



**HAL**  
open science

## Analyzing microtomography data with Python and the scikit-image library

Emmanuelle Gouillart, Juan Nunez-Iglesias, Stéfan van Der Walt

► **To cite this version:**

Emmanuelle Gouillart, Juan Nunez-Iglesias, Stéfan van Der Walt. Analyzing microtomography data with Python and the scikit-image library. *Advanced Structural and Chemical Imaging*, 2017, 2, pp.18. 10.1186/s40679-016-0031-0 . hal-01448271

**HAL Id: hal-01448271**

**<https://hal.science/hal-01448271>**

Submitted on 27 Jan 2017

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

## RESEARCH

# Analyzing microtomography data with Python and the scikit-image library

Emmanuelle Gouillart<sup>1\*</sup>, Juan Nunez-Iglesias<sup>2</sup> and Stéfan van der Walt<sup>3</sup>**Abstract**

The exploration and processing of images is a vital aspect of the scientific workflows of many X-ray imaging modalities. Users require tools that combine interactivity, versatility, and performance.

`scikit-image` is an open-source image processing toolkit for the Python language that supports a large variety of file formats and is compatible with 2-D and 3-D images. The toolkit exposes a simple programming interface, with thematic modules grouping functions according to their purpose, such as image restoration, segmentation, and measurements. `scikit-image` users benefit from a rich scientific Python ecosystem that contains many powerful libraries for tasks such as visualization or machine learning. `scikit-image` combines a gentle learning curve, versatile image processing capabilities, and the scalable performance required for the high-throughput analysis of X-ray imaging data.

**Keywords:** `scikit-image`; Python; image processing library; 3-D image

**Introduction**

The acquisition time of synchrotron tomography images has decreased dramatically over the last decade, from hours to seconds (Maire and Withers, 2014). New modalities such as single-bunch imaging provide a time resolution down to the nanosecond for radiography (Rack *et al.*, 2014). However, the time subsequently spent in processing the images has not decreased as much, so that the outcome of a successful synchrotron imaging run often takes weeks or even months to be transformed into scientific results.

Transforming billions of pixels and voxels to a few meaningful figures represents a tremendous data reduction. Often, the sequence of operations needed to produce these data is not known beforehand, or might be altered due to artifacts (Marone *et al.*, 2010), or to an unforeseen evolution of the sample. Image processing necessarily involves trial and error phases to choose the processing workflow. Therefore, image processing tools need to offer at the same time enough flexibility of use, a variety of algorithms, and efficient implementations to allow for fast iterations while adjusting the workflow.

Several software applications and libraries are available to synchrotron users to process their images. ImageJ (Abràmoff *et al.*, 2004; Schneider *et al.*, 2012) and its distribution Fiji (Schindelin *et al.*, 2012) is a popular general-purpose tool for 2-D and 3-D images, thanks to its intuitive menus and graphical tools, and the wealth of plugins contributed by a vivid community (Schindelin *et al.*, 2015). Software specialized in analyzing synchrotron data are available as well, such as XRDUA (De Nolf *et al.*, 2014) for diffraction images obtained in powder diffraction analysis, or For 3-D images, commercial tools such as Avizo 3D software (TM), or ToolIP/MAVikit (Fraunhofer Institute for Industrial Mathematics ITWM, 2016) are appreciated for an intuitive graphical pipeline and advanced 3D visualization. Some synchrotrons have even developed their own tools for volume processing, such as Pore3D (Brun *et al.*, 2010) at the Elettra facility. Alternatively, the use of a programming language gives finer control, better reproducibility, and more complex analysis possibilities, provided classical processing algorithms can be called from libraries – thereby limiting the complexity of the programming task and the risk of bugs. Matlab (TM) and its image processing toolbox are popular in the academic community of computer vision and image processing. The Python language is widely used in the scientific world and in synchrotron facilities. As a general-purpose language, Python is used in synchrotrons to control device servers (Brookhaven National Lab, 2016; Coutinho, T., 2016; Sugandhi *et al.*, 2016), to access raw data of X-ray detectors (Knudsen *et al.*, 2013), to reconstruct

\* Correspondence: [emmanuelle.gouillart@nsup.org](mailto:emmanuelle.gouillart@nsup.org)

<sup>1</sup>Surface du Verre et Interfaces, UMR 125 CNRS/Saint-Gobain, 93303 Aubervilliers, France

Full list of author information is available at the end of the article

tomography volumes from radiographs (Gürsoy et al., 2014; Mirone et al., 2014), and in data processing packages for macromolecular crystallography (Adams et al., 2010), azimuthal integration of diffraction data (Ashiotis et al., 2015), or fluorescence analysis (Solé et al., 2007; V. Armando Sole, 2016).

`scikit-image` (Van der Walt et al., 2014) is a general-purpose image processing library for the Python language, and a component of the ecosystem of Python scientific modules commonly known as Scientific Python (Oliphant, 2007). Like the rest of the ecosystem, `scikit-image` is released under a permissive open-source license and is available free of charge. Most of `scikit-image` is compatible with both 2-D and 3-D images, so that it can be used for a large number of imaging modalities, such as microscopy, radiography or tomography. In this article, we explain how `scikit-image` can be used for processing data acquired in X-ray imaging experiments, with a focus on microtomography 3-D images. This article does not intend to be a pedagogical tutorial on `scikit-image` for X-ray imaging, but rather to explain the rationale behind the package, and provide various examples of its capabilities.

## Methods – Overview and first steps

In this section, we provide a short overview of the typical use patterns of `scikit-image`, illustrated by short snippets of code. Since Python is a programming language, the user interacts with data objects and images through code, which is either entered and executed in an interactive interpreter, or written in text files (so-called scripts) that are executed.

Images are manipulated as numerical arrays, each with a single, uniform data type. This common format guarantees interoperability with other libraries and straightforward access to and interpretation of computer memory. The N-dimensional (2-D, 3-D, ...) numerical array object is provided by the NumPy module (Van Der Walt et al., 2011).

In image processing in Python, one of the first tasks then is to generate NumPy arrays, which is often achieved by reading data from files. We read one 2-dimensional image from a file and display it as follows:

```
from skimage import io
im = io.imread('figure_gallery.png')
io.imshow(im)
```

`skimage` is the name under which `scikit-image` is imported in Python code. Note that functions (such as `imread` that reads an image file, or `imshow` that displays an image) are found in thematic submodules of `skimage`, such as `io` for Input/Output.

A stack of 2-D images, such as tomography slices generated by a reconstruction algorithm, can be opened as an image collection or a 3-D array:

```
>>> from skimage import io
>>> image_collection = io.imread_collection('*.*.tif')
>>> image_3d = io.concatenate(image_collection)
>>> print(image_3d.shape)
(800, 1024, 1024)
```

Raw data formats can be opened using the NumPy functions `fromfile` (to load the array into memory) or `memmap` (to keep the array on disk). The following code creates an array from a raw image file of unsigned 16-bit integers with a header of 1024 bytes

```
im = np.memmap('image.edf', shape=(2048, 2048),
              offset=1024, dtype=np.uint16)
```

For every raw data specification, it is thus very easy to write a reader using `np.memmap` (see for example <https://github.com/jni/python-redshirt>). hdf5 files are accessed using modules such as `h5py`, `pytables`.

`scikit-image` has a simple Application Programming Interface (API), based almost exclusively on functions. Most functions take an image (*i.e.* a multi-dimensional array) as input parameter:

```
>>> from skimage import filters
>>> im_gaussian = filters.gaussian(im)
```

Optional parameters can be passed as Python *keyword arguments*, in addition to the image parameter.

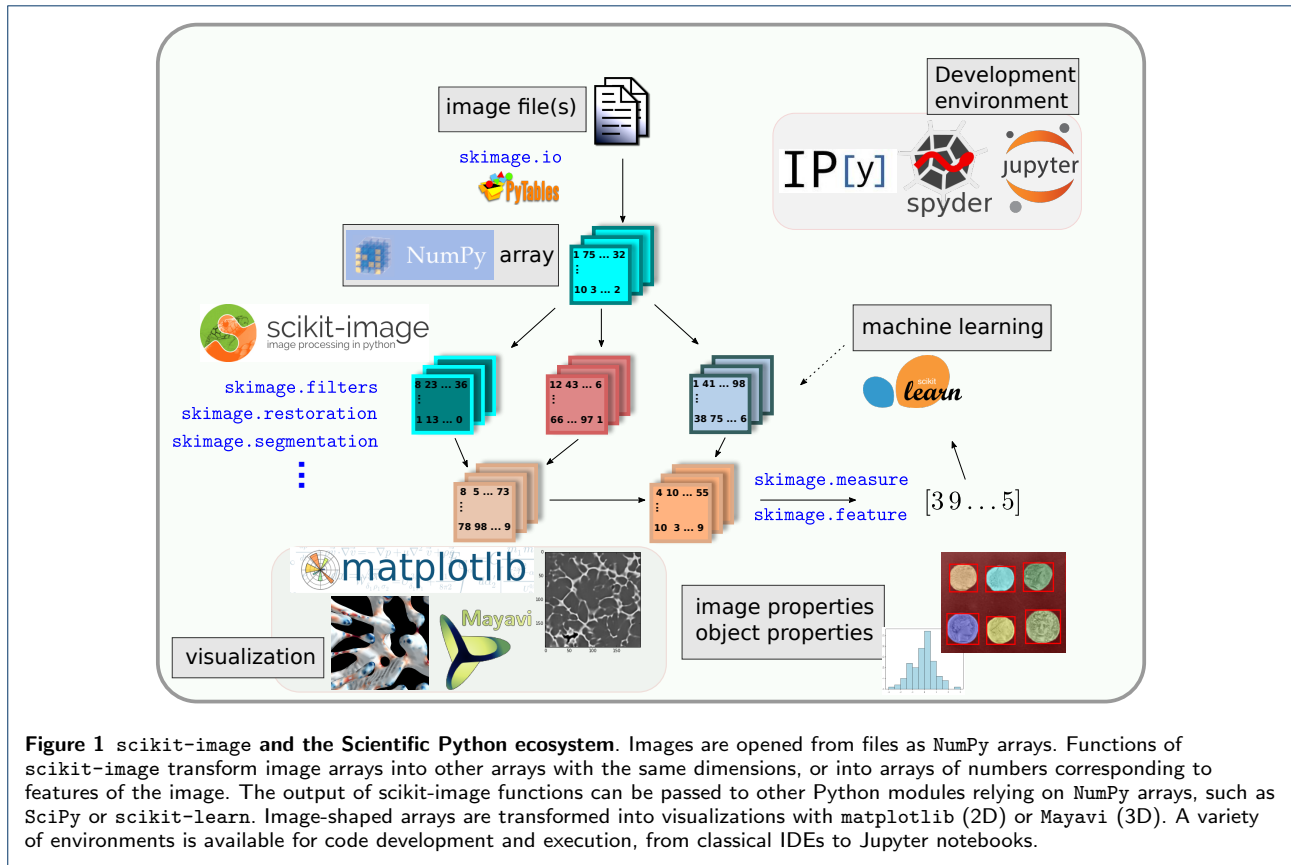
```
>>> im_gaussian = filters.gaussian(im, sigma=3,
                                mode='wrap')
```

A few functions require several arrays to be passed, such as the watershed segmentation algorithm that takes as parameters the image to be segmented, and an image of markers from which labels are propagated:

```
>>> from skimage import morphology
>>> labels = morphology.watershed(im, markers)
```

Therefore, the image processing workflow can be seen as a directed graph (a richer structure than a linear pipeline), where nodes are image-shaped arrays, and edges are functions of `scikit-image` transforming the arrays (see Fig. 1).

Most functions transparently handle 2-D, 3-D, or even higher-dimensional images as arguments, so the



same functions can be used to process tomography, microscopy, or natural images. The rest raise an error when passed a 3-D argument:

```
>>> filters.prewitt(im)
[... ]
ValueError: The parameter 'image' must be a 2-↔
dimensional array
```

However, the proportion of functions supporting 3D images is always increasing, thanks to the many contributors to the library.

While the majority of functions return processed images, returns can also be numerical value(s) such as pixel coordinates of objects of interest or statistical information about the image:

```
>>> from skimage import exposure
>>> counts, bins = exposure.histogram(im)
>>> counts.shape
(256,)
```

## The Python ecosystem

The benefits of scikit-image for image processing come not only from the features of the package alone,

but also from the rich environment surrounding scientific Python (Oliphant, 2007; Perez et al., 2011). Fig. 1 illustrates how several components of this ecosystem combine into a sophisticated image processing workflow.

*NumPy arrays* are the cornerstone of the Scientific Python ecosystem, and of scikit-image operations in particular. Cropping or downsampling an image, or retrieving pixels corresponding to a given label in a segmentation are all NumPy “one-liners”. To illustrate the compactness of NumPy code, consider modifying pixel values below a threshold. This operation can be written as

```
im[im < 0.5] = 0
```

exploiting the ability to index arrays with boolean arrays, also called *masking*. NumPy uses memory sparingly and avoids making new copies of arrays whenever possible, an important requirement when dealing with the gigabyte-sized images of tomography. For example, cropping a subvolume as follows does not create a copy of the original array

```
sub_volume = im[100:-100, 100:-100, 100:-100]
```

but instead refers to the correct memory offsets in the original.

*Interpreter and development environment.* While several interpreters are available to execute Python instructions and scripts interactively, the most popular in the scientific world is IPython (Pérez and Granger, 2007; Rossant, 2015). IPython is an advanced interpreter, which integrates syntax highlighting, text auto-completion, a debugger, introspection and profiling methods, and online help. Several Integrated Development Environments (IDEs) come bundled with IPython, together with other components such as a text editor. Notable examples include Spyder (Fig. 2), PyCharm, and Visual Studio Code.

The Jupyter notebook (Kluyver et al., 2016) is a web application that grew out of the IPython project. Jupyter notebooks provide an interactive development environment within a web browser, where live code can be enriched by explanatory text, equations and visualizations (Fig. 3). Jupyter notebooks render directly as webpages on GitHub, making them a straightforward tool to publish online a script and its output. As of July 2016, more than 500,000 Jupyter notebooks were posted on GitHub, demonstrating their wide adoption by the community as workflow-sharing tools (<http://archive.ipython.org/media/SciPy2016JupyterLab.pdf>).

*Visualization libraries.* Visualizing images is an important component of the image processing workflow, used to inspect the final result and to adjust the parameters of intermediate processing operations. `matplotlib` (Hunter et al., 2007) is the most popular 2D plotting library of the Python ecosystem. It can be used to visualize 2D data such as color or grayscale images, and 1D data such as contour lines, outlines of segmented regions, histograms of gray levels, etc. Although `matplotlib` has simple 3D plotting capabilities, we recommend using the `mayavi` module (Ramaachandran and Varoquaux, 2011) for applications requiring advanced 3D visualization, such as tomography. `mayavi` is based on the VTK toolkit. It exposes a simple API for visualizing data passed as `numpy` arrays. For example, visualizing the surface of binary data can be written as

```
# synthetic binary array from skimage.data
im = data.binary_blobs(length=400, n_dim=3).←
    astype(np.uint8)
# visualization
from mayavi import mlab
mlab.contour3d(im)
mlab.outline()
```

(see Fig. 4 for the resulting visualization).

For more advanced visualizations, a large majority of VTK capabilities can be accessed through `mayavi`'s pipeline API. `mayavi` offers a good trade-off between simplicity of use for common operations, and accessibility to more sophisticated capabilities such as responsive visualizations.

*Advanced toolkits for signal processing and data science.* `scikit-image` is only one Python module that can be used for data processing, among many others. A very popular module is `scikit-learn` (Pedregosa et al., 2011), a Python module for machine learning using `NumPy` arrays. Local features of an image (such as local statistics of gray levels, or geometric points of interest) or features of segmented objects (e.g. geometrical and intensity characteristics of segmented particles) can be extracted with functions from `skimage.feature` (see Fig. 1). It is then possible to use a *classification* algorithm from `scikit-learn` to label pixels (a segmentation task) or to classify whole images or objects that have already been segmented. The near-universal use of `NumPy` arrays ensures the interoperability between these packages, so that just a few lines of code are sufficient to create these sophisticated workflows.

The modularity of the Scientific Python ecosystem may be confusing at first sight, but the core modules of this ecosystem are almost perfectly compatible, thanks to the shared use of `NumPy` arrays and common development practices (although they are developed in parallel by different teams). Several “distributions”, such as Anaconda or Canopy, bundle together the most popular libraries, including `scikit-image`.

## Results

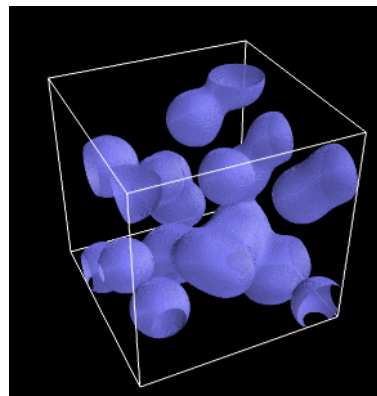
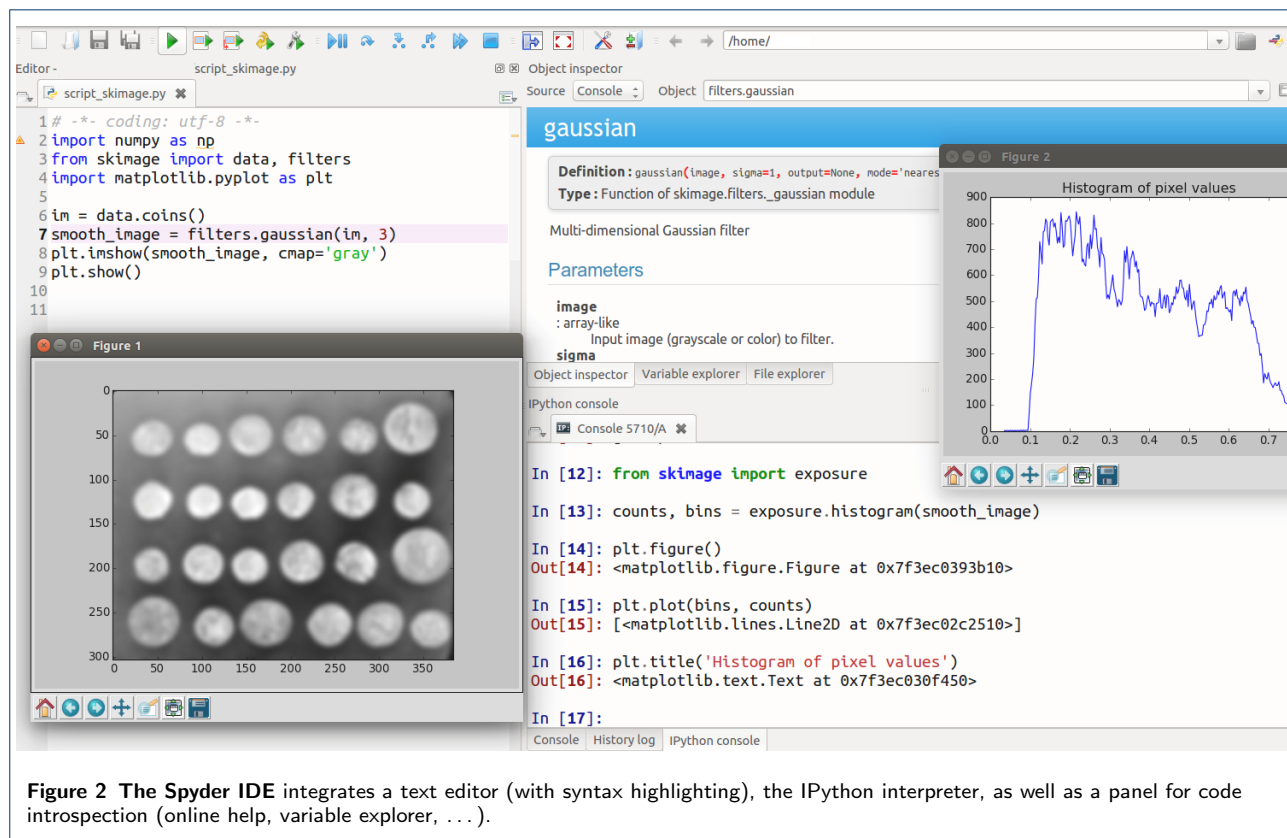


Figure 4 Simple 3-D visualization realized with Mayavi.



## Image processing capabilities

*Capabilities.* `scikit-image` offers most classical image processing operations, such as exposure and color adjustment, filtering, segmentation, feature extraction, geometric transformations, and measurements of region characteristics. In addition to common operations, some advanced algorithms are also implemented, a selection of which is illustrated in Fig. 5. In the following, we briefly illustrate how `scikit-image` can be used for some typical image processing tasks encountered when analyzing tomographic images: denoising, mid-range feature detection, segmentation and measurement of region properties. For the sake of brevity, other tasks such as contrast manipulation or geometric transformations are not described here; the interested reader is referred to the documentation of `scikit-image`.

Tomographic images often suffer from artifacts or poor signal-to-noise ratio. Therefore, *denoising* data is often the first step of an image processing workflow. Several denoising filters are available for restoring these images, ranging from general-purpose median and bilateral filters to those more suited to specific applications. For example, total-variation denoising (Chambolle, 2004; Getreuer, 2012) is ideal for restoring piecewise-constant images (see Fig. 5 a)), such as

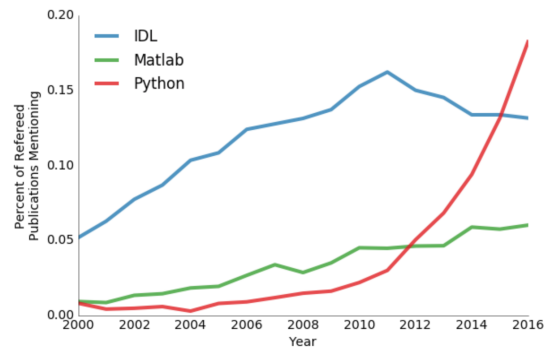
images with a small number of phases encountered in materials science (Bouttes et al., 2015). Conversely, images with a fine-grained texture are better preserved with non-local means denoising, a patch-based algorithm (Buades et al., 2005) (see Fig. 5 a)).

Detecting the presence of objects or extracting pixels corresponding to objects (a task known as segmentation) is an important task of image analysis for medical or materials science applications. `scikit-image` offers a wide variety of functions for *detecting geometrical features* of interest in an image. In order to detect thin boundaries, the ridges of an image can be identified as regions for which the leading eigenvalue of the local Hessian matrix is high (see Fig. 5 b)). In the Fourier space, peaks in 2D Bragg diffraction patterns can be extracted using blob detection methods (Ashiotis et al., 2015), such as the Laplacian of Gaussian method (see Fig. 5 b)).

*Segmentation* of regions of interest can be achieved using one of various strategies, depending on the characteristics of the image. Images with a clear contrast between regions can be segmented automatically thanks to several thresholding algorithms, including an adaptive local thresholding algorithm aimed at images with contrast variations. Super-pixel algorithms (Achanta et al., 2012; Felzenszwalb and Huttenlocher,

```
In [8]: def trendlines(queries, norm=False):
fig, ax = plt.subplots()
for lang in languages:
counts = queries[lang]
x = counts[:, 0]
y = np.copy(counts[:, 1])
if norm:
y /= counts[:, 2]
ax.plot(x, y * 100, label=lang, lw=4, alpha=0.8)
ax.set_xlim(np.min(x), np.max(x))
ax.set_xlabel('Year')
ax.set_ylabel('Percent of Refereed\nPublications Mentioning')
ax.legend(loc='upper left', frameon=False)
remove_chartjunk(ax, ['top', 'right'])

In [9]: trendlines(results, norm=True)
plt.savefig('python-vs-matlab-vs-IDL-in-astro.pdf')
```



There you have it: some time in early 2015, Python overtook IDL as the most mentioned (and probably the most used) programming language in astronomy!

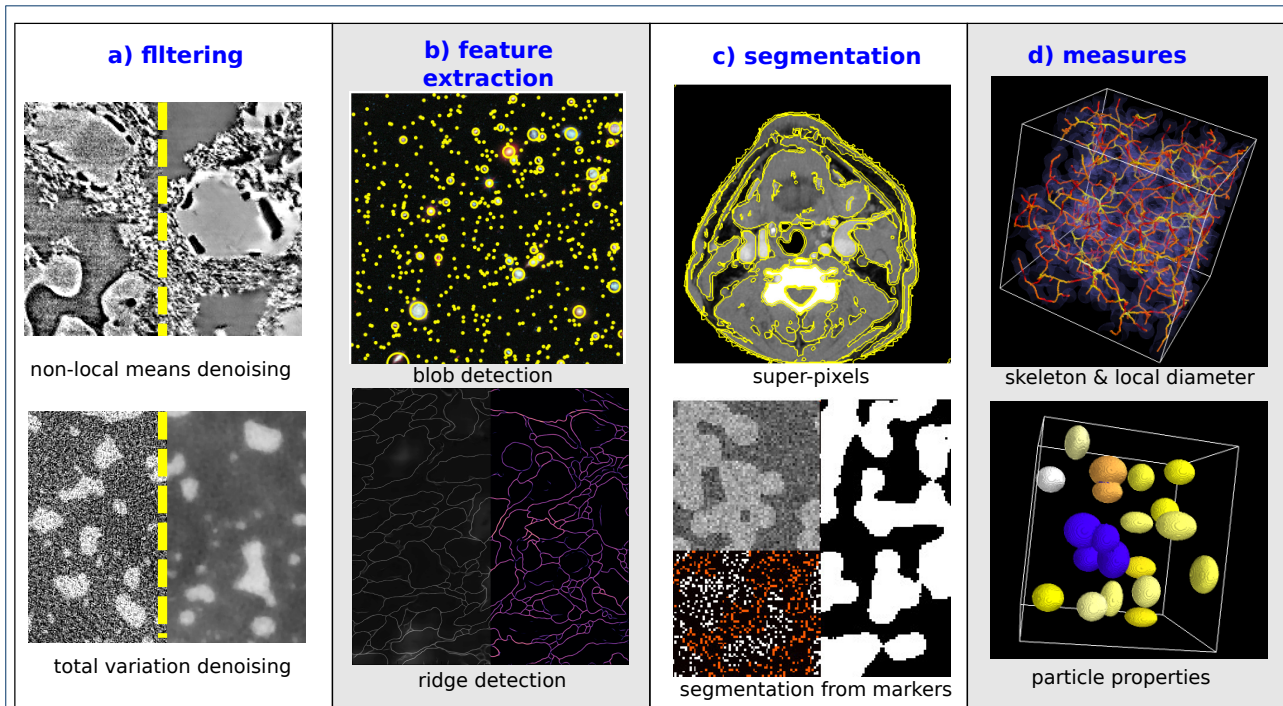
**Figure 3** The Jupyter notebook allows mixing of computer code (top), plot and text output (middle), and free-form narrative text (bottom). This makes it ideal to record and report code-based analyses. Screenshot from Nunez-Iglesias et al. (2016).

2004) create an over-segmentation of images in superpixels, by grouping pixels that are close together both in color- and spatial distance (see Fig. 5 c)). Region-growing algorithms, such as the morphological watershed or the random walker (Grady, 2006), propagate the labels of user-defined markers through the image (see Fig. 5 c)). The active contour algorithm (Kass et al., 1988) fits snake contours to features of the image, such as edges or high-brightness regions.

Following segmentation, the characteristics of labeled regions (particles, porosities, organs, ...) resulting from a segmentation can be measured using the `measure` submodule. The different connected components (e.g. bubbles or non-touching particles) of a binary image are labeled with the `measure.label` function. Properties of labeled regions such as size, extent, center of mass or mean intensity value are accessed with `measure.regionprops` (see Fig. 5 d)). Local characteristics of a region can be retrieved as well: Fig. 5 d) shows how the local diameter of open porosity is measured by combining a skeletonization of the porosity channels, and the distance transform to the other phase measured on the skeleton.

*Performance.* Given the large size of tomography datasets, the execution speed of image processing operations is of critical concern. `scikit-image` relies mostly on calls to NumPy operations, of which most are performed in optimized compiled code (C or Fortran). Performance-critical parts of `scikit-image` that cannot call efficient NumPy code are implemented in Cython. Cython (Behnel et al., 2011) is an extension of the Python language that supports explicit type declarations, and is compiled directly to C. Therefore, the performance of `scikit-image` can be close to the one of libraries written in a compiled language such as C++ or Java. For example, computing the watershed segmentation of a 2000x2000 array of floats into 1000 regions took about 1 s using 1 CPU of an off-the-shelf laptop, and 10 s for a 256x256x256 image segmented into 2000 regions. Similar timescales were obtained with `mahotas`, a Python package implemented exclusively in C++ (Coelho, 2013) (with a slight advantage for `mahotas`).

However, basic `scikit-image` code runs on a single core. Computing workstations and servers used for X-ray imaging typically have several tens of cores. Paral-



**Figure 5 Typical image processing operations with scikit-image.** Data are synthetic, unless stated otherwise. **a) Filtering** - Top: non-local means denoising of an image with a fine-grained texture, acquired by *in situ* synchrotron microtomography during glass melting (Gouillart et al., 2012). Bottom: total-variation denoising of an image with two phases, corresponding to phase-separating silicate melts observed by *in situ tomography* (Bouttes et al., 2015). **b) Feature extraction** - Top: Hubble deep field (NASA, public domain), blob detection using the Laplacian of Gaussian method. Bottom: ridge detection using the leading eigenvalue of the Hessian matrix, neuron image from CREMI challenge (<https://cremi.org/data/>). **c) Segmentation** - Top: super-pixel segmentation of a CT slice of the human head (ChumpusRex, Wikipedia, 2016), using Felzenszwalb's algorithm (Felzenszwalb and Huttenlocher, 2004). Bottom: random walker segmentation (right) of noisy image (top-left corner), using histogram-determined markers (bottom-left corner). **d) Measures** - Top: visualization of local diameter (color-coded on the skeleton curve) of an interconnected phase (represented in violet). Bottom: particles color-coded according to their extent.

lization of the computing workflow can be achieved in multiple ways. The most trivial parallelization scheme consists of applying the same workflow to different images, on different cores. However, finer-grained parallelization is preferable when prototyping the processing workflow.

An easy solution consists in dividing an image into smaller images (with or without overlap, depending on the operation), and to apply the same operation on the different sub-images, on different cores. Creating overlapping chunks is easy with the dedicated function `view_as_windows` (or `view_as_blocks` for contiguous non-overlapping chunks):

```
>>> from skimage import util, data
>>> im = data.camera()
>>> im.shape
(512, 512)
>>> # chunks of size 134 with 8-pixel overlap
>>> chunks = util.view_as_windows(im, (134, ←
134), step=126)
>>> chunks.shape
(4, 4, 134, 134)
```

The `joblib` library enables easy parallel processing. Looping over the different blocks, and dispatching the computation over several cores, is realized with the following syntax:

```
>>> from joblib import Parallel, delayed
>>> filtered_chunks = Parallel(n_jobs=4)(←
delayed(filters.gaussian)(chunks[i, j]) for ←
i in range(4) for j in range(4))
```

`scikit-image` also offers experimental support for a more integrated parallel processing pipeline, thanks to the `dask` (Rocklin, M. and `dask contributors`, 2016) module:

```
filtered_im = util.apply_parallel(filters.←
gaussian, im, depth=8)
```

The size of chunks is determined automatically from the number of available cpus, or can be specified by the user.



*Caching* provides another tool to speed up data analysis. A situation that often arises is that, while prototyping a workflow, scripts (containing the image processing pipeline) are run several times to experiment with parameters. *joblib* provides a caching mechanism that avoids the repetition of function calls, if their arguments have not changed:

```
>>> from skimage import filters, data
>>> from joblib import Memory
>>> mem = Memory(cachedir='/tmp/joblib')
>>> median = mem.cache(filters.median)
>>> im = data.camera()
>>> filtered_im = median(im, np.ones((3, 3)))
-----
[Memory] Calling skimage.filters.rank.generic.<-
median...
median(array([[156, ..., 152],
             ...,
             [121, ..., 111]], dtype=uint8), array([[<-
             1., 1., 1.],
             [ 1., 1., 1.],
             [ 1., 1., 1.]])
-----median - 0.0s, 0.0min
>>> filtered_again = median(im, np.ones((3, 3))<-
)
>>> # the above call did not trigger another <-
evaluation
```

Finally, we note that the biggest performance improvements often come from the improvement of algorithms (as opposed to computing architectures only). For example, non-local means denoising (Buades et al., 2005) is a costly operation, since it requires several nested loops, on all pixels and on neighboring patches to be compared with the pixel-centered patch. Thanks to the implementation of a more recent algorithm (Darbon et al., 2008) that modifies the internal organization of loops, it was possible to improve execution time by a factor of roughly ten times. The large size of the *scikit-image* community makes it likely for algorithmic improvements to be discussed regularly. During the code review process, a close watch is also kept on memory consumption, since for large image sizes, transfers between computer memory (RAM) and CPU cache are often a serious performance bottleneck.

## Documentation

The quality of software documentation is (perhaps especially) important in software aimed at scientists. *scikit-image* users have access to several kinds of documentation. All functions are documented using the NumPy documentation standard (Pawlik et al., 2015), which is universal across all major Scientific Python packages. The standard include a description of all input and output variables and their data types, together with explanations of what each function does and how to use it. Function documentation is accessible online or within the development environment itself (IPython, Spyder, Jupyter Notebook...).

In addition, a graphical gallery of examples ([http://scikit-image.org/docs/dev/auto\\_examples/](http://scikit-image.org/docs/dev/auto_examples/)), part of which is displayed in Fig. 6, showcases graphical examples of common image processing operations. The examples are organized as an array of thumbnails with a short title (see Fig. 6 left). These thumbnails link to the webpage of the corresponding example, which features a mini-tutorial on the image processing method, the code needed to run the example and the figure generated by the example. Since the graphical gallery is an efficient way to inform users about the features of *scikit-image*, every new feature integrated in the package must include an example for the gallery. Longer tutorials and a more narrative documentation is available as well in the online User Guide of *scikit-image*. The User Guide explains in particular "big picture", foundational aspects of *scikit-image*, such as its use of NumPy arrays as images, or how the package interacts with other parts of the scientific Python ecosystem.

Finally, tutorials on *scikit-image* are available in various places, either as YouTube videos, or in the SciPy Lecture Notes (Varoquaux et al., 2016), a comprehensive online book of Scientific Python tutorials.

## Development and use of *scikit-image*

*Who uses scikit-image.* Estimating the number of active users of an open-source package is a difficult task. Download statistics, for example, largely overestimate the number of active users, all the more if the package is bundled with others in a software distribution, such as Anaconda or Canopy. A view closer to reality can be obtained by analyzing the statistics of visits of the online help, available on the project website. As of the first half of 2016, 20000 unique visitors visited the *scikit-image* website every month at <http://scikit-image.org/>, from 138 countries.

The *scikit-image* paper of 2014 (Van der Walt et al., 2014) has been cited by 120 research works (as of August 2016, according to Google Scholar), among which studies that used X-ray imaging in fields such as medical imaging (Blackledge et al., 2016; Malan et al., 2016; Shen et al., 2015), materials science (Bouttes et al., 2015) or geoscience (Schlüter et al., 2014).

*Development process.* *scikit-image* is developed by a diverse team of volunteers. More than 170 individuals have contributed to the package. The large number of developers and users key to project's sustainability. The development process takes place on GitHub <https://github.com/scikit-image/scikit-image>, where users and developers propose and discuss new contributions, report bugs or submit ideas for improvements. A release cycle of one or two releases every year ensures that new features are propagated to users on a regular basis.

## Discussion – Current limitations and challenges

While we emphasize the assets of `scikit-image` for processing X-ray images, one should be aware of current limitations.

*Speed of execution.* Although `scikit-image` approaches the speed of execution of compiled (C++, Java) code, it cannot reach the performance of code optimized for the GPU, or the hand-tuned CPU-specific optimizations found in OpenCV (Pulli et al., 2012). At the moment, `scikit-image` is not the best tool for ultrafast computations where the workflow is known beforehand, simple and stable. However, it is an excellent tool for exploring image data interactively and testing different algorithms – an important component of data processing in scientific work –, and its speed of execution is sufficient for processing gigabyte-sized tomographic images in seconds to minutes. Moreover, the multiprocessing capability of `scikit-image` is likely to improve in the near future.

*3-D compatibility.* Currently, about two thirds of `scikit-image` functions transparently handle 2D or 3D arrays, with the remainder limited to 2D analysis, often unnecessarily. Improved support for 3D and higher-dimensional volumes is on the project roadmap.

*Documentation for domain-specific applications.* Some image processing libraries or applications address a specific scientific domain, such as CellProfiler (Carpenter et al., 2006; Lamprecht et al., 2007) for biological images. The documentation of such projects often showcases examples that are close to the experience of the targeted community. Since `scikit-image` is application-agnostic, applications such as tomographic imaging are not mentioned in detail in the documentation of `scikit-image`. A possible improvement would be to write comprehensive tutorials addressing specific communities, and to refer to these tutorials from the main `scikit-image` documentation.

## Getting started

Scientists interested in experimenting with `scikit-image` are invited to read the installation instructions at <http://scikit-image.org/docs/stable/install.html>. `scikit-image` can be installed either bundled in a Scientific Python distribution, such as Anaconda (conda command line) or Canopy, or stand alone (along with its dependencies) using a installer/packager such as pip or Ubuntu's Aptitude.

The Getting Started section of the online User Guide ([http://scikit-image.org/docs/dev/user\\_guide/getting\\_started.html](http://scikit-image.org/docs/dev/user_guide/getting_started.html)) provides a good launch

pad for beginners, and gently leads into other sections of the user guide. A gallery of examples ([http://scikit-image.org/docs/dev/auto\\_examples/](http://scikit-image.org/docs/dev/auto_examples/)) lets users find applications close to their needs. Although most examples in the gallery use 2D images, many are applicable to 3D images as well.

Assistance on matters not covered by the documentation is provided on the dedicated mailing-list [scikit-image@googlegroups.com](mailto:scikit-image@googlegroups.com) or on Stack-Overflow <http://stackoverflow.com/questions/tagged/scikit-image>.

## Conclusion

`scikit-image` offers a wide variety of image processing algorithms, using a simple interface natively compatible with 2D and 3D images. It is well integrated into the Scientific Python ecosystem, so that it interfaces well with visualization libraries and other data processing packages. `scikit-image` has seen tremendous growth since its creation in 2009, both in terms of users and included features. In addition to the growing number of scientific teams that use `scikit-image` for processing images of various X-ray modalities, domain-specific tools are now using `scikit-image` as a dependency to build upon. Examples include `tomopy` (Gürsoy et al., 2014) for tomographic reconstruction, or `DIOPTAS` (Prescher and Prakapenka, 2015) for the reduction and exploration of X-ray diffraction data. It is likely that more application-specific software will benefit from depending on `scikit-image` in the future, since `scikit-image` strives to be domain-agnostic and to keep the function interface stable. On the end-user side, future work includes better integration of parallel processing capabilities, completion of full 3-D compatibility, an enriched narrative documentation, speed enhancements, and expansion of the set of supported algorithms.

### Competing interests

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

### Author's contributions

E. Gouillart, J. Nunez Iglesias and S. van der Walt conducted the research and wrote the paper. All authors read and approved the final manuscript.

### Acknowledgements

The authors gratefully acknowledge the work of the contributors of `scikit-image`, and thank S. Deville and D. Vandembroucq for a careful reading of the paper and useful suggestions. E. Gouillart acknowledges the support of ANR project EDDAM ANR-11-BS09-027. CREMI (<https://cremi.org/data/>) is acknowledged for the membrane image of Fig. 5 (b).

### Author details

<sup>1</sup>Surface du Verre et Interfaces, UMR 125 CNRS/Saint-Gobain, 93303 Aubervilliers, France. <sup>2</sup>Victorian Life Sciences Computation Initiative, University of Melbourne, Carlton, VIC, Australia. <sup>3</sup>Division of Applied Mathematics, Stellenbosch University, Stellenbosch, South Africa.

## References

- Abràmoff, M.D., Magalhães, P.J., Ram, S.J.: Image processing with imagej. *Biophotonics international* **11**(7), 36–42 (2004)
- Achanta, R., Shaji, A., Smith, K., Lucchi, A., Fua, P., Süsstrunk, S.: Slic superpixels compared to state-of-the-art superpixel methods. *IEEE transactions on pattern analysis and machine intelligence* **34**(11), 2274–2282 (2012)
- Adams, P.D., Afonine, P.V., Bunkóczi, G., Chen, V.B., Davis, I.W., Echols, N., Headd, J.J., Hung, L.-W., Kapral, G.J., Grosse-Kunstleve, R.W., et al.: Phenix: a comprehensive python-based system for macromolecular structure solution. *Acta Crystallographica Section D: Biological Crystallography* **66**(2), 213–221 (2010)
- Ashiotis, G., Deschilde, A., Nawaz, Z., Wright, J.P., Karkoulis, D., Picca, F.E., Kieffer, J.: The fast azimuthal integration Python library: pyFAI. *Journal of applied crystallography* **48**(2), 510–519 (2015)
- Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D.S., Smith, K.: Cython: The best of both worlds. *Computing in Science & Engineering* **13**(2), 31–39 (2011)
- Blackledge, M.D., Collins, D.J., Koh, D.-M., Leach, M.O.: Rapid development of image analysis research tools: Bridging the gap between researcher and clinician with pyosirix. *Computers in biology and medicine* **69**, 203–212 (2016)
- Bouttes, D., Lambert, O., Claireaux, C., Woelfel, W., Dalmás, D., Gouillart, E., Lhuissier, P., Salvo, L., Boller, E., Vandembroucq, D.: Hydrodynamic coarsening in phase-separated silicate melts. *Acta Materialia* **92**, 233–242 (2015)
- Brookhaven National Lab: NSLS-II. <http://nsls-ii.github.io/> (2016)
- Brun, F., Mancini, L., Kasae, P., Favretto, S., Drossi, D., Tromba, G.: Pore3d: A software library for quantitative analysis of porous media. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* **615**(3), 326–332 (2010)
- Buades, A., Coll, B., Morel, J.-M.: A non-local algorithm for image denoising. In: *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference On*, vol. 2, pp. 60–65 (2005). IEEE
- Carpenter, A.E., Jones, T.R., Lamprecht, M.R., Clarke, C., Kang, I.H., Friman, O., Guertin, D.A., Chang, J.H., Lindquist, R.A., Moffat, J., et al.: Cellprofiler: image analysis software for identifying and quantifying cell phenotypes. *Genome biology* **7**(10), 100 (2006)
- Chambolle, A.: An algorithm for total variation minimization and applications. *Journal of Mathematical imaging and vision* **20**(1–2), 89–97 (2004)
- ChumpusRex, Wikipedia: Typical screen layout of workstation software used for reviewing multi-detector CT studies. <https://en.wikipedia.org/wiki/File:Ct-workstation-neck.jpg> (2016)
- Coelho, L.P.: Mahotas: Open source software for scriptable computer vision. *Journal of Open Research Software* **1**(1) (2013)
- Coutinho, T.: PyTango. [http://www.esrf.eu/computing/cs/tango/tango\\_doc/kernel\\_doc/pytango/latest/index.html](http://www.esrf.eu/computing/cs/tango/tango_doc/kernel_doc/pytango/latest/index.html) (2016)
- Darbon, J., Cunha, A., Chan, T.F., Osher, S., Jensen, G.J.: Fast nonlocal filtering applied to electron cryomicroscopy. In: *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, pp. 1331–1334 (2008). IEEE
- De Nolf, W., Vanmeert, F., Janssens, K.: Xrdua: crystalline phase distribution maps by two-dimensional scanning and tomographic (micro) x-ray powder diffraction. *Journal of applied crystallography* **47**(3), 1107–1117 (2014)
- Felzenszwalb, P.F., Huttenlocher, D.P.: Efficient graph-based image segmentation. *International Journal of Computer Vision* **59**(2), 167–181 (2004)
- Fraunhofer Institute for Industrial Mathematics ITWM: MAVI. <http://www.itwm.fraunhofer.de/en/departments/image-processing/microstructure-analysis/mavi.html> (2016)
- Getreuer, P.: Rudin-osher-fatemi total variation denoising using split bregman. *Image Processing On Line* **2**, 74–95 (2012)
- Gouillart, E., Toplis, M.J., Grynberg, J., Chopinet, M.-H., Sondergard, E., Salvo, L., Suéry, M., Di Michiel, M., Varoquaux, G.: In situ synchrotron microtomography reveals multiple reaction pathways during soda-lime glass synthesis. *Journal of the American Ceramic Society* **95**(5), 1504–1507 (2012)
- Grady, L.: Random walks for image segmentation. *IEEE transactions on pattern analysis and machine intelligence* **28**(11), 1768–83 (2006). doi:10.1109/TPAMI.2006.233
- Gürsoy, D., De Carlo, F., Xiao, X., Jacobsen, C.: Tomopy: a framework for the analysis of synchrotron tomographic data. *Journal of synchrotron radiation* **21**(5), 1188–1193 (2014)
- Hunter, J.D., et al.: Matplotlib: A 2d graphics environment. *Computing in science and engineering* **9**(3), 90–95 (2007)
- Kass, M., Witkin, A., Terzopoulos, D.: Snakes: Active contour models. *International journal of computer vision* **1**(4), 321–331 (1988)
- Kluyver, T., Ragan-Kelley, B., Pérez, F., Granger, B., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J., Grout, J., Corlay, S., et al.: Jupyter notebooks—a publishing format for reproducible computational workflows. *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, 87 (2016)
- Knudsen, E.B., Sørensen, H.O., Wright, J.P., Goret, G., Kieffer, J.: Fabio: easy access to two-dimensional x-ray detector images in python. *Journal of Applied Crystallography* **46**(2), 537–539 (2013)
- Lamprecht, M.R., Sabatini, D.M., Carpenter, A.E., et al.: Cellprofiler™: free, versatile software for automated biological image analysis. *Biotechniques* **42**(1), 71 (2007)
- Maire, E., Withers, P.: Quantitative x-ray tomography. *International materials reviews* **59**(1), 1–43 (2014)
- Malan, D.F., van der Walt, S.J., Raidou, R.G., van den Berg, B., Stoel, B.C., Botha, C.P., Nelissen, R.G., Valstar, E.R.: A fluoroscopy-based planning and guidance software tool for minimally invasive hip resection by cement injection. *International journal of computer assisted radiology and surgery* **11**(2), 281–296 (2016)
- Marone, F., Münch, B., Stampanoni, M.: Fast reconstruction algorithm dealing with tomography artifacts. In: *SPIE Optical Engineering+ Applications*, pp. 780410–780410 (2010). International Society for Optics and Photonics
- Mirone, A., Brun, E., Gouillart, E., Tafforeau, P., Kieffer, J.: The pyhst2 hybrid distributed code for high speed tomographic reconstruction with iterative reconstruction and a priori knowledge capabilities. *Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms* **324**, 41–48 (2014)
- Nunez-Iglesias, J., Beaumont, C., Robitaille, T.P.: Counting programming language mentions in astronomy papers (2016). doi:10.5281/zenodo.162794. <https://github.com/jni/programming-languages-in-astronomy/blob/1.1/programming-languages-in-ADS.ipynb>
- Oliphant, T.E.: Python for scientific computing. *Computing in Science & Engineering* **9**(3), 10–20 (2007)
- Pawlik, A., Segal, J., Sharp, H., Petre, M.: Crowdsourcing scientific software documentation: a case study of the numpy documentation project. *Computing in Science & Engineering* **17**(1), 28–36 (2015)
- Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al.: Scikit-learn: Machine learning in python. *Journal of Machine Learning Research* **12**(Oct), 2825–2830 (2011)
- Pérez, F., Granger, B.E.: Ipython: a system for interactive scientific computing. *Computing in Science & Engineering* **9**(3), 21–29 (2007)
- Perez, F., Granger, B.E., Hunter, J.D.: Python: an ecosystem for scientific computing. *Computing in Science & Engineering* **13**(2), 13–21 (2011)
- Prescher, C., Prakash, V.B.: Dioptas: a program for reduction of two-dimensional x-ray diffraction data and data exploration. *High Pressure Research* **35**(3), 223–230 (2015)
- Pullii, K., Baksheev, A., Korniyakov, K., Eruhmov, V.: Real-time computer vision with opencv. *Communications of the ACM* **55**(6), 61–69 (2012)
- Rack, A., Scheel, M., Hardy, L., Curfs, C., Bonnin, A., Reichert, H.: Exploiting coherence for real-time studies by single-bunch imaging. *Journal of synchrotron radiation* **21**(4), 815–818 (2014)
- Ramachandran, P., Varoquaux, G.: Mayavi: 3d visualization of scientific data. *Computing in Science & Engineering* **13**(2), 40–51 (2011)
- Rocklin, M. and dask contributors: dask. <http://dask.pydata.org/en/latest/> (2016)
- Rossant, C.: Learning IPython for Interactive Computing and Data Visualization. Packt Publishing Ltd, ??? (2015)
- Schindelin, J., Arganda-Carreras, I., Frise, E., Kaynig, V., Longair, M.,

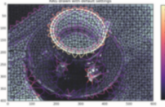
- Pietzsch, T., Preibisch, S., Rueden, C., Saalfeld, S., Schmid, B., *et al.*: Fiji: an open-source platform for biological-image analysis. *Nature methods* **9**(7), 676–682 (2012)
- Schindelin, J., Rueden, C.T., Hiner, M.C., Eliceiri, K.W.: The imagej ecosystem: An open platform for biomedical image analysis. *Molecular reproduction and development* **82**(7-8), 518–529 (2015)
- Schlüter, S., Sheppard, A., Brown, K., Wildenschild, D.: Image processing of multiphase images obtained via x-ray microtomography: a review. *Water Resources Research* **50**(4), 3615–3639 (2014)
- Schneider, C.A., Rasband, W.S., Eliceiri, K.W., *et al.*: Nih image to imagej: 25 years of image analysis. *Nat methods* **9**(7), 671–675 (2012)
- Shen, W., Zhou, M., Yang, F., Yang, C., Tian, J.: Multi-scale convolutional neural networks for lung nodule classification. In: *International Conference on Information Processing in Medical Imaging*, pp. 588–599 (2015). Springer
- Solé, V., Papillon, E., Cotte, M., Walter, P., Susini, J.: A multiplatform code for the analysis of energy-dispersive x-ray fluorescence spectra. *Spectrochimica Acta Part B: Atomic Spectroscopy* **62**(1), 63–68 (2007)
- Sugandhi, R., Swamy, R., Khirwadkar, S.: Use of epics and python technology for the development of a computational toolkit for high heat flux testing of plasma facing components. *Fusion Engineering and Design* (2016)
- V. Armando Sole: PyMca. <https://github.com/vasole/pymca> (2016)
- Van Der Walt, S., Colbert, S.C., Varoquaux, G.: The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering* **13**(2), 22–30 (2011)
- Van der Walt, S., Schönberger, J.L., Nunez-Iglesias, J., Boulogne, F., Warner, J.D., Yager, N., Gouillart, E., Yu, T.: scikit-image: image processing in python. *PeerJ* **2**, 453 (2014)
- Varoquaux, G., Gouillart, E., Vahtras, O., scipy-lecture-notes contributors: Scipy Lecture Notes. <http://www.scipy-lectures.org/> Accessed: 2016-08-10 (2016). <http://www.scipy-lectures.org/>

## Figures

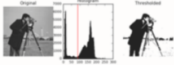
### List of Figures

- 1 **scikit-image and the Scientific Python ecosystem.** Images are opened from files as NumPy arrays. Functions of scikit-image transform image arrays into other arrays with the same dimensions, or into arrays of numbers corresponding to features of the image. The output of scikit-image functions can be passed to other Python modules relying on NumPy arrays, such as SciPy or scikit-learn. Image-shaped arrays are transformed into visualizations with matplotlib (2D) or Mayavi (3D). A variety of environments is available for code development and execution, from classical IDEs to Jupyter notebooks. . . . . 3
- 4 **Simple 3-D visualization** realized with Mayavi. . . . . 4
- 2 **The Spyder IDE** integrates a text editor (with syntax highlighting), the IPython interpreter, as well as a panel for code introspection (online help, variable explorer, . . .). . . . . 5
- 3 **The Jupyter notebook** allows mixing of computer code (top), plot and text output (middle), and free-form narrative text (bottom). This makes it ideal to record and report code-based analyses. Screenshot from Nunez-Iglesias et al. (2016). . . . 6
- 5 **Typical image processing operations with scikit-image.** Data are synthetic, unless stated otherwise. **a) Filtering** - Top: non-local means denoising of an image with a fine-grained texture, acquired by *in situ* synchrotron microtomography during glass melting (Gouillart et al., 2012). Bottom: total-variation denoising of an image with two phases, corresponding to phase-separating silicate melts observed by *in situ tomography* (Bouttes et al., 2015). **b) Feature extraction** - Top: Hubble deep field (NASA, public domain), blob detection using the Laplacian of Gaussian method. Bottom: ridge detection using the leading eigenvalue of the Hessian matrix, neuron image from CREMI challenge (<https://cremi.org/data/>). **c) Segmentation** - Top: super-pixel segmentation of a CT slice of the human head (ChumpusRex, Wikipedia, 2016), using Felzenszwalb's algorithm (Felzenszwalb and Huttenlocher, 2004). Bottom: random walker segmentation (right) of noisy image (top-left corner), using histogram-determined markers (bottom-left corner). **d) Measures** - Top: visualization of local diameter (color-coded on the skeleton curve) of an interconnected phase (represented in violet). Bottom: particles color-coded according to their extent. . . . . 7
- 6 **Gallery of examples of scikit-image.** The gallery of examples consists of an array of thumbnails (left), which link to example webpages, each centered on a specific image processing task. Each webpage includes Python code generating a figure, the figure itself, and a short tutorial explaining the image processing operations and the code. . . . . 13

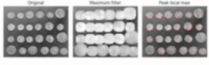
### Gallery of thumbnails



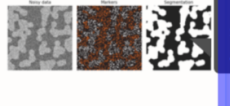
Drawing Region  
Adjacency Graphs (RAGs)



Thresholding




Finding local maxima



[Random walker segmentation](#)

The random walker algorithm [1] determines the segmentation of an image from a set of markers...



Label image regions

Measure region properties

### Code, figure and mini-tutorial

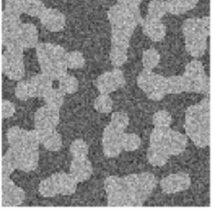
#### Random walker segmentation

The random walker algorithm [1] 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.

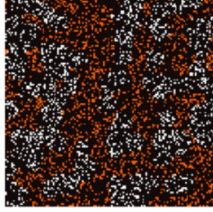
In this example, two phases are clearly visible, but the data are too noisy to perform the segmentation from the histogram only. We determine markers of the two phases from the extreme tails of the histogram of gray values, and use the random walker for the segmentation.

[1] *Random walks for image segmentation*, Leo Grady, IEEE Trans. Pattern Anal. Mach. Intell. 2006 Nov; 28(11):1768-83


Noisy data



Markers



Segmentation



```

import numpy as np
import matplotlib.pyplot as plt

from skimage.segmentation import random_walker
from skimage.data import binary_blobs
import skimage

# Generate noisy synthetic data
data = skimage.img_as_float(binary_blobs(length=128, seed=1))
data += 0.35 * np.random.randn(*data.shape)
markers = np.zeros(data.shape, dtype=np.uint)
markers[data < -0.3] = 1
markers[data > 1.3] = 2

# Run random walker algorithm
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', interpolation='nearest')
ax1.axis('off')
ax1.set_adjustable('box-forced')
ax1.set_title('Noisy data')
ax2.imshow(markers, cmap='hot', interpolation='nearest')
ax2.axis('off')
ax2.set_adjustable('box-forced')
ax2.set_title('Markers')
ax3.imshow(labels, cmap='gray', interpolation='nearest')
ax3.axis('off')
ax3.set_adjustable('box-forced')
ax3.set_title('Segmentation')

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

**Figure 6** Gallery of examples of scikit-image. The gallery of examples consists of an array of thumbnails (left), which link to example webpages, each centered on a specific image processing task. Each webpage includes Python code generating a figure, the figure itself, and a short tutorial explaining the image processing operations and the code.