<code>skimage.data.chelsea</code>	Chelsea the cat.
<code>skimage.data.clock</code>	Motion blurred clock.
<code>skimage.data.coffee</code>	Coffee cup.
<code>skimage.data.coins</code>	Greek coins from Pompeii.
<code>skimage.data.colorwheel</code>	Color Wheel.
<code>skimage.data.download_all</code>	Download all datasets for use with scikit-image offline.
<code>skimage.data.eagle</code>	A golden eagle.
<code>skimage.data.file_hash</code>	Calculate the hash of a given file.
<code>skimage.data.grass</code>	Grass.
<code>skimage.data.gravel</code>	Gravel
<code>skimage.data.horse</code>	Black and white silhouette of a horse.
<code>skimage.data.hubble_deep_field</code>	Hubble eXtreme Deep Field.
<code>skimage.data.human_mitosis</code>	Image of human cells undergoing mitosis.
<code>skimage.data.immunohistochemistry</code>	Immunohistochemical (IHC) staining with hematoxylin counterstaining.
<code>skimage.data.kidney</code>	Mouse kidney tissue.
<code>skimage.data.lbp_frontal_face_cascade_filename</code>	Return the path to the XML file containing the weak classifier cascade.
<code>skimage.datalfw_subset</code>	Subset of data from the LFW dataset.
<code>skimage.data.lily</code>	Lily of the valley plant stem.
<code>skimage.data.logo</code>	Scikit-image logo, a RGBA image.
<code>skimage.data.microaneurysms</code>	Gray-level "microaneurysms" image.
<code>skimage.data.moon</code>	Surface of the moon.
<code>skimage.data.nickel_solidification</code>	Image sequence of synchrotron x-radiographs showing the rapid solidification of a nickel alloy sample.
<code>skimage.data.page</code>	Scanned page.
<code>skimage.data.protein_transport</code>	Microscopy image sequence with fluorescence tagging of proteins re-localizing from the cytoplasmic area to the nuclear envelope.

continues on next page

Table 4 – continued from previous page

<code>skimage.data.retina</code>	Human retina.
<code>skimage.data.rocket</code>	Launch photo of DSCOVR on Falcon 9 by SpaceX.
<code>skimage.data.shepp_logan_phantom</code>	Shepp Logan Phantom.
<code>skimage.data.skin</code>	Microscopy image of dermis and epidermis (skin layers).
<code>skimage.data.stereo_motorcycle</code>	Rectified stereo image pair with ground-truth disparities.
<code>skimage.data.text</code>	Gray-level "text" image used for corner detection.
<code>skimage.data.vortex</code>	Case B1 image pair from the first PIV challenge.

skimage.data.astronaut()

Color image of the astronaut Eileen Collins.

Photograph of Eileen Collins, an American astronaut. She was selected as an astronaut in 1992 and first piloted the space shuttle STS-63 in 1995. She retired in 2006 after spending a total of 38 days, 8 hours and 10 minutes in outer space.

This image was downloaded from the NASA Great Images database <<https://flic.kr/p/r9qvLn>>`__.

No known copyright restrictions, released into the public domain.

Returns**astronaut**

`[(512, 512, 3) uint8 ndarray]` Astronaut image.

- *General-purpose images*
- *Block views on images/arrays*
- *RGB to grayscale*
- *Adapting gray-scale filters to RGB images*
- *Active Contour Model*
- *Rescale, resize, and downscale*
- *Build image pyramids*
- *Piecewise Affine Transformation*
- *Image Deconvolution*
- *Using window functions with images*
- *Image Deconvolution*
- *Estimate strength of blur*
- *Inpainting*

- Non-local means denoising for preserving textures
 - Histogram of Oriented Gradients
 - CENSURE feature detector
 - ORB feature detector and binary descriptor
 - Gabors / Primary Visual Cortex “Simple Cells” from an Image
 - BRIEF binary descriptor
 - SIFT feature detector and descriptor extractor
 - Comparison of segmentation and superpixel algorithms
 - Flood Fill
 - Face detection using a cascade classifier
-

`skimage.data.binary_blobs(length=512, blob_size_fraction=0.1, n_dim=2, volume_fraction=0.5, rng=None)`

Generate synthetic binary image with several rounded blob-like objects.

Parameters

length

[int, optional] Linear size of output image.

blob_size_fraction

[float, optional] Typical linear size of blob, as a fraction of `length`, should be smaller than 1.

n_dim

[int, optional] Number of dimensions of output image.

volume_fraction

[float, default 0.5] Fraction of image pixels covered by the blobs (where the output is 1). Should be in [0, 1].

rng

[{`numpy.random.Generator`, int}, optional] Pseudo-random number generator. By default, a PCG64 generator is used (see `numpy.random.default_rng()`). If `rng` is an int, it is used to seed the generator.

Returns

blobs

[ndarray of bools] Output binary image

Other Parameters**seed**

[DEPRECATED] Deprecated in favor of *rng*.

Deprecated since version 0.21.

Examples

```
>>> from skimage import data
>>> data.binary_blobs(length=5, blob_size_fraction=0.2)
array([[ True, False,  True,  True,  True],
       [ True,  True,  True, False,  True],
       [False,  True, False,  True,  True],
       [ True, False, False,  True,  True],
       [ True, False, False, False,  True]])
>>> blobs = data.binary_blobs(length=256, blob_size_fraction=0.1)
>>> # Finer structures
>>> blobs = data.binary_blobs(length=256, blob_size_fraction=0.05)
>>> # Blobs cover a smaller volume fraction of the image
>>> blobs = data.binary_blobs(length=256, volume_fraction=0.3)
```

- *General-purpose images*
- *Skeletonize*
- *Random walker segmentation*
- *Explore and visualize region properties with pandas*
- *Colocalization metrics*

skimage.data.brain()

Subset of data from the University of North Carolina Volume Rendering Test Data Set.

The full dataset is available at [?].

Returns**image**

[(10, 256, 256) uint16 ndarray]

Notes

The 3D volume consists of 10 layers from the larger volume.

References

[?]

- *Local Histogram Equalization*
 - *Rank filters*
-

`skimage.data.brick()`

Brick wall.

Returns

brick

`[(512, 512) uint8 image]` A small section of a brick wall.

Notes

The original image was downloaded from [CC0Textures](#) and licensed under the Creative Commons CC0 License.

A perspective transform was then applied to the image, prior to rotating it by 90 degrees, cropping and scaling it to obtain the final image.

- *General-purpose images*
 - *Gabor filter banks for texture classification*
 - *Local Binary Pattern for texture classification*
-

`skimage.data.camera()`

Gray-level “camera” image.

Can be used for segmentation and denoising examples.

Returns

camera

`[(512, 512) uint8 ndarray]` Camera image.

Notes

No copyright restrictions. CC0 by the photographer (Lav Varshney).

Changed in version 0.18: This image was replaced due to copyright restrictions. For more information, please see [?].

References

[?]

- *General-purpose images*
- *Using simple NumPy operations for manipulating images*
- *Tinting gray-scale images*
- *Straight line Hough transform*
- *Edge operators*
- *Structural similarity index*
- *Image Registration*
- *Masked Normalized Cross-Correlation*
- *Entropy*
- *Band-pass filtering by Difference of Gaussians*
- *Butterworth Filters*
- *Dense DAISY feature description*
- *GLCM Texture Features*
- *Thresholding*
- *Chan-Vese Segmentation*
- *Multi-Otsu Thresholding*
- *Morphological Snakes*
- *Flood Fill*
- *Thresholding*
- *Rank filters*
- *Li thresholding*

skimage.data.cat()

Chelsea the cat.

An example with texture, prominent edges in horizontal and diagonal directions, as well as features of differing scales.

Returns

chelsea

[(300, 451, 3) uint8 ndarray] Chelsea image.

Notes

No copyright restrictions. CC0 by the photographer (Stefan van der Walt).

- *General-purpose images*
 - *Render text onto an image*
-

skimage.data.cell()

Cell floating in saline.

This is a quantitative phase image retrieved from a digital hologram using the Python library `qpformat`. The image shows a cell with high phase value, above the background phase.

Because of a banding pattern artifact in the background, this image is a good test of thresholding algorithms. The pixel spacing is 0.107 μm .

These data were part of a comparison between several refractive index retrieval techniques for spherical objects as part of [?].

This image is CC0, dedicated to the public domain. You may copy, modify, or distribute it without asking permission.

Returns

cell

[(660, 550) uint8 array] Image of a cell.

References

[?]

- *Li thresholding*
-

`skimage.data.cells3d()`

3D fluorescence microscopy image of cells.

The returned data is a 3D multichannel array with dimensions provided in (z, c, y, x) order. Each voxel has a size of (0.29 0.26 0.26) micrometer. Channel 0 contains cell membranes, channel 1 contains nuclei.

Returns

cells3d: (60, 2, 256, 256) uint16 ndarray

The volumetric images of cells taken with an optical microscope.

Notes

The data for this was provided by the Allen Institute for Cell Science.

It has been downsampled by a factor of 4 in the row and column dimensions to reduce computational time.

The microscope reports the following voxel spacing in microns:

- Original voxel size is (0.290, 0.065, 0.065).
 - Scaling factor is (1, 4, 4) in each dimension.
 - After rescaling the voxel size is (0.29 0.26 0.26).
 - *Datasets with 3 or more spatial dimensions*
 - *3D adaptive histogram equalization*
 - *Use rolling-ball algorithm for estimating background intensity*
 - *Explore 3D images (of cells)*
-

`skimage.data.checkerboard()`

Checkerboard image.

Checkerboards are often used in image calibration, since the corner-points are easy to locate. Because of the many parallel edges, they also visualise distortions particularly well.

Returns

checkerboard

[(200, 200) uint8 ndarray] Checkerboard image.

- *General-purpose images*
 - *Swirl*
 - *Robust matching using RANSAC*
 - *Corner detection*
 - *Flood Fill*
-

skimage.data.chelsea()

Chelsea the cat.

An example with texture, prominent edges in horizontal and diagonal directions, as well as features of differing scales.

Returns

chelsea

[(300, 451, 3) uint8 ndarray] Chelsea image.

Notes

No copyright restrictions. CC0 by the photographer (Stefan van der Walt).

- *Histogram matching*
 - *Types of homographies*
 - *Calibrating Denoisers Using J-Invariance*
 - *Denoising a picture*
 - *Shift-invariant wavelet denoising*
 - *Phase Unwrapping*
 - *Wavelet denoising*
 - *Full tutorial on calibrating Denoisers Using J-Invariance*
 - *Flood Fill*
-

skimage.data.clock()

Motion blurred clock.

This photograph of a wall clock was taken while moving the camera in an approximately horizontal direction. It may be used to illustrate inverse filters and deconvolution.

Released into the public domain by the photographer (Stefan van der Walt).

Returns

clock

[(300, 400) uint8 ndarray] Clock image.

- *General-purpose images*
-

skimage.data.coffee()

Coffee cup.

This photograph is courtesy of Pikolo Espresso Bar. It contains several elliptical shapes as well as varying texture (smooth porcelain to coarse wood grain).

Returns

coffee

[(400, 600, 3) uint8 ndarray] Coffee image.

Notes

No copyright restrictions. CC0 by the photographer (Rachel Michetti).

- *RGB to HSV*
- *Histogram matching*
- *Circular and Elliptical Hough Transforms*
- *Region Boundary based RAGs*
- *RAG Thresholding*
- *Normalized Cut*
- *Drawing Region Adjacency Graphs (RAGs)*
- *RAG Merging*
- *Hierarchical Merging of Region Boundary RAGs*

`skimage.data.coins()`

Greek coins from Pompeii.

This image shows several coins outlined against a gray background. It is especially useful in, e.g. segmentation tests, where individual objects need to be identified against a background. The background shares enough grey levels with the coins that a simple segmentation is not sufficient.

Returns

`coins`

`[(303, 384) uint8 ndarray]` Coins image.

Notes

This image was downloaded from the [Brooklyn Museum Collection](#).

No known copyright restrictions.

- *Filtering regional maxima*
- *Circular and Elliptical Hough Transforms*
- *Hysteresis thresholding*
- *Mean filters*
- *Template Matching*
- *Multi-Block Local Binary Pattern for texture classification*
- *Sliding window histogram*
- *Find Regular Segments Using Compact Watershed*
- *Finding local maxima*
- *Expand segmentation labels without overlap*
- *Label image regions*
- *Find the intersection of two segmentations*
- *Measure region properties*
- *Morphological Snakes*
- *Evaluating segmentation metrics*
- *Use rolling-ball algorithm for estimating background intensity*

- *Visual image comparison*
 - *Comparing edge-based and region-based segmentation*
-

skimage.data.colorwheel()

Color Wheel.

Returns

colorwheel

[(370, 371, 3) uint8 image] A colorwheel.

- *General-purpose images*
-

skimage.data.download_all(*directory=None*)

Download all datasets for use with scikit-image offline.

Scikit-image datasets are no longer shipped with the library by default. This allows us to use higher quality datasets, while keeping the library download size small.

This function requires the installation of an optional dependency, `pooch`, to download the full dataset. Follow installation instruction found at

<https://scikit-image.org/docs/stable/install.html>

Call this function to download all sample images making them available offline on your machine.

Parameters

directory: path-like, optional

The directory where the dataset should be stored.

Raises

ModuleNotFoundError:

If `pooch` is not install, this error will be raised.

Notes

scikit-image will only search for images stored in the default directory. Only specify the directory if you wish to download the images to your own folder for a particular reason. You can access the location of the default data directory by inspecting the variable `skimage.data.data_dir`.

`skimage.data.eagle()`

A golden eagle.

Suitable for examples on segmentation, Hough transforms, and corner detection.

Returns

eagle

[(2019, 1826) uint8 ndarray] Eagle image.

Notes

No copyright restrictions. CC0 by the photographer (Dayane Machado).

- *Markers for watershed transform*
-

`skimage.data.file_hash(fname, alg='sha256')`

Calculate the hash of a given file.

Useful for checking if a file has changed or been corrupted.

Parameters

fname

[str] The name of the file.

alg

[str] The type of the hashing algorithm

Returns

hash

[str] The hash of the file.

Examples

```
>>> fname = "test-file-for-hash.txt"
>>> with open(fname, "w") as f:
...     __ = f.write("content of the file")
>>> print(file_hash(fname))
0fc74468e6a9a829f103d069aeb2bb4f8646bad58bf146bb0e3379b759ec4a00
>>> import os
>>> os.remove(fname)
```

`skimage.data.grass()`

Grass.

Returns

`grass`

`[(512, 512) uint8 image]` Some grass.

Notes

The original image was downloaded from [DeviantArt](#) and licensed under the Creative Commons CC0 License.

The downloaded image was cropped to include a region of (512, 512) pixels around the top left corner, converted to grayscale, then to uint8 prior to saving the result in PNG format.

- *Gabor filter banks for texture classification*
 - *Local Binary Pattern for texture classification*
-

`skimage.data.gravel()`

Gravel

Returns

`gravel`

`[(512, 512) uint8 image]` Grayscale gravel sample.

Notes

The original image was downloaded from [CC0Textures](#) and licensed under the Creative Commons CC0 License.

The downloaded image was then rescaled to (1024, 1024), then the top left (512, 512) pixel region was cropped prior to converting the image to grayscale and uint8 data type. The result was saved using the PNG format.

- *Band-pass filtering by Difference of Gaussians*
 - *Gabor filter banks for texture classification*
 - *Local Binary Pattern for texture classification*
-

`skimage.data.horse()`

Black and white silhouette of a horse.

This image was downloaded from [openclipart](#)

No copyright restrictions. CC0 given by owner (Andreas Preuss (marauder)).

Returns

horse

[(328, 400) bool ndarray] Horse image.

- *Convex Hull*
 - *Skeletonize*
 - *Morphological Filtering*
-

`skimage.data.hubble_deep_field()`

Hubble eXtreme Deep Field.

This photograph contains the Hubble Telescope's farthest ever view of the universe. It can be useful as an example for multi-scale detection.

Returns

hubble_deep_field

[(872, 1000, 3) uint8 ndarray] Hubble deep field image.

Notes

This image was downloaded from [HubbleSite](#).

The image was captured by NASA and may be freely used in the public domain.

- *Scientific images*
 - *Removing small objects in grayscale images with a top hat filter*
 - *Full tutorial on calibrating Denoisers Using J-Invariance*
 - *Blob Detection*
 - *Extrema*
-

`skimage.data.human_mitosis()`

Image of human cells undergoing mitosis.

Returns

human_mitosis: (512, 512) uint8 ndarray

Data of human cells undergoing mitosis taken during the preparation of the manuscript in [\[?\]](#).

Notes

Copyright David Root. Licensed under CC-0 [\[?\]](#).

References

[\[?\]](#), [\[?\]](#)

- *Segment human cells (in mitosis)*
-

`skimage.data.immunohistochemistry()`

Immunohistochemical (IHC) staining with hematoxylin counterstaining.

This picture shows colonic glands where the IHC expression of FHL2 protein is revealed with DAB. Hematoxylin counterstaining is applied to enhance the negative parts of the tissue.

This image was acquired at the Center for Microscopy And Molecular Imaging (CMMI).

No known copyright restrictions.

Returns

immunohistochemistry

`[(512, 512, 3) uint8 ndarray]` Immunohistochemistry image.

- *Scientific images*
 - *Separate colors in immunohistochemical staining*
 - *Apply maskSLIC vs SLIC*
-

`skimage.data.kidney()`

Mouse kidney tissue.

This biological tissue on a pre-prepared slide was imaged with confocal fluorescence microscopy (Nikon C1 inverted microscope). Image shape is (16, 512, 512, 3). That is 512x512 pixels in X-Y, 16 image slices in Z, and 3 color channels (emission wavelengths 450nm, 515nm, and 605nm, respectively). Real-space voxel size is 1.24 microns in X-Y, and 1.25 microns in Z. Data type is unsigned 16-bit integers.

Returns

kidney

`[(16, 512, 512, 3) uint16 ndarray]` Kidney 3D multichannel image.

Notes

This image was acquired by Genevieve Buckley at Monash Micro Imaging in 2018. License: CC0

- *Interact with 3D images (of kidney tissue)*
 - *Estimate anisotropy in a 3D microscopy image*
-

`skimage.data.lbp_frontal_face_cascade_filename()`

Return the path to the XML file containing the weak classifier cascade.

These classifiers were trained using LBP features. The file is part of the OpenCV repository [?].

References

[?]

- *Face detection using a cascade classifier*
-

skimage.data.lfw_subset()

Subset of data from the LFW dataset.

This database is a subset of the LFW database containing:

- 100 faces
- 100 non-faces

The full dataset is available at [?].

Returns

images

`[(200, 25, 25) uint8 ndarray]` 100 first images are faces and subsequent 100 are non-faces.

Notes

The faces were randomly selected from the LFW dataset and the non-faces were extracted from the background of the same dataset. The cropped ROIs have been resized to a 25 x 25 pixels.

References

[?], [?]

- *Specific images*
 - *Face classification using Haar-like feature descriptor*
-

skimage.data.lily()

Lily of the valley plant stem.

This plant stem on a pre-prepared slide was imaged with confocal fluorescence microscopy (Nikon C1 inverted microscope). Image shape is (922, 922, 4). That is 922x922 pixels in X-Y, with 4 color channels. Real-space voxel size is 1.24 microns in X-Y. Data type is unsigned 16-bit integers.

Returns

lily

`[(922, 922, 4) uint16 ndarray]` Lily 2D multichannel image.

Notes

This image was acquired by Genevieve Buckley at Monasoh Micro Imaging in 2018. License: CC0

- *Scientific images*
-

`skimage.data.logo()`

Scikit-image logo, a RGBA image.

Returns

logo

`[(500, 500, 4) uint8 ndarray]` Logo image.

`skimage.data.microaneurysms()`

Gray-level “microaneurysms” image.

Detail from an image of the retina (green channel). The image is a crop of image 07_dr.JPG from the High-Resolution Fundus (HRF) Image Database: <https://www5.cs.fau.de/research/data/fundus-images/>

Returns

microaneurysms

`[(102, 102) uint8 ndarray]` Retina image with lesions.

Notes

No copyright restrictions. CC0 given by owner (Andreas Maier).

References

[?]

- *Scientific images*
 - *Attribute operators*
-

skimage.data.moon()

Surface of the moon.

This low-contrast image of the surface of the moon is useful for illustrating histogram equalization and contrast stretching.

Returns**moon**

[(512, 512) uint8 ndarray] Moon image.

- *Scientific images*
 - *Gamma and log contrast adjustment*
 - *Histogram Equalization*
 - *Local Histogram Equalization*
 - *Assemble images with simple image stitching*
 - *Unsharp masking*
 - *Filling holes and finding peaks*
-

skimage.data.nickel_solidification()

Image sequence of synchrotron x-radiographs showing the rapid solidification of a nickel alloy sample.

Returns**nickel_solidification: (11, 384, 512) uint16 ndarray****Notes**

See info under *nickel_solidification.tif* at <https://gitlab.com/scikit-image/data/-/blob/master/README.md#data>.

- *Track solidification of a metallic alloy*
-

skimage.data.page()

Scanned page.

This image of printed text is useful for demonstrations requiring uneven background illumination.

Returns

page

[(191, 384) uint8 ndarray] Page image.

- *Attribute operators*
 - *Thresholding*
 - *Niblack and Sauvola Thresholding*
 - *Use rolling-ball algorithm for estimating background intensity*
 - *Thresholding*
 - *Rank filters*
-

skimage.data.protein_transport()

Microscopy image sequence with fluorescence tagging of proteins re-localizing from the cytoplasmic area to the nuclear envelope.

Returns

protein_transport: (15, 2, 180, 183) uint8 ndarray

Notes

See info under *NPCsingleNucleus.tif* at <https://gitlab.com/scikit-image/data/-/blob/master/README.md#data>.

- *Colocalization metrics*
 - *Measure fluorescence intensity at the nuclear envelope*
-

skimage.data.retina()

Human retina.

This image of a retina is useful for demonstrations requiring circular images.

Returns

retina

[(1411, 1411, 3) uint8 ndarray] Retina image in RGB.

Notes

This image was downloaded from *wikimedia*. This file is made available under the Creative Commons CC0 1.0 Universal Public Domain Dedication.

References

[?]

- *Scientific images*
- *Ridge operators*
- *Using Polar and Log-Polar Transformations for Registration*
- *Use pixel graphs to find an object's geodesic center*

skimage.data.rocket()

Launch photo of DSCOVR on Falcon 9 by SpaceX.

This is the launch photo of Falcon 9 carrying DSCOVR lifted off from SpaceX's Launch Complex 40 at Cape Canaveral Air Force Station, FL.

Returns

rocket

[(427, 640, 3) uint8 ndarray] Rocket image.

Notes

This image was downloaded from [SpaceX Photos](#).

The image was captured by SpaceX and released in the public domain.

`skimage.data.shepp_logan_phantom()`

Shepp Logan Phantom.

Returns

phantom

`[(400, 400) float64 image]` Image of the Shepp-Logan phantom in grayscale.

References

[?]

- *Scientific images*
 - *Radon transform*
 - *Morphological Filtering*
-

`skimage.data.skin()`

Microscopy image of dermis and epidermis (skin layers).

Hematoxylin and eosin stained slide at 10x of normal epidermis and dermis with a benign intradermal nevus.

Returns

skin

`[(960, 1280, 3) RGB image of uint8]`

Notes

This image requires an Internet connection the first time it is called, and to have the `pooch` package installed, in order to fetch the image file from the scikit-image datasets repository.

The source of this image is https://en.wikipedia.org/wiki/File:Normal_Epidermis_and_Dermis_with_Intradermal_Nevus_10x.JPG

The image was released in the public domain by its author Kilbad.

- *Scientific images*

- *Trainable segmentation using local features and random forests*
-

skimage.data.stereo_motorcycle()

Rectified stereo image pair with ground-truth disparities.

The two images are rectified such that every pixel in the left image has its corresponding pixel on the same scanline in the right image. That means that both images are warped such that they have the same orientation but a horizontal spatial offset (baseline). The ground-truth pixel offset in column direction is specified by the included disparity map.

The two images are part of the Middlebury 2014 stereo benchmark. The dataset was created by Nera Nesic, Porter Westling, Xi Wang, York Kitajima, Greg Krathwohl, and Daniel Scharstein at Middlebury College. A detailed description of the acquisition process can be found in [?].

The images included here are down-sampled versions of the default exposure images in the benchmark. The images are down-sampled by a factor of 4 using the function `skimage.transform.downscale_local_mean`. The calibration data in the following and the included ground-truth disparity map are valid for the down-sampled images:

Focal length:	994.978px
Principal point x:	311.193px
Principal point y:	254.877px
Principal point dx:	31.086px
Baseline:	193.001mm

Returns

`img_left`

[(500, 741, 3) uint8 ndarray] Left stereo image.

`img_right`

[(500, 741, 3) uint8 ndarray] Right stereo image.

`disp`

[(500, 741, 3) float ndarray] Ground-truth disparity map, where each value describes the offset in column direction between corresponding pixels in the left and the right stereo images. E.g. the corresponding pixel of `img_left[10, 10 + disp[10, 10]]` is `img_right[10, 10]`. NaNs denote pixels in the left image that do not have ground-truth.

Notes

The original resolution images, images with different exposure and lighting, and ground-truth depth maps can be found at the Middlebury website [?].

References

[?], [?]

- *Specific images*
 - *Fundamental matrix estimation*
 - *Registration using optical flow*
-

`skimage.data.text()`

Gray-level “text” image used for corner detection.

Returns

`text`

`[(172, 448) uint8 ndarray]` Text image.

Notes

This image was downloaded from Wikipedia <<https://en.wikipedia.org/wiki/File:Corner.png>>`__.

No known copyright restrictions, released into the public domain.

- *Active Contour Model*
 - *Using geometric transformations*
-

`skimage.data.vortex()`

Case B1 image pair from the first PIV challenge.

Returns

`image0, image1`

`[(512, 512) grayscale images]` A pair of images featuring synthetic moving particles.

Notes

This image was licensed as CC0 by its author, Prof. Koji Okamoto, with thanks to Prof. Jun Sakakibara, who maintains the PIV Challenge site.

References

[?], [?]

- *Specific images*
- *Registration using optical flow*

1.3.4 skimage.draw

<code>skimage.draw._bezier_segment</code>	Generate Bezier segment coordinates.
<code>skimage.draw.bezier_curve</code>	Generate Bezier curve coordinates.
<code>skimage.draw.circle_perimeter</code>	Generate circle perimeter coordinates.
<code>skimage.draw.circle_perimeter_aa</code>	Generate anti-aliased circle perimeter coordinates.
<code>skimage.draw.disk</code>	Generate coordinates of pixels within circle.
<code>skimage.draw.ellipse</code>	Generate coordinates of pixels within ellipse.
<code>skimage.draw.ellipse_perimeter</code>	Generate ellipse perimeter coordinates.
<code>skimage.draw.ellipsoid</code>	Generates ellipsoid with semimajor axes aligned with grid dimensions on grid with specified <i>spacing</i> .
<code>skimage.draw.ellipsoid_stats</code>	Calculates analytical surface area and volume for ellipsoid with semimajor axes aligned with grid dimensions of specified <i>spacing</i> .
<code>skimage.draw.line</code>	Generate line pixel coordinates.
<code>skimage.draw.line_aa</code>	Generate anti-aliased line pixel coordinates.
<code>skimage.draw.line_nd</code>	Draw a single-pixel thick line in n dimensions.
<code>skimage.draw.polygon</code>	Generate coordinates of pixels inside a polygon.
<code>skimage.draw.polygon2mask</code>	Create a binary mask from a polygon.
<code>skimage.draw.polygon_perimeter</code>	Generate polygon perimeter coordinates.
<code>skimage.draw.random_shapes</code>	Generate an image with random shapes, labeled with bounding boxes.
<code>skimage.draw.rectangle</code>	Generate coordinates of pixels within a rectangle.
<code>skimage.draw.rectangle_perimeter</code>	Generate coordinates of pixels that are exactly around a rectangle.
<code>skimage.draw.set_color</code>	Set pixel color in the image at the given coordinates.

`skimage.draw._bezier_segment()`

Generate Bezier segment coordinates.

Parameters

r0, c0

[int] Coordinates of the first control point.

r1, c1

[int] Coordinates of the middle control point.

r2, c2

[int] Coordinates of the last control point.

weight

[cnp.float64_t] Middle control point weight, it describes the line tension.

Returns

rr, cc

[(N,) ndarray of int] Indices of pixels that belong to the Bezier curve. May be used to directly index into an array, e.g. `img[rr, cc] = 1`.

Notes

The algorithm is the rational quadratic algorithm presented in reference [?].

References

[?]

`skimage.draw.bezier_curve(r0, c0, r1, c1, r2, c2, weight, shape=None)`

Generate Bezier curve coordinates.

Parameters

r0, c0

[int] Coordinates of the first control point.

r1, c1

[int] Coordinates of the middle control point.

r2, c2

[int] Coordinates of the last control point.

weight

[double] Middle control point weight, it describes the line tension.

shape

[tuple, optional] Image shape which is used to determine the maximum extent of output

pixel coordinates. This is useful for curves that exceed the image size. If None, the full extent of the curve is used.

Returns

rr, cc

[(N,) ndarray of int] Indices of pixels that belong to the Bezier curve. May be used to directly index into an array, e.g. `img[rr, cc] = 1`.

Notes

The algorithm is the rational quadratic algorithm presented in reference [?].

References

[?]

Examples

```
>>> import numpy as np
>>> from skimage.draw import bezier_curve
>>> img = np.zeros((10, 10), dtype=np.uint8)
>>> rr, cc = bezier_curve(1, 5, 5, -2, 8, 8, 2)
>>> img[rr, cc] = 1
>>> img
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
       [0, 0, 0, 1, 1, 0, 0, 0, 0, 0],
       [0, 0, 1, 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, 1, 1, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 1, 1, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 1, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]], dtype=uint8)
```

- *Shapes*

`skimage.draw.circle_perimeter(r, c, radius, method='bresenham', shape=None)`

Generate circle perimeter coordinates.

Parameters

r, c

[int] Centre coordinate of circle.

radius

[int] Radius of circle.

method

[{‘bresenham’, ‘andres’}, optional] bresenham : Bresenham method (default) andres : Andres method

shape

[tuple, optional] Image shape which is used to determine the maximum extent of output pixel coordinates. This is useful for circles that exceed the image size. If None, the full extent of the circle is used. Must be at least length 2. Only the first two values are used to determine the extent of the input image.

Returns**rr, cc**

[(N,) ndarray of int] Bresenham and Andres’ method: Indices of pixels that belong to the circle perimeter. May be used to directly index into an array, e.g. `img[rr, cc] = 1`.

Notes

Andres method presents the advantage that concentric circles create a disc whereas Bresenham can make holes. There is also less distortions when Andres circles are rotated. Bresenham method is also known as midpoint circle algorithm. Anti-aliased circle generator is available with `circle_perimeter_aa`.

References

[?], [?]

Examples

```
>>> from skimage.draw import circle_perimeter
>>> img = np.zeros((10, 10), dtype=np.uint8)
>>> rr, cc = circle_perimeter(4, 4, 3)
>>> img[rr, cc] = 1
>>> img
array([[0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 1, 0, 0, 0, 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, 1, 0],
       [0, 1, 0, 0, 0, 0, 0, 1, 0],
       [0, 0, 1, 0, 0, 0, 1, 0, 0],
```

(continues on next page)

(continued from previous page)

```
[0, 0, 0, 1, 1, 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]], dtype=uint8)
```

- *Shapes*
- *Circular and Elliptical Hough Transforms*

```
skimage.draw.circle_perimeter_aa(r, c, radius, shape=None)
```

Generate anti-aliased circle perimeter coordinates.

Parameters

r, c

[int] Centre coordinate of circle.

radius

[int] Radius of circle.

shape

[tuple, optional] Image shape which is used to determine the maximum extent of output pixel coordinates. This is useful for circles that exceed the image size. If None, the full extent of the circle is used. Must be at least length 2. Only the first two values are used to determine the extent of the input image.

Returns

rr, cc, val

[(N,) ndarray (int, int, float)] Indices of pixels (*rr*, *cc*) and intensity values (*val*). `img[rr, cc] = val`.

Notes

Wu's method draws anti-aliased circle. This implementation doesn't use lookup table optimization.

Use the function `draw.set_color` to apply `circle_perimeter_aa` results to color images.

References

[?]

Examples

```
>>> from skimage.draw import circle_perimeter_aa
>>> img = np.zeros((10, 10), dtype=np.uint8)
>>> rr, cc, val = circle_perimeter_aa(4, 4, 3)
>>> img[rr, cc] = val * 255
>>> img
array([[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
       [ 0,  0, 60, 211, 255, 211, 60,  0,  0,  0],
       [ 0, 60, 194, 43,  0, 43, 194, 60,  0,  0],
       [ 0, 211, 43,  0,  0, 43, 211,  0,  0],
       [ 0, 255,  0,  0,  0,  0, 255,  0,  0],
       [ 0, 211, 43,  0,  0, 43, 211,  0,  0],
       [ 0, 60, 194, 43,  0, 43, 194, 60,  0,  0],
       [ 0,  0, 60, 211, 255, 211, 60,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0]], dtype=uint8)
```

```
>>> from skimage import data, draw
>>> image = data.chelsea()
>>> rr, cc, val = draw.circle_perimeter_aa(r=100, c=100, radius=75)
>>> draw.set_color(image, (rr, cc), [1, 0, 0], alpha=val)
```

- *Shapes*
-

`skimage.draw.disk(center, radius, *, shape=None)`

Generate coordinates of pixels within circle.

Parameters

`center`

[tuple] Center coordinate of disk.

`radius`

[double] Radius of disk.

`shape`

[tuple, optional] Image shape as a tuple of size 2. Determines the maximum extent of output pixel coordinates. This is useful for disks that exceed the image size. If None, the full extent of the disk is used. The shape might result in negative coordinates and wraparound behaviour.

Returns**rr, cc**

[ndarray of int] Pixel coordinates of disk. May be used to directly index into an array, e.g.
`img[rr, cc] = 1.`

Examples

```
>>> import numpy as np
>>> from skimage.draw import disk
>>> shape = (4, 4)
>>> img = np.zeros(shape, dtype=np.uint8)
>>> rr, cc = disk((0, 0), 2, shape=shape)
>>> img[rr, cc] = 1
>>> img
array([[1, 1, 0, 0],
       [1, 1, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]], dtype=uint8)
>>> img = np.zeros(shape, dtype=np.uint8)
>>> # Negative coordinates in rr and cc perform a wraparound
>>> rr, cc = disk((0, 0), 2, shape=None)
>>> img[rr, cc] = 1
>>> img
array([[1, 1, 0, 1],
       [1, 1, 0, 1],
       [0, 0, 0, 0],
       [1, 1, 0, 1]], dtype=uint8)
>>> img = np.zeros((10, 10), dtype=np.uint8)
>>> rr, cc = disk((4, 4), 5)
>>> img[rr, cc] = 1
>>> img
array([[0, 0, 1, 1, 1, 1, 1, 0, 0, 0],
       [0, 1, 1, 1, 1, 1, 1, 0, 0, 0],
       [1, 1, 1, 1, 1, 1, 1, 1, 0, 0],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
       [0, 1, 1, 1, 1, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 1, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]], dtype=uint8)
```

- *Shapes*
- *Shape Index*

`skimage.draw.ellipse(r, c, r_radius, c_radius, shape=None, rotation=0.0)`

Generate coordinates of pixels within ellipse.

Parameters

r, c

[double] Centre coordinate of ellipse.

r_radius, c_radius

[double] Minor and major semi-axes. $(r/r_radius)^{**2} + (c/c_radius)^{**2} = 1$.

shape

[tuple, optional] Image shape which is used to determine the maximum extent of output pixel coordinates. This is useful for ellipses which exceed the image size. By default the full extent of the ellipse are used. Must be at least length 2. Only the first two values are used to determine the extent.

rotation

[float, optional (default 0.)] Set the ellipse rotation (rotation) in range (-PI, PI) in contra clock wise direction, so PI/2 degree means swap ellipse axis

Returns

rr, cc

[ndarray of int] Pixel coordinates of ellipse. May be used to directly index into an array, e.g. `img[rr, cc] = 1`.

Notes

The ellipse equation:

$$\begin{aligned} ((x * \cos(\alpha) + y * \sin(\alpha)) / x_radius)^{** 2} + \\ ((x * \sin(\alpha) - y * \cos(\alpha)) / y_radius)^{** 2} = 1 \end{aligned}$$

Note that the positions of `ellipse` without specified `shape` can have also, negative values, as this is correct on the plane. On the other hand using these ellipse positions for an image afterwards may lead to appearing on the other side of image, because `image[-1, -1] = image[end-1, end-1]`

```
>>> rr, cc = ellipse(1, 2, 3, 6)
>>> img = np.zeros((6, 12), dtype=np.uint8)
>>> img[rr, cc] = 1
>>> img
array([[1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1],
```

(continues on next page)

(continued from previous page)

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[1, 1, 1, 1, 1, 0, 0, 0, 1, 1]], dtype=uint8)
```

Examples

```
>>> from skimage.draw import ellipse
>>> img = np.zeros((10, 12), dtype=np.uint8)
>>> rr, cc = ellipse(5, 6, 3, 5, rotation=np.deg2rad(30))
>>> img[rr, cc] = 1
>>> img
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0],
       [0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
       [0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0],
       [0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]], dtype=uint8)
```

- *Shapes*
- *Approximate and subdivide polygons*
- *Masked Normalized Cross-Correlation*
- *Corner detection*
- *Measure region properties*

`skimage.draw.ellipse_perimeter(r, c, r_radius, c_radius, orientation=0, shape=None)`

Generate ellipse perimeter coordinates.

Parameters

r, c

[int] Centre coordinate of ellipse.

r_radius, c_radius

[int] Minor and major semi-axes. $(r/r_radius)^{**2} + (c/c_radius)^{**2} = 1$.

orientation

[double, optional] Major axis orientation in clockwise direction as radians.

shape

[tuple, optional] Image shape which is used to determine the maximum extent of output pixel coordinates. This is useful for ellipses that exceed the image size. If None, the full extent of the ellipse is used. Must be at least length 2. Only the first two values are used to determine the extent of the input image.

Returns**rr, cc**

[(N,) ndarray of int] Indices of pixels that belong to the ellipse perimeter. May be used to directly index into an array, e.g. `img[rr, cc] = 1`.

References

[?]

Examples

```
>>> from skimage.draw import ellipse_perimeter
>>> img = np.zeros((10, 10), dtype=np.uint8)
>>> rr, cc = ellipse_perimeter(5, 5, 3, 4)
>>> img[rr, cc] = 1
>>> img
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 1, 1, 1, 0, 0, 0],
       [0, 0, 1, 0, 0, 0, 0, 1, 0, 0],
       [0, 1, 0, 0, 0, 0, 0, 0, 0, 1],
       [0, 1, 0, 0, 0, 0, 0, 0, 0, 1],
       [0, 1, 0, 0, 0, 0, 0, 0, 0, 1],
       [0, 1, 0, 0, 0, 0, 0, 0, 0, 1],
       [0, 0, 1, 0, 0, 0, 0, 0, 1, 0],
       [0, 0, 0, 1, 1, 1, 1, 0, 0, 0]], dtype=uint8)
```

Note that the positions of `ellipse` without specified `shape` can have also, negative values, as this is correct on the plane. On the other hand using these ellipse positions for an image afterwards may lead to appearing on the other side of image, because `image[-1, -1] = image[end-1, end-1]`

```
>>> rr, cc = ellipse_perimeter(2, 3, 4, 5)
>>> img = np.zeros((9, 12), dtype=np.uint8)
>>> img[rr, cc] = 1
>>> img
array([[0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1],
       [0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0],
       [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0],
       [1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
```

(continues on next page)

(continued from previous page)

```
[0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0],  
[0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0],  
[1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]], dtype=uint8)
```

- *Shapes*
 - *Circular and Elliptical Hough Transforms*
-

`skimage.draw.ellipsoid(a, b, c, spacing=(1.0, 1.0, 1.0), levelset=False)`

Generates ellipsoid with semimajor axes aligned with grid dimensions on grid with specified *spacing*.

Parameters

a

[float] Length of semimajor axis aligned with x-axis.

b

[float] Length of semimajor axis aligned with y-axis.

c

[float] Length of semimajor axis aligned with z-axis.

spacing

[tuple of floats, length 3] Spacing in (x, y, z) spatial dimensions.

levelset

[bool] If True, returns the level set for this ellipsoid (signed level set about zero, with positive denoting interior) as np.float64. False returns a binarized version of said level set.

Returns

ellip

[(N, M, P) array] Ellipsoid centered in a correctly sized array for given *spacing*. Boolean dtype unless *levelset=True*, in which case a float array is returned with the level set above 0.0 representing the ellipsoid.

- *Marching Cubes*
-

`skimage.draw.ellipsoid_stats(a, b, c)`

Calculates analytical surface area and volume for ellipsoid with semimajor axes aligned with grid dimensions of specified *spacing*.

Parameters

a

[float] Length of semimajor axis aligned with x-axis.

b

[float] Length of semimajor axis aligned with y-axis.

c

[float] Length of semimajor axis aligned with z-axis.

Returns

vol

[float] Calculated volume of ellipsoid.

surf

[float] Calculated surface area of ellipsoid.

`skimage.draw.line(r0, c0, r1, c1)`

Generate line pixel coordinates.

Parameters

r0, c0

[int] Starting position (row, column).

r1, c1

[int] End position (row, column).

Returns

rr, cc

[(N,) ndarray of int] Indices of pixels that belong to the line. May be used to directly index into an array, e.g. `img[rr, cc] = 1`.

Notes

Anti-aliased line generator is available with `line_aa`.

Examples

```
>>> from skimage.draw import line
>>> img = np.zeros((10, 10), dtype=np.uint8)
>>> rr, cc = line(1, 1, 8, 8)
>>> img[rr, cc] = 1
>>> img
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]], dtype=uint8)
```

- *Shapes*
- *Straight line Hough transform*

`skimage.draw.line_aa(r0, c0, r1, c1)`

Generate anti-aliased line pixel coordinates.

Parameters

r0, c0

[int] Starting position (row, column).

r1, c1

[int] End position (row, column).

Returns

rr, cc, val

[(N,) ndarray (int, int, float)] Indices of pixels (*rr*, *cc*) and intensity values (*val*). `img[rr, cc] = val`.

References

[?]

Examples

```
>>> from skimage.draw import line_aa
>>> img = np.zeros((10, 10), dtype=np.uint8)
>>> rr, cc, val = line_aa(1, 1, 8, 8)
>>> img[rr, cc] = val * 255
>>> img
array([[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
       [ 0, 255, 74,  0,  0,  0,  0,  0,  0,  0],
       [ 0, 74, 255, 74,  0,  0,  0,  0,  0,  0],
       [ 0, 0, 74, 255, 74,  0,  0,  0,  0,  0],
       [ 0, 0, 0, 74, 255, 74,  0,  0,  0,  0],
       [ 0, 0, 0, 0, 74, 255, 74,  0,  0,  0],
       [ 0, 0, 0, 0, 0, 74, 255, 74,  0,  0],
       [ 0, 0, 0, 0, 0, 0, 74, 255, 74,  0],
       [ 0, 0, 0, 0, 0, 0, 0, 74, 255, 0],
       [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]], dtype=uint8)
```

- *Shapes*
-

`skimage.draw.line_nd(start, stop, *, endpoint=False, integer=True)`

Draw a single-pixel thick line in n dimensions.

The line produced will be ndim-connected. That is, two subsequent pixels in the line will be either direct or diagonal neighbors in n dimensions.

Parameters

start

[array-like, shape (N,)] The start coordinates of the line.

stop

[array-like, shape (N,)] The end coordinates of the line.

endpoint

[bool, optional] Whether to include the endpoint in the returned line. Defaults to False, which allows for easy drawing of multi-point paths.

integer

[bool, optional] Whether to round the coordinates to integer. If True (default), the returned coordinates can be used to directly index into an array. *False* could be used for e.g. vector drawing.

Returns**coords**

[tuple of arrays] The coordinates of points on the line.

Examples

```
>>> lin = line_nd((1, 1), (5, 2.5), endpoint=False)
>>> lin
(array([1, 2, 3, 4]), array([1, 1, 2, 2]))
>>> im = np.zeros((6, 5), dtype=int)
>>> im[lin] = 1
>>> im
array([[0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0],
       [0, 1, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0]])
>>> line_nd([2, 1, 1], [5, 5, 2.5], endpoint=True)
(array([2, 3, 4, 4, 5]), array([1, 2, 3, 4, 5]), array([1, 1, 2, 2, 2]))
```

`skimage.draw.polygon(r, c, shape=None)`

Generate coordinates of pixels inside a polygon.

Parameters**r**

[(N,) array_like] Row coordinates of the polygon's vertices.

c

[(N,) array_like] Column coordinates of the polygon's vertices.

shape

[tuple, optional] Image shape which is used to determine the maximum extent of output pixel coordinates. This is useful for polygons that exceed the image size. If None, the full extent of the polygon is used. Must be at least length 2. Only the first two values are used to determine the extent of the input image.

Returns**rr, cc**

[ndarray of int] Pixel coordinates of polygon. May be used to directly index into an array, e.g. `img[rr, cc] = 1`.

See also:

`polygon2mask`

Create a binary mask from a polygon.

Notes

This function ensures that `rr` and `cc` don't contain negative values. Pixels of the polygon that whose coordinates are smaller 0, are not drawn.

Examples

```
>>> import skimage as ski
>>> r = np.array([1, 2, 8])
>>> c = np.array([1, 7, 4])
>>> rr, cc = ski.draw.polygon(r, c)
>>> img = np.zeros((10, 10), dtype=int)
>>> img[rr, cc] = 1
>>> img
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 1, 1, 1, 0, 0, 0],
       [0, 0, 1, 1, 1, 1, 1, 0, 0, 0],
       [0, 0, 0, 1, 1, 1, 1, 0, 0, 0],
       [0, 0, 0, 1, 1, 1, 1, 0, 0, 0],
       [0, 0, 0, 1, 1, 1, 1, 0, 0, 0],
       [0, 0, 0, 0, 1, 1, 0, 0, 0, 0],
       [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
```

If the image `shape` is defined and vertices / points of the `polygon` are outside this coordinate space, only a part (or none at all) of the polygon's pixels is returned. Shifting the polygon's vertices by an offset can be used to move the polygon around and potentially draw an arbitrary sub-region of the polygon.

```
>>> offset = (2, -4)
>>> rr, cc = ski.draw.polygon(r - offset[0], c - offset[1], shape=img.shape)
>>> img = np.zeros((10, 10), dtype=int)
>>> img[rr, cc] = 1
>>> img
array([[0, 0, 0, 0, 0, 0, 1, 1, 1, 1],
       [0, 0, 0, 0, 0, 0, 1, 1, 1, 1],
       [0, 0, 0, 0, 0, 0, 0, 1, 1, 1],
       [0, 0, 0, 0, 0, 0, 0, 1, 1, 1],
       [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, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
```

(continues on next page)

(continued from previous page)

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],  
[0, 0, 0, 0, 0, 0, 0, 0]])
```

- *Shapes*
-

`skimage.draw.polygon2mask(image_shape, polygon)`

Create a binary mask from a polygon.

Parameters

`image_shape`

[tuple of size 2] The shape of the mask.

`polygon`

[(N, 2) array_like] The polygon coordinates of shape (N, 2) where N is the number of points.

Returns

`mask`

[2-D ndarray of type ‘bool’] The binary mask that corresponds to the input polygon.

See also:

`polygon`

Generate coordinates of pixels inside a polygon.

Notes

This function does not do any border checking. Parts of the polygon that are outside the coordinate space defined by `image_shape` are not drawn.

Examples

```
>>> import skimage as ski  
>>> image_shape = (10, 10)  
>>> polygon = np.array([[1, 1], [2, 7], [8, 4]])  
>>> mask = ski.draw.polygon2mask(image_shape, polygon)  
>>> mask.astype(int)  
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],  
       [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],  
       [0, 0, 1, 1, 1, 1, 1, 0, 0, 0],
```

(continues on next page)

(continued from previous page)

```
[0, 0, 1, 1, 1, 1, 1, 0, 0, 0],
[0, 0, 0, 1, 1, 1, 1, 0, 0, 0],
[0, 0, 0, 1, 1, 1, 0, 0, 0, 0],
[0, 0, 0, 0, 1, 1, 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])
```

If vertices / points of the *polygon* are outside the coordinate space defined by *image_shape*, only a part (or none at all) of the polygon is drawn in the mask.

```
>>> offset = np.array([[2, -4]])
>>> ski.draw.polygon2mask(image_shape, polygon - offset).astype(int)
array([[0, 0, 0, 0, 0, 0, 1, 1, 1, 1],
       [0, 0, 0, 0, 0, 0, 1, 1, 1, 1],
       [0, 0, 0, 0, 0, 0, 0, 1, 1, 1],
       [0, 0, 0, 0, 0, 0, 0, 1, 1, 1],
       [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, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
```

skimage.draw.polygon_perimeter(*r*, *c*, *shape=None*, *clip=False*)

Generate polygon perimeter coordinates.

Parameters

r

[(N,) ndarray] Row coordinates of vertices of polygon.

c

[(N,) ndarray] Column coordinates of vertices of polygon.

shape

[tuple, optional] Image shape which is used to determine maximum extents of output pixel coordinates. This is useful for polygons that exceed the image size. If None, the full extents of the polygon is used. Must be at least length 2. Only the first two values are used to determine the extent of the input image.

clip

[bool, optional] Whether to clip the polygon to the provided shape. If this is set to True, the drawn figure will always be a closed polygon with all edges visible.

Returns**rr, cc**

[ndarray of int] Pixel coordinates of polygon. May be used to directly index into an array, e.g. `img[rr, cc] = 1`.

Examples

```
>>> from skimage.draw import polygon_perimeter
>>> img = np.zeros((10, 10), dtype=np.uint8)
>>> rr, cc = polygon_perimeter([5, -1, 5, 10],
...                             [-1, 5, 11, 5],
...                             shape=img.shape, clip=True)
>>> img[rr, cc] = 1
>>> img
array([[0, 0, 0, 0, 1, 1, 1, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0, 1, 0, 0],
       [0, 0, 1, 0, 0, 0, 0, 0, 1, 0],
       [0, 1, 0, 0, 0, 0, 0, 0, 0, 1],
       [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
       [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
       [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
       [0, 1, 1, 0, 0, 0, 0, 0, 0, 1],
       [0, 0, 0, 1, 0, 0, 0, 1, 1, 0],
       [0, 0, 0, 0, 1, 1, 1, 0, 0, 0]], dtype=uint8)
```

`skimage.draw.random_shapes(image_shape, max_shapes, min_shapes=1, min_size=2, max_size=None, num_channels=3, shape=None, intensity_range=None, allow_overlap=False, num_trials=100, rng=None, *, channel_axis=-1)`

Generate an image with random shapes, labeled with bounding boxes.

The image is populated with random shapes with random sizes, random locations, and random colors, with or without overlap.

Shapes have random (row, col) starting coordinates and random sizes bounded by `min_size` and `max_size`. It can occur that a randomly generated shape will not fit the image at all. In that case, the algorithm will try again with new starting coordinates a certain number of times. However, it also means that some shapes may be skipped altogether. In that case, this function will generate fewer shapes than requested.

Parameters**image_shape**

[tuple] The number of rows and columns of the image to generate.

max_shapes

[int] The maximum number of shapes to (attempt to) fit into the shape.

min_shapes

[int, optional] The minimum number of shapes to (attempt to) fit into the shape.

min_size

[int, optional] The minimum dimension of each shape to fit into the image.

max_size

[int, optional] The maximum dimension of each shape to fit into the image.

num_channels

[int, optional] Number of channels in the generated image. If 1, generate monochrome images, else color images with multiple channels. Ignored if `multichannel` is set to False.

shape

[{rectangle, circle, triangle, ellipse, None} str, optional] The name of the shape to generate or `None` to pick random ones.

intensity_range

[{tuple of tuples of uint8, tuple of uint8}, optional] The range of values to sample pixel values from. For grayscale images the format is (min, max). For multichannel - ((min, max),) if the ranges are equal across the channels, and ((min_0, max_0), ... (min_N, max_N)) if they differ. As the function supports generation of uint8 arrays only, the maximum range is (0, 255). If `None`, set to (0, 254) for each channel reserving color of intensity = 255 for background.

allow_overlap

[bool, optional] If `True`, allow shapes to overlap.

num_trials

[int, optional] How often to attempt to fit a shape into the image before skipping it.

rng

[{`numpy.random.Generator`, int}, optional] Pseudo-random number generator. By default, a PCG64 generator is used (see `numpy.random.default_rng()`). If `rng` is an int, it is used to seed the generator.

channel_axis

[int or `None`, optional] If `None`, the image is assumed to be a grayscale (single channel) image. Otherwise, this parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

image

[`uint8` array] An image with the fitted shapes.

labels

[list] A list of labels, one per shape in the image. Each label is a (category, ((r0, r1), (c0, c1))) tuple specifying the category and bounding box coordinates of the shape.

Other Parameters**random_seed**

[DEPRECATED] Deprecated in favor of *rng*.

Deprecated since version 0.21.

Examples

```
>>> import skimage.draw
>>> image, labels = skimage.draw.random_shapes((32, 32), max_shapes=3)
>>> image
array([
[[255, 255, 255],
 [255, 255, 255],
 [255, 255, 255],
 ...,
 [255, 255, 255],
 [255, 255, 255],
 [255, 255, 255]]], dtype=uint8)
>>> labels
[('circle', ((22, 18), (25, 21))), ('triangle', ((5, 6), (13, 13)))]
```

- *Random Shapes*

skimage.draw.rectangle(*start*, *end=None*, *extent=None*, *shape=None*)

Generate coordinates of pixels within a rectangle.

Parameters**start**

[tuple] Origin point of the rectangle, e.g., ([*plane*,] *row*, *column*).

end

[tuple] End point of the rectangle ([*plane*,] *row*, *column*). For a 2D matrix, the slice defined by the rectangle is [*start*:(*end*+1)]. Either *end* or *extent* must be specified.

extent

[tuple] The extent (size) of the drawn rectangle. E.g., ([*num_planes*,] *num_rows*,

`num_cols`). Either `end` or `extent` must be specified. A negative extent is valid, and will result in a rectangle going along the opposite direction. If extent is negative, the `start` point is not included.

shape

[tuple, optional] Image shape used to determine the maximum bounds of the output coordinates. This is useful for clipping rectangles that exceed the image size. By default, no clipping is done.

Returns

coords

[array of int, shape (Ndim, Npoints)] The coordinates of all pixels in the rectangle.

Notes

This function can be applied to N-dimensional images, by passing `start` and `end` or `extent` as tuples of length N.

Examples

```
>>> import numpy as np
>>> from skimage.draw import rectangle
>>> img = np.zeros((5, 5), dtype=np.uint8)
>>> start = (1, 1)
>>> extent = (3, 3)
>>> rr, cc = rectangle(start, extent=extent, shape=img.shape)
>>> img[rr, cc] = 1
>>> img
array([[0, 0, 0, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 0, 0, 0]], dtype=uint8)
```

```
>>> img = np.zeros((5, 5), dtype=np.uint8)
>>> start = (0, 1)
>>> end = (3, 3)
>>> rr, cc = rectangle(start, end=end, shape=img.shape)
>>> img[rr, cc] = 1
>>> img
array([[0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 0, 0, 0]], dtype=uint8)
```

```
>>> import numpy as np
>>> from skimage.draw import rectangle
```

(continues on next page)

(continued from previous page)

```
>>> img = np.zeros((6, 6), dtype=np.uint8)
>>> start = (3, 3)
>>>
>>> rr, cc = rectangle(start, extent=(2, 2))
>>> img[rr, cc] = 1
>>> rr, cc = rectangle(start, extent=(-2, 2))
>>> img[rr, cc] = 2
>>> rr, cc = rectangle(start, extent=(-2, -2))
>>> img[rr, cc] = 3
>>> rr, cc = rectangle(start, extent=(2, -2))
>>> img[rr, cc] = 4
>>> print(img)
[[0 0 0 0 0 0]
 [0 3 3 2 2 0]
 [0 3 3 2 2 0]
 [0 4 4 1 1 0]
 [0 4 4 1 1 0]
 [0 0 0 0 0 0]]
```

`skimage.draw.rectangle_perimeter`(*start*, *end=None*, *extent=None*, *shape=None*, *clip=False*)

Generate coordinates of pixels that are exactly around a rectangle.

Parameters

start

[tuple] Origin point of the inner rectangle, e.g., (row, column).

end

[tuple] End point of the inner rectangle (row, column). For a 2D matrix, the slice defined by inner the rectangle is [start:(end+1)]. Either *end* or *extent* must be specified.

extent

[tuple] The extent (size) of the inner rectangle. E.g., (num_rows, num_cols). Either *end* or *extent* must be specified. Negative extents are permitted. See `rectangle` to better understand how they behave.

shape

[tuple, optional] Image shape used to determine the maximum bounds of the output coordinates. This is useful for clipping perimeters that exceed the image size. By default, no clipping is done. Must be at least length 2. Only the first two values are used to determine the extent of the input image.

clip

[bool, optional] Whether to clip the perimeter to the provided shape. If this is set to True, the drawn figure will always be a closed polygon with all edges visible.

Returns**coords**

[array of int, shape (2, Npoints)] The coordinates of all pixels in the rectangle.

Examples

```
>>> import numpy as np
>>> from skimage.draw import rectangle_perimeter
>>> img = np.zeros((5, 6), dtype=np.uint8)
>>> start = (2, 3)
>>> end = (3, 4)
>>> rr, cc = rectangle_perimeter(start, end=end, shape=img.shape)
>>> img[rr, cc] = 1
>>> img
array([[0, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 1, 0],
       [0, 0, 1, 0, 0, 1],
       [0, 0, 1, 0, 0, 1],
       [0, 0, 1, 1, 1, 0]], dtype=uint8)
```

```
>>> img = np.zeros((5, 5), dtype=np.uint8)
>>> r, c = rectangle_perimeter(start, (10, 10), shape=img.shape, clip=True)
>>> img[r, c] = 1
>>> img
array([[0, 0, 0, 0, 0],
       [0, 0, 1, 1, 0],
       [0, 0, 1, 0, 1],
       [0, 0, 1, 0, 1],
       [0, 0, 1, 1, 0]], dtype=uint8)
```

`skimage.draw.set_color(image, coords, color, alpha=1)`

Set pixel color in the image at the given coordinates.

Note that this function modifies the color of the image in-place. Coordinates that exceed the shape of the image will be ignored.

Parameters**image**

[(M, N, D) ndarray] Image

coords

[tuple of ((P,) ndarray, (P,) ndarray)] Row and column coordinates of pixels to be colored.

color

[(D,) ndarray] Color to be assigned to coordinates in the image.

alpha

[scalar or (N,) ndarray] Alpha values used to blend color with image. 0 is transparent, 1 is opaque.

Examples

```
>>> from skimage.draw import line, set_color
>>> img = np.zeros((10, 10), dtype=np.uint8)
>>> rr, cc = line(1, 1, 20, 20)
>>> set_color(img, (rr, cc), 1)
>>> img
array([[0, 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, 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]], dtype=uint8)
```

1.3.5 skimage.exposure

<code>skimage.exposure.adjust_gamma</code>	Performs Gamma Correction on the input image.
<code>skimage.exposure.adjust_log</code>	Performs Logarithmic correction on the input image.
<code>skimage.exposure.adjust_sigmoid</code>	Performs Sigmoid Correction on the input image.
<code>skimage.exposure.cumulative_distribution</code>	Return cumulative distribution function (cdf) for the given image.
<code>skimage.exposure.equalize_adapthist</code>	Contrast Limited Adaptive Histogram Equalization (CLAHE).
<code>skimage.exposure.equalize_hist</code>	Return image after histogram equalization.
<code>skimage.exposure.histogram</code>	Return histogram of image.
<code>skimage.exposure.is_low_contrast</code>	Determine if an image is low contrast.
<code>skimage.exposure.match_histograms</code>	Adjust an image so that its cumulative histogram matches that of another.
<code>skimage.exposure.rescale_intensity</code>	Return image after stretching or shrinking its intensity levels.

`skimage.exposure.adjust_gamma(image, gamma=1, gain=1)`

Performs Gamma Correction on the input image.

Also known as Power Law Transform. This function transforms the input image pixelwise according to the equation $O = I^{**\gamma}$ after scaling each pixel to the range 0 to 1.

Parameters

image

[ndarray] Input image.

gamma

[float, optional] Non negative real number. Default value is 1.

gain

[float, optional] The constant multiplier. Default value is 1.

Returns

out

[ndarray] Gamma corrected output image.

See also:

[adjust_log](#)

Notes

For gamma greater than 1, the histogram will shift towards left and the output image will be darker than the input image.

For gamma less than 1, the histogram will shift towards right and the output image will be brighter than the input image.

References

[?]

Examples

```
>>> from skimage import data, exposure, img_as_float
>>> image = img_as_float(data.moon())
>>> gamma_corrected = exposure.adjust_gamma(image, 2)
>>> # Output is darker for gamma > 1
>>> image.mean() > gamma_corrected.mean()
True
```

- *Gamma and log contrast adjustment*

- *Explore 3D images (of cells)*

`skimage.exposure.adjust_log(image, gain=1, inv=False)`

Performs Logarithmic correction on the input image.

This function transforms the input image pixelwise according to the equation $O = \text{gain} * \log(1 + I)$ after scaling each pixel to the range 0 to 1. For inverse logarithmic correction, the equation is $O = \text{gain} * (2^{**I} - 1)$.

Parameters

image

[ndarray] Input image.

gain

[float, optional] The constant multiplier. Default value is 1.

inv

[float, optional] If True, it performs inverse logarithmic correction, else correction will be logarithmic. Defaults to False.

Returns

out

[ndarray] Logarithm corrected output image.

See also:

[**adjust_gamma**](#)

References

[?]

- *Gamma and log contrast adjustment*

`skimage.exposure.adjust_sigmoid(image, cutoff=0.5, gain=10, inv=False)`

Performs Sigmoid Correction on the input image.

Also known as Contrast Adjustment. This function transforms the input image pixelwise according to the equation $O = 1/(1 + \exp^{*(\text{gain}*(\text{cutoff} - I))})$ after scaling each pixel to the range 0 to 1.

Parameters

image

[ndarray] Input image.

cutoff

[float, optional] Cutoff of the sigmoid function that shifts the characteristic curve in horizontal direction. Default value is 0.5.

gain

[float, optional] The constant multiplier in exponential's power of sigmoid function. Default value is 10.

inv

[bool, optional] If True, returns the negative sigmoid correction. Defaults to False.

Returns

out

[ndarray] Sigmoid corrected output image.

See also:

[*adjust_gamma*](#)

References

[?]

`skimage.exposure.cumulative_distribution(image, nbins=256)`

Return cumulative distribution function (cdf) for the given image.

Parameters

image

[array] Image array.

nbins

[int, optional] Number of bins for image histogram.

Returns**img_cdf**

[array] Values of cumulative distribution function.

bin_centers

[array] Centers of bins.

See also:***histogram*****References**

[?]

Examples

```
>>> from skimage import data, exposure, img_as_float
>>> image = img_as_float(data.camera())
>>> hi = exposure.histogram(image)
>>> cdf = exposure.cumulative_distribution(image)
>>> all(cdf[0] == np.cumsum(hi[0])/float(image.size))
True
```

- *Histogram matching*
- *Gamma and log contrast adjustment*
- *Histogram Equalization*
- *Local Histogram Equalization*
- *Explore 3D images (of cells)*

`skimage.exposure.equalize_adapthist(image, kernel_size=None, clip_limit=0.01, nbins=256)`

Contrast Limited Adaptive Histogram Equalization (CLAHE).

An algorithm for local contrast enhancement, that uses histograms computed over different tile regions of the image. Local details can therefore be enhanced even in regions that are darker or lighter than most of the image.

Parameters

image

[(N1, ..., NN[, C]) ndarray] Input image.

kernel_size

[int or array_like, optional] Defines the shape of contextual regions used in the algorithm. If iterable is passed, it must have the same number of elements as `image.ndim` (without color channel). If integer, it is broadcasted to each `image` dimension. By default, `kernel_size` is 1/8 of `image` height by 1/8 of its width.

clip_limit

[float, optional] Clipping limit, normalized between 0 and 1 (higher values give more contrast).

nbins

[int, optional] Number of gray bins for histogram (“data range”).

Returns

out

[(N1, ..., NN[, C]) ndarray] Equalized image with float64 dtype.

See also:

`equalize_hist`, `rescale_intensity`

Notes

•**For color images, the following steps are performed:**

- The image is converted to HSV color space
 - The CLAHE algorithm is run on the V (Value) channel
 - The image is converted back to RGB space and returned
- For RGBA images, the original alpha channel is removed.

Changed in version 0.17: The values returned by this function are slightly shifted upwards because of an internal change in rounding behavior.

References

[?], [?]

- *Histogram Equalization*
 - *3D adaptive histogram equalization*
-

```
skimage.exposure.equalize_hist(image, nbins=256, mask=None)
```

Return image after histogram equalization.

Parameters

image

[array] Image array.

nbins

[int, optional] Number of bins for image histogram. Note: this argument is ignored for integer images, for which each integer is its own bin.

mask

[ndarray of bools or 0s and 1s, optional] Array of same shape as *image*. Only points at which mask == True are used for the equalization, which is applied to the whole image.

Returns

out

[float array] Image array after histogram equalization.

Notes

This function is adapted from [?] with the author's permission.

References

[?], [?]

- *Histogram Equalization*
- *Local Histogram Equalization*
- *3D adaptive histogram equalization*
- *Visual image comparison*

- Explore 3D images (of cells)
 - Rank filters
-

```
skimage.exposure.histogram(image, nbins=256, source_range='image', normalize=False, *,  
                           channel_axis=None)
```

Return histogram of image.

Unlike `numpy.histogram`, this function returns the centers of bins and does not rebin integer arrays. For integer arrays, each integer value has its own bin, which improves speed and intensity-resolution.

If `channel_axis` is not set, the histogram is computed on the flattened image. For color or multichannel images, set `channel_axis` to use a common binning for all channels. Alternatively, one may apply the function separately on each channel to obtain a histogram for each color channel with separate binning.

Parameters

image

[array] Input image.

nbins

[int, optional] Number of bins used to calculate histogram. This value is ignored for integer arrays.

source_range

[string, optional] ‘image’ (default) determines the range from the input image. ‘dtype’ determines the range from the expected range of the images of that data type.

normalize

[bool, optional] If True, normalize the histogram by the sum of its values.

channel_axis

[int or None, optional] If None, the image is assumed to be a grayscale (single channel) image. Otherwise, this parameter indicates which axis of the array corresponds to channels.

Returns

hist

[array] The values of the histogram. When `channel_axis` is not None, `hist` will be a 2D array where the first axis corresponds to channels.

bin_centers

[array] The values at the center of the bins.

See also:

cumulative_distribution**Examples**

```
>>> from skimage import data, exposure, img_as_float
>>> image = img_as_float(data.camera())
>>> np.histogram(image, bins=2)
(array([ 93585, 168559]), array([0. , 0.5, 1. ]))
>>> exposure.histogram(image, nbins=2)
(array([ 93585, 168559]), array([0.25, 0.75]))
```

- *Histogram matching*
 - *Comparing edge-based and region-based segmentation*
 - *Rank filters*
-

`skimage.exposure.is_low_contrast(image, fraction_threshold=0.05, lower_percentile=1, upper_percentile=99, method='linear')`

Determine if an image is low contrast.

Parameters**image**

[array-like] The image under test.

fraction_threshold

[float, optional] The low contrast fraction threshold. An image is considered low- contrast when its range of brightness spans less than this fraction of its data type's full range. [?]

lower_percentile

[float, optional] Disregard values below this percentile when computing image contrast.

upper_percentile

[float, optional] Disregard values above this percentile when computing image contrast.

method

[str, optional] The contrast determination method. Right now the only available option is “linear”.

Returns

out

[bool] True when the image is determined to be low contrast.

Notes

For boolean images, this function returns False only if all values are the same (the method, threshold, and percentile arguments are ignored).

References

[?]

Examples

```
>>> image = np.linspace(0, 0.04, 100)
>>> is_low_contrast(image)
True
>>> image[-1] = 1
>>> is_low_contrast(image)
True
>>> is_low_contrast(image, upper_percentile=100)
False
```

`skimage.exposure.match_histograms(image, reference, *, channel_axis=None)`

Adjust an image so that its cumulative histogram matches that of another.

The adjustment is applied separately for each channel.

Parameters

image

[ndarray] Input image. Can be gray-scale or in color.

reference

[ndarray] Image to match histogram of. Must have the same number of channels as image.

channel_axis

[int or None, optional] If None, the image is assumed to be a grayscale (single channel) image. Otherwise, this parameter indicates which axis of the array corresponds to channels.

Returns

matched

[ndarray] Transformed input image.

Raises

ValueError

Thrown when the number of channels in the input image and the reference differ.

References

[?]

- *Histogram matching*

```
skimage.exposure.rescale_intensity(image, in_range='image', out_range='dtype')
```

Return image after stretching or shrinking its intensity levels.

The desired intensity range of the input and output, *in_range* and *out_range* respectively, are used to stretch or shrink the intensity range of the input image. See examples below.

Parameters

image

[array] Image array.

in_range, out_range

[str or 2-tuple, optional] Min and max intensity values of input and output image. The possible values for this parameter are enumerated below.

‘image’

Use image min/max as the intensity range.

‘dtype’

Use min/max of the image’s dtype as the intensity range.

dtype-name

Use intensity range based on desired *dtype*. Must be valid key in *DTYPE_RANGE*.

2-tuple

Use *range_values* as explicit min/max intensities.

Returns

out

[array] Image array after rescaling its intensity. This image is the same dtype as the input image.

See also:

[equalize_hist](#)

Notes

Changed in version 0.17: The dtype of the output array has changed to match the input dtype, or float if the output range is specified by a pair of values.

Examples

By default, the min/max intensities of the input image are stretched to the limits allowed by the image's dtype, since *in_range* defaults to 'image' and *out_range* defaults to 'dtype':

```
>>> image = np.array([51, 102, 153], dtype=np.uint8)
>>> rescale_intensity(image)
array([ 0, 127, 255], dtype=uint8)
```

It's easy to accidentally convert an image dtype from uint8 to float:

```
>>> 1.0 * image
array([ 51., 102., 153.])
```

Use *rescale_intensity* to rescale to the proper range for float dtypes:

```
>>> image_float = 1.0 * image
>>> rescale_intensity(image_float)
array([0., 0.5, 1.])
```

To maintain the low contrast of the original, use the *in_range* parameter:

```
>>> rescale_intensity(image_float, in_range=(0, 255))
array([0.2, 0.4, 0.6])
```

If the min/max value of *in_range* is more/less than the min/max image intensity, then the intensity levels are clipped:

```
>>> rescale_intensity(image_float, in_range=(0, 102))
array([0.5, 1., 1.])
```

If you have an image with signed integers but want to rescale the image to just the positive range, use the *out_range* parameter. In that case, the output dtype will be float:

```
>>> image = np.array([-10, 0, 10], dtype=np.int8)
>>> rescale_intensity(image, out_range=(0, 127))
array([ 0., 63.5, 127.])
```

To get the desired range with a specific dtype, use `.astype()`:

```
>>> rescale_intensity(image, out_range=(0, 127)).astype(np.int8)
array([ 0, 63, 127], dtype=int8)
```

If the input image is constant, the output will be clipped directly to the output range: >>> image = np.array([130, 130, 130], dtype=np.int32) >>> rescale_intensity(image, out_range=(0, 127)).astype(np.int32) array([127, 127, 127], dtype=int32)

- *Adapting gray-scale filters to RGB images*
- *Separate colors in immunohistochemical staining*
- *Histogram Equalization*
- *Robust matching using RANSAC*
- *Phase Unwrapping*
- *Histogram of Oriented Gradients*
- *Filling holes and finding peaks*
- *Random walker segmentation*
- *Extrema*
- *Explore 3D images (of cells)*
- *Rank filters*

1.3.6 skimage.feature

<code>skimage.feature.blob_dog</code>	Finds blobs in the given grayscale image.
<code>skimage.feature.blob_doh</code>	Finds blobs in the given grayscale image.
<code>skimage.feature.blob_log</code>	Finds blobs in the given grayscale image.
<code>skimage.feature.canny</code>	Edge filter an image using the Canny algorithm.
<code>skimage.feature.corner_fast</code>	Extract FAST corners for a given image.
<code>skimage.feature.corner_foerstner</code>	Compute Foerstner corner measure response image.
<code>skimage.feature.corner_harris</code>	Compute Harris corner measure response image.
<code>skimage.feature.corner_kitchen_rosenfeld</code>	Compute Kitchen and Rosenfeld corner measure response image.
<code>skimage.feature.corner_moravec</code>	Compute Moravec corner measure response image.
<code>skimage.feature.corner_orientations</code>	Compute the orientation of corners.
<code>skimage.feature.corner_peaks</code>	Find peaks in corner measure response image.
<code>skimage.feature.corner_shi_tomasi</code>	Compute Shi-Tomasi (Kanade-Tomasi) corner measure response image.
<code>skimage.feature.corner_subpix</code>	Determine subpixel position of corners.
<code>skimage.feature.daisy</code>	Extract DAISY feature descriptors densely for the given image.
<code>skimage.feature.draw_haar_like_feature</code>	Visualization of Haar-like features.
<code>skimage.feature.draw_multiblock_lbp</code>	Multi-block local binary pattern visualization.

continues on next page

Table 5 – continued from previous page

<code>skimage.feature.fisher_vector</code>	Compute the Fisher vector given some descriptors/vectors, and an associated estimated GMM.
<code>skimage.feature.graycomatrix</code>	Calculate the gray-level co-occurrence matrix.
<code>skimage.feature.grycoprops</code>	Calculate texture properties of a GLCM.
<code>skimage.feature.haar_like_feature</code>	Compute the Haar-like features for a region of interest (ROI) of an integral image.
<code>skimage.feature.haar_like_feature_coord</code>	Compute the coordinates of Haar-like features.
<code>skimage.feature.hessian_matrix</code>	Compute the Hessian matrix.
<code>skimage.feature.hessian_matrix_det</code>	Compute the approximate Hessian Determinant over an image.
<code>skimage.feature.hessian_matrix_eigvals</code>	Compute eigenvalues of Hessian matrix.
<code>skimage.feature.hog</code>	Extract Histogram of Oriented Gradients (HOG) for a given image.
<code>skimage.feature.learn_gmm</code>	Estimate a Gaussian mixture model (GMM) given a set of descriptors and number of modes (i.e.
<code>skimage.feature.local_binary_pattern</code>	Compute the local binary patterns (LBP) of an image.
<code>skimage.feature.match_descriptors</code>	Brute-force matching of descriptors.
<code>skimage.feature.match_template</code>	Match a template to a 2-D or 3-D image using normalized correlation.
<code>skimage.feature.multiblock_lbp</code>	Multi-block local binary pattern (MB-LBP).
<code>skimage.feature.multiscale_basic_features</code>	Local features for a single- or multi-channel nd image.
<code>skimage.feature.peak_local_max</code>	Find peaks in an image as coordinate list.
<code>skimage.feature.plot_matches</code>	Plot matched features.
<code>skimage.feature.shape_index</code>	Compute the shape index.
<code>skimage.feature.structure_tensor</code>	Compute structure tensor using sum of squared differences.
<code>skimage.feature.structure_tensor_eigenvalue</code>	Compute eigenvalues of structure tensor.
<code>skimage.feature.BRIEF</code>	BRIEF binary descriptor extractor.
<code>skimage.feature.CENSURE</code>	CENSURE keypoint detector.
<code>skimage.feature.Cascade</code>	Class for cascade of classifiers that is used for object detection.
<code>skimage.feature.ORB</code>	Oriented FAST and rotated BRIEF feature detector and binary descriptor extractor.
<code>skimage.feature.SIFT</code>	SIFT feature detection and descriptor extraction.

`skimage.feature.blob_dog(image, min_sigma=1, max_sigma=50, sigma_ratio=1.6, threshold=0.5, overlap=0.5, *, threshold_rel=None, exclude_border=False)`

Finds blobs in the given grayscale image.

Blobs are found using the Difference of Gaussian (DoG) method [?], [?]. For each blob found, the method returns its coordinates and the standard deviation of the Gaussian kernel that detected the blob.

Parameters

`image`

[ndarray] Input grayscale image, blobs are assumed to be light on dark background (white on black).

`min_sigma`

[scalar or sequence of scalars, optional] The minimum standard deviation for Gaussian kernel. Keep this low to detect smaller blobs. The standard deviations of the Gaussian filter are given for each axis as a sequence, or as a single number, in which case it is equal for all axes.

max_sigma

[scalar or sequence of scalars, optional] The maximum standard deviation for Gaussian kernel. Keep this high to detect larger blobs. The standard deviations of the Gaussian filter are given for each axis as a sequence, or as a single number, in which case it is equal for all axes.

sigma_ratio

[float, optional] The ratio between the standard deviation of Gaussian Kernels used for computing the Difference of Gaussians

threshold

[float or None, optional] The absolute lower bound for scale space maxima. Local maxima smaller than *threshold* are ignored. Reduce this to detect blobs with lower intensities. If *threshold_rel* is also specified, whichever threshold is larger will be used. If None, *threshold_rel* is used instead.

overlap

[float, optional] A value between 0 and 1. If the area of two blobs overlaps by a fraction greater than *threshold*, the smaller blob is eliminated.

threshold_rel

[float or None, optional] Minimum intensity of peaks, calculated as `max(dog_space) * threshold_rel`, where `dog_space` refers to the stack of Difference-of-Gaussian (DoG) images computed internally. This should have a value between 0 and 1. If None, *threshold* is used instead.

exclude_border

[tuple of ints, int, or False, optional] If tuple of ints, the length of the tuple must match the input array's dimensionality. Each element of the tuple will exclude peaks from within *exclude_border*-pixels of the border of the image along that dimension. If nonzero int, *exclude_border* excludes peaks from within *exclude_border*-pixels of the border of the image. If zero or False, peaks are identified regardless of their distance from the border.

Returns

A

[(n, image.ndim + sigma) ndarray] A 2d array with each row representing 2 coordinate values for a 2D image, or 3 coordinate values for a 3D image, plus the sigma(s) used. When a single sigma is passed, outputs are: (r, c, sigma) or (p, r, c, sigma) where (r, c) or (p, r, c) are coordinates of the blob and sigma is the standard deviation of the Gaussian kernel which detected the blob. When an anisotropic gaussian is used (sigmas per dimension), the detected sigma is returned for each dimension.

See also:

`skimage.filters.difference_of_gaussians`**Notes**

The radius of each blob is approximately $\sqrt{2}\sigma$ for a 2-D image and $\sqrt{3}\sigma$ for a 3-D image.

References

[?], [?]

Examples

```
>>> from skimage import data, feature
>>> coins = data.coins()
>>> feature.blob_dog(coins, threshold=.05, min_sigma=10, max_sigma=40)
array([[128., 155., 10.],
       [198., 155., 10.],
       [124., 338., 10.],
       [127., 102., 10.],
       [193., 281., 10.],
       [126., 208., 10.],
       [267., 115., 10.],
       [197., 102., 10.],
       [198., 215., 10.],
       [123., 279., 10.],
       [126., 46., 10.],
       [259., 247., 10.],
       [196., 43., 10.],
       [54., 276., 10.],
       [267., 358., 10.],
       [58., 100., 10.],
       [259., 305., 10.],
       [185., 347., 16.],
       [261., 174., 16.],
       [46., 336., 16.],
       [54., 217., 10.],
       [55., 157., 10.],
       [57., 41., 10.],
       [260., 47., 16.]])
```

- *Blob Detection*
-

`skimage.feature.blob_doh(image, min_sigma=1, max_sigma=30, num_sigma=10, threshold=0.01, overlap=0.5, log_scale=False, *, threshold_rel=None)`

Finds blobs in the given grayscale image.

Blobs are found using the Determinant of Hessian method [?]. For each blob found, the method returns its coordinates and the standard deviation of the Gaussian Kernel used for the Hessian matrix whose determinant detected the blob. Determinant of Hessians is approximated using [?].

Parameters

image

[2D ndarray] Input grayscale image. Blobs can either be light on dark or vice versa.

min_sigma

[float, optional] The minimum standard deviation for Gaussian Kernel used to compute Hessian matrix. Keep this low to detect smaller blobs.

max_sigma

[float, optional] The maximum standard deviation for Gaussian Kernel used to compute Hessian matrix. Keep this high to detect larger blobs.

num_sigma

[int, optional] The number of intermediate values of standard deviations to consider between *min_sigma* and *max_sigma*.

threshold

[float or None, optional] The absolute lower bound for scale space maxima. Local maxima smaller than *threshold* are ignored. Reduce this to detect blobs with lower intensities. If *threshold_rel* is also specified, whichever threshold is larger will be used. If None, *threshold_rel* is used instead.

overlap

[float, optional] A value between 0 and 1. If the area of two blobs overlaps by a fraction greater than *threshold*, the smaller blob is eliminated.

log_scale

[bool, optional] If set intermediate values of standard deviations are interpolated using a logarithmic scale to the base 10. If not, linear interpolation is used.

threshold_rel

[float or None, optional] Minimum intensity of peaks, calculated as `max(doh_space) * threshold_rel`, where `doh_space` refers to the stack of Determinant-of-Hessian (DoH) images computed internally. This should have a value between 0 and 1. If None, *threshold* is used instead.

Returns

A

[(n, 3) ndarray] A 2d array with each row representing 3 values, (y, x, sigma) where (y, x)

are coordinates of the blob and `sigma` is the standard deviation of the Gaussian kernel of the Hessian Matrix whose determinant detected the blob.

Notes

The radius of each blob is approximately `sigma`. Computation of Determinant of Hessians is independent of the standard deviation. Therefore detecting larger blobs won't take more time. In methods line `blob_dog()` and `blob_log()` the computation of Gaussians for larger `sigma` takes more time. The downside is that this method can't be used for detecting blobs of radius less than `3px` due to the box filters used in the approximation of Hessian Determinant.

References

[?], [?]

Examples

```
>>> from skimage import data, feature
>>> img = data.coins()
>>> feature.blob_doh(img)
array([[197.        , 153.        , 20.33333333],
       [124.        , 336.        , 20.33333333],
       [126.        , 153.        , 20.33333333],
       [195.        , 100.        , 23.55555556],
       [192.        , 212.        , 23.55555556],
       [121.        , 271.        , 30.        ],
       [126.        , 101.        , 20.33333333],
       [193.        , 275.        , 23.55555556],
       [123.        , 205.        , 20.33333333],
       [270.        , 363.        , 30.        ],
       [265.        , 113.        , 23.55555556],
       [262.        , 243.        , 23.55555556],
       [185.        , 348.        , 30.        ],
       [156.        , 302.        , 30.        ],
       [123.        , 44.         , 23.55555556],
       [260.        , 173.        , 30.        ],
       [197.        , 44.         , 20.33333333]])
```

- *Blob Detection*
-

```
skimage.feature.blob_log(image, min_sigma=1, max_sigma=50, num_sigma=10, threshold=0.2, overlap=0.5,
                        log_scale=False, *, threshold_rel=None, exclude_border=False)
```

Finds blobs in the given grayscale image.

Blobs are found using the Laplacian of Gaussian (LoG) method [?]. For each blob found, the method returns its coordinates and the standard deviation of the Gaussian kernel that detected the blob.

Parameters

image

[ndarray] Input grayscale image, blobs are assumed to be light on dark background (white on black).

min_sigma

[scalar or sequence of scalars, optional] the minimum standard deviation for Gaussian kernel. Keep this low to detect smaller blobs. The standard deviations of the Gaussian filter are given for each axis as a sequence, or as a single number, in which case it is equal for all axes.

max_sigma

[scalar or sequence of scalars, optional] The maximum standard deviation for Gaussian kernel. Keep this high to detect larger blobs. The standard deviations of the Gaussian filter are given for each axis as a sequence, or as a single number, in which case it is equal for all axes.

num_sigma

[int, optional] The number of intermediate values of standard deviations to consider between *min_sigma* and *max_sigma*.

threshold

[float or None, optional] The absolute lower bound for scale space maxima. Local maxima smaller than *threshold* are ignored. Reduce this to detect blobs with lower intensities. If *threshold_rel* is also specified, whichever threshold is larger will be used. If None, *threshold_rel* is used instead.

overlap

[float, optional] A value between 0 and 1. If the area of two blobs overlaps by a fraction greater than *threshold*, the smaller blob is eliminated.

log_scale

[bool, optional] If set intermediate values of standard deviations are interpolated using a logarithmic scale to the base 10. If not, linear interpolation is used.

threshold_rel

[float or None, optional] Minimum intensity of peaks, calculated as $\max(\log_space) * \text{threshold_rel}$, where *log_space* refers to the stack of Laplacian-of-Gaussian (LoG) images computed internally. This should have a value between 0 and 1. If None, *threshold* is used instead.

exclude_border

[tuple of ints, int, or False, optional] If tuple of ints, the length of the tuple must match the input array's dimensionality. Each element of the tuple will exclude peaks from within *exclude_border*-pixels of the border of the image along that dimension. If nonzero int, *exclude_border* excludes peaks from within *exclude_border*-pixels of the border of the image. If zero or False, peaks are identified regardless of their distance from the border.

Returns

A

`[(n, image.ndim + sigma) ndarray]` A 2d array with each row representing 2 coordinate values for a 2D image, or 3 coordinate values for a 3D image, plus the sigma(s) used. When a single sigma is passed, outputs are: `(r, c, sigma)` or `(p, r, c, sigma)` where `(r, c)` or `(p, r, c)` are coordinates of the blob and `sigma` is the standard deviation of the Gaussian kernel which detected the blob. When an anisotropic gaussian is used (sigmas per dimension), the detected sigma is returned for each dimension.

Notes

The radius of each blob is approximately $\sqrt{2}\sigma$ for a 2-D image and $\sqrt{3}\sigma$ for a 3-D image.

References

[?]

Examples

```
>>> from skimage import data, feature, exposure
>>> img = data.coins()
>>> img = exposure.equalize_hist(img) # improves detection
>>> feature.blob_log(img, threshold = .3)
array([[124.        , 336.        , 11.88888889],
       [198.        , 155.        , 11.88888889],
       [194.        , 213.        , 17.33333333],
       [121.        , 272.        , 17.33333333],
       [263.        , 244.        , 17.33333333],
       [194.        , 276.        , 17.33333333],
       [266.        , 115.        , 11.88888889],
       [128.        , 154.        , 11.88888889],
       [260.        , 174.        , 17.33333333],
       [198.        , 103.        , 11.88888889],
       [126.        , 208.        , 11.88888889],
       [127.        , 102.        , 11.88888889],
       [263.        , 302.        , 17.33333333],
       [197.        , 44.         , 11.88888889],
       [185.        , 344.        , 17.33333333],
       [126.        , 46.         , 11.88888889],
       [113.        , 323.        , 1.          ]])
```

- *Blob Detection*

```
skimage.feature.canny(image, sigma=1.0, low_threshold=None, high_threshold=None, mask=None,
                      use_quantiles=False, *, mode='constant', cval=0.0)
```

Edge filter an image using the Canny algorithm.

Parameters**image**

[2D array] Grayscale input image to detect edges on; can be of any dtype.

sigma

[float, optional] Standard deviation of the Gaussian filter.

low_threshold

[float, optional] Lower bound for hysteresis thresholding (linking edges). If None, low_threshold is set to 10% of dtype's max.

high_threshold

[float, optional] Upper bound for hysteresis thresholding (linking edges). If None, high_threshold is set to 20% of dtype's max.

mask

[array, dtype=bool, optional] Mask to limit the application of Canny to a certain area.

use_quantiles

[bool, optional] If True then treat low_threshold and high_threshold as quantiles of the edge magnitude image, rather than absolute edge magnitude values. If True then the thresholds must be in the range [0, 1].

mode

[str, {‘reflect’, ‘constant’, ‘nearest’, ‘mirror’, ‘wrap’}] The mode parameter determines how the array borders are handled during Gaussian filtering, 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’.

Returns**output**

[2D array (image)] The binary edge map.

See also:

`skimage.filters.sobel`

Notes

The steps of the algorithm are as follows:

- Smooth the image using a Gaussian with `sigma` width.
- Apply the horizontal and vertical Sobel operators to get the gradients within the image. The edge strength is the norm of the gradient.
- Thin potential edges to 1-pixel wide curves. First, find the normal to the edge at each point. This is done by looking at the signs and the relative magnitude of the X-Sobel and Y-Sobel to sort the points into 4 categories: horizontal, vertical, diagonal and antidiagonal. Then look in the normal and reverse directions to see if the values in either of those directions are greater than the point in question. Use interpolation to get a mix of points instead of picking the one that's the closest to the normal.
- Perform a hysteresis thresholding: first label all points above the high threshold as edges. Then recursively label any point above the low threshold that is 8-connected to a labeled point as an edge.

References

[?], [?]

Examples

```
>>> from skimage import feature
>>> rng = np.random.default_rng()
>>> # Generate noisy image of a square
>>> im = np.zeros((256, 256))
>>> im[64:-64, 64:-64] = 1
>>> im += 0.2 * rng.random(im.shape)
>>> # First trial with the Canny filter, with the default smoothing
>>> edges1 = feature.canny(im)
>>> # Increase the smoothing for better results
>>> edges2 = feature.canny(im, sigma=3)
```

- *Canny edge detector*
- *Straight line Hough transform*
- *Circular and Elliptical Hough Transforms*
- *Evaluating segmentation metrics*
- *Comparing edge-based and region-based segmentation*

```
skimage.feature.corner_fast(image, n=12, threshold=0.15)
```

Extract FAST corners for a given image.

Parameters

image

[(M, N) ndarray] Input image.

n

[int, optional] Minimum number of consecutive pixels out of 16 pixels on the circle that should all be either brighter or darker w.r.t testpixel. A point c on the circle is darker w.r.t test pixel p if $I_c < I_p - threshold$ and brighter if $I_c > I_p + threshold$. Also stands for the n in *FAST-n* corner detector.

threshold

[float, optional] Threshold used in deciding whether the pixels on the circle are brighter, darker or similar w.r.t. the test pixel. Decrease the threshold when more corners are desired and vice-versa.

Returns**response**

[ndarray] FAST corner response image.

References

[?], [?]

Examples

```
>>> from skimage.feature import corner_fast, corner_peaks
>>> square = np.zeros((12, 12))
>>> square[3:9, 3:9] = 1
>>> square.astype(int)
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0],
       [0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0],
       [0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0],
       [0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0],
       [0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0],
       [0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0],
       [0, 0, 0, 1, 1, 1, 1, 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, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
>>> corner_peaks(corner_fast(square, 9), min_distance=1)
array([[3, 3],
       [3, 8],
       [8, 3],
       [8, 8]])
```

`skimage.feature.corner_foerstner(image, sigma=1)`

Compute Foerstner corner measure response image.

This corner detector uses information from the auto-correlation matrix A:

$$\begin{bmatrix} A = [(imx^{**2}) & (imx*imy)] \\ [(imx*imy) & (imy^{**2})] \end{bmatrix} = \begin{bmatrix} [A_{xx} \quad A_{xy}] \\ [A_{xy} \quad A_{yy}] \end{bmatrix}$$

Where imx and imy are first derivatives, averaged with a gaussian filter. The corner measure is then defined as:

$$\begin{aligned} w &= \det(A) / \text{trace}(A) && (\text{size of error ellipse}) \\ q &= 4 * \det(A) / \text{trace}(A)^{**2} && (\text{roundness of error ellipse}) \end{aligned}$$

Parameters

image

[(M, N) ndarray] Input image.

sigma

[float, optional] Standard deviation used for the Gaussian kernel, which is used as weighting function for the auto-correlation matrix.

Returns

w

[ndarray] Error ellipse sizes.

q

[ndarray] Roundness of error ellipse.

References

[?], [?]

Examples

```
>>> from skimage.feature import corner_foerstner, corner_peaks
>>> square = np.zeros([10, 10])
>>> square[2:8, 2:8] = 1
>>> square.astype(int)
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 1, 1, 1, 0, 0, 0],
       [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
```

(continues on next page)

(continued from previous page)

```
[0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
[0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
[0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
>>> w, q = corner_foerstner(square)
>>> accuracy_thresh = 0.5
>>> roundness_thresh = 0.3
>>> foerstner = (q > roundness_thresh) * (w > accuracy_thresh) * w
>>> corner_peaks(foerstner, min_distance=1)
array([[2, 2],
       [2, 7],
       [7, 2],
       [7, 7]])
```

`skimage.feature.corner_harris(image, method='k', k=0.05, eps=1e-06, sigma=1)`

Compute Harris corner measure response image.

This corner detector uses information from the auto-correlation matrix A:

$$\begin{bmatrix} \text{imx}^{**2} & (\text{imx} \cdot \text{imy}) \\ (\text{imx} \cdot \text{imy}) & \text{imy}^{**2} \end{bmatrix} = \begin{bmatrix} \text{Axx} & \text{Axy} \\ \text{Axy} & \text{Ayy} \end{bmatrix}$$

Where imx and imy are first derivatives, averaged with a gaussian filter. The corner measure is then defined as:

$$\det(A) - k * \text{trace}(A)^{**2}$$

or:

$$2 * \det(A) / (\text{trace}(A) + \text{eps})$$

Parameters

image

[(M, N) ndarray] Input image.

method

[{'k', 'eps'}, optional] Method to compute the response image from the auto-correlation matrix.

k

[float, optional] Sensitivity factor to separate corners from edges, typically in range [0, 0.2]. Small values of k result in detection of sharp corners.

eps

[float, optional] Normalisation factor (Noble's corner measure).

sigma

[float, optional] Standard deviation used for the Gaussian kernel, which is used as weighting function for the auto-correlation matrix.

Returns**response**

[ndarray] Harris response image.

References

[?]

Examples

```
>>> from skimage.feature import corner_harris, corner_peaks
>>> square = np.zeros([10, 10])
>>> square[2:8, 2:8] = 1
>>> square.astype(int)
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
>>> corner_peaks(corner_harris(square), min_distance=1)
array([[2, 2],
       [2, 7],
       [7, 2],
       [7, 7]])
```

- Robust matching using RANSAC
- Assemble images with simple image stitching
- Corner detection
- BRIEF binary descriptor

```
skimage.feature.corner_kitchen_rosenfeld(image, mode='constant', cval=0)
```

Compute Kitchen and Rosenfeld corner measure response image.

The corner measure is calculated as follows:

```
(imxx * imy**2 + imyy * imx**2 - 2 * imxy * imx * imy)
/ (imx**2 + imy**2)
```

Where imx and imy are the first and imxx, imxy, imyy the second derivatives.

Parameters

image

[(M, N) ndarray] Input image.

mode

[{‘constant’, ‘reflect’, ‘wrap’, ‘nearest’, ‘mirror’}, optional] How to handle values outside the image borders.

cval

[float, optional] Used in conjunction with mode ‘constant’, the value outside the image boundaries.

Returns

response

[ndarray] Kitchen and Rosenfeld response image.

References

[?]

`skimage.feature.corner_moravec(image, window_size=1)`

Compute Moravec corner measure response image.

This is one of the simplest corner detectors and is comparatively fast but has several limitations (e.g. not rotation invariant).

Parameters

image

[(M, N) ndarray] Input image.

window_size

[int, optional] Window size.

Returns

response

[ndarray] Moravec response image.

References

[?]

Examples

```
>>> from skimage.feature import corner_moravec
>>> square = np.zeros([7, 7])
>>> square[3, 3] = 1
>>> square.astype(int)
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 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]])
>>> corner_moravec(square).astype(int)
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 0, 0, 0],
       [0, 0, 1, 2, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
```

skimage.feature.corner_orientations(*image*, *corners*, *mask*)

Compute the orientation of corners.

The orientation of corners is computed using the first order central moment i.e. the center of mass approach. The corner orientation is the angle of the vector from the corner coordinate to the intensity centroid in the local neighborhood around the corner calculated using first order central moment.

Parameters**image**

[(M, N) array] Input grayscale image.

corners

[(K, 2) array] Corner coordinates as (row, col).

mask

[2D array] Mask defining the local neighborhood of the corner used for the calculation of the central moment.

Returns

orientations

[(K, 1) array] Orientations of corners in the range [-pi, pi].

References

[?], [?]

Examples

```
>>> from skimage.morphology import octagon
>>> from skimage.feature import (corner_fast, corner_peaks,
...                               corner_orientations)
>>> square = np.zeros((12, 12))
>>> square[3:9, 3:9] = 1
>>> square.astype(int)
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0],
       [0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0],
       [0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0],
       [0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0],
       [0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0],
       [0, 0, 0, 1, 1, 1, 1, 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, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
>>> corners = corner_peaks(corner_fast(square, 9), min_distance=1)
>>> corners
array([[3, 3],
       [3, 8],
       [8, 3],
       [8, 8]])
>>> orientations = corner_orientations(square, corners, octagon(3, 2))
>>> np.rad2deg(orientations)
array([-45., 135., -45., -135.])
```

`skimage.feature.corner_peaks(image, min_distance=1, threshold_abs=None, threshold_rel=None, exclude_border=True, indices=True, num_peaks=inf, footprint=None, labels=None, *, num_peaks_per_label=inf, p_norm=inf)`

Find peaks in corner measure response image.

This differs from `skimage.feature.peak_local_max` in that it suppresses multiple connected peaks with the same accumulator value.

Parameters

image

[(M, N) ndarray] Input image.

min_distance

[int, optional] The minimal allowed distance separating peaks.

*

[*] See `skimage.feature.peak_local_max()`.

p_norm

[float] Which Minkowski p-norm to use. Should be in the range [1, inf]. A finite large p may cause a ValueError if overflow can occur. inf corresponds to the Chebyshev distance and 2 to the Euclidean distance.

Returns

output

[ndarray or ndarray of bools]

- If `indices = True` : (row, column, ...) coordinates of peaks.
- If `indices = False` : Boolean array shaped like `image`, with peaks represented by True values.

See also:

`skimage.feature.peak_local_max`

Notes

Changed in version 0.18: The default value of `threshold_rel` has changed to None, which corresponds to letting `skimage.feature.peak_local_max` decide on the default. This is equivalent to `threshold_rel=0`.

The `num_peaks` limit is applied before suppression of connected peaks. To limit the number of peaks after suppression, set `num_peaks=np.inf` and post-process the output of this function.

Examples

```
>>> from skimage.feature import peak_local_max
>>> response = np.zeros((5, 5))
>>> response[2:4, 2:4] = 1
>>> response
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 1., 1., 0.],
       [0., 0., 1., 1., 0.],
       [0., 0., 0., 0., 0.]])
>>> peak_local_max(response)
array([[2, 2],
       [2, 3],
       [3, 2],
       [3, 3]])
>>> corner_peaks(response)
array([[2, 2]])
```

- Robust matching using RANSAC
- Assemble images with simple image stitching
- Corner detection
- BRIEF binary descriptor

`skimage.feature.corner_shi_tomasi(image, sigma=1)`

Compute Shi-Tomasi (Kanade-Tomasi) corner measure response image.

This corner detector uses information from the auto-correlation matrix A:

$$\begin{bmatrix} \text{imx}^{**2} & \text{imx} * \text{imy} \\ \text{imx} * \text{imy} & \text{imy}^{**2} \end{bmatrix} = \begin{bmatrix} \text{Axx} & \text{Axy} \\ \text{Axy} & \text{Ayy} \end{bmatrix}$$

Where imx and imy are first derivatives, averaged with a gaussian filter. The corner measure is then defined as the smaller eigenvalue of A:

$$((\text{Axx} + \text{Ayy}) - \sqrt{(\text{Axx} - \text{Ayy})^{**2} + 4 * \text{Axy}^{**2}}) / 2$$

Parameters

image

[(M, N) ndarray] Input image.

sigma

[float, optional] Standard deviation used for the Gaussian kernel, which is used as weighting function for the auto-correlation matrix.

Returns**response**

[ndarray] Shi-Tomasi response image.

References

[?]

Examples

```
>>> from skimage.feature import corner_shi_tomasi, corner_peaks
>>> square = np.zeros([10, 10])
>>> square[2:8, 2:8] = 1
>>> square.astype(int)
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
>>> corner_peaks(corner_shi_tomasi(square), min_distance=1)
array([[2, 2],
       [2, 7],
       [7, 2],
       [7, 7]])
```

`skimage.feature.corner_subpix(image, corners, window_size=11, alpha=0.99)`

Determine subpixel position of corners.

A statistical test decides whether the corner is defined as the intersection of two edges or a single peak. Depending on the classification result, the subpixel corner location is determined based on the local covariance of the grey-values. If the significance level for either statistical test is not sufficient, the corner cannot be classified, and the output subpixel position is set to NaN.

Parameters**image**

[(M, N) ndarray] Input image.

corners

`[(K, 2) ndarray]` Corner coordinates (*row, col*).

window_size

[int, optional] Search window size for subpixel estimation.

alpha

[float, optional] Significance level for corner classification.

Returns**positions**

`[(K, 2) ndarray]` Subpixel corner positions. NaN for “not classified” corners.

References

[?], [?]

Examples

```
>>> from skimage.feature import corner_harris, corner_peaks, corner_subpix
>>> img = np.zeros((10, 10))
>>> img[:5, :5] = 1
>>> img[5:, 5:] = 1
>>> img.astype(int)
array([[1, 1, 1, 1, 1, 0, 0, 0, 0, 0],
       [1, 1, 1, 1, 1, 0, 0, 0, 0, 0],
       [1, 1, 1, 1, 1, 0, 0, 0, 0, 0],
       [1, 1, 1, 1, 1, 0, 0, 0, 0, 0],
       [1, 1, 1, 1, 1, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
       [0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
       [0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
       [0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
       [0, 0, 0, 0, 1, 1, 1, 1, 1, 1]])
>>> coords = corner_peaks(corner_harris(img), min_distance=2)
>>> coords_subpix = corner_subpix(img, coords, window_size=7)
>>> coords_subpix
array([[4.5, 4.5]])
```

- Robust matching using RANSAC
- Corner detection

```
skimage.feature.daisy(image, step=4, radius=15, rings=3, histograms=8, orientations=8, normalization='l1',
                      sigmas=None, ring_radii=None, visualize=False)
```

Extract DAISY feature descriptors densely for the given image.

DAISY is a feature descriptor similar to SIFT formulated in a way that allows for fast dense extraction. Typically, this is practical for bag-of-features image representations.

The implementation follows Tola et al. [?] but deviate on the following points:

- Histogram bin contribution are smoothed with a circular Gaussian window over the tonal range (the angular range).
- The sigma values of the spatial Gaussian smoothing in this code do not match the sigma values in the original code by Tola et al. [?]. In their code, spatial smoothing is applied to both the input image and the center histogram. However, this smoothing is not documented in [?] and, therefore, it is omitted.

Parameters

image

[(M, N) array] Input image (grayscale).

step

[int, optional] Distance between descriptor sampling points.

radius

[int, optional] Radius (in pixels) of the outermost ring.

rings

[int, optional] Number of rings.

histograms

[int, optional] Number of histograms sampled per ring.

orientations

[int, optional] Number of orientations (bins) per histogram.

normalization

[['l1' | 'l2' | 'daisy' | 'off'], optional] How to normalize the descriptors

- 'l1': L1-normalization of each descriptor.
- 'l2': L2-normalization of each descriptor.
- 'daisy': L2-normalization of individual histograms.
- 'off': Disable normalization.

sigmas

[1D array of float, optional] Standard deviation of spatial Gaussian smoothing for the center histogram and for each ring of histograms. The array of sigmas should be sorted from the center and out. I.e. the first sigma value defines the spatial smoothing of the center

histogram and the last sigma value defines the spatial smoothing of the outermost ring. Specifying sigmas overrides the following parameter.

```
rings = len(sigmas) - 1
```

ring_radii

[1D array of int, optional] Radius (in pixels) for each ring. Specifying ring_radii overrides the following two parameters.

```
rings = len(ring_radii) radius = ring_radii[-1]
```

If both sigmas and ring_radii are given, they must satisfy the following predicate since no radius is needed for the center histogram.

```
len(ring_radii) == len(sigmas) + 1
```

visualize

[bool, optional] Generate a visualization of the DAISY descriptors

Returns

descs

[array] Grid of DAISY descriptors for the given image as an array dimensionality (P, Q, R) where

```
P = ceil((M - radius*2) / step) Q = ceil((N - radius*2) / step) R =  
(rings * histograms + 1) * orientations
```

descs_img

[(M, N, 3) array (only if visualize==True)] Visualization of the DAISY descriptors.

References

[?], [?]

- *Dense DAISY feature description*

```
skimage.feature.draw_haar_like_feature(image, r, c, width, height, feature_coord,  
color_positive_block=(1.0, 0.0, 0.0), color_negative_block=(0.0,  
1.0, 0.0), alpha=0.5, max_n_features=None, rng=None)
```

Visualization of Haar-like features.

Parameters

image

[(M, N) ndarray] The region of an integral image for which the features need to be computed.

r

[int] Row-coordinate of top left corner of the detection window.

c

[int] Column-coordinate of top left corner of the detection window.

width

[int] Width of the detection window.

height

[int] Height of the detection window.

feature_coord

[ndarray of list of tuples or None, optional] The array of coordinates to be extracted. This is useful when you want to recompute only a subset of features. In this case *feature_type* needs to be an array containing the type of each feature, as returned by *haar_like_feature_coord()*. By default, all coordinates are computed.

color_positive_block

[tuple of 3 floats] Floats specifying the color for the positive block. Corresponding values define (R, G, B) values. Default value is red (1, 0, 0).

color_negative_block

[tuple of 3 floats] Floats specifying the color for the negative block. Corresponding values define (R, G, B) values. Default value is blue (0, 1, 0).

alpha

[float] Value in the range [0, 1] that specifies opacity of visualization. 1 - fully transparent, 0 - opaque.

max_n_features

[int, default=None] The maximum number of features to be returned. By default, all features are returned.

rng

[{`numpy.random.Generator`, int}, optional] Pseudo-random number generator. By default, a PCG64 generator is used (see `numpy.random.default_rng()`). If *rng* is an int, it is used to seed the generator.

The *rng* is used when generating a set of features smaller than the total number of available features.

Returns

features

[(M, N), ndarray] An image in which the different features will be added.

Other Parameters

random_state

[DEPRECATED] Deprecated in favor of *rng*.

Deprecated since version 0.21.

Examples

```
>>> import numpy as np
>>> from skimage.feature import haar_like_feature_coord
>>> from skimage.feature import draw_haar_like_feature
>>> feature_coord, _ = haar_like_feature_coord(2, 2, 'type-4')
>>> image = draw_haar_like_feature(np.zeros((2, 2)),
...                                 0, 0, 2, 2,
...                                 feature_coord,
...                                 max_n_features=1)
>>> image
array([[ [0., 0.5, 0.],
       [0.5, 0., 0.] ],
      [[0.5, 0., 0.],
       [0., 0.5, 0.]]])
```

- Haar-like feature descriptor
- Face classification using Haar-like feature descriptor

`skimage.feature.draw_multiblock_lbp(image, r, c, width, height, lbp_code=0, color_greater_block=(1, 1, 1), color_less_block=(0, 0.69, 0.96), alpha=0.5)`

Multi-block local binary pattern visualization.

Blocks with higher sums are colored with alpha-blended white rectangles, whereas blocks with lower sums are colored alpha-blended cyan. Colors and the *alpha* parameter can be changed.

Parameters

image

[ndarray of float or uint] Image on which to visualize the pattern.

r

[int] Row-coordinate of top left corner of a rectangle containing feature.

c

[int] Column-coordinate of top left corner of a rectangle containing feature.

width

[int] Width of one of 9 equal rectangles that will be used to compute a feature.

height

[int] Height of one of 9 equal rectangles that will be used to compute a feature.

lbp_code

[int] The descriptor of feature to visualize. If not provided, the descriptor with 0 value will be used.

color_greater_block

[tuple of 3 floats] Floats specifying the color for the block that has greater intensity value. They should be in the range [0, 1]. Corresponding values define (R, G, B) values. Default value is white (1, 1, 1).

color_smaller_block

[tuple of 3 floats] Floats specifying the color for the block that has smaller intensity value. They should be in the range [0, 1]. Corresponding values define (R, G, B) values. Default value is cyan (0, 0.69, 0.96).

alpha

[float] Value in the range [0, 1] that specifies opacity of visualization. 1 - fully transparent, 0 - opaque.

Returns

output

[ndarray of float] Image with MB-LBP visualization.

References

[?]

- *Multi-Block Local Binary Pattern for texture classification*

`skimage.feature.fisher_vector(descriptors, gmm, *, improved=False, alpha=0.5)`

Compute the Fisher vector given some descriptors/vectors, and an associated estimated GMM.

Parameters

descriptors

[np.ndarray, shape=(n_descriptors, descriptor_length)] NumPy array of the descriptors for which the Fisher vector representation is to be computed.

gmm

[sklearn.mixture.GaussianMixture] An estimated GMM object, which contains the necessary parameters needed to compute the Fisher vector.

improved

[bool, default=False] Flag denoting whether to compute improved Fisher vectors or not. Improved Fisher vectors are L2 and power normalized. Power normalization is simply $f(z) = \text{sign}(z) \text{pow}(\text{abs}(z), \alpha)$ for some $0 \leq \alpha \leq 1$.

alpha

[float, default=0.5] The parameter for the power normalization step. Ignored if improved=False.

Returns**fisher_vector**

[np.ndarray] The computation Fisher vector, which is given by a concatenation of the gradients of a GMM with respect to its parameters (mixture weights, means, and covariance matrices). For D-dimensional input descriptors or vectors, and a K-mode GMM, the Fisher vector dimensionality will be $2KD + K$. Thus, its dimensionality is invariant to the number of descriptors/vectors.

References

[?], [?]

Examples

```
>>> import pytest
>>> _ = pytest.importorskip('sklearn')
>>> from skimage.feature import fisher_vector, learn_gmm
>>> sift_for_images = [np.random.random((10, 128)) for _ in range(10)]
>>> num_modes = 16
>>> # Estimate 16-mode GMM with these synthetic SIFT vectors
>>> gmm = learn_gmm(sift_for_images, n_modes=num_modes)
>>> test_image_descriptors = np.random.random((25, 128))
>>> # Compute the Fisher vector
>>> fv = fisher_vector(test_image_descriptors, gmm)
```

- Fisher vector feature encoding

`skimage.feature.graycomatrix(image, distances, angles, levels=None, symmetric=False, normed=False)`

Calculate the gray-level co-occurrence matrix.

A gray level co-occurrence matrix is a histogram of co-occurring grayscale values at a given offset over an image.

Changed in version 0.19: `greymatrix` was renamed to `graymatrix` in 0.19.

Parameters

image

[array_like] Integer typed input image. Only positive valued images are supported. If type is other than uint8, the argument `levels` needs to be set.

distances

[array_like] List of pixel pair distance offsets.

angles

[array_like] List of pixel pair angles in radians.

levels

[int, optional] The input image should contain integers in [0, `levels`-1], where levels indicate the number of gray-levels counted (typically 256 for an 8-bit image). This argument is required for 16-bit images or higher and is typically the maximum of the image. As the output matrix is at least `levels` x `levels`, it might be preferable to use binning of the input image rather than large values for `levels`.

symmetric

[bool, optional] If True, the output matrix $P[:, :, d, \theta]$ is symmetric. This is accomplished by ignoring the order of value pairs, so both (i, j) and (j, i) are accumulated when (i, j) is encountered for a given offset. The default is False.

normed

[bool, optional] If True, normalize each matrix $P[:, :, d, \theta]$ by dividing by the total number of accumulated co-occurrences for the given offset. The elements of the resulting matrix sum to 1. The default is False.

Returns

P

[4-D ndarray] The gray-level co-occurrence histogram. The value $P[i,j,d,\theta]$ is the number of times that gray-level j occurs at a distance d and at an angle θ from gray-level i . If `normed` is `False`, the output is of type uint32, otherwise it is float64. The dimensions are: `levels` x `levels` x number of distances x number of angles.

References

[?], [?], [?], [?]

Examples

Compute 2 GLCMs: One for a 1-pixel offset to the right, and one for a 1-pixel offset upwards.

```
>>> image = np.array([[0, 0, 1, 1],
...                   [0, 0, 1, 1],
...                   [0, 2, 2, 2],
...                   [2, 2, 3, 3]], dtype=np.uint8)
>>> result = graycomatrix(image, [1], [0, np.pi/4, np.pi/2, 3*np.pi/4],
...                         levels=4)
>>> result[:, :, 0, 0]
array([[2, 2, 1, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 1],
       [0, 0, 0, 1]], dtype=uint32)
>>> result[:, :, 0, 1]
array([[1, 1, 3, 0],
       [0, 1, 1, 0],
       [0, 0, 0, 2],
       [0, 0, 0, 0]], dtype=uint32)
>>> result[:, :, 0, 2]
array([[3, 0, 2, 0],
       [0, 2, 2, 0],
       [0, 0, 1, 2],
       [0, 0, 0, 0]], dtype=uint32)
>>> result[:, :, 0, 3]
array([[2, 0, 0, 0],
       [1, 1, 2, 0],
       [0, 0, 2, 1],
       [0, 0, 0, 0]], dtype=uint32)
```

- *GLCM Texture Features*

`skimage.feature.graycoprops(P, prop='contrast')`

Calculate texture properties of a GLCM.

Compute a feature of a gray level co-occurrence matrix to serve as a compact summary of the matrix. The properties are computed as follows:

- ‘contrast’: $\sum_{i,j=0}^{levels-1} P_{i,j}(i - j)^2$
- ‘dissimilarity’: $\sum_{i,j=0}^{levels-1} P_{i,j}|i - j|$
- ‘homogeneity’: $\sum_{i,j=0}^{levels-1} \frac{P_{i,j}}{1+(i-j)^2}$
- ‘ASM’: $\sum_{i,j=0}^{levels-1} P_{i,j}^2$
- ‘energy’: \sqrt{ASM}

- ‘correlation’:

$$\sum_{i,j=0}^{levels-1} P_{i,j} \left[\frac{(i - \mu_i)(j - \mu_j)}{\sqrt{(\sigma_i^2)(\sigma_j^2)}} \right]$$

Each GLCM is normalized to have a sum of 1 before the computation of texture properties.

Changed in version 0.19: `greycoprops` was renamed to `graycoprops` in 0.19.

Parameters

P

[ndarray] Input array. P is the gray-level co-occurrence histogram for which to compute the specified property. The value $P[i,j,d,theta]$ is the number of times that gray-level j occurs at a distance d and at an angle θ from gray-level i .

prop

[{‘contrast’, ‘dissimilarity’, ‘homogeneity’, ‘energy’, ‘correlation’, ‘ASM’}, optional] The property of the GLCM to compute. The default is ‘contrast’.

Returns

results

[2-D ndarray] 2-dimensional array. $results[d, a]$ is the property ‘prop’ for the d ’th distance and the a ’th angle.

References

[?]

Examples

Compute the contrast for GLCMs with distances [1, 2] and angles [0 degrees, 90 degrees]

```
>>> image = np.array([[0, 0, 1, 1],
...                   [0, 0, 1, 1],
...                   [0, 2, 2, 2],
...                   [2, 2, 3, 3]], dtype=np.uint8)
>>> g = graycomatrix(image, [1, 2], [0, np.pi/2], levels=4,
...                   normed=True, symmetric=True)
>>> contrast = graycoprops(g, 'contrast')
>>> contrast
array([[0.58333333, 1.          ],
       [1.25        , 2.75        ]])
```

- *GLCM Texture Features*

```
skimage.feature.haar_like_feature(int_image, r, c, width, height, feature_type=None, feature_coord=None)
```

Compute the Haar-like features for a region of interest (ROI) of an integral image.

Haar-like features have been successfully used for image classification and object detection [?]. It has been used for real-time face detection algorithm proposed in [?].

Parameters

int_image

[(M, N) ndarray] Integral image for which the features need to be computed.

r

[int] Row-coordinate of top left corner of the detection window.

c

[int] Column-coordinate of top left corner of the detection window.

width

[int] Width of the detection window.

height

[int] Height of the detection window.

feature_type

[str or list of str or None, optional] The type of feature to consider:

- ‘type-2-x’: 2 rectangles varying along the x axis;
- ‘type-2-y’: 2 rectangles varying along the y axis;
- ‘type-3-x’: 3 rectangles varying along the x axis;
- ‘type-3-y’: 3 rectangles varying along the y axis;
- ‘type-4’: 4 rectangles varying along x and y axis.

By default all features are extracted.

If using with *feature_coord*, it should correspond to the feature type of each associated coordinate feature.

feature_coord

[ndarray of list of tuples or None, optional] The array of coordinates to be extracted. This is useful when you want to recompute only a subset of features. In this case *feature_type* needs to be an array containing the type of each feature, as returned by *haar_like_feature_coord()*. By default, all coordinates are computed.

Returns

haar_features

`[(n_features,) ndarray of int or float]` Resulting Haar-like features. Each value is equal to the subtraction of sums of the positive and negative rectangles. The data type depends of the data type of `int_image`: `int` when the data type of `int_image` is `uint` or `int` and `float` when the data type of `int_image` is `float`.

Notes

When extracting those features in parallel, be aware that the choice of the backend (i.e. multiprocessing vs threading) will have an impact on the performance. The rule of thumb is as follows: use multiprocessing when extracting features for all possible ROI in an image; use threading when extracting the feature at specific location for a limited number of ROIs. Refer to the example *Face classification using Haar-like feature descriptor* for more insights.

References

[?], [?], [?]

Examples

```
>>> import numpy as np
>>> from skimage.transform import integral_image
>>> from skimage.feature import haar_like_feature
>>> img = np.ones((5, 5), dtype=np.uint8)
>>> img_ii = integral_image(img)
>>> feature = haar_like_feature(img_ii, 0, 0, 5, 5, 'type-3-x')
>>> feature
array([-1, -2, -3, -4, -5, -1, -2, -3, -4, -5, -1, -2, -3, -4, -5, -1, -2,
       -3, -4, -1, -2, -3, -4, -1, -2, -3, -4, -1, -2, -3, -1, -2,
       -2, -3, -1, -2, -1, -2, -1, -2, -1, -1, -1])
```

You can compute the feature for some pre-computed coordinates.

```
>>> from skimage.feature import haar_like_feature_coord
>>> feature_coord, feature_type = zip(
...     *[haar_like_feature_coord(5, 5, feat_t)
...       for feat_t in ('type-2-x', 'type-3-x')])
>>> # only select one feature over two
>>> feature_coord = np.concatenate([x[::2] for x in feature_coord])
>>> feature_type = np.concatenate([x[::2] for x in feature_type])
>>> feature = haar_like_feature(img_ii, 0, 0, 5, 5,
...                               feature_type=feature_type,
...                               feature_coord=feature_coord)
>>> feature
array([ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
       0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
       0,  0,  0,  0,  0,  0,  0,  0,  0, -1, -3, -5, -2, -4, -1,
      -3, -5, -2, -4, -2, -4, -2, -1, -3, -2, -1, -1, -1, -1, -1])
```

- *Face classification using Haar-like feature descriptor*

skimage.feature.haar_like_feature_coord(*width*, *height*, *feature_type=None*)

Compute the coordinates of Haar-like features.

Parameters**width**

[int] Width of the detection window.

height

[int] Height of the detection window.

feature_type

[str or list of str or None, optional] The type of feature to consider:

- ‘type-2-x’: 2 rectangles varying along the x axis;
- ‘type-2-y’: 2 rectangles varying along the y axis;
- ‘type-3-x’: 3 rectangles varying along the x axis;
- ‘type-3-y’: 3 rectangles varying along the y axis;
- ‘type-4’: 4 rectangles varying along x and y axis.

By default all features are extracted.

Returns**feature_coord**

[(*n_features*, *n_rectangles*, 2, 2), ndarray of list of tuple coord] Coordinates of the rectangles for each feature.

feature_type

[(*n_features*,), ndarray of str] The corresponding type for each feature.

Examples

```
>>> import numpy as np
>>> from skimage.transform import integral_image
>>> from skimage.feature import haar_like_feature_coord
>>> feat_coord, feat_type = haar_like_feature_coord(2, 2, 'type-4')
>>> feat_coord
array([ list([(0, 0), (0, 0)], [(0, 1), (0, 1)],
           [(1, 1), (1, 1)], [(1, 0), (1, 0)]]), dtype=object)
>>> feat_type
array(['type-4'], dtype=object)
```

- *Haar-like feature descriptor*
 - *Face classification using Haar-like feature descriptor*
-

```
skimage.feature.hessian_matrix(image, sigma=1, mode='constant', cval=0, order='rc',
                               use_gaussian_derivatives=None)
```

Compute the Hessian matrix.

In 2D, the Hessian matrix is defined as:

```
H = [Hrr Hrc]
     [Hrc Hcc]
```

which is computed by convolving the image with the second derivatives of the Gaussian kernel in the respective r- and c-directions.

The implementation here also supports n-dimensional data.

Parameters

image

[ndarray] Input image.

sigma

[float] Standard deviation used for the Gaussian kernel, which is used as weighting function for the auto-correlation matrix.

mode

[{'constant', 'reflect', 'wrap', 'nearest', 'mirror'}, optional] How to handle values outside the image borders.

cval

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

order

[{'rc', 'xy'}, optional] For 2D images, this parameter allows for the use of reverse or forward order of the image axes in gradient computation. 'rc' indicates the use of the first axis initially (Hrr, Hrc, Hcc), whilst 'xy' indicates the usage of the last axis initially (Hxx, Hxy, Hyy). Images with higher dimension must always use 'rc' order.

use_gaussian_derivatives

[boolean, optional] Indicates whether the Hessian is computed by convolving with Gaussian derivatives, or by a simple finite-difference operation.

Returns

H_elems

[list of ndarray] Upper-diagonal elements of the hessian matrix for each pixel in the input image. In 2D, this will be a three element list containing [Hrr, Hrc, Hcc]. In nD, the list will contain $(n^{**2} + n) / 2$ arrays.

Notes

The distributive property of derivatives and convolutions allows us to restate the derivative of an image, I, smoothed with a Gaussian kernel, G, as the convolution of the image with the derivative of G.

$$\frac{\partial}{\partial x_i}(I * G) = I * \left(\frac{\partial}{\partial x_i} G \right)$$

When `use_gaussian_derivatives` is `True`, this property is used to compute the second order derivatives that make up the Hessian matrix.

When `use_gaussian_derivatives` is `False`, simple finite differences on a Gaussian-smoothed image are used instead.

Examples

```
>>> from skimage.feature import hessian_matrix
>>> square = np.zeros((5, 5))
>>> square[2, 2] = 4
>>> Hrr, Hrc, Hcc = hessian_matrix(square, sigma=0.1, order='rc',
...                                     use_gaussian_derivatives=False)
>>> Hrc
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  1.,  0., -1.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0., -1.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
```

`skimage.feature.hessian_matrix_det(image, sigma=1, approximate=True)`

Compute the approximate Hessian Determinant over an image.

The 2D approximate method uses box filters over integral images to compute the approximate Hessian Determinant.

Parameters

image

[ndarray] The image over which to compute the Hessian Determinant.

sigma

[float, optional] Standard deviation of the Gaussian kernel used for the Hessian matrix.

approximate

[bool, optional] If True and the image is 2D, use a much faster approximate computation. This argument has no effect on 3D and higher images.

Returns

out

[array] The array of the Determinant of Hessians.

Notes

For 2D images when `approximate=True`, the running time of this method only depends on size of the image. It is independent of `sigma` as one would expect. The downside is that the result for `sigma` less than 3 is not accurate, i.e., not similar to the result obtained if someone computed the Hessian and took its determinant.

References

[?]

`skimage.feature.hessian_matrix_eigvals(H_elems)`

Compute eigenvalues of Hessian matrix.

Parameters

`H_elems`

[list of ndarray] The upper-diagonal elements of the Hessian matrix, as returned by `hessian_matrix`.

Returns

`eigs`

[ndarray] The eigenvalues of the Hessian matrix, in decreasing order. The eigenvalues are the leading dimension. That is, `eigs[i, j, k]` contains the ith-largest eigenvalue at position (j, k).

Examples

```
>>> from skimage.feature import hessian_matrix, hessian_matrix_eigvals
>>> square = np.zeros((5, 5))
>>> square[2, 2] = 4
>>> H_elems = hessian_matrix(square, sigma=0.1, order='rc',
...                             use_gaussian_derivatives=False)
>>> hessian_matrix_eigvals(H_elems)[0]
array([[ 0.,  0.,  2.,  0.,  0.],
       [ 0.,  1.,  0.,  1.,  0.],
       [ 2.,  0., -2.,  0.,  2.],
       [ 0.,  1.,  0.,  1.,  0.],
       [ 0.,  0.,  2.,  0.,  0.]])
```

`skimage.feature.hog(image, orientations=9, pixels_per_cell=(8, 8), cells_per_block=(3, 3),
block_norm='L2-Hys', visualize=False, transform_sqrt=False, feature_vector=True, *,
channel_axis=None)`

Extract Histogram of Oriented Gradients (HOG) for a given image.

Compute a Histogram of Oriented Gradients (HOG) by

1. (optional) global image normalization
2. computing the gradient image in *row* and *col*
3. computing gradient histograms
4. normalizing across blocks
5. flattening into a feature vector

Parameters

image

[$(M, N[, C])$ ndarray] Input image.

orientations

[int, optional] Number of orientation bins.

pixels_per_cell

[2-tuple (int, int), optional] Size (in pixels) of a cell.

cells_per_block

[2-tuple (int, int), optional] Number of cells in each block.

block_norm

[str {'L1', 'L1-sqrt', 'L2', 'L2-Hys'}, optional] Block normalization method:

L1

Normalization using L1-norm.

L1-sqrt

Normalization using L1-norm, followed by square root.

L2

Normalization using L2-norm.

L2-Hys

Normalization using L2-norm, followed by limiting the maximum values to 0.2 (*Hys* stands for *hysteresis*) and renormalization using L2-norm. (default) For details, see [?], [?].

visualize

[bool, optional] Also return an image of the HOG. For each cell and orientation bin, the image contains a line segment that is centered at the cell center, is perpendicular to the midpoint of the range of angles spanned by the orientation bin, and has intensity proportional to the corresponding histogram value.

transform_sqrt

[bool, optional] Apply power law compression to normalize the image before processing. DO NOT use this if the image contains negative values. Also see *notes* section below.

feature_vector

[bool, optional] Return the data as a feature vector by calling .ravel() on the result just before returning.

channel_axis

[int or None, optional] If None, the image is assumed to be a grayscale (single channel) image. Otherwise, this parameter indicates which axis of the array corresponds to channels.

New in version 0.19: *channel_axis* was added in 0.19.

Returns

out

[(n_blocks_row, n_blocks_col, n_cells_row, n_cells_col, n_orient) ndarray] HOG descriptor for the image. If *feature_vector* is True, a 1D (flattened) array is returned.

hog_image

[(M, N) ndarray, optional] A visualisation of the HOG image. Only provided if *visualize* is True.

Notes

The presented code implements the HOG extraction method from [?] with the following changes: (I) blocks of (3, 3) cells are used ((2, 2) in the paper); (II) no smoothing within cells (Gaussian spatial window with sigma=8pix in the paper); (III) L1 block normalization is used (L2-Hys in the paper).

Power law compression, also known as Gamma correction, is used to reduce the effects of shadowing and illumination variations. The compression makes the dark regions lighter. When the kwarg `transform_sqrt` is set to True, the function computes the square root of each color channel and then applies the hog algorithm to the image.

References

[?], [?], [?], [?]

- *Histogram of Oriented Gradients*

`skimage.feature.learn_gmm(descriptors, *, n_modes=32, gm_args=None)`

Estimate a Gaussian mixture model (GMM) given a set of descriptors and number of modes (i.e. Gaussians). This function is essentially a wrapper around the scikit-learn implementation of GMM, namely the `sklearn.mixture.GaussianMixture()` class.

Due to the nature of the Fisher vector, the only enforced parameter of the underlying scikit-learn class is the `covariance_type`, which must be ‘diag’.

There is no simple way to know what value to use for `n_modes` a-priori. Typically, the value is usually one of {16, 32, 64, 128}. One may train a few GMMs and choose the one that maximises the log probability of the GMM, or choose `n_modes` such that the downstream classifier trained on the resultant Fisher vectors has maximal performance.

Parameters

descriptors

[np.ndarray (N, M) or list [(N1, M), (N2, M), ...]] List of NumPy arrays, or a single NumPy array, of the descriptors used to estimate the GMM. The reason a list of NumPy arrays is permissible is because often when using a Fisher vector encoding, descriptors/vectors are computed separately for each sample/image in the dataset, such as SIFT vectors for each image. If a list is passed in, then each element must be a NumPy array in which the number of rows may differ (e.g. different number of SIFT vector for each image), but the number of columns for each must be the same (i.e. the dimensionality must be the same).

n_modes

[int] The number of modes/Gaussians to estimate during the GMM estimate.

gm_args

[dict] Keyword arguments that can be passed into the underlying scikit-learn `sklearn.mixture.GaussianMixture()` class.

Returns**gmm**

[`sklearn.mixture.GaussianMixture()`] The estimated GMM object, which contains the necessary parameters needed to compute the Fisher vector.

References

[?]

Examples

```
>>> import pytest
>>> _ = pytest.importorskip('sklearn')
>>> from skimage.feature import fisher_vector
>>> rng = np.random.Generator(np.random.PCG64())
>>> sift_for_images = [rng.standard_normal((10, 128)) for _ in range(10)]
>>> num_modes = 16
>>> # Estimate 16-mode GMM with these synthetic SIFT vectors
>>> gmm = learn_gmm(sift_for_images, n_modes=num_modes)
```

- *Fisher vector feature encoding*
-

`skimage.feature.local_binary_pattern(image, P, R, method='default')`

Compute the local binary patterns (LBP) of an image.

LBP is a visual descriptor often used in texture classification.

Parameters**image**

[(M, N) array] 2D grayscale image.

P

[int] Number of circularly symmetric neighbor set points (quantization of the angular space).

R

[float] Radius of circle (spatial resolution of the operator).

method

[str {‘default’, ‘ror’, ‘uniform’, ‘nri_uniform’, ‘var’}, optional] Method to determine the pattern:

default

Original local binary pattern which is grayscale invariant but not rotation invariant.

ror

Extension of default pattern which is grayscale invariant and rotation invariant.

uniform

Uniform pattern which is grayscale invariant and rotation invariant, offering finer quantization of the angular space. For details, see [?].

nri_uniform

Variant of uniform pattern which is grayscale invariant but not rotation invariant. For details, see [?] and [?].

var

Variance of local image texture (related to contrast) which is rotation invariant but not grayscale invariant.

Returns**output**

[(M, N) array] LBP image.

References

[?], [?], [?]

- *Local Binary Pattern for texture classification*

```
skimage.feature.match_descriptors(descriptors1, descriptors2, metric=None, p=2, max_distance=inf,
cross_check=True, max_ratio=1.0)
```

Brute-force matching of descriptors.

For each descriptor in the first set this matcher finds the closest descriptor in the second set (and vice-versa in the case of enabled cross-checking).

Parameters**descriptors1**

[(M, P) array] Descriptors of size P about M keypoints in the first image.

descriptors2

[(N, P) array] Descriptors of size P about N keypoints in the second image.

metric

[{‘euclidean’, ‘cityblock’, ‘minkowski’, ‘hamming’, …} , optional] The metric to compute the distance between two descriptors. See `scipy.spatial.distance.cdist` for all possible types. The hamming distance should be used for binary descriptors. By default the L2-norm is used for all descriptors of dtype float or double and the Hamming distance is used for binary descriptors automatically.

p

[int, optional] The p-norm to apply for `metric='minkowski'`.

max_distance

[float, optional] Maximum allowed distance between descriptors of two keypoints in separate images to be regarded as a match.

cross_check

[bool, optional] If True, the matched keypoints are returned after cross checking i.e. a matched pair (keypoint1, keypoint2) is returned if keypoint2 is the best match for keypoint1 in second image and keypoint1 is the best match for keypoint2 in first image.

max_ratio

[float, optional] Maximum ratio of distances between first and second closest descriptor in the second set of descriptors. This threshold is useful to filter ambiguous matches between the two descriptor sets. The choice of this value depends on the statistics of the chosen descriptor, e.g., for SIFT descriptors a value of 0.8 is usually chosen, see D.G. Lowe, “Distinctive Image Features from Scale-Invariant Keypoints”, International Journal of Computer Vision, 2004.

Returns**matches**

[(Q, 2) array] Indices of corresponding matches in first and second set of descriptors, where `matches[:, 0]` denote the indices in the first and `matches[:, 1]` the indices in the second set of descriptors.

- *Fundamental matrix estimation*
 - *ORB feature detector and binary descriptor*
 - *BRIEF binary descriptor*
 - *SIFT feature detector and descriptor extractor*
-

`skimage.feature.match_template(image, template, pad_input=False, mode='constant', constant_values=0)`

Match a template to a 2-D or 3-D image using normalized correlation.

The output is an array with values between -1.0 and 1.0. The value at a given position corresponds to the correlation coefficient between the image and the template.

For `pad_input=True` matches correspond to the center and otherwise to the top-left corner of the template. To find the best match you must search for peaks in the response (output) image.

Parameters

image

[(M, N[, D]) array] 2-D or 3-D input image.

template

[(m, n[, d]) array] Template to locate. It must be ($m \leq M, n \leq N[, d \leq D]$).

pad_input

[bool] If True, pad `image` so that output is the same size as the image, and output values correspond to the template center. Otherwise, the output is an array with shape ($M - m + 1, N - n + 1$) for an (M, N) image and an (m, n) template, and matches correspond to origin (top-left corner) of the template.

mode

[see `numpy.pad`, optional] Padding mode.

constant_values

[see `numpy.pad`, optional] Constant values used in conjunction with `mode='constant'`.

Returns

output

[array] Response image with correlation coefficients.

Notes

Details on the cross-correlation are presented in [?]. This implementation uses FFT convolutions of the image and the template. Reference [?] presents similar derivations but the approximation presented in this reference is not used in our implementation.

References

[?], [?]

Examples

```
>>> template = np.zeros((3, 3))
>>> template[1, 1] = 1
>>> template
array([[0., 0., 0.],
       [0., 1., 0.],
       [0., 0., 0.]])
>>> image = np.zeros((6, 6))
>>> image[1, 1] = 1
>>> image[4, 4] = -1
>>> image
array([[ 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., -1.,  0.]])
>>> result = match_template(image, template)
>>> np.round(result, 3)
array([[ 1. , -0.125,  0. ,  0. ],
       [-0.125, -0.125,  0. ,  0. ],
       [ 0. ,  0. ,  0.125,  0.125],
       [ 0. ,  0. ,  0.125, -1. ]])
>>> result = match_template(image, template, pad_input=True)
>>> np.round(result, 3)
array([[-0.125, -0.125, -0.125,  0. ,  0. ,  0. ],
       [-0.125,  1. , -0.125,  0. ,  0. ,  0. ],
       [-0.125, -0.125, -0.125,  0. ,  0. ,  0. ],
       [ 0. ,  0. ,  0. ,  0.125,  0.125,  0.125],
       [ 0. ,  0. ,  0. ,  0.125, -1. ,  0.125],
       [ 0. ,  0. ,  0. ,  0.125,  0.125,  0.125]])
```

- *Template Matching*
-

`skimage.feature.multiblock_lbp(int_image, r, c, width, height)`

Multi-block local binary pattern (MB-LBP).

The features are calculated similarly to local binary patterns (LBPs), (See `local_binary_pattern()`) except that summed blocks are used instead of individual pixel values.

MB-LBP is an extension of LBP that can be computed on multiple scales in constant time using the integral image. Nine equally-sized rectangles are used to compute a feature. For each rectangle, the sum of the pixel intensities is computed. Comparisons of these sums to that of the central rectangle determine the feature, similarly to LBP.

Parameters

`int_image`

`[(N, M) array]` Integral image.

r

[int] Row-coordinate of top left corner of a rectangle containing feature.

c

[int] Column-coordinate of top left corner of a rectangle containing feature.

width

[int] Width of one of the 9 equal rectangles that will be used to compute a feature.

height

[int] Height of one of the 9 equal rectangles that will be used to compute a feature.

Returns**output**

[int] 8-bit MB-LBP feature descriptor.

References

[?]

- *Multi-Block Local Binary Pattern for texture classification*

```
skimage.feature.multiscale_basic_features(image, intensity=True, edges=True, texture=True,
                                         sigma_min=0.5, sigma_max=16, num_sigma=None,
                                         num_workers=None, *, channel_axis=None)
```

Local features for a single- or multi-channel nd image.

Intensity, gradient intensity and local structure are computed at different scales thanks to Gaussian blurring.

Parameters**image**

[ndarray] Input image, which can be grayscale or multichannel.

intensity

[bool, default True] If True, pixel intensities averaged over the different scales are added to the feature set.

edges

[bool, default True] If True, intensities of local gradients averaged over the different scales are added to the feature set.

texture

[bool, default True] If True, eigenvalues of the Hessian matrix after Gaussian blurring at different scales are added to the feature set.

sigma_min

[float, optional] Smallest value of the Gaussian kernel used to average local neighborhoods before extracting features.

sigma_max

[float, optional] Largest value of the Gaussian kernel used to average local neighborhoods before extracting features.

num_sigma

[int, optional] Number of values of the Gaussian kernel between sigma_min and sigma_max. If None, sigma_min multiplied by powers of 2 are used.

num_workers

[int or None, optional] The number of parallel threads to use. If set to `None`, the full set of available cores are used.

channel_axis

[int or None, optional] If `None`, the image is assumed to be a grayscale (single channel) image. Otherwise, this parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns**features**

[np.ndarray] Array of shape `image.shape + (n_features,)`. When `channel_axis` is not `None`, all channels are concatenated along the features dimension. (i.e. `n_features == n_features_singlechannel * n_channels`)

- *Trainable segmentation using local features and random forests*
-

```
skimage.feature.peak_local_max(image, min_distance=1, threshold_abs=None, threshold_rel=None,  
                               exclude_border=True, num_peaks=inf, footprint=None, labels=None,  
                               num_peaks_per_label=inf, p_norm=inf)
```

Find peaks in an image as coordinate list.

Peaks are the local maxima in a region of $2 * \text{min_distance} + 1$ (i.e. peaks are separated by at least `min_distance`).

If both `threshold_abs` and `threshold_rel` are provided, the maximum of the two is chosen as the minimum intensity threshold of peaks.

Changed in version 0.18: Prior to version 0.18, peaks of the same height within a radius of `min_distance` were all returned, but this could cause unexpected behaviour. From 0.18 onwards, an arbitrary peak within the region is returned. See issue [gh-2592](#).

Parameters

image

[ndarray] Input image.

min_distance

[int, optional] The minimal allowed distance separating peaks. To find the maximum number of peaks, use `min_distance=1`.

threshold_abs

[float or None, optional] Minimum intensity of peaks. By default, the absolute threshold is the minimum intensity of the image.

threshold_rel

[float or None, optional] Minimum intensity of peaks, calculated as `max(image) * threshold_rel`.

exclude_border

[int, tuple of ints, or bool, optional] If positive integer, `exclude_border` excludes peaks from within `exclude_border`-pixels of the border of the image. If tuple of non-negative ints, the length of the tuple must match the input array's dimensionality. Each element of the tuple will exclude peaks from within `exclude_border`-pixels of the border of the image along that dimension. If True, takes the `min_distance` parameter as value. If zero or False, peaks are identified regardless of their distance from the border.

num_peaks

[int, optional] Maximum number of peaks. When the number of peaks exceeds `num_peaks`, return `num_peaks` peaks based on highest peak intensity.

footprint

[ndarray of bools, optional] If provided, `footprint == 1` represents the local region within which to search for peaks at every point in `image`.

labels

[ndarray of ints, optional] If provided, each unique region `labels == value` represents a unique region to search for peaks. Zero is reserved for background.

num_peaks_per_label

[int, optional] Maximum number of peaks for each label.

p_norm

[float] Which Minkowski p-norm to use. Should be in the range [1, inf]. A finite large p may cause a ValueError if overflow can occur. `inf` corresponds to the Chebyshev distance and 2 to the Euclidean distance.

Returns

output

[ndarray] The coordinates of the peaks.

See also:

`skimage.feature.corner_peaks`

Notes

The peak local maximum function returns the coordinates of local peaks (maxima) in an image. Internally, a maximum filter is used for finding local maxima. This operation dilates the original image. After comparison of the dilated and original images, this function returns the coordinates of the peaks where the dilated image equals the original image.

Examples

```
>>> img1 = np.zeros((7, 7))
>>> img1[3, 4] = 1
>>> img1[3, 2] = 1.5
>>> img1
array([[0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 1.5, 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.]])
```

```
>>> peak_local_max(img1, min_distance=1)
array([[3, 2],
       [3, 4]])
```

```
>>> peak_local_max(img1, min_distance=2)
array([[3, 2]])
```

```
>>> img2 = np.zeros((20, 20, 20))
>>> img2[10, 10, 10] = 1
>>> img2[15, 15, 15] = 1
>>> peak_idx = peak_local_max(img2, exclude_border=0)
>>> peak_idx
array([[10, 10, 10],
       [15, 15, 15]])
```

```
>>> peak_mask = np.zeros_like(img2, dtype=bool)
>>> peak_mask[tuple(peak_idx.T)] = True
>>> np.argwhere(peak_mask)
```

(continues on next page)

(continued from previous page)

```
array([[10, 10, 10],  
       [15, 15, 15]])
```

- *Finding local maxima*
 - *Watershed segmentation*
 - *Segment human cells (in mitosis)*
-

```
skimage.feature.plot_matches(ax, image1, image2, keypoints1, keypoints2, matches, keypoints_color='k',  
                           matches_color=None, only_matches=False, alignment='horizontal')
```

Plot matched features.

Parameters

ax

[matplotlib.axes.Axes] Matches and image are drawn in this ax.

image1

[(N, M [, 3]) array] First grayscale or color image.

image2

[(N, M [, 3]) array] Second grayscale or color image.

keypoints1

[(K1, 2) array] First keypoint coordinates as (row, col).

keypoints2

[(K2, 2) array] Second keypoint coordinates as (row, col).

matches

[(Q, 2) array] Indices of corresponding matches in first and second set of descriptors, where
`matches[:, 0]` denote the indices in the first and `matches[:, 1]` the indices in the
second set of descriptors.

keypoints_color

[matplotlib color, optional] Color for keypoint locations.

matches_color

[matplotlib color, optional] Color for lines which connect keypoint matches. By default the
color is chosen randomly.

only_matches

[bool, optional] Whether to only plot matches and not plot the keypoint locations.

alignment

[{‘horizontal’, ‘vertical’}, optional] Whether to show images side by side, ‘horizontal’, or one above the other, ‘vertical’.

- *Fundamental matrix estimation*
 - *Robust matching using RANSAC*
 - *ORB feature detector and binary descriptor*
 - *BRIEF binary descriptor*
 - *SIFT feature detector and descriptor extractor*
-

`skimage.feature.shape_index(image, sigma=1, mode='constant', cval=0)`

Compute the shape index.

The shape index, as defined by Koenderink & van Doorn [?], is a single valued measure of local curvature, assuming the image as a 3D plane with intensities representing heights.

It is derived from the eigenvalues of the Hessian, and its value ranges from -1 to 1 (and is undefined (=NaN) in flat regions), with following ranges representing following shapes:

Table 6: Ranges of the shape index and corresponding shapes.

Interval (s in ...)	Shape
[-1, -7/8)	Spherical cup
[-7/8, -5/8)	Through
[-5/8, -3/8)	Rut
[-3/8, -1/8)	Saddle rut
[-1/8, +1/8)	Saddle
[+1/8, +3/8)	Saddle ridge
[+3/8, +5/8)	Ridge
[+5/8, +7/8)	Dome
[+7/8, +1]	Spherical cap

Parameters**image**

[(M, N) ndarray] Input image.

sigma

[float, optional] Standard deviation used for the Gaussian kernel, which is used for smoothing the input data before Hessian eigen value calculation.

mode

[{‘constant’, ‘reflect’, ‘wrap’, ‘nearest’, ‘mirror’}, optional] How to handle values outside the image borders

cval

[float, optional] Used in conjunction with mode ‘constant’, the value outside the image boundaries.

Returns

s

[ndarray] Shape index

References

[?]

Examples

```
>>> from skimage.feature import shape_index
>>> square = np.zeros((5, 5))
>>> square[2, 2] = 4
>>> s = shape_index(square, sigma=0.1)
>>> s
array([[ nan,  nan, -0.5,  nan,  nan],
       [ nan, -0. ,  nan, -0. ,  nan],
       [-0.5,  nan, -1. ,  nan, -0.5],
       [ nan, -0. ,  nan, -0. ,  nan],
       [ nan,  nan, -0.5,  nan,  nan]])
```

- *Shape Index*

`skimage.feature.structure_tensor(image, sigma=1, mode='constant', cval=0, order='rc')`

Compute structure tensor using sum of squared differences.

The (2-dimensional) structure tensor A is defined as:

A = [Arr Arc] [Arc Acc]

which is approximated by the weighted sum of squared differences in a local window around each pixel in the image. This formula can be extended to a larger number of dimensions (see [?]).

Parameters

image

[ndarray] Input image.

sigma

[float or array-like of float, optional] Standard deviation used for the Gaussian kernel, which is used as a weighting function for the local summation of squared differences. If sigma is an iterable, its length must be equal to *image.ndim* and each element is used for the Gaussian kernel applied along its respective axis.

mode

[{‘constant’, ‘reflect’, ‘wrap’, ‘nearest’, ‘mirror’}, optional] How to handle values outside the image borders.

val

[float, optional] Used in conjunction with mode ‘constant’, the value outside the image boundaries.

order

[{‘rc’, ‘xy’}, optional] NOTE: ‘xy’ is only an option for 2D images, higher dimensions must always use ‘rc’ order. This parameter allows for the use of reverse or forward order of the image axes in gradient computation. ‘rc’ indicates the use of the first axis initially (Arr, Arc, Acc), whilst ‘xy’ indicates the usage of the last axis initially (Axx, Axy, Ayy).

Returns

A_elems

[list of ndarray] Upper-diagonal elements of the structure tensor for each pixel in the input image.

See also:

[*structure_tensor_eigenvalues*](#)

References

[?]

Examples

```
>>> from skimage.feature import structure_tensor
>>> square = np.zeros((5, 5))
>>> square[2, 2] = 1
>>> Arr, Arc, Acc = structure_tensor(square, sigma=0.1, order='rc')
>>> Acc
array([[0., 0., 0., 0., 0.],
       [0., 1., 0., 1., 0.],
       [0., 4., 0., 4., 0.],
       [0., 1., 0., 1., 0.],
       [0., 0., 0., 0., 0.]])
```

- Estimate anisotropy in a 3D microscopy image

`skimage.feature.structure_tensor_eigenvalues(A_elems)`

Compute eigenvalues of structure tensor.

Parameters

A_elems

[list of ndarray] The upper-diagonal elements of the structure tensor, as returned by `structure_tensor`.

Returns

ndarray

The eigenvalues of the structure tensor, in decreasing order. The eigenvalues are the leading dimension. That is, the coordinate [i, j, k] corresponds to the ith-largest eigenvalue at position (j, k).

See also:

`structure_tensor`

Examples

```
>>> from skimage.feature import structure_tensor
>>> from skimage.feature import structure_tensor_eigenvalues
>>> square = np.zeros((5, 5))
>>> square[2, 2] = 1
>>> A_elems = structure_tensor(square, sigma=0.1, order='rc')
>>> structure_tensor_eigenvalues(A_elems)[0]
array([[0., 0., 0., 0., 0.],
       [0., 2., 4., 2., 0.],
       [0., 4., 0., 4., 0.],
       [0., 2., 4., 2., 0.],
       [0., 0., 0., 0., 0.]])
```

- Estimate anisotropy in a 3D microscopy image

`class skimage.feature.BRIEF(descriptor_size=256, patch_size=49, mode='normal', sigma=1, rng=1)`

Bases: `DescriptorExtractor`

BRIEF binary descriptor extractor.

BRIEF (Binary Robust Independent Elementary Features) is an efficient feature point descriptor. It is highly discriminative even when using relatively few bits and is computed using simple intensity difference tests.

For each keypoint, intensity comparisons are carried out for a specifically distributed number N of pixel-pairs resulting in a binary descriptor of length N. For binary descriptors the Hamming distance can be used for feature matching, which leads to lower computational cost in comparison to the L2 norm.

Parameters

`descriptor_size`

[int, optional] Size of BRIEF descriptor for each keypoint. Sizes 128, 256 and 512 recommended by the authors. Default is 256.

`patch_size`

[int, optional] Length of the two dimensional square patch sampling region around the keypoints. Default is 49.

`mode`

[{'normal', 'uniform'}, optional] Probability distribution for sampling location of decision pixel-pairs around keypoints.

`rng`

[{`numpy.random.Generator`, int}, optional] Pseudo-random number generator (RNG). By default, a PCG64 generator is used (see `numpy.random.default_rng()`). If `rng` is an int, it is used to seed the generator.

The PRNG is used for the random sampling of the decision pixel-pairs. From a square window with length `patch_size`, pixel pairs are sampled using the `mode` parameter to build the descriptors using intensity comparison.

For matching across images, the same *rng* should be used to construct descriptors. To facilitate this:

- (a) *rng* defaults to 1
- (b) Subsequent calls of the `extract` method will use the same *rng*/seed.

sigma

[float, optional] Standard deviation of the Gaussian low-pass filter applied to the image to alleviate noise sensitivity, which is strongly recommended to obtain discriminative and good descriptors.

Examples

```
>>> from skimage.feature import (corner_harris, corner_peaks, BRIEF,
...                               match_descriptors)
>>> import numpy as np
>>> square1 = np.zeros((8, 8), dtype=np.int32)
>>> square1[2:6, 2:6] = 1
>>> square1
array([[0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 1, 0, 0, 0],
       [0, 0, 1, 1, 1, 0, 0, 0],
       [0, 0, 1, 1, 1, 0, 0, 0],
       [0, 0, 1, 1, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0]], dtype=int32)
>>> square2 = np.zeros((9, 9), dtype=np.int32)
>>> square2[2:7, 2:7] = 1
>>> square2
array([[0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 1, 1, 0, 0, 0],
       [0, 0, 1, 1, 1, 1, 0, 0, 0],
       [0, 0, 1, 1, 1, 1, 0, 0, 0],
       [0, 0, 1, 1, 1, 1, 0, 0, 0],
       [0, 0, 1, 1, 1, 1, 0, 0, 0],
       [0, 0, 1, 1, 1, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0]], dtype=int32)
>>> keypoints1 = corner_peaks(corner_harris(square1), min_distance=1)
>>> keypoints2 = corner_peaks(corner_harris(square2), min_distance=1)
>>> extractor = BRIEF(patch_size=5)
>>> extractor.extract(square1, keypoints1)
>>> descriptors1 = extractor.descriptors
>>> extractor.extract(square2, keypoints2)
>>> descriptors2 = extractor.descriptors
>>> matches = match_descriptors(descriptors1, descriptors2)
>>> matches
array([[0, 0],
       [1, 1],
       [2, 2],
       [3, 3]])
```

(continues on next page)

(continued from previous page)

```
>>> keypoints1[matches[:, 0]]  
array([[2, 2],  
       [2, 5],  
       [5, 2],  
       [5, 5]])  
>>> keypoints2[matches[:, 1]]  
array([[2, 2],  
       [2, 6],  
       [6, 2],  
       [6, 6]])
```

Attributes

descriptors

`[(Q, descriptor_size) array of dtype bool]` 2D ndarray of binary descriptors of size `descriptor_size` for Q keypoints after filtering out border keypoints with value at an index `(i, j)` either being `True` or `False` representing the outcome of the intensity comparison for i-th keypoint on j-th decision pixel-pair. It is `Q == np.sum(mask)`.

mask

`[(N,) array of dtype bool]` Mask indicating whether a keypoint has been filtered out (`False`) or is described in the `descriptors` array (`True`).

`__init__(descriptor_size=256, patch_size=49, mode='normal', sigma=1, rng=1)`

- *BRIEF binary descriptor*

`extract(image, keypoints)`

Extract BRIEF binary descriptors for given keypoints in image.

Parameters

image

[2D array] Input image.

keypoints

[`(N, 2)` array] Keypoint coordinates as `(row, col)`.

`class skimage.feature.CENSURE(min_scale=1, max_scale=7, mode='DoB', non_max_threshold=0.15, line_threshold=10)`

Bases: `FeatureDetector`

CENSURE keypoint detector.

min_scale

[int, optional] Minimum scale to extract keypoints from.

max_scale

[int, optional] Maximum scale to extract keypoints from. The keypoints will be extracted from all the scales except the first and the last i.e. from the scales in the range [min_scale + 1, max_scale - 1]. The filter sizes for different scales is such that the two adjacent scales comprise of an octave.

mode

[{'DoB', 'Octagon', 'STAR'}, optional] Type of bi-level filter used to get the scales of the input image. Possible values are 'DoB', 'Octagon' and 'STAR'. The three modes represent the shape of the bi-level filters i.e. box(square), octagon and star respectively. For instance, a bi-level octagon filter consists of a smaller inner octagon and a larger outer octagon with the filter weights being uniformly negative in both the inner octagon while uniformly positive in the difference region. Use STAR and Octagon for better features and DoB for better performance.

non_max_threshold

[float, optional] Threshold value used to suppress maximas and minimas with a weak magnitude response obtained after Non-Maximal Suppression.

line_threshold

[float, optional] Threshold for rejecting interest points which have ratio of principal curvatures greater than this value.

References

[?], [?]

Examples

```
>>> from skimage.data import astronaut
>>> from skimage.color import rgb2gray
>>> from skimage.feature import CENSURE
>>> img = rgb2gray(astronaut()[100:300, 100:300])
>>> censure = CENSURE()
>>> censure.detect(img)
>>> censure.keypoints
array([[ 4, 148],
       [12,  73],
       [21, 176],
       [91,  22],
       [93,  56],
       [94,  22],
       [95,  54],
       [100,  51],
       [103,  51],
       [106,  67],
       [108,  15],
```

(continues on next page)

(continued from previous page)

```
[117,  20],  
[122,  60],  
[125,  37],  
[129,  37],  
[133,  76],  
[145,  44],  
[146,  94],  
[150, 114],  
[153,  33],  
[154, 156],  
[155, 151],  
[184,  63]])  
>>> censure.scales  
array([2, 6, 6, 2, 4, 3, 2, 3, 2, 6, 3, 2, 2, 3, 2, 2, 2, 3, 2, 2, 4, 2,  
       2])
```

Attributes

keypoints

[(N, 2) array] Keypoint coordinates as (row, col).

scales

[(N,) array] Corresponding scales.

__init__(min_scale=1, max_scale=7, mode='DoB', non_max_threshold=0.15, line_threshold=10)

- *CENSURE feature detector*

detect(*image*)

Detect CENSURE keypoints along with the corresponding scale.

Parameters

image

[2D ndarray] Input image.

class skimage.feature.Cascade

Bases: `object`

Class for cascade of classifiers that is used for object detection.

The main idea behind cascade of classifiers is to create classifiers of medium accuracy and ensemble them into one strong classifier instead of just creating a strong one. The second advantage of cascade classifier is that easy

examples can be classified only by evaluating some of the classifiers in the cascade, making the process much faster than the process of evaluating a one strong classifier.

Notes

The cascade approach was first described by Viola and Jones [?], [?], although these initial publications used a set of Haar-like features. This implementation instead uses multi-scale block local binary pattern (MB-LBP) features [?].

References

[?], [?], [?]

Attributes

eps

[cnp.float32_t] Accuracy parameter. Increasing it, makes the classifier detect less false positives but at the same time the false negative score increases.

stages_number

[Py_ssize_t] Amount of stages in a cascade. Each cascade consists of stumps i.e. trained features.

stumps_number

[Py_ssize_t] The overall amount of stumps in all the stages of cascade.

features_number

[Py_ssize_t] The overall amount of different features used by cascade. Two stumps can use the same features but has different trained values.

window_width

[Py_ssize_t] The width of a detection window that is used. Objects smaller than this window can't be detected.

window_height

[Py_ssize_t] The height of a detection window.

stages

[Stage*] A pointer to the C array that stores stages information using a Stage struct.

features

[MLBPF*] A pointer to the C array that stores MLBPF features using an MLBPF struct.

LUTs

[cnp.uint32_t*] A pointer to the C array with look-up tables that are used by trained MLBPF features (MLBPFStumps) to evaluate a particular region.

`__init__()`

Initialize cascade classifier.

Parameters

`xml_file`

[file's path or file's object] A file in a OpenCv format from which all the cascade classifier's parameters are loaded.

`eps`

[cnp.float32_t] Accuracy parameter. Increasing it, makes the classifier detect less false positives but at the same time the false negative score increases.

- *Face detection using a cascade classifier*

`detect_multiscale()`

Search for the object on multiple scales of input image.

The function takes the input image, the scale factor by which the searching window is multiplied on each step, minimum window size and maximum window size that specify the interval for the search windows that are applied to the input image to detect objects.

Parameters

`img`

[2-D or 3-D ndarray] Ndarray that represents the input image.

`scale_factor`

[cnp.float32_t] The scale by which searching window is multiplied on each step.

`step_ratio`

[cnp.float32_t] The ratio by which the search step in multiplied on each scale of the image. 1 represents the exhaustive search and usually is slow. By setting this parameter to higher values the results will be worse but the computation will be much faster. Usually, values in the interval [1, 1.5] give good results.

`min_size`

[tuple (int, int)] Minimum size of the search window.

`max_size`

[tuple (int, int)] Maximum size of the search window.

`min_neighbor_number`

[int] Minimum amount of intersecting detections in order for detection to be approved by the function.

intersection_score_threshold

[cnp.float32_t] The minimum value of value of ratio (intersection area) / (small rectangle ratio) in order to merge two detections into one.

Returns**output**

[list of dicts] Dict have form {‘r’: int, ‘c’: int, ‘width’: int, ‘height’: int}, where ‘r’ represents row position of top left corner of detected window, ‘c’ - col position, ‘width’ - width of detected window, ‘height’ - height of detected window.

eps**features_number****stages_number****stumps_number****window_height****window_width**

```
class skimage.feature.ORB(downscale=1.2, n_scales=8, n_keypoints=500, fast_n=9, fast_threshold=0.08,
                           harris_k=0.04)
```

Bases: FeatureDetector, DescriptorExtractor

Oriented FAST and rotated BRIEF feature detector and binary descriptor extractor.

Parameters**n_keypoints**

[int, optional] Number of keypoints to be returned. The function will return the best *n_keypoints* according to the Harris corner response if more than *n_keypoints* are detected. If not, then all the detected keypoints are returned.

fast_n

[int, optional] The *n* parameter in `skimage.feature.corner_fast`. Minimum number

of consecutive pixels out of 16 pixels on the circle that should all be either brighter or darker w.r.t test-pixel. A point c on the circle is darker w.r.t test pixel p if $I_c < I_p - \text{threshold}$ and brighter if $I_c > I_p + \text{threshold}$. Also stands for the n in FAST-n corner detector.

fast_threshold

[float, optional] The threshold parameter in `feature.corner_fast`. Threshold used to decide whether the pixels on the circle are brighter, darker or similar w.r.t. the test pixel. Decrease the threshold when more corners are desired and vice-versa.

harris_k

[float, optional] The k parameter in `skimage.feature.corner_harris`. Sensitivity factor to separate corners from edges, typically in range [0, 0.2]. Small values of k result in detection of sharp corners.

downscale

[float, optional] Downscale factor for the image pyramid. Default value 1.2 is chosen so that there are more dense scales which enable robust scale invariance for a subsequent feature description.

n_scales

[int, optional] Maximum number of scales from the bottom of the image pyramid to extract the features from.

References

[?]

Examples

```
>>> from skimage.feature import ORB, match_descriptors
>>> img1 = np.zeros((100, 100))
>>> img2 = np.zeros_like(img1)
>>> rng = np.random.default_rng(19481137) # do not copy this value
>>> square = rng.random((20, 20))
>>> img1[40:60, 40:60] = square
>>> img2[53:73, 53:73] = square
>>> detector_extractor1 = ORB(n_keypoints=5)
>>> detector_extractor2 = ORB(n_keypoints=5)
>>> detector_extractor1.detect_and_extract(img1)
>>> detector_extractor2.detect_and_extract(img2)
>>> matches = match_descriptors(detector_extractor1.descriptors,
...                               detector_extractor2.descriptors)
>>> matches
array([[0, 0],
       [1, 1],
       [2, 2],
       [3, 4],
       [4, 3]])
>>> detector_extractor1.keypoints[matches[:, 0]]
```

(continues on next page)

(continued from previous page)

```
array([[59., 59.],
       [40., 40.],
       [57., 40.],
       [46., 58.],
       [58.8, 58.8]])
>>> detector_extractor2.keypoints[matches[:, 1]]
array([[72., 72.],
       [53., 53.],
       [70., 53.],
       [59., 71.],
       [72., 72.]])
```

Attributes

keypoints

[(N, 2) array] Keypoint coordinates as (row, col).

scales

[(N,) array] Corresponding scales.

orientations

[(N,) array] Corresponding orientations in radians.

responses

[(N,) array] Corresponding Harris corner responses.

descriptors

[(Q, descriptor_size) array of dtype bool] 2D array of binary descriptors of size *descriptor_size* for Q keypoints after filtering out border keypoints with value at an index (i, j) either being True or False representing the outcome of the intensity comparison for i-th keypoint on j-th decision pixel-pair. It is Q == np.sum(mask).

__init__(downscale=1.2, n_scales=8, n_keypoints=500, fast_n=9, fast_threshold=0.08, harris_k=0.04)

- *Fundamental matrix estimation*
- *ORB feature detector and binary descriptor*
- *Fisher vector feature encoding*

detect(*image*)

Detect oriented FAST keypoints along with the corresponding scale.

Parameters

image

[2D array] Input image.

detect_and_extract(*image*)

Detect oriented FAST keypoints and extract rBRIEF descriptors.

Note that this is faster than first calling *detect* and then *extract*.

Parameters

image

[2D array] Input image.

extract(*image*, *keypoints*, *scales*, *orientations*)

Extract rBRIEF binary descriptors for given keypoints in image.

Note that the keypoints must be extracted using the same *downscale* and *n_scales* parameters. Additionally, if you want to extract both keypoints and descriptors you should use the faster *detect_and_extract*.

Parameters

image

[2D array] Input image.

keypoints

[(N, 2) array] Keypoint coordinates as (row, col).

scales

[(N,) array] Corresponding scales.

orientations

[(N,) array] Corresponding orientations in radians.

```
class skimage.feature.SIFT(upsampling=2, n_octaves=8, n_scales=3, sigma_min=1.6, sigma_in=0.5,
                           c_dog=0.013333333333333334, c_edge=10, n_bins=36, lambda_ori=1.5,
                           c_max=0.8, lambda_descr=6, n_hist=4, n_ori=8)
```

Bases: FeatureDetector, DescriptorExtractor

SIFT feature detection and descriptor extraction.

Parameters

upsampling

[int, optional] Prior to the feature detection the image is upscaled by a factor of 1 (no upscaling), 2 or 4. Method: Bi-cubic interpolation.

n_octaves

[int, optional] Maximum number of octaves. With every octave the image size is halved and the sigma doubled. The number of octaves will be reduced as needed to keep at least 12 pixels along each dimension at the smallest scale.

n_scales

[int, optional] Maximum number of scales in every octave.

sigma_min

[float, optional] The blur level of the seed image. If upsampling is enabled sigma_min is scaled by factor 1/upsampling

sigma_in

[float, optional] The assumed blur level of the input image.

c_dog

[float, optional] Threshold to discard low contrast extrema in the DoG. Its final value is dependent on n_scales by the relation: $\text{final_c_dog} = (2^{(1/n_scales)-1}) / (2^{(1/3)-1}) * \text{c_dog}$

c_edge

[float, optional] Threshold to discard extrema that lie in edges. If H is the Hessian of an extremum, its “edgeness” is described by $\text{tr}(H)^2/\det(H)$. If the edgeness is higher than $(\text{c_edge} + 1)^2/\text{c_edge}$, the extremum is discarded.

n_bins

[int, optional] Number of bins in the histogram that describes the gradient orientations around keypoint.

lambda_ori

[float, optional] The window used to find the reference orientation of a keypoint has a width of $6 * \text{lambda_ori} * \text{sigma}$ and is weighted by a standard deviation of $2 * \text{lambda_ori} * \text{sigma}$.

c_max

[float, optional] The threshold at which a secondary peak in the orientation histogram is accepted as orientation

lambda_descr

[float, optional] The window used to define the descriptor of a keypoint has a width of $2 * \text{lambda_descr} * \text{sigma} * (\text{n_hist}+1)/\text{n_hist}$ and is weighted by a standard deviation of $\text{lambda_descr} * \text{sigma}$.

n_hist

[int, optional] The window used to define the descriptor of a keypoint consists of n_hist * n_hist histograms.

n_ori

[int, optional] The number of bins in the histograms of the descriptor patch.

Notes

The SIFT algorithm was developed by David Lowe [?], [?] and later patented by the University of British Columbia. Since the patent expired in 2020 it's free to use. The implementation here closely follows the detailed description in [?], including use of the same default parameters.

References

[?], [?], [?]

Examples

```
>>> from skimage.feature import SIFT, match_descriptors
>>> from skimage.data import camera
>>> from skimage.transform import rotate
>>> img1 = camera()
>>> img2 = rotate(camera(), 90)
>>> detector_extractor1 = SIFT()
>>> detector_extractor2 = SIFT()
>>> detector_extractor1.detect_and_extract(img1)
>>> detector_extractor2.detect_and_extract(img2)
>>> matches = match_descriptors(detector_extractor1.descriptors,
...                               detector_extractor2.descriptors,
...                               max_ratio=0.6)
>>> matches[10:15]
array([[ 10,  412],
       [ 11,  417],
       [ 12,  407],
       [ 13,  411],
       [ 14,  406]])
>>> detector_extractor1.keypoints[matches[10:15, 0]]
array([[ 95,  214],
       [ 97,  211],
       [ 97,  218],
       [102,  215],
       [104,  218]])
>>> detector_extractor2.keypoints[matches[10:15, 1]]
array([[297,   95],
       [301,   97],
       [294,   97],
       [297,  102],
       [293,  104]])
```

Attributes

delta_min

[float] The sampling distance of the first octave. It's final value is 1/upsampling.

float_dtype

[type] The datatype of the image.

scalespace_sigmas

[(n_octaves, n_scales + 3) array] The sigma value of all scales in all octaves.

keypoints

[(N, 2) array] Keypoint coordinates as (row, col).

positions

[(N, 2) array] Subpixel-precision keypoint coordinates as (row, col).

sigmas

[(N,) array] The corresponding sigma (blur) value of a keypoint.

scales

[(N,) array] The corresponding scale of a keypoint.

orientations

[(N,) array] The orientations of the gradient around every keypoint.

octaves

[(N,) array] The corresponding octave of a keypoint.

descriptors

[(N, n_hist*n_hist*n_ori) array] The descriptors of a keypoint.

```
__init__(upsampling=2, n_octaves=8, n_scales=3, sigma_min=1.6, sigma_in=0.5,  
        c_dog=0.01333333333333334, c_edge=10, n_bins=36, lambda_ori=1.5, c_max=0.8,  
        lambda_descr=6, n_hist=4, n_ori=8)
```

- *SIFT feature detector and descriptor extractor*

property deltas

The sampling distances of all octaves

detect(image)

Detect the keypoints.

Parameters

image

[2D array] Input image.

`detect_and_extract(image)`

Detect the keypoints and extract their descriptors.

Parameters

image

[2D array] Input image.

`extract(image)`

Extract the descriptors for all keypoints in the image.

Parameters

image

[2D array] Input image.

1.3.7 skimage.filters

<code>skimage.filters.apply_hysteresis_threshold</code>	Apply hysteresis thresholding to <code>image</code> .
<code>skimage.filters.butterworth</code>	Apply a Butterworth filter to enhance high or low frequency features.
<code>skimage.filters.compute_hessian_eigenvalues</code>	Deprecated: <code>compute_hessian_eigenvalues</code> is deprecated since version 0.20 and will be removed in version 0.22.
<code>skimage.filters.correlate_sparse</code>	Compute valid cross-correlation of <code>padded_array</code> and <code>kernel</code> .
<code>skimage.filters.difference_of_gaussians</code>	Find features between <code>low_sigma</code> and <code>high_sigma</code> in size.
<code>skimage.filters.farid</code>	Find the edge magnitude using the Farid transform.
<code>skimage.filters.farid_h</code>	Find the horizontal edges of an image using the Farid transform.
<code>skimage.filters.farid_v</code>	Find the vertical edges of an image using the Farid transform.
<code>skimage.filters.filter_forward</code>	Apply the given filter to data.
<code>skimage.filters.filter_inverse</code>	Apply the filter in reverse to the given data.
<code>skimage.filters.frangi</code>	Filter an image with the Frangi vesselness filter.
<code>skimage.filters.gabor</code>	Return real and imaginary responses to Gabor filter.
<code>skimage.filters.gabor_kernel</code>	Return complex 2D Gabor filter kernel.

continues on next page

Table 7 – continued from previous page

<code>skimage.filters.gaussian</code>	Multi-dimensional Gaussian filter.
<code>skimage.filters.hessian</code>	Filter an image with the Hybrid Hessian filter.
<code>skimage.filters.inverse</code>	Deprecated: <code>inverse</code> is deprecated since version 0.20 and will be removed in version 0.22.
<code>skimage.filters.laplace</code>	Find the edges of an image using the Laplace operator.
<code>skimage.filters.median</code>	Return local median of an image.
<code>skimage.filters.meijering</code>	Filter an image with the Meijering neuriteness filter.
<code>skimage.filters.prewitt</code>	Find the edge magnitude using the Prewitt transform.
<code>skimage.filters.prewitt_h</code>	Find the horizontal edges of an image using the Prewitt transform.
<code>skimage.filters.prewitt_v</code>	Find the vertical edges of an image using the Prewitt transform.
<code>skimage.filters.rank_order</code>	Return an image of the same shape where each pixel is the index of the pixel value in the ascending order of the unique values of <code>image</code> , aka the rank-order value.
<code>skimage.filters.roberts</code>	Find the edge magnitude using Roberts' cross operator.
<code>skimage.filters.roberts_neg_diag</code>	Find the cross edges of an image using the Roberts' Cross operator.
<code>skimage.filters.roberts_pos_diag</code>	Find the cross edges of an image using Roberts' cross operator.
<code>skimage.filters.sato</code>	Filter an image with the Sato tubeness filter.
<code>skimage.filters.scharr</code>	Find the edge magnitude using the Scharr transform.
<code>skimage.filters.scharr_h</code>	Find the horizontal edges of an image using the Scharr transform.
<code>skimage.filters.scharr_v</code>	Find the vertical edges of an image using the Scharr transform.
<code>skimage.filters.sobel</code>	Find edges in an image using the Sobel filter.
<code>skimage.filters.sobel_h</code>	Find the horizontal edges of an image using the Sobel transform.
<code>skimage.filters.sobel_v</code>	Find the vertical edges of an image using the Sobel transform.
<code>skimage.filters.threshold_isodata</code>	Return threshold value(s) based on ISODATA method.
<code>skimage.filters.threshold_li</code>	Compute threshold value by Li's iterative Minimum Cross Entropy method.
<code>skimage.filters.threshold_local</code>	Compute a threshold mask image based on local pixel neighborhood.
<code>skimage.filters.threshold_mean</code>	Return threshold value based on the mean of grayscale values.
<code>skimage.filters.threshold_minimum</code>	Return threshold value based on minimum method.
<code>skimage.filters.threshold_multithreshold</code>	Generate <code>classes</code> -1 threshold values to divide gray levels in <code>image</code> , following Otsu's method for multiple classes.
<code>skimage.filters.threshold_niblack</code>	Applies Niblack local threshold to an array.
<code>skimage.filters.threshold_otsu</code>	Return threshold value based on Otsu's method.
<code>skimage.filters.threshold_sauvola</code>	Applies Sauvola local threshold to an array.
<code>skimage.filters.threshold_triangle</code>	Return threshold value based on the triangle algorithm.
<code>skimage.filters.threshold_yen</code>	Return threshold value based on Yen's method.
<code>skimage.filters.try_all_threshold</code>	Returns a figure comparing the outputs of different thresholding methods.
<code>skimage.filters.unsharp_mask</code>	Unsharp masking filter.
<code>skimage.filters.wiener</code>	Minimum Mean Square Error (Wiener) inverse filter.
<code>skimage.filters.window</code>	Return an n-dimensional window of a given size and dimensionality.

continues on next page

Table 7 – continued from previous page

<code>skimage.filters.LPIFilter2D</code>	Linear Position-Invariant Filter (2-dimensional)
<code>skimage.filters.rank</code>	

`skimage.filters.apply_hysteresis_threshold(image, low, high)`

Apply hysteresis thresholding to `image`.

This algorithm finds regions where `image` is greater than `high` OR `image` is greater than `low` and that region is connected to a region greater than `high`.

Parameters

`image`

[array, shape (M,[N, ..., P])] Grayscale input image.

`low`

[float, or array of same shape as `image`] Lower threshold.

`high`

[float, or array of same shape as `image`] Higher threshold.

Returns

`thresholded`

[array of bool, same shape as `image`] Array in which True indicates the locations where `image` was above the hysteresis threshold.

References

[?]

Examples

```
>>> image = np.array([1, 2, 3, 2, 1, 2, 1, 3, 2])
>>> apply_hysteresis_threshold(image, 1.5, 2.5).astype(int)
array([0, 1, 1, 1, 0, 0, 0, 1, 1])
```

- *Hysteresis thresholding*
 - *Use pixel graphs to find an object's geodesic center*
-

```
skimage.filters.butterworth(image, cutoff_frequency_ratio=0.005, high_pass=True, order=2.0,
                             channel_axis=None, *, squared_butterworth=True, npad=0)
```

Apply a Butterworth filter to enhance high or low frequency features.

This filter is defined in the Fourier domain.

Parameters

image

[$(M[, N[, \dots, P]][, C])$ ndarray] Input image.

cutoff_frequency_ratio

[float, optional] Determines the position of the cut-off relative to the shape of the FFT. Receives a value between [0, 0.5].

high_pass

[bool, optional] Whether to perform a high pass filter. If False, a low pass filter is performed.

order

[float, optional] Order of the filter which affects the slope near the cut-off. Higher order means steeper slope in frequency space.

channel_axis

[int, optional] If there is a channel dimension, provide the index here. If None (default) then all axes are assumed to be spatial dimensions.

squared_butterworth

[bool, optional] When True, the square of a Butterworth filter is used. See notes below for more details.

npad

[int, optional] Pad each edge of the image by $npad$ pixels using `numpy.pad`'s `mode='edge'` extension.

Returns

result

[ndarray] The Butterworth-filtered image.

Notes

A band-pass filter can be achieved by combining a high-pass and low-pass filter. The user can increase `npad` if boundary artifacts are apparent.

The “Butterworth filter” used in image processing textbooks (e.g. [?], [?]) is often the square of the traditional Butterworth filters as described by [?], [?]. The squared version will be used here if `squared_butterworth` is set to True. The lowpass, squared Butterworth filter is given by the following expression for the lowpass case:

$$H_{low}(f) = \frac{1}{1 + \left(\frac{f}{cf_s}\right)^{2n}}$$

with the highpass case given by

$$H_{hi}(f) = 1 - H_{low}(f)$$

where $f = \sqrt{\sum_{d=0}^{\text{ndim}} f_d^2}$ is the absolute value of the spatial frequency, f_s is the sampling frequency, c the `cutoff_frequency_ratio`, and n is the filter `order` [?]. When `squared_butterworth=False`, the square root of the above expressions are used instead.

Note that `cutoff_frequency_ratio` is defined in terms of the sampling frequency, f_s . The FFT spectrum covers the Nyquist range ($[-f_s/2, f_s/2]$) so `cutoff_frequency_ratio` should have a value between 0 and 0.5. The frequency response (gain) at the cutoff is 0.5 when `squared_butterworth` is true and $1/\sqrt{2}$ when it is false.

References

[?], [?], [?], [?]

Examples

Apply a high-pass and low-pass Butterworth filter to a grayscale and color image respectively:

```
>>> from skimage.data import camera, astronaut
>>> from skimage.filters import butterworth
>>> high_pass = butterworth(camera(), 0.07, True, 8)
>>> low_pass = butterworth(astronaut(), 0.01, False, 4, channel_axis=-1)
```

- *Butterworth Filters*

```
skimage.filters.compute_hessian_eigenvalues(image, sigma, sorting='none', mode='constant', cval=0,
                                             use_gaussian_derivatives=False)
```

Deprecated: `compute_hessian_eigenvalues` is deprecated since version 0.20 and will be removed in version 0.22. Use `skimage.feature.hessian_matrix_eigvals` on the results of `skimage.feature.hessian_matrix` instead.

Compute Hessian eigenvalues of nD images.

For 2D images, the computation uses a more efficient, skimage-based algorithm.

Parameters

image

[(N, ..., M) ndarray] Array with input image data.

sigma

[float] Smoothing factor of image for detection of structures at different (sigma) scales.

sorting

[{'val', 'abs', 'none'}, optional] Sorting of eigenvalues by values ('val') or absolute values ('abs'), or without sorting ('none'). Default is 'none'.

mode

[{'constant', 'reflect', 'wrap', 'nearest', 'mirror'}, optional] How to handle values outside the image borders.

cval

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

use_gaussian_derivatives

[boolean, optional] Indicates whether the Hessian is computed by convolving with Gaussian derivatives, or by a simple finite-difference operation.

Returns**eigenvalues**

[(D, N, ..., M) ndarray] Array with (sorted) eigenvalues of Hessian eigenvalues for each pixel of the input image.

`skimage.filters.correlate_sparse(image, kernel, mode='reflect')`

Compute valid cross-correlation of *padded_array* and *kernel*.

This function is *fast* when *kernel* is large with many zeros.

See `scipy.ndimage.correlate` for a description of cross-correlation.

Parameters**image**

[ndarray, dtype float, shape (M, N,[...,] P)] The input array. If mode is 'valid', this array should already be padded, as a margin of the same shape as kernel will be stripped off.

kernel

[ndarray, dtype float shape (Q, R,[...,] S)] The kernel to be correlated. Must have the same number of dimensions as *padded_array*. For high performance, it should be sparse (few nonzero entries).

mode

[string, optional] See `scipy.ndimage.correlate` for valid modes. Additionally, mode ‘valid’ is accepted, in which case no padding is applied and the result is the result for the smaller image for which the kernel is entirely inside the original data.

Returns**result**

[array of float, shape (M, N,[...], P)] The result of cross-correlating *image* with *kernel*. If mode ‘valid’ is used, the resulting shape is (M-Q+1, N-R+1,[...], P-S+1).

`skimage.filters.difference_of_gaussians(image, low_sigma, high_sigma=None, *, mode='nearest', eval=0, channel_axis=None, truncate=4.0)`

Find features between `low_sigma` and `high_sigma` in size.

This function uses the Difference of Gaussians method for applying band-pass filters to multi-dimensional arrays. The input array is blurred with two Gaussian kernels of differing sigmas to produce two intermediate, filtered images. The more-blurred image is then subtracted from the less-blurred image. The final output image will therefore have had high-frequency components attenuated by the smaller-sigma Gaussian, and low frequency components will have been removed due to their presence in the more-blurred intermediate.

Parameters**image**

[ndarray] Input array to filter.

low_sigma

[scalar or sequence of scalars] Standard deviation(s) for the Gaussian kernel with the smaller sigmas across all axes. The standard deviations are given for each axis as a sequence, or as a single number, in which case the single number is used as the standard deviation value for all axes.

high_sigma

[scalar or sequence of scalars, optional (default is None)] Standard deviation(s) for the Gaussian kernel with the larger sigmas across all axes. The standard deviations are given for each axis as a sequence, or as a single number, in which case the single number is used as the standard deviation value for all axes. If None is given (default), sigmas for all axes are calculated as 1.6 * `low_sigma`.

mode

[{‘reflect’, ‘constant’, ‘nearest’, ‘mirror’, ‘wrap’}, optional] The `mode` parameter determines how the array borders are handled, where `cval` is the value when `mode` is equal to ‘constant’. Default is ‘nearest’.

cval

[scalar, optional] Value to fill past edges of input if `mode` is ‘constant’. Default is 0.0

channel_axis

[int or None, optional] If None, the image is assumed to be a grayscale (single channel) image. Otherwise, this parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

truncate

[float, optional (default is 4.0)] Truncate the filter at this many standard deviations.

Returns**filtered_image**

[ndarray] the filtered array.

See also:

`skimage.feature.blob_dog`

Notes

This function will subtract an array filtered with a Gaussian kernel with sigmas given by `high_sigma` from an array filtered with a Gaussian kernel with sigmas provided by `low_sigma`. The values for `high_sigma` must always be greater than or equal to the corresponding values in `low_sigma`, or a `ValueError` will be raised.

When `high_sigma` is none, the values for `high_sigma` will be calculated as 1.6x the corresponding values in `low_sigma`. This ratio was originally proposed by Marr and Hildreth (1980) [?] and is commonly used when approximating the inverted Laplacian of Gaussian, which is used in edge and blob detection.

Input image is converted according to the conventions of `img_as_float`.

Except for sigma values, all parameters are used for both filters.

References

[?]

Examples

Apply a simple Difference of Gaussians filter to a color image:

```
>>> from skimage.data import astronaut
>>> from skimage.filters import difference_of_gaussians
>>> filtered_image = difference_of_gaussians(astronaut(), 2, 10,
...                                             channel_axis=-1)
```

Apply a Laplacian of Gaussian filter as approximated by the Difference of Gaussians filter:

```
>>> filtered_image = difference_of_gaussians(astronaut(), 2,
...                                             channel_axis=-1)
```

Apply a Difference of Gaussians filter to a grayscale image using different sigma values for each axis:

```
>>> from skimage.data import camera
>>> filtered_image = difference_of_gaussians(camera(), (2,5), (3,20))
```

- *Using Polar and Log-Polar Transformations for Registration*
 - *Band-pass filtering by Difference of Gaussians*
-

`skimage.filters.farid(image, mask=None, *, axis=None, mode='reflect', cval=0.0)`

Find the edge magnitude using the Farid transform.

Parameters

image

[array] The input image.

mask

[array of bool, optional] Clip the output image to this mask. (Values where mask=0 will be set to 0.)

axis

[int or sequence of int, optional] Compute the edge filter along this axis. If not provided, the edge magnitude is computed. This is defined as:

```
farid_mag = np.sqrt(sum([farid(image, axis=i)**2
                         for i in range(image.ndim)]) / image.ndim)
```

The magnitude is also computed if axis is a sequence.

mode

[str or sequence of str, optional] The boundary mode for the convolution. See `scipy.ndimage.convolve` for a description of the modes. This can be either a single boundary mode or one boundary mode per axis.

cval

[float, optional] When `mode` is 'constant', this is the constant used in values outside the boundary of the image data.

Returns

output

[array of float] The Farid edge map.

See also:

`farid_h, farid_v`

horizontal and vertical edge detection.

`scharr, sobel, prewitt, skimage.feature.canny`**Notes**

Take the square root of the sum of the squares of the horizontal and vertical derivatives to get a magnitude that is somewhat insensitive to direction. Similar to the Scharr operator, this operator is designed with a rotation invariance constraint.

References

[?], [?]

Examples

```
>>> from skimage import data
>>> camera = data.camera()
>>> from skimage import filters
>>> edges = filters.farid(camera)
```

`skimage.filters.farid_h(image, *, mask=None)`

Find the horizontal edges of an image using the Farid transform.

Parameters**image**

[2-D array] Image to process.

mask

[2-D array, optional] An optional mask to limit the application to a certain area. Note that pixels surrounding masked regions are also masked to prevent masked regions from affecting the result.

Returns**output**

[2-D array] The Farid edge map.

Notes

The kernel was constructed using the 5-tap weights from [1].

References

[?], [?]

- *Edge operators*
-

`skimage.filters.farid_v(image, *, mask=None)`

Find the vertical edges of an image using the Farid transform.

Parameters

image

[2-D array] Image to process.

mask

[2-D array, optional] An optional mask to limit the application to a certain area. Note that pixels surrounding masked regions are also masked to prevent masked regions from affecting the result.

Returns

output

[2-D array] The Farid edge map.

Notes

The kernel was constructed using the 5-tap weights from [1].

References

[?]

- *Edge operators*
-

```
skimage.filters.filter_forward(data, impulse_response=None, filter_params=None,
                               predefined_filter=None)
```

Apply the given filter to data.

Parameters

data

[(M, N) ndarray] Input data.

impulse_response

[callable $f(r, c, \text{**}filter_params)$] Impulse response of the filter. See `LPIFilter2D.__init__`.

filter_params

[dict, optional] Additional keyword parameters to the `impulse_response` function.

Other Parameters

predefined_filter

[`LPIFilter2D`] If you need to apply the same filter multiple times over different images, construct the `LPIFilter2D` and specify it here.

Examples

Gaussian filter without normalization:

```
>>> def filt_func(r, c, sigma=1):
...     return np.exp(-(r**2 + c**2)/(2 * sigma**2))
>>>
>>> from skimage import data
>>> filtered = filter_forward(data.coins(), filt_func)
```

```
skimage.filters.filter_inverse(data, impulse_response=None, filter_params=None, max_gain=2,
                               predefined_filter=None)
```

Apply the filter in reverse to the given data.

Parameters

data

[(M,N) ndarray] Input data.

impulse_response

[callable $f(r, c, \text{**}filter_params)$] Impulse response of the filter. See `LPIFilter2D`. This is a required argument unless a `predefined_filter` is provided.

filter_params

[dict, optional] Additional keyword parameters to the impulse_response function.

max_gain

[float, optional] Limit the filter gain. Often, the filter contains zeros, which would cause the inverse filter to have infinite gain. High gain causes amplification of artefacts, so a conservative limit is recommended.

Other Parameters

predefined_filter

[LPIFilter2D, optional] If you need to apply the same filter multiple times over different images, construct the LPIFilter2D and specify it here.

`skimage.filters.frangi(image, sigmas=range(1, 10, 2), scale_range=None, scale_step=None, alpha=0.5, beta=0.5, gamma=None, black_ridges=True, mode='reflect', cval=0)`

Filter an image with the Frangi vesselness filter.

This filter can be used to detect continuous ridges, e.g. vessels, wrinkles, rivers. It can be used to calculate the fraction of the whole image containing such objects.

Defined only for 2-D and 3-D images. Calculates the eigenvectors of the Hessian to compute the similarity of an image region to vessels, according to the method described in [?].

Parameters

image

[(N, M[, P]) ndarray] Array with input image data.

sigmas

[iterable of floats, optional] Sigmas used as scales of filter, i.e., `np.arange(scale_range[0], scale_range[1], scale_step)`

scale_range

[2-tuple of floats, optional] The range of sigmas used.

scale_step

[float, optional] Step size between sigmas.

alpha

[float, optional] Frangi correction constant that adjusts the filter's sensitivity to deviation from a plate-like structure.

beta

[float, optional] Frangi correction constant that adjusts the filter's sensitivity to deviation from a blob-like structure.

gamma

[float, optional] Frangi correction constant that adjusts the filter's sensitivity to areas of high variance/texture/structure. The default, None, uses half of the maximum Hessian norm.

black_ridges

[boolean, optional] When True (the default), the filter detects black ridges; when False, it detects white ridges.

mode

[{‘constant’, ‘reflect’, ‘wrap’, ‘nearest’, ‘mirror’}, optional] How to handle values outside the image borders.

cval

[float, optional] Used in conjunction with mode ‘constant’, the value outside the image boundaries.

Returns**out**

[(N, M[, P]) ndarray] Filtered image (maximum of pixels across all scales).

See also:

[meijering](#)
[sato](#)
[hessian](#)

Notes

Earlier versions of this filter were implemented by Marc Schrijver, (November 2001), D. J. Kroon, University of Twente (May 2009) [?], and D. G. Ellis (January 2017) [?].

References

[?], [?], [?]

- *Ridge operators*

```
skimage.filters.gabor(image, frequency, theta=0, bandwidth=1, sigma_x=None, sigma_y=None, n_stds=3,
                      offset=0, mode='reflect', cval=0)
```

Return real and imaginary responses to Gabor filter.

The real and imaginary parts of the Gabor filter kernel are applied to the image and the response is returned as a pair of arrays.

Gabor filter is a linear filter with a Gaussian kernel which is modulated by a sinusoidal plane wave. Frequency and orientation representations of the Gabor filter are similar to those of the human visual system. Gabor filter banks are commonly used in computer vision and image processing. They are especially suitable for edge detection and texture classification.

Parameters

image

[2-D array] Input image.

frequency

[float] Spatial frequency of the harmonic function. Specified in pixels.

theta

[float, optional] Orientation in radians. If 0, the harmonic is in the x-direction.

bandwidth

[float, optional] The bandwidth captured by the filter. For fixed bandwidth, `sigma_x` and `sigma_y` will decrease with increasing frequency. This value is ignored if `sigma_x` and `sigma_y` are set by the user.

sigma_x, sigma_y

[float, optional] Standard deviation in x- and y-directions. These directions apply to the kernel *before* rotation. If `theta = pi/2`, then the kernel is rotated 90 degrees so that `sigma_x` controls the *vertical* direction.

n_stds

[scalar, optional] The linear size of the kernel is `n_stds` (3 by default) standard deviations.

offset

[float, optional] Phase offset of harmonic function in radians.

mode

[{‘constant’, ‘nearest’, ‘reflect’, ‘mirror’, ‘wrap’ }, optional] Mode used to convolve image with a kernel, passed to `ndi.convolve`

cval

[scalar, optional] Value to fill past edges of input if `mode` of convolution is ‘constant’. The parameter is passed to `ndi.convolve`.

Returns

real, imag

[arrays] Filtered images using the real and imaginary parts of the Gabor filter kernel. Images are of the same dimensions as the input one.

References

[?], [?]

Examples

```
>>> from skimage.filters import gabor
>>> from skimage import data, io
>>> from matplotlib import pyplot as plt
```

```
>>> image = data.coins()
>>> # detecting edges in a coin image
>>> filt_real, filt_imag = gabor(image, frequency=0.6)
>>> plt.figure()
>>> io.imshow(filt_real)
>>> io.show()
```

```
>>> # less sensitivity to finer details with the lower frequency kernel
>>> filt_real, filt_imag = gabor(image, frequency=0.1)
>>> plt.figure()
>>> io.imshow(filt_real)
>>> io.show()
```

`skimage.filters.gabor_kernel`(*frequency*, *theta*=0, *bandwidth*=1, *sigma_x*=None, *sigma_y*=None, *n_stds*=3, *offset*=0, *dtype*=<class 'numpy.complex128'>)

Return complex 2D Gabor filter kernel.

Gabor kernel is a Gaussian kernel modulated by a complex harmonic function. Harmonic function consists of an imaginary sine function and a real cosine function. Spatial frequency is inversely proportional to the wavelength of the harmonic and to the standard deviation of a Gaussian kernel. The bandwidth is also inversely proportional to the standard deviation.

Parameters

frequency

[float] Spatial frequency of the harmonic function. Specified in pixels.

theta

[float, optional] Orientation in radians. If 0, the harmonic is in the x-direction.

bandwidth

[float, optional] The bandwidth captured by the filter. For fixed bandwidth, *sigma_x* and *sigma_y* will decrease with increasing frequency. This value is ignored if *sigma_x* and *sigma_y* are set by the user.

sigma_x, sigma_y

[float, optional] Standard deviation in x- and y-directions. These directions apply to the kernel *before* rotation. If $\theta = \pi/2$, then the kernel is rotated 90 degrees so that `sigma_x` controls the *vertical* direction.

n_stds

[scalar, optional] The linear size of the kernel is `n_stds` (3 by default) standard deviations

offset

[float, optional] Phase offset of harmonic function in radians.

dtype

[{np.complex64, np.complex128}] Specifies if the filter is single or double precision complex.

Returns**g**

[complex array] Complex filter kernel.

References

[?], [?]

Examples

```
>>> from skimage.filters import gabor_kernel
>>> from skimage import io
>>> from matplotlib import pyplot as plt
```

```
>>> gk = gabor_kernel(frequency=0.2)
>>> plt.figure()
>>> io.imshow(gk.real)
>>> io.show()
```

```
>>> # more ripples (equivalent to increasing the size of the
>>> # Gaussian spread)
>>> gk = gabor_kernel(frequency=0.2, bandwidth=0.1)
>>> plt.figure()
>>> io.imshow(gk.real)
>>> io.show()
```

- *Gabor filter banks for texture classification*
-

```
skimage.filters.gaussian(image, sigma=1, output=None, mode='nearest', cval=0, preserve_range=False, truncate=4.0, *, channel_axis=<ChannelAxisNotSet>)
```

Multi-dimensional Gaussian filter.

Parameters

image

[array-like] Input image (grayscale or color) to filter.

sigma

[scalar or sequence of scalars, optional] Standard deviation for Gaussian kernel. The standard deviations of the Gaussian filter are given for each axis as a sequence, or as a single number, in which case it is equal for all axes.

output

[array, optional] The `output` parameter passes an array in which to store the filter output.

mode

[{‘reflect’, ‘constant’, ‘nearest’, ‘mirror’, ‘wrap’}, optional] The `mode` parameter determines how the array borders are handled, where `cval` is the value when mode is equal to ‘constant’. Default is ‘nearest’.

cval

[scalar, optional] Value to fill past edges of input if `mode` is ‘constant’. Default is 0.0

preserve_range

[bool, optional] If True, keep the original range of values. Otherwise, the input `image` is converted according to the conventions of `img_as_float` (Normalized first to values [-1.0 ; 1.0] or [0 ; 1.0] depending on `dtype` of input)

For more information, see: https://scikit-image.org/docs/dev/user_guide/data_types.html

truncate

[float, optional] Truncate the filter at this many standard deviations.

channel_axis

[int or None, optional] If None, the image is assumed to be a grayscale (single channel) image. Otherwise, this parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Warning: Automatic detection of the color channel based on the old deprecated `multi-channel=None` was broken in version 0.19. In 0.20 this behavior is fixed. The last axis of an `image` with dimensions (M, N, 3) is interpreted as a color channel if `channel_axis` is not set by the user (signaled by the default proxy value `ChannelAxisNotSet`). Starting with 0.22, `channel_axis=None` will be used as the new default value.

Returns

filtered_image

[ndarray] the filtered array

Notes

This function is a wrapper around `scipy.ndi.gaussian_filter()`.

Integer arrays are converted to float.

The `output` should be floating point data type since `gaussian` converts to float provided `image`. If `output` is not provided, another array will be allocated and returned as the result.

The multi-dimensional filter is implemented as a sequence of one-dimensional convolution filters. The intermediate arrays are stored in the same data type as the output. Therefore, for output types with a limited precision, the results may be imprecise because intermediate results may be stored with insufficient precision.

Examples

```
>>> a = np.zeros((3, 3))
>>> a[1, 1] = 1
>>> a
array([[0., 0., 0.],
       [0., 1., 0.],
       [0., 0., 0.]])
>>> gaussian(a, sigma=0.4) # mild smoothing
array([[0.00163116, 0.03712502, 0.00163116],
       [0.03712502, 0.84496158, 0.03712502],
       [0.00163116, 0.03712502, 0.00163116]])
>>> gaussian(a, sigma=1) # more smoothing
array([[0.05855018, 0.09653293, 0.05855018],
       [0.09653293, 0.15915589, 0.09653293],
       [0.05855018, 0.09653293, 0.05855018]])
>>> # Several modes are possible for handling boundaries
>>> gaussian(a, sigma=1, mode='reflect')
array([[0.08767308, 0.12075024, 0.08767308],
       [0.12075024, 0.16630671, 0.12075024],
       [0.08767308, 0.12075024, 0.08767308]])
>>> # For RGB images, each is filtered separately
>>> from skimage.data import astronaut
>>> image = astronaut()
>>> filtered_img = gaussian(image, sigma=1, channel_axis=-1)
```

- *Active Contour Model*
- *Assemble images with simple image stitching*
- *Colocalization metrics*
- *Track solidification of a metallic alloy*

- Measure fluorescence intensity at the nuclear envelope
-

```
skimage.filters.hessian(image, sigmas=range(1, 10, 2), scale_range=None, scale_step=None, alpha=0.5,  
beta=0.5, gamma=15, black_ridges=True, mode='reflect', cval=0)
```

Filter an image with the Hybrid Hessian filter.

This filter can be used to detect continuous edges, e.g. vessels, wrinkles, rivers. It can be used to calculate the fraction of the whole image containing such objects.

Defined only for 2-D and 3-D images. Almost equal to Frangi filter, but uses alternative method of smoothing. Refer to [?] to find the differences between Frangi and Hessian filters.

Parameters

image

[(N, M[, P]) ndarray] Array with input image data.

sigmas

[iterable of floats, optional] Sigmas used as scales of filter, i.e., np.arange(scale_range[0], scale_range[1], scale_step)

scale_range

[2-tuple of floats, optional] The range of sigmas used.

scale_step

[float, optional] Step size between sigmas.

beta

[float, optional] Frangi correction constant that adjusts the filter's sensitivity to deviation from a blob-like structure.

gamma

[float, optional] Frangi correction constant that adjusts the filter's sensitivity to areas of high variance/texture/structure.

black_ridges

[boolean, optional] When True (the default), the filter detects black ridges; when False, it detects white ridges.

mode

[{‘constant’, ‘reflect’, ‘wrap’, ‘nearest’, ‘mirror’}, optional] How to handle values outside the image borders.

cval

[float, optional] Used in conjunction with mode ‘constant’, the value outside the image boundaries.

Returns

out

[(N, M[, P]) ndarray] Filtered image (maximum of pixels across all scales).

See also:

[*meijering*](#)
[*sato*](#)
[*frangi*](#)

Notes

Written by Marc Schrijver (November 2001) Re-Written by D. J. Kroon University of Twente (May 2009) [?]

References

[?], [?]

- *Ridge operators*

```
skimage.filters.inverse(data, impulse_response=None, filter_params=None, max_gain=2,
                       predefined_filter=None)
```

Deprecated: `inverse` is deprecated since version 0.20 and will be removed in version 0.22. Use `skimage.filters.filter_inverse` instead.

```
skimage.filters.laplace(image, ksize=3, mask=None)
```

Find the edges of an image using the Laplace operator.

Parameters

image

[ndarray] Image to process.

ksize

[int, optional] Define the size of the discrete Laplacian operator such that it will have a size of `(ksize,) * image.ndim`.

mask

[ndarray, optional] An optional mask to limit the application to a certain area. Note that pixels surrounding masked regions are also masked to prevent masked regions from affecting the result.

Returns**output**

[ndarray] The Laplace edge map.

Notes

The Laplacian operator is generated using the function `skimage.restoration.uft.laplacian()`.

`skimage.filters.median(image, footprint=None, out=None, mode='nearest', cval=0.0, behavior='ndimage')`

Return local median of an image.

Parameters**image**

[array-like] Input image.

footprint

[ndarray, optional] If `behavior=='rank'`, `footprint` is a 2-D array of 1's and 0's. If `behavior=='ndimage'`, `footprint` is a N-D array of 1's and 0's with the same number of dimension than `image`. If None, `footprint` will be a N-D array with 3 elements for each dimension (e.g., vector, square, cube, etc.)

out

[ndarray, (same dtype as image), optional] If None, a new array is allocated.

mode

[{'reflect', 'constant', 'nearest', 'mirror', 'wrap'}, optional] The mode parameter determines how the array borders are handled, where `cval` is the value when mode is equal to 'constant'. Default is 'nearest'.

New in version 0.15: mode is used when `behavior='ndimage'`.

cval

[scalar, optional] Value to fill past edges of input if mode is 'constant'. Default is 0.0

New in version 0.15: `cval` was added in 0.15 is used when `behavior='ndimage'`.

behavior

[{'ndimage', 'rank'}, optional] Either to use the old behavior (i.e., < 0.15) or the new behavior. The old behavior will call the `skimage.filters.rank.median()`. The new behavior will call the `scipy.ndimage.median_filter()`. Default is 'ndimage'.

New in version 0.15: behavior is introduced in 0.15

Changed in version 0.16: Default behavior has been changed from 'rank' to 'ndimage'

Returns**out**

[2-D array (same dtype as input image)] Output image.

See also:

[`skimage.filters.rank.median`](#)

Rank-based implementation of the median filtering offering more flexibility with additional parameters but dedicated for unsigned integer images.

Examples

```
>>> from skimage import data
>>> from skimage.morphology import disk
>>> from skimage.filters import median
>>> img = data.camera()
>>> med = median(img, disk(5))
```

`skimage.filters.meijering(image, sigmas=range(1, 10, 2), alpha=None, black_ridges=True, mode='reflect', cval=0)`

Filter an image with the Meijering neuriteness filter.

This filter can be used to detect continuous ridges, e.g. neurites, wrinkles, rivers. It can be used to calculate the fraction of the whole image containing such objects.

Calculates the eigenvectors of the Hessian to compute the similarity of an image region to neurites, according to the method described in [?].

Parameters**image**

[(N, M[, ..., P]) ndarray] Array with input image data.

sigmas

[iterable of floats, optional] Sigmas used as scales of filter

alpha

[float, optional] Shaping filter constant, that selects maximally flat elongated features. The default, None, selects the optimal value $-1/(ndim+1)$.

black_ridges

[boolean, optional] When True (the default), the filter detects black ridges; when False, it detects white ridges.

mode

[{‘constant’, ‘reflect’, ‘wrap’, ‘nearest’, ‘mirror’}, optional] How to handle values outside the image borders.

cval

[float, optional] Used in conjunction with mode ‘constant’, the value outside the image boundaries.

Returns**out**

[(N, M[, …, P]) ndarray] Filtered image (maximum of pixels across all scales).

See also:

sato

frangi

hessian

References

[?]

- *Ridge operators*

`skimage.filters.prewitt(image, mask=None, *, axis=None, mode='reflect', cval=0.0)`

Find the edge magnitude using the Prewitt transform.

Parameters**image**

[array] The input image.

mask

[array of bool, optional] Clip the output image to this mask. (Values where mask=0 will be set to 0.)

axis

[int or sequence of int, optional] Compute the edge filter along this axis. If not provided, the edge magnitude is computed. This is defined as:

```
prw_mag = np.sqrt(sum([prewitt(image, axis=i)**2
                      for i in range(image.ndim)])) / image.ndim
```

The magnitude is also computed if axis is a sequence.

mode

[str or sequence of str, optional] The boundary mode for the convolution. See [scipy.ndimage.convolve](#) for a description of the modes. This can be either a single boundary mode or one boundary mode per axis.

eval

[float, optional] When *mode* is 'constant', this is the constant used in values outside the boundary of the image data.

Returns

output

[array of float] The Prewitt edge map.

See also:

[`prewitt_h`, `prewitt_v`](#)

horizontal and vertical edge detection.

[`sobel`, `scharr`, `farid`, `skimage.feature.canny`](#)

Notes

The edge magnitude depends slightly on edge directions, since the approximation of the gradient operator by the Prewitt operator is not completely rotation invariant. For a better rotation invariance, the Scharr operator should be used. The Sobel operator has a better rotation invariance than the Prewitt operator, but a worse rotation invariance than the Scharr operator.

Examples

```
>>> from skimage import data
>>> from skimage import filters
>>> camera = data.camera()
>>> edges = filters.prewitt(camera)
```

- *Edge operators*

`skimage.filters.prewitt_h(image, mask=None)`

Find the horizontal edges of an image using the Prewitt transform.

Parameters

image

[2-D array] Image to process.

mask

[2-D array, optional] An optional mask to limit the application to a certain area. Note that pixels surrounding masked regions are also masked to prevent masked regions from affecting the result.

Returns**output**

[2-D array] The Prewitt edge map.

Notes

We use the following kernel:

$$\begin{matrix} 1/3 & 1/3 & 1/3 \\ 0 & 0 & 0 \\ -1/3 & -1/3 & -1/3 \end{matrix}$$

- *Edge operators*

`skimage.filters.prewitt_v(image, mask=None)`

Find the vertical edges of an image using the Prewitt transform.

Parameters**image**

[2-D array] Image to process.

mask

[2-D array, optional] An optional mask to limit the application to a certain area. Note that pixels surrounding masked regions are also masked to prevent masked regions from affecting the result.

Returns**output**

[2-D array] The Prewitt edge map.

Notes

We use the following kernel:

```
1/3  0  -1/3
1/3  0  -1/3
1/3  0  -1/3
```

- *Edge operators*
-

`skimage.filters.rank_order(image)`

Return an image of the same shape where each pixel is the index of the pixel value in the ascending order of the unique values of `image`, aka the rank-order value.

Parameters

image

[ndarray]

Returns

labels

[ndarray of unsigned integers, of shape `image.shape`] New array where each pixel has the rank-order value of the corresponding pixel in `image`. Pixel values are between 0 and `n - 1`, where `n` is the number of distinct unique values in `image`. The dtype of this array will be determined by `np.min_scalar_type(image.size)`.

original_values

[1-D ndarray] Unique original values of `image`. This will have the same dtype as `image`.

Examples

```
>>> a = np.array([[1, 4, 5], [4, 4, 1], [5, 1, 1]])
>>> a
array([[1, 4, 5],
       [4, 4, 1],
       [5, 1, 1]])
>>> rank_order(a)
(array([[0, 1, 2],
       [1, 1, 0],
       [2, 0, 0]], dtype=uint8), array([1, 4, 5]))
>>> b = np.array([-1., 2.5, 3.1, 2.5])
>>> rank_order(b)
(array([0, 1, 2, 1], dtype=uint8), array([-1., 2.5, 3.1]))
```

```
skimage.filters.roberts(image, mask=None)
```

Find the edge magnitude using Roberts' cross operator.

Parameters

image

[2-D array] Image to process.

mask

[2-D array, optional] An optional mask to limit the application to a certain area. Note that pixels surrounding masked regions are also masked to prevent masked regions from affecting the result.

Returns

output

[2-D array] The Roberts' Cross edge map.

See also:

`roberts_pos_diag, roberts_neg_diag`

diagonal edge detection.

`sobel, scharr, prewitt, skimage.feature.canny`

Examples

```
>>> from skimage import data
>>> camera = data.camera()
>>> from skimage import filters
>>> edges = filters.roberts(camera)
```

- *Edge operators*
-

```
skimage.filters.roberts_neg_diag(image, mask=None)
```

Find the cross edges of an image using the Roberts' Cross operator.

The kernel is applied to the input image to produce separate measurements of the gradient component one orientation.

Parameters**image**

[2-D array] Image to process.

mask

[2-D array, optional] An optional mask to limit the application to a certain area. Note that pixels surrounding masked regions are also masked to prevent masked regions from affecting the result.

Returns**output**

[2-D array] The Robert's edge map.

Notes

We use the following kernel:

0	1
-1	0

skimage.filters.roberts_pos_diag(image, mask=None)

Find the cross edges of an image using Roberts' cross operator.

The kernel is applied to the input image to produce separate measurements of the gradient component one orientation.

Parameters**image**

[2-D array] Image to process.

mask

[2-D array, optional] An optional mask to limit the application to a certain area. Note that pixels surrounding masked regions are also masked to prevent masked regions from affecting the result.

Returns**output**

[2-D array] The Robert's edge map.

Notes

We use the following kernel:

1	0
0	-1

`skimage.filters.sato(image, sigmas=range(1, 10, 2), black_ridges=True, mode='reflect', cval=0)`

Filter an image with the Sato tubeness filter.

This filter can be used to detect continuous ridges, e.g. tubes, wrinkles, rivers. It can be used to calculate the fraction of the whole image containing such objects.

Defined only for 2-D and 3-D images. Calculates the eigenvectors of the Hessian to compute the similarity of an image region to tubes, according to the method described in [?].

Parameters

image

[$(N, M[, P])$ ndarray] Array with input image data.

sigmas

[iterable of floats, optional] Sigmas used as scales of filter.

black_ridges

[boolean, optional] When True (the default), the filter detects black ridges; when False, it detects white ridges.

mode

[{'constant', 'reflect', 'wrap', 'nearest', 'mirror'}, optional] How to handle values outside the image borders.

cval

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

Returns

out

[$(N, M[, P])$ ndarray] Filtered image (maximum of pixels across all scales).

See also:

`meijering`
`frangi`
`hessian`

References

[?]

- *Ridge operators*
 - *Use pixel graphs to find an object's geodesic center*
-

`skimage.filters.scharr(image, mask=None, *, axis=None, mode='reflect', cval=0.0)`

Find the edge magnitude using the Scharr transform.

Parameters

image

[array] The input image.

mask

[array of bool, optional] Clip the output image to this mask. (Values where mask=0 will be set to 0.)

axis

[int or sequence of int, optional] Compute the edge filter along this axis. If not provided, the edge magnitude is computed. This is defined as:

```
sch_mag = np.sqrt(sum([scharr(image, axis=i)**2  
                      for i in range(image.ndim)])) / image.ndim
```

The magnitude is also computed if axis is a sequence.

mode

[str or sequence of str, optional] The boundary mode for the convolution. See `scipy.ndimage.convolve` for a description of the modes. This can be either a single boundary mode or one boundary mode per axis.

cval

[float, optional] When `mode` is 'constant', this is the constant used in values outside the boundary of the image data.

Returns

output

[array of float] The Scharr edge map.

See also:

scharr_h, scharr_v

horizontal and vertical edge detection.

sobel, prewitt, farid, skimage.feature.canny

Notes

The Scharr operator has a better rotation invariance than other edge filters such as the Sobel or the Prewitt operators.

References

[?], [?]

Examples

```
>>> from skimage import data
>>> from skimage import filters
>>> camera = data.camera()
>>> edges = filters.scharr(camera)
```

- *Edge operators*

`skimage.filters.scharr_h(image, mask=None)`

Find the horizontal edges of an image using the Scharr transform.

Parameters**image**

[2-D array] Image to process.

mask

[2-D array, optional] An optional mask to limit the application to a certain area. Note that pixels surrounding masked regions are also masked to prevent masked regions from affecting the result.

Returns**output**

[2-D array] The Scharr edge map.

Notes

We use the following kernel:

3	10	3
0	0	0
-3	-10	-3

References

[?]

- *Edge operators*

skimage.filters.scharr_v(image, mask=None)

Find the vertical edges of an image using the Scharr transform.

Parameters

image

[2-D array] Image to process

mask

[2-D array, optional] An optional mask to limit the application to a certain area. Note that pixels surrounding masked regions are also masked to prevent masked regions from affecting the result.

Returns

output

[2-D array] The Scharr edge map.

Notes

We use the following kernel:

3	0	-3
10	0	-10
3	0	-3

References

[?]

- Edge operators

`skimage.filters.sobel(image, mask=None, *, axis=None, mode='reflect', cval=0.0)`

Find edges in an image using the Sobel filter.

Parameters

image

[array] The input image.

mask

[array of bool, optional] Clip the output image to this mask. (Values where mask=0 will be set to 0.)

axis

[int or sequence of int, optional] Compute the edge filter along this axis. If not provided, the edge magnitude is computed. This is defined as:

```
sobel_mag = np.sqrt(sum([sobel(image, axis=i)**2
                        for i in range(image.ndim)])) / image.ndim
```

The magnitude is also computed if axis is a sequence.

mode

[str or sequence of str, optional] The boundary mode for the convolution. See `scipy.ndimage.convolve` for a description of the modes. This can be either a single boundary mode or one boundary mode per axis.

cval

[float, optional] When `mode` is 'constant', this is the constant used in values outside the boundary of the image data.

Returns

output

[array of float] The Sobel edge map.

See also:

`sobel_h`, `sobel_v`

horizontal and vertical edge detection.

`scharr`, `prewitt`, `farid`, `skimage.feature.canny`

References

[?], [?]

Examples

```
>>> from skimage import data
>>> from skimage import filters
>>> camera = data.camera()
>>> edges = filters.sobel(camera)
```

- *Adapting gray-scale filters to RGB images*
- *Edge operators*
- *Hysteresis thresholding*
- *Region Boundary based RAGs*
- *Find Regular Segments Using Compact Watershed*
- *Expand segmentation labels without overlap*
- *Comparison of segmentation and superpixel algorithms*
- *Find the intersection of two segmentations*
- *Hierarchical Merging of Region Boundary RAGs*
- *Flood Fill*
- *Evaluating segmentation metrics*
- *Comparing edge-based and region-based segmentation*

`skimage.filters.sobel_h(image, mask=None)`

Find the horizontal edges of an image using the Sobel transform.

Parameters

`image`

[2-D array] Image to process.

mask

[2-D array, optional] An optional mask to limit the application to a certain area. Note that pixels surrounding masked regions are also masked to prevent masked regions from affecting the result.

Returns**output**

[2-D array] The Sobel edge map.

Notes

We use the following kernel:

1	2	1
0	0	0
-1	-2	-1

- *Edge operators*

skimage.filters.sobel_v(image, mask=None)

Find the vertical edges of an image using the Sobel transform.

Parameters**image**

[2-D array] Image to process.

mask

[2-D array, optional] An optional mask to limit the application to a certain area. Note that pixels surrounding masked regions are also masked to prevent masked regions from affecting the result.

Returns**output**

[2-D array] The Sobel edge map.

Notes

We use the following kernel:

```
1  0  -1
2  0  -2
1  0  -1
```

- *Edge operators*
-

`skimage.filters.threshold_isodata(image=None, nbins=256, return_all=False, *, hist=None)`

Return threshold value(s) based on ISODATA method.

Histogram-based threshold, known as Ridler-Calvard method or inter-means. Threshold values returned satisfy the following equality:

```
threshold = (image[image <= threshold].mean() +
              image[image > threshold].mean()) / 2.0
```

That is, returned thresholds are intensities that separate the image into two groups of pixels, where the threshold intensity is midway between the mean intensities of these groups.

For integer images, the above equality holds to within one; for floating-point images, the equality holds to within the histogram bin-width.

Either image or hist must be provided. In case hist is given, the actual histogram of the image is ignored.

Parameters

image

[(N, M, \dots, P)] ndarray] Grayscale input image.

nbins

[int, optional] Number of bins used to calculate histogram. This value is ignored for integer arrays.

return_all

[bool, optional] If False (default), return only the lowest threshold that satisfies the above equality. If True, return all valid thresholds.

hist

[array, or 2-tuple of arrays, optional] Histogram to determine the threshold from and a corresponding array of bin center intensities. Alternatively, only the histogram can be passed.

Returns

threshold

[float or int or array] Threshold value(s).

References

[?], [?], [?]

Examples

```
>>> from skimage.data import coins
>>> image = coins()
>>> thresh = threshold_isodata(image)
>>> binary = image > thresh
```

`skimage.filters.threshold_li(image, *, tolerance=None, initial_guess=None, iter_callback=None)`

Compute threshold value by Li's iterative Minimum Cross Entropy method.

Parameters**image**

[(N, M[, ..., P]) ndarray] Grayscale input image.

tolerance

[float, optional] Finish the computation when the change in the threshold in an iteration is less than this value. By default, this is half the smallest difference between intensity values in `image`.

initial_guess

[float or Callable[[array[float]], float], optional] Li's iterative method uses gradient descent to find the optimal threshold. If the image intensity histogram contains more than two modes (peaks), the gradient descent could get stuck in a local optimum. An initial guess for the iteration can help the algorithm find the globally-optimal threshold. A float value defines a specific start point, while a callable should take in an array of image intensities and return a float value. Example valid callables include `numpy.mean` (default), `lambda arr: numpy.quantile(arr, 0.95)`, or even `skimage.filters.threshold_otsu()`.

iter_callback

[Callable[[float], Any], optional] A function that will be called on the threshold at every iteration of the algorithm.

Returns

threshold

[float] Upper threshold value. All pixels with an intensity higher than this value are assumed to be foreground.

References

[?], [?], [?], [?]

Examples

```
>>> from skimage.data import camera
>>> image = camera()
>>> thresh = threshold_li(image)
>>> binary = image > thresh
```

- *Li thresholding*
-

`skimage.filters.threshold_local(image, block_size=3, method='gaussian', offset=0, mode='reflect', param=None, cval=0)`

Compute a threshold mask image based on local pixel neighborhood.

Also known as adaptive or dynamic thresholding. The threshold value is the weighted mean for the local neighborhood of a pixel subtracted by a constant. Alternatively the threshold can be determined dynamically by a given function, using the ‘generic’ method.

Parameters**image**

[(N, M[, ..., P]) ndarray] Grayscale input image.

block_size

[int or sequence of int] Odd size of pixel neighborhood which is used to calculate the threshold value (e.g. 3, 5, 7, ..., 21, ...).

method

[{‘generic’, ‘gaussian’, ‘mean’, ‘median’}, optional] Method used to determine adaptive threshold for local neighborhood in weighted mean image.

- ‘generic’: use custom function (see `param` parameter)
- ‘gaussian’: apply gaussian filter (see `param` parameter for custom sigma value)
- ‘mean’: apply arithmetic mean filter
- ‘median’: apply median rank filter

By default the ‘gaussian’ method is used.

offset

[float, optional] Constant subtracted from weighted mean of neighborhood to calculate the local threshold value. Default offset is 0.

mode

[{‘reflect’, ‘constant’, ‘nearest’, ‘mirror’, ‘wrap’}, optional] The mode parameter determines how the array borders are handled, where `cval` is the value when mode is equal to ‘constant’. Default is ‘reflect’.

param

[{int, function}, optional] Either specify sigma for ‘gaussian’ method or function object for ‘generic’ method. This functions takes the flat array of local neighborhood as a single argument and returns the calculated threshold for the centre pixel.

cval

[float, optional] Value to fill past edges of input if mode is ‘constant’.

Returns**threshold**

[(N, M[, ..., P]) ndarray] Threshold image. All pixels in the input image higher than the corresponding pixel in the threshold image are considered foreground.

References

[?]

Examples

```
>>> from skimage.data import camera
>>> image = camera()[:50, :50]
>>> binary_image1 = image > threshold_local(image, 15, 'mean')
>>> func = lambda arr: arr.mean()
>>> binary_image2 = image > threshold_local(image, 15, 'generic',
...                                         param=func)
```

- *Thresholding*

```
skimage.filters.threshold_mean(image)
```

Return threshold value based on the mean of grayscale values.

Parameters

image

[(N, M[, ..., P]) ndarray] Grayscale input image.

Returns

threshold

[float] Upper threshold value. All pixels with an intensity higher than this value are assumed to be foreground.

References

[?]

Examples

```
>>> from skimage.data import camera
>>> image = camera()
>>> thresh = threshold_mean(image)
>>> binary = image > thresh
```

- *Thresholding*

`skimage.filters.threshold_minimum(image=None, nbins=256, max_num_iter=10000, *, hist=None)`

Return threshold value based on minimum method.

The histogram of the input `image` is computed if not provided and smoothed until there are only two maxima. Then the minimum in between is the threshold value.

Either `image` or `hist` must be provided. In case `hist` is given, the actual histogram of the image is ignored.

Parameters

image

[(N, M[, ..., P]) ndarray, optional] Grayscale input image.

nbins

[int, optional] Number of bins used to calculate histogram. This value is ignored for integer arrays.

max_num_iter

[int, optional] Maximum number of iterations to smooth the histogram.

hist

[array, or 2-tuple of arrays, optional] Histogram to determine the threshold from and a corresponding array of bin center intensities. Alternatively, only the histogram can be passed.

Returns**threshold**

[float] Upper threshold value. All pixels with an intensity higher than this value are assumed to be foreground.

Raises**RuntimeError**

If unable to find two local maxima in the histogram or if the smoothing takes more than 1e4 iterations.

References

[?], [?]

Examples

```
>>> from skimage.data import camera
>>> image = camera()
>>> thresh = threshold_minimum(image)
>>> binary = image > thresh
```

- *Thresholding*
- *Track solidification of a metallic alloy*

```
skimage.filters.threshold_multithreshold(image=None, classes=3, nbins=256, *, hist=None)
```

Generate *classes*-1 threshold values to divide gray levels in *image*, following Otsu's method for multiple classes.

The threshold values are chosen to maximize the total sum of pairwise variances between the thresholded graylevel classes. See Notes and [?] for more details.

Either image or hist must be provided. If hist is provided, the actual histogram of the image is ignored.

Parameters

image

[$(N, M[, \dots, P])$ ndarray, optional] Grayscale input image.

classes

[int, optional] Number of classes to be thresholded, i.e. the number of resulting regions.

nbins

[int, optional] Number of bins used to calculate the histogram. This value is ignored for integer arrays.

hist

[array, or 2-tuple of arrays, optional] Histogram from which to determine the threshold, and optionally a corresponding array of bin center intensities. If no hist provided, this function will compute it from the image (see notes).

Returns

thresh

[array] Array containing the threshold values for the desired classes.

Raises

ValueError

If `image` contains less grayscale value than the desired number of classes.

Notes

This implementation relies on a Cython function whose complexity is $O\left(\frac{Ch^{C-1}}{(C-1)!}\right)$, where h is the number of histogram bins and C is the number of classes desired.

If no hist is given, this function will make use of `skimage.exposure.histogram`, which behaves differently than `np.histogram`. While both allowed, use the former for consistent behaviour.

The input image must be grayscale.

References

[?], [?]

Examples

```
>>> from skimage.color import label2rgb
>>> from skimage import data
>>> image = data.camera()
>>> thresholds = threshold_multiotsu(image)
>>> regions = np.digitize(image, bins=thresholds)
>>> regions_colorized = label2rgb(regions)
```

- *Multi-Otsu Thresholding*
- *Use pixel graphs to find an object's geodesic center*
- *Segment human cells (in mitosis)*

`skimage.filters.threshold_niblack(image, window_size=15, k=0.2)`

Applies Niblack local threshold to an array.

A threshold T is calculated for every pixel in the image using the following formula:

$$T = m(x,y) - k * s(x,y)$$

where $m(x,y)$ and $s(x,y)$ are the mean and standard deviation of pixel (x,y) neighborhood defined by a rectangular window with size w times w centered around the pixel. k is a configurable parameter that weights the effect of standard deviation.

Parameters

image

[(N, M, \dots, P) ndarray] Grayscale input image.

window_size

[int, or iterable of int, optional] Window size specified as a single odd integer (3, 5, 7, ...), or an iterable of length `image.ndim` containing only odd integers (e.g. (1, 5, 5)).

k

[float, optional] Value of parameter k in threshold formula.

Returns

threshold

[(N, M) ndarray] Threshold mask. All pixels with an intensity higher than this value are assumed to be foreground.

Notes

This algorithm is originally designed for text recognition.

The Bradley threshold is a particular case of the Niblack one, being equivalent to

```
>>> from skimage import data
>>> image = data.page()
>>> q = 1
>>> threshold_image = threshold_niblack(image, k=0) * q
```

for some value q. By default, Bradley and Roth use q=1.

References

[?], [?]

Examples

```
>>> from skimage import data
>>> image = data.page()
>>> threshold_image = threshold_niblack(image, window_size=7, k=0.1)
```

- *Niblack and Sauvola Thresholding*
-

`skimage.filters.threshold_otsu(image=None, nbins=256, *, hist=None)`

Return threshold value based on Otsu's method.

Either image or hist must be provided. If hist is provided, the actual histogram of the image is ignored.

Parameters

image

[$(N, M[\dots, P])$ ndarray, optional] Grayscale input image.

nbins

[int, optional] Number of bins used to calculate histogram. This value is ignored for integer arrays.

hist

[array, or 2-tuple of arrays, optional] Histogram from which to determine the threshold, and optionally a corresponding array of bin center intensities. If no hist provided, this function will compute it from the image.

Returns

threshold

[float] Upper threshold value. All pixels with an intensity higher than this value are assumed to be foreground.

Notes

The input image must be grayscale.

References

[?]

Examples

```
>>> from skimage.data import camera
>>> image = camera()
>>> thresh = threshold_otsu(image)
>>> binary = image <= thresh
```

- *Thresholding*
- *Niblack and Sauvola Thresholding*
- *Label image regions*
- *Measure region properties*
- *Colocalization metrics*
- *Thresholding*
- *Measure fluorescence intensity at the nuclear envelope*
- *Rank filters*

`skimage.filters.threshold_sauvola(image, window_size=15, k=0.2, r=None)`

Applies Sauvola local threshold to an array. Sauvola is a modification of Niblack technique.

In the original method a threshold T is calculated for every pixel in the image using the following formula:

$$T = m(x,y) * (1 + k * ((s(x,y) / R) - 1))$$

where $m(x,y)$ and $s(x,y)$ are the mean and standard deviation of pixel (x,y) neighborhood defined by a rectangular window with size w times w centered around the pixel. k is a configurable parameter that weights the effect of standard deviation. R is the maximum standard deviation of a grayscale image.

Parameters

image

[(N, M[, ..., P]) ndarray] Grayscale input image.

window_size

[int, or iterable of int, optional] Window size specified as a single odd integer (3, 5, 7, ...), or an iterable of length `image.ndim` containing only odd integers (e.g. (1, 5, 5)).

k

[float, optional] Value of the positive parameter k.

r

[float, optional] Value of R, the dynamic range of standard deviation. If None, set to the half of the image dtype range.

Returns

threshold

[(N, M) ndarray] Threshold mask. All pixels with an intensity higher than this value are assumed to be foreground.

Notes

This algorithm is originally designed for text recognition.

References

[?]

Examples

```
>>> from skimage import data
>>> image = data.page()
>>> t_sauvola = threshold_sauvola(image, window_size=15, k=0.2)
>>> binary_image = image > t_sauvola
```

- Niblack and Sauvola Thresholding

`skimage.filters.threshold_triangle(image, nbins=256)`

Return threshold value based on the triangle algorithm.

Parameters

image

[(N, M[, ..., P]) ndarray] Grayscale input image.

nbins

[int, optional] Number of bins used to calculate histogram. This value is ignored for integer arrays.

Returns

threshold

[float] Upper threshold value. All pixels with an intensity higher than this value are assumed to be foreground.

References

[?], [?]

Examples

```
>>> from skimage.data import camera
>>> image = camera()
>>> thresh = threshold_triangle(image)
>>> binary = image > thresh
```

skimage.filters.threshold_yen(image=None, nbins=256, *, hist=None)

Return threshold value based on Yen's method. Either image or hist must be provided. In case hist is given, the actual histogram of the image is ignored.

Parameters

image

[(N, M[, ..., P]) ndarray] Grayscale input image.

nbins

[int, optional] Number of bins used to calculate histogram. This value is ignored for integer arrays.

hist

[array, or 2-tuple of arrays, optional] Histogram from which to determine the threshold, and

optionally a corresponding array of bin center intensities. An alternative use of this function is to pass it only hist.

Returns

threshold

[float] Upper threshold value. All pixels with an intensity higher than this value are assumed to be foreground.

References

[?], [?], [?]

Examples

```
>>> from skimage.data import camera
>>> image = camera()
>>> thresh = threshold_yen(image)
>>> binary = image <= thresh
```

skimage.filters.try_all_threshold(*image*, *figsize*=(8, 5), *verbose*=True)

Returns a figure comparing the outputs of different thresholding methods.

Parameters

image

[(N, M) ndarray] Input image.

figsize

[tuple, optional] Figure size (in inches).

verbose

[bool, optional] Print function name for each method.

Returns

fig, ax

[tuple] Matplotlib figure and axes.

Notes

The following algorithms are used:

- isodata
- li
- mean
- minimum
- otsu
- triangle
- yen

Examples

```
>>> from skimage.data import text
>>> fig, ax = try_all_threshold(text(), figsize=(10, 6), verbose=False)
```

- *Thresholding*
- *Thresholding*

```
skimage.filters.unsharp_mask(image, radius=1.0, amount=1.0, preserve_range=False, *,  
                             channel_axis=None)
```

Unsharp masking filter.

The sharp details are identified as the difference between the original image and its blurred version. These details are then scaled, and added back to the original image.

Parameters

image

[[P, ...,]M[, N][, C] ndarray] Input image.

radius

[scalar or sequence of scalars, optional] If a scalar is given, then its value is used for all dimensions. If sequence is given, then there must be exactly one radius for each dimension except the last dimension for multichannel images. Note that 0 radius means no blurring, and negative values are not allowed.

amount

[scalar, optional] The details will be amplified with this factor. The factor could be 0 or negative. Typically, it is a small positive number, e.g. 1.0.

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`. Also see https://scikit-image.org/docs/dev/user_guide/data_types.html

channel_axis

[int or None, optional] If None, the image is assumed to be a grayscale (single channel) image. Otherwise, this parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns**output**

[[P, ...,]M[, N][, C] ndarray of float] Image with unsharp mask applied.

Notes

Unsharp masking is an image sharpening technique. It is a linear image operation, and numerically stable, unlike deconvolution which is an ill-posed problem. Because of this stability, it is often preferred over deconvolution.

The main idea is as follows: sharp details are identified as the difference between the original image and its blurred version. These details are added back to the original image after a scaling step:

$$\text{enhanced image} = \text{original} + \text{amount} * (\text{original} - \text{blurred})$$

When applying this filter to several color layers independently, color bleeding may occur. More visually pleasing result can be achieved by processing only the brightness/lightness/intensity channel in a suitable color space such as HSV, HSL, YUV, or YCbCr.

Unsharp masking is described in most introductory digital image processing books. This implementation is based on [?].

References

[?], [?]

Examples

```
>>> array = np.ones(shape=(5, 5), dtype=np.uint8)*100
>>> array[2, 2] = 120
>>> array
array([[100, 100, 100, 100, 100],
       [100, 100, 100, 100, 100],
       [100, 100, 120, 100, 100],
       [100, 100, 100, 100, 100],
       [100, 100, 100, 100, 100]], dtype=uint8)
>>> np.around(unsharp_mask(array, radius=0.5, amount=2), 2)
array([[0.39, 0.39, 0.39, 0.39, 0.39],
       [0.39, 0.39, 0.38, 0.39, 0.39],
```

(continues on next page)

(continued from previous page)

```
[0.39, 0.38, 0.53, 0.38, 0.39],
[0.39, 0.39, 0.38, 0.39, 0.39],
[0.39, 0.39, 0.39, 0.39, 0.39]])
```

```
>>> array = np.ones(shape=(5,5), dtype=np.int8)*100
>>> array[2,2] = 127
>>> np.around(unsharp_mask(array, radius=0.5, amount=2),2)
array([[0.79, 0.79, 0.79, 0.79, 0.79],
       [0.79, 0.78, 0.75, 0.78, 0.79],
       [0.79, 0.75, 1. , 0.75, 0.79],
       [0.79, 0.78, 0.75, 0.78, 0.79],
       [0.79, 0.79, 0.79, 0.79, 0.79]])
```

```
>>> np.around(unsharp_mask(array, radius=0.5, amount=2, preserve_range=True), 2)
array([[100. , 100. , 99.99, 100. , 100. ],
       [100. , 99.39, 95.48, 99.39, 100. ],
       [99.99, 95.48, 147.59, 95.48, 99.99],
       [100. , 99.39, 95.48, 99.39, 100. ],
       [100. , 100. , 99.99, 100. , 100. ]])
```

- *Unsharp masking*

`skimage.filters.wiener(data, impulse_response=None, filter_params=None, K=0.25, predefined_filter=None)`

Minimum Mean Square Error (Wiener) inverse filter.

Parameters

data

[(M,N) ndarray] Input data.

K

[float or (M,N) ndarray] Ratio between power spectrum of noise and undegraded image.

impulse_response

[callable $f(r, c, \text{**filter_params})$] Impulse response of the filter. See `LPIFilter2D.__init__`.

filter_params

[dict, optional] Additional keyword parameters to the `impulse_response` function.

Other Parameters

predefined_filter

[LPIFilter2D] If you need to apply the same filter multiple times over different images, construct the LPIFilter2D and specify it here.

`skimage.filters.window(window_type, shape, warp_kwargs=None)`

Return an n-dimensional window of a given size and dimensionality.

Parameters

`window_type`

[string, float, or tuple] The type of window to be created. Any window type supported by `scipy.signal.get_window` is allowed here. See notes below for a current list, or the SciPy documentation for the version of SciPy on your machine.

`shape`

[tuple of int or int] The shape of the window along each axis. If an integer is provided, a 1D window is generated.

`warp_kwargs`

[dict] Keyword arguments passed to `skimage.transform.warp` (e.g., `warp_kwargs={'order':3}` to change interpolation method).

Returns

`nd_window`

[ndarray] A window of the specified shape. `dtype` is `np.float64`.

Notes

This function is based on `scipy.signal.get_window` and thus can access all of the window types available to that function (e.g., "hann", "boxcar"). Note that certain window types require parameters that have to be supplied with the window name as a tuple (e.g., ("tukey", 0.8)). If only a float is supplied, it is interpreted as the beta parameter of the Kaiser window.

See https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.windows.get_window.html for more details.

Note that this function generates a double precision array of the specified shape and can thus generate very large arrays that consume a large amount of available memory.

The approach taken here to create nD windows is to first calculate the Euclidean distance from the center of the intended nD window to each position in the array. That distance is used to sample, with interpolation, from a 1D window returned from `scipy.signal.get_window`. The method of interpolation can be changed with the `order` keyword argument passed to `skimage.transform.warp`.

Some coordinates in the output window will be outside of the original signal; these will be filled in with zeros.

Window types: - boxcar - triang - blackman - hamming - hann - bartlett - flattop - parzen - bohman - blackman-harris - nuttall - barthann - kaiser (needs beta) - gaussian (needs standard deviation) - general_gaussian (needs

power, width) - slepian (needs width) - dpss (needs normalized half-bandwidth) - chebwin (needs attenuation) - exponential (needs decay scale) - tukey (needs taper fraction)

References

[?]

Examples

Return a Hann window with shape (512, 512):

```
>>> from skimage.filters import window
>>> w = window('hann', (512, 512))
```

Return a Kaiser window with beta parameter of 16 and shape (256, 256, 35):

```
>>> w = window(16, (256, 256, 35))
```

Return a Tukey window with an alpha parameter of 0.8 and shape (100, 300):

```
>>> w = window('tukey', 0.8, (100, 300))
```

- *Using Polar and Log-Polar Transformations for Registration*
- *Using window functions with images*
- *Band-pass filtering by Difference of Gaussians*

`class skimage.filters.LPIFilter2D(impulse_response, **filter_params)`

Bases: `object`

Linear Position-Invariant Filter (2-dimensional)

`__init__(impulse_response, **filter_params)`

Parameters

impulse_response

[callable $f(r, c, \text{**filter_params})$] Function that yields the impulse response. r and c are 1-dimensional vectors that represent row and column positions, in other words coordinates are $(r[0], c[0]), (r[1], c[1])$ etc. **filter_params are passed through.

In other words, `impulse_response` would be called like this:

```
>>> def impulse_response(r, c, **filter_params):
...     pass
>>>
>>> r = [0, 0, 0, 1, 1, 1, 2, 2, 2]
```

(continues on next page)

(continued from previous page)

```
>>> c = [0, 1, 2, 0, 1, 2, 0, 1, 2]
>>> filter_params = {'kw1': 1, 'kw2': 2, 'kw3': 3}
>>> impulse_response(r, c, **filter_params)
```

Examples

Gaussian filter without normalization of coefficients:

```
>>> def filt_func(r, c, sigma=1):
...     return np.exp(-(r**2 + c**2)/(2 * sigma**2))
>>> filter = LPIFilter2D(filt_func)
```

1.3.8 skimage.filters.rank

<code>skimage.filters.rank.autolevel</code>	Auto-level image using local histogram.
<code>skimage.filters.rank.autolevel_percentile</code>	Return grayscale local autolevel of an image.
<code>skimage.filters.rank.enhance_contrast</code>	Enhance contrast of an image.
<code>skimage.filters.rank.</code>	Enhance contrast of an image.
<code>enhance_contrast_percentile</code>	
<code>skimage.filters.rank.entropy</code>	Local entropy.
<code>skimage.filters.rank.equalize</code>	Equalize image using local histogram.
<code>skimage.filters.rank.geometric_mean</code>	Return local geometric mean of an image.
<code>skimage.filters.rank.gradient</code>	Return local gradient of an image (i.e.
<code>skimage.filters.rank.gradient_percentile</code>	Return local gradient of an image (i.e.
<code>skimage.filters.rank.majority</code>	Assign to each pixel the most common value within its neighborhood.
<code>skimage.filters.rank.maximum</code>	Return local maximum of an image.
<code>skimage.filters.rank.mean</code>	Return local mean of an image.
<code>skimage.filters.rank.mean_bilateral</code>	Apply a flat kernel bilateral filter.
<code>skimage.filters.rank.mean_percentile</code>	Return local mean of an image.
<code>skimage.filters.rank.median</code>	Return local median of an image.
<code>skimage.filters.rank.minimum</code>	Return local minimum of an image.
<code>skimage.filters.rank.modal</code>	Return local mode of an image.
<code>skimage.filters.rank.noise_filter</code>	Noise feature.
<code>skimage.filters.rank.otsu</code>	Local Otsu's threshold value for each pixel.
<code>skimage.filters.rank.percentile</code>	Return local percentile of an image.
<code>skimage.filters.rank.pop</code>	Return the local number (population) of pixels.
<code>skimage.filters.rank.pop_bilateral</code>	Return the local number (population) of pixels.
<code>skimage.filters.rank.pop_percentile</code>	Return the local number (population) of pixels.
<code>skimage.filters.rank.subtract_mean</code>	Return image subtracted from its local mean.
<code>skimage.filters.rank.</code>	Return image subtracted from its local mean.
<code>subtract_mean_percentile</code>	
<code>skimage.filters.rank.sum</code>	Return the local sum of pixels.
<code>skimage.filters.rank.sum_bilateral</code>	Apply a flat kernel bilateral filter.
<code>skimage.filters.rank.sum_percentile</code>	Return the local sum of pixels.
<code>skimage.filters.rank.threshold</code>	Local threshold of an image.
<code>skimage.filters.rank.threshold_percentile</code>	Local threshold of an image.
<code>skimage.filters.rank.windowed_histogram</code>	Normalized sliding window histogram

```
skimage.filters.rank.autolevel(image, footprint, out=None, mask=None, shift_x=False, shift_y=False, shift_z=False)
```

Auto-level image using local histogram.

This filter locally stretches the histogram of gray values to cover the entire range of values from “white” to “black”.

Parameters

image

[([P,] M, N) ndarray (uint8, uint16)] Input image.

footprint

[ndarray] The neighborhood expressed as an ndarray of 1’s and 0’s.

out

[([P,] M, N) array (same dtype as input)] If None, a new array is allocated.

mask

[ndarray (integer or float), optional] Mask array that defines (>0) area of the image included in the local neighborhood. If None, the complete image is used (default).

shift_x, shift_y, shift_z

[int] Offset added to the footprint center point. Shift is bounded to the footprint sizes (center must be inside the given footprint).

Returns

out

[([P,] M, N) ndarray (same dtype as input image)] Output image.

Examples

```
>>> from skimage import data
>>> from skimage.morphology import disk, ball
>>> from skimage.filters.rank import autolevel
>>> import numpy as np
>>> img = data.camera()
>>> rng = np.random.default_rng()
>>> volume = rng.integers(0, 255, size=(10,10,10), dtype=np.uint8)
>>> auto = autolevel(img, disk(5))
>>> auto_vol = autolevel(volume, ball(5))
```

- *Rank filters*

```
skimage.filters.rank.autolevel_percentile(image, footprint, out=None, mask=None, shift_x=False,  
shift_y=False, p0=0, p1=1)
```

Return grayscale local autolevel of an image.

This filter locally stretches the histogram of grayvalues to cover the entire range of values from “white” to “black”.

Only grayvalues between percentiles [p0, p1] are considered in the filter.

Parameters

image

[2-D array (uint8, uint16)] Input image.

footprint

[2-D array] The neighborhood expressed as a 2-D array of 1’s and 0’s.

out

[2-D array (same dtype as input)] If None, a new array is allocated.

mask

[ndarray] Mask array that defines (>0) area of the image included in the local neighborhood.
If None, the complete image is used (default).

shift_x, shift_y

[int] Offset added to the footprint center point. Shift is bounded to the footprint sizes (center must be inside the given footprint).

p0, p1

[float in [0, ..., 1]] Define the [p0, p1] percentile interval to be considered for computing the value.

Returns

out

[2-D array (same dtype as input image)] Output image.

- *Rank filters*
-

```
skimage.filters.rank.enhance_contrast(image, footprint, out=None, mask=None, shift_x=False,  
shift_y=False, shift_z=False)
```

Enhance contrast of an image.

This replaces each pixel by the local maximum if the pixel gray value is closer to the local maximum than the local minimum. Otherwise it is replaced by the local minimum.

Parameters**image**

[([P,] M, N) ndarray (uint8, uint16)] Input image.

footprint

[ndarray] The neighborhood expressed as an ndarray of 1's and 0's.

out

[([P,] M, N) array (same dtype as input)] If None, a new array is allocated.

mask

[ndarray (integer or float), optional] Mask array that defines (>0) area of the image included in the local neighborhood. If None, the complete image is used (default).

shift_x, shift_y, shift_z

[int] Offset added to the footprint center point. Shift is bounded to the footprint sizes (center must be inside the given footprint).

Returns**out**

[([P,] M, N) ndarray (same dtype as input image)] Output image

Examples

```
>>> from skimage import data
>>> from skimage.morphology import disk, ball
>>> from skimage.filters.rank import enhance_contrast
>>> import numpy as np
>>> img = data.camera()
>>> rng = np.random.default_rng()
>>> volume = rng.integers(0, 255, size=(10,10,10), dtype=np.uint8)
>>> out = enhance_contrast(img, disk(5))
>>> out_vol = enhance_contrast(volume, ball(5))
```

- *Rank filters*

`skimage.filters.rank.enhance_contrast_percentile(image, footprint, out=None, mask=None, shift_x=False, shift_y=False, p0=0, p1=1)`

Enhance contrast of an image.

This replaces each pixel by the local maximum if the pixel grayvalue is closer to the local maximum than the local minimum. Otherwise it is replaced by the local minimum.

Only grayvalues between percentiles [p0, p1] are considered in the filter.

Parameters

image

[2-D array (uint8, uint16)] Input image.

footprint

[2-D array] The neighborhood expressed as a 2-D array of 1's and 0's.

out

[2-D array (same dtype as input)] If None, a new array is allocated.

mask

[ndarray] Mask array that defines (>0) area of the image included in the local neighborhood.
If None, the complete image is used (default).

shift_x, shift_y

[int] Offset added to the footprint center point. Shift is bounded to the footprint sizes (center must be inside the given footprint).

p0, p1

[float in [0, ..., 1]] Define the [p0, p1] percentile interval to be considered for computing the value.

Returns

out

[2-D array (same dtype as input image)] Output image.

- *Rank filters*
-

```
skimage.filters.rank.entropy(image, footprint, out=None, mask=None, shift_x=False, shift_y=False,  
shift_z=False)
```

Local entropy.

The entropy is computed using base 2 logarithm i.e. the filter returns the minimum number of bits needed to encode the local gray level distribution.

Parameters

image

[([P,] M, N) ndarray (uint8, uint16)] Input image.

footprint

[ndarray] The neighborhood expressed as an ndarray of 1's and 0's.

out

[([P,] M, N) array (same dtype as input)] If None, a new array is allocated.

mask

[ndarray (integer or float), optional] Mask array that defines (>0) area of the image included in the local neighborhood. If None, the complete image is used (default).

shift_x, shift_y, shift_z

[int] Offset added to the footprint center point. Shift is bounded to the footprint sizes (center must be inside the given footprint).

Returns**out**

[([P,] M, N) ndarray (float)] Output image.

References

[?]

Examples

```
>>> from skimage import data
>>> from skimage.filters.rank import entropy
>>> from skimage.morphology import disk, ball
>>> import numpy as np
>>> img = data.camera()
>>> rng = np.random.default_rng()
>>> volume = rng.integers(0, 255, size=(10,10,10), dtype=np.uint8)
>>> ent = entropy(img, disk(5))
>>> ent_vol = entropy(volume, ball(5))
```

- *Tinting gray-scale images*
- *Entropy*
- *Rank filters*

`skimage.filters.rank.equalize(image, footprint, out=None, mask=None, shift_x=False, shift_y=False, shift_z=False)`

Equalize image using local histogram.

Parameters

image

[([P,] M, N) ndarray (uint8, uint16)] Input image.

footprint

[ndarray] The neighborhood expressed as an ndarray of 1's and 0's.

out

[([P,] M, N) array (same dtype as input)] If None, a new array is allocated.

mask

[ndarray (integer or float), optional] Mask array that defines (>0) area of the image included in the local neighborhood. If None, the complete image is used (default).

shift_x, shift_y, shift_z

[int] Offset added to the footprint center point. Shift is bounded to the footprint sizes (center must be inside the given footprint).

Returns

out

[([P,] M, N) ndarray (same dtype as input image)] Output image.

Examples

```
>>> from skimage import data
>>> from skimage.morphology import disk, ball
>>> from skimage.filters.rank import equalize
>>> import numpy as np
>>> img = data.camera()
>>> rng = np.random.default_rng()
>>> volume = rng.integers(0, 255, size=(10,10,10), dtype=np.uint8)
>>> equ = equalize(img, disk(5))
>>> equ_vol = equalize(volume, ball(5))
```

- Local Histogram Equalization

- Rank filters

```
skimage.filters.rank.geometric_mean(image, footprint, out=None, mask=None, shift_x=False,
shift_y=False, shift_z=False)
```

Return local geometric mean of an image.

Parameters**image**

[([P,] M, N) ndarray (uint8, uint16)] Input image.

footprint

[ndarray] The neighborhood expressed as an ndarray of 1's and 0's.

out

[([P,] M, N) array (same dtype as input)] If None, a new array is allocated.

mask

[ndarray (integer or float), optional] Mask array that defines (>0) area of the image included in the local neighborhood. If None, the complete image is used (default).

shift_x, shift_y, shift_z

[int] Offset added to the footprint center point. Shift is bounded to the footprint sizes (center must be inside the given footprint).

Returns**out**

[([P,] M, N) ndarray (same dtype as input image)] Output image.

References

[?]

Examples

```
>>> from skimage import data
>>> from skimage.morphology import disk, ball
>>> from skimage.filters.rank import mean
>>> import numpy as np
>>> img = data.camera()
>>> rng = np.random.default_rng()
>>> volume = rng.integers(0, 255, size=(10,10,10), dtype=np.uint8)
>>> avg = geometric_mean(img, disk(5))
>>> avg_vol = geometric_mean(volume, ball(5))
```

`skimage.filters.rank.gradient(image, footprint, out=None, mask=None, shift_x=False, shift_y=False, shift_z=False)`

Return local gradient of an image (i.e. local maximum - local minimum).

Parameters

image

[([P,] M, N) ndarray (uint8, uint16)] Input image.

footprint

[ndarray] The neighborhood expressed as an ndarray of 1's and 0's.

out

[([P,] M, N) array (same dtype as input)] If None, a new array is allocated.

mask

[ndarray (integer or float), optional] Mask array that defines (>0) area of the image included in the local neighborhood. If None, the complete image is used (default).

shift_x, shift_y, shift_z

[int] Offset added to the footprint center point. Shift is bounded to the footprint sizes (center must be inside the given footprint).

Returns

out

[([P,] M, N) ndarray (same dtype as input image)] Output image.

Examples

```
>>> from skimage import data
>>> from skimage.morphology import disk, ball
>>> from skimage.filters.rank import gradient
>>> import numpy as np
>>> img = data.camera()
>>> rng = np.random.default_rng()
>>> volume = rng.integers(0, 255, size=(10,10,10), dtype=np.uint8)
>>> out = gradient(img, disk(5))
>>> out_vol = gradient(volume, ball(5))
```

- *Markers for watershed transform*

- *Rank filters*

```
skimage.filters.rank.gradient_percentile(image, footprint, out=None, mask=None, shift_x=False, shift_y=False, p0=0, p1=1)
```

Return local gradient of an image (i.e. local maximum - local minimum).

Only grayvalues between percentiles [p0, p1] are considered in the filter.

Parameters

image

[2-D array (uint8, uint16)] Input image.

footprint

[2-D array] The neighborhood expressed as a 2-D array of 1's and 0's.

out

[2-D array (same dtype as input)] If None, a new array is allocated.

mask

[ndarray] Mask array that defines (>0) area of the image included in the local neighborhood.
If None, the complete image is used (default).

shift_x, shift_y

[int] Offset added to the footprint center point. Shift is bounded to the footprint sizes (center must be inside the given footprint).

p0, p1

[float in [0, ..., 1]] Define the [p0, p1] percentile interval to be considered for computing the value.

Returns

out

[2-D array (same dtype as input image)] Output image.

```
skimage.filters.rank.majority(image, footprint, *, out=None, mask=None, shift_x=False, shift_y=False, shift_z=False)
```

Assign to each pixel the most common value within its neighborhood.

Parameters

image

[ndarray] Image array (uint8, uint16 array).

footprint

[2-D array (integer or float)] The neighborhood expressed as a 2-D array of 1's and 0's.

out

[ndarray (integer or float), optional] If None, a new array will be allocated.

mask

[ndarray (integer or float), optional] Mask array that defines (>0) area of the image included in the local neighborhood. If None, the complete image is used (default).

shift_x, shift_y

[int, optional] Offset added to the footprint center point. Shift is bounded to the footprint sizes (center must be inside the given footprint).

Returns**out**

[2-D array (same dtype as input image)] Output image.

Examples

```
>>> from skimage import data
>>> from skimage.filters.rank import majority
>>> from skimage.morphology import disk, ball
>>> import numpy as np
>>> img = data.camera()
>>> rng = np.random.default_rng()
>>> volume = rng.integers(0, 255, size=(10,10,10), dtype=np.uint8)
>>> maj_img = majority(img, disk(5))
>>> maj_img_vol = majority(volume, ball(5))
```

```
skimage.filters.rank.maximum(image, footprint, out=None, mask=None, shift_x=False, shift_y=False, shift_z=False)
```

Return local maximum of an image.

Parameters

image

[([P,] M, N) ndarray (uint8, uint16)] Input image.

footprint

[ndarray] The neighborhood expressed as an ndarray of 1's and 0's.

out

[([P,] M, N) array (same dtype as input)] If None, a new array is allocated.

mask

[ndarray (integer or float), optional] Mask array that defines (>0) area of the image included in the local neighborhood. If None, the complete image is used (default).

shift_x, shift_y, shift_z

[int] Offset added to the footprint center point. Shift is bounded to the footprint sizes (center must be inside the given footprint).

Returns**out**

[([P,] M, N) ndarray (same dtype as input image)] Output image.

See also:

[*skimage.morphology.dilation*](#)

Notes

The lower algorithm complexity makes *skimage.filters.rank.maximum* more efficient for larger images and footprints.

Examples

```
>>> from skimage import data
>>> from skimage.morphology import disk, ball
>>> from skimage.filters.rank import maximum
>>> import numpy as np
>>> img = data.camera()
>>> rng = np.random.default_rng()
>>> volume = rng.integers(0, 255, size=(10,10,10), dtype=np.uint8)
>>> out = maximum(img, disk(5))
>>> out_vol = maximum(volume, ball(5))
```

- *Rank filters*

`skimage.filters.rank.mean(image, footprint, out=None, mask=None, shift_x=False, shift_y=False, shift_z=False)`

Return local mean of an image.

Parameters

image

[([P,] M, N) ndarray (uint8, uint16)] Input image.

footprint

[ndarray] The neighborhood expressed as an ndarray of 1's and 0's.

out

[([P,] M, N) array (same dtype as input)] If None, a new array is allocated.

mask

[ndarray (integer or float), optional] Mask array that defines (>0) area of the image included in the local neighborhood. If None, the complete image is used (default).

shift_x, shift_y, shift_z

[int] Offset added to the footprint center point. Shift is bounded to the footprint sizes (center must be inside the given footprint).

Returns**out**

[([P,] M, N) ndarray (same dtype as input image)] Output image.

Examples

```
>>> from skimage import data
>>> from skimage.morphology import disk, ball
>>> from skimage.filters.rank import mean
>>> import numpy as np
>>> img = data.camera()
>>> rng = np.random.default_rng()
>>> volume = rng.integers(0, 255, size=(10,10,10), dtype=np.uint8)
>>> avg = mean(img, disk(5))
>>> avg_vol = mean(volume, ball(5))
```

- *Mean filters*
- *Segment human cells (in mitosis)*
- *Rank filters*

```
skimage.filters.rank.mean_bilateral(image, footprint, out=None, mask=None, shift_x=False,
shift_y=False, s0=10, s1=10)
```

Apply a flat kernel bilateral filter.

This is an edge-preserving and noise reducing denoising filter. It averages pixels based on their spatial closeness and radiometric similarity.

Spatial closeness is measured by considering only the local pixel neighborhood given by a footprint (structuring element).

Radiometric similarity is defined by the graylevel interval $[g-s_0, g+s_1]$ where g is the current pixel graylevel.

Only pixels belonging to the footprint and having a graylevel inside this interval are averaged.

Parameters

image

[2-D array (uint8, uint16)] Input image.

footprint

[2-D array] The neighborhood expressed as a 2-D array of 1's and 0's.

out

[2-D array (same dtype as input)] If None, a new array is allocated.

mask

[ndarray] Mask array that defines (>0) area of the image included in the local neighborhood.
If None, the complete image is used (default).

shift_x, shift_y

[int] Offset added to the footprint center point. Shift is bounded to the footprint sizes (center must be inside the given footprint).

s0, s1

[int] Define the $[s_0, s_1]$ interval around the grayvalue of the center pixel to be considered for computing the value.

Returns

out

[2-D array (same dtype as input image)] Output image.

See also:

[denoise_bilateral](#)

Examples

```
>>> import numpy as np
>>> from skimage import data
>>> from skimage.morphology import disk
>>> from skimage.filters.rank import mean_bilateral
>>> img = data.camera().astype(np.uint16)
>>> bilat_img = mean_bilateral(img, disk(20), s0=10, s1=10)
```

- *Mean filters*
 - *Rank filters*
-

`skimage.filters.rank.mean_percentile(image, footprint, out=None, mask=None, shift_x=False, shift_y=False, p0=0, p1=1)`

Return local mean of an image.

Only grayvalues between percentiles [p0, p1] are considered in the filter.

Parameters

image

[2-D array (uint8, uint16)] Input image.

footprint

[2-D array] The neighborhood expressed as a 2-D array of 1's and 0's.

out

[2-D array (same dtype as input)] If None, a new array is allocated.

mask

[ndarray] Mask array that defines (>0) area of the image included in the local neighborhood.
If None, the complete image is used (default).

shift_x, shift_y

[int] Offset added to the footprint center point. Shift is bounded to the footprint sizes (center must be inside the given footprint).

p0, p1

[float in [0, ..., 1]] Define the [p0, p1] percentile interval to be considered for computing the value.

Returns

out

[2-D array (same dtype as input image)] Output image.

- *Mean filters*
-

```
skimage.filters.rank.median(image, footprint=None, out=None, mask=None, shift_x=False, shift_y=False, shift_z=False)
```

Return local median of an image.

Parameters**image**

[([P,] M, N) ndarray (uint8, uint16)] Input image.

footprint

[ndarray] The neighborhood expressed as an ndarray of 1's and 0's. If None, a full square of size 3 is used.

out

[([P,] M, N) array (same dtype as input)] If None, a new array is allocated.

mask

[ndarray (integer or float), optional] Mask array that defines (>0) area of the image included in the local neighborhood. If None, the complete image is used (default).

shift_x, shift_y, shift_z

[int] Offset added to the footprint center point. Shift is bounded to the footprint sizes (center must be inside the given footprint).

Returns**out**

[([P,] M, N) ndarray (same dtype as input image)] Output image.

See also:***skimage.filters.median***

Implementation of a median filtering which handles images with floating precision.

Examples

```
>>> from skimage import data
>>> from skimage.morphology import disk, ball
>>> from skimage.filters.rank import median
>>> import numpy as np
>>> img = data.camera()
>>> rng = np.random.default_rng()
>>> volume = rng.integers(0, 255, size=(10,10,10), dtype=np.uint8)
>>> med = median(img, disk(5))
>>> med_vol = median(volume, ball(5))
```

- *Markers for watershed transform*
 - *Rank filters*
-

`skimage.filters.rank.minimum(image, footprint, out=None, mask=None, shift_x=False, shift_y=False, shift_z=False)`

Return local minimum of an image.

Parameters

image

[([P,] M, N) ndarray (uint8, uint16)] Input image.

footprint

[ndarray] The neighborhood expressed as an ndarray of 1's and 0's.

out

[([P,] M, N) array (same dtype as input)] If None, a new array is allocated.

mask

[ndarray (integer or float), optional] Mask array that defines (>0) area of the image included in the local neighborhood. If None, the complete image is used (default).

shift_x, shift_y, shift_z

[int] Offset added to the footprint center point. Shift is bounded to the footprint sizes (center must be inside the given footprint).

Returns

out

[([P,] M, N) ndarray (same dtype as input image)] Output image.

See also:

`skimage.morphology.erosion`**Notes**

The lower algorithm complexity makes `skimage.filters.rank.minimum` more efficient for larger images and footprints.

Examples

```
>>> from skimage import data
>>> from skimage.morphology import disk, ball
>>> from skimage.filters.rank import minimum
>>> import numpy as np
>>> img = data.camera()
>>> rng = np.random.default_rng()
>>> volume = rng.integers(0, 255, size=(10,10,10), dtype=np.uint8)
>>> out = minimum(img, disk(5))
>>> out_vol = minimum(volume, ball(5))
```

- *Rank filters*

`skimage.filters.rank.modal(image, footprint, out=None, mask=None, shift_x=False, shift_y=False, shift_z=False)`

Return local mode of an image.

The mode is the value that appears most often in the local histogram.

Parameters**image**

[([P,] M, N) ndarray (uint8, uint16)] Input image.

footprint

[ndarray] The neighborhood expressed as an ndarray of 1's and 0's.

out

[([P,] M, N) array (same dtype as input)] If None, a new array is allocated.

mask

[ndarray (integer or float), optional] Mask array that defines (>0) area of the image included in the local neighborhood. If None, the complete image is used (default).

shift_x, shift_y, shift_z

[int] Offset added to the footprint center point. Shift is bounded to the footprint sizes (center must be inside the given footprint).

Returns**out**

[([P,] M, N) ndarray (same dtype as input image)] Output image.

Examples

```
>>> from skimage import data
>>> from skimage.morphology import disk, ball
>>> from skimage.filters.rank import modal
>>> import numpy as np
>>> img = data.camera()
>>> rng = np.random.default_rng()
>>> volume = rng.integers(0, 255, size=(10,10,10), dtype=np.uint8)
>>> out = modal(img, disk(5))
>>> out_vol = modal(volume, ball(5))
```

`skimage.filters.rank.noise_filter(image, footprint, out=None, mask=None, shift_x=False, shift_y=False, shift_z=False)`

Noise feature.

Parameters**image**

[([P,] M, N) ndarray (uint8, uint16)] Input image.

footprint

[ndarray] The neighborhood expressed as an ndarray of 1's and 0's.

out

[([P,] M, N) array (same dtype as input)] If None, a new array is allocated.

mask

[ndarray (integer or float), optional] Mask array that defines (>0) area of the image included in the local neighborhood. If None, the complete image is used (default).

shift_x, shift_y, shift_z

[int] Offset added to the footprint center point. Shift is bounded to the footprint sizes (center must be inside the given footprint).

Returns

out

[([P,] M, N) ndarray (same dtype as input image)] Output image.

References

[?]

Examples

```
>>> from skimage import data
>>> from skimage.morphology import disk, ball
>>> from skimage.filters.rank import noise_filter
>>> import numpy as np
>>> img = data.camera()
>>> rng = np.random.default_rng()
>>> volume = rng.integers(0, 255, size=(10,10,10), dtype=np.uint8)
>>> out = noise_filter(img, disk(5))
>>> out_vol = noise_filter(volume, ball(5))
```

`skimage.filters.rank.otsu(image, footprint, out=None, mask=None, shift_x=False, shift_y=False, shift_z=False)`

Local Otsu's threshold value for each pixel.

Parameters**image**

[([P,] M, N) ndarray (uint8, uint16)] Input image.

footprint

[ndarray] The neighborhood expressed as an ndarray of 1's and 0's.

out

[([P,] M, N) array (same dtype as input)] If None, a new array is allocated.

mask

[ndarray (integer or float), optional] Mask array that defines (>0) area of the image included in the local neighborhood. If None, the complete image is used (default).

shift_x, shift_y, shift_z

[int] Offset added to the footprint center point. Shift is bounded to the footprint sizes (center must be inside the given footprint).

Returns

out

[([P,] M, N) ndarray (same dtype as input image)] Output image.

References

[?]

Examples

```
>>> from skimage import data
>>> from skimage.filters.rank import otsu
>>> from skimage.morphology import disk, ball
>>> import numpy as np
>>> img = data.camera()
>>> rng = np.random.default_rng()
>>> volume = rng.integers(0, 255, size=(10,10,10), dtype=np.uint8)
>>> local_otsu = otsu(img, disk(5))
>>> thresh_image = img >= local_otsu
>>> local_otsu_vol = otsu(volume, ball(5))
>>> thresh_image_vol = volume >= local_otsu_vol
```

- *Thresholding*
 - *Rank filters*
-

`skimage.filters.rank.percentile(image, footprint, out=None, mask=None, shift_x=False, shift_y=False, p0=0)`

Return local percentile of an image.

Returns the value of the p0 lower percentile of the local grayvalue distribution.

Only grayvalues between percentiles [p0, p1] are considered in the filter.

Parameters

image

[2-D array (uint8, uint16)] Input image.

footprint

[2-D array] The neighborhood expressed as a 2-D array of 1's and 0's.

out

[2-D array (same dtype as input)] If None, a new array is allocated.

mask

[ndarray] Mask array that defines (>0) area of the image included in the local neighborhood.
If None, the complete image is used (default).

shift_x, shift_y

[int] Offset added to the footprint center point. Shift is bounded to the footprint sizes (center must be inside the given footprint).

p0

[float in [0, ..., 1]] Set the percentile value.

Returns**out**

[2-D array (same dtype as input image)] Output image.

```
skimage.filters.rank.pop(image, footprint, out=None, mask=None, shift_x=False, shift_y=False, shift_z=False)
```

Return the local number (population) of pixels.

The number of pixels is defined as the number of pixels which are included in the footprint and the mask.

Parameters**image**

[([P,] M, N) ndarray (uint8, uint16)] Input image.

footprint

[ndarray] The neighborhood expressed as an ndarray of 1's and 0's.

out

[([P,] M, N) array (same dtype as input)] If None, a new array is allocated.

mask

[ndarray (integer or float), optional] Mask array that defines (>0) area of the image included in the local neighborhood. If None, the complete image is used (default).

shift_x, shift_y, shift_z

[int] Offset added to the footprint center point. Shift is bounded to the footprint sizes (center must be inside the given footprint).

Returns**out**

[([P,] M, N) ndarray (same dtype as input image)] Output image.

Examples

```
>>> from skimage.morphology import square, cube # Need to add 3D example
>>> import skimage.filters.rank as rank
>>> img = 255 * np.array([[0, 0, 0, 0, 0],
...                         [0, 1, 1, 1, 0],
...                         [0, 1, 1, 1, 0],
...                         [0, 1, 1, 1, 0],
...                         [0, 0, 0, 0, 0]], dtype=np.uint8)
>>> rank.pop(img, square(3))
array([[4, 6, 6, 6, 4],
       [6, 9, 9, 9, 6],
       [6, 9, 9, 9, 6],
       [6, 9, 9, 9, 6],
       [4, 6, 6, 6, 4]], dtype=uint8)
```

```
skimage.filters.rank.pop_bilateral(image, footprint, out=None, mask=None, shift_x=False, shift_y=False,
                                     s0=10, s1=10)
```

Return the local number (population) of pixels.

The number of pixels is defined as the number of pixels which are included in the footprint and the mask. Additionally pixels must have a graylevel inside the interval $[g-s_0, g+s_1]$ where g is the grayvalue of the center pixel.

Parameters

image

[2-D array (uint8, uint16)] Input image.

footprint

[2-D array] The neighborhood expressed as a 2-D array of 1's and 0's.

out

[2-D array (same dtype as input)] If None, a new array is allocated.

mask

[ndarray] Mask array that defines (>0) area of the image included in the local neighborhood.
If None, the complete image is used (default).

shift_x, shift_y

[int] Offset added to the footprint center point. Shift is bounded to the footprint sizes (center must be inside the given footprint).

s0, s1

[int] Define the $[s_0, s_1]$ interval around the grayvalue of the center pixel to be considered for computing the value.

Returns**out**

[2-D array (same dtype as input image)] Output image.

Examples

```
>>> import numpy as np
>>> from skimage.morphology import square
>>> import skimage.filters.rank as rank
>>> img = 255 * np.array([[0, 0, 0, 0, 0],
...                      [0, 1, 1, 1, 0],
...                      [0, 1, 1, 1, 0],
...                      [0, 1, 1, 1, 0],
...                      [0, 0, 0, 0, 0]], dtype=np.uint16)
>>> rank.pop_bilateral(img, square(3), s0=10, s1=10)
array([[3, 4, 3, 4, 3],
       [4, 4, 6, 4, 4],
       [3, 6, 9, 6, 3],
       [4, 4, 6, 4, 4],
       [3, 4, 3, 4, 3]], dtype=uint16)
```

`skimage.filters.rank.pop_percentile(image, footprint, out=None, mask=None, shift_x=False, shift_y=False, p0=0, p1=1)`

Return the local number (population) of pixels.

The number of pixels is defined as the number of pixels which are included in the footprint and the mask.

Only grayvalues between percentiles [p0, p1] are considered in the filter.

Parameters**image**

[2-D array (uint8, uint16)] Input image.

footprint

[2-D array] The neighborhood expressed as a 2-D array of 1's and 0's.

out

[2-D array (same dtype as input)] If None, a new array is allocated.

mask

[ndarray] Mask array that defines (>0) area of the image included in the local neighborhood.
If None, the complete image is used (default).

shift_x, shift_y

[int] Offset added to the footprint center point. Shift is bounded to the footprint sizes (center must be inside the given footprint).

p0, p1

[float in [0, …, 1]] Define the [p0, p1] percentile interval to be considered for computing the value.

Returns

out

[2-D array (same dtype as input image)] Output image.

`skimage.filters.rank.subtract_mean(image, footprint, out=None, mask=None, shift_x=False, shift_y=False, shift_z=False)`

Return image subtracted from its local mean.

Parameters

image

[([P,] M, N) ndarray (uint8, uint16)] Input image.

footprint

[ndarray] The neighborhood expressed as an ndarray of 1's and 0's.

out

[([P,] M, N) array (same dtype as input)] If None, a new array is allocated.

mask

[ndarray (integer or float), optional] Mask array that defines (>0) area of the image included in the local neighborhood. If None, the complete image is used (default).

shift_x, shift_y, shift_z

[int] Offset added to the footprint center point. Shift is bounded to the footprint sizes (center must be inside the given footprint).

Returns

out

[([P,] M, N) ndarray (same dtype as input image)] Output image.

Notes

Subtracting the mean value may introduce underflow. To compensate this potential underflow, the obtained difference is downscaled by a factor of 2 and shifted by $n_bins / 2 - 1$, the median value of the local histogram ($n_bins = \max(3, \text{image}.max()) + 1$ for 16-bits images and 256 otherwise).

Examples

```
>>> from skimage import data
>>> from skimage.morphology import disk, ball
>>> from skimage.filters.rank import subtract_mean
>>> import numpy as np
>>> img = data.camera()
>>> rng = np.random.default_rng()
>>> volume = rng.integers(0, 255, size=(10,10,10), dtype=np.uint8)
>>> out = subtract_mean(img, disk(5))
>>> out_vol = subtract_mean(volume, ball(5))
```

`skimage.filters.rank.subtract_mean_percentile(image, footprint, out=None, mask=None, shift_x=False, shift_y=False, p0=0, p1=1)`

Return image subtracted from its local mean.

Only grayvalues between percentiles [p0, p1] are considered in the filter.

Parameters

image

[2-D array (uint8, uint16)] Input image.

footprint

[2-D array] The neighborhood expressed as a 2-D array of 1's and 0's.

out

[2-D array (same dtype as input)] If None, a new array is allocated.

mask

[ndarray] Mask array that defines (>0) area of the image included in the local neighborhood.
If None, the complete image is used (default).

shift_x, shift_y

[int] Offset added to the footprint center point. Shift is bounded to the footprint sizes (center must be inside the given footprint).

p0, p1

[float in [0, ..., 1]] Define the [p0, p1] percentile interval to be considered for computing the value.

Returns

out

[2-D array (same dtype as input image)] Output image.

`skimage.filters.rank.sum(image, footprint, out=None, mask=None, shift_x=False, shift_y=False, shift_z=False)`

Return the local sum of pixels.

Note that the sum may overflow depending on the data type of the input array.

Parameters

image

[([P,] M, N) ndarray (uint8, uint16)] Input image.

footprint

[ndarray] The neighborhood expressed as an ndarray of 1's and 0's.

out

[([P,] M, N) array (same dtype as input)] If None, a new array is allocated.

mask

[ndarray (integer or float), optional] Mask array that defines (>0) area of the image included in the local neighborhood. If None, the complete image is used (default).

shift_x, shift_y, shift_z

[int] Offset added to the footprint center point. Shift is bounded to the footprint sizes (center must be inside the given footprint).

Returns

out

[([P,] M, N) ndarray (same dtype as input image)] Output image.

Examples

```
>>> from skimage.morphology import square, cube # Need to add 3D example
>>> import skimage.filters.rank as rank          # Cube seems to fail but
>>> img = np.array([[0, 0, 0, 0, 0],
...                  [0, 1, 1, 1, 0],
...                  [0, 1, 1, 1, 0],
...                  [0, 1, 1, 1, 0],
...                  [0, 0, 0, 0, 0]], dtype=np.uint8)
>>> rank.sum(img, square(3))
array([[1, 2, 3, 2, 1],
       [2, 4, 6, 4, 2],
       [3, 6, 9, 6, 3],
       [2, 4, 6, 4, 2],
       [1, 2, 3, 2, 1]], dtype=uint8)
```

`skimage.filters.rank.sum_bilateral(image, footprint, out=None, mask=None, shift_x=False, shift_y=False, s0=10, s1=10)`

Apply a flat kernel bilateral filter.

This is an edge-preserving and noise reducing denoising filter. It averages pixels based on their spatial closeness and radiometric similarity.

Spatial closeness is measured by considering only the local pixel neighborhood given by a footprint (structuring element).

Radiometric similarity is defined by the graylevel interval $[g-s_0, g+s_1]$ where g is the current pixel graylevel.

Only pixels belonging to the footprint AND having a graylevel inside this interval are summed.

Note that the sum may overflow depending on the data type of the input array.

Parameters

image

[2-D array (uint8, uint16)] Input image.

footprint

[2-D array] The neighborhood expressed as a 2-D array of 1's and 0's.

out

[2-D array (same dtype as input)] If None, a new array is allocated.

mask

[ndarray] Mask array that defines (>0) area of the image included in the local neighborhood.
If None, the complete image is used (default).

shift_x, shift_y

[int] Offset added to the footprint center point. Shift is bounded to the footprint sizes (center must be inside the given footprint).

s0, s1

[int] Define the [s0, s1] interval around the grayvalue of the center pixel to be considered for computing the value.

Returns**out**

[2-D array (same dtype as input image)] Output image.

See also:

[denoise_bilateral](#)

Examples

```
>>> import numpy as np
>>> from skimage import data
>>> from skimage.morphology import disk
>>> from skimage.filters.rank import sum_bilateral
>>> img = data.camera().astype(np.uint16)
>>> bilat_img = sum_bilateral(img, disk(10), s0=10, s1=10)
```

```
skimage.filters.rank.sum_percentile(image, footprint, out=None, mask=None, shift_x=False,
shift_y=False, p0=0, p1=1)
```

Return the local sum of pixels.

Only grayvalues between percentiles [p0, p1] are considered in the filter.

Note that the sum may overflow depending on the data type of the input array.

Parameters**image**

[2-D array (uint8, uint16)] Input image.

footprint

[2-D array] The neighborhood expressed as a 2-D array of 1's and 0's.

out

[2-D array (same dtype as input)] If None, a new array is allocated.

mask

[ndarray] Mask array that defines (>0) area of the image included in the local neighborhood. If None, the complete image is used (default).

shift_x, shift_y

[int] Offset added to the footprint center point. Shift is bounded to the footprint sizes (center must be inside the given footprint).

p0, p1

[float in [0, …, 1]] Define the [p0, p1] percentile interval to be considered for computing the value.

Returns**out**

[2-D array (same dtype as input image)] Output image.

```
skimage.filters.rank.threshold(image, footprint, out=None, mask=None, shift_x=False, shift_y=False, shift_z=False)
```

Local threshold of an image.

The resulting binary mask is True if the gray value of the center pixel is greater than the local mean.

Parameters**image**

[([P,] M, N) ndarray (uint8, uint16)] Input image.

footprint

[ndarray] The neighborhood expressed as an ndarray of 1's and 0's.

out

[([P,] M, N) array (same dtype as input)] If None, a new array is allocated.

mask

[ndarray (integer or float), optional] Mask array that defines (>0) area of the image included in the local neighborhood. If None, the complete image is used (default).

shift_x, shift_y, shift_z

[int] Offset added to the footprint center point. Shift is bounded to the footprint sizes (center must be inside the given footprint).

Returns**out**

[([P,] M, N) ndarray (same dtype as input image)] Output image.

Examples

```
>>> from skimage.morphology import square, cube # Need to add 3D example
>>> from skimage.filters.rank import threshold
>>> img = 255 * np.array([[0, 0, 0, 0, 0],
...                         [0, 1, 1, 1, 0],
...                         [0, 1, 1, 1, 0],
...                         [0, 1, 1, 1, 0],
...                         [0, 0, 0, 0, 0]], dtype=np.uint8)
>>> threshold(img, square(3))
array([[0, 0, 0, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 0, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 0, 0, 0]], dtype=uint8)
```

```
skimage.filters.rank.threshold_percentile(image, footprint, out=None, mask=None, shift_x=False, shift_y=False, p0=0)
```

Local threshold of an image.

The resulting binary mask is True if the grayvalue of the center pixel is greater than the local mean.

Only grayvalues between percentiles [p0, p1] are considered in the filter.

Parameters

image

[2-D array (uint8, uint16)] Input image.

footprint

[2-D array] The neighborhood expressed as a 2-D array of 1's and 0's.

out

[2-D array (same dtype as input)] If None, a new array is allocated.

mask

[ndarray] Mask array that defines (>0) area of the image included in the local neighborhood.
If None, the complete image is used (default).

shift_x, shift_y

[int] Offset added to the footprint center point. Shift is bounded to the footprint sizes (center must be inside the given footprint).

p0

[float in [0, ..., 1]] Set the percentile value.

Returns

out

[2-D array (same dtype as input image)] Output image.

```
skimage.filters.rank.windowed_histogram(image, footprint, out=None, mask=None, shift_x=False,  
shift_y=False, n_bins=None)
```

Normalized sliding window histogram

Parameters**image**

[2-D array (integer or float)] Input image.

footprint

[2-D array (integer or float)] The neighborhood expressed as a 2-D array of 1's and 0's.

out

[2-D array (integer or float), optional] If None, a new array is allocated.

mask

[ndarray (integer or float), optional] Mask array that defines (>0) area of the image included in the local neighborhood. If None, the complete image is used (default).

shift_x, shift_y

[int, optional] Offset added to the footprint center point. Shift is bounded to the footprint sizes (center must be inside the given footprint).

n_bins

[int or None] The number of histogram bins. Will default to `image.max() + 1` if None is passed.

Returns**out**

[3-D array (float)] Array of dimensions (H,W,N), where (H,W) are the dimensions of the input image and N is `n_bins` or `image.max() + 1` if no value is provided as a parameter. Effectively, each pixel is a N-D feature vector that is the histogram. The sum of the elements in the feature vector will be 1, unless no pixels in the window were covered by both footprint and mask, in which case all elements will be 0.

Examples

```
>>> from skimage import data
>>> from skimage.filters.rank import windowed_histogram
>>> from skimage.morphology import disk, ball
>>> import numpy as np
>>> img = data.camera()
>>> rng = np.random.default_rng()
>>> volume = rng.integers(0, 255, size=(10,10,10), dtype=np.uint8)
>>> hist_img = windowed_histogram(img, disk(5))
```

- Sliding window histogram

1.3.9 skimage.future

Functionality with an experimental API.

Warning: Although you can count on the functions in this package being around in the future, the API may change with any version update **and will not follow the skimage two-version deprecation path**. Therefore, use the functions herein with care, and do not use them in production code that will depend on updated skimage versions.

<code>skimage.future.fit_segmenter</code>	Segmentation using labeled parts of the image and a classifier.
<code>skimage.future.manual_lasso_segmentation</code>	Return a label image based on freeform selections made with the mouse.
<code>skimage.future.manual_polygon_segmentation</code>	Return a label image based on polygon selections made with the mouse.
<code>skimage.future.predict_segmenter</code>	Segmentation of images using a pretrained classifier.
<code>skimage.future.TrainableSegmenter</code>	Estimator for classifying pixels.

`skimage.future.fit_segmenter(labels, features, clf)`

Segmentation using labeled parts of the image and a classifier.

Parameters

`labels`

[ndarray of ints] Image of labels. Labels ≥ 1 correspond to the training set and label 0 to unlabeled pixels to be segmented.

`features`

[ndarray] Array of features, with the first dimension corresponding to the number of features, and the other dimensions correspond to `labels.shape`.

clf

[classifier object] classifier object, exposing a `fit` and a `predict` method as in scikit-learn's API, for example an instance of `RandomForestClassifier` or `LogisticRegression` classifier.

Returns**clf**

[classifier object] classifier trained on `labels`

Raises

NotFittedError if `self.clf` has not been fitted yet (use `self.fit`).

-
- *Trainable segmentation using local features and random forests*

skimage.future.manual_lasso_segmentation(image, alpha=0.4, return_all=False)

Return a label image based on freeform selections made with the mouse.

Parameters**image**

[(M, N[, 3]) array] Grayscale or RGB image.

alpha

[float, optional] Transparency value for polygons drawn over the image.

return_all

[bool, optional] If True, an array containing each separate polygon drawn is returned. (The polygons may overlap.) If False (default), latter polygons “overwrite” earlier ones where they overlap.

Returns**labels**

[array of int, shape ([Q,]M, N)] The segmented regions. If mode is ‘*separate*’, the leading dimension of the array corresponds to the number of regions that the user drew.

Notes

Press and hold the left mouse button to draw around each object.

Examples

```
>>> from skimage import data, future, io
>>> camera = data.camera()
>>> mask = future.manual_lasso_segmentation(camera)
>>> io.imshow(mask)
>>> io.show()
```

```
skimage.future.manual_polygon_segmentation(image, alpha=0.4, return_all=False)
```

Return a label image based on polygon selections made with the mouse.

Parameters

image

[(M, N[, 3]) array] Grayscale or RGB image.

alpha

[float, optional] Transparency value for polygons drawn over the image.

return_all

[bool, optional] If True, an array containing each separate polygon drawn is returned. (The polygons may overlap.) If False (default), latter polygons “overwrite” earlier ones where they overlap.

Returns

labels

[array of int, shape ([Q,]M, N)] The segmented regions. If mode is ‘separate’, the leading dimension of the array corresponds to the number of regions that the user drew.

Notes

Use left click to select the vertices of the polygon and right click to confirm the selection once all vertices are selected.

Examples

```
>>> from skimage import data, future, io
>>> camera = data.camera()
>>> mask = future.manual_polygon_segmentation(camera)
>>> io.imshow(mask)
>>> io.show()
```

`skimage.future.predict_segmenter(features, clf)`

Segmentation of images using a pretrained classifier.

Parameters

features

[ndarray] Array of features, with the last dimension corresponding to the number of features, and the other dimensions are compatible with the shape of the image to segment, or a flattened image.

clf

[classifier object] trained classifier object, exposing a `predict` method as in scikit-learn's API, for example an instance of `RandomForestClassifier` or `LogisticRegression` classifier. The classifier must be already trained, for example with `skimage.segmentation.fit_segmenter()`.

Returns

output

[ndarray] Labeled array, built from the prediction of the classifier.

- *Trainable segmentation using local features and random forests*

`class skimage.future.TrainableSegmenter(clf=None, features_func=None)`

Bases: `object`

Estimator for classifying pixels.

Parameters

clf

[classifier object, optional] classifier object, exposing a `fit` and a `predict` method as in scikit-learn's API, for example an instance of `RandomForestClassifier` or `LogisticRegression` classifier.

features_func

[function, optional] function computing features on all pixels of the image, to be passed to the classifier. The output should be of shape (`m_features`, *`labels.shape`). If None, `skimage.feature.multiscale_basic_features()` is used.

Methods

<code>fit(image, labels)</code>	Train classifier using partially labeled (annotated) image.
<code>predict(image)</code>	Segment new image using trained internal classifier.

compute_features

`__init__(clf=None, features_func=None)`

`compute_features(image)`

`fit(image, labels)`

Train classifier using partially labeled (annotated) image.

Parameters**image**

[ndarray] Input image, which can be grayscale or multichannel, and must have a number of dimensions compatible with `self.features_func`.

labels

[ndarray of ints] Labeled array of shape compatible with `image` (same shape for a single-channel image). Labels ≥ 1 correspond to the training set and label 0 to unlabeled pixels to be segmented.

`predict(image)`

Segment new image using trained internal classifier.

Parameters**image**

[ndarray] Input image, which can be grayscale or multichannel, and must have a number of dimensions compatible with `self.features_func`.

Raises

NotFittedError if `self.clf` has not been fitted yet (use `self.fit`).

1.3.10 skimage.graph

<code>skimage.graph.central_pixel</code>	Find the pixel with the highest closeness centrality.
<code>skimage.graph.cut_normalized</code>	Perform Normalized Graph cut on the Region Adjacency Graph.
<code>skimage.graph.cut_threshold</code>	Combine regions separated by weight less than threshold.
<code>skimage.graph.merge_hierarchical</code>	Perform hierarchical merging of a RAG.
<code>skimage.graph.pixel_graph</code>	Create an adjacency graph of pixels in an image.
<code>skimage.graph.rag_boundary</code>	Comouter RAG based on region boundaries
<code>skimage.graph.rag_mean_color</code>	Compute the Region Adjacency Graph using mean colors.
<code>skimage.graph.route_through_array</code>	Simple example of how to use the MCP and MCP_Geometric classes.
<code>skimage.graph.shortest_path</code>	Find the shortest path through an n-d array from one side to another.
<code>skimage.graph.MCP</code>	A class for finding the minimum cost path through a given n-d costs array.
<code>skimage.graph.MCP_Connect</code>	Connect source points using the distance-weighted minimum cost function.
<code>skimage.graph.MCP_Flexible</code>	Find minimum cost paths through an N-d costs array.
<code>skimage.graph.MCP_Geometric</code>	Find distance-weighted minimum cost paths through an n-d costs array.
<code>skimage.graph.RAG</code>	The Region Adjacency Graph (RAG) of an image, subclasses <code>networkx.Graph</code>

`skimage.graph.central_pixel(graph, nodes=None, shape=None, partition_size=100)`

Find the pixel with the highest closeness centrality.

Closeness centrality is the inverse of the total sum of shortest distances from a node to every other node.

Parameters**graph**

[scipy.sparse.csr_matrix] The sparse matrix representation of the graph.

nodes

[array of int] The raveled index of each node in graph in the image. If not provided, the returned value will be the index in the input graph.

shape

[tuple of int] The shape of the image in which the nodes are embedded. If provided, the returned coordinates are a NumPy multi-index of the same dimensionality as the input shape. Otherwise, the returned coordinate is the raveled index provided in *nodes*.

partition_size

[int] This function computes the shortest path distance between every pair of nodes in the graph. This can result in a very large (N^2) matrix. As a simple performance tweak, the distance values are computed in lots of *partition_size*, resulting in a memory requirement of only *partition_size* $\cdot N$.

Returns**position**

[int or tuple of int] If shape is given, the coordinate of the central pixel in the image. Otherwise, the raveled index of that pixel.

distances

[array of float] The total sum of distances from each node to each other reachable node.

- Use pixel graphs to find an object's geodesic center
-

```
skimage.graph.cut_normalized(labels, rag, thresh=0.001, num_cuts=10, in_place=True, max_edge=1.0, *,  
    rng=None)
```

Perform Normalized Graph cut on the Region Adjacency Graph.

Given an image's labels and its similarity RAG, recursively perform a 2-way normalized cut on it. All nodes belonging to a subgraph that cannot be cut further are assigned a unique label in the output.

Parameters**labels**

[ndarray] The array of labels.

rag

[RAG] The region adjacency graph.

thresh

[float] The threshold. A subgraph won't be further subdivided if the value of the N-cut exceeds *thresh*.

num_cuts

[int] The number of N-cuts to perform before determining the optimal one.

in_place

[bool] If set, modifies *rag* in place. For each node *n* the function will set a new attribute *rag.nodes[n]['ncut_label']*.

max_edge

[float, optional] The maximum possible value of an edge in the RAG. This corresponds to an edge between identical regions. This is used to put self edges in the RAG.

rng

[{`numpy.random.Generator`, int}, optional] Pseudo-random number generator. By default, a PCG64 generator is used (see `numpy.random.default_rng()`). If *rng* is an int, it is used to seed the generator.

The *rng* is used to determine the starting point of `scipy.sparse.linalg.eigsh`.

Returns**out**

[ndarray] The new labeled array.

Other Parameters**random_state**

[DEPRECATED] Deprecated in favor of *rng*.

Deprecated since version 0.21.

References

[?]

Examples

```
>>> from skimage import data, segmentation, graph
>>> img = data.astronaut()
>>> labels = segmentation.slic(img)
>>> rag = graph.rag_mean_color(img, labels, mode='similarity')
>>> new_labels = graph.cut_normalized(labels, rag)
```

- *Normalized Cut*

`skimage.graph.cut_threshold(labels, rag, thresh, in_place=True)`

Combine regions separated by weight less than threshold.

Given an image's labels and its RAG, output new labels by combining regions whose nodes are separated by a weight less than the given threshold.

Parameters

labels

[ndarray] The array of labels.

rag

[RAG] The region adjacency graph.

thresh

[float] The threshold. Regions connected by edges with smaller weights are combined.

in_place

[bool] If set, modifies *rag* in place. The function will remove the edges with weights less than *thresh*. If set to *False* the function makes a copy of *rag* before proceeding.

Returns

out

[ndarray] The new labelled array.

References

[?]

Examples

```
>>> from skimage import data, segmentation, graph
>>> img = data.astronaut()
>>> labels = segmentation.slic(img)
>>> rag = graph.rag_mean_color(img, labels)
>>> new_labels = graph.cut_threshold(labels, rag, 10)
```

- *RAG Thresholding*

`skimage.graph.merge_hierarchical(labels, rag, thresh, rag_copy, in_place_merge, merge_func, weight_func)`

Perform hierarchical merging of a RAG.

Greedily merges the most similar pair of nodes until no edges lower than *thresh* remain.

Parameters**labels**

[ndarray] The array of labels.

rag

[RAG] The Region Adjacency Graph.

thresh

[float] Regions connected by an edge with weight smaller than *thresh* are merged.

rag_copy

[bool] If set, the RAG copied before modifying.

in_place_merge

[bool] If set, the nodes are merged in place. Otherwise, a new node is created for each merge..

merge_func

[callable] This function is called before merging two nodes. For the RAG *graph* while merging *src* and *dst*, it is called as follows `merge_func(graph, src, dst)`.

weight_func

[callable] The function to compute the new weights of the nodes adjacent to the merged node. This is directly supplied as the argument *weight_func* to *merge_nodes*.

Returns**out**

[ndarray] The new labeled array.

- *RAG Merging*
- *Hierarchical Merging of Region Boundary RAGs*

`skimage.graph.pixel_graph(image, *, mask=None, edge_function=None, connectivity=1, spacing=None)`

Create an adjacency graph of pixels in an image.

Pixels where the mask is True are nodes in the returned graph, and they are connected by edges to their neighbors according to the connectivity parameter. By default, the *value* of an edge when a mask is given, or when the image is itself the mask, is the euclidean distance between the pixels.

However, if an int- or float-valued image is given with no mask, the value of the edges is the absolute difference in intensity between adjacent pixels, weighted by the euclidean distance.

Parameters

image

[array] The input image. If the image is of type bool, it will be used as the mask as well.

mask

[array of bool] Which pixels to use. If None, the graph for the whole image is used.

edge_function

[callable] A function taking an array of pixel values, and an array of neighbor pixel values, and an array of distances, and returning a value for the edge. If no function is given, the value of an edge is just the distance.

connectivity

[int] The square connectivity of the pixel neighborhood: the number of orthogonal steps allowed to consider a pixel a neighbor. See [scipy.ndimage.generate_binary_structure](#) for details.

spacing

[tuple of float] The spacing between pixels along each axis.

Returns

graph

[scipy.sparse.csr_matrix] A sparse adjacency matrix in which entry (i, j) is 1 if nodes i and j are neighbors, 0 otherwise.

nodes

[array of int] The nodes of the graph. These correspond to the raveled indices of the nonzero pixels in the mask.

- Use pixel graphs to find an object's geodesic center
-

skimage.graph.rag_boundary(labels, edge_map, connectivity=2)

Comouter RAG based on region boundaries

Given an image's initial segmentation and its edge map this method constructs the corresponding Region Adjacency Graph (RAG). Each node in the RAG represents a set of pixels within the image with the same label in *labels*. The weight between two adjacent regions is the average value in *edge_map* along their boundary.

labels

[ndarray] The labelled image.

edge_map

[ndarray] This should have the same shape as that of *labels*. For all pixels along the boundary between 2 adjacent regions, the average value of the corresponding pixels in *edge_map* is the edge weight between them.

connectivity

[int, optional] Pixels with a squared distance less than *connectivity* from each other are considered adjacent. It can range from 1 to *labels.ndim*. Its behavior is the same as *connectivity* parameter in `scipy.ndimage.generate_binary_structure`.

Examples

```
>>> from skimage import data, segmentation, filters, color, graph
>>> img = data.chelsea()
>>> labels = segmentation.slic(img)
>>> edge_map = filters.sobel(color.rgb2gray(img))
>>> rag = graph.rag_boundary(labels, edge_map)
```

- *Region Boundary based RAGs*
- *Hierarchical Merging of Region Boundary RAGs*

`skimage.graph.rag_mean_color(image, labels, connectivity=2, mode='distance', sigma=255.0)`

Compute the Region Adjacency Graph using mean colors.

Given an image and its initial segmentation, this method constructs the corresponding Region Adjacency Graph (RAG). Each node in the RAG represents a set of pixels within *image* with the same label in *labels*. The weight between two adjacent regions represents how similar or dissimilar two regions are depending on the *mode* parameter.

Parameters**image**

[ndarray, shape(M, N, [..., P], 3)] Input image.

labels

[ndarray, shape(M, N, [..., P])] The labelled image. This should have one dimension less than *image*. If *image* has dimensions (M, N, 3) *labels* should have dimensions (M, N).

connectivity

[int, optional] Pixels with a squared distance less than *connectivity* from each other are

considered adjacent. It can range from 1 to *labels.ndim*. Its behavior is the same as *connectivity* parameter in `scipy.ndimage.generate_binary_structure`.

mode

[{‘distance’, ‘similarity’}, optional] The strategy to assign edge weights.

‘distance’ : The weight between two adjacent regions is the $|c_1 - c_2|$, where c_1 and c_2 are the mean colors of the two regions. It represents the Euclidean distance in their average color.

‘similarity’ : The weight between two adjacent is $e^{-d^2/\sigma}$ where $d = |c_1 - c_2|$, where c_1 and c_2 are the mean colors of the two regions. It represents how similar two regions are.

sigma

[float, optional] Used for computation when *mode* is “similarity”. It governs how close to each other two colors should be, for their corresponding edge weight to be significant. A very large value of *sigma* could make any two colors behave as though they were similar.

Returns

out

[RAG] The region adjacency graph.

References

[?]

Examples

```
>>> from skimage import data, segmentation, graph
>>> img = data.astronaut()
>>> labels = segmentation.slic(img)
>>> rag = graph.rag_mean_color(img, labels)
```

- *RAG Thresholding*
- *Normalized Cut*
- *Drawing Region Adjacency Graphs (RAGs)*
- *RAG Merging*

`skimage.graph.route_through_array(array, start, end, fully_connected=True, geometric=True)`

Simple example of how to use the MCP and MCP_Geometric classes.

See the MCP and MCP_Geometric class documentation for explanation of the path-finding algorithm.

Parameters**array**

[ndarray] Array of costs.

start

[iterable] n-d index into `array` defining the starting point

end

[iterable] n-d index into `array` defining the end point

fully_connected

[bool (optional)] If True, diagonal moves are permitted, if False, only axial moves.

geometric

[bool (optional)] If True, the MCP_Geometric class is used to calculate costs, if False, the MCP base class is used. See the class documentation for an explanation of the differences between MCP and MCP_Geometric.

Returns**path**

[list] List of n-d index tuples defining the path from `start` to `end`.

cost

[float] Cost of the path. If `geometric` is False, the cost of the path is the sum of the values of `array` along the path. If `geometric` is True, a finer computation is made (see the documentation of the MCP_Geometric class).

See also:

`MCP`, `MCP_Geometric`

Examples

```
>>> import numpy as np
>>> from skimage.graph import route_through_array
>>>
>>> image = np.array([[1, 3], [10, 12]])
>>> image
array([[ 1,  3],
       [10, 12]])
>>> # Forbid diagonal steps
>>> route_through_array(image, [0, 0], [1, 1], fully_connected=False)
```

(continues on next page)

(continued from previous page)

```
(([0, 0], [0, 1], [1, 1]), 9.5)
>>> # Now allow diagonal steps: the path goes directly from start to end
>>> route_through_array(image, [0, 0], [1, 1])
(([0, 0], [1, 1]), 9.19238815542512)
>>> # Cost is the sum of array values along the path (16 = 1 + 3 + 12)
>>> route_through_array(image, [0, 0], [1, 1], fully_connected=False,
... geometric=False)
(([0, 0], [0, 1], [1, 1]), 16.0)
>>> # Larger array where we display the path that is selected
>>> image = np.arange(36).reshape((6, 6))
>>> image
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35]])
>>> # Find the path with lowest cost
>>> indices, weight = route_through_array(image, (0, 0), (5, 5))
>>> indices = np.stack(indices, axis=-1)
>>> path = np.zeros_like(image)
>>> path[indices[0], indices[1]] = 1
>>> path
array([[1, 1, 1, 1, 1, 0],
       [0, 0, 0, 0, 0, 1],
       [0, 0, 0, 0, 0, 1],
       [0, 0, 0, 0, 0, 1],
       [0, 0, 0, 0, 0, 1],
       [0, 0, 0, 0, 0, 1]])
```

`skimage.graph.shortest_path(arr, reach=1, axis=-1, output_indexlist=False)`

Find the shortest path through an n-d array from one side to another.

Parameters

arr

[ndarray of float64]

reach

[int, optional] By default (`reach = 1`), the shortest path can only move one row up or down for every step it moves forward (i.e., the path gradient is limited to 1). `reach` defines the number of elements that can be skipped along each non-axis dimension at each step.

axis

[int, optional] The axis along which the path must always move forward (default -1)

output_indexlist

[bool, optional] See return value *p* for explanation.

Returns

p

[iterable of int] For each step along *axis*, the coordinate of the shortest path. If *output_indexlist* is True, then the path is returned as a list of n-d tuples that index into *arr*. If False, then the path is returned as an array listing the coordinates of the path along the non-axis dimensions for each step along the axis dimension. That is, *p.shape == (arr.shape[axis], arr.ndim-1)* except that *p* is squeezed before returning so if *arr.ndim == 2*, then *p.shape == (arr.shape[axis],)*

cost

[float] Cost of path. This is the absolute sum of all the differences along the path.

class `skimage.graph.MCP(costs, offsets=None, fully_connected=True, sampling=None)`

Bases: `object`

A class for finding the minimum cost path through a given n-d costs array.

Given an n-d costs array, this class can be used to find the minimum-cost path through that array from any set of points to any other set of points. Basic usage is to initialize the class and call `find_costs()` with a one or more starting indices (and an optional list of end indices). After that, call `traceback()` one or more times to find the path from any given end-position to the closest starting index. New paths through the same costs array can be found by calling `find_costs()` repeatedly.

The cost of a path is calculated simply as the sum of the values of the *costs* array at each point on the path. The class `MCP_Geometric`, on the other hand, accounts for the fact that diagonal vs. axial moves are of different lengths, and weights the path cost accordingly.

Array elements with infinite or negative costs will simply be ignored, as will paths whose cumulative cost overflows to infinite.

Parameters

costs

[ndarray]

offsets

[iterable, optional] A list of offset tuples: each offset specifies a valid move from a given n-d position. If not provided, offsets corresponding to a singly- or fully-connected n-d neighborhood will be constructed with `make_offsets()`, using the *fully_connected* parameter value.

fully_connected

[bool, optional] If no *offsets* are provided, this determines the connectivity of the generated neighborhood. If true, the path may go along diagonals between elements of the *costs* array; otherwise only axial moves are permitted.

sampling

[tuple, optional] For each dimension, specifies the distance between two cells/voxels. If not given or None, the distance is assumed unit.

Attributes**offsets**

[ndarray] Equivalent to the *offsets* provided to the constructor, or if none were so provided, the offsets created for the requested n-d neighborhood. These are useful for interpreting the *traceback* array returned by the *find_costs()* method.

__init__(costs, offsets=None, fully_connected=True, sampling=None)

See class documentation.

find_costs()

Find the minimum-cost path from the given starting points.

This method finds the minimum-cost path to the specified ending indices from any one of the specified starting indices. If no end positions are given, then the minimum-cost path to every position in the costs array will be found.

Parameters**starts**

[iterable] A list of n-d starting indices (where n is the dimension of the *costs* array). The minimum cost path to the closest/cheapest starting point will be found.

ends

[iterable, optional] A list of n-d ending indices.

find_all_ends

[bool, optional] If ‘True’ (default), the minimum-cost-path to every specified end-position will be found; otherwise the algorithm will stop when a path is found to any end-position. (If no *ends* were specified, then this parameter has no effect.)

Returns**cumulative_costs**

[ndarray] Same shape as the *costs* array; this array records the minimum cost path from the nearest/cheapest starting index to each index considered. (If *ends* were specified, not all elements in the array will necessarily be considered: positions not evaluated will have a cumulative cost of inf. If *find_all_ends* is ‘False’, only one of the specified end-positions will have a finite cumulative cost.)

traceback

[ndarray] Same shape as the *costs* array; this array contains the offset to any given index from its predecessor index. The offset indices index into the *offsets* attribute, which is a array of n-d offsets. In the 2-d case, if *offsets*[traceback[x, y]] is (-1, -1), that means that the predecessor of [x, y] in the minimum cost path to some start position is [x+1, y+1]. Note that if the *offset_index* is -1, then the given index was not considered.

goal_reached()

int goal_reached(int index, float cumcost) This method is called each iteration after popping an index from the heap, before examining the neighbors.

This method can be overloaded to modify the behavior of the MCP algorithm. An example might be to stop the algorithm when a certain cumulative cost is reached, or when the front is a certain distance away from the seed point.

This method should return 1 if the algorithm should not check the current point's neighbors and 2 if the algorithm is now done.

offsets

traceback(end)

Trace a minimum cost path through the pre-calculated traceback array.

This convenience function reconstructs the the minimum cost path to a given end position from one of the starting indices provided to *find_costs()*, which must have been called previously. This function can be called as many times as desired after *find_costs()* has been run.

Parameters

end

[iterable] An n-d index into the *costs* array.

Returns

traceback

[list of n-d tuples] A list of indices into the *costs* array, starting with one of the start positions passed to *find_costs()*, and ending with the given *end* index. These indices specify the minimum-cost path from any given start index to the *end* index. (The total cost of that path can be read out from the *cumulative_costs* array returned by *find_costs()*.)

class skimage.graph.MCP_Connect(costs, offsets=None, fully_connected=True)

Bases: *MCP*

Connect source points using the distance-weighted minimum cost function.

A front is grown from each seed point simultaneously, while the origin of the front is tracked as well. When two fronts meet, *create_connection()* is called. This method must be overloaded to deal with the found edges in a way that is appropriate for the application.

`__init__(*args, **kwargs)`

`create_connection()`

`create_connection id1, id2, pos1, pos2, cost1, cost2)`

Overload this method to keep track of the connections that are found during MCP processing. Note that a connection with the same ids can be found multiple times (but with different positions and costs).

At the time that this method is called, both points are “frozen” and will not be visited again by the MCP algorithm.

Parameters

id1

[int] The seed point id where the first neighbor originated from.

id2

[int] The seed point id where the second neighbor originated from.

pos1

[tuple] The index of of the first neighbor in the connection.

pos2

[tuple] The index of of the second neighbor in the connection.

cost1

[float] The cumulative cost at *pos1*.

cost2

[float] The cumulative costs at *pos2*.

`find_costs()`

Find the minimum-cost path from the given starting points.

This method finds the minimum-cost path to the specified ending indices from any one of the specified starting indices. If no end positions are given, then the minimum-cost path to every position in the costs array will be found.

Parameters

starts

[iterable] A list of n-d starting indices (where n is the dimension of the *costs* array). The minimum cost path to the closest/cheapest starting point will be found.

ends

[iterable, optional] A list of n-d ending indices.

find_all_ends

[bool, optional] If ‘True’ (default), the minimum-cost-path to every specified end-position will be found; otherwise the algorithm will stop when a path is found to any end-position. (If no *ends* were specified, then this parameter has no effect.)

Returns**cumulative_costs**

[ndarray] Same shape as the *costs* array; this array records the minimum cost path from the nearest/cheapest starting index to each index considered. (If *ends* were specified, not all elements in the array will necessarily be considered: positions not evaluated will have a cumulative cost of inf. If *find_all_ends* is ‘False’, only one of the specified end-positions will have a finite cumulative cost.)

traceback

[ndarray] Same shape as the *costs* array; this array contains the offset to any given index from its predecessor index. The offset indices index into the *offsets* attribute, which is a array of n-d offsets. In the 2-d case, if *offsets*[*traceback*[*x*, *y*]] is (-1, -1), that means that the predecessor of [*x*, *y*] in the minimum cost path to some start position is [*x*+1, *y*+1]. Note that if the *offset_index* is -1, then the given index was not considered.

goal_reached()

int goal_reached(int index, float cumcost) This method is called each iteration after popping an index from the heap, before examining the neighbors.

This method can be overloaded to modify the behavior of the MCP algorithm. An example might be to stop the algorithm when a certain cumulative cost is reached, or when the front is a certain distance away from the seed point.

This method should return 1 if the algorithm should not check the current point’s neighbors and 2 if the algorithm is now done.

offsets**traceback(end)**

Trace a minimum cost path through the pre-calculated traceback array.

This convenience function reconstructs the the minimum cost path to a given end position from one of the starting indices provided to *find_costs()*, which must have been called previously. This function can be called as many times as desired after *find_costs()* has been run.

Parameters

end

[iterable] An n-d index into the *costs* array.

Returns

traceback

[list of n-d tuples] A list of indices into the *costs* array, starting with one of the start positions passed to *find_costs()*, and ending with the given *end* index. These indices specify the minimum-cost path from any given start index to the *end* index. (The total cost of that path can be read out from the *cumulative_costs* array returned by *find_costs()*.)

class `skimage.graph.MCP_Flexible(costs, offsets=None, fully_connected=True)`

Bases: *MCP*

Find minimum cost paths through an N-d costs array.

See the documentation for MCP for full details. This class differs from MCP in that several methods can be overloaded (from pure Python) to modify the behavior of the algorithm and/or create custom algorithms based on MCP. Note that *goal_reached* can also be overloaded in the MCP class.

__init__(costs, offsets=None, fully_connected=True, sampling=None)

See class documentation.

examine_neighbor(index, new_index, offset_length)

This method is called once for every pair of neighboring nodes, as soon as both nodes are frozen.

This method can be overloaded to obtain information about neighboring nodes, and/or to modify the behavior of the MCP algorithm. One example is the *MCP_Connect* class, which checks for meeting fronts using this hook.

find_costs()

Find the minimum-cost path from the given starting points.

This method finds the minimum-cost path to the specified ending indices from any one of the specified starting indices. If no end positions are given, then the minimum-cost path to every position in the costs array will be found.

Parameters

starts

[iterable] A list of n-d starting indices (where n is the dimension of the *costs* array). The minimum cost path to the closest/cheapest starting point will be found.

ends

[iterable, optional] A list of n-d ending indices.

find_all_ends

[bool, optional] If ‘True’ (default), the minimum-cost-path to every specified end-position will be found; otherwise the algorithm will stop when a path is found to any end-position. (If no *ends* were specified, then this parameter has no effect.)

Returns

cumulative_costs

[ndarray] Same shape as the *costs* array; this array records the minimum cost path from the nearest/cheapest starting index to each index considered. (If *ends* were specified, not all elements in the array will necessarily be considered: positions not evaluated will have a cumulative cost of inf. If *find_all_ends* is ‘False’, only one of the specified end-positions will have a finite cumulative cost.)

traceback

[ndarray] Same shape as the *costs* array; this array contains the offset to any given index from its predecessor index. The offset indices index into the *offsets* attribute, which is a array of n-d offsets. In the 2-d case, if *offsets*[*traceback*[*x*, *y*]] is (-1, -1), that means that the predecessor of [*x*, *y*] in the minimum cost path to some start position is [*x*+1, *y*+1]. Note that if the *offset_index* is -1, then the given index was not considered.

goal_reached()

int *goal_reached*(int *index*, float *cumcost*) This method is called each iteration after popping an index from the heap, before examining the neighbors.

This method can be overloaded to modify the behavior of the MCP algorithm. An example might be to stop the algorithm when a certain cumulative cost is reached, or when the front is a certain distance away from the seed point.

This method should return 1 if the algorithm should not check the current point’s neighbors and 2 if the algorithm is now done.

offsets

traceback(*end*)

Trace a minimum cost path through the pre-calculated traceback array.

This convenience function reconstructs the the minimum cost path to a given end position from one of the starting indices provided to *find_costs()*, which must have been called previously. This function can be called as many times as desired after *find_costs()* has been run.

Parameters

end

[iterable] An n-d index into the *costs* array.

Returns

traceback

[list of n-d tuples] A list of indices into the *costs* array, starting with one of the start positions passed to `find_costs()`, and ending with the given *end* index. These indices specify the minimum-cost path from any given start index to the *end* index. (The total cost of that path can be read out from the *cumulative_costs* array returned by `find_costs()`.)

travel_cost(*old_cost*, *new_cost*, *offset_length*)

This method calculates the travel cost for going from the current node to the next. The default implementation returns *new_cost*. Overload this method to adapt the behaviour of the algorithm.

update_node(*index*, *new_index*, *offset_length*)

This method is called when a node is updated, right after *new_index* is pushed onto the heap and the traceback map is updated.

This method can be overloaded to keep track of other arrays that are used by a specific implementation of the algorithm. For instance the MCP_Connect class uses it to update an id map.

class `skimage.graph.MCP_Geometric(costs, offsets=None, fully_connected=True)`

Bases: *MCP*

Find distance-weighted minimum cost paths through an n-d costs array.

See the documentation for MCP for full details. This class differs from MCP in that the cost of a path is not simply the sum of the costs along that path.

This class instead assumes that the costs array contains at each position the “cost” of a unit distance of travel through that position. For example, a move (in 2-d) from (1, 1) to (1, 2) is assumed to originate in the center of the pixel (1, 1) and terminate in the center of (1, 2). The entire move is of distance 1, half through (1, 1) and half through (1, 2); thus the cost of that move is $(1/2)*costs[1,1] + (1/2)*costs[1,2]$.

On the other hand, a move from (1, 1) to (2, 2) is along the diagonal and is $\sqrt{2}$ in length. Half of this move is within the pixel (1, 1) and the other half in (2, 2), so the cost of this move is calculated as $(\sqrt{2}/2)*costs[1,1] + (\sqrt{2}/2)*costs[2,2]$.

These calculations don’t make a lot of sense with offsets of magnitude greater than 1. Use the *sampling* argument in order to deal with anisotropic data.

__init__(*costs*, *offsets=None*, *fully_connected=True*, *sampling=None*)

See class documentation.

find_costs()

Find the minimum-cost path from the given starting points.

This method finds the minimum-cost path to the specified ending indices from any one of the specified starting indices. If no end positions are given, then the minimum-cost path to every position in the costs array will be found.

Parameters**starts**

[iterable] A list of n-d starting indices (where n is the dimension of the *costs* array). The minimum cost path to the closest/cheapest starting point will be found.

ends

[iterable, optional] A list of n-d ending indices.

find_all_ends

[bool, optional] If ‘True’ (default), the minimum-cost-path to every specified end-position will be found; otherwise the algorithm will stop when a path is found to any end-position. (If no *ends* were specified, then this parameter has no effect.)

Returns**cumulative_costs**

[ndarray] Same shape as the *costs* array; this array records the minimum cost path from the nearest/cheapest starting index to each index considered. (If *ends* were specified, not all elements in the array will necessarily be considered: positions not evaluated will have a cumulative cost of inf. If *find_all_ends* is ‘False’, only one of the specified end-positions will have a finite cumulative cost.)

traceback

[ndarray] Same shape as the *costs* array; this array contains the offset to any given index from its predecessor index. The offset indices index into the *offsets* attribute, which is a array of n-d offsets. In the 2-d case, if *offsets*[*traceback*[*x*, *y*]] is (-1, -1), that means that the predecessor of [*x*, *y*] in the minimum cost path to some start position is [*x*+1, *y*+1]. Note that if the *offset_index* is -1, then the given index was not considered.

goal_reached()

int *goal_reached*(int *index*, float *cumcost*) This method is called each iteration after popping an index from the heap, before examining the neighbors.

This method can be overloaded to modify the behavior of the MCP algorithm. An example might be to stop the algorithm when a certain cumulative cost is reached, or when the front is a certain distance away from the seed point.

This method should return 1 if the algorithm should not check the current point’s neighbors and 2 if the algorithm is now done.

offsets**traceback(*end*)**

Trace a minimum cost path through the pre-calculated traceback array.

This convenience function reconstructs the the minimum cost path to a given end position from one of the starting indices provided to *find_costs()*, which must have been called previously. This function can be called as many times as desired after *find_costs()* has been run.

Parameters

end

[iterable] An n-d index into the *costs* array.

Returns**traceback**

[list of n-d tuples] A list of indices into the *costs* array, starting with one of the start positions passed to `find_costs()`, and ending with the given *end* index. These indices specify the minimum-cost path from any given start index to the *end* index. (The total cost of that path can be read out from the *cumulative_costs* array returned by `find_costs()`.)

class `skimage.graph.RAG(label_image=None, connectivity=1, data=None, **attr)`

Bases: `Graph`

The Region Adjacency Graph (RAG) of an image, subclasses `networkx.Graph`

Parameters**label_image**

[array of int] An initial segmentation, with each region labeled as a different integer. Every unique value in `label_image` will correspond to a node in the graph.

connectivity

[int in {1, ..., `label_image.ndim`}, optional] The connectivity between pixels in `label_image`. For a 2D image, a connectivity of 1 corresponds to immediate neighbors up, down, left, and right, while a connectivity of 2 also includes diagonal neighbors. See `scipy.ndimage.generate_binary_structure`.

data

[`networkx` Graph specification, optional] Initial or additional edges to pass to the NetworkX Graph constructor. See `networkx.Graph`. Valid edge specifications include edge list (list of tuples), NumPy arrays, and SciPy sparse matrices.

****attr**

[keyword arguments, optional] Additional attributes to add to the graph.

__init__(label_image=None, connectivity=1, data=None, **attr)

Initialize a graph with edges, name, or graph attributes.

Parameters**incoming_graph_data**

[input graph (optional, default: None)] Data to initialize graph. If None (default) an empty

graph is created. The data can be an edge list, or any NetworkX graph object. If the corresponding optional Python packages are installed the data can also be a 2D NumPy array, a SciPy sparse array, or a PyGraphviz graph.

attr

[keyword arguments, optional (default= no attributes)] Attributes to add to graph as key=value pairs.

See also:

convert**Examples**

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G = nx.Graph(name="my graph")
>>> e = [(1, 2), (2, 3), (3, 4)] # list of edges
>>> G = nx.Graph(e)
```

Arbitrary graph attribute pairs (key=value) may be assigned

```
>>> G = nx.Graph(e, day="Friday")
>>> G.graph
{'day': 'Friday'}
```

- *Region Boundary based RAGs*
- *RAG Thresholding*
- *Normalized Cut*
- *Drawing Region Adjacency Graphs (RAGs)*
- *Region Adjacency Graphs*
- *RAG Merging*
- *Hierarchical Merging of Region Boundary RAGs*

add_edge(*u*, *v*, *attr_dict=None*, *attr*)**

Add an edge between *u* and *v* while updating max node id.

See also:

`networkx.Graph.add_edge()`.

add_edges_from(*ebunch_to_add*, *attr*)**

Add all the edges in *ebunch_to_add*.

Parameters

`ebunch_to_add`

[container of edges] Each edge given in the container will be added to the graph. The edges must be given as 2-tuples (u, v) or 3-tuples (u, v, d) where d is a dictionary containing edge data.

`attr`

[keyword arguments, optional] Edge data (or labels or objects) can be assigned using keyword arguments.

See also:

`add_edge`

add a single edge

`add_weighted_edges_from`

convenient way to add weighted edges

Notes

Adding the same edge twice has no effect but any edge data will be updated when each duplicate edge is added.

Edge attributes specified in an ebunch take precedence over attributes specified via keyword arguments.

When adding edges from an iterator over the graph you are changing, a *RuntimeError* can be raised with message: *RuntimeError: dictionary changed size during iteration*. This happens when the graph's underlying dictionary is modified during iteration. To avoid this error, evaluate the iterator into a separate object, e.g. by using `list(iterator_of_edges)`, and pass this object to `G.add_edges_from`.

Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edges_from([(0, 1), (1, 2)]) # using a list of edge tuples
>>> e = zip(range(0, 3), range(1, 4))
>>> G.add_edges_from(e) # Add the path graph 0-1-2-3
```

Associate data to edges

```
>>> G.add_edges_from([(1, 2), (2, 3)], weight=3)
>>> G.add_edges_from([(3, 4), (1, 4)], label="WN2898")
```

Evaluate an iterator over a graph if using it to modify the same graph

```
>>> G = nx.Graph([(1, 2), (2, 3), (3, 4)])
>>> # Grow graph by one new node, adding edges to all existing nodes.
>>> # wrong way - will raise RuntimeError
>>> # G.add_edges_from(((5, n) for n in G.nodes))
```

(continues on next page)

(continued from previous page)

```
>>> # correct way - note that there will be no self-edge for node 5
>>> G.add_edges_from(list((5, n) for n in G.nodes))
```

add_node(*n*, *attr_dict=None*, *attr*)**

Add node *n* while updating the maximum node id.

See also:

`networkx.Graph.add_node()`.

add_nodes_from(*nodes_for_adding*, *attr*)**

Add multiple nodes.

Parameters**nodes_for_adding**

[iterable container] A container of nodes (list, dict, set, etc.). OR A container of (node, attribute dict) tuples. Node attributes are updated using the attribute dict.

attr

[keyword arguments, optional (default= no attributes)] Update attributes for all nodes in nodes. Node attributes specified in nodes as a tuple take precedence over attributes specified via keyword arguments.

See also:

add_node**Notes**

When adding nodes from an iterator over the graph you are changing, a *RuntimeError* can be raised with message: *RuntimeError: dictionary changed size during iteration*. This happens when the graph's underlying dictionary is modified during iteration. To avoid this error, evaluate the iterator into a separate object, e.g. by using `list(iterator_of_nodes)`, and pass this object to `G.add_nodes_from`.

Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_nodes_from("Hello")
>>> K3 = nx.Graph([(0, 1), (1, 2), (2, 0)])
>>> G.add_nodes_from(K3)
>>> sorted(G.nodes(), key=str)
[0, 1, 2, 'H', 'e', 'l', 'o']
```

Use keywords to update specific node attributes for every node.

```
>>> G.add_nodes_from([1, 2], size=10)
>>> G.add_nodes_from([3, 4], weight=0.4)
```

Use (node, attrdict) tuples to update attributes for specific nodes.

```
>>> G.add_nodes_from([(1, dict(size=11)), (2, {"color": "blue"})])
>>> G.nodes[1]["size"]
11
>>> H = nx.Graph()
>>> H.add_nodes_from(G.nodes(data=True))
>>> H.nodes[1]["size"]
11
```

Evaluate an iterator over a graph if using it to modify the same graph

```
>>> G = nx.Graph([(0, 1), (1, 2), (3, 4)])
>>> # wrong way - will raise RuntimeError
>>> # G.add_nodes_from(n + 1 for n in G.nodes)
>>> # correct way
>>> G.add_nodes_from(list(n + 1 for n in G.nodes))
```

add_weighted_edges_from(*ebunch_to_add*, *weight='weight'*, ***attr*)

Add weighted edges in *ebunch_to_add* with specified weight attr

Parameters

ebunch_to_add

[container of edges] Each edge given in the list or container will be added to the graph.
The edges must be given as 3-tuples (u, v, w) where w is a number.

weight

[string, optional (default= ‘weight’)] The attribute name for the edge weights to be added.

attr

[keyword arguments, optional (default= no attributes)] Edge attributes to add/update for all edges.

See also:

add_edge

add a single edge

add_edges_from

add multiple edges

Notes

Adding the same edge twice for Graph/DiGraph simply updates the edge data. For MultiGraph/MultiDiGraph, duplicate edges are stored.

When adding edges from an iterator over the graph you are changing, a *RuntimeError* can be raised with message: *RuntimeError: dictionary changed size during iteration*. This happens when the graph's underlying dictionary is modified during iteration. To avoid this error, evaluate the iterator into a separate object, e.g. by using `list(iterator_of_edges)`, and pass this object to `G.add_weighted_edges_from`.

Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_weighted_edges_from([(0, 1, 3.0), (1, 2, 7.5)])
```

Evaluate an iterator over edges before passing it

```
>>> G = nx.Graph([(1, 2), (2, 3), (3, 4)])
>>> weight = 0.1
>>> # Grow graph by one new node, adding edges to all existing nodes.
>>> # wrong way - will raise RuntimeError
>>> # G.add_weighted_edges_from(((5, n, weight) for n in G.nodes))
>>> # correct way - note that there will be no self-edge for node 5
>>> G.add_weighted_edges_from(list((5, n, weight) for n in G.nodes))
```

`property adj`

Graph adjacency object holding the neighbors of each node.

This object is a read-only dict-like structure with node keys and neighbor-dict values. The neighbor-dict is keyed by neighbor to the edge-data-dict. So `G.adj[3][2]['color'] = 'blue'` sets the color of the edge (3, 2) to “blue”.

Iterating over `G.adj` behaves like a dict. Useful idioms include `for nbr, data in G.adj[n].items():`.

The neighbor information is also provided by subscripting the graph. So `for nbr, foovalue in G[node].data('foo', default=1):` works.

For directed graphs, `G.adj` holds outgoing (successor) info.

`adjacency()`

Returns an iterator over (node, adjacency dict) tuples for all nodes.

For directed graphs, only outgoing neighbors/adjacencies are included.

`Returns`

`adj_iter`

[iterator] An iterator over (node, adjacency dictionary) for all nodes in the graph.

Examples

```
>>> G = nx.path_graph(4) # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> [(n, nbrdict) for n, nbrdict in G.adjacency()]
[(0, {1: {}}), (1, {0: {}}, 2: {})), (2, {1: {}, 3: {}}), (3, {2: {}})]
```

adjlist_inner_dict_factory

alias of `dict`

adjlist_outer_dict_factory

alias of `dict`

clear()

Remove all nodes and edges from the graph.

This also removes the name, and all graph, node, and edge attributes.

Examples

```
>>> G = nx.path_graph(4) # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.clear()
>>> list(G.nodes)
[]
>>> list(G.edges)
[]
```

clear_edges()

Remove all edges from the graph without altering nodes.

Examples

```
>>> G = nx.path_graph(4) # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.clear_edges()
>>> list(G.nodes)
[0, 1, 2, 3]
>>> list(G.edges)
[]
```

copy()

Copy the graph with its max node id.

See also:

`networkx.Graph.copy()`.

property degree

A DegreeView for the Graph as G.degree or G.degree().

The node degree is the number of edges adjacent to the node. The weighted node degree is the sum of the edge weights for edges incident to that node.

This object provides an iterator for (node, degree) as well as lookup for the degree for a single node.

Parameters

nbunch

[single node, container, or all nodes (default= all nodes)] The view will only report edges incident to these nodes.

weight

[string or None, optional (default=None)] The name of an edge attribute that holds the numerical value used as a weight. If None, then each edge has weight 1. The degree is the sum of the edge weights adjacent to the node.

Returns

DegreeView or int

If multiple nodes are requested (the default), returns a *DegreeView* mapping nodes to their degree. If a single node is requested, returns the degree of the node as an integer.

Examples

```
>>> G = nx.path_graph(4) # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.degree[0] # node 0 has degree 1
1
>>> list(G.degree([0, 1, 2]))
[(0, 1), (1, 2), (2, 2)]
```

edge_attr_dict_factory

alias of `dict`

edge_subgraph(edges)

Returns the subgraph induced by the specified edges.

The induced subgraph contains each edge in `edges` and each node incident to any one of those edges.

Parameters

edges

[iterable] An iterable of edges in this graph.

Returns**G**

[Graph] An edge-induced subgraph of this graph with the same edge attributes.

Notes

The graph, edge, and node attributes in the returned subgraph view are references to the corresponding attributes in the original graph. The view is read-only.

To create a full graph version of the subgraph with its own copy of the edge or node attributes, use:

```
G.edge_subgraph(edges).copy()
```

Examples

```
>>> G = nx.path_graph(5)
>>> H = G.edge_subgraph([(0, 1), (3, 4)])
>>> list(H.nodes)
[0, 1, 3, 4]
>>> list(H.edges)
[(0, 1), (3, 4)]
```

property edges

An EdgeView of the Graph as G.edges or G.edges().

edges(self, nbunch=None, data=False, default=None)

The EdgeView provides set-like operations on the edge-tuples as well as edge attribute lookup. When called, it also provides an EdgeDataView object which allows control of access to edge attributes (but does not provide set-like operations). Hence, `G.edges[u, v]['color']` provides the value of the color attribute for edge (u, v) while `for (u, v, c) in G.edges.data('color', default='red')`: iterates through all the edges yielding the color attribute with default ‘red’ if no color attribute exists.

Parameters**nbunch**

[single node, container, or all nodes (default= all nodes)] The view will only report edges from these nodes.

data

[string or bool, optional (default=False)] The edge attribute returned in 3-tuple $(u, v, ddict[data])$. If True, return edge attribute dict in 3-tuple $(u, v, ddict)$. If False, return 2-tuple (u, v) .

default

[value, optional (default=None)] Value used for edges that don't have the requested attribute. Only relevant if data is not True or False.

Returns

edges

[EdgeView] A view of edge attributes, usually it iterates over (u, v) or (u, v, d) tuples of edges, but can also be used for attribute lookup as `edges[u, v]['foo']`.

Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored. For directed graphs this returns the out-edges.

Examples

```
>>> G = nx.path_graph(3) # or MultiGraph, etc
>>> G.add_edge(2, 3, weight=5)
>>> [e for e in G.edges]
[(0, 1), (1, 2), (2, 3)]
>>> G.edges.data() # default data is {} (empty dict)
EdgeDataView([(0, 1, {}), (1, 2, {}), (2, 3, {'weight': 5})])
>>> G.edges.data("weight", default=1)
EdgeDataView([(0, 1, 1), (1, 2, 1), (2, 3, 5)])
>>> G.edges([0, 3]) # only edges from these nodes
EdgeDataView([(0, 1), (3, 2)])
>>> G.edges(0) # only edges from node 0
EdgeDataView([(0, 1)])
```

`fresh_copy()`

Return a fresh copy graph with the same data structure.

A fresh copy has no nodes, edges or graph attributes. It is the same data structure as the current graph. This method is typically used to create an empty version of the graph.

This is required when subclassing Graph with networkx v2 and does not cause problems for v1. Here is more detail from the network migrating from 1.x to 2.x document:

With the new GraphViews (SubGraph, ReversedGraph, etc) you can't assume that ``G.__class__()`` will create a new instance of the same graph type as ``G``. In fact, the call signature for ``__class__`` differs depending on whether ``G`` is a view or a base class. For v2.x you should use ``G.fresh_copy()`` to create a null graph of the correct type---ready to fill with nodes and edges.

`get_edge_data(u, v, default=None)`

Returns the attribute dictionary associated with edge (u, v).

This is identical to `G[u][v]` except the default is returned instead of an exception if the edge doesn't exist.

Parameters**u, v**

[nodes]

default: any Python object (default=None)

Value to return if the edge (u, v) is not found.

Returns**edge_dict**

[dictionary] The edge attribute dictionary.

Examples

```
>>> G = nx.path_graph(4) # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G[0][1]
{}
```

Warning: Assigning to $G[u]/v$ is not permitted. But it is safe to assign attributes $G[u]/v/['foo']$

```
>>> G[0][1]["weight"] = 7
>>> G[0][1]["weight"]
7
>>> G[1][0]["weight"]
7
```

```
>>> G = nx.path_graph(4) # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.get_edge_data(0, 1) # default edge data is {}
{}
>>> e = (0, 1)
>>> G.get_edge_data(*e) # tuple form
{}
>>> G.get_edge_data("a", "b", default=0) # edge not in graph, return 0
0
```

graph_attr_dict_factoryalias of `dict`**has_edge(*u, v*)**

Returns True if the edge (u, v) is in the graph.

This is the same as v in $G[u]$ without KeyError exceptions.**Parameters**

u, v

[nodes] Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.

Returns**edge_ind**

[bool] True if edge is in the graph, False otherwise.

Examples

```
>>> G = nx.path_graph(4) # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.has_edge(0, 1) # using two nodes
True
>>> e = (0, 1)
>>> G.has_edge(*e) # e is a 2-tuple (u, v)
True
>>> e = (0, 1, {"weight": 7})
>>> G.has_edge(*e[:2]) # e is a 3-tuple (u, v, data_dictionary)
True
```

The following syntax are equivalent:

```
>>> G.has_edge(0, 1)
True
>>> 1 in G[0] # though this gives KeyError if 0 not in G
True
```

has_node(n)

Returns True if the graph contains the node n.

Identical to *n* in *G*

Parameters**n**

[node]

Examples

```
>>> G = nx.path_graph(3) # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.has_node(0)
True
```

It is more readable and simpler to use

```
>>> 0 in G
True
```

`is_directed()`

Returns True if graph is directed, False otherwise.

`is_multigraph()`

Returns True if graph is a multigraph, False otherwise.

```
merge_nodes(src, dst, weight_func=<function min_weight>, in_place=True, extra_arguments=None,
extra_keywords=None)
```

Merge node *src* and *dst*.

The new combined node is adjacent to all the neighbors of *src* and *dst*. *weight_func* is called to decide the weight of edges incident on the new node.

Parameters

`src, dst`

[int] Nodes to be merged.

`weight_func`

[callable, optional] Function to decide the attributes of edges incident on the new node. For each neighbor *n* for *src* and *dst*, *weight_func* will be called as follows: *weight_func(src, dst, n, *extra_arguments, **extra_keywords)*. *src*, *dst* and *n* are IDs of vertices in the RAG object which is in turn a subclass of `networkx.Graph`. It is expected to return a dict of attributes of the resulting edge.

`in_place`

[bool, optional] If set to *True*, the merged node has the id *dst*, else merged node has a new id which is returned.

`extra_arguments`

[sequence, optional] The sequence of extra positional arguments passed to *weight_func*.

`extra_keywords`

[dictionary, optional] The dict of keyword arguments passed to the *weight_func*.

Returns

id

[int] The id of the new node.

Notes

If *in_place* is *False* the resulting node has a new id, rather than *dst*.

property name

String identifier of the graph.

This graph attribute appears in the attribute dict G.graph keyed by the string “*name*”. as well as an attribute (technically a property) *G.name*. This is entirely user controlled.

nbunch_iter(nbunch=None)

Returns an iterator over nodes contained in nbunch that are also in the graph.

The nodes in nbunch are checked for membership in the graph and if not are silently ignored.

Parameters**nbunch**

[single node, container, or all nodes (default= all nodes)] The view will only report edges incident to these nodes.

Returns**niter**

[iterator] An iterator over nodes in nbunch that are also in the graph. If nbunch is None, iterate over all nodes in the graph.

Raises**NetworkXError**

If nbunch is not a node or sequence of nodes. If a node in nbunch is not hashable.

See also:

[Graph.__iter__](#)

Notes

When nbunch is an iterator, the returned iterator yields values directly from nbunch, becoming exhausted when nbunch is exhausted.

To test whether nbunch is a single node, one can use “if nbunch in self:”, even after processing with this routine.

If nbunch is not a node or a (possibly empty) sequence/iterator or None, a `NetworkXError` is raised. Also, if any object in nbunch is not hashable, a `NetworkXError` is raised.

`neighbors(n)`

Returns an iterator over all neighbors of node n.

This is identical to `iter(G[n])`

Parameters

`n`

[node] A node in the graph

Returns

`neighbors`

[iterator] An iterator over all neighbors of node n

Raises

`NetworkXError`

If the node n is not in the graph.

Notes

Alternate ways to access the neighbors are `G.adj[n]` or `G[n]`:

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge("a", "b", weight=7)
>>> G["a"]
AtlasView({'b': {'weight': 7}})
>>> G = nx.path_graph(4)
>>> [n for n in G[0]]
[1]
```

Examples

```
>>> G = nx.path_graph(4) # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> [n for n in G.neighbors(0)]
[1]
```

`next_id()`

Returns the *id* for the new node to be inserted.

The current implementation returns one more than the maximum *id*.

Returns

`id`

[int] The *id* of the new node to be inserted.

`node_attr_dict_factory`

alias of `dict`

`node_dict_factory`

alias of `dict`

`property nodes`

A NodeView of the Graph as `G.nodes` or `G.nodes()`.

Can be used as `G.nodes` for data lookup and for set-like operations. Can also be used as `G.nodes(data='color', default=None)` to return a NodeDataView which reports specific node data but no set operations. It presents a dict-like interface as well with `G.nodes.items()` iterating over `(node, nodedata)` 2-tuples and `G.nodes[3]['foo']` providing the value of the `foo` attribute for node 3. In addition, a view `G.nodes.data('foo')` provides a dict-like interface to the `foo` attribute of each node. `G.nodes.data('foo', default=1)` provides a default for nodes that do not have attribute `foo`.

Parameters

`data`

[string or bool, optional (default=False)] The node attribute returned in 2-tuple (n, ddict[data]). If True, return entire node attribute dict as (n, ddict). If False, return just the nodes n.

`default`

[value, optional (default=None)] Value used for nodes that don't have the requested attribute. Only relevant if data is not True or False.

Returns

NodeView

Allows set-like operations over the nodes as well as node attribute dict lookup and calling to get a NodeDataView. A NodeDataView iterates over (n, data) and has no set operations. A NodeView iterates over n and includes set operations.

When called, if data is False, an iterator over nodes. Otherwise an iterator of 2-tuples (node, attribute value) where the attribute is specified in *data*. If data is True then the attribute becomes the entire data dictionary.

Notes

If your node data is not needed, it is simpler and equivalent to use the expression `for n in G`, or `list(G)`.

Examples

There are two simple ways of getting a list of all nodes in the graph:

```
>>> G = nx.path_graph(3)
>>> list(G.nodes)
[0, 1, 2]
>>> list(G)
[0, 1, 2]
```

To get the node data along with the nodes:

```
>>> G.add_node(1, time="5pm")
>>> G.nodes[0]["foo"] = "bar"
>>> list(G.nodes(data=True))
[(0, {'foo': 'bar'}), (1, {'time': '5pm'}), (2, {})]
>>> list(G.nodes.data())
[(0, {'foo': 'bar'}), (1, {'time': '5pm'}), (2, {})]
```

```
>>> list(G.nodes(data="foo"))
[(0, 'bar'), (1, None), (2, None)]
>>> list(G.nodes.data("foo"))
[(0, 'bar'), (1, None), (2, None)]
```

```
>>> list(G.nodes(data="time"))
[(0, None), (1, '5pm'), (2, None)]
>>> list(G.nodes.data("time"))
[(0, None), (1, '5pm'), (2, None)]
```

```
>>> list(G.nodes(data="time", default="Not Available"))
[(0, 'Not Available'), (1, '5pm'), (2, 'Not Available')]
>>> list(G.nodes.data("time", default="Not Available"))
[(0, 'Not Available'), (1, '5pm'), (2, 'Not Available')]
```

If some of your nodes have an attribute and the rest are assumed to have a default attribute value you can create a dictionary from node/attribute pairs using the *default* keyword argument to guarantee the value is never None:

```
>>> G = nx.Graph()
>>> G.add_node(0)
>>> G.add_node(1, weight=2)
>>> G.add_node(2, weight=3)
>>> dict(G.nodes(data="weight", default=1))
{0: 1, 1: 2, 2: 3}
```

number_of_edges(*u=None, v=None*)

Returns the number of edges between two nodes.

Parameters**u, v**

[nodes, optional (default=all edges)] If *u* and *v* are specified, return the number of edges between *u* and *v*. Otherwise return the total number of all edges.

Returns**nedges**

[int] The number of edges in the graph. If nodes *u* and *v* are specified return the number of edges between those nodes. If the graph is directed, this only returns the number of edges from *u* to *v*.

See also:**size****Examples**

For undirected graphs, this method counts the total number of edges in the graph:

```
>>> G = nx.path_graph(4)
>>> G.number_of_edges()
3
```

If you specify two nodes, this counts the total number of edges joining the two nodes:

```
>>> G.number_of_edges(0, 1)
1
```

For directed graphs, this method can count the total number of directed edges from *u* to *v*:

```
>>> G = nx.DiGraph()
>>> G.add_edge(0, 1)
>>> G.add_edge(1, 0)
```

(continues on next page)

(continued from previous page)

```
>>> G.number_of_edges(0, 1)
1
```

number_of_nodes()

Returns the number of nodes in the graph.

Returns

nnodes

[int] The number of nodes in the graph.

See also:

order

identical method

__len__

identical method

Examples

```
>>> G = nx.path_graph(3) # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.number_of_nodes()
3
```

order()

Returns the number of nodes in the graph.

Returns

nnodes

[int] The number of nodes in the graph.

See also:

number_of_nodes

identical method

__len__

identical method

Examples

```
>>> G = nx.path_graph(3) # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.order()
3
```

remove_edge(*u*, *v*)

Remove the edge between *u* and *v*.

Parameters

u, v

[nodes] Remove the edge between nodes *u* and *v*.

Raises

NetworkXError

If there is not an edge between *u* and *v*.

See also:

[*remove_edges_from*](#)

remove a collection of edges

Examples

```
>>> G = nx.path_graph(4) # or DiGraph, etc
>>> G.remove_edge(0, 1)
>>> e = (1, 2)
>>> G.remove_edge(*e) # unpacks e from an edge tuple
>>> e = (2, 3, {"weight": 7}) # an edge with attribute data
>>> G.remove_edge(*e[:2]) # select first part of edge tuple
```

remove_edges_from(*ebunch*)

Remove all edges specified in *ebunch*.

Parameters

ebunch: list or container of edge tuples

Each edge given in the list or container will be removed from the graph. The edges can be:

- 2-tuples (*u*, *v*) edge between *u* and *v*.
- 3-tuples (*u*, *v*, *k*) where *k* is ignored.

See also:

`remove_edge`

remove a single edge

Notes

Will fail silently if an edge in ebunch is not in the graph.

Examples

```
>>> G = nx.path_graph(4) # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> ebunch = [(1, 2), (2, 3)]
>>> G.remove_edges_from(ebunch)
```

`remove_node(n)`

Remove node n.

Removes the node n and all adjacent edges. Attempting to remove a non-existent node will raise an exception.

Parameters

n

[node] A node in the graph

Raises

NetworkXError

If n is not in the graph.

See also:

`remove_nodes_from`

Examples

```
>>> G = nx.path_graph(3) # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> list(G.edges)
[(0, 1), (1, 2)]
>>> G.remove_node(1)
>>> list(G.edges)
[]
```

`remove_nodes_from(nodes)`

Remove multiple nodes.

Parameters

`nodes`

[iterable container] A container of nodes (list, dict, set, etc.). If a node in the container is not in the graph it is silently ignored.

See also:

`remove_node`

Notes

When removing nodes from an iterator over the graph you are changing, a *RuntimeError* will be raised with message: *RuntimeError: dictionary changed size during iteration*. This happens when the graph's underlying dictionary is modified during iteration. To avoid this error, evaluate the iterator into a separate object, e.g. by using `list(iterator_of_nodes)`, and pass this object to `G.remove_nodes_from`.

Examples

```
>>> G = nx.path_graph(3) # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> e = list(G.nodes)
>>> e
[0, 1, 2]
>>> G.remove_nodes_from(e)
>>> list(G.nodes)
[]
```

Evaluate an iterator over a graph if using it to modify the same graph

```
>>> G = nx.Graph([(0, 1), (1, 2), (3, 4)])
>>> # this command will fail, as the graph's dict is modified during iteration
>>> # G.remove_nodes_from(n for n in G.nodes if n < 2)
>>> # this command will work, since the dictionary underlying graph is not_
>>> ~modified
>>> G.remove_nodes_from(list(n for n in G.nodes if n < 2))
```

size(*weight=None*)

Returns the number of edges or total of all edge weights.

Parameters**weight**

[string or None, optional (default=None)] The edge attribute that holds the numerical value used as a weight. If None, then each edge has weight 1.

Returns**size**

[numeric] The number of edges or (if weight keyword is provided) the total weight sum.

If weight is None, returns an int. Otherwise a float (or more general numeric if the weights are more general).

See also:**[number_of_edges](#)****Examples**

```
>>> G = nx.path_graph(4) # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.size()
3
```

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge("a", "b", weight=2)
>>> G.add_edge("b", "c", weight=4)
>>> G.size()
2
>>> G.size(weight="weight")
6.0
```

subgraph(*nodes*)

Returns a SubGraph view of the subgraph induced on *nodes*.

The induced subgraph of the graph contains the nodes in *nodes* and the edges between those nodes.

Parameters**nodes**

[list, iterable] A container of nodes which will be iterated through once.

Returns**G**

[SubGraph View] A subgraph view of the graph. The graph structure cannot be changed but node/edge attributes can and are shared with the original graph.

Notes

The graph, edge and node attributes are shared with the original graph. Changes to the graph structure is ruled out by the view, but changes to attributes are reflected in the original graph.

To create a subgraph with its own copy of the edge/node attributes use: G.subgraph(nodes).copy()

For an inplace reduction of a graph to a subgraph you can remove nodes: G.remove_nodes_from([n for n in G if n not in set(nodes)])

Subgraph views are sometimes NOT what you want. In most cases where you want to do more than simply look at the induced edges, it makes more sense to just create the subgraph as its own graph with code like:

```
# Create a subgraph SG based on a (possibly multigraph) G
SG = G.__class__()
SG.add_nodes_from((n, G.nodes[n]) for n in largest_wcc)
if SG.is_multigraph():
    SG.add_edges_from((n, nbr, key, d)
                      for n, nbrs in G.adj.items() if n in largest_wcc
                      for nbr, keydict in nbrs.items() if nbr in largest_wcc
                      for key, d in keydict.items())
else:
    SG.add_edges_from((n, nbr, d)
                      for n, nbrs in G.adj.items() if n in largest_wcc
                      for nbr, d in nbrs.items() if nbr in largest_wcc)
SG.graph.update(G.graph)
```

Examples

```
>>> G = nx.path_graph(4) # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> H = G.subgraph([0, 1, 2])
>>> list(H.edges)
[(0, 1), (1, 2)]
```

to_directed(as_view=False)

Returns a directed representation of the graph.

Returns**G**

[DiGraph] A directed graph with the same name, same nodes, and with each edge (u, v, data) replaced by two directed edges (u, v, data) and (v, u, data).

Notes

This returns a “deepcopy” of the edge, node, and graph attributes which attempts to completely copy all of the data and references.

This is in contrast to the similar `D=DiGraph(G)` which returns a shallow copy of the data.

See the Python copy module for more information on shallow and deep copies, <https://docs.python.org/3/library/copy.html>.

Warning: If you have subclassed Graph to use dict-like objects in the data structure, those changes do not transfer to the DiGraph created by this method.

Examples

```
>>> G = nx.Graph()  # or MultiGraph, etc
>>> G.add_edge(0, 1)
>>> H = G.to_directed()
>>> list(H.edges)
[(0, 1), (1, 0)]
```

If already directed, return a (deep) copy

```
>>> G = nx.DiGraph()  # or MultiDiGraph, etc
>>> G.add_edge(0, 1)
>>> H = G.to_directed()
>>> list(H.edges)
[(0, 1)]
```

`to_directed_class()`

Returns the class to use for empty directed copies.

If you subclass the base classes, use this to designate what directed class to use for `to_directed()` copies.

`to_undirected(as_view=False)`

Returns an undirected copy of the graph.

Parameters

`as_view`

[bool (optional, default=False)] If True return a view of the original undirected graph.

Returns

`G`

[Graph/MultiGraph] A deepcopy of the graph.

See also:

Graph, copy, add_edge, add_edges_from**Notes**

This returns a “deepcopy” of the edge, node, and graph attributes which attempts to completely copy all of the data and references.

This is in contrast to the similar $G = nx.DiGraph(D)$ which returns a shallow copy of the data.

See the Python copy module for more information on shallow and deep copies, <https://docs.python.org/3/library/copy.html>.

Warning: If you have subclassed DiGraph to use dict-like objects in the data structure, those changes do not transfer to the Graph created by this method.

Examples

```
>>> G = nx.path_graph(2) # or MultiGraph, etc
>>> H = G.to_directed()
>>> list(H.edges)
[(0, 1), (1, 0)]
>>> G2 = H.to_undirected()
>>> list(G2.edges)
[(0, 1)]
```

to_undirected_class()

Returns the class to use for empty undirected copies.

If you subclass the base classes, use this to designate what directed class to use for *to_directed()* copies.

update(edges=None, nodes=None)

Update the graph using nodes/edges/graphs as input.

Like dict.update, this method takes a graph as input, adding the graph’s nodes and edges to this graph. It can also take two inputs: edges and nodes. Finally it can take either edges or nodes. To specify only nodes the keyword *nodes* must be used.

The collections of edges and nodes are treated similarly to the *add_edges_from/add_nodes_from* methods. When iterated, they should yield 2-tuples (u, v) or 3-tuples (u, v, datadict).

Parameters**edges**

[Graph object, collection of edges, or None] The first parameter can be a graph or some edges. If it has attributes *nodes* and *edges*, then it is taken to be a Graph-like object and those attributes are used as collections of nodes and edges to be added to the graph. If the first parameter does not have those attributes, it is treated as a collection of edges and added to the graph. If the first argument is None, no edges are added.

nodes

[collection of nodes, or None] The second parameter is treated as a collection of nodes to be added to the graph unless it is None. If *edges* is *None* and *nodes* is *None* an exception is raised. If the first parameter is a Graph, then *nodes* is ignored.

See also:

`add_edges_from`

add multiple edges to a graph

`add_nodes_from`

add multiple nodes to a graph

Notes

If you want to update the graph using an adjacency structure it is straightforward to obtain the edges/nodes from adjacency. The following examples provide common cases, your adjacency may be slightly different and require tweaks of these examples:

```
>>> # dict-of-set/list/tuple
>>> adj = {1: {2, 3}, 2: {1, 3}, 3: {1, 2}}
>>> e = [(u, v) for u, nbrs in adj.items() for v in nbrs]
>>> G.update(edges=e, nodes=adj)
```

```
>>> DG = nx.DiGraph()
>>> # dict-of-dict-of-attribute
>>> adj = {1: {2: 1.3, 3: 0.7}, 2: {1: 1.4}, 3: {1: 0.7}}
>>> e = [
...     (u, v, {"weight": d})
...     for u, nbrs in adj.items()
...     for v, d in nbrs.items()
... ]
>>> DG.update(edges=e, nodes=adj)
```

```
>>> # dict-of-dict-of-dict
>>> adj = {1: {2: {"weight": 1.3}, 3: {"color": 0.7, "weight": 1.2}}}
>>> e = [
...     (u, v, {"weight": d})
...     for u, nbrs in adj.items()
...     for v, d in nbrs.items()
... ]
>>> DG.update(edges=e, nodes=adj)
```

```
>>> # predecessor adjacency (dict-of-set)
>>> pred = {1: {2, 3}, 2: {3}, 3: {}}
>>> e = [(v, u) for u, nbrs in pred.items() for v in nbrs]
```

```
>>> # MultiGraph dict-of-dict-of-dict-of-attribute
>>> MDG = nx.MultiDiGraph()
```

(continues on next page)

(continued from previous page)

```
>>> adj = {
...     1: {2: {0: {"weight": 1.3}, 1: {"weight": 1.2}}}, 
...     3: {2: {0: {"weight": 0.7}}}, 
... }
>>> e = [
...     (u, v, ekey, d)
...     for u, nbrs in adj.items()
...     for v, keydict in nbrs.items()
...     for ekey, d in keydict.items()
... ]
>>> MDG.update(edges=e)
```

Examples

```
>>> G = nx.path_graph(5)
>>> G.update(nx.complete_graph(range(4, 10)))
>>> from itertools import combinations
>>> edges = (
...     (u, v, {"power": u * v})
...     for u, v in combinations(range(10, 20), 2)
...     if u * v < 225
... )
>>> nodes = [1000] # for singleton, use a container
>>> G.update(edges, nodes)
```

1.3.11 skimage.io

Utilities to read and write images in various formats.

The following plug-ins are available:

Plugin	Description
imread	Image reading and writing via imread
fits	FITS image reading via PyFITS
imageio	Image reading via the ImageIO Library
gdal	Image reading via the GDAL Library (www.gdal.org)
simpleitk	Image reading and writing via SimpleITK
pil	Image reading via the Python Imaging Library
matplotlib	Display or save images using Matplotlib
tifffile	Load and save TIFF and TIFF-based images using tifffile.py

<code>skimage.io.call_plugin</code>	Find the appropriate plugin of 'kind' and execute it.
<code>skimage.io.concatenate_images</code>	Concatenate all images in the image collection into an array.
<code>skimage.io.find_available_plugins</code>	List available plugins.
<code>skimage.io.imread</code>	Load an image from file.
<code>skimage.io.imread_collection</code>	Load a collection of images.
<code>skimage.io.imread_collection_wrapper</code>	
<code>skimage.io.imsave</code>	Save an image to file.
<code>skimage.io.imshow</code>	Display an image.
<code>skimage.io.imshow_collection</code>	Display a collection of images.
<code>skimage.io.load_sift</code>	Read SIFT or SURF features from externally generated file.
<code>skimage.io.load_surf</code>	Read SIFT or SURF features from externally generated file.
<code>skimage.io.plugin_info</code>	Return plugin meta-data.
<code>skimage.io.plugin_order</code>	Return the currently preferred plugin order.
<code>skimage.io.pop</code>	Pop an image from the shared image stack.
<code>skimage.io.push</code>	Push an image onto the shared image stack.
<code>skimage.io.reset_plugins</code>	
<code>skimage.io.show</code>	Display pending images.
<code>skimage.io.use_plugin</code>	Set the default plugin for a specified operation.
<code>skimage.io.ImageCollection</code>	Load and manage a collection of image files.
<code>skimage.io.MultiImage</code>	A class containing all frames from multi-frame TIFF images.
<code>skimage.io.collection</code>	Data structures to hold collections of images, with optional caching.
<code>skimage.io.manage_plugins</code>	Handle image reading, writing and plotting plugins.
<code>skimage.io.sift</code>	
<code>skimage.io.util</code>	

`skimage.io.call_plugin(kind, *args, **kwargs)`

Find the appropriate plugin of ‘kind’ and execute it.

Parameters

`kind`

[{‘imshow’, ‘imsave’, ‘imread’, ‘imread_collection’}] Function to look up.

`plugin`

[str, optional] Plugin to load. Defaults to None, in which case the first matching plugin is used.

`*args, **kwargs`

[arguments and keyword arguments] Passed to the plugin function.

`skimage.io.concatenate_images(ic)`

Concatenate all images in the image collection into an array.

Parameters

ic

[an iterable of images] The images to be concatenated.

Returns

array_cat

[ndarray] An array having one more dimension than the images in *ic*.

Raises

ValueError

If images in *ic* don't have identical shapes.

See also:

`ImageCollection.concatenate`, `MultiImage.concatenate`

Notes

`concatenate_images` receives any iterable object containing images, including `ImageCollection` and `MultiImage`, and returns a NumPy array.

`skimage.io.find_available_plugins(loaded=False)`

List available plugins.

Parameters

loaded

[bool] If True, show only those plugins currently loaded. By default, all plugins are shown.

Returns

p

[dict] Dictionary with plugin names as keys and exposed functions as values.

`skimage.io.imread(fname, as_gray=False, plugin=None, **plugin_args)`

Load an image from file.

Parameters

fname

[str or pathlib.Path] Image file name, e.g. `test.jpg` or URL.

as_gray

[bool, optional] If True, convert color images to gray-scale (64-bit floats). Images that are already in gray-scale format are not converted.

plugin

[str, optional] Name of plugin to use. By default, the different plugins are tried (starting with `imageio`) until a suitable candidate is found. If not given and `fname` is a tiff file, the `tifffile` plugin will be used.

Returns

img_array

[ndarray] The different color bands/channels are stored in the third dimension, such that a gray-image is $M \times N$, an RGB-image $M \times N \times 3$ and an RGBA-image $M \times N \times 4$.

Other Parameters

plugin_args

[keywords] Passed to the given plugin.

`skimage.io.imread_collection(load_pattern, conserve_memory=True, plugin=None, **plugin_args)`

Load a collection of images.

Parameters

load_pattern

[str or list] List of objects to load. These are usually filenames, but may vary depending on the currently active plugin. See the docstring for `ImageCollection` for the default behaviour of this parameter.

conserve_memory

[bool, optional] If True, never keep more than one in memory at a specific time. Otherwise, images will be cached once they are loaded.

Returns**ic**

[ImageCollection] Collection of images.

Other Parameters**plugin_args**

[keywords] Passed to the given plugin.

skimage.io.imread_collection_wrapper(imread)**skimage.io.imsave(fname, arr, plugin=None, check_contrast=True, **plugin_args)**

Save an image to file.

Parameters**fname**

[str or pathlib.Path] Target filename.

arr

[ndarray of shape (M,N) or (M,N,3) or (M,N,4)] Image data.

plugin

[str, optional] Name of plugin to use. By default, the different plugins are tried (starting with imageio) until a suitable candidate is found. If not given and fname is a tiff file, the tifffile plugin will be used.

check_contrast

[bool, optional] Check for low contrast and print warning (default: True).

Other Parameters**plugin_args**

[keywords] Passed to the given plugin.

Notes

When saving a JPEG, the compression ratio may be controlled using the `quality` keyword argument which is an integer with values in [1, 100] where 1 is worst quality and smallest file size, and 100 is best quality and largest file size (default 75). This is only available when using the PIL and imageio plugins.

`skimage.io.imshow(arr, plugin=None, **plugin_args)`

Display an image.

Parameters

arr

[ndarray or str] Image data or name of image file.

plugin

[str] Name of plugin to use. By default, the different plugins are tried (starting with imageio) until a suitable candidate is found.

Other Parameters

plugin_args

[keywords] Passed to the given plugin.

- *RAG Merging*
-

`skimage.io.imshow_collection(ic, plugin=None, **plugin_args)`

Display a collection of images.

Parameters

ic

[ImageCollection] Collection to display.

plugin

[str] Name of plugin to use. By default, the different plugins are tried until a suitable candidate is found.

Other Parameters

plugin_args

[keywords] Passed to the given plugin.

skimage.io.load_sift(*f*)

Read SIFT or SURF features from externally generated file.

This routine reads SIFT or SURF files generated by binary utilities from <http://people.cs.ubc.ca/~lowe/keypoints/> and <http://www.vision.ee.ethz.ch/~surf/>.

This routine *does not* generate SIFT/SURF features from an image. These algorithms are patent encumbered. Please use `skimage.feature.CENSURE` instead.

Parameters**filelike**

[string or open file] Input file generated by the feature detectors from <http://people.cs.ubc.ca/~lowe/keypoints/> or <http://www.vision.ee.ethz.ch/~surf/> .

mode

[{‘SIFT’, ‘SURF’}, optional] Kind of descriptor used to generate *filelike*.

Returns**data**

[record array with fields]

•row: int

row position of feature

•column: int

column position of feature

•scale: float

feature scale

•orientation: float

feature orientation

•data: array

feature values

`skimage.io.load_surf(f)`

Read SIFT or SURF features from externally generated file.

This routine reads SIFT or SURF files generated by binary utilities from <http://people.cs.ubc.ca/~lowe/keypoints/> and <http://www.vision.ee.ethz.ch/~surf/>.

This routine *does not* generate SIFT/SURF features from an image. These algorithms are patent encumbered. Please use `skimage.feature.CENSURE` instead.

Parameters

filelike

[string or open file] Input file generated by the feature detectors from <http://people.cs.ubc.ca/~lowe/keypoints/> or <http://www.vision.ee.ethz.ch/~surf/> .

mode

[{‘SIFT’, ‘SURF’}, optional] Kind of descriptor used to generate *filelike*.

Returns

data

[record array with fields]

•row: int

row position of feature

•column: int

column position of feature

•scale: float

feature scale

•orientation: float

feature orientation

•data: array

feature values

`skimage.io.plugin_info(plugin)`

Return plugin meta-data.

Parameters

plugin

[str] Name of plugin.

Returns**m**

[dict] Meta data as specified in plugin .ini.

skimage.io.plugin_order()

Return the currently preferred plugin order.

Returns**p**

[dict] Dictionary of preferred plugin order, with function name as key and plugins (in order of preference) as value.

skimage.io.pop()

Pop an image from the shared image stack.

Returns**img**

[ndarray] Image popped from the stack.

skimage.io.push(*img*)

Push an image onto the shared image stack.

Parameters**img**

[ndarray] Image to push.

`skimage.io.reset_plugins()`

`skimage.io.show()`

Display pending images.

Launch the event loop of the current gui plugin, and display all pending images, queued via `imshow`. This is required when using `imshow` from non-interactive scripts.

A call to `show` will block execution of code until all windows have been closed.

Examples

```
>>> import skimage.io as io

>>> rng = np.random.default_rng()
>>> for i in range(4):
...     ax_im = io.imshow(rng.random((50, 50)))
>>> io.show()
```

- *Region Boundary based RAGs*
 - *RAG Merging*
-

`skimage.io.use_plugin(name, kind=None)`

Set the default plugin for a specified operation. The plugin will be loaded if it hasn't been already.

Parameters

name

[str] Name of plugin.

kind

[{‘imsave’, ‘imread’, ‘imshow’, ‘imread_collection’, ‘imshow_collection’}, optional] Set the plugin for this function. By default, the plugin is set for all functions.

See also:

`available_plugins`

List of available plugins

Examples

To use Matplotlib as the default image reader, you would write:

```
>>> from skimage import io
>>> io.use_plugin('matplotlib', 'imread')
```

To see a list of available plugins run `io.available_plugins`. Note that this lists plugins that are defined, but the full list may not be usable if your system does not have the required libraries installed.

```
class skimage.io.ImageCollection(load_pattern, conserve_memory=True, load_func=None,
                                  **load_func_kwargs)
```

Bases: `object`

Load and manage a collection of image files.

Parameters

`load_pattern`

[str or list of str] Pattern string or list of strings to load. The filename path can be absolute or relative.

`conserve_memory`

[bool, optional] If True, `ImageCollection` does not keep more than one in memory at a specific time. Otherwise, images will be cached once they are loaded.

Other Parameters

`load_func`

[callable] `imread` by default. See notes below.

`**load_func_kwargs`

[dict] Any other keyword arguments are passed to `load_func`.

Notes

Note that files are always returned in alphanumerical order. Also note that slicing returns a new `ImageCollection`, *not* a view into the data.

`ImageCollection` image loading can be customized through `load_func`. For an `ImageCollection` `ic`, `ic[5]` calls `load_func(load_pattern[5])` to load that image.

For example, here is an `ImageCollection` that, for each video provided, loads every second frame:

```
import imageio.v3 as iio3
import itertools

def vidread_step(f, step):
    vid = iio3.imiter(f)
```

(continues on next page)

(continued from previous page)

```

return list(itertools.islice(vid, None, None, step))

video_file = 'no_time_for_that_tiny.gif'
ic = ImageCollection(video_file, load_func=vidread_step, step=2)

ic # is an ImageCollection object of length 1 because 1 video is provided

x = ic[0]
x[5] # the 10th frame of the first video

```

Alternatively, if `load_func` is provided and `load_pattern` is a sequence, an `ImageCollection` of corresponding length will be created, and the individual images will be loaded by calling `load_func` with the matching element of the `load_pattern` as its first argument. In this case, the elements of the sequence do not need to be names of existing files (or strings at all). For example, to create an `ImageCollection` containing 500 images from a video:

```

class FrameReader:
    def __init__(self, f):
        self.f = f
    def __call__(self, index):
        return iio3.imread(self.f, index=index)

ic = ImageCollection(range(500), load_func=FrameReader('movie.mp4'))

ic # is an ImageCollection object of length 500

```

Another use of `load_func` would be to convert all images to uint8:

```

def imread_convert(f):
    return imread(f).astype(np.uint8)

ic = ImageCollection('/tmp/*.*.png', load_func=imread_convert)

```

Examples

```
>>> import imageio.v3 as iio3
>>> import skimage.io as io
```

Where your images are located >>> data_dir = os.path.join(os.path.dirname(__file__), ‘..data’)

```

>>> coll = io.ImageCollection(data_dir + '/chess*.png')
>>> len(coll)
2
>>> coll[0].shape
(200, 200)

```

```
>>> image_col = io.ImageCollection([f'{data_dir}/*.*.png', '{data_dir}/*.*.jpg'])
```

```

>>> class MultiReader:
...     def __init__(self, f):

```

(continues on next page)

(continued from previous page)

```
...     self.f = f
...     def __call__(self, index):
...         return iio3.imread(self.f, index=index)
...
>>> filename = data_dir + '/no_time_for_that_tiny.gif'
>>> ic = io.ImageCollection(range(24), load_func=MultiReader(filename))
>>> len(image_col)
23
>>> isinstance(ic[0], np.ndarray)
True
```

Attributes

files

[list of str] If a pattern string is given for *load_pattern*, this attribute stores the expanded file list. Otherwise, this is equal to *load_pattern*.

__init__(load_pattern, conserve_memory=True, load_func=None, **load_func_kwargs)

Load and manage a collection of images.

concatenate()

Concatenate all images in the collection into an array.

Returns

ar

[np.ndarray] An array having one more dimension than the images in *self*.

Raises

ValueError

If images in the *ImageCollection* don't have identical shapes.

See also:

concatenate_images

property conserve_memory

property files**reload(*n=None*)**

Clear the image cache.

Parameters**n**

[None or int] Clear the cache for this image only. By default, the entire cache is erased.

class skimage.io.MultiImage(*filename, conserve_memory=True, dtype=None, **imread_kwargs*)

Bases: *ImageCollection*

A class containing all frames from multi-frame TIFF images.

Parameters**load_pattern**

[str or list of str] Pattern glob or filenames to load. The path can be absolute or relative.

conserve_memory

[bool, optional] Whether to conserve memory by only caching the frames of a single image.
Default is True.

Notes

MultiImage returns a list of image-data arrays. In this regard, it is very similar to *ImageCollection*, but the two differ in their treatment of multi-frame images.

For a TIFF image containing N frames of size WxH, *MultiImage* stores all frames of that image as a single element of shape (*N, W, H*) in the list. *ImageCollection* instead creates N elements of shape (*W, H*).

For an animated GIF image, *MultiImage* reads only the first frame, while *ImageCollection* reads all frames by default.

Examples

```
# Where your images are located >>> data_dir = os.path.join(os.path.dirname(__file__), '../data')
```

```
>>> multipage_tiff = data_dir + '/multipage.tif'  
>>> multi_img = MultiImage(multipage_tiff)  
>>> len(multi_img) # multi_img contains one element  
1  
>>> multi_img[0].shape # this element is a two-frame image of shape:  
(2, 15, 10)
```

```
>>> image_col = ImageCollection(multipage_tiff)
>>> len(image_col) # image_col contains two elements
2
>>> for frame in image_col:
...     print(frame.shape) # each element is a frame of shape (15, 10)
...
(15, 10)
(15, 10)
```

`__init__(filename, conserve_memory=True, dtype=None, **imread_kwargs)`

Load a multi-img.

`concatenate()`

Concatenate all images in the collection into an array.

Returns

`ar`

[np.ndarray] An array having one more dimension than the images in *self*.

Raises

`ValueError`

If images in the *ImageCollection* don't have identical shapes.

See also:

`concatenate_images`

`property conserve_memory`

`property filename`

`property files`

`reload(n=None)`

Clear the image cache.

Parameters

n

[None or int] Clear the cache for this image only. By default, the entire cache is erased.

1.3.12 skimage.measure

<code>skimage.measure.approximate_polygon</code>	Approximate a polygonal chain with the specified tolerance.
<code>skimage.measure.block_reduce</code>	Downsample image by applying function <i>func</i> to local blocks.
<code>skimage.measure.blur_effect</code>	Compute a metric that indicates the strength of blur in an image (0 for no blur, 1 for maximal blur).
<code>skimage.measure.euler_number</code>	Calculate the Euler characteristic in binary image.
<code>skimage.measure.find_contours</code>	Find iso-valued contours in a 2D array for a given level value.
<code>skimage.measure.grid_points_in_poly</code>	Test whether points on a specified grid are inside a polygon.
<code>skimage.measure.inertia_tensor</code>	Compute the inertia tensor of the input image.
<code>skimage.measure.inertia_tensor_eigvals</code>	Compute the eigenvalues of the inertia tensor of the image.
<code>skimage.measure.intersection_coeff</code>	Fraction of a channel's segmented binary mask that overlaps with a second channel's segmented binary mask.
<code>skimage.measure.label</code>	Label connected regions of an integer array.
<code>skimage.measure.manders_coloc_coeff</code>	Manders' colocalization coefficient between two channels.
<code>skimage.measure.manders_overlap_coeff</code>	Manders' overlap coefficient
<code>skimage.measure.marching_cubes</code>	Marching cubes algorithm to find surfaces in 3d volumetric data.
<code>skimage.measure.mesh_surface_area</code>	Compute surface area, given vertices and triangular faces.
<code>skimage.measure.moments</code>	Calculate all raw image moments up to a certain order.
<code>skimage.measure.moments_central</code>	Calculate all central image moments up to a certain order.
<code>skimage.measure.moments_coords</code>	Calculate all raw image moments up to a certain order.
<code>skimage.measure.moments_coords_central</code>	Calculate all central image moments up to a certain order.
<code>skimage.measure.moments_hu</code>	Calculate Hu's set of image moments (2D-only).
<code>skimage.measure.moments_normalized</code>	Calculate all normalized central image moments up to a certain order.
<code>skimage.measure.pearson_corr_coeff</code>	Calculate Pearson's Correlation Coefficient between pixel intensities in channels.
<code>skimage.measure.perimeter</code>	Calculate total perimeter of all objects in binary image.
<code>skimage.measure.perimeter_crofton</code>	Calculate total Crofton perimeter of all objects in binary image.
<code>skimage.measure.points_in_poly</code>	Test whether points lie inside a polygon.
<code>skimage.measure.profile_line</code>	Return the intensity profile of an image measured along a scan line.

continues on next page

Table 9 – continued from previous page

<code>skimage.measure.ransac</code>	Fit a model to data with the RANSAC (random sample consensus) algorithm.
<code>skimage.measure.regionprops</code>	Measure properties of labeled image regions.
<code>skimage.measure.regionprops_table</code>	Compute image properties and return them as a pandas-compatible table.
<code>skimage.measure.shannon_entropy</code>	Calculate the Shannon entropy of an image.
<code>skimage.measure.subdivide_polygon</code>	Subdivision of polygonal curves using B-Splines.
<code>skimage.measure.CircleModel</code>	Total least squares estimator for 2D circles.
<code>skimage.measure.EllipseModel</code>	Total least squares estimator for 2D ellipses.
<code>skimage.measure.LineModelND</code>	Total least squares estimator for N-dimensional lines.

skimage.measure.approximate_polygon(*coords*, *tolerance*)

Approximate a polygonal chain with the specified tolerance.

It is based on the Douglas-Peucker algorithm.

Note that the approximated polygon is always within the convex hull of the original polygon.

Parameters**coords**

[(N, 2) array] Coordinate array.

tolerance

[float] Maximum distance from original points of polygon to approximated polygonal chain.
If tolerance is 0, the original coordinate array is returned.

Returns**coords**

[(M, 2) array] Approximated polygonal chain where M <= N.

References

[?]

- *Approximate and subdivide polygons*

skimage.measure.block_reduce(*image*, *block_size*=2, *func*=<function sum>, *cval*=0, *func_kwarg*s=None)

Downsample image by applying function *func* to local blocks.

This function is useful for max and mean pooling, for example.

Parameters

image

[ndarray] N-dimensional input image.

block_size

[array_like or int] Array containing down-sampling integer factor along each axis. Default block_size is 2.

func

[callable] Function object which is used to calculate the return value for each local block. This function must implement an axis parameter. Primary functions are `numpy.sum`, `numpy.min`, `numpy.max`, `numpy.mean` and `numpy.median`. See also `func_kwargs`.

cval

[float] Constant padding value if image is not perfectly divisible by the block size.

func_kwargs

[dict] Keyword arguments passed to `func`. Notably useful for passing dtype argument to `np.mean`. Takes dictionary of inputs, e.g.: `func_kwargs={'dtype': np.float16}`.

Returns

image

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

Examples

```
>>> from skimage.measure import block_reduce
>>> image = np.arange(3*3*4).reshape(3, 3, 4)
>>> image
array([[[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]],
      [[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]],
      [[24, 25, 26, 27],
       [28, 29, 30, 31],
       [32, 33, 34, 35]]])
>>> block_reduce(image, block_size=(3, 3, 1), func=np.mean)
array([[[16., 17., 18., 19.]]])
>>> image_max1 = block_reduce(image, block_size=(1, 3, 4), func=np.max)
>>> image_max1
array([[[[11]],
       [[23]]],
```

(continues on next page)

(continued from previous page)

```
[[35]])]
>>> image_max2 = block_reduce(image, block_size=(3, 1, 4), func=np.max)
>>> image_max2
array([[[27],
       [31],
       [35]]])
```

```
skimage.measure.blur_effect(image, h_size=11, channel_axis=None, reduce_func=<function amax>)
```

Compute a metric that indicates the strength of blur in an image (0 for no blur, 1 for maximal blur).

Parameters

image

[ndarray] RGB or grayscale nD image. The input image is converted to grayscale before computing the blur metric.

h_size

[int, optional] Size of the re-blurring filter.

channel_axis

[int or None, optional] If None, the image is assumed to be grayscale (single-channel). Otherwise, this parameter indicates which axis of the array corresponds to color channels.

reduce_func

[callable, optional] Function used to calculate the aggregation of blur metrics along all axes. If set to None, the entire list is returned, where the i-th element is the blur metric along the i-th axis.

Returns

blur

[float (0 to 1) or list of floats] Blur metric: by default, the maximum of blur metrics along all axes.

Notes

h_size must keep the same value in order to compare results between images. Most of the time, the default size (11) is enough. This means that the metric can clearly discriminate blur up to an average 11x11 filter; if blur is higher, the metric still gives good results but its values tend towards an asymptote.

References

[?]

- *Estimate strength of blur*
-

`skimage.measure.euler_number(image, connectivity=None)`

Calculate the Euler characteristic in binary image.

For 2D objects, the Euler number is the number of objects minus the number of holes. For 3D objects, the Euler number is obtained as the number of objects plus the number of holes, minus the number of tunnels, or loops.

Parameters

image: (N, M) ndarray or (N, M, D) ndarray.

2D or 3D images. If image is not binary, all values strictly greater than zero are considered as the object.

connectivity

[int, optional] Maximum number of orthogonal hops to consider a pixel/voxel as a neighbor. Accepted values are ranging from 1 to `input.ndim`. If `None`, a full connectivity of `input.ndim` is used. 4 or 8 neighborhoods are defined for 2D images (connectivity 1 and 2, respectively). 6 or 26 neighborhoods are defined for 3D images, (connectivity 1 and 3, respectively). Connectivity 2 is not defined.

Returns

euler_number

[int] Euler characteristic of the set of all objects in the image.

Notes

The Euler characteristic is an integer number that describes the topology of the set of all objects in the input image. If object is 4-connected, then background is 8-connected, and conversely.

The computation of the Euler characteristic is based on an integral geometry formula in discretized space. In practice, a neighborhood configuration is constructed, and a LUT is applied for each configuration. The coefficients used are the ones of Ohser et al.

It can be useful to compute the Euler characteristic for several connectivities. A large relative difference between results for different connectivities suggests that the image resolution (with respect to the size of objects and holes) is too low.

References

[?], [?]

Examples

- *Euler number*

```
skimage.measure.find_contours(image, level=None, fully_connected='low', positive_orientation='low', *, mask=None)
```

Find iso-valued contours in a 2D array for a given level value.

Uses the “marching squares” method to compute the iso-valued contours of the input 2D array for a particular level value. Array values are linearly interpolated to provide better precision for the output contours.

Parameters

image

[2D ndarray of double] Input image in which to find contours.

level

[float, optional] Value along which to find contours in the array. By default, the level is set to $(\max(\text{image}) + \min(\text{image})) / 2$

Changed in version 0.18: This parameter is now optional.

fully_connected

[str, {'low', 'high'}] Indicates whether array elements below the given level value are to be considered fully-connected (and hence elements above the value will only be face connected), or vice-versa. (See notes below for details.)

positive_orientation

[str, {'low', 'high'}] Indicates whether the output contours will produce positively-oriented polygons around islands of low- or high-valued elements. If 'low' then contours will wind counter-clockwise around elements below the iso-value. Alternately, this means that low-valued elements are always on the left of the contour. (See below for details.)

mask

[2D ndarray of bool, or None] A boolean mask, True where we want to draw contours. Note that NaN values are always excluded from the considered region (mask is set to False wherever array is NaN).

Returns

contours

[list of (n,2)-ndarrays] Each contour is an ndarray of shape (n, 2), consisting of n (row, column) coordinates along the contour.

See also:

[`skimage.measure.marching_cubes`](#)

Notes

The marching squares algorithm is a special case of the marching cubes algorithm [?]. A simple explanation is available here:

<http://users.polytech.unice.fr/~lingrand/MarchingCubes/algo.html>

There is a single ambiguous case in the marching squares algorithm: when a given 2 x 2-element square has two high-valued and two low-valued elements, each pair diagonally adjacent. (Where high- and low-valued is with respect to the contour value sought.) In this case, either the high-valued elements can be ‘connected together’ via a thin isthmus that separates the low-valued elements, or vice-versa. When elements are connected together across a diagonal, they are considered ‘fully connected’ (also known as ‘face+vertex-connected’ or ‘8-connected’). Only high-valued or low-valued elements can be fully-connected, the other set will be considered as ‘face-connected’ or ‘4-connected’. By default, low-valued elements are considered fully-connected; this can be altered with the ‘fully_connected’ parameter.

Output contours are not guaranteed to be closed: contours which intersect the array edge or a masked-off region (either where mask is False or where array is NaN) will be left open. All other contours will be closed. (The closed-ness of a contours can be tested by checking whether the beginning point is the same as the end point.)

Contours are oriented. By default, array values lower than the contour value are to the left of the contour and values greater than the contour value are to the right. This means that contours will wind counter-clockwise (i.e. in ‘positive orientation’) around islands of low-valued pixels. This behavior can be altered with the ‘positive_orientation’ parameter.

The order of the contours in the output list is determined by the position of the smallest x, y (in lexicographical order) coordinate in the contour. This is a side-effect of how the input array is traversed, but can be relied upon.

Warning: Array coordinates/values are assumed to refer to the *center* of the array element. Take a simple example input: `[0, 1]`. The interpolated position of 0.5 in this array is midway between the 0-element (at $x=0$) and the 1-element (at $x=1$), and thus would fall at $x=0.5$.

This means that to find reasonable contours, it is best to find contours midway between the expected “light” and “dark” values. In particular, given a binarized array, *do not* choose to find contours at the low or high value of the array. This will often yield degenerate contours, especially around structures that are a single array element wide. Instead choose a middle value, as above.

References

[?]

Examples

```
>>> a = np.zeros((3, 3))
>>> a[0, 0] = 1
>>> a
array([[1., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
>>> find_contours(a, 0.5)
[array([[0. , 0.5],
       [0.5, 0. ]]])
```

- *Contour finding*
- *Approximate and subdivide polygons*
- *Measure region properties*

`skimage.measure.grid_points_in_poly(shape, verts, binarize=True)`

Test whether points on a specified grid are inside a polygon.

For each (r, c) coordinate on a grid, i.e. $(0, 0)$, $(0, 1)$ etc., test whether that point lies inside a polygon.

You can control the output type with the *binarize* flag. Please refer to its documentation for further details.

Parameters

`shape`

[tuple (M, N)] Shape of the grid.

verts

[(V, 2) array] Specify the V vertices of the polygon, sorted either clockwise or anti-clockwise. The first point may (but does not need to be) duplicated.

binarize: bool

If *True*, the output of the function is a boolean mask. Otherwise, it is a labeled array. The labels are: 0 - outside, 1 - inside, 2 - vertex, 3 - edge.

Returns**mask**

[(M, N) ndarray] If *binarize* is True, the output is a boolean mask. True means the corresponding pixel falls inside the polygon. If *binarize* is False, the output is a labeled array, with pixels having a label between 0 and 3. The meaning of the values is: 0 - outside, 1 - inside, 2 - vertex, 3 - edge.

See also:

[*points_in_poly*](#)

`skimage.measure.inertia_tensor(image, mu=None, *, spacing=None)`

Compute the inertia tensor of the input image.

Parameters**image**

[array] The input image.

mu

[array, optional] The pre-computed central moments of *image*. The inertia tensor computation requires the central moments of the image. If an application requires both the central moments and the inertia tensor (for example, `skimage.measure.regionprops`), then it is more efficient to pre-compute them and pass them to the inertia tensor call.

spacing: tuple of float, shape (ndim,)

The pixel spacing along each axis of the image.

Returns**T**

[array, shape (image.ndim, image.ndim)] The inertia tensor of the input image. $T_{i,j}$ contains the covariance of image intensity along axes i and j .

References

[?], [?]

`skimage.measure.inertia_tensor_eigvals(image, mu=None, T=None, *, spacing=None)`

Compute the eigenvalues of the inertia tensor of the image.

The inertia tensor measures covariance of the image intensity along the image axes. (See `inertia_tensor`.) The relative magnitude of the eigenvalues of the tensor is thus a measure of the elongation of a (bright) object in the image.

Parameters

`image`

[array] The input image.

`mu`

[array, optional] The pre-computed central moments of `image`.

`T`

[array, shape (`image.ndim`, `image.ndim`)] The pre-computed inertia tensor. If `T` is given, `mu` and `image` are ignored.

`spacing: tuple of float, shape (ndim,)`

The pixel spacing along each axis of the image.

Returns

`eigvals`

[list of float, length `image.ndim`] The eigenvalues of the inertia tensor of `image`, in descending order.

Notes

Computing the eigenvalues requires the inertia tensor of the input image. This is much faster if the central moments (`mu`) are provided, or, alternatively, one can provide the inertia tensor (`T`) directly.

`skimage.measure.intersection_coeff(image0_mask, image1_mask, mask=None)`

Fraction of a channel's segmented binary mask that overlaps with a second channel's segmented binary mask.

Parameters

image0_mask

[(M, N) ndarray of dtype bool] Image mask of channel A.

image1_mask

[(M, N) ndarray of dtype bool] Image mask of channel B. Must have same dimensions as *image0_mask*.

mask

[(M, N) ndarray of dtype bool, optional] Only *image0_mask* and *image1_mask* pixels within this region of interest mask are included in the calculation. Must have same dimensions as *image0_mask*.

Returns**Intersection coefficient, float**

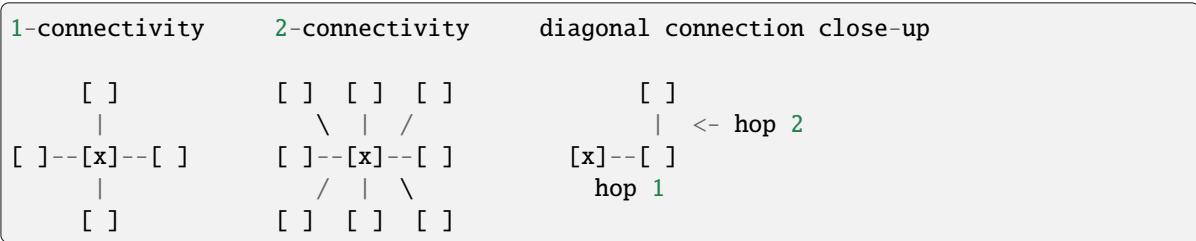
Fraction of *image0_mask* that overlaps with *image1_mask*.

- *Colocalization metrics*
-

skimage.measure.label(label_image, background=None, return_num=False, connectivity=None)

Label connected regions of an integer array.

Two pixels are connected when they are neighbors and have the same value. In 2D, they can be neighbors either in a 1- or 2-connected sense. The value refers to the maximum number of orthogonal hops to consider a pixel/voxel a neighbor:

**Parameters****label_image**

[ndarray of dtype int] Image to label.

background

[int, optional] Consider all pixels with this value as background pixels, and label them as 0. By default, 0-valued pixels are considered as background pixels.

return_num

[bool, optional] Whether to return the number of assigned labels.

connectivity

[int, optional] Maximum number of orthogonal hops to consider a pixel/voxel as a neighbor. Accepted values are ranging from 1 to `input.ndim`. If `None`, a full connectivity of `input.ndim` is used.

Returns**labels**

[ndarray of dtype int] Labeled array, where all connected regions are assigned the same integer value.

num

[int, optional] Number of labels, which equals the maximum label index and is only returned if `return_num` is *True*.

See also:

[regionprops](#)

[regionprops_table](#)

References

[?], [?]

Examples

```
>>> import numpy as np
>>> x = np.eye(3).astype(int)
>>> print(x)
[[1 0 0]
 [0 1 0]
 [0 0 1]]
>>> print(label(x, connectivity=1))
[[1 0 0]
 [0 2 0]
 [0 0 3]]
>>> print(label(x, connectivity=2))
[[1 0 0]
 [0 1 0]
 [0 0 1]]
>>> print(label(x, background=-1))
[[1 2 2]
 [2 1 2]]
```

(continues on next page)

(continued from previous page)

```
[2 2 1]
>>> x = np.array([[1, 0, 0],
...                 [1, 1, 5],
...                 [0, 0, 0]])
>>> print(label(x))
[[1 0 0]
 [1 1 2]
 [0 0 0]]
```

- *Expand segmentation labels without overlap*
- *Label image regions*
- *Find the intersection of two segmentations*
- *Extrema*
- *Explore and visualize region properties with pandas*
- *Measure region properties*
- *Euler number*
- *Evaluating segmentation metrics*
- *Segment human cells (in mitosis)*
- *Track solidification of a metallic alloy*

```
skimage.measure.manders_coloc_coeff(image0, image1_mask, mask=None)
```

Manders' colocalization coefficient between two channels.

Parameters

image0

[(M, N) ndarray] Image of channel A. All pixel values should be non-negative.

image1_mask

[(M, N) ndarray of dtype bool] Binary mask with segmented regions of interest in channel B. Must have same dimensions as *image0*.

mask

[(M, N) ndarray of dtype bool, optional] Only *image0* pixel values within this region of interest mask are included in the calculation. Must have same dimensions as *image0*.

Returns**mcc**

[float] Manders' colocalization coefficient.

Notes

Manders' Colocalization Coefficient (MCC) is the fraction of total intensity of a certain channel (channel A) that is within the segmented region of a second channel (channel B) [?]. It ranges from 0 for no colocalisation to 1 for complete colocalisation. It is also referred to as M1 and M2.

MCC is commonly used to measure the colocalization of a particular protein in a subcellular compartment. Typically a segmentation mask for channel B is generated by setting a threshold that the pixel values must be above to be included in the MCC calculation. In this implementation, the channel B mask is provided as the argument *image1_mask*, allowing the exact segmentation method to be decided by the user beforehand.

The implemented equation is:

$$r = \frac{\sum A_{i,coloc}}{\sum A_i}$$

where

A_i is the value of the i^{th} pixel in *image0* $A_{i,coloc} = A_i$ if $Bmask_i > 0$ $Bmask_i$ is the value of the i^{th} pixel in *mask*

MCC is sensitive to noise, with diffuse signal in the first channel inflating its value. Images should be processed to remove out of focus and background light before the MCC is calculated [?].

References

[?], [?]

- *Colocalization metrics*

`skimage.measure.manders_overlap_coeff(image0, image1, mask=None)`

Manders' overlap coefficient

Parameters**image0**

[(M, N) ndarray] Image of channel A. All pixel values should be non-negative.

image1

[(M, N) ndarray] Image of channel B. All pixel values should be non-negative. Must have same dimensions as *image0*

mask

[(M, N) ndarray of dtype bool, optional] Only *image0* and *image1* pixel values within this region of interest mask are included in the calculation. Must have same dimensions as *image0*.

Returns**moc: float**

Manders' Overlap Coefficient of pixel intensities between the two images.

Notes

Manders' Overlap Coefficient (MOC) is given by the equation [?]:

$$r = \frac{\sum A_i B_i}{\sqrt{\sum A_i^2 \sum B_i^2}}$$

where

A_i is the value of the i^{th} pixel in *image0* B_i is the value of the i^{th} pixel in *image1*

It ranges between 0 for no colocalization and 1 for complete colocalization of all pixels.

MOC does not take into account pixel intensities, just the fraction of pixels that have positive values for both channels[R2208c1a5d6e1-2]_. [?]. Its usefulness has been criticized as it changes in response to differences in both co-occurrence and correlation and so a particular MOC value could indicate a wide range of colocalization patterns [?] [?].

References

[?], [?], [?], [?], [?]

```
skimage.measure.marching_cubes(volume, level=None, *, spacing=(1.0, 1.0, 1.0),
                                gradient_direction='descent', step_size=1, allow_degenerate=True,
                                method='lewiner', mask=None)
```

Marching cubes algorithm to find surfaces in 3d volumetric data.

In contrast with Lorensen et al. approach [?], Lewiner et al. algorithm is faster, resolves ambiguities, and guarantees topologically correct results. Therefore, this algorithm generally a better choice.

Parameters**volume**

[(M, N, P) array] Input data volume to find isosurfaces. Will internally be converted to float32 if necessary.

level

[float, optional] Contour value to search for isosurfaces in *volume*. If not given or None, the average of the min and max of vol is used.

spacing

[length-3 tuple of floats, optional] Voxel spacing in spatial dimensions corresponding to numpy array indexing dimensions (M, N, P) as in *volume*.

gradient_direction

[string, optional] Controls if the mesh was generated from an isosurface with gradient descent toward objects of interest (the default), or the opposite, considering the *left-hand* rule. The two options are: * descent : Object was greater than exterior * ascent : Exterior was greater than object

step_size

[int, optional] Step size in voxels. Default 1. Larger steps yield faster but coarser results. The result will always be topologically correct though.

allow_degenerate

[bool, optional] Whether to allow degenerate (i.e. zero-area) triangles in the end-result. Default True. If False, degenerate triangles are removed, at the cost of making the algorithm slower.

method: {'lewiner', 'lorensen'}, optional

Whether the method of Lewiner et al. or Lorensen et al. will be used.

mask

[(M, N, P) array, optional] Boolean array. The marching cube algorithm will be computed only on True elements. This will save computational time when interfaces are located within certain region of the volume M, N, P-e.g. the top half of the cube-and also allow to compute finite surfaces-i.e. open surfaces that do not end at the border of the cube.

Returns**verts**

[(V, 3) array] Spatial coordinates for V unique mesh vertices. Coordinate order matches input *volume* (M, N, P). If *allow_degenerate* is set to True, then the presence of degenerate triangles in the mesh can make this array have duplicate vertices.

faces

[(F, 3) array] Define triangular faces via referencing vertex indices from *verts*. This algorithm specifically outputs triangles, so each face has exactly three indices.

normals

[(V, 3) array] The normal direction at each vertex, as calculated from the data.

values

[(V,) array] Gives a measure for the maximum value of the data in the local region near each vertex. This can be used by visualization tools to apply a colormap to the mesh.

See also:

`skimage.measure.mesh_surface_area`
`skimage.measure.find_contours`

Notes

The algorithm [?] is an improved version of Chernyaev's Marching Cubes 33 algorithm. It is an efficient algorithm that relies on heavy use of lookup tables to handle the many different cases, keeping the algorithm relatively easy. This implementation is written in Cython, ported from Lewiner's C++ implementation.

To quantify the area of an isosurface generated by this algorithm, pass verts and faces to `skimage.measure.mesh_surface_area`.

Regarding visualization of algorithm output, to contour a volume named `myvolume` about the level 0.0, using the `mayavi` package:

```
>>>
>> from mayavi import mlab
>> verts, faces, _, _ = marching_cubes(myvolume, 0.0)
>> mlab.triangular_mesh([vert[0] for vert in verts],
>>                      [vert[1] for vert in verts],
>>                      [vert[2] for vert in verts],
>>                      faces)
>> mlab.show()
```

Similarly using the `visvis` package:

```
>>>
>> import visvis as vv
>> verts, faces, normals, values = marching_cubes(myvolume, 0.0)
>> vv.mesh(np.fliplr(verts), faces, normals, values)
>> vv.use().Run()
```

To reduce the number of triangles in the mesh for better performance, see this [example](#) using the `mayavi` package.

References

[?], [?]

- *Marching Cubes*

`skimage.measure.mesh_surface_area(verts,faces)`

Compute surface area, given vertices and triangular faces.

Parameters**verts**

[(V, 3) array of floats] Array containing (x, y, z) coordinates for V unique mesh vertices.

faces

[(F, 3) array of ints] List of length-3 lists of integers, referencing vertex coordinates as provided in *verts*.

Returns**area**

[float] Surface area of mesh. Units now [coordinate units] $^{**} 2$.

See also:

[`skimage.measure.marching_cubes`](#)

Notes

The arguments expected by this function are the first two outputs from `skimage.measure.marching_cubes`. For unit correct output, ensure correct *spacing* was passed to `skimage.measure.marching_cubes`.

This algorithm works properly only if the *faces* provided are all triangles.

`skimage.measure.moments(image, order=3, *, spacing=None)`

Calculate all raw image moments up to a certain order.

The following properties can be calculated from raw image moments:

- Area as: `M[0, 0]`.
- Centroid as: `{M[1, 0] / M[0, 0], M[0, 1] / M[0, 0]}`.

Note that raw moments are neither translation, scale nor rotation invariant.

Parameters**image**

[nD double or uint8 array] Rasterized shape as image.

order

[int, optional] Maximum order of moments. Default is 3.

spacing: tuple of float, shape (ndim,)

The pixel spacing along each axis of the image.

Returns

m

[(order + 1, order + 1) array] Raw image moments.

References

[?], [?], [?], [?]

Examples

```
>>> image = np.zeros((20, 20), dtype=np.float64)
>>> image[13:17, 13:17] = 1
>>> M = moments(image)
>>> centroid = (M[1, 0] / M[0, 0], M[0, 1] / M[0, 0])
>>> centroid
(14.5, 14.5)
```

`skimage.measure.moments_central(image, center=None, order=3, *, spacing=None, **kwargs)`

Calculate all central image moments up to a certain order.

The center coordinates (cr, cc) can be calculated from the raw moments as: $\{M[1, 0] / M[0, 0], M[0, 1] / M[0, 0]\}$.

Note that central moments are translation invariant but not scale and rotation invariant.

Parameters

image

[nD double or uint8 array] Rasterized shape as image.

center

[tuple of float, optional] Coordinates of the image centroid. This will be computed if it is not provided.

order

[int, optional] The maximum order of moments computed.

spacing: tuple of float, shape (ndim,)

The pixel spacing along each axis of the image.

Returns**mu**

[(order + 1, order + 1) array] Central image moments.

References

[?], [?], [?], [?]

Examples

```
>>> image = np.zeros((20, 20), dtype=np.float64)
>>> image[13:17, 13:17] = 1
>>> M = moments(image)
>>> centroid = (M[1, 0] / M[0, 0], M[0, 1] / M[0, 0])
>>> moments_central(image, centroid)
array([[16.,  0., 20.,  0.],
       [0.,  0.,  0.,  0.],
       [20.,  0., 25.,  0.],
       [0.,  0.,  0.,  0.]])
```

`skimage.measure.moments_coords(coords, order=3)`

Calculate all raw image moments up to a certain order.

The following properties can be calculated from raw image moments:

- Area as: $M[0, 0]$.
- Centroid as: $\{M[1, 0] / M[0, 0], M[0, 1] / M[0, 0]\}$.

Note that raw moments are neither translation, scale nor rotation invariant.

Parameters**coords**

[(N, D) double or uint8 array] Array of N points that describe an image of D dimensionality in Cartesian space.

order

[int, optional] Maximum order of moments. Default is 3.

Returns

M

[(order + 1, order + 1, ...) array] Raw image moments. (D dimensions)

References

[?]

Examples

```
>>> coords = np.array([[row, col]
...                     for row in range(13, 17)
...                     for col in range(14, 18)], dtype=np.float64)
>>> M = moments_coords(coords)
>>> centroid = (M[1, 0] / M[0, 0], M[0, 1] / M[0, 0])
>>> centroid
(14.5, 15.5)
```

`skimage.measure.moments_coords_central(coords, center=None, order=3)`

Calculate all central image moments up to a certain order.

The following properties can be calculated from raw image moments:

- Area as: $M[0, 0]$.
- Centroid as: $\{M[1, 0] / M[0, 0], M[0, 1] / M[0, 0]\}$.

Note that raw moments are neither translation, scale nor rotation invariant.

Parameters

coords

[(N, D) double or uint8 array] Array of N points that describe an image of D dimensionality in Cartesian space. A tuple of coordinates as returned by `np.nonzero` is also accepted as input.

center

[tuple of float, optional] Coordinates of the image centroid. This will be computed if it is not provided.

order

[int, optional] Maximum order of moments. Default is 3.

Returns

Mc

`[(order + 1, order + 1, ...)]` Central image moments. (D dimensions)

References

[?]

Examples

```
>>> coords = np.array([[row, col]
...                     for row in range(13, 17)
...                     for col in range(14, 18)])
>>> moments_coords_central(coords)
array([[16.,  0., 20.,  0.],
       [ 0.,  0.,  0.,  0.],
       [20.,  0., 25.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

As seen above, for symmetric objects, odd-order moments (columns 1 and 3, rows 1 and 3) are zero when centered on the centroid, or center of mass, of the object (the default). If we break the symmetry by adding a new point, this no longer holds:

```
>>> coords2 = np.concatenate((coords, [[17, 17]]), axis=0)
>>> np.round(moments_coords_central(coords2),
...            decimals=2)
array([[17. ,  0. , 22.12, -2.49],
       [ 0. ,  3.53,  1.73,  7.4 ],
       [25.88,  6.02, 36.63,  8.83],
       [ 4.15, 19.17, 14.8 , 39.6 ]])
```

Image moments and central image moments are equivalent (by definition) when the center is (0, 0):

```
>>> np.allclose(moments_coords(coords),
...               moments_coords_central(coords, (0, 0)))
True
```

skimage.measure.moments_hu(*nu*)

Calculate Hu's set of image moments (2D-only).

Note that this set of moments is proved to be translation, scale and rotation invariant.

Parameters**nu**

`[(M, M) array]` Normalized central image moments, where M must be ≥ 4 .

Returns

nu

[(7,) array] Hu's set of image moments.

References

[?], [?], [?], [?], [?]

Examples

```
>>> image = np.zeros((20, 20), dtype=np.float64)
>>> image[13:17, 13:17] = 0.5
>>> image[10:12, 10:12] = 1
>>> mu = moments_central(image)
>>> nu = moments_normalized(mu)
>>> moments_hu(nu)
array([0.74537037, 0.35116598, 0.10404918, 0.04064421, 0.00264312,
       0.02408546, 0.])
```

`skimage.measure.moments_normalized(mu, order=3, spacing=None)`

Calculate all normalized central image moments up to a certain order.

Note that normalized central moments are translation and scale invariant but not rotation invariant.

Parameters

mu

[(M,[...], M) array] Central image moments, where M must be greater than or equal to order.

order

[int, optional] Maximum order of moments. Default is 3.

Returns

nu

[(order + 1,[...], order + 1) array] Normalized central image moments.

References

[?], [?], [?], [?]

Examples

```
>>> image = np.zeros((20, 20), dtype=np.float64)
>>> image[13:17, 13:17] = 1
>>> m = moments(image)
>>> centroid = (m[0, 1] / m[0, 0], m[1, 0] / m[0, 0])
>>> mu = moments_central(image, centroid)
>>> moments_normalized(mu)
array([[ nan,         nan,  0.078125,  0.        ],
       [ nan,  0.        ,  0.        ,  0.        ],
       [ 0.078125,  0.        ,  0.00610352,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        ]])
```

`skimage.measure.pearson_corr_coeff(image0, image1, mask=None)`

Calculate Pearson's Correlation Coefficient between pixel intensities in channels.

Parameters

image0

[(M, N) ndarray] Image of channel A.

image1

[(M, N) ndarray] Image of channel 2 to be correlated with channel B. Must have same dimensions as *image0*.

mask

[(M, N) ndarray of dtype bool, optional] Only *image0* and *image1* pixels within this region of interest mask are included in the calculation. Must have same dimensions as *image0*.

Returns

pcc

[float] Pearson's correlation coefficient of the pixel intensities between the two images, within the mask if provided.

p-value

[float] Two-tailed p-value.

Notes

Pearson's Correlation Coefficient (PCC) measures the linear correlation between the pixel intensities of the two images. Its value ranges from -1 for perfect linear anti-correlation to +1 for perfect linear correlation. The calculation of the p-value assumes that the intensities of pixels in each input image are normally distributed.

Scipy's implementation of Pearson's correlation coefficient is used. Please refer to it for further information and caveats [?].

$$r = \frac{\sum(A_i - m_{Ai})(B_i - m_{Bi})}{\sqrt{\sum(A_i - m_{Ai})^2} \sqrt{\sum(B_i - m_{Bi})^2}}$$

where

A_i is the value of the i^{th} pixel in *image0* B_i is the value of the i^{th} pixel in *image1*, m_{Ai} is the mean of the pixel values in *image0* m_{Bi} is the mean of the pixel values in *image1*

A low PCC value does not necessarily mean that there is no correlation between the two channel intensities, just that there is no linear correlation. You may wish to plot the pixel intensities of each of the two channels in a 2D scatterplot and use Spearman's rank correlation if a non-linear correlation is visually identified [?]. Also consider if you are interested in correlation or co-occurrence, in which case a method involving segmentation masks (e.g. MCC or intersection coefficient) may be more suitable [?] [?].

Providing the mask of only relevant sections of the image (e.g., cells, or particular cellular compartments) and removing noise is important as the PCC is sensitive to these measures [?] [?].

References

[?], [?], [?], [?]

- *Colocalization metrics*
-

`skimage.measure.perimeter(image, neighborhood=4)`

Calculate total perimeter of all objects in binary image.

Parameters

image

[(N, M) ndarray] 2D binary image.

neighborhood

[4 or 8, optional] Neighborhood connectivity for border pixel determination. It is used to compute the contour. A higher neighborhood widens the border on which the perimeter is computed.

Returns

perimeter

[float] Total perimeter of all objects in binary image.

References

[?]

Examples

```
>>> from skimage import data, util
>>> from skimage.measure import label
>>> # coins image (binary)
>>> img_coins = data.coins() > 110
>>> # total perimeter of all objects in the image
>>> perimeter(img_coins, neighborhood=4)
7796.867...
>>> perimeter(img_coins, neighborhood=8)
8806.268...
```

- *Measure perimeters with different estimators*

`skimage.measure.perimeter_crofton(image, directions=4)`

Calculate total Crofton perimeter of all objects in binary image.

Parameters

`image`

`[(N, M) ndarray]` 2D image. If image is not binary, all values strictly greater than zero are considered as the object.

`directions`

`[2 or 4, optional]` Number of directions used to approximate the Crofton perimeter. By default, 4 is used: it should be more accurate than 2. Computation time is the same in both cases.

Returns

`perimeter`

`[float]` Total perimeter of all objects in binary image.

Notes

This measure is based on Crofton formula [1], which is a measure from integral geometry. It is defined for general curve length evaluation via a double integral along all directions. In a discrete space, 2 or 4 directions give a quite good approximation, 4 being more accurate than 2 for more complex shapes.

Similar to `perimeter()`, this function returns an approximation of the perimeter in continuous space.

References

[?], [?]

Examples

```
>>> from skimage import data, util
>>> from skimage.measure import label
>>> # coins image (binary)
>>> img_coins = data.coins() > 110
>>> # total perimeter of all objects in the image
>>> perimeter_crofton(img_coins, directions=2)
8144.578...
>>> perimeter_crofton(img_coins, directions=4)
7837.077...
```

- Measure perimeters with different estimators
-

`skimage.measure.points_in_poly(points, verts)`

Test whether points lie inside a polygon.

Parameters

points

[(N, 2) array] Input points, (x, y).

verts

[(M, 2) array] Vertices of the polygon, sorted either clockwise or anti-clockwise. The first point may (but does not need to be) duplicated.

Returns

mask

[(N,) array of bool] True if corresponding point is inside the polygon.

See also:

grid_points_in_poly

```
skimage.measure.profile_line(image, src, dst, linewidth=1, order=None, mode='reflect', cval=0.0, *,  
    reduce_func=<function mean>)
```

Return the intensity profile of an image measured along a scan line.

Parameters**image**

[ndarray, shape (M, N[, C])] The image, either grayscale (2D array) or multichannel (3D array, where the final axis contains the channel information).

src

[array_like, shape (2,)] The coordinates of the start point of the scan line.

dst

[array_like, shape (2,)] The coordinates of the end point of the scan line. The destination point is *included* in the profile, in contrast to standard numpy indexing.

linewidth

[int, optional] Width of the scan, perpendicular to the line

order

[int in {0, 1, 2, 3, 4, 5}, optional] The order of the spline interpolation, default is 0 if `image.dtype` is bool and 1 otherwise. The order has to be in the range 0-5. See `skimage.transform.warp` for detail.

mode

[{‘constant’, ‘nearest’, ‘reflect’, ‘mirror’, ‘wrap’}, optional] How to compute any values falling outside of the image.

cval

[float, optional] If `mode` is ‘constant’, what constant value to use outside the image.

reduce_func

[callable, optional] Function used to calculate the aggregation of pixel values perpendicular to the `profile_line` direction when `linewidth > 1`. If set to None the unreduced array will be returned.

Returns**return_value**

[array] The intensity profile along the scan line. The length of the profile is the ceil of the computed length of the scan line.

Examples

```
>>> x = np.array([[1, 1, 1, 2, 2, 2]])
>>> img = np.vstack([np.zeros_like(x), x, x, x, np.zeros_like(x)])
>>> img
array([[0, 0, 0, 0, 0, 0],
       [1, 1, 1, 2, 2, 2],
       [1, 1, 1, 2, 2, 2],
       [1, 1, 1, 2, 2, 2],
       [0, 0, 0, 0, 0, 0]])
>>> profile_line(img, (2, 1), (2, 4))
array([1., 1., 2., 2.])
>>> profile_line(img, (1, 0), (1, 6), cval=4)
array([1., 1., 1., 2., 2., 2.])
```

The destination point is included in the profile, in contrast to standard numpy indexing. For example:

```
>>> profile_line(img, (1, 0), (1, 6)) # The final point is out of bounds
array([1., 1., 1., 2., 2., 2.])
>>> profile_line(img, (1, 0), (1, 5)) # This accesses the full first row
array([1., 1., 1., 2., 2.])
```

For different reduce_func inputs:

```
>>> profile_line(img, (1, 0), (1, 3), linewidth=3, reduce_func=np.mean)
array([0.66666667, 0.66666667, 0.66666667, 1.33333333])
>>> profile_line(img, (1, 0), (1, 3), linewidth=3, reduce_func=np.max)
array([1, 1, 1, 2])
>>> profile_line(img, (1, 0), (1, 3), linewidth=3, reduce_func=np.sum)
array([2, 2, 2, 4])
```

The unreduced array will be returned when *reduce_func* is None or when *reduce_func* acts on each pixel value individually.

```
>>> profile_line(img, (1, 2), (4, 2), linewidth=3, order=0,
...     reduce_func=None)
array([[1, 1, 2],
       [1, 1, 2],
       [1, 1, 2],
       [0, 0, 0]])
>>> profile_line(img, (1, 0), (1, 3), linewidth=3, reduce_func=np.sqrt)
array([[1.          , 1.          , 0.          ],
       [1.          , 1.          , 0.          ],
       [1.          , 1.          , 0.          ],
       [1.41421356, 1.41421356, 0.          ]])
```

```
skimage.measure.ransac(data, model_class, min_samples, residual_threshold, is_data_valid=None,
                       is_model_valid=None, max_trials=100, stop_sample_num=inf,
                       stop_residuals_sum=0, stop_probability=1, rng=None, initial_inliers=None)
```

Fit a model to data with the RANSAC (random sample consensus) algorithm.

RANSAC is an iterative algorithm for the robust estimation of parameters from a subset of inliers from the complete data set. Each iteration performs the following tasks:

1. Select *min_samples* random samples from the original data and check whether the set of data is valid (see *is_data_valid*).
2. Estimate a model to the random subset (*model_cls.estimate(*data[random_subset])*) and check whether the estimated model is valid (see *is_model_valid*).
3. Classify all data as inliers or outliers by calculating the residuals to the estimated model (*model_cls.residuals(*data)*) - all data samples with residuals smaller than the *residual_threshold* are considered as inliers.
4. Save estimated model as best model if number of inlier samples is maximal. In case the current estimated model has the same number of inliers, it is only considered as the best model if it has less sum of residuals.

These steps are performed either a maximum number of times or until one of the special stop criteria are met. The final model is estimated using all inlier samples of the previously determined best model.

Parameters

data

[[list, tuple of] (N, ...) array] Data set to which the model is fitted, where N is the number of data points and the remaining dimension are depending on model requirements. If the model class requires multiple input data arrays (e.g. source and destination coordinates of `skimage.transform.AffineTransform`), they can be optionally passed as tuple or list. Note, that in this case the functions `estimate(*data)`, `residuals(*data)`, `is_model_valid(model, *random_data)` and `is_data_valid(*random_data)` must all take each data array as separate arguments.

model_class

[object] Object with the following object methods:

- `success = estimate(*data)`
- `residuals(*data)`

where *success* indicates whether the model estimation succeeded (*True* or *None* for success, *False* for failure).

min_samples

[int in range (0, N)] The minimum number of data points to fit a model to.

residual_threshold

[float larger than 0] Maximum distance for a data point to be classified as an inlier.

is_data_valid

[function, optional] This function is called with the randomly selected data before the model is fitted to it: `is_data_valid(*random_data)`.

is_model_valid

[function, optional] This function is called with the estimated model and the randomly selected data: `is_model_valid(model, *random_data)`, .

max_trials

[int, optional] Maximum number of iterations for random sample selection.

stop_sample_num

[int, optional] Stop iteration if at least this number of inliers are found.

stop_residuals_sum

[float, optional] Stop iteration if sum of residuals is less than or equal to this threshold.

stop_probability

[float in range [0, 1], optional] RANSAC iteration stops if at least one outlier-free set of the training data is sampled with `probability >= stop_probability`, depending on the current best model's inlier ratio and the number of trials. This requires to generate at least N samples (trials):

$$N \geq \log(1 - \text{probability}) / \log(1 - e^{**m})$$

where the probability (confidence) is typically set to a high value such as 0.99, e is the current fraction of inliers w.r.t. the total number of samples, and m is the `min_samples` value.

rng

[{`numpy.random.Generator`, int}, optional] Pseudo-random number generator. By default, a PCG64 generator is used (see `numpy.random.default_rng()`). If `rng` is an int, it is used to seed the generator.

initial_inliers

[array-like of bool, shape (N,), optional] Initial samples selection for model estimation

Returns

model

[object] Best model with largest consensus set.

inliers

[(N,) array] Boolean mask of inliers classified as True.

Other Parameters

random_state

[DEPRECATED] Deprecated in favor of `rng`.

Deprecated since version 0.21.

References

[?]

Examples

Generate ellipse data without tilt and add noise:

```
>>> t = np.linspace(0, 2 * np.pi, 50)
>>> xc, yc = 20, 30
>>> a, b = 5, 10
>>> x = xc + a * np.cos(t)
>>> y = yc + b * np.sin(t)
>>> data = np.column_stack([x, y])
>>> rng = np.random.default_rng(203560) # do not copy this value
>>> data += rng.normal(size=data.shape)
```

Add some faulty data:

```
>>> data[0] = (100, 100)
>>> data[1] = (110, 120)
>>> data[2] = (120, 130)
>>> data[3] = (140, 130)
```

Estimate ellipse model using all available data:

```
>>> model = EllipseModel()
>>> model.estimate(data)
True
>>> np.round(model.params)
array([ 72.,  75.,  77.,  14.,   1.])
```

Estimate ellipse model using RANSAC:

```
>>> ransac_model, inliers = ransac(data, EllipseModel, 20, 3, max_trials=50)
>>> abs(np.round(ransac_model.params))
array([20., 30., 10.,  6.,  2.])
>>> inliers
array([False, False, False, False, True,  True,  True,  True,
       True,  True,  True,  True,  True,  True,  True,  True,
       True,  True,  True,  True,  True,  True,  True,  True,
       True,  True,  True,  True,  True], dtype=bool)
>>> sum(inliers) > 40
True
```

RANSAC can be used to robustly estimate a geometric transformation. In this section, we also show how to use a proportion of the total samples, rather than an absolute number.

```
>>> from skimage.transform import SimilarityTransform
>>> rng = np.random.default_rng()
>>> src = 100 * rng.random((50, 2))
```

(continues on next page)

(continued from previous page)

```
>>> model0 = SimilarityTransform(scale=0.5, rotation=1,
...                                translation=(10, 20))
>>> dst = model0(src)
>>> dst[0] = (10000, 10000)
>>> dst[1] = (-100, 100)
>>> dst[2] = (50, 50)
>>> ratio = 0.5 # use half of the samples
>>> min_samples = int(ratio * len(src))
>>> model, inliers = ransac(
...     (src, dst),
...     SimilarityTransform,
...     min_samples,
...     10,
...     initial_inliers=np.ones(len(src), dtype=bool),
... )
>>> inliers
array([False, False, False,  True,  True,  True,  True,  True,  True,
       True,  True,  True,  True,  True,  True,  True,  True,
       True,  True,  True,  True,  True,  True,  True,  True,  True,
       True,  True,  True,  True,  True,  True,  True,  True,  True,
       True,  True,  True,  True,  True])
```

- *Fundamental matrix estimation*
- *Robust line model estimation using RANSAC*
- *Robust matching using RANSAC*
- *Assemble images with simple image stitching*

`skimage.measure.regionprops(label_image, intensity_image=None, cache=True, *, extra_properties=None, spacing=None, offset=None)`

Measure properties of labeled image regions.

Parameters

`label_image`

`[(M, N[, P]) ndarray]` Labeled input image. Labels with value 0 are ignored.

Changed in version 0.14.1: Previously, `label_image` was processed by `numpy.squeeze` and so any number of singleton dimensions was allowed. This resulted in inconsistent handling of images with singleton dimensions. To recover the old behaviour, use `regionprops(np.squeeze(label_image), ...)`.

`intensity_image`

`[(M, N[, P][, C]) ndarray, optional]` Intensity (i.e., input) image with same size as labeled

image, plus optionally an extra dimension for multichannel data. Currently, this extra channel dimension, if present, must be the last axis. Default is None.

Changed in version 0.18.0: The ability to provide an extra dimension for channels was added.

cache

[bool, optional] Determine whether to cache calculated properties. The computation is much faster for cached properties, whereas the memory consumption increases.

extra_properties

[Iterable of callables] Add extra property computation functions that are not included with skimage. The name of the property is derived from the function name, the dtype is inferred by calling the function on a small sample. If the name of an extra property clashes with the name of an existing property the extra property will not be visible and a UserWarning is issued. A property computation function must take a region mask as its first argument. If the property requires an intensity image, it must accept the intensity image as the second argument.

spacing: tuple of float, shape (ndim,)

The pixel spacing along each axis of the image.

offset

[array-like of int, shape (*label_image.ndim*,), optional] Coordinates of the origin (“top-left” corner) of the label image. Normally this is ([0,]0, 0), but it might be different if one wants to obtain regionprops of subvolumes within a larger volume.

Returns

properties

[list of RegionProperties] Each item describes one labeled region, and can be accessed using the attributes listed below.

See also:

label

Notes

The following properties can be accessed as attributes or keys:

num_pixels

[int] Number of foreground pixels.

area

[float] Area of the region i.e. number of pixels of the region scaled by pixel-area.

area_bbox

[float] Area of the bounding box i.e. number of pixels of bounding box scaled by pixel-area.

area_convex

[float] Area of the convex hull image, which is the smallest convex polygon that encloses the region.

area_filled

[float] Area of the region with all the holes filled in.

axis_major_length

[float] The length of the major axis of the ellipse that has the same normalized second central moments as the region.

axis_minor_length

[float] The length of the minor axis of the ellipse that has the same normalized second central moments as the region.

bbox

[tuple] Bounding box (`min_row`, `min_col`, `max_row`, `max_col`). Pixels belonging to the bounding box are in the half-open interval `[min_row; max_row)` and `[min_col; max_col)`.

centroid

[array] Centroid coordinate tuple (`row`, `col`).

centroid_local

[array] Centroid coordinate tuple (`row`, `col`), relative to region bounding box.

centroid_weighted

[array] Centroid coordinate tuple (`row`, `col`) weighted with intensity image.

centroid_weighted_local

[array] Centroid coordinate tuple (`row`, `col`), relative to region bounding box, weighted with intensity image.

coords_scaled

[`(N, 2)` ndarray] Coordinate list (`row`, `col`) ``of the region scaled by ``spacing.

coords

[`(N, 2)` ndarray] Coordinate list (`row`, `col`) of the region.

eccentricity

[float] Eccentricity of the ellipse that has the same second-moments as the region. The eccentricity is the ratio of the focal distance (distance between focal points) over the major axis length. The value is in the interval [0, 1]. When it is 0, the ellipse becomes a circle.

equivalent_diameter_area

[float] The diameter of a circle with the same area as the region.

euler_number

[int] Euler characteristic of the set of non-zero pixels. Computed as number of connected components subtracted by number of holes (input.ndim connectivity). In 3D, number of connected components plus number of holes subtracted by number of tunnels.

extent

[float] Ratio of pixels in the region to pixels in the total bounding box. Computed as area / (rows * cols)

feret_diameter_max

[float] Maximum Feret's diameter computed as the longest distance between points around a region's convex hull contour as determined by `find_contours`. [?]

image

[(H, J) ndarray] Sliced binary region image which has the same size as bounding box.

image_convex

[(H, J) ndarray] Binary convex hull image which has the same size as bounding box.

image_filled

[(H, J) ndarray] Binary region image with filled holes which has the same size as bounding box.

image_intensity

[ndarray] Image inside region bounding box.

inertia_tensor

[ndarray] Inertia tensor of the region for the rotation around its mass.

inertia_tensor_eigvals

[tuple] The eigenvalues of the inertia tensor in decreasing order.

intensity_max

[float] Value with the greatest intensity in the region.

intensity_mean

[float] Value with the mean intensity in the region.

intensity_min

[float] Value with the least intensity in the region.

label

[int] The label in the labeled input image.

moments

[(3, 3) ndarray] Spatial moments up to 3rd order:

```
m_ij = sum{ array(row, col) * row^i * col^j }
```

where the sum is over the *row*, *col* coordinates of the region.

moments_central

[(3, 3) ndarray] Central moments (translation invariant) up to 3rd order:

```
mu_ij = sum{ array(row, col) * (row - row_c)^i * (col - col_c)^j }
```

where the sum is over the *row*, *col* coordinates of the region, and *row_c* and *col_c* are the coordinates of the region's centroid.

moments_hu

[tuple] Hu moments (translation, scale and rotation invariant).

moments_normalized

[(3, 3) ndarray] Normalized moments (translation and scale invariant) up to 3rd order:

```
nmu_ij = mu_ij / m_00^[(i+j)/2 + 1]
```

where *m_00* is the zeroth spatial moment.

moments_weighted

[(3, 3) ndarray] Spatial moments of intensity image up to 3rd order:

```
wm_ij = sum{ array(row, col) * row^i * col^j }
```

where the sum is over the *row*, *col* coordinates of the region.

moments_weighted_central

[(3, 3) ndarray] Central moments (translation invariant) of intensity image up to 3rd order:

```
wmu_ij = sum{ array(row, col) * (row - row_c)^i * (col - col_c)^j }
```

where the sum is over the *row*, *col* coordinates of the region, and *row_c* and *col_c* are the coordinates of the region's weighted centroid.

moments_weighted_hu

[tuple] Hu moments (translation, scale and rotation invariant) of intensity image.

moments_weighted_normalized

[(3, 3) ndarray] Normalized moments (translation and scale invariant) of intensity image up to 3rd order:

```
wnu_ij = wmu_ij / wm_00^[(i+j)/2 + 1]
```

where *wm_00* is the zeroth spatial moment (intensity-weighted area).

orientation

[float] Angle between the 0th axis (rows) and the major axis of the ellipse that has the same second moments as the region, ranging from $-pi/2$ to $pi/2$ counter-clockwise.

perimeter

[float] Perimeter of object which approximates the contour as a line through the centers of border pixels using a 4-connectivity.

perimeter_crofton

[float] Perimeter of object approximated by the Crofton formula in 4 directions.

slice

[tuple of slices] A slice to extract the object from the source image.

solidity

[float] Ratio of pixels in the region to pixels of the convex hull image.

Each region also supports iteration, so that you can do:

```
for prop in region:  
    print(prop, region[prop])
```

References

[?], [?], [?], [?], [?]

Examples

```
>>> from skimage import data, util  
>>> from skimage.measure import label, regionprops  
>>> img = util.img_as_ubyte(data.coins()) > 110  
>>> label_img = label(img, connectivity=img.ndim)  
>>> props = regionprops(label_img)  
>>> # centroid of first labeled object  
>>> props[0].centroid  
(22.72987986048314, 81.91228523446583)  
>>> # centroid of first labeled object  
>>> props[0]['centroid']  
(22.72987986048314, 81.91228523446583)
```

Add custom measurements by passing functions as extra_properties

```
>>> from skimage import data, util  
>>> from skimage.measure import label, regionprops  
>>> import numpy as np  
>>> img = util.img_as_ubyte(data.coins()) > 110  
>>> label_img = label(img, connectivity=img.ndim)  
>>> def pixelcount(regionmask):  
...     return np.sum(regionmask)  
>>> props = regionprops(label_img, extra_properties=(pixelcount,))  
>>> props[0].pixelcount  
7741  
>>> props[1]['pixelcount']  
42
```

- *Label image regions*
 - *Measure region properties*
-

```
skimage.measure.regionprops_table(label_image, intensity_image=None, properties=('label', 'bbox'), *, cache=True, separator='-', extra_properties=None, spacing=None)
```

Compute image properties and return them as a pandas-compatible table.

The table is a dictionary mapping column names to value arrays. See Notes section below for details.

New in version 0.16.

Parameters

label_image

[(N, M[, P]) ndarray] Labeled input image. Labels with value 0 are ignored.

intensity_image

[(M, N[, P][, C]) ndarray, optional] Intensity (i.e., input) image with same size as labeled image, plus optionally an extra dimension for multichannel data. Currently, this extra channel dimension, if present, must be the last axis. Default is None.

Changed in version 0.18.0: The ability to provide an extra dimension for channels was added.

properties

[tuple or list of str, optional] Properties that will be included in the resulting dictionary. For a list of available properties, please see `regionprops()`. Users should remember to add “label” to keep track of region identities.

cache

[bool, optional] Determine whether to cache calculated properties. The computation is much faster for cached properties, whereas the memory consumption increases.

separator

[str, optional] For non-scalar properties not listed in OBJECT_COLUMNS, each element will appear in its own column, with the index of that element separated from the property name by this separator. For example, the inertia tensor of a 2D region will appear in four columns: `inertia_tensor-0-0`, `inertia_tensor-0-1`, `inertia_tensor-1-0`, and `inertia_tensor-1-1` (where the separator is -).

Object columns are those that cannot be split in this way because the number of columns would change depending on the object. For example, `image` and `coords`.

extra_properties

[Iterable of callables] Add extra property computation functions that are not included with skimage. The name of the property is derived from the function name, the dtype is inferred by calling the function on a small sample. If the name of an extra property clashes with the name of an existing property the extra property will not be visible and a UserWarning is issued. A property computation function must take a region mask as its first argument. If

the property requires an intensity image, it must accept the intensity image as the second argument.

spacing: tuple of float, shape (ndim,)

The pixel spacing along each axis of the image.

Returns

out_dict

[dict] Dictionary mapping property names to an array of values of that property, one value per region. This dictionary can be used as input to pandas DataFrame to map property names to columns in the frame and regions to rows. If the image has no regions, the arrays will have length 0, but the correct type.

Notes

Each column contains either a scalar property, an object property, or an element in a multidimensional array.

Properties with scalar values for each region, such as “eccentricity”, will appear as a float or int array with that property name as key.

Multidimensional properties *of fixed size* for a given image dimension, such as “centroid” (every centroid will have three elements in a 3D image, no matter the region size), will be split into that many columns, with the name {property_name}{separator}{element_num} (for 1D properties), {property_name}{separator}{elem_num0}{separator}{elem_num1} (for 2D properties), and so on.

For multidimensional properties that don’t have a fixed size, such as “image” (the image of a region varies in size depending on the region size), an object array will be used, with the corresponding property name as the key.

Examples

```
>>> from skimage import data, util, measure
>>> image = data.coins()
>>> label_image = measure.label(image > 110, connectivity=image.ndim)
>>> props = measure.regionprops_table(label_image, image,
...                                         properties=['label',
...                                         'inertia_tensor',
...                                         'inertia_tensor_eigvals'])
>>> props
{'label': array([ 1,  2, ...]), ...
 'inertia_tensor-0-0': array([ 4.012...e+03,   8.51..., ...]), ...
 ...,
 'inertia_tensor_eigvals-1': array([ 2.67...e+02,   2.83..., ...])}
```

The resulting dictionary can be directly passed to pandas, if installed, to obtain a clean DataFrame:

```
>>> import pandas as pd
>>> data = pd.DataFrame(props)
>>> data.head()
   label  inertia_tensor-0-0 ...  inertia_tensor_eigvals-1
0      1          4012.909888 ...            267.065503
```

(continues on next page)

(continued from previous page)

1	2	8.514739	...	2.834806
2	3	0.666667	...	0.000000
3	4	0.000000	...	0.000000
4	5	0.222222	...	0.111111

[5 rows x 7 columns]

If we want to measure a feature that does not come as a built-in property, we can define custom functions and pass them as `extra_properties`. For example, we can create a custom function that measures the intensity quartiles in a region:

```
>>> from skimage import data, util, measure
>>> import numpy as np
>>> def quartiles(regionmask, intensity):
...     return np.percentile(intensity[regionmask], q=(25, 50, 75))
>>>
>>> image = data.coins()
>>> label_image = measure.label(image > 110, connectivity=image.ndim)
>>> props = measure.regionprops_table(label_image, intensity_image=image,
...                                     properties=('label',),
...                                     extra_properties=(quartiles,))
>>> import pandas as pd
>>> pd.DataFrame(props).head()
   label  quartiles-0  quartiles-1  quartiles-2
0      1       117.00       123.0       130.0
1      2       111.25       112.0       114.0
2      3       111.00       111.0       111.0
3      4       111.00       111.5       112.5
4      5       112.50       113.0       114.0
```

- Explore and visualize region properties with `pandas`
- Measure region properties
- Track solidification of a metallic alloy
- Measure fluorescence intensity at the nuclear envelope

`skimage.measure.shannon_entropy(image, base=2)`

Calculate the Shannon entropy of an image.

The Shannon entropy is defined as $S = -\sum(pk * \log(pk))$, where pk are frequency/probability of pixels of value k .

Parameters

`image`

[(N, M) ndarray] Grayscale input image.

base

[float, optional] The logarithmic base to use.

Returns**entropy**

[float]

Notes

The returned value is measured in bits or shannon (Sh) for base=2, natural unit (nat) for base=np.e and hartley (Hart) for base=10.

References

[?], [?]

Examples

```
>>> from skimage import data
>>> from skimage.measure import shannon_entropy
>>> shannon_entropy(data.camera())
7.231695011055706
```

skimage.measure.subdivide_polygon(coords, degree=2, preserve_ends=False)

Subdivision of polygonal curves using B-Splines.

Note that the resulting curve is always within the convex hull of the original polygon. Circular polygons stay closed after subdivision.

Parameters**coords**

[(N, 2) array] Coordinate array.

degree

{1, 2, 3, 4, 5, 6, 7}, optional] Degree of B-Spline. Default is 2.

preserve_ends

[bool, optional] Preserve first and last coordinate of non-circular polygon. Default is False.

Returns

coords

[(M, 2) array] Subdivided coordinate array.

References

[?]

- *Approximate and subdivide polygons*

class skimage.measure.CircleModel

Bases: `BaseModel`

Total least squares estimator for 2D circles.

The functional model of the circle is:

$$r^2 = (x - xc)^2 + (y - yc)^2$$

This estimator minimizes the squared distances from all points to the circle:

$$\min\{ \sum((r - \sqrt{(x_i - xc)^2 + (y_i - yc)^2})^2) \}$$

A minimum number of 3 points is required to solve for the parameters.

Notes

The estimation is carried out using a 2D version of the spherical estimation given in [?].

References

[?]

Examples

```
>>> t = np.linspace(0, 2 * np.pi, 25)
>>> xy = CircleModel().predict_xy(t, params=(2, 3, 4))
>>> model = CircleModel()
>>> model.estimate(xy)
True
>>> tuple(np.round(model.params, 5))
(2.0, 3.0, 4.0)
>>> res = model.residuals(xy)
>>> np.abs(np.round(res, 9))
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0.])
```

Attributes

params

[tuple] Circle model parameters in the following order xc, yc, r .

__init__(self)**estimate(self, data)**

Estimate circle model from data using total least squares.

Parameters**data**

[(N, 2) array] N points with (x, y) coordinates, respectively.

Returns**success**

[bool] True, if model estimation succeeds.

predict_xy(self, t, params=None)

Predict x- and y-coordinates using the estimated model.

Parameters**t**

[array] Angles in circle in radians. Angles start to count from positive x-axis to positive y-axis in a right-handed system.

params

[(3,) array, optional] Optional custom parameter set.

Returns**xy**

[(..., 2) array] Predicted x- and y-coordinates.

residuals(self, data)

Determine residuals of data to model.

For each point the shortest distance to the circle is returned.

Parameters

data

[(N, 2) array] N points with (x, y) coordinates, respectively.

Returns

residuals

[(N,) array] Residual for each data point.

class skimage.measure.EllipseModel

Bases: `BaseModel`

Total least squares estimator for 2D ellipses.

The functional model of the ellipse is:

```
xt = xc + a*cos(theta)*cos(t) - b*sin(theta)*sin(t)
yt = yc + a*sin(theta)*cos(t) + b*cos(theta)*sin(t)
d = sqrt((x - xt)**2 + (y - yt)**2)
```

where (xt, yt) is the closest point on the ellipse to (x, y). Thus d is the shortest distance from the point to the ellipse.

The estimator is based on a least squares minimization. The optimal solution is computed directly, no iterations are required. This leads to a simple, stable and robust fitting method.

The `params` attribute contains the parameters in the following order:

```
xc, yc, a, b, theta
```

Examples

```
>>> xy = EllipseModel().predict_xy(np.linspace(0, 2 * np.pi, 25),
...                                 params=(10, 15, 8, 4, np.deg2rad(30)))
>>> ellipse = EllipseModel()
>>> ellipse.estimate(xy)
True
>>> np.round(ellipse.params, 2)
array([10. , 15. , 8. , 4. , 0.52])
>>> np.round(abs(ellipse.residuals(xy)), 5)
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0.])
```

Attributes

params

[tuple] Ellipse model parameters in the following order xc, yc, a, b, θ .

__init__()**estimate(data)**

Estimate ellipse model from data using total least squares.

Parameters**data**

[(N, 2) array] N points with (x, y) coordinates, respectively.

Returns**success**

[bool] True, if model estimation succeeds.

References

[?]

predict_xy(t, params=None)

Predict x- and y-coordinates using the estimated model.

Parameters**t**

[array] Angles in circle in radians. Angles start to count from positive x-axis to positive y-axis in a right-handed system.

params

[(5,) array, optional] Optional custom parameter set.

Returns**xy**

[(..., 2) array] Predicted x- and y-coordinates.

residuals(*data*)

Determine residuals of data to model.

For each point the shortest distance to the ellipse is returned.

Parameters**data**

[(N, 2) array] N points with (x, y) coordinates, respectively.

Returns**residuals**

[(N,) array] Residual for each data point.

class skimage.measure.LineModelND

Bases: `BaseModel`

Total least squares estimator for N-dimensional lines.

In contrast to ordinary least squares line estimation, this estimator minimizes the orthogonal distances of points to the estimated line.

Lines are defined by a point (origin) and a unit vector (direction) according to the following vector equation:

```
X = origin + lambda * direction
```

Examples

```
>>> x = np.linspace(1, 2, 25)
>>> y = 1.5 * x + 3
>>> lm = LineModelND()
>>> lm.estimate(np.stack([x, y], axis=-1))
True
>>> tuple(np.round(lm.params, 5))
(array([1.5 , 5.25]), array([0.5547 , 0.83205]))
>>> res = lm.residuals(np.stack([x, y], axis=-1))
>>> np.abs(np.round(res, 9))
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0.])
>>> np.round(lm.predict_y(x[:5]), 3)
array([4.5 , 4.562, 4.625, 4.688, 4.75 ])
>>> np.round(lm.predict_x(y[:5]), 3)
array([1.    , 1.042, 1.083, 1.125, 1.167])
```

Attributes

params

[tuple] Line model parameters in the following order *origin, direction*.

__init__()

- Robust line model estimation using RANSAC

estimate(data)

Estimate line model from data.

This minimizes the sum of shortest (orthogonal) distances from the given data points to the estimated line.

Parameters**data**

[(N, dim) array] N points in a space of dimensionality dim >= 2.

Returns**success**

[bool] True, if model estimation succeeds.

predict(x, axis=0, params=None)

Predict intersection of the estimated line model with a hyperplane orthogonal to a given axis.

Parameters**x**

[(n, 1) array] Coordinates along an axis.

axis

[int] Axis orthogonal to the hyperplane intersecting the line.

params

[(2,) array, optional] Optional custom parameter set in the form (*origin, direction*).

Returns**data**

[(n, m) array] Predicted coordinates.

Raises

ValueError

If the line is parallel to the given axis.

predict_x(y, params=None)

Predict x-coordinates for 2D lines using the estimated model.

Alias for:

```
predict(y, axis=1)[:, 0]
```

Parameters

y

[array] y-coordinates.

params

[(2,) array, optional] Optional custom parameter set in the form (*origin*, *direction*).

Returns

x

[array] Predicted x-coordinates.

predict_y(x, params=None)

Predict y-coordinates for 2D lines using the estimated model.

Alias for:

```
predict(x, axis=0)[:, 1]
```

Parameters

x

[array] x-coordinates.

params

[(2,) array, optional] Optional custom parameter set in the form (*origin*, *direction*).

Returns**y**

[array] Predicted y-coordinates.

residuals(*data*, *params=None*)

Determine residuals of data to model.

For each point, the shortest (orthogonal) distance to the line is returned. It is obtained by projecting the data onto the line.

Parameters**data**

[(N, dim) array] N points in a space of dimension dim.

params[(2,) array, optional] Optional custom parameter set in the form (*origin*, *direction*).**Returns****residuals**

[(N,) array] Residual for each data point.

1.3.13 skimage.metrics

<code>skimage.metrics.adapted_rand_error</code>	Compute Adapted Rand error as defined by the SNEMI3D contest.
<code>skimage.metrics.contingency_table</code>	Return the contingency table for all regions in matched segmentations.
<code>skimage.metrics.hausdorff_distance</code>	Calculate the Hausdorff distance between nonzero elements of given images.
<code>skimage.metrics.hausdorff_pair</code>	Returns pair of points that are Hausdorff distance apart between nonzero elements of given images.
<code>skimage.metrics.mean_squared_error</code>	Compute the mean-squared error between two images.
<code>skimage.metrics.normalized_mutual_information</code>	Compute the normalized mutual information (NMI).
<code>skimage.metrics.normalized_root_mse</code>	Compute the normalized root mean-squared error (NRMSE) between two images.
<code>skimage.metrics.peak_signal_noise_ratio</code>	Compute the peak signal to noise ratio (PSNR) for an image.
<code>skimage.metrics.structural_similarity</code>	Compute the mean structural similarity index between two images.
<code>skimage.metrics.variation_of_information</code>	Return symmetric conditional entropies associated with the VI.

```
skimage.metrics.adapted_rand_error(image_true=None, image_test=None, *, table=None,
                                    ignore_labels=(0,), alpha=0.5)
```

Compute Adapted Rand error as defined by the SNEMI3D contest. [?]

Parameters

image_true

[ndarray of int] Ground-truth label image, same shape as im_test.

image_test

[ndarray of int] Test image.

table

[scipy.sparse array in crs format, optional] A contingency table built with skimage.evaluate.contingency_table. If None, it will be computed on the fly.

ignore_labels

[sequence of int, optional] Labels to ignore. Any part of the true image labeled with any of these values will not be counted in the score.

alpha

[float, optional] Relative weight given to precision and recall in the adapted Rand error calculation.

Returns

are

[float] The adapted Rand error.

prec

[float] The adapted Rand precision: this is the number of pairs of pixels that have the same label in the test label image *and* in the true image, divided by the number in the test image.

rec

[float] The adapted Rand recall: this is the number of pairs of pixels that have the same label in the test label image *and* in the true image, divided by the number in the true image.

Notes

Pixels with label 0 in the true segmentation are ignored in the score.

The adapted Rand error is calculated as follows:

$1 - \frac{\sum_{ij} p_{ij}^2}{\alpha \sum_k s_k^2 + (1-\alpha) \sum_k t_k^2}$, where p_{ij} is the probability that a pixel has the same label in the test image *and* in the true image, t_k is the probability that a pixel has label k in the true image, and s_k is the probability that a pixel has label k in the test image.

Default behavior is to weight precision and recall equally in the adapted Rand error calculation. When alpha = 0, adapted Rand error = recall. When alpha = 1, adapted Rand error = precision.

References

[?]

- *Evaluating segmentation metrics*

`skimage.metrics.contingency_table(im_true, im_test, *, ignore_labels=None, normalize=False)`

Return the contingency table for all regions in matched segmentations.

Parameters

im_true

[ndarray of int] Ground-truth label image, same shape as im_test.

im_test

[ndarray of int] Test image.

ignore_labels

[sequence of int, optional] Labels to ignore. Any part of the true image labeled with any of these values will not be counted in the score.

normalize

[bool] Determines if the contingency table is normalized by pixel count.

Returns

cont

[scipy.sparse.csr_matrix] A contingency table. $cont[i, j]$ will equal the number of voxels labeled i in *im_true* and j in *im_test*.

```
skimage.metrics.hausdorff_distance(image0, image1, method='standard')
```

Calculate the Hausdorff distance between nonzero elements of given images.

Parameters

image0, image1

[ndarray] Arrays where True represents a point that is included in a set of points. Both arrays must have the same shape.

method

[{'standard', 'modified'}, optional, default = 'standard'] The method to use for calculating the Hausdorff distance. standard is the standard Hausdorff distance, while modified is the modified Hausdorff distance.

Returns

distance

[float] The Hausdorff distance between coordinates of nonzero pixels in image0 and image1, using the Euclidean distance.

Notes

The Hausdorff distance [?] is the maximum distance between any point on image0 and its nearest point on image1, and vice-versa. The Modified Hausdorff Distance (MHD) has been shown to perform better than the directed Hausdorff Distance (HD) in the following work by Dubuisson et al. [?]. The function calculates forward and backward mean distances and returns the largest of the two.

References

[?], [?]

Examples

```
>>> points_a = (3, 0)
>>> points_b = (6, 0)
>>> shape = (7, 1)
>>> image_a = np.zeros(shape, dtype=bool)
>>> image_b = np.zeros(shape, dtype=bool)
>>> image_a[points_a] = True
>>> image_b[points_b] = True
>>> hausdorff_distance(image_a, image_b)
3.0
```

- *Hausdorff Distance*

skimage.metrics.hausdorff_pair(image0, image1)

Returns pair of points that are Hausdorff distance apart between nonzero elements of given images.

The Hausdorff distance [?] is the maximum distance between any point on `image0` and its nearest point on `image1`, and vice-versa.

Parameters**image0, image1**

[ndarray] Arrays where True represents a point that is included in a set of points. Both arrays must have the same shape.

Returns**point_a, point_b**

[array] A pair of points that have Hausdorff distance between them.

References

[?]

Examples

```
>>> points_a = (3, 0)
>>> points_b = (6, 0)
>>> shape = (7, 1)
>>> image_a = np.zeros(shape, dtype=bool)
>>> image_b = np.zeros(shape, dtype=bool)
>>> image_a[points_a] = True
>>> image_b[points_b] = True
>>> hausdorff_pair(image_a, image_b)
(array([3, 0]), array([6, 0]))
```

- *Hausdorff Distance*
-

skimage.metrics.mean_squared_error(image0, image1)

Compute the mean-squared error between two images.

Parameters

image0, image1

[ndarray] Images. Any dimensionality, must have same shape.

Returns

mse

[float] The mean-squared error (MSE) metric.

Notes

Changed in version 0.16: This function was renamed from `skimage.measure.compare_mse` to `skimage.metrics.mean_squared_error`.

- *Structural similarity index*
 - *Full tutorial on calibrating Denoisers Using J-Invariance*
-

skimage.metrics.normalized_mutual_information(image0, image1, *, bins=100)

Compute the normalized mutual information (NMI).

The normalized mutual information of A and B is given by:

```
.. math::
```

$$Y(A, B) = \frac{H(A) + H(B)}{H(A, B)}$$

where $H(X) := -\sum_{x \in X} x \log x$ is the entropy.

It was proposed to be useful in registering images by Colin Studholme and colleagues [?]. It ranges from 1 (perfectly uncorrelated image values) to 2 (perfectly correlated image values, whether positively or negatively).

Parameters

image0, image1

[ndarray] Images to be compared. The two input images must have the same number of dimensions.

bins

[int or sequence of int, optional] The number of bins along each axis of the joint histogram.

Returns

nmi

[float] The normalized mutual information between the two arrays, computed at the granularity given by `bins`. Higher NMI implies more similar input images.

Raises**ValueError**

If the images don't have the same number of dimensions.

Notes

If the two input images are not the same shape, the smaller image is padded with zeros.

References

[?]

`skimage.metrics.normalized_root_mse(image_true, image_test, *, normalization='euclidean')`

Compute the normalized root mean-squared error (NRMSE) between two images.

Parameters**image_true**

[ndarray] Ground-truth image, same shape as `im_test`.

image_test

[ndarray] Test image.

normalization

[{‘euclidean’, ‘min-max’, ‘mean’}, optional] Controls the normalization method to use in the denominator of the NRMSE. There is no standard method of normalization across the literature [?]. The methods available here are as follows:

- ‘euclidean’ : normalize by the averaged Euclidean norm of `im_true`:

$$\text{NRMSE} = \text{RMSE} * \sqrt{N} / \| \text{im_true} \|$$

where $\| . \|$ denotes the Frobenius norm and $N = \text{im_true.size}$. This result is equivalent to:

$$\text{NRMSE} = \| \text{im_true} - \text{im_test} \| / \| \text{im_true} \|.$$

- ‘min-max’ : normalize by the intensity range of `im_true`.
- ‘mean’ : normalize by the mean of `im_true`

Returns

nrmse

[float] The NRMSE metric.

Notes

Changed in version 0.16: This function was renamed from `skimage.measure.compare_nrmse` to `skimage.metrics.normalized_root_mse`.

References

[?]

`skimage.metrics.peak_signal_noise_ratio(image_true, image_test, *, data_range=None)`

Compute the peak signal to noise ratio (PSNR) for an image.

Parameters

image_true

[ndarray] Ground-truth image, same shape as im_test.

image_test

[ndarray] Test image.

data_range

[int, optional] The data range of the input image (distance between minimum and maximum possible values). By default, this is estimated from the image data-type.

Returns

psnr

[float] The PSNR metric.

Notes

Changed in version 0.16: This function was renamed from `skimage.measure.compare_psnr` to `skimage.metrics.peak_signal_noise_ratio`.

References

[?]

- *Assemble images with simple image stitching*
- *Shift-invariant wavelet denoising*
- *Non-local means denoising for preserving textures*
- *Wavelet denoising*
- *Full tutorial on calibrating Denoisers Using J-Invariance*

```
skimage.metrics.structural_similarity(im1, im2, *, win_size=None, gradient=False, data_range=None,
                                      channel_axis=None, gaussian_weights=False, full=False,
                                      **kwargs)
```

Compute the mean structural similarity index between two images. Please pay attention to the `data_range` parameter with floating-point images.

Parameters

im1, im2

[ndarray] Images. Any dimensionality with same shape.

win_size

[int or None, optional] The side-length of the sliding window used in comparison. Must be an odd value. If `gaussian_weights` is True, this is ignored and the window size will depend on `sigma`.

gradient

[bool, optional] If True, also return the gradient with respect to im2.

data_range

[float, optional] The data range of the input image (distance between minimum and maximum possible values). By default, this is estimated from the image data type. This estimate may be wrong for floating-point image data. Therefore it is recommended to always pass this value explicitly (see note below).

channel_axis

[int or None, optional] If None, the image is assumed to be a grayscale (single channel) image. Otherwise, this parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

gaussian_weights

[bool, optional] If True, each patch has its mean and variance spatially weighted by a normalized Gaussian kernel of width `sigma=1.5`.

full

[bool, optional] If True, also return the full structural similarity image.

Returns

mssim

[float] The mean structural similarity index over the image.

grad

[ndarray] The gradient of the structural similarity between `im1` and `im2` [?]. This is only returned if `gradient` is set to True.

S

[ndarray] The full SSIM image. This is only returned if `full` is set to True.

Other Parameters

use_sample_covariance

[bool] If True, normalize covariances by $N-1$ rather than, N where N is the number of pixels within the sliding window.

K1

[float] Algorithm parameter, K1 (small constant, see [?]).

K2

[float] Algorithm parameter, K2 (small constant, see [?]).

sigma

[float] Standard deviation for the Gaussian when `gaussian_weights` is True.

Notes

If `data_range` is not specified, the range is automatically guessed based on the image data type. However for floating-point image data, this estimate yields a result double the value of the desired range, as the `dtype_range` in `skimage.util.dtype.py` has defined intervals from -1 to +1. This yields an estimate of 2, instead of 1, which is most often required when working with image data (as negative light intensities are nonsensical). In case of working with YCbCr-like color data, note that these ranges are different per channel (Cb and Cr have double the range of Y), so one cannot calculate a channel-averaged SSIM with a single call to this function, as identical ranges are assumed for each channel.

To match the implementation of Wang et al. [?], set `gaussian_weights` to `True`, `sigma` to 1.5, `use_sample_covariance` to `False`, and specify the `data_range` argument.

Changed in version 0.16: This function was renamed from `skimage.measure.compare_ssim` to `skimage.metrics.structural_similarity`.

References

[?], [?]

- *Structural similarity index*

`skimage.metrics.variation_of_information(image0=None, image1=None, *, table=None, ignore_labels=())`

Return symmetric conditional entropies associated with the VI. [?]

The variation of information is defined as $VI(X,Y) = H(X|Y) + H(Y|X)$. If X is the ground-truth segmentation, then $H(X|Y)$ can be interpreted as the amount of under-segmentation and $H(Y|X)$ as the amount of over-segmentation. In other words, a perfect over-segmentation will have $H(X|Y)=0$ and a perfect under-segmentation will have $H(Y|X)=0$.

Parameters

`image0, image1`

[ndarray of int] Label images / segmentations, must have same shape.

`table`

[scipy.sparse array in csr format, optional] A contingency table built with `skimage.evaluate.contingency_table`. If `None`, it will be computed with `skimage.evaluate.contingency_table`. If given, the entropies will be computed from this table and any images will be ignored.

`ignore_labels`

[sequence of int, optional] Labels to ignore. Any part of the true image labeled with any of these values will not be counted in the score.

Returns

[ndarray of float, shape (2,)] The conditional entropies of $\text{image1}|\text{image0}$ and $\text{image0}|\text{image1}$.

References

[?]

- *Evaluating segmentation metrics*

1.3.14 skimage.morphology

<code>skimage.morphology.area_closing</code>	Perform an area closing of the image.
<code>skimage.morphology.area_opening</code>	Perform an area opening of the image.
<code>skimage.morphology.ball</code>	Generates a ball-shaped footprint.
<code>skimage.morphology.binary_closing</code>	Return fast binary morphological closing of an image.
<code>skimage.morphology.binary_dilation</code>	Return fast binary morphological dilation of an image.
<code>skimage.morphology.binary_erosion</code>	Return fast binary morphological erosion of an image.
<code>skimage.morphology.binary_opening</code>	Return fast binary morphological opening of an image.
<code>skimage.morphology.black_tophat</code>	Return black top hat of an image.
<code>skimage.morphology.closing</code>	Return grayscale morphological closing of an image.
<code>skimage.morphology.convex_hull_image</code>	Compute the convex hull image of a binary image.
<code>skimage.morphology.convex_hull_object</code>	Compute the convex hull image of individual objects in a binary image.
<code>skimage.morphology.cube</code>	Generates a cube-shaped footprint.
<code>skimage.morphology.diameter_closing</code>	Perform a diameter closing of the image.
<code>skimage.morphology.diameter_opening</code>	Perform a diameter opening of the image.
<code>skimage.morphology.diamond</code>	Generates a flat, diamond-shaped footprint.
<code>skimage.morphology.dilation</code>	Return grayscale morphological dilation of an image.
<code>skimage.morphology.disk</code>	Generates a flat, disk-shaped footprint.
<code>skimage.morphology.ellipse</code>	Generates a flat, ellipse-shaped footprint.
<code>skimage.morphology.erosion</code>	Return grayscale morphological erosion of an image.
<code>skimage.morphology.flood</code>	Mask corresponding to a flood fill.
<code>skimage.morphology.flood_fill</code>	Perform flood filling on an image.
<code>skimage.morphology.footprint_from_sequence</code>	Convert a footprint sequence into an equivalent ndarray.
<code>skimage.morphology.h_maxima</code>	Determine all maxima of the image with height $\geq h$.
<code>skimage.morphology.h_minima</code>	Determine all minima of the image with depth $\geq h$.
<code>skimage.morphology.isotropic_closing</code>	Return binary morphological closing of an image.
<code>skimage.morphology.isotropic_dilation</code>	Return binary morphological dilation of an image.
<code>skimage.morphology.isotropic_erosion</code>	Return binary morphological erosion of an image.
<code>skimage.morphology.isotropic_opening</code>	Return binary morphological opening of an image.
<code>skimage.morphology.label</code>	Label connected regions of an integer array.
<code>skimage.morphology.local_maxima</code>	Find local maxima of n-dimensional array.
<code>skimage.morphology.local_minima</code>	Find local minima of n-dimensional array.
<code>skimage.morphology.max_tree</code>	Build the max tree from an image.
<code>skimage.morphology.max_tree_local_maxima</code>	Determine all local maxima of the image.
<code>skimage.morphology.medial_axis</code>	Compute the medial axis transform of a binary image.
<code>skimage.morphology.octagon</code>	Generates an octagon shaped footprint.
<code>skimage.morphology.octahedron</code>	Generates a octahedron-shaped footprint.
<code>skimage.morphology.opening</code>	Return grayscale morphological opening of an image.

continues on next page

Table 10 – continued from previous page

<code>skimage.morphology.reconstruction</code>	Perform a morphological reconstruction of an image.
<code>skimage.morphology.rectangle</code>	Generates a flat, rectangular-shaped footprint.
<code>skimage.morphology.remove_small_holes</code>	Remove contiguous holes smaller than the specified size.
<code>skimage.morphology.remove_small_objects</code>	Remove objects smaller than the specified size.
<code>skimage.morphology.skeletonize</code>	Compute the skeleton of a binary image.
<code>skimage.morphology.skeletonize_3d</code>	Compute the skeleton of a binary image.
<code>skimage.morphology.square</code>	Generates a flat, square-shaped footprint.
<code>skimage.morphology.star</code>	Generates a star shaped footprint.
<code>skimage.morphology.thin</code>	Perform morphological thinning of a binary image.
<code>skimage.morphology.white_tophat</code>	Return white top hat of an image.

`skimage.morphology.area_closing`(*image*, *area_threshold*=64, *connectivity*=1, *parent*=None, *tree_traverser*=None)

Perform an area closing of the image.

Area closing removes all dark structures of an image with a surface smaller than *area_threshold*. The output image is larger than or equal to the input image for every pixel and all local minima have at least a surface of *area_threshold* pixels.

Area closings are similar to morphological closings, but they do not use a fixed footprint, but rather a deformable one, with surface = *area_threshold*.

In the binary case, area closings are equivalent to `remove_small_holes`; this operator is thus extended to gray-level images.

Technically, this operator is based on the max-tree representation of the image.

Parameters

image

[ndarray] The input image for which the `area_closing` is to be calculated. This image can be of any type.

area_threshold

[unsigned int] The size parameter (number of pixels). The default value is arbitrarily chosen to be 64.

connectivity

[unsigned int, optional] The neighborhood connectivity. The integer represents the maximum number of orthogonal steps to reach a neighbor. In 2D, it is 1 for a 4-neighborhood and 2 for a 8-neighborhood. Default value is 1.

parent

[ndarray, int64, optional] Parent image representing the max tree of the inverted image. The value of each pixel is the index of its parent in the ravelled array. See Note for further details.

tree_traverser

[1D array, int64, optional] The ordered pixel indices (referring to the ravelled array). The pixels are ordered such that every pixel is preceded by its parent (except for the root which has no parent).

Returns

output

[ndarray] Output image of the same shape and type as input image.

See also:

`skimage.morphology.area_opening`
`skimage.morphology.diameter_opening`
`skimage.morphology.diameter_closing`
`skimage.morphology.max_tree`
`skimage.morphology.remove_small_objects`
`skimage.morphology.remove_small_holes`

Notes

If a max-tree representation (parent and tree_traverser) are given to the function, they must be calculated from the inverted image for this function, i.e.: >>> P, S = max_tree(invert(f)) >>> closed = diameter_closing(f, 3, parent=P, tree_traverser=S)

References

[?], [?], [?], [?], [?]

Examples

We create an image (quadratic function with a minimum in the center and 4 additional local minima.

```
>>> w = 12
>>> x, y = np.mgrid[0:w,0:w]
>>> f = 180 + 0.2*((x - w/2)**2 + (y-w/2)**2)
>>> f[2:3,1:5] = 160; f[2:4,9:11] = 140; f[9:11,2:4] = 120
>>> f[9:10,9:11] = 100; f[10,10] = 100
>>> f = f.astype(int)
```

We can calculate the area closing:

```
>>> closed = area_closing(f, 8, connectivity=1)
```

All small minima are removed, and the remaining minima have at least a size of 8.

```
skimage.morphology.area_opening(image, area_threshold=64, connectivity=1, parent=None,  
tree_traverser=None)
```

Perform an area opening of the image.

Area opening removes all bright structures of an image with a surface smaller than `area_threshold`. The output image is thus the largest image smaller than the input for which all local maxima have at least a surface of `area_threshold` pixels.

Area openings are similar to morphological openings, but they do not use a fixed footprint, but rather a deformable one, with surface = `area_threshold`. Consequently, the `area_opening` with `area_threshold=1` is the identity.

In the binary case, area openings are equivalent to `remove_small_objects`; this operator is thus extended to gray-level images.

Technically, this operator is based on the max-tree representation of the image.

Parameters

image

[ndarray] The input image for which the `area_opening` is to be calculated. This image can be of any type.

area_threshold

[unsigned int] The size parameter (number of pixels). The default value is arbitrarily chosen to be 64.

connectivity

[unsigned int, optional] The neighborhood connectivity. The integer represents the maximum number of orthogonal steps to reach a neighbor. In 2D, it is 1 for a 4-neighborhood and 2 for a 8-neighborhood. Default value is 1.

parent

[ndarray, int64, optional] Parent image representing the max tree of the image. The value of each pixel is the index of its parent in the ravelled array.

tree_traverser

[1D array, int64, optional] The ordered pixel indices (referring to the ravelled array). The pixels are ordered such that every pixel is preceded by its parent (except for the root which has no parent).

Returns

output

[ndarray] Output image of the same shape and type as the input image.

See also:

`skimage.morphology.area_closing`
`skimage.morphology.diameter_opening`
`skimage.morphology.diameter_closing`
`skimage.morphology.max_tree`
`skimage.morphology.remove_small_objects`
`skimage.morphology.remove_small_holes`

References

[?], [?], [?], [?], [?]

Examples

We create an image (quadratic function with a maximum in the center and 4 additional local maxima.

```
>>> w = 12
>>> x, y = np.mgrid[0:w,0:w]
>>> f = 20 - 0.2*((x - w/2)**2 + (y-w/2)**2)
>>> f[2:3,1:5] = 40; f[2:4,9:11] = 60; f[9:11,2:4] = 80
>>> f[9:10,9:11] = 100; f[10,10] = 100
>>> f = f.astype(int)
```

We can calculate the area opening:

```
>>> open = area_opening(f, 8, connectivity=1)
```

The peaks with a surface smaller than 8 are removed.

`skimage.morphology.ball(radius, dtype=<class 'numpy.uint8'>, *, strict_radius=True, decomposition=None)`

Generates a ball-shaped footprint.

This is the 3D equivalent of a disk. A pixel is within the neighborhood if the Euclidean distance between it and the origin is no greater than radius.

Parameters

radius

[int] The radius of the ball-shaped footprint.

Returns

footprint

[ndarray or tuple] The footprint where elements of the neighborhood are 1 and 0 otherwise.

Other Parameters

`dtype`

[data-type, optional] The data type of the footprint.

`strict_radius`

[bool, optional] If False, extend the radius by 0.5. This allows the circle to expand further within a cube that remains of size $2 * \text{radius} + 1$ along each axis. This parameter is ignored if decomposition is not None.

`decomposition`

[{None, ‘sequence’}, optional] If None, a single array is returned. For ‘sequence’, a tuple of smaller footprints is returned. Applying this series of smaller footprints will give a result equivalent to a single, larger footprint, but with better computational performance. For ball footprints, the sequence decomposition is not exactly equivalent to decomposition=None. See Notes for more details.

Notes

The disk produced by the decomposition=’sequence’ mode is not identical to that with decomposition=None. Here we extend the approach taken in [?] for disks to the 3D case, using 3-dimensional extensions of the “square”, “diamond” and “t-shaped” elements from that publication. All of these elementary elements have size $(3,) * \text{ndim}$. We numerically computed the number of repetitions of each element that gives the closest match to the ball computed with kwargs `strict_radius=False, decomposition=None`.

Empirically, the equivalent composite footprint to the sequence decomposition approaches a rhombicuboctahedron (26-faces [?]).

References

[?], [?]

- *Generate footprints (structuring elements)*
- *Decompose flat footprints (structuring elements)*
- *Local Histogram Equalization*
- *Rank filters*

`skimage.morphology.binary_closing(image, footprint=None, out=None)`

Return fast binary morphological closing of an image.

This function returns the same result as grayscale closing but performs faster for binary images.

The morphological closing on an image is defined as a dilation followed by an erosion. Closing can remove small dark spots (i.e. “pepper”) and connect small bright cracks. This tends to “close” up (dark) gaps between (bright) features.

Parameters

image

[ndarray] Binary input image.

footprint

[ndarray or tuple, optional] The neighborhood expressed as a 2-D array of 1's and 0's. If None, use a cross-shaped footprint (connectivity=1). The footprint can also be provided as a sequence of smaller footprints as described in the notes below.

out

[ndarray of bool, optional] The array to store the result of the morphology. If None, is passed, a new array will be allocated.

Returns

closing

[ndarray of bool] The result of the morphological closing.

See also:

[`skimage.morphology.isotropic_closing`](#)

Notes

The footprint can also be provided as a sequence of 2-tuples where the first element of each 2-tuple is a footprint ndarray and the second element is an integer describing the number of times it should be iterated. For example `footprint=[(np.ones((9, 1)), 1), (np.ones((1, 9)), 1)]` would apply a 9x1 footprint followed by a 1x9 footprint resulting in a net effect that is the same as `footprint=np.ones((9, 9))`, but with lower computational cost. Most of the builtin footprints such as `skimage.morphology.disk` provide an option to automatically generate a footprint sequence of this type.

- *Flood Fill*
-

`skimage.morphology.binary_dilation(image, footprint=None, out=None)`

Return fast binary morphological dilation of an image.

This function returns the same result as grayscale dilation but performs faster for binary images.

Morphological dilation sets a pixel at (i, j) to the maximum over all pixels in the neighborhood centered at (i, j) . Dilation enlarges bright regions and shrinks dark regions.

Parameters

image

[ndarray] Binary input image.

footprint

[ndarray or tuple, optional] The neighborhood expressed as a 2-D array of 1's and 0's. If None, use a cross-shaped footprint (connectivity=1). The footprint can also be provided as a sequence of smaller footprints as described in the notes below.

out

[ndarray of bool, optional] The array to store the result of the morphology. If None is passed, a new array will be allocated.

Returns**dilated**

[ndarray of bool or uint] The result of the morphological dilation with values in [False, True].

See also:

[`skimage.morphology.isotropic_dilation`](#)

Notes

The footprint can also be provided as a sequence of 2-tuples where the first element of each 2-tuple is a footprint ndarray and the second element is an integer describing the number of times it should be iterated. For example `footprint=[(np.ones((9, 1)), 1), (np.ones((1, 9)), 1)]` would apply a 9x1 footprint followed by a 1x9 footprint resulting in a net effect that is the same as `footprint=np.ones((9, 9))`, but with lower computational cost. Most of the builtin footprints such as `skimage.morphology.disk` provide an option to automatically generate a footprint sequence of this type.

- *Inpainting*
- *Measure fluorescence intensity at the nuclear envelope*

`skimage.morphology.binary_erosion(image, footprint=None, out=None)`

Return fast binary morphological erosion of an image.

This function returns the same result as grayscale erosion but performs faster for binary images.

Morphological erosion sets a pixel at (i, j) to the minimum over all pixels in the neighborhood centered at (i, j) . Erosion shrinks bright regions and enlarges dark regions.

Parameters

image

[ndarray] Binary input image.

footprint

[ndarray or tuple, optional] The neighborhood expressed as a 2-D array of 1's and 0's. If None, use a cross-shaped footprint (connectivity=1). The footprint can also be provided as a sequence of smaller footprints as described in the notes below.

out

[ndarray of bool, optional] The array to store the result of the morphology. If None is passed, a new array will be allocated.

Returns

eroded

[ndarray of bool or uint] The result of the morphological erosion taking values in [False, True].

See also:

[`skimage.morphology.isotropic_erosion`](#)

Notes

The footprint can also be provided as a sequence of 2-tuples where the first element of each 2-tuple is a footprint ndarray and the second element is an integer describing the number of times it should be iterated. For example `footprint=[(np.ones((9, 1)), 1), (np.ones((1, 9)), 1)]` would apply a 9x1 footprint followed by a 1x9 footprint resulting in a net effect that is the same as `footprint=np.ones((9, 9))`, but with lower computational cost. Most of the builtin footprints such as `skimage.morphology.disk` provide an option to automatically generate a footprint sequence of this type.

- *Measure fluorescence intensity at the nuclear envelope*
-

`skimage.morphology.binary_opening(image, footprint=None, out=None)`

Return fast binary morphological opening of an image.

This function returns the same result as grayscale opening but performs faster for binary images.

The morphological opening on an image is defined as an erosion followed by a dilation. Opening can remove small bright spots (i.e. “salt”) and connect small dark cracks. This tends to “open” up (dark) gaps between (bright) features.

Parameters

image

[ndarray] Binary input image.

footprint

[ndarray or tuple, optional] The neighborhood expressed as a 2-D array of 1's and 0's. If None, use a cross-shaped footprint (connectivity=1). The footprint can also be provided as a sequence of smaller footprints as described in the notes below.

out

[ndarray of bool, optional] The array to store the result of the morphology. If None is passed, a new array will be allocated.

Returns**opening**

[ndarray of bool] The result of the morphological opening.

See also:

`skimage.morphology.isotropic_opening`

Notes

The footprint can also be provided as a sequence of 2-tuples where the first element of each 2-tuple is a footprint ndarray and the second element is an integer describing the number of times it should be iterated. For example `footprint=[[np.ones((9, 1)), 1], [np.ones((1, 9)), 1]]` would apply a 9x1 footprint followed by a 1x9 footprint resulting in a net effect that is the same as `footprint=np.ones((9, 9))`, but with lower computational cost. Most of the builtin footprints such as `skimage.morphology.disk` provide an option to automatically generate a footprint sequence of this type.

- *Flood Fill*

`skimage.morphology.black_tophat(image, footprint=None, out=None)`

Return black top hat of an image.

The black top hat of an image is defined as its morphological closing minus the original image. This operation returns the dark spots of the image that are smaller than the footprint. Note that dark spots in the original image are bright spots after the black top hat.

Parameters**image**

[ndarray] Image array.

footprint

[ndarray or tuple, optional] The neighborhood expressed as a 2-D array of 1's and 0's. If None, use a cross-shaped footprint (connectivity=1). The footprint can also be provided as a sequence of smaller footprints as described in the notes below.

out

[ndarray, optional] The array to store the result of the morphology. If None is passed, a new array will be allocated.

Returns**out**

[array, same shape and type as *image*] The result of the morphological black top hat.

See also:

[white_tophat](#)

Notes

The footprint can also be provided as a sequence of 2-tuples where the first element of each 2-tuple is a footprint ndarray and the second element is an integer describing the number of times it should be iterated. For example `footprint=[(np.ones((9, 1)), 1), (np.ones((1, 9)), 1)]` would apply a 9x1 footprint followed by a 1x9 footprint resulting in a net effect that is the same as `footprint=np.ones((9, 9))`, but with lower computational cost. Most of the builtin footprints such as `skimage.morphology.disk` provide an option to automatically generate a footprint sequence of this type.

References

[?]

Examples

```
>>> # Change dark peak to bright peak and subtract background
>>> import numpy as np
>>> from skimage.morphology import square
>>> dark_on_gray = np.array([[7, 6, 6, 6, 7],
...                           [6, 5, 4, 5, 6],
...                           [6, 4, 0, 4, 6],
...                           [6, 5, 4, 5, 6],
...                           [7, 6, 6, 6, 7]], dtype=np.uint8)
>>> black_tophat(dark_on_gray, square(3))
array([[0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 1, 5, 1, 0],
```

(continues on next page)

(continued from previous page)

```
[0, 0, 1, 0, 0],  
[0, 0, 0, 0, 0]], dtype=uint8)
```

- *Morphological Filtering*

skimage.morphology.closing(*image*, *footprint=None*, *out=None*)

Return grayscale morphological closing of an image.

The morphological closing of an image is defined as a dilation followed by an erosion. Closing can remove small dark spots (i.e. “pepper”) and connect small bright cracks. This tends to “close” up (dark) gaps between (bright) features.

Parameters

image

[ndarray] Image array.

footprint

[ndarray or tuple, optional] The neighborhood expressed as a 2-D array of 1’s and 0’s. If None, use a cross-shaped footprint (connectivity=1). The footprint can also be provided as a sequence of smaller footprints as described in the notes below.

out

[ndarray, optional] The array to store the result of the morphology. If None, a new array will be allocated.

Returns

closing

[array, same shape and type as *image*] The result of the morphological closing.

Notes

The footprint can also be provided as a sequence of 2-tuples where the first element of each 2-tuple is a footprint ndarray and the second element is an integer describing the number of times it should be iterated. For example `footprint=[(np.ones((9, 1)), 1), (np.ones((1, 9)), 1)]` would apply a 9x1 footprint followed by a 1x9 footprint resulting in a net effect that is the same as `footprint=np.ones((9, 9))`, but with lower computational cost. Most of the builtin footprints such as `skimage.morphology.disk` provide an option to automatically generate a footprint sequence of this type.

Examples

```
>>> # Close a gap between two bright lines
>>> import numpy as np
>>> from skimage.morphology import square
>>> broken_line = np.array([[0, 0, 0, 0, 0],
...                         [0, 0, 0, 0, 0],
...                         [1, 1, 0, 1, 1],
...                         [0, 0, 0, 0, 0],
...                         [0, 0, 0, 0, 0]], dtype=np.uint8)
>>> closing(broken_line, square(3))
array([[0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [1, 1, 1, 1, 1],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]], dtype=uint8)
```

- *Attribute operators*
 - *Label image regions*
 - *Morphological Filtering*
-

`skimage.morphology.convex_hull_image(image, offset_coordinates=True, tolerance=1e-10, include_borders=True)`

Compute the convex hull image of a binary image.

The convex hull is the set of pixels included in the smallest convex polygon that surround all white pixels in the input image.

Parameters

`image`

[array] Binary input image. This array is cast to bool before processing.

`offset_coordinates`

[bool, optional] If True, a pixel at coordinate, e.g., (4, 7) will be represented by coordinates (3.5, 7), (4.5, 7), (4, 6.5), and (4, 7.5). This adds some “extent” to a pixel when computing the hull.

`tolerance`

[float, optional] Tolerance when determining whether a point is inside the hull. Due to numerical floating point errors, a tolerance of 0 can result in some points erroneously being classified as being outside the hull.

`include_borders: bool, optional`

If False, vertices/edges are excluded from the final hull mask.

Returns**hull**

[(M, N) array of bool] Binary image with pixels in convex hull set to True.

References

[?]

- *Convex Hull*
- *Morphological Filtering*

`skimage.morphology.convex_hull_object(image, *, connectivity=2)`

Compute the convex hull image of individual objects in a binary image.

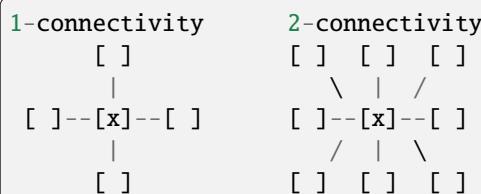
The convex hull is the set of pixels included in the smallest convex polygon that surround all white pixels in the input image.

Parameters**image**

[(M, N) ndarray] Binary input image.

connectivity

{1, 2}, int, optional] Determines the neighbors of each pixel. Adjacent elements within a squared distance of `connectivity` from pixel center are considered neighbors.:

**Returns****hull**

[ndarray of bool] Binary image with pixels inside convex hull set to True.

Notes

This function uses `skimage.morphology.label` to define unique objects, finds the convex hull of each using `convex_hull_image`, and combines these regions with logical OR. Be aware the convex hulls of unconnected objects may overlap in the result. If this is suspected, consider using `convex_hull_image` separately on each object or adjust `connectivity`.

`skimage.morphology.cube(width, dtype=<class 'numpy.uint8'>, *, decomposition=None)`

Generates a cube-shaped footprint.

This is the 3D equivalent of a square. Every pixel along the perimeter has a chessboard distance no greater than radius (`radius=floor(width/2)`) pixels.

Parameters

width

[int] The width, height and depth of the cube.

Returns

footprint

[ndarray or tuple] The footprint where elements of the neighborhood are 1 and 0 otherwise. When `decomposition` is None, this is just a numpy.ndarray. Otherwise, this will be a tuple whose length is equal to the number of unique structuring elements to apply (see Notes for more detail)

Other Parameters

dtype

[data-type, optional] The data type of the footprint.

decomposition

[{None, ‘separable’, ‘sequence’}, optional] If None, a single array is returned. For ‘sequence’, a tuple of smaller footprints is returned. Applying this series of smaller footprints will give an identical result to a single, larger footprint, but often with better computational performance. See Notes for more details.

Notes

When `decomposition` is not `None`, each element of the `footprint` tuple is a 2-tuple of the form `(ndarray, num_iter)` that specifies a footprint array and the number of iterations it is to be applied.

For binary morphology, using `decomposition='sequence'` was observed to give better performance, with the magnitude of the performance increase rapidly increasing with footprint size. For grayscale morphology with square footprints, it is recommended to use `decomposition=None` since the internal SciPy functions that are called already have a fast implementation based on separable 1D sliding windows.

The ‘sequence’ decomposition mode only supports odd valued `width`. If `width` is even, the sequence used will be identical to the ‘separable’ mode.

- *Generate footprints (structuring elements)*
 - *Decompose flat footprints (structuring elements)*
-

`skimage.morphology.diameter_closing(image, diameter_threshold=8, connectivity=1, parent=None, tree_traverser=None)`

Perform a diameter closing of the image.

Diameter closing removes all dark structures of an image with maximal extension smaller than `diameter_threshold`. The maximal extension is defined as the maximal extension of the bounding box. The operator is also called Bounding Box Closing. In practice, the result is similar to a morphological closing, but long and thin structures are not removed.

Technically, this operator is based on the max-tree representation of the image.

Parameters

`image`

[ndarray] The input image for which the `diameter_closing` is to be calculated. This image can be of any type.

`diameter_threshold`

[unsigned int] The maximal extension parameter (number of pixels). The default value is 8.

`connectivity`

[unsigned int, optional] The neighborhood connectivity. The integer represents the maximum number of orthogonal steps to reach a neighbor. In 2D, it is 1 for a 4-neighborhood and 2 for a 8-neighborhood. Default value is 1.

`parent`

[ndarray, int64, optional] Precomputed parent image representing the max tree of the inverted image. This function is fast, if precomputed parent and `tree_traverser` are provided. See Note for further details.

`tree_traverser`

[1D array, int64, optional] Precomputed traverser, where the pixels are ordered such that

every pixel is preceded by its parent (except for the root which has no parent). This function is fast, if precomputed parent and tree_traverser are provided. See Note for further details.

Returns

output

[ndarray] Output image of the same shape and type as input image.

See also:

`skimage.morphology.area_opening`
`skimage.morphology.area_closing`
`skimage.morphology.diameter_opening`
`skimage.morphology.max_tree`

Notes

If a max-tree representation (parent and tree_traverser) are given to the function, they must be calculated from the inverted image for this function, i.e.: >>> P, S = max_tree(invert(f)) >>> closed = diameter_closing(f, 3, parent=P, tree_traverser=S)

References

[?], [?]

Examples

We create an image (quadratic function with a minimum in the center and 4 additional local minima.

```
>>> w = 12
>>> x, y = np.mgrid[0:w,0:w]
>>> f = 180 + 0.2*((x - w/2)**2 + (y-w/2)**2)
>>> f[2:3,1:5] = 160; f[2:4,9:11] = 140; f[9:11,2:4] = 120
>>> f[9:10,9:11] = 100; f[10,10] = 100
>>> f = f.astype(int)
```

We can calculate the diameter closing:

```
>>> closed = diameter_closing(f, 3, connectivity=1)
```

All small minima with a maximal extension of 2 or less are removed. The remaining minima have all a maximal extension of at least 3.

- *Attribute operators*
-

```
skimage.morphology.diameter_opening(image, diameter_threshold=8, connectivity=1, parent=None,  
tree_traverser=None)
```

Perform a diameter opening of the image.

Diameter opening removes all bright structures of an image with maximal extension smaller than diameter_threshold. The maximal extension is defined as the maximal extension of the bounding box. The operator is also called Bounding Box Opening. In practice, the result is similar to a morphological opening, but long and thin structures are not removed.

Technically, this operator is based on the max-tree representation of the image.

Parameters

image

[ndarray] The input image for which the area_opening is to be calculated. This image can be of any type.

diameter_threshold

[unsigned int] The maximal extension parameter (number of pixels). The default value is 8.

connectivity

[unsigned int, optional] The neighborhood connectivity. The integer represents the maximum number of orthogonal steps to reach a neighbor. In 2D, it is 1 for a 4-neighborhood and 2 for a 8-neighborhood. Default value is 1.

parent

[ndarray, int64, optional] Parent image representing the max tree of the image. The value of each pixel is the index of its parent in the ravelled array.

tree_traverser

[1D array, int64, optional] The ordered pixel indices (referring to the ravelled array). The pixels are ordered such that every pixel is preceded by its parent (except for the root which has no parent).

Returns

output

[ndarray] Output image of the same shape and type as the input image.

See also:

`skimage.morphology.area_opening`
`skimage.morphology.area_closing`
`skimage.morphology.diameter_closing`
`skimage.morphology.max_tree`

References

[?], [?]

Examples

We create an image (quadratic function with a maximum in the center and 4 additional local maxima.

```
>>> w = 12
>>> x, y = np.mgrid[0:w,0:w]
>>> f = 20 - 0.2*((x - w/2)**2 + (y-w/2)**2)
>>> f[2:3,1:5] = 40; f[2:4,9:11] = 60; f[9:11,2:4] = 80
>>> f[9:10,9:11] = 100; f[10,10] = 100
>>> f = f.astype(int)
```

We can calculate the diameter opening:

```
>>> open = diameter_opening(f, 3, connectivity=1)
```

The peaks with a maximal extension of 2 or less are removed. The remaining peaks have all a maximal extension of at least 3.

`skimage.morphology.diamond(radius, dtype=<class 'numpy.uint8'>, *, decomposition=None)`

Generates a flat, diamond-shaped footprint.

A pixel is part of the neighborhood (i.e. labeled 1) if the city block/Manhattan distance between it and the center of the neighborhood is no greater than radius.

Parameters

radius

[int] The radius of the diamond-shaped footprint.

Returns

footprint

[ndarray or tuple] The footprint where elements of the neighborhood are 1 and 0 otherwise. When *decomposition* is None, this is just a numpy.ndarray. Otherwise, this will be a tuple whose length is equal to the number of unique structuring elements to apply (see Notes for more detail)

Other Parameters

dtype

[data-type, optional] The data type of the footprint.

decomposition

[{None, ‘sequence’}, optional] If None, a single array is returned. For ‘sequence’, a tuple of smaller footprints is returned. Applying this series of smaller footprints will give an identical result to a single, larger footprint, but with better computational performance. See Notes for more details.

Notes

When *decomposition* is not None, each element of the *footprint* tuple is a 2-tuple of the form (*ndarray*, *num_iter*) that specifies a footprint array and the number of iterations it is to be applied.

For either binary or grayscale morphology, using *decomposition='sequence'* was observed to have a performance benefit, with the magnitude of the benefit increasing with increasing footprint size.

- *Generate footprints (structuring elements)*
 - *Decompose flat footprints (structuring elements)*
-

`skimage.morphology.dilation(image, footprint=None, out=None, shift_x=False, shift_y=False)`

Return grayscale morphological dilation of an image.

Morphological dilation sets the value of a pixel to the maximum over all pixel values within a local neighborhood centered about it. The values where the footprint is 1 define this neighborhood. Dilation enlarges bright regions and shrinks dark regions.

Parameters

image

[*ndarray*] Image array.

footprint

[*ndarray* or tuple, optional] The neighborhood expressed as a 2-D array of 1’s and 0’s. If None, use a cross-shaped footprint (connectivity=1). The footprint can also be provided as a sequence of smaller footprints as described in the notes below.

out

[*ndarray*, optional] The array to store the result of the morphology. If None is passed, a new array will be allocated.

shift_x, shift_y

[bool, optional] Shift footprint about center point. This only affects 2D eccentric footprints (i.e., footprints with even-numbered sides).

Returns

dilated

[`uint8` array, same shape and type as *image*] The result of the morphological dilation.

Notes

For `uint8` (and `uint16` up to a certain bit-depth) data, the lower algorithm complexity makes the `skimage.filters.rank.maximum` function more efficient for larger images and footprints.

The footprint can also be a provided as a sequence of 2-tuples where the first element of each 2-tuple is a footprint ndarray and the second element is an integer describing the number of times it should be iterated. For example `footprint=[(np.ones((9, 1)), 1), (np.ones((1, 9)), 1)]` would apply a 9x1 footprint followed by a 1x9 footprint resulting in a net effect that is the same as `footprint=np.ones((9, 9))`, but with lower computational cost. Most of the builtin footprints such as `skimage.morphology.disk` provide an option to automatically generate a footprint sequence of this type.

Examples

```
>>> # Dilation enlarges bright regions
>>> import numpy as np
>>> from skimage.morphology import square
>>> bright_pixel = np.array([[0, 0, 0, 0, 0],
...                           [0, 0, 0, 0, 0],
...                           [0, 0, 1, 0, 0],
...                           [0, 0, 0, 0, 0],
...                           [0, 0, 0, 0, 0]], dtype=np.uint8)
>>> dilation(bright_pixel, square(3))
array([[0, 0, 0, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 0, 0, 0]], dtype=uint8)
```

- *Expand segmentation labels without overlap*
- *Morphological Filtering*
- *Rank filters*

`skimage.morphology.disk(radius, dtype=<class 'numpy.uint8'>, *, strict_radius=True, decomposition=None)`

Generates a flat, disk-shaped footprint.

A pixel is within the neighborhood if the Euclidean distance between it and the origin is no greater than radius (This is only approximately True, when `decomposition == 'sequence'`).

Parameters

radius

[int] The radius of the disk-shaped footprint.

Returns**footprint**

[ndarray] The footprint where elements of the neighborhood are 1 and 0 otherwise.

Other Parameters**dtype**

[data-type, optional] The data type of the footprint.

strict_radius

[bool, optional] If False, extend the radius by 0.5. This allows the circle to expand further within a cube that remains of size $2 * \text{radius} + 1$ along each axis. This parameter is ignored if decomposition is not None.

decomposition

[{None, ‘sequence’, ‘crosses’ }, optional] If None, a single array is returned. For ‘sequence’, a tuple of smaller footprints is returned. Applying this series of smaller footprints will give a result equivalent to a single, larger footprint, but with better computational performance. For disk footprints, the ‘sequence’ or ‘crosses’ decompositions are not always exactly equivalent to decomposition=None. See Notes for more details.

Notes

When `decomposition` is not None, each element of the `footprint` tuple is a 2-tuple of the form (`ndarray`, `num_iter`) that specifies a footprint array and the number of iterations it is to be applied.

The disk produced by the `decomposition='sequence'` mode may not be identical to that with `decomposition=None`. A disk footprint can be approximated by applying a series of smaller footprints of extent 3 along each axis. Specific solutions for this are given in [?] for the case of 2D disks with radius 2 through 10. Here, we numerically computed the number of repetitions of each element that gives the closest match to the disk computed with kwargs `strict_radius=False`, `decomposition=None`.

Empirically, the series decomposition at large radius approaches a hexadecagon (a 16-sided polygon [?]). In [?], the authors demonstrate that a hexadecagon is the closest approximation to a disk that can be achieved for decomposition with footprints of shape (3, 3).

The disk produced by the `decomposition='crosses'` is often but not always identical to that with `decomposition=None`. It tends to give a closer approximation than `decomposition='sequence'`, at a performance that is fairly comparable. The individual cross-shaped elements are not limited to extent (3, 3) in size. Unlike the ‘sequence’ decomposition, the ‘crosses’ decomposition can also accurately approximate the shape of disks with `strict_radius=True`. The method is based on an adaption of algorithm 1 given in [?].

References

[?], [?], [?], [?]

- *Generate footprints (structuring elements)*
- *Decompose flat footprints (structuring elements)*
- *Local Histogram Equalization*
- *Removing small objects in grayscale images with a top hat filter*
- *Mean filters*
- *Entropy*
- *Inpainting*
- *Sliding window histogram*
- *Apply maskSLIC vs SLIC*
- *Markers for watershed transform*
- *Flood Fill*
- *Morphological Filtering*
- *Segment human cells (in mitosis)*
- *Thresholding*
- *Rank filters*

```
skimage.morphology.ellipse(width, height, dtype=<class 'numpy.uint8'>, *, decomposition=None)
```

Generates a flat, ellipse-shaped footprint.

Every pixel along the perimeter of ellipse satisfies the equation $(x/\text{width}+1)^{**2} + (y/\text{height}+1)^{**2} = 1$.

Parameters

width

[int] The width of the ellipse-shaped footprint.

height

[int] The height of the ellipse-shaped footprint.

Returns

footprint

[ndarray] The footprint where elements of the neighborhood are 1 and 0 otherwise. The footprint will have shape (2 * height + 1, 2 * width + 1).

Other Parameters**dtype**

[data-type, optional] The data type of the footprint.

decomposition

[{None, ‘crosses’}, optional] If None, a single array is returned. For ‘sequence’, a tuple of smaller footprints is returned. Applying this series of smaller footprints will give an identical result to a single, larger footprint, but with better computational performance. See Notes for more details.

Notes

When *decomposition* is not None, each element of the *footprint* tuple is a 2-tuple of the form (*ndarray*, *num_iter*) that specifies a footprint array and the number of iterations it is to be applied.

The ellipse produced by the *decomposition='crosses'* is often but not always identical to that with *decomposition=None*. The method is based on an adaption of algorithm 1 given in [?].

References

[?]

Examples

```
>>> from skimage.morphology import footprints
>>> footprints.ellipse(5, 3)
array([[0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [0, 0, 1, 1, 1, 1, 1, 1, 0, 0]], dtype=uint8)
```

- Decompose flat footprints (structuring elements)

`skimage.morphology.erosion(image, footprint=None, out=None, shift_x=False, shift_y=False)`

Return grayscale morphological erosion of an image.

Morphological erosion sets a pixel at (i,j) to the minimum over all pixels in the neighborhood centered at (i,j). Erosion shrinks bright regions and enlarges dark regions.

Parameters

image

[ndarray] Image array.

footprint

[ndarray or tuple, optional] The neighborhood expressed as a 2-D array of 1's and 0's. If None, use a cross-shaped footprint (connectivity=1). The footprint can also be provided as a sequence of smaller footprints as described in the notes below.

out

[ndarrays, optional] The array to store the result of the morphology. If None is passed, a new array will be allocated.

shift_x, shift_y

[bool, optional] shift footprint about center point. This only affects eccentric footprints (i.e. footprint with even numbered sides).

Returns

eroded

[array, same shape as *image*] The result of the morphological erosion.

Notes

For uint8 (and uint16 up to a certain bit-depth) data, the lower algorithm complexity makes the `skimage.filters.rank.minimum` function more efficient for larger images and footprints.

The footprint can also be provided as a sequence of 2-tuples where the first element of each 2-tuple is a footprint ndarray and the second element is an integer describing the number of times it should be iterated. For example `footprint=[(np.ones((9, 1)), 1), (np.ones((1, 9)), 1)]` would apply a 9x1 footprint followed by a 1x9 footprint resulting in a net effect that is the same as `footprint=np.ones((9, 9))`, but with lower computational cost. Most of the builtin footprints such as `skimage.morphology.disk` provide an option to automatically generate a footprint sequence of this type.

Examples

```
>>> # Erosion shrinks bright regions
>>> import numpy as np
>>> from skimage.morphology import square
>>> bright_square = np.array([[0, 0, 0, 0, 0],
...                           [0, 1, 1, 1, 0],
...                           [0, 1, 1, 1, 0],
...                           [0, 1, 1, 1, 0],
...                           [0, 0, 0, 0, 0]], dtype=np.uint8)
>>> erosion(bright_square, square(3))
```

(continues on next page)

(continued from previous page)

```
array([[0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]], dtype=uint8)
```

- *Morphological Filtering*
-

`skimage.morphology.flood(image, seed_point, *, footprint=None, connectivity=None, tolerance=None)`

Mask corresponding to a flood fill.

Starting at a specific *seed_point*, connected points equal or within *tolerance* of the seed value are found.

Parameters

image

[ndarray] An n-dimensional array.

seed_point

[tuple or int] The point in *image* used as the starting point for the flood fill. If the image is 1D, this point may be given as an integer.

footprint

[ndarray, optional] The footprint (structuring element) used to determine the neighborhood of each evaluated pixel. It must contain only 1's and 0's, have the same number of dimensions as *image*. If not given, all adjacent pixels are considered as part of the neighborhood (fully connected).

connectivity

[int, optional] A number used to determine the neighborhood of each evaluated pixel. Adjacent pixels whose squared distance from the center is less than or equal to *connectivity* are considered neighbors. Ignored if *footprint* is not None.

tolerance

[float or int, optional] If None (default), adjacent values must be strictly equal to the initial value of *image* at *seed_point*. This is fastest. If a value is given, a comparison will be done at every point and if within tolerance of the initial value will also be filled (inclusive).

Returns

mask

[ndarray] A Boolean array with the same shape as *image* is returned, with True values for areas connected to and equal (or within tolerance of) the seed point. All other values are False.

Notes

The conceptual analogy of this operation is the ‘paint bucket’ tool in many raster graphics programs. This function returns just the mask representing the fill.

If indices are desired rather than masks for memory reasons, the user can simply run `numpy.nonzero` on the result, save the indices, and discard this mask.

Examples

```
>>> from skimage.morphology import flood
>>> image = np.zeros((4, 7), dtype=int)
>>> image[1:3, 1:3] = 1
>>> image[3, 0] = 1
>>> image[1:3, 4:6] = 2
>>> image[3, 6] = 3
>>> image
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 1, 1, 0, 2, 2, 0],
       [0, 1, 1, 0, 2, 2, 0],
       [1, 0, 0, 0, 0, 0, 3]])
```

Fill connected ones with 5, with full connectivity (diagonals included):

```
>>> mask = flood(image, (1, 1))
>>> image_flooded = image.copy()
>>> image_flooded[mask] = 5
>>> image_flooded
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 5, 5, 0, 2, 2, 0],
       [0, 5, 5, 0, 2, 2, 0],
       [5, 0, 0, 0, 0, 0, 3]])
```

Fill connected ones with 5, excluding diagonal points (connectivity 1):

```
>>> mask = flood(image, (1, 1), connectivity=1)
>>> image_flooded = image.copy()
>>> image_flooded[mask] = 5
>>> image_flooded
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 5, 5, 0, 2, 2, 0],
       [0, 5, 5, 0, 2, 2, 0],
       [1, 0, 0, 0, 0, 0, 3]])
```

Fill with a tolerance:

```
>>> mask = flood(image, (0, 0), tolerance=1)
>>> image_flooded = image.copy()
>>> image_flooded[mask] = 5
>>> image_flooded
array([[5, 5, 5, 5, 5, 5, 5],
       [5, 5, 5, 5, 2, 2, 5],
```

(continues on next page)

(continued from previous page)

```
[5, 5, 5, 5, 2, 2, 5],  
[5, 5, 5, 5, 5, 3]])
```

```
skimage.morphology.flood_fill(image, seed_point, new_value, *, footprint=None, connectivity=None,  
                               tolerance=None, in_place=False)
```

Perform flood filling on an image.

Starting at a specific *seed_point*, connected points equal or within *tolerance* of the seed value are found, then set to *new_value*.

Parameters

image

[ndarray] An n-dimensional array.

seed_point

[tuple or int] The point in *image* used as the starting point for the flood fill. If the image is 1D, this point may be given as an integer.

new_value

[*image* type] New value to set the entire fill. This must be chosen in agreement with the dtype of *image*.

footprint

[ndarray, optional] The footprint (structuring element) used to determine the neighborhood of each evaluated pixel. It must contain only 1's and 0's, have the same number of dimensions as *image*. If not given, all adjacent pixels are considered as part of the neighborhood (fully connected).

connectivity

[int, optional] A number used to determine the neighborhood of each evaluated pixel. Adjacent pixels whose squared distance from the center is less than or equal to *connectivity* are considered neighbors. Ignored if *footprint* is not None.

tolerance

[float or int, optional] If None (default), adjacent values must be strictly equal to the value of *image* at *seed_point* to be filled. This is fastest. If a tolerance is provided, adjacent points with values within plus or minus tolerance from the seed point are filled (inclusive).

in_place

[bool, optional] If True, flood filling is applied to *image* in place. If False, the flood filled result is returned without modifying the input *image* (default).

Returns

filled

[ndarray] An array with the same shape as *image* is returned, with values in areas connected to and equal (or within tolerance of) the seed point replaced with *new_value*.

Notes

The conceptual analogy of this operation is the ‘paint bucket’ tool in many raster graphics programs.

Examples

```
>>> from skimage.morphology import flood_fill
>>> image = np.zeros((4, 7), dtype=int)
>>> image[1:3, 1:3] = 1
>>> image[3, 0] = 1
>>> image[1:3, 4:6] = 2
>>> image[3, 6] = 3
>>> image
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 1, 1, 0, 2, 2, 0],
       [0, 1, 1, 0, 2, 2, 0],
       [1, 0, 0, 0, 0, 0, 3]])
```

Fill connected ones with 5, with full connectivity (diagonals included):

```
>>> flood_fill(image, (1, 1), 5)
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 5, 5, 0, 2, 2, 0],
       [0, 5, 5, 0, 2, 2, 0],
       [5, 0, 0, 0, 0, 0, 3]])
```

Fill connected ones with 5, excluding diagonal points (connectivity 1):

```
>>> flood_fill(image, (1, 1), 5, connectivity=1)
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 5, 5, 0, 2, 2, 0],
       [0, 5, 5, 0, 2, 2, 0],
       [1, 0, 0, 0, 0, 0, 3]])
```

Fill with a tolerance:

```
>>> flood_fill(image, (0, 0), 5, tolerance=1)
array([[5, 5, 5, 5, 5, 5, 5],
       [5, 5, 5, 5, 2, 2, 5],
       [5, 5, 5, 5, 2, 2, 5],
       [5, 5, 5, 5, 5, 5, 3]])
```

`skimage.morphology.footprint_from_sequence(footprints)`

Convert a footprint sequence into an equivalent ndarray.

Parameters**footprints**

[tuple of 2-tuples] A sequence of footprint tuples where the first element of each tuple is an array corresponding to a footprint and the second element is the number of times it is to be applied. Currently all footprints should have odd size.

Returns**footprint**

[ndarray] An single array equivalent to applying the sequence of `footprints`.

- *Decompose flat footprints (structuring elements)*
-

skimage.morphology.h_maxima(image, h, footprint=None)

Determine all maxima of the image with height $\geq h$.

The local maxima are defined as connected sets of pixels with equal gray level strictly greater than the gray level of all pixels in direct neighborhood of the set.

A local maximum M of height h is a local maximum for which there is at least one path joining M with an equal or higher local maximum on which the minimal value is $f(M) - h$ (i.e. the values along the path are not decreasing by more than h with respect to the maximum's value) and no path to an equal or higher local maximum for which the minimal value is greater.

The global maxima of the image are also found by this function.

Parameters**image**

[ndarray] The input image for which the maxima are to be calculated.

h

[unsigned integer] The minimal height of all extracted maxima.

footprint

[ndarray, optional] The neighborhood expressed as an n-D array of 1's and 0's. Default is the ball of radius 1 according to the maximum norm (i.e. a 3x3 square for 2D images, a 3x3x3 cube for 3D images, etc.)

Returns**h_max**

[ndarray] The local maxima of height $\geq h$ and the global maxima. The resulting image is a binary image, where pixels belonging to the determined maxima take value 1, the others take value 0.

See also:

`skimage.morphology.extrema.h_minima`
`skimage.morphology.extrema.local_maxima`
`skimage.morphology.extrema.local_minima`

References

[?]

Examples

```
>>> import numpy as np
>>> from skimage.morphology import extrema
```

We create an image (quadratic function with a maximum in the center and 4 additional constant maxima. The heights of the maxima are: 1, 21, 41, 61, 81

```
>>> w = 10
>>> x, y = np.mgrid[0:w, 0:w]
>>> f = 20 - 0.2*((x - w/2)**2 + (y-w/2)**2)
>>> f[2:4,2:4] = 40; f[2:4,7:9] = 60; f[7:9,2:4] = 80; f[7:9,7:9] = 100
>>> f = f.astype(int)
```

We can calculate all maxima with a height of at least 40:

```
>>> maxima = extrema.h_maxima(f, 40)
```

The resulting image will contain 3 local maxima.

- *Extrema*
-

`skimage.morphology.h_minima(image, h, footprint=None)`

Determine all minima of the image with depth $\geq h$.

The local minima are defined as connected sets of pixels with equal gray level strictly smaller than the gray levels of all pixels in direct neighborhood of the set.

A local minimum M of depth h is a local minimum for which there is at least one path joining M with an equal or lower local minimum on which the maximal value is $f(M) + h$ (i.e. the values along the path are not increasing by more than h with respect to the minimum's value) and no path to an equal or lower local minimum for which the maximal value is smaller.

The global minima of the image are also found by this function.

Parameters

image

[ndarray] The input image for which the minima are to be calculated.

h

[unsigned integer] The minimal depth of all extracted minima.

footprint

[ndarray, optional] The neighborhood expressed as an n-D array of 1's and 0's. Default is the ball of radius 1 according to the maximum norm (i.e. a 3x3 square for 2D images, a 3x3x3 cube for 3D images, etc.)

Returns

h_min

[ndarray] The local minima of depth $\geq h$ and the global minima. The resulting image is a binary image, where pixels belonging to the determined minima take value 1, the others take value 0.

See also:

`skimage.morphology.extrema.h_maxima`
`skimage.morphology.extrema.local_maxima`
`skimage.morphology.extrema.local_minima`

References

[?]

Examples

```
>>> import numpy as np
>>> from skimage.morphology import extrema
```

We create an image (quadratic function with a minimum in the center and 4 additional constant maxima. The depth of the minima are: 1, 21, 41, 61, 81

```
>>> w = 10
>>> x, y = np.mgrid[0:w,0:w]
>>> f = 180 + 0.2*((x - w/2)**2 + (y-w/2)**2)
>>> f[2:4,2:4] = 160; f[2:4,7:9] = 140; f[7:9,2:4] = 120; f[7:9,7:9] = 100
>>> f = f.astype(int)
```

We can calculate all minima with a depth of at least 40:

```
>>> minima = extrema.h_minima(f, 40)
```

The resulting image will contain 3 local minima.

`skimage.morphology.isotropic_closing(image, radius, out=None, spacing=None)`

Return binary morphological closing of an image.

This function returns the same result as `binary skimage.morphology.binary_closing()` but performs faster for large circular structuring elements. This works by thresholding the exact Euclidean distance map [?], [?]. The implementation is based on: func:scipy.ndimage.distance_transform_edt.

Parameters

image

[ndarray] Binary input image.

radius

[float] The radius with which the regions should be closed.

out

[ndarray of bool, optional] The array to store the result of the morphology. If None, is passed, a new array will be allocated.

spacing

[float, or sequence of float, optional] Spacing of elements along each dimension. If a sequence, must be of length equal to the input's dimension (number of axes). If a single number, this value is used for all axes. If not specified, a grid spacing of unity is implied.

Returns

closed

[ndarray of bool] The result of the morphological closing.

References

[?], [?]

`skimage.morphology.isotropic_dilation(image, radius, out=None, spacing=None)`

Return binary morphological dilation of an image.

This function returns the same result as `binary skimage.morphology.binary_dilation()` but performs faster for large circular structuring elements. This works by applying a threshold to the exact Euclidean distance map of the inverted image [?], [?]. The implementation is based on: func:scipy.ndimage.distance_transform_edt.

Parameters

image

[ndarray] Binary input image.

radius

[float] The radius by which regions should be dilated.

out

[ndarray of bool, optional] The array to store the result of the morphology. If None is passed, a new array will be allocated.

spacing

[float, or sequence of float, optional] Spacing of elements along each dimension. If a sequence, must be of length equal to the input's dimension (number of axes). If a single number, this value is used for all axes. If not specified, a grid spacing of unity is implied.

Returns

dilated

[ndarray of bool] The result of the morphological dilation with values in [False, True].

References

[?], [?]

skimage.morphology.isotropic_erosion(image, radius, out=None, spacing=None)

Return binary morphological erosion of an image.

This function returns the same result as `skimage.morphology.binary_erosion()` but performs faster for large circular structuring elements. This works by applying a threshold to the exact Euclidean distance map of the image [?], [?]. The implementation is based on: func:`scipy.ndimage.distance_transform_edt`.

Parameters

image

[ndarray] Binary input image.

radius

[float] The radius by which regions should be eroded.

out

[ndarray of bool, optional] The array to store the result of the morphology. If None, a new array will be allocated.

spacing

[float, or sequence of float, optional] Spacing of elements along each dimension. If a sequence, must be of length equal to the input's dimension (number of axes). If a single number, this value is used for all axes. If not specified, a grid spacing of unity is implied.

Returns

eroded

[ndarray of bool] The result of the morphological erosion taking values in [False, True].

References

[?], [?]

`skimage.morphology.isotropic_opening(image, radius, out=None, spacing=None)`

Return binary morphological opening of an image.

This function returns the same result as `skimage.morphology.binary_opening()` but performs faster for large circular structuring elements. This works by thresholding the exact Euclidean distance map [?], [?]. The implementation is based on: func:`scipy.ndimage.distance_transform_edt`.

Parameters

image

[ndarray] Binary input image.

radius

[float] The radius with which the regions should be opened.

out

[ndarray of bool, optional] The array to store the result of the morphology. If None is passed, a new array will be allocated.

spacing

[float, or sequence of float, optional] Spacing of elements along each dimension. If a sequence, must be of length equal to the input's dimension (number of axes). If a single number, this value is used for all axes. If not specified, a grid spacing of unity is implied.

Returns

opened

[ndarray of bool] The result of the morphological opening.

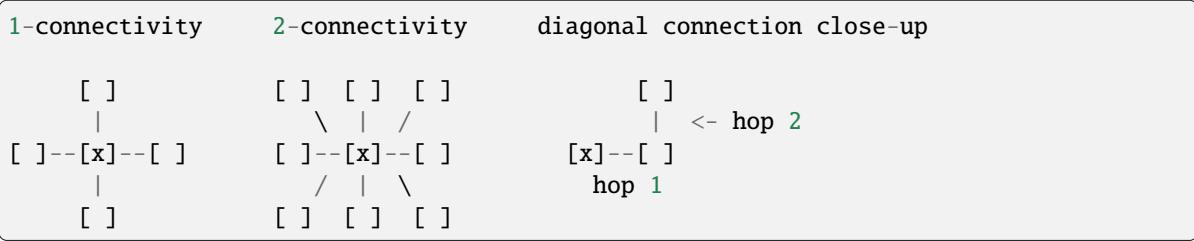
References

[?], [?]

`skimage.morphology.label(label_image, background=None, return_num=False, connectivity=None)`

Label connected regions of an integer array.

Two pixels are connected when they are neighbors and have the same value. In 2D, they can be neighbors either in a 1- or 2-connected sense. The value refers to the maximum number of orthogonal hops to consider a pixel/voxel a neighbor:

**Parameters****label_image**

[ndarray of dtype int] Image to label.

background

[int, optional] Consider all pixels with this value as background pixels, and label them as 0.
By default, 0-valued pixels are considered as background pixels.

return_num

[bool, optional] Whether to return the number of assigned labels.

connectivity

[int, optional] Maximum number of orthogonal hops to consider a pixel/voxel as a neighbor. Accepted values are ranging from 1 to `input.ndim`. If `None`, a full connectivity of `input.ndim` is used.

Returns**labels**

[ndarray of dtype int] Labeled array, where all connected regions are assigned the same integer value.

num

[int, optional] Number of labels, which equals the maximum label index and is only returned if `return_num` is *True*.

See also:

`regionprops`
`regionprops_table`

References

[?], [?]

Examples

```
>>> import numpy as np
>>> x = np.eye(3).astype(int)
>>> print(x)
[[1 0 0]
 [0 1 0]
 [0 0 1]]
>>> print(label(x, connectivity=1))
[[1 0 0]
 [0 2 0]
 [0 0 3]]
>>> print(label(x, connectivity=2))
[[1 0 0]
 [0 1 0]
 [0 0 1]]
>>> print(label(x, background=-1))
[[1 2 2]
 [2 1 2]
 [2 2 1]]
>>> x = np.array([[1, 0, 0],
...                 [1, 1, 5],
...                 [0, 0, 0]])
>>> print(label(x))
[[1 0 0]
 [1 1 2]
 [0 0 0]]
```

```
skimage.morphology.local_maxima(image, footprint=None, connectivity=None, indices=False,
                                 allow_borders=True)
```

Find local maxima of n-dimensional array.

The local maxima are defined as connected sets of pixels with equal gray level (plateaus) strictly greater than the gray levels of all pixels in the neighborhood.

Parameters

image

[ndarray] An n-dimensional array.

footprint

[ndarray, optional] The footprint (structuring element) used to determine the neighborhood of each evaluated pixel (True denotes a connected pixel). It must be a boolean array and have the same number of dimensions as *image*. If neither *footprint* nor *connectivity* are given, all adjacent pixels are considered as part of the neighborhood.

connectivity

[int, optional] A number used to determine the neighborhood of each evaluated pixel. Adjacent pixels whose squared distance from the center is less than or equal to *connectivity* are considered neighbors. Ignored if *footprint* is not None.

indices

[bool, optional] If True, the output will be a tuple of one-dimensional arrays representing the indices of local maxima in each dimension. If False, the output will be a boolean array with the same shape as *image*.

allow_borders

[bool, optional] If true, plateaus that touch the image border are valid maxima.

Returns

maxima

[ndarray or tuple[ndarray]] If *indices* is false, a boolean array with the same shape as *image* is returned with True indicating the position of local maxima (False otherwise). If *indices* is true, a tuple of one-dimensional arrays containing the coordinates (indices) of all found maxima.

Warns

UserWarning

If *allow_borders* is false and any dimension of the given *image* is shorter than 3 samples, maxima can't exist and a warning is shown.

See also:

`skimage.morphology.local_minima`
`skimage.morphology.h_maxima`
`skimage.morphology.h_minima`

Notes

This function operates on the following ideas:

1. Make a first pass over the image's last dimension and flag candidates for local maxima by comparing pixels in only one direction. If the pixels aren't connected in the last dimension all pixels are flagged as candidates instead.

For each candidate:

2. Perform a flood-fill to find all connected pixels that have the same gray value and are part of the plateau.
3. Consider the connected neighborhood of a plateau: if no bordering sample has a higher gray level, mark the plateau as a definite local maximum.

Examples

```
>>> from skimage.morphology import local_maxima
>>> image = np.zeros((4, 7), dtype=int)
>>> image[1:3, 1:3] = 1
>>> image[3, 0] = 1
>>> image[1:3, 4:6] = 2
>>> image[3, 6] = 3
>>> image
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 1, 1, 0, 2, 2, 0],
       [0, 1, 1, 0, 2, 2, 0],
       [1, 0, 0, 0, 0, 0, 3]])
```

Find local maxima by comparing to all neighboring pixels (maximal connectivity):

```
>>> local_maxima(image)
array([[False, False, False, False, False, False, False],
       [False, True, True, False, False, False, False],
       [False, True, True, False, False, False, False],
       [True, False, False, False, False, False, True]])
>>> local_maxima(image, indices=True)
(array([1, 1, 2, 2, 3, 3]), array([1, 2, 1, 2, 0, 6]))
```

Find local maxima without comparing to diagonal pixels (connectivity 1):

```
>>> local_maxima(image, connectivity=1)
array([[False, False, False, False, False, False, False],
       [False, True, True, False, True, True, False],
       [False, True, True, False, True, True, False],
       [True, False, False, False, False, False, True]])
```

and exclude maxima that border the image edge:

```
>>> local_maxima(image, connectivity=1, allow_borders=False)
array([[False, False, False, False, False, False, False],
       [False, True, True, False, True, True, False],
       [False, True, True, False, True, True, False],
       [False, False, False, False, False, False, False]])
```

- *Extrema*
-

```
skimage.morphology.local_minima(image, footprint=None, connectivity=None, indices=False,  
                                 allow_borders=True)
```

Find local minima of n-dimensional array.

The local minima are defined as connected sets of pixels with equal gray level (plateaus) strictly smaller than the gray levels of all pixels in the neighborhood.

Parameters

image

[ndarray] An n-dimensional array.

footprint

[ndarray, optional] The footprint (structuring element) used to determine the neighborhood of each evaluated pixel (True denotes a connected pixel). It must be a boolean array and have the same number of dimensions as *image*. If neither *footprint* nor *connectivity* are given, all adjacent pixels are considered as part of the neighborhood.

connectivity

[int, optional] A number used to determine the neighborhood of each evaluated pixel. Adjacent pixels whose squared distance from the center is less than or equal to *connectivity* are considered neighbors. Ignored if *footprint* is not None.

indices

[bool, optional] If True, the output will be a tuple of one-dimensional arrays representing the indices of local minima in each dimension. If False, the output will be a boolean array with the same shape as *image*.

allow_borders

[bool, optional] If true, plateaus that touch the image border are valid minima.

Returns

minima

[ndarray or tuple[ndarray]] If *indices* is false, a boolean array with the same shape as *image* is returned with True indicating the position of local minima (False otherwise). If *indices* is true, a tuple of one-dimensional arrays containing the coordinates (indices) of all found minima.

See also:

[`skimage.morphology.local_maxima`](#)
[`skimage.morphology.h_maxima`](#)
[`skimage.morphology.h_minima`](#)

Notes

This function operates on the following ideas:

1. Make a first pass over the image's last dimension and flag candidates for local minima by comparing pixels in only one direction. If the pixels aren't connected in the last dimension all pixels are flagged as candidates instead.

For each candidate:

2. Perform a flood-fill to find all connected pixels that have the same gray value and are part of the plateau.
3. Consider the connected neighborhood of a plateau: if no bordering sample has a smaller gray level, mark the plateau as a definite local minimum.

Examples

```
>>> from skimage.morphology import local_minima
>>> image = np.zeros((4, 7), dtype=int)
>>> image[1:3, 1:3] = -1
>>> image[3, 0] = -1
>>> image[1:3, 4:6] = -2
>>> image[3, 6] = -3
>>> image
array([[ 0,  0,  0,  0,  0,  0],
       [ 0, -1, -1,  0, -2, -2,  0],
       [ 0, -1, -1,  0, -2, -2,  0],
       [-1,  0,  0,  0,  0, -3]])
```

Find local minima by comparing to all neighboring pixels (maximal connectivity):

```
>>> local_minima(image)
array([[False, False, False, False, False, False, False],
       [False, True, True, False, False, False, False],
       [False, True, True, False, False, False, False],
       [True, False, False, False, False, False, True]])
>>> local_minima(image, indices=True)
(array([1, 1, 2, 2, 3, 3]), array([1, 2, 1, 2, 0, 6]))
```

Find local minima without comparing to diagonal pixels (connectivity 1):

```
>>> local_minima(image, connectivity=1)
array([[False, False, False, False, False, False, False],
       [False, True, True, False, True, True, False],
       [False, True, True, False, True, True, False],
       [True, False, False, False, False, False, True]])
```

and exclude minima that border the image edge:

```
>>> local_minima(image, connectivity=1, allow_borders=False)
array([[False, False, False, False, False, False, False],
       [False, True, True, False, True, True, False],
       [False, True, True, False, True, True, False],
       [False, False, False, False, False, False, False]])
```

`skimage.morphology.max_tree(image, connectivity=1)`

Build the max tree from an image.

Component trees represent the hierarchical structure of the connected components resulting from sequential thresholding operations applied to an image. A connected component at one level is parent of a component at a higher level if the latter is included in the first. A max-tree is an efficient representation of a component tree. A connected component at one level is represented by one reference pixel at this level, which is parent to all other pixels at that level and to the reference pixel at the level above. The max-tree is the basis for many morphological operators, namely connected operators.

Parameters

image

[ndarray] The input image for which the max-tree is to be calculated. This image can be of any type.

connectivity

[unsigned int, optional] The neighborhood connectivity. The integer represents the maximum number of orthogonal steps to reach a neighbor. In 2D, it is 1 for a 4-neighborhood and 2 for a 8-neighborhood. Default value is 1.

Returns

parent

[ndarray, int64] Array of same shape as image. The value of each pixel is the index of its parent in the ravelled array.

tree_traverser

[1D array, int64] The ordered pixel indices (referring to the ravelled array). The pixels are ordered such that every pixel is preceded by its parent (except for the root which has no parent).

References

[?], [?], [?], [?]

Examples

We create a small sample image (Figure 1 from [4]) and build the max-tree.

```
>>> image = np.array([[15, 13, 16], [12, 12, 10], [16, 12, 14]])  
>>> P, S = max_tree(image, connectivity=2)
```

- *Max-tree*
-

`skimage.morphology.max_tree_local_maxima(image, connectivity=1, parent=None, tree_traverser=None)`

Determine all local maxima of the image.

The local maxima are defined as connected sets of pixels with equal gray level strictly greater than the gray levels of all pixels in direct neighborhood of the set. The function labels the local maxima.

Technically, the implementation is based on the max-tree representation of an image. The function is very efficient if the max-tree representation has already been computed. Otherwise, it is preferable to use the function `local_maxima`.

Parameters

image

[ndarray] The input image for which the maxima are to be calculated.

connectivity

[unsigned int, optional] The neighborhood connectivity. The integer represents the maximum number of orthogonal steps to reach a neighbor. In 2D, it is 1 for a 4-neighborhood and 2 for a 8-neighborhood. Default value is 1.

parent

[ndarray, int64, optional] The value of each pixel is the index of its parent in the ravelled array.

tree_traverser

[1D array, int64, optional] The ordered pixel indices (referring to the ravelled array). The pixels are ordered such that every pixel is preceded by its parent (except for the root which has no parent).

Returns

local_max

[ndarray, uint64] Labeled local maxima of the image.

See also:

`skimage.morphology.local_maxima`
`skimage.morphology.max_tree`

References

[?], [?], [?], [?], [?]

Examples

We create an image (quadratic function with a maximum in the center and 4 additional constant maxima.

```
>>> w = 10
>>> x, y = np.mgrid[0:w,0:w]
>>> f = 20 - 0.2*((x - w/2)**2 + (y-w/2)**2)
>>> f[2:4,2:4] = 40; f[2:4,7:9] = 60; f[7:9,2:4] = 80; f[7:9,7:9] = 100
>>> f = f.astype(int)
```

We can calculate all local maxima:

```
>>> maxima = max_tree_local_maxima(f)
```

The resulting image contains the labeled local maxima.

`skimage.morphology.medial_axis(image, mask=None, return_distance=False, *, rng=None)`

Compute the medial axis transform of a binary image.

Parameters

image

[binary ndarray, shape (M, N)] The image of the shape to be skeletonized.

mask

[binary ndarray, shape (M, N), optional] If a mask is given, only those elements in *image* with a true value in *mask* are used for computing the medial axis.

return_distance

[bool, optional] If true, the distance transform is returned as well as the skeleton.

rng

[{`numpy.random.Generator`, int}, optional] Pseudo-random number generator. By default, a PCG64 generator is used (see `numpy.random.default_rng()`). If *rng* is an int, it is used to seed the generator.

The PRNG determines the order in which pixels are processed for tiebreaking.

New in version 0.19.

Returns

out

[ndarray of bools] Medial axis transform of the image

dist

[ndarray of ints, optional] Distance transform of the image (only returned if *return_distance* is True)

Other Parameters

random_state

[DEPRECATED] Deprecated in favor of *rng*.

Deprecated since version 0.21.

See also:

`skeletonize()`

Notes

This algorithm computes the medial axis transform of an image as the ridges of its distance transform.

The different steps of the algorithm are as follows

- A lookup table is used, that assigns 0 or 1 to each configuration of the 3x3 binary square, whether the central pixel should be removed or kept. We want a point to be removed if it has more than one neighbor and if removing it does not change the number of connected components.
- The distance transform to the background is computed, as well as the cornerness of the pixel.
- The foreground (value of 1) points are ordered by the distance transform, then the cornerness.
- A cython function is called to reduce the image to its skeleton. It processes pixels in the order determined at the previous step, and removes or maintains a pixel according to the lookup table. Because of the ordering, it is possible to process all pixels in only one pass.

Examples

```
>>> square = np.zeros((7, 7), dtype=np.uint8)
>>> square[1:-1, 2:-2] = 1
>>> square
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 0, 0, 0],
       [0, 0, 1, 1, 0, 0, 0],
       [0, 0, 1, 1, 0, 0, 0],
       [0, 0, 1, 1, 0, 0, 0],
       [0, 0, 1, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]], dtype=uint8)
>>> medial_axis(square).astype(np.uint8)
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 0, 1, 0, 0],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 1, 0, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]], dtype=uint8)
```

- *Skeletonize*

`skimage.morphology.octagon(m, n, dtype=<class 'numpy.uint8'>, *, decomposition=None)`

Generates an octagon shaped footprint.

For a given size of (m) horizontal and vertical sides and a given (n) height or width of slanted sides octagon is generated. The slanted sides are 45 or 135 degrees to the horizontal axis and hence the widths and heights are equal. The overall size of the footprint along a single axis will be $m + 2 * n$.

Parameters

m

[int] The size of the horizontal and vertical sides.

n

[int] The height or width of the slanted sides.

Returns

footprint

[ndarray or tuple] The footprint where elements of the neighborhood are 1 and 0 otherwise. When *decomposition* is None, this is just a numpy.ndarray. Otherwise, this will be a tuple whose length is equal to the number of unique structuring elements to apply (see Notes for more detail)

Other Parameters

dtype

[data-type, optional] The data type of the footprint.

decomposition

[{None, ‘sequence’}, optional] If None, a single array is returned. For ‘sequence’, a tuple of smaller footprints is returned. Applying this series of smaller footprints will give an identical result to a single, larger footprint, but with better computational performance. See Notes for more details.

Notes

When *decomposition* is not None, each element of the *footprint* tuple is a 2-tuple of the form (*ndarray*, *num_iter*) that specifies a footprint array and the number of iterations it is to be applied.

For either binary or grayscale morphology, using *decomposition='sequence'* was observed to have a performance benefit, with the magnitude of the benefit increasing with increasing footprint size.

- *Generate footprints (structuring elements)*
 - *Decompose flat footprints (structuring elements)*
-

`skimage.morphology.octahedron(radius, dtype=<class 'numpy.uint8'>, *, decomposition=None)`

Generates a octahedron-shaped footprint.

This is the 3D equivalent of a diamond. A pixel is part of the neighborhood (i.e. labeled 1) if the city block/Manhattan distance between it and the center of the neighborhood is no greater than radius.

Parameters

radius

[int] The radius of the octahedron-shaped footprint.

Returns

footprint

[ndarray or tuple] The footprint where elements of the neighborhood are 1 and 0 otherwise. When *decomposition* is None, this is just a numpy.ndarray. Otherwise, this will be a tuple whose length is equal to the number of unique structuring elements to apply (see Notes for more detail)

Other Parameters

dtype

[data-type, optional] The data type of the footprint.

decomposition

[{None, ‘sequence’}, optional] If None, a single array is returned. For ‘sequence’, a tuple of smaller footprints is returned. Applying this series of smaller footprints will give an identical result to a single, larger footprint, but with better computational performance. See Notes for more details.

Notes

When *decomposition* is not None, each element of the *footprint* tuple is a 2-tuple of the form (*ndarray*, *num_iter*) that specifies a footprint array and the number of iterations it is to be applied.

For either binary or grayscale morphology, using *decomposition='sequence'* was observed to have a performance benefit, with the magnitude of the benefit increasing with increasing footprint size.

- *Generate footprints (structuring elements)*
 - *Decompose flat footprints (structuring elements)*
-

skimage.morphology.opening(image, footprint=None, out=None)

Return grayscale morphological opening of an image.

The morphological opening of an image is defined as an erosion followed by a dilation. Opening can remove small bright spots (i.e. “salt”) and connect small dark cracks. This tends to “open” up (dark) gaps between (bright) features.

Parameters**image**

[*ndarray*] Image array.

footprint

[*ndarray* or tuple, optional] The neighborhood expressed as a 2-D array of 1’s and 0’s. If None, use a cross-shaped footprint (connectivity=1). The footprint can also be provided as a sequence of smaller footprints as described in the notes below.

out

[*ndarray*, optional] The array to store the result of the morphology. If None is passed, a new array will be allocated.

Returns

opening

[array, same shape and type as *image*] The result of the morphological opening.

Notes

The footprint can also be provided as a sequence of 2-tuples where the first element of each 2-tuple is a footprint ndarray and the second element is an integer describing the number of times it should be iterated. For example `footprint=[(np.ones((9, 1)), 1), (np.ones((1, 9)), 1)]` would apply a 9x1 footprint followed by a 1x9 footprint resulting in a net effect that is the same as `footprint=np.ones((9, 9))`, but with lower computational cost. Most of the builtin footprints such as `skimage.morphology.disk` provide an option to automatically generate a footprint sequence of this type.

Examples

```
>>> # Open up gap between two bright regions (but also shrink regions)
>>> import numpy as np
>>> from skimage.morphology import square
>>> bad_connection = np.array([[1, 0, 0, 0, 1],
...                             [1, 1, 0, 1, 1],
...                             [1, 1, 1, 1, 1],
...                             [1, 1, 0, 1, 1],
...                             [1, 0, 0, 0, 1]], dtype=np.uint8)
>>> opening(bad_connection, square(3))
array([[0, 0, 0, 0, 0],
       [1, 1, 0, 1, 1],
       [1, 1, 0, 1, 1],
       [1, 1, 0, 1, 1],
       [0, 0, 0, 0, 0]], dtype=uint8)
```

- *Apply maskSLIC vs SLIC*
 - *Morphological Filtering*
-

`skimage.morphology.reconstruction(seed, mask, method='dilation', footprint=None, offset=None)`

Perform a morphological reconstruction of an image.

Morphological reconstruction by dilation is similar to basic morphological dilation: high-intensity values will replace nearby low-intensity values. The basic dilation operator, however, uses a footprint to determine how far a value in the input image can spread. In contrast, reconstruction uses two images: a “seed” image, which specifies the values that spread, and a “mask” image, which gives the maximum allowed value at each pixel. The mask image, like the footprint, limits the spread of high-intensity values. Reconstruction by erosion is simply the inverse: low-intensity values spread from the seed image and are limited by the mask image, which represents the minimum allowed value.

Alternatively, you can think of reconstruction as a way to isolate the connected regions of an image. For dilation, reconstruction connects regions marked by local maxima in the seed image: neighboring pixels less-than-or-equal-to those seeds are connected to the seeded region. Local maxima with values larger than the seed image will get truncated to the seed value.

Parameters

seed

[ndarray] The seed image (a.k.a. marker image), which specifies the values that are dilated or eroded.

mask

[ndarray] The maximum (dilation) / minimum (erosion) allowed value at each pixel.

method

[{‘dilation’|‘erosion’}, optional] Perform reconstruction by dilation or erosion. In dilation (or erosion), the seed image is dilated (or eroded) until limited by the mask image. For dilation, each seed value must be less than or equal to the corresponding mask value; for erosion, the reverse is true. Default is ‘dilation’.

footprint

[ndarray, optional] The neighborhood expressed as an n-D array of 1’s and 0’s. Default is the n-D square of radius equal to 1 (i.e. a 3x3 square for 2D images, a 3x3x3 cube for 3D images, etc.)

offset

[ndarray, optional] The coordinates of the center of the footprint. Default is located on the geometrical center of the footprint, in that case footprint dimensions must be odd.

Returns

reconstructed

[ndarray] The result of morphological reconstruction.

Notes

The algorithm is taken from [?]. Applications for grayscale reconstruction are discussed in [?] and [?].

References

[?], [?], [?]

Examples

```
>>> import numpy as np
>>> from skimage.morphology import reconstruction
```

First, we create a sinusoidal mask image with peaks at middle and ends.

```
>>> x = np.linspace(0, 4 * np.pi)
>>> y_mask = np.cos(x)
```

Then, we create a seed image initialized to the minimum mask value (for reconstruction by dilation, min-intensity values don't spread) and add "seeds" to the left and right peak, but at a fraction of peak value (1).

```
>>> y_seed = y_mask.min() * np.ones_like(x)
>>> y_seed[0] = 0.5
>>> y_seed[-1] = 0
>>> y_rec = reconstruction(y_seed, y_mask)
```

The reconstructed image (or curve, in this case) is exactly the same as the mask image, except that the peaks are truncated to 0.5 and 0. The middle peak disappears completely: Since there were no seed values in this peak region, its reconstructed value is truncated to the surrounding value (-1).

As a more practical example, we try to extract the bright features of an image by subtracting a background image created by reconstruction.

```
>>> y, x = np.mgrid[:20:0.5, :20:0.5]
>>> bumps = np.sin(x) + np.sin(y)
```

To create the background image, set the mask image to the original image, and the seed image to the original image with an intensity offset, h .

```
>>> h = 0.3
>>> seed = bumps - h
>>> background = reconstruction(seed, bumps)
```

The resulting reconstructed image looks exactly like the original image, but with the peaks of the bumps cut off. Subtracting this reconstructed image from the original image leaves just the peaks of the bumps

```
>>> hdome = bumps - background
```

This operation is known as the h-dome of the image and leaves features of height h in the subtracted image.

- *Filtering regional maxima*
- *Filling holes and finding peaks*

`skimage.morphology.rectangle(nrows, ncols, dtype=<class 'numpy.uint8'>, *, decomposition=None)`

Generates a flat, rectangular-shaped footprint.

Every pixel in the rectangle generated for a given width and given height belongs to the neighborhood.

Parameters

nrows

[int] The number of rows of the rectangle.

ncols

[int] The number of columns of the rectangle.

Returns

footprint

[ndarray or tuple] A footprint consisting only of ones, i.e. every pixel belongs to the neighborhood. When *decomposition* is None, this is just a numpy.ndarray. Otherwise, this will be a tuple whose length is equal to the number of unique structuring elements to apply (see Notes for more detail)

Other Parameters

dtype

[data-type, optional] The data type of the footprint.

decomposition

[{None, ‘separable’, ‘sequence’}, optional] If None, a single array is returned. For ‘sequence’, a tuple of smaller footprints is returned. Applying this series of smaller footprints will give an identical result to a single, larger footprint, but often with better computational performance. See Notes for more details. With ‘separable’, this function uses separable 1D footprints for each axis. Whether ‘sequence’ or ‘separable’ is computationally faster may be architecture-dependent.

Notes

When *decomposition* is not None, each element of the *footprint* tuple is a 2-tuple of the form (ndarray, num_iter) that specifies a footprint array and the number of iterations it is to be applied.

For binary morphology, using *decomposition='sequence'* was observed to give better performance, with the magnitude of the performance increase rapidly increasing with footprint size. For grayscale morphology with rectangular footprints, it is recommended to use *decomposition=None* since the internal SciPy functions that are called already have a fast implementation based on separable 1D sliding windows.

The *sequence* decomposition mode only supports odd valued *nrows* and *ncols*. If either *nrows* or *ncols* is even, the sequence used will be identical to *decomposition='separable'*.

- The use of *width* and *height* has been deprecated in version 0.18.0. Use *nrows* and *ncols* instead.
- *Generate footprints (structuring elements)*
- *Decompose flat footprints (structuring elements)*

`skimage.morphology.remove_small_holes(ar, area_threshold=64, connectivity=1, *, out=None)`

Remove contiguous holes smaller than the specified size.

Parameters

ar

[ndarray (arbitrary shape, int or bool type)] The array containing the connected components of interest.

area_threshold

[int, optional (default: 64)] The maximum area, in pixels, of a contiguous hole that will be filled. Replaces *min_size*.

connectivity

[int, {1, 2, ..., ar.ndim}, optional (default: 1)] The connectivity defining the neighborhood of a pixel.

out

[ndarray] Array of the same shape as *ar* and bool dtype, into which the output is placed. By default, a new array is created.

Returns

out

[ndarray, same shape and type as input *ar*] The input array with small holes within connected components removed.

Raises

TypeError

If the input array is of an invalid type, such as float or string.

ValueError

If the input array contains negative values.

Notes

If the array type is int, it is assumed that it contains already-labeled objects. The labels are not kept in the output image (this function always outputs a bool image). It is suggested that labeling is completed after using this function.

Examples

```
>>> from skimage import morphology
>>> a = np.array([[1, 1, 1, 1, 1, 0],
...                 [1, 1, 1, 0, 1, 0],
...                 [1, 0, 0, 1, 1, 0],
...                 [1, 1, 1, 1, 1, 0]], bool)
>>> b = morphology.remove_small_holes(a, 2)
>>> b
array([[ True,  True,  True,  True,  True, False],
       [ True,  True,  True,  True,  True, False],
       [ True, False, False,  True,  True, False],
       [ True,  True,  True,  True,  True, False]])
>>> c = morphology.remove_small_holes(a, 2, connectivity=2)
>>> c
array([[ True,  True,  True,  True,  True, False],
       [ True,  True,  True, False,  True, False],
       [ True, False, False,  True,  True, False],
       [ True,  True,  True,  True,  True, False]])
>>> d = morphology.remove_small_holes(a, 2, out=a)
>>> d is a
True
```

- *Apply maskSLIC vs SLIC*
- *Measure region properties*

`skimage.morphology.remove_small_objects(ar, min_size=64, connectivity=1, *, out=None)`

Remove objects smaller than the specified size.

Expects `ar` to be an array with labeled objects, and removes objects smaller than `min_size`. If `ar` is bool, the image is first labeled. This leads to potentially different behavior for bool and 0-and-1 arrays.

Parameters

ar

[ndarray (arbitrary shape, int or bool type)] The array containing the objects of interest. If the array type is int, the ints must be non-negative.

min_size

[int, optional (default: 64)] The smallest allowable object size.

connectivity

[int, {1, 2, ..., ar.ndim}, optional (default: 1)] The connectivity defining the neighborhood of a pixel. Used during labelling if *ar* is bool.

out

[ndarray] Array of the same shape as *ar*, into which the output is placed. By default, a new array is created.

Returns**out**

[ndarray, same shape and type as input *ar*] The input array with small connected components removed.

Raises**TypeError**

If the input array is of an invalid type, such as float or string.

ValueError

If the input array contains negative values.

Examples

```
>>> from skimage import morphology
>>> a = np.array([[0, 0, 0, 1, 0],
...                 [1, 1, 1, 0, 0],
...                 [1, 1, 1, 0, 1]], bool)
>>> b = morphology.remove_small_objects(a, 6)
>>> b
array([[False, False, False, False, False],
       [ True,  True,  True, False, False],
       [ True,  True,  True, False, False]])
>>> c = morphology.remove_small_objects(a, 7, connectivity=2)
>>> c
array([[False, False, False,  True, False],
       [ True,  True,  True, False, False],
       [ True,  True,  True, False, False]])
>>> d = morphology.remove_small_objects(a, 6, out=a)
>>> d is a
True
```

- *Apply maskSLIC vs SLIC*
- *Measure region properties*

- *Evaluating segmentation metrics*
 - *Comparing edge-based and region-based segmentation*
-

`skimage.morphology.skeletonize(image, *, method=None)`

Compute the skeleton of a binary image.

Thinning is used to reduce each connected component in a binary image to a single-pixel wide skeleton.

Parameters

image

[ndarray, 2D or 3D] An image containing the objects to be skeletonized. Zeros represent background, nonzero values are foreground.

method

[{'zhang', 'lee'}, optional] Which algorithm to use. Zhang's algorithm [?] only works for 2D images, and is the default for 2D. Lee's algorithm [?] works for 2D or 3D images and is the default for 3D.

Returns

skeleton

[ndarray] The thinned image.

See also:

`medial_axis`

References

[?], [?]

Examples

```
>>> X, Y = np.ogrid[0:9, 0:9]
>>> ellipse = (1./3 * (X - 4)**2 + (Y - 4)**2 < 3**2).astype(np.uint8)
>>> ellipse
array([[0, 0, 0, 1, 1, 1, 0, 0, 0],
       [0, 0, 1, 1, 1, 1, 0, 0, 0],
       [0, 0, 1, 1, 1, 1, 0, 0, 0],
       [0, 0, 1, 1, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 1, 1, 0, 0],
```

(continues on next page)

(continued from previous page)

```
[0, 0, 1, 1, 1, 1, 0, 0],  
[0, 0, 1, 1, 1, 1, 0, 0],  
[0, 0, 1, 1, 1, 1, 0, 0],  
[0, 0, 0, 1, 1, 1, 0, 0]], dtype=uint8)  
>>> skel = skeletonize(ellipse)  
>>> skel.astype(np.uint8)  
array([[0, 0, 0, 0, 0, 0, 0, 0, 0],  
       [0, 0, 0, 0, 0, 0, 0, 0, 0],  
       [0, 0, 0, 0, 0, 0, 0, 0, 0],  
       [0, 0, 0, 0, 1, 0, 0, 0, 0],  
       [0, 0, 0, 0, 1, 0, 0, 0, 0],  
       [0, 0, 0, 0, 1, 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]], dtype=uint8)
```

- *Skeletonize*
- *Use pixel graphs to find an object's geodesic center*
- *Morphological Filtering*

skimage.morphology.skeletonize_3d(image)

Compute the skeleton of a binary image.

Thinning is used to reduce each connected component in a binary image to a single-pixel wide skeleton.

Parameters

image

[ndarray, 2D or 3D] A binary image containing the objects to be skeletonized. Zeros represent background, nonzero values are foreground.

Returns

skeleton

[ndarray] The thinned image.

See also:

[skeletonize](#), [medial_axis](#)

Notes

The method of [?] uses an octree data structure to examine a 3x3x3 neighborhood of a pixel. The algorithm proceeds by iteratively sweeping over the image, and removing pixels at each iteration until the image stops changing. Each iteration consists of two steps: first, a list of candidates for removal is assembled; then pixels from this list are rechecked sequentially, to better preserve connectivity of the image.

The algorithm this function implements is different from the algorithms used by either `skeletonize` or `medial_axis`, thus for 2D images the results produced by this function are generally different.

References

[?]

`skimage.morphology.square(width, dtype=<class 'numpy.uint8'>, *, decomposition=None)`

Generates a flat, square-shaped footprint.

Every pixel along the perimeter has a chessboard distance no greater than radius (`radius=floor(width/2)`) pixels.

Parameters

width

[int] The width and height of the square.

Returns

footprint

[ndarray or tuple] The footprint where elements of the neighborhood are 1 and 0 otherwise. When `decomposition` is None, this is just a numpy.ndarray. Otherwise, this will be a tuple whose length is equal to the number of unique structuring elements to apply (see Notes for more detail)

Other Parameters

dtype

[data-type, optional] The data type of the footprint.

decomposition

[{None, ‘separable’, ‘sequence’}, optional] If None, a single array is returned. For ‘sequence’, a tuple of smaller footprints is returned. Applying this series of smaller footprints will give an identical result to a single, larger footprint, but often with better computational performance. See Notes for more details. With ‘separable’, this function uses separable 1D footprints for each axis. Whether ‘sequence’ or ‘separable’ is computationally faster may be architecture-dependent.

Notes

When `decomposition` is not `None`, each element of the `footprint` tuple is a 2-tuple of the form `(ndarray, num_iter)` that specifies a footprint array and the number of iterations it is to be applied.

For binary morphology, using `decomposition='sequence'` or `decomposition='separable'` were observed to give better performance than `decomposition=None`, with the magnitude of the performance increase rapidly increasing with footprint size. For grayscale morphology with square footprints, it is recommended to use `decomposition=None` since the internal SciPy functions that are called already have a fast implementation based on separable 1D sliding windows.

The ‘sequence’ decomposition mode only supports odd valued `width`. If `width` is even, the sequence used will be identical to the ‘separable’ mode.

- *Generate footprints (structuring elements)*
 - *Decompose flat footprints (structuring elements)*
 - *Attribute operators*
 - *Label image regions*
-

```
skimage.morphology.star(a, dtype=<class 'numpy.uint8'>)
```

Generates a star shaped footprint.

Start has 8 vertices and is an overlap of square of size $2*a + 1$ with its 45 degree rotated version. The slanted sides are 45 or 135 degrees to the horizontal axis.

Parameters

a

[int] Parameter deciding the size of the star structural element. The side of the square array returned is $2*a + 1 + 2*\lfloor a / 2 \rfloor$.

Returns

footprint

[ndarray] The footprint where elements of the neighborhood are 1 and 0 otherwise.

Other Parameters

dtype

[data-type, optional] The data type of the footprint.

- *Generate footprints (structuring elements)*

```
skimage.morphology.thin(image, max_num_iter=None)
```

Perform morphological thinning of a binary image.

Parameters

image

[binary (M, N) ndarray] The image to be thinned.

max_num_iter

[int, number of iterations, optional] Regardless of the value of this parameter, the thinned image is returned immediately if an iteration produces no change. If this parameter is specified it thus sets an upper bound on the number of iterations performed.

Returns

out

[ndarray of bool] Thinned image.

See also:

[**skeletonize**](#), [**medial_axis**](#)

Notes

This algorithm [?] works by making multiple passes over the image, removing pixels matching a set of criteria designed to thin connected regions while preserving eight-connected components and 2 x 2 squares [?]. In each of the two sub-iterations the algorithm correlates the intermediate skeleton image with a neighborhood mask, then looks up each neighborhood in a lookup table indicating whether the central pixel should be deleted in that sub-iteration.

References

[?], [?]

Examples

```
>>> square = np.zeros((7, 7), dtype=np.uint8)
>>> square[1:-1, 2:-2] = 1
>>> square[0, 1] = 1
>>> square
array([[0, 1, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 0, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]], dtype=uint8)
>>> skel = thin(square)
>>> skel.astype(np.uint8)
array([[0, 1, 0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]], dtype=uint8)
```

- *Skeletonize*
-

`skimage.morphology.white_tophat(image, footprint=None, out=None)`

Return white top hat of an image.

The white top hat of an image is defined as the image minus its morphological opening. This operation returns the bright spots of the image that are smaller than the footprint.

Parameters

`image`

[ndarray] Image array.

`footprint`

[ndarray or tuple, optional] The neighborhood expressed as a 2-D array of 1's and 0's. If None, use a cross-shaped footprint (connectivity=1). The footprint can also be provided as a sequence of smaller footprints as described in the notes below.

`out`

[ndarray, optional] The array to store the result of the morphology. If None is passed, a new array will be allocated.

Returns

out

[array, same shape and type as *image*] The result of the morphological white top hat.

See also:

`black_tophat`

Notes

The footprint can also be provided as a sequence of 2-tuples where the first element of each 2-tuple is a footprint ndarray and the second element is an integer describing the number of times it should be iterated. For example `footprint=[(np.ones((9, 1)), 1), (np.ones((1, 9)), 1)]` would apply a 9x1 footprint followed by a 1x9 footprint resulting in a net effect that is the same as `footprint=np.ones((9, 9))`, but with lower computational cost. Most of the builtin footprints such as `skimage.morphology.disk` provide an option to automatically generate a footprint sequence of this type.

References

[?]

Examples

```
>>> # Subtract gray background from bright peak
>>> import numpy as np
>>> from skimage.morphology import square
>>> bright_on_gray = np.array([[2, 3, 3, 3, 2],
...                             [3, 4, 5, 4, 3],
...                             [3, 5, 9, 5, 3],
...                             [3, 4, 5, 4, 3],
...                             [2, 3, 3, 3, 2]], dtype=np.uint8)
>>> white_tophat(bright_on_gray, square(3))
array([[0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 1, 5, 1, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0]], dtype=uint8)
```

- *Removing small objects in grayscale images with a top hat filter*
- *Morphological Filtering*

1.3.15 skimage.registration

<code>skimage.registration.optical_flow_ilk</code>	Coarse to fine optical flow estimator.
<code>skimage.registration.optical_flow_tvl1</code>	Coarse to fine optical flow estimator.
<code>skimage.registration.phase_cross_correlation</code>	Efficient subpixel image translation registration by cross-correlation.

```
skimage.registration.optical_flow_ilk(reference_image, moving_image, *, radius=7, num_warp=10,
gaussian=False, prefilter=False, dtype=<class 'numpy.float32'>)
```

Coarse to fine optical flow estimator.

The iterative Lucas-Kanade (iLK) solver is applied at each level of the image pyramid. iLK [?] is a fast and robust alternative to TVL1 algorithm although less accurate for rendering flat surfaces and object boundaries (see [?]).

Parameters

`reference_image`

[ndarray, shape (M, N[, P[, ...]])] The first gray scale image of the sequence.

`moving_image`

[ndarray, shape (M, N[, P[, ...]])] The second gray scale image of the sequence.

`radius`

[int, optional] Radius of the window considered around each pixel.

`num_warp`

[int, optional] Number of times moving_image is warped.

`gaussian`

[bool, optional] If True, a Gaussian kernel is used for the local integration. Otherwise, a uniform kernel is used.

`prefilter`

[bool, optional] Whether to prefilter the estimated optical flow before each image warp. When True, a median filter with window size 3 along each axis is applied. This helps to remove potential outliers.

`dtype`

[dtype, optional] Output data type: must be floating point. Single precision provides good results and saves memory usage and computation time compared to double precision.

Returns

`flow`

[ndarray, shape ((reference_image.ndim, M, N[, P[, ...]]))] The estimated optical flow components for each axis.

Notes

- The implemented algorithm is described in **Table2** of [?].
- Color images are not supported.

References

[?], [?]

Examples

```
>>> from skimage.color import rgb2gray
>>> from skimage.data import stereo_motorcycle
>>> from skimage.registration import optical_flow_ilk
>>> reference_image, moving_image, disp = stereo_motorcycle()
>>> # --- Convert the images to gray level: color is not supported.
>>> reference_image = rgb2gray(reference_image)
>>> moving_image = rgb2gray(moving_image)
>>> flow = optical_flow_ilk(moving_image, reference_image)
```

- *Registration using optical flow*

`skimage.registration.optical_flow_tvl1(reference_image, moving_image, *, attachment=15, tightness=0.3, num_warp=5, num_iter=10, tol=0.0001, prefilter=False, dtype=<class 'numpy.float32'>)`

Coarse to fine optical flow estimator.

The TV-L1 solver is applied at each level of the image pyramid. TV-L1 is a popular algorithm for optical flow estimation introduced by Zack et al. [?], improved in [?] and detailed in [?].

Parameters

reference_image

[ndarray, shape (M, N[, P[, ...]])] The first gray scale image of the sequence.

moving_image

[ndarray, shape (M, N[, P[, ...]])] The second gray scale image of the sequence.

attachment

[float, optional] Attachment parameter (λ in [?]). The smaller this parameter is, the smoother the returned result will be.

tightness

[float, optional] Tightness parameter (τ in [?]). It should have a small value in order to maintain attachment and regularization parts in correspondence.

num_warp

[int, optional] Number of times moving_image is warped.

num_iter

[int, optional] Number of fixed point iteration.

tol

[float, optional] Tolerance used as stopping criterion based on the L^2 distance between two consecutive values of (u, v).

prefilter

[bool, optional] Whether to prefilter the estimated optical flow before each image warp. When True, a median filter with window size 3 along each axis is applied. This helps to remove potential outliers.

dtype

[dtype, optional] Output data type: must be floating point. Single precision provides good results and saves memory usage and computation time compared to double precision.

Returns

flow

[ndarray, shape ((image0.ndim, M, N[, P[, ...]])] The estimated optical flow components for each axis.

Notes

Color images are not supported.

References

[?], [?], [?]

Examples

```
>>> from skimage.color import rgb2gray
>>> from skimage.data import stereo_motorcycle
>>> from skimage.registration import optical_flow_tvl1
>>> image0, image1, disp = stereo_motorcycle()
>>> # --- Convert the images to gray level: color is not supported.
>>> image0 = rgb2gray(image0)
>>> image1 = rgb2gray(image1)
>>> flow = optical_flow_tvl1(image1, image0)
```

- Registration using optical flow

`skimage.registration.phase_cross_correlation(reference_image, moving_image, *, upsample_factor=1, space='real', disambiguate=False, return_error=True, reference_mask=None, moving_mask=None, overlap_ratio=0.3, normalization='phase')`

Efficient subpixel image translation registration by cross-correlation.

This code gives the same precision as the FFT upsampled cross-correlation in a fraction of the computation time and with reduced memory requirements. It obtains an initial estimate of the cross-correlation peak by an FFT and then refines the shift estimation by upsampling the DFT only in a small neighborhood of that estimate by means of a matrix-multiply DFT [?].

Parameters

`reference_image`

[array] Reference image.

`moving_image`

[array] Image to register. Must be same dimensionality as `reference_image`.

`upsample_factor`

[int, optional] Upsampling factor. Images will be registered to within 1 / `upsample_factor` of a pixel. For example `upsample_factor == 20` means the images will be registered within 1/20th of a pixel. Default is 1 (no upsampling). Not used if any of `reference_mask` or `moving_mask` is not None.

`space`

[string, one of “real” or “fourier”, optional] Defines how the algorithm interprets input data. “real” means data will be FFT’d to compute the correlation, while “fourier” data will bypass FFT of input data. Case insensitive. Not used if any of `reference_mask` or `moving_mask` is not None.

`disambiguate`

[bool] The shift returned by this function is only accurate *modulo* the image shape, due to the periodic nature of the Fourier transform. If this parameter is set to True, the *real*

space cross-correlation is computed for each possible shift, and the shift with the highest cross-correlation within the overlapping area is returned.

return_error

[bool, {"always"}, optional] Returns error and phase difference if "always" is given. If False, or either `reference_mask` or `moving_mask` are given, only the shift is returned.

reference_mask

[ndarray] Boolean mask for `reference_image`. The mask should evaluate to True (or 1) on valid pixels. `reference_mask` should have the same shape as `reference_image`.

moving_mask

[ndarray or None, optional] Boolean mask for `moving_image`. The mask should evaluate to True (or 1) on valid pixels. `moving_mask` should have the same shape as `moving_image`. If None, `reference_mask` will be used.

overlap_ratio

[float, optional] Minimum allowed overlap ratio between images. The correlation for translations corresponding with an overlap ratio lower than this threshold will be ignored. A lower `overlap_ratio` leads to smaller maximum translation, while a higher `overlap_ratio` leads to greater robustness against spurious matches due to small overlap between masked images. Used only if one of `reference_mask` or `moving_mask` is not None.

normalization

[{"phase", None}] The type of normalization to apply to the cross-correlation. This parameter is unused when masks (`reference_mask` and `moving_mask`) are supplied.

Returns

shift

[ndarray] Shift vector (in pixels) required to register `moving_image` with `reference_image`. Axis ordering is consistent with the axis order of the input array.

error

[float] Translation invariant normalized RMS error between `reference_image` and `moving_image`. For masked cross-correlation this error is not available and NaN is returned if `return_error` is "always".

phasediff

[float] Global phase difference between the two images (should be zero if images are non-negative). For masked cross-correlation this phase difference is not available and NaN is returned if `return_error` is "always".

Notes

The use of cross-correlation to estimate image translation has a long history dating back to at least [?]. The “phase correlation” method (selected by `normalization="phase"`) was first proposed in [?]. Publications [?] and [?] use an unnormalized cross-correlation (`normalization=None`). Which form of normalization is better is application-dependent. For example, the phase correlation method works well in registering images under different illumination, but is not very robust to noise. In a high noise scenario, the unnormalized method may be preferable.

When masks are provided, a masked normalized cross-correlation algorithm is used [?], [?].

References

[?], [?], [?], [?], [?], [?]

- *Image Registration*
- *Masked Normalized Cross-Correlation*
- *Using Polar and Log-Polar Transformations for Registration*

1.3.16 skimage.restoration

Image restoration module.

<code>skimage.restoration.ball_kernel</code>	Create a ball kernel for <code>restoration.rolling_ball</code> .
<code>skimage.restoration.calibrate_denoiser</code>	Calibrate a denoising function and return optimal J-invariant version.
<code>skimage.restoration.cycle_spin</code>	Cycle spinning (repeatedly apply <code>func</code> to shifted versions of <code>x</code>).
<code>skimage.restoration.denoise_bilateral</code>	Denoise image using bilateral filter.
<code>skimage.restoration.denoise_invariant</code>	Apply a J-invariant version of a denoising function.
<code>skimage.restoration.denoise_nl_means</code>	Perform non-local means denoising on 2D-4D grayscale or RGB images.
<code>skimage.restoration.denoise_tv_bregman</code>	Perform total variation denoising using split-Bregman optimization.
<code>skimage.restoration.denoise_tv_chambolle</code>	Perform total variation denoising in nD.
<code>skimage.restoration.denoise_wavelet</code>	Perform wavelet denoising on an image.
<code>skimage.restoration.ellipsoid_kernel</code>	Create an ellipsoid kernel for <code>restoration.rolling_ball</code> .
<code>skimage.restoration.estimate_sigma</code>	Robust wavelet-based estimator of the (Gaussian) noise standard deviation.
<code>skimage.restoration.inpaint_biharmonic</code>	Inpaint masked points in image with biharmonic equations.
<code>skimage.restoration.richardson_lucy</code>	Richardson-Lucy deconvolution.
<code>skimage.restoration.rolling_ball</code>	Estimate background intensity by rolling/translating a kernel.
<code>skimage.restoration.unsupervised_wiener</code>	Unsupervised Wiener-Hunt deconvolution.
<code>skimage.restoration.unwrap_phase</code>	Recover the original from a wrapped phase image.
<code>skimage.restoration.wiener</code>	Wiener-Hunt deconvolution

`skimage.restoration.ball_kernel(radius, ndim)`

Create a ball kernel for restoration.rolling_ball.

Parameters

radius

[int] Radius of the ball.

ndim

[int] Number of dimensions of the ball. `ndim` should match the dimensionality of the image the kernel will be applied to.

Returns

kernel

[ndarray] The kernel containing the surface intensity of the top half of the ellipsoid.

See also:

[`rolling_ball`](#)

`skimage.restoration.calibrate_denoiser(image, denoise_function, denoise_parameters, *, stride=4, approximate_loss=True, extra_output=False)`

Calibrate a denoising function and return optimal J-invariant version.

The returned function is partially evaluated with optimal parameter values set for denoising the input image.

Parameters

image

[ndarray] Input data to be denoised (converted using `img_as_float`).

denoise_function

[function] Denoising function to be calibrated.

denoise_parameters

[dict of list] Ranges of parameters for `denoise_function` to be calibrated over.

stride

[int, optional] Stride used in masking procedure that converts `denoise_function` to J-invariance.

approximate_loss

[bool, optional] Whether to approximate the self-supervised loss used to evaluate the denoiser by only computing it on one masked version of the image. If False, the runtime will be a factor of $stride^{ndim}$ longer.

extra_output

[bool, optional] If True, return parameters and losses in addition to the calibrated denoising function

Returns**best_denoise_function**

[function] The optimal J-invariant version of *denoise_function*.

If *extra_output* is True, the following tuple is also returned:
(parameters_tested, losses)

[tuple (list of dict, list of int)] List of parameters tested for *denoise_function*, as a dictionary of kwargs Self-supervised loss for each set of parameters in *parameters_tested*.

Notes

The calibration procedure uses a self-supervised mean-square-error loss to evaluate the performance of J-invariant versions of *denoise_function*. The minimizer of the self-supervised loss is also the minimizer of the ground-truth loss (i.e., the true MSE error) [1]. The returned function can be used on the original noisy image, or other images with similar characteristics.

Increasing the stride increases the performance of *best_denoise_function*

at the expense of increasing its runtime. It has no effect on the runtime of the calibration.

References

[?]

Examples

```
>>> from skimage import color, data
>>> from skimage.restoration import denoise_wavelet
>>> import numpy as np
>>> img = color.rgb2gray(data.astronaut()[:50, :50])
>>> rng = np.random.default_rng()
>>> noisy = img + 0.5 * img.std() * rng.standard_normal(img.shape)
>>> parameters = {'sigma': np.arange(0.1, 0.4, 0.02)}
>>> denoising_function = calibrate_denoiser(noisy, denoise_wavelet,
...                                         denoise_parameters=parameters)
>>> denoised_img = denoising_function(img)
```

- *Calibrating Denoisers Using J-Invariance*

- Full tutorial on calibrating Denoisers Using J-Invariance
-

```
skimage.restoration.cycle_spin(x, func, max_shifts, shift_steps=1, num_workers=None, func_kw=None, *  
                               channel_axis=None)
```

Cycle spinning (repeatedly apply func to shifted versions of x).

Parameters

x

[array-like] Data for input to func.

func

[function] A function to apply to circularly shifted versions of x. Should take x as its first argument. Any additional arguments can be supplied via func_kw.

max_shifts

[int or tuple] If an integer, shifts in range(0, max_shifts+1) will be used along each axis of x. If a tuple, range(0, max_shifts[i]+1) will be along axis i.

shift_steps

[int or tuple, optional] The step size for the shifts applied along axis, i, are:: range((0, max_shifts[i]+1, shift_steps[i])). If an integer is provided, the same step size is used for all axes.

num_workers

[int or None, optional] The number of parallel threads to use during cycle spinning. If set to None, the full set of available cores are used.

func_kw

[dict, optional] Additional keyword arguments to supply to func.

channel_axis

[int or None, optional] If None, the image is assumed to be a grayscale (single channel) image. Otherwise, this parameter indicates which axis of the array corresponds to channels.

New in version 0.19: channel_axis was added in 0.19.

Returns

avg_y

[np.ndarray] The output of func(x, **func_kw) averaged over all combinations of the specified axis shifts.

Notes

Cycle spinning was proposed as a way to approach shift-invariance via performing several circular shifts of a shift-variant transform [?].

For a n-level discrete wavelet transforms, one may wish to perform all shifts up to `max_shifts = 2**n - 1`. In practice, much of the benefit can often be realized with only a small number of shifts per axis.

For transforms such as the blockwise discrete cosine transform, one may wish to evaluate shifts up to the block size used by the transform.

References

[?]

Examples

```
>>> import skimage.data
>>> from skimage import img_as_float
>>> from skimage.restoration import denoise_wavelet, cycle_spin
>>> img = img_as_float(skimage.data.camera())
>>> sigma = 0.1
>>> img = img + sigma * np.random.standard_normal(img.shape)
>>> denoised = cycle_spin(img, func=denoise_wavelet,
...                         max_shifts=3)
```

- Shift-invariant wavelet denoising

`skimage.restoration.denoise_bilateral(image, win_size=None, sigma_color=None, sigma_spatial=1, bins=10000, mode='constant', cval=0, *, channel_axis=None)`

Denoise image using bilateral filter.

Parameters

image

[ndarray, shape (M, N[, 3])] Input image, 2D grayscale or RGB.

win_size

[int] Window size for filtering. If `win_size` is not specified, it is calculated as `max(5, 2 * ceil(3 * sigma_spatial) + 1)`.

sigma_color

[float] Standard deviation for grayvalue/color distance (radiometric similarity). A larger value results in averaging of pixels with larger radiometric differences. If `None`, the standard deviation of `image` will be used.

sigma_spatial

[float] Standard deviation for range distance. A larger value results in averaging of pixels with larger spatial differences.

bins

[int] Number of discrete values for Gaussian weights of color filtering. A larger value results in improved accuracy.

mode

[{‘constant’, ‘edge’, ‘symmetric’, ‘reflect’, ‘wrap’}] How to handle values outside the image borders. See `numpy.pad` for detail.

cval

[string] Used in conjunction with mode ‘constant’, the value outside the image boundaries.

channel_axis

[int or None, optional] If `None`, the image is assumed to be grayscale (single-channel). Otherwise, this parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

denoised

[ndarray] Denoised image.

Notes

This is an edge-preserving, denoising filter. It averages pixels based on their spatial closeness and radiometric similarity [?].

Spatial closeness is measured by the Gaussian function of the Euclidean distance between two pixels and a certain standard deviation (`sigma_spatial`).

Radiometric similarity is measured by the Gaussian function of the Euclidean distance between two color values and a certain standard deviation (`sigma_color`).

Note that, if the image is of any `int` dtype, `image` will be converted using the `img_as_float` function and thus the standard deviation (`sigma_color`) will be in range [0, 1].

For more information on scikit-image’s data type conversions and how images are rescaled in these conversions, see: https://scikit-image.org/docs/stable/user_guide/data_types.html.

References

[?]

Examples

```
>>> from skimage import data, img_as_float
>>> astro = img_as_float(data.astronaut())
>>> astro = astro[220:300, 220:320]
>>> rng = np.random.default_rng()
>>> noisy = astro + 0.6 * astro.std() * rng.random(astro.shape)
>>> noisy = np.clip(noisy, 0, 1)
>>> denoised = denoise_bilateral(noisy, sigma_color=0.05, sigma_spatial=15,
...                                 channel_axis=-1)
```

- Denoising a picture
- Rank filters

`skimage.restoration.denoise_invariant(image, denoise_function, *, stride=4, masks=None, denoiser_kwargs=None)`

Apply a J-invariant version of a denoising function.

Parameters

image

[ndarray ([M[, N[, ...P]][, C]) of ints, uints or floats] Input data to be denoised. *image* can be of any numeric type, but it is cast into a ndarray of floats (using *img_as_float*) for the computation of the denoised image.

denoise_function

[function] Original denoising function.

stride

[int, optional] Stride used in masking procedure that converts *denoise_function* to J-invariance.

masks

[list of ndarray, optional] Set of masks to use for computing J-invariant output. If *None*, a full set of masks covering the image will be used.

denoiser_kwargs:

Keyword arguments passed to *denoise_function*.

Returns

output

[ndarray] Denoised image, of same shape as *image*.

Notes

A denoising function is J-invariant if the prediction it makes for each pixel does not depend on the value of that pixel in the original image. The prediction for each pixel may instead use all the relevant information contained in the rest of the image, which is typically quite significant. Any function can be converted into a J-invariant one using a simple masking procedure, as described in [1].

The pixel-wise error of a J-invariant denoiser is uncorrelated to the noise, so long as the noise in each pixel is independent. Consequently, the average difference between the denoised image and the noisy image, the *self-supervised loss*, is the same as the difference between the denoised image and the original clean image, the *ground-truth loss* (up to a constant).

This means that the best J-invariant denoiser for a given image can be found using the noisy data alone, by selecting the denoiser minimizing the self-supervised loss.

References

[?]

Examples

```
>>> import skimage
>>> from skimage.restoration import denoise_invariant, denoise_wavelet
>>> image = skimage.util.img_as_float(skimage.data.chelsea())
>>> noisy = skimage.util.random_noise(image, var=0.2 ** 2)
>>> denoised = denoise_invariant(noisy, denoise_function=denoise_wavelet)
```

- Full tutorial on calibrating Denoisers Using J-Invariance
-

`skimage.restoration.denoise_nl_means(image, patch_size=7, patch_distance=11, h=0.1, fast_mode=True, sigma=0.0, *, preserve_range=False, channel_axis=None)`

Perform non-local means denoising on 2D-4D grayscale or RGB images.

Parameters**image**

[2D or 3D ndarray] Input image to be denoised, which can be 2D or 3D, and grayscale or RGB (for 2D images only, see `channel_axis` parameter). There can be any number of channels (does not strictly have to be RGB).

patch_size

[int, optional] Size of patches used for denoising.

patch_distance

[int, optional] Maximal distance in pixels where to search patches used for denoising.

h

[float, optional] Cut-off distance (in gray levels). The higher h, the more permissive one is in accepting patches. A higher h results in a smoother image, at the expense of blurring features. For a Gaussian noise of standard deviation sigma, a rule of thumb is to choose the value of h to be sigma of slightly less.

fast_mode

[bool, optional] If True (default value), a fast version of the non-local means algorithm is used. If False, the original version of non-local means is used. See the Notes section for more details about the algorithms.

sigma

[float, optional] The standard deviation of the (Gaussian) noise. If provided, a more robust computation of patch weights is computed that takes the expected noise variance into account (see Notes below).

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`. Also see https://scikit-image.org/docs/dev/user_guide/data_types.html

channel_axis

[int or None, optional] If None, the image is assumed to be a grayscale (single channel) image. Otherwise, this parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns**result**

[ndarray] Denoised image, of same shape as `image`.

Notes

The non-local means algorithm is well suited for denoising images with specific textures. The principle of the algorithm is to average the value of a given pixel with values of other pixels in a limited neighborhood, provided that the *patches* centered on the other pixels are similar enough to the patch centered on the pixel of interest.

In the original version of the algorithm [?], corresponding to `fast=False`, the computational complexity is:

```
image.size * patch_size ** image.ndim * patch_distance ** image.ndim
```

Hence, changing the size of patches or their maximal distance has a strong effect on computing times, especially for 3-D images.

However, the default behavior corresponds to `fast_mode=True`, for which another version of non-local means [?] is used, corresponding to a complexity of:

```
image.size * patch_distance ** image.ndim
```

The computing time depends only weakly on the patch size, thanks to the computation of the integral of patches distances for a given shift, that reduces the number of operations [?]. Therefore, this algorithm executes faster than the classic algorithm (`fast_mode=False`), at the expense of using twice as much memory. This implementation has been proven to be more efficient compared to other alternatives, see e.g. [?].

Compared to the classic algorithm, all pixels of a patch contribute to the distance to another patch with the same weight, no matter their distance to the center of the patch. This coarser computation of the distance can result in a slightly poorer denoising performance. Moreover, for small images (images with a linear size that is only a few times the patch size), the classic algorithm can be faster due to boundary effects.

The image is padded using the `reflect` mode of `skimage.util.pad` before denoising.

If the noise standard deviation, `sigma`, is provided a more robust computation of patch weights is used. Subtracting the known noise variance from the computed patch distances improves the estimates of patch similarity, giving a moderate improvement to denoising performance [?]. It was also mentioned as an option for the fast variant of the algorithm in [?].

When `sigma` is provided, a smaller `h` should typically be used to avoid oversmoothing. The optimal value for `h` depends on the image content and noise level, but a reasonable starting point is `h = 0.8 * sigma` when `fast_mode` is `True`, or `h = 0.6 * sigma` when `fast_mode` is `False`.

References

[?], [?], [?], [?]

Examples

```
>>> a = np.zeros((40, 40))
>>> a[10:-10, 10:-10] = 1.
>>> rng = np.random.default_rng()
>>> a += 0.3 * rng.standard_normal(a.shape)
>>> denoised_a = denoise_nl_means(a, 7, 5, 0.1)
```

- Non-local means denoising for preserving textures
- Full tutorial on calibrating Denoisers Using J-Invariance

`skimage.restoration.denoise_tv_bregman(image, weight=5.0, max_num_iter=100, eps=0.001, isotropic=True, *, channel_axis=None)`

Perform total variation denoising using split-Bregman optimization.

Given f , a noisy image (input data), total variation denoising (also known as total variation regularization) aims to find an image u with less total variation than f , under the constraint that u remain similar to f . This can be expressed by the Rudin–Osher–Fatemi (ROF) minimization problem:

$$\min_u \sum_{i=0}^{N-1} \left(|\nabla u_i| + \frac{\lambda}{2} (f_i - u_i)^2 \right)$$

where λ is a positive parameter. The first term of this cost function is the total variation; the second term represents data fidelity. As $\lambda \rightarrow 0$, the total variation term dominates, forcing the solution to have smaller total variation, at the expense of looking less like the input data.

This code is an implementation of the split Bregman algorithm of Goldstein and Osher to solve the ROF problem ([?], [?], [?]).

Parameters

image

[ndarray] Input image to be denoised (converted using `img_as_float()`).

weight

[float, optional] Denoising weight. It is equal to $\frac{\lambda}{2}$. Therefore, the smaller the `weight`, the more denoising (at the expense of less similarity to `image`).

eps

[float, optional] Tolerance $\varepsilon > 0$ for the stop criterion: The algorithm stops when $\|u_n - u_{n-1}\|_2 < \varepsilon$.

max_num_iter

[int, optional] Maximal number of iterations used for the optimization.

isotropic

[boolean, optional] Switch between isotropic and anisotropic TV denoising.

channel_axis

[int or None, optional] If `None`, the image is assumed to be grayscale (single-channel). Otherwise, this parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

u

[ndarray] Denoised image.

See also:

`denoise_tv_chambolle`

Perform total variation denoising in nD.

Notes

Ensure that `channel_axis` parameter is set appropriately for color images.

The principle of total variation denoising is explained in [?]. It is about minimizing the total variation of an image, which can be roughly described as the integral of the norm of the image gradient. Total variation denoising tends to produce cartoon-like images, that is, piecewise-constant images.

References

[?], [?], [?], [?]

```
skimage.restoration.denoise_tv_chambolle(image, weight=0.1, eps=0.0002, max_num_iter=200, *,  
                                         channel_axis=None)
```

Perform total variation denoising in nD.

Given f , a noisy image (input data), total variation denoising (also known as total variation regularization) aims to find an image u with less total variation than f , under the constraint that u remain similar to f . This can be expressed by the Rudin–Osher–Fatemi (ROF) minimization problem:

$$\min_u \sum_{i=0}^{N-1} \left(|\nabla u_i| + \frac{\lambda}{2} (f_i - u_i)^2 \right)$$

where λ is a positive parameter. The first term of this cost function is the total variation; the second term represents data fidelity. As $\lambda \rightarrow 0$, the total variation term dominates, forcing the solution to have smaller total variation, at the expense of looking less like the input data.

This code is an implementation of the algorithm proposed by Chambolle in [?] to solve the ROF problem.

Parameters

image

[ndarray] Input image to be denoised. If its dtype is not float, it gets converted with `img_as_float()`.

weight

[float, optional] Denoising weight. It is equal to $\frac{1}{\lambda}$. Therefore, the greater the `weight`, the more denoising (at the expense of fidelity to `image`).

eps

[float, optional] Tolerance $\varepsilon > 0$ for the stop criterion (compares to absolute value of relative difference of the cost function E): The algorithm stops when $|E_{n-1} - E_n| < \varepsilon * E_0$.

max_num_iter

[int, optional] Maximal number of iterations used for the optimization.

channel_axis

[int or None, optional] If `None`, the image is assumed to be grayscale (single-channel). Otherwise, this parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

u

[ndarray] Denoised image.

See also:

[`denoise_tv_bregman`](#)

Perform total variation denoising using split-Bregman optimization.

Notes

Make sure to set the `channel_axis` parameter appropriately for color images.

The principle of total variation denoising is explained in [?]. It is about minimizing the total variation of an image, which can be roughly described as the integral of the norm of the image gradient. Total variation denoising tends to produce cartoon-like images, that is, piecewise-constant images.

References

[?], [?]

Examples

2D example on astronaut image:

```
>>> from skimage import color, data
>>> img = color.rgb2gray(data.astronaut())[:50, :50]
>>> rng = np.random.default_rng()
>>> img += 0.5 * img.std() * rng.standard_normal(img.shape)
>>> denoised_img = denoise_tv_chambolle(img, weight=60)
```

3D example on synthetic data:

```
>>> x, y, z = np.ogrid[0:20, 0:20, 0:20]
>>> mask = (x - 22)**2 + (y - 20)**2 + (z - 17)**2 < 8**2
>>> mask = mask.astype(float)
>>> rng = np.random.default_rng()
>>> mask += 0.2 * rng.standard_normal(mask.shape)
>>> res = denoise_tv_chambolle(mask, weight=100)
```

- *Denoising a picture*
- *Full tutorial on calibrating Denoisers Using J-Invariance*
- *Track solidification of a metallic alloy*

```
skimage.restoration.denoise_wavelet(image, sigma=None, wavelet='db1', mode='soft',
                                     wavelet_levels=None, convert2ycbcr=False, method='BayesShrink',
                                     rescale_sigma=True, *, channel_axis=None)
```

Perform wavelet denoising on an image.

Parameters

image

[ndarray ([M[, N[, ...P]][, C]]) of ints, uints or floats] Input data to be denoised. *image* can be of any numeric type, but it is cast into an ndarray of floats for the computation of the denoised image.

sigma

[float or list, optional] The noise standard deviation used when computing the wavelet detail coefficient threshold(s). When None (default), the noise standard deviation is estimated via the method in [?].

wavelet

[string, optional] The type of wavelet to perform and can be any of the options pywt.wavelist outputs. The default is 'db1'. For example, *wavelet* can be any of {'db2', 'haar', 'sym9'} and many more.

mode

[{'soft', 'hard'}, optional] An optional argument to choose the type of denoising performed. It noted that choosing soft thresholding given additive noise finds the best approximation of the original image.

wavelet_levels

[int or None, optional] The number of wavelet decomposition levels to use. The default is three less than the maximum number of possible decomposition levels.

convert2ycbcr

[bool, optional] If True and *channel_axis* is set, do the wavelet denoising in the YCbCr colorspace instead of the RGB color space. This typically results in better performance for RGB images.

method

[{'BayesShrink', 'VisuShrink'}, optional] Thresholding method to be used. The currently supported methods are "BayesShrink" [?] and "VisuShrink" [?]. Defaults to "BayesShrink".

rescale_sigma

[bool, optional] If False, no rescaling of the user-provided *sigma* will be performed. The default of True rescales sigma appropriately if the image is rescaled internally.

New in version 0.16: *rescale_sigma* was introduced in 0.16

channel_axis

[int or None, optional] If `None`, the image is assumed to be grayscale (single-channel). Otherwise, this parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns**out**

[ndarray] Denoised image.

Notes

The wavelet domain is a sparse representation of the image, and can be thought of similarly to the frequency domain of the Fourier transform. Sparse representations have most values zero or near-zero and truly random noise is (usually) represented by many small values in the wavelet domain. Setting all values below some threshold to 0 reduces the noise in the image, but larger thresholds also decrease the detail present in the image.

If the input is 3D, this function performs wavelet denoising on each color plane separately.

Changed in version 0.16: For floating point inputs, the original input range is maintained and there is no clipping applied to the output. Other input types will be converted to a floating point value in the range [-1, 1] or [0, 1] depending on the input image range. Unless `rescale_sigma = False`, any internal rescaling applied to the image will also be applied to `sigma` to maintain the same relative amplitude.

Many wavelet coefficient thresholding approaches have been proposed. By default, `denoise_wavelet` applies BayesShrink, which is an adaptive thresholding method that computes separate thresholds for each wavelet subband as described in [?].

If `method == "VisuShrink"`, a single “universal threshold” is applied to all wavelet detail coefficients as described in [?]. This threshold is designed to remove all Gaussian noise at a given `sigma` with high probability, but tends to produce images that appear overly smooth.

Although any of the wavelets from PyWavelets can be selected, the thresholding methods assume an orthogonal wavelet transform and may not choose the threshold appropriately for biorthogonal wavelets. Orthogonal wavelets are desirable because white noise in the input remains white noise in the subbands. Biorthogonal wavelets lead to colored noise in the subbands. Additionally, the orthogonal wavelets in PyWavelets are orthonormal so that noise variance in the subbands remains identical to the noise variance of the input. Example orthogonal wavelets are the Daubechies (e.g. ‘db2’) or symmlet (e.g. ‘sym2’) families.

References

[?], [?]

Examples

```
>>> from skimage import color, data
>>> img = img_as_float(data.astronaut())
>>> img = color.rgb2gray(img)
>>> rng = np.random.default_rng()
>>> img += 0.1 * rng.standard_normal(img.shape)
>>> img = np.clip(img, 0, 1)
>>> denoised_img = denoise_wavelet(img, sigma=0.1, rescale_sigma=True)
```

- *Calibrating Denoisers Using J-Invariance*
 - *Denoising a picture*
 - *Shift-invariant wavelet denoising*
 - *Wavelet denoising*
 - *Full tutorial on calibrating Denoisers Using J-Invariance*
-

`skimage.restoration.ellipsoid_kernel(shape, intensity)`

Create an ellipsoid kernel for `restoration.rolling_ball`.

Parameters

shape

[arraylike] Length of the principal axis of the ellipsoid (excluding the intensity axis). The kernel needs to have the same dimensionality as the image it will be applied to.

intensity

[int] Length of the intensity axis of the ellipsoid.

Returns

kernel

[ndarray] The kernel containing the surface intensity of the top half of the ellipsoid.

See also:

[`rolling_ball`](#)

- *Use rolling-ball algorithm for estimating background intensity*
-

```
skimage.restoration.estimate_sigma(image, average_sigmas=False, *, channel_axis=None)
```

Robust wavelet-based estimator of the (Gaussian) noise standard deviation.

Parameters

image

[ndarray] Image for which to estimate the noise standard deviation.

average_sigmas

[bool, optional] If true, average the channel estimates of *sigma*. Otherwise return a list of sigmas corresponding to each channel.

channel_axis

[int or None, optional] If *None*, the image is assumed to be grayscale (single-channel). Otherwise, this parameter indicates which axis of the array corresponds to channels.

New in version 0.19: *channel_axis* was added in 0.19.

Returns

sigma

[float or list] Estimated noise standard deviation(s). If *multichannel* is True and *average_sigmas* is False, a separate noise estimate for each channel is returned. Otherwise, the average of the individual channel estimates is returned.

Notes

This function assumes the noise follows a Gaussian distribution. The estimation algorithm is based on the median absolute deviation of the wavelet detail coefficients as described in section 4.2 of [?].

References

[?]

Examples

```
>>> import skimage.data
>>> from skimage import img_as_float
>>> img = img_as_float(skimage.data.camera())
>>> sigma = 0.1
>>> rng = np.random.default_rng()
>>> img = img + sigma * rng.standard_normal(img.shape)
>>> sigma_hat = estimate_sigma(img, channel_axis=None)
```

- Denoising a picture

- Non-local means denoising for preserving textures
 - Wavelet denoising
 - Full tutorial on calibrating Denoisers Using J-Invariance
-

`skimage.restoration.inpaint_biharmonic(image, mask, *, split_into_regions=False, channel_axis=None)`

Inpaint masked points in image with biharmonic equations.

Parameters

image

`[(M[, N[, ..., P]][, C]) ndarray]` Input image.

mask

`[(M[, N[, ..., P]]) ndarray]` Array of pixels to be inpainted. Have to be the same shape as one of the ‘image’ channels. Unknown pixels have to be represented with 1, known pixels - with 0.

split_into_regions

[boolean, optional] If True, inpainting is performed on a region-by-region basis. This is likely to be slower, but will have reduced memory requirements.

channel_axis

[int or None, optional] If None, the image is assumed to be a grayscale (single channel) image. Otherwise, this parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

out

`[(M[, N[, ..., P]][, C]) ndarray]` Input image with masked pixels inpainted.

References

[?], [?]

Examples

```
>>> img = np.tile(np.square(np.linspace(0, 1, 5)), (5, 1))
>>> mask = np.zeros_like(img)
>>> mask[2, 2:] = 1
>>> mask[1, 3:] = 1
>>> mask[0, 4:] = 1
>>> out = inpaint_biharmonic(img, mask)
```

- *Inpainting*

`skimage.restoration.richardson_lucy(image, psf, num_iter=50, clip=True, filter_epsilon=None)`

Richardson-Lucy deconvolution.

Parameters

`image`

[ndarray] Input degraded image (can be n-dimensional).

`psf`

[ndarray] The point spread function.

`num_iter`

[int, optional] Number of iterations. This parameter plays the role of regularisation.

`clip`

[boolean, optional] True by default. If true, pixel value of the result above 1 or under -1 are thresholded for skimage pipeline compatibility.

`filter_epsilon: float, optional`

Value below which intermediate results become 0 to avoid division by small numbers.

Returns

`im_deconv`

[ndarray] The deconvolved image.

References

[?]

Examples

```
>>> from skimage import img_as_float, data, restoration
>>> camera = img_as_float(data.camera())
>>> from scipy.signal import convolve2d
>>> psf = np.ones((5, 5)) / 25
>>> camera = convolve2d(camera, psf, 'same')
>>> rng = np.random.default_rng()
>>> camera += 0.1 * camera.std() * rng.standard_normal(camera.shape)
>>> deconvolved = restoration.richardson_lucy(camera, psf, 5)
```

- *Image Deconvolution*
-

`skimage.restoration.rolling_ball(image, *, radius=100, kernel=None, nansafe=False, num_threads=None)`

Estimate background intensity by rolling/translating a kernel.

This rolling ball algorithm estimates background intensity for a ndimage in case of uneven exposure. It is a generalization of the frequently used rolling ball algorithm [?].

Parameters

image

[ndarray] The image to be filtered.

radius

[int, optional] Radius of a ball shaped kernel to be rolled/translated in the image. Used if `kernel = None`.

kernel

[ndarray, optional] The kernel to be rolled/translated in the image. It must have the same number of dimensions as `image`. Kernel is filled with the intensity of the kernel at that position.

nansafe: bool, optional

If `False` (default) assumes that none of the values in `image` are `np.nan`, and uses a faster implementation.

num_threads: int, optional

The maximum number of threads to use. If `None` use the OpenMP default value; typically equal to the maximum number of virtual cores. Note: This is an upper limit to the number of threads. The exact number is determined by the system's OpenMP library.

Returns**background**

[ndarray] The estimated background of the image.

Notes

For the pixel that has its background intensity estimated (without loss of generality at `center`) the rolling ball method centers `kernel` under it and raises the kernel until the surface touches the image umbra at some `pos=(y, x)`. The background intensity is then estimated using the image intensity at that position (`image[pos]`) plus the difference of `kernel[center]` - `kernel[pos]`.

This algorithm assumes that dark pixels correspond to the background. If you have a bright background, invert the image before passing it to the function, e.g., using `utils.invert`. See the gallery example for details.

This algorithm is sensitive to noise (in particular salt-and-pepper noise). If this is a problem in your image, you can apply mild gaussian smoothing before passing the image to this function.

References

[?]

Examples

```
>>> import numpy as np
>>> from skimage import data
>>> from skimage.restoration import rolling_ball
>>> image = data.coins()
>>> background = rolling_ball(data.coins())
>>> filtered_image = image - background
```

```
>>> import numpy as np
>>> from skimage import data
>>> from skimage.restoration import rolling_ball, ellipsoid_kernel
>>> image = data.coins()
>>> kernel = ellipsoid_kernel((101, 101), 75)
>>> background = rolling_ball(data.coins(), kernel=kernel)
>>> filtered_image = image - background
```

- Use rolling-ball algorithm for estimating background intensity

`skimage.restoration.unsupervised_wiener(image, psf, reg=None, user_params=None, is_real=True, clip=True, *, rng=None)`

Unsupervised Wiener-Hunt deconvolution.

Return the deconvolution with a Wiener-Hunt approach, where the hyperparameters are automatically estimated. The algorithm is a stochastic iterative process (Gibbs sampler) described in the reference below. See also `wiener` function.

Parameters

image

[(M, N) ndarray] The input degraded image.

psf

[ndarray] The impulse response (input image's space) or the transfer function (Fourier space). Both are accepted. The transfer function is automatically recognized as being complex (`np.iscomplexobj(psf)`).

reg

[ndarray, optional] The regularisation operator. The Laplacian by default. It can be an impulse response or a transfer function, as for the psf.

user_params

[dict, optional] Dictionary of parameters for the Gibbs sampler. See below.

clip

[boolean, optional] True by default. If true, pixel values of the result above 1 or under -1 are thresholded for skimage pipeline compatibility.

rng

[{`numpy.random.Generator`, int}, optional] Pseudo-random number generator. By default, a PCG64 generator is used (see `numpy.random.default_rng()`). If `rng` is an int, it is used to seed the generator.

New in version 0.19.

Returns

x_postmean

[(M, N) ndarray] The deconvolved image (the posterior mean).

chains

[dict] The keys `noise` and `prior` contain the chain list of noise and prior precision respectively.

Other Parameters

The keys of ``user_params`` are:

threshold

[float] The stopping criterion: the norm of the difference between successive approximated solution (empirical mean of object samples, see Notes section). 1e-4 by default.

burnin

[int] The number of sample to ignore to start computation of the mean. 15 by default.

min_num_iter

[int] The minimum number of iterations. 30 by default.

max_num_iter

[int] The maximum number of iterations if `threshold` is not satisfied. 200 by default.

callback

[callable (None by default)] A user provided callable to which is passed, if the function exists, the current image sample for whatever purpose. The user can store the sample, or compute other moments than the mean. It has no influence on the algorithm execution and is only for inspection.

random_state

[DEPRECATED] Deprecated in favor of `rng`.

Deprecated since version 0.21.

Notes

The estimated image is design as the posterior mean of a probability law (from a Bayesian analysis). The mean is defined as a sum over all the possible images weighted by their respective probability. Given the size of the problem, the exact sum is not tractable. This algorithm use of MCMC to draw image under the posterior law. The practical idea is to only draw highly probable images since they have the biggest contribution to the mean. At the opposite, the less probable images are drawn less often since their contribution is low. Finally, the empirical mean of these samples give us an estimation of the mean, and an exact computation with an infinite sample set.

References

[?]

Examples

```
>>> from skimage import color, data, restoration
>>> img = color.rgb2gray(data.astronaut())
>>> from scipy.signal import convolve2d
>>> psf = np.ones((5, 5)) / 25
>>> img = convolve2d(img, psf, 'same')
>>> rng = np.random.default_rng()
>>> img += 0.1 * img.std() * rng.standard_normal(img.shape)
>>> deconvolved_img = restoration.unsupervised_wiener(img, psf)
```

- *Image Deconvolution*

`skimage.restoration.unwrap_phase(image, wrap_around=False, rng=None)`

Recover the original from a wrapped phase image.

From an image wrapped to lie in the interval [-pi, pi), recover the original, unwrapped image.

Parameters

image

[1D, 2D or 3D ndarray of floats, optionally a masked array] The values should be in the range [-pi, pi). If a masked array is provided, the masked entries will not be changed, and their values will not be used to guide the unwrapping of neighboring, unmasked values. Masked 1D arrays are not allowed, and will raise a *ValueError*.

wrap_around

[bool or sequence of bool, optional] When an element of the sequence is *True*, the unwrapping process will regard the edges along the corresponding axis of the image to be connected and use this connectivity to guide the phase unwrapping process. If only a single boolean is given, it will apply to all axes. Wrap around is not supported for 1D arrays.

rng

[{`numpy.random.Generator`, int}, optional] Pseudo-random number generator. By default, a PCG64 generator is used (see `numpy.random.default_rng()`). If *rng* is an int, it is used to seed the generator.

Unwrapping relies on a random initialization. This sets the PRNG to use to achieve deterministic behavior.

Returns

image_unwrapped

[array_like, double] Unwrapped image of the same shape as the input. If the input *image* was a masked array, the mask will be preserved.

Other Parameters

seed

[DEPRECATED] Deprecated in favor of *rng*.

Deprecated since version 0.21.

Raises

ValueError

If called with a masked 1D array or called with a 1D array and *wrap_around=True*.

References

[?], [?]

Examples

```
>>> c0, c1 = np.ogrid[-1:1:128j, -1:1:128j]
>>> image = 12 * np.pi * np.exp(-(c0**2 + c1**2))
>>> image_wrapped = np.angle(np.exp(1j * image))
>>> image_unwrapped = unwrap_phase(image_wrapped)
>>> np.std(image_unwrapped - image) < 1e-6 # A constant offset is normal
True
```

- *Phase Unwrapping*

`skimage.restoration.wiener(image, psf, balance, reg=None, is_real=True, clip=True)`

Wiener-Hunt deconvolution

Return the deconvolution with a Wiener-Hunt approach (i.e. with Fourier diagonalisation).

Parameters

image

[ndarray] Input degraded image (can be n-dimensional).

psf

[ndarray] Point Spread Function. This is assumed to be the impulse response (input image space) if the data-type is real, or the transfer function (Fourier space) if the data-type is complex. There is no constraints on the shape of the impulse response. The transfer function must be of shape (N_1, N_2, \dots, N_D) if `is_real` is `True`, $(N_1, N_2, \dots, N_D // 2 + 1)$ otherwise (see `np.fft.rfftn`).

balance

[float] The regularisation parameter value that tunes the balance between the data adequacy that improve frequency restoration and the prior adequacy that reduce frequency restoration (to avoid noise artifacts).

reg

[ndarray, optional] The regularisation operator. The Laplacian by default. It can be an impulse response or a transfer function, as for the `psf`. Shape constraint is the same as for the `psf` parameter.

is_real

[boolean, optional] True by default. Specify if `psf` and `reg` are provided with hermitian hypothesis, that is only half of the frequency plane is provided (due to the redundancy of

Fourier transform of real signal). It's apply only if `psf` and/or `reg` are provided as transfer function. For the hermitian property see `uft` module or `np.fft.rfftn`.

clip

[boolean, optional] True by default. If True, pixel values of the result above 1 or under -1 are thresholded for skimage pipeline compatibility.

Returns

`im_deconv`

[(M, N) ndarray] The deconvolved image.

Notes

This function applies the Wiener filter to a noisy and degraded image by an impulse response (or PSF). If the data model is

$$y = Hx + n$$

where n is noise, H the PSF and x the unknown original image, the Wiener filter is

$$\hat{x} = F^\dagger(|\Lambda_H|^2 + \lambda|\Lambda_D|^2)\Lambda_H^\dagger Fy$$

where F and F^\dagger are the Fourier and inverse Fourier transforms respectively, Λ_H the transfer function (or the Fourier transform of the PSF, see [Hunt] below) and Λ_D the filter to penalize the restored image frequencies (Laplacian by default, that is penalization of high frequency). The parameter λ tunes the balance between the data (that tends to increase high frequency, even those coming from noise), and the regularization.

These methods are then specific to a prior model. Consequently, the application or the true image nature must correspond to the prior model. By default, the prior model (Laplacian) introduce image smoothness or pixel correlation. It can also be interpreted as high-frequency penalization to compensate the instability of the solution with respect to the data (sometimes called noise amplification or “explosive” solution).

Finally, the use of Fourier space implies a circulant property of H , see [?].

References

[?], [?]

Examples

```
>>> from skimage import color, data, restoration
>>> img = color.rgb2gray(data.astronaut())
>>> from scipy.signal import convolve2d
>>> psf = np.ones((5, 5)) / 25
>>> img = convolve2d(img, psf, 'same')
>>> rng = np.random.default_rng()
>>> img += 0.1 * img.std() * rng.standard_normal(img.shape)
>>> deconvolved_img = restoration.wiener(img, psf, 0.1)
```

1.3.17 skimage.segmentation

<code>skimage.segmentation.active_contour</code>	Active contour model.
<code>skimage.segmentation.chan_vese</code>	Chan-Vese segmentation algorithm.
<code>skimage.segmentation.checkerboard_level_set</code>	Create a checkerboard level set with binary values.
<code>skimage.segmentation.clear_border</code>	Clear objects connected to the label image border.
<code>skimage.segmentation.disk_level_set</code>	Create a disk level set with binary values.
<code>skimage.segmentation.expand_labels</code>	Expand labels in label image by distance pixels without overlapping.
<code>skimage.segmentation.felzenszwalb</code>	Computes Felzenszwalb's efficient graph based image segmentation.
<code>skimage.segmentation.find_boundaries</code>	Return bool array where boundaries between labeled regions are True.
<code>skimage.segmentation.flood</code>	Mask corresponding to a flood fill.
<code>skimage.segmentation.flood_fill</code>	Perform flood filling on an image.
<code>skimage.segmentation.inverse_gaussian_gradient</code>	Inverse of gradient magnitude.
<code>skimage.segmentation.join_segmentations</code>	Return the join of the two input segmentations.
<code>skimage.segmentation.mark_boundaries</code>	Return image with boundaries between labeled regions highlighted.
<code>skimage.segmentation.morphological_chan_vese</code>	Morphological Active Contours without Edges (Mor-phACWE)
<code>skimage.segmentation.morphological_geodesic_active_contour</code>	Morphological Geodesic Active Contours (Mor-phGAC).
<code>skimage.segmentation.quickshift</code>	Segment image using quickshift clustering in Color-(x,y) space.
<code>skimage.segmentation.random_walker</code>	Random walker algorithm for segmentation from markers.
<code>skimage.segmentation.relabel_sequential</code>	Relabel arbitrary labels to {offset, .
<code>skimage.segmentation.slic</code>	Segments image using k-means clustering in Color-(x,y,z) space.
<code>skimage.segmentation.watershed</code>	Find watershed basins in <i>image</i> flooded from given <i>markers</i> .

`skimage.segmentation.active_contour`(*image*, *snake*, *alpha*=0.01, *beta*=0.1, *w_line*=0, *w_edge*=1, *gamma*=0.01, *max_px_move*=1.0, *max_num_iter*=2500, *convergence*=0.1, *, *boundary_condition*='periodic')

Active contour model.

Active contours by fitting snakes to features of images. Supports single and multichannel 2D images. Snakes can be periodic (for segmentation) or have fixed and/or free ends. The output snake has the same length as the input boundary. As the number of points is constant, make sure that the initial snake has enough points to capture the details of the final contour.

Parameters

`image`

[(N, M) or (N, M, 3) ndarray] Input image.

snake

[$(N, 2)$ ndarray] Initial snake coordinates. For periodic boundary conditions, endpoints must not be duplicated.

alpha

[float, optional] Snake length shape parameter. Higher values makes snake contract faster.

beta

[float, optional] Snake smoothness shape parameter. Higher values makes snake smoother.

w_line

[float, optional] Controls attraction to brightness. Use negative values to attract toward dark regions.

w_edge

[float, optional] Controls attraction to edges. Use negative values to repel snake from edges.

gamma

[float, optional] Explicit time stepping parameter.

max_px_move

[float, optional] Maximum pixel distance to move per iteration.

max_num_iter

[int, optional] Maximum iterations to optimize snake shape.

convergence

[float, optional] Convergence criteria.

boundary_condition

[string, optional] Boundary conditions for the contour. Can be one of ‘periodic’, ‘free’, ‘fixed’, ‘free-fixed’, or ‘fixed-free’. ‘periodic’ attaches the two ends of the snake, ‘fixed’ holds the end-points in place, and ‘free’ allows free movement of the ends. ‘fixed’ and ‘free’ can be combined by parsing ‘fixed-free’, ‘free-fixed’. Parsing ‘fixed-fixed’ or ‘free-free’ yields same behaviour as ‘fixed’ and ‘free’, respectively.

Returns

snake

[$(N, 2)$ ndarray] Optimised snake, same shape as input parameter.

References

[?]

Examples

```
>>> from skimage.draw import circle_perimeter
>>> from skimage.filters import gaussian
```

Create and smooth image:

```
>>> img = np.zeros((100, 100))
>>> rr, cc = circle_perimeter(35, 45, 25)
>>> img[rr, cc] = 1
>>> img = gaussian(img, 2, preserve_range=False)
```

Initialize spline:

```
>>> s = np.linspace(0, 2*np.pi, 100)
>>> init = 50 * np.array([np.sin(s), np.cos(s)]).T + 50
```

Fit spline to image:

```
>>> snake = active_contour(img, init, w_edge=0, w_line=1, coordinates='rc')
>>> dist = np.sqrt((45-snake[:, 0])**2 + (35-snake[:, 1])**2)
>>> int(np.mean(dist))
25
```

- Active Contour Model

`skimage.segmentation.chan_vese(image, mu=0.25, lambda1=1.0, lambda2=1.0, tol=0.001, max_num_iter=500, dt=0.5, init_level_set='checkerboard', extended_output=False)`

Chan-Vese segmentation algorithm.

Active contour model by evolving a level set. Can be used to segment objects without clearly defined boundaries.

Parameters

image

[(M, N) ndarray] Grayscale image to be segmented.

mu

[float, optional] ‘edge length’ weight parameter. Higher *mu* values will produce a ‘round’ edge, while values closer to zero will detect smaller objects.

lambda1

[float, optional] ‘difference from average’ weight parameter for the output region with value

‘True’. If it is lower than *lambda2*, this region will have a larger range of values than the other.

lambda2

[float, optional] ‘difference from average’ weight parameter for the output region with value ‘False’. If it is lower than *lambda1*, this region will have a larger range of values than the other.

tol

[float, positive, optional] Level set variation tolerance between iterations. If the L2 norm difference between the level sets of successive iterations normalized by the area of the image is below this value, the algorithm will assume that the solution was reached.

max_num_iter

[uint, optional] Maximum number of iterations allowed before the algorithm interrupts itself.

dt

[float, optional] A multiplication factor applied at calculations for each step, serves to accelerate the algorithm. While higher values may speed up the algorithm, they may also lead to convergence problems.

init_level_set

[str or (M, N) ndarray, optional] Defines the starting level set used by the algorithm. If a string is inputted, a level set that matches the image size will automatically be generated. Alternatively, it is possible to define a custom level set, which should be an array of float values, with the same shape as ‘image’. Accepted string values are as follows.

‘checkerboard’

the starting level set is defined as $\sin(x/5\pi)*\sin(y/5\pi)$, where x and y are pixel coordinates. This level set has fast convergence, but may fail to detect implicit edges.

‘disk’

the starting level set is defined as the opposite of the distance from the center of the image minus half of the minimum value between image width and image height. This is somewhat slower, but is more likely to properly detect implicit edges.

‘small disk’

the starting level set is defined as the opposite of the distance from the center of the image minus a quarter of the minimum value between image width and image height.

extended_output

[bool, optional] If set to True, the return value will be a tuple containing the three return values (see below). If set to False which is the default value, only the ‘segmentation’ array will be returned.

Returns

segmentation

[(M, N) ndarray, bool] Segmentation produced by the algorithm.

phi

[(M, N) ndarray of floats] Final level set computed by the algorithm.

energies

[list of floats] Shows the evolution of the ‘energy’ for each step of the algorithm. This should allow to check whether the algorithm converged.

Notes

The Chan-Vese Algorithm is designed to segment objects without clearly defined boundaries. This algorithm is based on level sets that are evolved iteratively to minimize an energy, which is defined by weighted values corresponding to the sum of differences intensity from the average value outside the segmented region, the sum of differences from the average value inside the segmented region, and a term which is dependent on the length of the boundary of the segmented region.

This algorithm was first proposed by Tony Chan and Luminita Vese, in a publication entitled “An Active Contour Model Without Edges” [?].

This implementation of the algorithm is somewhat simplified in the sense that the area factor ‘nu’ described in the original paper is not implemented, and is only suitable for grayscale images.

Typical values for *lambda1* and *lambda2* are 1. If the ‘background’ is very different from the segmented object in terms of distribution (for example, a uniform black image with figures of varying intensity), then these values should be different from each other.

Typical values for *mu* are between 0 and 1, though higher values can be used when dealing with shapes with very ill-defined contours.

The ‘energy’ which this algorithm tries to minimize is defined as the sum of the differences from the average within the region squared and weighed by the ‘lambda’ factors to which is added the length of the contour multiplied by the ‘mu’ factor.

Supports 2D grayscale images only, and does not implement the area term described in the original article.

References

[?], [?], [?]

- *Chan-Vese Segmentation*

```
skimage.segmentation.checkerboard_level_set(image_shape, square_size=5)
```

Create a checkerboard level set with binary values.

Parameters

image_shape

[tuple of positive integers] Shape of the image.

square_size

[int, optional] Size of the squares of the checkerboard. It defaults to 5.

Returns

out

[array with shape *image_shape*] Binary level set of the checkerboard.

See also:

[*disk_level_set*](#)

- *Morphological Snakes*
-

`skimage.segmentation.clear_border(labels, buffer_size=0, bgval=0, mask=None, *, out=None)`

Clear objects connected to the label image border.

Parameters

labels

[(M[, N[, ..., P]]) array of int or bool] Imaging data labels.

buffer_size

[int, optional] The width of the border examined. By default, only objects that touch the outside of the image are removed.

bgval

[float or int, optional] Cleared objects are set to this value.

mask

[ndarray of bool, same shape as *image*, optional.] Image data mask. Objects in labels image overlapping with False pixels of mask will be removed. If defined, the argument buffer_size will be ignored.

out

[ndarray] Array of the same shape as *labels*, into which the output is placed. By default, a new array is created.

Returns**out**

[(M[, N[, ..., P]]) array] Imaging data labels with cleared borders

Examples

```
>>> import numpy as np
>>> from skimage.segmentation import clear_border
>>> labels = np.array([[0, 0, 0, 0, 0, 0, 0, 1, 0],
...                     [1, 1, 0, 0, 1, 0, 0, 1, 0],
...                     [1, 1, 0, 1, 0, 1, 0, 0, 0],
...                     [0, 0, 0, 1, 1, 1, 1, 0, 0],
...                     [0, 1, 1, 1, 1, 1, 1, 1, 0],
...                     [0, 0, 0, 0, 0, 0, 0, 0, 0]])
>>> clear_border(labels)
array([[0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0, 0, 0],
       [0, 0, 1, 0, 1, 0, 0, 0, 0],
       [0, 0, 0, 1, 1, 1, 0, 0, 0],
       [0, 1, 1, 1, 1, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0]])
>>> mask = np.array([[0, 0, 1, 1, 1, 1, 1, 1, 1],
...                   [0, 0, 1, 1, 1, 1, 1, 1, 1],
...                   [1, 1, 1, 1, 1, 1, 1, 1, 1],
...                   [1, 1, 1, 1, 1, 1, 1, 1, 1],
...                   [1, 1, 1, 1, 1, 1, 1, 1, 1],
...                   [1, 1, 1, 1, 1, 1, 1, 1, 1]]).astype(bool)
>>> clear_border(labels, mask=mask)
array([[0, 0, 0, 0, 0, 0, 1, 0],
       [0, 0, 0, 1, 0, 0, 1, 0],
       [0, 0, 1, 0, 1, 0, 0, 0],
       [0, 0, 1, 1, 1, 1, 0, 0],
       [0, 1, 1, 1, 1, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0]])
```

- *Label image regions*
- *Colocalization metrics*
- *Measure fluorescence intensity at the nuclear envelope*

`skimage.segmentation.disk_level_set(image_shape, *, center=None, radius=None)`

Create a disk level set with binary values.

Parameters

image_shape

[tuple of positive integers] Shape of the image

center

[tuple of positive integers, optional] Coordinates of the center of the disk given in (row, column). If not given, it defaults to the center of the image.

radius

[float, optional] Radius of the disk. If not given, it is set to the 75% of the smallest image dimension.

Returns

out

[array with shape *image_shape*] Binary level set of the disk with the given *radius* and *center*.

See also:

[*checkerboard_level_set*](#)

`skimage.segmentation.expand_labels(label_image, distance=1)`

Expand labels in label image by *distance* pixels without overlapping.

Given a label image, `expand_labels` grows label regions (connected components) outwards by up to *distance* pixels without overflowing into neighboring regions. More specifically, each background pixel that is within Euclidean distance of $\leq \text{distance}$ pixels of a connected component is assigned the label of that connected component. Where multiple connected components are within *distance* pixels of a background pixel, the label value of the closest connected component will be assigned (see Notes for the case of multiple labels at equal distance).

Parameters

label_image

[ndarray of dtype int] label image

distance

[float] Euclidean distance in pixels by which to grow the labels. Default is one.

Returns

enlarged_labels

[ndarray of dtype int] Labeled array, where all connected regions have been enlarged

See also:

`skimage.measure.label()`, `skimage.segmentation.watershed()`,
`skimage.morphology.dilation()`

Notes

Where labels are spaced more than `distance` pixels are apart, this is equivalent to a morphological dilation with a disc or hyperball of radius `distance`. However, in contrast to a morphological dilation, `expand_labels` will not expand a label region into a neighboring region.

This implementation of `expand_labels` is derived from CellProfiler [?], where it is known as module “IdentifySecondaryObjects (Distance-N)” [?].

There is an important edge case when a pixel has the same distance to multiple regions, as it is not defined which region expands into that space. Here, the exact behavior depends on the upstream implementation of `scipy.ndimage.distance_transform_edt`.

References

[?], [?]

Examples

```
>>> labels = np.array([0, 1, 0, 0, 0, 0, 2])
>>> expand_labels(labels, distance=1)
array([1, 1, 1, 0, 0, 2, 2])
```

Labels will not overwrite each other:

```
>>> expand_labels(labels, distance=3)
array([1, 1, 1, 1, 2, 2, 2])
```

In case of ties, behavior is undefined, but currently resolves to the label closest to $(0, \dots)^{\text{ndim}}$ in lexicographical order.

```
>>> labels_tied = np.array([0, 1, 0, 2, 0])
>>> expand_labels(labels_tied, 1)
array([1, 1, 1, 2, 2])
>>> labels2d = np.array(
...     [[0, 1, 0, 0],
...      [2, 0, 0, 0],
...      [0, 3, 0, 0]])
...
>>> expand_labels(labels2d, 1)
array([[2, 1, 1, 0],
       [2, 2, 0, 0],
       [2, 3, 3, 0]])
```

- *Expand segmentation labels without overlap*

```
skimage.segmentation.felzenszwalb(image, scale=1, sigma=0.8, min_size=20, *, channel_axis=-1)
```

Computes Felsenszwalb's efficient graph based image segmentation.

Produces an oversegmentation of a multichannel (i.e. RGB) image using a fast, minimum spanning tree based clustering on the image grid. The parameter `scale` sets an observation level. Higher scale means less and larger segments. `sigma` is the diameter of a Gaussian kernel, used for smoothing the image prior to segmentation.

The number of produced segments as well as their size can only be controlled indirectly through `scale`. Segment size within an image can vary greatly depending on local contrast.

For RGB images, the algorithm uses the euclidean distance between pixels in color space.

Parameters

`image`

[`(width, height, 3)` or `(width, height)` ndarray] Input image.

`scale`

[float] Free parameter. Higher means larger clusters.

`sigma`

[float] Width (standard deviation) of Gaussian kernel used in preprocessing.

`min_size`

[int] Minimum component size. Enforced using postprocessing.

`channel_axis`

[int or None, optional] If None, the image is assumed to be a grayscale (single channel) image. Otherwise, this parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

`segment_mask`

[`(width, height)` ndarray] Integer mask indicating segment labels.

Notes

The k parameter used in the original paper renamed to *scale* here.

References

[?]

Examples

```
>>> from skimage.segmentation import felzenszwalb
>>> from skimage.data import coffee
>>> img = coffee()
>>> segments = felzenszwalb(img, scale=3.0, sigma=0.95, min_size=5)
```

- Comparison of segmentation and superpixel algorithms

`skimage.segmentation.find_boundaries(label_img, connectivity=1, mode='thick', background=0)`

Return bool array where boundaries between labeled regions are True.

Parameters

`label_img`

[array of int or bool] An array in which different regions are labeled with either different integers or boolean values.

`connectivity`

[int in {1, ..., $label_img.ndim$ }, optional] A pixel is considered a boundary pixel if any of its neighbors has a different label. *connectivity* controls which pixels are considered neighbors. A connectivity of 1 (default) means pixels sharing an edge (in 2D) or a face (in 3D) will be considered neighbors. A connectivity of $label_img.ndim$ means pixels sharing a corner will be considered neighbors.

`mode`

[string in {'thick', 'inner', 'outer', 'subpixel'}] How to mark the boundaries:

- thick: any pixel not completely surrounded by pixels of the same label (defined by *connectivity*) is marked as a boundary. This results in boundaries that are 2 pixels thick.
- inner: outline the pixels *just inside* of objects, leaving background pixels untouched.
- outer: outline pixels in the background around object boundaries. When two objects touch, their boundary is also marked.
- subpixel: return a doubled image, with pixels *between* the original pixels marked as boundary where appropriate.

background

[int, optional] For modes ‘inner’ and ‘outer’, a definition of a background label is required. See *mode* for descriptions of these two.

Returns**boundaries**

[array of bool, same shape as *label_img*] A bool image where True represents a boundary pixel. For *mode* equal to ‘subpixel’, *boundaries.shape[i]* is equal to $2 * \text{label_img}.\text{shape}[i] - 1$ for all *i* (a pixel is inserted in between all other pairs of pixels).

Examples

```
>>> labels = np.array([[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, 5, 5, 5, 0, 0],
...                     [0, 0, 1, 1, 1, 5, 5, 5, 0, 0],
...                     [0, 0, 1, 1, 1, 5, 5, 5, 0, 0],
...                     [0, 0, 1, 1, 1, 5, 5, 5, 0, 0],
...                     [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
...                     [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]], dtype=np.uint8)
>>> find_boundaries(labels, mode='thick').astype(np.uint8)
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 1, 1, 1, 0, 0, 0],
       [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
       [0, 1, 1, 1, 1, 1, 0, 1, 1, 0],
       [0, 1, 1, 0, 1, 1, 0, 1, 1, 0],
       [0, 1, 1, 1, 1, 1, 0, 1, 1, 0],
       [0, 0, 1, 1, 1, 1, 1, 1, 1, 0],
       [0, 0, 0, 0, 1, 1, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]], dtype=uint8)
>>> find_boundaries(labels, mode='inner').astype(np.uint8)
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 1, 1, 1, 0, 0, 0],
       [0, 0, 1, 1, 1, 1, 0, 1, 0, 0],
       [0, 0, 1, 0, 1, 1, 0, 1, 0, 0],
       [0, 0, 1, 1, 1, 1, 0, 1, 0, 0],
       [0, 0, 0, 0, 1, 1, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]], dtype=uint8)
>>> find_boundaries(labels, mode='outer').astype(np.uint8)
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 1, 0, 0, 1, 0],
       [0, 1, 0, 0, 1, 1, 0, 0, 1, 0],
       [0, 1, 0, 0, 1, 1, 0, 0, 1, 0],
       [0, 1, 0, 0, 1, 1, 0, 0, 1, 0],
```

(continues on next page)

(continued from previous page)

```
[0, 0, 1, 1, 1, 0, 0, 1, 0],
[0, 0, 0, 0, 1, 1, 1, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0]], dtype=uint8)
>>> labels_small = labels[:, ::2, ::3]
>>> labels_small
array([[0, 0, 0, 0],
       [0, 0, 5, 0],
       [0, 1, 5, 0],
       [0, 0, 5, 0],
       [0, 0, 0, 0]], dtype=uint8)
>>> find_boundaries(labels_small, mode='subpixel').astype(np.uint8)
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 1, 1, 0],
       [0, 0, 0, 1, 0, 1, 0],
       [0, 1, 1, 0, 1, 0, 0],
       [0, 1, 0, 1, 0, 1, 0],
       [0, 1, 1, 1, 0, 1, 0],
       [0, 0, 0, 1, 0, 1, 0],
       [0, 0, 0, 1, 1, 1, 0],
       [0, 0, 0, 0, 0, 0, 0]], dtype=uint8)
>>> bool_image = np.array([[False, False, False, False, False],
...                         [False, False, False, False, False],
...                         [False, False, True, True, True],
...                         [False, False, True, True, True],
...                         [False, False, True, True, True]],
...                         dtype=bool)
>>> find_boundaries(bool_image)
array([[False, False, False, False, False],
       [False, False, True, True, True],
       [False, True, True, True, True],
       [False, True, True, False, False],
       [False, True, True, False, False]])
```

`skimage.segmentation.flood(image, seed_point, *, footprint=None, connectivity=None, tolerance=None)`

Mask corresponding to a flood fill.

Starting at a specific `seed_point`, connected points equal or within `tolerance` of the seed value are found.

Parameters

`image`

[ndarray] An n-dimensional array.

`seed_point`

[tuple or int] The point in `image` used as the starting point for the flood fill. If the image is 1D, this point may be given as an integer.

`footprint`

[ndarray, optional] The footprint (structuring element) used to determine the neighborhood of each evaluated pixel. It must contain only 1's and 0's, have the same number of dimensions as *image*. If not given, all adjacent pixels are considered as part of the neighborhood (fully connected).

connectivity

[int, optional] A number used to determine the neighborhood of each evaluated pixel. Adjacent pixels whose squared distance from the center is less than or equal to *connectivity* are considered neighbors. Ignored if *footprint* is not None.

tolerance

[float or int, optional] If None (default), adjacent values must be strictly equal to the initial value of *image* at *seed_point*. This is fastest. If a value is given, a comparison will be done at every point and if within tolerance of the initial value will also be filled (inclusive).

Returns

mask

[ndarray] A Boolean array with the same shape as *image* is returned, with True values for areas connected to and equal (or within tolerance of) the seed point. All other values are False.

Notes

The conceptual analogy of this operation is the ‘paint bucket’ tool in many raster graphics programs. This function returns just the mask representing the fill.

If indices are desired rather than masks for memory reasons, the user can simply run `numpy.nonzero` on the result, save the indices, and discard this mask.

Examples

```
>>> from skimage.morphology import flood
>>> image = np.zeros((4, 7), dtype=int)
>>> image[1:3, 1:3] = 1
>>> image[3, 0] = 1
>>> image[1:3, 4:6] = 2
>>> image[3, 6] = 3
>>> image
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 1, 1, 0, 2, 2, 0],
       [0, 1, 1, 0, 2, 2, 0],
       [1, 0, 0, 0, 0, 0, 3]])
```

Fill connected ones with 5, with full connectivity (diagonals included):

```
>>> mask = flood(image, (1, 1))
>>> image_flooded = image.copy()
>>> image_flooded[mask] = 5
```

(continues on next page)

(continued from previous page)

```
>>> image_flooded
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 5, 5, 0, 2, 2, 0],
       [0, 5, 5, 0, 2, 2, 0],
       [5, 0, 0, 0, 0, 0, 3]])
```

Fill connected ones with 5, excluding diagonal points (connectivity 1):

```
>>> mask = flood(image, (1, 1), connectivity=1)
>>> image_flooded = image.copy()
>>> image_flooded[mask] = 5
>>> image_flooded
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 5, 5, 0, 2, 2, 0],
       [0, 5, 5, 0, 2, 2, 0],
       [1, 0, 0, 0, 0, 0, 3]])
```

Fill with a tolerance:

```
>>> mask = flood(image, (0, 0), tolerance=1)
>>> image_flooded = image.copy()
>>> image_flooded[mask] = 5
>>> image_flooded
array([[5, 5, 5, 5, 5, 5, 5],
       [5, 5, 5, 5, 2, 2, 5],
       [5, 5, 5, 5, 2, 2, 5],
       [5, 5, 5, 5, 5, 5, 3]])
```

- *Flood Fill*

`skimage.segmentation.flood_fill(image, seed_point, new_value, *, footprint=None, connectivity=None, tolerance=None, in_place=False)`

Perform flood filling on an image.

Starting at a specific `seed_point`, connected points equal or within `tolerance` of the seed value are found, then set to `new_value`.

Parameters

`image`

[ndarray] An n-dimensional array.

`seed_point`

[tuple or int] The point in `image` used as the starting point for the flood fill. If the image is 1D, this point may be given as an integer.

`new_value`

[*image* type] New value to set the entire fill. This must be chosen in agreement with the dtype of *image*.

footprint

[ndarray, optional] The footprint (structuring element) used to determine the neighborhood of each evaluated pixel. It must contain only 1's and 0's, have the same number of dimensions as *image*. If not given, all adjacent pixels are considered as part of the neighborhood (fully connected).

connectivity

[int, optional] A number used to determine the neighborhood of each evaluated pixel. Adjacent pixels whose squared distance from the center is less than or equal to *connectivity* are considered neighbors. Ignored if *footprint* is not None.

tolerance

[float or int, optional] If None (default), adjacent values must be strictly equal to the value of *image* at *seed_point* to be filled. This is fastest. If a tolerance is provided, adjacent points with values within plus or minus tolerance from the seed point are filled (inclusive).

in_place

[bool, optional] If True, flood filling is applied to *image* in place. If False, the flood filled result is returned without modifying the input *image* (default).

Returns

filled

[ndarray] An array with the same shape as *image* is returned, with values in areas connected to and equal (or within tolerance of) the seed point replaced with *new_value*.

Notes

The conceptual analogy of this operation is the ‘paint bucket’ tool in many raster graphics programs.

Examples

```
>>> from skimage.morphology import flood_fill
>>> image = np.zeros((4, 7), dtype=int)
>>> image[1:3, 1:3] = 1
>>> image[3, 0] = 1
>>> image[1:3, 4:6] = 2
>>> image[3, 6] = 3
>>> image
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 1, 1, 0, 2, 2, 0],
       [0, 1, 1, 0, 2, 2, 0],
       [1, 0, 0, 0, 0, 0, 3]])
```

Fill connected ones with 5, with full connectivity (diagonals included):

```
>>> flood_fill(image, (1, 1), 5)
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 5, 5, 0, 2, 2, 0],
       [0, 5, 5, 0, 2, 2, 0],
       [5, 0, 0, 0, 0, 0, 3]])
```

Fill connected ones with 5, excluding diagonal points (connectivity 1):

```
>>> flood_fill(image, (1, 1), 5, connectivity=1)
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 5, 5, 0, 2, 2, 0],
       [0, 5, 5, 0, 2, 2, 0],
       [1, 0, 0, 0, 0, 0, 3]])
```

Fill with a tolerance:

```
>>> flood_fill(image, (0, 0), 5, tolerance=1)
array([[5, 5, 5, 5, 5, 5, 5],
       [5, 5, 5, 5, 2, 2, 5],
       [5, 5, 5, 5, 2, 2, 5],
       [5, 5, 5, 5, 5, 5, 3]])
```

- *Flood Fill*

skimage.segmentation.inverse_gaussian_gradient(*image*, *alpha*=100.0, *sigma*=5.0)

Inverse of gradient magnitude.

Compute the magnitude of the gradients in the image and then inverts the result in the range [0, 1]. Flat areas are assigned values close to 1, while areas close to borders are assigned values close to 0.

This function or a similar one defined by the user should be applied over the image as a preprocessing step before calling *morphological_geodesic_active_contour*.

Parameters

image

[*M, N* or *L, M, N* array] Grayscale image or volume.

alpha

[float, optional] Controls the steepness of the inversion. A larger value will make the transition between the flat areas and border areas steeper in the resulting array.

sigma

[float, optional] Standard deviation of the Gaussian filter applied over the image.

Returns

gimage

[(M, N) or (L, M, N) array] Preprocessed image (or volume) suitable for *morphological_geodesic_active_contour*.

- *Morphological Snakes*
 - *Evaluating segmentation metrics*
-

`skimage.segmentation.join_segmentations(s1, s2, return_mapping: bool = False)`

Return the join of the two input segmentations.

The join J of S1 and S2 is defined as the segmentation in which two voxels are in the same segment if and only if they are in the same segment in *both* S1 and S2.

Parameters

s1, s2

[numpy arrays] s1 and s2 are label fields of the same shape.

return_mapping

[bool, optional] If true, return mappings for joined segmentation labels to the original labels.

Returns

j

[numpy array] The join segmentation of s1 and s2.

map_j_to_s1

[ArrayMap, optional] Mapping from labels of the joined segmentation j to labels of s1.

map_j_to_s2

[ArrayMap, optional] Mapping from labels of the joined segmentation j to labels of s2.

Examples

```
>>> from skimage.segmentation import join_segmentations
>>> s1 = np.array([[0, 0, 1, 1],
...                 [0, 2, 1, 1],
...                 [2, 2, 2, 1]])
>>> s2 = np.array([[0, 1, 1, 0],
...                 [0, 1, 1, 0],
...                 [0, 1, 1, 1]])
>>> join_segmentations(s1, s2)
```

(continues on next page)

(continued from previous page)

```

array([[0, 1, 3, 2],
       [0, 5, 3, 2],
       [4, 5, 5, 3]])
>>> j, m1, m2 = join_segmentations(s1, s2, return_mapping=True)
>>> m1
ArrayMap(array([0, 1, 2, 3, 4, 5]), array([0, 0, 1, 1, 2, 2]))
>>> np.all(m1[j] == s1)
True
>>> np.all(m2[j] == s2)
True

```

- Find the intersection of two segmentations

`skimage.segmentation.mark_boundaries(image, label_img, color=(1, 1, 0), outline_color=None, mode='outer', background_label=0)`

Return image with boundaries between labeled regions highlighted.

Parameters

image

[(M, N[, 3]) array] Grayscale or RGB image.

label_img

[(M, N) array of int] Label array where regions are marked by different integer values.

color

[length-3 sequence, optional] RGB color of boundaries in the output image.

outline_color

[length-3 sequence, optional] RGB color surrounding boundaries in the output image. If None, no outline is drawn.

mode

[string in {‘thick’, ‘inner’, ‘outer’, ‘subpixel’}, optional] The mode for finding boundaries.

background_label

[int, optional] Which label to consider background (this is only useful for modes `inner` and `outer`).

Returns

marked

[(M, N, 3) array of float] An image in which the boundaries between labels are superimposed on the original image.

See also:

`find_boundaries`

- *Apply maskSLIC vs SLIC*
 - *Comparison of segmentation and superpixel algorithms*
 - *RAG Merging*
 - *Trainable segmentation using local features and random forests*
 - *Evaluating segmentation metrics*
-

```
skimage.segmentation.morphological_chan_vese(image, num_iter, init_level_set='checkerboard',
                                             smoothing=1, lambda1=1, lambda2=1,
                                             iter_callback=<function <lambda>>)
```

Morphological Active Contours without Edges (MorphACWE)

Active contours without edges implemented with morphological operators. It can be used to segment objects in images and volumes without well defined borders. It is required that the inside of the object looks different on average than the outside (i.e., the inner area of the object should be darker or lighter than the outer area on average).

Parameters

image

[(M, N) or (L, M, N) array] Grayscale image or volume to be segmented.

num_iter

[uint] Number of num_iter to run

init_level_set

[str, (M, N) array, or (L, M, N) array] Initial level set. If an array is given, it will be binarized and used as the initial level set. If a string is given, it defines the method to generate a reasonable initial level set with the shape of the *image*. Accepted values are ‘checkerboard’ and ‘disk’. See the documentation of `checkerboard_level_set` and `disk_level_set` respectively for details about how these level sets are created.

smoothing

[uint, optional] Number of times the smoothing operator is applied per iteration. Reasonable values are around 1-4. Larger values lead to smoother segmentations.

lambda1

[float, optional] Weight parameter for the outer region. If *lambda1* is larger than *lambda2*, the outer region will contain a larger range of values than the inner region.

lambda2

[float, optional] Weight parameter for the inner region. If *lambda2* is larger than *lambda1*, the inner region will contain a larger range of values than the outer region.

iter_callback

[function, optional] If given, this function is called once per iteration with the current level set as the only argument. This is useful for debugging or for plotting intermediate results during the evolution.

Returns**out**

[(M, N) or (L, M, N) array] Final segmentation (i.e., the final level set)

See also:

[*disk_level_set*](#), [*checkerboard_level_set*](#)

Notes

This is a version of the Chan-Vese algorithm that uses morphological operators instead of solving a partial differential equation (PDE) for the evolution of the contour. The set of morphological operators used in this algorithm are proved to be infinitesimally equivalent to the Chan-Vese PDE (see [?]). However, morphological operators do not suffer from the numerical stability issues typically found in PDEs (it is not necessary to find the right time step for the evolution), and are computationally faster.

The algorithm and its theoretical derivation are described in [?].

References

[?]

- *Morphological Snakes*

```
skimage.segmentation.morphological_geodesic_active_contour(gimage, num_iter, init_level_set='disk',
                                                       smoothing=1, threshold='auto',
                                                       balloon=0, iter_callback=<function
                                                       <lambda>>)
```

Morphological Geodesic Active Contours (MorphGAC).

Geodesic active contours implemented with morphological operators. It can be used to segment objects with visible but noisy, cluttered, broken borders.

Parameters

gimage

[(M, N) or (L, M, N) array] Preprocessed image or volume to be segmented. This is very rarely the original image. Instead, this is usually a preprocessed version of the original image that enhances and highlights the borders (or other structures) of the object to segment. `morphological_geodesic_active_contour` will try to stop the contour evolution in areas where `gimage` is small. See `morphsnakes.inverse_gaussian_gradient` as an example function to perform this preprocessing. Note that the quality of `morphological_geodesic_active_contour` might greatly depend on this preprocessing.

num_iter

[uint] Number of num_iter to run.

init_level_set

[str, (M, N) array, or (L, M, N) array] Initial level set. If an array is given, it will be binarized and used as the initial level set. If a string is given, it defines the method to generate a reasonable initial level set with the shape of the `image`. Accepted values are ‘checkerboard’ and ‘disk’. See the documentation of `checkerboard_level_set` and `disk_level_set` respectively for details about how these level sets are created.

smoothing

[uint, optional] Number of times the smoothing operator is applied per iteration. Reasonable values are around 1-4. Larger values lead to smoother segmentations.

threshold

[float, optional] Areas of the image with a value smaller than this threshold will be considered borders. The evolution of the contour will stop in these areas.

balloon

[float, optional] Balloon force to guide the contour in non-informative areas of the image, i.e., areas where the gradient of the image is too small to push the contour towards a border. A negative value will shrink the contour, while a positive value will expand the contour in these areas. Setting this to zero will disable the balloon force.

iter_callback

[function, optional] If given, this function is called once per iteration with the current level set as the only argument. This is useful for debugging or for plotting intermediate results during the evolution.

Returns**out**

[(M, N) or (L, M, N) array] Final segmentation (i.e., the final level set)

See also:

`inverse_gaussian_gradient`, `disk_level_set`, `checkerboard_level_set`

Notes

This is a version of the Geodesic Active Contours (GAC) algorithm that uses morphological operators instead of solving partial differential equations (PDEs) for the evolution of the contour. The set of morphological operators used in this algorithm are proved to be infinitesimally equivalent to the GAC PDEs (see [?]). However, morphological operators do not suffer from the numerical stability issues typically found in PDEs (e.g., it is not necessary to find the right time step for the evolution), and are computationally faster.

The algorithm and its theoretical derivation are described in [?].

References

[?]

- *Morphological Snakes*
- *Evaluating segmentation metrics*

```
skimage.segmentation.quickshift(image, ratio=1.0, kernel_size=5, max_dist=10, return_tree=False,
                                 sigma=0, convert2lab=True, rng=42, *, channel_axis=-1)
```

Segment image using quickshift clustering in Color-(x,y) space.

Produces an oversegmentation of the image using the quickshift mode-seeking algorithm.

Parameters

image

[(width, height, channels) ndarray] Input image. The axis corresponding to color channels can be specified via the *channel_axis* argument.

ratio

[float, optional, between 0 and 1] Balances color-space proximity and image-space proximity. Higher values give more weight to color-space.

kernel_size

[float, optional] Width of Gaussian kernel used in smoothing the sample density. Higher means fewer clusters.

max_dist

[float, optional] Cut-off point for data distances. Higher means fewer clusters.

return_tree

[bool, optional] Whether to return the full segmentation hierarchy tree and distances.

sigma

[float, optional] Width for Gaussian smoothing as preprocessing. Zero means no smoothing.

convert2lab

[bool, optional] Whether the input should be converted to Lab colorspace prior to segmentation. For this purpose, the input is assumed to be RGB.

rng

[{`numpy.random.Generator`, int}, optional] Pseudo-random number generator. By default, a PCG64 generator is used (see `numpy.random.default_rng()`). If `rng` is an int, it is used to seed the generator.

The PRNG is used to break ties, and is seeded with 42 by default.

channel_axis

[int, optional] The axis of *image* corresponding to color channels. Defaults to the last axis.

Returns

segment_mask

[(width, height) ndarray] Integer mask indicating segment labels.

Other Parameters

random_seed

[DEPRECATED] Deprecated in favor of `rng`.

Deprecated since version 0.21.

Notes

The authors advocate to convert the image to Lab color space prior to segmentation, though this is not strictly necessary. For this to work, the image must be given in RGB format.

References

[?]

- *Comparison of segmentation and superpixel algorithms*

```
skimage.segmentation.random_walker(data, labels, beta=130, mode='cg_j', tol=0.001, copy=True,
                                    return_full_prob=False, spacing=None, *, prob_tol=0.001,
                                    channel_axis=None)
```

Random walker algorithm for segmentation from markers.

Random walker algorithm is implemented for gray-level or multichannel images.

Parameters

data

[array_like] Image to be segmented in phases. Gray-level *data* can be two- or three-dimensional; multichannel data can be three- or four- dimensional with *channel_axis* specifying the dimension containing channels. Data spacing is assumed isotropic unless the *spacing* keyword argument is used.

labels

[array of ints, of same shape as *data* without channels dimension] Array of seed markers labeled with different positive integers for different phases. Zero-labeled pixels are unlabeled pixels. Negative labels correspond to inactive pixels that are not taken into account (they are removed from the graph). If labels are not consecutive integers, the labels array will be transformed so that labels are consecutive. In the multichannel case, *labels* should have the same shape as a single channel of *data*, i.e. without the final dimension denoting channels.

beta

[float, optional] Penalization coefficient for the random walker motion (the greater *beta*, the more difficult the diffusion).

mode

[string, available options {‘cg’, ‘cg_j’, ‘cg_mg’, ‘bf’}] Mode for solving the linear system in the random walker algorithm.

- ‘bf’ (brute force): an LU factorization of the Laplacian is computed. This is fast for small images (<1024x1024), but very slow and memory-intensive for large images (e.g., 3-D volumes).
- ‘cg’ (conjugate gradient): the linear system is solved iteratively using the Conjugate Gradient method from `scipy.sparse.linalg`. This is less memory-consuming than the brute force method for large images, but it is quite slow.
- ‘cg_j’ (conjugate gradient with Jacobi preconditioner): the Jacobi preconditioner is applied during the Conjugate gradient method iterations. This may accelerate the convergence of the ‘cg’ method.
- ‘cg_mg’ (conjugate gradient with multigrid preconditioner): a preconditioner is computed using a multigrid solver, then the solution is computed with the Conjugate Gradient method. This mode requires that the `pyamg` module is installed.

tol

[float, optional] Tolerance to achieve when solving the linear system using the conjugate gradient based modes (‘cg’, ‘cg_j’ and ‘cg_mg’).

copy

[bool, optional] If copy is False, the *labels* array will be overwritten with the result of the segmentation. Use `copy=False` if you want to save on memory.

return_full_prob

[bool, optional] If True, the probability that a pixel belongs to each of the labels will be returned, instead of only the most likely label.

spacing

[iterable of floats, optional] Spacing between voxels in each spatial dimension. If *None*, then the spacing between pixels/voxels in each dimension is assumed 1.

prob_tol

[float, optional] Tolerance on the resulting probability to be in the interval [0, 1]. If the tolerance is not satisfied, a warning is displayed.

channel_axis

[int or None, optional] If None, the image is assumed to be a grayscale (single channel) image. Otherwise, this parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

output

[ndarray]

- If `return_full_prob` is False, array of ints of same shape and data type as `labels`, in which each pixel has been labeled according to the marker that reached the pixel first by anisotropic diffusion.
- If `return_full_prob` is True, array of floats of shape (`nlabels`, `labels.shape`). `output[label_nb, i, j]` is the probability that label `label_nb` reaches the pixel (i, j) first.

See also:

`skimage.morphology.watershed`

watershed segmentation A segmentation algorithm based on mathematical morphology and “flooding” of regions from markers.

Notes

Multichannel inputs are scaled with all channel data combined. Ensure all channels are separately normalized prior to running this algorithm.

The `spacing` argument is specifically for anisotropic datasets, where data points are spaced differently in one or more spatial dimensions. Anisotropic data is commonly encountered in medical imaging.

The algorithm was first proposed in [?].

The algorithm solves the diffusion equation at infinite times for sources placed on markers of each phase in turn. A pixel is labeled with the phase that has the greatest probability to diffuse first to the pixel.

The diffusion equation is solved by minimizing $x \cdot T L x$ for each phase, where L is the Laplacian of the weighted graph of the image, and x is the probability that a marker of the given phase arrives first at a pixel by diffusion ($x=1$ on markers of the phase, $x=0$ on the other markers, and the other coefficients are looked for). Each pixel is attributed the label for which it has a maximal value of x . The Laplacian L of the image is defined as:

- $L_{ii} = d_i$, the number of neighbors of pixel i (the degree of i)
- $L_{ij} = -w_{ij}$ if i and j are adjacent pixels

The weight w_{ij} is a decreasing function of the norm of the local gradient. This ensures that diffusion is easier between pixels of similar values.

When the Laplacian is decomposed into blocks of marked and unmarked pixels:

$$\begin{aligned} L &= M B T \\ B A \end{aligned}$$

with first indices corresponding to marked pixels, and then to unmarked pixels, minimizing $x.T L x$ for one phase amount to solving:

$$A x = -B x_m$$

where $x_m = 1$ on markers of the given phase, and 0 on other markers. This linear system is solved in the algorithm using a direct method for small images, and an iterative method for larger images.

References

[?]

Examples

```
>>> rng = np.random.default_rng()
>>> a = np.zeros((10, 10)) + 0.2 * rng.random((10, 10))
>>> a[5:8, 5:8] += 1
>>> b = np.zeros_like(a, dtype=np.int32)
>>> b[3, 3] = 1 # Marker for first phase
>>> b[6, 6] = 2 # Marker for second phase
>>> random_walker(a, b)
array([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 2, 2, 2, 1, 1],
       [1, 1, 1, 1, 2, 2, 2, 1, 1, 1],
       [1, 1, 1, 1, 2, 2, 2, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]], dtype=int32)
```

- *Random walker segmentation*

`skimage.segmentation.relabel_sequential(label_field, offset=1)`

Relabel arbitrary labels to $\{offset, \dots, offset + \text{number_of_labels}\}$.

This function also returns the forward map (mapping the original labels to the reduced labels) and the inverse map (mapping the reduced labels back to the original ones).

Parameters

label_field

[numpy array of int, arbitrary shape] An array of labels, which must be non-negative integers.

offset

[int, optional] The return labels will start at *offset*, which should be strictly positive.

Returns

relabeled

[numpy array of int, same shape as *label_field*] The input label field with labels mapped to $\{ \text{offset}, \dots, \text{number_of_labels} + \text{offset} - 1 \}$. The data type will be the same as *label_field*, except when offset + number_of_labels causes overflow of the current data type.

forward_map

[ArrayMap] The map from the original label space to the returned label space. Can be used to re-apply the same mapping. See examples for usage. The output data type will be the same as *relabeled*.

inverse_map

[ArrayMap] The map from the new label space to the original space. This can be used to reconstruct the original label field from the relabeled one. The output data type will be the same as *label_field*.

Notes

The label 0 is assumed to denote the background and is never remapped.

The forward map can be extremely big for some inputs, since its length is given by the maximum of the label field. However, in most situations, *label_field.max()* is much smaller than *label_field.size*, and in these cases the forward map is guaranteed to be smaller than either the input or output images.

Examples

```
>>> from skimage.segmentation import relabel_sequential
>>> label_field = np.array([1, 1, 5, 5, 8, 99, 42])
>>> relab, fw, inv = relabel_sequential(label_field)
>>> relab
array([1, 1, 2, 2, 3, 5, 4])
>>> print(fw)
ArrayMap:
  1 → 1
  5 → 2
  8 → 3
```

(continues on next page)

(continued from previous page)

Segments image using k-means clustering in Color-(x,y,z) space.

Parameters

image

[2D, 3D or 4D ndarray] Input image, which can be 2D or 3D, and grayscale or multichannel (see *channel_axis* parameter). Input image must either be NaN-free or the NaN's must be masked out

n segments

[int, optional] The (approximate) number of labels in the segmented output image.

compactness

[float, optional] Balances color proximity and space proximity. Higher values give more weight to space proximity, making superpixel shapes more square/cubic. In SLICO mode, this is the initial compactness. This parameter depends strongly on image contrast and on the shapes of objects in the image. We recommend exploring possible values on a log scale, e.g., 0.01, 0.1, 1, 10, 100, before refining around a chosen value.

max_num_iter

[int, optional] Maximum number of iterations of k-means.

sigma

[float or array-like of floats, optional] Width of Gaussian smoothing kernel for pre-processing for each dimension of the image. The same sigma is applied to each dimension

in case of a scalar value. Zero means no smoothing. Note that *sigma* is automatically scaled if it is scalar and if a manual voxel spacing is provided (see Notes section). If *sigma* is array-like, its size must match *image*'s number of spatial dimensions.

spacing

[array-like of floats, optional] The voxel spacing along each spatial dimension. By default, *slic* assumes uniform spacing (same voxel resolution along each spatial dimension). This parameter controls the weights of the distances along the spatial dimensions during k-means clustering.

convert2lab

[bool, optional] Whether the input should be converted to Lab colorspace prior to segmentation. The input image *must* be RGB. Highly recommended. This option defaults to True when *channel_axis` is not None *and* ``image.shape[-1] == 3.*

enforce_connectivity

[bool, optional] Whether the generated segments are connected or not

min_size_factor

[float, optional] Proportion of the minimum segment size to be removed with respect to the supposed segment size `depth*width*height/n_segments`

max_size_factor

[float, optional] Proportion of the maximum connected segment size. A value of 3 works in most of the cases.

slic_zero

[bool, optional] Run SLIC-zero, the zero-parameter mode of SLIC. [?]

start_label

[int, optional] The labels' index start. Should be 0 or 1.

New in version 0.17: `start_label` was introduced in 0.17

mask

[ndarray, optional] If provided, superpixels are computed only where mask is True, and seed points are homogeneously distributed over the mask using a k-means clustering strategy. Mask number of dimensions must be equal to image number of spatial dimensions.

New in version 0.17: `mask` was introduced in 0.17

channel_axis

[int or None, optional] If None, the image is assumed to be a grayscale (single channel) image. Otherwise, this parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

labels

[2D or 3D array] Integer mask indicating segment labels.

Raises**ValueError**

If `convert2lab` is set to True but the last array dimension is not of length 3.

ValueError

If `start_label` is not 0 or 1.

ValueError

If `image` contains unmasked NaN values.

ValueError

If `image` contains unmasked infinite values.

ValueError

If `image` is 2D but `channel_axis` is -1 (the default).

Notes

- If $\sigma > 0$, the image is smoothed using a Gaussian kernel prior to segmentation.
- If σ is scalar and $spacing$ is provided, the kernel width is divided along each dimension by the spacing. For example, if `sigma=1` and `spacing=[5, 1, 1]`, the effective σ is `[0.2, 1, 1]`. This ensures sensible smoothing for anisotropic images.
- The image is rescaled to be in $[0, 1]$ prior to processing (masked values are ignored).
- Images of shape $(M, N, 3)$ are interpreted as 2D RGB images by default. To interpret them as 3D with the last dimension having length 3, use `channel_axis=None`.
- `start_label` is introduced to handle the issue [?]. Label indexing starts at 1 by default.

References

[?], [?], [?], [?]

Examples

```
>>> from skimage.segmentation import slic
>>> from skimage.data import astronaut
>>> img = astronaut()
>>> segments = slic(img, n_segments=100, compactness=10)
```

Increasing the compactness parameter yields more square regions:

```
>>> segments = slic(img, n_segments=100, compactness=20)
```

- *Region Boundary based RAGs*
 - *RAG Thresholding*
 - *Normalized Cut*
 - *Drawing Region Adjacency Graphs (RAGs)*
 - *Apply maskSLIC vs SLIC*
 - *Comparison of segmentation and superpixel algorithms*
 - *Find the intersection of two segmentations*
 - *RAG Merging*
 - *Hierarchical Merging of Region Boundary RAGs*
-

`skimage.segmentation.watershed(image, markers=None, connectivity=1, offset=None, mask=None, compactness=0, watershed_line=False)`

Find watershed basins in *image* flooded from given *markers*.

Parameters

image

[ndarray (2-D, 3-D, ...)] Data array where the lowest value points are labeled first.

markers

[int, or ndarray of int, same shape as *image*, optional] The desired number of markers, or an array marking the basins with the values to be assigned in the label matrix. Zero means not a marker. If None (no markers given), the local minima of the image are used as markers.

connectivity

[ndarray, optional] An array with the same number of dimensions as *image* whose non-zero elements indicate neighbors for connection. Following the scipy convention, default is a one-connected array of the dimension of the image.

offset

[array_like of shape *image.ndim*, optional] offset of the connectivity (one offset per dimension)

mask

[ndarray of bools or 0s and 1s, optional] Array of same shape as *image*. Only points at which mask == True will be labeled.

compactness

[float, optional] Use compact watershed [?] with given compactness parameter. Higher values result in more regularly-shaped watershed basins.

watershed_line

[bool, optional] If watershed_line is True, a one-pixel wide line separates the regions obtained by the watershed algorithm. The line has the label 0. Note that the method used for adding this line expects that marker regions are not adjacent; the watershed line may not catch borders between adjacent marker regions.

Returns**out**

[ndarray] A labeled matrix of the same type and shape as markers

See also:

[`skimage.segmentation.random_walker`](#)

random walker segmentation A segmentation algorithm based on anisotropic diffusion, usually slower than the watershed but with good results on noisy data and boundaries with holes.

Notes

This function implements a watershed algorithm [?] [?] that apportions pixels into marked basins. The algorithm uses a priority queue to hold the pixels with the metric for the priority queue being pixel value, then the time of entry into the queue - this settles ties in favor of the closest marker.

Some ideas taken from Soille, “Automated Basin Delineation from Digital Elevation Models Using Mathematical Morphology”, Signal Processing 20 (1990) 171-182

The most important insight in the paper is that entry time onto the queue solves two problems: a pixel should be assigned to the neighbor with the largest gradient or, if there is no gradient, pixels on a plateau should be split between markers on opposite sides.

This implementation converts all arguments to specific, lowest common denominator types, then passes these to a C algorithm.

Markers can be determined manually, or automatically using for example the local minima of the gradient of the image, or the local maxima of the distance function to the background for separating overlapping objects (see example).

References

[?], [?], [?]

Examples

The watershed algorithm is useful to separate overlapping objects.

We first generate an initial image with two overlapping circles:

```
>>> x, y = np.indices((80, 80))
>>> x1, y1, x2, y2 = 28, 28, 44, 52
>>> r1, r2 = 16, 20
>>> mask_circle1 = (x - x1)**2 + (y - y1)**2 < r1**2
>>> mask_circle2 = (x - x2)**2 + (y - y2)**2 < r2**2
>>> image = np.logical_or(mask_circle1, mask_circle2)
```

Next, we want to separate the two circles. We generate markers at the maxima of the distance to the background:

```
>>> from scipy import ndimage as ndi
>>> distance = ndi.distance_transform_edt(image)
>>> from skimage.feature import peak_local_max
>>> max_coords = peak_local_max(distance, labels=image,
...                               footprint=np.ones((3, 3)))
>>> local_maxima = np.zeros_like(image, dtype=bool)
>>> local_maxima[tuple(max_coords.T)] = True
>>> markers = ndi.label(local_maxima)[0]
```

Finally, we run the watershed on the image and markers:

```
>>> labels = watershed(-distance, markers, mask=image)
```

The algorithm works also for 3-D images, and can be used for example to separate overlapping spheres.

- *Find Regular Segments Using Compact Watershed*
- *Expand segmentation labels without overlap*
- *Watershed segmentation*
- *Markers for watershed transform*
- *Comparison of segmentation and superpixel algorithms*
- *Find the intersection of two segmentations*
- *Evaluating segmentation metrics*
- *Comparing edge-based and region-based segmentation*
- *Segment human cells (in mitosis)*

1.3.18 skimage.transform

This module includes tools to transform images and volumetric data.

- Geometric transformation: These transforms change the shape or position of an image. They are useful for tasks such as image registration, alignment, and geometric correction. Examples: `AffineTransform`, `ProjectiveTransform`, `EuclideanTransform`.
- Image resizing and rescaling: These transforms change the size or resolution of an image. They are useful for tasks such as down-sampling an image to reduce its size or up-sampling an image to increase its resolution. Examples: `resize()`, `rescale()`.
- Feature detection and extraction: These transforms identify and extract specific features or patterns in an image. They are useful for tasks such as object detection, image segmentation, and feature matching. Examples: `hough_circle()`, `pyramid_expand()`, `radon()`.
- Image transformation: These transforms change the appearance of an image without changing its content. They are useful for tasks such as creating image mosaics, applying artistic effects, and visualizing image data. Examples: `warp()`, `iradon()`.

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

continues on next page

Table 11 – continued from previous page

<code>skimage.transform.warp_coords</code>	Build the source coordinates for the output of a 2-D image warp.
<code>skimage.transform.warp_polar</code>	Remap image to polar or log-polar coordinates space.
<code>skimage.transform.AffineTransform</code>	Affine transformation.
<code>skimage.transform.EssentialMatrixTransform</code>	Essential matrix transformation.
<code>skimage.transform.EuclideanTransform</code>	Euclidean transformation, also known as a rigid transform.
<code>skimage.transform.FundamentalMatrixTransform</code>	Fundamental matrix transformation.
<code>skimage.transform.PiecewiseAffineTransform</code>	Piecewise affine transformation.
<code>skimage.transform.PolynomialTransform</code>	2D polynomial transformation.
<code>skimage.transform.ProjectiveTransform</code>	Projective transformation.
<code>skimage.transform.SimilarityTransform</code>	Similarity transformation.

`skimage.transform.downscale_local_mean(image, factors, cval=0, clip=True)`

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 interpolation in `skimage.transform.resize` and `skimage.transform.rescale` this function 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.

`clip`

[bool, optional] Unused, but kept here for API consistency with the other transforms in this module. (The local mean will never fall outside the range of values in the input image, assuming the provided *cval* also falls within that range.)

Returns

`image`

[ndarray] Down-sampled image with same number of dimensions as input image. For integer inputs, the output dtype will be `float64`. See `numpy.mean()` for details.

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]])
```

- *Rescale, resize, and downscale*
-

`skimage.transform.estimate_transform(ttype, src, dst, *args, **kwargs)`

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_like 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 order, default order is 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
>>> import skimage as ski

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

>>> tform = ski.transform.estimate_transform('similarity', src, dst)

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

>>> # warp image using the estimated transformation
>>> image = ski.data.camera()

>>> ski.transform.warp(image, inverse_map=tform.inverse)

>>> # create transformation with explicit parameters
>>> tform2 = ski.transform.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
```

skimage.transform.frt2(a)

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

[?]

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)
```

`skimage.transform.hough_circle(image, radius, normalize=True, full_output=False)`

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=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)
```

- Circular and Elliptical Hough Transforms
-

```
skimage.transform.hough_circle_peaks(hspaces, radii, min_xdistance=1, min_ydistance=1,
                                      threshold=None, num_peaks=inf, total_num_peaks=inf,
                                      normalize=False)
```

Return peaks in a circle Hough transform.

Identifies most prominent circles separated by certain distances in given Hough spaces. Non-maximum suppression with different sizes is applied separately in the first and second dimension of the Hough space to identify peaks. For circles with different radius but close in distance, only the one with highest peak is kept.

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.

Notes

Circles with bigger radius have higher peaks in Hough space. If larger circles are preferred over smaller ones, *normalize* should be False. Otherwise, circles will be returned in the order of decreasing voting number.

Examples

```
>>> from skimage import transform, 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 = transform.hough_circle(img, radius)
>>> accum, cx, cy, rad = hough_circle_peaks(hspaces, [radius,])
```

- Circular and Elliptical Hough Transforms

`skimage.transform.hough_ellipse(image, threshold=4, accuracy=1, min_size=4, max_size=None)`

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, yc, xc, 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

[?]

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)]
```

- Circular and Elliptical Hough Transforms
-

`skimage.transform.hough_line(image, theta=None)`

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 in the range [-pi/2, 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
>>> rng = np.random.default_rng()
>>> img += rng.random(img.shape) * 0.95
```

Apply the Hough transform:

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

- *Straight line Hough transform*

`skimage.transform.hough_line_peaks(hspace, angles, dists, min_distance=9, min_angle=10, threshold=None, num_peaks=inf)`

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.
 $(\text{angles}[-1] - \text{angles}[0] == PI)$.

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=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

- 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.

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 [?] for an overview. The idea for this algorithm is due to Vlad Negnevitski.

References

[?]

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
```

```
skimage.transform.integral_image(image, *, dtype=None)
```

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.

Notes

For better accuracy and to avoid potential overflow, the data type of the output may differ from the input's when the default `dtype` of `None` is used. For inputs with integer `dtype`, the behavior matches that for `numpy.cumsum()`. Floating point inputs will be promoted to at least double precision. The user can set `dtype` to override this behavior.

References

[?]

- *Multi-Block Local Binary Pattern for texture classification*
- *Face classification using Haar-like feature descriptor*

```
skimage.transform.integrate(ii, start, end)
```

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 $[(row_win_1, col_win_1, \dots), (row_win_2, col_win_2, \dots), \dots]$.

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 $[(row_win_1, col_win_1, \dots), (row_win_2, col_win_2, \dots), \dots]$.

Returns**S**

[scalar or ndarray] Integral (sum) over the given window(s).

Examples

```
>>> arr = np.ones((5, 6), dtype=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.])
```

`skimage.transform.iradon(radon_image, theta=None, output_size=None, filter_name='ramp', interpolation='linear', circle=True, preserve_range=True)`

Inverse radon transform.

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

Parameters**radon_image**

[array] 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, 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, optional] Number of rows and columns in the reconstruction.

filter_name

[str, optional] 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] 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`.

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`. Also see https://scikit-image.org/docs/dev/user_guide/data_types.html

Returns

reconstructed

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

Changed in version 0.19: In `iradon`, `filter` argument is deprecated in favor of `filter_name`.

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

[?], [?]

- *Radon transform*

```
skimage.transform.iradon_sart(radon_image, theta=None, image=None, projection_shifts=None, clip=None, relaxation=0.15, dtype=None)
```

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

relaxation

[float, optional] 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.

dtype

[dtype, optional] Output data type, must be floating point. By default, if input data type is not float, input is cast to double, otherwise dtype is set to input data type.

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

[?], [?], [?], [?], [?]

- *Radon transform*
-

`skimage.transform.matrix_transform(coords, matrix)`

Apply 2D matrix transform.

Parameters

coords

`[(N, 2) array_like]` x, y coordinates to transform

matrix

`[(3, 3) array_like]` Homogeneous transformation matrix.

Returns

coords

`[(N, 2) array]` Transformed coordinates.

`skimage.transform.order_angles_golden_ratio(theta)`

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

[?], [?]

```
skimage.transform.probabilistic_hough_line(image, threshold=10, line_length=50, line_gap=10,
                                            theta=None, rng=None)
```

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 aggressively.

theta

[1D ndarray, dtype=double, optional] Angles at which to compute the transform, in radians. Defaults to a vector of 180 angles evenly spaced in the range [-pi/2, pi/2).

rng

[{`numpy.random.Generator`, int}, optional] Pseudo-random number generator. By default, a PCG64 generator is used (see `numpy.random.default_rng()`). If `rng` is an int, it is used to seed the generator.

Returns

lines

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

Other Parameters

seed

[DEPRECATED] Deprecated in favor of *rng*.

Deprecated since version 0.21.

References

[?]

- *Straight line Hough transform*
-

```
skimage.transform.pyramid_expand(image, upscale=2, sigma=None, order=1, mode='reflect', cval=0,
                                 preserve_range=False, *, channel_axis=None)
```

Upsample and then smooth image.

Parameters

image

[ndarray] Input image.

upscale

[float, optional] Upscale factor.

sigma

[float, optional] Sigma for Gaussian filter. Default is $2 * \text{upscale} / 6.0$ 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'.

eval

[float, optional] Value to fill past edges of input if mode is ‘constant’.

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*. Also see https://scikit-image.org/docs/dev/user_guide/data_types.html

channel_axis

[int or None, optional] If None, the image is assumed to be a grayscale (single channel) image. Otherwise, this parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns**out**

[array] Upsampled and smoothed float image.

References

[?]

```
skimage.transform.pyramid_gaussian(image, max_layer=-1, downscale=2, sigma=None, order=1,
mode='reflect', cval=0, preserve_range=False, *, channel_axis=None)
```

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, optional] 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 $2 * \text{downscale} / 6.0$ 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'.

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`. Also see https://scikit-image.org/docs/dev/user_guide/data_types.html

channel_axis

[int or None, optional] If None, the image is assumed to be a grayscale (single channel) image. Otherwise, this parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

pyramid

[generator] Generator yielding pyramid layers as float images.

References

[?]

- *Build image pyramids*

```
skimage.transform.pyramid_laplacian(image, max_layer=-1, downscale=2, sigma=None, order=1,
                                     mode='reflect', cval=0, preserve_range=False, *,
                                     channel_axis=None)
```

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, optional] 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 $2 * \text{downscale} / 6.0$ 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'.

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`. Also see https://scikit-image.org/docs/dev/user_guide/data_types.html

channel_axis

[int or None, optional] If None, the image is assumed to be a grayscale (single channel) image. Otherwise, this parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

pyramid

[generator] Generator yielding pyramid layers as float images.

References

[?], [?]

```
skimage.transform.pyramid_reduce(image, downscale=2, sigma=None, order=1, mode='reflect', cval=0,  
                                 preserve_range=False, *, channel_axis=None)
```

Smooth and then downsample image.

Parameters**image**

[ndarray] Input image.

downscale

[float, optional] Downscale factor.

sigma

[float, optional] Sigma for Gaussian filter. Default is $2 * \text{downscale} / 6.0$ 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'.

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`. Also see https://scikit-image.org/docs/dev/user_guide/data_types.html

channel_axis

[int or None, optional] If None, the image is assumed to be a grayscale (single channel) image. Otherwise, this parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

Returns

`out`

[array] Smoothed and downsampled float image.

References

[?]

`skimage.transform.radon(image, theta=None, circle=True, *, preserve_range=False)`

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

Parameters

`image`

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

`theta`

[array_like, optional] Projection angles (in degrees). If `None`, the value is set to `np.arange(180)`.

`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)`.

`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`. Also see https://scikit-image.org/docs/dev/user_guide/data_types.html

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 (<https://www.clear.rice.edu/elec431/projects96/DSP/bpanalysis.html>)

References

[?], [?]

- *Radon transform*
-

```
skimage.transform.rescale(image, scale, order=None, mode='reflect', cval=0, clip=True,
                         preserve_range=False, anti_aliasing=None, anti_aliasing_sigma=None, *,
                         channel_axis=None)
```

Scale image by a certain factor.

Performs interpolation to up-scale or down-scale N-dimensional images. Note that anti-aliasing should be enabled when down-sizing images to avoid aliasing artifacts. For down-sampling with an integer factor also see `skimage.transform.downscale_local_mean`.

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 0 if image.dtype is bool and 1 otherwise. 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`.

eval

[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`. Also see https://scikit-image.org/docs/dev/user_guide/data_types.html

anti_aliasing

[bool, optional] Whether to apply a Gaussian filter to smooth the image prior to down-scaling. It is crucial to filter when down-sampling the image to avoid aliasing artifacts. If input image data type is bool, no anti-aliasing is applied.

anti_aliasing_sigma

[{float, tuple of floats}, optional] Standard deviation for Gaussian filtering to avoid aliasing artifacts. By default, this value is chosen as $(s - 1) / 2$ where s is the down-scaling factor.

channel_axis

[int or None, optional] If None, the image is assumed to be a grayscale (single channel) image. Otherwise, this parameter indicates which axis of the array corresponds to channels.

New in version 0.19: `channel_axis` was added in 0.19.

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 rescale
>>> image = data.camera()
>>> rescale(image, 0.1).shape
(51, 51)
>>> rescale(image, 0.5).shape
(256, 256)
```

- *Interpolation: Edge Modes*
- *Rescale, resize, and downscale*

- *Radon transform*
 - *Using Polar and Log-Polar Transformations for Registration*
-

```
skimage.transform.resize(image, output_shape, order=None, mode='reflect', cval=0, clip=True,  
                        preserve_range=False, anti_aliasing=None, anti_aliasing_sigma=None)
```

Resize image to match a certain size.

Performs interpolation to up-size or down-size N-dimensional images. Note that anti-aliasing should be enabled when down-sizing images to avoid aliasing artifacts. For downsampling with an integer factor also see `skimage.transform.downscale_local_mean`.

Parameters

image

[ndarray] Input image.

output_shape

[iterable] 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 0 if `image.dtype` is `bool` and 1 otherwise. 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*. Also see https://scikit-image.org/docs/dev/user_guide/data_types.html

anti_aliasing

[bool, optional] Whether to apply a Gaussian filter to smooth the image prior to downampling. It is crucial to filter when downsampling the image to avoid aliasing artifacts. If not specified, it is set to True when downsampling an image whose data type is not bool. It is also set to False when using nearest neighbor interpolation (`order == 0`) with integer input data type.

anti_aliasing_sigma

[{float, tuple of floats}, optional] Standard deviation for Gaussian filtering used when anti-aliasing. By default, this value is chosen as $(s - 1) / 2$ where s is the downsampling factor, where $s > 1$. For the up-size case, $s < 1$, no anti-aliasing is performed prior to rescaling.

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)).shape
(100, 100)
```

- *Interpolation: Edge Modes*
- *Rescale, resize, and downscale*
- *Fisher vector feature encoding*

```
skimage.transform.resize_local_mean(image, output_shape, grid_mode=True, preserve_range=False, *, channel_axis=None)
```

Resize an array with the local mean / bilinear scaling.

Parameters

image

[ndarray] Input image. If this is a multichannel image, the axis corresponding to channels should be specified using *channel_axis*

output_shape

[iterable] Size of the generated output image. When *channel_axis* is not None, the *channel_axis* should either be omitted from *output_shape* or the *output_shape*[*channel_axis*] must match *image.shape*[*channel_axis*]. If the length of *output_shape* exceeds *image.ndim*, additional singleton dimensions will be appended to the input *image* as needed.

grid_mode

[bool, optional] Defines *image* pixels position: if True, pixels are assumed to be at grid intersections, otherwise at cell centers. As a consequence, for example, a 1d signal of length 5 is considered to have length 4 when *grid_mode* is False, but length 5 when *grid_mode* is True. See the following visual illustration:

```
| pixel 1 | pixel 2 | pixel 3 | pixel 4 | pixel 5 |
 |<----->|
           vs.
|<----->|
```

The starting point of the arrow in the diagram above corresponds to coordinate location 0 in each mode.

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*. Also see https://scikit-image.org/docs/dev/user_guide/data_types.html

Returns**resized**

[ndarray] Resized version of the input.

See also:

[**resize**](#), [**downscale_local_mean**](#)

Notes

This method is sometimes referred to as “area-based” interpolation or “pixel mixing” interpolation [?]. When `grid_mode` is True, it is equivalent to using OpenCV’s resize with `INTER_AREA` interpolation mode. It is commonly used for image downsizing. If the downsizing factors are integers, then `downscale_local_mean` should be preferred instead.

References

[?]

Examples

```
>>> from skimage import data
>>> from skimage.transform import resize_local_mean
>>> image = data.camera()
>>> resize_local_mean(image, (100, 100)).shape
(100, 100)
```

`skimage.transform.rotate(image, angle, resize=False, center=None, order=None, mode='constant', cval=0, clip=True, preserve_range=False)`

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=(cols / 2 - 0.5, rows / 2 - 0.5)`. Please note that this parameter is (cols, rows), contrary to normal skimage ordering.

Returns

rotated

[ndarray] Rotated version of the input.

Other Parameters

order

[int, optional] The order of the spline interpolation, default is 0 if image.dtype is bool and 1 otherwise. 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`. Also see https://scikit-image.org/docs/dev/user_guide/data_types.html

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 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)
```

- *Types of homographies*
- *Assemble images with simple image stitching*

- *Using Polar and Log-Polar Transformations for Registration*
 - *ORB feature detector and binary descriptor*
 - *BRIEF binary descriptor*
 - *SIFT feature detector and descriptor extractor*
 - *Sliding window histogram*
 - *Local Binary Pattern for texture classification*
 - *Measure perimeters with different estimators*
 - *Measure region properties*
 - *Visual image comparison*
-

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

Perform a swirl transformation.

Parameters

image

[ndarray] Input image.

center

[(column, row) 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 0 if image.dtype is bool and 1 otherwise. 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`. Also see https://scikit-image.org/docs/dev/user_guide/data_types.html

- *Swirl*
-

`skimage.transform.warp(image, inverse_map, map_args=None, output_shape=None, order=None, mode='constant', cval=0.0, clip=True, preserve_range=False)`

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

Default is 0 if `image.dtype` is bool and 1 otherwise.

`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`.

eval

[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*. Also see https://scikit-image.org/docs/dev/user_guide/data_types.html

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])
>>> rng = np.random.default_rng()
>>> cube = rng.random(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)
```

- *Swirl*
- *Piecewise Affine Transformation*
- *Using geometric transformations*
- *Types of homographies*
- *Robust matching using RANSAC*
- *Registration using optical flow*
- *Assemble images with simple image stitching*
- *CENSURE feature detector*
- *Corner detection*
- *ORB feature detector and binary descriptor*
- *BRIEF binary descriptor*
- *SIFT feature detector and descriptor extractor*

```
skimage.transform.warp_coords(coord_map, shape, dtype=<class 'numpy.float64'>)
```

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)
```

```
skimage.transform.warp_polar(image, center=None, *, radius=None, output_shape=None, scaling='linear',  
                             channel_axis=None, **kwargs)
```

Remap image to polar or log-polar coordinates space.

Parameters

image

[ndarray] Input image. Only 2-D arrays are accepted by default. 3-D arrays are accepted if a *channel_axis* is specified.

center

[tuple (row, col), optional] Point in image that represents the center of the transformation (i.e., the origin in cartesian space). Values can be of type *float*. If no value is given, the center is assumed to be the center point of the image.

radius

[float, optional] Radius of the circle that bounds the area to be transformed.

output_shape

[tuple (row, col), optional]

scaling

[{'linear', 'log'}, optional] Specify whether the image warp is polar or log-polar. Defaults to 'linear'.

channel_axis

[int or None, optional] If None, the image is assumed to be a grayscale (single channel) image. Otherwise, this parameter indicates which axis of the array corresponds to channels.

New in version 0.19: *channel_axis* was added in 0.19.

**kwargs

[keyword arguments] Passed to *transform.warp*.

Returns

warped

[ndarray] The polar or log-polar warped image.

Examples

Perform a basic polar warp on a grayscale image:

```
>>> from skimage import data
>>> from skimage.transform import warp_polar
>>> image = data.checkerboard()
>>> warped = warp_polar(image)
```

Perform a log-polar warp on a grayscale image:

```
>>> warped = warp_polar(image, scaling='log')
```

Perform a log-polar warp on a grayscale image while specifying center, radius, and output shape:

```
>>> warped = warp_polar(image, (100, 100), radius=100,
...                      output_shape=image.shape, scaling='log')
```

Perform a log-polar warp on a color image:

```
>>> image = data.astronaut()
>>> warped = warp_polar(image, scaling='log', channel_axis=-1)
```

- *Using Polar and Log-Polar Transformations for Registration*

```
class skimage.transform.AffineTransform(matrix=None, scale=None, rotation=None, shear=None,
                                       translation=None, *, dimensionality=2)
```

Bases: *ProjectiveTransform*

Affine transformation.

Has the following form:

```
X = a0 * x + a1 * y + a2
    = sx * x * [cos(rotation) + tan(shear_y) * sin(rotation)]
    - sy * y * [tan(shear_x) * cos(rotation) + sin(rotation)]
    + translation_x

Y = b0 * x + b1 * y + b2
    = sx * x * [sin(rotation) - tan(shear_y) * cos(rotation)]
    - sy * y * [tan(shear_x) * sin(rotation) - cos(rotation)]
    + translation_y
```

where *sx* and *sy* are scale factors in the x and y directions.

This is equivalent to applying the operations in the following order:

1. Scale
2. Shear
3. Rotate
4. Translate

The homogeneous transformation matrix is:

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

In 2D, the transformation parameters can be given as the homogeneous transformation matrix, above, or as the implicit parameters, scale, rotation, shear, and translation in x (a2) and y (b2). For 3D and higher, only the matrix form is allowed.

In narrower transforms, such as the Euclidean (only rotation and translation) or Similarity (rotation, translation, and a global scale factor) transforms, it is possible to specify 3D transforms using implicit parameters also.

Parameters

matrix

`[(D+1, D+1) array_like, optional]` Homogeneous transformation matrix. If this matrix is provided, it is an error to provide any of scale, rotation, shear, or translation.

scale

`[{s as float or (sx, sy) as array, list or tuple}, optional]` Scale factor(s). If a single value, it will be assigned to both sx and sy. Only available for 2D.

New in version 0.17: Added support for supplying a single scalar value.

rotation

`[float, optional]` Rotation angle, clockwise, as radians. Only available for 2D.

shear

`[float or 2-tuple of float, optional]` The x and y shear angles, clockwise, by which these axes are rotated around the origin [2]. If a single value is given, take that to be the x shear angle, with the y angle remaining 0. Only available in 2D.

translation

`[(tx, ty) as array, list or tuple, optional]` Translation parameters. Only available for 2D.

dimensionality

`[int, optional]` The dimensionality of the transform. This is not used if any other parameters are provided.

Raises

ValueError

If both `matrix` and any of the other parameters are provided.

References

[?], [?]

Examples

```
>>> import numpy as np
>>> import skimage as ski
>>> img = ski.data.astronaut()
```

Define source and destination points:

```
>>> src = np.array([[150, 150],
...                 [250, 100],
...                 [150, 200]])
>>> dst = np.array([[200, 200],
...                 [300, 150],
...                 [150, 400]])
```

Estimate the transformation matrix:

```
>>> tform = ski.transform.AffineTransform()
>>> tform.estimate(src, dst)
True
```

Apply the transformation:

```
>>> warped = ski.transform.warp(img, inverse_map=tform.inverse)
```

Attributes

params

[(D+1, D+1) array] Homogeneous transformation matrix.

`__init__(matrix=None, scale=None, rotation=None, shear=None, translation=None, *, dimensionality=2)`

- *Types of homographies*
- *Robust matching using RANSAC*
- *CENSURE feature detector*
- *Corner detection*
- *ORB feature detector and binary descriptor*
- *BRIEF binary descriptor*

- SIFT feature detector and descriptor extractor

property dimensionality

The dimensionality of the transformation.

estimate(src, dst, weights=None)

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:

$$\begin{aligned} X &= (a_0*x + a_1*y + a_2) / (c_0*x + c_1*y + 1) \\ Y &= (b_0*x + b_1*y + b_2) / (c_0*x + c_1*y + 1) \end{aligned}$$

These equations can be transformed to the following form:

$$\begin{aligned} 0 &= a_0*x + a_1*y + a_2 - c_0*x*X - c_1*y*X - X \\ 0 &= b_0*x + b_1*y + b_2 - c_0*x*Y - c_1*y*Y - Y \end{aligned}$$

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:

$$\begin{aligned} A &= [[x \ y \ 1 \ 0 \ 0 \ 0 \ -x*X \ -y*X \ -X] \\ &\quad [0 \ 0 \ 0 \ x \ y \ 1 \ -x*Y \ -y*Y \ -Y]] \\ &\quad \dots \\ &\quad \dots \\ &\quad] \\ x.T &= [a_0 \ a_1 \ a_2 \ b_0 \ b_1 \ b_2 \ c_0 \ c_1 \ c_3] \end{aligned}$$

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 c_3 .

Weights can be applied to each pair of corresponding points to indicate, particularly in an overdetermined system, if point pairs have higher or lower confidence or uncertainties associated with them. From the matrix treatment of least squares problems, these weight values are normalised, square-rooted, then built into a diagonal matrix, by which A is multiplied.

In case of the affine transformation the coefficients c_0 and c_1 are 0. Thus the system of equations is:

$$\begin{aligned} A &= [[x \ y \ 1 \ 0 \ 0 \ 0 \ -X] \\ &\quad [0 \ 0 \ 0 \ x \ y \ 1 \ -Y]] \\ &\quad \dots \\ &\quad \dots \\ &\quad] \\ x.T &= [a_0 \ a_1 \ a_2 \ b_0 \ b_1 \ b_2 \ c_3] \end{aligned}$$

Parameters

src

[(N, 2) array_like] Source coordinates.

dst

[(N, 2) array_like] Destination coordinates.

weights

[(N,) array_like, optional] Relative weight values for each pair of points.

Returns

success

[bool] True, if model estimation succeeds.

property inverse

Return a transform object representing the inverse.

residuals(src, dst)

Determine residuals of transformed destination coordinates.

For each transformed source coordinate the Euclidean distance to the respective destination coordinate is determined.

Parameters

src

[(N, 2) array] Source coordinates.

dst

[(N, 2) array] Destination coordinates.

Returns

residuals

[(N,) array] Residual for coordinate.

property rotation

property scale

property shear

property translation

```
class skimage.transform.EssentialMatrixTransform(rotation=None, translation=None, matrix=None, *, dimensionality=2)
```

Bases: *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_like, optional] Rotation matrix of the relative camera motion.

translation

[(3, 1) array_like, optional] Translation vector of the relative camera motion. The vector must have unit length.

matrix

[(3, 3) array_like, optional] Essential matrix.

References

[?]

Examples

```
>>> import numpy as np
>>> import skimage as ski
>>>
>>> tform_matrix = ski.transform.EssentialMatrixTransform(
...     rotation=np.eye(3), translation=np.array([0, 0, 1])
... )
>>> tform_matrix.params
array([[ 0., -1.,  0.],
       [ 1.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

(continues on next page)

(continued from previous page)

```
>>> src = np.array([[ 1.839035,  1.924743],
...                  [ 0.543582,  0.375221],
...                  [ 0.47324 ,  0.142522],
...                  [ 0.96491 ,  0.598376],
...                  [ 0.102388,  0.140092],
...                  [15.994343,  9.622164],
...                  [ 0.285901,  0.430055],
...                  [ 0.09115 ,  0.254594]])
>>> dst = np.array([[1.002114,  1.129644],
...                  [1.521742,  1.846002],
...                  [1.084332,  0.275134],
...                  [0.293328,  0.588992],
...                  [0.839509,  0.08729 ],
...                  [1.779735,  1.116857],
...                  [0.878616,  0.602447],
...                  [0.642616,  1.028681]])
>>> tform_matrix.estimate(src, dst)
True
>>> tform_matrix.residuals(src, dst)
array([0.4245518687, 0.0146044753, 0.1384703409, 0.1214095141,
       0.2775934609, 0.3245311807, 0.0021077555, 0.2651228318])
```

Attributes

params

[(3, 3) array] Essential matrix.

`__init__(rotation=None, translation=None, matrix=None, *, dimensionality=2)`

`estimate(src, dst)`

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_like] Source coordinates.

dst

[(N, 2) array_like] Destination coordinates.

Returns

success

[bool] True, if model estimation succeeds.

property inverse

Return a transform object representing the inverse.

See Hartley & Zisserman, Ch. 8: Epipolar Geometry and the Fundamental Matrix, for an explanation of why $F \cdot T$ gives the inverse.

residuals(src, dst)

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.

```
class skimage.transform.EuclideanTransform(matrix=None, rotation=None, translation=None, *, dimensionality=2)
```

Bases: *ProjectiveTransform*

Euclidean transformation, also known as a rigid transform.

Has the following form:

$$\begin{aligned} X &= a_0 * x - b_0 * y + a_1 = \\ &= x * \cos(\text{rotation}) - y * \sin(\text{rotation}) + a_1 \\ Y &= b_0 * x + a_0 * y + b_1 = \\ &= x * \sin(\text{rotation}) + y * \cos(\text{rotation}) + b_1 \end{aligned}$$

where the homogeneous transformation matrix is:

$$\begin{bmatrix} [a_0 & b_0 & a_1] \\ [b_0 & a_0 & b_1] \\ [0 & 0 & 1] \end{bmatrix}$$

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

[(D+1, D+1) array_like, optional] Homogeneous transformation matrix.

rotation

[float or sequence of float, optional] Rotation angle, clockwise, as radians. If given as a vector, it is interpreted as Euler rotation angles [?]. Only 2D (single rotation) and 3D (Euler rotations) values are supported. For higher dimensions, you must provide or estimate the transformation matrix.

translation

[sequence of float, length D, optional] Translation parameters for each axis.

dimensionality

[int, optional] The dimensionality of the transform.

References

[?]

Attributes

params

[(D+1, D+1) array] Homogeneous transformation matrix.

`__init__(matrix=None, rotation=None, translation=None, *, dimensionality=2)`

- *Using geometric transformations*
- *Types of homographies*
- *Assemble images with simple image stitching*

property dimensionality

The dimensionality of the transformation.

`estimate(src, dst)`

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_like] Source coordinates.

dst

[(N, 2) array_like] Destination coordinates.

Returns

success

[bool] True, if model estimation succeeds.

property inverse

Return a transform object representing the inverse.

residuals(src, dst)

Determine residuals of transformed destination coordinates.

For each transformed source coordinate the Euclidean distance to the respective destination coordinate is determined.

Parameters

src

[(N, 2) array] Source coordinates.

dst

[(N, 2) array] Destination coordinates.

Returns

residuals

[(N,) array] Residual for coordinate.

property rotation

property translation

```
class skimage.transform.FundamentalMatrixTransform(matrix=None, *, dimensionality=2)
```

Bases: `_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_like, optional]` Fundamental matrix.

References

[?]

Attributes

`params`

`[(3, 3) array]` Fundamental matrix.

```
__init__(matrix=None, *, dimensionality=2)
```

- *Fundamental matrix estimation*

`estimate(src, dst)`

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_like]` Source coordinates.

`dst`

`[(N, 2) array_like]` Destination coordinates.

Returns**success**

[bool] True, if model estimation succeeds.

property inverse

Return a transform object representing the inverse.

See Hartley & Zisserman, Ch. 8: Epipolar Geometry and the Fundamental Matrix, for an explanation of why F.T gives the inverse.

residuals(src, dst)

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.

class skimage.transform.PiecewiseAffineTransform

Bases: `_GeometricTransform`

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__()`

- *Piecewise Affine Transformation*

`estimate(src, dst)`

Estimate the transformation from a set of corresponding points.

Number of source and destination coordinates must match.

Parameters

`src`

`[(N, D) array_like]` Source coordinates.

`dst`

`[(N, D) array_like]` Destination coordinates.

Returns

`success`

`[bool]` True, if all pieces of the model are successfully estimated.

`property inverse`

Return a transform object representing the inverse.

`residuals(src, dst)`

Determine residuals of transformed destination coordinates.

For each transformed source coordinate the Euclidean distance to the respective destination coordinate is determined.

Parameters

`src`

`[(N, 2) array]` Source coordinates.

`dst`

`[(N, 2) array]` Destination coordinates.

Returns

residuals

[(N,) array] Residual for coordinate.

class skimage.transform.PolynomialTransform(*params=None*, *, *dimensionality=2*)

Bases: _GeometricTransform

2D polynomial transformation.

Has the following 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_like, 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*, *, *dimensionality=2*)

estimate(*src*, *dst*, *order=2*, *weights=None*)

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.

Weights can be applied to each pair of corresponding points to indicate, particularly in an overdetermined system, if point pairs have higher or lower confidence or uncertainties associated with them. From the matrix treatment of least squares problems, these weight values are normalised, square-rooted, then built into a diagonal matrix, by which A is multiplied.

Parameters

src

[N, 2] array_like] Source coordinates.

dst

[N, 2] array_like] Destination coordinates.

order

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

weights

[N,) array_like, optional] Relative weight values for each pair of points.

Returns

success

[bool] True, if model estimation succeeds.

property inverse

Return a transform object representing the inverse.

residuals(src, dst)

Determine residuals of transformed destination coordinates.

For each transformed source coordinate the Euclidean distance to the respective destination coordinate is determined.

Parameters

src

[(N, 2) array] Source coordinates.

dst

[(N, 2) array] Destination coordinates.

Returns**residuals**

[(N,) array] Residual for coordinate.

```
class skimage.transform.ProjectiveTransform(matrix=None, *, dimensionality=2)
```

Bases: `_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 $H\mathbf{x}$:

$$\begin{bmatrix} [a_0 & a_1 & a_2] \\ [b_0 & b_1 & b_2] \\ [c_0 & c_1 & 1] \end{bmatrix}.$$

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

$$\begin{bmatrix} [\cos(\theta) & -\sin(\theta) & 0] \\ [\sin(\theta) & \cos(\theta) & 0] \\ [0 & 0 & 1] \end{bmatrix}$$

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

$$\begin{bmatrix} [1 & 0 & 10] \\ [0 & 1 & 20] \\ [0 & 0 & 1] \end{bmatrix}.$$

Parameters**matrix**

[(D+1, D+1) array_like, optional] Homogeneous transformation matrix.

dimensionality

[int, optional] The number of dimensions of the transform. This is ignored if `matrix` is not `None`.

Attributes

params

`[(D+1, D+1) array]` Homogeneous transformation matrix.

`__init__(matrix=None, *, dimensionality=2)`

- *Using geometric transformations*
- *Types of homographies*
- *Robust matching using RANSAC*
- *Assemble images with simple image stitching*
- *CENSURE feature detector*
- *Corner detection*
- *ORB feature detector and binary descriptor*
- *BRIEF binary descriptor*
- *SIFT feature detector and descriptor extractor*

property dimensionality

The dimensionality of the transformation.

`estimate(src, dst, weights=None)`

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:

$$\begin{aligned} X &= (a_0*x + a_1*y + a_2) / (c_0*x + c_1*y + 1) \\ Y &= (b_0*x + b_1*y + b_2) / (c_0*x + c_1*y + 1) \end{aligned}$$

These equations can be transformed to the following form:

$$\begin{aligned} 0 &= a_0*x + a_1*y + a_2 - c_0*x*X - c_1*y*X - X \\ 0 &= b_0*x + b_1*y + b_2 - c_0*x*Y - c_1*y*Y - Y \end{aligned}$$

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:

$$\begin{aligned} A &= [[x \ y \ 1 \ 0 \ 0 \ 0 \ -x*X \ -y*X \ -X] \\ &\quad [0 \ 0 \ 0 \ x \ y \ 1 \ -x*Y \ -y*Y \ -Y]] \\ &\quad \dots \\ &\quad \dots \end{aligned}$$

(continues on next page)

(continued from previous page)

```

    ]
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.

Weights can be applied to each pair of corresponding points to indicate, particularly in an overdetermined system, if point pairs have higher or lower confidence or uncertainties associated with them. From the matrix treatment of least squares problems, these weight values are normalised, square-rooted, then built into a diagonal matrix, by which A is multiplied.

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_like] Source coordinates.

dst

[(N, 2) array_like] Destination coordinates.

weights

[(N,) array_like, optional] Relative weight values for each pair of points.

Returns

success

[bool] True, if model estimation succeeds.

property inverse

Return a transform object representing the inverse.

residuals(src, dst)

Determine residuals of transformed destination coordinates.

For each transformed source coordinate the Euclidean distance to the respective destination coordinate is determined.

Parameters**src**

[(N, 2) array] Source coordinates.

dst

[(N, 2) array] Destination coordinates.

Returns**residuals**

[(N,) array] Residual for coordinate.

```
class skimage.transform.SimilarityTransform(matrix=None, scale=None, rotation=None,
                                         translation=None, *, dimensionality=2)
```

Bases: *EuclideanTransform*

Similarity transformation.

Has the following form in 2D:

$$\begin{aligned} X &= a_0 * x - b_0 * y + a_1 = \\ &= s * x * \cos(\text{rotation}) - s * y * \sin(\text{rotation}) + a_1 \\ Y &= b_0 * x + a_0 * y + b_1 = \\ &= s * x * \sin(\text{rotation}) + s * y * \cos(\text{rotation}) + b_1 \end{aligned}$$

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

$$\begin{bmatrix} [a_0 & b_0 & a_1] \\ [b_0 & a_0 & b_1] \\ [0 & 0 & 1] \end{bmatrix}$$

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

Parameters**matrix**

[(dim+1, dim+1) array_like, optional] Homogeneous transformation matrix.

scale

[float, optional] Scale factor. Implemented only for 2D and 3D.

rotation

[float, optional] Rotation angle, clockwise, as radians. Implemented only for 2D and 3D. For 3D, this is given in ZYX Euler angles.

translation

[(dim,) array_like, optional] x, y[, z] translation parameters. Implemented only for 2D and 3D.

Attributes**params**

[(dim+1, dim+1) array] Homogeneous transformation matrix.

__init__(matrix=None, scale=None, rotation=None, translation=None, *, dimensionality=2)

- *Using geometric transformations*
- *Types of homographies*

property dimensionality

The dimensionality of the transformation.

estimate(src, dst)

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_like] Source coordinates.

dst

[(N, 2) array_like] Destination coordinates.

Returns**success**

[bool] True, if model estimation succeeds.

property inverse

Return a transform object representing the inverse.

residuals(*src*, *dst*)

Determine residuals of transformed destination coordinates.

For each transformed source coordinate the Euclidean distance to the respective destination coordinate is determined.

Parameters

src

[(N, 2) array] Source coordinates.

dst

[(N, 2) array] Destination coordinates.

Returns

residuals

[(N,) array] Residual for coordinate.

property rotation

property scale

property translation

1.3.19 skimage.util

<code>skimage.util.apply_parallel</code>	Map a function in parallel across an array.
<code>skimage.util.compare_images</code>	Return an image showing the differences between two images.
<code>skimage.util.crop</code>	Crop array <i>ar</i> by <i>crop_width</i> along each dimension.
<code>skimage.util.dtype_limits</code>	Return intensity limits, i.e. (min, max) tuple, of the image's dtype.
<code>skimage.util.img_as_bool</code>	Convert an image to boolean format.
<code>skimage.util.img_as_float</code>	Convert an image to floating point format.
<code>skimage.util.img_as_float32</code>	Convert an image to single-precision (32-bit) floating point format.
<code>skimage.util.img_as_float64</code>	Convert an image to double-precision (64-bit) floating point format.
<code>skimage.util.img_as_int</code>	Convert an image to 16-bit signed integer format.
<code>skimage.util.img_as_ubyte</code>	Convert an image to 8-bit unsigned integer format.
<code>skimage.util.img_as_uint</code>	Convert an image to 16-bit unsigned integer format.
<code>skimage.util.invert</code>	Invert an image.
<code>skimage.util.label_points</code>	Assign unique integer labels to coordinates on an image mask
<code>skimage.util.map_array</code>	Map values from input array from <i>input_vals</i> to <i>output_vals</i> .
<code>skimage.util.montage</code>	Create a montage of several single- or multichannel images.
<code>skimage.util.random_noise</code>	Function to add random noise of various types to a floating-point image.
<code>skimage.util.regular_grid</code>	Find <i>n_points</i> regularly spaced along <i>ar_shape</i> .
<code>skimage.util.regular_seeds</code>	Return an image with <code>~n_points</code> regularly-spaced nonzero pixels.
<code>skimage.util.slice_along_axes</code>	Slice an image along given axes.
<code>skimage.util.unique_rows</code>	Remove repeated rows from a 2D array.
<code>skimage.util.view_as_blocks</code>	Block view of the input n-dimensional array (using re-striding).
<code>skimage.util.view_as_windows</code>	Rolling window view of the input n-dimensional array.

`skimage.util.apply_parallel(function, array, chunks=None, depth=0, mode=None, extra_arguments=(), extra_keywords=None, *, dtype=None, compute=None, channel_axis=None)`

Map a function in parallel across an array.

Split an array into possibly overlapping chunks of a given depth and boundary type, call the given function in parallel on the chunks, combine the chunks and return the resulting array.

Parameters

function

[function] Function to be mapped which takes an array as an argument.

array

[numpy array or dask array] Array which the function will be applied to.

chunks

[int, tuple, or tuple of tuples, optional] A single integer is interpreted as the length of one side of a square chunk that should be tiled across the array. One tuple of length `array.ndim` represents the shape of a chunk, and it is tiled across the array. A list of tuples of length `ndim`, where each sub-tuple is a sequence of chunk sizes along the corresponding dimension. If `None`, the array is broken up into chunks based on the number of available cpus. More information about chunks is in the documentation [here](#). When `channel_axis` is not `None`, the tuples can be length `ndim - 1` and a single chunk will be used along the channel axis.

depth

[int or sequence of int, optional] The depth of the added boundary cells. A tuple can be used to specify a different depth per array axis. Defaults to zero. When `channel_axis` is not `None`, and a tuple of length `ndim - 1` is provided, a depth of 0 will be used along the channel axis.

mode

[{‘reflect’, ‘symmetric’, ‘periodic’, ‘wrap’, ‘nearest’, ‘edge’}, optional] Type of external boundary padding.

extra_arguments

[tuple, optional] Tuple of arguments to be passed to the function.

extra_keywords

[dictionary, optional] Dictionary of keyword arguments to be passed to the function.

dtype

[data-type or `None`, optional] The data-type of the `function` output. If `None`, Dask will attempt to infer this by calling the function on data of shape `(1,) * ndim`. For functions expecting RGB or multichannel data this may be problematic. In such cases, the user should manually specify this `dtype` argument instead.

New in version 0.18: `dtype` was added in 0.18.

compute

[bool, optional] If `True`, compute eagerly returning a NumPy Array. If `False`, compute lazily returning a Dask Array. If `None` (default), compute based on array type provided (eagerly for NumPy Arrays and lazily for Dask Arrays).

channel_axis

[int or `None`, optional] If `None`, the image is assumed to be a grayscale (single channel) image. Otherwise, this parameter indicates which axis of the array corresponds to channels.

Returns

out

[ndarray or dask Array] Returns the result of the applying the operation. Type is dependent on the `compute` argument.

Notes

Numpy edge modes ‘symmetric’, ‘wrap’, and ‘edge’ are converted to the equivalent dask boundary modes ‘reflect’, ‘periodic’ and ‘nearest’, respectively. Setting `compute=False` can be useful for chaining later operations. For example region selection to preview a result or storing large data to disk instead of loading in memory.

```
skimage.util.compare_images(image1, image2, method='diff', *, n_tiles=(8, 8))
```

Return an image showing the differences between two images.

New in version 0.16.

Parameters

image1, image2

[2-D array] Images to process, must be of the same shape.

method

[string, optional] Method used for the comparison. Valid values are {‘diff’, ‘blend’, ‘checkerboard’}. Details are provided in the note section.

n_tiles

[tuple, optional] Used only for the *checkerboard* method. Specifies the number of tiles (row, column) to divide the image.

Returns

comparison

[2-D array] Image showing the differences.

Notes

‘diff’ computes the absolute difference between the two images. ‘blend’ computes the mean value. ‘checkerboard’ makes tiles of dimension `n_tiles` that display alternatively the first and the second image.

- *Edge operators*
- *Visual image comparison*

```
skimage.util.crop(ar, crop_width, copy=False, order='K')
```

Crop array `ar` by `crop_width` along each dimension.

Parameters

ar

[array-like of rank N] Input array.

crop_width

[{sequence, int}] Number of values to remove from the edges of each axis. ((before_1, after_1), ... (before_N, after_N)) specifies unique crop widths at the start and end of each axis. ((before, after),) or (before, after) specifies a fixed start and end crop for every axis. (n,) or n for integer n is a shortcut for before = after = n for all axes.

copy

[bool, optional] If *True*, ensure the returned array is a contiguous copy. Normally, a crop operation will return a discontiguous view of the underlying input array.

order

[{‘C’, ‘F’, ‘A’, ‘K’}, optional] If `copy==True`, control the memory layout of the copy. See `np.copy`.

Returns

cropped

[array] The cropped array. If `copy=False` (default), this is a sliced view of the input array.

skimage.util.dtype_limits(*image*, *clip_negative=False*)

Return intensity limits, i.e. (min, max) tuple, of the image’s dtype.

Parameters

image

[ndarray] Input image.

clip_negative

[bool, optional] If True, clip the negative range (i.e. return 0 for min intensity) even if the image dtype allows negative values.

Returns

imin, imax

[tuple] Lower and upper intensity limits.

`skimage.util.img_as_bool(image, force_copy=False)`

Convert an image to boolean format.

Parameters**image**

[ndarray] Input image.

force_copy

[bool, optional] Force a copy of the data, irrespective of its current dtype.

Returns**out**

[ndarray of bool (*bool_*)] Output image.

Notes

The upper half of the input dtype's positive range is True, and the lower half is False. All negative values (if present) are False.

`skimage.util.img_as_float(image, force_copy=False)`

Convert an image to floating point format.

This function is similar to `img_as_float64`, but will not convert lower-precision floating point arrays to `float64`.

Parameters**image**

[ndarray] Input image.

force_copy

[bool, optional] Force a copy of the data, irrespective of its current dtype.

Returns**out**

[ndarray of float] Output image.

Notes

The range of a floating point image is [0.0, 1.0] or [-1.0, 1.0] when converting from unsigned or signed datatypes, respectively. If the input image has a float type, intensity values are not modified and can be outside the ranges [0.0, 1.0] or [-1.0, 1.0].

`skimage.util.img_as_float32(image, force_copy=False)`

Convert an image to single-precision (32-bit) floating point format.

Parameters

image

[ndarray] Input image.

force_copy

[bool, optional] Force a copy of the data, irrespective of its current dtype.

Returns

out

[ndarray of float32] Output image.

Notes

The range of a floating point image is [0.0, 1.0] or [-1.0, 1.0] when converting from unsigned or signed datatypes, respectively. If the input image has a float type, intensity values are not modified and can be outside the ranges [0.0, 1.0] or [-1.0, 1.0].

`skimage.util.img_as_float64(image, force_copy=False)`

Convert an image to double-precision (64-bit) floating point format.

Parameters

image

[ndarray] Input image.

force_copy

[bool, optional] Force a copy of the data, irrespective of its current dtype.

Returns

out

[ndarray of float64] Output image.

Notes

The range of a floating point image is [0.0, 1.0] or [-1.0, 1.0] when converting from unsigned or signed datatypes, respectively. If the input image has a float type, intensity values are not modified and can be outside the ranges [0.0, 1.0] or [-1.0, 1.0].

`skimage.util.img_as_int(image, force_copy=False)`

Convert an image to 16-bit signed integer format.

Parameters**image**

[ndarray] Input image.

force_copy

[bool, optional] Force a copy of the data, irrespective of its current dtype.

Returns**out**

[ndarray of int16] Output image.

Notes

The values are scaled between -32768 and 32767. If the input data-type is positive-only (e.g., uint8), then the output image will still only have positive values.

`skimage.util.img_as_ubyte(image, force_copy=False)`

Convert an image to 8-bit unsigned integer format.

Parameters**image**

[ndarray] Input image.

force_copy

[bool, optional] Force a copy of the data, irrespective of its current dtype.

Returns

out

[ndarray of ubyte (uint8)] Output image.

Notes

Negative input values will be clipped. Positive values are scaled between 0 and 255.

`skimage.util.img_as_uint(image, force_copy=False)`

Convert an image to 16-bit unsigned integer format.

Parameters

image

[ndarray] Input image.

force_copy

[bool, optional] Force a copy of the data, irrespective of its current dtype.

Returns

out

[ndarray of uint16] Output image.

Notes

Negative input values will be clipped. Positive values are scaled between 0 and 65535.

`skimage.util.invert(image, signed_float=False)`

Invert an image.

Invert the intensity range of the input image, so that the dtype maximum is now the dtype minimum, and vice-versa. This operation is slightly different depending on the input dtype:

- unsigned integers: subtract the image from the dtype maximum
- signed integers: subtract the image from -1 (see Notes)
- floats: subtract the image from 1 (if signed_float is False, so we assume the image is unsigned), or from 0 (if signed_float is True).

See the examples for clarification.

Parameters

image

[ndarray] Input image.

signed_float

[bool, optional] If True and the image is of type float, the range is assumed to be [-1, 1]. If False and the image is of type float, the range is assumed to be [0, 1].

Returns

inverted

[ndarray] Inverted image.

Notes

Ideally, for signed integers we would simply multiply by -1. However, signed integer ranges are asymmetric. For example, for np.int8, the range of possible values is [-128, 127], so that $-128 * -1$ equals -128! By subtracting from -1, we correctly map the maximum dtype value to the minimum.

Examples

```
>>> img = np.array([[100, 0, 200],
...                  [0, 50, 0],
...                  [30, 0, 255]], np.uint8)
>>> invert(img)
array([[155, 255, 55],
       [255, 205, 255],
       [225, 255, 0]], dtype=uint8)
>>> img2 = np.array([[ -2, 0, -128],
...                   [127, 0, 5]], np.int8)
>>> invert(img2)
array([[ 1, -1, 127],
       [-128, -1, -6]], dtype=int8)
>>> img3 = np.array([[ 0., 1., 0.5, 0.75]])
>>> invert(img3)
array([[1., 0., 0.5, 0.25]])
>>> img4 = np.array([[ 0., 1., -1., -0.25]])
>>> invert(img4, signed_float=True)
array([[-0., -1., 1., 0.25]])
```

- *Convex Hull*
- *Skeletonize*
- *Use rolling-ball algorithm for estimating background intensity*

skimage.util.label_points(*coords*, *output_shape*)

Assign unique integer labels to coordinates on an image mask

Parameters

coords: ndarray

An array of N coordinates with dimension D

output_shape: tuple

The shape of the mask on which *coords* are labelled

Returns

labels: ndarray

A mask of zeroes containing unique integer labels at the *coords*

Notes

- The labels are assigned to coordinates that are converted to integer and considered to start from 0.
- Coordinates that are out of range of the mask raise an IndexError.
- Negative coordinates raise a ValueError

Examples

```
>>> import numpy as np
>>> from skimage.util._label import label_points
>>> coords = np.array([[0, 1], [2, 2]])
>>> output_shape = (5, 5)
>>> mask = label_points(coords, output_shape)
>>> mask
array([[0, 1, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 2, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]], dtype=uint64)
```

skimage.util.map_array(*input_arr*, *input_vals*, *output_vals*, *out=None*)

Map values from input array from *input_vals* to *output_vals*.

Parameters

input_arr

[array of int, shape (M[, N][, P][, ...])] The input label image.

input_vals

[array of int, shape (N,)] The values to map from.

output_vals

[array, shape (N,)] The values to map to.

out: array, same shape as `input_arr`

The output array. Will be created if not provided. It should have the same dtype as *output_vals*.

Returns**out**

[array, same shape as *input_arr*] The array of mapped values.

```
skimage.util.montage(arr_in, fill='mean', rescale_intensity=False, grid_shape=None, padding_width=0, *,  
channel_axis=None)
```

Create a montage of several single- or multichannel images.

Create a rectangular montage from an input array representing an ensemble of equally shaped single- (gray) or multichannel (color) images.

For example, `montage(arr_in)` called with the following *arr_in*

1	2	3
---	---	---

will return

1	2
3	
•	

where the '*' patch will be determined by the *fill* parameter.

Parameters**arr_in**

[(K, M, N[, C]) ndarray] An array representing an ensemble of *K* images of equal shape.

fill

[float or array-like of floats or ‘mean’, optional] Value to fill the padding areas and/or the extra tiles in the output array. Has to be *float* for single channel collections. For multichannel collections has to be an array-like of shape of number of channels. If *mean*, uses the mean value over all images.

rescale_intensity

[bool, optional] Whether to rescale the intensity of each image to [0, 1].

grid_shape

[tuple, optional] The desired grid shape for the montage (*ntiles_row*, *ntiles_column*). The default aspect ratio is square.

padding_width

[int, optional] The size of the spacing between the tiles and between the tiles and the borders. If non-zero, makes the boundaries of individual images easier to perceive.

channel_axis

[int or None, optional] If None, the image is assumed to be a grayscale (single channel) image. Otherwise, this parameter indicates which axis of the array corresponds to channels.

Returns

arr_out

[(K*(M+p)+p, K*(N+p)+p[, C]) ndarray] Output array with input images glued together (including padding *p*).

Examples

```
>>> import numpy as np
>>> from skimage.util import montage
>>> arr_in = np.arange(3 * 2 * 2).reshape(3, 2, 2)
>>> arr_in
array([[[ 0,  1],
       [ 2,  3]],
      [[ 4,  5],
       [ 6,  7]],
      [[ 8,  9],
       [10, 11]]])
>>> arr_out = montage(arr_in)
>>> arr_out.shape
(4, 4)
>>> arr_out
array([[[ 0,  1,  4,  5],
       [ 2,  3,  6,  7],
       [ 8,  9,  5,  5],
       [10, 11,  5,  5]])]
>>> arr_in.mean()
5.5
```

(continues on next page)

(continued from previous page)

```
>>> arr_out_nonsquare = montage(arr_in, grid_shape=(1, 3))
>>> arr_out_nonsquare
array([[ 0,  1,  4,  5,  8,  9],
       [ 2,  3,  6,  7, 10, 11]])
>>> arr_out_nonsquare.shape
(2, 6)
```

- *Gabors / Primary Visual Cortex “Simple Cells” from an Image*
- *Explore 3D images (of cells)*

`skimage.util.random_noise(image, mode='gaussian', rng=None, clip=True, **kwargs)`

Function to add random noise of various types to a floating-point image.

Parameters

`image`

[ndarray] Input image data. Will be converted to float.

`mode`

[str, optional] One of the following strings, selecting the type of noise to add:

‘gaussian’ (default)

Gaussian-distributed additive noise.

‘localvar’

Gaussian-distributed additive noise, with specified local variance at each point of *image*.

‘poisson’

Poisson-distributed noise generated from the data.

‘salt’

Replaces random pixels with 1.

‘pepper’

Replaces random pixels with 0 (for unsigned images) or -1 (for signed images).

‘s&p’

Replaces random pixels with either 1 or *low_val*, where *low_val* is 0 for unsigned images or -1 for signed images.

‘speckle’

Multiplicative noise using `out = image + n * image`, where *n* is Gaussian noise with specified mean & variance.

rng

[{`numpy.random.Generator`, int}, optional] Pseudo-random number generator. By default, a PCG64 generator is used (see `numpy.random.default_rng()`). If `rng` is an int, it is used to seed the generator.

clip

[bool, optional] If True (default), the output will be clipped after noise applied for modes ‘speckle’, ‘poisson’, and ‘gaussian’. This is needed to maintain the proper image data range. If False, clipping is not applied, and the output may extend beyond the range [-1, 1].

mean

[float, optional] Mean of random distribution. Used in ‘gaussian’ and ‘speckle’. Default : 0.

var

[float, optional] Variance of random distribution. Used in ‘gaussian’ and ‘speckle’. Note: variance = (standard deviation) ** 2. Default : 0.01

local_vars

[ndarray, optional] Array of positive floats, same shape as *image*, defining the local variance at every image point. Used in ‘localvar’.

amount

[float, optional] Proportion of image pixels to replace with noise on range [0, 1]. Used in ‘salt’, ‘pepper’, and ‘salt & pepper’. Default : 0.05

salt_vs_pepper

[float, optional] Proportion of salt vs. pepper noise for ‘s&p’ on range [0, 1]. Higher values represent more salt. Default : 0.5 (equal amounts)

Returns

out

[ndarray] Output floating-point image data on range [0, 1] or [-1, 1] if the input *image* was unsigned or signed, respectively.

Other Parameters

seed

[DEPRECATED] Deprecated in favor of `rng`.

Deprecated since version 0.21.

Notes

Speckle, Poisson, Localvar, and Gaussian noise may generate noise outside the valid image range. The default is to clip (not alias) these values, but they may be preserved by setting `clip=False`. Note that in this case the output may contain values outside the ranges [0, 1] or [-1, 1]. Use this option with care.

Because of the prevalence of exclusively positive floating-point images in intermediate calculations, it is not possible to intuit if an input is signed based on `dtype` alone. Instead, negative values are explicitly searched for. Only if found does this function assume signed input. Unexpected results only occur in rare, poorly exposes cases (e.g. if all values are above 50 percent gray in a signed *image*). In this event, manually scaling the input to the positive domain will solve the problem.

The Poisson distribution is only defined for positive integers. To apply this noise type, the number of unique values in the image is found and the next round power of two is used to scale up the floating-point result, after which it is scaled back down to the floating-point image range.

To generate Poisson noise against a signed image, the signed image is temporarily converted to an unsigned image in the floating point domain, Poisson noise is generated, then it is returned to the original range.

- *Canny edge detector*
- *Assemble images with simple image stitching*
- *Calibrating Denoisers Using J-Invariance*
- *Denoising a picture*
- *Shift-invariant wavelet denoising*
- *Non-local means denoising for preserving textures*
- *Wavelet denoising*
- *Full tutorial on calibrating Denoisers Using J-Invariance*

`skimage.util.regular_grid(ar_shape, n_points)`

Find `n_points` regularly spaced along `ar_shape`.

The returned points (as slices) should be as close to cubically-spaced as possible. Essentially, the points are spaced by the Nth root of the input array size, where N is the number of dimensions. However, if an array dimension cannot fit a full step size, it is “discarded”, and the computation is done for only the remaining dimensions.

Parameters

`ar_shape`

[array-like of ints] The shape of the space embedding the grid. `len(ar_shape)` is the number of dimensions.

`n_points`

[int] The (approximate) number of points to embed in the space.

Returns**slices**

[tuple of slice objects] A slice along each dimension of *ar_shape*, such that the intersection of all the slices give the coordinates of regularly spaced points.

Changed in version 0.14.1: In scikit-image 0.14.1 and 0.15, the return type was changed from a list to a tuple to ensure compatibility with Numpy 1.15 and higher. If your code requires the returned result to be a list, you may convert the output of this function to a list with:

```
>>> result = list(regular_grid(ar_shape=(3, 20, 40), n_points=8))
```

Examples

```
>>> ar = np.zeros((20, 40))
>>> g = regular_grid(ar.shape, 8)
>>> g
(slice(5, None, 10), slice(5, None, 10))
>>> ar[g] = 1
>>> ar.sum()
8.0
>>> ar = np.zeros((20, 40))
>>> g = regular_grid(ar.shape, 32)
>>> g
(slice(2, None, 5), slice(2, None, 5))
>>> ar[g] = 1
>>> ar.sum()
32.0
>>> ar = np.zeros((3, 20, 40))
>>> g = regular_grid(ar.shape, 8)
>>> g
(slice(1, None, 3), slice(5, None, 10), slice(5, None, 10))
>>> ar[g] = 1
>>> ar.sum()
8.0
```

- Find Regular Segments Using Compact Watershed
-

`skimage.util.regular_seeds(ar_shape, n_points, dtype=<class 'int'>)`

Return an image with ~`n_points` regularly-spaced nonzero pixels.

Parameters**ar_shape**

[tuple of int] The shape of the desired output image.

n_points

[int] The desired number of nonzero points.

dtype

[numpy data type, optional] The desired data type of the output.

Returns**seed_img**

[array of int or bool] The desired image.

Examples

```
>>> regular_seeds((5, 5), 4)
array([[0, 0, 0, 0, 0],
       [0, 1, 0, 2, 0],
       [0, 0, 0, 0, 0],
       [0, 3, 0, 4, 0],
       [0, 0, 0, 0, 0]])
```

skimage.util.slice_along_axes(*image*, *slices*, *axes=None*, *copy=False*)

Slice an image along given axes.

Parameters**image**

[ndarray] Input image.

slices

[list of 2-tuple (a, b) where a < b.] For each axis in *axes*, a corresponding 2-tuple (*min_val*, *max_val*) to slice with (as with Python slices, *max_val* is non-inclusive).

axes

[int or tuple, optional] Axes corresponding to the limits given in *slices*. If None, axes are in ascending order, up to the length of *slices*.

copy

[bool, optional] If True, ensure that the output is not a view of *image*.

Returns

out

[ndarray] The region of *image* corresponding to the given slices and axes.

Examples

```
>>> from skimage import data
>>> img = data.camera()
>>> img.shape
(512, 512)
>>> cropped_img = slice_along_axes(img, [(0, 100)])
>>> cropped_img.shape
(100, 512)
>>> cropped_img = slice_along_axes(img, [(0, 100), (0, 100)])
>>> cropped_img.shape
(100, 100)
>>> cropped_img = slice_along_axes(img, [(0, 100), (0, 75)], axes=[1, 0])
>>> cropped_img.shape
(75, 100)
```

skimage.util.unique_rows(ar)

Remove repeated rows from a 2D array.

In particular, if given an array of coordinates of shape (Npoints, Ndim), it will remove repeated points.

Parameters

ar

[2-D ndarray] The input array.

Returns

ar_out

[2-D ndarray] A copy of the input array with repeated rows removed.

Raises

ValueError

[if *ar* is not two-dimensional.]

Notes

The function will generate a copy of *ar* if it is not C-contiguous, which will negatively affect performance for large input arrays.

Examples

```
>>> ar = np.array([[1, 0, 1],
...                 [0, 1, 0],
...                 [1, 0, 1]], np.uint8)
>>> unique_rows(ar)
array([[0, 1, 0],
       [1, 0, 1]], dtype=uint8)
```

`skimage.util.view_as_blocks(arr_in, block_shape)`

Block view of the input n-dimensional array (using re-striding).

Blocks are non-overlapping views of the input array.

Parameters

`arr_in`

[ndarray] N-d input array.

`block_shape`

[tuple] The shape of the block. Each dimension must divide evenly into the corresponding dimensions of *arr_in*.

Returns

`arr_out`

[ndarray] Block view of the input array.

Examples

```
>>> import numpy as np
>>> from skimage.util.shape import view_as_blocks
>>> A = np.arange(4*4).reshape(4,4)
>>> A
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> B = view_as_blocks(A, block_shape=(2, 2))
```

(continues on next page)

(continued from previous page)

```
>>> B[0, 0]
array([[0, 1],
       [4, 5]])
>>> B[0, 1]
array([[2, 3],
       [6, 7]])
>>> B[1, 0, 1, 1]
13
```

```
>>> A = np.arange(4*4*6).reshape(4,4,6)
>>> A
array([[[ 0,  1,  2,  3,  4,  5],
        [ 6,  7,  8,  9, 10, 11],
        [12, 13, 14, 15, 16, 17],
        [18, 19, 20, 21, 22, 23]],
       [[24, 25, 26, 27, 28, 29],
        [30, 31, 32, 33, 34, 35],
        [36, 37, 38, 39, 40, 41],
        [42, 43, 44, 45, 46, 47]],
       [[48, 49, 50, 51, 52, 53],
        [54, 55, 56, 57, 58, 59],
        [60, 61, 62, 63, 64, 65],
        [66, 67, 68, 69, 70, 71]],
       [[72, 73, 74, 75, 76, 77],
        [78, 79, 80, 81, 82, 83],
        [84, 85, 86, 87, 88, 89],
        [90, 91, 92, 93, 94, 95]]])
>>> B = view_as_blocks(A, block_shape=(1, 2, 2))
>>> B.shape
(4, 2, 3, 1, 2, 2)
>>> B[2:, 0, 2]
array([[[[52, 53],
        [58, 59]]],
       [[[76, 77],
        [82, 83]]]])
```

- *Block views on images/arrays*

skimage.util.view_as_windows(arr_in, window_shape, step=1)

Rolling window view of the input n-dimensional array.

Windows are overlapping views of the input array, with adjacent windows shifted by a single row or column (or an index of a higher dimension).

Parameters

arr_in

[ndarray] N-d input array.

window_shape

[integer or tuple of length arr_in.ndim] Defines the shape of the elementary n-dimensional orthotope (better known as hyperrectangle [?]) of the rolling window view. If an integer is given, the shape will be a hypercube of sidelength given by its value.

step

[integer or tuple of length arr_in.ndim] Indicates step size at which extraction shall be performed. If integer is given, then the step is uniform in all dimensions.

Returns**arr_out**

[ndarray] (rolling) window view of the input array.

Notes

One should be very careful with rolling views when it comes to memory usage. Indeed, although a ‘view’ has the same memory footprint as its base array, the actual array that emerges when this ‘view’ is used in a computation is generally a (much) larger array than the original, especially for 2-dimensional arrays and above.

For example, let us consider a 3 dimensional array of size (100, 100, 100) of `float64`. This array takes about $8*100**3$ Bytes for storage which is just 8 MB. If one decides to build a rolling view on this array with a window of (3, 3, 3) the hypothetical size of the rolling view (if one was to reshape the view for example) would be $8*(100-3+1)**3*3**3$ which is about 203 MB! The scaling becomes even worse as the dimension of the input array becomes larger.

References

[?]

Examples

```
>>> import numpy as np
>>> from skimage.util.shape import view_as_windows
>>> A = np.arange(4*4).reshape(4,4)
>>> A
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> window_shape = (2, 2)
>>> B = view_as_windows(A, window_shape)
>>> B[0, 0]
array([[ 0,  1],
       [ 4,  5]])
>>> B[0, 1]
array([[ 1,  2],
       [ 5,  6]])
```

```
>>> A = np.arange(10)
>>> A
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> window_shape = (3,)
>>> B = view_as_windows(A, window_shape)
>>> B.shape
(8, 3)
>>> B
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4],
       [3, 4, 5],
       [4, 5, 6],
       [5, 6, 7],
       [6, 7, 8],
       [7, 8, 9]])
```

```
>>> A = np.arange(5*4).reshape(5, 4)
>>> A
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])
>>> window_shape = (4, 3)
>>> B = view_as_windows(A, window_shape)
>>> B.shape
(2, 2, 4, 3)
>>> B
array([[[[ 0,  1,  2],
          [ 4,  5,  6],
          [ 8,  9, 10],
          [12, 13, 14]]],
       [[[ 1,  2,  3],
          [ 5,  6,  7],
          [ 9, 10, 11],
          [13, 14, 15]]],
       [[[ 4,  5,  6],
          [ 8,  9, 10],
          [12, 13, 14],
          [16, 17, 18]]],
       [[[ 5,  6,  7],
          [ 9, 10, 11],
          [13, 14, 15],
          [17, 18, 19]]]])
```

- *Gabors / Primary Visual Cortex “Simple Cells” from an Image*
-

1.3.20 License

```
Files: *
Copyright: 2009-2022 the scikit-image team
License: BSD-3-Clause

Files: doc/source/themes/scikit-image/layout.html
Copyright: 2007-2010 the Sphinx team
License: BSD-3-Clause

Files: skimage/feature/_canny.py
       skimage/filters/edges.py
       skimage/filters/_rank_order.py
       skimage/morphology/_skeletonize.py
       skimage/morphology/tests/test_watershed.py
       skimage/morphology/watershed.py
       skimage/segmentation/heap_general.pxi
       skimage/segmentation/heap_watershed.pxi
       skimage/segmentation/_watershed.py
       skimage/segmentation/_watershed_cy.pyx
Copyright: 2003-2009 Massachusetts Institute of Technology
           2009-2011 Broad Institute
           2003 Lee Kamentsky
           2003-2005 Peter J. Verveer
License: BSD-3-Clause

Files: skimage/filters/thresholding.py
       skimage/graph/_mcp.pyx
       skimage/graph/heap.pyx
Copyright: 2009-2015 Board of Regents of the University of
           Wisconsin-Madison, Broad Institute of MIT and Harvard,
           and Max Planck Institute of Molecular Cell Biology and
           Genetics
           2009 Zachary Pincus
           2009 Almar Klein
License: BSD-2-Clause

File: skimage/morphology/grayreconstruct.py
      skimage/morphology/tests/test_reconstruction.py
Copyright: 2003-2009 Massachusetts Institute of Technology
           2009-2011 Broad Institute
           2003 Lee Kamentsky
License: BSD-3-Clause

File: skimage/morphology/_grayreconstruct.pyx
Copyright: 2003-2009 Massachusetts Institute of Technology
           2009-2011 Broad Institute
           2003 Lee Kamentsky
           2022 Gregory Lee (added a 64-bit integer variant for large images)
License: BSD-3-Clause

File: skimage/segmentation/_expand_labels.py
Copyright: 2020 Broad Institute
```

(continues on next page)

(continued from previous page)

2020 CellProfiler team

License: BSD-3-Clause

File: skimage/exposure/_adaphist.py

Copyright: 1994 Karel Zuiderveld

License: BSD-3-Clause

Function: skimage/morphology/_skeletonize_cy.pyx:_skeletonize_loop

Copyright: 2003-2009 Massachusetts Institute of Technology

2009-2011 Broad Institute

2003 Lee Kamentsky

License: BSD-3-Clause

Function: skimage/_shared/version_requirements.py:_check_version

Copyright: 2013 The IPython Development Team

License: BSD-3-Clause

Function: skimage/_shared/version_requirements.py:is_installed

Copyright: 2009-2011 Pierre Raybaut

License: MIT

File: skimage/feature/fisher_vector.py

Copyright: 2014 2014 Dan Oneata

License: MIT

License: BSD-2-Clause

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE HOLDERS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

License: BSD-3-Clause

Redistribution and use in source and binary forms, with or without

(continues on next page)

(continued from previous page)

modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE HOLDERS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

License: MIT

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.

1.4 Release notes

This is the list of changes to scikit-image between each release. For full details, see the [commit logs](#).

1.4.1 scikit-image 0.X.0 release notes

scikit-image is an image processing library for the scientific Python ecosystem that includes algorithms for segmentation, geometric transformations, feature detection, registration, color space manipulation, analysis, filtering, morphology, and more.

For more information, examples, and documentation, please visit our website:

<https://scikit-image.org>

New Features

Improvements

API Changes

- `data.image_fetcher` is now private (gh-6854)
- `data.create_image_fetcher` is now private (gh-6854)

Bugfixes

Deprecations

Contributors to this release

1.4.2 scikit-image 0.9.0 release notes

We're happy to announce the release of scikit-image v0.9.0!

scikit-image is an image processing toolbox for SciPy that includes algorithms for segmentation, geometric transformations, color space manipulation, analysis, filtering, morphology, feature detection, and more.

For more information, examples, and documentation, please visit our website:

<http://scikit-image.org>

New Features

scikit-image now runs without translation under both Python 2 and 3.

In addition to several bug fixes, speed improvements and examples, the 204 pull requests merged for this release include the following new features (PR number in brackets):

Segmentation:

- 3D support in SLIC segmentation (#546)
- SLIC voxel spacing (#719)
- Generalized anisotropic spacing support for random_walker (#775)

- Yen threshold method (#686)

Transforms and filters:

- SART algorithm for tomography reconstruction (#584)
- Gabor filters (#371)
- Hough transform for ellipses (#597)
- Fast resampling of nD arrays (#511)
- Rotation axis center for Radon transforms with inverses. (#654)
- Reconstruction circle in inverse Radon transform (#567)
- Pixelwise image adjustment curves and methods (#505)

Feature detection:

- [experimental API] BRIEF feature descriptor (#591)
- [experimental API] Censure (STAR) Feature Detector (#668)
- Octagon structural element (#669)
- Add non rotation invariant uniform LBPs (#704)

Color and noise:

- Add deltaE color comparison and lab2lch conversion (#665)
- Isotropic denoising (#653)
- Generator to add various types of random noise to images (#625)
- Color deconvolution for immunohistochemical images (#441)
- Color label visualization (#485)

Drawing and visualization:

- Wu's anti-aliased circle, line, bezier curve (#709)
- Linked image viewers and docked plugins (#575)
- Rotated ellipse + bezier curve drawing (#510)
- PySide & PyQt4 compatibility in skimage-viewer (#551)

Other:

- Python 3 support without 2to3. (#620)
- 3D Marching Cubes (#469)
- Line, Circle, Ellipse total least squares fitting and RANSAC algorithm (#440)
- N-dimensional array padding (#577)
- Add a wrapper around `scipy.ndimage.gaussian_filter` with useful default behaviors. (#712)
- Predefined structuring elements for 3D morphology (#484)

API changes

The following backward-incompatible API changes were made between 0.8 and 0.9:

- No longer wrap `imread` output in an `Image` class
- Change default value of `sigma` parameter in `skimage.segmentation.slic` to 0
- `hough_circle` now returns a stack of arrays that are the same size as the input image. Set the `full_output` flag to True for the old behavior.
- The following functions were deprecated over two releases: `skimage.filter.denoise_tv_chambolle`, `skimage.morphology.is_local_maximum`, `skimage.transform.hough`, `skimage.transform.probabilistic_hough`, `skimage.transform.hough`. Their functionality still exists, but under different names.

Contributors to this release

This release was made possible by the collaborative efforts of many contributors, both new and old. They are listed in alphabetical order by surname:

- Ankit Agrawal
- K.-Michael Aye
- Chris Beaumont
- François Boulogne
- Luis Pedro Coelho
- Marianne Corvellec
- Olivier Debeir
- Ferdinand Deger
- Kemal Eren
- Jostein Bø Fløystad
- Christoph Gohlke
- Emmanuelle Gouillart
- Christian Horea
- Thouis (Ray) Jones
- Almar Klein
- Xavier Moles Lopez
- Alexis Mignon
- Juan Nunez-Iglesias
- Zachary Pincus
- Nicolas Pinto
- Davin Potts
- Malcolm Reynolds
- Umesh Sharma
- Johannes Schönberger

- Chintak Sheth
- Kirill Shklovsky
- Steven Sylvester
- Matt Terry
- Riaan van den Dool
- Stéfan van der Walt
- Josh Warner
- Adam Wisniewski
- Yang Zetian
- Tony S Yu

1.4.3 scikits-image 0.8.0 release notes

We're happy to announce the 8th version of scikit-image!

scikit-image is an image processing toolbox for SciPy that includes algorithms for segmentation, geometric transformations, color space manipulation, analysis, filtering, morphology, feature detection, and more.

For more information, examples, and documentation, please visit our website:

<http://scikit-image.org>

New Features

- New rank filter package with many new functions and a very fast underlying local histogram algorithm, especially for large structuring elements `skimage.filter.rank`. *
- New function for small object removal `skimage.morphology.remove_small_objects`
- New circular hough transformation `skimage.transform.hough_circle`
- New function to draw circle perimeter `skimage.draw.circle_perimeter` and ellipse perimeter `skimage.draw.ellipse_perimeter`
- New dense DAISY feature descriptor `skimage.feature.daisy`
- New bilateral filter `skimage.filter.denoise_bilateral`
- New faster TV denoising filter based on split-Bregman algorithm `skimage.filter.denoise_tv_bregman`
- New linear hough peak detection `skimage.transform.hough_peaks`
- New Scharr edge detection `skimage.filter.scharr`
- New geometric image scaling as convenience function `skimage.transform.rescale`
- New theme for documentation and website
- Faster median filter through vectorization `skimage.filter.median_filter`
- Grayscale images supported for SLIC segmentation
- Unified peak detection with more options `skimage.feature.peak_local_max`
- `imread` can read images via URL and knows more formats `skimage.io.imread`

Additionally, this release adds lots of bug fixes, new examples, and performance enhancements.

Contributors to this release

This release was only possible due to the efforts of many contributors, both new and old.

- Adam Ginsburg
- Anders Boesen Lindbo Larsen
- Andreas Mueller
- Christoph Gohlke
- Christos Psaltis
- Colin Lea
- François Boulogne
- Jan Margeta
- Johannes Schönberger
- Josh Warner (Mac)
- Juan Nunez-Iglesias
- Luis Pedro Coelho
- Marianne Corvellec
- Matt McCormick
- Nicolas Pinto
- Olivier Debeir
- Paul Ivanov
- Sergey Karayev
- Stefan van der Walt
- Steven Silvester
- Thouis (Ray) Jones
- Tony S Yu

1.4.4 scikits-image 0.7.0 release notes

We're happy to announce the 7th version of scikits-image!

Scikits-image is an image processing toolbox for SciPy that includes algorithms for segmentation, geometric transformations, color space manipulation, analysis, filtering, morphology, feature detection, and more.

For more information, examples, and documentation, please visit our website

<http://scikit-image.org>

New Features

It's been only 3 months since scikits-image 0.6 was released, but in that short time, we've managed to add plenty of new features and enhancements, including

- Geometric image transforms
- 3 new image segmentation routines (Felsenzwalb, Quickshift, SLIC)
- Local binary patterns for texture characterization
- Morphological reconstruction
- Polygon approximation
- CIE Lab color space conversion
- Image pyramids
- Multispectral support in random walker segmentation
- Slicing, concatenation, and natural sorting of image collections
- Perimeter and coordinates measurements in regionprops
- An extensible image viewer based on Qt and Matplotlib, with plugins for edge detection, line-profiling, and viewing image collections

Plus, this release adds a number of bug fixes, new examples, and performance enhancements.

Contributors to this release

This release was only possible due to the efforts of many contributors, both new and old.

- Andreas Mueller
- Andreas Wuerl
- Andy Wilson
- Brian Holt
- Christoph Gohlke
- Dharhas Pothina
- Emmanuelle Gouillart
- Guillaume Gay
- Josh Warner
- James Bergstra
- Johannes Schonberger
- Jonathan J. Helmus
- Juan Nunez-Iglesias
- Leon Tietz
- Marianne Corvellec
- Matt McCormick
- Neil Yager

- Nicolas Pinto
- Nicolas Poilvert
- Pavel Campr
- Petter Strandmark
- Stefan van der Walt
- Tim Sheerman-Chase
- Tomas Kazmar
- Tony S Yu
- Wei Li

1.4.5 scikits-image 0.6 release notes

We're happy to announce the 6th version of scikits-image!

Scikits-image is an image processing toolbox for SciPy that includes algorithms for segmentation, geometric transformations, color space manipulation, analysis, filtering, morphology, feature detection, and more.

For more information, examples, and documentation, please visit our website

<http://skimage.org>

New Features

- Packaged in Debian as `python-skimage`
 - Template matching
 - Fast user-defined image warping
 - Adaptive thresholding
 - Structural similarity index
 - Polygon, circle and ellipse drawing
 - Peak detection
 - Region properties
 - TiffFile I/O plugin
- ... along with some bug fixes and performance tweaks.

Contributors to this release

- Vincent Albufera
- David Cournapeau
- Christoph Gohlke
- Emmanuelle Gouillart
- Pieter Holtzhausen
- Zachary Pincus

- Johannes Schönberger
- Tom (tangofoxtrotmike)
- James Turner
- Stefan van der Walt
- Tony S Yu

1.4.6 scikits-image 0.5 release notes

We're happy to announce the 0.5 release of scikits-image, our image processing toolbox for SciPy.

For more information, please visit our website

<http://scikits-image.org>

New Features

- Consistent intensity rescaling and improved range conversion.
- Random walker segmentation.
- Harris corner detection.
- Otsu thresholding.
- Block views, window views and montage.
- Plugin for Christoph Gohlke's "tifffile".
- Peak detection.
- Improved FreeImage wrappers and meta-data reading.
- 8-neighbor and background labelling.

... along with updates to the documentation and website, and a number of bug fixes.

Contributors to this release

- Andreas Mueller
- Brian Holt
- Christoph Gohlke
- Emmanuelle Gouillart
- Michael Aye
- Nelle Varoquaux
- Nicolas Pinto
- Nicolas Poilvert
- Pieter Holtzhausen
- Stefan van der Walt
- Tony S Yu
- Warren Weckesser

- Zachary Pincus

1.4.7 scikits-image 0.4 release notes

We're happy to announce the 0.4 release of scikits-image, an image processing toolbox for SciPy.

Please visit our examples gallery to see what we've been up to:

http://scikits-image.org/docs/0.4/auto_examples/

Note that, in this release, we renamed the module from `scikits.image` to `skimage`, to work around name space conflicts with other scikits (similarly, the machine learning scikit is now imported as `sklearn`).

A big shout-out also to everyone currently at SciPy India; have fun, and remember to join the scikits-image sprint!

This release runs under all major operating systems where Python (>=2.6 or 3.x), NumPy and SciPy can be installed.

For more information, visit our website

<http://scikits-image.org>

New Features

- Module rename from `scikits.image` to `skimage`
 - Contour finding
 - Grey-level co-occurrence matrices
 - Skeletonization and medial axis transform
 - Convex hull images
 - New test data sets
 - GDAL I/O plugin
- ... as well as some bug fixes.

Contributors to this release

- Andreas Mueller
- Christopher Gohlke
- Emmanuelle Gouillart
- Neil Yager
- Nelle Varoquaux
- Riaan van den Dool
- Stefan van der Walt
- Thouis (Ray) Jones
- Tony S Yu
- Zachary Pincus

1.4.8 scikits.image 0.3 release notes

After a brief (!) absence, we're back with a new and shiny version of scikits.image, the image processing toolbox for SciPy.

This release runs under all major operating systems where Python (>=2.6 or 3.x), NumPy and SciPy can be installed.

For more information, visit our website

<http://scikits-image.org>

or the examples gallery at

http://scikits-image.org/docs/0.3/auto_examples/

New Features

- Shortest paths
 - Total variation denoising
 - Hough and probabilistic Hough transforms
 - Radon transform with reconstruction
 - Histogram of gradients
 - Morphology, including watershed, connected components
 - Faster homography transformations (rotations, zoom, etc.)
 - Image dtype conversion routines
 - Line drawing
 - Better image collection handling
 - Constant time median filter
 - Edge detection (canny, sobel, etc.)
 - IO: Freeimage, FITS, Qt and other image loaders; video support.
 - SIFT feature loader
 - Example data-sets
- ... as well as many bug fixes and minor updates.

Contributors for this release

Martin Bergtholdt Luis Pedro Coelho Chris Colbert Damian Eads Dan Farmer Emmanuelle Gouillart Brian Holt Pieter Holtzhausen Thouis (Ray) Jones Lee Kamentsky Almar Klein Kyle Mandli Andreas Mueller Neil Muller Zachary Pincus James Turner Stefan van der Walt Gael Varoquaux Tony Yu

1.4.9 scikit-image 0.21.0 release notes

We're happy to announce the release of scikit-image 0.21.0! scikit-image is an image processing toolbox for SciPy that includes algorithms for segmentation, geometric transformations, color space manipulation, analysis, filtering, morphology, feature detection, and more.

For more information, examples, and documentation, please visit our website: <https://scikit-image.org>

Highlights

- Last release to support Python 3.8
- Unified API for PRNGs

New Features

- Implement Fisher vectors in scikit-image ([#5349](#)).
- Add support for y-dimensional shear to the AffineTransform ([#6752](#)).

API Changes

In this release, we unify the way seeds are specified for algorithms that make use of pseudo-random numbers. Before, various keyword arguments (`sample_seed`, `seed`, `random_seed`, and `random_state`) served the same purpose in different places. These have all been replaced with a single `rng` argument, that handles both integer seeds and NumPy Generators. Please see the related SciPy discussion, as well as [Scientific Python SPEC 7](#) that attempts to summarize the argument.

- Unify API on seed keyword for random seeds / generator ([#6258](#)).
- Refactor `_invariant_denoise` to `denoise_invariant` ([#6660](#)).
- Expose `color.get_xyz_coords` in public API ([#6696](#)).
- Make `join_segmentations` return array maps from output to input labels ([#6786](#)).
- Unify pseudo-random seeding interface ([#6922](#)).
- Change geometric transform inverse to property ([#6926](#)).

Enhancements

- Bounding box crop ([#5499](#)).
- Add support for y-dimensional shear to the AffineTransform ([#6752](#)).
- Make `join_segmentations` return array maps from output to input labels ([#6786](#)).
- Check if `spacing` parameter is tuple in `regionprops` ([#6907](#)).
- Enable use of `rescale_intensity` with dask array ([#6910](#)).

Performance

- Add lazy loading to skimage.color submodule ([#6967](#)).
- Add Lazy loading to skimage.draw submodule ([#6971](#)).
- Add Lazy loader to skimage.exposure ([#6978](#)).
- Add lazy loading to skimage.future module ([#6981](#)).

Bug Fixes

- Fix and refactor *deprecated* decorator to *deprecate_func* ([#6594](#)).
- Refactor *_invariant_denoise* to *denoise_invariant* ([#6660](#)).
- Expose *color.get_xyz_coords* in public API ([#6696](#)).
- shift and normalize data before fitting circle or ellipse ([#6703](#)).
- Showcase pydata-sphinx-theme ([#6714](#)).
- Fix matrix calculation for shear angle in *AffineTransform* ([#6717](#)).
- Fix threshold_li(): prevent log(0) on single-value background. ([#6745](#)).
- Fix copy-paste error in *footprints.diamond* test case ([#6756](#)).
- Update .devpy/cmds.py to match latest devpy ([#6789](#)).
- Avoid installation of rtoml via conda in installation guide ([#6792](#)).
- Raise error in skeletonize for invalid value to method param ([#6805](#)).
- Sign error fix in measure.regionprops for orientations of 45 degrees ([#6836](#)).
- Fix returned data type in *segmentation.watershed* ([#6839](#)).
- Handle NaNs when clipping in *transform.resize* ([#6852](#)).
- Fix failing regionprop_table for multichannel properties ([#6861](#)).
- Do not allow 64-bit integer inputs; add test to ensure masked and unmasked modes are aligned ([#6875](#)).
- Fix typo in apply_parallel introduced in #6876 ([#6881](#)).
- Fix LPI filter for data with even dimensions ([#6883](#)).
- Use legacy datasets without creating a *data_dir* ([#6886](#)).
- Raise error when source_range is not correct ([#6898](#)).
- apply spacing rescaling when computing centroid_weighted ([#6900](#)).
- Corrected energy calculation in Chan Vese ([#6902](#)).
- Add missing backticks to DOI role in docstring of *area_opening* ([#6913](#)).
- Fix inclusion of *random.js* in HTML output ([#6935](#)).
- Fix URL of random gallery links ([#6937](#)).
- Use context manager to ensure urlopen buffer is closed ([#6942](#)).
- Fix sparse index type casting in skimage.graph._ncut ([#6975](#)).

Maintenance

- Fix and refactor *deprecated* decorator to *deprecate_func* (#6594).
- allow trivial ransac call (#6755).
- Fix copy-paste error in *footprints.diamond* test case (#6756).
- Use imageio v3 API (#6764).
- Unpin scipy dependency (#6773).
- Update .devpy/cmds.py to match latest devpy (#6789).
- Relicense CLAHE code under BSD-3-Clause (#6795).
- Relax reproduce section in bug issue template (#6825).
- Rename devpy to spin (#6842).
- Speed up threshold_local function by fixing call to _supported_float_type (#6847).
- Specify kernel for ipywidgets (#6849).
- Make *image_fetcher* and *create_image_fetcher* in *data* private (#6855).
- Update references to outdated dev.py with spin (#6856).
- Bump 0.21 removals to 0.22 (#6868).
- Update dependencies (#6869).
- Update pre-commits (#6870).
- Add test for radon transform on circular phantom (#6873).
- Do not allow 64-bit integer inputs; add test to ensure masked and unmasked modes are aligned (#6875).
- Don't use mutable types as default values for arguments (#6876).
- Point *version_switcher.json* URL at dev docs (#6882).
- Add back parallel tests that were removed as part of Meson build (#6884).
- Use legacy datasets without creating a *data_dir* (#6886).
- Remove old doc cruft (#6901).
- Temporarily pin imageio to <2.28 (#6909).
- Unify pseudo-random seeding interface follow-up (#6924).
- Use pytest.warn instead of custom context manager (#6931).
- Follow-up to move to pydata-sphinx-theme (#6933).
- Mark functions as *noexcept* to support Cython 3 (#6936).
- Skip unstable test in *ransac*'s docstring (#6938).
- Stabilize EllipseModel fitting parameters (#6943).
- Point logo in generated HTML docs at scikit-image.org (#6947).
- If user provides RNG, spawn it before deepcopying (#6948).
- Skip ransac doctest (#6953).
- Expose *GeometricTransform.residuals* in HTML doc (#6968).
- Fix NumPy 1.25 deprecation warnings (#6969).

- Revert jupyterlite (#6972).
- Don't test numpy nightlies due to transcendental functions issue (#6973).
- Ignore tight layout warning from matplotlib pre-release (#6976).
- Remove temporary constraint <2.28 for imageio (#6980).

Documentation

- Document boundary behavior of *draw.polygon* and *draw.polygon2mask* (#6690).
- Showcase pydata-sphinx-theme (#6714).
- Merge duplicate instructions for setting up build environment. (#6770).
- Add docstring to *skimage.color* module (#6777).
- DOC: Fix underline length in *docstring_add_deprecated* (#6778).
- Link full license to README (#6779).
- Fix conda instructions for dev env setup. (#6781).
- Update docstring in *skimage.future* module (#6782).
- Remove outdated build instructions from README (#6788).
- Add docstring to the *transform* module (#6797).
- Handle pip-only dependencies when using conda. (#6806).
- Added examples to the EssentialMatrixTransform class and its estimation function (#6832).
- Fix returned data type in *segmentation.watershed* (#6839).
- Update references to outdated dev.py with spin (#6856).
- Added example to AffineTransform class (#6859).
- Update _warps_cy.pyx (#6867).
- Point *version_switcher.json* URL at dev docs (#6882).
- Fix docstring underline lengths (#6895).
- ENH Add JupyterLite button to gallery examples (#6911).
- Add missing backticks to DOI role in docstring of *area_opening* (#6913).
- Add 0.21 release notes (#6925).
- Simplify installation instruction document (#6927).
- Follow-up to move to pydata-sphinx-theme (#6933).
- Update release notes (#6944).
- MNT Fix typo in JupyterLite comment (#6945).
- Point logo in generated HTML docs at scikit-image.org (#6947).
- Add missing PRs to release notes (#6949).
- fix bad link in CODE_OF_CONDUCT.md (#6952).
- Expose *GeometricTransform.residuals* in HTML doc (#6968).

Infrastructure

- Showcase pydata-sphinx-theme ([#6714](#)).
- Prepare CI configuration for merge queue ([#6771](#)).
- Pin to devpy 0.1 tag ([#6816](#)).
- Relax reproduce section in bug issue template ([#6825](#)).
- Rename devpy to spin ([#6842](#)).
- Use lazy loader 0.2 ([#6844](#)).
- Cleanup cruft in tools ([#6846](#)).
- Update pre-commits ([#6870](#)).
- Remove *codecov* dependency which disappeared from PyPI ([#6887](#)).
- Add CircleCI API token; fixes status link to built docs ([#6894](#)).
- Temporarily pin imageio to <2.28 ([#6909](#)).
- Add PR links to release notes generating script ([#6917](#)).
- Use official meson-python release ([#6928](#)).
- Fix inclusion of *random.js* in HTML output ([#6935](#)).
- Fix URL of random gallery links ([#6937](#)).
- Respect SPHINXOPTS and add –install-deps flags to *spin docs* ([#6940](#)).
- Build skimage before generating docs ([#6946](#)).
- Enable testing against nightly upstream wheels ([#6956](#)).
- Add nightly wheel builder ([#6957](#)).
- Run weekly tests on nightly wheels ([#6959](#)).
- CI: ensure that a “type: “ label is present on each PR ([#6960](#)).
- Add PR milestone labeler ([#6977](#)).

33 authors added to this release (alphabetical)

- Adam J. Stewart (@adamjstewart)
- Adeyemi Biola (@decorouz)
- aeisenbarth (@aeisenbarth)
- Ananya Srivastava (@ana42742)
- Bohumír Zámečník (@bzamecnik)
- Carlos Horn (@carloshorn)
- Daniel Angelov (@23pointsNorth)
- DavidTorpey (@DavidTorpey)
- Dipkumar Patel (@immortal3)
- Enrico Tagliavini (@enricotagliavini)
- Eric Prestat (@ericpre)

- GGoussar (@GGoussar)
- Gregory Lee (@grlee77)
- harshitha kolipaka (@harshithakolipaka)
- Hayato Ikoma (@hayatoikoma)
- i-aki-y (@i-aki-y)
- Jake Martin (@jakeMartin1234)
- Jarrod Millman (@jarrodmillman)
- Juan Nunez-Iglesias (@jni)
- Kevin MEETOOA (@kevinmeetooa)
- Lars Grüter (@lagru)
- Loïc Estève (@lesteve)
- mahamtariq58 (@mahamtariq58)
- Marianne Corvellec (@mkcor)
- Mark Harfouche (@hmaarrfk)
- Matthias Bussonnier (@Carreau)
- Matus Valo (@matusvalo)
- Michael Görner (@v4hn)
- Ramyashri Padmanabhakumar (@rum1887)
- scott-vsi (@scott-vsi)
- Sean Quinn (@seanpquinn)
- Stefan van der Walt (@stefanv)
- Tony Reina (@tonyreina)

27 reviewers added to this release (alphabetical)

- Adeyemi Biola (@decorouz)
- aeisenbarth (@aeisenbarth)
- Ananya Srivastava (@ana42742)
- Brigitta Sipőcz (@bsipocz)
- Carlos Horn (@carloshorn)
- Cris Luengo (@crisluento)
- DavidTorpey (@DavidTorpey)
- Dipkumar Patel (@immortal3)
- Enrico Tagliavini (@enricotagliavini)
- Gregory Lee (@grlee77)
- Henry Pinkard (@henrypinkard)
- i-aki-y (@i-aki-y)

- Jarrod Millman (@jarrodmillman)
- Juan Nunez-Iglesias (@jni)
- Kevin MEETOOA (@kevinmeetooa)
- kzuiderveld (@kzuiderveld)
- Lars Grüter (@lagru)
- Marianne Corvellec (@mkcor)
- Mark Harfouche (@hmaarrfk)
- Ramyashri Padmanabhakumar (@rum1887)
- Riadh Fezzani (@rfezzani)
- Sean Quinn (@seanpquinn)
- Sebastian Berg (@seberg)
- Sebastian Wallkötter (@FirefoxMetzger)
- Stefan van der Walt (@stefanv)
- Tony Reina (@tonyreina)
- Tony Reina (@tony-res)

1.4.10 scikit-image 0.20.0 release notes

scikit-image is an image processing toolbox built on SciPy that includes algorithms for segmentation, geometric transformations, color space manipulation, analysis, filtering, morphology, feature detection, and more.

For more information, examples, and documentation, please visit our website: <https://scikit-image.org>

With this release, many of the functions in `skimage.measure` now support anisotropic images with different voxel spacings.

Many performance improvements were made, such as support for footprint decomposition in `skimage.morphology`

Four new gallery examples were added to the documentation, including the new interactive example “Track solidification of a metallic alloy”.

This release completes the transition to a more flexible `channel_axis` parameter for indicating multi-channel images, and includes several other deprecations that make the API more consistent and expressive.

Finally, in preparation for the removal of `distutils` in the upcoming Python 3.12 release, we replaced our build system with `meson` and a static `pyproject.toml` specification.

This release supports Python 3.8–3.11.

New features and improvements

- Support footprint decomposition to several footprint generating and consuming functions in `skimage.morphology`. By decomposing a footprint into several smaller ones, morphological operations can potentially be sped up. The decomposed footprint can be generated with the new `decomposition` parameter of the functions `rectangle`, `diamond`, `disk`, `cube`, `octahedron`, `ball`, and `octagon` in `skimage.morphology`. The `footprint` parameter of the functions `binary_erosion`, `binary_dilation`, `binary_opening`, `binary_closing`, `erosion`, `dilation`, `opening`, `closing`, `white_tophat`, and `black_tophat` in `skimage.morphology` now accepts a sequence of 2-element tuples (`footprint_i`, `num_iter_i`) where each entry, `i`, of the sequence contains a footprint and the number of times it should be iteratively applied. This is the form produced by the footprint decompositions mentioned above ([#5482](#), [#6151](#)).
- Support anisotropic images with different voxel spacings. Spacings can be defined with the new parameter `spacing` of the following functions in `skimage.measure`: `regionprops`, `regionprops_table`, `moments`, `moments_central`, `moments_normalized`, `centroid`, `inertia_tensor`, and `inertia_tensor_eigvals`. Voxel spacing is taken into account for the following existing properties in `skimage.measure`. `regionprops`: `area`, `area_bbox`, `centroid`, `area_convex`, `extent`, `feret_diameter_max`, `area_filled`, `inertia_tensor`, `moments`, `moments_central`, `moments_hu`, `moments_normalized`, `perimeter`, `perimeter_crofton`, `solidity`, `moments_weighted_central`, and `moments_weighted_hu`. The new properties `num_pixels` and `coords_scaled` are available as well. See the respective docstrings for more details ([#6296](#)).
- Add isotropic binary morphological operators `isotropic_closing`, `isotropic_dilation`, `isotropic_erosion`, and `isotropic_opening` in `skimage.morphology`. These functions return the same results as their non-isotropic counterparts but perform faster for large circular structuring elements ([#6492](#)).
- Add new colocalization metrics `pearson_corr_coeff`, `manders_coloc_coeff`, `manders_overlap_coeff` and `intersection_coeff` to `skimage.measure` ([#6189](#)).
- Support the Modified Hausdorff Distance (MHD) metric in `skimage.metrics.hausdorff_distance` via the new parameter `method`. The MHD can be more robust against outliers than the directed Hausdorff Distance (HD) ([#5581](#)).
- Add two datasets `skimage.data.protein_transport` and `skimage.data.nickel_solidification` ([#6087](#)).
- Add new parameter `use_gaussian_derivatives` to `skimage.feature.hessian_matrix` which allows the computation of the Hessian matrix by convolving with Gaussian derivatives ([#6149](#)).
- Add new parameters `squared_butterworth` and `npad` to `skimage.filters.butterworth`, which support traditional or squared filtering and edge padding, respectively ([#6251](#)).
- Support construction of a `skimage.io.ImageCollection` from a `load_pattern` with an arbitrary sequence as long as a matching `load_func` is provided ([#6276](#)).
- Add new parameter `alpha` to `skimage.metrics.adapted_rand_error` allowing control over the weight given to precision and recall ([#6472](#)).
- Add new parameter `binarize` to `skimage.measure.grid_points_in_poly` to optionally return labels that tell whether a pixel is inside, outside, or on the border of the polygon ([#6515](#)).
- Add new parameter `include_borders` to `skimage.measure.convex_hull_image` to optionally exclude vertices or edges from the final hull mask ([#6515](#)).
- Add new parameter `offsets` to `skimage.measure.regionprops` that optionally allows specifying the coordinates of the origin and affects the properties `coords_scaled` and `coords` ([#3706](#)).
- Add new parameter `disambiguate` to `skimage.registration.phase_cross_correlation` to optionally disambiguate periodic shifts ([#6617](#)).

- Support n-dimensional images in `skimage.filters.farid` (Farid & Simoncelli filter) (#6257).
- Support n-dimensional images in `skimage.restoration.wiener` (#6454).
- Support three dimensions for the properties `rotation` and `translation` in `skimage.transform.EuclideanTransform` as well as for `skimage.transform.SimilarityTransform.scale` (#6367).
- Allow footprints with non-adjacent pixels as neighbors in `skimage.morphology.flood_fill` (#6236).
- Support array-likes consistently in `AffineTransform`, `EssentialMatrixTransform`, `EuclideanTransform`, `FundamentalMatrixTransform`, `GeometricTransform`, `PiecewiseAffineTransform`, `PolynomialTransform`, `ProjectiveTransform`, `SimilarityTransform`, `estimate_transform`, and `matrix_transform` in `skimage.transform` (#6270).

Performance

- Improve performance (~2x speedup) of `skimage.feature.canny` by porting a part of its implementation to Cython (#6387).
- Improve performance (~2x speedup) of `skimage.feature.hessian_matrix_eigvals` and 2D `skimage.feature.structure_tensor_eigenvalues` (#6441).
- Improve performance of `skimage.measure.moments_central` by avoiding redundant computations (#6188).
- Reduce import time of `skimage.io` by loading the `matplotlib` plugin only when required (#6550).
- Incorporate RANSAC improvements from scikit-learn into `skimage.measure.ransac` which decrease the number of iterations (#6046).
- Improve histogram matching performance on unsigned integer data with `skimage.exposure.match_histograms`. (#6209, #6354).
- Reduce memory consumption of the ridge filters `meijering`, `sato`, `frangi`, and `hessian` in `skimage.filters` (#6509).
- Reduce memory consumption of `blob_dog`, `blob_log`, and `blob_doh` in `skimage.feature` (#6597).
- Use minimal required unsigned integer size internally in `skimage.morphology.reconstruction` which allows to operate the function with higher precision or on larger arrays. Previously, `int32` was used. (#6342).
- Use minimal required unsigned integer size in `skimage.filters.rank_order` which allows to operate the function with higher precision or on larger arrays. Previously, the returned `labels` and `original_values` were always of type `uint32`. (#6342).

Changes and new deprecations

- Set Python 3.8 as the minimal supported version (#6679).
- Rewrite `skimage.filters.meijering`, `skimage.filters.sato`, `skimage.filters.frangi`, and `skimage.filters.hessian` to match the published algorithms more closely. This change is backward incompatible and will lead to different output values compared to the previous implementation. The Hessian matrix calculation is now done more accurately. The filters will now be correctly set to zero whenever one of the Hessian eigenvalues has a sign which is incompatible with a ridge of the desired polarity. The gamma constant of the Frangi filter is now set adaptively based on the maximum Hessian norm (#6446).
- Move functions in `skimage.future.graph` to `skimage.graph`. This affects `cut_threshold`, `cut_normalized`, `merge_hierarchical`, `rag_mean_color`, `RAG`, `show_rag`, and `rag_boundary` (#6674).
- Return `False` in `skimage.measure.LineModelND.estimate` instead of raising an error if the model is under-determined (#6453).

- Return `False` in `skimage.measure.CircleModel.estimate` instead of warning if the model is under-determined ([#6453](#)).
- Rename `skimage.filters.inverse` to `skimage.filters.inverse_filter`. `skimage.filters.inverse` is deprecated and will be removed in the next release ([#6418](#), [#6701](#)).
- Update minimal supported dependencies to `numpy>=1.20` ([#6565](#)).
- Update minimal supported dependencies to `scipy>=1.8` ([#6564](#)).
- Update minimal supported dependencies to `networkx>=2.8` ([#6564](#)).
- Update minimal supported dependency to `pillow>=9.0.1` ([#6402](#)).
- Update minimal supported dependency to `setuptools` 67 ([#6754](#)).
- Update optional, minimal supported dependency to `matplotlib>=3.3` ([#6383](#)).
- Warn for non-integer image inputs to `skimage.feature.local_binary_pattern`. Applying the function to floating-point images may give unexpected results when small numerical differences between adjacent pixels are present ([#6272](#)).
- Warn if `skimage.registration.phase_cross_correlation` returns only the shift vector. Starting with the next release this function will always return a tuple of three (shift vector, error, phase difference). Use `return_error="always"` to silence this warning and switch to this new behavior ([#6543](#)).
- Warn in `skimage.metrics.structural_similarity`, if `data_range` is not specified in case of floating point data ([#6612](#)).
- Automatic detection of the color channel is deprecated in `skimage.filters.gaussian` and a warning is emitted if the parameter `channel_axis` is not set explicitly ([#6583](#)).

Completed deprecations

- Remove `skimage.viewer` which was scheduled for removal in the postponed version 1.0 ([#6160](#)).
- Remove deprecated parameter `indices` from `skimage.feature.peak_local_max` ([#6161](#)).
- Remove `skimage.feature.structure_tensor_eigvals` (it was replaced by `skimage.feature.structure_tensor_eigenvalues`) and change the default parameter value in `skimage.feature.structure_tensor` to `order="rc"` ([#6162](#)).
- Remove deprecated parameter `array` in favor of `image` from `skimage.measure.find_contours` ([#6163](#)).
- Remove deprecated Qt IO plugin and the `skivi` console script ([#6164](#)).
- Remove deprecated parameter value `method='lorensen'` in `skimage.measure.marching_cubes` ([#6230](#)).
- Remove deprecated parameter `multichannel`; use `channel_axis` instead. This affects `skimage.draw.random_shapes`, `skimage.exposure.match_histograms`, `skimage.feature.multiscale_basic_features`, `skimage.feature.hog`, `skimage.feature.difference_of_gaussians`, `skimage.filters.unsharp_mask`, and `skimage.metrics.structural_similarity`. In `skimage.restoration`, this affects `cycle_spin`, `denoise_bilateral`, `denoise_tv_bregman`, `denoise_tv_chambolle`, `denoise_wavelet`, `estimate_sigma`, `inpaint_biharmonic`, and `denoise_nl_means`. In `skimage.segmentation`, this affects `felzenszwalb`, `random_walker`, and `slic`. In `skimage.transform`, this affects `rescale`, `warp_polar`, `pyramid_reduce`, `pyramid_expand`, `pyramid_gaussian`, and `pyramid_laplacian`. In `skimage.util`, this affects `montage` and `apply_parallel` ([#6583](#)).
- Remove deprecated parameter `selem`; use `footprint` instead. In `skimage.filters`, this affects `median`, `autolevel_percentile`, `gradient_percentile`, `mean_percentile`, `subtract_mean_percentile`, `enhance_contrast_percentile`, `percentile`, `pop_percentile`,

`sum_percentile`, `threshold_percentile`, `mean_bilateral`, `pop_bilateral`, `sum_bilateral`, `autolevel`, `equalize`, `gradient`, `maximum`, `mean`, `geometric_mean`, `subtract_mean`, `median`, `minimum`, `modal`, `enhance_contrast`, `pop`, `sum`, `threshold`, `noise_filter`, `entropy`, `otsu`, `windowed_histogram`, and `majority`. In `skimage.morphology`, this affects `flood_fill`, `flood`, `binary_erosion`, `binary_dilation`, `binary_opening`, `binary_closing`, `h_maxima`, `h_minima`, `local_maxima`, `local_minima`, `erosion`, `dilation`, `opening`, `closing`, `white_tophat`, `black_tophat`, and `reconstruction` ([#6583](#)).

- Remove deprecated parameter `max_iter` from `skimage.filters.threshold_minimum`, `skimage.morphology.thin`, and `skimage.segmentation.chan_vese`; use `max_num_iter` instead ([#6583](#)).
- Remove deprecated parameter `max_iterations` from `skimage.segmentation.active_contour`; use `max_num_iter` instead ([#6583](#)).
- Remove deprecated parameter `input` from `skimage.measure.label`; use `label_image` instead ([#6583](#)).
- Remove deprecated parameter `coordinates` from `skimage.measure.regionprops` and `skimage.segmentation.active_contour` ([#6583](#)).
- Remove deprecated parameter `neighbourhood` from `skimage.measure.perimeter`; use `neighborhood` instead ([#6583](#)).
- Remove deprecated parameters `height` and `width` from `skimage.morphology.rectangle`; use `ncols` and `nrows` instead ([#6583](#)).
- Remove deprecated parameter `in_place` from `skimage.morphology.remove_small_objects`, `skimage.morphology.remove_small_holes`, and `skimage.segmentation.clear_border`; use `out` instead ([#6583](#)).
- Remove deprecated parameter `iterations` from `skimage.restoration.richardson_lucy`, `skimage.segmentation.morphological_chan_vese`, and `skimage.segmentation.morphological_geodesic_active_contour`; use `num_iter` instead ([#6583](#)).
- Remove support for deprecated keys `"min_iter"` and `"max_iter"` in `skimage.restoration.unsupervised_wiener`'s parameter `user_params`; use `"min_num_iter"` and `"max_num_iter"` instead ([#6583](#)).
- Remove deprecated functions `greycomatrix` and `greycoprops` from `skimage.feature` ([#6583](#)).
- Remove deprecated submodules `skimage.morphology.grey` and `skimage.morphology.greyreconstruct`; use `skimage.morphology` instead ([#6583](#)).
- Remove deprecated submodule `skimage.morphology.selem`; use `skimage.morphology.footprints` instead ([#6583](#)).
- Remove deprecated `skimage.future.graph.ncut` (it was replaced by `skimage.graph.cut_normalized`) ([#6685](#)).

Bug fixes

- Fix round-off error in `skimage.exposure.adjust_gamma` ([#6285](#)).
- Round and convert output coordinates of `skimage.draw.rectangle` to `int` even if the input coordinates use `float`. This fix ensures that the output can be used for indexing similar to other draw functions ([#6501](#)).
- Avoid unexpected exclusion of peaks near the image border in `skimage.feature.peak_local_max` if the peak value is smaller 0 ([#6502](#)).
- Avoid anti-aliasing in `skimage.transform.resize` by default when using nearest neighbor interpolation (`order == 0`) with an integer input data type ([#6503](#)).

- Use mask during rescaling in `skimage.segmentation.slic`. Previously, the mask was ignored when rescaling the image to make choice of compactness insensitive to the image values. The new behavior makes it possible to mask values such as `numpy.nan` or `numpy.inf`. Additionally, raise an error if the input `image` has two dimensions and a `channel_axis` is specified - indicating that the image is multi-channel (#6525).
- Fix unexpected error when passing a tuple to the parameter `exclude_border` in `skimage.feature.blob_dog` and `skimage.feature.blob_log` (#6533).
- Raise a specific error message in `skimage.segmentation.random_walker` if no seeds are provided as positive values in the parameter `labels` (#6562).
- Raise a specific error message when accessing region properties from `skimage.measure.regionprops` when the required `intensity_image` is unavailable (#6584).
- Avoid errors in `skimage.feature.ORB.detect_and_extract` by breaking early if the octave image is too small (#6590).
- Fix `skimage.restoration.inpaint_biharmonic` for images with Fortran-ordered memory layout (#6263).
- Fix automatic detection of the color channel in `skimage.filters.gaussian` (this behavior is deprecated, see new deprecations) (#6583).
- Fix stacklevel of warning in `skimage.color.lab2rgb` (#6616).
- Fix the order of return values for `skimage.feature.hessian_matrix` and raise an error if `order='xy'` is requested for images with more than 2 dimensions (#6624).
- Fix misleading exception in functions in `skimage.filters.rank` that did not mention that 2D images are also supported (#6666).
- Fix in-place merging of weights in `skimage.graph.RAG.merge_nodes` (#6692).
- Fix growing memory error and silence compiler warning in internal `heappush` function (#6727).
- Fix compilation warning about struct initialization in `Cascade.detect_multi_scale` (#6728).

Documentation

New

- Add gallery example “Decompose flat footprints (structuring elements)” (#6151).
- Add gallery example “Butterworth Filters” and improve docstring of `skimage.filters.butterworth` (#6251).
- Add gallery example “Render text onto an image” (#6431).
- Add gallery example “Track solidification of a metallic alloy” (#6469).
- Add gallery example “Colocalization metrics” (#6189).
- Add support page (`.github/SUPPORT.md`) to help users from GitHub find appropriate support resources (#6171, #6575).
- Add `CITATION.bib` to repository to help with citing scikit-image (#6195).
- Add usage instructions for new Meson-based build system with `dev.py` (#6600).

Improved & updated

- Improve gallery example “Measure perimeters with different estimators” (#6200, #6121).
- Adapt gallery example “Build image pyramids” to more diversified shaped images and downsample factors (#6293).
- Adapt gallery example “Explore 3D images (of cells)” with interactive slice explorer using plotly (#4953).
- Clarify meaning of the `weights` term and rewrite docstrings of `skimage.restoration.denoise_tv_bregman` and `skimage.restoration.denoise_tv_chambolle` (#6544).
- Describe the behavior of `skimage.io.MultiImage` more precisely in its docstring (#6290, #6292).
- Clarify that the enabled `watershed_line` parameter will not catch borders between adjacent marker regions in `skimage.segmentation.watershed` (#6280).
- Clarify that `skimage.morphology.skeletonize` accepts an `image` of any input type (#6322).
- Use gridded thumbnails in our gallery to demonstrate the different images and datasets available in `skimage.data` (#6298, #6300, #6301).
- Tweak balance in the docstring example of `skimage.restoration.wiener` for a less blurry result (#6265).
- Document support for Path objects in `skimage.io.imread` and `skimage.io.imsave` (#6361).
- Improve error message in `skimage.filters.threshold_multitotsu` if the discretized image cannot be thresholded (#6375).
- Show original unlabeled image as well in the gallery example “Expand segmentation labels without overlap” (#6396).
- Document refactoring of `grey*` to `skimage.feature.graymatrix` and `skimage.feature.graycoprops` in the release 0.19 (#6420).
- Document inclusion criteria for new functionality in core developer guide (#6488).
- Print the number of segments after applying the Watershed in the gallery example “Comparison of segmentation and superpixel algorithms” (#6535).
- Replace issue templates with issue forms (#6554, #6576).
- Expand reviewer guidelines in pull request template (#6208).
- Provide pre-commit PR instructions in pull request template (#6578).
- Warn about and explain the handling of floating-point data in the docstring of `skimage.metrics.structural_similarity` (#6595).
- Fix intensity autoscaling in animated `imshow` in gallery example “Measure fluorescence intensity at the nuclear envelope” (#6599).
- Clarify dependency on `scikit-image[data]` and `pooch` in `INSTALL.rst` (#6619).
- Don’t use confusing loop in installation instructions for `conda` (#6672).
- Document value ranges of $L^*a^*b^*$ and L^*Ch in `lab2xyz`, `rgb2lab`, `lab2lch`, and `lch2lab` in `skimage.color` (#6688, #6697, #6719).
- Use more consistent style in docstring of `skimage.feature.local_binary_pattern` (#6736).

Fixes, spelling & minor tweaks

- Remove deprecated reference and use `skimage.measure.marching_cubes` in gallery example “Marching Cubes” ([#6377](#)).
- List only the two primary OS-independent methods of installing scikit-image ([#6557](#), [#6560](#)).
- Fix description of `connectivity` parameter in the docstring of `skimage.morphology.flood` ([#6534](#)).
- Fix formatting in the docstring of `skimage.metrics.hausdorff_distance` ([#6203](#)).
- Fix typo in docstring of `skimage.measure.moments_hu` ([#6016](#)).
- Fix formatting of mode parameter in `skimage.util.random_noise` ([#6532](#)).
- Fix broken links in SKIP 3 ([#6445](#)).
- Fix broken link in docstring of `skimage.filters.sobel` ([#6474](#)).
- Change “neighbour” to EN-US spelling “neighbor” ([#6204](#)).
- Add missing copyrights to LICENSE.txt and use formatting according to SPDX identifiers ([#6419](#)).
- Include `skimage.morphology.footprint_from_sequence` in the public API documentation ([#6555](#)).
- Correct note about return type in the docstring of `skimage.exposure.rescale_intensity` ([#6582](#)).
- Stop using the `git://` connection protocol and remove references to it ([#6201](#), [#6283](#)).
- Update scikit-image’s mailing addresses to the new domain discuss.scientific-python.org ([#6255](#)).
- Remove references to deprecated mailing list in `doc/source/user_guide/getting_help.rst` ([#6575](#)).
- Use “center” in favor of “centre”, and “color” in favor of “colour” gallery examples ([#6421](#), [#6422](#)).
- Replace reference to `api_changes.rst` with `release_dev.rst` ([#6495](#)).
- Clarify header pointing to notes for latest version released ([#6508](#)).
- Add missing spaces to error message in `skimage.measure.regionprops` ([#6545](#)).
- Apply codespell to fix common spelling mistakes ([#6537](#)).
- Add missing space in math directive in `normalized_mutual_information`’s docstring ([#6549](#)).
- Fix lengths of docstring heading underline in `skimage.morphology.isotropic_` functions ([#6628](#)).
- Fix plot order due to duplicate examples with the file name `plot_thresholding.py` ([#6644](#)).
- Get rid of numpy deprecation warning in gallery example `plot_equalize` ([#6650](#)).
- Fix swapping of opening and closing in gallery example `plot_rank_filters` ([#6652](#)).
- Get rid of numpy deprecation warning in gallery example `plot_log_gamma.py` ([#6655](#)).
- Remove warnings and unnecessary messages in gallery example “Tinting gray-scale images” ([#6656](#)).
- Update the contribution guide to recommend creating the `virtualenv` outside the source tree ([#6675](#)).
- Fix typo in docstring of `skimage.data.coffee` ([#6740](#)).
- Add missing backtick in docstring of `skimage.graph.merge_nodes` ([#6741](#)).
- Fix typo in `skimage.metrics.variation_of_information` ([#6768](#)).

Other and development related updates

Governance & planning

- Add draft of SKIP 4 “Transitioning to scikit-image 2.0” (#6339, #6353).

Maintenance

- Prepare release notes for v0.20.0 (#6556, #6766).
- Add and test alternative build system based on Meson as an alternative to the deprecated distutils system (#6536).
- Use `cnp.float32_t` and `cnp.float64_t` over `float` and `double` in Cython code (#6303).
- Move `skimage/measure/mc_meta` folder into `tools/precompute/` folder to avoid its unnecessary distribution to users (#6294).
- Remove unused function `getLutNames` in `tools/precompute/mc_meta/createluts.py` (#6294).
- Point urls for data files to a specific commit (#6297).
- DropCodecov badge from project README (#6302).
- Remove undefined reference to '`python_to_notebook`' in `doc/ext/notebook_doc.py` (#6307).
- Parameterize tests in `skimage.measure.tests.test_moments` (#6323).
- Avoid unnecessary copying in `skimage.morphology.skeletonize` and update code style and tests (#6327).
- Fix typo in `_probabilistic_hough_line` (#6373).
- Derive `OBJECT_COLUMNS` from `COL_DTYPES` in `skimage.measure._regionprops` (#6389).
- Support `loadtxt` of NumPy 1.23 with `skimage/feature/orb_descriptor_positions.txt` (#6400).
- Exclude pillow 9.1.1 from supported requirements (#6384).
- Use the same numpy version dependencies for building as used by default (#6409).
- Forward-port v0.19.1 and v0.19.2 release notes (#6253).
- Forward-port v0.19.3 release notes (#6416).
- Exclude submodules of `doc.*` from package install (#6428).
- Substitute deprecated `vertices` with `simplices` in `skimage.transform._geometric` (#6430).
- Fix minor typo in `skimage.filters.sato` (#6434).
- Simplify sort-by-absolute-value in ridge filters (#6440).
- Removed completed items in `TODO.txt` (#6442).
- Remove duplicate import in `skimage.feature._canny` (#6457).
- Use `with open(...)` as `f` instead of `f = open(...)` (#6458).
- Use context manager when possible (#6484).
- Use `broadcast_to` instead of `as_strided` to generate broadcasted arrays (#6476).
- Use `moving_image` in docstring of `skimage.registration._optical_flow._tv1` (#6480).
- Use `pyplot.get_cmap` instead of deprecated `cm.get_cmap` in `skimage.future.graph.show_rag` for compatibility with matplotlib 3.3 to 3.6 (#6483, #6490).
- Update `plot_euler_number.py` for matplotlib 3.6 compatibility (#6522).
- Make non-functional change to `build.txt` to fix cache issue on CircleCI (#6528).

- Update deprecated field `license_file` to `license_files` in `setup.cfg` (#6529).
- Ignore codespell fixes with git blame (#6539).
- Remove `FUNDING.yml` in preference of org version (#6553).
- Handle pending changes to `tifffile.imwrite` defaults and avoid test warnings (#6460).
- Handle deprecation by updating to `networkx.to_scipy_sparse_array` (#6564).
- Update minimum supported numpy, scipy, and networkx (#6385).
- Apply linting results after enabling pre-commit in CI (#6568).
- Refactor lazy loading to use stubs & `lazy_loader` package (#6577).
- Update sphinx configuration (#6579).
- Update `pyproject.toml` to support Python 3.11 and to fix 32-bit pinned packages on Windows (#6519).
- Update primary email address in mailmap entry for grlee77 (#6639).
- Handle new warnings introduced in NumPy 1.24 (#6637).
- Remove unnecessary dependency on ninja in `pyproject.toml` (#6634).
- Pin to latest meson-python $\geq 0.11.0$ (#6627).
- Increase warning stacklevel by 1 in `skimage.color.lab2xyz` (#6613).
- Update OpenBLAS to v0.3.17 (#6607, #6610).
- Fix Meson build on windows in sync with SciPy (#6609).
- Set `check: true` for `run_command` in `skimage/meson.build` (#6606).
- Add `dev.py` and `setup` commands (#6600).
- Organize `dev.py` commands into sections (#6629).
- Remove `thumbnail_size` in config since `sphinx-gallery` $\geq 0.9.0$ (#6647).
- Add new test cases for `skimage.transform.resize` (#6669).
- Use meson-python main branch (#6671).
- Simplify `QhullError` import (#6677).
- Remove old SciPy cruft (#6678, #6681).
- Remove old references to `imread` package (#6680).
- Remove pillow cruft (and a few other cleanups) (#6683).
- Remove leftover `gtk_plugin.ini` (#6686).
- Prepare v0.20.0rc0 (#6706).
- Remove pre-release suffix for Python 3.11 (#6709).
- Loosen tests for SciPy 1.10 (#6715).
- Specify C flag only if supported by compiler (#6716).
- Extract version info from `skimage/__init__.py` in `skimage/meson.build` (#6723).
- Fix Cython errors/warnings (#6725).
- Generate `pyproject` deps from requirements (#6726).
- MAINT: Use `uintptr_t` to calculate new heap ptr positions (#6734).

- Bite the bullet: remove distutils and setup.py ([#6738](#)).
- Use meson-python developer version ([#6753](#)).
- Require `setuptools` 65.6+ ([#6751](#)).
- Remove `setup.cfg`, use `pypackage.toml` instead ([#6758](#)).
- Update `pypackage.toml` to use `meson-python>=0.13.0rc0` ([#6759](#)).

Benchmarks

- Add benchmarks for `morphology.local_maxima` ([#3255](#)).
- Add benchmarks for `skimage.morphology.reconstruction` ([#6342](#)).
- Update benchmark environment to Python 3.10 and NumPy 1.23 ([#6511](#)).

CI & automation

- Add Github actions/stale to label “dormant” issues and PRs ([#6506](#), [#6546](#), [#6552](#)).
- Fix the autogeneration of API docs for lazy loaded subpackages ([#6075](#)).
- Checkout gh-pages with a shallow clone ([#6085](#)).
- Fix dev doc build ([#6091](#)).
- Fix CI by excluding Pillow 9.1.0 ([#6315](#)).
- Pin pip pip to <22.1 in `tools/github/before_install.sh` ([#6379](#)).
- Update GH actions from v2 to v3 ([#6382](#)).
- Update to supported CircleCI images ([#6401](#)).
- Use artifact-redirector ([#6407](#)).
- Forward-port gh-6369: Fix windows wheels: use vsdevcmd.bat to make sure rc.exe is on the path ([#6417](#)).
- Restrict GitHub Actions permissions to required ones ([#6426](#), [#6504](#)).
- Update to Ubuntu LTS version on Actions workflows ([#6478](#)).
- Relax label name comparison in `benchmarks.yaml` workflow ([#6520](#)).
- Add linting via pre-commit ([#6563](#)).
- Add CI tests for Python 3.11 ([#6566](#)).
- Fix CI for Scipy 1.9.2 ([#6567](#)).
- Test optional Py 3.10 dependencies on MacOS ([#6580](#)).
- Pin setuptools in GHA MacOS workflow and `azure-pipelines.yml` ([#6626](#)).
- Build Python 3.11 wheels ([#6581](#)).
- Fix doc build on CircleCI and add ccache ([#6646](#)).
- Build wheels on CI via branch rather than tag ([#6668](#)).
- Do not build wheels on pushes to main ([#6673](#)).
- Use `tools/github/before_install.sh` wheels workflow ([#6718](#)).
- Use Ruff for linting ([#6729](#)).
- Use test that can fail for sdist ([#6731](#)).
- Fix fstring in `skimage._shared._warnings.expected_warnings` ([#6733](#)).

- Build macosx/py38 wheel natively ([#6743](#)).
- Remove CircleCI URL check ([#6749](#)).
- CI Set MACOSX_DEPLOYMENT_TARGET=10.9 for Wheels ([#6750](#)).
- Add temporary workaround until new meson-python release ([#6757](#)).
- Update action to use new environment file ([#6762](#)).
- Autogenerate pyproject.toml ([#6763](#)).

71 authors contributed to this release [alphabetical by first name or login]

- Adeel Hassan
- Albert Y. Shih
- AleixBP (AleixBP)
- Alex (sashashura)
- Alexandr Kalinin
- Alexandre de Siqueira
- Amin (MOAMSA)
- Antony Lee
- Balint Varga
- Ben Greiner
- bsmietanka (bsmietanka)
- Chris Roat
- Chris Wood
- Daria
- Dave Mellert
- Dudu Lasry
- Elena Pascal
- Eli Schwartz
- Fabian Schneider
- forgeRW (forgeRW)
- Frank A. Krueger
- Gregory Lee
- Gus Becker
- Hande Gözükhan
- Jacob Rosenthal
- James Gao
- Jan Kadlec
- Jan-Hendrik Müller

- Jan-Lukas Wynen
- Jarrod Millman
- Jeremy Muhlich
- johnthagen (johnthagen)
- Joshua Newton
- Juan DF
- Juan Nunez-Iglesias
- Judd Storrs
- Larry Bradley
- Lars Grüter
- lihaitao (li1127217ye)
- Lucas Johnson
- Malinda (maldil)
- Marianne Corvellec
- Mark Harfouche
- Martijn Courteaux
- Marvin Albert
- Matthew Brett
- Matthias Busonnier
- Miles Lucas
- Nathan Chan
- Naveen
- OBgoneSouth (OBgoneSouth)
- Oren Amsalem
- Preston Buscay
- Peter Sobolewski
- Peter Bell
- Ray Bell
- Riadh Fezzani
- Robin Thibaut
- Ross Barnowski
- samtygier (samtygier)
- Sandeep N Menon
- Sanghyeok Hyun
- Sebastian Berg
- Sebastian Wallkötter

- Simon-Martin Schröder
- Stefan van der Walt
- Teemu Kumpumäki
- Thanushi Peiris
- Thomas Voigtmann
- Tim-Oliver Buchholz
- Tyler Reddy

42 reviewers contributed to this release [alphabetical by first name or login]

- Abhijeet Parida
- Albert Y. Shih
- Alex (sashashura)
- Alexandre de Siqueira
- Antony Lee
- Ben Greiner
- Carlo Dri
- Chris Roat
- Daniele Nicolodi
- Daria
- Dudu Lasry
- Eli Schwartz
- François Boulogne
- Gregory Lee
- Gus Becker
- Jacob Rosenthal
- James Gao
- Jan-Hendrik Müller
- Jarrod Millman
- Juan DF
- Juan Nunez-Iglesias
- Lars Grüter
- Malinda (maldil)
- Marianne Corvellec
- Mark Harfouche
- Martijn Courteaux
- Marvin Albert

- Matthias Bussonnier
- Oren Amsalem
- Ralf Gommers
- Riadh Fezzani
- Robert Haase
- Robin Thibaut
- Sandeep N Menon
- Sanghyeok Hyun
- Sebastian Berg
- Sebastian Wallkötter
- Simon-Martin Schröder
- Stefan van der Walt
- Thanushi Peiris
- Thomas Voigtmann
- Tim-Oliver Buchholz

1.4.11 scikit-image 0.19.3 release notes

We're happy to announce the release of scikit-image v0.19.3!

scikit-image is an image processing toolbox for SciPy that includes algorithms for segmentation, geometric transformations, color space manipulation, analysis, filtering, morphology, feature detection, and more.

For more information, examples, and documentation, please visit our website:

<https://scikit-image.org>

Bugs Fixed

- Revert unintentional change to default multichannel behavior introduced in v0.19.0 for `skimage.restoration.cycle_spin` (now defaults to single channel again)
- Fix corner case with an optimal angle of 0 degrees in `hough_line_peaks`
- Fixed the gallery example involving registration with log-polar transformations
- Update test suite for compatibility with the most recent `tifffile` release.
- `warp/rotate`: fixed a bug with clipping when `cval` is not in the input range
- Fix computation of histogram bins for multichannel integer-valued images

General Maintenance

- Update `skimage.future.manual_polygon_segmentation` to work with Matplotlib 3.5.
- Update `skimage.io.imread` to avoid warnings when using `imageio>=2.16.2`.
- Now compatible with Pillow ≥ 9.1 (palette may contain <256 entries)
- Added support for NumPy 1.23

Pull Requests Included

- Backport PR #6306 on branch v0.19.x (Fix for error in ‘Using Polar and Log-Polar Transformations for Registration’) (#6312)
- Backport PR #6271 on branch v0.19.x (hough_line_peaks fix for corner case with optimal angle=0) (#6313)
- Backport PR #6261 on branch v0.19.x (Ignore sparse matrix deprecation warning) (#6316)
- backport PR 6328: Fix issue with newer versions of matplotlib in manual segmentation (#6334)
- Backport PR #6343 on branch v0.19.x (avoid warnings about change to v3 API from imageio) (#6344)
- Backport PR #6355 on branch v0.19.x (remove use of deprecated kwargs from `test_tifffile_kwarg_passthrough`) (#6357)
- Backport PR #6352 on branch v0.19.x (Fix channel_axis default for cycle_spin) (#6358)
- Backport PR #6348 on branch v0.19.x (Fix smoothed image computation when mask is None in canny) (#6359)
- Backport PR #6361 on branch v0.19.x (Document support for Path objects in io functions) (#6363)
- Backport PR #6400 on branch v0.19.x (Add support for NumPy 1.23) (#6403)
- Backport PR #6335 on branch v0.19.x (warp/rotate: fixed a bug with clipping when eval is not in the input range) (#6411)
- Backport PR #6413 on branch v0.19.x (Fix computation of histogram bins for multichannel integer-valued images) (#6414)

10 authors added to this release [alphabetical by first name or login]

- Albert Y. Shih
- Bartłomiej Śmiertanka
- Dave Mellert
- Gregory Lee
- Graham Inggs
- Jarrod Millman
- John Hagen
- Mark Harfouche
- Riadh Fezzani
- Stefan van der Walt

7 reviewers added to this release [alphabetical by first name or login]

- Alexandre de Siqueira
- Gregory Lee
- Jarrod Millman
- Juan Nunez-Iglesias
- Lars Grüter
- Mark Harfouche
- Riadh Fezzani

1.4.12 scikit-image 0.19.2 release notes

We're happy to announce the release of scikit-image v0.19.2! This is primarily a bug fix release, although there is one new gallery example related to detection of fluorescence at the nuclear envelope of mammalian cells.

scikit-image is an image processing toolbox for SciPy that includes algorithms for segmentation, geometric transformations, color space manipulation, analysis, filtering, morphology, feature detection, and more.

For more information, examples, and documentation, please visit our website:

<https://scikit-image.org>

Pull Requests Included

- fix mistake in tests.yml made during backport (gh-6129)
- Backport PR #6145 on branch v0.19.x (Fix channel_axis handling in pyramid_gaussian and pyramid_laplace) (gh-6155)
- Backport PR #6130 on branch v0.19.x (bump deprecated Azure windows environment) (gh-6131)
- Backport PR #6148 on branch v0.19.x (deprecate n_iter_max (should be max_num_iter)) (gh-6156)
- Backport PR #6152 on branch v0.19.x (specify python version used by mybinder.org for gallery demos) (gh-6157)
- Backport PR #6139 on branch v0.19.x (fix phase_cross_correlation typo) (gh-6158)
- Backport PR #6133 on branch v0.19.x (Update user warning message for viewer module.) (gh-6159)
- Backport PR #6169 on branch v0.19.x (Fix unintended change to output dtype of match_histograms) (gh-6172)
- Backport PR #6184 on branch v0.19.x (Fix SIFT wrong octave indices + typo) (gh-6186)
- Backport PR #6191 on branch v0.19.x (Fix issue6190 - inconsistent default parameters in pyramids.py) (gh-6193)
- Backport PR #6207 on branch v0.19.x (Always set params to nan when ProjectiveTransform.estimate fails) (gh-6210)
- Backport PR #5262 on branch v0.19.x (Add textbook-like tutorial on measuring fluorescence at nuclear envelope.) (gh-6213)
- Backport PR #6087 on branch v0.19.x (Add two datasets for use in upcoming scientific tutorials.) (gh-6215)
- Backport PR #6214 on branch v0.19.x (EuclideanTransform.estimate should return False when NaNs are present) (gh-6221)

- Backport PR #6219 on branch v0.19.x (Allow the output_shape argument to be any iterable for resize and resize_local_mean) (gh-6222)
- Backport PR #6223 on branch v0.19.x (Update filename in testing instructions.) (gh-6225)
- Backport PR #6231 on branch v0.19.x (Update imports/refs from deprecated scipy.ndimage.filters namespace) (gh-6233)
- Backport PR #6229 on branch v0.19.x (Remove redundant testing on Appveyor) (gh-6234)
- Backport PR #6183 on branch v0.19.x (Fix decorators warnings stacklevel) (gh-6238)
- Backport PR #6239 on branch v0.19.x (DOC: fix SciPy intersphinx) (gh-6241)
- Backport PR #6232 on branch v0.19.x (Include Cython sources via package_data) (gh-6244)
- Backport PR #6227 on branch v0.19.x (Fix calculation of Z normal in marching cubes) (gh-6245)
- Backport PR #6242 on branch v0.19.x (Fix bug in SLIC superpixels with `enforce_connectivity=True` and `start_label > 0`) (gh-6246)
- Backport PR #6211 on branch v0.19.x (PiecewiseAffineTransform.estimate return should reflect underlying transforms) gh-6247
- update MacOS libomp installation in wheel building script (gh-6249)

9 authors added to this release [alphabetical by first name or login]

- Chris Roat
- Fabian Schneider
- Gregory Lee
- Hande Gözükhan
- Larry Bradley
- Marianne Corvellec
- Mark Harfouche
- Miles Lucas
- Riadh Fezzani

8 reviewers added to this release [alphabetical by first name or login]

- Alexandre de Siqueira
- Gregory Lee
- Juan Nunez-Iglesias
- Marianne Corvellec
- Mark Harfouche
- Riadh Fezzani
- Robert Haase
- Stefan van der Walt

1.4.13 scikit-image 0.19.1 release notes

We're happy to announce the release of scikit-image v0.19.1!

This is a small bug fix release that resolves a couple of backwards compatibility issues and a couple of issues with the wheels on PyPI. Specifically, MacOs wheels for Apple M1 (arm64) on PyPI were broken in 0.19.0, but should now be repaired. The arm64 wheels are for MacOs ≥ 12 only. Wheel sizes are also greatly reduced relative to 0.19.0 by stripping debug symbols from the binaries and making sure that Cython-generated source files are not bundled in the wheels.

Pull Requests Included

- Backport PR #6089 on branch v0.19.x (Skip tests requiring fetched data) (gh-6115)
- Backport PR #6097 on branch v0.19.x (restore non-underscore functions in skimage.data) (gh-6099)
- Backport PR #6095 on branch v0.19.x (Preserve backwards compatibility for *channel_axis* parameter in transform functions) (gh-6100)
- Backport PR #6103 on branch v0.19.x (make rank filter test comparisons robust across architectures) (gh-6106)
- Backport PR #6105 on branch v0.19.x (pass a specific random_state into ransac in test_ransac_geometric) (gh-6107)
- Fix two equality comparison bugs in the wheel build script (gh-6098)
- Backport of gh-6109 (Add linker flags to strip debug symbols during wheel building) (gh-6110)
- Pin setuptools maximum in v0.19.x to avoid breaking on planned distutils API changes (gh-6112)
- Avoid potential circular import of rgb2gray (gh-6113)
- Backport PR #6089 on branch v0.19.x (Skip tests requiring fetched data) (gh-6115)
- Backport PR #6118 on branch v0.19.x (Fixes to tests.yml and fixes for expected warnings) (gh-6127)
- Backport PR #6114 on branch v0.19.x (relax test condition to make it more robust to variable CI load) (gh-6128)

3 authors added to this release [alphabetical by first name or login]

- Gregory R. Lee
- Joshua Newton
- Mark Harfouche

5 reviewers added to this release [alphabetical by first name or login]

- Gregory R. Lee
- Juan Nunez-Iglesias
- Marianne Corvellec
- Mark Harfouche
- Stefan van der Walt

1.4.14 scikit-image 0.19.0 release notes

We're happy to announce the release of scikit-image v0.19.0!

scikit-image is an image processing toolbox for SciPy that includes algorithms for segmentation, geometric transformations, color space manipulation, analysis, filtering, morphology, feature detection, and more.

For more information, examples, and documentation, please visit our website:

<https://scikit-image.org>

A highlight of this release is the addition of the popular scale-invariant feature transform (SIFT) feature detector and descriptor. This release also introduces a perceptual blur metric, new pixel graph algorithms, and most functions now operate in single-precision when single-precision inputs are provided. Many other bug fixes, enhancements and performance improvements are detailed below.

A significant change in this release is in the treatment of multichannel images. The existing `multichannel` argument to functions has been deprecated in favor of a new `channel_axis` argument. `channel_axis` can be used to specify which axis of an array contains channel information (with `channel_axis=None` indicating a grayscale image).

scikit-image now uses “lazy loading”, which enables users to access the functions from all `skimage` submodules without the overhead of eagerly importing all submodules. As a concrete example, after calling “import `skimage`” a user can directly call a function such as `skimage.transform.warp` whereas previously it would have been required to first “import `skimage.transform`”.

An exciting change on the development side is the introduction of support for Pythran as an alternative to Cython for generating compiled code. We plan to keep Cython support as well going forward, so developers are free to use either one as appropriate. For those curious about Pythran, a good overview was given in the SciPy 2021 presentation, “Building SciPy Kernels with Pythran” (<https://www.youtube.com/watch?v=6a9D9WL6ZjQ>).

This release now supports Python 3.7-3.10. Apple M1 architecture (arm64) support is new to this release. MacOS 12 wheels are provided for Python 3.8-3.10.

New Features

- Added support for processing images with channels located along any array axis. This is in contrast to previous releases where channels were required to be the last axis of an image. See more info on the new `channel_axis` argument under the API section of the release notes.
- A no-reference measure of perceptual blur was added (`skimage.measure.blur_effect`).
- Non-local means (`skimage.restoration.denoise_nl_means`) now supports 3D multichannel, 4D and 4D multichannel data when `fast_mode=True`.
- An n-dimensional Fourier-domain Butterworth filter (`skimage.filters.butterworth`) was added.
- Color conversion functions now have a new `channel_axis` keyword argument that allows specification of which axis of an array corresponds to channels. For backwards compatibility, this parameter defaults to `channel_axis=-1`, indicating that channels are along the last axis.
- Added a new keyword only parameter `random_state` to `morphology.medial_axis` and `restoration.unsupervised_wiener`.
- Seeding random number generators will not give the same results as the underlying generator was updated to use `numpy.random.Generator`.
- Added `saturation` parameter to `skimage.color.label2rgb`
- Added normalized mutual information metric `skimage.metrics.normalized_mutual_information`
- `threshold_local` now supports n-dimensional inputs and anisotropic `block_size`

- New `skimage.util.label_points` function for assigning labels to points.
- Added nD support to several geometric transform classes
- Added `skimage.metrics.hausdorff_pair` to find points separated by the Hausdorff distance.
- Additional colorspace illuminants and observers parameter options were added to `skimage.color.lab2rgb`, `skimage.color.rgb2lab`, `skimage.color.xyz2lab`, `skimage.color.lab2xyz`, `skimage.color.xyz2luv` and `skimage.color.luv2xyz`.
- `skimage.filters.threshold_multiotsu` has a new `hist` keyword argument to allow use with a user-supplied histogram. (gh-5543)
- `skimage.restoration.denoise_bilateral` added support for images containing negative values. (gh-5527)
- The `skimage.feature` functions `blob_dog`, `blob_doh` and `blob_log` now support a `threshold_rel` keyword argument that can be used to specify a relative threshold (in range [0, 1]) rather than an absolute one. (gh-5517)
- Implement lazy submodule importing (gh-5101)
- Implement weighted estimation of geometric transform matrices (gh-5601)
- Added new pixel graph algorithms in `skimage.graph`: `pixel_graph` generates a graph (network) of pixels according to their adjacency, and `central_pixel` finds the geodesic center of the pixels. (gh-5602)
- scikit-image now supports use of Pythran in contributed code. (gh-3226)

Documentation

- A new doc tutorial presenting a 3D biomedical imaging example has been added to the gallery (gh-4946). The technical content benefited from conversations with Genevieve Buckley, Kevin Mader, and Volker Hilsenstein.
- New gallery example for 3D structure tensor.
- New gallery example displaying a 3D dataset.
- Extended rolling ball example with ECG data (1D).
- The stain unmixing gallery example was fixed and now displays proper separation of the stains.
- Documentation has been added to the contributing notes about how to submit a gallery example.
- Autoformat docstrings in morphology.
- Display plotly figures from gallery example even when running script at CLI.
- Single out docs-only PRs in review process.
- Use matplotlib's infinite axline to demonstrate hough transform.
- Clarify disk documentation inconsistency regarding 'shape'.
- docs: fix simple typo, conversions -> conversions.
- Fixes to linspace in example.
- Minor fixes to Hough line transform code and examples.
- Added 1/2 pixel bounds to extent of displayed images in several examples.
- Add release step on github to RELEASE.txt.
- Remove reference to opencv in threshold_local documentation.
- Update structure_tensor docstring to include per-axis sigma.

- Fix typo in _shared/utils.py docs.
- Proofread and crosslink examples with immunohistochemistry image.
- Spelling correction: witch -> which.
- Mention possible filters in radon_transform -> filtered-back-projection
- Fix dtype info in documentation for watershed.
- Proofread gallery example for Radon transform.
- Use internal function for noise + clarify code in Canny example.
- Make more comprehensive ‘see also’ sections in filters.
- Specify the release note version instead of the misleading *latest*.
- Remove misleading comment in plot_thresholding.py example.
- Fix sphinx layout to make the search engine work with recent sphinx versions.
- Draw node IDs in RAG example.
- Update sigma_color description in denoise_bilateral.
- Update intersphinx fallback inventories + add matplotlib fallback inventory.
- Fix numpy deprecation in plot_local_equalize.py.
- Rename label variable in plot_regionprops.py to circumvent link issue in docs.
- Avoid duplicate API documentation for ImageViewer, CollectionViewer.
- Fix ‘blog_dog’ typo in gaussian docs.
- Update reference link documentation in the adjust_sigmoid function.
- Fix reference to multiscale_basic_features in TrainableSegmenter.
- Slight shape_index docstring modification to specify 2D array.
- Add stitching gallery example (gh-5365)
- Add draft SKIP3: transition to scikit-image 1.0 (gh-5475)
- Mention commit messages in the contribution guidelines. (gh-5504)
- Fix and standardize docstrings for blob detection functions. (gh-5547)
- Update the User Guide to reflect usage of channel_axis rather than multichannel. (gh-5554)
- Update the user guide to use channel_axis rather than multichannel (gh-5556)
- Add hyperlinks to referenced documentation places. (gh-5560)
- Update branching instructions to change the location of the pooch repo. (gh-5565)
- Add Notes and References section to the Cascade class docstring. (gh-5568)
- Clarify 2D vs nD in skimage.feature.corner docstrings (gh-5569)
- Fix math formulas in plot_swirl.py example. (gh-5574)
- Update references in texture feature detectors docstrings (gh-5578)
- Update mailing list location to discuss.scientific-python.org forum (gh-5951)
- DOC: Fix docstring in rescale_intensity() (gh-5964)
- Fix slic documentation (gh-5975)

- Update docstring for dilation, which is now nD. (gh-5978)
- Change stitching gallery example thumbnail (gh-5985)
- Add circle and disk to glossary.md (gh-5590)
- Update pixel graphs example (gh-5991)
- Separate entries that have the same description in glossary.md (gh-5592)
- Do not use space before colon in directive name (gh-6002)

Improvements

- Many more functions throughout the library now have single precision (float32) support.
- Biharmonic inpainting (`skimage.restoration.inpaint_biharmonic`) was refactored and is orders of magnitude faster than before.
- Salt-and-pepper noise generation with `skimage.util.random_noise` is now faster.
- The performance of the SLIC superpixels algorithm (`skimage.segmentation.slice`) was improved for the case where a mask is supplied by the user (gh-4903). The specific superpixels produced by masked SLIC will not be identical to those produced by prior releases.
- `exposure.adjust_gamma` has been accelerated for `uint8` images thanks to a LUT (gh-4966).
- `measure.label` has been accelerated for boolean input images, by using `scipy.ndimage`'s implementation for this case (gh-4945).
- `util.apply_parallel` now works with multichannel data (gh-4927).
- `skimage.feature.peak_local_max` supports now any Minkowski distance.
- Fast, non-Cython implementation for `skimage.filters.correlate_sparse`.
- For efficiency, the histogram is now precomputed within `skimage.filters.try_all_threshold`.
- Faster `skimage.filters.find_local_max` when given a finite `num_peaks`.
- All filters in the `skimage.filters.rank` module now release the GIL, enabling multithreaded use.
- `skimage.restoration.denoise_tv_bregman` and `skimage.restoration.denoise_bilateral` now release the GIL, enabling multithreaded use.
- A `skimage.color.label2rgb` performance regression was addressed.
- Improve numerical precision in `CircleModel.estimate`. (gh-5190)
- Add default keyword argument values to `skimage.restoration.denoise_tv_bregman`, `skimage.measure.block_reduce`, and `skimage.filters.threshold_local`. (gh-5454)
- Make matplotlib an optional dependency (gh-5990)
- single precision support in `skimage.filters` (gh-5354)
- Support nD images and labels in `label2rgb` (gh-5550)
- Regionprops table performance refactor (gh-5576)
- add regionprops benchmark script (gh-5579)
- remove use of `apply_along_axes` from `greycomatrix` & `greycoprops` (gh-5580)
- refactor gabor_kernel for efficiency (gh-5582)
- remove need for `channel_as_last_axis` decorator in `skimage.filters` (gh-5584)

- replace use of `scipy.ndimage.gaussian_filter` with `skimage.filters.gaussian` (gh-5872)
- add `channel_axis` argument to `quickshift` (gh-5987)
- add MacOS arm64 wheels (gh-6068)

API Changes

- The `multichannel` boolean argument has been deprecated. All functions with multichannel support now use an integer `channel_axis` to specify which axis corresponds to channels. Setting `channel_axis` to `None` is used to indicate that the image is grayscale. Specifically, existing code with `multichannel=True` should be updated to use `channel_axis=-1` and code with `multichannel=False` should now specify `channel_axis=None`.
- Most functions now return `float32` images when the input has `float32` dtype.
- A default value has been added to `measure.find_contours`, corresponding to the half distance between the min and max values of the image (gh-4862).
- `data.cat` has been introduced as an alias of `data.chelsea` for a more descriptive name.
- The `level` parameter of `measure.find_contours` is now a keyword argument, with a default value set to `(max(image) - min(image)) / 2`.
- `p_norm` argument was added to `skimage.feature.peak_local_max` to add support for Minkowski distances.
- `skimage.transforms.integral_image` now promotes floating point inputs to double precision by default (for accuracy). A new `dtype` keyword argument can be used to override this behavior when desired.
- Color conversion functions now have a new `channel_axis` keyword argument (see **New Features** section).
- SLIC superpixel segmentation outputs may differ from previous versions for data that was not already scaled to $[0, 1]$ range. There is now an automatic internal rescaling of the input to $[0, 1]$ so that the `compactness` parameter has an effect that is independent of the input image's scaling.
- A bug fix to the phase normalization applied within `skimage.register.phase_cross_correlation` may result in a different result as compared to prior releases. The prior behavior of “unnormalized” cross correlation is still available by explicitly setting `normalization=None`. There is no change to the masked cross-correlation case, which uses a different algorithm.

Bugfixes

- Input `labels` argument renumbering in `skimage.feature.peak_local_max` is avoided (gh-5047).
- fix clip bug in resize when anti_aliasing is applied (gh-5202)
- Nonzero values at the image edge are no longer incorrectly marked as a boundary when using `find_boundaries` with mode='subpixel' (gh-5447).
- Fix return dtype of `_label2rgb_avg` function.
- Ensure `skimage.color.separate_stains` does not return negative values.
- Prevent integer overflow in `EllipseModel`.
- Fixed off-by one error in pixel bins in Hough line transform, `skimage.transform.hough_line`.
- Handle 1D arrays properly in `skimage.filters.gaussian`.
- Fix Laplacian matrix size bug in `skimage.segmentation.random_walker`.
- Regionprops table (`skimage.measure.regionprops_table`) dtype bugfix.
- Fix `skimage.transform.rescale` when using a small scale factor.

- Fix `skimage.measure.label` segfault.
- Watershed (`skimage.segmentation.watershed`): consider connectivity when calculating markers.
- Fix `skimage.transform.warp` output dtype when `order=0`.
- Fix multichannel `intensity_image` extra_properties in `regionprops`.
- Fix error message for `skimage.metric.structural_similarity` when image is too small.
- Do not mark image edges in ‘subpixel’ mode of `skimage.segmentation.find_boundaries`.
- Fix behavior of `skimage.exposure.is_low_contrast` for boolean inputs.
- Fix wrong syntax for the string argument of `ValueError` in `skimage.metric.structural_similarity`.
- Fixed NaN issue in `skimage.filters.threshold_otsu`.
- Fix `skimage.feature.blob_dog` docstring example and normalization.
- Fix uint8 overflow in `skimage.exposure.adjust_gamma`.
- Work with pooch 1.5.0 for fetching data (gh-5529).
- The `offsets` attribute of `skimage.graph.MCP` is now public. (gh-5547)
- Fix `io.imread` behavior with `pathlib.Path` inputs (gh-5543)
- Make scikit-image imports from Pooch, compatible with `pooch >= 1.5.0`. (gh-5529)
- Fix several broken doctests and restore doctesting on GitHub Actions. (gh-5505)
- Fix broken doctests in `skimage.exposure.histogram` and `skimage.measure.regionprops_table`. (gh-5522)
- Rescale image consistently during SLIC superpixel segmentation. (gh-5518)
- Correct phase correlation in `skimage.register.phase_cross_correlation`. (gh-5461)
- Fix hidden attribute ‘`offsets`’ in `skimage.graph.MCP` (gh-5551)
- fix `phase_cross_correlation` for 3D with reference masks (gh-5559)
- fix return shape of `blob_log` and `blob_dog` when no peaks are found (gh-5567)
- Fix `find contours` key error (gh-5577)
- Refactor `measure.ransac` and add warning when the estimated model is not valid (gh-5583)
- Restore integer image rescaling for edge filters (gh-5589)
- `trainable_segmentation`: re-raise in error case (gh-5600)
- allow `regionprops_table` to be called with deprecated property names (gh-5908)
- Fix weight calculation in fast mode of non-local means (gh-5923)
- fix for #5948: lower boundary 1 for `kernel_size` in `equalize_adapthist` (gh-5949)
- convert `pathlib.Path` to `str` in `imsave` (gh-5971)
- Fix slic spacing (gh-5974)
- Add small regularization to avoid zero-division in `richardson_lucy` (gh-5976)
- Fix benchmark suite (watershed function was moved) (gh-5982)
- catch `QhullError` and return empty array (`convex_hull`) (gh-6008)
- add property getters for all newly deprecated `regionprops` names (gh-6000)

- Fix the estimation of ellipsoid axis lengths in the 3D case (gh-6013)
- Fix peak local max segfault (gh-6035)
- Avoid circular import errors when EAGER_IMPORT=1 (gh-6042)
- remove all use of the deprecated distutils package (gh-6044)

Deprecations

Completed deprecations from prior releases

- In `measure.label`, the deprecated `neighbors` parameter has been removed (use `connectivity` instead).
- The deprecated `skimage.color.rgb2grey` and `skimage.color.grey2rgb` functions have been removed (use `skimage.color.rgb2gray` and `skimage.color.gray2rgb` instead).
- `skimage.color.rgb2gray` no longer allows grayscale or RGBA inputs.
- The deprecated `alpha` parameter of `skimage.color.gray2rgb` has now been removed. Use `skimage.color.gray2rgba` for conversion to RGBA.
- Attempting to warp a boolean image with `order > 0` now raises a `ValueError`.
- When warping or rescaling boolean images, setting `anti-aliasing=True` will raise a `ValueError`.
- The `bg_label` parameter of `skimage.color.label2rgb` is now 0.
- The deprecated `filter` parameter of `skimage.transform.iradon` has now been removed (use `filter_name` instead).
- The deprecated `skimage.draw.circle` function has been removed (use `skimage.draw.disk` instead).
- The deprecated `skimage.feature.register_translation` function has been removed (use `skimage.registration.phase_cross_correlation` instead).
- The deprecated `skimage.feature.masked_register_translation` function has been removed (use `skimage.registration.phase_cross_correlation` instead).
- The deprecated `skimage.measure.marching_cubes_classic` function has been removed (use `skimage.measure.marching_cubes` instead).
- The deprecated `skimage.measure.marching_cubes_lewiner` function has been removed (use `skimage.measure.marching_cubes` instead).
- The deprecated `skimage.segmentation.circle_level_set` function has been removed (use `skimage.segmentation.disk_level_set` instead).
- The deprecated `inplace` parameter of `skimage.morphology.flood_fill`
- The deprecated `skimage.util.pad` function has been removed (use `numpy.pad` instead). been removed (use `in_place` instead).
- The default mode in `skimage.filters.hessian` is now '`reflect`'.
- The default boundary mode in `skimage.filters.sato` is now '`reflect`'.
- The default boundary mode in `skimage.measure.profile_line` is now '`reflect`'.
- The default value of `preserve_range` in `skimage.restoration.denoise_nl_means` is now `False`.
- The default value of `start_label` in `skimage.segmentation.slic` is now 1.

Newly introduced deprecations:

- The `multichannel` argument is now deprecated throughout the library and will be removed in 1.0. The new `channel_axis` argument should be used instead. Existing code with `multichannel=True` should be updated to use `channel_axis=-1` and code with `multichannel=False` should now specify `channel_axis=None`.
- `skimage.feature.greycomatrix` and `skimage.feature.greycoprops` are deprecated in favor of `skimage.feature.graycomatrix` and `skimage.feature.grycoprops`.
- The `skimage.morphology.grey` module has been renamed `skimage.morphology.gray`. The old name is deprecated.
- The `skimage.morphology.greyreconstruct` module has been renamed `skimage.morphology.grayreconstruct`. The old name is deprecated.
- see **API Changes** section regarding functions with deprecated argument names related to the number of iterations. `num_iterations` and `max_num_iter` are now used throughout the library.
- see **API Changes** section on deprecation of the `selem` argument in favor of `footprint` throughout the library
- Deprecate `in_place` in favor of the use of an explicit `out` argument in `skimage.morphology.remove_small_objects`, `skimage.morphology.remove_small_holes` and `skimage.segmentation.clear_border`
- The input argument of `skimage.measure.label` has been renamed `label_image`. The old name is deprecated.
- standardize on `num_iter` for parameters describing the number of iterations and `max_num_iter` for parameters specifying an iteration limit. Functions where the old argument names have now been deprecated are:

```
skimage.filters.threshold_minimum
skimage.morphology.thin
skimage.restoration.denoise_tv_bregman
skimage.restoration.richardson_lucy
skimage.segmentation.active_contour
skimage.segmentation.chan_vese
skimage.segmentation.morphological_chan_vese
skimage.segmentation.morphological_geodesic_active_contour
skimage.segmentation.slic
```

- The names of several parameters in `skimage.measure.regionprops` have been updated so that properties are better grouped by the first word(s) of the name. The old names will continue to work for backwards compatibility. The specific names that were updated are:

Old Name	New Name
<code>max_intensity</code>	<code>intensity_max</code>
<code>mean_intensity</code>	<code>intensity_mean</code>
<code>min_intensity</code>	<code>intensity_min</code>
<code>bbox_area</code>	<code>area_bbox</code>
<code>convex_area</code>	<code>area_convex</code>
<code>filled_area</code>	<code>area_filled</code>
<code>convex_image</code>	<code>image_convex</code>
<code>filled_image</code>	<code>image_filled</code>
<code>intensity_image</code>	<code>image_intensity</code>

(continues on next page)

(continued from previous page)

local_centroid	centroid_local
weighted_centroid	centroid_weighted
weighted_local_centroid	centroid_weighted_local
major_axis_length	axis_major_length
minor_axis_length	axis_minor_length
weighted_moments	moments_weighted
weighted_moments_central	moments_weighted_central
weighted_moments_hu	moments_weighted_hu
weighted_moments_normalized	moments_weighted_normalized
equivalent_diameter	equivalent_diameter_area

- The `selem` argument has been renamed to `footprint` throughout the library. The `selem` argument is now deprecated.

Development process

- Test setup and teardown functions added to allow raising an error on any uncaught warnings via `SKIMAGE_TEST_STRICT_WARNINGS_GLOBAL` environment variable.
- Increase automation in release process.
- Release wheels before source
- update minimum supported Matplotlib, NumPy, SciPy and Pillow
- Pin pillow to !=8.3.0
- Rename *master* to *main* throughout
- Ensure that `README.txt` has write permissions for subsequent imports.
- Run face classification gallery example with a single thread
- Enable pip and skimage.data caching on Azure
- Fix CircleCI and Azure CI caching.
- Address Cython warnings.
- Disable calls to `plotly.io.show` when running on Azure.
- Remove legacy Travis-CI scripts and update contributor documentation accordingly.
- Increase cibuildwheel verbosity.
- Update pip during dev environment installation.
- Add benchmark checks to CI.
- Resolve stochastic rank filter test failures on CI.
- Ensure that `README.txt` has write permissions for subsequent imports.
- Decorators for helping with the transition between the keyword argument `multichannel` and `channel_axis`.
- Add missing import in `lch2lab` docstring example (gh-5998)

- Prefer importing build_py and sdist from setuptools (gh-6007)
- Reintroduce skimage.test utility (gh-5909)

Other Updates

- Refactor np.random.x to use np.random.Generator.
- Avoid warnings about use of deprecated `scipy.linalg.pinv2`.
- Simplify resize implementation using new SciPy 1.6 zoom option.
- Fix duplicate test function names in `test_unsharp_mask.py`.
- Benchmarks: fix `ResizeLocalMeanSuite.time_resize_local_mean` signature.
- Prefer use of new-style NumPy random API in tests (gh-5450)
- Add fixture enforcing SimpleITK I/O in `test_simpleitk.py` (gh-5526)
- MNT: Remove unused stat import from skimage data (gh-5566)
- MAINT: Remove unused imports (gh-5595)
- MAINT: Refactor duplicated tests, remove unnecessary assignments and variables (gh-5596)
- Remove obsolete lazy import (gh-5992)
- Lazily load `data_dir` into the top-level namespace (gh-5996)
- Update scipy requirement to 1.4.1 and use `scipy.fft` instead of `scipy.fftpack` (gh-5999)
- Remove lines generating Requires metadata (gh-6017)
- Update wheel builds to include Python 3.10 (gh-6021)
- Update `pyproject.toml` to handle Python 3.10 and Apple arm64 (gh-6022)
- Add python 3.10 test runs on GitHub Actions and Appveyor (gh-6027)
- Pin sphinx to <4.3 until new sphinx-gallery release is available (gh-6029)
- Relax a couple of equality tests causing i686 test failures on cibuildwheel (gh-6031)
- Avoid matplotlib import overhead during ‘import skimage’ (gh-6032)
- Update sphinx gallery pin (gh-6034)

Contributors to this release

80 authors added to this release [alphabetical by first name or login]

- Abhinavmishra8960 (Abhinavmishra8960)
- abouyssو
- Alessia Marcolini
- Alex Brooks
- Alexandre de Siqueira
- Andres Fernandez
- Andrew Hurlbatt

- andrewnags (andrewnags)
- Antoine Bierret
- BMaster123 (BMaster123)
- Boaz Mohar
- Bozhidar Karaargirov
- Carlos Andrés Álvarez Restrepo
- Christoph Gohlke
- Christoph Sommer
- Clement Ng
- cmarasinou
- Cris Luengo
- David Manthey
- Devanshu Shah
- Dhiraj Kumar Sah
- divyank agarwal
- Egor Panfilov
- Emmanuelle Gouillart
- Erik Reed
- erykoff (erykoff)
- Fabian Schneider
- Felipe Gutierrez-Barragan
- François Boulogne
- Fred Bunt
- Fukai Yohsuke
- Gregory R. Lee
- Hari Prasad
- Harish Venkataraman
- Harshit Dixit
- Ian Hunt-Isaak
- Jaime Rodríguez-Guerra
- Jan-Hendrik Müller
- Janakarajan Natarajan
- Jenny Vo
- john lee
- Jonathan Striebel
- Joseph Fox-Rabinovitz

- Juan Antonio Barragan Noguera
- Juan Nunez-Iglesias
- Julien Jerphanion
- Jurneo
- klaussfreire (klaussfreire)
- Larkinnjm1 (Larkinnjm1)
- Lars Grüter
- Mads Dyrmann
- Marianne Corvellec
- Marios Achilias
- Mark Boer
- Mark Harfouche
- Matthias Bussonnier
- Mauro Silberberg
- Max Frei
- michalkrawczyk (michalkrawczyk)
- Niels Cautaerts
- Pamphile ROY
- Pradyumna Rahul
- R
- Raphael
- Riadh Fezzani
- Robert Haase
- Sebastian Gonzalez Tirado
- Sebastián Vanrell
- serge-sans-paille (serge-sans-paille)
- Stefan van der Walt
- t.ae
- that1solodev (Xyno18)
- Thomas Walter
- Tim Gates
- Tom Flux
- Vinicius D. Cerutti
- Volker Hilsenstein
- WeiChungChang
- yacht

- Yash-10 (Yash-10)

63 reviewers added to this release [alphabetical by first name or login]

- Abhinavmishra8960
- Alessia Marcolini
- Alex Brooks
- Alexandre de Siqueira
- Andres Fernandez
- Andrew Hurlbatt
- andrewnags
- BMaster123
- Boaz Mohar
- Carlos Andrés Álvarez Restrepo
- Clement Ng
- Cris Luengo
- Dan Schult
- David Manthey
- Egor Panfilov
- Emmanuelle Gouillart
- erykoff
- Fabian Schneider
- Felipe Gutierrez-Barragan
- François Boulogne
- Fukai Yohsuke
- Genevieve Buckley
- Gregory R. Lee
- Jan Eglinger
- Jan-Hendrik Müller
- Janakarajan Natarajan
- Jarrod Millman
- Jirka Borovec
- Joan Massich
- Johannes Schönberger
- john lee
- Jon Crall
- Joseph Fox-Rabinovitz

- Josh Warner
- Juan Nunez-Iglesias
- Julien Jerphanion
- Kenneth Hoste
- klaussfreire
- Larkinnjm1
- Lars Grüter
- Marianne Corvellec
- Mark Boer
- Mark Harfouche
- Matthias Bussonnier
- Max Frei
- michalkrawczyk
- Niels Cautaerts
- Pamphile ROY
- Pomax
- R
- Raphael
- Riadh Fezzani
- Robert Kern
- Ross Barnowski
- Sebastian Berg
- Sebastian Gonzalez Tirado
- Sebastian Wallkötter
- serge-sans-paille
- Stefan van der Walt
- t.ae
- Vinicius D. Cerutti
- Volker Hilsenstein
- Yash-10

1.4.15 scikit-image 0.18.3 release notes

We're happy to announce the release of scikit-image v0.18.3!

scikit-image is an image processing toolbox for SciPy that includes algorithms for segmentation, geometric transformations, color space manipulation, analysis, filtering, morphology, feature detection, and more.

This is a small bugfix release for compatibility with Pooch 1.5 and SciPy 1.7.

Bug fixes

- Only import from Pooch's public API. This resolves an import failure with Pooch 1.5.0. (#5531, backport of #5529)
- Do not use deprecated `scipy.linalg.pinv2` in `random_walker` when using the multigrid solver. (#5531, backport of #5437)

3 authors added to this release [alphabetical by first name or login]

David Manthey Gregory Lee Mark Harfouche

3 reviewers added to this release [alphabetical by first name or login]

Gregory Lee Juan Nunez-Iglesias Mark Harfouche

1.4.16 scikit-image 0.18.2 release notes

We're happy to announce the release of scikit-image v0.18.2!

scikit-image is an image processing toolbox for SciPy that includes algorithms for segmentation, geometric transformations, color space manipulation, analysis, filtering, morphology, feature detection, and more.

This release mostly serves to add wheels for the aarch64 architecture; it also fixes a couple of minor bugs.

For more information, examples, and documentation, please visit our website:

<https://scikit-image.org>

Bug fixes

- allow either SyntaxError or OSError for truncated JPG (#5315, #5334)
- Fix sphinx: role already being registered (#5319, #5335)

Development process

- Update pyproject.toml to ensure pypy compatibility and aarch compatibility (#5326, #5328)
- Build aarch64 wheels (#5197, #5210)
- See if latest Ubuntu image fixes QEMU CPU detection issue (#5227, #5233)
- Rename *master* to *main* throughout (#5243, #5295)
- Fixup test for INSTALL_FROM_SDIST (#5283, #5296)
- Remove unnecessary manual installation of packages from before_install (#5298)
- Use manylinux2010 for python 3.9+ (#5303, #5310)
- add numpy version specification on aarch for cpython 3.8 (#5374, #5375)

7 authors added to this release [alphabetical by first name or login]

- François Boulogne
- Janakarajan Natarajan
- Juan Nunez-Iglesias
- John Lee
- Mark Harfouche
- MeeseeksMachine
- Stéfan van der Walt

9 reviewers added to this release [alphabetical by first name or login]

- Alexandre de Siqueira
- Gregory R. Lee
- Juan Nunez-Iglesias
- Marianne Corvellec
- Mark Harfouche
- Matti Picus
- Matthias Bussonnier
- Riadh Fezzani
- Stéfan van der Walt

1.4.17 scikit-image 0.18.1 release notes

This is a bug fix release and contains the following two bug fixes:

- Fix indexing error for labelling in large (>2GB) arrays (#5143, #5151)
- Only use `retry_if_failed` with recent `pooch` (#5148)

See below for the new features and API changes in 0.18.0.

1.4.18 scikit-image 0.18.0 release notes

We're happy to announce the release of scikit-image v0.18.0!

scikit-image is an image processing toolbox for SciPy that includes algorithms for segmentation, geometric transformations, color space manipulation, analysis, filtering, morphology, feature detection, and more.

This release of scikit-image drops support for Python 3.6 in accordance with the [NEP-29 Python and Numpy version support community standard](#): Python 3.7 or newer is required to run this version.

For more information, examples, and documentation, please visit our website:

<https://scikit-image.org>

New Features

- Add the iterative Lucas-Kanade (iLK) optical flow method (#4161)
- Add Feret diameter in region properties (#4379, #4820)
- Add functions to compute Euler number and Crofton perimeter estimation (#4380)
- Add a function to compute the Hausdorff distance (#4382)
- Added 3D support for many filters in `skimage.filters.rank`.
- An experimental implementation of trainable pixel segmentation, aiming for compatibility with the scikit-learn API, has been added to `skimage.future`. Try it out! (#4739)
- Add a new function `segmentation.expand_labels` to dilate labels while preventing overlap (#4795)
- It is now possible to pass extra measurement functions to `measure.regionprops` and `regionprops_table` (#4810)
- Add rolling ball algorithm for background subtraction (#4851)
- New sample images have been added in the data subpackage: `data.eagle` (#4922), `data.human_mitosis` (#4939), `data.cells3d` (#4951), and `data.vortex` (#5041). Also note that the image for `data.camera` has been changed due to copyright issues (#4913).
- `skimage.feature.structure_tensor` now supports 3D (and nD) images as input (#5002)
- Many thresholding methods can now receive a precomputed histogram as input, resulting in significant speedups if multiple methods are tried on the same image, or if a fast histogram method is used. (#5006)
- `measure.regionprops` now supports multichannel intensity images (#5037)

Documentation

- Add an example to the flood fill tutorial (#4619)
- Docstring enhancements for marching cubes and find_contours (#4641)
- A new tutorial presenting a cell biology example has been added to the gallery (#4648). Special thanks to Pierre Poulain and Fred Bernard (Université de Paris and Institut Jacques Monod) for scientific review of this example!
- Improve register rotation example with notes and references (#4723)
- Add versionadded for new scalar type support for “scale” param in `transform.AffineTransform` (#4733)
- New tutorial on [visualizing 3D data](#) (#4850)
- Add example for 3D adaptive histogram equalization (AHE) (#4658)
- Automatic formatting of docstrings for improved consistency (#4849)
- Improved docstring for `rgb2lab` (#4839) and `marching_cubes` (#4846)
- Improved docstring for `measure.marching_cubes`, mentioning how to decimate a mesh using mayavi (#4846)
- Document how to contribute a gallery example. (#4857)
- Fix and improve entropy example (#4904)
- expand the benchmarking section of the developer docs (#4905)
- Improved docstring for `util.random_noise` (#5001)
- Improved docstrings for `morphology.h_maxima` and `morphology.h_minima` (#4929).
- Improved docstring for `util.img_as_int` (#4888).
- A new example demonstrates interactive exploration of regionprops results using the PyData stack (pandas, seaborn) at <https://scikit-image.org/docs/dev/auto_examples/segmentation/plot_regionprops.html>` (#5010).
- Documentation has been added to explain [how to download example datasets](#) which are not installed with scikit-image (#4984). Similarly, the contributor guide has been updated to mention how to host new datasets in a gitlab repository (#4892).
- The [benchmarking section of the developer documentation](#) has been expanded (#4905).
- Added links to the image.sc forum in example pages (#5094, #5096)
- Added missing datasets to gallery examples (#5116, #5118)
- Add farid filters in `__all__`, to populate the documentation (#5128, #5129)
- Proofread gallery example for rank filters. (#5126, #5136)

Improvements

- float32 support for SLIC (#4683), ORB (#4684, #4697), BRIEF (#4685), `pyramid_gaussian` (#4696), Richardson-Lucy deconvolution (#4880)
- In `skimage.restoration.richardson_lucy`, computations are now done in single-precision when the input image is single-precision. This can give a substantial performance improvement when working with single precision data.
- Richardson-Lucy deconvolution now has a `filter_epsilon` keyword argument to avoid division by very small numbers (#4823)
- Add default level parameter (max-min) / 2 in `measure.find_contours` (#4862)

- The performance of the SLIC superpixels algorithm (`skimage.segmentation.slice`) was improved for the case where a mask is supplied by the user (#4903). The specific superpixels produced by masked SLIC will not be identical to those produced by prior releases.
- `exposure.adjust_gamma` has been accelerated for `uint8` images by using a look-up table (LUT) (#4966).
- `measure.label` has been accelerated for boolean input images, by using `scipy.ndimage`'s implementation for this case (#4945).
- `util.apply_parallel` now works with multichannel data (#4927).
- `skimage.feature.peak_local_max` supports now any Minkowski distance.
- We now use sparse cross-correlation to accelerate local thresholding functions (#4912)
- `morphology.convex_hull_image` now uses much less memory by checking hull inequalities in sequence (#5020)
- Polygon rasterization is more precise and will no longer potentially exclude input vertices. (#5029)
- Add data optional requirements to allow pip install scikit-image[data] (#5105, #5111)
- OpenMP support in MSVC (#4924, #5111)
- Restandardize handling of Multi-Image files (#2815, #5132)
- Consistent zoom boundary behavior across SciPy versions (#5131, #5133)

API Changes

- `skimage.restoration.richardson_lucy` returns a single-precision output when the input is single-precision. Prior to this release, double-precision was always used. (#4880)
- The default value of `threshold_rel` in `skimage.feature.corner` has changed from 0.1 to None, which corresponds to letting `skimage.feature.peak_local_max` decide on the default. This is currently equivalent to `threshold_rel=0`.
- In `measure.label`, the deprecated `neighbors` parameter has been removed. (#4942)
- The image returned by `data.camera` has changed because of copyright issues (#4913).

Bug fixes

- A bug in `label2rgb` has been fixed when the input image had `np.uint8` dtype (#4661)
- Fixed incorrect implementation of `skimage.color.separate_stains` (#4725)
- Many bug fixes have been made in `peak_local_max` (#2592, #4756, #4760, #5047)
- Fix bug in `random_walker` when input labels have negative values (#4771)
- PSF flipping is now correct for Richardson-Lucy deconvolution work in >2D (#4823)
- Fix `equalize_adapthist` (CLAHE) for clip value 1.0 (#4828)
- For the RANSAC algorithm, improved the case where all data points are outliers, which was previously raising an error (#4844)
- An error-causing bug has been corrected for the `bg_color` parameter in `label2rgb` when its value is a string (#4840)
- A normalization bug was fixed in `metrics.variation_of_information` (#4875)

- Euler characteristic property of `skimage.measure.regionprops` was erroneous for 3D objects, since it did not take tunnels into account. A new implementation based on integral geometry fixes this bug (#4380).
- In `skimage.morphology.selem.rectangle` the `height` argument controlled the width and the `width` argument controlled the height. They have been replaced with `nrow` and `ncol`. (#4906)
- `skimage.segmentation.flood_fill` and `skimage.segmentation.flood` now consistently handle negative values for `seed_point`.
- Segmentation faults in `segmentation.flood` have been fixed (#4948, #4972)
- A segfault in `draw.polygon` for the case of 0-d input has been fixed (#4943).
- In `registration.phase_cross_correlation`, a `ValueError` is raised when NaNs are found in the computation (as a result of NaNs in input images). Before this fix, an incorrect value could be returned where the input images had NaNs (#4886).
- Fix edge filters not respecting padding mode (#4907)
- Use `v{}` for version tags with `pooch` (#5104, #5110)
- Fix compilation error in XCode 12 (#5107, #5111)

Deprecations

- The `indices` argument in `skimage.feature.peak_local_max` has been deprecated. Indices will always be returned. (#4752)
- In `skimage.feature.structure_tensor`, an `order` argument has been introduced which will default to ‘rc’ starting in version 0.20. (#4841)
- `skimage.feature.structure_tensor_eigvals` has been deprecated and will be removed in version 0.20. Use `skimage.feature.structure_tensor_eigenvalues` instead.
- The `skimage.viewer` subpackage and the `skivi` script have been deprecated and will be removed in version 0.20. For interactive visualization we recommend using dedicated tools such as `napari` or `plotly`. In a similar vein, the `qt` and `skivi` plugins of `skimage.io` have been deprecated and will be removed in version 0.20. (#4941, #4954)
- In `skimage.morphology.selem.rectangle` the arguments `width` and `height` have been deprecated. Use `nrow` and `ncol` instead.
- The explicit setting `threshold_rel=0` was removed from the Examples of the following docstrings: `skimage.feature.BRIEF`, `skimage.feature.corner_harris`, `skimage.feature.corner_shi_tomasi`, `skimage.feature.corner_foerstner`, `skimage.feature.corner_fast`, `skimage.feature.corner_subpix`, `skimage.feature.corner_peaks`, `skimage.feature.corner_orientations`, and `skimage.feature._detect_octave`.
- In `skimage.restoration._denoise`, the warning regarding `rescale_sigma=None` was removed.
- In `skimage.restoration._cycle_spin`, the `# doctest: +SKIP` was removed.

Development process

- Fix #3327: Add functionality for benchmark coverage (#3329)
- Release process notes have been improved. (#4228)
- `pyproject.toml` has been added to the sdist.
- Build and deploy dev/master documentation using GitHub Actions (#4852)
- Website now deploys itself (#4870)
- build doc on circle ci and link artifact (#4881)
- Benchmarks can now run on older scikit-image commits (#4891)
- Website analytics are tracked using plausible.io and can be visualized on <https://plausible.io/scikit-image.org> (#4893)
- Artifacts for the documentation build are now found in each pull request (#4881).
- Documentation source files can now be written in Markdown in addition to ReST, thanks to `myst` (#4863).
- update trove classifiers and tests for Python 3.9 + fix pytest config (#5052)
- fix Azure Pipelines, pytest config, and trove classifiers for Python 3.8 (#5054)
- Moved our testing from Travis to GitHub Actions (#5074)
- We now build our wheels on GitHub Actions on the main repo using `cibuildwheel`. Many thanks to the matplotlib and scikit-learn developers for paving the way for us! (#5080)
- Disable Travis-CI builds (#5099, #5111)
- Improvements to CircleCI build: no parallelization and caching) (#5097, #5119)

Other Pull Requests

- Manage iradon input and output data type (#4298)
- random walker: Display a warning when the probability is outside [0,1] for a given tol (#4631)
- MAINT: remove unused cython file (#4633)
- Forget legacy data dir (#4662)
- Setup longdesc markdown and switch to 0.18dev (#4663)
- Optional pooch dependency (#4666)
- Adding new default values to functions on doc/examples/segmentation/plot_ncut (#4676)
- Reintroduced convert with a strong deprecation warning (#4681)
- In release notes, better describe skimage's relationship to ecosystem (#4689)
- Perform some todo tasks for 0.18 (#4690)
- Perform todo tasks for 0.17! (#4691)
- suppressing warnings from gallery examples (#4692)
- release notes for 0.17.2 (#4702)
- Fix gallery example mentioning deprecated argument (#4706)
- Specify the encoding of files opened in the setup phase (#4713)

- Remove duplicate fused type definition (#4724)
- Blacklist cython version 0.29.18 (#4730)
- Fix CI failures related to conversion of np.floating to dtype (#4731)
- Fix Ci failures related to array ragged input numpy deprecation (#4735)
- Unwrap decorators before resolving link to source (sphinx.ext.linkcode) (#4740)
- Fix plotting error in j-invariant denoising tutorial (#4744)
- Highlight all source lines with HTML doc “source” links (sphinx.ext.linkcode) (#4746)
- Turn checklist boxes into bullet points inside the pull request template (#4747)
- Deprecate (min_distance < 1) and (footprint.size < 2) in peak_local_max (#4753)
- forbid dask 2.17.0 to fix CI (#4758)
- try to fix ci which is broken because of pyqt5 last version (#4788)
- Remove unused variable in j invariant docs (#4792)
- include all md files in manifest.in (#4793)
- Remove additional “::” to make plot directive work. (#4798)
- Use optipng to compress images/thumbnails in our gallery (#4800)
- Fix runtime warning in blob.py (#4803)
- Add TODO task for sphinx-gallery>=0.9.0 to remove enforced thumbnail_size (#4804)
- Change SSIM code example to use real MSE (#4807)
- Let biomed example load image data with Pooch. (#4809)
- Tweak threshold_otsu error checking - closes #4811 (#4812)
- Ensure assert messages from Cython rank filters are informative (#4815)
- Simplify equivalent_diameter function (#4819)
- DOC: update subpackage descriptions (#4825)
- style: be explicit when stacking arrays (#4826)
- MAINT: import Iterable from collections.abc (Python 3.9 compatibility) (#4834)
- Silence several warnings in the test suite (#4837)
- Silence a few RuntimeWarnings in the test suite (#4838)
- handle color string mapping correctly (#4840)
- DOC: Autoformat docstrings in io.*.py (#4845)
- Update min req for pillow due to CVE-2020-10379 and co. (#4861)
- DOC: First pass at format conversion, rst -> myst (#4863)
- Fixed typo in comment (#4867)
- Alternative wording for install guide PR #4750 (#4871)
- DOC: Clarify condition on unique vertices returned by marching cubes (#4872)
- Remove unmaintained wiki page link in contributor guidelines (#4873)
- new matomo config (#4879)

- Fix Incorrect documentation for skimage.util.img_as_int Issue (#4888)
- Minor edit for proper doc rendering (#4897)
- Changelog back-log (#4898)
- minor refactoring in phase_cross_correlation (#4901)
- Fix draw.circle/disk deprecation message, fixes #4884 (#4908)
- Add versionchanged tag for new opt param in measure.find_contours() (#4909)
- Declare build dependencies (#4920)
- Replace words with racial connotations (#4921)
- Fixes to apply_parallel for functions working with multichannel data (#4927)
- Improve description of h_maxima and h_minima functions (#4928) (#4929)
- CI: Skip doc build for PYTHONOPTIMIZE=2 (#4930)
- MAINT: Remove custom fused type in skimage/morphology/_max_tree.pyx (#4931)
- MAINT: remove numpydoc option, issue fixed in numpydoc 1.0 (#4932)
- modify development version string to allow use with NumpyVersion (#4947)
- CI: Add verbose option to avoid travis timeout for OSX install script (#4956)
- Fix CI: ban sphinx-gallery 0.8.0 (#4960)
- Alias for data.chelsea: data.cat() (#4962)
- Fix typo. (#4963)
- CI: Use Travis wait improved to avoid timeout for OSX builds (#4965)
- Small enhancement in “Contour finding” example: Removed unused variable n (#4967)
- MAINT: remove unused imports (#4968)
- MAINT: Remove conditional import on networkx (#4970)
- forbid latest version of pyqt (#4973)
- Remove warnings/explicit settings on feature, restoration (#4974)
- Docstring improvements for label and regionprops_label (#4983)
- try to fix timeout problem with circleci (#4986)
- improve Euler number example (#4989)
- [website] Standardize Documentation index page. (#4990)
- Proofread INSTALL file. (#4991)
- Catch leftover typos in INSTALL file. (#4992)
- Let tifffile.imread handle additional keyword arguments (#4997)
- Update docstring for random_noise function (#5001)
- Update sphinx mapping for sklearn and numpy (#5003)
- Update docstring slic superpixels (#5014)
- Bump numpy versions to match scipy (kinda) (#5016)
- Fix usage of numpy.pad for old versions of numpy (#5017)

- [MRG] Update documentation to new data.camera() (#5018)
- bumped plotly requirement for docs (#5021)
- Fix IndexError when calling hough_line_peaks with too few angles (#5024)
- Code simplification after latest numpy bump (#5027)
- Fixes broken link to CODE_OF_CONDUCT.md (#5030)
- Specify whether core dev should merge right after second approving review. (#5040)
- Update pytest configuration to include `test_` functions (#5044)
- MAINT Build fix for pyodide (#5059)
- reduce OSX build time so that Travis is happy (#5067)
- DOC: document the normalized kernel in prewitt_h, prewitt_v (#5076)
- Some minor tweaks to CI (#5079)
- removed usage of numpy's private functions from util.arraycrop (#5081)
- peak_local_max: remove deprecated *indices* argument from examples (#5082)
- Replace np.bool, np.float, and np.int with bool, float, and int (#5103, #5108)
- change plausible script to track outbound links (#5115, #5123)
- Remove Python 3.6 support (#5117, #5125)
- Optimize ensure_spacing (#5062, #5135)

52 authors added to this release [alphabetical by first name or login]

A warm thank you to all contributors who added to this release. A fraction of contributors were first-time contributors to open source and a much larger fraction first-time contributors to scikit-image. It's a great feeling for maintainers to welcome new contributors, and the diversity of scikit-image contributors is surely a big strength of the package.

- Abhishek Arya
- Abhishek Patil
- Alexandre de Siqueira
- Ben Nathanson
- Cameron Blocker
- Chris Roat
- Christoph Gohlke
- Clement Ng
- Corey Harris
- David McMahon
- David Mellert
- Devi Sandeep
- Egor Panfilov
- Emmanuelle Gouillart
- François Boulogne

- Genevieve Buckley
- Gregory R. Lee
- Harry Kwon
- iofall (cedarfall)
- Jan Funke
- Juan Nunez-Iglesias
- Julian Gilbey
- Julien Jerphanion
- kalpana
- kolibril13 (kolibril13)
- Kushaan Gupta
- Lars Grüter
- Marianne Corvellec
- Mark Harfouche
- Marvin Albert
- Matthias Busonnier
- Max Frei
- Nathan
- neeraj3029 (neeraj3029)
- Nick
- notmatthancock (matt)
- OGordon100 (OGordon100)
- Owen Solberg
- Riadh Fezzani
- Robert Haase
- Roman Yurchak
- Ronak Sharma
- Ross Barnowski
- Ruby Werman
- ryanlu41 (ryanlu41)
- Sebastian Wallkötter
- Shyam Saladi
- Stefan van der Walt
- Terence Honles
- Volker Hilsenstein
- Wendy Mak

- Yogendra Sharma

41 reviewers added to this release [alphabetical by first name or login]

- Abhishek Arya
- Abhishek Patil
- Alexandre de Siqueira
- Ben Nathanson
- Chris Roat
- Clement Ng
- Corey Harris
- Cris Luengo
- David Mellert
- Egor Panfilov
- Emmanuelle Gouillart
- François Boulogne
- Gregory R. Lee
- Harry Kwon
- Jan Funke
- Juan Nunez-Iglesias
- Julien Jerphanion
- kalpana
- Kushaan Gupta
- Lars Grüter
- Marianne Corvellec
- Mark Harfouche
- Marvin Albert
- neeraj3029
- Nick
- OGordon100
- Riadh Fezzani
- Robert Haase
- Ross Barnowski
- Ruby Werman
- ryanlu41
- Scott Trinkle
- Sebastian Wallkötter

- Stanley_Wang
- Stefan van der Walt
- Steven Brown
- Stuart Mumford
- Terence Honles
- Volker Hilsenstein
- Wendy Mak

1.4.19 scikit-image 0.17.2 release notes

We're happy to announce the release of scikit-image v0.17.2, which is a bug-fix release.

Bug fixes

- We made `pooch` an optional dependency, since it has been added as required dependency by mistake (#4666), and we fixed a bug about the path used for `pooch` to download data (#4662)
- The support of float 32 images was corrected for `slic` segmentation, `ORB` and `BRIEF` feature detectors (#4683, #4684, #4685, #4696, #4697)
- **We removed deprecated arguments (#4691)**
 - `mask`, `shift_x`, and `shift_y` from `skimage.filters.median`
 - `beta1` and `beta2` from `skimage.filters.frangi`
 - `beta1` and `beta2` from `skimage.filters.hessian`
 - `dtype` from `skimage.io.imread`
 - `img` from `skimage.morphology.skeletonize_3d`.
- Gallery examples were updated to suppress warnings and take into account new default values in some functions (#4692 and #4676)

6 authors added to this release [alphabetical by first name or login]

- Alexandre de Siqueira
- Emmanuelle Gouillart
- François Boulogne
- Juan Nunez-Iglesias
- Mark Harfouche
- Riadh Fezzani

1.4.20 scikit-image 0.17.1 release notes

We're happy to announce the release of scikit-image v0.17.1!

scikit-image is an image processing toolbox for SciPy that includes algorithms for segmentation, geometric transformations, color space manipulation, analysis, filtering, morphology, feature detection, and more.

For more information, examples, and documentation, please visit our website:

<https://scikit-image.org>

Many thanks to the 54 authors who contributed the amazing number of 213 merged pull requests! scikit-image is a community-based project and we are happy that this number includes first-time contributors to scikit-image.

Special thanks for the release to the Cython team, who helped us make our code compatible with their coming Cython 3.0 release.

New Features

- Hyperparameter calibration of denoising algorithms with `restoration.calibrate_denoiser` (#3824), with corresponding gallery example and tutorial.
- `measure.profile_line` has a new `reduce_func` parameter to accept a reduction operation to be computed on pixel values along the profile (#4206)
- nD windows for reducing spectral leakage when computing the FFT of n-dimensional images, with `filters.window` (#4252) (with new gallery example)
- Add Minkowski distance metric support to `corner_peak` (#4218)
- `util.map_array` was introduced to map a set of pixel values to another one (for example to map region labels to the size of regions in an image of labels) #4612 and #4646
- Masked marching cubes (#3829)
- The SLIC superpixel algorithm now accepts a mask to exclude some parts of the image and force the superpixel boundaries to follow the boundary of the mask (#3850)
- Pooch – on the fly download of datasets from github: we introduced the possibility to include larger datasets in the `data` submodule, thanks to the `pooch` library. `data.download_all` fetches all datasets. (#3945)
- Starting with this version, our gallery examples now have links to run the example notebook on a binder instance. (#4543)

New doc tutorials and gallery examples have been added to the use of `regionprops_table` (#4348) geometrical transformations (#4385), and the registration of rotation and scaling with no shared center (#4515). A new section on registration has been added to the gallery (#4575).

Improvements

- scikit-image aims at being fully compatible with 3D arrays, and when possible with nD arrays. nD support has been added to color conversion functions (#4418), to the CLAHE `exposure.equalize_adapthist` algorithm (#4598) and to the Sobel, Scharr, and Prewitt filters (#4347).
- Multichannel support for `denoise_tv_bregman` (#4446)
- The memory footprint of `segmentation.relabel_sequential` has been reduced in the case of labels much larger than the number of labels (#4612)
- Random ellipses are now possible in `draw.random_shapes` (#4493)

- Add border conditions to ridge filters (#4396)
- *segmentation.random_walker* new Jacobi preconditioned conjugate gradient mode (#4359) and minor corrections #4630
- Warn when rescaling with NaN in exposure.intensity_range (#4265)

We have also improved the consistency of several functions regarding the way they handle data types

- Make dtype consistent in filters.rank functions (#4289)
- Fix colorconv float32 to double cast (#4296)
- Prevent radon from upcasting float32 arrays to double (#4297)
- Manage iradon_sart input and output data type (#4300)

API Changes

- When used with floating point inputs, *denoise_wavelet* no longer rescales the range of the data or clips the output to the range [0, 1] or [-1, 1]. For non-float inputs, rescaling and clipping still occurs as in prior releases (although with a bugfix related to the scaling of *sigma*).
- For 2D input, edge filters (Sobel, Scharr, Prewitt, Roberts, and Farid) no longer set the boundary pixels to 0 when a mask is not supplied. This was changed because the boundary mode for *scipy.ndimage.convolve* is now 'reflect', which allows meaningful values at the borders for these filters. To retain the old behavior, pass *mask=np.ones(image.shape, dtype=bool)* (#4347)
- When *out_range* is a range of numbers and not a dtype in *skimage.exposure.rescale_intensity()*, the output data type will always be float (#4585)
- The values returned by *skimage.exposure.equalize_adapthist()* will be slightly different from previous versions due to different rounding behavior (#4585)
- Move masked_register_translation from feature to registration (#4503)
- Move register_translation from skimage.feature to skimage.registration (#4502)
- Move watershed from morphology to segmentation (#4443)
- Rename draw.circle() to draw.disk() (#4428)
- The forward and backward maps returned by *skimage.segmentation.relabel_sequential()* are no longer NumPy arrays, but more memory-efficient *ArrayMap* objects that behave the same way for mapping. See the *relabel_sequential* documentation for more details. To get NumPy arrays back, cast it as a NumPy array: *np.asarray(forward_map)* (#4612)

Bugfixes

- *denoise_wavelet*: For user-supplied *sigma*, if the input image gets rescaled via *img_as_float*, the same scaling will be applied to *sigma* to preserve the relative scale of the noise estimate. To restore the old, behaviour, the user can manually specify *rescale_sigma=False*.
- Fix Frangi artefacts around the image (#4343)
- Fix Negative eigenvalue in inertia_tensor_eigvals due to floating point precision (#4589)
- Fix morphology.flood for F-ordered images (#4556)
- Fix h_maxima/minima strange behaviors on floating point image input (#4496)
- Fix peak_local_max coordinates ordering (#4501)

- Sort naturally peaks coordinates of same amplitude in peak_local_max (#4582)
- Fix denoise_nl_means data type management (#4322)
- Update rescale_intensity to prevent under/overflow and produce proper output dtype (#4585)

(other small bug fixes are part of the list of other pull requests at the end)

Deprecations

The minimal supported Python version by this release is 3.6.

- Parameter `inplace` in `skimage.morphology.flood_fill` has been deprecated in favor of `in_place` and will be removed in version scikit-image 0.19.0 (#4250).
- `skimage.segmentation.circle_level_set` has been deprecated and will be removed in 0.19. Use `skimage.segmentation.disk_level_set` instead.
- `skimage.draw.circle` has been deprecated and will be removed in 0.19. Use `skimage.draw.disk` instead.
- Deprecate filter argument in `iradon` due to clash with python keyword (#4158)
- Deprecate marching_cubes_classic (#4287)
- Change `label2rgb` default background value from -1 to 0 (#4614)
- Deprecate `rgb2grey` and `grey2rgb` (#4420)
- Complete deprecation of `circle` in `morphsnakes` (#4467)
- Deprecate non RGB image conversion in `rgb2gray` (#4838, #4439), and deprecate non gray scale image conversion in `gray2rgb` (#4440)

The list of other pull requests is given at the end of this document, after the list of authors and reviewers.

54 authors added to this release [alphabetical by first name or login]

- aadideshpande (aadideshpande)
- Alexandre de Siqueira
- Asaf Kali
- Cedric
- D-Bhatta (D-Bhatta)
- Danielle
- Davis Bennett
- Dhiren Serai
- Dylan Cutler
- Egor Panfilov
- Emmanuelle Gouillart
- Eoghan O'Connell
- Eric Jelli
- Eric Perlman
- erjel (erjel)

- Evan Widloski
- François Boulogne
- Gregory R. Lee
- Hazen Babcock
- Jan Eglinger
- Joshua Batson
- Juan Nunez-Iglesias
- Justin Terry
- kalvdans (kalvdans)
- Karthikeyan Singaravelan
- Lars Grüter
- Leengit (Leengit)
- leGIT-bot (leGIT-bot)
- LGiki
- Marianne Corvellec
- Mark Harfouche
- Marvin Albert
- mellertd (Dave Mellert)
- Miguel de la Varga
- Mostafa Alaa
- Mojdeh Rastgoo (mrastgoo)
- notmatthancock (matt)
- Ole Streicher
- Riadh Fezzani
- robroooh (robroooh)
- SamirNasibli
- schneefux (schneefux)
- Scott Sievert
- Stefan van der Walt
- Talley Lambert
- Tim Head (betatim)
- Thomas A Caswell
- Timothy Sweetser
- Tony Tung
- Uwe Schmidt
- VolkerH (VolkerH)

- Xiaoyu Wu
- Yuanjin Lu
- Zaccharie Ramzi
- Zhōu Bówēi

35 reviewers added to this release [alphabetical by first name or login]

- Alexandre de Siqueira
- Asaf Kali
- D-Bhatta
- Egor Panfilov
- Emmanuelle Gouillart
- Eoghan O'Connell
- erjel
- François Boulogne
- Gregory R. Lee
- Hazen Babcock
- Jacob Quinn Shenker
- Jirka Borovec
- Josh Warner
- Joshua Batson
- Juan Nunez-Iglesias
- Justin Terry
- Lars Grüter
- Leengit
- leGIT-bot
- Marianne Corvellec
- Mark Harfouche
- Marvin Albert
- mellertd
- Miguel de la Varga
- Riadh Fezzani
- robroooh
- SamirNasibli
- Stefan van der Walt
- Timothy Sweetser
- Tony Tung

- Uwe Schmidt
- VolkerH
- Xiaoyu Wu
- Zhōu Bówēi

Other Pull Requests

- [WIP] DOC changing the doc in plot_glcms (#2789)
- Document tophat in the gallery (#3609)
- More informative error message on boolean images for regionprops (#4156)
- Refactor/fix threshold_multotsu (#4178)
- Sort the generated API documentation alphabetically (#4208)
- Fix the random Linux build fails in travis CI (#4227)
- Initialize starting vector for `scipy.sparse.linalg.eigsh` to ensure reproducibility in graph_cut (#4251)
- Add histogram matching test (#4254)
- MAINT: use SciPy's implementation of convolution method (#4267)
- Improve CSS for SKIP rendering (#4271)
- Add toggle for prompts in docstring examples next to copybutton (#4273)
- Tight layout for glcm example in gallery (#4285)
- Forward port 0.16.2 release notes (#4290)
- Fix typo in `hog` docstring (#4302)
- pyramid functions take `preserve_range` kwarg (#4310)
- Create test and fix types (#4311)
- Deprecate `numpy.pad` wrapping (#4313)
- Clarify merge policy in core contributor guide (#4315)
- Regionprops is empty bug (#4316)
- Add check to avoid import crashing (#4319)
- Fix typo in `simple_metrics` docstring (#4323)
- Make `peak_local_max` `exclude_border` independent and anisotropic (#4325)
- Fix `blob_log`/`blob_dog` and their corresponding tests (#4327)
- Add section on closing issues to core dev guide (#4328)
- Use gaussian filter output array if provided (#4329)
- Move cython pinning forward (#4330)
- Add python 3.8 to the build matrix (#4331)
- Avoid importing mathematical functions from `scipy` as told ;) (#4332)
- Add `dtype` keyword argument to `block_reduce` and small documentation changes (#4334)
- Add explicit use of 32-bit int in `fast_exp` (#4338)
- Fix single precision cast to double in `slic` (#4339)

- Change *measure.block_reduce* to accept explicit *func_kwarg*s kwd (#4341)
- Fix *equalize_adapthist* border artifacts (#4349)
- Make *hough_circle_peaks* respect *min_xdistance*, *min_ydistance* (#4350)
- Deprecate *CONTRIBUTORS.txt* and replace by git shortlog command (#4351)
- Add warning on pillow version if reading a MPO image (#4354)
- Minor documentation improvement in *measure.block_reduce* (#4355)
- Add example to highlight *regionprops_table* (#4356)
- Remove code that tries to avoid upgrading large dependencies from *setup.py* (#4362)
- Fix float32 promotion in cubic interpolation (#4363)
- Update to the new way of generating Sphinx search box (#4367)
- clarify *register_translation* example description (#4368)
- Bump *scipy* minimum version to 1.0.1 (#4372)
- Fixup OSX Builds by skipping building with *numpy* 1.18.0 (#4376)
- Bump *pywavelets* to 0.5.2 (#4377)
- mini-galleries for classes as well in API doc (#4381)
- *gallery*: Fix typo + reduce the angle to a reasonable value (#4386)
- *setup*: read long description from *README* (#4392)
- Do not depend on test execution order for success (#4393)
- *_adapthist* module refactoring and memory use reduction (#4395)
- Documentation fixes for transform (*rescale*, *warp_polar*) (#4401)
- *DOC*: specify the meaning of *m* in *ransac* formula (#4404)
- Updating *link* to values in core developer guide (#4405)
- Fix *subtract_mean* underflow correction (#4409)
- Fix hanging documentation build in Azure (#4411)
- Fix warnings regarding invalid escape sequences. (#4414)
- Fix the URLs in *skimage.transform.pyramids* (#4415)
- Fix *profile_line* interpolation errors (#4416)
- *MAINT*: replace *circle_level_set* by *disk_level_set* (#4421)
- Add *stacklevel=2* to deprecation warnings in *skimage.measure.marching_cubes* (#4422)
- Deprecate *rank.tophat* and *rank.bottomhat* (#4423)
- Add *gray2rgba* and deprecate *RGBA* support in *gray2rgb* (#4424)
- *ISSUE_TEMPLATE*: add note about *image.sc* forum (#4429)
- Fix the link in *skips.1-governance* (#4432)
- Fix the dead link in *skimage.feature.canny* (#4433)
- Fix *use_quantiles* behavior in *canny* (#4437)
- Remove redundant checks for threshold values in Canny (#4441)

- Difference of Gaussians function (#4445)
- Fix test for denoise_tv_bregman accepting float32 and float64 as inputs (#4448)
- Standardize colon usage in docstrings (#4449)
- Bump numpy version to 1.15.1 (#4452)
- Set minimum tifffile version to fix numpy incompatibility (#4453)
- Cleanup warnings regarding denoise_wavelet (#4456)
- Address FutureWarning from numpy in subtype check in reginoprops (#4457)
- Skip warnings in doctests for warning module (#4458)
- Skip doctests for deprecated functions rank.tophat rank.bottomhat since they emit warnings (#4459)
- Skip morphology.watershed doctest since it was moved and emits a warning (#4460)
- Use rgba2rgb directly where rgb kind is inferred (#4461)
- Cleanup corner peaks warnings (#4463)
- Fix edgecase bugs in segmentation.relabel_sequential (#4465)
- Fix deltaE cmc close colors bug (#4469)
- Fix bool array warping (#4470)
- Fix bool array profile_line (#4471)
- Fix values link in governance (#4472)
- Improving example on filters (#4479)
- reduce runtime of non-local means tests (#4480)
- Add sponsor button (#4481)
- reduced the duration of the longest tests (#4487)
- tiny improvements to haar feature examples (#4490)
- Add min version to sphinx-gallery >= 0.3.1 to work with py3.8 (#4498)
- Fix KeyError in find_contours (#4505)
- Fix bool array save with imageio plugin (#4512)
- Fixing order of elements in docstrings of skimage/color/colorconv (#4518)
- Fix exposure_adapthist return when clip_limit == 1 (#4519)
- Adding info on venv activation on Windows (#4521)
- Fix similarity transform scale (#4524)
- Added explanation in the example of *segmentation/plot_label.py* to make the background transparent (#4527)
- Add example code for generating structuring elements. (#4528)
- Block imread version 0.7.2 due to build failure (#4529)
- Maint: edits to suppress some warnings (unused imports, blank lines) (#4530)
- MNT: remove duplicate nogil specification (#4546)
- Block pillow 7.1.0, see #4548 (#4551)
- Fix binder requirements (#4555)

- Do not enforce pil plugin in skimage.data (#4560)
- Remove “backport to 0.14” in github template (#4561)
- Fix inconsistency in docstring (filters.median) (#4562)
- Disable key check for texlive in travis-mac as a temporary workaround (#4565)
- Bump Pywavelets min requirement to 1.1.1 (#4568)
- Strip backslash in sphinx 3.0.0 (#4569)
- Remove binary specification from match_descriptors docstring (#4571)
- Remove importing skimage.transform as tf (#4576)
- Add note to remove option in doc config when numpydoc will be patched (#4578)
- update task in TODO.txt (#4579)
- Rename convert to _convert, as it is a private function (#4590)
- Do not overwrite data module in plot_skeleton.py (#4591)
- [CI fix] add import_array in cython files where numpy is cimport-ed (#4592)
- Recommend cnp.import_array in contribution guide (#4593)
- Add example of natsort usage in documentation (#4599)
- Fix broken and permanently moved links (#4600)
- Fix typo in cython import_array (#4602)
- Update min required sphinx version for sphinx-copybutton (#4604)
- Clarify error message when montaging multichannel nD images and multichannel=False (#4607)
- Fix register_translation warning message (#4609)
- Add notes on deprecation warnings in marching_cube_* and gray2rgb (#4610)
- Improve loading speed of our gallery by reducing the thumbnail size (#4613)
- Fixed wrong behaviour of *exposure.rescale_intensity* for constant input. (#4615)
- Change math formatting in the docstrings (#4617)
- Add .mypy_cache to the gitignore (#4620)
- typo fixes for register rotation gallery example (#4623)
- Userguide: add a visualization chapter (#4627)
- Fix deprecation warnings due to invalid escape sequences. (#4628)
- add docstring examples for moments_hu and centroid (#4632)
- Update pooch registry with new file location (#4635)
- Misleading “ValueError: Input array has to be either 3- or 4-dimensional” in montage (#4638)
- Fix broken link (#4639)
- AffineTransform: Allow a single value for ‘scale’ to apply to both sx & sy (#4642)
- Fix CI - cython 3.0a4 (#4643)
- Fix sphinx (#4644)
- Fix ArrayMap test (#4645)

- Remove copy of tifffile; install from pip (#4235)
- Refactor/move neighborhood utility functions in morphology (#4209)

1.4.21 scikit-image 0.16.2 release notes

We're happy to announce the release of scikit-image v0.16.2!

scikit-image is an image processing toolbox for SciPy that includes algorithms for segmentation, geometric transformations, color space manipulation, analysis, filtering, morphology, feature detection, and more.

This is a bug fix release that addresses several critical issues from 0.16.1.

Bug fixes

- Migrate to networkx 2.x (#4236, #4237)
- Sync required numpy and dask to runtime versions (#4233, #4239)
- Fix wrong argument parsing in structural_similarity (#4246, #4247)
- Fix active contour gallery example after change to rc coordinates (#4257, #4262)

4 authors added to this release [alphabetical by first name or login]

- François Boulogne
- Jarrod Millman
- Mark Harfouche
- Ondrej Pesek

6 reviewers added to this release [alphabetical by first name or login]

- Alexandre de Siqueira
- Egor Panfilov
- François Boulogne
- Juan Nunez-Iglesias
- Mark Harfouche
- Nelle Varoquaux

1.4.22 scikit-image 0.16.1 release notes

We're happy to announce the release of scikit-image v0.16.1!

scikit-image is an image processing toolbox for SciPy that includes algorithms for segmentation, geometric transformations, color space manipulation, analysis, filtering, morphology, feature detection, and more.

For more information, examples, and documentation, please visit our website:

<https://scikit-image.org>

Starting from this release, scikit-image will follow the recently introduced NumPy deprecation policy, *NEP 29* <https://github.com/numpy/numpy/blob/master/doc/neps/nep-0029-deprecation_policy.rst>. Accordingly, scikit-image 0.16 drops support for Python 3.5. This release of scikit-image officially supports Python 3.6 and 3.7.

Special thanks to Matthias Bussonnier for [Frappuccino](#), which helped us catch all API changes and nail down the APIs for new features.

New Features

- New `skimage.evaluate` module containing simple metrics (mse, nrme, psd) and segmentation metrics (adapted rand error, variation of information) (#4025)
- n-dimensional TV-L1 optical flow algorithm for registration – `skimage.registration.optical_flow_tv11` (#3983)
- Draw a line in an n-dimensional array – `skimage.draw.line_nd` (#2043)
- 2D Farid & Simoncelli edge filters - `skimage.filters.farid`, `skimage.filters.farid_h`, and `skimage.filters.farid_v` (#3775)
- 2D majority voting filter assigning to each pixel the most commonly occurring value within its neighborhood – `skimage.filters.majority` (#3836, #3839)
- Multi-level threshold “multi-Otsu” method, a thresholding algorithm used to separate the pixels of an input image into several classes by maximizing the variances between classes – `skimage.filters.threshold_multithreshold` (#3872, #4174)
- New example data – `skimage.data.shepp_logan_phantom`, `skimage.data.colorwheel`, `skimage.data.brick`, `skimage.data.grass`, `skimage.data.roughwall`, `skimage.data.cell` (#3958, #3966)
- Compute and format image region properties as a table – `skimage.measure.regionprops_table` (#3959)
- Convert a polygon into a mask – `skimage.draw.poly2mask` (#3971, #3977)
- Visual image comparison helper `skimage.util.compare_images`, that returns an image showing the difference between two input images (#4089)
- `skimage.transform.warp_polar` to remap image into polar or log-polar coordinates. (#4097)

Improvements

- RANSAC: new option to set initial samples selected for initialization (#2992)
- Better repr and str for `skimage.transform.ProjectiveTransform` (#3525, #3967)
- Better error messages and data type stability to `skimage.segmentation.relabel_sequential` (#3740)
- Improved compatibility with dask arrays in some image thresholding methods (#3823)
- `skimage.io.ImageCollection` can now receive lists of patterns (#3928)
- Speed up `skimage.feature.peak_local_max` (#3984)
- Better error message when incorrect value for keyword argument `kind` in `skimage.color.label2rgb` (#4055)
- All functions from `skimage.drawing` now supports multi-channel 2D images (#4134)

API Changes

- Deprecated subpackage `skimage.novice` has been removed.
- Default value of `multichannel` parameters has been set to `False` in `skimage.transform.rescale`, `skimage.transform.pyramid_reduce`, `skimage.transform.pyramid_laplacian`, `skimage.transform.pyramid_gaussian`, and `skimage.transform.pyramid_expand`. Guessing is no longer performed for 3D arrays.
- Deprecated argument `visualise` has been removed from `skimage.feature.hog`. Use `visualize` instead."
- `skimage.transform.seam_carve` has been completely removed from the library due to licensing restrictions.
- Parameter `as_grey` has been removed from `skimage.data.load` and `skimage.io.imread`. Use `as_gray` instead.
- Parameter `min_size` has been removed from `skimage.morphology.remove_small_holes`. Use `area_threshold` instead.
- Deprecated `correct_mesh_orientation` in `skimage.measure` has been removed.
- `skimage.measure._regionprops` has been completely switched to using row-column coordinates. Old x-y interface is not longer available.
- Default value of `behavior` parameter has been set to `ndimage` in `skimage.filters.median`.
- Parameter `flatten` in `skimage.io.imread` has been removed in favor of `as_gray`.
- Parameters `Hxx`, `Hxy`, `Hyx` have been removed from `skimage.feature.corner.hessian_matrix_eigvals` in favor of `H_elems`.
- Default value of `order` parameter has been set to `rc` in `skimage.feature.hessian_matrix`.
- `skimage.util.img_as_*` functions no longer raise precision and/or loss warnings.

Bugfixes

- Corrected error with scales attribute in ORB.detect_and_extract (#2835) The scales attribute wasn't taking into account the mask, and thus was using an incorrect array size.
- Correct for bias in Inverse Random Transform (`skimage.transform.irandon`) (#3067) Fixed by using the Ramp filter equation in the spatial domain as described in the reference
- Fix a rounding issue that caused a rotated image to have a different size than the input (`skimage.transform.rotate`) (#3173)
- RANSAC uses random subsets of the original data and not bootstraps. (#3901, #3915)
- Canny now produces the same output regardless of `dtype` (#3919)
- Geometry Transforms: avoid division by zero & some degenerate cases (#3926)
- Fixed float32 support in `denoise_bilateral` and `denoise_tv_bregman` (#3936)
- Fixed computation of Meijering filter and avoid ZeroDivisionError (#3957)
- Fixed `skimage.filters.threshold_li` to prevent being stuck on stationnary points, and thus at local minima or maxima (#3966)
- Edited `skimage.exposure.rescale_intensity` to return input image instead of nans when all 0 (#4015)
- Fixed `skimage.morphology.medial_axis`. A wrong indentation in Cython caused the function to not behave as intended. (#4060)
- Fixed `skimage.restoration.denoise_bilateral` by correcting the padding in the gaussian filter(#4080)

- Fixed `skimage.measure.find_contours` when input image contains NaN. Contours interesting NaN will be left open (#4150)
- Fixed `skimage.feature.blob_log` and `skimage.feature.blob_dog` for 3D images and anisotropic data (#4162)
- Fixed `skimage.exposure.adjust_gamma`, `skimage.exposure.adjust_log`, and `skimage.exposure.adjust_sigmoid` such that when provided with a 1 by 1 ndarray, it returns 1 by 1 ndarrays and not single number floats (#4169)

Deprecations

- Parameter `neighbors` in `skimage.measure.convex_hull_object` has been deprecated in favor of `connectivity` and will be removed in version 0.18.0.
- The following functions are deprecated in favor of the `skimage.metrics` module (#4025):
 - `skimage.measure.compare_mse`
 - `skimage.measure.compare_nrmse`
 - `skimage.measure.compare_psnr`
 - `skimage.measure.compare_ssim`
- The function `skimage.color.guess_spatial_dimensions` is deprecated and will be removed in 0.18 (#4031)
- The argument `bc` in `skimage.segmentation.active_contour` is deprecated.
- The function `skimage.data.load` is deprecated and will be removed in 0.18 (#4061)
- The function `skimage.transform.match_histogram` is deprecated in favor of `skimage.exposure.match_histogram` (#4107)
- The parameter `neighbors` of `skimage.morphology.convex_hull_object` is deprecated.
- The `skimage.transform.randon_transform` function will convert input image of integer type to float by default in 0.18. To preserve current behaviour, set the new argument `preserve_range` to True. (#4131)

Documentation improvements

- DOC: Improve the documentation of `transform.resize` with respect to the `anti_aliasing_sigma` parameter (#3911)
- Fix URL for stain deconvolution reference (#3862)
- Fix doc for denoise gaussian (#3869)
- DOC: various enhancements (cross links, gallery, ref...), mainly for corner detection (#3996)
- [DOC] clarify that the `inertia_tensor` may be nD in documentation (#4013)
- [DOC] How to test and write benchmarks (#4016)
- Spellcheck @CONTRIBUTING.txt (#4008)
- Spellcheck @doc/examples/segmentation/plot_watershed.py (#4009)
- Spellcheck @doc/examples/segmentation/plot_thresholding.py (#4010)
- Spellcheck @skimage/morphology/binary.py (#4011)
- Spellcheck @skimage/morphology/extrema.py (#4012)
- docs update for `downscale_local_mean` and N-dimensional images (#4079)

- Remove fancy language from 0.15 release notes (#3827)
- Documentation formatting / compilation fixes (#3838)
- Remove duplicated section in INSTALL.txt. (#3876)
- ENH: doc of ridge functions (#3933)
- Fix docstring for Threshold Niblack (#3917)
- adding docs to circle_perimeter_aa (#4155)
- Update link to NumPy docstring standard in Contribution Guide (replaces #4191) (#4192)
- DOC: Improve downscale_local_mean() docstring (#4180)
- DOC: enhance the result display in ransac gallery example (#4109)
- Gallery: use fstrings for better readability (#4110)
- MNT: Document stacklevel parameter in contribution guide (#4066)
- Fix minor typo (#3988)
- MIN: docstring improvements in canny functions (#3920)
- Minor docstring fixes for #4150 (#4184)
- Fix *full* parameter description in compare_ssim (#3860)
- State Bradley threshold equivalence in Niblack docstring (#3891)
- Add plt.show() to example-code for consistency. (#3908)
- CC0 is not equivalent to public domain. Fix the note of the horse image (#3931)
- Update the joblib link in tutorial_parallelization.rst (#3943)
- Fix plot_edge_filter.py references (#3946)
- Add missing argument to docstring of PaintTool (#3970)
- Improving documentation and tests for directional filters (#3956)
- Added new thorough examples on the inner working of `skimage.filters.threshold_li` (#3966)
- matplotlib: remove interpolation=nearest, none in our examples (#4002)
- fix URL encoding for wikipedia references in filters.rank.entropy and filters.rank.shannon_entropy docstring (#4007)
- Fixup integer division in examples (#4032)
- Update the links the installation guide (#4118)
- Gallery hough line transform (#4124)
- Cross-linking between function documentation should now be much improved! (#4188)
- Better documentation of the num_peaks of `skimage.feature.corner_peaks` (#4195)

Other Pull Requests

- Add benchmark suite for exposure module (#3312)
- Remove precision and sign loss warnings from `skimage.util.img_as_` (#3575)
- Propose SKIPs and add mission/vision/values, governance (#3585)
- Use user-installed tifffile if available (#3650)
- Simplify benchmarks pinnings (#3711)
- Add project_urls to setup for PyPI and other services (#3834)
- Address deprecations for 0.16 release (#3841)
- Followup deprecations for 0.16 (#3851)
- Build and test the docs in Azure (#3873)
- Pin numpydoc to pre-0.8 to fix dev docs formatting (#3893)
- Change all HTTP links to HTTPS (#3896)
- Skip extra deps on OSX (#3898)
- Add location for Sphinx 2.0.1 search results; clean up templates (#3899)
- Fix CSS styling of Sphinx 2.0.1 + numpydoc 0.9 rendered docs (#3900)
- Travis CI: The sudo: tag is deprecated in Travis (#4164)
- MNT Preparing the 0.16 release (#4204)
- FIX generate_release_note when contributor_set contains None (#4205)
- Specify that travis should use Ubuntu xenial (14.04) not trusty (16.04) (#4082)
- MNT: set stack level accordingly in lab2xyz (#4067)
- MNT: fixup stack level for filters ridges (#4068)
- MNT: remove unused import *deprecated* from filters.thresholding (#4069)
- MNT: Set stacklevel correctly in io matplotlib plugin (#4070)
- MNT: set stacklevel accordingly in felzenszwalb_cython (#4071)
- MNT: Set stacklevel accordingly in img_as_* (convert) (#4072)
- MNT: set stacklevel accordingly in util.shape (#4073)
- MNT: remove extreaneous matplotlib warning (#4074)
- Suppress warnings in tests for viewer (#4017)
- Suppress warnings in test suite regarding measure.label (#4018)
- Suppress warnings in test_rank due to type conversion (#4019)
- Add todo item for imread plugin testing (#3907)
- Remove matplotlib agg warning when using the sphinx gallery. (#3897)
- Forward-port release notes for 0.14.4 (#4137)
- Add tests for pathological arrays in threshold_li (#4143)
- setup.py: Fail gracefully when NumPy is not installed (#4181)
- Drop Python 3.5 support (#4102)

- Force imageio reader to return NumPy arrays (#3837)
- Fixing connecting to GitHub with SSH info. (#3875)
- Small fix to an error message of `skimage.measure.regionprops` (#3884)
- Unify skeletonize and skeletonize 3D APIs (#3904)
- Add location for Sphinx 2.0.1 search results; clean up templates (#3910)
- Pin numpy version forward (#3925)
- Replacing pyfits with Astropy to read FITS (#3930)
- Add warning for future dtype kwarg removal (#3932)
- MAINT: cleanup regionprop add PYTHONOPTIMIZE=2 to travis array (#3934)
- Adding complexity and new tests for filters.threshold_morphology (#3935)
- Fixup dtype kwarg warning in certain image plugins (#3948)
- don't cast integer to float before using it as integer in numpy logspace (#3949)
- avoid low contrast image save in a doctest. (#3953)
- MAINT: Remove unused `_convert_input` from filters.`_gaussian` (#4001)
- Set minimum version for imread so that it compiles from source on linux in test builds (#3960)
- Cleanup plugin utilization in data.load and testsuite (#3961)
- Select minimum imageio such that it is compatible with pathlib (#3969)
- Remove pytest-faulthandler from test dependencies (#3987)
- Fix tifffile and `__array_function__` failures in our CI (#3992)
- MAINT: Do not use assert in code, raise an exception instead. (#4006)
- Enable packagers to disable failures on warnings. (#4021)
- Fix numpy 117 rc and dask in thresholding filters (#4022)
- silence r,c warnings when property does not depend on r,c (#4027)
- remove warning filter, fix doc wrt r,c (#4028)
- Import Iterable from collections.abc (#4033)
- Import Iterable from collections.abc in vendored tiff file code (#4034)
- Correction of typos after #4025 (#4036)
- Rename internal function called `assert_*` -> `check_*` (#4037)
- Improve import time (#4039)
- Remove .meeseeksdev.yml (#4045)
- Fix mpl deprecation on grid() (#4049)
- Fix gallery after deprecation from #4025 (#4050)
- fix mpl future deprecation normed -> density (#4053)
- Add shape= to circle perimeter in hough_circle example (#4047)
- Critical: address internal warnings in test suite related to metrics 4025 (#4063)
- Use functools instead of a real function for the internal warn function (#4062)

- Test rank capture warnings in threadsafe manner (#4064)
- Make use of FFTs more consistent across the library (#4084)
- Fixup region props test (#4099)
- Turn single backquotes to double backquotes in filters (#4127)
- Refactor radon transform module (#4136)
- Fix broken import of rgb2gray in benchmark suite (#4176)
- Fix doc building issues with SKIPs (#4182)
- Remove several `__future__` imports (#4198)
- Restore deprecated coordinates arg to regionprops (#4144)
- Refactor/optimize threshold_multotsu (#4167)
- Remove Python2-specific code (#4170)
- *view_as_windows* incorrectly assumes that a contiguous array is needed (#4171)
- Handle case in which NamedTemporaryFile fails (#4172)
- Fix incorrect resolution date on SKIP1 (#4183)
- API updates before 0.16 (#4187)
- Fix conversion to float32 dtype (#4193)

Contributors to this release

- Abhishek Arya
- Alexandre de Siqueira
- Alexis Mignon
- Anthony Carapetis
- Bastian Eichenberger
- Bharat Raghunathan
- Christian Clauss
- Clement Ng
- David Breuer
- David Haberthür
- Dominik Kutra
- Dominik Straub
- Egor Panfilov
- Emmanuelle Gouillart
- Etienne Landuré
- François Boulogne
- Genevieve Buckley
- Gregory R. Lee

- Hadrien Mary
- Hamdi Sahloul
- Holly Gibbs
- Huang-Wei Chang
- i3v (i3v)
- Jarrod Millman
- Jirka Borovec
- Johan Jeppsson
- Johannes Schönberger
- Jon Crall
- Josh Warner
- Juan Nunez-Iglesias
- Kaligule (Kaligule)
- kczimm (kczimm)
- Lars Grueter
- Shachar Ben Harim
- Luis F. de Figueiredo
- Mark Harfouche
- Mars Huang
- Dave Mellert
- Nelle Varoquaux
- Ollin Boer Bohan
- Patrick J Zager
- Riadh Fezzani
- Ryan Avery
- Srinath Kailasa
- Stefan van der Walt
- Stuart Berg
- Uwe Schmidt

Reviewers for this release

- Alexandre de Siqueira
- Anthony Carapetis
- Bastian Eichenberger
- Clement Ng
- David Breuer
- Egor Panfilov
- Emmanuelle Gouillart
- Etienne Landuré
- François Boulogne
- Genevieve Buckley
- Gregory R. Lee
- Hadrien Mary
- Hamdi Sahloul
- Holly Gibbs
- Jarrod Millman
- Jirka Borovec
- Johan Jeppsson
- Johannes Schönberger
- Jon Crall
- Josh Warner
- jrmarscha
- Juan Nunez-Iglesias
- kczimm
- Lars Grueter
- leGIT-bot
- Mark Harfouche
- Mars Huang
- Dave Mellert
- Paul Müller
- Phil Starkey
- Ralf Gommers
- Riadh Fezzani
- Ryan Avery
- Sebastian Berg
- Stefan van der Walt

- Uwe Schmidt

1.4.23 scikit-image 0.15.0 release notes

We're happy to announce the release of scikit-image v0.15.0!

scikit-image is an image processing toolbox for SciPy that includes algorithms for segmentation, geometric transformations, color space manipulation, analysis, filtering, morphology, feature detection, and more.

For more information, examples, and documentation, please visit our website:

<https://scikit-image.org>

0.15 is the first scikit-image release that is only compatible with Python 3.5 and above. Python 2.7 users should strongly consider upgrading to Python 3.5+, or use the 0.14 long term support releases.

New Features

- N-dimensional flood fill, with tolerance (#3245)
- Attribute operators (#2680)
- Extension of register_translation to enable subpixel precision in 3D and optionally disable error calculation (#2880)
- unsharp mask filtering (#2772)
- New options connectivity, indices and allow_borders for skimage.morphology.local_maxima and local_minima. (#3022)
- Image translation registration for masked data (skimage.feature.masked_register_translation) (#3334)
- Frangi (vesselness), Meijering (neuriteness), and Sato (tubeness) filters (#3515)
- Allow float->float conversion of any range (#3052)
- Let lower precision float arrays pass through img_as_float (#3110)
- Lazy apply_parallel (allows optimization of dask array operations) (#3121)
- Add range option for histogram. (#2479)
- Add histogram matching (#3568)

Improvements

- Replace morphology.local_maxima with faster flood-fill based Cython version (#3022)
- skivi is now using qtpy for Qt4/Qt5/PySide/PySide2 compatibility (a new optional dependency).
- Performance is now monitored by [Airspeed Velocity](#). Benchmark results will appear at <https://pandas.pydata.org/speed/> (#3137)
- Speed up inner loop of GLCM (#3378)
- Allow tuple to define kernel in threshold_niblack and threshold_sauvola (#3596)
- Add support for anisotropic blob detection in blob_log and blob_dog (#3690)

API Changes

- `skimage.transform.seam_carve` has been removed because the algorithm is patented. (#3751)
- Parameter `dynamic_range` in `skimage.measure.compare_psnr` has been removed. Use parameter `data_range` instead. (#3313)
- `imageio` is now the preferred plugin for reading and writing images. (#3126)
- `imageio` is now a dependency of scikit-image. (#3126)
- `regular_grid` now returns a tuple instead of a list for compatibility with numpy 1.15 (#3238)
- `colorconv.separate_stains` and `colorconv.combine_stains` now uses base10 instead of the natural logarithm as discussed in issue #2995. (#3146)
- Default value of `clip_negative` parameter in `skimage.util.dtype_limits` has been set to `False`.
- Default value of `circle` parameter in `skimage.transform.radon` has been set to `True`.
- Default value of `circle` parameter in `skimage.transform.iradon` has been set to `True`.
- Default value of `mode` parameter in `skimage.transform.swirl` has been set to `reflect`.
- Deprecated `skimage.filters.threshold_adaptive` has been removed. Use `skimage.filters.threshold_local` instead.
- Default value of `multichannel` parameter in `skimage.restoration.denoise_bilateral` has been set to `False`.
- Default value of `multichannel` parameter in `skimage.restoration.denoise_nl_means` has been set to `False`.
- Default value of `mode` parameter in `skimage.transform.resize` and `skimage.transform.rescale` has been set to `reflect`.
- Default value of `anti_aliasing` parameter in `skimage.transform.resize` and `skimage.transform.rescale` has been set to `True`.
- Removed the `skimage.test` function. This functionality can be achieved by calling `pytest` directly.
- `morphology.local_maxima` now returns a boolean array (#3749)

Bugfixes

- Correct bright ridge detection for Frangi filter (#2700)
- `skimage.morphology.local_maxima` and `skimage.morphology.local_minima` no longer raise an error if any dimension of the image is smaller 3 and the keyword `allow_borders` was false.
- `skimage.morphology.local_maxima` and `skimage.morphology.local_minima` will return a boolean array instead of an array of 0s and 1s if the parameter `indices` was false.
- When `compare_ssim` is used with `gaussian_weights` set to `True`, the boundary crop used when computing the mean structural similarity will now exactly match the width of the Gaussian used. The Gaussian filter window is also now truncated at 3.5 rather than 4.0 standard deviations to exactly match the original publication on the SSIM. These changes should produce only a very small change in the computed SSIM value. There is no change to the existing behavior when `gaussian_weights` is `False`. (#3802)
- erroneous use of cython wrap around (#3481)
- Speed up block reduce by providing the appropriate parameters to numpy (#3522)
- Add `urllib.request` again (#3766)

- Repeat pixels in reflect mode when image has dimension 1 (#3174)
- Improve Li thresholding (#3402, 3622)

Deprecations

- Python 2 support has been dropped. Users should have Python >= 3.5. (#3000)
- `skimage.util.montage2d` has been removed. Use `skimage.util.montage` instead.
- `skimage.novice` is deprecated and will be removed in 0.16.
- `skimage.transform.resize` and `skimage.transform.rescale` option `anti_aliasing` has been enabled by default.
- `regionprops` will use row-column coordinates in 0.16. You can start using them now with `regionprops(..., coordinates='rc')`. You can silence warning messages, and retain the old behavior, with `regionprops(..., coordinates='xy')`. However, that option will go away in 0.16 and result in an error. This change has a number of consequences. Specifically, the “orientation” region property will measure the anticlockwise angle from a *vertical* line, i.e. from the vector (1, 0) in row-column coordinates.
- `skimage.morphology.remove_small_holes` `min_size` argument is deprecated and will be removed in 0.16. Use `area_threshold` instead.
- `skimage.filters.median` will change behavior in the future to have an identical behavior as `scipy.ndimage.median_filter`. This behavior can be set already using `behavior='ndimage'`. In 0.16, it will be the default behavior and removed in 0.17 as well as the parameter of the previous behavior (i.e., `mask`, `shift_x`, `shift_y`) will be removed.

Documentation improvements

- Correct rotate method’s center parameter doc (#3341)
- Add Sphinx copybutton (#3530)
- Add glossary to the documentation (#3626)
- Add image of retina to our data (#3748)
- Add microaneurysms() to gallery (#3765)
- Better document remove_small_objects behaviour: int vs bool (#2830)
- Linking preserve_range parameter calls to docs (#3109)
- Update the documentation regarding datalocality (#3127)
- Specify conda-forge channel for scikit-image conda install (#3189)
- Turn DOIs into web links in docstrings (#3367)
- Update documentation for regionprops (#3602)
- DOC: Improve the RANSAC gallery example (#3554)
- DOC: “feature.peak_local_max” : explanation of multiple same-intensity peaks returned by the function; added details on `exclude_border` parameter (#3600)

Improvements

- MNT: handle a deprecation warning for np.linspace and floats for the num parameter (#3453)
- TST: numpy empty arrays are not inherently Falsy (#3455)
- handle warning in scipy cdist for unused parameters (#3456)
- MNT: don't use filter_warnings in test suite. (#3459)
- Add doc notes on setting up the build environment (#3472)
- Release the GIL in numerous cython functions (#3490)
- Cython touchups to use float32 and float64 (#3493)
- rank_filters: Change how the bitdepth and max_bin are computed to ensure exact warnings. (#3501)
- Rank: Optimize OTSU filter (#3504)
- Rank - Fix rank entropy and OTSU tests (#3506)
- delay importing pyplot in manual segmentation (#3533)
- Get rid of the requirements-parser dependency (#3534)
- filter warning from `correct_mesh_orientation` in tests (#3549)
- cloudpickle is really a doc dependency, not a core one (#3634)
- optional dependencies on pip (#3645)
- Fewer test warnings in 3.7 (#3687)
- collections.abc nit (#3692)
- Streamlined issue template (#3697)
- Tighten the PR Template (#3701)
- Use language level to 3 in cython for future compatibility (#3707)
- Update ISSUE_TEMPLATE.md with info about numpy and skimage versions (#3730)
- Use relative imports for many cython modules (#3759)
- Pass tests that don't raise floating point exceptions on arm with soft-fp (#3337)

Other improvements

- BUG: Fix greycoprops correlation always returning 1 (#2532)
- Add section on API discovery via `skimage.lookfor` (#2539)
- Speedup 2D warping for affine transformations (#2902)
- Credit Reviewers in Release Notes (#2927)
- Added small galleries in the API (#2940)
- Use skimage gaussian filter to avoid integer rounding artifacts (#2983)
- Remove Python 2 compatibility (#3000)
- Add `rectangle_perimeter` feature to `skimage.draw` (#3069)
- Update installation instructions to reference existing requirements specification (#3113)
- Updated release notes with pre 0.13.1 phase (#3114)

- Release guidelines update (#3115)
- Ensure we are installing with / running on Python 3 (#3119)
- Hide warnings in test_unsharp_mask (#3130)
- Process 0.15 deprecations (#3132)
- Documentation: always use dev branch javascript (#3136)
- Add initial airspeed velocity (asv) framework (#3137)
- Suppress warnings for flatten during io testing (#3143)
- Recover from exceptions in filters.try_all_threshold() (#3149)
- Fix skimage.test() to run the unittests (#3152)
- skivi: Use qtpy to handle different Qt versions (#3157)
- Refactor python version checking. (#3160)
- Move data_dir to within data/__init__.py (#3161)
- Move the definition of lookfor out of __init__.py (#3162)
- Normalize the package number to PEP440 (#3163)
- Remove skimage.test as it was never used. (#3164)
- Added a message about qtpy to the INSTALL.rst (#3168)
- Regression fix: Travis should fail if tests fail (#3170)
- Set minimum cython version to 0.23.4 (#3171)
- Add rgba2rgb to API docs (#3175)
- Minor doc formatting fixes in video.rst (#3176)
- Decrease the verbosity of the testing (#3182)
- Speedup rgb2gray using matrix multiply (#3187)
- Add instructions for meeseeksdev to PR template (#3194)
- Remove installation instructions for video packages (#3197)
- Big image labeling fix (#3202)
- Handle dask deprecation in cycle_spin (#3205)
- Fix Qt viewer painttool indexing (#3210)
- build_versions.py is no longer hard coded. (#3211)
- Remove dtype constructor call in exposure.rescale_intensity (#3213)
- Various updates to the ASV benchmarks (#3215)
- Add a link to stack overflow on github README (#3217)
- MAINT: remove encoding information in file headers (python 3) (#3219)
- Build tools: Dedicate a –pre build in appveyor and ensure other builds don't download –pre (#3222)
- Fix the human readable error message on a bad build. (#3223)
- Respect input array type in apply_parallel by default (#3225)
- Travis cleanup pip commands (#3227)

- Add benchmarks for morphology.watershed (#3234)
- Correcte docstring formatting so that code block is displayed as code (#3236)
- Defer skimage.io import of matplotlib.pyplot until needed (#3243)
- Add benchmark for Sobel filters (#3249)
- Remove cython md5 hashing since it breaks the build process (#3254)
- Fix typo in documentation. (#3262)
- Issue 3156: skimage/_init__.py Update docstring and fix import * (#3265)
- Object detector module (#3267)
- Do not import submodules while building (#3270)
- Add benchmark suite for canny (#3271)
- improve segmentation.felzenszwalb document #3264 (#3272)
- Update _canny.py (#3276)
- Add benchmark suite for histogram equalization (#3285)
- fix link to equalist_hist blog reference (#3287)
- .gitignore: novice: Ignore save-demo.jpg (#3289)
- Guide the user of denoise_wavelet to choose an orthogonal wavelet. (#3290)
- Remove unused lib in skimage/_init__.py (#3291)
- BUILD: Add pyproject.toml to ensure cython is present (#3295)
- Handle intersphinx and mpl deprecation warnings in docs (#3300)
- Minor PEP8 fixes (#3305)
- cython: check for presence of cpp files during install from sdist (#3311)
- appveyor: don't upload any artifacts (#3315)
- Add benchmark suite for hough_line() (#3319)
- Novice skip url test (#3320)
- Remove benchmarks from wheel (#3321)
- Add license file to the wheel (binary) distribution (#3322)
- codecov: ignore build scripts in coverage and don't comment on PRs (#3326)
- Matplotlib 2.2.3 + PyQt5.11 (#3345)
- Allow @hmaarrfk to mention MeeseeksDev to backport. (#3357)
- Add Python 3.7 to the test matrix (#3359)
- Fix deprecated keyword from dask (#3366)
- Incompatible modes with anti-aliasing in skimage.transform.resize (#3368)
- Missing eval parameter in threshold_local (#3370)
- Avoid Sphinx 1.7.8 (#3381)
- Show our data in the gallery (#3388)
- Minor updates to grammar in numpy images page (#3389)

- assert_all_close doesn't exist, make it `assert_array_equal` (#3391)
- Better behavior of Gaussian filter for arrays with a large number of dimensions (#3394)
- Allow import/execution with -OO (#3398)
- Mark tests known to fail on 32bit architectures with xfail (#3399)
- Hardcode the inputs to `test_ssim_grad` (#3403)
- TST: make `test_wavelet_denoising_levels` compatible with PyWavelets 1.0 (#3406)
- Allow `tifffile.py` to handle I/O. (#3409)
- Add explicit Trove classifier for Python 3 (#3415)
- Fix error in `contribs.py` (#3418)
- MAINT: remove pyside restriction since we don't support Python 3.4 anymore (#3421)
- Build tools: simplify how `MPL_DIR` is obtained. (#3422)
- Build tools: Don't run tests twice in travis. (#3423)
- Build tools: Add an OSX build with optional dependencies. (#3424)
- MAINT: Reverted the changes in #3300 that broke the `MINIMUM_REQUIREMENTS` tests (#3427)
- MNT: Convert links using http to https (#3428)
- MAINT: Use upstream colormaps now that matplotlib has been upgraded (#3429)
- Build tools: Make `pyamg` an optional dependency and remove custom logic (#3431)
- Build tools: Fix PyQt installed in minimum requirements build (#3432)
- MNT: multiprocessing should always be available since we depend on python >=2.7 (#3434)
- MAINT Use `np.full` instead of `cst*np.ones` (#3440)
- DOC: Fix LaTeX build via `make latexpdf` (#3441)
- Update instructions et al for releases after 0.14.1 (#3442)
- Remove code specific to python 2 (#3443)
- Fix default value of `methods` in `_try_all` to avoid exception (#3444)
- Fix `morphology.local_maxima` for input with any dimension < 3 (#3447)
- Use raw strings to avoid unknown escape symbol warnings (#3450)
- Speed up `xyz2rgb` by clipping output in place (#3451)
- MNT; handle deprecation warnings in `tifffile` (#3452)
- Build tools: TST: filter away novice deprecation warnings during testing (#3454)
- Build tools: don't use the `pytest.fixtures` decorator anymore in class fixtures (#3458)
- Preserving the `fill_value` of a masked array (#3461)
- Fix `VisibleDeprecationWarning` from `np.histogram`, `normed=True` (#3463)
- Build Tools: DOC: Document that now `PYTHONOPTIMIZE` build is blocked by SciPy (#3470)
- DOC: Replace broken links by webarchive equivalent links (#3471)
- FIX: making the `plot_marching_cubes` example visible. (#3474)
- Avoid Travis failure regarding `skimage.lookfor` (#3477)

- Fix Python executable for sphinx-build in docs Makefile (#3478)
- Build Tools: Block specific Cython versions (#3479)
- Fix typos (#3480)
- Add “optional” indications to docstrings (#3495)
- Rename ‘mnxc’ (masked normalize cross-correlation) to something more descriptive (#3497)
- Random walker bug fix: no error should be raised when there is nothing to do (#3500)
- Various minor edits for active contour (#3508)
- Fix range for uint32 dtype in user guide (#3512)
- Raise meaningful exception in warping when image is empty (#3518)
- DOC: Development installation instructions for Ubuntu are missing tkinter (#3520)
- Better gallery examples and tests for masked translation registration (#3528)
- DOC: make more docstrings compliant with our standards (#3529)
- Build tools: Remove restriction on simpleitk for python 3.7 (#3535)
- Speedup and add benchmark for `skeletonize_3d` (#3536)
- Update requirements/README.md on justification of matplotlib 3.0.0 in favor of #3476 (#3542)
- Doc enhancements around denoising features. (#3553)
- Use ‘getconf _NPROCESSORS_ONLN’ as fallback for nproc in Makefile of docs (#3563)
- Fix matplotlib set_*lim API deprecations (#3564)
- Switched from np.power to np.cbrt (#3570)
- Filtered out DeprecationWarning for matrix subclass (#3572)
- Add RGB to grayscale example to gallery (#3574)
- Build tools: Refactor check_sdist so that it takes a filename as a parameter (#3579)
- Turn dask to an optional requirement (#3582)
- `_marching_cubes_lewiner_cy`: mark char as signed (#3587)
- Hyperlink DOIs to preferred resolver (#3589)
- Missing parameter description in `morphology.reconstruction` docstring #3581 (#3591)
- Update chat location (#3598)
- Remove orphan code (`skimage/filters/_ctmf.pyx`). (#3601)
- More explicit example title, better list rendering in `plot_cycle_spinning.py` (#3606)
- Add rgb to hsv example in the gallery (#3607)
- Update documentation of `perimeter` and add input validation (#3608)
- Additional mask option to `clear_border` (#3610)
- Set up CI with Azure Pipelines (#3612)
- [MRG] EHN: median filters will accept floating image (#3616)
- Update Travis-CI to xcode 10.1 (#3617)
- Minor tweaks to `_mean_std` code (#3619)

- Add explicit ordering of gallery sections (#3627)
- Delete broken links (#3628)
- Build tools: Fix test_mpl_imshow for matplotlib 2.2.3 and numpy 1.16 (#3635)
- First draft of core dev guide (#3636)
- Add more details about the home page build process (#3639)
- Ensure images resources with long querystrings can be read (#3642)
- Delay matplotlib import in skimage/future/manual_segmentation.py (#3648)
- make the low contrast check optional when saving images (#3653)
- Correctly ignore release notes auto-generated for docs (#3656)
- Remove MANIFEST file when making the ‘clean’ target (#3657)
- Clarify return values in _overlap docstrings in feature/blob.py (#3660)
- Contribution script: allow specification of GitHub development branch (#3661)
- Update core dev guide: deprecation, contributor guide, required experience (#3662)
- Add release notes for 0.14.2 (#3664)
- FIX gallery: Add multichannel=True to match_histogram (#3672)
- MAINT Minor code style improvements (#3673)
- Pass parameters through tifffile plugin (#3675)
- DOC unused im3d_t in example (#3677)
- Remove wrong cast of Py_ssize_t to int (#3682)
- Build tools: allow python 3.7 to fail, but travis to continue (#3683)
- Build tools: remove pyproject.toml (#3688)
- Fix ValueError: not enough values to unpack (#3703)
- Several fixes for heap.pyx (#3704)
- Enable the faulthandler module during testing (#3708)
- Build tools: Fix Python 3.7 builds on travis (#3709)
- Replace np.einsum with np.tensordot in _upsampled_dft (#3710)
- Fix potential use of NULL pointers (#3717)
- Fix potential memory leak (#3718)
- Fix potential use of NULL pointers (#3719)
- Fix and improve core_cy.pyx (#3720)
- Build tools: Downgrade Xcode to 9.4 on master (#3723)
- Improve visual_test.py (#3732)
- Updated painttool to work with color images and properly scale labels. (#3733)
- Add image.sc forum badge to README (#3738)
- Block PyQt 5.12.0 on Travis (#3743)
- Build tools: Fix matplotlib + qt 5.12 the same way upstream does it (#3744)

- gallery: remove xx or yy sorted directory names (#3761)
- Allow for f-contiguous 2D arrays in convex_hull_image (#3762)
- Build tools: Set astropy minimum requirement to 1.2 to help the CIs. (#3767)
- Avoid NumPy warning while stacking arrays. (#3768)
- Set CC0 for microaneurysms (#3778)
- Unify LICENSE files for easier interpretation (#3791)
- Readme: Remove expectation for future fix from matplotlib (#3794)
- Improved documentation/test in flood() (#3796)
- Use ssize_t in denoise cython (#3800)
- Removed non-existent parameter in docstring (#3803)
- Remove redundant point in draw.polygon docstring example (#3806)
- Ensure watershed auto-markers respect mask (#3809)

75 authors added to this release [alphabetical by first name or login]

- Abhishek Arya
- Adrian Roth
- alexis-cvetkov (Alexis Cvetkov-Iliev)
- Ambrose J Carr
- Arthur Imbert
- blochl (Leonid Bloch)
- Brian Smith
- Casper da Costa-Luis
- Christian Rauch
- Christoph Deil
- Christoph Gohlke
- Constantin Pape
- David Breuer
- Egor Panfilov
- Emmanuelle Gouillart
- fivemok
- François Boulogne
- François Cokelaer
- François-Michel De Rainville
- Genevieve Buckley
- Gregory R. Lee
- Gregory Starck

- Guillaume Lemaitre
- Hugo
- jakirkham (John Kirkham)
- Jan
- Jan Eglinger
- Jathrone
- Jeremy Metz
- Jesse Pangburn
- Johannes Schönberger
- Jonathan J. Helmus
- Josh Warner
- Jotham Apaloo
- Juan Nunez-Iglesias
- Justin
- Katrin Leinweber
- Kim Newell
- Kira Evans
- Kirill Klimov
- Lars Grueter
- Laurent P. René de Cotret
- Legodev
- mamrehn
- Marcel Beining
- Mark Harfouche
- Matt McCormick
- Matthias Bussonnier
- mrastgoo
- Nehal J Wani
- Nelle Varoquaux
- Onomatopeia
- Oscar Javier Hernandez
- Page-David
- PeterJackNaylor
- PinkFloyded
- R S Nikhil Krishna
- ratijas

- Rob
- robroooh
- Roman Yurchak
- Sarkis Dallakian
- Scott Staniewicz
- Sean Budd
- shcrela
- Stefan van der Walt
- Taylor D. Scott
- Thein Oo
- Thomas Walter
- Tom Augspurger
- Tommy Löfstedt
- Tony Tung
- Vilim Štih
- yangfl
- Zhanwen “Phil” Chen

46 reviewers added to this release [alphabetical by first name or login]

- Abhishek Arya
- Adrian Roth
- Alexandre de Siqueira
- Ambrose J Carr
- Arthur Imbert
- Brian Smith
- Christian Rauch
- Christoph Gohlke
- David Breuer
- Egor Panfilov
- Emmanuelle Gouillart
- Evan Putra Limanto
- François Boulogne
- François Cokelaer
- Gregory R. Lee
- Grégory Starck
- Guillaume Lemaitre

- Ilya Flyamer
- jakirkham
- Jarrod Millman
- Johannes Schönberger
- Josh Warner
- Jotham Apaloo
- Juan Nunez-Iglesias
- Justin
- Lars Grueter
- Laurent P. René de Cotret
- Marcel Beining
- Mark Harfouche
- Matthew Brett
- Matthew Rocklin
- Matti Picus
- mrastgoo
- Onomatopeia
- PeterJackNaylor
- Rob
- Roman Yurchak
- Scott Staniewicz
- Stefan van der Walt
- Thein Oo
- Thomas A Caswell
- Thomas Walter
- Tom Augspurger
- Tomas Kazmar
- Tommy Löfstedt
- Vilim Štih

1.4.24 scikit-image 0.14.4 release notes

We're happy to announce the release of scikit-image v0.14.4!

As a reminder, 0.14.x is the final version of scikit-image with support for Python 2.7, and will receive critical bug fixes until Jan 1, 2020. If you are using Python 3.5 or later, you should upgrade to scikit-image 0.15.x.

This is a bugfix release, and contains the following changes from v0.14.3:

Bug Fixes

- Fix float32 support in denoise_bilateral and denoise_tv_bregman (#3937)
- Fixup test for RANSAC: don't pick duplicate samples #3901 (#3916)

Other Pull Requests

- Backport PR #3943 on branch v0.14.x (Update the joblib link in tutorial_parallelization.rst) (#3944)

3 authors added to this release [alphabetical by first name or login]

- Alexandre de Siqueira
- Juan Nunez-Iglesias
- Mark Harfouche

1.4.25 scikit-image 0.14.3 release notes

As a reminder, 0.14.x is the final version of scikit-image with support for Python 2.7, and will receive critical bug fixes until Jan 1, 2020. If you are using Python 3.5 or later, you should upgrade to scikit-image 0.15.x.

This is a bugfix release, and contains the following changes from v0.14.2:

API Changes

- morphology.local_maxima now returns a boolean array instead of uint8 (#3749, #3752)

Bug Fixes

- _marching_cubes_lewiner_cy: mark char as signed (#3587, #3678)
- Fix potential use of NULL pointer (#3696)
- pypi: explicitly exclude Python 3.1, 3.2, and 3.3 (#3726)
- Reduce default tolerance in threshold_li (#3622) (#3781)
- Denoising functions now accept float32 images (#3449) (#3486) (#3880)

Other Pull Requests

- BLD: pin cython's language_level (#3716)
- Build tools: Upgrade xcode to 9.4 on v0.14.x branch (#3724)
- Get rid of the requirements-parser dependency (#3534, #3727)
- Add small galleries in the API (#2940, #3728)
- Correctly ignore release notes auto-generated for docs (#3656, #3737)
- Fix qt 5.12 pinning for 0.14.x branch. (#3744, #3753)
- Minor fixes to documentation and testing infrastructure - backports #3870 and #3869 (#3881)
- Set astropy minimum requirement to 1.2 to help the CIs. (#3767, #3770)
- Avoid NumPy warning while stacking arrays. (#3768, #3771)
- Fix human readable error message on a bad build. (#3223, #3790)
- Unify LICENSE files for easier interpretation (#3791, #3792)
- Documentation formatting / compilation fixes - Backport of #3838 to v0.14.x (#3885)
- Fix build by using latest wheel package (scikit-image/scikit-image-wheels#10)

12 authors added to this release [alphabetical by first name]

- Andrew Murray
- Christoph Gohlke
- Egor Panfilov
- François Boulogne
- Johannes Schönberger
- Juan Nunez-Iglesias
- Lars Grueter
- Mark Harfouche
- Matthew Bowden
- Nehal J Wani
- Nelle Varoquaux
- Stefan van der Walt
- Thomas A Caswell

... and, as always, a special mention to Matthias Bussonnier's Meeseeks Box, which remains invaluable for our backports.

4 committers added to this release [alphabetical by first name or login]

- Josh Warner
- Juan Nunez-Iglesias
- Mark Harfouche
- Stefan van der Walt

5 reviewers added to this release [alphabetical by first name or login]

- Egor Panfilov
- François Boulogne
- Juan Nunez-Iglesias
- Mark Harfouche
- Stefan van der Walt

1.4.26 scikit-image 0.14.2 release notes

This release handles an incompatibility between scikit-image and NumPy 1.16.0, released on January 13th 2019.

It contains the following changes from 0.14.1:

API changes

- `skimage.measure.regionprops` no longer removes singleton dimensions from label images (#3284). To recover the old behavior, replace `regionprops(label_image)` calls with `regionprops(np.squeeze(label_image))`

Bug fixes

- Address deprecation of NumPy `_validate_lengths` (backport of #3556)
- Correctly handle the maximum number of lines in Hough transforms (backport of #3514)
- Correctly implement early stopping criterion for rank kernel noise filter (backport of #3503)
- Fix `skimage.measure.regionprops` for 1x1 inputs (backport of #3284)

Enhancements

- Rewrite of `local_maxima` with flood-fill (backport of #3022, #3447)

Build Process & Testing

- Dedicate a --pre build in appveyor (backport of #3222)
- Avoid Travis-CI failure regarding `skimage.lookfor` (backport of #3477)
- Stop using the `pytest.fixtures` decorator (#3558)
- Filter out DeprecationPendingWarning for matrix subclass (#3637)
- Fix matplotlib test warnings and circular import (#3632)

Contributors & Reviewers

- François Boulogne
- Emmanuelle Gouillart
- Lars Grüter
- Mark Harfouche
- Juan Nunez-Iglesias
- Egor Panfilov
- Stefan van der Walt

1.4.27 scikit-image 0.14.1 release notes

We're happy to announce the release of scikit-image v0.14.1!

scikit-image is an image processing toolbox for SciPy that includes algorithms for segmentation, geometric transformations, color space manipulation, analysis, filtering, morphology, feature detection, and more.

This is our first release under our Long Term Support for 0.14 policy. As a reminder, 0.14 is the last release to support Python 2.7, but it will be updated with bug fixes and popular features until January 1st, 2020.

This release contains the following changes from 0.14.0:

Bug fixes

- `skimage.color.adapt_rgb` was applying input functions to the wrong axis (#3097)
- `CollectionViewer` now indexes correctly (it had been broken by an update to NumPy indexing) (#3288)
- Handle deprecated indexing-by-list and NumPy `matrix` from NumPy 1.15 (#3238, #3242, #3292)
- Fix incorrect inertia tensor calculation (#3303) (Special thanks to JP Cornil for reporting this bug and for their patient help with this fix)
- Fix missing comma in `__all__` listing of `moments_coord_central`, so it and `moments_normalized` can now be correctly imported from the `measure` namespace (#3374)
- Fix background color in `label2rgb(..., kind='avg')` (#3280)

Enhancements

- “Reflect” mode in transforms now works fine when an image dimension has size 1 (#3174)
- `img_as_float` now allows single-precision (32-bit) float arrays to pass through unmodified, rather than being up-converted to 64-bit (#3110, #3052, #3391)
- Speed up `rgb2gray` computation (#3187)
- The scikit-image viewer now works with different PyQt versions (#3157)
- The `cycle_spin` function for enhanced denoising works single-threaded when dask is not installed now (#3218)
- scikit-image’s `io` module will no longer inadvertently set the matplotlib backend when imported (#3243)
- Fix deprecated `get` keyword from dask in favor of `scheduler` (#3366)
- Add missing `cval` parameter to `threshold_local` (#3370)

API changes

- Remove deprecated `dynamic_range` in `measure.compare_psnr` (#3313)

Documentation

- Improve the documentation on data locality (#3127)
- Improve the documentation on dealing with video (#3176)
- Update broken link for Canny filter documentation (#3276)
- Fix incorrect documentation for the `center` parameter of `skimage.transform.rotate` (#3341)
- Fix incorrect formatting of docstring in `measure.profile_line` (#3236)

Build process / development

- Ensure Cython is 0.23.4 or newer (#3171)
- Suppress warnings during testing (#3143)
- Fix `skimage.test` (#3152)
- Don’t upload artifacts to AppVeyor (there is no way to delete them) (#3315)
- Remove `import *` from the scikit-image package root (#3265)
- Allow named non-core contributors to issue MeeseeksDev commands (#3357, #3358)
- Add testing in Python 3.7 (#3359)
- Add license file to the binary distribution (#3322)
- `lookfor` is no longer defined in `__init__.py` but rather imported to it (#3162)
- Add `pyproject.toml` to ensure Cython is present before building (#3295)
- Add explicit Python version Trove classifiers for PyPI (#3417)
- Ignore known test failures in 32-bit releases, allowing 32-bit wheel builds (#3434)
- Ignore failure to raise floating point warnings on certain ARM platforms (#3337)
- Fix tests to be compatible with PyWavelets 1.0 (#3406)

Credits

Made with commits from (alphabetical by last name):

- François Boulogne
- Genevieve Buckley
- Sean Budd
- Matthias Bussonnier
- Sarkis Dallakian
- Christoph Deil
- François-Michel De Rainville
- Emmanuelle Gouillart
- Yaroslav Halchenko
- Mark Harfouche
- Jonathan Helmus
- Gregory Lee
- @Legodev
- Matt McCormick
- Juan Nunez-Iglesias
- Egor Panfilov
- Jesse Pangburn
- Johannes Schönberger
- Stefan van der Walt

Reviewed by (alphabetical by last name):

- François Boulogne
- Emmanuelle Gouillart
- Mark Harfouche
- Juan Nunez-Iglesias
- Egor Panfilov
- Stéfan van der Walt
- Josh Warner

And with the special support of [MeeseeksDev](<https://github.com/MeeseeksBox>), created by Matthias Bussonnier

1.4.28 scikit-image 0.14.0 release notes

We're happy to announce the release of scikit-image v0.14.0!

scikit-image is an image processing toolbox for SciPy that includes algorithms for segmentation, geometric transformations, color space manipulation, analysis, filtering, morphology, feature detection, and more.

This is the last major release with official support for Python 2.7. Future releases will be developed using Python 3-only syntax.

However, 0.14 is a long-term support (LTS) release and will receive bug fixes and backported features deemed important (by community demand) until January 1st 2020 (end of maintenance for Python 2.7; see PEP 373 for details).

For more information, examples, and documentation, please visit our website:

<http://scikit-image.org>

New Features

- Lookfor function to search across the library: `skimage.lookfor`. (#2713)
- nD support for `skimage.transform.rescale`, `skimage.transform.resize`, and `skimage.transform.pyramid_*` transforms. (#1522)
- Chan-Vese segmentation algorithm. (#1957)
- Manual segmentation with `matplotlib` for fast data annotation: `skimage.future.manual_polygon_segmentation`, `skimage.future.manual_lasso_segmentation`. (#2584)
- Hysteresis thresholding: `skimage.filters.apply_hysteresis_threshold`. (#2665)
- Segmentation with morphological snakes: `skimage.segmentation.morphological_chan_vese` (2D), `skimage.segmentation.morphological_geodesic_active_contour` (2D and 3D). (#2791)
- nD support for image moments: `skimage.measure.moments_central`, `skimage.measure.moments_central`, `skimage.measure.moments_normalized`, `skimage.measure.moments_hu`. This change leads to 3D/nD compatibility for many regionprops. (#2603)
- Image moments from coordinate input: `skimage.measure.moments_coords`, `skimage.measure.moments_coords_central`. (#2859)
- Added 3D support to `blob_dog` and `blob_log`. (#2854)
- Inertia tensor and its eigenvalues can now be computed outside of regionprops; available in `skimage.measure.inertia_tensor`. (#2603)
- Cycle-spinning function for approximating shift-invariance by averaging results from a series of spatial shifts: `skimage.restoration.cycle_spin`. (#2647)
- Haar-like feature: `skimage.feature.haar_like_feature`, `skimage.feature.haar_like_feature_coord`, `skimage.feature.draw_haar_like_feature`. (#2848)
- Data generation with `random_shapes` function: `skimage.draw.random_shapes`. (#2773)
- Subset of LFW (Labeled Faces in the Wild) database: `skimage.data.cbcl_face_database`. (#2905)
- Fully reworked montage function (now with a better padding behavior): `skimage.util.montage`. (#2626)
- YDbDr colorspace conversion routines: `skimage.color.rgb2ydbdr`, `skimage.color.ydbdr2rgb`. (#3018)

Improvements

- VisuShrink method for `skimage.restoration.denoise_wavelet`. (#2470)
- New `max_ratio` parameter for `skimage.feature.match_descriptors`. (#2472)
- `skimage.transform.resize` and `skimage.transform.rescale` have a new `anti_aliasing` option to avoid aliasing artifacts when down-sampling images. (#2802)
- Support for multichannel images for `skimage.feature.hog`. (#2870)
- Non-local means denoising (`skimage.restoration.denoise_nl_means`) has a new optional parameter, `sigma`, that can be used to specify the noise standard deviation. This enables noise-robust patch distance estimation. (#2890)
- Mixed dtypes support for `skimage.measure.compare_ssim`, `skimage.measure.compare_psnr`, etc. (#2893)
- New `alignment` parameter in `skimage.feature.plot_matches`. (#2955)
- New `seed` parameter in `skimage.transform.probabilistic_hough_line`. (#2960)
- Various performance improvements. (#2821, #2878, #2967, #3035, #3056, #3100)

Bugfixes

- Fixed `skimage.measure.regionprops.bbox_area` returning incorrect value. (#2837)
- Changed gradient and L2-Hys norm computation in `skimage.feature.hog` to closely follow the paper. (#2864)
- Fixed `skimage.color.convert_colorspace` not working for YCbCr, YPbPr. (#2780)
- Fixed incorrect composition of projective transformation with inverse transformation. (#2826)
- Fixed bug in random walker appearing when seed pixels are isolated inside pruned zones. (#2946)
- Fixed `rescale` not working properly with different rescale factors in multichannel case. (#2959)
- Fixed float and integer dtype support in `skimage.util.invert`. (#3030)
- Fixed `skimage.measure.find_contours` raising `StopIteration` on Python 3.7. (#3038)
- Fixed platform-specific issues appearing in Windows and/or 32-bit environments. (#2867, #3033)

API Changes

- `skimage.util.montage`. namespace has been removed, and `skimage.util.montage2d` function is now available as `skimage.util.montage2d`.
- `skimage.morphology.binary_erosion` now uses `True` as border value, and is now consistent with `skimage.morphology.erosion`.

Deprecations

- `freeimage` plugin has been removed from `skimage.io`.
- `skimage.util.montage2d` is deprecated and will be removed in 0.15. Use `skimage.util.montage` function instead.
- `skimage.novice` is deprecated and will be removed in 0.16.
- `skimage.transform.resize` and `skimage.transform.rescale` have a new `anti_aliasing` option that avoids aliasing artifacts when down-sampling images. This option will be enabled by default in 0.15.
- `regionprops` will use row-column coordinates in 0.16. You can start using them now with `regionprops(..., coordinates='rc')`. You can silence warning messages, and retain the old behavior, with `regionprops(..., coordinates='xy')`. However, that option will go away in 0.16 and result in an error. This change has a number of consequences. Specifically, the “orientation” region property will measure the anticlockwise angle from a *vertical* line, i.e. from the vector $(1, 0)$ in row-column coordinates.
- `skimage.morphology.remove_small_holes` `min_size` argument is deprecated and will be removed in 0.16. Use `area_threshold` instead.

Contributors to this release

- Alvin
- Norman Barker
- Brad Bazemore
- Leonid Bloch
- Benedikt Boecking
- Jirka Borovec
- François Boulogne
- Larry Bradley
- Robert Bradshaw
- Matthew Brett
- Floris van Breugel
- Alex Chum
- Yannick Copin
- Nethanel Elzas
- Kira Evans
- Christoph Gohlke
- GGoussar
- Jens Glaser
- Peter Goldsborough
- Emmanuelle Gouillart
- Ben Hadfield
- Mark Harfouche

- Scott Heatwole
- Gregory R. Lee
- Guillaume Lemaitre
- Theodore Lindsay
- Kevin Mader
- Jarrod Millman
- Vinicius Monego
- Pradyumna Narayana
- Juan Nunez-Iglesias
- Kesavan PS
- Egor Panfilov
- Oleksandr Pavlyk
- Justin Pinkney
- Robert Pollak
- Jonathan Reich
- Émile Robitaille
- Rose Zhao
- Alex Rothberg
- Arka Sadhu
- Max Schambach
- Johannes Schönberger
- Sourav Singh
- Kesavan Subburam
- Matt Swain
- Saurav R. Tuladhar
- Nelle Varoquaux
- Viraj
- David Volgyes
- Stefan van der Walt
- Thomas Walter
- Scott Warchal
- Josh Warner
- Nicholas Weir
- Sera Yang
- Chiang, Yi-Yo
- corrado9999

- ed1d1a8d
- eepaillard
- leaprovenzano
- mikigom
- mrastgoo
- mutterer
- pmneila
- timhok
- zhongzyd

We'd also like to thank all the people who contributed their time to perform the reviews:

- Leonid Bloch
- Jirka Borovec
- François Boulogne
- Matthew Brett
- Thomas A Caswell
- Kira Evans
- Peter Goldsborough
- Emmanuelle Gouillart
- Almar Klein
- Gregory R. Lee
- Joan Massich
- Juan Nunez-Iglesias
- Faraz Oloumi
- Daniil Pakhomov
- Egor Panfilov
- Dan Schult
- Johannes Schönberger
- Steven Sylvester
- Alexandre de Siqueira
- Nelle Varoquaux
- Stefan van der Walt
- Josh Warner
- Eric Wieser

Full list of changes

This release is the result of 14 months of work. It contains the following 186 merged pull requests by 67 committers:

- n-dimensional rescale, resize, and pyramid transforms (#1522)
- Segmentation: Implementation of a simple Chan-Vese Algorithm (#1957)
- JPEG quality argument in imsave (#2063)
- improve geometric models fitting (line, circle) using LSM (#2433)
- Improve input parameter handling in `_sift_read` (#2452)
- Remove broken test in `_shared/tests/test_interpolation.py` (#2454)
- [MRG] Pytest migration (#2468)
- Add VisuShrink method for `denoise_wavelet` (#2470)
- Ratio test for descriptor matching (#2472)
- Make HOG visualization use midpoints of orientation bins (#2525)
- DOC: Add example for rescaling/resizing/downscaling (#2560)
- Gallery random walker: Rescale image range to -1, 1 (#2575)
- Update conditional requirement for PySide (#2578)
- Add configuration file for `pep8_speaks` (#2579)
- Manual segmentation tool with matplotlib (#2584)
- Website updates (documentation build) (#2585)
- Update the release process notes (#2593)
- Defer matplotlib imports (#2596)
- Spelling: replaces colour by color (#2598)
- Add nD support to image moments computation (#2603)
- Set xlim and ylim in rescale gallery example (#2606)
- Reduce runtime of local_maxima gallery example (#2608)
- MAINT `_shared.testing` now contains pytest's useful functions (#2614)
- error message misspelled, integral to integer (#2615)
- Respect standard notations for images in functions arguments (#2617)
- MAINT: remove unused argument in private inpainting function (#2618)
- MAINT: some minor edits on Chan Vese segmentation (#2619)
- Fix UserWarning: Unknown section Example (#2620)
- Eliminate some TODOs for 0.14 (#2621)
- Clean up and fix bug in ssim tests (#2622)
- Add padding_width to montage2d and add montage_rgb (#2626)
- Add tests covering erroneous input to morphology.watershed (#2631)
- Fix name of code coverage tool (#2638)
- MAINT: Remove undefined attributes in skimage.filters (#2643)

- Improve the support for 1D images in `color.gray2rgb` (#2645)
- ENH: add cycle spinning routine (#2647)
- `as_gray` replaces `as_grey` in `imread()` and `load()` (#2652)
- Fix AppVeyor pytest execution (#2658)
- More TODOs for 0.14 (#2659)
- pin sphinx to <1.6 (#2662)
- MAINT: use relative imports instead of absolute ones (#2664)
- Add hysteresis thresholding function (#2665)
- Improve hysteresis docstring (#2669)
- Add helper functions `img_as_float32` and `img_as_float64` (#2673)
- Remove unnecessary assignment in pxd file. (#2683)
- Unused var and function call in documentation example (#2684)
- Make `imshow_collection` to plot images on a grid of convenient aspect ratio (#2689)
- Fix typo in Chan-Vese docstrings (#2692)
- Fix data type error with `marching_cubes_lewiner(allow_degenerate=False)` (#2694)
- Add handling for uniform arrays when finding local extrema. (#2699)
- Avoid unnecessary copies in `skimage.morphology.label` (#2701)
- Deprecate `visualise` in favor of `visualize` in `skimage.feature.hog` (#2705)
- Remove alpha channel when saving to jpg format (#2706)
- Tweak in-place installation instructions (#2712)
- Add `skimage.lookfor` function (#2713)
- Speedup image dtype conversion by switching to `asarray` (#2715)
- MAINT reorganizing CI-related scripts (#2718)
- added rect function to draw module (#2719)
- Remove duplicate parameter in `skimage.io.imread` docstring (#2725)
- Add support for 1D arrays for grey erosion (#2727)
- Build with Xcode 9 beta 3, MacOS 10.12 (#2730)
- Travis docs one platform (#2732)
- Install documentation build requirements on Travis-CI (#2737)
- Add reference papers for `restoration.inpaint_biharmonic` (#2738)
- Completely remove `freeimage` plugin from `skimage.io` (#2744)
- Implementation and test fix for shannon_entropy calculation. (#2749)
- Minor cleanup (#2750)
- Add notes on testing to `CONTRIBUTING` (#2751)
- Update OSX install script (#2752)
- fix bug in horizontal seam_carve and seam_carve test. issue :#2545 (#2754)

- Recommend merging instead of rebasing, to lower contribution barrier (#2757)
- updated second link, first link still has paywall (#2768)
- DOC: set_color docstring, in-place said explicitly (#2771)
- Add module for generating random, labeled shapes (#2773)
- Ignore known failures (#2774)
- Update testdoc (#2775)
- Remove bento support (#2776)
- AppVeyor supports dot-file-style (#2779)
- Fix bug in `color.convert_colorspace` for YCbCr, YPbPr (#2780)
- Reorganizing requirements (#2781)
- WIP: Deal with long running command on travis (#2782)
- Deprecate the novice module (#2742) (#2784)
- Document mentioning deprecations in the release notes (#2785)
- [WIP] FIX Swirl center coordinates are reversed (#2790)
- Implementation of the Morphological Snakes (#2791)
- Merge TASKS.txt with CONTRIBUTING.txt (#2800)
- Add Gaussian filter-based antialiasing to resize (#2802)
- Add morphological snakes to release notes (#2803)
- Return empty array if hough_line_peaks detects nothing (#2805)
- Add W503 to pep8speaks ignore. (#2816)
- Slice PIL palette correctly using extreme image value. (#2818)
- Move INSTALL to top-level (#2819)
- Make simple watershed fast again (#2821)
- The gallery now points to the stable docs (#2822)
- Adapt AppVeyor to use Python.org dist, and remove install script (#2823)
- Remove pytest yield (#2824)
- Bug fix in projective transformation composition with inverse transformation (#2826)
- FIX: add estimate_sigma to __all__ in restoration module (#2829)
- Switch from LaTeX to MathJax in doc build (#2832)
- Docstring fixes for better formula formatting (#2834)
- Fix regionprops.bbox_area bug (#2837)
- MAINT: add Python 3.6 to appveyor, small edits (#2840)
- Allow convex area calculation in 3D for regionprops (#2847)
- [MRG] DOC fix documentation build (#2851)
- Change default args from list to tuple in `feature.draw_multiblock_lbp` (#2852)
- Add 3D support to `blob_dog` and `blob_log` (#2854)

- Update compare_nrmse docstring (#2855)
- Fix link order in example (#2858)
- Add Computation of Image Moments to Coordinates (#2859)
- Revert gradient formula, modify the deprecation warning, and fix L2-Hys norm in `skimage.feature.hog` (#2864)
- OverflowError: Python int too large to convert to C long on win-amd64-py2.7 (#2867)
- Fix `skimage.measure.centroid` and add test coverage (#2869)
- Add multichannel support to `feature.hog` (#2870)
- Remove scipy version check in `active_contour` (#2871)
- Update DOI reference in `measure.compare_ssim` (#2872)
- Fix randomness and expected ranges for RGB in `test_random_shapes`. (#2877)
- NL means fixes for large datasets (#2878)
- Make `test_random_shapes` use internally shipped testing tools (#2879)
- DOC: Update docstring for `is_low_contrast` to match function signature (#2883)
- Update URL in RAG docstring (#2885)
- Fix spelling typo in NL means docstring (#2887)
- noise-robust patch distance estimation for non-local means (#2890)
- Allow mixed dtypes in `compare_ssim`, `compare_psnr`, etc. (#2893)
- EHN add Haar-like feature (#2896)
- Add CBCL face database subset to `skimage.data` (#2897)
- EXA example for haar like features (#2898)
- Install documentation dependencies on all builds (#2900)
- Improve LineModelND doc strings (#2903)
- Add a subset of LFW dataset to `skimage.data` (#2905)
- Update default parameter values in the docstring of `skimage.restoration.unsupervised_wiener` (#2906)
- Revert “Add CBCL face database subset to `skimage.data`” (#2907)
- remove unused parameter ‘n_segments’ in `_enforce_label_connectivity_cython()` (#2908)
- Update six version to make `pytest_cov` work (#2909)
- Fix typos in `draw._random_shapes._generate_triangle_mask` docstring (#2914)
- do not assume 3 channels during non-local means denoising (#2922)
- add missing cdef in `_integral_image_3d` (non-local means) (#2923)
- Replace `morphology.remove_small_holes` argument `min_size` with `area_threshold` (#2924)
- Ensure warning to provide bool array is warranted (#2930)
- Remove copyright notice with permission of the author (Thomas Lewiner) (#2932)
- Fix link to Windows binaries in README. (#2934)
- Handle NumPy 1.14 API changes (#2935)

- Specify *gradient* parameter docstring in *compare_ssim* (#2937)
- Fixed broken link on LBP documentation (#2941)
- Corrected bug related to border value of morphology.binary_erosion (#2945)
- Correct bug in random walker when seed pixels are isolated inside pruned zones (#2946)
- Fix Cython compilation warnings in NL Means and Watershed (#2947)
- Add *alignment* parameter to *feature.plot_matches* (#2955)
- Raise warning when attempting to save boolean image (#2957)
- Allow different rescale factors in multichannel warp (#2959)
- Add seed parameter to probabilistic_hough_line (#2960)
- Minor style fixes for #2946 (#2961)
- Build on fewer AppVeyor platforms to avoid timeout (#2962)
- Watershed segmentation: make usable for large arrays (#2967)
- Mark data_range as being a float (#2971)
- Use correct NumPy version comparison in pytest configuration (#2975)
- Handle matplotlib 2.2 pre-release deprecations (#2977)
- Bugfix LineModelND.residuals does not use the optional parameter *params* (#2979)
- Return empty list on flat images with hough_ellipse #2820 (#2996)
- Add release notes for 0.13.1 (#2999)
- MAINT: PIL removed saving RGBA images as jpeg files (#3004)
- Ensure stdev is always nonnegative in *_mean_std* (#3008)
- Add citation information to README (#3013)
- Add YDbDr colorspace conversion routines (#3018)
- Minor style and documentation updates for #2859 (#3023)
- *draw.random_shapes* API improvements (#3029)
- Type dependent inversion (#3030)
- Fix ValueError: Buffer dtype mismatch, expected ‘int64_t’ but got ‘int’ on win_amd64 (#3033)
- Replace pow function calls in Cython modules to fix performance issues on Windows (#3035)
- Add *__pycache__* and .cache to .gitignore. (#3037)
- Fix RuntimeError: generator raised StopIteration on Python 3.7 (#3038)
- Fix invert tests (#3039)
- Fix examples not displaying figures (#3040)
- Correct reference for the coins sample image (#3042)
- Switch to basis numpy int dtypes in *dtype_range* (#3050)
- speedup img_as_float by making division multiplication and avoiding unnecessary allocation (#3056)
- For sparse CG solver, provide atol=0 keyword for SciPy >= 1.1 (#3063)
- Update dependencies and deprecations to fix Travis builds (#3072)

- Sanitizing marching_cubes_lewiner spacing input argument (#3074)
- Allow convex_hull_image on empty images (#3076)
- v0.13.x: Backport NumPy 1.14 compatibility (#3085)
- Force Appveyor to fail on failed tests (#3093)
- Add `threshold_local` to `filters` module namespace (#3096)
- Replace grey by gray where no deprecation is needed (#3098)
- Optimize `_probabilistic_hough_line` function (#3100)
- Rebuild docs upon deploy to ensure Javascript is generated (#3104)
- Fix random gallery script generation (#3106)

scikit-image 0.13.1 is a bug-fix and compatibility update. See below for the many new features in 0.13.0.

The main contribution in 0.13.1 is Jarrod Millman's valiant work to ensure scikit-image works with both NetworkX 1.11 and 2.0 (#2766). Additional updates include:

- Bug fix in similarity transform estimation, by GitHub user @zhongzyd (#2690)
- Bug fixes in `skimage.util.plot_matches` and `denoise_wavelet`, by Gregory Lee (#2650, #2640)
- Documentation updates by Egor Panfilov (#2716) and Jirka Borovec (#2524)
- Documentation build fixes by Gregory Lee (#2666, #2731), Nelle Varoquaux (#2722), and Stéfan van der Walt (#2723, #2810)

1.4.29 scikit-image 0.13.0 release notes

We're happy to (finally) announce the release of scikit-image v0.13.0!

scikit-image is an image processing toolbox for SciPy that includes algorithms for segmentation, geometric transformations, color space manipulation, analysis, filtering, morphology, feature detection, and more.

For more information, examples, and documentation, please visit our website:

<http://scikit-image.org>

and our gallery of examples

http://scikit-image.org/docs/dev/auto_examples/

Highlights

This release is the result of a year of work, with over 200 pull requests by 82 contributors. Highlights include:

- Improved n-dimensional image support. This release adds nD support to:
 - `regionprops` computation for centroids (#2083)
 - `segmentation.clear_border` (#2087)
 - Hessian matrix (#2194)
- In addition, the following new functions support nD images:
 - new wavelet denoising function, `restoration.denoise_wavelet` (#1833, #2190, #2238, #2240, #2241, #2242, #2462)

- new thresholding functions, `filters.threshold_sauvola` and `filters.threshold_niblack` (#2266, #2441)
- new local maximum, local minimum, hmaxima, hminima functions (#2449)
- Grey level co-occurrence matrix (GLCM) now works with uint16 images
- `filters.try_all_threshold` to rapidly see output of various thresholding methods
- Frangi and Hessian filters (2D only) (#2153)
- New *compact watershed* algorithm in `segmentation.watershed` (#2211)
- New *shape index* algorithm in `feature.shape_index` (#2312)

New functions and features

- Add threshold minimum algorithm (#2104)
- Implement mean and triangle thresholding (#2126)
- Add Frangi and Hessian filters (#2153)
- add bbox_area to region properties (#2187)
- colorconv: Add rgba2rgb() (#2181)
- Lewiner marching cubes algorithm (#2052)
- image inversion (#2199)
- wavelet denoising (from #1833) (#2190)
- routine to estimate the noise standard deviation from an image (#1837)
- Add compact watershed and clean up existing watershed (#2211)
- Added the missing ‘grey2rgb’ function. (#2316)
- Shape index (#2312)
- Fundamental and essential matrix 8-point algorithm (#1357)
- Add YUV, YIQ, YPbPr, YCbCr colorspaces
- Detection of local extrema from morphology (#2449)
- shannon entropy (#2416)

Documentation improvements

- add details about github SSH keys in contributing page (#2073)
- Add example for felzenszwalb image segmentation (#2096)
- Sphinx gallery for example gallery (#2078)
- Improved region boundary RAG docs (#2106)
- Add gallery Lucy-Richardson deconvolution algorithm (#2376)
- Gallery: Use Horse to illustrate Convex Hull (#2431)
- Add working with OpenCV in user guide (#2519)

Code improvements

- Remove lena image from test suite (#1985)
- Remove duplicate mean calculation in skimage.feature.match_template (#1980)
- Add nD support to clear_border (#2087)
- Add uint16 images support for co-occurrence matrix (#2095)
- Add default parameters for Gaussian and median filters (#2151)
- try_all to choose the best threshold algorithm (#2110)
- Add support for multichannel in Felzenszwalb segmentation (#2134)
- Improved SimilarityTransform, new EuclideanTransform class (#2044)
- ENH: Speed up Hessian matrix computation (#2194)
- add n-dimensional support to denoise_wavelet (#2242)
- Speedup inpaint_biharmonic (#2234)
- Update hessian matrix code to include order kwarg (#2327)
- Handle cases for label2rgb where input labels are negative and/or nonconsecutive (#2370)
- Added watershed_line parameter (#2393)

API Changes

- Remove deprecated filter module. Use filters instead. (#2023)
- Remove skimage.filters.canny links. Use feature.canny instead. (#2024)
- Removed Python 2.6 support and related checks (#2033)
- Remove deprecated {h/v}sobel, {h/v}prewitt, {h/v}schar, roberts_{positive/negative} filters (#2159)
- Remove deprecated _mode_deprecations (#2156)
- Remove deprecated None defaults in rescale_intensity (#2161)
- Parameters ntiles_x and ntiles_y have been removed from exposure.equalize_adapthist
- The minimum NumPy version is now 1.11, and the minimum SciPy version is now 0.17

Deprecations

- clip_negative will be set to false by default in version 0.15 (func: dtype_limits) (#2228)
- Deprecate “dynamic_range” in favor of “data_range” (#2384)
- The default value of the circle argument to radon and iradon transforms will be True in 0.15 (#2235)
- The default value of multichannel for denoise_bilateral and denoise_nl_means will be False in 0.15
- The default value of block_norm in feature.hog will be L2-Hysteresis in 0.15.
- The threshold_adaptive function is deprecated. Use threshold_local instead.
- The default value of mode in transform.swirl, resize, and rescale will be “reflect” in 0.15.

Contributors to this release

- AbdealiJK
- Rodrigo Benenson
- Vighnesh Birodkar
- Jirka Borovec
- François Boulogne
- Matthew Brett
- Sarwat Fatima
- Rachel Finck
- Joe Futrelle
- Jeroen Van Goey
- Christoph Gohlke
- Roman Golovanov
- Emmanuelle Gouillart
- Anshita Gupta
- David Haberthür
- Jeff Hemmelgarn
- Hiyorimi
- Daniel Hyams
- Alex Izvorski
- Kyle Jackson
- Jirka
- JohnnyTeutonic
- Kevin Keraudren
- Almar Klein
- Yu Kobayashi
- Moriyoshi Koizumi
- Lachlan
- LachlanD
- George Laurent
- Gregory R. Lee
- Evan Limanto
- Ben Longo
- Victor MARTIN
- Oliver Mader
- Ken'ichi Matsui

- Jeremy Metz
- Jeysen Molina
- Michael Mueller
- Juan Nunez-Iglesias
- Egor Panfilov
- Paul
- PengchengAi
- Francisco de la Peña
- Pavlin Poličar
- Orion Poplawski
- Zoe Richards
- Todd V. Rovito
- Christian Sachs
- Sanya
- Johannes Schönberger
- Pavel Shevchuk
- Scott Sievert
- Steven Sylvester
- Shaun Singh
- Sourav Singh
- Alexandre Fioravante de Siqueira
- Samuel St-Jean
- Noah Stier
- Ole Streicher
- Martin Thoma
- Matěj Týč
- Viraj
- Stefan van der Walt
- Josh Warner
- Olivia Wilson
- Robin Wilson
- Martin Zackrisson
- Yue Zheng
- Nick Zoghb
- alexandrejaguar
- almar

- cespenel
- danielballan
- dmesejo
- eli
- jwittenbach
- lgeorge
- mljli
- rjeli
- skrish13
- tseclaudia
- walter

Pull requests merged in this release

- Warn if user tries to build with older Cython version (#1986)
- Remove lena image from test suite (#1985)
- Add inpaint to module init (#1987)
- Pre-calculate template mean (#1980)
- rgb2grey -> grey2rgb (#1989)
- Also expose rgb2gray as rgb2grey (#1990)
- Remove all .md5 files on clean (#1992)
- avoid deprecation warnings when calling compute_ssim with multichannel=True (#1994)
- DOC: Suggest multichannel=True in compute_ssim error (#1999)
- [DOC] add link to guide (#2001)
- Fix docs->doc in CONTRIBUTING (#2009)
- Turn dask into an optional dependency (#2013)
- Correct regexp for catching mpl warnings (#2014)
- BUILD: Use –pre flag for Travis pip installs. (#1938)
- Github templates (#1954)
- added doc to PaintTool (#1934)
- skimage.segmentation.quickshift signature is missing from API docs (#2017)
- MAINT: Upgrade tifffile (#2016)
- Modified .gitignore to properly ignore auto_example files (#1966)
- MAINT: Switch from coveralls -> codecov in CI build (#2015)
- skimage.segmentation.quickshift signature is missing from API docs, third attempt (#2021)
- MAINT: Remove deprecated filter module (#2023)
- Remove skimage.filters.canny links (#2024)

- Document regionprops bbox property. (#2030)
- Fix URL to texturematch paper (#2031)
- Improved skimage.segmentation.active_contour input arguments' dtype support (#2032)
- Fix local test function (#2034)
- Removed Python 2.6 support and related checks (#2033)
- Test on OSX (#2038)
- Change coverage badge to codecov (#2055)
- TST: Speed up bilateral filter tests (#2061)
- Speed up colorconv._convert (#2064)
- FIX: Fix import of ‘warn’ in qt_plugin (#2070)
- Add YUV, YIQ, YPbPr, YCbCr colorspaces
- adding details about github SSH keys in contributing page (#2073)
- ENH: Pass np.random.RandomState to RANSAC (#2072)
- Handle IO objects with tifffile (#2046)
- Updated centroid to use coords - works in 3d (#2083)
- [WIP] Hierarchical Merging of Region Boundary RAGs (#2058)
- Add nD support to clear_border (#2087)
- DOC: update for new API (minor) (#2090)
- Add example for felzenszwalb image segmentation (#2096)
- DOC: add space before column on variable def (minor...) (#2102)
- DOC: Guide new contributors to HTTPS, not SSH (#2082)
- Add François Boulogne to the mailmap (#2117)
- Move skimage.filters.rank description and todos from README into docstring. (#2115)
- Fixing Error and documentation on Otsu Threshold (#2118)
- Add scuinto’s second email address to mailmap (#2122)
- MAINT: around label and regionprops functions. (#2100)
- Add threshold minimum algorithm (#2104)
- Sphinx gallery for example gallery (#2078)
- DOC: make a title shorter in gallery (#2128)
- DOC: refactor axes with lists (#2129)
- DOC ENH + API fix on houghline transform (#2089)
- Fix indentation for example script (#2136)
- Implement mean and triangle thresholding (#2126)
- Move skimage.measure.label references to the docstring (#2143)
- Fix outdated GraphicsGems link (#2149)
- Docstring (#2145)

- Add uint16 images support for co-occurrence matrix (#2095)
- Remove deprecated {h/v}sobel, {h/v}prewitt, {h/v}scharr, roberts_{positive/negative} filters (#2159)
- Remove deprecated _mode_deprecations (#2156)
- Default parameters (#2151)
- ENH: try_all to choose the best threshold algorithm and DOC refactoring (#2110)
- BUGFIX: inverse_map should not be None (#2160)
- Switched felzenszwalb gray to multichannel version (#2134)
- Writing, style, and PEP8 fixes for greycomatrix (#2157)
- Add Frangi and Hessian filters (#2153)
- Improved SimilarityTransform, new EuclideanTransform class (#2044)
- color.colorconv: Fix documentation of rgb2gray() (#2169)
- fix region merging in segmentation.felzenszwalb (#2164)
- Remove deprecated None defaults in rescale_intensity (#2161)
- DOC: add a note to template_match (#2176)
- Added chapter title formatting for numpy_images.rst (#2177)
- Fix threshold_triangle to work with non-integer images. (#2171)
- Improved region boundary RAG docs (#2106)
- ENH add bbox_area to region properties (#2187)
- colorconv: Add rgba2rgb() (#2181)
- DOC: add DOI to references (#2188)
- remove local threshold in try_all_threshold (#2180)
- DOC: add a note on warning treatment (#2198)
- ENH: Speed up Hessian matrix computation (#2194)
- Add missing unittests for data and convert horse to binary (#2196)
- Fix ssim example (#2208)
- [MRG] MAINT: Replaced gaussian_filter with filters.gaussian (#2210)
- [MRG] DOC: corrected mssim docstring to return float (#2218)
- FEAT: Lewiner marching cubes algorithm (#2052)
- Fix bug in salt and pepper noise (#2223)
- TST: Updated AppVeyor to use Conda, added msvc_runtime (#2217)
- Improve docstrings for captions (#2185)
- Add task update version on wikipedia (#2230)
- NEW + DOC: image inversion (#2199)
- ENH: Implements wavelet denoising (from #1833) (#2190)
- TEST: define seed in setup() / Fix random test failure (#2227)
- add n-dimensional support to denoise_wavelet (#2242)

- API: clip_negative will be set to false by default in version 0.15 (func: dtype_limits) (#2228)
- Speedup inpaint_biharmonic (#2234)
- MAINT dtype.py (PEP8) (#2231)
- Removed unused extend_image (#2251)
- ENH: routine to estimate the noise standard deviation from an image (#1837)
- Restrict sphinx builds to a single process. Remove vendored numpydoc. (#2257)
- Added more specific check for image shape in threshold_otsu warning (#2259)
- Allow running setup.py egg_info without numpy installed. (#2260)
- Add compact watershed and clean up existing watershed (#2211)
- Use numpy.pad directly, removing most shipped code in util.pad (#2265)
- DOC: fix references (#2262)
- DOC: tiny fixes in gallery (#2226)
- DOC: fix typo (#2274)
- Update Manifest.in (#2255)
- Bugfix unbounded correlation – Dhyams fix for match template (#2263)
- DOC: Refactor example skeletonize in the gallery (#2141)
- [MRG+1] Insert metadata in docstrings of images in skimage.data.* (#2236)
- MAINT: Radon (docstring, API, PEP8) (#2235)
- [MRG+2] MAINT: Fix numpy deprecation (#2283)
- Reduce whitespace around plots (#2144)
- [MRG+1] By default, clear_border is not inplace (#2285)
- Remove unused imports in transform.{pyx/pxd} (#2288)
- [MRG+1] Add community guidelines to doc navigation (#2287)
- Adding colors to the IHC (#2279)
- FIX: select num_peaks if labels is specified (#2098)
- [MRG+1] Add felzenszwalb shape validation (#2286)
- [MRG+1] more closely match the BayesShrink paper in _wavelet_threshold (#2241)
- Remove usages of subplots_adjust (#2289)
- [MRG+1] Change documentation page favicon (#2291)
- [MRG+1] TST: prefer assert_ from numpy.testing over assert (#2298)
- TSTFIX: Bug fix for development version of scipy (#2302)
- Enhance compare_ssim docstring (#2314)
- Added the missing ‘grey2rgb’ function. (#2316)
- PEP8 (#2304)
- Made Python wrappers for public Cython functions (#2303)
- Update mailing list location (#2328)

- Shape Index (#2312)
- Add pywavelets to runtime requirements in DEPENDS.txt (#2238)
- Refactor variable names in `skimage.draw` (#2321)
- Fix display problem when printing error messages (#2326)
- Added catch for zero image in `threshold_li` (#2338)
- FIX: Modified `peak_local_max` to use `relabel_sequential` (#2341)
- Update favicon in `_static` (#2355)
- Remove incorrect input type assumption in doctings for `rgb2HSV` and h... (#2354)
- Update the default boundary mode in `transform.swirl` (#2331)
- Update `imread()` document (#2358)
- Check for valid mode in `random_walker()`. (#2362)
- Fix 1 broken test in `_shared` not executed by nose/travis (#2229)
- Update hessian matrix code to include order kwarg (#2327)
- Clarify purpose of `beta1` and `beta2` parameters in documentations of sk... (#2382)
- Handle cases for `label2RGB` where input labels are negative and/or nonconsecutive (#2370)
- Update `exposure.equalize_adapthist` args and docstring (#2220)
- Fix (x, y) origin description in user guide (#2385)
- Update docstring for `show_rag` method (#2375)
- Fix display problem when printing error messages (#2372)
- Added a check for empty array in `_shared.utils.py` (#2364)
- Fix no peaks blob log (#2349)
- ENH: Extend `draw.ellipse` with orientation kwarg (#2366)
- Fundamental and essential matrix 8-point algorithm (#1357)
- Fix reference to travis notes (#2403)
- Fix deprecated option in sphinx that causes warning treated as error in travis (#2395)
- Update Travis Script (#2374)
- Remove the freeimage plugin (#1933)
- Fix shape type for histogram (#2417)
- Add illuminant and observer parameters to the `rgb2lab` and `lab2rgb` functions. (#2306)
- PEP8 (#2413)
- MAINT: merge lists of dtypes (#2420)
- Made (partially) pep8-compliant (#2392)
- Added titles and text to make `plot_brief.py` example more clear (#2193)
- DOC: Add reference to standard illuminant (#2418)
- Added titles and text to the subplots to make it easier to new comers for `plot_censure.py` example (#2191)
- Deprecate “dynamic_range” in favor of “data_range” (#2384)

- Make PR 2266 n-D compatible (#4)
- Add new “thin” method based on Guo and Hall 1989 (#2294)
- local threshold niblack sauvola (from Jeysommc PR) (#2266)
- stable ellipse fitting (#2394)
- Add gallery Lucy-Richardson deconvolution algorithm (#2376)
- Improve SIFT loader docstring according to comments and StackOverflow (#2404)
- Change to Javascript loading of search index (patch by Julian Taylor) (#2438)
- Fix segfault in connected components (patch by Yaroslav Halchenko) (#2437)
- Refactor `util/dtype.py` (#2425)
- ENH: Gallery, various little stylish corrections (DFT example). (#2430)
- Make `peak_local_max` return indices sorted, always (#2435)
- Correct comment of `probabilistic_hough_line()`. (#2448)
- Added watershed_line parameter (#2393)
- Solved Gaussian value range #2383 (#2388)
- Gallery: Use Horse to illustrate Convex Hull (#2431)
- MRG: update build matrix for Python 3.6 (#2451)
- Wavelet denoising in YCbCr color space (#2240)
- Gallery: Use gray cmap for coins (#2459)
- Bug fix for Sauvola and Niblack thresholding (#2441)
- MAINT: removes `_wavelet_threshold` docstring (#2460)
- BUG: fix `denoise_wavelet` for odd-length input (#2462)
- MAINT: warns for new multichannel default in `denoise_{bilateral, nl_means}` (#2467)
- Various enhancements in gallery for denoising (#2461)
- Tool for checking completeness of `sdist` (#2085)
- Add different `skimage.hog` blocks normalization methods (#2040)
- DOC: fix typos and add references (#2478)
- update sphinx gallery to 0.1.8 (#2474)
- DOC: Fix typo in gaussian filter docstring (#2487)
- Add `threshold_local`, deprecate old `threshold_adaptive` API (#2490)
- Default edge mode change for `resize` and `rescale` (#2484)
- Add `dask[array]` to optional requirements (#2494)
- DOC: Adds an instruction to `CONTRIBUTING.txt` & Updates the git install link for Windows (#2495)
- ENH: generalize `hough_peak` functions (#2109)
- Fix gallery examples (#2504)
- Bump min `scipy` version (#2254)
- DOC: `img_as_float` add note about range if input `dtype` is float (#2499)

- Update tifffile for 2017.01.12 changes (#2497)
- Replace local_sum by block_reduce in docstrings. (#2498)
- MAINT: pass scipys truncate parameter to gaussian filter API (#2508)
- DOC: gallery: join segmentation: enhancement (#2507)
- Tidy up the deployment of dev docs (#2516)
- Do not require cython for normal builds (#2509)
- Fix broken test_ncut_stable_subgraph for Python 3.6, enable Python 3.6 in Travis (#2511)
- Improved background labeling (#2381)
- For imread's load_func, make the img_num argument optional (#2054)
- Make compatible with current networkx master (#2455)
- Miscellaneous tidying in HOG code (#2526)
- BUG: Fix NumPy error when no descriptors are returned by ORB (#2537)
- BUG: ValueError in restoration.denoise_bilateral for zeros image (#2533)
- Fix link to Python XY (#2542)
- TST: fix ValueError with scipy-0.19.0rc2 (#2544)
- DOC: Update URL for data.coins() (#2548)
- Replace GRIN URL with Flickr URL (#2547)
- Have threshold_minimum return identical results on i686 and x86_64 (#2549)
- Minor Fix (Issue #2554) (#2556)
- Remove offset parameter from filters.threshold_sauvola docstring (#2566)
- Practical guide to reading video files (#1012)
- Remove dask from requirements.txt (#2572)
- Fix morphology.watershed error message (#2570)
- DOC: Added working with OpenCV in user guide (#2519)
- NEW: add shannon entropy (#2416)
- Fix typo in ylabel of GLCM demo (#2576)
- Detection of local extrema from morphology (#2449)
- Add extrema functions to __init__ (#2588)

1.4.30 scikit-image 0.12 release notes

We're happy to announce the release of scikit-image v0.12!

scikit-image is an image processing toolbox for SciPy that includes algorithms for segmentation, geometric transformations, color space manipulation, analysis, filtering, morphology, feature detection, and more.

For more information, examples, and documentation, please visit our website:

<http://scikit-image.org>

and our gallery of examples

http://scikit-image.org/docs/dev/auto_examples/

Highlights and new features

For this release, we merged over 200 pull requests with bug fixes, cleanups, improved documentation and new features. Highlights include:

- 3D skeletonization (#1923)
- Parallelization framework using `dask:skimage.util.apply_parallel` (#1493)
- Laplacian operator `filters.laplace` (#1763)
- Synthetic 2-D and 3-D binary data with rounded blobs (#1485)
- Plugin for `imageio` library (#1575)
- Inpainting algorithm (#1804)
- New handling of background pixels for `measure.label`: 0-valued pixels are considered as background by default, and the label of background pixels is 0.
- Partial support of 3-D images for `skimage.measure.regionprops` (#1505)
- Multi-block local binary patterns (MB-LBP) for texture classification (#1536)
- Seam Carving (resizing without distortion) (#1459)
- Simple image comparison metrics (PSNR, NRMSE) (#1897)
- Region boundary based region adjacency graphs (RAG) (#1495)
- Construction of RAG from label images (#1826)
- `morphology.remove_small_holes` now complements `morphology.remove_small_objects` (#1689)
- Faster cython implementation of `morphology.skeletonize`
- More appropriate default weights in `restoration.denoise_tv_chambolle` and `feature.peak_local_max`
- Correction of bug in the computation of the Euler characteristic in `measure.regionprops`.
- Replace jet by viridis as default colormap
- Alpha layer support for `color.gray2rgb`
- The measure of structural similarity (`measure.compare_ssim`) is now n-dimensional and supports color channels as well.
- We fixed an issue related to incorrect propagation in plateaus in `segmentation.watershed`

Documentation:

- New organization of gallery of examples in sections
- More frequent (automated) updates of online documentation

API Changes

- `equalize_adapthist` now takes a `kernel_size` keyword argument, replacing the `ntiles_*` arguments.
- The functions `blob_dog`, `blob_log` and `blob_doh` now return float arrays instead of integer arrays.
- `transform.integrate` now takes lists of tuples instead of integers to define the window over which to integrate.
- `reverse_map` parameter in `skimage.transform.warp` has been removed.
- `enforce_connectivity` in `skimage.segmentation.slic` defaults to True.
- `skimage.measure.fit.BaseModel._params`, `skimage.transform.ProjectiveTransform._matrix`, `skimage.transform.PolynomialTransform._params`, `skimage.transform.PiecewiseAffineTransform.affines_*` attributes have been removed.
- `skimage.filters.denoise_*` have moved to `skimage.restoration.denoise_*`.

Deprecations

- `filters.gaussian_filter` has been renamed `filters.gaussian`
- `filters.gabor_filter` has been renamed `filters.gabor`
- `restoration.nl_means_denoising` has been renamed `restoration.denoise_nl_means`
- `measure.LineModel` was deprecated in favor of `measure.LineModelND`
- `measure.structural_similarity` has been renamed `measure.compare_ssim`
- `data.lena` has been deprecated, and gallery examples use instead the `data.astronaut()` picture.

Contributors to this release

(Listed alphabetically by last name)

- K.-Michael Aye
- Connelly Barnes
- Sumit Binnani
- Vighnesh Birodkar
- François Boulogne
- Matthew Brett
- Steven Brown
- Arnaud De Bruecker
- Olivier Debeir
- Charles Deledalle
- endolith
- Eric Lubeck
- Victor Escorcia
- Ivo Flipse
- Joel Frederico

- Cédric Gilon
- Christoph Gohlke
- Korijn van Golen
- Emmanuelle Gouillart
- J. Goutin
- Blake Griffith
- M. Hawker
- Jonathan Helmus
- Dhruv Jawali
- Lee Kamentsky
- Kevin Keraudren
- Julius Bier Kirkegaard
- David Koeller
- Gustav Larsson
- Gregory R. Lee
- Guillaume Lemaître
- Benny Lichtner
- Himanshu Mishra
- Juan Nunez-Iglesias
- Ömer Özak
- Leena P.
- Michael Pacer
- Daniil Pakhomov
- David Perez-Suarez
- Egor Panfilov
- David PS
- Sergio Pascual
- Ariel Rokem
- Nicolas Rougier
- Christian Sachs
- Kshitij Saraogi
- Martin Savc
- Johannes Schönberger
- Arve Seljebu
- Tim Sheerman-Chase
- Scott Sievert

- Steven Silvester
- Alexandre Fioravante de Siqueira
- Daichi Suzuo
- Noah Trebesch
- Pratap Vardhan
- Gael Varoquaux
- Stefan van der Walt
- Joshua Warner
- Josh Warner
- Warren Weckesser
- Daniel Wennberg
- John Wiggins
- Robin Wilson
- Olivia Wilson

1.4.31 scikit-image 0.11.0 release notes

We're happy to announce the release of scikit-image v0.11.0!

scikit-image is an image processing toolbox for SciPy that includes algorithms for segmentation, geometric transformations, color space manipulation, analysis, filtering, morphology, feature detection, and more.

For more information, examples, and documentation, please visit our website:

<http://scikit-image.org>

Highlights

For this release, we merged over 200 pull requests with bug fixes, cleanups, improved documentation and new features. Highlights include:

- Region Adjacency Graphs - Color distance RAGs (#1031) - Threshold Cut on RAGs (#1031) - Similarity RAGs (#1080) - Normalized Cut on RAGs (#1080) - RAG drawing (#1087) - Hierarchical merging (#1100)
- Sub-pixel shift registration (#1066)
- Non-local means denoising (#874)
- Sliding window histogram (#1127)
- More illuminants in color conversion (#1130)
- Handling of CMYK images (#1360)
- *stop_probability* for RANSAC (#1176)
- Li thresholding (#1376)
- Signed edge operators (#1240)
- Full ndarray support for *peak_local_max* (#1355)
- Improve conditioning of geometric transformations (#1319)

- Standardize handling of multi-image files (#1200)
- Ellipse structuring element (#1298)
- Multi-line drawing tool (#1065), line handle style (#1179)
- Point in polygon testing (#1123)
- Rotation around a specified center (#1168)
- Add *shape* option to drawing functions (#1222)
- Faster regionprops (#1351)
- `skimage.future` package (#1365)
- More robust I/O module (#1189)

API Changes

- The `skimage.filter` subpackage has been renamed to `skimage.filters`.
- Some edge detectors returned values greater than 1—their results are now appropriately scaled with a factor of $\sqrt{2}$.

Contributors to this release

(Listed alphabetically by last name)

- Fedor Baart
- Vighnesh Birodkar
- François Boulogne
- Nelson Brown
- Alexey Buzmakov
- Julien Coste
- Phil Elson
- Adam Feuer
- Jim Fienup
- Geoffrey French
- Emmanuelle Gouillart
- Charles Harris
- Jonathan Helmus
- Alexander Iacchetta
- Ivana Kajić
- Kevin Keraudren
- Almar Klein
- Gregory R. Lee
- Jeremy Metz

- Stuart Mumford
- Damian Nadales
- Pablo Márquez Neila
- Juan Nunez-Iglesias
- Rebecca Roisin
- Jasper St. Pierre
- Jacopo Sabbatini
- Michael Sarahan
- Salvatore Scaramuzzino
- Phil Schaf
- Johannes Schönberger
- Tim Seifert
- Arve Seljebu
- Steven Sylvester
- Julian Taylor
- Matěj Týč
- Alexey Umnov
- Pratap Vardhan
- Stefan van der Walt
- Joshua Warner
- Tony S Yu

1.4.32 scikit-image 0.10.0 release notes

We're happy to announce the release of scikit-image v0.10.0!

scikit-image is an image processing toolbox for SciPy that includes algorithms for segmentation, geometric transformations, color space manipulation, analysis, filtering, morphology, feature detection, and more.

For more information, examples, and documentation, please visit our website:

<http://scikit-image.org>

New Features

In addition to many bug fixes, (speed) improvements and new examples, the 118 pull requests (1112 commits) merged for this release include the following new or improved features (PR number in brackets):

- BRIEF, ORB and CENSURE features (#834)
- Blob Detection (#903)
- Phase unwrapping (#644)
- Wiener deconvolution (#800)

- IPython notebooks in examples gallery (#1000)
- Luv colorspace conversion (#798)
- Viewer overlays (#810)
- ISODATA thresholding (#859)
- A new rank filter for summation (#844)
- Faster MCP with anisotropy support (#854)
- N-d peak finding (*peak_local_max*) (#906)
- *imread_collection* support for all plugins (#862)
- Enforce SLIC superpixels connectivity and add SLIC-zero (#857, #864)
- Correct mesh orientation (for use in external visualizers) in marching cubes algorithm (#882)
- Loading from URL and alpha support in novice module (#916, #946)
- Equality for regionprops (#956)

API changes

The following backward-incompatible API changes were made between 0.9 and 0.10:

- Removed deprecated functions in *skimage.filter.rank.**
- Removed deprecated parameter *epsilon* of *skimage.viewer.LineProfile*
- Removed backwards-compatibility of *skimage.measure.regionprops*
- Removed *{ratio, sigma}* deprecation warnings of *skimage.segmentation.slic* and also remove explicit *sigma* parameter from doc-string example
- Changed default mode of random_walker segmentation to ‘cg_mg’ > ‘cg’ > ‘bf’, depending on which optional dependencies are available.
- Removed deprecated *out* parameter of *skimage.morphology.binary_**
- Removed deprecated parameter *depth* in *skimage.segmentation.random_walker*
- Removed deprecated logger function in *skimage/_init_.py*
- Removed deprecated function *filter.median_filter*
- Removed deprecated *skimage.color.is_gray* and *skimage.color.is_rgb* functions
- Removed deprecated *skimage.segmentation.visualize_boundaries*
- Removed deprecated *skimage.morphology.greyscale_**
- Removed deprecated *skimage.exposure.equalize*

Contributors to this release

This release was made possible by the collaborative efforts of many contributors, both new and old. They are listed in alphabetical order by surname:

- Raphael Ackermann
- Ankit Agrawal
- Maximilian Albert
- Pietro Berkes
- Vighnesh Birodkar
- François Boulogne
- Olivier Debeir
- Christoph Deil
- Jaidev Deshpande
- Jaime Frio
- Jostein Bø Fløystad
- Neeraj Gangwar
- Christoph Gohlke
- Michael Hansen
- Almar Klein
- Jeremy Metz
- Juan Nunez-Iglesias
- François Orieux
- Guillem Palou
- Rishabh Raj
- Thomas Robitaille
- Michal Romaniuk
- Johannes L. Schönberger
- Steven Sylvester
- Julian Taylor
- Gregor Thalhammer
- Matthew Trentacoste
- Siva Prasad Varma
- Guillem Palou Visa
- Stefan van der Walt
- Joshua Warner
- Tony S Yu
- radioxoma

1.5 Development

Below you will find resources about the development of scikit-image.

1.5.1 How to contribute to scikit-image

Developing Open Source is great fun! Join us on the [scikit-image developer forum](#) and tell us which of the following challenges you'd like to solve.

- Mentoring is available for those new to scientific programming in Python.
- If you're looking for something to implement or to fix, you can browse the [open issues on GitHub](#).
- The technical detail of the *development process* is summed up below. Refer to the *gitwash* for a step-by-step tutorial.

- *Development process*
- *Divergence between upstream main and your feature branch*
- *Guidelines*
- *Stylistic Guidelines*
- *Testing*
- *Test coverage*
- *Building docs*
 - *Requirements*
 - *Fixing Warnings*
- *Deprecation cycle*
- *Bugs*
- *Benchmarks*
 - *Prerequisites*
 - *Writing a benchmark*
 - *Testing the benchmarks locally*
 - *Running your benchmark*
 - *Comparing results to main*

Development process

Here's the long and short of it:

1. If you are a first-time contributor:
 - Go to <https://github.com/scikit-image/scikit-image> and click the “fork” button to create your own copy of the project.
 - Clone the project to your local computer:

```
git clone https://github.com/your-username/scikit-image.git
```

- Change the directory:

```
cd scikit-image
```

- Add the upstream repository:

```
git remote add upstream https://github.com/scikit-image/scikit-image.git
```

- Now, you have remote repositories named:

- `upstream`, which refers to the `scikit-image` repository
- `origin`, which refers to your personal fork

- Next, you need to set up your build environment. Please refer to *Build environment setup* for instructions.

- Finally, we recommend you use a pre-commit hook, which runs some auto-formatters when you do a `git commit`:

```
pre-commit install
```

Note: Although our code is hosted on [github](#), our datasets are stored on [gitlab](#) and fetched with [pooch](#). New datasets must be submitted on gitlab. Once merged, the data registry `skimage/data/_registry.py` in the main Github repository can be updated.

2. Develop your contribution:

- Pull the latest changes from upstream:

```
git checkout main
git pull upstream main
```

- Create a branch for the feature you want to work on. Use a sensible name, such as ‘transform-speedups’:

```
git checkout -b transform-speedups
```

- Commit locally as you progress (with `git add` and `git commit`). Please write [good commit messages](#).

3. To submit your contribution:

- Push your changes back to your fork on GitHub:

```
git push origin transform-speedups
```

- Enter your GitHub username and password (repeat contributors or advanced users can remove this step by connecting to GitHub with SSH).
- Go to GitHub. The new branch will show up with a green “pull request” button – click it.
- If you want, post on the [developer forum](#) to explain your changes or to ask for review.

For a more detailed discussion, read these [detailed documents](#) on how to use Git with `scikit-image` (*Working with scikit-image source code*).

4. Review process:

- Reviewers (the other developers and interested community members) will write inline and/or general comments on your pull request (PR) to help you improve its implementation, documentation, and style. Every single developer working on the project has their code reviewed, and we've come to see it as a friendly conversation from which we all learn and the overall code quality benefits. Therefore, please don't let the review discourage you from contributing: its only aim is to improve the quality of the project, not to criticize (we are, after all, very grateful for the time you're donating!).
- To update your pull request, make your changes on your local repository and commit. As soon as those changes are pushed up (to the same branch as before) the pull request will update automatically.
- Continuous integration (CI) services are triggered after each pull request submission to build the package, run unit tests, measure code coverage, and check the coding style (PEP8) of your branch. The tests must pass before your PR can be merged. If CI fails, you can find out why by clicking on the “failed” icon (red cross) and inspecting the build and test logs.
- A pull request must be approved by two core team members before merging.

5. Document changes

If your change introduces any API modifications, please update `doc/release/release_dev.rst`.

If your change introduces a deprecation, add a reminder to `TODO.txt` for the team to remove the deprecated functionality in the future.

Note: To reviewers: if it is not obvious from the PR description, add a short explanation of what a branch did to the merge message and, if closing a bug, also add “Closes #123” where 123 is the issue number.

Divergence between upstream main and your feature branch

If GitHub indicates that the branch of your PR can no longer be merged automatically, merge the main branch into yours:

```
git fetch upstream main  
git merge upstream/main
```

If any conflicts occur, they need to be fixed before continuing. See which files are in conflict using:

```
git status
```

Which displays a message like:

```
Unmerged paths:  
(use "git add <file>..." to mark resolution)  
  
both modified: file_with_conflict.txt
```

Inside the conflicted file, you'll find sections like these:

```
The way the text looks in your branch
```

Choose one version of the text that should be kept, and delete the rest:

```
The way the text looks in your branch
```

Now, add the fixed file:

```
git add file_with_conflict.txt
```

Once you've fixed all merge conflicts, do:

```
git commit
```

Note: Advanced Git users are encouraged to rebase instead of merge, but we squash and merge most PRs either way.

Guidelines

- All code should have tests (see *test coverage* below for more details).
- All code should be documented, to the same standard as NumPy and SciPy.
- For new functionality, always add an example to the gallery (see *Sphinx-Gallery* below for more details).
- No changes are ever merged without review and approval by two core team members. There are two exceptions to this rule. First, pull requests which affect only the documentation require review and approval by only one core team member in most cases. If the maintainer feels the changes are large or likely to be controversial, two reviews should still be encouraged. The second case is that of minor fixes which restore CI to a working state, because these should be merged fairly quickly. Reach out on the developer forum if you get no response to your pull request. **Never merge your own pull request.**

Stylistic Guidelines

- Set up your editor to remove trailing whitespace. Follow [PEP08](#).
- Use numpy data types instead of strings (np.uint8 instead of "uint8").
- Use the following import conventions:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import ndimage as ndi

# only in Cython code
cimport numpy as cnp
cnp.import_array()
```

- When documenting array parameters, use `image : (M, N) ndarray` and then refer to M and N in the docstring, if necessary.
- Refer to array dimensions as (plane), row, column, not as x, y, z. See *Coordinate conventions* in the user guide for more information.
- Functions should support all input image dtypes. Use utility functions such as `img_as_float` to help convert to an appropriate type. The output format can be whatever is most efficient. This allows us to string together several functions into a pipeline, e.g.:

```
hough(canny(my_image))
```

- Use `Py_ssize_t` as data type for all indexing, shape and size variables in C/C++ and Cython code.

- Use relative module imports, i.e. `from .._shared import xyz` rather than `from skimage._shared import xyz`.
- Wrap Cython code in a pure Python function, which defines the API. This improves compatibility with code introspection tools, which are often not aware of Cython code.
- For Cython functions, release the GIL whenever possible, using `with nogil:`.

Testing

See the testing section of the Installation guide.

Test coverage

Tests for a module should ideally cover all code in that module, i.e., statement coverage should be at 100%.

To measure the test coverage, install `pytest-cov` (using `pip install -r requirements/test.txt`) and then run:

```
$ spin coverage
```

This will print a report with one line for each file in `skimage`, detailing the test coverage:

Name	Stmts	Exec	Cover	Missing
<code>skimage/color/colorconv</code>	77	77	100%	
<code>skimage/filter/__init__</code>	1	1	100%	
...				

Building docs

To build the HTML documentation, you can run:

```
spin docs
```

Then, all the HTML files will be generated in `scikit-image/doc/build/html/`. To rebuild a full clean documentation, run:

```
spin docs --clean
```

Requirements

Sphinx, Sphinx-Gallery, and LaTeX are needed to build the documentation.

Sphinx:

Sphinx and other python packages needed to build the documentation can be installed using: `scikit-image/requirements/docs.txt` file.

```
pip install -r requirements/docs.txt
```

Sphinx-Gallery:

The above install command includes the installation of `Sphinx-Gallery`, which we use to create the *Examples*. Refer to the Sphinx-Gallery documentation for complete instructions on syntax and usage.

If you are contributing an example to the gallery or editing an existing one, build the docs (see above) and open a web browser to check how your edits render at `scikit-image/doc/build/html/auto_examples/`: navigate to the file you have added or changed.

When adding an example, visit also `scikit-image/doc/build/html/auto_examples/index.html` to check how the new thumbnail renders on the gallery's homepage. To change the thumbnail image, please refer to [this section](#) of the Sphinx-Gallery docs.

Note that gallery examples should have a maximum figure width of 8 inches.

LaTeX Ubuntu:

```
sudo apt-get install -qq texlive texlive-latex-extra dvipng
```

LaTeX Mac:

Install the full `MacTex` installation or install the smaller `BasicTex` and add `ucs` and `dvipng` packages:

```
sudo tlmgr install ucs dvipng
```

Fixing Warnings

- “citation not found: R###” There is probably an underscore after a reference in the first line of a docstring (e.g. [1]_). Use this method to find the source file: `$ cd doc/build; grep -rin R###`
- “Duplicate citation R##, other instance in...”” There is probably a [2] without a [1] in one of the docstrings
- Make sure to use pre-sphinxification paths to images (not the `_images` directory)

Deprecation cycle

If the behavior of the library has to be changed, a deprecation cycle must be followed to warn users.

- a deprecation cycle is *not* necessary when:
 - adding a new function, or
 - adding a new keyword argument to the *end* of a function signature, or
 - fixing what was buggy behavior

• a deprecation cycle is necessary for *any breaking API change*, meaning a

change where the function, invoked with the same arguments, would return a different result after the change. This includes:

- changing the order of arguments or keyword arguments, or
- adding arguments or keyword arguments to a function, or
- changing a function’s name or submodule, or
- changing the default value of a function’s arguments.

Usually, our policy is to put in place a deprecation cycle over two releases.

For the sake of illustration, we consider the modification of a default value in a function signature. In version N (therefore, next release will be N+1), we have

```
def a_function(image, rescale=True):
    out = do_something(image, rescale=rescale)
    return out
```

that has to be changed to

```
def a_function(image, rescale=None):
    if rescale is None:
        warn('The default value of rescale will change '
              'to `False` in version N+3.', stacklevel=2)
        rescale = True
    out = do_something(image, rescale=rescale)
    return out
```

and in version N+3

```
def a_function(image, rescale=False):
    out = do_something(image, rescale=rescale)
    return out
```

Here is the process for a 2-release deprecation cycle:

- In the signature, set default to *None*, and modify the docstring to specify that it's *True*.
- In the function, _if_ rescale is set to *None*, set to *True* and warn that the default will change to *False* in version N+3.
- In doc/release/release_dev.rst, under deprecations, add “In *a_function*, the *rescale* argument will default to *False* in N+3.”
- In TODO.txt, create an item in the section related to version N+3 and write “change rescale default to False in *a_function*”.

Note that the 2-release deprecation cycle is not a strict rule and in some cases, the developers can agree on a different procedure upon justification (like when we can't detect the change, or it involves moving or deleting an entire function for example).

Scikit-image uses warnings to highlight changes in its API so that users may update their code accordingly. The `stacklevel` argument sets the location in the callstack where the warnings will point. In most cases, it is appropriate to set the `stacklevel` to 2. When warnings originate from helper routines internal to the scikit-image library, it is may be more appropriate to set the `stacklevel` to 3. For more information, see the documentation of the `warn` function in the Python standard library.

To test if your warning is being emitted correctly, try calling the function from an IPython console. It should point you to the console input itself instead of being emitted by the files in the scikit-image library.

- **Good:** ipython:1: UserWarning: ...
- **Bad:** scikit-image/skimage/measure/_structural_similarity.py:155: UserWarning:

Bugs

Please report bugs on [GitHub](#).

Benchmarks

While not mandatory for most pull requests, we ask that performance related PRs include a benchmark in order to clearly depict the use-case that is being optimized for. A historical view of our snapshots can be found on at the following website.

In this section we will review how to setup the benchmarks, and three commands `spin asv -- dev`, `spin asv -- run` and `spin asv -- continuous`.

Prerequisites

Begin by installing `airspeed velocity` in your development environment. Prior to installation, be sure to activate your development environment, then if using `venv` you may install the requirement with:

```
source skimage-dev/bin/activate
pip install asv
```

If you are using `conda`, then the command:

```
conda activate skimage-dev
conda install asv
```

is more appropriate. Once installed, it is useful to run the command:

```
spin asv -- machine
```

To let `airspeed velocity` know more information about your machine.

Writing a benchmark

To write benchmark, add a file in the `benchmarks` directory which contains a class with one `setup` method and at least one method prefixed with `time_`.

The `time_` method should only contain code you wish to benchmark. Therefore it is useful to move everything that prepares the benchmark scenario into the `setup` method. This function is called before calling a `time_` method and its execution time is not factored into the benchmarks.

Take for example the `TransformSuite` benchmark:

```
import numpy as np
from skimage import transform

class TransformSuite:
    """Benchmark for transform routines in scikit-image."""

    def setup(self):
        self.image = np.zeros((2000, 2000))
        idx = np.arange(500, 1500)
        self.image[idx[::-1], idx] = 255
        self.image[idx, idx] = 255

    def time_hough_line(self):
        result1, result2, result3 = transform.hough_line(self.image)
```

Here, the creation of the image is completed in the `setup` method, and not included in the reported time of the benchmark.

It is also possible to benchmark features such as peak memory usage. To learn more about the features of `asv`, please refer to the official [airspeed velocity documentation](#).

Also, the benchmark files need to be importable when benchmarking old versions of scikit-image. So if anything from scikit-image is imported at the top level, it should be done as:

```
try:  
    from skimage import metrics  
except ImportError:  
    pass
```

The benchmarks themselves don't need any guarding against missing features, only the top-level imports.

To allow tests of newer functions to be marked as “n/a” (not available) rather than “failed” for older versions, the setup method itself can raise a `NotImplemented` error. See the following example for the registration module:

```
try:  
    from skimage import registration  
except ImportError:  
    raise NotImplementedError("registration module not available")
```

Testing the benchmarks locally

Prior to running the true benchmark, it is often worthwhile to test that the code is free of typos. To do so, you may use the command:

```
spin asv -- dev -b TransformSuite
```

Where the `TransformSuite` above will be run once in your current environment to test that everything is in order.

Running your benchmark

The command above is fast, but doesn't test the performance of the code adequately. To do that you may want to run the benchmark in your current environment to see the performance of your change as you are developing new features. The command `asv run -E existing` will specify that you wish to run the benchmark in your existing environment. This will save a significant amount of time since building scikit-image can be a time consuming task:

```
spin asv -- run -E existing -b TransformSuite
```

Comparing results to main

Often, the goal of a PR is to compare the results of the modifications in terms speed to a snapshot of the code that is in the main branch of the `scikit-image` repository. The command `asv continuous` is of help here:

```
spin asv -- continuous main -b TransformSuite
```

This call will build out the environments specified in the `asv.conf.json` file and compare the performance of the benchmark between your current commit and the code in the main branch.

The output may look something like:

```
$ spin asv -- continuous main -b TransformSuite  
· Creating environments  
· Discovering benchmarks  
.. Uninstalling from conda-py3.7-cython-numpy1.15-scipy  
.. Installing 544c0fe3 <benchmark_docs> into conda-py3.7-cython-numpy1.15-scipy.
```

(continues on next page)

(continued from previous page)

```

· Running 4 total benchmarks (2 commits * 2 environments * 1 benchmarks)
[ 0.00%] · For scikit-image commit 37c764cb <benchmark_docs~1> (round 1/2):
[...]
[100.00%] ... .ansform.TransformSuite.time_hough_line           33.2±2ms

BENCHMARKS NOT SIGNIFICANTLY CHANGED.

```

In this case, the differences between HEAD and main are not significant enough for airspeed velocity to report.

It is also possible to get a comparison of results for two specific revisions for which benchmark results have previously been run via the `asv compare` command:

```
spin asv -- compare v0.14.5 v0.17.2
```

Finally, one can also run ASV benchmarks only for a specific commit hash or release tag by appending `^!` to the commit or tag name. For example to run the skimage.filter module benchmarks on release v0.17.2:

```
spin asv -- run -b Filter v0.17.2^!
```

1.5.2 Working with *scikit-image* source code

Contents:

Introduction

These pages describe a [git](#) and [github](#) workflow for the `scikit-image` project.

There are several different workflows here, for different ways of working with *scikit-image*.

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

For general resources for learning git, see *git resources*.

Install git

Overview

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

In detail

See the git page for the most recent information.

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

There are good instructions here: <https://book.git-scm.com/book/en/v2/Getting-Started-Installing-Git>

Following the latest source

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

The steps are:

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

Get the local copy of the code

From the command line:

```
git clone https://github.com/scikit-image/scikit-image.git
```

You now have a copy of the code tree in the new `scikit-image` directory.

Updating the code

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

```
cd scikit-image  
git pull
```

The tree in `scikit-image` will now have the latest changes from the initial repository.

Making a patch

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

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

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

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

Making patches

Overview

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

Then, send the generated patch files to the [scikit-image developer forum](#) — where we will thank you warmly.

In detail

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

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

2. If you don't already have one, clone a copy of the scikit-image repository:

```
git clone https://github.com/scikit-image/scikit-image.git
cd scikit-image
```

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

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

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

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

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

- When you have finished, check you have committed all your changes:

```
git status
```

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

```
git format-patch -M -C main
```

You will now have several files named for the commits:

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

Send these files to the [scikit-image developer forum](#).

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

```
git checkout main
```

Moving from patching to development

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

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

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

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

Git for development

Contents:

Making your own copy (fork) of scikit-image

You need to do this only once. The instructions here are very similar to the instructions at <https://help.github.com/en/github/getting-started-with-github/fork-a-repo> — please see that page for more detail. We're repeating some of it here just to give the specifics for the [scikit-image](#) project, and to suggest some default names.

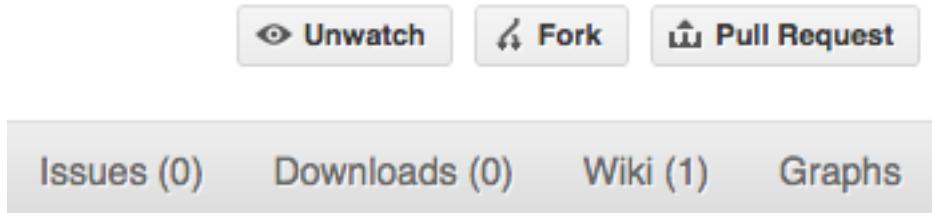
Set up and configure a github account

If you don't have a github account, go to the [github page](#), and make one.

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

Create your own forked copy of scikit-image

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



Now, after a short pause and some ‘Hardcore forking action’, you should find yourself at the home page for your own forked copy of [scikit-image](#).

Set up your fork

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

Overview

```
git clone git@github.com:your-user-name/scikit-image.git
cd scikit-image
git remote add upstream https://github.com/scikit-image/scikit-image.git
```

In detail

Clone your fork

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

```
* main
remotes/origin/main
```

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

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

Linking your repository to the upstream repo

```
cd scikit-image
git remote add upstream https://github.com/scikit-image/scikit-image.git
```

`upstream` here is just the arbitrary name we're using to refer to the main `scikit-image` repository at `scikit-image` `github`.

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

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

```
upstream  https://github.com/scikit-image/scikit-image.git (fetch)
upstream  https://github.com/scikit-image/scikit-image.git (push)
origin    git@github.com:your-user-name/scikit-image.git (fetch)
origin    git@github.com:your-user-name/scikit-image.git (push)
```

Configure git

Overview

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

Here is an example `.gitconfig` file:

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

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

[core]
    editor = vim

[merge]
    summary = true
```

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

```
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
git config --global alias.ci "commit -a"
git config --global alias.co checkout
git config --global alias.st "status -a"
git config --global alias.stat "status -a"
git config --global alias.br branch
```

(continues on next page)

(continued from previous page)

```
git config --global alias.wdiff "diff --color-words"
git config --global core.editor vim
git config --global merge.summary true
```

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

In detail

`user.name` and `user.email`

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

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

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

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

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

Aliases

You might well benefit from some aliases to common commands.

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

The following `git config --global` commands:

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

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

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

Editor

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

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

Merging

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

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

Or from the command line:

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

Fancy log output

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

```
lg = log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr)
˓→%C(bold blue)[%an]%Creset' --abbrev-commit --date=relative
```

You use the alias with:

```
git lg
```

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

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

(continues on next page)

(continued from previous page)

Terhorst]

```
| |\ \
| | /
```

Thanks to Yury V. Zaytsev for posting it.

Development workflow

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

Workflow summary

In what follows we'll refer to the upstream scikit-image `main` branch, as "trunk".

- Don't use your `main` branch for anything. Consider deleting it.
- When you are starting a new set of changes, fetch any changes from trunk, and start a new *feature branch* from that.
- Make a new branch for each separable set of changes — "one task, one branch" ([ipython git workflow](#)).
- Name your branch for the purpose of the changes - e.g. `bugfix-for-issue-14` or `refactor-database-code`.
- If you can possibly avoid it, avoid merging trunk or any other branches into your feature branch while you are working.
- If you do find yourself merging from trunk, consider *Rebasing on trunk*
- Ask on the [scikit-image developer forum](#) if you get stuck.
- Ask for code review!

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

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

Consider deleting your main branch

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

Update the mirror of trunk

First make sure you have done *Linking your repository to the upstream repo*.

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

```
git fetch upstream
```

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

Make a new feature branch

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

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

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

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

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

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

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

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

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

The editing workflow

Overview

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

In more detail

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

```
# On branch ny-new-feature
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   README
#
# Untracked files:
```

(continues on next page)

(continued from previous page)

```
#   (use "git add <file>..." to include in what will be committed)
#
# INSTALL
no changes added to commit (use "git add" and/or "git commit -a")
```

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

Ask for your changes to be reviewed or merged

When you are ready to ask for someone to review your code and consider a merge:

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



3. Click on the ‘Pull request’ button:



[Downloads \(0\)](#) [Wiki \(1\)](#) [Graphs](#)

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

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

Some other things you might want to do

Delete a branch on github

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

(Note the colon : before test-branch. See also: <https://help.github.com/en/github/using-git/managing-remote-repositories>

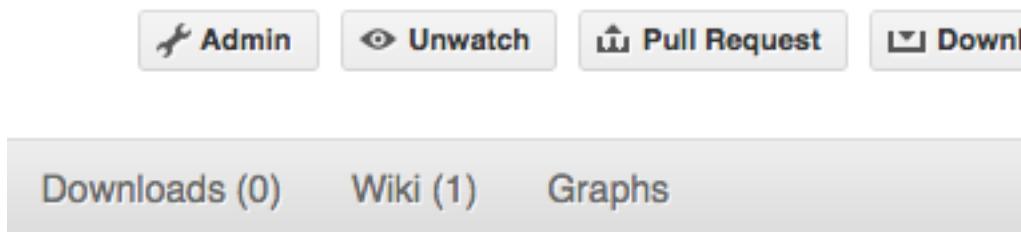
Several people sharing a single repository

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

First fork scikit-image into your account, as from *Making your own copy (fork) of scikit-image*.

Then, go to your forked repository github page, say <https://github.com/your-user-name/scikit-image>

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



Now all those people can do:

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

Remember that links starting with git@ use the ssh protocol.

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

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

Explore your repository

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

```
gitk --all
```

To see a linear list of commits for this branch:

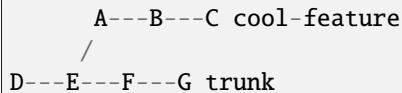
```
git log
```

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

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

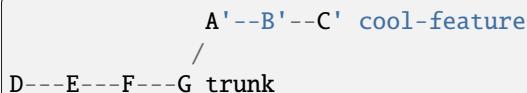
Rebasing on trunk

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



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

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



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

To do a rebase on trunk:

```

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

```

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

```
git rebase upstream/main
```

When all looks good you can delete your backup branch:

```
git branch -D tmp
```

If it doesn't look good you may need to have a look at *Recovering from mess-ups*.

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

Recovering from mess-ups

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

If you mess up during a rebase:

```
git rebase --abort
```

If you notice you messed up after the rebase:

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

If you forgot to make a backup branch:

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

8630830 cool-feature@{0}: commit: BUG: io: close file handles immediately
278dd2a cool-feature@{1}: rebase finished: refs/heads/my-feature-branch onto ↵
→ 11ee694744f2552d
26aa21a cool-feature@{2}: commit: BUG: lib: make seek_gzip_factory not leak gzip obj
...
# reset the branch to where it was before the botched rebase
git reset --hard cool-feature@{2}
```

Rewriting commit history

Note: Do this only for your own feature branches.

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

This can be done via *interactive rebasing*.

Suppose that the commit history looks like this:

```
git log --oneline
eadc391 Fix some remaining bugs
a815645 Modify it so that it works
2dec1ac Fix a few bugs + disable
13d7934 First implementation
6ad92e5 * masked is now an instance of a new object, MaskedConstant
29001ed Add pre-nep for a copule of structured_array_extensions.
...
```

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

- Rewrite the commit message for 13d7934 to something more sensible.
- Combine the commits 2dec1ac, a815645, eadc391 into a single one.

We do as follows:

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

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

```
pick 13d7934 First implementation
pick 2dec1ac Fix a few bugs + disable
pick a815645 Modify it so that it works
pick eadc391 Fix some remaining bugs

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

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

```
r 13d7934 First implementation
pick 2dec1ac Fix a few bugs + disable
f a815645 Modify it so that it works
f eadc391 Fix some remaining bugs
```

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

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

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

and the history looks now like this:

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

If it went wrong, recovery is again possible as explained *above*.

Maintainer workflow

This page is for maintainers — those of us who merge our own or other peoples' changes into the upstream repository. Being as how you're a maintainer, you are completely on top of the basic stuff in *Development workflow*.

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

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

```
git remote add upstream-rw git@github.com:scikit-image/scikit-image.git  
git fetch upstream-rw
```

Integrating changes

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

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

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

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

A few commits

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

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

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

A long series of commits

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

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

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

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

Check the history

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

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

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

Push to trunk

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

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

git resources

Tutorials and summaries

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

Advanced git workflow

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

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

Manual pages online

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

- [git add](#)
- [git branch](#)
- [git checkout](#)
- [git clone](#)
- [git commit](#)
- [git config](#)
- [git diff](#)
- [git log](#)
- [git pull](#)
- [git push](#)
- [git remote](#)
- [git status](#)

1.5.3 Core Developer Guide

Welcome, new core developer! The core team appreciate the quality of your work, and enjoy working with you; we have therefore invited you to join us. Thank you for your numerous contributions to the project so far.

This document offers guidelines for your new role. First and foremost, you should familiarize yourself with the project's *mission, vision, and values*. When in doubt, always refer back here.

As a core team member, you gain the responsibility of shepherding other contributors through the review process; here are some guidelines.

All Contributors Are Treated The Same

You now have the ability to push changes directly to the main branch, but should never do so; instead, continue making pull requests as before and in accordance with the *general contributor guide*.

As a core contributor, you gain the ability to merge or approve other contributors' pull requests. Much like nuclear launch keys, it is a shared power: you must merge *only after* another core has approved the pull request, *and* after you yourself have carefully reviewed it. (See *Reviewing* and especially *Merge Only Changes You Understand* below.) To ensure a clean git history, use GitHub's [Squash and Merge](#) feature to merge, unless you have a good reason not to do so.

Reviewing

How to Conduct A Good Review

Always be kind to contributors. Nearly all of scikit-image is volunteer work, for which we are tremendously grateful. Provide constructive criticism on ideas and implementations, and remind yourself of how it felt when your own work was being evaluated as a novice.

scikit-image strongly values mentorship in code review. New users often need more handholding, having little to no git experience. Repeat yourself liberally, and, if you don't recognize a contributor, point them to our development guide, or other GitHub workflow tutorials around the web. Do not assume that they know how GitHub works (e.g., many don't realize that adding a commit automatically updates a pull request). Gentle, polite, kind encouragement can make the difference between a new core developer and an abandoned pull request.

When reviewing, focus on the following:

1. **API:** The API is what users see when they first use scikit-image. APIs are difficult to change once released, so should be simple, functional (i.e. not carry state), consistent with other parts of the library, and should avoid modifying input variables. Please familiarize yourself with the project's [deprecation policy](#)
2. **Documentation:** Any new feature should have a gallery example, that not only illustrates but explains it.
3. **The algorithm:** You should understand the code being modified or added before approving it. (See *Merge Only Changes You Understand* below.) Implementations should do what they claim, and be simple, readable, and efficient.
4. **Tests:** All contributions to the library *must* be tested, and each added line of code should be covered by at least one test. Good tests not only execute the code, but explores corner cases. It is tempting not to review tests, but please do so.
5. **Licensing:** New contributions should be available under the same license as or be compatible with scikit-image's license. Examples of BSD-compatible licenses are the [MIT License](#) and [Apache License 2.0](#). When in doubt, ask the team for help. If you, the contributor, are not the copyright holder of the submitted code, please ask the original authors for approval and include their names in LICENSE.txt. You can use the other entries in that file as templates.
6. **Established methods:** In general, we are looking to include algorithms and methods which are established, well documented in the literature and widely used by the imaging community. While this is not a hard requirement, new contributions should be consistent with *our mission*.

Other changes may be *nitpicky*: spelling mistakes, formatting, etc. Do not ask contributors to make these changes, and instead make the changes by [pushing to their branch](#) or using GitHub's [suggestion feature](#). (The latter is preferred because it gives the contributor a choice in whether to accept the changes.)

Our default merge policy is to squash all PR commits into a single commit. Users who wish to bring the latest changes from `main` into their branch should be advised to merge, not to rebase. Even when merge conflicts arise, don't ask for a rebase unless you know that a contributor is experienced with git. Instead, rebase the branch yourself, force-push to their branch, and advise the contributor on how to force-pull. If the contributor is no longer active, you may take over their branch by submitting a new pull request and closing the original. In doing so, ensure you communicate that you are not throwing the contributor's work away!

Please add a note to a pull request after you push new changes; GitHub does not send out notifications for these.

Merge Only Changes You Understand

Long-term maintainability is an important concern. Code doesn't merely have to *work*, but should be *understood* by multiple core developers. Changes will have to be made in the future, and the original contributor may have moved on.

Therefore, *do not merge a code change unless you understand it*. Ask for help freely: we have a long history of consulting community members, or even external developers, for added insight where needed, and see this as a great learning opportunity.

While we collectively “own” any patches (and bugs!) that become part of the code base, you are vouching for changes you merge. Please take that responsibility seriously.

In practice, if you are the second core developer reviewing and approving a given pull request, you typically merge it (again, using GitHub’s Squash and Merge feature) in the wake of your approval. What are the exceptions to this process? If the pull request has been particularly controversial or the subject of much debate (e.g., involving API changes), then you would want to wait a few days before merging. This waiting time gives others a chance to speak up in case they are not fine with the current state of the pull request. Another exceptional situation is one where the first approving review happened a long time ago and many changes have taken place in the meantime.

When squashing commits GitHub concatenates all commit messages. Please edit the resulting message so that it gives a concise, tidy overview of changes. For example, you may want to grab the description from the PR itself, and delete lines such as “pep8 fix”, “apply review comments”, etc. Please retain all Co-authored-by entries.

Closing issues and pull requests

Sometimes, an issue must be closed that was not fully resolved. This can be for a number of reasons:

- the person behind the original post has not responded to calls for clarification, and none of the core developers have been able to reproduce their issue;
- fixing the issue is difficult, and it is deemed too niche a use case to devote sustained effort or prioritize over other issues; or
- the use case or feature request is something that core developers feel does not belong in scikit-image,

among others. Similarly, pull requests sometimes need to be closed without merging, because:

- the pull request implements a niche feature that we consider not worth the added maintenance burden;
- the pull request implements a useful feature, but requires significant effort to bring up to scikit-image’s standards, and the original contributor has moved on, and no other developer can be found to make the necessary changes; or
- the pull request makes changes that do not align with our values, such as increasing the code complexity of a function significantly to implement a marginal speedup,

among others.

All these may be valid reasons for closing, but we must be wary not to alienate contributors by closing an issue or pull request without an explanation. When closing, your message should:

- explain clearly how the decision was made to close. This is particularly important when the decision was made in a community meeting, which does not have as visible a record as the comments thread on the issue itself;
- thank the contributor(s) for their work; and
- provide a clear path for the contributor or anyone else to appeal the decision.

These points help ensure that all contributors feel welcome and empowered to keep contributing, regardless of the outcome of past contributions.

Further resources

As a core member, you should be familiar with community and developer resources such as:

- Our [contributor guide](#)
- Our [community guidelines](#)
- [PEP8](#) for Python style
- [PEP257](#) and the [NumPy documentation guide](#) for docstrings. (NumPy docstrings are a superset of PEP257. You should read both.)
- The scikit-image tag on StackOverflow
- The scikit-image tag on [forum.image.sc](#)
- Our [developer forum](#)
- Our [chat room](#)

You are not required to monitor all of the social resources.

Inviting New Core Members

Any core member may nominate other contributors to join the core team. Nominations happen on a private email list, skimage-core@discuss.scientific-python.org. As of this writing, there is no hard-and-fast rule about who can be nominated; at a minimum, they should have: been part of the project for at least six months, contributed significant changes of their own, contributed to the discussion and review of others' work, and collaborated in a way befitting our community values.

Contribute To This Guide!

This guide reflects the experience of the current core developers. We may well have missed things that, by now, have become second nature—things that you, as a new team member, will spot more easily. Please ask the other core developers if you have any questions, and submit a pull request with insights gained.

Conclusion

We are excited to have you on board! We look forward to your contributions to the code base and the community. Thank you in advance!

1.5.4 scikit-image proposals (SKIPS)

SKIPS document any major changes or proposals to the scikit-image project.

Sections

SKIP 0 — Purpose and Process

Author

Jarrod Millman <millman@berkeley.edu>

Author

Juan Nunez-Iglesias <juan.nunez-iglesias@monash.edu>

Author

Stéfan van der Walt <stefanv@berkeley.edu>

Status

Active

Type

Process

Created

2019-07-30

What is a SKIP?

SKIP stands for **scikit-image proposal**. A SKIP is a design document providing information to the community, or describing a new feature for scikit-image or its processes or environment. The SKIP should provide a rationale for the proposed change as well as a concise technical specification, if applicable.

We intend SKIPS to be the primary mechanisms for proposing major new features, for collecting community input on an issue, and for documenting the design decisions that have gone into scikit-image. The SKIP author is responsible for building consensus within the community and documenting dissenting opinions.

Because the SKIPS are maintained as text files in a versioned repository, their revision history is the historical record of the feature proposal¹.

Types

There are three kinds of SKIPS:

1. A **Standards Track** SKIP describes a new feature or implementation for scikit-image.
2. An **Informational** SKIP describes a scikit-image design issue, or provides general guidelines or information to the Python community, but does not propose a new feature. Informational SKIPS do not necessarily represent a scikit-image community consensus or recommendation, so users and implementers are free to ignore Informational SKIPS.
3. A **Process** SKIP describes a process surrounding scikit-image, or proposes a change to (or an event in) a process. Process SKIPS are like Standards Track SKIPS but apply to areas other than the scikit-image library itself. They may propose an implementation, but not to scikit-image's codebase; they require community consensus.

¹ This historical record is available by the normal git commands for retrieving older revisions, and can also be browsed on [GitHub](#).

Examples include procedures, guidelines, changes to the decision-making process, and changes to the tools or environment used in scikit-image development. Any meta-SKIP is also considered a Process SKIP.

SKIP Workflow

The SKIP process begins with a new idea for scikit-image. A SKIP should contain a single key proposal or new idea. Small enhancements or patches often don't need a SKIP and can be injected into the scikit-image development workflow with a pull request to the scikit-image [repo](#). The more focused the SKIP, the more likely it is to be accepted.

Each SKIP must have a champion—someone who writes the SKIP using the style and format described below, shepherds the discussions in the appropriate forums, and attempts to build community consensus around the idea. The SKIP champion (a.k.a. Author) should first attempt to ascertain whether the idea is suitable for a SKIP. Posting to the scikit-image [developer forum](#) is the best way to do this.

The proposal should be submitted as a draft SKIP via a [GitHub pull request](#) to the `doc/source/skips` directory with the name `skip-<n>.rst` where `<n>` is an appropriately assigned number (e.g., `skip-35.rst`). The draft must use the *SKIP X — Template and Instructions* file.

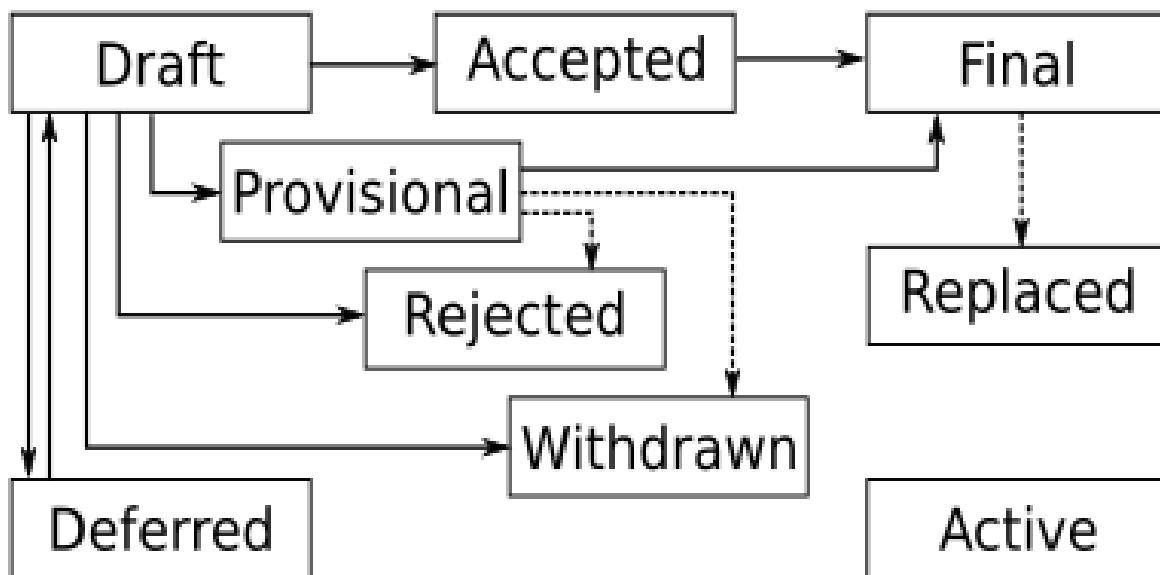
Once the PR is in place, the SKIP should be announced on the developer forum for discussion (comments on the PR itself should be restricted to minor editorial and technical fixes).

At the earliest convenience, the PR should be merged (regardless of whether it is accepted during discussion). A SKIP that outlines a coherent argument and that is considered reasonably complete should be merged optimistically, regardless of whether it is accepted during discussion. Additional PRs may be made by the author to update or expand the SKIP, or by maintainers to set its status, discussion URL, etc.

Standards Track SKIPS consist of two parts, a design document and a reference implementation. It is generally recommended that at least a prototype implementation be co-developed with the SKIP, as ideas that sound good in principle sometimes turn out to be impractical. Often it makes sense for the prototype implementation to be made available as PR to the scikit-image repo, as long as it is properly marked as WIP (work in progress).

Review and Resolution

SKIPS are discussed on the developer forum. The possible paths of the status of SKIPS are as follows:



All SKIPs should be created with the **Draft** status.

Eventually, after discussion, there may be a consensus that the SKIP should be accepted – see the next section for details. At this point the status becomes **Accepted**.

Once a SKIP has been **Accepted**, the reference implementation must be completed. When the reference implementation is complete and incorporated into the main source code repository, the status will be changed to **Final**.

To allow gathering of additional design and interface feedback before committing to long term stability for a language feature or standard library API, a SKIP may also be marked as “**Provisional**”. This is short for “**Provisionally Accepted**”, and indicates that the proposal has been accepted for inclusion in the reference implementation, but additional user feedback is needed before the full design can be considered “**Final**”. Unlike regular accepted SKIPs, provisionally accepted SKIPs may still be **Rejected** or **Withdrawn** even after the related changes have been included in a release.

Wherever possible, it is considered preferable to reduce the scope of a proposal to avoid the need to rely on the “**Provisional**” status (e.g. by deferring some features to later SKIPs), as this status can lead to version compatibility challenges in the wider ecosystem.

A SKIP can also be assigned status **Deferred**. The SKIP author or a core developer can assign the SKIP this status when no progress is being made on the SKIP.

A SKIP can also be **Rejected**. Perhaps after all is said and done it was not a good idea. It is still important to have a record of this fact. The **Withdrawn** status is similar—it means that the SKIP author themselves has decided that the SKIP is actually a bad idea, or has accepted that a competing proposal is a better alternative.

When a SKIP is **Accepted**, **Rejected**, or **Withdrawn**, the SKIP should be updated accordingly. In addition to updating the status field, at the very least the **Resolution** header should be added with a link to the relevant post on the discussion forum.

SKIPs can also be **Superseded** by a different SKIP, rendering the original obsolete. The **Replaced-By** and **Replaces** headers should be added to the original and new SKIPs respectively.

Process SKIPs may also have a status of **Active** if they are never meant to be completed, e.g. SKIP 0 (this SKIP).

How a SKIP becomes Accepted

A SKIP is **Accepted** by consensus of all interested contributors. We need a concrete way to tell whether consensus has been reached. When you think a SKIP is ready to accept, start a topic on the developer forum with a subject like:

Proposal to accept SKIP #<number>: <title>

In the body of your email, you should:

- link to the latest version of the SKIP,
- briefly describe any major points of contention and how they were resolved,
- include a sentence like: “If there are no substantive objections within 7 days from this email, then the SKIP will be accepted; see SKIP 0 for more details.”

For an equivalent example in the NumPy library, see: <https://mail.python.org/pipermail/numpy-discussion/2018-June/078345.html>

After you send the email, you should make sure to link to the email thread from the **Discussion** section of the SKIP, so that people can find it later.

Generally the SKIP author will be the one to send this email, but anyone can do it – the important thing is to make sure that everyone knows when a SKIP is on the verge of acceptance, and give them a final chance to respond. If there’s some special reason to extend this final comment period beyond 7 days, then that’s fine, just say so in the email. You shouldn’t do less than 7 days, because sometimes people are travelling or similar and need some time to respond.

In general, the goal is to make sure that the community has consensus, not provide a rigid policy for people to try to game. When in doubt, err on the side of asking for more feedback and looking for opportunities to compromise.

If the final comment period passes without any substantive objections, then the SKIP can officially be marked Accepted. You should send a followup email notifying the list (celebratory emoji optional but encouraged), and then update the SKIP by setting its :Status: to Accepted, and its :Resolution: header to a link to your followup email.

If there *are* substantive objections, then the SKIP remains in Draft state, discussion continues as normal, and it can be proposed for acceptance again later once the objections are resolved.

In unusual cases, when no consensus can be reached between core developers, the scikit-image Steering Council may be asked to decide whether a controversial SKIP is Accepted.

Maintenance

In general, Standards track SKIPS are no longer modified after they have reached the Final state, as the code and project documentation are considered the ultimate reference for the implemented feature. They may, however, be updated under special circumstances.

Process SKIPS may be updated over time to reflect changes to development practices and other details. The precise process followed in these cases will depend on the nature and purpose of the SKIP being updated.

Format and Template

SKIPS are UTF-8 encoded text files using the reStructuredText format. Please see the *SKIP X — Template and Instructions* file and the *reStructuredTextPrimer* for more information. We use Sphinx to convert SKIPS to HTML for viewing on the web².

Header Preamble

Each SKIP must begin with a header preamble. The headers must appear in the following order. Headers marked with * are optional. All other headers are required.

```
:Author: <list of authors' real names and optionally, email addresses>
>Status: <Draft | Active | Accepted | Deferred | Rejected |
          Withdrawn | Final | Superseded>
>Type: <Standards Track | Process>
:Created: <date created on, in dd-mmm-yyyy format>
* :Requires: <skip numbers>
* :skimage-Version: <version number>
* :Replaces: <skip number>
* :Replaced-By: <skip number>
* :Resolution: <url>
```

The Author header lists the names, and optionally the email addresses of all the authors of the SKIP. The format of the Author header value must be

Random J. User <address@dom.ain>

if the email address is included, and just

Random J. User

² The URL for viewing SKIPS on the web is <https://scikit-image.org/docs/stable/skips/>

if the address is not given. If there are multiple authors, each should be on a separate line.

Discussion

- <https://github.com/scikit-image/scikit-image/pull/3585>

References and Footnotes

Copyright

This document has been placed in the public domain.

SKIP 1 — scikit-image governance and decision-making

Author

Juan Nunez-Iglesias <juan.nunez-iglesias@monash.edu>

Author

Stéfan van der Walt <stefanv@berkeley.edu>

Author

Josh Warner

Author

François Boulogne

Author

Emmanuelle Gouillart

Author

Mark Harfouche

Author

Lars Grüter

Author

Egor Panfilov

Status

Final

Type

Process

Created

2019-07-02

Resolved

2019-09-25

Resolution<https://github.com/scikit-image/scikit-image/pull/4182>**skimage-Version**

0.16

Abstract

The purpose of this document is to formalize the governance process used by the scikit-image project, to clarify how decisions are made and how the various elements of our community interact.

This is a consensus-based community project. Anyone with an interest in the project can join the community, contribute to the project design, and participate in the decision making process. This document describes how that participation takes place, how to find consensus, and how deadlocks are resolved.

Roles And Responsibilities**The Community**

The scikit-image community consists of anyone using or working with the project in any way.

Contributors

A community member can become a contributor by interacting directly with the project in concrete ways, such as:

- proposing a change to the code via a GitHub pull request;
- reporting issues on our [GitHub issues page](#);
- proposing a change to the documentation, [website](#), or [tutorials](#) via a GitHub pull request;
- discussing the design of the library, website, or tutorials on the [developer forum](#), in the [project chat room](#), or in existing issues and pull requests; or
- reviewing [open pull requests](#),

among other possibilities. Any community member can become a contributor, and all are encouraged to do so. By contributing to the project, community members can directly help to shape its future.

Contributors are encouraged to read the *contributing guide*.

Core developers

Core developers are community members that have demonstrated continued commitment to the project through ongoing contributions. They have shown they can be trusted to maintain scikit-image with care. Becoming a core developer allows contributors to merge approved pull requests, cast votes for and against merging a pull-request, and be involved in deciding major changes to the API, and thereby more easily carry on with their project related activities. Core developers appear as organization members on the scikit-image [GitHub organization](#). Core developers are expected to review code contributions while adhering to the *core developer guide*.

New core developers can be nominated by any existing core developer. Discussion about new core developer nominations is one of the few activities that takes place on the project's private management list. The decision to invite a new core developer must be made by "lazy consensus", meaning unanimous agreement by all responding existing core developers. Invitation must take place at least one week after initial nomination, to allow existing members time to voice any objections.

Steering Council

The Steering Council (SC) members are core developers who have additional responsibilities to ensure the smooth running of the project. SC members are expected to participate in strategic planning, approve changes to the governance model, and make decisions about funding granted to the project itself. (Funding to community members is theirs to pursue and manage.) The purpose of the SC is to ensure smooth progress from the big-picture perspective. Changes that impact the full project require analysis informed by long experience with both the project and the larger ecosystem. When the core developer community (including the SC members) fails to reach such a consensus in a reasonable timeframe, the SC is the entity that resolves the issue.

The steering council is fixed in size to five members. This may be expanded in the future. The initial steering council of scikit-image consists of Stéfan van der Walt, Juan Nunez-Iglesias, Emmanuelle Gouillart, Josh Warner, and Zachary Pincus. The SC membership is revisited every January. SC members who do not actively engage with the SC duties are expected to resign. New members are added by nomination by a core developer. Nominees should have demonstrated long-term, continued commitment to the project and its *values*. A nomination will result in discussion that cannot take more than a month and then admission to the SC by consensus.

The scikit-image steering council may be contacted at skimage-steering@groups.io.

Decision Making Process

Decisions about the future of the project are made through discussion with all members of the community. All non-sensitive project management discussion takes place on the project [developer forum](#) and the [issue tracker](#). Occasionally, sensitive discussion may occur on a private list.

Decisions should be made in accordance with the *mission, vision and values* of the scikit-image project.

Scikit-image uses a "consensus seeking" process for making decisions. The group tries to find a resolution that has no open objections among core developers. Core developers are expected to distinguish between fundamental objections to a proposal and minor perceived flaws that they can live with, and not hold up the decision-making process for the latter. If no option can be found without objections, the decision is escalated to the SC, which will itself use consensus seeking to come to a resolution. In the unlikely event that there is still a deadlock, the proposal will move forward if it has the support of a simple majority of the SC. Any vote must be backed by a *scikit-image proposal (SKIP)*.

Decisions (in addition to adding core developers and SC membership as above) are made according to the following rules:

- **Minor documentation changes**, such as typo fixes, or addition / correction of a sentence (but no change of the scikit-image.org landing page or the "about" page), require approval by a core developer *and* no disagreement or requested changes by a core developer on the issue or pull request page (lazy consensus). Core developers

are expected to give “reasonable time” to others to give their opinion on the pull request if they’re not confident others would agree.

- **Code changes and major documentation changes** require agreement by *two* core developers *and* no disagreement or requested changes by a core developer on the issue or pull-request page (lazy consensus).
- **Changes to the API principles** require a *Improvement proposals (SKIPs)* and follow the decision-making process outlined above.
- **Changes to this governance model or our mission, vision, and values** require a *Improvement proposals (SKIPs)* and follow the decision-making process outlined above, *unless* there is unanimous agreement from core developers on the change.

If an objection is raised on a lazy consensus, the proposer can appeal to the community and core developers and the change can be approved or rejected by escalating to the SC, and if necessary, a SKIP (see below).

Improvement proposals (SKIPs)

For all votes, a formal proposal must have been made public and discussed before the vote. After discussion has taken place, the key advocate of the proposal must create a consolidated document summarizing the discussion, called a SKIP, on which the core team votes. The lifetime of a SKIP detailed in *SKIP 0 — Purpose and Process*.

A list of all existing SKIPS is available [here](#).

Copyright

This document is based on the [scikit-learn governance document](#) and is placed in the public domain.

SKIP 2 — scikit-image mission statement

Author

Juan Nunez-Iglesias <juan.nunez-iglesias@monash.edu>

Author

Stéfan van der Walt <stefanv@berkeley.edu>

Author

Josh Warner

Author

François Boulogne

Author

Emmanuelle Gouillart

Author

Mark Harfouche

Author

Lars Grüter

Author

Egor Panfilov

Author

Gregory Lee

Status

Active

Type

Process

Created

2018-12-08

Resolved

Resolution

skimage-Version

0.16

Abstract

scikit-image should adopt the document below as its mission statement. This statement will feature prominently in the scikit-image home page and readme, as well as the contributor and core developer guides. Decisions about the API and the future of the library would be referenced against this document. (See *SKIP 1 — scikit-image governance and decision-making*.)

In July 2018, I (Juan) published a blog post that broadly outlined what I would want from a roadmap for scikit-image¹, but requested comments from the community before it would be finalized. I consider that we have collected comments for long enough and can move forward with formal adoption. Most comments were positive, so below I'll just summarize the “negative” comments under “rejected ideas”.

Detailed description

(Or: What problem does this proposal solve?)

Over the past few years, scikit-image has been slightly “adrift”, with new and old contributors coming in, adding what small bits they need at the time, and disappearing again for a while. This is *fine* and we want to encourage more of it, but it also lacks direction. Additionally, without a concerted roadmap to concentrate effort, many of these contributions just fall by the wayside, as it is difficult for new contributors to live up to our stringent (and largely unwritten) standards of code.

¹ <https://ilovesymposia.com/2018/07/13/the-road-to-scikit-image-1-0/>

Implementation

Our mission

scikit-image aims to be the reference library for scientific image analysis in Python. We accomplish this by:

- being **easy to use and install**. We are careful in taking on new dependencies, and sometimes cull existing ones, or make them optional. All functions in our API have thorough docstrings clarifying expected inputs and outputs.
- providing a **consistent API**. Conceptually identical arguments have the same name and position in a function signature.
- **ensuring correctness**. Test coverage is close to 100% and code is reviewed by at least two core developers before being included in the library.
- **caring for users' data**. We have a functional² API and don't modify input arrays unless explicitly directed to do so.
- promoting **education in image processing**, with extensive pedagogical documentation.

Our values

- We are inclusive. We continue to welcome and mentor newcomers who are making their first contribution.
- We are community-driven. Decisions about the API and features are driven by our users' requirements, not by the whims of the core team. (See *SKIP 1 — scikit-image governance and decision-making*.)
- We serve scientific applications primarily, over "consumer" image editing in the vein of Photoshop or GIMP. This often means prioritizing n-dimensional data support, and rejecting implementations of "flashy" filters that have little scientific value.
- We value simple, readable implementations over getting every last ounce of performance. Readable code that is easy to understand, for newcomers and maintainers alike, makes it easier to contribute new code as well as prevent bugs. This means that we will prefer a 20% slowdown if it reduces lines of code two-fold, for example.
- We value education and documentation. All functions should have NumPy-style docstrings³, preferably with examples, as well as gallery examples that showcase how that function is used in a scientific application. Core developers take an active role in finishing documentation examples.
- We don't do magic. We use NumPy arrays instead of fancy façade objects¹², and we prefer to educate users rather than make decisions on their behalf. This does not preclude sensible defaults.⁴

This document

Much in the same way that the Zen of Python⁵ and PEP8 guide style and implementation details in most Python code, this guide is meant to guide decisions about the future of scikit-image, be it in terms of code style, whether to accept new functionality, or whether to take on new dependencies, among other things.

² https://en.wikipedia.org/wiki/Functional_programming

³ https://docs.scipy.org/doc/numpy/docs/howto_document.html

¹² The use of NumPy arrays was the most supported of the statement's components, together with the points about inclusivity, mentorship, and documentation. We had +1s from Mark Harfouche, Royi Avital, and Greg Lee, among others.

⁴ <https://forum.image.sc/t/request-for-comment-road-to-scikit-image-1-0/20099/4>

⁵ <https://www.python.org/dev/peps/pep-0020/>

References

To find out more about the history of this document, please read the following:

- Original blog post⁷
- The GitHub issue⁶
- The image.sc forum post⁷
- The SKIP GitHub pull request⁸

Backward compatibility

This SKIP formalizes what had been the unwritten culture of scikit-image, so it does not raise any backward compatibility concerns.

Alternatives

Two topics in the original discussion were ultimately rejected, detailed below:

Handling metadata

In my original post, I suggested that scikit-image should have some form of metadata handling before 1.0. Among others, Mark Harfouche, Curtis Rueden, and Dan Allan all advised that (a) maybe scikit-image doesn't *need* to handle metadata, and can instead focus on being a robust lower-level library that another like XArray can use to include metadata handling, and (b) anyway, metadata support can be added later without breaking the 1.0 API. I think these are very good points and furthermore metadata handling is super hard and I don't mind keeping this off our plate for the moment.

Magical thinking

Philipp Hanslovsky suggested⁹ that, regarding "doing magic", it is advisable in some contexts, and a good solution is to provide a magic layer built on top of the non-magical one. I agree with this assessment, but, until 1.0, scikit-image should remain the non-magic layer.

Discussion

See References below.

⁶ <https://github.com/scikit-image/scikit-image/issues/3263>

⁷ <https://forum.image.sc/t/request-for-comment-road-to-scikit-image-1-0/20099>

⁸ <https://github.com/scikit-image/scikit-image/pull/3585>

⁹ <https://forum.image.sc/t/request-for-comment-road-to-scikit-image-1-0/20099/3>

References

Copyright

This document is dedicated to the public domain with the Creative Commons CC0 license¹⁰. Attribution to this source is encouraged where appropriate, as per CC0+BY¹¹.

SKIP 3 — Transitioning to scikit-image 1.0

Author

Juan Nunez-Iglesias <juan.nunez-iglesias@monash.edu>

Status

Final

Type

Standards Track

Created

2021-07-15

Resolved

2021-09-13

Resolution

Rejected

Version effective

None

Abstract

scikit-image is preparing to release version 1.0. This is potentially an opportunity to clean up the API, including backwards incompatible changes. Some of these changes involve changing return values without changing function signatures, which can ordinarily only be done by adding an otherwise useless keyword argument (such as `new_return_style=True`) whose default value changes over several releases. The result is *still* a backwards incompatible change, but made over a longer time period.

Despite being in beta and in a 0.x series of releases, scikit-image is used extremely broadly, and any backwards incompatible changes are likely to be disruptive. This SKIP proposes a process to ensure that the community is aware of upcoming changes, and can adapt their libraries *or* their declared scikit-image version dependencies accordingly.

¹⁰ CC0 1.0 Universal (CC0 1.0) Public Domain Dedication, <https://creativecommons.org/publicdomain/zero/1.0/>

¹¹ <https://dancohen.org/2013/11/26/cc0-by/>

Motivation and Scope

scikit-image has grown organically over the past 12 years, with functionality being added by a broad community of contributors from different backgrounds. This has resulted in various parts of the API being inconsistent: for example, `skimage.transform.warp` inverts the order of coordinates, so that a translation of (45, 32) actually moves the values in a NumPy array by 32 along the 0th axis, and 45 along the 1st, *but only in 2D*.

Additionally, as our user base has grown, it has become apparent that certain early API choices turned out to be more confusing than helpful. For example, scikit-image will automatically convert images to various data types, *rescaling them in the process*. A `uint8` image in the range [0, 255] will automatically be converted to a `float64` image in [0, 1]. This might initially seem reasonable, but, for consistency, `uint16` images in [0, 65535] are rescaled to [0, 1] floats, and `uint16` images with 12-bit range in [0, 4095], which are common in microscopy, are rescaled to [0, 0.0625]. These silent conversions have resulted in much user confusion.

Changing this convention would require adding a `preserve_range=` keyword argument to almost *all* scikit-image functions, whose default value would change from `False` to `True` over 4 versions. Eventually, the change would be backwards-incompatible, no matter how gentle we made the deprecation curve.

Given the accumulation of potential API changes that have turned out to be too burdensome and noisy to fix with a standard deprecation cycle, principally because they involve changing function outputs for the same inputs, it makes sense to make all those changes in a transition to version 1.0 – semantic versioning, which we use, explicitly allows breaking API changes on major version updates⁶. However, we must acknowledge that (1) an enormous number of projects depend on scikit-image and would thus be affected by backwards incompatible changes, and (2) it is not yet common practice in the scientific Python community to put upper version bounds on dependencies, so it is very unlikely that anyone used `scikit-image<1.*` in their dependency list (though this is slowly changing⁵).

Given the above, we need to come up with a way to notify all our users that this change is coming, while also allowing them to silence any warnings once they have been noted.

Detailed description

It is beyond the scope of this document to list all of the proposed API changes for scikit-image 1.0, many of which have yet to be decided upon. Indeed, the scope and ambition of the 1.0 transition could grow if this SKIP is accepted. The SKIP instead proposes a mechanism for warning users about upcoming breaking changes. A meta-issue tracking the proposed changes can be found on GitHub, [scikit-image/scikit-image#5439](https://github.com/scikit-image/scikit-image/issues/5439)⁷. Some examples are briefly included below for illustrative purposes:

- Stop rescaling input arrays when the `dtype` must be coerced to `float`.
- Stop swapping coordinate axis order in different contexts, such as drawing or warping.
- Allow automatic return of non-NumPy types, so long as they are coercible to NumPy with `numpy.asarray`.
- Harmonizing similar parameters in different functions to have the same name; for example, we currently have `random_seed`, `random_state`, `seed`, or `sample_seed` in different functions, all to mean the same thing.
- Changing `measure.regionprops` to return a dictionary instead of a list.
- Combine functions that have the same purpose, such as `watershed`, `slic`, or `felzenszwalb`, into a common namespace. This would make it easier for new users to find out which functions they should try out for a specific task.

The question is, how do we make this transition while causing as little disruption as possible?

This document proposes releasing 0.19 as the final 0.x series release, then immediately releasing a nearly identical 0.20 release that warns users about breaking changes in 1.0, thus giving them an opportunity to pin their scikit-image

⁶ <https://semver.org/>

⁵ <https://github.com/scipy/scipy/pull/12862>

⁷ <https://github.com/scikit-image/scikit-image/issues/5439>

dependency to 0.19.x. The warning would also point users to a transition guide to prepare their code for 1.0. See *Implementation* for details.

This approach ensures that all users get ample warning, and a chance to ensure that their scripts and libraries will continue to work after 1.0 is released. Users who don't have the time or inclination to make the transition will be able to pin their dependencies correctly. Those who prefer to be on the cutting edge will also be able to plan around the 1.0 release and update their code correctly, in sync with scikit-image.

Related Work

pandas released 1.0.0 in January 2020, including many backwards-incompatible API changes³. `scipy` released version 1.0 in 2017, but, given its stage of maturity and position at the base of the scientific Python ecosystem, opted not to make major breaking changes⁴. However, SciPy has adopted a policy of adding upper-bounds on dependencies⁵, acknowledging that the ecosystem as a whole makes backwards incompatible changes on a 2 version deprecation cycle.

Implementation

The details of the proposal are as follows:

- scikit-image 0.19 will be the final *true* 0.x release. It contains some new features, bug fixes, and several API changes following on from deprecations in 0.17.
- shortly after 0.19, we release 0.20, which is identical except that it emits a warning at import time. The warning reads something like the following: “scikit-image 1.0 will be released later this year and will contain breaking changes. To ensure your code keeps running, please install `scikit-image<=0.19.*`. To silence this warning but still depend on scikit-image after 1.0 is released, install `scikit-image!=0.20.*`.” The warning also contains a link for further details, and instructions for managing the dependency in both conda and pip environments.
- After 0.20, we make all the API changes we need, without deprecation cycles. Importantly, for every API change, we add a line to a “scikit-image 1.0 transition guide” in the documentation, which maps every changed functionality in the library from its old form to its new form. These changes are tracked on a GitHub issue⁷ and in the 1.0 milestone⁸.
- Once the transition has happened in the repository, we release 1.0.0a0, an alpha release which contains a global warning pointing to the transition guide, as well as all of the new functionality. We also release 0.21, which contains the same warning but is functionally identical to 0.19. This gives authors who chose to pin to `scikit-image!=0.20.*` a chance to make the migration to 1.0.
- After at least one month, we release 1.0.
- We continue to maintain a 0.19.x branch with bug fixes for a year, in order to give users time to transition to the new API.

³ <https://pandas.pydata.org/pandas-docs/stable/whatsnew/v1.0.0.html#backwards-incompatible-api-changes>

⁴ <https://docs.scipy.org/doc/scipy/release.1.0.0.html>

⁸ <https://github.com/scikit-image/scikit-image/milestones/1.0>

Backward compatibility

This proposal breaks backwards compatibility in numerous places in the library.

Alternatives

New package naming

Instead of breaking compatibility in the `scikit-image` package, we could leave that package at 0.19, and release a *new* package, e.g. `scikit-image1`, which starts at 1.0 and imports as `skimage1`. This would obviate the need for users to pin their scikit-image version — users depending on `skimage` 0.x would be able to use that library “in perpetuity.”

Ultimately, the core developers felt that this approach could unnecessarily fragment the community, between those that continue using 0.19 and those that shift to 1.0. Ultimately, the transition of downstream code to 1.0 would be equally painful as the proposed approach, but the pressure to make the switch would be decreased, as everyone installing `scikit-image` would still get the old version.

Continuous deprecation over multiple versions

This transition could occur gradually over many versions. For example, for functions automatically converting and rescaling float inputs, we could add a `preserve_range` keyword argument that would initially default to False, but the default value of False would be deprecated, with users getting a warning to switch to True. After the switch, we could (optionally) deprecate the argument, arriving, after a further two releases, at the same place: scikit-image no longer rescales data automatically, there are no unnecessary keyword arguments lingering all over the API.

Of course, this kind of operation would have to be done simultaneously over all of the above proposed changes.

Ultimately, the core team felt that this approach generates more work for both the scikit-image developers and the developers of downstream libraries, for dubious benefit: ultimately, later versions of scikit-image will still be incompatible with prior versions, although over a longer time scale.

Not making the proposed API changes

Another possibility is to reject backwards incompatible API changes outright, except in extreme cases. The core team feels that this is essentially equivalent to pinning the library at 0.19.

Discussion

In early July 2021, the core team held a series of meetings to discuss this approach. The minutes of this meeting are in the scikit-image meeting notes repository⁹.

Ongoing discussion will happen on the user forum¹⁰, the developer forum¹¹, and GitHub discussion⁷. Specific links to relevant posts will be added to this document before acceptance.

⁹ <https://github.com/scikit-image/meeting-notes/blob/main/2021/july-api-meetings.md>

¹⁰ <https://forum.image.sc/tag/scikit-image>

¹¹ <https://discuss.scientific-python.org/c/contributor/skimage>

Resolution

This SKIP was discussed most extensively in a thread on the mailing list in July 2021¹². In the end, many and core developers felt that this plan posed too big a risk of either changing code behavior silently or eroding goodwill in the community, or both. Matthew Brett wrote¹³:

I'm afraid I wasn't completely sure whether the 1.0 option would result in breaking what I call the Konrad Hinsen rule for scientific software:

“” Under (virtually) no circumstances should new versions of a scientific package silently give substantially different results for the same function / method call from a previous version of the package. “”

Matthew further wrote¹⁴ that if we *don't* break the Hinsen rule, but instead break users' unpinned scripts, we will lose a lot of goodwill from the community:

If you make all these break (if they are lucky) or give completely wrong results, it's hard to imagine you aren't going to cause significant damage to the rest-of-iceberg body of users who are not on the mailing list.

Riadh Fezzani, one of our core developers, felt strongly that SemVer⁷ was sufficient to protect users¹⁵:

In scikit-image, we adopted the semantic versioning as it is largely adopted in the engineering community. This convention manages API breaking and that's what we are doing by releasing v1.0

Even taking this view, though, it cannot address the issue of external scikit-image “documentation”, such as a decade's worth of accumulated StackOverflow answers, that would be made obsolete by a breaking 1.0 release, as pointed out by Josh Warner¹⁶:

It's also worth considering that there is a substantial corpus of scikit-image teaching material out there. The majority we do not control, so cannot be updated or edited. The first hits on YouTube for tutorials are not the most recent, but older ones with lots of views.

Nor can it address the issue of *gradually* migrating a code base from the old API to the new API, as pointed out by Tom Caswell¹⁷:

Put another way, you do not want to put a graduate student in the position of saying “I _want_ to use the new API, but I have 10k LoC of inherited code using the old API

Ultimately, all these concerns add up to a compelling case to rejecting the SKIP. Juan Nunez-Iglesias wrote on the mailing list¹⁸:

My proposal going forward is to reject SKIP-3 and create a SKIP-4 proposing the skimage2 package.

The SKIP is therefore rejected.

¹² <https://mail.python.org/archives/list/scikit-image@python.org/thread/DSV6PEYVJ4RZRUWWV5SBNF7FFRERTSCF/>

¹³ <https://mail.python.org/archives/list/scikit-image@python.org/message/UYARUQM5LBWXIAWBAPNHIQIDRKUUDTEK/>

¹⁴ <https://mail.python.org/archives/list/scikit-image@python.org/message/63ZGG7DY5SWVM62XASHMCPFAG6KPJCMT/>

¹⁵ <https://mail.python.org/archives/list/scikit-image@python.org/message/HXI7YVCN6IFF5TL54JBP5QRUDHKTTYRR/>

¹⁶ <https://mail.python.org/archives/list/scikit-image@python.org/message/HRZGMOJLD2WDIO3JXQV3PRWKIUOVOF7P/>

¹⁷ <https://mail.python.org/archives/list/scikit-image@python.org/message/GFXBQYKDACDCH7BGNEGOU7LKHR2LPFX6/>

¹⁸ <https://mail.python.org/archives/list/scikit-image@python.org/message/5J4W63BXFQTT4GHPTZFH3AM4QHAXOW5R/>

References and Footnotes

All SKIPs should be declared as dedicated to the public domain with the CC0 license¹, as in *Copyright*, below, with attribution encouraged with CC0+BY².

Copyright

This document is dedicated to the public domain with the Creative Commons CC0 license³. Attribution to this source is encouraged where appropriate, as per CC0+BY⁴.

SKIP 4 — Transitioning to scikit-image 2.0

Author

Juan Nunez-Iglesias <juan.nunez-iglesias@monash.edu>

Author

Lars Grüter

Status

Draft

Type

Standards Track

Created

2022-04-08

Resolved

<null>

Resolution

<null>

Version effective

None

Abstract

scikit-image is preparing to release version 1.0. This *was seen* as an opportunity to clean up the API, including backwards incompatible changes. Some of these changes involve changing return values without changing function signatures, which can ordinarily only be done by adding an otherwise useless keyword argument (such as `new_return_style=True`) whose default value changes over several releases. The result is *still* a backwards incompatible change, but made over a longer time period.

¹ CC0 1.0 Universal (CC0 1.0) Public Domain Dedication, <https://creativecommons.org/publicdomain/zero/1.0/>

² <https://dancohen.org/2013/11/26/cc0-by/>

Despite being in beta and in a 0.x series of releases, scikit-image is used extremely broadly, and any backwards incompatible changes are likely to be disruptive. Given the rejection of *SKIP-3*, this document proposes an alternative pathway to create a new API. The new pathway involves the following steps:

- Any pending deprecations that were scheduled for v0.20 and v0.21 are finalised (the new API suggested by deprecation messages in v0.19 becomes the only API).
- This is released as 1.0.
- At this point, the branch `main` changes the package and import names to `skimage2`, and the API is free to evolve.

Further motivation for the API changes is explained below, and largely duplicated from *SKIP-3*.

Motivation and Scope

scikit-image has grown organically over the past 12+ years, with functionality being added by a broad community of contributors from different backgrounds. This has resulted in various parts of the API being inconsistent: for example, `skimage.transform.warp` inverts the order of coordinates, so that a translation of (45, 32) actually moves the values in a NumPy array by 32 along the 0th axis, and 45 along the 1st, *but only in 2D*. In 3D, a translation of (45, 32, 77) moves the values in each axis by the number in the corresponding position.

Additionally, as our user base has grown, it has become apparent that certain early API choices turned out to be more confusing than helpful. For example, scikit-image will automatically convert images to various data types, *rescaling them in the process*. A `uint8` image in the range [0, 255] will automatically be converted to a `float64` image in [0, 1]. This might initially seem reasonable, but, for consistency, `uint16` images in [0, 65535] are rescaled to [0, 1] floats, and `uint16` images with 12-bit range in [0, 4095], which are common in microscopy, are rescaled to [0, 0.0625]. These silent conversions have resulted in much user confusion.

Changing this convention would require adding a `preserve_range=` keyword argument to almost *all* scikit-image functions, whose default value would change from `False` to `True` over 4 versions. Eventually, the change would be backwards-incompatible, no matter how gentle we made the deprecation curve.

Other major functions, such as `skimage.measure.regionprops`, could use an API tweak, for example by returning a dictionary mapping labels to properties, rather than a list.

Given the accumulation of potential API changes that have turned out to be too burdensome and noisy to fix with a standard deprecation cycle, principally because they involve changing function outputs for the same inputs, it makes sense to make all those changes in a transition to version 2.0.

Although semantic versioning⁶ technically allows API changes with major version bumps, we must acknowledge that (1) an enormous number of projects depend on scikit-image and would thus be affected by backwards incompatible changes, and (2) it is not yet common practice in the scientific Python community to put upper version bounds on dependencies, so it is very unlikely that anyone used `scikit-image<1.*` or `scikit-image<2.*` in their dependency list. This implies that releasing a version 2.0 of scikit-image with breaking API changes would disrupt a large number of users. Additionally, such wide-sweeping changes would invalidate a large number of StackOverflow and other user guides. Finally, releasing a new version with a large number of changes prevents users from gradually migrating to the new API: an old code base must be migrated wholesale because it is impossible to depend on both versions of the API. This would represent an enormous barrier of entry for many users.

Given the above, this SKIP proposes that we release a new package where we can apply everything we have learned from over a decade of development, without disrupting our existing user base.

⁶ <https://semver.org/>

Detailed description

It is beyond the scope of this document to list all of the proposed API changes for skimage2, many of which have yet to be decided upon. Indeed, the scope and ambition of the 2.0 transition could grow if this SKIP is accepted. This SKIP instead proposes a mechanism for managing the transition without breaking users' code. A meta-issue tracking the proposed changes can be found on GitHub, [scikit-image/scikit-image#5439](https://github.com/scikit-image/scikit-image/issues/5439)⁷. Some examples are briefly included below for illustrative purposes:

- Stop rescaling input arrays when the dtype must be coerced to float.
- Stop swapping coordinate axis order in different contexts, such as drawing or warping.
- Allow automatic return of non-NumPy types, so long as they are coercible to NumPy with `numpy.asarray`.
- Harmonizing similar parameters in different functions to have the same name; for example, we currently have `random_seed`, `random_state`, `seed`, or `sample_seed` in different functions, all to mean the same thing.
- Changing `measure.regionprops` to return a dictionary instead of a list.
- Combine functions that have the same purpose, such as `watershed`, `slic`, or `felzenszwalb`, into a common namespace. This would make it easier for new users to find out which functions they should try out for a specific task. It would also help the community grow around common APIs, where now scikit-image APIs are essentially unique for each function.

To make this transition with a minimum amount of user disruption, this SKIP proposes releasing a new library, skimage2, that would replace the existing library, *but only if users explicitly opt-in*. Additionally, by releasing a new library, users could depend *both* on scikit-image (1.0) and on skimage2, allowing users to migrate their code progressively.

Related Work

pandas released 1.0.0 in January 2020, including many backwards-incompatible API changes³. `scipy` released version 1.0 in 2017, but, given its stage of maturity and position at the base of the scientific Python ecosystem, opted not to make major breaking changes⁴. However, SciPy has adopted a policy of adding upper-bounds on dependencies⁵, acknowledging that the ecosystem as a whole makes backwards incompatible changes on a 2 version deprecation cycle.

Several libraries have successfully migrated their user community to a new namespace with a version number on it, such as OpenCV (imported as `cv2`) and BeautifulSoup (imported as `bs4`), Jinja (`jinja2`) and psycopg (currently imported as `psycopg2`). Further afield, R's `ggplot` is used as `ggplot2`.

Implementation

The details of the proposal are as follows:

- scikit-image 0.19 will be followed by scikit-image 1.0. Every deprecation message will be removed from 1.0, and the API will be considered the scikit-image 1.0 API.
- After 1.0, the main branch will be changed to (a) change the import name to skimage2, (b) change the package name to skimage2, and (c) change the version number to 2.0-dev.
- There will be *no* “scikit-image” package on PyPI with version 2.0. Users who `pip install scikit-image` will always get the 1.x version of the package. To install scikit-image 2.0, users will need to `pip install skimage2`, `conda install skimage2`, or similar.
- After consensus has been reached on the new API, skimage2 will be released.

⁷ <https://github.com/scikit-image/scikit-image/issues/5439>

³ <https://pandas.pydata.org/pandas-docs/stable/whatsnew/v1.0.0.html#backwards-incompatible-api-changes>

⁴ <https://docs.scipy.org/doc/scipy/reference/release.1.0.0.html>

⁵ <https://github.com/scipy/scipy/pull/12862>

- Following the release of `skimage2`, a release of scikit-image 1.1 is made. This release is identical to 1.0 (including bugfixes) but will advise users to either (a) upgrade to `skimage2` or (b) pin the package to `scikit-image<1.1` to avoid the warning.
- scikit-image 1.0.x and 1.1.x will receive critical bug fixes for an unspecified period of time, depending on the severity of the bug and the amount of effort involved.

Backward compatibility

This proposal breaks backward compatibility in numerous places in the library. However, it does so in a new namespace, so that this proposal does not raise backward compatibility concerns for our users. That said, the authors will attempt to limit the number of backward incompatible changes to those likely to substantially improve the overall user experience. It is anticipated that porting `skimage` code to `skimage2` will be a straightforward process and we will publish a user guide for making the transition by the time of the `skimage2` release. Users will be notified about these resources - among other things - by a warning in scikit-image 1.1.

Alternatives

Releasing the new API in the same package using semantic versioning

This is *SKIP-3*, which was rejected after discussion with the community.

Continuous deprecation over multiple versions

This transition could occur gradually over many versions. For example, for functions automatically converting and rescaling float inputs, we could add a `preserve_range` keyword argument that would initially default to False, but the default value of False would be deprecated, with users getting a warning to switch to True. After the switch, we could (optionally) deprecate the argument, arriving, after a further two releases, at the same place: scikit-image no longer rescales data automatically, there are no unnecessary keyword arguments lingering all over the API.

Of course, this kind of operation would have to be done simultaneously over all of the above proposed changes.

Ultimately, the core team felt that this approach generates more work for both the scikit-image developers and the developers of downstream libraries, for dubious benefit: ultimately, later versions of scikit-image will still be incompatible with prior versions, although over a longer time scale.

A single package containing both versions

Since the import name is changing, it would be possible to make a single package with both the `skimage` and `skimage2` namespaces shipping together, at least for some time. This option is attractive but it implies longer-term maintenance of the 1.0 namespace, for which we might lack maintainer time, or a long deprecation cycle for the 1.0 namespace, which would ultimately result in a lot of unhappy users getting deprecation messages from their scikit-image use.

Not making the proposed API changes

Another possibility is to reject backwards incompatible API changes outright, except in extreme cases. The core team feels that this is essentially equivalent to pinning the library at 0.19.

“scikit-image2” as the new package name

The authors acknowledge that the new names should be chosen with care to keep the disruption to scikit-image’s user base and community as small as possible. However, to protect users without upper version constraints from accidentally upgrading to the new API, the package name `scikit-image` must be changed. Changing the import name `skimage` is similarly advantageous because it allows using both APIs in the same environment.

This document suggests just `skimage2` as the single new name for scikit-image’s API version 2.0, both for the import name and the name on PyPI, conda-forge and elsewhere. The following arguments were given in favor of this:

- Only one new name is introduced with the project thereby keeping the number of associated names as low as possible.
- With this change, the import and package name match.
- Users might be confused whether they should install `scikit-image2` or `scikit-image-2`. It was felt that `skimage2` avoids this confusion.
- Users who know what `skimage` is and see `skimage2` in an install instruction somewhere, will likely be able to infer that it is a newer version of the package.
- It is unlikely that users will be aware of the new API 2.0 but not of the new package name. A proposed release of scikit-image 1.1 might point users to `skimage2` during the installation and update process and thereby clearly communicate the successors name.

The following arguments were made against naming the package `skimage2`:

- According to the “Principle of least astonishment”, `scikit-image2` might be considered the least surprising evolution of the package name.
- It breaks with the convention that is followed by other scikits including scikit-image. (It was pointed out that this convention has not been true for some time and introducing a version number in the name is a precedent anyway.)

The earlier section “Related Work” describes how other projects dealt with similar problems.

Discussion

This SKIP is the result of discussion of *SKIP-3*. See the “Resolution” section of that document for further background on the motivation for this SKIP.

Resolution

References and Footnotes

All SKIPS should be declared as dedicated to the public domain with the CC0 license¹, as in *Copyright*, below, with attribution encouraged with CC0+BY².

¹ CC0 1.0 Universal (CC0 1.0) Public Domain Dedication, <https://creativecommons.org/publicdomain/zero/1.0/>

² <https://dancohen.org/2013/11/26/cc0-by/>

Copyright

This document is dedicated to the public domain with the Creative Commons CC0 license⁷. Attribution to this source is encouraged where appropriate, as per CC0+BY⁷.

SKIP X — Template and Instructions

Author

<list of authors' real names and, optionally, email addresses>

Status

<Draft | Active | Accepted | Deferred | Rejected | Withdrawn | Final | Superseded>

Type

<Standards Track | Process>

Created

<date created on, in yyyy-mm-dd format>

Resolved

<date resolved, in yyyy-mm-dd format>

Resolution

<url> (required for Accepted | Rejected | Withdrawn)

Version effective

<version-number> (for accepted SKIPs)

Abstract

The abstract should be a short description of what the SKIP will achieve.

Motivation and Scope

This section describes the need for the proposed change. It should describe the existing problem, who it affects, what it is trying to solve, and why. This section should explicitly address the scope of and key requirements for the proposed change.

Detailed description

This section should provide a detailed description of the proposed change. It should include examples of how the new functionality would be used, intended use-cases, and pseudocode illustrating its use.

Related Work

This section should list relevant and/or similar technologies, possibly in other libraries. It does not need to be comprehensive, just list the major examples of prior and relevant art.

Implementation

This section lists the major steps required to implement the SKIP. Where possible, it should be noted where one step is dependent on another, and which steps may be optionally omitted. Where it makes sense, each step should include a link related pull requests as the implementation progresses.

Any pull requests or developmt branches containing work on this SKIP should be linked to from here. (A SKIP does not need to be implemented in a single pull request if it makes sense to implement it in discrete phases).

Backward compatibility

This section describes the ways in which the SKIP breaks backward compatibility.

Alternatives

If there were any alternative solutions to solving the same problem, they should be discussed here, along with a justification for the chosen approach.

Discussion

This section may just be a bullet list including links to any discussions regarding the SKIP, but could also contain additional comments about that discussion:

- This includes links to discussion forum threads or relevant GitHub discussions.

References and Footnotes

All SKIPS should be declared as dedicated to the public domain with the CC0 license¹, as in *Copyright*, below, with attribution encouraged with CC0+BY².

¹ CC0 1.0 Universal (CC0 1.0) Public Domain Dedication, <https://creativecommons.org/publicdomain/zero/1.0/>

² <https://dancohen.org/2013/11/26/cc0-by/>

Copyright

This document is dedicated to the public domain with the Creative Commons CC0 license⁷. Attribution to this source is encouraged where appropriate, as per CC0+BY⁷.

1.6 About

Get to know the project and the community. Learn where we are going and how we work together.

1.6.1 Our mission

scikit-image aims to be the reference library for scientific image analysis in Python. We accomplish this by:

- being **easy to use and install**. We are careful in taking on new dependencies, and sometimes cull existing ones, or make them optional. All functions in our API have thorough docstrings clarifying expected inputs and outputs.
- providing a **consistent API**. Conceptually identical arguments have the same name and position in a function signature.
- **ensuring correctness**. Test coverage is close to 100% and code is reviewed by at least two core developers before being included in the library.
- **caring for users' data**. We have a [functional API](#) and don't modify input arrays unless explicitly directed to do so.
- promoting **education in image processing**, with extensive pedagogical documentation.

Our values

- We are inclusive. We continue to welcome and mentor newcomers who are making their first contribution.
- We are community-driven. Decisions about the API and features are driven by our users' requirements, not by the whims of the core team. (See [SKIP 1 — scikit-image governance and decision-making](#).)
- We serve scientific applications primarily, over “consumer” image editing in the vein of Photoshop or GIMP. This often means prioritizing n-dimensional data support, and rejecting implementations of “flashy” filters that have little scientific value.
- We value simple, readable implementations over getting every last ounce of performance. Readable code that is easy to understand, for newcomers and maintainers alike, makes it easier to contribute new code as well as prevent bugs. This means that we will prefer a 20% slowdown if it reduces lines of code two-fold, for example.
- We value education and documentation. All functions should have NumPy-style [docstrings](#), preferably with examples, as well as gallery examples that showcase how that function is used in a scientific application. Core developers take an active role in finishing documentation examples.
- We don't do magic. We use NumPy arrays instead of fancy façade objects¹, and we prefer to educate users rather than make decisions on their behalf. This does not preclude [sensible defaults](#).

¹ The use of NumPy arrays was the most supported of the statement's components, together with the points about inclusivity, mentorship, and documentation. We had +1s from Mark Harfouche, Royi Avital, and Greg Lee, among others.

This document

Much in the same way that the [Zen of Python](#) and PEP8 guide style and implementation details in most Python code, this guide is meant to guide decisions about the future of scikit-image, be it in terms of code style, whether to accept new functionality, or whether to take on new dependencies, among other things.

References

To find out more about the history of this document, please read the following:

- [Original blog post](#)
- [The GitHub issue](#)
- [The image.sc forum post](#)
- [The SKIP GitHub pull request](#)

Copyright

This document is dedicated to the public domain with the Creative Commons CC0 [license](#). Attribution to this source is encouraged where appropriate, as per [CC0+BY](#).

1.6.2 scikit-image Code of Conduct

scikit-image adopts the [SciPy code of conduct](#).

Introduction

This code of conduct applies to all spaces managed by the scikit-image project, including all public and private forums, issue trackers, wikis, blogs, Twitter, and any other communication channel used by our community. The scikit-image project does not organise in-person events, however events related to our community should have a code of conduct similar in spirit to this one.

This code of conduct should be honored by everyone who participates in the scikit-image community formally or informally, or claims any affiliation with the project, in any project-related activities and especially when representing the project, in any role.

This code is not exhaustive or complete. It serves to distill our common understanding of a collaborative, shared environment and goals. Please try to follow this code in spirit as much as in letter, to create a friendly and productive environment that enriches the surrounding community.

Specific Guidelines

We strive to:

1. Be open. We invite anyone to participate in our community. We prefer to use public methods of communication for project-related messages, unless discussing something sensitive. This applies to messages for help or project-related support, too; not only is a public support request much more likely to result in an answer to a question, it also ensures that any inadvertent mistakes in answering are more easily detected and corrected.

2. Be empathetic, welcoming, friendly, and patient. We work together to resolve conflict, and assume good intentions. We may all experience some frustration from time to time, but we do not allow frustration to turn into a personal attack. A community where people feel uncomfortable or threatened is not a productive one.
3. Be collaborative. Our work will be used by other people, and in turn we will depend on the work of others. When we make something for the benefit of the project, we are willing to explain to others how it works, so that they can build on the work to make it even better. Any decision we make will affect users and colleagues, and we take those consequences seriously when making decisions.
4. Be inquisitive. Nobody knows everything! Asking questions early avoids many problems later, so we encourage questions, although we may direct them to the appropriate forum. We will try hard to be responsive and helpful.
5. Be careful in the words that we choose. We are careful and respectful in our communication and we take responsibility for our own speech. Be kind to others. Do not insult or put down other participants. We will not accept harassment or other exclusionary behaviour, such as:
 - Violent threats or language directed against another person.
 - Sexist, racist, or otherwise discriminatory jokes and language.
 - Posting sexually explicit or violent material.
 - Posting (or threatening to post) other people's personally identifying information ("doxing").
 - Sharing private content, such as emails sent privately or non-publicly, or unlogged forums such as IRC channel history, without the sender's consent.
 - Personal insults, especially those using racist or sexist terms.
 - Unwelcome sexual attention.
 - Excessive profanity. Please avoid swearwords; people differ greatly in their sensitivity to swearing.
 - Repeated harassment of others. In general, if someone asks you to stop, then stop.
 - Advocating for, or encouraging, any of the above behaviour.

Diversity Statement

The scikit-image project welcomes and encourages participation by everyone. We are committed to being a community that everyone enjoys being part of. Although we may not always be able to accommodate each individual's preferences, we try our best to treat everyone kindly.

No matter how you identify yourself or how others perceive you: we welcome you. Though no list can hope to be comprehensive, we explicitly honour diversity in: age, culture, ethnicity, genotype, gender identity or expression, language, national origin, neurotype, phenotype, political beliefs, profession, race, religion, sexual orientation, socioeconomic status, subculture and technical ability, to the extent that these do not conflict with this code of conduct.

Though we welcome people fluent in all languages, scikit-image development is conducted in English.

Standards for behaviour in the scikit-image community are detailed in the Code of Conduct above. Participants in our community should uphold these standards in all their interactions and help others to do so as well (see next section).

Reporting Guidelines

We know that it is painfully common for internet communication to start at or devolve into obvious and flagrant abuse. We also recognize that sometimes people may have a bad day, or be unaware of some of the guidelines in this Code of Conduct. Please keep this in mind when deciding on how to respond to a breach of this Code.

For clearly intentional breaches, report those to the Code of Conduct committee (see below). For possibly unintentional breaches, you may reply to the person and point out this code of conduct (either in public or in private, whatever is most appropriate). If you would prefer not to do that, please feel free to report to the Code of Conduct Committee directly, or ask the Committee for advice, in confidence.

You can report issues to the scikit-image Code of Conduct committee, at skimage-conduct@groups.io. Currently, the committee consists of:

- Emmanuelle Gouillard emmanuelle.gouillard@normalesup.org (chair)
- Carol Willing willingc@gmail.com
- François Boulogne devel@sciunto.org
- Egor Panfilov egor.v.panfilov@gmail.com

If your report involves any members of the committee, or if they feel they have a conflict of interest in handling it, then they will recuse themselves from considering your report. Alternatively, if for any reason you feel uncomfortable making a report to the committee, then you can also contact:

- Any member of the *scikit-image Steering Council*, or
- Senior NumFOCUS staff: conduct@numfocus.org.

Incident reporting resolution & Code of Conduct enforcement

This section summarizes the most important points, more details can be found in the reporting manual.

We will investigate and respond to all complaints. The scikit-image Code of Conduct Committee and the scikit-image Steering Committee (if involved) will protect the identity of the reporter, and treat the content of complaints as confidential (unless the reporter agrees otherwise).

In case of severe and obvious breaches, e.g. personal threat or violent, sexist or racist language, we will immediately disconnect the originator from scikit-image communication channels; please see the manual for details.

In cases not involving clear severe and obvious breaches of this code of conduct, the process for acting on any received code of conduct violation report will be:

1. acknowledge report is received
2. reasonable discussion/feedback
3. mediation (if feedback didn't help, and only if both reporter and reportee agree to this)
4. enforcement via transparent decision (see CoC_resolutions) by the Code of Conduct Committee

The committee will respond to any report as soon as possible, and at most within 72 hours.

Endnotes

We are thankful to the groups behind the following documents, from which we drew content and inspiration:

- [The SciPy project](#)
- [The Apache Foundation Code of Conduct](#)
- [The Contributor Covenant](#)
- [Jupyter Code of Conduct](#)
- [Open Source Guides - Code of Conduct](#)

BIBLIOGRAPHY

- [Zha84] A fast parallel algorithm for thinning digital patterns, T. Y. Zhang and C. Y. Suen, Communications of the ACM, March 1984, Volume 27, Number 3.
- [Lee94] T.-C. Lee, R.L. Kashyap and C.-N. Chu, Building skeleton models via 3-D medial surface/axis thinning algorithms. Computer Vision, Graphics, and Image Processing, 56(6):462-478, 1994.
- [1] <https://web.archive.org/web/20160624145052/http://www.mecourse.com/landinig/software/cdeconv/cdeconv.html>
- [2] A. C. Ruifrok and D. A. Johnston, “Quantification of histochemical staining by color deconvolution,” Anal. Quant. Cytol. Histol., vol. 23, no. 4, pp. 291–299, Aug. 2001.
- [1] https://en.wikipedia.org/wiki/Color_difference
- [2] A. R. Robertson, “The CIE 1976 color-difference formulae,” Color Res. Appl. 2, 7-11 (1977).
- [1] https://en.wikipedia.org/wiki/Color_difference
- [2] [http://www.ece.rochester.edu/~gsharma/ciede2000/ciede2000noteCRNA.pdf DOI:10.1364/AO.33.008069](http://www.ece.rochester.edu/~gsharma/ciede2000/ciede2000noteCRNA.pdf)
- [3] M. Melgosa, J. Quesada, and E. Hita, “Uniformity of some recent color metrics tested with an accurate color-difference tolerance dataset,” Appl. Opt. 33, 8069-8077 (1994).
- [1] https://en.wikipedia.org/wiki/Color_difference
- [2] http://www.brucelindbloom.com/index.html?Eqn_DeltaE_CIE94.html
- [1] https://en.wikipedia.org/wiki/Color_difference
- [2] http://www.brucelindbloom.com/index.html?Eqn_DeltaE_CIE94.html
- [3] F. J. J. Clarke, R. McDonald, and B. Rigg, “Modification to the JPC79 colour-difference formula,” J. Soc. Dyers Colour. 100, 128-132 (1984).
- [1] A. C. Ruifrok and D. A. Johnston, “Quantification of histochemical staining by color deconvolution.,” Analytical and quantitative cytology and histology / the International Academy of Cytology [and] American Society of Cytology, vol. 23, no. 4, pp. 291-9, Aug. 2001.
- [1] https://en.wikipedia.org/wiki/HSL_and_HSV
- [1] <http://www.easyrgb.com/en/math.php>
- [2] https://en.wikipedia.org/wiki/CIELAB_color_space
- [3] https://en.wikipedia.org/wiki/HCL_color_space
- [1] https://en.wikipedia.org/wiki/Standard_illuminant
- [2] https://en.wikipedia.org/wiki/CIELAB_color_space

[1] <http://www.easyrgb.com/en/math.php>
[2] https://en.wikipedia.org/wiki/CIELAB_color_space
[1] <http://www.easyrgb.com/en/math.php>
[2] https://en.wikipedia.org/wiki/HCL_color_space
[3] https://en.wikipedia.org/wiki/CIELAB_color_space
[1] <http://www.easyrgb.com/en/math.php>
[2] <https://en.wikipedia.org/wiki/CIELUV>
[1] <http://poynton.ca/PDFs/ColorFAQ.pdf>
[1] A. C. Ruifrok and D. A. Johnston, “Quantification of histochemical staining by color deconvolution,” Analytical and quantitative cytology and histology / the International Academy of Cytology [and] American Society of Cytology, vol. 23, no. 4, pp. 291-9, Aug. 2001.
[1] https://en.wikipedia.org/wiki/HSL_and_HSV
[1] https://en.wikipedia.org/wiki/Standard_illuminant
[1] <http://www.easyrgb.com/en/math.php>
[2] <https://en.wikipedia.org/wiki/CIELUV>
[1] https://en.wikipedia.org/wiki/CIE_1931_color_space
[1] https://en.wikipedia.org/wiki/CIE_1931_color_space
[1] <https://en.wikipedia.org/wiki/YCbCr>
[1] <https://en.wikipedia.org/wiki/YDbDr>
[1] <https://en.wikipedia.org/wiki/YPbPr>
[1] <https://en.wikipedia.org/wiki/YUV>
[1] https://en.wikipedia.org/wiki/Alpha_compositing#Alpha_blending
[1] https://en.wikipedia.org/wiki/CIE_1931_color_space
[1] <https://web.archive.org/web/20160624145052/http://www.mecourse.com/landinig/software/cdeconv/cdeconv.html>
[2] <https://github.com/DIPlib/diplib/>
[3] A. C. Ruifrok and D. A. Johnston, “Quantification of histochemical staining by color deconvolution,” Anal. Quant. Cytol. Histol., vol. 23, no. 4, pp. 291–299, Aug. 2001.
[1] <http://www.easyrgb.com/en/math.php>
[2] https://en.wikipedia.org/wiki/CIELAB_color_space
[1] <http://www.easyrgb.com/en/math.php>
[2] <https://en.wikipedia.org/wiki/CIELUV>
[1] https://en.wikipedia.org/wiki/CIE_1931_color_space
[1] https://en.wikipedia.org/wiki/Standard_illuminant#White_points_of_standard_illuminants
[2] https://en.wikipedia.org/wiki/CIE_1931_color_space#Meaning_of_X,_Y_and_Z
[3] <https://www.rdocumentation.org/packages/grDevices/versions/3.6.2/topics/convertColor>
[1] <https://en.wikipedia.org/wiki/YCbCr>

- [1] <https://en.wikipedia.org/wiki/YDbDr>
- [1] <https://en.wikipedia.org/wiki/YPbPr>
- [1] <https://en.wikipedia.org/wiki/YUV>
- [1] <https://graphics.stanford.edu/data/voldata/>
- [1] <https://github.com/scikit-image/scikit-image/issues/3927>
- [1] Paul Müller, Mirjam Schürmann, Salvatore Girardo, Gheorghe Cojoc, and Jochen Guck. “Accurate evaluation of size and refractive index for spherical objects in quantitative phase imaging.” *Optics Express* 26(8): 10729-10743 (2018). DOI:[10.1364/OE.26.010729](https://doi.org/10.1364/OE.26.010729)
- [1] Moffat J, Grueneberg DA, Yang X, Kim SY, Kloepfer AM, Hinkle G, Piqani B, Eisenhaure TM, Luo B, Grenier JK, Carpenter AE, Foo SY, Stewart SA, Stockwell BR, Hacohen N, Hahn WC, Lander ES, Sabatini DM, Root DE (2006) A lentiviral RNAi library for human and mouse genes applied to an arrayed viral high-content screen. *Cell*, 124(6):1283-98 / :DOI: [10.1016/j.cell.2006.01.040](https://doi.org/10.1016/j.cell.2006.01.040) PMID 16564017
- [2] GitHub licensing discussion <https://github.com/CellProfiler/examples/issues/41>
- [1] OpenCV lbpcascade trained files <https://github.com/opencv/opencv/tree/master/data/lbpcascades>
- [1] Huang, G., Mattar, M., Lee, H., & Learned-Miller, E. G. (2012). Learning to align from scratch. In Advances in Neural Information Processing Systems (pp. 764-772).
- [2] <http://vis-www.cs.umass.edu/lfw/>
- [1] Budai, A., Bock, R, Maier, A., Hornegger, J., Michelson, G. (2013). Robust Vessel Segmentation in Fundus Images. International Journal of Biomedical Imaging, vol. 2013, 2013. DOI:[10.1155/2013/154860](https://doi.org/10.1155/2013/154860)
- [1] Häggström, Mikael (2014). “Medical gallery of Mikael Häggström 2014”. *WikiJournal of Medicine* 1 (2). DOI:[10.15347/wjm/2014.008](https://doi.org/10.15347/wjm/2014.008). ISSN 2002-4436. Public Domain
- [1] L. A. Shepp and B. F. Logan, “The Fourier reconstruction of a head section,” in IEEE Transactions on Nuclear Science, vol. 21, no. 3, pp. 21-43, June 1974. DOI:[10.1109/TNS.1974.6499235](https://doi.org/10.1109/TNS.1974.6499235)
- [1] D. Scharstein, H. Hirschmueller, Y. Kitajima, G. Krathwohl, N. Nesic, X. Wang, and P. Westling. High-resolution stereo datasets with subpixel-accurate ground truth. In German Conference on Pattern Recognition (GCPR 2014), Muenster, Germany, September 2014.
- [2] <http://vision.middlebury.edu/stereo/data-scenes2014/>
- [1] Particle Image Velocimetry (PIV) Challenge site <http://pivchallenge.org>
- [2] 1st PIV challenge Case B: <http://pivchallenge.org/pub/index.html#b>
- [1] A Rasterizing Algorithm for Drawing Curves, A. Zingl, 2012 <http://members.chello.at/easyfilter/Bresenham.pdf>
- [1] A Rasterizing Algorithm for Drawing Curves, A. Zingl, 2012 <http://members.chello.at/easyfilter/Bresenham.pdf>
- [1] J.E. Bresenham, “Algorithm for computer control of a digital plotter”, IBM Systems journal, 4 (1965) 25-30.
- [2] E. Andres, “Discrete circles, rings and spheres”, Computers & Graphics, 18 (1994) 695-706.
- [1] X. Wu, “An efficient antialiasing technique”, In ACM SIGGRAPH Computer Graphics, 25 (1991) 143-152.
- [1] A Rasterizing Algorithm for Drawing Curves, A. Zingl, 2012 <http://members.chello.at/easyfilter/Bresenham.pdf>
- [1] A Rasterizing Algorithm for Drawing Curves, A. Zingl, 2012 <http://members.chello.at/easyfilter/Bresenham.pdf>
- [1] https://en.wikipedia.org/wiki/Gamma_correction

- [1] <http://www.ece.ucsb.edu/Faculty/Manjunath/courses/ece178W03/EnhancePart1.pdf>
- [1] Gustav J. Braun, “Image Lightness Rescaling Using Sigmoidal Contrast Enhancement Functions”, <http://markfairchild.org/PDFs/PAP07.pdf>
- [1] https://en.wikipedia.org/wiki/Cumulative_distribution_function
- [1] <http://tog.acm.org/resources/GraphicsGems/>
- [2] <https://en.wikipedia.org/wiki/CLAHE#CLAHE>
- [1] <http://www.janeriksolem.net/histogram-equalization-with-python-and.html>
- [2] https://en.wikipedia.org/wiki/Histogram_equalization
- [1] https://scikit-image.org/docs/dev/user_guide/data_types.html
- [1] <http://paulbourke.net/miscellaneous/equalisation/>
- [1] https://en.wikipedia.org/wiki/Blob_detection#The_difference_of_Gaussians_approach
- [2] Lowe, D. G. “Distinctive Image Features from Scale-Invariant Keypoints.” International Journal of Computer Vision 60, 91–110 (2004). <https://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf> DOI:10.1023/B:VISI.0000029664.99615.94
- [1] https://en.wikipedia.org/wiki/Blob_detection#The_determinant_of_the_Hessian
- [2] Herbert Bay, Andreas Ess, Tinne Tuytelaars, Luc Van Gool, “SURF: Speeded Up Robust Features” ftp://ftp.vision.ee.ethz.ch/publications/articles/eth_biwi_00517.pdf
- [1] https://en.wikipedia.org/wiki/Blob_detection#The_Laplacian_of_Gaussian
- [1] Canny, J., A Computational Approach To Edge Detection, IEEE Trans. Pattern Analysis and Machine Intelligence, 8:679-714, 1986 DOI:10.1109/TPAMI.1986.4767851
- [2] William Green’s Canny tutorial https://en.wikipedia.org/wiki/Canny_edge_detector
- [1] Rosten, E., & Drummond, T. (2006, May). Machine learning for high-speed corner detection. In European conference on computer vision (pp. 430-443). Springer, Berlin, Heidelberg. DOI:10.1007/11744023_34 http://www.edwardrosten.com/work/rosten_2006_machine.pdf
- [2] Wikipedia, ‘Features from accelerated segment test’, https://en.wikipedia.org/wiki/Features_from_accelerated_segment_test
- [1] Förstner, W., & Gülch, E. (1987, June). A fast operator for detection and precise location of distinct points, corners and centres of circular features. In Proc. ISPRS intercommission conference on fast processing of photogrammetric data (pp. 281-305). <https://cseweb.ucsd.edu/classes/sp02/cse252/foerstner/foerstner.pdf>
- [2] https://en.wikipedia.org/wiki/Corner_detection
- [1] https://en.wikipedia.org/wiki/Corner_detection
- [1] Kitchen, L., & Rosenfeld, A. (1982). Gray-level corner detection. Pattern recognition letters, 1(2), 95-102. DOI:10.1016/0167-8655(82)90020-4
- [1] https://en.wikipedia.org/wiki/Corner_detection
- [1] Ethan Rublee, Vincent Rabaud, Kurt Konolige and Gary Bradski ‘ORB : An efficient alternative to SIFT and SURF’ http://www.vision.cs.chubu.ac.jp/CV-R/pdf/Rublee_iccv2011.pdf
- [2] Paul L. Rosin, “Measuring Corner Properties” <http://users.cs.cf.ac.uk/Paul.Rosin/corner2.pdf>
- [1] https://en.wikipedia.org/wiki/Corner_detection
- [1] Förstner, W., & Gülch, E. (1987, June). A fast operator for detection and precise location of distinct points, corners and centres of circular features. In Proc. ISPRS intercommission conference on fast processing of photogrammetric data (pp. 281-305). <https://cseweb.ucsd.edu/classes/sp02/cse252/foerstner/foerstner.pdf>

- [2] https://en.wikipedia.org/wiki/Corner_detection
- [1] Tola et al. “Daisy: An efficient dense descriptor applied to wide- baseline stereo.” Pattern Analysis and Machine Intelligence, IEEE Transactions on 32.5 (2010): 815-830.
- [2] <http://cvlab.epfl.ch/software/daisy>
- [1] L. Zhang, R. Chu, S. Xiang, S. Liao, S.Z. Li. “Face Detection Based on Multi-Block LBP Representation”, In Proceedings: Advances in Biometrics, International Conference, ICB 2007, Seoul, Korea. <http://www.cbsr.ia.ac.cn/users/scliao/papers/Zhang-ICB07-MBLBP.pdf> DOI:10.1007/978-3-540-74549-5_2
- [1] Perronnin, F. and Dance, C. Fisher kernels on Visual Vocabularies for Image Categorization, IEEE Conference on Computer Vision and Pattern Recognition, 2007
- [2] Perronnin, F. and Sanchez, J. and Mensink T. Improving the Fisher Kernel for Large-Scale Image Classification, ECCV, 2010
- [1] M. Hall-Beyer, 2007. GLCM Texture: A Tutorial <https://prism.ucalgary.ca/handle/1880/51900> DOI:10.11575/PRISM/33280
- [2] R.M. Haralick, K. Shanmugam, and I. Dinstein, “Textural features for image classification”, IEEE Transactions on Systems, Man, and Cybernetics, vol. SMC-3, no. 6, pp. 610-621, Nov. 1973. DOI:10.1109/TSMC.1973.4309314
- [3] M. Nadler and E.P. Smith, Pattern Recognition Engineering, Wiley-Interscience, 1993.
- [4] Wikipedia, https://en.wikipedia.org/wiki/Co-occurrence_matrix
- [1] M. Hall-Beyer, 2007. GLCM Texture: A Tutorial v. 1.0 through 3.0. The GLCM Tutorial Home Page, <https://prism.ucalgary.ca/handle/1880/51900> DOI:10.11575/PRISM/33280
- [1] https://en.wikipedia.org/wiki/Haar-like_feature
- [2] Oren, M., Papageorgiou, C., Sinha, P., Osuna, E., & Poggio, T. (1997, June). Pedestrian detection using wavelet templates. In Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on (pp. 193-199). IEEE. <http://tinyurl.com/y6ulxfta> DOI:10.1109/CVPR.1997.609319
- [3] Viola, Paul, and Michael J. Jones. “Robust real-time face detection.” International journal of computer vision 57.2 (2004): 137-154. <https://www.merl.com/publications/docs/TR2004-043.pdf> DOI:10.1109/CVPR.2001.990517
- [1] Herbert Bay, Andreas Ess, Tinne Tuytelaars, Luc Van Gool, “SURF: Speeded Up Robust Features” ftp://ftp.vision.ee.ethz.ch/publications/articles/eth_biwi_00517.pdf
- [1] https://en.wikipedia.org/wiki/Histogram_of_oriented_gradients
- [2] Dalal, N and Triggs, B, Histograms of Oriented Gradients for Human Detection, IEEE Computer Society Conference on Computer Vision and Pattern Recognition 2005 San Diego, CA, USA, <https://lear.inrialpes.fr/people/triggs/pubs/Dalal-cvpr05.pdf>, DOI:10.1109/CVPR.2005.177
- [3] Lowe, D.G., Distinctive image features from scale-invariant keypoints, International Journal of Computer Vision (2004) 60: 91, <http://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf>, DOI:10.1023/B:VISI.0000029664.99615.94
- [4] Dalal, N, Finding People in Images and Videos, Human-Computer Interaction [cs.HC], Institut National Polytechnique de Grenoble - INPG, 2006, <https://tel.archives-ouvertes.fr/tel-00390303/file/NavneetDalalThesis.pdf>
- [1] <https://scikit-learn.org/stable/modules/generated/sklearn.mixture.GaussianMixture.html>
- [1] T. Ojala, M. Pietikainen, T. Maenpaa, “Multiresolution gray-scale and rotation invariant texture classification with local binary patterns”, IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 24, no. 7, pp. 971-987, July 2002 DOI:10.1109/TPAMI.2002.1017623

- [2] T. Ahonen, A. Hadid and M. Pietikainen. “Face recognition with local binary patterns”, in Proc. Eighth European Conf. Computer Vision, Prague, Czech Republic, May 11-14, 2004, pp. 469-481, 2004. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.214.6851> DOI:10.1007/978-3-540-24670-1_36
- [3] T. Ahonen, A. Hadid and M. Pietikainen, “Face Description with Local Binary Patterns: Application to Face Recognition”, IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 28, no. 12, pp. 2037-2041, Dec. 2006 DOI:10.1109/TPAMI.2006.244
- [1] J. P. Lewis, “Fast Normalized Cross-Correlation”, Industrial Light and Magic.
- [2] Briechle and Hanebeck, “Template Matching using Fast Normalized Cross Correlation”, Proceedings of the SPIE (2001). DOI:10.1117/12.421129
- [1] L. Zhang, R. Chu, S. Xiang, S. Liao, S.Z. Li. “Face Detection Based on Multi-Block LBP Representation”, In Proceedings: Advances in Biometrics, International Conference, ICB 2007, Seoul, Korea. <http://www.cbsr.ia.ac.cn/users/scliao/papers/Zhang-ICB07-MBLBP.pdf> DOI:10.1007/978-3-540-74549-5_2
- [1] Koenderink, J. J. & van Doorn, A. J., “Surface shape and curvature scales”, Image and Vision Computing, 1992, 10, 557-564. DOI:10.1016/0262-8856(92)90076-F
- [1] https://en.wikipedia.org/wiki/Structure_tensor
- [1] Motilal Agrawal, Kurt Konolige and Morten Rufus Blas “CENSURE: Center Surround Extremas for Realtime Feature Detection and Matching”, https://link.springer.com/chapter/10.1007/978-3-540-88693-8_8 DOI:10.1007/978-3-540-88693-8_8
- [2] Adam Schmidt, Marek Kraft, Michal Fularz and Zuzanna Domagala “Comparative Assessment of Point Feature Detectors and Descriptors in the Context of Robot Navigation” http://yadda.icm.edu.pl/yadda/element/bwmeta1.element.baztech-268aaf28-0faf-4872-a4df-7e2e61cb364c/c/Schmidt_comparative.pdf DOI:10.1.1.465.1117
- [1] Viola, P. and Jones, M. “Rapid object detection using a boosted cascade of simple features,” In: Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001, pp. I-I. DOI:10.1109/CVPR.2001.990517
- [2] Viola, P. and Jones, M.J, “Robust Real-Time Face Detection”, International Journal of Computer Vision 57, 137–154 (2004). DOI:10.1023/B:VISI.0000013087.49260.fb
- [3] Liao, S. et al. Learning Multi-scale Block Local Binary Patterns for Face Recognition. International Conference on Biometrics (ICB), 2007, pp. 828-837. In: Lecture Notes in Computer Science, vol 4642. Springer, Berlin, Heidelberg. DOI:10.1007/978-3-540-74549-5_87
- [1] Ethan Rublee, Vincent Rabaud, Kurt Konolige and Gary Bradski “ORB: An efficient alternative to SIFT and SURF” http://www.vision.cs.chubu.ac.jp/CV-R/pdf/Rublee_iccv2011.pdf
- [1] D.G. Lowe. “Object recognition from local scale-invariant features”, Proceedings of the Seventh IEEE International Conference on Computer Vision, 1999, vol.2, pp. 1150-1157. DOI:10.1109/ICCV.1999.790410
- [2] D.G. Lowe. “Distinctive Image Features from Scale-Invariant Keypoints”, International Journal of Computer Vision, 2004, vol. 60, pp. 91–110. DOI:10.1023/B:VISI.0000029664.99615.94
- [3] I. R. Otero and M. Delbracio. “Anatomy of the SIFT Method”, Image Processing On Line, 4 (2014), pp. 370–396. DOI:10.5201/ipol.2014.82
- [1] J. Canny. A computational approach to edge detection. IEEE Transactions on Pattern Analysis and Machine Intelligence. 1986; vol. 8, pp.679-698. DOI:10.1109/TPAMI.1986.4767851
- [1] Russ, John C., et al. The Image Processing Handbook, 3rd. Ed. 1999, CRC Press, LLC.
- [2] Birchfield, Stan. Image Processing and Analysis. 2018. Cengage Learning.
- [3] Butterworth, Stephen. “On the theory of filter amplifiers.” Wireless Engineer 7.6 (1930): 536-541.
- [4] https://en.wikipedia.org/wiki/Butterworth_filter

- [1] Marr, D. and Hildreth, E. Theory of Edge Detection. Proc. R. Soc. Lond. Series B 207, 187-217 (1980). <https://doi.org/10.1098/rspb.1980.0020>
- [1] Farid, H. and Simoncelli, E. P., “Differentiation of discrete multidimensional signals”, IEEE Transactions on Image Processing 13(4): 496-508, 2004. DOI:[10.1109/TIP.2004.823819](https://doi.org/10.1109/TIP.2004.823819)
- [2] Wikipedia, “Farid and Simoncelli Derivatives.” Available at: <https://en.wikipedia.org/wiki/Image_derivatives#Farid_and_Simoncelli_Derivatives>
- [1] Farid, H. and Simoncelli, E. P., “Differentiation of discrete multidimensional signals”, IEEE Transactions on Image Processing 13(4): 496-508, 2004. DOI:[10.1109/TIP.2004.823819](https://doi.org/10.1109/TIP.2004.823819)
- [2] Farid, H. and Simoncelli, E. P. “Optimally rotation-equivariant directional derivative kernels”, In: 7th International Conference on Computer Analysis of Images and Patterns, Kiel, Germany. Sep, 1997.
- [1] Farid, H. and Simoncelli, E. P., “Differentiation of discrete multidimensional signals”, IEEE Transactions on Image Processing 13(4): 496-508, 2004. DOI:[10.1109/TIP.2004.823819](https://doi.org/10.1109/TIP.2004.823819)
- [1] Frangi, A. F., Niessen, W. J., Vincken, K. L., & Viergever, M. A. (1998,). Multiscale vessel enhancement filtering. In International Conference on Medical Image Computing and Computer-Assisted Intervention (pp. 130-137). Springer Berlin Heidelberg. DOI:[10.1007/BFb0056195](https://doi.org/10.1007/BFb0056195)
- [2] Kroon, D. J.: Hessian based Frangi vesselness filter.
- [3] Ellis, D. G.: <https://github.com/ellisdg/frangi3d/tree/master/frangi>
- [1] https://en.wikipedia.org/wiki/Gabor_filter
- [2] <https://web.archive.org/web/20180127125930/http://mplab.ucsd.edu/tutorials/gabor.pdf>
- [1] https://en.wikipedia.org/wiki/Gabor_filter
- [2] <https://web.archive.org/web/20180127125930/http://mplab.ucsd.edu/tutorials/gabor.pdf>
- [1] Ng, C. C., Yap, M. H., Costen, N., & Li, B. (2014,). Automatic wrinkle detection using hybrid Hessian filter. In Asian Conference on Computer Vision (pp. 609-622). Springer International Publishing. DOI:[10.1007/978-3-319-16811-1_40](https://doi.org/10.1007/978-3-319-16811-1_40)
- [2] Kroon, D. J.: Hessian based Frangi vesselness filter.
- [1] Meijering, E., Jacob, M., Sarria, J. C., Steiner, P., Hirlong, H., Unser, M. (2004). Design and validation of a tool for neurite tracing and analysis in fluorescence microscopy images. Cytometry Part A, 58(2), 167-176. DOI:[10.1002/cyto.a.20022](https://doi.org/10.1002/cyto.a.20022)
- [1] Sato, Y., Nakajima, S., Shiraga, N., Atsumi, H., Yoshida, S., Koller, T., ..., Kikinis, R. (1998). Three-dimensional multi-scale line filter for segmentation and visualization of curvilinear structures in medical images. Medical image analysis, 2(2), 143-168. DOI:[10.1016/S1361-8415\(98\)80009-1](https://doi.org/10.1016/S1361-8415(98)80009-1)
- [1] D. Kroon, 2009, Short Paper University Twente, Numerical Optimization of Kernel Based Image Derivatives.
- [2] https://en.wikipedia.org/wiki/Sobel_operator#Alternative_operators
- [1] D. Kroon, 2009, Short Paper University Twente, Numerical Optimization of Kernel Based Image Derivatives.
- [1] D. Kroon, 2009, Short Paper University Twente, Numerical Optimization of Kernel Based Image Derivatives.
- [1] D. Kroon, 2009, Short Paper University Twente, Numerical Optimization of Kernel Based Image Derivatives.
- [2] https://en.wikipedia.org/wiki/Sobel_operator
- [1] Ridler, TW & Calvard, S (1978), “Picture thresholding using an iterative selection method” IEEE Transactions on Systems, Man and Cybernetics 8: 630-632, DOI:[10.1109/TSMC.1978.4310039](https://doi.org/10.1109/TSMC.1978.4310039)
- [2] Sezgin M. and Sankur B. (2004) “Survey over Image Thresholding Techniques and Quantitative Performance Evaluation” Journal of Electronic Imaging, 13(1): 146-165, http://www.busim.ee.boun.edu.tr/~sankur/SankurFolder/Threshold_survey.pdf DOI:[10.1117/1.1631315](https://doi.org/10.1117/1.1631315)

- [3] ImageJ AutoThresholder code, http://fiji.sc/wiki/index.php/Auto_Threshold
- [1] Li C.H. and Lee C.K. (1993) “Minimum Cross Entropy Thresholding” Pattern Recognition, 26(4): 617-625 DOI:10.1016/0031-3203(93)90115-D
- [2] Li C.H. and Tam P.K.S. (1998) “An Iterative Algorithm for Minimum Cross Entropy Thresholding” Pattern Recognition Letters, 18(8): 771-776 DOI:10.1016/S0167-8655(98)00057-9
- [3] Sezgin M. and Sankur B. (2004) “Survey over Image Thresholding Techniques and Quantitative Performance Evaluation” Journal of Electronic Imaging, 13(1): 146-165 DOI:10.1117/1.1631315
- [4] ImageJ AutoThresholder code, http://fiji.sc/wiki/index.php/Auto_Threshold
- [1] Gonzalez, R. C. and Wood, R. E. “Digital Image Processing (2nd Edition).” Prentice-Hall Inc., 2002: 600–612. ISBN: 0-201-18075-8
- [1] C. A. Glasbey, “An analysis of histogram-based thresholding algorithms,” CVGIP: Graphical Models and Image Processing, vol. 55, pp. 532-537, 1993. DOI:10.1006/cgip.1993.1040
- [1] C. A. Glasbey, “An analysis of histogram-based thresholding algorithms,” CVGIP: Graphical Models and Image Processing, vol. 55, pp. 532-537, 1993.
- [2] Prewitt, JMS & Mendelsohn, ML (1966), “The analysis of cell images”, Annals of the New York Academy of Sciences 128: 1035-1053 DOI:10.1111/j.1749-6632.1965.tb11715.x
- [1] Liao, P-S., Chen, T-S. and Chung, P-C., “A fast algorithm for multilevel thresholding”, Journal of Information Science and Engineering 17 (5): 713-727, 2001. Available at: <https://ftp.iis.sinica.edu.tw/JISE/2001/200109_01.pdf> DOI:10.6688/JISE.2001.17.5.1
- [2] Tosa, Y., “Multi-Otsu Threshold”, a java plugin for ImageJ. Available at: <http://imagej.net/plugins/download/Multi_OtsuThreshold.java>
- [1] W. Niblack, An introduction to Digital Image Processing, Prentice-Hall, 1986.
- [2] D. Bradley and G. Roth, “Adaptive thresholding using Integral Image”, Journal of Graphics Tools 12(2), pp. 13-21, 2007. DOI:10.1080/2151237X.2007.10129236
- [1] Wikipedia, https://en.wikipedia.org/wiki/Otsu's_Method
- [1] J. Sauvola and M. Pietikainen, “Adaptive document image binarization,” Pattern Recognition 33(2), pp. 225-236, 2000. DOI:10.1016/S0031-3203(99)00055-2
- [1] Zack, G. W., Rogers, W. E. and Latt, S. A., 1977, Automatic Measurement of Sister Chromatid Exchange Frequency, Journal of Histochemistry and Cytochemistry 25 (7), pp. 741-753 DOI:10.1177/25.7.70454
- [2] ImageJ AutoThresholder code, http://fiji.sc/wiki/index.php/Auto_Threshold
- [1] Yen J.C., Chang F.J., and Chang S. (1995) “A New Criterion for Automatic Multilevel Thresholding” IEEE Trans. on Image Processing, 4(3): 370-378. DOI:10.1109/83.366472
- [2] Sezgin M. and Sankur B. (2004) “Survey over Image Thresholding Techniques and Quantitative Performance Evaluation” Journal of Electronic Imaging, 13(1): 146-165, DOI:10.1117/1.1631315 http://www.busim.ee.boun.edu.tr/~sankur/SankurFolder/Threshold_survey.pdf
- [3] ImageJ AutoThresholder code, http://fiji.sc/wiki/index.php/Auto_Threshold
- [1] Maria Petrou, Costas Petrou “Image Processing: The Fundamentals”, (2010), ed ii., page 357, ISBN 13: 9781119994398 DOI:10.1002/9781119994398
- [2] Wikipedia. Unsharp masking https://en.wikipedia.org/wiki/Unsharp_masking
- [1] Two-dimensional window design, Wikipedia, https://en.wikipedia.org/wiki/Two_dimensional_window_design
- [1] [https://en.wikipedia.org/wiki/Entropy_\(information_theory\)](https://en.wikipedia.org/wiki/Entropy_(information_theory))

- [1] Gonzalez, R. C. and Woods, R. E. “Digital Image Processing (3rd Edition).” Prentice-Hall Inc, 2006.
- [1] N. Hashimoto et al. Referenceless image quality evaluation for whole slide imaging. *J Pathol Inform* 2012;3:9.
- [1] https://en.wikipedia.org/wiki/Otsu's_method
- [1] Shi, J.; Malik, J., “Normalized cuts and image segmentation”, *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 22, no. 8, pp. 888-905, August 2000.
- [1] Alain Tremeau and Philippe Colantoni “Regions Adjacency Graph Applied To Color Image Segmentation” DOI:10.1109/83.841950
- [1] Alain Tremeau and Philippe Colantoni “Regions Adjacency Graph Applied To Color Image Segmentation” DOI:10.1109/83.841950
- [1] https://en.wikipedia.org/wiki/Ramer-Douglas-Peucker_algorithm
- [1] Frederique Crete, Thierry Dolmire, Patricia Ladret, and Marina Nicolas “The blur effect: perception and estimation with a new no-reference perceptual blur metric” *Proc. SPIE 6492, Human Vision and Electronic Imaging XII*, 64920I (2007) <https://hal.archives-ouvertes.fr/hal-00232709> DOI:10.1117/12.702790
- [1] S. Rivollier. Analyse d'image geometrique et morphometrique par diagrammes de forme et voisinages adaptatifs generaux. PhD thesis, 2010. Ecole Nationale Supérieure des Mines de Saint-Etienne. <https://tel.archives-ouvertes.fr/tel-00560838>
- [2] Ohser J., Nagel W., Schladitz K. (2002) The Euler Number of Discretized Sets - On the Choice of Adjacency in Homogeneous Lattices. In: Mecke K., Stoyan D. (eds) Morphology of Condensed Matter. Lecture Notes in Physics, vol 600. Springer, Berlin, Heidelberg.
- [1] Lorensen, William and Harvey E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics (SIGGRAPH 87 Proceedings)* 21(4) July 1987, p. 163-170. DOI:10.1145/37401.37422
- [1] https://en.wikipedia.org/wiki/Moment_of_inertia#Inertia_tensor
- [2] Bernd Jähne. Spatio-Temporal Image Processing: Theory and Scientific Applications. (Chapter 8: Tensor Methods) Springer, 1993.
- [1] Christophe Fiorio and Jens Gustedt, “Two linear time Union-Find strategies for image processing”, *Theoretical Computer Science* 154 (1996), pp. 165-181.
- [2] Kensheng Wu, Ekow Otoo and Arie Shoshani, “Optimizing connected component labeling algorithms”, Paper LBNL-56864, 2005, Lawrence Berkeley National Laboratory (University of California), <http://repositories.cdlib.org/lbln/LBNL-56864>
- [1] Manders, E.M.M., Verbeek, F.J. and Aten, J.A. (1993), Measurement of co-localization of objects in dual-colour confocal images. *Journal of Microscopy*, 169: 375-382. <https://doi.org/10.1111/j.1365-2818.1993.tb03313.x> <https://imagej.net/media/manders.pdf>
- [2] Dunn, K. W., Kamocka, M. M., & McDonald, J. H. (2011). A practical guide to evaluating colocalization in biological microscopy. *American journal of physiology. Cell physiology*, 300(4), C723–C742. <https://doi.org/10.1152/ajpcell.00462.2010>
- [1] Manders, E.M.M., Verbeek, F.J. and Aten, J.A. (1993), Measurement of co-localization of objects in dual-colour confocal images. *Journal of Microscopy*, 169: 375-382. <https://doi.org/10.1111/j.1365-2818.1993.tb03313.x> <https://imagej.net/media/manders.pdf>
- [2] Dunn, K. W., Kamocka, M. M., & McDonald, J. H. (2011). A practical guide to evaluating colocalization in biological microscopy. *American journal of physiology. Cell physiology*, 300(4), C723–C742. <https://doi.org/10.1152/ajpcell.00462.2010>
- [3] Bolte, S. and Cordelières, F.P. (2006), A guided tour into subcellular colocalization analysis in light microscopy. *Journal of Microscopy*, 224: 213-232. <https://doi.org/10.1111/j.1365-2818.2006.01>

- [4] Adler J, Parmryd I. (2010), Quantifying colocalization by correlation: the Pearson correlation coefficient is superior to the Mander's overlap coefficient. *Cytometry A*. Aug;77(8):733-42.<https://doi.org/10.1002/cyto.a.20896>
- [5] Adler, J, Parmryd, I. Quantifying colocalization: The case for discarding the Manders overlap coefficient. *Cytometry*. 2021; 99: 910– 920. <https://doi.org/10.1002/cyto.a.24336>
- [1] Thomas Lewiner, Helio Lopes, Antonio Wilson Vieira and Geovan Tavares. Efficient implementation of Marching Cubes' cases with topological guarantees. *Journal of Graphics Tools* 8(2) pp. 1-15 (december 2003). DOI:[10.1080/10867651.2003.10487582](https://doi.org/10.1080/10867651.2003.10487582)
- [2] Lorensen, William and Harvey E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics (SIGGRAPH 87 Proceedings)* 21(4) July 1987, p. 163-170). DOI:[10.1145/37401.37422](https://doi.org/10.1145/37401.37422)
- [1] Wilhelm Burger, Mark Burge. Principles of Digital Image Processing: Core Algorithms. Springer-Verlag, London, 2009.
- [2] B. Jähne. Digital Image Processing. Springer-Verlag, Berlin-Heidelberg, 6. edition, 2005.
- [3] T. H. Reiss. Recognizing Planar Objects Using Invariant Image Features, from Lecture notes in computer science, p. 676. Springer, Berlin, 1993.
- [4] https://en.wikipedia.org/wiki/Image_moment
- [1] Wilhelm Burger, Mark Burge. Principles of Digital Image Processing: Core Algorithms. Springer-Verlag, London, 2009.
- [2] B. Jähne. Digital Image Processing. Springer-Verlag, Berlin-Heidelberg, 6. edition, 2005.
- [3] T. H. Reiss. Recognizing Planar Objects Using Invariant Image Features, from Lecture notes in computer science, p. 676. Springer, Berlin, 1993.
- [4] https://en.wikipedia.org/wiki/Image_moment
- [1] Johannes Kilian. Simple Image Analysis By Moments. Durham University, version 0.2, Durham, 2001.
- [1] Johannes Kilian. Simple Image Analysis By Moments. Durham University, version 0.2, Durham, 2001.
- [1] M. K. Hu, “Visual Pattern Recognition by Moment Invariants”, *IRE Trans. Info. Theory*, vol. IT-8, pp. 179-187, 1962
- [2] Wilhelm Burger, Mark Burge. Principles of Digital Image Processing: Core Algorithms. Springer-Verlag, London, 2009.
- [3] B. Jähne. Digital Image Processing. Springer-Verlag, Berlin-Heidelberg, 6. edition, 2005.
- [4] T. H. Reiss. Recognizing Planar Objects Using Invariant Image Features, from Lecture notes in computer science, p. 676. Springer, Berlin, 1993.
- [5] https://en.wikipedia.org/wiki/Image_moment
- [1] Wilhelm Burger, Mark Burge. Principles of Digital Image Processing: Core Algorithms. Springer-Verlag, London, 2009.
- [2] B. Jähne. Digital Image Processing. Springer-Verlag, Berlin-Heidelberg, 6. edition, 2005.
- [3] T. H. Reiss. Recognizing Planar Objects Using Invariant Image Features, from Lecture notes in computer science, p. 676. Springer, Berlin, 1993.
- [4] https://en.wikipedia.org/wiki/Image_moment
- [1] [# noqa">https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.pearsonr.html # noqa](https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.pearsonr.html)
- [2] [# noqa">https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.spearmanr.html # noqa](https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.spearmanr.html)

- [3] Dunn, K. W., Kamocka, M. M., & McDonald, J. H. (2011). A practical guide to evaluating colocalization in biological microscopy. *American journal of physiology. Cell physiology*, 300(4), C723–C742. <https://doi.org/10.1152/ajpcell.00462.2010>
- [4] Bolte, S. and Cordelières, F.P. (2006), A guided tour into subcellular colocalization analysis in light microscopy. *Journal of Microscopy*, 224: 213-232. <https://doi.org/10.1111/j.1365-2818.2006.01706.x>
- [1] K. Benkrid, D. Crookes. Design and FPGA Implementation of a Perimeter Estimator. The Queen's University of Belfast. <http://www.cs.qub.ac.uk/~d.crookes/webpubs/papers/perimeter.doc>
- [1] https://en.wikipedia.org/wiki/Crofton_formula
- [2] S. Rivollier. Analyse d'image geometrique et morphometrique par diagrammes de forme et voisinages adaptatifs generaux. PhD thesis, 2010. Ecole Nationale Supérieure des Mines de Saint-Etienne. <https://tel.archives-ouvertes.fr/tel-00560838>
- [1] “RANSAC”, Wikipedia, <https://en.wikipedia.org/wiki/RANSAC>
- [1] Wilhelm Burger, Mark Burge. Principles of Digital Image Processing: Core Algorithms. Springer-Verlag, London, 2009.
- [2] B. Jähne. Digital Image Processing. Springer-Verlag, Berlin-Heidelberg, 6. edition, 2005.
- [3] T. H. Reiss. Recognizing Planar Objects Using Invariant Image Features, from Lecture notes in computer science, p. 676. Springer, Berlin, 1993.
- [4] https://en.wikipedia.org/wiki/Image_moment
- [5] W. Pabst, E. Gregorová. Characterization of particles and particle systems, pp. 27-28. ICT Prague, 2007. https://old.vscht.cz/sil/keramika/Characterization_of_particles/CPPS%20_English%20version_.pdf
- [1] [https://en.wikipedia.org/wiki/Entropy_\(information_theory\)](https://en.wikipedia.org/wiki/Entropy_(information_theory))
- [2] https://en.wiktionary.org/wiki/Shannon_entropy
- [1] <http://mrl.nyu.edu/publications/subdiv-course2000/coursenotes00.pdf>
- [1] Jekel, Charles F. Obtaining non-linear orthotropic material models for pvc-coated polyester via inverse bubble inflation. Thesis (MEng), Stellenbosch University, 2016. Appendix A, pp. 83-87. <https://hdl.handle.net/10019.1/98627>
- [1] Halir, R.; Flusser, J. “Numerically stable direct least squares fitting of ellipses”. In Proc. 6th International Conference in Central Europe on Computer Graphics and Visualization. WSCG (Vol. 98, pp. 125-132).
- [1] Arganda-Carreras I, Turaga SC, Berger DR, et al. (2015) Crowdsourcing the creation of image segmentation algorithms for connectomics. *Front. Neuroanat.* 9:142. DOI:[10.3389/fnana.2015.00142](https://doi.org/10.3389/fnana.2015.00142)
- [1] http://en.wikipedia.org/wiki/Hausdorff_distance
- [2] M. P. Dubuisson and A. K. Jain. A Modified Hausdorff distance for object matching. In ICPR94, pages A:566-568, Jerusalem, Israel, 1994. DOI:[10.1109/ICPR.1994.576361](https://doi.org/10.1109/ICPR.1994.576361) <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.1.8155>
- [1] http://en.wikipedia.org/wiki/Hausdorff_distance
- [1] C. Studholme, D.L.G. Hill, & D.J. Hawkes (1999). An overlap invariant entropy measure of 3D medical image alignment. *Pattern Recognition* 32(1):71-86 DOI:[10.1016/S0031-3203\(98\)00091-0](https://doi.org/10.1016/S0031-3203(98)00091-0)
- [1] https://en.wikipedia.org/wiki/Root-mean-square_deviation
- [1] https://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio
- [1] Wang, Z., Bovik, A. C., Sheikh, H. R., & Simoncelli, E. P. (2004). Image quality assessment: From error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13, 600-612. <https://ece.uwaterloo.ca/~z70wang/publications/ssim.pdf>, DOI:[10.1109/TIP.2003.819861](https://doi.org/10.1109/TIP.2003.819861)

- [2] Avanaki, A. N. (2009). Exact global histogram specification optimized for structural similarity. *Optical Review*, 16, 613-621. [arXiv:0901.0065](https://arxiv.org/abs/0901.0065) DOI:[10.1007/s10043-009-0119-z](https://doi.org/10.1007/s10043-009-0119-z)
- [1] Marina Meilă (2007), Comparing clusterings—an information based distance, *Journal of Multivariate Analysis*, Volume 98, Issue 5, Pages 873-895, ISSN 0047-259X, DOI:[10.1016/j.jmva.2006.11.013](https://doi.org/10.1016/j.jmva.2006.11.013).
- [1] Vincent L., Proc. “Grayscale area openings and closings, their efficient implementation and applications”, EURASIP Workshop on Mathematical Morphology and its Applications to Signal Processing, Barcelona, Spain, pp.22-27, May 1993.
- [2] Soille, P., “Morphological Image Analysis: Principles and Applications” (Chapter 6), 2nd edition (2003), ISBN 3540429883. DOI:[10.1007/978-3-662-05088-0](https://doi.org/10.1007/978-3-662-05088-0)
- [3] Salembier, P., Oliveras, A., & Garrido, L. (1998). Antiextensive Connected Operators for Image and Sequence Processing. *IEEE Transactions on Image Processing*, 7(4), 555-570. DOI:[10.1109/83.663500](https://doi.org/10.1109/83.663500)
- [4] Najman, L., & Couprise, M. (2006). Building the component tree in quasi-linear time. *IEEE Transactions on Image Processing*, 15(11), 3531-3539. DOI:[10.1109/TIP.2006.877518](https://doi.org/10.1109/TIP.2006.877518)
- [5] Carlinet, E., & Geraud, T. (2014). A Comparative Review of Component Tree Computation Algorithms. *IEEE Transactions on Image Processing*, 23(9), 3885-3895. DOI:[10.1109/TIP.2014.2336551](https://doi.org/10.1109/TIP.2014.2336551)
- [1] Vincent L., Proc. “Grayscale area openings and closings, their efficient implementation and applications”, EURASIP Workshop on Mathematical Morphology and its Applications to Signal Processing, Barcelona, Spain, pp.22-27, May 1993.
- [2] Soille, P., “Morphological Image Analysis: Principles and Applications” (Chapter 6), 2nd edition (2003), ISBN 3540429883. DOI:[10.1007/978-3-662-05088-0](https://doi.org/10.1007/978-3-662-05088-0)
- [3] Salembier, P., Oliveras, A., & Garrido, L. (1998). Antiextensive Connected Operators for Image and Sequence Processing. *IEEE Transactions on Image Processing*, 7(4), 555-570. DOI:[10.1109/83.663500](https://doi.org/10.1109/83.663500)
- [4] Najman, L., & Couprise, M. (2006). Building the component tree in quasi-linear time. *IEEE Transactions on Image Processing*, 15(11), 3531-3539. DOI:[10.1109/TIP.2006.877518](https://doi.org/10.1109/TIP.2006.877518)
- [5] Carlinet, E., & Geraud, T. (2014). A Comparative Review of Component Tree Computation Algorithms. *IEEE Transactions on Image Processing*, 23(9), 3885-3895. DOI:[10.1109/TIP.2014.2336551](https://doi.org/10.1109/TIP.2014.2336551)
- [1] Park, H and Chin R.T. Decomposition of structuring elements for optimal implementation of morphological operations. In Proceedings: 1997 IEEE Workshop on Nonlinear Signal and Image Processing, London, UK. <https://www.iwaenc.org/proceedings/1997/nsip97/pdf/scan/ns970226.pdf>
- [2] <https://en.wikipedia.org/wiki/Rhombicuboctahedron>
- [1] https://en.wikipedia.org/wiki/Top-hat_transform
- [1] <https://blogs.mathworks.com/steve/2011/10/04/binary-image-convex-hull-algorithm-notes/>
- [1] Walter, T., & Klein, J.-C. (2002). Automatic Detection of Microaneurysms in Color Fundus Images of the Human Retina by Means of the Bounding Box Closing. In A. Colosimo, P. Sirabella, A. Giuliani (Eds.), Medical Data Analysis. Lecture Notes in Computer Science, vol 2526, pp. 210-220. Springer Berlin Heidelberg. DOI:[10.1007/3-540-36104-9_23](https://doi.org/10.1007/3-540-36104-9_23)
- [2] Carlinet, E., & Geraud, T. (2014). A Comparative Review of Component Tree Computation Algorithms. *IEEE Transactions on Image Processing*, 23(9), 3885-3895. DOI:[10.1109/TIP.2014.2336551](https://doi.org/10.1109/TIP.2014.2336551)
- [1] Walter, T., & Klein, J.-C. (2002). Automatic Detection of Microaneurysms in Color Fundus Images of the Human Retina by Means of the Bounding Box Closing. In A. Colosimo, P. Sirabella, A. Giuliani (Eds.), Medical Data Analysis. Lecture Notes in Computer Science, vol 2526, pp. 210-220. Springer Berlin Heidelberg. DOI:[10.1007/3-540-36104-9_23](https://doi.org/10.1007/3-540-36104-9_23)
- [2] Carlinet, E., & Geraud, T. (2014). A Comparative Review of Component Tree Computation Algorithms. *IEEE Transactions on Image Processing*, 23(9), 3885-3895. DOI:[10.1109/TIP.2014.2336551](https://doi.org/10.1109/TIP.2014.2336551)

- [1] Park, H and Chin R.T. Decomposition of structuring elements for optimal implementation of morphological operations. In Proceedings: 1997 IEEE Workshop on Nonlinear Signal and Image Processing, London, UK. <https://www.iwaenc.org/proceedings/1997/nsip97/pdf/scan/ns970226.pdf>
- [2] <https://en.wikipedia.org/wiki/Hexadecagon>
- [3] Vanrell, M and Vitrià, J. Optimal 3×3 decomposable disks for morphological transformations. *Image and Vision Computing*, Vol. 15, Issue 11, 1997. DOI:[10.1016/S0262-8856\(97\)00026-7](https://doi.org/10.1016/S0262-8856(97)00026-7)
- [4] Li, D. and Ritter, G.X. Decomposition of Separable and Symmetric Convex Templates. Proc. SPIE 1350, *Image Algebra and Morphological Image Processing*, (1 November 1990). DOI:[10.1117/12.23608](https://doi.org/10.1117/12.23608)
- [1] Li, D. and Ritter, G.X. Decomposition of Separable and Symmetric Convex Templates. Proc. SPIE 1350, *Image Algebra and Morphological Image Processing*, (1 November 1990). DOI:[10.1117/12.23608](https://doi.org/10.1117/12.23608)
- [1] Soille, P., “Morphological Image Analysis: Principles and Applications” (Chapter 6), 2nd edition (2003), ISBN 3540429883.
- [1] Soille, P., “Morphological Image Analysis: Principles and Applications” (Chapter 6), 2nd edition (2003), ISBN 3540429883.
- [1] Cuisenaire, O. and Macq, B., “Fast Euclidean morphological operators using local distance transformation by propagation, and applications,” *Image Processing And Its Applications*, 1999. Seventh International Conference on (Conf. Publ. No. 465), 1999, pp. 856-860 vol.2. DOI:[10.1049/cp:19990446](https://doi.org/10.1049/cp:19990446)
- [2] Ingemar Ragnemalm, Fast erosion and dilation by contour processing and thresholding of distance maps, *Pattern Recognition Letters*, Volume 13, Issue 3, 1992, Pages 161-166. DOI:[10.1016/0167-8655\(92\)90055-5](https://doi.org/10.1016/0167-8655(92)90055-5)
- [1] Cuisenaire, O. and Macq, B., “Fast Euclidean morphological operators using local distance transformation by propagation, and applications,” *Image Processing And Its Applications*, 1999. Seventh International Conference on (Conf. Publ. No. 465), 1999, pp. 856-860 vol.2. DOI:[10.1049/cp:19990446](https://doi.org/10.1049/cp:19990446)
- [2] Ingemar Ragnemalm, Fast erosion and dilation by contour processing and thresholding of distance maps, *Pattern Recognition Letters*, Volume 13, Issue 3, 1992, Pages 161-166. DOI:[10.1016/0167-8655\(92\)90055-5](https://doi.org/10.1016/0167-8655(92)90055-5)
- [1] Cuisenaire, O. and Macq, B., “Fast Euclidean morphological operators using local distance transformation by propagation, and applications,” *Image Processing And Its Applications*, 1999. Seventh International Conference on (Conf. Publ. No. 465), 1999, pp. 856-860 vol.2. DOI:[10.1049/cp:19990446](https://doi.org/10.1049/cp:19990446)
- [2] Ingemar Ragnemalm, Fast erosion and dilation by contour processing and thresholding of distance maps, *Pattern Recognition Letters*, Volume 13, Issue 3, 1992, Pages 161-166. DOI:[10.1016/0167-8655\(92\)90055-5](https://doi.org/10.1016/0167-8655(92)90055-5)
- [1] Cuisenaire, O. and Macq, B., “Fast Euclidean morphological operators using local distance transformation by propagation, and applications,” *Image Processing And Its Applications*, 1999. Seventh International Conference on (Conf. Publ. No. 465), 1999, pp. 856-860 vol.2. DOI:[10.1049/cp:19990446](https://doi.org/10.1049/cp:19990446)
- [2] Ingemar Ragnemalm, Fast erosion and dilation by contour processing and thresholding of distance maps, *Pattern Recognition Letters*, Volume 13, Issue 3, 1992, Pages 161-166. DOI:[10.1016/0167-8655\(92\)90055-5](https://doi.org/10.1016/0167-8655(92)90055-5)
- [1] Christophe Fiorio and Jens Gustedt, “Two linear time Union-Find strategies for image processing”, *Theoretical Computer Science* 154 (1996), pp. 165-181.
- [2] Kensheng Wu, Ekow Otoo and Arie Shoshani, “Optimizing connected component labeling algorithms”, Paper LBNL-56864, 2005, Lawrence Berkeley National Laboratory (University of California), <http://repositories.cdlib.org/lbnl/LBNL-56864>
- [1] Salembier, P., Oliveras, A., & Garrido, L. (1998). Antiextensive Connected Operators for Image and Sequence Processing. *IEEE Transactions on Image Processing*, 7(4), 555-570. DOI:[10.1109/83.663500](https://doi.org/10.1109/83.663500)
- [2] Berger, C., Geraud, T., Levillain, R., Widynski, N., Baillard, A., Bertin, E. (2007). Effective Component Tree Computation with Application to Pattern Recognition in Astronomical Imaging. In International Conference on Image Processing (ICIP) (pp. 41-44). DOI:[10.1109/ICIP.2007.4379949](https://doi.org/10.1109/ICIP.2007.4379949)

- [3] Najman, L., & Couprivé, M. (2006). Building the component tree in quasi-linear time. *IEEE Transactions on Image Processing*, 15(11), 3531-3539. DOI:[10.1109/TIP.2006.877518](https://doi.org/10.1109/TIP.2006.877518)
- [4] Carlinet, E., & Géraud, T. (2014). A Comparative Review of Component Tree Computation Algorithms. *IEEE Transactions on Image Processing*, 23(9), 3885-3895. DOI:[10.1109/TIP.2014.2336551](https://doi.org/10.1109/TIP.2014.2336551)
- [1] Vincent L., Proc. “Grayscale area openings and closings, their efficient implementation and applications”, EURASIP Workshop on Mathematical Morphology and its Applications to Signal Processing, Barcelona, Spain, pp.22-27, May 1993.
- [2] Soille, P., “Morphological Image Analysis: Principles and Applications” (Chapter 6), 2nd edition (2003), ISBN 3540429883. DOI:[10.1007/978-3-662-05088-0](https://doi.org/10.1007/978-3-662-05088-0)
- [3] Salember, P., Oliveras, A., & Garrido, L. (1998). Antiextensive Connected Operators for Image and Sequence Processing. *IEEE Transactions on Image Processing*, 7(4), 555-570. DOI:[10.1109/83.663500](https://doi.org/10.1109/83.663500)
- [4] Najman, L., & Couprivé, M. (2006). Building the component tree in quasi-linear time. *IEEE Transactions on Image Processing*, 15(11), 3531-3539. DOI:[10.1109/TIP.2006.877518](https://doi.org/10.1109/TIP.2006.877518)
- [5] Carlinet, E., & Géraud, T. (2014). A Comparative Review of Component Tree Computation Algorithms. *IEEE Transactions on Image Processing*, 23(9), 3885-3895. DOI:[10.1109/TIP.2014.2336551](https://doi.org/10.1109/TIP.2014.2336551)
- [1] Robinson, “Efficient morphological reconstruction: a downhill filter”, *Pattern Recognition Letters* 25 (2004) 1759-1767.
- [2] Vincent, L., “Morphological Grayscale Reconstruction in Image Analysis: Applications and Efficient Algorithms”, *IEEE Transactions on Image Processing* (1993)
- [3] Soille, P., “Morphological Image Analysis: Principles and Applications”, Chapter 6, 2nd edition (2003), ISBN 3540429883.
- [Lee94] T.-C. Lee, R.L. Kashyap and C.-N. Chu, Building skeleton models via 3-D medial surface/axis thinning algorithms. *Computer Vision, Graphics, and Image Processing*, 56(6):462-478, 1994.
- [Zha84] A fast parallel algorithm for thinning digital patterns, T. Y. Zhang and C. Y. Suen, *Communications of the ACM*, March 1984, Volume 27, Number 3.
- [Lee94] T.-C. Lee, R.L. Kashyap and C.-N. Chu, Building skeleton models via 3-D medial surface/axis thinning algorithms. *Computer Vision, Graphics, and Image Processing*, 56(6):462-478, 1994.
- [1] Z. Guo and R. W. Hall, “Parallel thinning with two-subiteration algorithms,” Comm. ACM, vol. 32, no. 3, pp. 359-373, 1989. DOI:[10.1145/62065.62074](https://doi.org/10.1145/62065.62074)
- [2] Lam, L., Seong-Whan Lee, and Ching Y. Suen, “Thinning Methodologies-A Comprehensive Survey,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol 14, No. 9, p. 879, 1992. DOI:[10.1109/34.161346](https://doi.org/10.1109/34.161346)
- [1] https://en.wikipedia.org/wiki/Top-hat_transform
- [1] Le Besnerais, G., & Champagnat, F. (2005, September). Dense optical flow by iterative local window registration. In *IEEE International Conference on Image Processing 2005* (Vol. 1, pp. I-137). IEEE. DOI:[10.1109/ICIP.2005.1529706](https://doi.org/10.1109/ICIP.2005.1529706)
- [2] Plyer, A., Le Besnerais, G., & Champagnat, F. (2016). Massively parallel Lucas Kanade optical flow for real-time video processing applications. *Journal of Real-Time Image Processing*, 11(4), 713-730. DOI:[10.1007/s11554-014-0423-0](https://doi.org/10.1007/s11554-014-0423-0)
- [1] Zach, C., Pock, T., & Bischof, H. (2007, September). A duality based approach for realtime TV-L 1 optical flow. In *Joint pattern recognition symposium* (pp. 214-223). Springer, Berlin, Heidelberg. DOI:[10.1007/978-3-540-74936-3_22](https://doi.org/10.1007/978-3-540-74936-3_22)

- [2] Wedel, A., Pock, T., Zach, C., Bischof, H., & Cremers, D. (2009). An improved algorithm for TV-L 1 optical flow. In Statistical and geometrical approaches to visual motion analysis (pp. 23-45). Springer, Berlin, Heidelberg. [DOI:10.1007/978-3-642-03061-1_2](https://doi.org/10.1007/978-3-642-03061-1_2)
- [3] Pérez, J. S., Meinhardt-Holzapfel, E., & Facciolo, G. (2013). TV-L1 optical flow estimation. *Image Processing On Line*, 2013, 137-150. [DOI:10.5201/ipol.2013.26](https://doi.org/10.5201/ipol.2013.26)
- [1] Manuel Guizar-Sicairos, Samuel T. Thurman, and James R. Fienup, “Efficient subpixel image registration algorithms,” *Optics Letters* 33, 156-158 (2008). [DOI:10.1364/OL.33.000156](https://doi.org/10.1364/OL.33.000156)
- [2] P. Anuta, Spatial registration of multispectral and multitemporal digital imagery using fast Fourier transform techniques, *IEEE Trans. Geosci. Electron.*, vol. 8, no. 4, pp. 353–368, Oct. 1970. [DOI:10.1109/TGE.1970.271435](https://doi.org/10.1109/TGE.1970.271435).
- [3] C. D. Kuglin D. C. Hines. The phase correlation image alignment method, Proceeding of IEEE International Conference on Cybernetics and Society, pp. 163-165, New York, NY, USA, 1975, pp. 163–165.
- [4] James R. Fienup, “Invariant error metrics for image reconstruction” *Optics Letters* 36, 8352-8357 (1997). [DOI:10.1364/AO.36.008352](https://doi.org/10.1364/AO.36.008352)
- [5] Dirk Padfield. Masked Object Registration in the Fourier Domain. *IEEE Transactions on Image Processing*, vol. 21(5), pp. 2706-2718 (2012). [DOI:10.1109/TIP.2011.2181402](https://doi.org/10.1109/TIP.2011.2181402)
- [6] D. Padfield. “Masked FFT registration”. In Proc. Computer Vision and Pattern Recognition, pp. 2918-2925 (2010). [DOI:10.1109/CVPR.2010.5540032](https://doi.org/10.1109/CVPR.2010.5540032)
- [1] J. Batson & L. Royer. Noise2Self: Blind Denoising by Self-Supervision, International Conference on Machine Learning, p. 524-533 (2019).
- [1] R.R. Coifman and D.L. Donoho. “Translation-Invariant De-Noising”. *Wavelets and Statistics, Lecture Notes in Statistics*, vol.103. Springer, New York, 1995, pp.125-150. [DOI:10.1007/978-1-4612-2544-7_9](https://doi.org/10.1007/978-1-4612-2544-7_9)
- [1] C. Tomasi and R. Manduchi. “Bilateral Filtering for Gray and Color Images.” *IEEE International Conference on Computer Vision* (1998) 839-846. [DOI:10.1109/ICCV.1998.710815](https://doi.org/10.1109/ICCV.1998.710815)
- [1] J. Batson & L. Royer. Noise2Self: Blind Denoising by Self-Supervision, International Conference on Machine Learning, p. 524-533 (2019).
- [1] A. Buades, B. Coll, & J-M. Morel. A non-local algorithm for image denoising. In CVPR 2005, Vol. 2, pp. 60-65, IEEE. [DOI:10.1109/CVPR.2005.38](https://doi.org/10.1109/CVPR.2005.38)
- [2] J. Darbon, A. Cunha, T.F. Chan, S. Osher, and G.J. Jensen, Fast nonlocal filtering applied to electron cryomicroscopy, in 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro, 2008, pp. 1331-1334. [DOI:10.1109/ISBI.2008.4541250](https://doi.org/10.1109/ISBI.2008.4541250)
- [3] Jacques Froment. Parameter-Free Fast Pixelwise Non-Local Means Denoising. *Image Processing On Line*, 2014, vol. 4, pp. 300-326. [DOI:10.5201/ipol.2014.120](https://doi.org/10.5201/ipol.2014.120)
- [4] A. Buades, B. Coll, & J-M. Morel. Non-Local Means Denoising. *Image Processing On Line*, 2011, vol. 1, pp. 208-212. [DOI:10.5201/ipol.2011.bcm_nlm](https://doi.org/10.5201/ipol.2011.bcm_nlm)
- [1] Tom Goldstein and Stanley Osher, “The Split Bregman Method For L1 Regularized Problems”, [https://ww3.math.ucla.edu/camreport/cam08-29.pdf](https://www.math.ucla.edu/camreport/cam08-29.pdf)
- [2] Pascal Getreuer, “Rudin–Osher–Fatemi Total Variation Denoising using Split Bregman” in *Image Processing On Line* on 2012-05-19, https://www.ipol.im/pub/art/2012/g-tvd/article_lr.pdf
- [3] https://web.math.ucsb.edu/~cgarcia/UGProjects/BregmanAlgorithms_JacquelineBush.pdf
- [4] https://en.wikipedia.org/wiki/Total_variation_denoising
- [1] A. Chambolle, An algorithm for total variation minimization and applications, *Journal of Mathematical Imaging and Vision*, Springer, 2004, 20, 89-97.

- [2] https://en.wikipedia.org/wiki/Total_variation_denoising
- [1] Chang, S. Grace, Bin Yu, and Martin Vetterli. “Adaptive wavelet thresholding for image denoising and compression.” *Image Processing, IEEE Transactions on* 9.9 (2000): 1532-1546. DOI:10.1109/83.862633
- [2] D. L. Donoho and I. M. Johnstone. “Ideal spatial adaptation by wavelet shrinkage.” *Biometrika* 81.3 (1994): 425-455. DOI:10.1093/biomet/81.3.425
- [1] D. L. Donoho and I. M. Johnstone. “Ideal spatial adaptation by wavelet shrinkage.” *Biometrika* 81.3 (1994): 425-455. DOI:10.1093/biomet/81.3.425
- [1] S.B.Damelin and N.S.Hoang. “On Surface Completion and Image Inpainting by Biharmonic Functions: Numerical Aspects”, *International Journal of Mathematics and Mathematical Sciences*, Vol. 2018, Article ID 3950312 DOI:10.1155/2018/3950312
- [2] C. K. Chui and H. N. Mhaskar, MRA Contextual-Recovery Extension of Smooth Functions on Manifolds, *Appl. and Comp. Harmonic Anal.*, 28 (2010), 104-113, DOI:10.1016/j.acha.2009.04.004
- [1] https://en.wikipedia.org/wiki/Richardson%E2%80%93Lucy_deconvolution
- [1] Sternberg, Stanley R. “Biomedical image processing.” *Computer* 1 (1983): 22-34. DOI:10.1109/MC.1983.1654163
- [1] François Orieux, Jean-François Giovannelli, and Thomas Rodet, “Bayesian estimation of regularization and point spread function parameters for Wiener-Hunt deconvolution”, *J. Opt. Soc. Am. A* 27, 1593-1607 (2010)
<https://www.osapublishing.org/josaa/abstract.cfm?URI=josaa-27-7-1593>
<https://hal.archives-ouvertes.fr/hal-00674508>
- [1] Miguel Arevalillo Herraez, David R. Burton, Michael J. Lalor, and Munther A. Gdeisat, “Fast two-dimensional phase-unwrapping algorithm based on sorting by reliability following a noncontinuous path”, *Journal Applied Optics*, Vol. 41, No. 35 (2002) 7437,
- [2] Abdul-Rahman, H., Gdeisat, M., Burton, D., & Lalor, M., “Fast three-dimensional phase-unwrapping algorithm based on sorting by reliability following a non-continuous path. In W. Osten, C. Gorecki, & E. L. Novak (Eds.), *Optical Metrology* (2005) 32–40, International Society for Optics and Photonics.
- [1] François Orieux, Jean-François Giovannelli, and Thomas Rodet, “Bayesian estimation of regularization and point spread function parameters for Wiener-Hunt deconvolution”, *J. Opt. Soc. Am. A* 27, 1593-1607 (2010)
<https://www.osapublishing.org/josaa/abstract.cfm?URI=josaa-27-7-1593>
<https://hal.archives-ouvertes.fr/hal-00674508>
- [2] B. R. Hunt “A matrix theory proof of the discrete convolution theorem”, *IEEE Trans. on Audio and Electroacoustics*, vol. au-19, no. 4, pp. 285-288, dec. 1971
- [1] Kass, M.; Witkin, A.; Terzopoulos, D. “Snakes: Active contour models”. *International Journal of Computer Vision* 1 (4): 321 (1988). DOI:10.1007/BF00133570
- [1] An Active Contour Model without Edges, Tony Chan and Luminita Vese, *Scale-Space Theories in Computer Vision*, 1999, DOI:10.1007/3-540-48236-9_13
- [2] Chan-Vese Segmentation, Pascal Getreuer *Image Processing On Line*, 2 (2012), pp. 214-224, DOI:10.5201/ipol.2012.g-cv
- [3] The Chan-Vese Algorithm - Project Report, Rami Cohen, 2011 arXiv:1107.2782
- [1] <https://cellprofiler.org>
- [2] <https://github.com/CellProfiler/CellProfiler/blob/082930ea95add7b72243a4fa3d39ae5145995e9c/cellprofiler/modules/identifysecondaryobjects.py#L559>

- [1] Efficient graph-based image segmentation, Felzenszwalb, P.F. and Huttenlocher, D.P. International Journal of Computer Vision, 2004
- [1] A Morphological Approach to Curvature-based Evolution of Curves and Surfaces, Pablo Márquez-Neila, Luis Baumela, Luis Álvarez. In IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI), 2014, DOI:10.1109/TPAMI.2013.106
- [1] A Morphological Approach to Curvature-based Evolution of Curves and Surfaces, Pablo Márquez-Neila, Luis Baumela, Luis Álvarez. In IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI), 2014, DOI:10.1109/TPAMI.2013.106
- [1] Quick shift and kernel methods for mode seeking, Vedaldi, A. and Soatto, S. European Conference on Computer Vision, 2008
- [1] Leo Grady, Random walks for image segmentation, IEEE Trans Pattern Anal Mach Intell. 2006 Nov;28(11):1768-83. DOI:10.1109/TPAMI.2006.233.
- [1] Radhakrishna Achanta, Appu Shaji, Kevin Smith, Aurelien Lucchi, Pascal Fua, and Sabine Süsstrunk, SLIC Superpixels Compared to State-of-the-art Superpixel Methods, TPAMI, May 2012. DOI:10.1109/TPAMI.2012.120
- [2] <https://www.epfl.ch/labs/ivrl/research/slic-superpixels/#SLICO>
- [3] Irving, Benjamin. “maskSLIC: regional superpixel generation with application to local pathology characterisation in medical images.”, 2016, arXiv:1606.09518
- [4] <https://github.com/scikit-image/scikit-image/issues/3722>
- [1] https://en.wikipedia.org/wiki/Watershed_%28image_processing%29
- [2] <http://cmm.ensmp.fr/~beucher/wtshed.html>
- [3] Peer Neubert & Peter Protzel (2014). Compact Watershed and Preemptive SLIC: On Improving Trade-offs of Superpixel Segmentation Algorithms. ICPR 2014, pp 996-1001. DOI:10.1109/ICPR.2014.181 https://www.tu-chemnitz.de/etit/proaut/publications/cws_pSLIC_ICPR.pdf
- [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)
- [1] Xie, Yonghong, and Qiang Ji. “A new efficient ellipse detection method.” Pattern Recognition, 2002. Proceedings. 16th International Conference on. Vol. 2. IEEE, 2002
- [1] A. Kingston and I. Svalbe, “Projective transforms on periodic discrete image arrays,” in P. Hawkes (Ed), Advances in Imaging and Electron Physics, 139 (2006)
- [1] F.C. Crow, “Summed-area tables for texture mapping,” ACM SIGGRAPH Computer Graphics, vol. 18, 1984, pp. 207-212.
- [1] AC Kak, M Slaney, “Principles of Computerized Tomographic Imaging”, IEEE Press 1988.
- [2] 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
- [1] AC Kak, M Slaney, “Principles of Computerized Tomographic Imaging”, IEEE Press 1988.
- [2] AH Andersen, AC Kak, “Simultaneous algebraic reconstruction technique (SART): a superior implementation of the ART algorithm”, Ultrasonic Imaging 6 pp 81–94 (1984)
- [3] S Kaczmarz, “Angenäherte auflösung von systemen linearer gleichungen”, Bulletin International de l’Academie Polonaise des Sciences et des Lettres 35 pp 355–357 (1937)
- [4] 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.

- [5] Kaczmarz' method, Wikipedia, https://en.wikipedia.org/wiki/Kaczmarz_method
- [1] 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.
- [2] 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.
- [1] 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.
- [1] http://persci.mit.edu/pub_pdfs/pyramid83.pdf
- [1] http://persci.mit.edu/pub_pdfs/pyramid83.pdf
- [1] http://persci.mit.edu/pub_pdfs/pyramid83.pdf
- [2] http://sepwww.stanford.edu/data/media/public/sep/morgan/texturmatch/paper_html/node3.html
- [1] http://persci.mit.edu/pub_pdfs/pyramid83.pdf
- [1] AC Kak, M Slaney, "Principles of Computerized Tomographic Imaging", IEEE Press 1988.
- [2] 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
- [1] <http://entropymine.com/imageworsener/pixelmixing/>
- [1] Wikipedia, "Affine transformation", https://en.wikipedia.org/wiki/Affine_transformation#Image_transformation
- [2] Wikipedia, "Shear mapping", https://en.wikipedia.org/wiki/Shear_mapping
- [1] Hartley, Richard, and Andrew Zisserman. Multiple view geometry in computer vision. Cambridge university press, 2003.
- [1] https://en.wikipedia.org/wiki/Rotation_matrix#In_three_dimensions
- [1] Hartley, Richard, and Andrew Zisserman. Multiple view geometry in computer vision. Cambridge university press, 2003.
- [1] <https://en.wikipedia.org/wiki/Hyperrectangle>