pi2 Documentation

Arttu Miettinen

May 04, 2023

Contents

This is documentation for pi2/itl2, an image analysis program/library designed for processing and analysis of terapixel-scale volume images *locally or on a computer cluster*. Some typical uses of this program are, e.g., *tracing blood vessels in tomographic images*, *analysis of fibre orientation in composite materials*, or *stitching of volumetric mosaic images*.

The pi2 environment can be used from e.g. *Python*, *C/C++*, and *.NET* programs, and as a *standalone executable* (like ImageMagick).

Some of the capabilities of pi2 include

- *Saturating image arithmetic*
- *Fast non-rigid stitching*
- *Filtering* for noise reduction: *bilateral*, *approximate bilateral*, *variance weighted mean*, *median*, *Gaussian*, *high-pass*, ...
- Post-processing of segmented images: *small region removal* (volume opening), *minimum* and *maximum* filtering (optionally with fast approximate spherical structuring element), *opening* and *closing*, ...
- *Skeletonization* (*line* and *plate+line* skeletons), skeleton *tracing* with measurements (branch length, cross-sectional area...).
- *Detection of interfaces* using an Edwards-Wilkinson surface model
- *Orientation analysis* using the structure tensor approach
- Fast separable *distance map* and *local thickness map*
- *Watershed segmentation*, *region growing*, estimation of surface curvature by *quadric fitting* or using an *implicit form*.
- *Particle/blob/connected component analysis* and *visualization*.
- *Generation* of 3D shapes like spheres, boxes, ...
- *I/O* with `.raw`, `.tif`, `.vol`, and `.nrrd` files; `.tif` and `.png` image sequences; *memory-mapped* `.raw` files.
- Supports (at least) unsigned and signed 8-, 16-, 32-, and 64-bit integer and 32-bit floating point pixels.
- Most operations can be performed on arbitrarily large images either locally or on a *computer cluster* with lazy evaluation.

CHAPTER 1

Installation

Versions for Windows and some Linuxes can be downloaded from the GitHub Releases page.

Windows users must have the Visual Studio Redistributable package installed. Other installation steps are not necessary, just unpack the .zip archive to a suitable folder.

Linux users must ensure that fftw3, libpng, libtiff, zlib, and a recent version of glibc are installed.

For other platforms, refer to prerequisites and build instructions at GitHub.

CHAPTER 2

Quick start examples

Python script or iPython console:

```
from pi2py2 import *
pi = Pi2()
img = pi.newimage(ImageDataType.UINT8, 100, 100, 100)
pi.noise(img, 100, 25)
pi.writetif(img, './noise')
```

Linux shell with pi2 as a standalone program:

```
./pi2 "newimage(img, uint8, 100, 100, 100); noise(img, 100, 25); writetif(img, ./
↪noise);"
```

Windows Command Prompt with pi2 as a standalone program:

```
pi2 newimage(img, uint8, 100, 100, 100); noise(img, 100, 25); writetif(img, ./noise);
```

In a C# program:

```
Pi2 pi = new Pi2();
Pi2Image img = pi.NewImage(ImageDataType.UInt8, 100, 100, 100);
pi.Noise(img, 100, 25);
pi.WriteTif(img, "./noise");
```

For more complicated examples, please refer to the *Examples* page.

CHAPTER 3

Further reading

3.1 Using pi2 from Python

Pi2 can be used from Python scripts and from iPython console. In order to initialize Pi2, the package pi2py must be imported and a Pi2 object created:

```
import sys
sys.path.append("path/to/pi2/folder")

from pi2py2 import *
pi = Pi2()
```

Note The `'path/to/pi2/folder'` should point to the folder of the compiled binary distribution of pi2, not to the source code repository. Typically, the correct folder will be `pi2/bin-linux64/release-nocl` or something similar, where the first folder is the root of the source code repository.

Note Windows users should beware that Python uses the standard directory separator backslash -character as escape character. This means you need to specify the path either using forward slashes "/" (`"path/to/folder"`), double -characters (`"path\\to\\folder"`), or a raw string (`r"path\to\folder"`).

If the pi2py2.py file is in the current working directory, then the `sys.path.append` command does not need to be run. Pi2py is written for Python versions >= 3.6.

After initialization, all the functionality of Pi2 can be accessed through the Pi2 object:

```
img = pi.newimage(ImageDataType.UINT8, 100, 100, 100)
pi.noise(img, 100, 25)
pi.writetif(img, './noise')
```

For reference on the available functionality, use either iPython help or the *Command reference* page. Note that there are some small differences between arguments specified in the *Command reference* and what pi2py2 expects. The largest difference is that instead of image name, you can pass object returned by the newimage command.

There are many Python examples available here: https://github.com/arttumiettinen/pi2/blob/master/python_scripts/pi2py2_examples.py Most of them are documented in the *Examples* page.

The Python bindings in the pi2py2.py file are (mostly) self-generating so they are always (mostly) up-to-date.

3.1.1 Pi2 and NumPy

NumPy arrays can be used together with Pi2 in a few ways.

1. NumPy arrays can be passed to Pi2 functions as input images. Outputting to NumPy arrays is not supported at the moment. Additionally, 3-element NumPy arrays can be used in all occasions where a 3-element vector is required.

2. Image data can be retrieved as a Numpy array using `Pi2Image.to_numpy()` function. The function returns the data in the image as NumPy array of appropriate shape and data type. The NumPy array is a copy of the original image data so changes made to it are not reflected in the Pi2 system.

Retrieving image data as a NumPy array causes it to be read into RAM in its entirety. This might not be desirable in distributed computing mode, if the image is large.

3. NumPy array can be copied into a Pi2 image using `Pi2Image.from_numpy(array)` function. The data of the array is copied from NumPy into the Pi2 system, so changes made by Pi2 are not reflected in the NumPy array.

For 'power users' there is also a method `Pi2Image.to_numpy_pointer()` that returns reference to the image data in the Pi2 system as a NumPy array. Changes made to the array are reflected in the Pi2 system and vice versa. However, please note that Pi2 commands that change the size of the image will re-allocate the memory reserved for the image and this process invalidates any NumPy arrays returned by the `to_numpy_pointer()` method. Accessing invalidated array might result in program crash. Arrays returned by the `to_numpy()` method are not affected by this problem.

If Pi2 is in distributed computing mode, calling `Pi2Image.to_numpy_pointer()` will cause the image to be read into RAM in its entirety. In this mode, changes made to the NumPy array are not reflected to the image until `Pi2Image.flush_pointer()` method is called.

3.2 Using pi2 from shell or command line

Pi2 can be used as a standalone program from the shell (Linux) or command prompt (Windows). There are two ways to achieve this.

3.2.1 With a Pi2 script file

Create a text file that contains Pi2 commands that you want to run, e.g:

```
newimage(img, uint8, 100, 100, 100);
noise(img, 100, 25);
writetif(img, ./noise);
```

Save the file and then run:

```
./pi2 file_name
```

or in Windows:

```
pi2 file_name
```

where `file_name` is the name of the script file. If no file name is given, the program tries to open `pi.txt` or `pi2.txt` in the current directory and use that as an input file.

3.2.2 Without intermediate script file

The Pi2 commands can be given directly as command-line arguments to the pi2 executable. For example, in Linux:

```
./pi2 "newimage(img, uint8, 100, 100, 100); noise(img, 100, 25); writetif(img, ./
↪noise);"
```

or in Windows:

```
pi2 newimage(img, uint8, 100, 100, 100); noise(img, 100, 25); writetif(img, ./noise);
```

3.2.3 Pi2 command syntax

The syntax of the Pi2 script files (and command-lines) is pretty simple and forgiving. There are only a few rules:

- Each command is in the form `name(arg1, arg2, ..., argN)`. There are no, e.g. arithmetic operations or conditional statements. List of allowed commands is found in the *Command reference* page.
- Commands can be separated by `;`-character or newline. Newlines can be in either Linux or Windows format.
- Whitespace is not significant. Values do **not** need to be quoted unless they contain significant whitespace or commas, and in that case they may be quoted either with single quotes `'` or double quotes `"`.
- Anything after `%`, `#`, or `//` is comment until the next newline.
- Vector values can be expressed with `[x, y, z]`, where `x`, `y`, and `z` are the three components of the vector. If only one component `x` is specified, the resulting vector will be `[x, x, x]`.
- Image names must be alphanumeric and start with a letter.
- Argument that has a default value does not need to be given, unless arguments after that are given.

3.3 Using pi2 from .NET programs

Pi2 can be used from .NET languages like C# and Visual Basic .NET. In short, you will need to add a reference to pi2cs library and create an instance of the Pi2 class. The methods of the class can then be used to access Pi2 functionality.

In practice, all this is done using code like this:

```
Pi2 pi = new Pi2();
Pi2Image img = pi.NewImage(ImageDataType.UInt8, 100, 100, 100);
pi.Noise(img, 100, 25);
pi.WriteTif(img, "./noise");
```

The methods of the Pi2 class are documented in the standard XML documentation and they mostly correspond to what is found in *Command reference*.

Depending on the agenda of the authors, the functionality in the Pi2 class might lag behind what is available in the Python interface. However, all functionality is available through a low-level wrapper class PiLib. In particular, it exposes method `RunAndCheck` that can be used to run arbitrary Pi2 commands (see *Command reference*), and `GetImage` that returns pointer to image data stored in the Pi2 system.

3.3.1 Viewing Pi2 images

The pi2cs library contains controls Pi2PictureViewer, Pi2PictureToolStrip and Pi2PictureBox that can be used to show Pi2 images on Windows Forms. Of these three, Pi2PictureBox is a simple picture box mostly corresponding to the standard Windows Forms PictureBox, but it shows image from the Pi2 system. Pi2PictureViewer and Pi2PictureToolStrip provide more advanced user interface with functionality such as zooming and annotations. At the time of writing, work with these controls is still very much in progress.

3.4 Using pi2 from C++ code and other programming languages

3.4.1 Using full pi2 functionality

Pi2 functionality including distributed processing can be used from C, C++, and many other programming languages by using the pilib shared library/dynamic library. Examples of the process can be found in:

- C and C++: https://github.com/arttumiettinen/pi2/blob/master/pi2/pi2.cpp and https://github.com/arttumiettinen/pi2/blob/master/pilib/pilib.h
- C#: https://github.com/arttumiettinen/pi2/blob/master/pi2cs/Pi2.cs and https://github.com/arttumiettinen/pi2/blob/master/pi2cs/PiLib.cs
- Python: https://github.com/arttumiettinen/pi2/blob/master/python_scripts/generic/pi2py2.py

Basically, the shared library contains a few functions that you use to interact with the pi2 system.

First, you will need to initialize pi2 by calling

```
void* createPI();
```

that returns a handle that must be passed to all other function calls.

Pi2 commands can be run using

```
uint8_t run(void* pi, const char* commands);
```

The arguments are the handle returned by `createPI()` and arbitrary pi2 command string in the same format that is used in the *stand-alone mode*.

If the return value of `run` is zero, the execution of the commands failed. In this case the error message can be accessed using functions:

```
const char* lastErrorMessage(void* pi);
int32_t lastErrorLine(void* pi);
```

These return pointer to error message and line number causing the error, respectively. The error can be cleared with

```
void clearLastError(void* pi);
```

but it is not strictly necessary as new errors will overwrite old ones anyway.

Image data stored in the pi2 system can be accessed with

```
void* getImage(void* pi, const char* imgName, int64_t* width, int64_t* height, int64_
↪t* depth, int32_t* dataType);
```

that returns a pointer to image data, given handle returned by `createPI` and the name of the image. The width, height, depth, and dataType parameters are filled by the function to image dimensions and data type.

After you are done, the pi2 system should be torn down by passing the handle to

```
void destroyPI(void* pi).
```

More information and API documentation can be found in source files

- C/C++ https://github.com/arttumiettinen/pi2/blob/master/pilib/pilib.h.
- C#: https://github.com/arttumiettinen/pi2/blob/master/pi2cs/PiLib.cs

3.4.2 Using non-distributed functionality only as a C++ template library

For C++ developers pi2 offers also a (perhaps) simpler way to access the non-distributed processing functionality. The image processing algorithms are available as a mostly template-based C++ library in the itl2 folder of the source code. The library works in pretty simple imperative mindset:

```
#include "image.h"
#include "noise.h"
#include "io/itltiff.h"

using namespace itl2;

void main()
{
        // Create image whose pixel data type is 8-bit unsigned integer and␣
↪dimensions are 100 x 100 x 100.
        Image<uint8_t> image(100, 100, 100);

        // Add noise to the image
        noise(image, 100, 25);

        // Write to .tiff file. The writed function adds .tif in the end of file name␣
↪if the suffix is not there.
        itl2::tiff::writed(image, "./noise");
}
```

In some cases you need to link with libitl2/itl2.lib. Please see header files in itl2 folder and subfolders for API documentation.

Note: Using the library this way has not been thoroughly tested so you might encounter some problems!

3.5 Distributed processing and cluster configuration

One of the main reasons for existence of the pi2 system is its ability to do user-transparent distributed processing. Let us look at a simple example in Python code:

```
# Initialization
from pi2py2 import *
pi = Pi2()

# Read two images
img1 = pi.read('./first_image')
img2 = pi.read('./second_image')

# Add them together
```

(continues on next page)

(continued from previous page)

```
pi.add(img1, img2)

# Save
pi.writeraw(img1, './added_image')
```

This code initializes the pi2 system, reads two images from disk (in any supported format), adds `img2` to `img1` and finally saves the result to disk.

If the images do not fit into the RAM of the computer, we have a problem. The input images must be read in smaller pieces, operations performed on the pieces, and the result must be written into the output file at the correct location. Doing all this is a bit complicated, particularly if the processing is something more complicated than simple addition. Luckily, pi2 is able to do all this with only minimal changes to the code:

```
# Initialization
from pi2py2 import *
pi = Pi2()

# Enable distributed processing
pi.distribute(Distributor.LOCAL)

# Read two images
img1 = pi.read('./first_image')
img2 = pi.read('./second_image')

# Add them together
pi.add(img1, img2)

# Save
pi.writeraw(img1, './added_image')
```

We only need to add the line `pi.distribute(...)` in order to enable the distributed computing mode. The argument specifies what kind of distribution strategy to use. Currently, supported strategies are `Distributor.LOCAL` and `Distributor.SLURM`.

The local mode divides the input and output images into smaller pieces and processes the pieces one-by-one on your local computer. This mode can be used to process datasets that do not fit into the RAM of your computer.

The Slurm mode assumes that you are running on a computer cluster using the Slurm Workload Manager. It divides the datasets similarly to the local mode, but submits processing of each piece to the cluster as a Slurm job. This way all the cluster nodes available to you can be benefited from, and processing of very large (even terabyte-scale) images is pretty fast.

The distributed operating mode is supported by most of the pi2 commands. See the *Command reference* for details.

To fully benefit from the distributed mode, some settings must be tuned as detailed in the following sections.

3.5.1 Configuration for local distribution mode

Configuration for local distribution mode are found in file local_config.txt found in the same folder than the pi2 executable. In the configuration file, lines starting with `;`-character are comments. In the default configuration file the comments are used to describe the different settings. The most important setting is `max_memory` that gives the approximate amount of memory (in megabytes) that the pi2 system may use at once. If set to zero, pi2 uses 85 % of total RAM in the computer. For descriptions of the other settings, please refer to the comments in the default configuration file.

For quick testing, the `maxmemory` parameter can also be set using the *maxmemory* command, but changes made with the command are not saved into the configuration files.

3.5.2 Configuration for SLURM cluster

Slurm cluster configuration is located in file slurm_config.txt found in the same folder than the pi2 executable. The file is similar to the local_config.txt (see above), but it has more parameters available.

The pi2 system divides the jobs into three categories: fast, normal, and slow. The `extra_args_*` settings can be used to submit the jobs into different partitions. This is useful as partitions for fast jobs usually have shorter queueing time.

The `max_memory` setting (and `maxmemory` command) works the same than in local processing, but now it should be set to the amount of usable RAM in a compute node. If set to zero, the system finds out the node with the smallest amount of memory using `sinfo --Node --Format=memory`, and sets `max_memory` to 90 % of that. In very large clusters the `sinfo` command may be slow, so you might want to set the `max_memory` parameter manually.

If you encounter out-of-memory errors or jobs crash without reason, try decreasing the `max_memory` parameter.

When a job fails, the pi2 system will try to re-submit it a few times. The maximum number of re-submissions is given by the `max_resubmit_count` parameter.

For more thorough descriptions of the parameters please refer to comments in the default configuration file.

3.6 Non-rigid stitching & the NRStitcher program

NRStitcher is a Python program for non-rigid stitching of terabyte-scale volume images. Algorithms that it implements are discussed in *Arttu Miettinen, Ioannis Vogiatzis Oikonomidis, Anne Bonnin, Marco Stampanoni. NRStitcher: non-rigid stitching of terapixel-scale volumetric images, btz423, Bioinformatics, 2019*.

Basically, NRStitcher is used to construct a large (volume) image from overlapping tiles. The program is able to account for small deformations in the tiles. NRStitcher uses pi2 system for image processing and runs on Slurm cluster or locally.

3.6.1 Acquisition of the tile images

In order to generate input data for the NRStitcher program, one typically uses some kind of a microscope to acquire a number of overlapping images. Large-ish overlaps are easier to stitch with the NRStitcher. For microtomographic images overlaps of 10 % - 30 % are typically used: the well-reconstructed region in the images is cylindrical, so approximately 30 % overlap between neighbouring cylinders is required in the radial direction. In the axial direction 10 % overlap seems to be fine in most cases.

The images must be saved in a format supported by the pi2 system, e.g. `.raw`, `.tif`, `.nrrd`, or sequence of `.tif` or `.png` slices.

The order of acquisition can be anything (snake, row-by-row, random, etc.) as long as approximate locations of the tiles are known.

3.6.2 Masking

Sometimes the tile images contain bad areas that must not be used in the stitching process. In order to facilitate this, pixel value 0 is handled specially in the stitching process. It is used to denote pixels with "no usable data" and can therefore be used to mask some regions of the tile image out of the final result. For example, in the case of tomographic images, pixel value 0 could be used to get rid of the badly reconstructed region near the edges of the image. In many cases this seems to be unnecessary, though.

In the case your valid pixel data contains zeroes, consider mapping the pixel values e.g. to range [1, max], where max denotes the maximal representable number in the pixel data type. After stitching, you may re-map back to the original range. The *linmap* command can be used for this purpose.

3.6.3 Stitching

In order to stitch the tiles, one must first create a new folder that will contain a stitch settings file, temporary files, and the final stitched image.

The stitch settings file must be generated according to the template. The name of the settings file must be `stitch_settings.txt` and it must be saved into the output folder. Most of the settings can be initially left to their default values, but the `[positions]` section must be modified so that it contains paths and file names of the tiles and their positions. The section contains one line for each tile. The line is formatted like `image_name = x, y, z`, where `image_name` gives the path and file name of the tile, and `x, y, z` is the location of that tile in pixels.

For example, positions of a four-tile image could be defined like this:

```
[positions]
wc1_150x150x1.raw = 0, 80, 0
wc2_150x150x1.raw = 110, 60, 0
wc3_150x150x1.raw = 220, 40, 0
wc4_150x150x1.raw = 330, 20, 0
```

If the stitching is to be run on a computer cluster, the cluster settings in the configuration file must be changed to match the capabilities of the available compute nodes.

If you are working at the SLS TOMCAT beamline, there is a script that converts a set of CT scan log files to a stitch settings file. Please refer to the internal documentation for details.

Often it is useful to try the stitching with downscaled versions of the tiles to see if everything works. This can be conveniently done using the parameter `binning`. Specify value larger than one to downscale the tiles by that amount before stitching. Usually binning 4 is a good starting point.

When the parameter file is ready, the stitching process is started by running the Python file `nr_stitcher.py` in the output folder.

After the script is finished, the largest `.raw` image in the output folder is the stitched image. Please note that if there are multiple non-connected sets of tiles in the output, multiple output images will be created.

Before running `nr_stitcher.py` again with different parameters, please remove all temporary files from the output folder by running `rm-stitch-temp.sh` or `rm-stitch-temp.cmd`, or specify redo=True in the configuration file.

3.6.4 Example

An example of tiles and the corresponding settings file can be found in the examples archive distributed along each pi2 version. See https://github.com/arttumiettinen/pi2/releases.

3.6.5 Known problems

- **Stitching result is not good, there are doubled details visible in the regions where one or more tiles overlap.**
  This is usually caused by too large or too small values of `point_spacing` and `coarse_block_radius` parameters.

- **Error ": No such file or directory" when running in Linux.** The error means that the line endings in the .py files have been converted to Windows format. This might happen for various not-so-obvious reasons, e.g. when transferring the Python code files by email. Easy workaround is to prepend 'python' to the commands, e.g.

```
python /path_to_scripts/nr_stitcher.py
```

  In some cases it is necessary to explicitly specify python3 so that the operating system does not decide to use python 2.x versions that are incompatible:

```
python3 /path_to_scripts/nr_stitcher.py
```

- **The same cluster jobs are being submitted over and over again.** The cluster jobs fail for reason or another, and the NRStitcher is unable to catch the error. Check Slurm output files `*-out.txt` and `*-err.txt` for more information. Sometimes simply re-running the nr_stitcher_nongrid.py fixes the problem.
- **Large blocks of output image are black.** If using a cluster, this problem is usually caused by failed SLURM jobs in the finalization phase. Each black block corresponds to one failed job. The most probable causes for this problem are out of memory error caused by too large `max_block_size` parameter in the stitch settings file, or jobs running too long and being killed by SLURM.

  In order to find out the reason for the error, please check the SLURM output files `*-out.txt` and `*-err.txt`. They will contain an indication of the reason why jobs failed if they failed.

  In any case, you might want to try again with smaller `max_block_size` parameter and larger job run time (using e.g. `cluster_extra_params = --time=48:00:00` in the stitch settings file)

  In some cases the problems are intermittent and you will get away by simply re-running `nr_stitcher.py`. It tries its best not to re-calculate the good regions.

3.7 Examples

This page contains some examples on how to use pi2 for simple (and some not so simple) tasks. All the examples are given in Python 3.6. They can be found in a single Python file here.

The examples require a few small images that can be downloaded from here, or from GitHub.

In order to run the examples, please run this initialization first (make sure that the file paths point to correct locations):

```
import numpy as np
from pi2py2 import *
pi = Pi2()

# Define convenience functions that return input and output file names.
# This is just to avoid copying the paths to all the examples in case they change.
def input_file(filename='t1-head_256x256x129.raw'):
        return '../../test_input_data/' + filename

def input_file_bin():
        return '../../test_input_data/t1-head_bin_256x256x129.raw'

def output_file(name):
        return '../../test_output_data/pi2py2/' + name
```

3.7.1 Binning

This example shows how to use different *binning* modes and what is their effect.

The default binning calculates average of each $(binsize)^3$ block of the input image, and makes that one pixel of the output image. The 'min' and 'max' modes take the minimum and maximum of the block, respectively, and make that value one pixel of the output image.

```
def binning_scaling():
        """
        Demonstrates binning.
        """

        # Read image
        img = pi.read(input_file())

        # Normal binning
        binned_mean = pi.newlike(img)
        pi.bin(img, binned_mean, 4)
        pi.writeraw(binned_mean, output_file('binning_mean'))

        # Min binning
        binned_min = pi.newlike(img)
        pi.bin(img, binned_min, 4, 'min')
        pi.writeraw(binned_min, output_file('binning_min'))

        # Max binning
        binned_max = pi.newlike(img)
        pi.bin(img, binned_max, 4, 'max')
        pi.writeraw(binned_max, output_file('binning_max'))
```

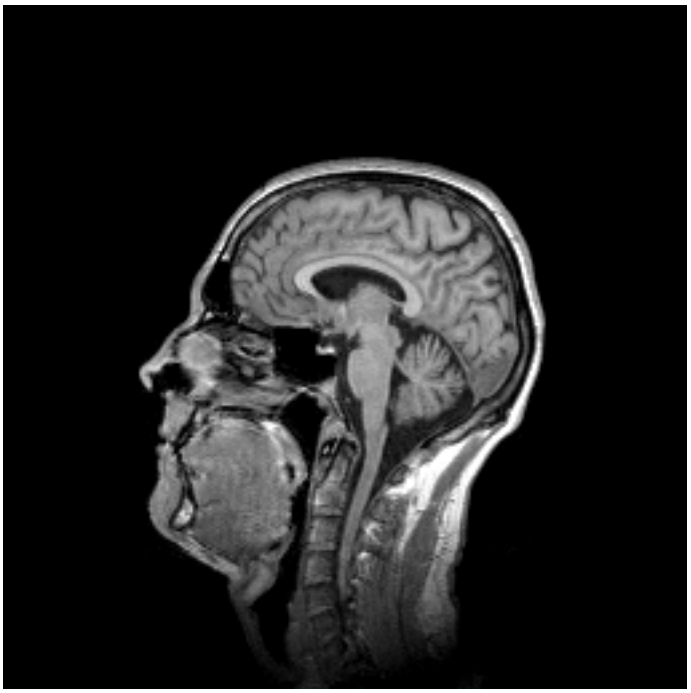

Fig. 1: One slice of the input image.

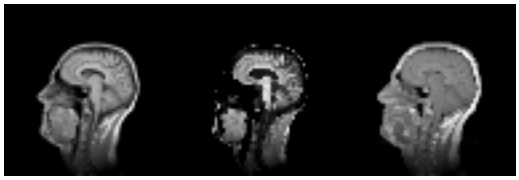

Fig. 2: Images binned to 25 % of original size using 'mean', 'min', and 'max' modes, from left to right.

3.7.2 Bivariate histogram

This example shows how to calculate the bivariate histogram of two images (of the same size). Here we calculate bivariate histogram of an image and its local thickness transform.

```
def bivariate_histogram():
        """
        Demonstrates determination of bivariate histogram.
        """

        # Read original
        img = pi.read(input_file())

        # Create a copy of the image and binarize it
        bin = pi.newlike(img)
        pi.copy(img, bin)
        pi.threshold(bin, 185)

        # Calculate thickness map.
        # Convert input image first to uint32 so that it can hold even large values.
        pi.convert(bin, ImageDataType.UINT32)
        tmap = pi.newlike(bin, ImageDataType.FLOAT32)
        pi.tmap(bin, tmap)

        # Calculate bivariate histogram of original and thickness map
        hist = pi.newimage(ImageDataType.FLOAT32)
        bins1 = pi.newimage(ImageDataType.FLOAT32)
        bins2 = pi.newimage(ImageDataType.FLOAT32)
        gray_max = 500   # Maximum gray value in the histogram bins
        thick_max = 30   # Maximum thickness in the histogram bins
        pi.hist2(img, 0, gray_max, 60, tmap, 0, thick_max, 15, hist, bins1, bins2) #␣
↪60 gray value bins, 15 thickness bins

        # Write output files to disk
        pi.writetif(img, output_file('original'))
        pi.writetif(tmap, output_file('thickness_map'))
        pi.writetif(hist, output_file('histogram'))
        pi.writetif(bins1, output_file('bins1'))
        pi.writetif(bins2, output_file('bins2'))

        # Get histogram data
        h = hist.get_data()

        # Make plot
        import matplotlib.pyplot as plt
        fig = plt.figure(figsize=(4.5, 6))

        # Note that here we plot the transpose of the histogram as that fits better␣
↪into an image.
```

(continues on next page)

(continued from previous page)

```
        # We also flip it upside down so that both x- and y-values increase in the␣
↪usual directions.
        pltimg = plt.imshow(np.flipud(h.transpose()), vmin=0, vmax=1.5e4, extent=(0,␣
↪thick_max, 0, gray_max), aspect=1/8)
        cbar = fig.colorbar(pltimg)
        plt.xlabel('Thickness [pix]')
        plt.ylabel('Gray value [pix]')
        cbar.set_label('Count', rotation=90)
        plt.tight_layout()
        plt.show(block=False)
        plt.savefig(output_file('bivariate_histogram.png'))
```

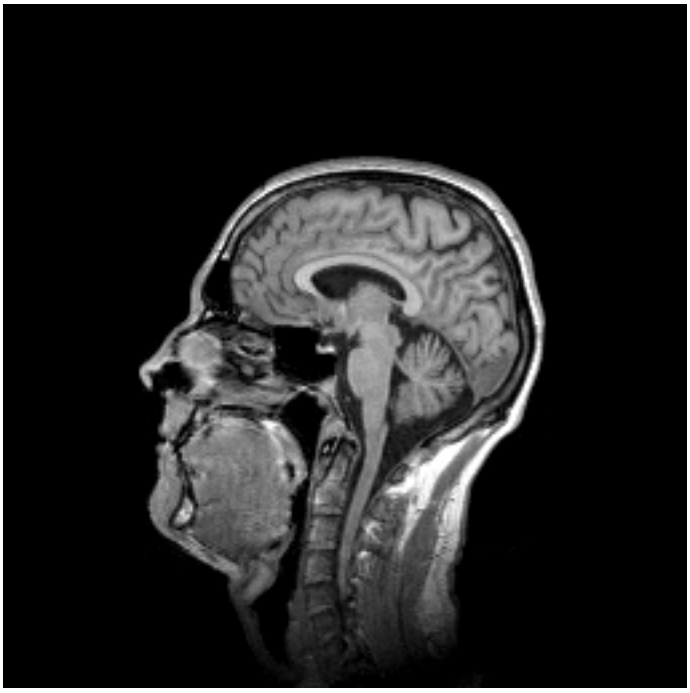

Fig. 3: One slice of the input image.

3.7.3 Byte order, big-endian, little-endian

This example demonstrates some issues with big-endian and little-endian .raw files.

The pi2 system always assumes that .raw files are read in native byte order of the host computer. If this is not the case, the byte order can be changed using *swapbyteorder* command

```
def big_endian_and_little_endian():
        """
        Demonstrates how to change the endianness of pixel values.
        """

        # The images are in the native byte order
        geometry = pi.newimage(ImageDataType.UINT16, 100, 100)
        pi.ramp(geometry, 1)
```

(continues on next page)

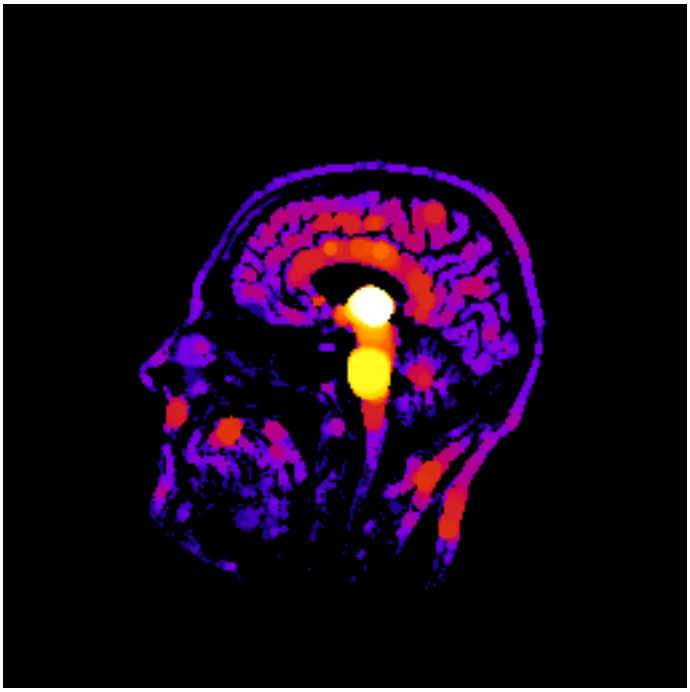

Fig. 4: Local thickness map of the input. Brighter colors correspond to thicker regions.

(continued from previous page)

```
        # By default, images are written in the native byte order.
        # This usually means little endian at the time this was written, and at the
        # processors this code has ever run.
        pi.writeraw(geometry, output_file("little_endian_ramp"))

        # Change to big endian format can be made by running swapbyteorder command
        pi.swapbyteorder(geometry)
        pi.writeraw(geometry, output_file("big_endian_ramp"))

        # NOTE: Here the geometry image is stored in big endian format, and you␣
↪cannot do
        # any calculations on it until you run the swapbyteorder command on it again!
        # For example, writing something else than .raw files for image in non-native␣
↪byte
        # order does not make sense! The .tif file saved below will
        # contain pixel values in wrong byte order.
        pi.writetif(geometry, output_file("big_endian_ramp.tif"))
```

3.7.4 Create and access images

This example shows how to create images, retrieve their dimensions and data type, save them, and access the images as NumPy arrays:

```
def create_and_access_images():
        """
        Demonstrates creation of image and accessing its data.
        """
```

(continues on next page)

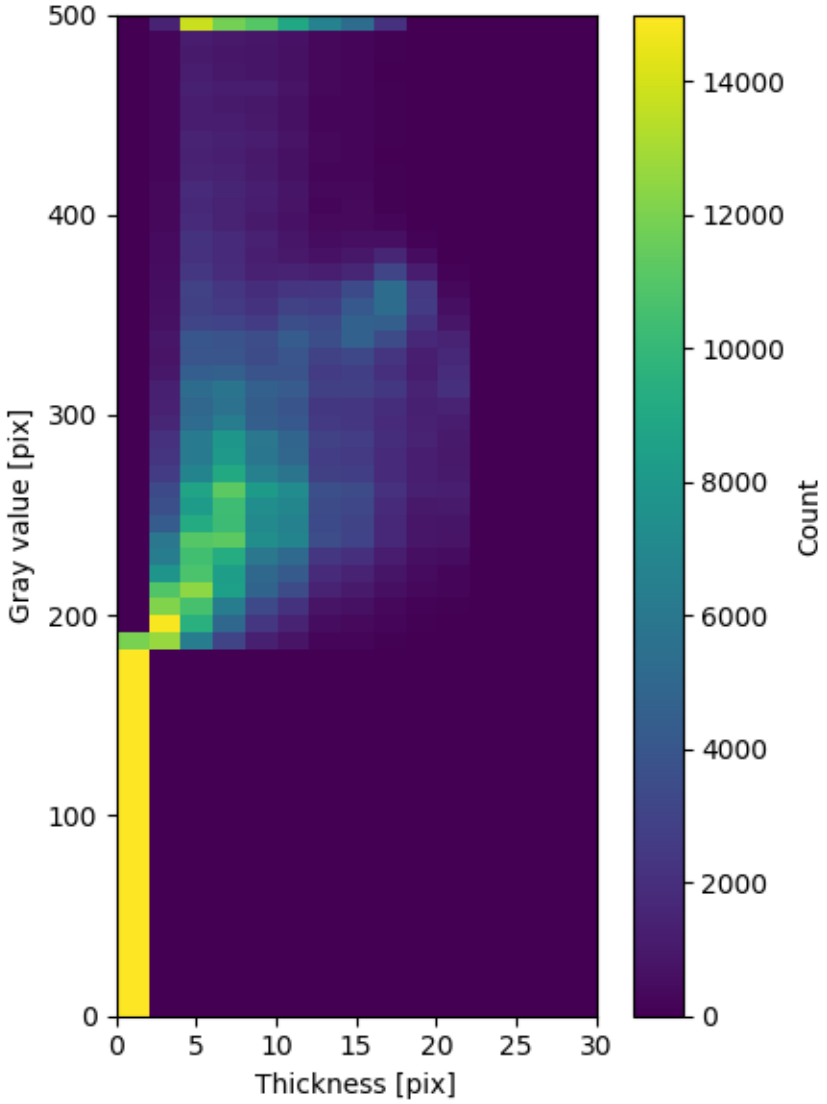

Fig. 5: Bivariate histogram of local thickness and original gray value, showing that larger thicknesses tend to be found in brighter regions. The bright row at gray value 500 is caused by out-of-bounds values (gray value > 500) included in the last bin.

(continued from previous page)

```
        # newimage command is used to create new images
        img1 = pi.newimage(ImageDataType.UINT8, 10, 20, 30)
        print(f"Width = {img1.get_width()}")
        print(f"Height = {img1.get_height()}")
        print(f"Depth = {img1.get_depth()}")
        print(f"Data type = {img1.get_data_type()}")
        print(f"All in one: {img1}")

        # newlike command can be used to create new image that is similar to existing␣
↪image,
        # This line creates second image that has the same dimensions than img1 and␣
↪pixel
        # data type UINT16.
        img2 = pi.newlike(img1, ImageDataType.UINT16)
        print(f"img2 is {img2}")

        # This line creates second image that has the same pixel data type than img1,
        # but its dimensions are 50x50x50.
        img3 = pi.newlike(img1, ImageDataType.UNKNOWN, 50, 50, 50)
        print(f"img3 is {img3}")

        # The data in the image can be retrieved as a NumPy array
        data = img1.get_data()
        print(f"When converted to a NumPy array, the shape of the image is {data.
↪shape} and data type is {data.dtype}.")

        # Image data can also be set from a NumPy array
        data = np.eye(100, 100)
        img1.set_data(data)

        # This writes img1 to disk as a .raw file.
        # The dimensions of the image and the .raw suffix are automatically appended␣
↪to
        # the image name given as second argument.
        pi.writeraw(img1, output_file("img1"))

        # NumPy arrays can be used directly as input in commands.
        # Changes made by Pi2 are NOT reflected in the NumPy arrays as
        # in some cases that would require re-shaping of the arrays, and that
        # does not seem to be wise...
        pi.writetif(data, output_file("numpy_array_tif"))
```

Output shown by the code above:

```
Width = 10
Height = 20
Depth = 30
Data type = uint8
All in one: (10, 20, 30), uint8
img2 is (10, 20, 30), uint16
img3 is (50, 50, 50), uint8
When converted to a NumPy array, the shape of the image is (20, 10, 30) and data type␣
↪is uint8.
```

3.7.5 File format conversions

File format can be changed by reading the file and then writing in desired format. In many cases the conversion works in distributed mode, too:

```
def convert_format():
        """
        Demonstrates file format conversion.
        """

        # Read image
        img = pi.read(input_file('simple_structures.vol'))

        # Write in any supported format
        pi.writeraw(img, output_file('vol_to_raw'))
        pi.writetif(img, output_file('vol_to_tif'))
```

3.7.6 Filtering, normal and distributed mode

This example demonstrates *Gaussian filtering* of an image in normal mode and in *distributed mode*. For demonstration, the *max_memory* setting of the distributed mode is set such that we need two jobs even for the small test image used in the example. The filtering result is calculated also in normal mode, and the results between the normal mode and the distributed mode are compared.

The example generalizes easily to other filtering methods like *vawe*, *bilateral*, *maximum*, *opening*, *closing*, etc.

```
def filtering():
        """
        Calculates Gaussian blur of image normally and using distributed processing.
        Calculates difference of the two versions.
        The example generalizes to any available filtering procedure like
        vawefilter, bilateralfilter, maxfilter, openingfilter, etc.
        """

        # Gaussian filtering (local sequential 'distributed' processing)
        # --------------------------------------------------------------

        # Enable distributed mode
        pi.distribute(Distributor.LOCAL)

        # For demonstration, set memory per one job to low value.
        # 25 megabytes results in 2 jobs for the default input image in this example.
        # Typically you would set this value in local_config.txt file.
        pi.maxmemory(25)

        # Read image
        img = pi.read(input_file())

        # Create output image
        filtered = pi.newlike(img)

        # Filter
        pi.gaussfilter(img, filtered, 5)

        # Write output to disk.
        # The distributed mode saves internal temporary images as .raw files or .png
```

(continues on next page)

(continued from previous page)

```
        # sequences. Writeraw command in distributed mode therefore often converts
        # into a simple file rename.
        pi.writeraw(filtered, output_file('head_gauss_distributed'))

        # Disable distributed mode
        pi.maxmemory(0) # Sets max memory to automatically determined value
        pi.distribute(Distributor.NONE)


        # Gaussian filtering (local processing)
        # -------------------------------------

        # This code is the same than in distributed case above, but without
        # pi.distribute-commands.
        img = pi.read(input_file())
        filtered = pi.newlike(img)
        pi.gaussfilter(img, filtered, 5)
        pi.writeraw(filtered, output_file('head_gauss_normal'))


        # Calculate difference of results of normal and distributed processing
        # --------------------------------------------------------------------

        # Read both images
        img = pi.read(output_file('head_gauss_normal'))
        img2 = pi.read(output_file('head_gauss_distributed'))

        # Convert them to float32 so that negative values can be represented, too.
        pi.convert(img, ImageDataType.FLOAT32)
        pi.convert(img2, ImageDataType.FLOAT32)

        # Subtract img2 from img
        pi.subtract(img, img2)
        pi.writeraw(img, output_file('gauss_difference'))

        # Calculate absolute value of each pixel
        pi.abs(img)

        # Calculate maximal value and place it to image M.
        # M will be a 1x1x1 image.
        M = pi.newimage(ImageDataType.FLOAT32)
        pi.maxval(img, M)

        # Get the value of the first pixel of image M.
        # In this case M is a 1x1x1 image so we have only one pixel anyway.
        M = M.get_value()
        print(f"Maximal difference = {M}")
```

Output:

```
Enabling distributed computing mode using local sequential processing.
Using 13.51 GiB RAM per task.
Using 25 MiB RAM per task.
Submitting 2 jobs, each estimated to require at most 21 MiB of RAM...
-- Individual job progress information is cut away for clarity --
Waiting for jobs to finish...
Using 13.51 GiB RAM per task.
```

(continues on next page)

(continued from previous page)

```
Distributed computing mode is disabled.
Maximal difference = 0.0
```

3.7.7 Finding surface using Edwards-Wilkinson model

This example shows how to find a surface or interface from a grayscale image.

Surface recognition is done using the 'Carpet' algorithm according to Turpeinen - Interface Detection Using a Quenched-Noise Version of the Edwards-Wilkinson Equation. The algorithm places a surface at $z = 0$, and moves it towards larger z values according to the Edwards-Wilkinson equation. The movement of the surface stops when it encounters enough pixels with value above specific stopping gray level. The surface does not move through small holes in the object as it has controllable amount of surface tension. The output of the algorithm is a height map that gives the z-position of the surface for each (x, y)-location. In addition to the height map, a visualization of the evolution of the surface can be made.

After finding the surface, the example demonstrates how to visualize it.

Fig. 6: Visualization of the evolution of an Edwards-Wilkinson surface on top of the original data. The figure shows one xz-slice through the original image. The surface has been drawn with red. The animation shows the surface as a function of iteration (i.e. time).

```
def find_surface():
        """
        Demonstrates surface recognition using Edwards-Wilkinson model.
        """

        # Read image
        orig = pi.read(input_file())

        # Crop smaller piece so that the surface does not 'fall through'
        # as there's nothing in the edges of the image.
        # Usually surfaces are recognized from e.g. images of paper sheet
        # where the sheet spans the whole image.
        img = pi.newimage(orig.get_data_type())
        pi.crop(orig, img, [80, 90, 0], [85, 120, orig.get_depth()])

        pi.writeraw(img, output_file('findsurface_geometry'))

        # This image will hold the height map of the surface
        hmap = pi.newimage(ImageDataType.FLOAT32)

        # ... and this one will be a visualization of surface evolution
        vis = pi.newimage(orig.get_data_type())

        # Now find the surface.
        # We will stop at gray level 100 and perform 60 iterations
        # with surface tension 1.0.
        pi.findsurface(img, hmap, 100, Direction.DOWN, 1.0, 60, 1, vis, img.height() /
↪ 2, 900)

        # Save the result surface
        pi.writeraw(hmap, output_file('findsurface_height_map'))
```

(continues on next page)

(continued from previous page)

```
        # Save the visualization
        pi.writeraw(vis, output_file('findsurface_evolution'))



        # We can also draw the surface to the image
        surf_vis = pi.newimage()
        pi.set(surf_vis, img)
        pi.drawheightmap(surf_vis, hmap, 900)
        pi.writeraw(surf_vis, output_file('findsurface_full_vis'))

        # Or set pixels below or above the surface.
        # This is useful to, e.g., get rid of background noise above
        # or below the surface of the sample.
        before_vis = pi.newimage()
        pi.set(before_vis, img)
        pi.setbeforeheightmap(before_vis, hmap, 900)
        pi.writeraw(before_vis, output_file('findsurface_before_vis'))

        after_vis = pi.newimage()
        pi.set(after_vis, img)
        pi.setafterheightmap(after_vis, hmap, 900)
        pi.writeraw(after_vis, output_file('findsurface_after_vis'))
```

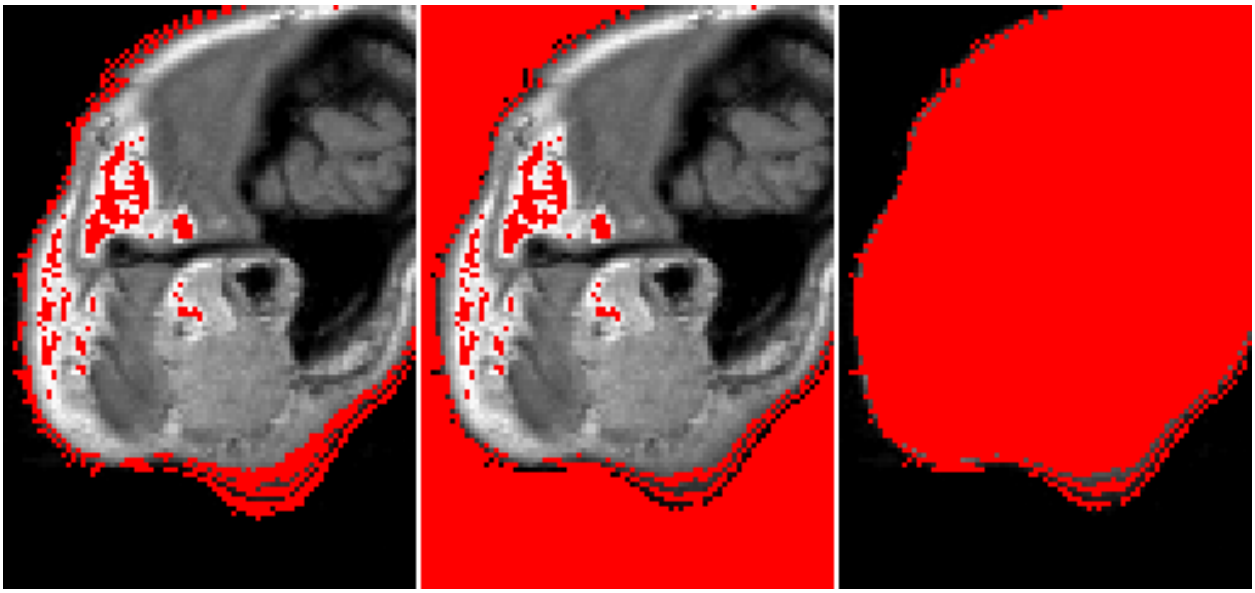

Fig. 7: Visualizations of the found surface (red) at single xy-slice of the input (grayscale). The left panel shows the surface, the middle panel shows all the pixels that are located above the surface and the right panel all the pixels that are below it.

3.7.8 Greedy coloring

This example demonstrates *coloring* of regions with minimal number of colors, while ensuring that neighbouring regions have different colors. See also https://en.wikipedia.org/wiki/Four_color_theorem.

The coloring is made using a greedy algorithm, so the number of colors is not necessarily minimal.

```
def greedy_coloring():
        """
        Shows how to change colors of regions.
        """

        # Create image
        img = pi.newimage(ImageDataType.UINT8, 200, 200, 1)

        # Plot some overlapping spheres.
        # Each of the spheres has different color
        pi.sphere(img, [100, 100, 0], 50, 7)
        pi.sphere(img, [150, 100, 0], 50, 10)
        pi.sphere(img, [50, 100, 0], 50, 15)
        pi.sphere(img, [100, 50, 0], 50, 20)
        pi.sphere(img, [150, 150, 0], 50, 25)

        # Save the initial image
        pi.writeraw(img, output_file("before_coloring"))

        # Minimize number of colors in the image, still making
        # sure that all neighbouring spheres have different color.
        pi.greedycoloring(img)

        # Save the result
        pi.writeraw(img, output_file("after_coloring"))
```

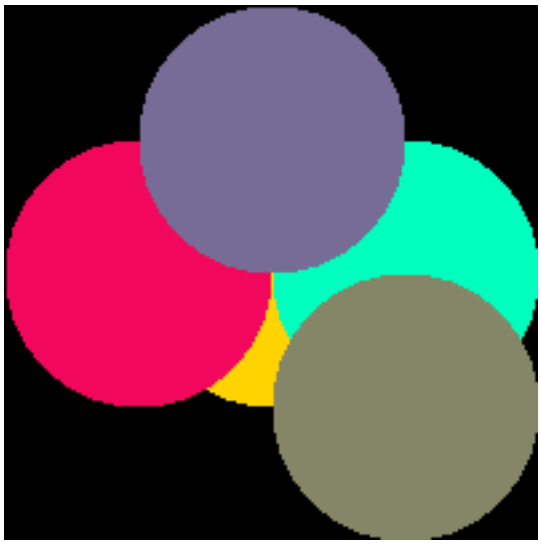

Fig. 8: Input image for greedy coloring. Pixel values are converted to colors using random look-up table, background is black.

3.7.9 Help

This example shows how to use the *help* command and commands related to it:

```
def help():
        """
        Demonstrates Pi2 help commands.
```

(continues on next page)

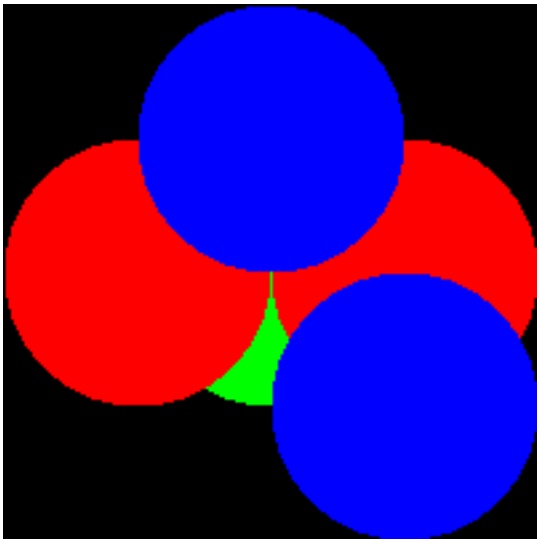

Fig. 9: Output image from greedy coloring. Pixel values are converted to colors using random look-up table, background is black.

(continued from previous page)

```
        IPython docstrings contain the same information.
        """

        # Show generic info
        pi.info()

        # Show license(s) related to the pi2
        pi.license()

        # List all available commands
        pi.help()

        # Show help for convert command
        pi.help('convert')
```

Output of this example very lengthy and is not shown here. Part of the output can be seen on the *License*, *Command reference*, and *convert* pages.

In addition to the *help* command, the iPython help system (TAB and ?) and .NET XML API documentation are supported.

3.7.10 Histogram

This example shows how to calculate histogram of an image. In addition to the bin counts, the *hist* command outputs also bin starts.

In the end of the example the histogram is plotted using PyPlot.

```
def histogram():
        """
        Demonstrates histogram calculation.
        """
```

(continues on next page)

(continued from previous page)

```
        # Read image
        img = pi.read(input_file())

        # Calculate histogram
        hist_min = 0        # Start of the first bin
        hist_max = 1000     # End of the last bin
        bin_count = 100     # Count of bins
        hist = pi.newimage(ImageDataType.FLOAT32)
        bins = pi.newimage(ImageDataType.FLOAT32)
        pi.hist(img, hist, bins, hist_min, hist_max, bin_count)


        # Get histogram data as NumPy array
        data = hist.get_data()
        bin_starts = bins.get_data()

        # Most of the background pixels are in the first bin.
        # We set it to zero so that the output image becomes nicely scaled
        # automatically.
        data[0] = 0

        # Bin edges can be generated like this if they have not been grabbed
        # from hist(...) command
        #bin_edges = np.linspace(hist_min, hist_max, bin_count + 1)

        # Convert bin starts to bin centers
        bin_size = bin_starts[1] - bin_starts[0]
        bin_centers = bin_starts + bin_size / 2


        # Plot the histogram
        import matplotlib.pyplot as plt
        plt.bar(bin_centers, data, 0.8*bin_size)
        plt.xlabel('Gray value')
        plt.ylabel('Count')
        plt.tight_layout()
        plt.show(block=False)
        plt.savefig(output_file('histogram.png'))
```

3.7.11 Filling a cavity with a level set method

This example shows how to fill a concave cavity using a level set method.

```
def levelset_fill_cavity():
        """
        Demonstrates how to fill a non-convex cavity using a level-set method.
        """

        # Create an image that contains a rectangle with a cavity in its edge.
        # For this example we make a 2D image for easier visualization,
        # but everything should work the same for a 3D volume image, too.
        c = 30 # Radius of the image
        geom = pi.newimage(ImageDataType.UINT8, 2 * c, 2 * c)
        pi.box(geom, [c - 15, c - 15, 0], 30, 128)      # The box
```

(continues on next page)

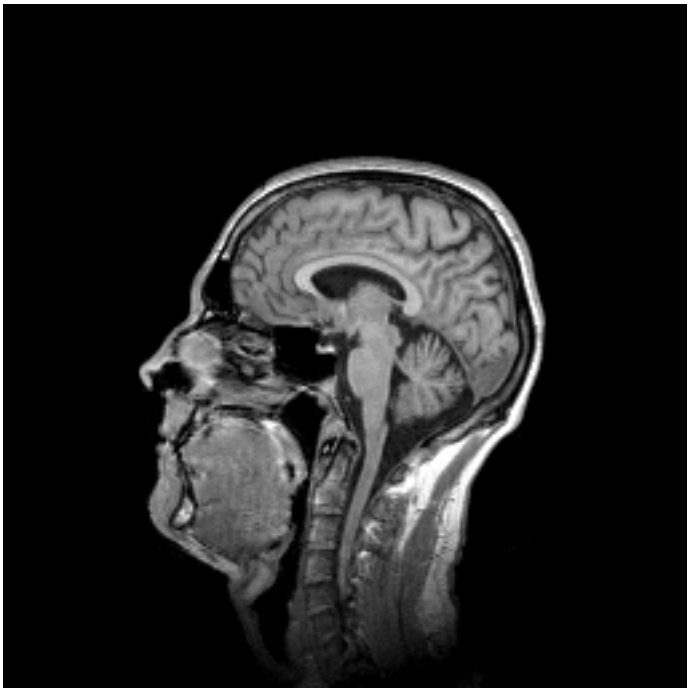

Fig. 10: Slice of the input image.

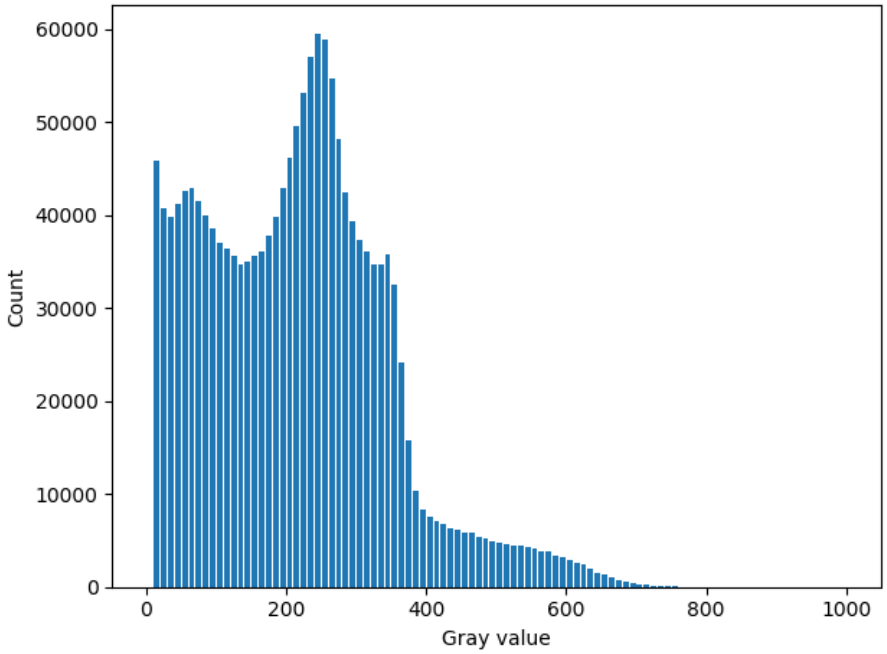

Fig. 11: Histogram of the input image. The peak corresponding to the background has been erased.

Fig. 12: Edge of segmented region shown as a red outline on top of the original image. The animation shows evolution of the edge as a function of iteration number.

(continued from previous page)

```
        pi.sphere(geom, [c, c, 0], 8, 0)                # Part of cavity
        pi.sphere(geom, [c, c - 6, 0], 6, 0)            # Part of cavity
        pi.sphere(geom, [c, c - 12, 0], 6, 0)           # Part of cavity

        # Noise to make the image more realistic
        pi.noise(geom, 0, 20)

        # Save the geometry
        pi.writetif(geom, output_file('levelset_geom'))


        # Initialize an image that holds the level set function.
        # After iteration below, the segmentation is the part of phi
        # that has positive value.
        phi = pi.newlike(geom, ImageDataType.FLOAT32)

        # Update phi iteratively
        for n in np.arange(0, 200):
                print(f"Iteration {n}" )

                # Construct force field that acts on the surface defined by phi
                # The force will be the sum of three terms.
                F = pi.newlike(phi)



                # First term: the force is positive inside the object and negative␣
↪everywhere else.
                # This makes the surface take the shape of the object.
                pi.copy(geom, F)
                pi.divide(F, 128)
                pi.subtract(F, 0.5)



                # Second term: Penalize high curvature by making curvy
                # regions less curved.
                kappa = pi.newlike(phi)
                pi.meancurvature(phi, kappa, 0.5)

                # Multiply the curvature values to scale them correctly.
                pi.multiply(kappa, -5)

                # Remove negative curvature values. They correspond to
                # convex shapes, and we want to zero those so that
                # they don't have any effect on the surface.
                pi.max(kappa, 0)

                # Add kappa to the force term
                pi.add(F, kappa)



                # Third term: Normal force
                # This term makes the surface move towards its normal.
                # This term is not required in this example.
                #L = pi.newlike(phi)
```

(continues on next page)

(continued from previous page)

```
                #pi.gradientmagnitude(phi, L, 0.75, 0)
                #pi.multiply(L, 0.01)
                #pi.add(F, L)


                # Smooth the total force a little bit
                # This is not strictly by the book, but smoothing seems to
                # make phi converge faster to a smoother result.
                tmp = pi.newlike(F)
                pi.gaussfilter(F, tmp, 0.5, BoundaryCondition.NEAREST)
                pi.set(F, tmp)


                # Multiply by time step
                dt = 1
                pi.multiply(F, dt)

                # Add force*dt to the surface
                pi.add(phi, F)


                # Convert phi to segmentation and save it for visualization purposes
                vis = pi.newlike(phi)
                pi.copy(phi, vis)
                pi.threshold(vis, 0)
                pi.convert(vis, ImageDataType.UINT8)
                pi.writetif(vis, output_file(f"./levelset_result/{n}"))
```

3.7.12 Line-enhancing filtering

This example demonstrates line-enhancing filtering according to *Frangi*.

When using the *frangifilter* command, the scaling of the output values depends on the pixel data type of the input image. The results of the Frangi filtering are always in the range $[0, 1]$. In order to be able to express the results in an image of integer pixel data type, the Frangi filter values are scaled to range $[0, M]$, where M is the maximum value that can be expressed with the pixel data type.

If the image is of floating point pixel data type, the values are not scaled.

```
def linefilters():
        """
        Calculates Frangi line-enhancing filter.
        """

        # Read image
        img = pi.read(input_file())

        # Frangi filter, 16-bit result
        frangi_16bit = pi.newlike(img, ImageDataType.UINT16)
        pi.frangifilter(img, frangi_16bit, 3.0)
        pi.writeraw(frangi_16bit, output_file('frangi_16bit'))


        # Frangi filter, 32-bit result. Notice different scaling in the output,␣
↪compared to the 16-bit version.
```

(continues on next page)

(continued from previous page)

```
        frangi_32bit = pi.newlike(img, ImageDataType.FLOAT32)
        pi.convert(img, ImageDataType.FLOAT32)
        pi.frangifilter(img, frangi_32bit, 3.0)
        pi.writeraw(frangi_32bit, output_file('frangi_32bit'))
```

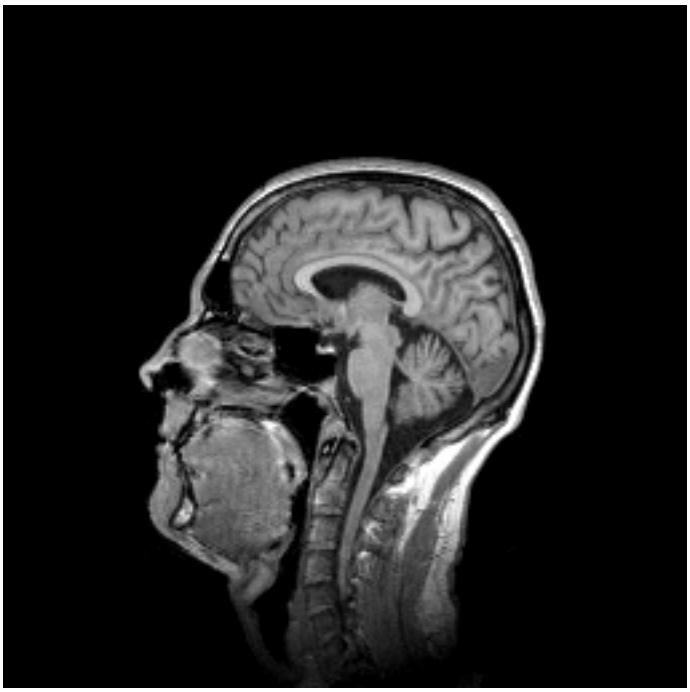

Fig. 13: One slice of the input image.

3.7.13 Local thickness / opening transform

This example demonstrates how to calculate *local thickness map*:

```
def thickmap():
        """
        Demonstrates use of thickness map functions.
        """

        # Read input image. Input must be binary (or it must be made binary,
        # see e.g. 'threshold' command)
        geom = pi.read(input_file_bin())

        # Create output image
        tmap = pi.newimage(ImageDataType.FLOAT32)

        # Convert the input to large enough data type so that it can hold squared␣
↪distance values.
        # If the data type is too small, an error is raised.
        pi.convert(geom, ImageDataType.UINT32)
        pi.tmap(geom, tmap)
        pi.writeraw(tmap, output_file('head_tmap'))
```

(continues on next page)

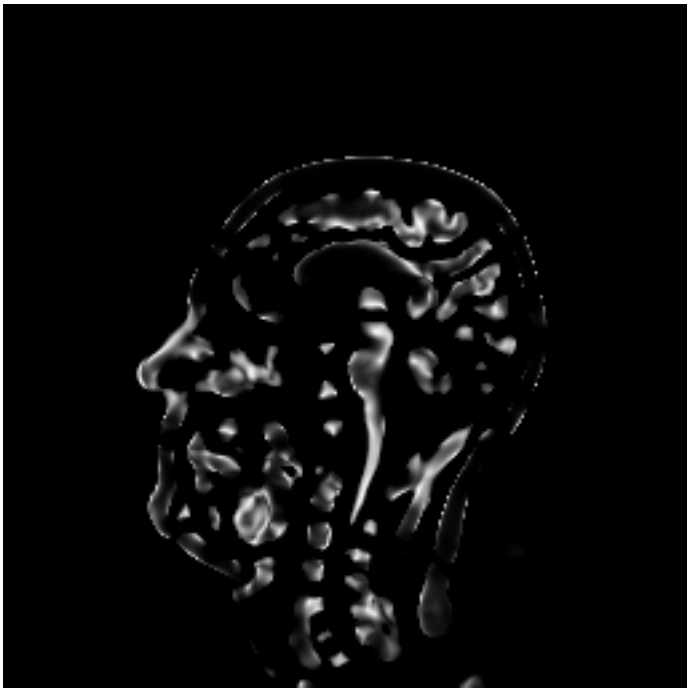

Fig. 14: Result of line-enhancing Frangi filter.

(continued from previous page)

```
        # If the default version consumes too much memory, there is also a slower␣
↪memory-saver version.
        # It is activated by setting last parameter of tmap to True:
        geom2 = pi.read(input_file_bin())
        tmap2 = pi.newimage(ImageDataType.FLOAT32)
        pi.convert(geom2, ImageDataType.UINT32)
        pi.tmap(geom2, tmap2, 0, True)
        pi.writeraw(tmap, output_file('head_tmap_memsave'))
```

3.7.14 Montage

This example shows how to make a *montage* of a 3D image. The montage shows multiple 2D slices of the 3D image in a single view.

```
def montage():
        """
        Demonstrates creation of a montage from a 3D image.
        """

        # Read the 3D image
        img = pi.read(input_file())

        # Create empty image for the montage
        montage = pi.newimage(img.get_data_type())

        # Make the montage, 4x3 slices, scaled to 0.5 of
        # original size
```

(continues on next page)

Fig. 15: One slice of the input image.

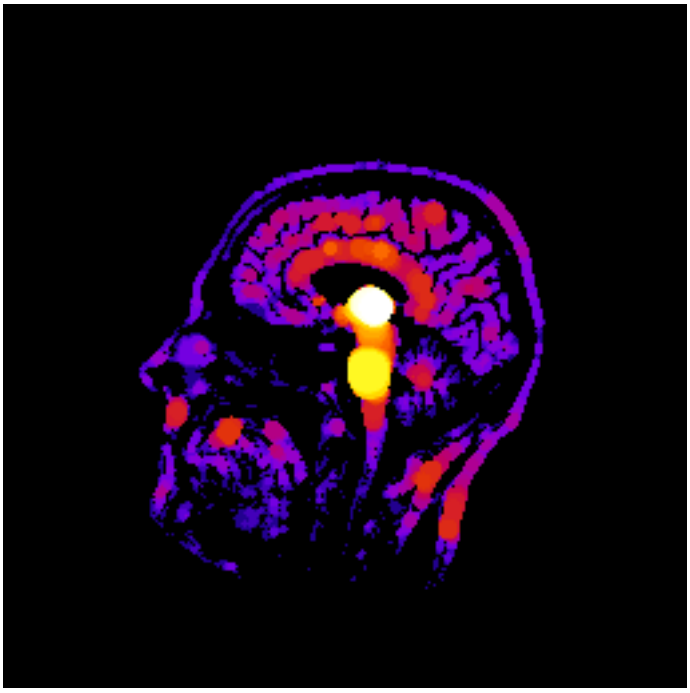

Fig. 16: Local thickness map of the input. Brighter colors correspond to thicker regions.

(continued from previous page)

```
        pi.montage(img, montage, 4, 3, 0.5)

        # Save the output
        pi.writetif(montage, output_file('montage'))
```

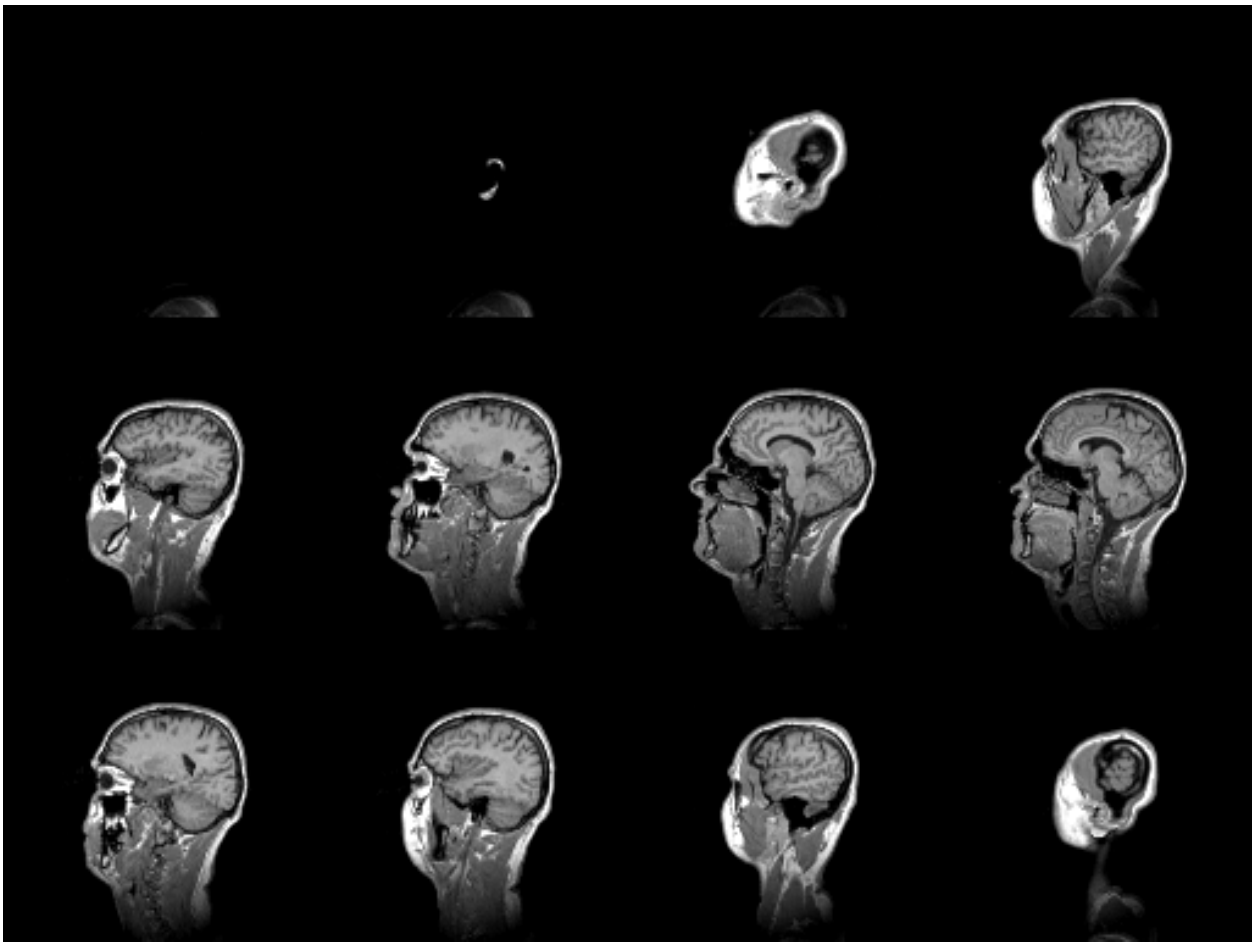

Fig. 17: Montage of the input image.

3.7.15 Estimation of orientation using the structure tensor method

This example shows how to estimate orientation of cylindrical structures using the structure tensor method.

First, the example generates an image containing randomly oriented cylinders. Orientation of each cylinder is drawn from a von Mises-Fisher distribution, that allows specifying a main direction and spread of the orientations. The cylinders are drawn into a pi2 image. In the code below, the sampling procedure is skipped as it is somewhat lengthy and not directly related to pi2. The sampling code is shown in the bottom of this page.

```
def generate_orientation_image(main_az=np.pi/4, main_pol=np.pi/2, kappa=3, n=150):
        """
        Generates test image for orientation analysis demonstrations.
        By default, plots 100 capsules whose orientations are taken from from von␣
↪Mises-
```

(continues on next page)

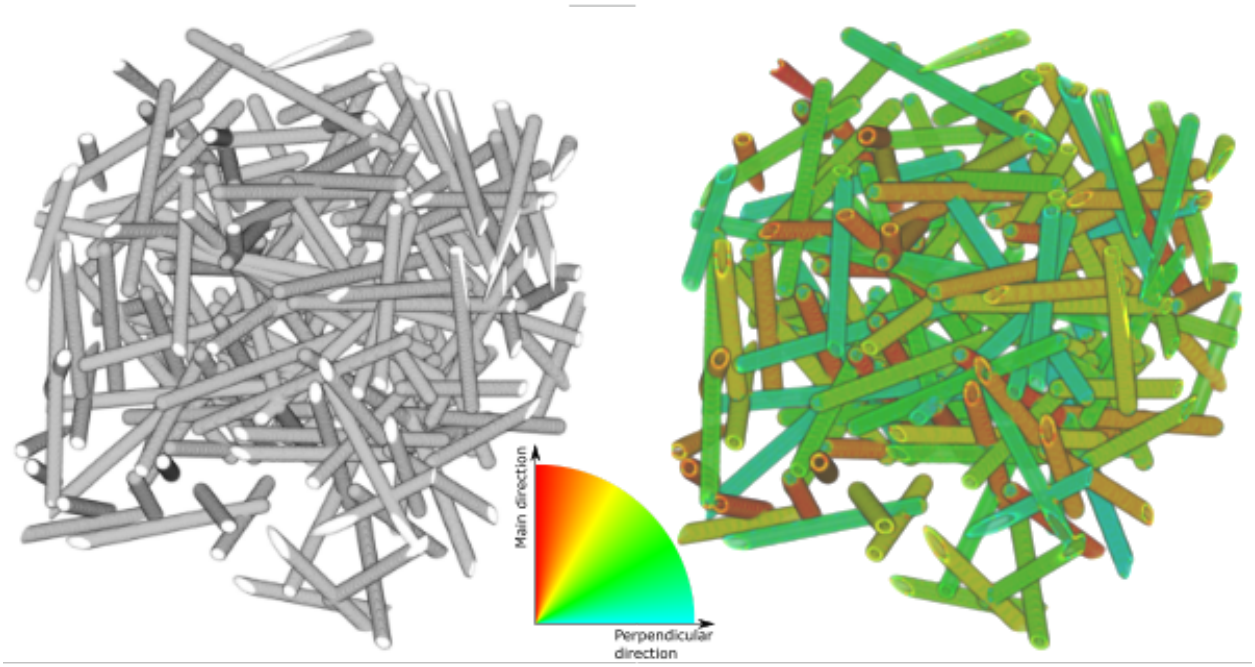

Fig. 18: 3D visualization of the generated input image (left panel), and similar visualization where the fibres have been colored according to the local orientation (right).

(continued from previous page)

```
        Fisher distribution (main direction = x + 45 deg towards y, kappa = 3)

        Returns the resulting pi2 image and direction vectors.
        """

        directions = sample_orientations_from_vonmises_fisher(main_az, main_pol,␣
↪kappa, n)

        size = 300
        img = pi.newimage(ImageDataType.UINT8, [size, size, size])
        for dir in directions:

                L = 100
                r = 5
                pos = np.random.rand(1, 3) * size

                pi.capsule(img, pos - L * dir, pos + L * dir, r, 255)

        return img, directions
```

The generated image is then analyzed using the *cylinderorientation* command. The command returns orientation of cylindrical structures at each pixel, in spherical coordinates. The azimuthal and polar (ϕ and θ) coordinates are visualized below.

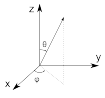

Fig. 19: Spherical coordinate system used in pi2. Here, ϕ is the azimuthal angle and θ is the polar angle.

A visualization of the orientations is made using the *mainorientationcolor* command. There, each pixel is assigned a color based on angle between a selected main orientation and the local orientation in the pixel.

In the end, the example plots the true orientation distribution of the cylinders and the distribution estimated from the image by statistical binning of local orientation angles. Both distributions are plotted into the same figure, shown below.

```
def orientation_analysis():
        """
        Demonstrates how to determine and visualize orientation of structures.
        """

        # Create test image with given main fibre direction
        main_azimuthal = np.pi/4
        main_polar = np.pi/2
        img, true_orientations = generate_orientation_image(main_azimuthal, main_
↪polar)

        # Save it for later visualization
        pi.writetif(img, output_file('cylinders'))

        # Calculate orientation of cylinders
        # Note that we need to convert the input image to float32
        # format as it is used as an output image, too.
        # Additionally we need images to store the azimuthal and polar
        # orientation angles.
        pi.convert(img, ImageDataType.FLOAT32)
        azimuthal = pi.newimage(ImageDataType.FLOAT32)
        polar = pi.newimage(ImageDataType.FLOAT32)
        pi.cylinderorientation(img, azimuthal, polar, 1, 1)

        # Now img has been replaced with 'orientation energy'
        pi.writetif(img, output_file('cylinders_energy'))

        # Make a color-coded visualization of the orientations
        r = pi.newimage()
        g = pi.newimage()
        b = pi.newimage()
        pi.mainorientationcolor(img, azimuthal, polar, main_azimuthal, main_polar, r,␣
↪g, b)
        #pi.axelssoncolor(img, azimuthal, polar, r, g, b) # This is another␣
↪possibility if main orientation is not available.
        pi.writeraw(r, g, b, output_file('cylinders_main_orientation_color'))


        # Make orientation histogram.
        # Energy is used as weight
        hist = pi.newimage(ImageDataType.FLOAT32)
        bins1 = pi.newimage(ImageDataType.FLOAT32)
        bins2 = pi.newimage(ImageDataType.FLOAT32)
        pi.whist2(azimuthal, -np.pi, np.pi, 20, polar, 0, np.pi, 10, img, hist, bins1,
↪ bins2) # 20 azimuthal angle bins, 10 polar angle bins




        # Make a plot that compares the true orientation distribution to the␣
↪estimated one
```

(continues on next page)

(continued from previous page)

```
        import matplotlib.pyplot as plt
        fig = plt.figure(figsize=(4.5, 6))

        # First plot the true orientations
        plt.subplot(2, 1, 1)

        # Convert directions to polar coordinates using the same convention that pi2␣
↪uses
        azs = []
        pols = []
        for dir in true_orientations:
                x = dir[0]
                y = dir[1]
                z = dir[2]
                r = np.sqrt(x * x + y * y + z * z)
                azimuthal = np.arctan2(y, x)
                polar = np.arccos(z / r)
                azs.append(azimuthal)
                pols.append(polar)

        # Calculate orientation histogram using the NumPy method
        hst, xedges, yedges = np.histogram2d(azs, pols, range=[[-np.pi, np.pi], [0,␣
↪np.pi]], bins=[20, 10])

        # Plot the histogram
        pltimg = plt.imshow(hst.transpose(), extent=(xedges[0], xedges[-1], yedges[0],
↪ yedges[-1]))
        cbar = fig.colorbar(pltimg)
        plt.xlabel('Azimuthal angle [rad]')
        plt.ylabel('Polar angle [rad]')
        plt.title('True distribution of cylinder orientations')


        # Now plot the histogram estimated from the image
        plt.subplot(2, 1, 2)
        pltimg = plt.imshow(hist.get_data(), extent=(-np.pi, np.pi, 0, np.pi))
        cbar = fig.colorbar(pltimg)
        plt.xlabel('Azimuthal angle [rad]')
        plt.ylabel('Polar angle [rad]')
        plt.title('Distribution estimated from the image')

        # Show and print the figure
        plt.tight_layout()
        plt.show(block=False)

        plt.savefig(output_file('bivariate_histogram_comparison.png'))
```

The code used to sample the von Mises-Fisher distribution:

```
def sample_orientations_from_vonmises_fisher(main_az, main_pol, kappa, n):
        """
        Sample directions from von Mises-Fisher distribution.
        main_az and main_pol give the azimuthal and polar angles of the main␣
↪direction.
        kappa indicates the spread of the directions around the main direction.
        Large kappa means small spread.
        n gives the number of directions to generate.
```

(continues on next page)

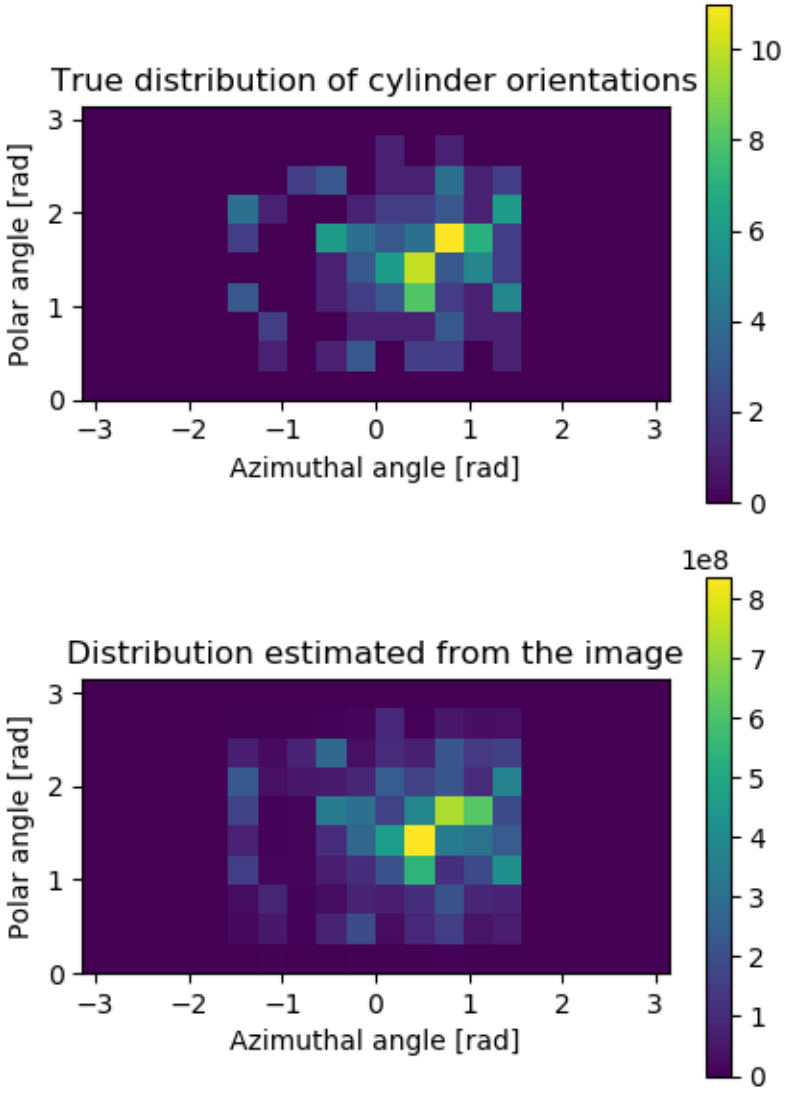

Fig. 20: Orientation distributions of cylinders in the image generated in the example. The top panel shows the true distribution, and the bottom panel shows the distribution estimated from the 3D image using the structure tensor method. The distributions show directions corresponding to the whole sphere, but notice that the half-sphere corresponding to the negative x-values is empty. This happens because the orientations are symmetrical, i.e. directions $-\vec{r}$ and $\vec{r}$ describe the same orientation, and therefore half of the possible directions are redundant.

(continued from previous page)

```
        Returns n 3-component unit vectors.

        This code mostly from https://stats.stackexchange.com/questions/156729/
↪sampling-from-von-mises-fisher-distribution-in-python
        but its correctness has not been checked. It seems to
        create plausible results, though.
        """

        import scipy as sc
        import scipy.stats
        import scipy.linalg as la

        def sample_tangent_unit(mu):
                mat = np.matrix(mu)

                if mat.shape[1]>mat.shape[0]:
                        mat = mat.T

                U,_,_ = la.svd(mat)
                nu = np.matrix(np.random.randn(mat.shape[0])).T
                x = np.dot(U[:,1:],nu[1:,:])
                return x/la.norm(x)


        def rW(n, kappa, m):
                dim = m-1
                b = dim / (np.sqrt(4*kappa*kappa + dim*dim) + 2*kappa)
                x = (1-b) / (1+b)
                c = kappa*x + dim*np.log(1-x*x)

                y = []
                for i in range(0,n):
                        done = False
                        while not done:
                                z = sc.stats.beta.rvs(dim/2,dim/2)
                                w = (1 - (1+b)*z) / (1 - (1-b)*z)
                                u = sc.stats.uniform.rvs()
                                if kappa*w + dim*np.log(1-x*w) - c >= np.log(u):
                                        done = True
                        y.append(w)

                return np.array(y)


        def rvMF(n,theta):
                dim = len(theta)
                kappa = np.linalg.norm(theta)
                mu = theta / kappa

                w = rW(n, kappa, dim)

                result = []
                for sample in range(0,n):

                        v = sample_tangent_unit(mu).transpose()
                        v = np.asarray(v.transpose()).squeeze()
```

(continues on next page)

(continued from previous page)

```
                        result.append(np.sqrt(1-w[sample]**2)*v + w[sample]*mu)

                return result




        # First sample directions from the von Mises-Fisher distribution
        main_dir = np.array([np.cos(main_az) * np.sin(main_pol), 1 * np.sin(main_az)␣
↪* np.sin(main_pol), 1 * np.cos(main_pol)])
        directions = rvMF(n, kappa * main_dir)

        # Convert to orientations (v and -v are the same)
        # by ensuring that all directions have positive x coordinate.
        # This is the same convention used in pi2.
        for dir in directions:
                if dir[0] < 0:
                        dir *= -1

        return directions
```

3.7.16 Particle analysis

This example demonstrates how to analyze the shape of objects ('particles') in a binary image.

First, the example generates artificial particles using the *sphere* and *box* commands.

```
def generate_particles(sphere_count=1000, box_count=1000):
        """
        analyze_particles demo uses this to generate an image containing some random␣
↪regions.
        """

        import random

        img = pi.newimage(ImageDataType.UINT8, 500, 500, 500)

        # Generate some spheres
        for i in range(0, sphere_count):
                pos = [random.randint(0, 500), random.randint(0, 500), random.
↪randint(0, 500)]
                r = random.randint(1, 20)
                pi.sphere(img, pos, r, 255)

        # Generate some boxes
        for i in range(0, box_count):
                pos = [random.randint(0, 500), random.randint(0, 500), random.
↪randint(0, 500)]
                size = [random.randint(1, 20), random.randint(1, 20), random.
↪randint(1, 20)]
                pi.box(img, pos, size, 255)

        pi.writeraw(img, output_file('particles'))
```

Next, the available analyzers are queried using command *listanalyzers*. The command lists all available analyzers and columns they produce to the output. At the time of writing the analyzers are

coordinates

Shows (zero-based) coordinates of a pixel that is guaranteed to be inside the particle. Output column names are 'X', 'Y', and 'Z'.

isonedge

Tests whether the particle touches image edge. If it does, returns 1 in a column 'Is on edge'; if it doesn't, returns zero.

volume

Shows total volume of the particle in pixels. Outputs one column with name 'Volume'.

pca

Calculates orientation of the particle using principal component analysis. Outputs:

- Centroid of the particle in columns 'CX', 'CY', and 'CZ'.
- Meridional eccentricity in column 'e (meridional)'.
- Standard deviations of the projections of the particle points to the principal axes in columns 'l1', 'l2', and 'l3'.
- Orientations of the three principal axes of the particle, in spherical coordinates, in columns 'phiN' and 'thetaN', where N is in 1...3, phiN is the azimuthal angle and thetaN is the polar angle.
- Maximum distance from the centroid to the edge of the particle in colum 'rmax'.
- Maximum diameter of the projection of the particle on each principal component in columns 'd1', 'd2', and 'd3'.
- Scaling factor for a bounding ellipsoid in column 'bounding scale'. An ellipsoid that bounds the particle and whose semi-axes correspond to the principal components has semi-axis lengths blN, where b is the bounding scale and lN are the lengths of the principal axes.

surfacearea

Shows surface area of particles determined with the Marching Cubes method. Outputs one column 'Surface area'.

bounds

Calculates axis-aligned bounding box of the particle. Returns minimum and maximum coordinates of particle points in all coordinate dimensions. Outputs columns 'minX' and 'maxX' for each dimension, where X is either x, y, or z.

boundingsphere

Shows position and radius of the smallest possible sphere that contains all the points in the particle. Outputs columns 'bounding sphere X', 'bounding sphere Y', 'bounding sphere Z', and 'bounding sphere radius'. Uses the MiniBall algorithm from Welzl, E. - Smallest enclosing disks (balls and ellipsoids).

The actual particle analysis is performed with *analyzeparticles* command. It needs as input the binary image, image for results, and a string that contains the names of the desired analyzers.

The output image contains one row for each particle in the image. The meanings of columns can be found using command *headers*. Its output in this case is `Volume [pixel], X [pixel], Y [pixel], Z [pixel], minx, maxx, miny, maxy, minz, maxz, bounding sphere X [pixel], bounding sphere Y [pixel], bounding sphere Z [pixel], bounding sphere radius [pixel], Is on edge [0/1]`.

Finally, the results are converted to NumPy format and the volume distribution of the particles is plotted.

```
def analyze_particles():
        """
        Demonstrates particle (i.e, region or blob) analysis, and some drawing␣
↪commands.
        """

        # Generate particle image
        generate_particles()

        # Show analyzer names.
        # This does not do anything else than shows which analyzers are available.
        pi.listanalyzers()

        # Make a list of analyzers that we want to use
        analyzers = 'volume coordinates bounds boundingsphere isonedge'


        # Read generated data file
        img = pi.read(output_file('particles'))

        # Analyze particles
        result = pi.newimage(ImageDataType.FLOAT32)
        pi.analyzeparticles(img, result, analyzers)

        # Show titles of data columns
        print('Titles of columns in data table:')
        pi.headers(analyzers)

        # Get result data from the pi2 system
        pa = result.get_data()

        # Volume is in the first column (see output of 'headers' command)
        volume = pa[:, 0]

        # Plot volume histogram
        import matplotlib.pyplot as plt
        plt.hist(volume, density=True, bins=30)
        plt.xlabel('Volume [pixel]')
        plt.ylabel('Probability')
        plt.tight_layout()
        plt.show(block=False)
        plt.savefig(output_file('volume_distribution.png'))
```

Output:

```
coordinates
Shows (zero-based) coordinates of a pixel that is guaranteed to be inside the␣
↪particle. Output column names are X, Y, and Z.

isonedge
Tests whether the particle touches image edge. If it does, returns 1.0 in a column
↪'isonedge'; if it doesn't, returns zero.

volume
Shows total volume of the particle in pixels. Outputs one column with name 'volume'.

bounds
```

(continues on next page)

(continued from previous page)

```
Calculates axis-aligned bounding box of the particle. Returns minimum and maximum␣
↪coordinates of particle points in all coordinate dimensions. Outputs columns 'minX'␣
↪and 'maxX' for each dimension, where X is either x, y, or z.

boundingsphere
Shows position and radius of the smallest possible sphere that contains all the␣
↪points in the particle. Outputs columns 'bounding sphere X', 'bounding sphere Y',
↪'bounding sphere Z', and 'bounding sphere radius'.
Analyzing in 8 blocks.
Processing 464 particles on block edge boundaries...
Find edge points of particles...
Build forest...
Determine bounding boxes...
Divide boxes to a grid...
Determine overlaps...
Combine particles...
Analyze combined particles...
Titles of columns in data table:
Volume [pixel], X [pixel], Y [pixel], Z [pixel], minx, maxx, miny, maxy, minz, maxz,␣
↪bounding sphere X [pixel], bounding sphere Y [pixel], bounding sphere Z [pixel],␣
↪bounding sphere radius [pixel], Is on edge [0/1]
```

3.7.17 Filter particle analysis results, visualize particles

This example demonstrates use of Python commands to filter particle analysis results. After filtering, the remaining particles are filled with another color.

The example also makes another visualization of the particles drawn as ellipsoids. Similar *drawellipsoids* function call can be used to draw, e.g., bounding spheres of the particles, too.

The image of the particles is generated similarly to the *particle analysis example*.

```
def fill_particles():
        """
        Demonstrates analysis and filling of particles.
        """

        # Generate particle image
        generate_particles()


        # Read generated data file
        img = pi.read(output_file('particles'))

        # Analyze particles
        analyzers = 'volume coordinates pca'
        result = pi.newimage(ImageDataType.FLOAT32)
        pi.analyzeparticles(img, result, analyzers)

        # Get result data from the pi2 system
        pa = result.get_data()

        # Volume is in the first column (see the output of the 'headers' command)
        volume = pa[:, 0]
```

(continues on next page)

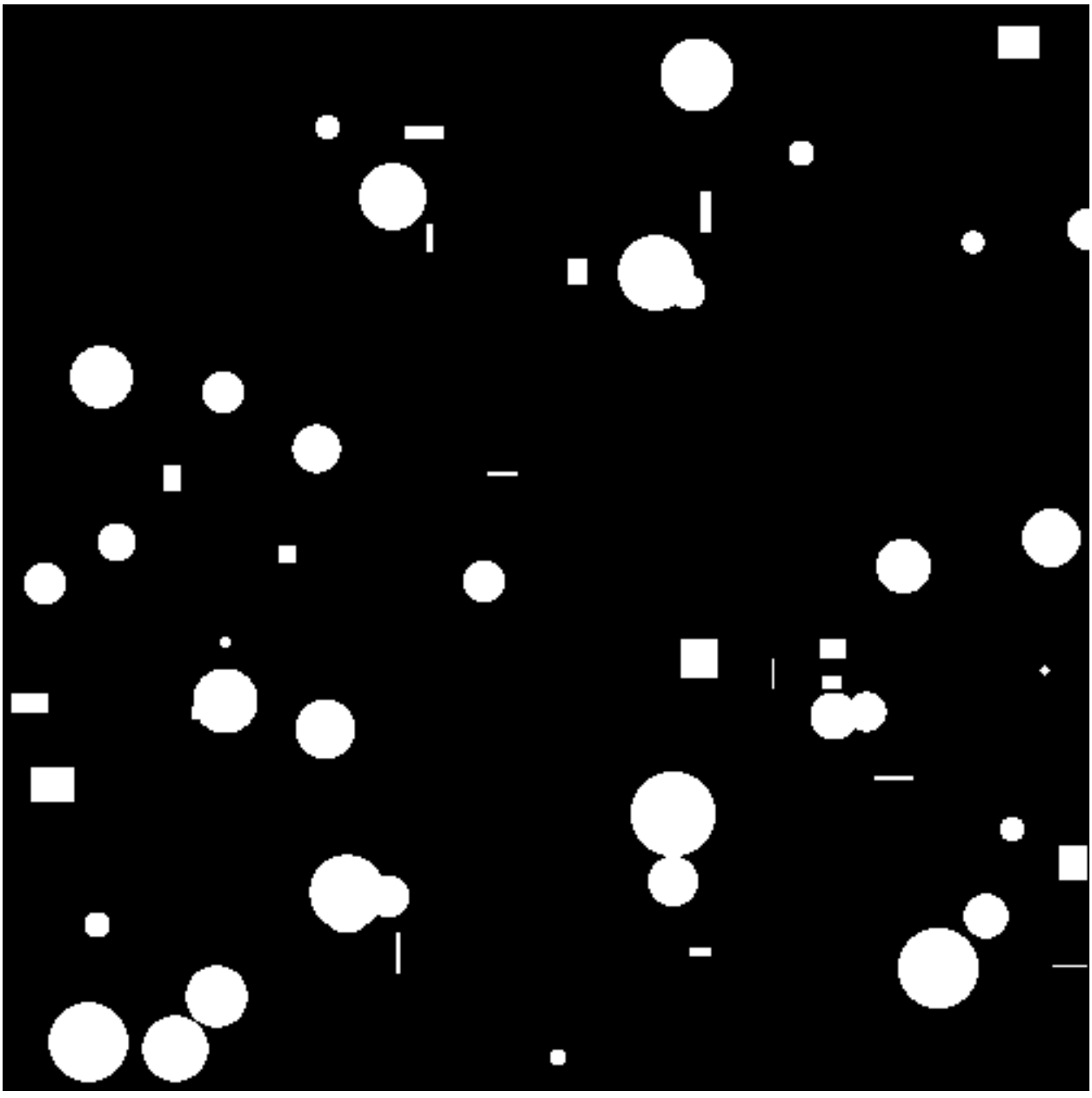

Fig. 21: One slice of the input image. Background is black, particles are white.

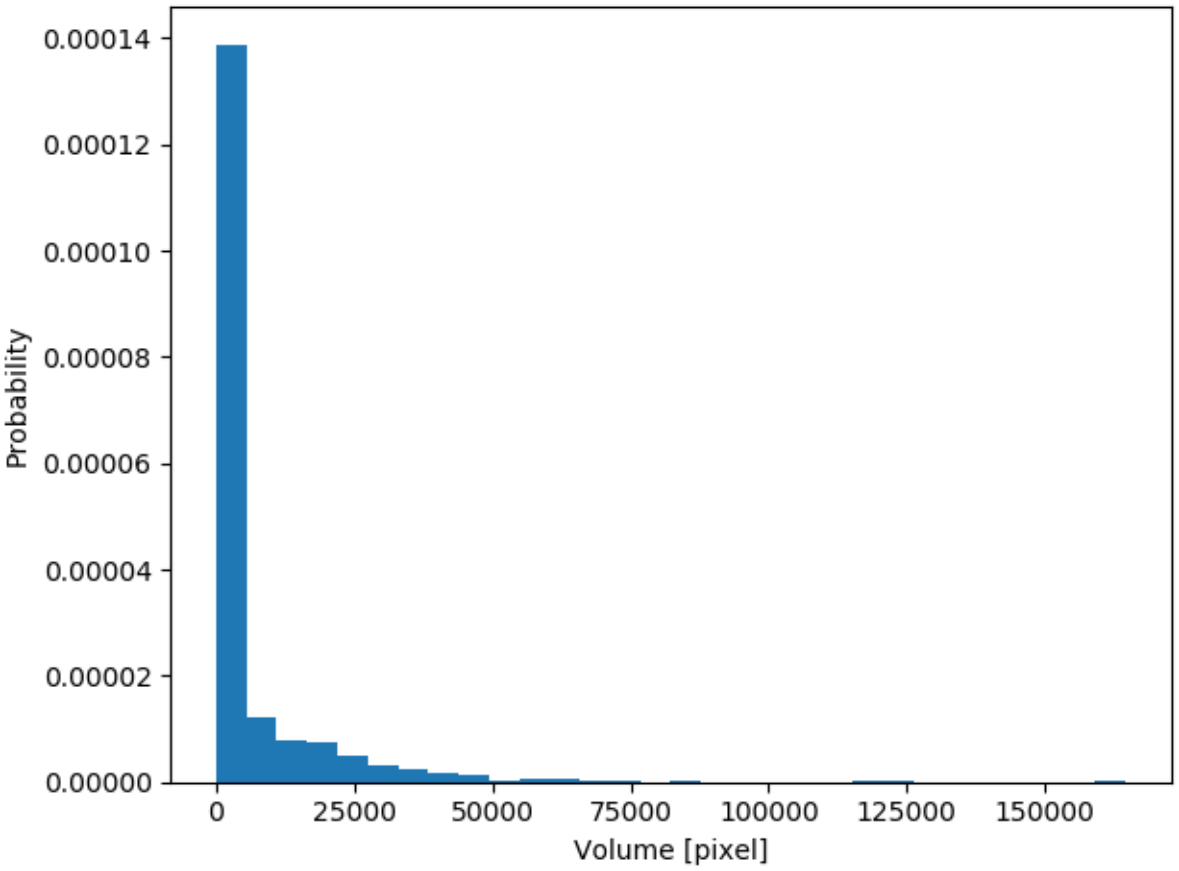

Fig. 22: Distribution of particle volume.

(continued from previous page)

```
        print(f"Before filtering the shape of the results matrix is {pa.shape}")

        # Filter the measurement results using suitable Python methods.
        # Here we take only the largest particles
        limit = 20000
        pa = pa[volume > limit, :]

        print(f"After filtering the shape of the results matrix is {pa.shape}")

        # We could use set_data(...) function to push the filtered results
        # back to pi2, but we may as well use the NumPy array pa directly
        # in pi2 commands.
        #result.set_data(pa)

        # Fill the particles that we left into the pa array (i.e. the big ones)
        # Note that (for simple distributed processing support) the fillparticles
        # sets all non-filled particles to 1.
        img = pi.read(output_file('particles'))
        pi.fillparticles(img, analyzers, pa, 2)
        # Set 1 -> 255 and 2 -> 128 so that the colors of the filled particles
        # correspond to the original colors.
        pi.replace(img, 1, 255)
        pi.replace(img, 2, 128)
        pi.writeraw(img, output_file('particles_large_colored'))



        # Make another visualization by drawing an ellipsoid approximation of each␣
↪particle
        # on top of the particles
        ellipsoid_vis = pi.read(output_file('particles'))

        # This draws the particles as ellipsoids whose volume equals particle volume,␣
↪with color 128
        pi.drawellipsoids(ellipsoid_vis, analyzers, result, 128, EllipsoidType.VOLUME)

        # Save the result
        pi.writeraw(ellipsoid_vis, output_file('particles_ellipsoids'))
```

Partial output:

```
Before filtering the shape of the results matrix is (1549, 4)
After filtering the shape of the results matrix is (170, 4)
```

3.7.18 Read and write images

This example shows how to read and write images in various file formats:

```
def read_and_write_image():
        """
        Demonstrates reading and writing images.
        """

        # .tif image
        img1 = pi.read(input_file('t1-head.tif'))
```

(continues on next page)

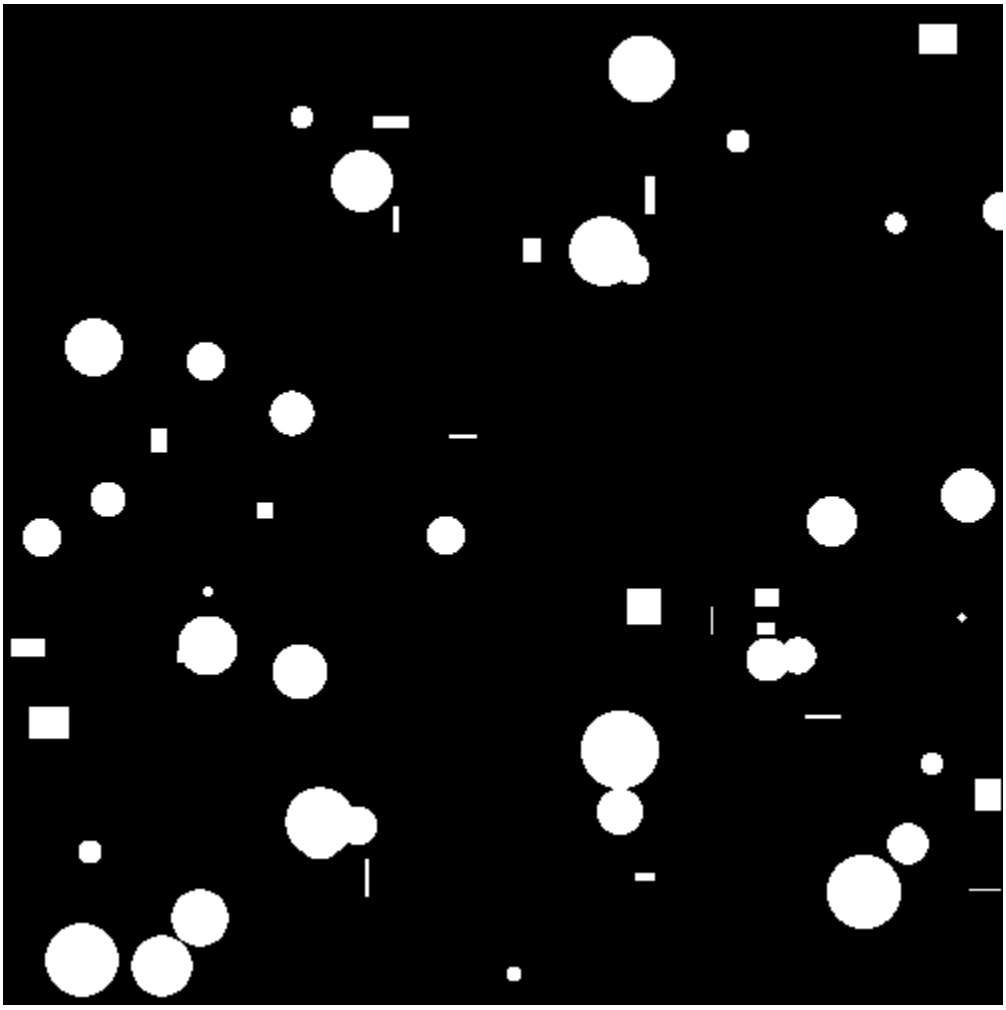

Fig. 23: Slice of the binary input image. Background is black, particles are white.

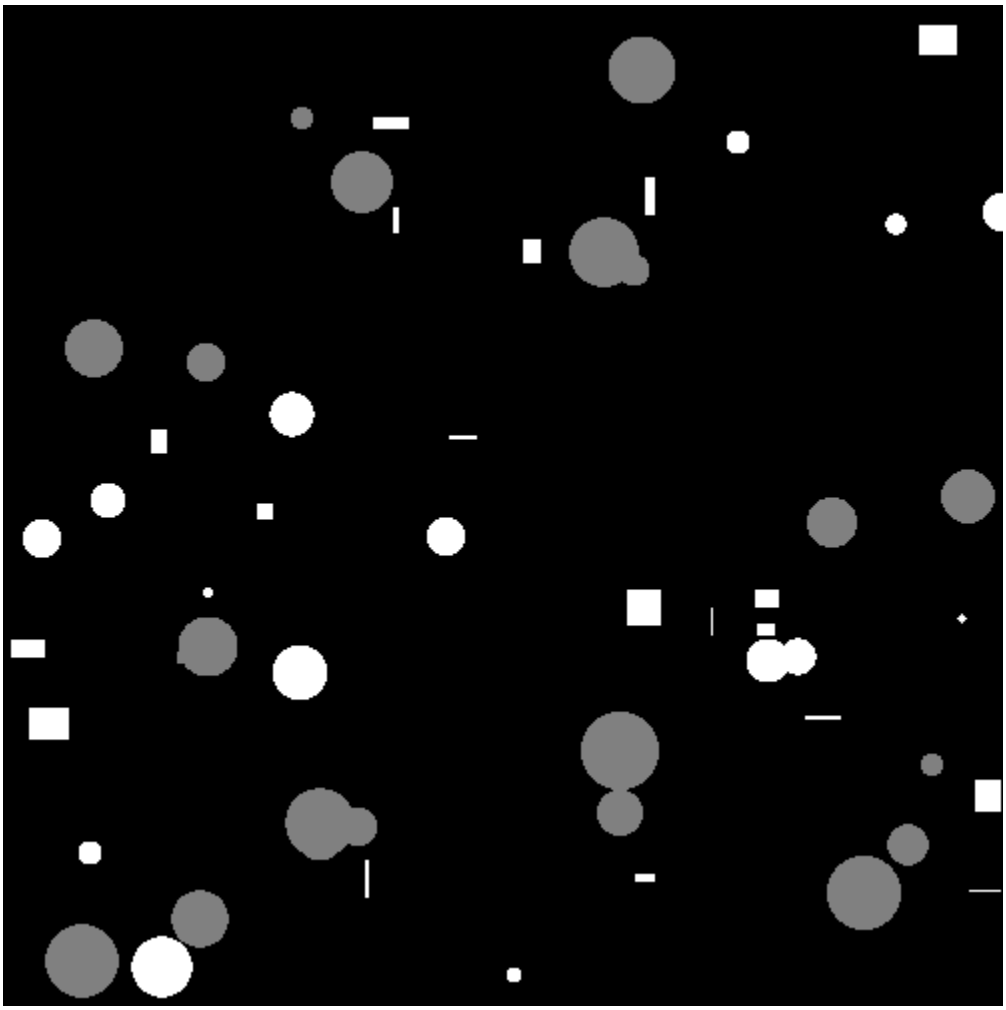

Fig. 24: Slice of the output of *fillparticles*. Background is black, original particles are white, and filled particles are gray.

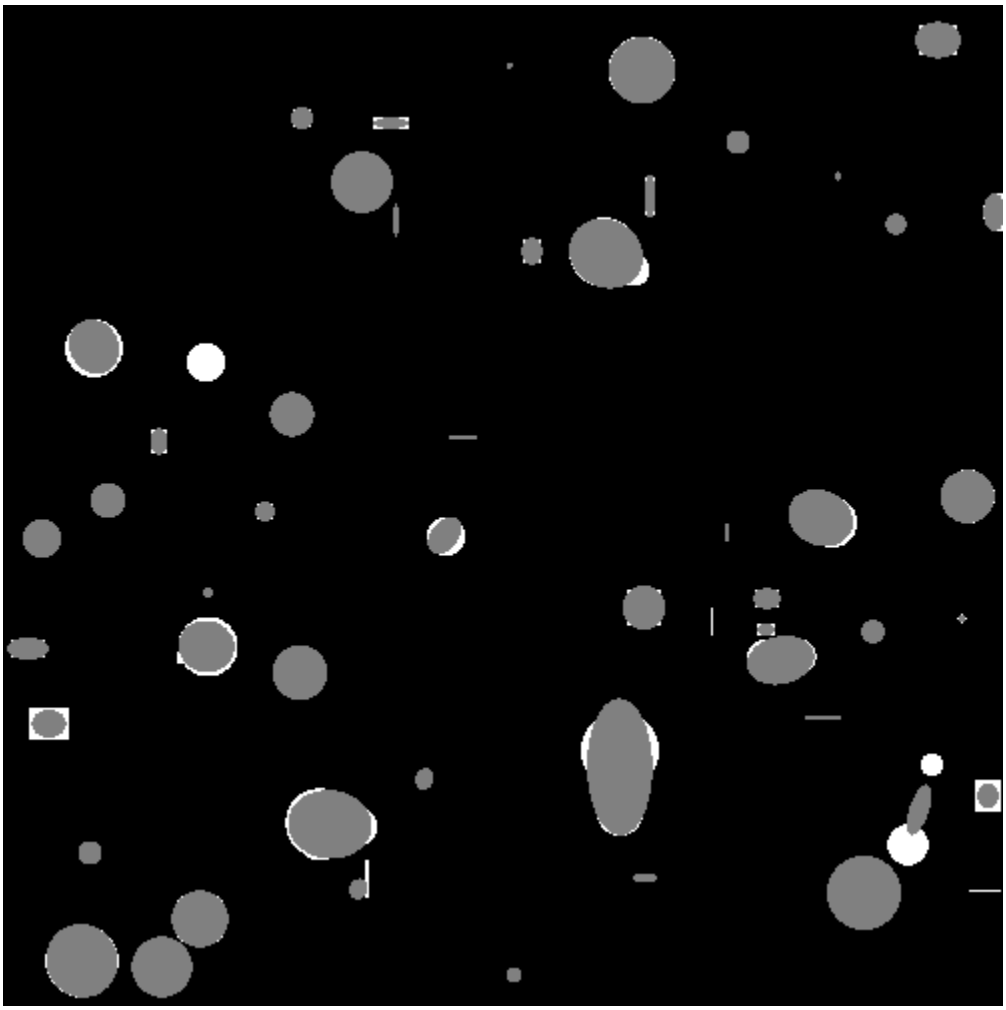

Fig. 25: Slice of the output of *drawellipsoids*. Background is black, original particles are white, and ellipsoids corresponding to the particles are gray.

(continued from previous page)

```
        # .png image
        img2 = pi.read(input_file('uint8.png'))

        # .raw file
        img3 = pi.read(input_file('t1-head_bin_'))

        # Save sequence. The @(3) declares that the file name should contain 3␣
↪numeric digits in the place of @, e.g. head_001_bin.tif.
        pi.writesequence(img3, output_file('binary_head/head_@(3)_bin.tif'))

        # Read sequence back. Notice that reading does not support (3) directive in␣
↪the sequence template!
        img4 = pi.read(output_file('binary_head/head_@_bin.tif'))

        # Images can also be read into existing images.
        pi.read(output_file('binary_head/head_@_bin.tif'), target_image=img1)

        # When saving .raw files, the dimensions are appended to the file name
        pi.writeraw(img2, output_file('uint8_as_raw'))

        # When reading .raw files, it is not necessary to give the dimensions and .
↪raw suffix
        # if the given part of the file name identifies an unique .raw file
        img2 = pi.read(output_file('uint8_as_raw'))
```

3.7.19 Image rotations

This example shows how to use *rotate*, *rot90cw*, *rot90ccw*, and *reslice* commands. First the example rotates input image 90 degrees clockwise, then 90 degrees around x-axis using the *reslice* command, and finally -60 degrees around vector $[1, 1, 0]$ using the *rotate* command.

```
def rotations():
        """
        Demonstrates rotations and re-slicing.
        """

        # Read image
        img = pi.read(input_file())

        # Rotate 90 degrees clockwise (around z-axis)
        rot90 = pi.newimage()
        pi.rot90cw(img, rot90)
        pi.writetif(rot90, output_file('rotate_90_clockwise'))

        # Reslice (rotate 90 degrees around x- or y-axis)
        top = pi.newimage()
        pi.reslice(img, top, ResliceDirection.TOP)
        pi.writetif(top, output_file('reslice_top'))

        # General rotation.
        # NOTE:
        # - The size of the output must be set to desired value before the call
        #   to rotate function.
        # - The angle is given in radians. Here it is -60 degrees.
```

(continues on next page)

(continued from previous page)

```
        # - The axis is given as a vector and it does not need to be a unit vector.
        grot = pi.newimage(img.get_data_type(), img.get_dimensions())
        pi.rotate(img, grot, -60/180*3.14, [1, 1, 0])
        pi.writetif(grot, output_file('general_rotation'))
```

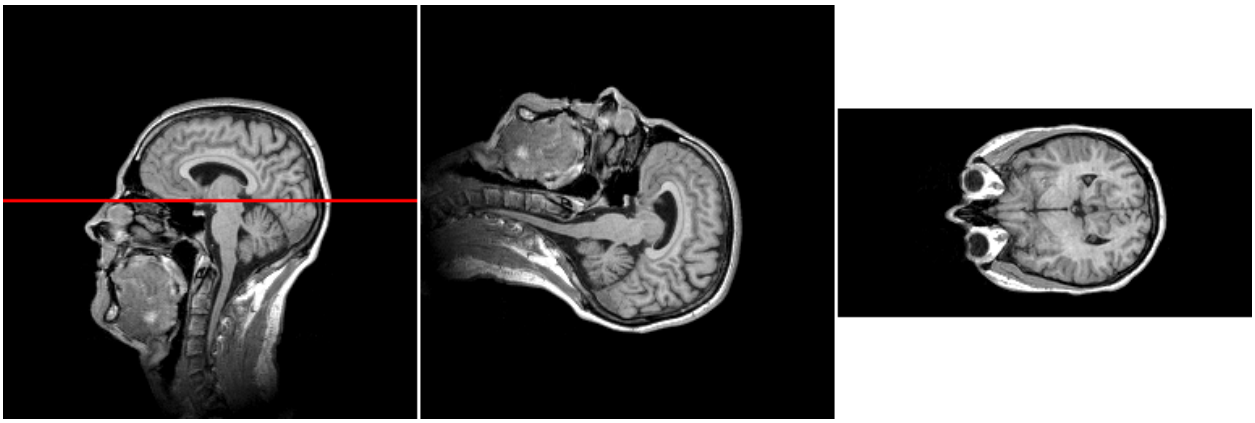

Fig. 26: One slice of the input image (left), one slice of the input rotated 90 degrees clockwise (middle), and one slice of the input rotated 90 degrees around x-axis using the *reslice* command. The location of the slice in the rightmost panel is shown with red line in the leftmost panel.

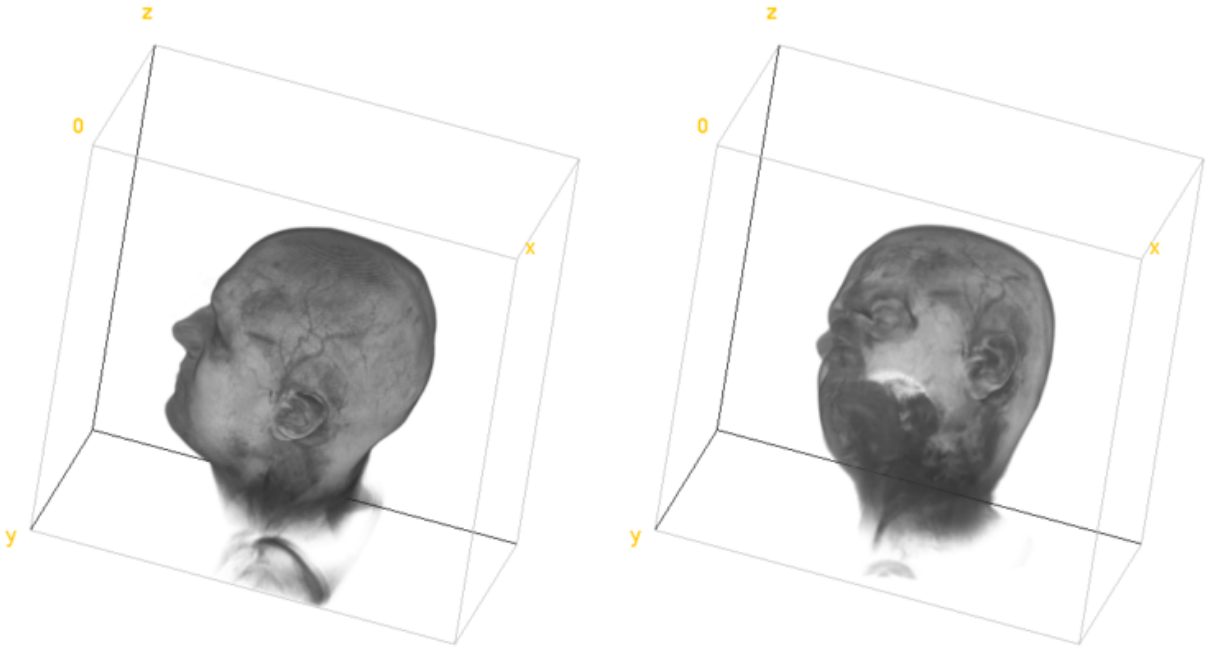

Fig. 27: 3D visualization of the original image (left) and the original rotated -60 degrees around $[1, 1, 0]$ (right).

3.7.20 Seeded distance map

This example shows how to use the *seeded distance map command*. The example creates simple geometry consisting of four spheres, and calculates seeded distance map using an arbitrary point inside the top-left sphere as a seed point.

```
def seeded_distance_map():
        """
        Demonstrates calculation of a seeded distance map.
        """

        # Create geometry
        geometry = pi.newimage(ImageDataType.UINT8, 50, 50, 50)
        pi.sphere(geometry, [15, 15, 25], 10, 255)
        pi.sphere(geometry, [15, 15 + 18, 25], 10.0, 255)
        pi.sphere(geometry, [15 + 18, 15 + 18, 25], 10.0, 255)
        pi.sphere(geometry, [15 + 22, 15, 25], 10.0, 255)
        pi.writetif(geometry, output_file("sdmap_geometry"))

        # Create seeds
        seeds = pi.newlike(geometry)
        pi.set(seeds, [24, 19, 25], 255)
        pi.writetif(seeds, output_file("sdmap_seeds"))

        # Calculate seeded distance map
        sdmap = pi.newlike(geometry, ImageDataType.FLOAT32)
        pi.sdmap(seeds, geometry, sdmap)
        pi.writetif(sdmap, output_file("sdmap_result"))
```

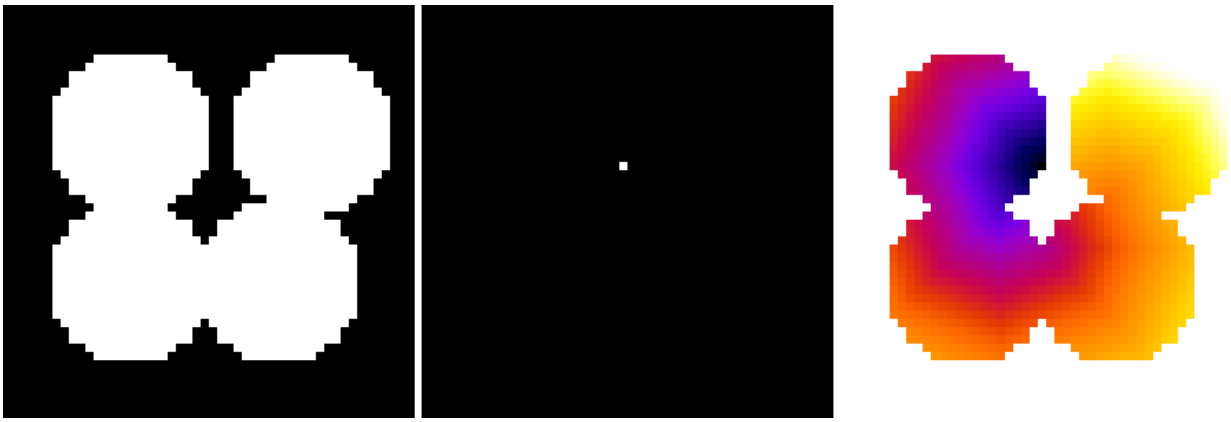

Fig. 28: The geometry (left) and seed points (middle) given as input to the seeded distance map algorithm. The resulting output distance map (right). Only the central slice of each volume image is shown.

3.7.21 Simple image math and saturation arithmetic

This example demonstrates how to do simple mathematical operations on images. All the operations are saturating, i.e. out-of-bounds results are clamped to the allowable value range of the pixel data type instead of wrapping around

Some operations support input images of different size. In this case, out-of-bounds pixels are taken from the nearest valid location in the edge of the image. As a result, one can e.g. multiply each slice of a 3D image by a 2D image. This process is often used to mask bad regions away (e.g. regions outside of the completely reconstructed cylinder in a tomographic image).

```
def math():
        """
        Demonstrates use of simple image math.
        """
```

(continues on next page)

(continued from previous page)

```
        # Add images
        img = pi.read(input_file())
        pi.add(img, img)
        pi.writeraw(img, output_file('head_added_to_itself'))


        # Subtract images
        img = pi.read(input_file())
        pi.subtract(img, img)
        pi.writeraw(img, output_file('head_subtracted_from_itself'))


        # Add constant
        # The math operations are saturating, i.e. if the result of an operation is␣
↪out of
        # bounds that can be represented with pixel data type, the value is clipped
        # to the bounds.
        # For example,
        # 200 + 200 = 255 for uint8 image,
        # 200 + 200 = 400 for uint16 image,
        # 2*30000 = 65535 for uint16 image,
        # etc.
        img = pi.read(input_file())
        pi.add(img, 65400) # Add large value to partially saturate 16-bit range
        pi.writeraw(img, output_file('head_saturated'))


        # Do you have a 2D mask that you would like to apply to
        # all slices of a 3D stack?
        # No problem, just specify True for 'allow dimension broadcast' parameter:
        img = pi.read(input_file())

        # Create mask whose size is the same than the size of the original but it␣
↪contains
        # only one slice. Then draw a circle into it, with color 1.
        mask = pi.newimage(img.get_data_type(), img.get_width(), img.get_height())
        pi.sphere(mask, [img.get_width() / 2, img.get_height() / 2, 0], img.get_
↪width() / 4, 1)
        pi.writetif(mask, output_file('mask'))

        pi.multiply(img, mask, True)
        pi.writetif(img, output_file('head_masked'))
```

3.7.22 Skeleton, saving to VTK format

This example demonstrates determination of a *line skeleton* of a binary image. The skeletonization process reduces all foreground structures into their approximate centerlines.

The skeleton can be *traced* into a graph structure, where skeleton branches are edges and their intersections are vertices. The traced branches contain information about the branch, e.g., its length.

Finally, the traced skeleton is saved into a .vtk file. The generated .vtk file is compatible, e.g., with ParaView:

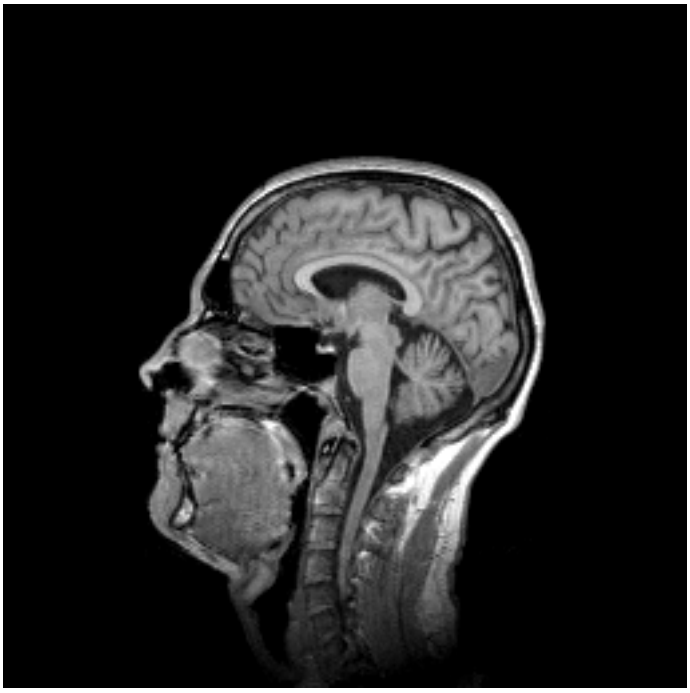

Fig. 29: One slice of the input image. Grayscale range is $[0, 463]$.

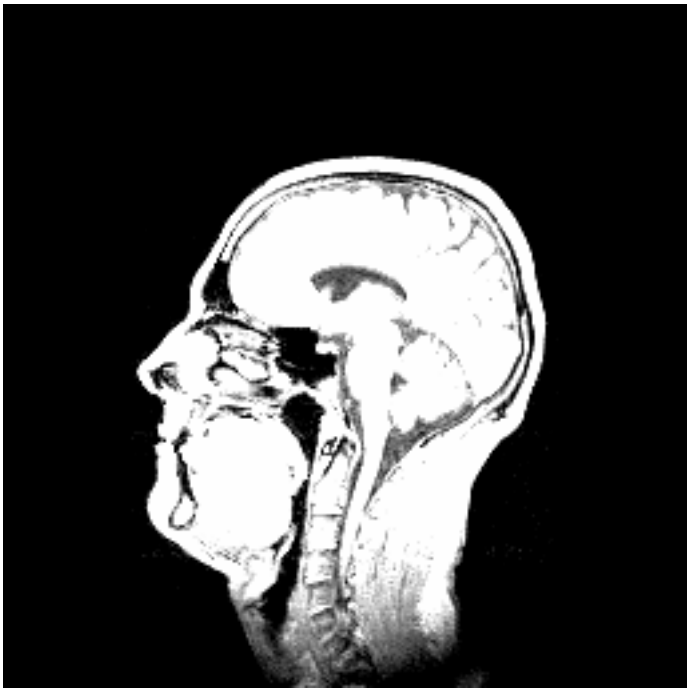

Fig. 30: Saturated output image, input image + 65400. Notice that many regions are saturated to white, corresponding to the maximum value of the pixel data type. Grayscale range is $[65400, 65535]$.

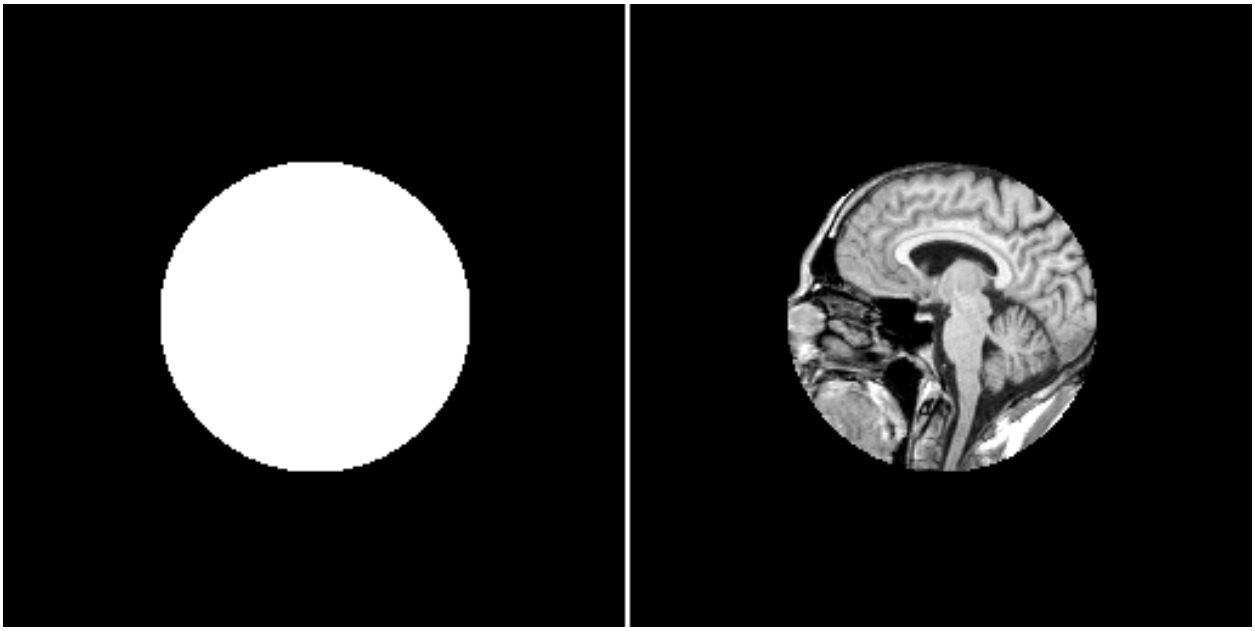

Fig. 31: Mask (left) and masking result (right). In the mask (left), white pixels correspond to value 1 and black pixels to value 0.

```
def skeleton_vtk():
        """
        Creates skeleton, traces it to a graph/network structure, and saves
        the skeleton branches in a .vtk file.
        """

        # Generate a simple image for demonstration
        geometry = pi.newimage(ImageDataType.UINT8, 100, 100)

        p0 = [1, 1, 0]
        p1 = [25, 50, 0]
        p2 = [75, 50, 0]
        p3 = [5, 95, 0]
        p4 = [95, 95, 0]

        pi.line(geometry, p0, p1, 255)
        pi.line(geometry, p1, p2, 255)
        pi.line(geometry, p0, p2, 255)
        pi.line(geometry, p1, p3, 255)
        pi.line(geometry, p2, p4, 255)

        # Save the geometry
        pi.writeraw(geometry, output_file("geometry"))

        # Convert geometry to line skeleton
        pi.lineskeleton(geometry)

        # Write the skeleton so that it can be visualized
        pi.writeraw(geometry, output_file("skeleton"))

        # Create images for tracing the skeleton into a graph
```

(continues on next page)

(continued from previous page)

```
        vertices = pi.newimage(ImageDataType.FLOAT32)
        edges = pi.newimage(ImageDataType.UINT64)
        measurements = pi.newimage(ImageDataType.FLOAT32)
        points = pi.newimage(ImageDataType.INT32)

        # Trace the skeleton
        # The last 1 gives count of threads. For images with small number of branches
        # (like here) single-threaded processing is faster.
        pi.tracelineskeleton(geometry, vertices, edges, measurements, points, True, 1)

        # Convert the traced skeleton into points and lines format, as the .vtk files
        # need that format.
        vtkpoints = pi.newimage()
        vtklines = pi.newimage()
        pi.getpointsandlines(vertices, edges, measurements, points, vtkpoints,␣
↪vtklines)

        pi.writevtk(vtkpoints, vtklines, output_file("vtk_test_file"))
```

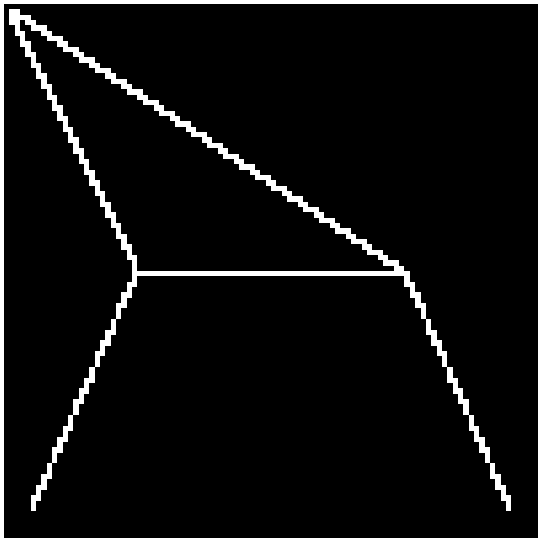

Fig. 32: Input image.

3.7.23 Skeleton types

This examples shows various skeleton types available in pi2. Skeletonization is a process where a binary image is converted to single pixel thick lines or planes that still represent the geometry (and perhaps topology) of the object in the original binary image.

Pi2 supports three types of skeletons.

- The command *surfaceskeleton* with default arguments creates skeletons that consist of 1-pixel thick planes and in some cases the planes may degenerate into 1-pixel thick lines.
- The command *surfaceskeleton* with 'retain surfaces' argument set to false creates skeletons that consist of 1-pixel thick lines where possible. Cavities in the original binary image are still surrounded by a 1-pixel thick plane.
- The command *lineskeleton* creates skeletons that consist strictly of 1-pixel thick lines.

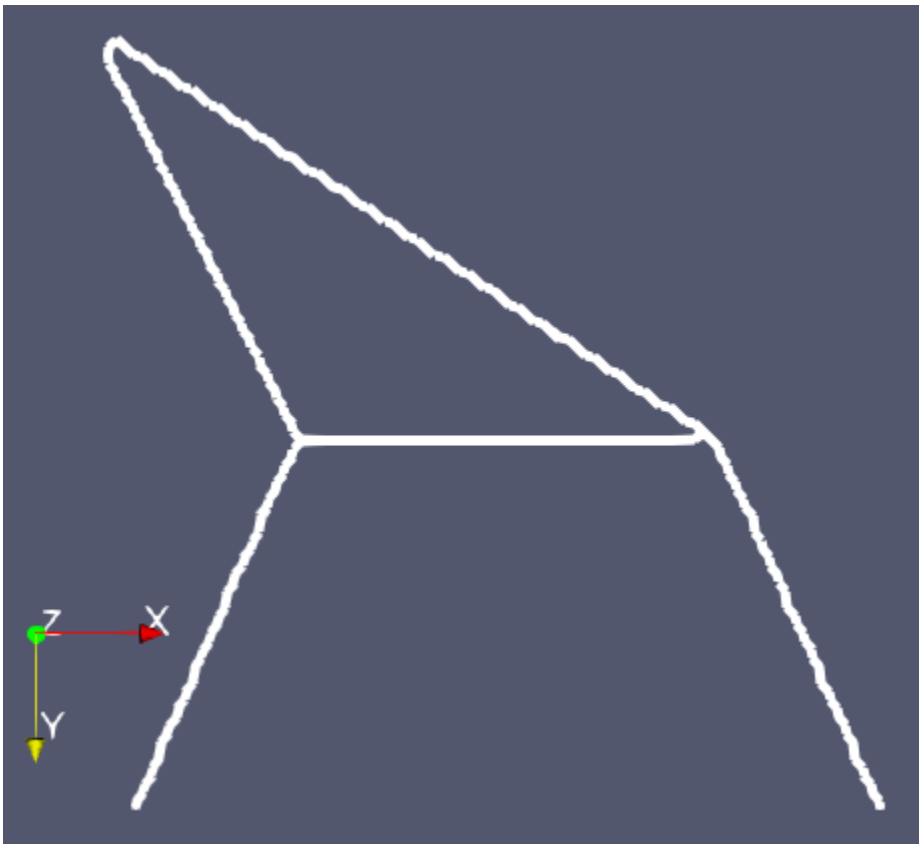

Fig. 33: Traced skeleton visualized in ParaView.

Note that skeletons calculated in distributed processing mode might not be equal to skeletons calculated in normal processing mode. In both cases the skeletons should be correct, though.

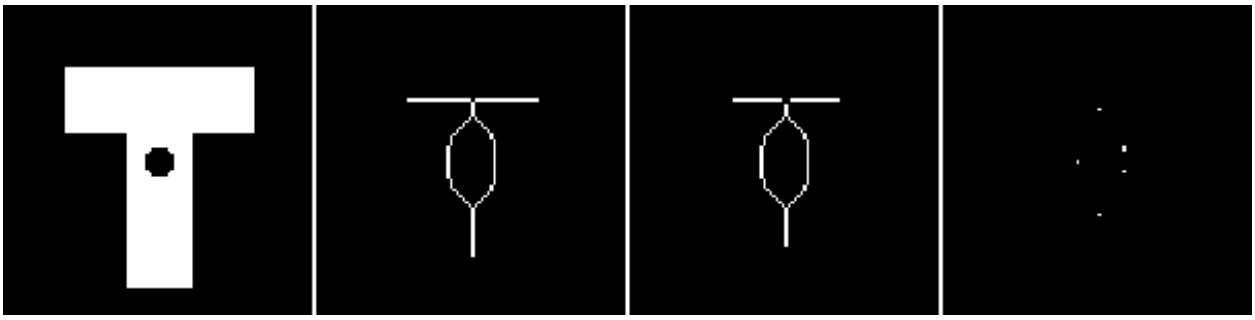

Fig. 34: From left to right: Slice of the input image, slice of a surface skeleton, slice of a surface skeleton with 'retain surfaces' set to false, slice of a line skeleton.

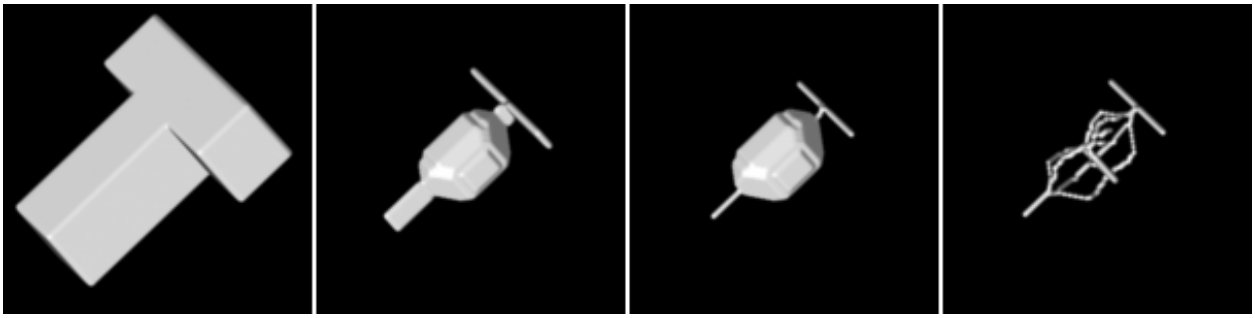

Fig. 35: From left to right: Visualization of the input image, visualization of a surface skeleton, visualization of a surface skeleton with 'retain surfaces' set to false, visualization of a line skeleton.

Code used to create image data for the figures above:

```
def skeleton_types():
        """
        Demonstrates differences between skeletonization algorithms available in pi2.
        """

        # Create a 3D image that contains a thick T letter
        img = pi.newimage(ImageDataType.UINT8, 100, 100, 100)
        pi.box(img, [20, 20, 40], [61, 21, 26], 255)
        pi.box(img, [40, 20, 40], [21, 71, 26], 255)

        # Add a cavity (hole) to the T
        pi.sphere(img, [50, 50, 50], 5, 0)

        # Save the input geometry
        pi.writeraw(img, output_file('T'))

        # Create a skeleton that contains 1-pixel thick planes and 1-pixel thick␣
↪lines.
        surface_skele_true = pi.newlike(img)
        pi.set(surface_skele_true, img)
        pi.surfaceskeleton(surface_skele_true, True)
        pi.writeraw(surface_skele_true, output_file('T_surface_skele_true'))
```

(continues on next page)

(continued from previous page)

```
        # Create a skeleton that contains 1-pixel thick planes only around cavities
        # and 1-pixel thick lines elsewhere.
        surface_skele_false = pi.newlike(img)
        pi.set(surface_skele_false, img)
        pi.surfaceskeleton(surface_skele_false, False)
        pi.writeraw(surface_skele_false, output_file('T_surface_skele_false'))

        # Create a skeleton that contains only 1-pixel thick lines.
        line_skele = pi.newlike(img)
        pi.set(line_skele, img)
        pi.lineskeleton(line_skele)
        pi.writeraw(line_skele, output_file('T_line_skele'))
```

3.7.24 Vessel graph

This example demonstrates generation of a vessel graph structure from a tomographic image of vasculature in mouse brain. The analysis process is similar to that used in [1], where it was used to trace vessels in tomographic images of corrosion cast models of cerebral vasculature.

The analysis process consists of thresholding the original image in order to convert it to binary form, where the vessels are white and everything else is black. The binary image is skeletonized into single pixel wide lines, and the lines are traced into a graph structure. Spurious branches are removed using a custom condition based on the ratio of length and diameter of the branch. The average diameter of the branches is calculated and all the information is saved into a .vtk file for further analysis. The figures below represent the same slice through the tomographic image in various processing phases. Also, 3D visualization of the original image and the traced vessels are provided below.

For running this example code, a demonstration image package available at the GitHub releases page is required.

```
def vessel_tracing():

        # Make binary image where vessels are foreground
        img = pi.read(input_file('orig_basal_ganglia_217'))
        pi.threshold(img, 18000) # The threshold value should be chosen using e.g.␣
↪Otsu method
        pi.closingfilter(img, 1, True, NeighbourhoodType.BOX) # This closes small␣
↪holes in the vessels
        pi.convert(img, ImageDataType.UINT8)
        pi.multiply(img, 255)

        # In order to fill cavities in the vessels, fill the background by a␣
↪temporary color.
        # Here we perform the background fill by a flood fill starting from non-
↪vessel points
        # in the image edge. Here we try to fill from all corners of the image.
        # Note that it is possible that all corners are occupied by vessels and in␣
↪this case
        # this filling method does not work, but in practice that situation is very␣
↪rare.
        fill(img, 0, 0, 0)
        fill(img, img.get_width() - 1, 0, 0)
        fill(img, 0, img.get_height() - 1, 0)
        fill(img, img.get_width() - 1, img.get_height() - 1, 0)
        fill(img, 0, 0, img.get_depth() - 1)
        fill(img, img.get_width() - 1, 0, img.get_depth() - 1)
        fill(img, 0, img.get_height() - 1, img.get_depth() - 1)
```

(continues on next page)

(continued from previous page)

```
        fill(img, img.get_width() - 1, img.get_height() - 1, img.get_depth() - 1)
        # Here the vessels, cavities and background have colors 255, 0, and 128,␣
↪respectively.
        # Set cavities and vessels to 255 and background to 0.
        pi.replace(img, 255, 0)
        pi.replace(img, 0, 255)
        pi.replace(img, 128, 0)

        pi.writeraw(img, output_file('vessel_bin'))

        # Now calculate skeleton. We use the surfaceskeleton function and specify␣
↪False as second
        # argument to indicate that we want a line skeleton. In many cases this␣
↪method makes
        # more stable skeletons than the lineskeleton function.
        pi.surfaceskeleton(img, False)
        pi.writeraw(img, output_file('vessel_skele'))

        # Below we will need the distance map of vessel phase, so we calculate it␣
↪here.
        img = pi.read(output_file('vessel_bin'))
        dmap = pi.newimage(ImageDataType.FLOAT32)
        pi.dmap(img, dmap)
        pi.writeraw(dmap, output_file('vessel_dmap'))
        dmap_data = dmap.get_data()


        # Trace skeleton
        skeleton = pi.read(output_file('vessel_skele'))
        smoothing_sigma = 2
        max_displacement = 2
        vertices = pi.newimage(ImageDataType.FLOAT32)
        edges = pi.newimage(ImageDataType.UINT64)
        measurements = pi.newimage(ImageDataType.FLOAT32)
        points = pi.newimage(ImageDataType.INT32)
        pi.tracelineskeleton(skeleton, vertices, edges, measurements, points, True, 1,
↪ smoothing_sigma, max_displacement)


        # Next, we will remove all edges that has at least one free and and whose
        # L/r < 2.
        # First, get edges, vertices, and branch length as NumPy arrays.
        old_edges = edges.get_data()
        vert_coords = vertices.get_data()

        # The tracelineskeleton measures branch length by anchored convolution and␣
↪returns it in the
        # measurements image.
        meas_data = measurements.get_data()
        length_data = meas_data[:, 1]


        # Calculate degree of each vertex
        deg = {}
        for i in range(0, vert_coords.shape[0]):
                deg[i] = 0
```

(continues on next page)

(continued from previous page)

```
        for i in range(0, old_edges.shape[0]):
                deg[old_edges[i, 0]] += 1
                deg[old_edges[i, 1]] += 1

        # Determine which edges should be removed
        remove_flags = []
        for i in range(0, old_edges.shape[0]):
                n1 = old_edges[i, 0]
                n2 = old_edges[i, 1]

                # Remove edge if it has at least one free end point, and if L/r < 2,␣
↪where
                # r = max(r_1, r_2) and r_1 and r_2 are radii at the end points or␣
↪the edge.
                should_remove = False
                if deg[n1] == 1 or deg[n2] == 1:

                        p1 = vert_coords[n1, :]
                        p2 = vert_coords[n2, :]

                        r1 = dmap_data[int(p1[1]), int(p1[0]), int(p1[2])]
                        r2 = dmap_data[int(p2[1]), int(p2[0]), int(p2[2])]

                        r = max(r1, r2)
                        L = length_data[i]

                        if L < 2 * r:
                                should_remove = True

                # Remove very short isolated branches, too.
                if deg[n1] == 1 and deg[n2] == 1:
                        L = length_data[i]
                        if L < 5 / 0.75: # (5 um) / (0.75 um/pixel)
                                should_remove = True

                remove_flags.append(should_remove)

        remove_flags = np.array(remove_flags).astype(np.uint8)
        print(f"Before dynamic pruning: {old_edges.shape[0]} edges")
        print(f"Removing {np.sum(remove_flags)} edges")

        # This call adjusts the vertices, edges, and measurements images such that
        # the edges for which remove_flags entry is True are removed from the graph.
        pi.removeedges(vertices, edges, measurements, points, remove_flags, True,␣
↪True)



        # Convert to vtk format in order to get radius for each point and line
        vtkpoints = pi.newimage()
        vtklines = pi.newimage()
        pi.getpointsandlines(vertices, edges, measurements, points, vtkpoints,␣
↪vtklines)

        # Get radius for each point
        points_data = vtkpoints.get_data()
        radius_points = np.zeros([points_data.shape[0]])
```

(continues on next page)

(continued from previous page)

```
        for i in range(0, points_data.shape[0]):
                p = points_data[i, :]
                r = dmap_data[int(p[1]), int(p[0]), int(p[2])]
                radius_points[i] = r


        # Get average radius for each branch
        # Notice that the vtklines image has a special format that is detailed in
        # the documentation of getpointsandlines function.
        lines_data = vtklines.get_data()
        radius_lines = []
        i = 0
        edge_count = lines_data[i]
        i += 1
        for k in range(0, edge_count):
                count = lines_data[i]
                i += 1

                R = 0
                for n in range(0, count):
                        index = lines_data[i]
                        i += 1
                        p = points_data[index, :]
                        R += dmap_data[int(p[1]), int(p[0]), int(p[2])]
                R /= count

                radius_lines.append(R)

        radius_lines = np.array(radius_lines)


        # Convert to vtk format again, now with smoothing the point coordinates to␣
↪get non-jagged branches.
        vtkpoints = pi.newimage()
        vtklines = pi.newimage()
        pi.getpointsandlines(vertices, edges, measurements, points, vtkpoints,␣
↪vtklines, smoothing_sigma, max_displacement)


        # Write to file
        pi.writevtk(vtkpoints, vtklines, output_file('vessels'), "radius", radius_
↪points, "radius", radius_lines)
```

References

[1] Wälchli T., Bisschop J., Miettinen A. et al. Hierarchical imaging and computational analysis of three-dimensional vascular network architecture in the entire postnatal and adult mouse brain. Nature Protocols, 2021.

3.7.25 Watershed segmentation

This example demonstrates how to perform watershed segmentation.

In pi2, you will have to first generate seeds image that contains colored regions that will be *grown* in the watershed process. The filling priority of pixels must also be calculated somehow, *usually* this is the *gradient* of the input image, but in this example we use the input image itself.

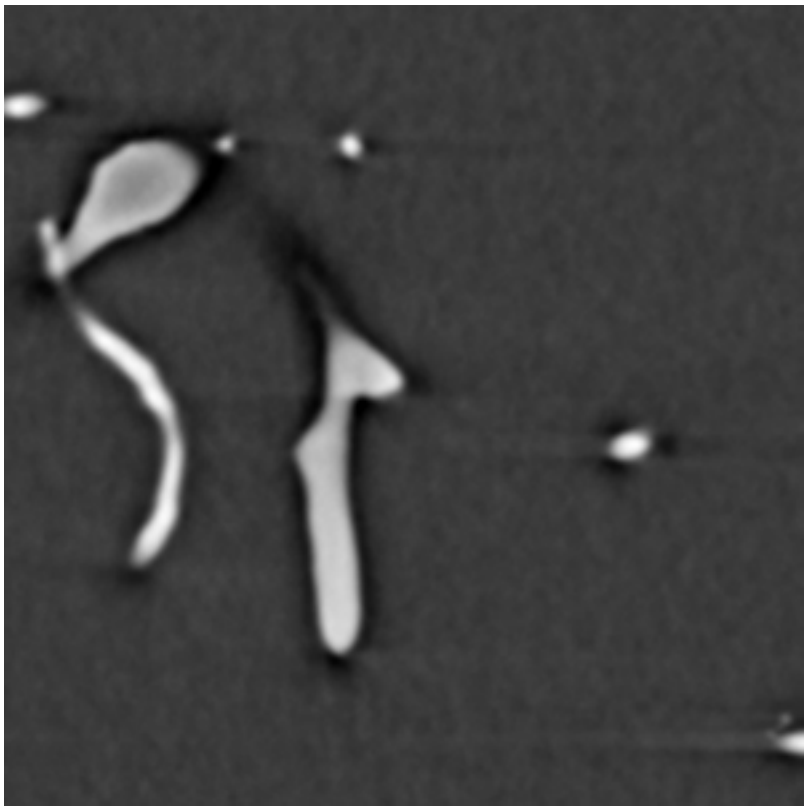

Fig. 36: Input image. The bright regions are corrosion-cast blood vessels.

Fig. 37: Segmented vessels.

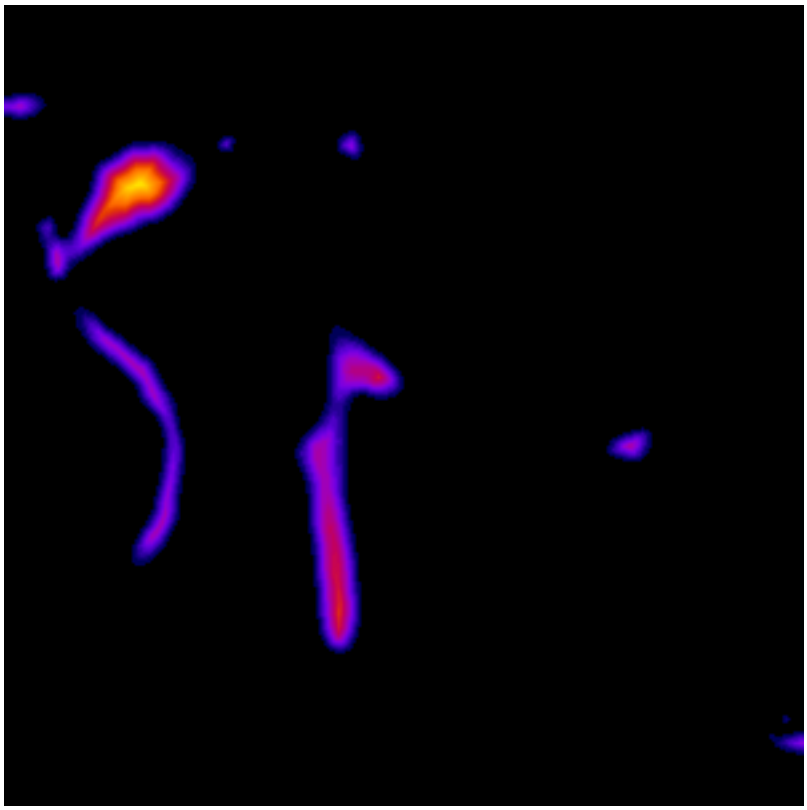

Fig. 38: Distance map of the segmented vessels.

Fig. 39: Skeleton of the segmented vessels.

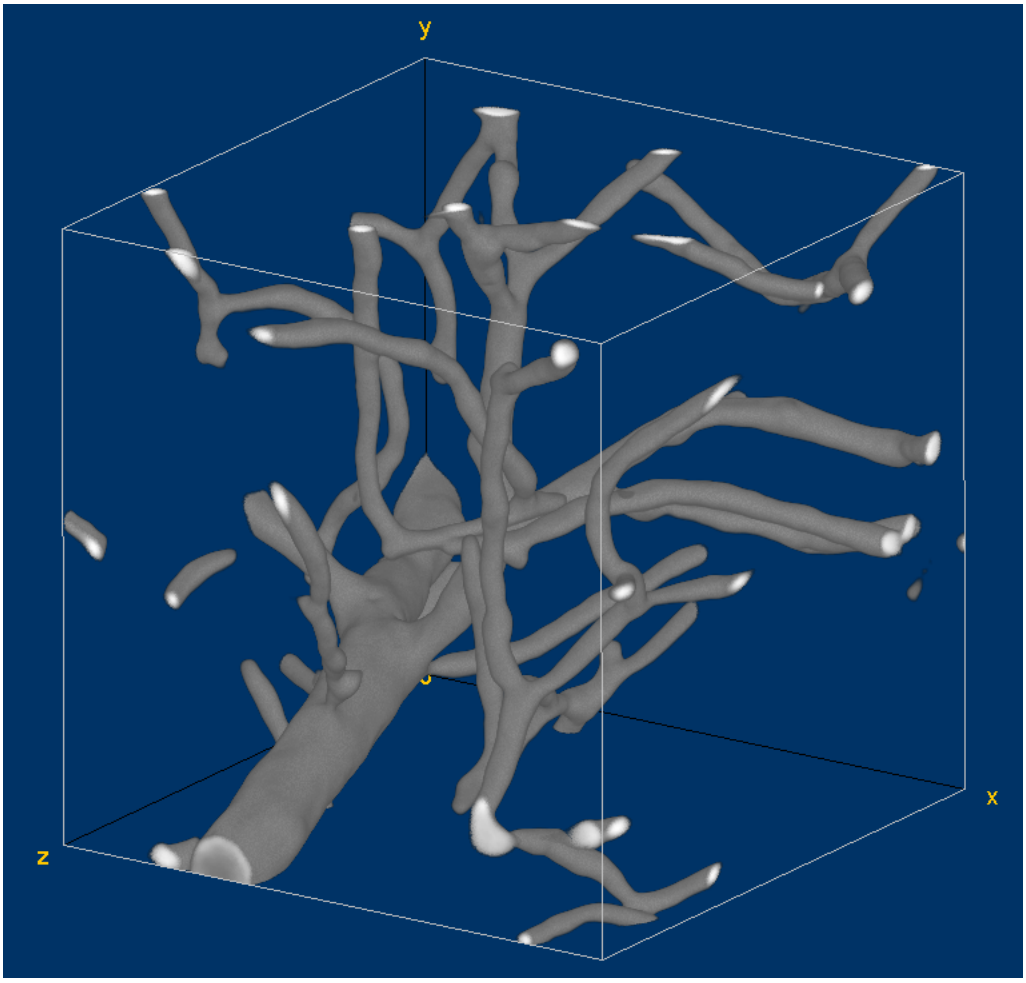

Fig. 40: Volume visualization of the original vessel image. This visualization has been generated with the Volume Viewer plugin of Fiji.

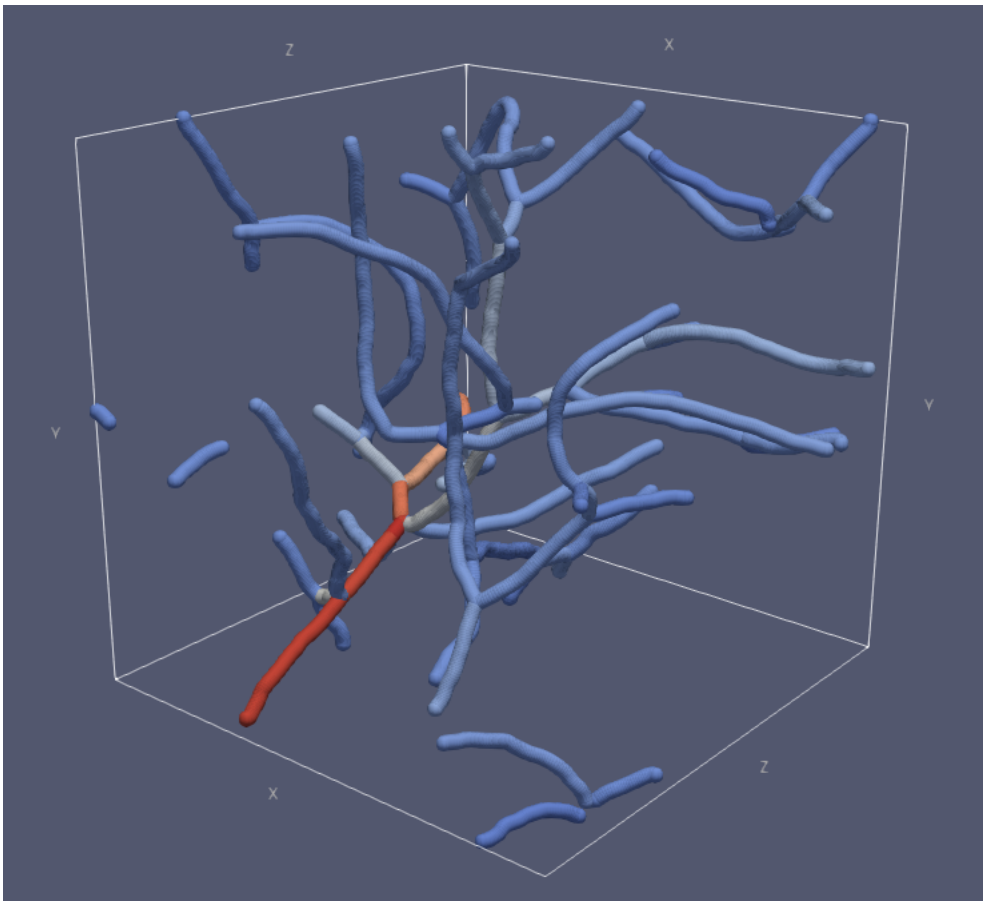

Fig. 41: Visualization of the traced vessels. Each vessel branch has been colored based on the average diameter of that branch. Bright colors correspond to large diameters. This visualization has been generated with ParaView.

```
def watershed():
        """
        Demonstrates Meyer's flooding algorithm for calculation of watershed.
        """

        # Read image
        weights = pi.read(input_file('t1-head.tif'))

        # Create new image, taking unspecified properties from the old image
        labels = pi.newlike(weights, 'uint8')

        # Set some seeds
        pi.set(labels, [110, 90, 63], 100) # Brain
        pi.set(labels, [182, 165, 63], 200) # Skull

        # Save seeds so that they can be viewed later
        pi.writetif(labels, output_file('meyer_grow_seeds'))

        # Grow the seeds. (Normally you would use gradient of input image etc. as␣
↪weight)
        pi.grow(labels, weights)

        # Save result
        pi.writetif(labels, output_file('meyer_grow'))
```

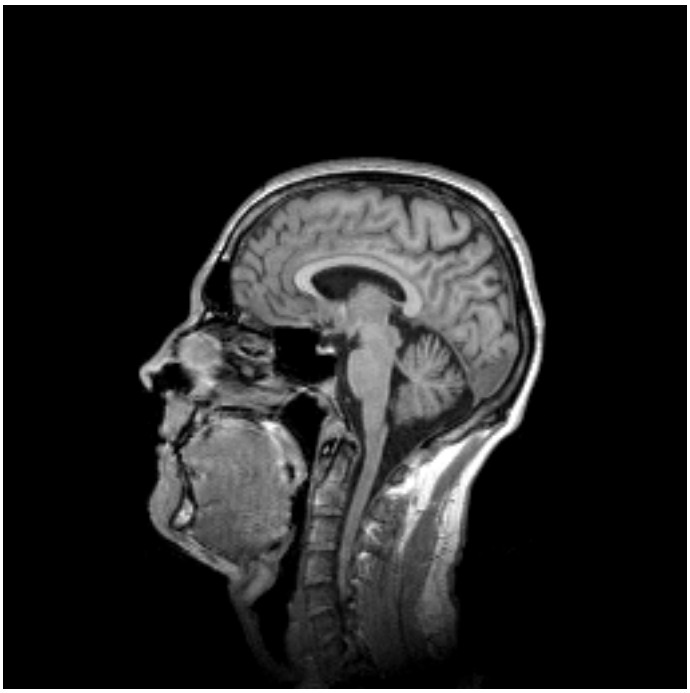

Fig. 42: One slice of the input image.

3.7.26 Watershed segmentation of particles

This example shows how to make a watershed segmentation where seeds come from local maxima of distance map. The maxima may be filtered to avoid over-segmentation.

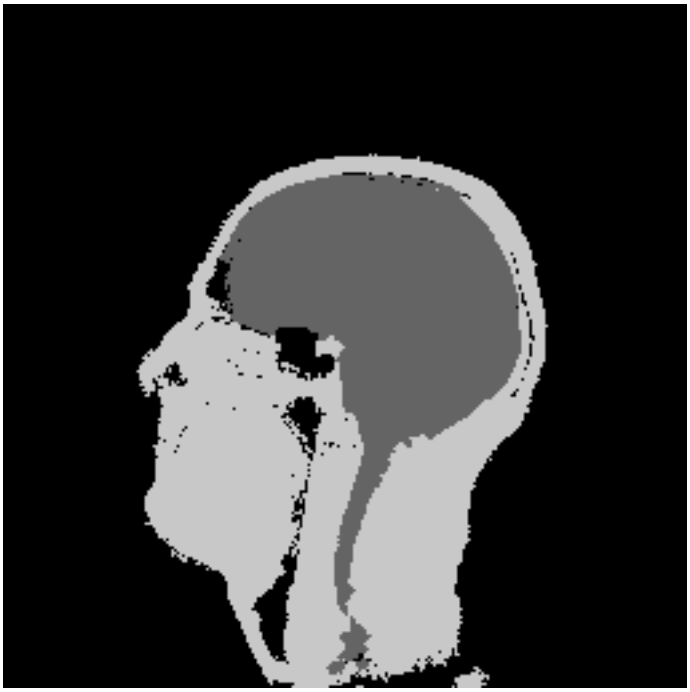

Fig. 43: Result of watershed segmentation.

The image data is generated similarly to what was done in *particle analysis example*.

```
def particle_segmentation():
        """
        Demonstrates segmentation of particles using watershed seeded
        by (filtered) local maxima of distance map.
        """

        # Generate particle image
        generate_particles(1000, 0)


        # Read generated data file
        img = pi.read(output_file('particles'))

        # Convert to wider data type so that we can label each particle
        pi.convert(img, ImageDataType.UINT16)

        # Calculate distance map
        dmap = pi.newimage(ImageDataType.FLOAT32)
        pi.dmap(img, dmap)

        # Find maxima
        # Note that in some cases the system is able to automatically
        # change the data type of output images, so we don't have to
        # specify any data type in the pi.newimage() command.
        # This does not work always, though, as there might be many
        # possible output data types.
        maxima = pi.newimage()
        pi.localmaxima(dmap, maxima)
```

(continues on next page)

(continued from previous page)

```
        # Remove unnecessary maxima to avoid over-segmentation
        pi.cleanmaxima(dmap, maxima)

        # Create image with labeled maxima only.
        # First set all pixels to zero, then label maxima.
        pi.set(img, 0)
        pi.labelmaxima(img, maxima)

        # Grow labels back to original geometry, using distance map value
        # as filling priority.
        pi.grow(img, dmap)

        # Save result
        pi.writeraw(img, output_file('particles_watershed'))
```

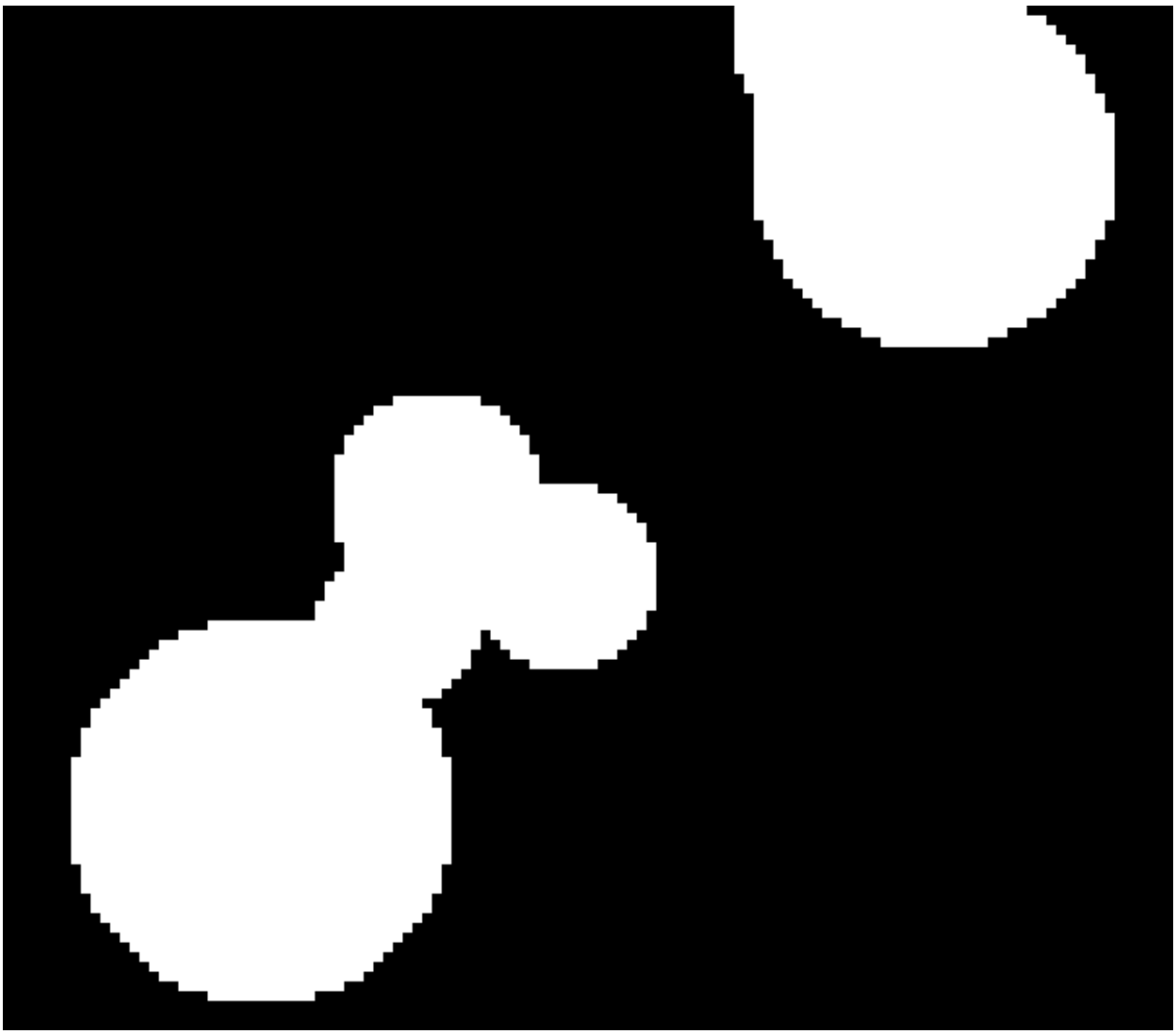

Fig. 44: One slice of the input image.

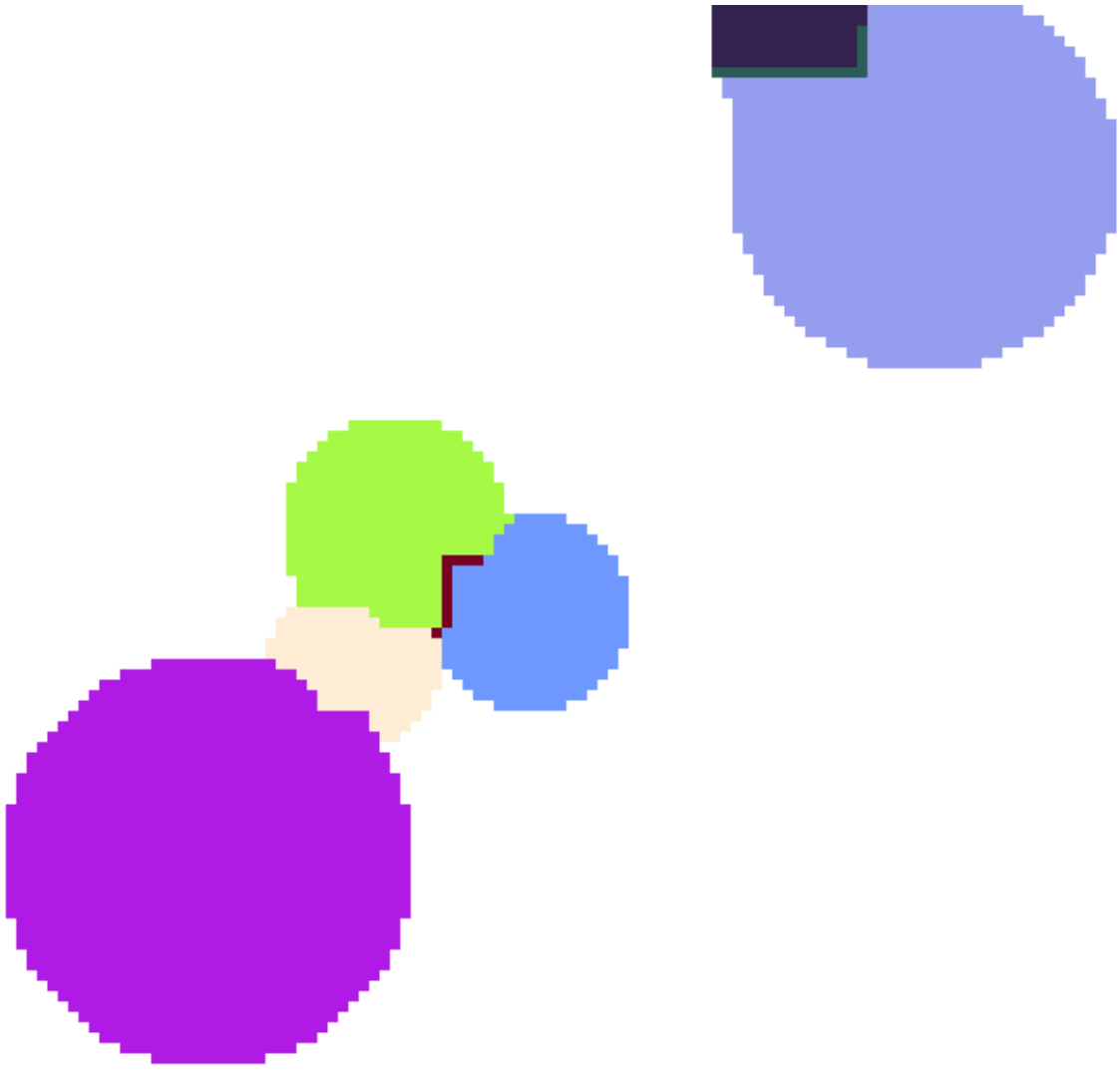

Fig. 45: Result of watershed segmentation without using *cleanmaxima*. Random colors have been assigned to the segmented regions. Notice extraneous regions between some of the spheres.

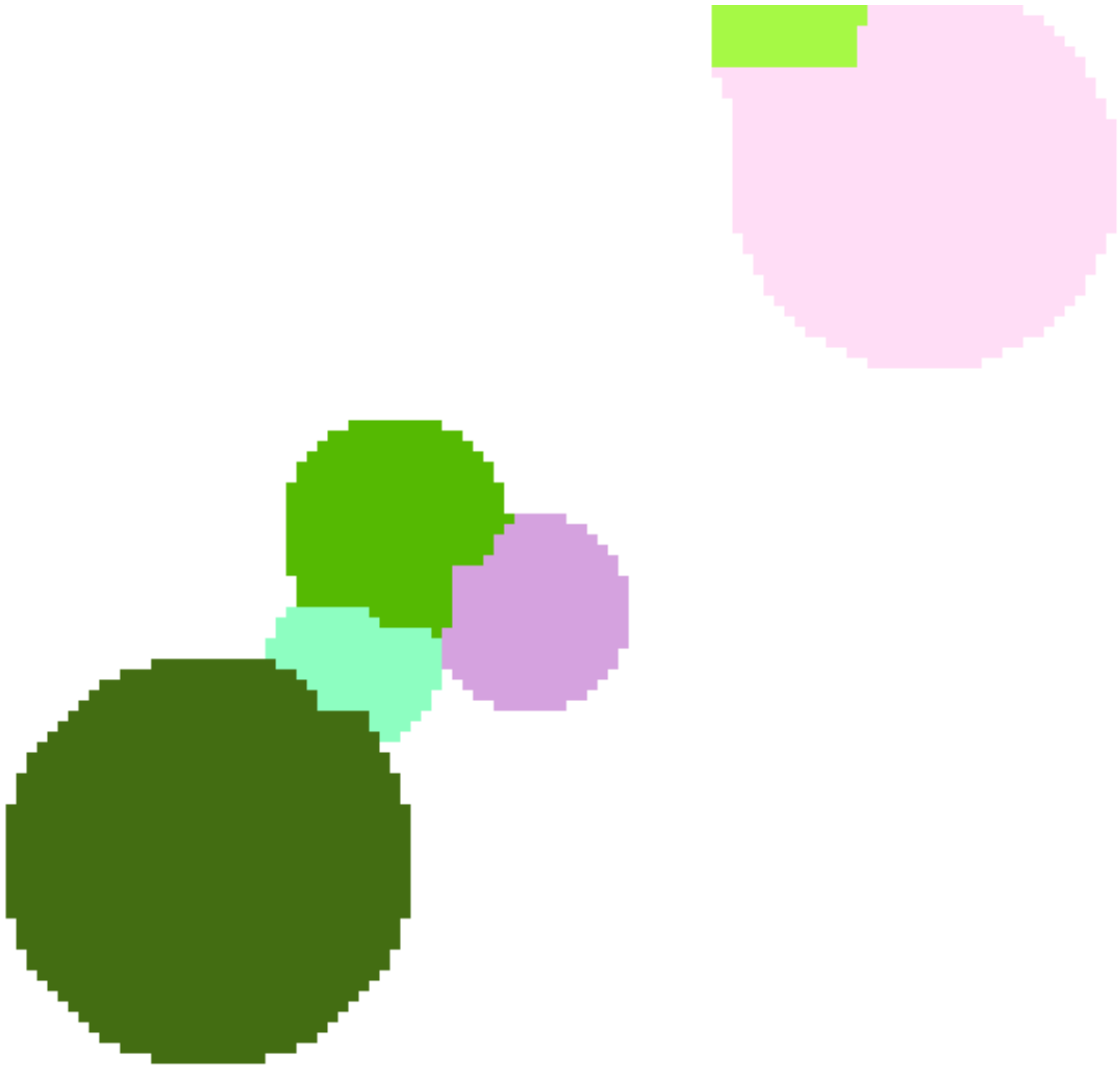

Fig. 46: Result of watershed segmentation after using *cleanmaxima*. Random colors have been assigned to the segmented regions. There are no extraneous regions between the spheres as in the previous figure.

3.8 Command reference

This section contains documentation for each command available in the pi2 system. The same information is also available using commands:

```
pi2 help
pi2 help(command_name)
```

where `command_name` is the name of the command whose help is to be retrieved.

If using pi2 from ipython, the help is available using the standard TAB and ?-help functionality.

3.8.1 abs

Syntax: `abs(image)`

Calculates absolute value of pixel values.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to process.

3.8.2 add

There are 2 forms of this command.

`add(image, parameter image, allow broadcast)`

Adds two images. Output is placed to the first image. The operation is performed using saturation arithmetic.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to process.

parameter image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image, complex32 image

Parameter image.

allow broadcast [input]

Data type: boolean

Default value: False

Set to true to allow size of parameter image differ from size of input image. If there is a need to access pixel outside of parameter image, the nearest value inside the image is taken instead. If set to false, dimensions of input and parameter images must be equal. If set to true, the parameter image is always loaded in its entirety in distributed processing mode.

`add(image, x)`

Adds and image and a constant. The operation is performed using saturation arithmetic.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

x [input]

Data type: real

Constant to add to the image.

3.8.3 analyzelabels

Syntax: `analyzelabels(image, results, analyzers)`

Analyzes labeled regions of the input image. The regions do not need to be connected. Region having value zero is skipped.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

The input image where each particle is labeled with different color.

results [output]

Data type: float32 image

Analysis results image.

analyzers [input]

Data type: string

Default value: "coordinates, volume"

List of names of analyzers to use. Use command *listanalyzers* to see all the names that can be specified. Separate the analyzer names with any non-alphanumeric character sequence.

See also

analyzeparticles, *listanalyzers*, *headers*, *fillparticles*, *drawellipsoids*, *label*, *analyzelabels*, *regionremoval*, *greedycoloring*, *csa*

3.8.4 analyzeparticles

Syntax: `analyzeparticles(input image, results, analyzers, connectivity, volume limit, single-threaded)`

Analyzes shape of blobs or other particles (separate nonzero regions) in the input image. Assumes all the particles have the same color. All the nonzero pixels in the input image will be set to same value. Output image will contain results of the measurements. There will be one row for each particle found in the input image. Use command *headers* to get interpretation of the columns. The order of the particles in the results may be different in normal and distributed processing modes. If you wish to analyze labeled particles, see *analyzelabels*.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image. The particles in this image will be filled with temporary color.

results [output]

Data type: float32 image

Image where analysis results are placed. This image will contain one row for each particle found in the input image. Use command *headers* to retrieve meanings of columns.

analyzers [input]

Data type: string

Default value: "coordinates, volume"

List of names of analyzers to use. Use command *listanalyzers* to see all the names that can be specified. Separate the analyzer names with any non-alphanumeric character sequence.

connectivity [input]

Data type: connectivity

Default value: Nearest

Connectivity of the particles. Can be Nearest for connectivity to nearest neighbours only, or All for connectivity to all neighbours.

volume limit [input]

Data type: positive integer

Default value: 0

Maximum size of particles to consider, in pixels. Specify zero to consider all particles.

single-threaded [input]

Data type: boolean

Default value: False

Set to true to use single-threaded processing. That might be faster for some cases. Does not have any effect if distributed processing is enabled.

See also

analyzeparticles, *listanalyzers*, *headers*, *fillparticles*, *drawellipsoids*, *label*, *analyzelabels*, *regionremoval*, *greedycoloring*, *csa*

3.8.5 arg

Syntax: `arg(image)`

Calculates argument of pixel values.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: complex32 image

Image to process.

3.8.6 autothreshold

Syntax: `autothreshold(image, method, argument 1, argument 2, argument 3, argument 4)`

Thresholds image by automatically determined threshold value. The supported thresholding methods are

Otsu

Otsu's thresholding algorithm.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Described in N. Otsu, A Threshold Selection Method from Gray-Level Histograms, IEEE Transactions on Systems, Man, and Cybernetics 9(1), 1979.

Original C++ code by Jordan Bevik, ported to ImageJ plugin by Gabriel Landini, and finally ported to pi2.

Huang

Huang's fuzzy thresholding method.

Uses Shannon's entropy function (one can also use Yager's entropy function).

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Described in L.-K. Huang and M.-J.J. Wang, Image Thresholding by Minimizing the Measures of Fuzziness, Pattern Recognition 28(1), 1995.

Original code by M. Emre Celebi, ported to ImageJ plugin by G. Landini from E. Celebi's fourier_0.8 routines, then ported to pi2.

Intermodes

Assumes a bimodal histogram. The histogram needs is iteratively smoothed using a running average of size 3 until there are only two local maxima at j and k. The threshold t is (j+k)/2.

Images with histograms having extremely unequal peaks or a broad and flat valleys are unsuitable for this method.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Described in J. M. S. Prewitt and M. L. Mendelsohn, The analysis of cell images, Annals of the New York Academy of Sciences 128, 1966.

Ported to ImageJ plugin by Gabriel Landini from Antti Niemisto's Matlab code (GPL), and then to pi2.

IsoData

Iterative procedure based on the isodata algorithm described in T.W. Ridler and S. Calvard, Picture thresholding using an iterative selection method, IEEE Transactions on System, Man and Cybernetics, SMC-8, 1978.

The procedure divides the image into objects and background by taking an initial threshold, then the averages of the pixels at or below the threshold and pixels above are computed. The averages of those two values are computed, the threshold is incremented and the process is repeated until the threshold is larger than the composite average. That is, threshold = (average background + average objects) / 2

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

The code implementing this method is probably originally from NIH Image, then ported to ImageJ, and then to pi2.

Li

Li's Minimum Cross Entropy thresholding method.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

This implementation is based on the iterative version of the algorithm, described in C.H. Li and P.K.S. Tam, An Iterative Algorithm for Minimum Cross Entropy Thresholding, Pattern Recognition Letters 18(8), 1998.

Ported to ImageJ plugin by G.Landini from E Celebi's fourier_0.8 routines, and then to pi2.

MaxEntropy

Implements Kapur-Sahoo-Wong (Maximum Entropy) thresholding method described in J.N. Kapur, P.K. Sahoo, and A.K.C Wong, A New Method for Gray-Level Picture Thresholding Using the Entropy of the Histogram, Graphical Models and Image Processing 29(3), 1985.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Original code by M. Emre Celebi, ported to ImageJ plugin by G.Landini, then ported to pi2.

Mean

The threshold is shifted mean of the greyscale data, i.e. $t = \mu - c$, where t is the threshold, μ is the mean of the image or the neighbourhood, and c is the shift.

First argument is the shift value c.

Described in C. A. Glasbey, An analysis of histogram-based thresholding algorithms, CVGIP: Graphical Models and Image Processing 55, 1993.

MinError

Implements minimum error thresholding method described in J. Kittler and J. Illingworth, Minimum error thresholding, Pattern Recognition 19, 1986.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Code is originally from Antti Niemisto's Matlab code (GPL), then ported to ImageJ by Gabriel Landini, and then to pi2.

Minimum

Assumes a bimodal histogram. The histogram needs is iteratively smoothed (using a running average of size 3) until there are only two local maxima. Threshold t is such that $y_{t-1} > y_t$ and $y_t \leq y_{t+1}$. Images with histograms having extremely unequal peaks or a broad and flat valleys are unsuitable for this method.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Described in J. M. S. Prewitt and M. L. Mendelsohn, The analysis of cell images, Annals of the New York Academy of Sciences 128, 1966.

Original Matlab code by Antti Niemisto, ported to ImageJ by Gabriel Landini, then ported to pi2.

Moments

Attempts to preserve the moments of the original image in the thresholded result.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Described in W. Tsai, Moment-preserving thresholding: a new approach, Computer Vision, Graphics, and Image Processing 29, 1985.

Original code by M. Emre Celebi ported to ImageJ by Gabriel Landini, and then to pi2.

Percentile

Assumes the fraction of foreground pixels to be given value.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram. The fourth argument is the fraction of foreground pixels.

Described in W. Doyle, Operation useful for similarity-invariant pattern recognition, Journal of the Association for Computing Machinery 9, 1962.

Original code by Antti Niemisto, ported to ImageJ by Gabriel Landini, then to pi2.

RenyiEntropy

Similar to the MaxEntropy method, but using Renyi's entropy instead.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Described in J.N. Kapur, P.K. Sahoo, and A.K.C Wong, A New Method for Gray-Level Picture Thresholding Using the Entropy of the Histogram, Graphical Models and Image Processing 29(3), 1985.

Original code by M. Emre Celebi, ported to ImageJ plugin by G.Landini, then ported to pi2.

Shanbhag

Described in A.G. Shanhbag, Utilization of Information Measure as a Means of Image Thresholding, Graphical Models and Image Processing 56(5), 1994.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Original code by M. Emre Celebi, ported to ImageJ plugin by Gabriel Landini, then to pi2.

Triangle

The triangle algorithm assumes a peak near either end of the histogram, finds minimum near the other end, draws a line between the minimum and maximum, and sets threshold t to a value for which the point $(t, y(t))$ is furthest away from the line (where y is the histogram).

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Described in G.W. Zack, W.E. Rogers, and S.A. Latt, Automatic Measurement of Sister Chromatid Exchange Frequency, Journal of Histochemistry and Cytochemistry 25(7), 1977.

Original code by Johannes Schindelin, modified by Gabriel Landini, then ported to pi2.

Yen

Yen thresholding method described in J.C. Yen, F.K. Chang, and S. Chang, A New Criterion for Automatic Multilevel Thresholding, IEEE Transactions on Image Processing 4(3), 1995.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Original code by M. Emre Celebi, ported to ImageJ plugin by Gabriel Landini, then to pi2.

Median

The threshold is shifted median of the greyscale data, i.e. $t = M - c$, where t is the threshold, M is the median of the image, and c is the shift.

The median is calculated from image histogram, so its accuracy might degrade if too small bin count is used. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram. The fourth argument is the shift value c.

MidGrey

The threshold is $(m + M)/2 - c$, where m and M are the minimum and the maximum value of the image or neighbourhood, respectively. The value c is a user-specified shift.

The first argument is the shift value c.

Niblack

Niblack's thresholding method.

This method is mostly suited for local thresholding.

The threshold is $\mu + k * \sigma - c$, where μ and σ are the mean and the standard deviation of the image or the neighbourhood, respectively. The values k and c are user-specified scaling constant and shift.

The first argument is the scaling constant k. The second argument is the shift value c.

Described in W. Niblack, An introduction to Digital Image Processing, Prentice-Hall, 1986.

Phansalkar

Phansalkar's thresholding method.

This method is mostly suited for local thresholding.

Threshold is $\mu * (1.0 + p * \exp(-q * \mu) + k * (\sigma/r - 1))$ where μ and σ are the mean and the standard deviation of the image or the neighbourhood, respectively. The values k, r, p, and q are user-specified parameters of the method. The default values are $k = 0.25$, $r = 0.5$, $p = 2.0$, and $q = 10.0$.

The first four arguments are the parameters k, r, p, and q.

Described in N. Phansalskar, S. More, and A. Sabale, et al., Adaptive local thresholding for detection of nuclei in diversity stained cytology images, International Conference on Communications and Signal Processing (ICCSP), 2011.

Sauvola

Sauvola's thresholding method.

This method is mostly suited for local thresholding.

The threshold is $\mu * (1 + k * (\sigma/r - 1))$, where μ and σ are the mean and the standard deviation of the image or the neighbourhood, respectively. The values k and r are user-specified scaling constants.

The first argument is the scaling constant k. The second argument is the scaling constant r.

Described in Sauvola, J and Pietaksinen, M, Adaptive Document Image Binarization, Pattern Recognition 33(2), 2000.

Bernsen

Finds Bernsen's thresholding method.

This method is mostly suited for local thresholding.

The method uses a user-provided contrast threshold. If the local contrast (max - min) is above or equal to the contrast threshold, the threshold is set at the local midgrey value (the mean of the minimum and maximum grey values in the local window (or whole image in the case of global thresholding)). If the local contrast is below the contrast threshold the neighbourhood is considered to consist only of one class and the pixel is set to object or background depending on the value of the midgrey.

The first argument is the local contrast threshold.

Described in J. Bernsen, Dynamic Thresholding of Grey-Level Images, Proceedings of the 8th International Conference on Pattern Recognition, 1986.

The original code is written by Gabriel Landini, and it has been ported to pi2.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

method [input]

Data type: string

Default value: Otsu

Thresholding method that will be applied.

argument 1 [input]

Data type: real

Default value: nan

Argument for the thresholding method. The purpose of this argument depends on the method, see the list above. Specify nan in order to use a method-specific default value.

argument 2 [input]

Data type: real

Default value: nan

Argument for the thresholding method. The purpose of this argument depends on the method, see the list above. Specify nan in order to use a method-specific default value.

argument 3 [input]

Data type: real

Default value: nan

Argument for the thresholding method. The purpose of this argument depends on the method, see the list above. Specify nan in order to use a method-specific default value.

argument 4 [input]

Data type: real

Default value: nan

Argument for the thresholding method. The purpose of this argument depends on the method, see the list above. Specify nan in order to use a method-specific default value.

See also

localthreshold, *threshold*

3.8.7 autothresholdvalue

Syntax: `autothresholdvalue(input image, output image, method, argument 1, argument 2, argument 3, argument 4)`

Calculates threshold value for the input image, according to a selected thresholding method, and places it into the output image. The supported thresholding methods are

Otsu

Otsu's thresholding algorithm.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Described in N. Otsu, A Threshold Selection Method from Gray-Level Histograms, IEEE Transactions on Systems, Man, and Cybernetics 9(1), 1979.

Original C++ code by Jordan Bevik, ported to ImageJ plugin by Gabriel Landini, and finally ported to pi2.

Huang

Huang's fuzzy thresholding method.

Uses Shannon's entropy function (one can also use Yager's entropy function).

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Described in L.-K. Huang and M.-J.J. Wang, Image Thresholding by Minimizing the Measures of Fuzziness, Pattern Recognition 28(1), 1995.

Original code by M. Emre Celebi, ported to ImageJ plugin by G. Landini from E. Celebi's fourier_0.8 routines, then ported to pi2.

Intermodes

Assumes a bimodal histogram. The histogram needs is iteratively smoothed using a running average of size 3 until there are only two local maxima at j and k. The threshold t is (j+k)/2.

Images with histograms having extremely unequal peaks or a broad and flat valleys are unsuitable for this method.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Described in J. M. S. Prewitt and M. L. Mendelsohn, The analysis of cell images, Annals of the New York Academy of Sciences 128, 1966.

Ported to ImageJ plugin by Gabriel Landini from Antti Niemisto's Matlab code (GPL), and then to pi2.

IsoData

Iterative procedure based on the isodata algorithm described in T.W. Ridler and S. Calvard, Picture thresholding using an iterative selection method, IEEE Transactions on System, Man and Cybernetics, SMC-8, 1978.

The procedure divides the image into objects and background by taking an initial threshold, then the averages of the pixels at or below the threshold and pixels above are computed. The averages of those two values are computed, the threshold is incremented and the process is repeated until the threshold is larger than the composite average. That is, threshold = (average background + average objects) / 2

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

The code implementing this method is probably originally from NIH Image, then ported to ImageJ, and then to pi2.

Li

Li's Minimum Cross Entropy thresholding method.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

This implementation is based on the iterative version of the algorithm, described in C.H. Li and P.K.S. Tam, An Iterative Algorithm for Minimum Cross Entropy Thresholding, Pattern Recognition Letters 18(8), 1998.

Ported to ImageJ plugin by G.Landini from E Celebi's fourier_0.8 routines, and then to pi2.

MaxEntropy

Implements Kapur-Sahoo-Wong (Maximum Entropy) thresholding method described in J.N. Kapur, P.K. Sahoo, and A.K.C Wong, A New Method for Gray-Level Picture Thresholding Using the Entropy of the Histogram, Graphical Models and Image Processing 29(3), 1985.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Original code by M. Emre Celebi, ported to ImageJ plugin by G.Landini, then ported to pi2.

Mean

The threshold is shifted mean of the greyscale data, i.e. $t = \mu - c$, where t is the threshold, μ is the mean of the image or the neighbourhood, and c is the shift.

First argument is the shift value c.

Described in C. A. Glasbey, An analysis of histogram-based thresholding algorithms, CVGIP: Graphical Models and Image Processing 55, 1993.

MinError

Implements minimum error thresholding method described in J. Kittler and J. Illingworth, Minimum error thresholding, Pattern Recognition 19, 1986.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Code is originally from Antti Niemisto's Matlab code (GPL), then ported to ImageJ by Gabriel Landini, and then to pi2.

Minimum

Assumes a bimodal histogram. The histogram needs is iteratively smoothed (using a running average of size 3) until there are only two local maxima. Threshold t is such that $y_{t-1} > y_t$ and $y_t \leq y_{t+1}$. Images with histograms having extremely unequal peaks or a broad and flat valleys are unsuitable for this method.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Described in J. M. S. Prewitt and M. L. Mendelsohn, The analysis of cell images, Annals of the New York Academy of Sciences 128, 1966.

Original Matlab code by Antti Niemisto, ported to ImageJ by Gabriel Landini, then ported to pi2.

Moments

Attempts to preserve the moments of the original image in the thresholded result.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Described in W. Tsai, Moment-preserving thresholding: a new approach, Computer Vision, Graphics, and Image Processing 29, 1985.

Original code by M. Emre Celebi ported to ImageJ by Gabriel Landini, and then to pi2.

Percentile

Assumes the fraction of foreground pixels to be given value.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram. The fourth argument is the fraction of foreground pixels.

Described in W. Doyle, Operation useful for similarity-invariant pattern recognition, Journal of the Association for Computing Machinery 9, 1962.

Original code by Antti Niemisto, ported to ImageJ by Gabriel Landini, then to pi2.

RenyiEntropy

Similar to the MaxEntropy method, but using Renyi's entropy instead.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Described in J.N. Kapur, P.K. Sahoo, and A.K.C Wong, A New Method for Gray-Level Picture Thresholding Using the Entropy of the Histogram, Graphical Models and Image Processing 29(3), 1985.

Original code by M. Emre Celebi, ported to ImageJ plugin by G.Landini, then ported to pi2.

Shanbhag

Described in A.G. Shanhbag, Utilization of Information Measure as a Means of Image Thresholding, Graphical Models and Image Processing 56(5), 1994.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Original code by M. Emre Celebi, ported to ImageJ plugin by Gabriel Landini, then to pi2.

Triangle

The triangle algorithm assumes a peak near either end of the histogram, finds minimum near the other end, draws a line between the minimum and maximum, and sets threshold t to a value for which the point $(t, y(t))$ is furthest away from the line (where y is the histogram).

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Described in G.W. Zack, W.E. Rogers, and S.A. Latt, Automatic Measurement of Sister Chromatid Exchange Frequency, Journal of Histochemistry and Cytochemistry 25(7), 1977.

Original code by Johannes Schindelin, modified by Gabriel Landini, then ported to pi2.

Yen

Yen thresholding method described in J.C. Yen, F.K. Chang, and S. Chang, A New Criterion for Automatic Multilevel Thresholding, IEEE Transactions on Image Processing 4(3), 1995.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Original code by M. Emre Celebi, ported to ImageJ plugin by Gabriel Landini, then to pi2.

Median

The threshold is shifted median of the greyscale data, i.e. $t = M - c$, where t is the threshold, M is the median of the image, and c is the shift.

The median is calculated from image histogram, so its accuracy might degrade if too small bin count is used. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram. The fourth argument is the shift value c.

MidGrey

The threshold is $(m + M)/2 - c$, where m and M are the minimum and the maximum value of the image or neighbourhood, respectively. The value c is a user-specified shift.

The first argument is the shift value c.

Niblack

Niblack's thresholding method.

This method is mostly suited for local thresholding.

The threshold is $\mu+k*\sigma-c$, where μ and σ are the mean and the standard deviation of the image or the neighbourhood, respectively. The values k and c are user-specified scaling constant and shift.

The first argument is the scaling constant k. The second argument is the shift value c.

Described in W. Niblack, An introduction to Digital Image Processing, Prentice-Hall, 1986.

Phansalkar

Phansalkar's thresholding method.

This method is mostly suited for local thresholding.

Threshold is $\mu * (1.0 + p * \exp(-q * \mu) + k * (\sigma/r - 1))$ where μ and σ are the mean and the standard deviation of the image or the neighbourhood, respectively. The values k, r, p, and q are user-specified parameters of the method. The default values are $k = 0.25$, $r = 0.5$, $p = 2.0$, and $q = 10.0$.

The first four arguments are the parameters k, r, p, and q.

Described in N. Phansalskar, S. More, and A. Sabale, et al., Adaptive local thresholding for detection of nuclei in diversity stained cytology images, International Conference on Communications and Signal Processing (ICCSP), 2011.

Sauvola

Sauvola's thresholding method.

This method is mostly suited for local thresholding.

The threshold is $\mu * (1 + k * (\sigma/r - 1))$, where μ and σ are the mean and the standard deviation of the image or the neighbourhood, respectively. The values k and r are user-specified scaling constants.

The first argument is the scaling constant k. The second argument is the scaling constant r.

Described in Sauvola, J and Pietaksinen, M, Adaptive Document Image Binarization, Pattern Recognition 33(2), 2000.

Bernsen

Finds Bernsen's thresholding method.

This method is mostly suited for local thresholding.

The method uses a user-provided contrast threshold. If the local contrast (max - min) is above or equal to the contrast threshold, the threshold is set at the local midgrey value (the mean of the minimum and maximum grey values in the local window (or whole image in the case of global thresholding)). If the local contrast is below the contrast threshold the neighbourhood is considered to consist only of one class and the pixel is set to object or background depending on the value of the midgrey.

The first argument is the local contrast threshold.

Described in J. Bernsen, Dynamic Thresholding of Grey-Level Images, Proceedings of the 8th International Conference on Pattern Recognition, 1986.

The original code is written by Gabriel Landini, and it has been ported to pi2.

Note: In Python/pi2py2, the output image is not specified, and the result value is returned by the function.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Output image.

method [input]

Data type: string

Default value: Otsu

Thresholding method that will be applied.

argument 1 [input]

Data type: real

Default value: nan

Argument for the thresholding method. The purpose of this argument depends on the method, see the list above. Specify nan in order to use a method-specific default value.

argument 2 [input]

Data type: real

Default value: nan

Argument for the thresholding method. The purpose of this argument depends on the method, see the list above. Specify nan in order to use a method-specific default value.

argument 3 [input]

Data type: real

Default value: nan

Argument for the thresholding method. The purpose of this argument depends on the method, see the list above. Specify nan in order to use a method-specific default value.

argument 4 [input]

Data type: real

Default value: nan

Argument for the thresholding method. The purpose of this argument depends on the method, see the list above. Specify nan in order to use a method-specific default value.

See also

autothreshold, *autothresholdvalue*, *localthreshold*, *threshold*, *doublethreshold*, *dualthreshold*, *thresholdrange*

3.8.8 axelssoncolor

Syntax: `axelssoncolor(geometry, phi, theta, r, g, b)`

Color coding of orientation data used in Axelsson - Estimating 3D fibre orientation in volume images. This color coding is most suited to materials where most orientations are in the xy-plane, e.g. paper or cardboard. In the output, hue describes angle between the positive x-axis and the projection of the orientation vector to the xy-plane, i.e. the azimuthal component of the orientation direction. Absolute value of the z-coordinate of the orientation direction is mapped to saturation, maximum being at the xy-plane. Value is mapped to the pixel value in the geometry image normalized such that the maximum value of the geometry image is mapped to 1, and zero to 0.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

geometry [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

An image that contains the geometry for which local orientation has been determined.

phi [input]

Data type: float32 image

The azimuthal angle of the local orientation direction. The angle is given in radians and measured from positive x-axis towards positive y-axis and is given in range $[-\pi, \pi]$.

theta [input]

Data type: float32 image

The polar angle of the local orientation direction. The angle is given in radians and measured from positive z-axis towards xy-plane. The values are in range $[0, \pi]$.

r [output]

Data type: uint8 image

Red channel of the result will be stored into this image.

g [output]

Data type: uint8 image

Green channel of the result will be stored into this image.

b [output]

Data type: uint8 image

Blue channel of the result will be stored into this image.

See also

cylindricality, *cylinderorientation*, *plateorientation*, *mainorientationcolor*, *axelssoncolor*, *orientationdifference*

3.8.9 bandpassfilter

Syntax: `bandpassfilter(image, minimum size, maximum size)`

Bandpass filtering. Removes gray value variations smaller or larger in spatial extent than specified.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image [input & output]

Data type: float32 image

Image to process.

minimum size [input]

Data type: real

Default value: 3

Variations smaller than than this value are removed.

maximum size [input]

Data type: real

Default value: 40

Variations larger than this value are removed.

See also

fft, *ifft*, *bandpassfilter*, *highpassfilter*

3.8.10 bilateralfilter

Syntax: `bilateralfilter(input image, output image, spatial sigma, radiometric sigma, boundary condition)`

Bilateral filtering. Removes noise from the image while trying to preserve sharp edges. The filter is realized as a weighted local average, where weight value depends on both spatial and radiometric distance to the central pixel. See also C. Tomasi, R. Manduchi, Bilateral filtering for gray and color images, Sixth International Conference on Computer Vision, Bombay, 1998.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Output image.

spatial sigma [input]

Data type: real

Standard deviation of Gaussian kernel used for spatial smoothing.

radiometric sigma [input]

Data type: real

Standard deviation of Gaussian kernel used to avoid smoothing edges of features. Order of magnitude must be similar to difference between gray levels of background and objects.

boundary condition [input]

Data type: boundary condition

Default value: Nearest

Type of boundary condition. Zero indicates that values outside of image bounds are taken to be zero. Nearest indicates that the nearest value inside the image is to be used in place of values outside of image bounds.

See also

gaussfilter, *bilateralfilter*, *bilateralfilterapprox*, *vawefilter*, *openingfilter*, *closingfilter*, *minfilter*, *maxfilter*, *medianfilter*, *variancefilter*, *stddevfilter*, *bandpassfilter*, *highpassfilter*

3.8.11 bilateralfilterapprox

Syntax: `bilateralfilterapprox(input image, output image, spatial sigma, radiometric sigma, sample count)`

Approximate bilateral filtering. Removes noise from the image while trying to preserve sharp edges. The filter is realized as a weighted local average, where weight value depends on both spatial and radiometric distance to the central pixel. The difference to exact bilateral filter is that the approximate version does not process all pixels in a neighbourhood but only a random subset of them. As a result, the filtering operation is much faster but the output may contain some random noise.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Output image.

spatial sigma [input]

Data type: real

Standard deviation of Gaussian kernel used for spatial smoothing.

radiometric sigma [input]

Data type: real

Standard deviation of Gaussian kernel used to avoid smoothing edges of features. Order of magnitude must be similar to difference between gray levels of background and objects.

sample count [input]

Data type: positive integer

Default value: 0

Count of samples processed in each neighbourhood. Specify zero to determine sample count automatically.

See also

gaussfilter, *bilateralfilter*, *bilateralfilterapprox*, *vawefilter*, *openingfilter*, *closingfilter*, *minfilter*, *maxfilter*, *medianfilter*, *variancefilter*, *stddevfilter*, *bandpassfilter*, *highpassfilter*

3.8.12 bin

Syntax: `bin(input image, output image, factor, binning type)`

Reduces size of input image by given integer factor. Each output pixel corresponds to factor^dimensionality block of pixels in the input image.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image

Output image.

factor [input]

Data type: 3-component integer vector

Binning factor in each coordinate direction. Value 2 makes the output image dimension half of the input image dimension, 3 makes them one third etc.

binning type [input]

Data type: string

Default value: mean

Name of binning type to be performed. Currently 'mean', 'sum', 'min' and 'max' are supported.

See also

scale, *bin*, *maskedbin*, *scalelabels*

3.8.13 blockmatch

There are 2 forms of this command.

`blockmatch(reference image, deformed image, xmin, xmax, xstep, ymin, ymax, ystep, zmin, zmax, zstep, initial shift, file name prefix, comparison radius, subpixel accuracy)`

Calculates displacement field between two images. NOTE: This command is currently implemented in very old format, and thus it forcibly saves the results to a file.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

reference image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Reference image (non-moving image).

deformed image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Deformed image (image to register to non-moving image).

xmin [input]

Data type: integer

X-coordinate of the first calculation point in the reference image.

xmax [input]

Data type: integer

X-coordinate of the last calculation point in the reference image.

xstep [input]

Data type: integer

Step between calculation points in x-direction.

ymin [input]

Data type: integer

Y-coordinate of the first calculation point in the reference image.

ymax [input]

Data type: integer

Y-coordinate of the last calculation point in the reference image.

ystep [input]

Data type: integer

Step between calculation points in y-direction.

zmin [input]

Data type: integer

Z-coordinate of the first calculation point in the reference image.

zmax [input]

Data type: integer

Z-coordinate of the last calculation point in the reference image.

zstep [input]

Data type: integer

Step between calculation points in z-direction.

initial shift [input]

Data type: 3-component real vector

Initial shift between the images.

file name prefix [input]

Data type: string

Prefix (and path) of files to write. The command will save point grid in the reference image, corresponding points in the deformed image, and goodness-of-fit. If the files exists, the current contents are erased.

comparison radius [input]

Data type: 3-component integer vector

Default value: "[25, 25, 25]"

Radius of comparison region.

subpixel accuracy [input]

Data type: string

Default value: centroid

Subpixel accuracy mode. Can be 'none', 'quadratic', or 'centroid'.

See also

blockmatch, *blockmatchmemsave*, *pullback*, *pointstodeformed*

`blockmatch(reference image, deformed image, x grid, y grid, z grid, initial shift, file name prefix, coarse comparison radius, coarse binning, fine comparison radius, fine binning, subpixel accuracy)`

Calculates displacement field between two images with two-step multi-resolution approach, where coarse displacement is first calculated with larger block size (and binning) and the result is refined in second phase with smaller block size (and binning). NOTE: This command is currently implemented in very old format, and thus it forcibly saves the results to a file.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

reference image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Reference image (non-moving image).

deformed image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Deformed image (image to register to non-moving image).

x grid [input]

Data type: 3-component integer vector

Calculation point grid definition in X-direction. The format is [start coordinate, end coordinate, step].

y grid [input]

Data type: 3-component integer vector

Calculation point grid definition in Y-direction. The format is [start coordinate, end coordinate, step].

z grid [input]

Data type: 3-component integer vector

Calculation point grid definition in Z-direction. The format is [start coordinate, end coordinate, step].

initial shift [input]

Data type: 3-component real vector

Initial shift between the images.

file name prefix [input]

Data type: string

Prefix (and path) of files to write. The command will save point grid in the reference image, corresponding points in the deformed image, and goodness-of-fit. If the files exists, the current contents are erased.

coarse comparison radius [input]

Data type: 3-component integer vector

Default value: "[25, 25, 25]"

Radius of comparison region for coarse matching.

coarse binning [input]

Data type: positive integer

Default value: 2

Amount of resolution reduction in coarse matching phase.

fine comparison radius [input]

Data type: 3-component integer vector

Default value: "[10, 10, 10]"

Radius of comparison region for fine (full-resolution) matching.

fine binning [input]

Data type: positive integer

Default value: 1

Amount of resolution reduction in fine matching phase. Set to same value than coarse binning to skip fine matching phase.

subpixel accuracy [input]

Data type: string

Default value: centroid

Subpixel accuracy mode. Can be 'none', 'quadratic', or 'centroid'.

See also

blockmatch, *blockmatchmemsave*, *pullback*, *pointstodeformed*

3.8.14 blockmatchmemsave

Syntax: `blockmatchmemsave(reference image file, deformed image file, xmin, xmax, xstep, ymin, ymax, ystep, zmin, zmax, zstep, initial shift, file name prefix, coarse comparison radius, coarse binning, fine comparison radius, fine binning, normalize, subpixel accuracy)`

Calculates displacement between two images, loads only overlapping region from disk. NOTE: This command is currently implemented in very old format, and thus it forcibly saves the results to a file.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

reference image file [input]

Data type: string

Name of reference image file (non-moving image).

deformed image file [input]

Data type: string

Name of deformed image file (image to register to non-moving image).

xmin [input]

Data type: integer

X-coordinate of the first calculation point in the reference image.

xmax [input]

Data type: integer

X-coordinate of the last calculation point in the reference image.

xstep [input]

Data type: integer

Step between calculation points in x-direction.

ymin [input]

Data type: integer

Y-coordinate of the first calculation point in the reference image.

ymax [input]

Data type: integer

Y-coordinate of the last calculation point in the reference image.

ystep [input]

Data type: integer

Step between calculation points in y-direction.

zmin [input]

Data type: integer

Z-coordinate of the first calculation point in the reference image.

zmax [input]

Data type: integer

Z-coordinate of the last calculation point in the reference image.

zstep [input]

Data type: integer

Step between calculation points in z-direction.

initial shift [input]

Data type: 3-component real vector

Initial shift between the images.

file name prefix [input]

Data type: string

Prefix (and path) of files to write. The command will save point grid in the reference image, corresponding points in the deformed image, and goodness-of-fit. If the files exists, the current contents are erased.

coarse comparison radius [input]

Data type: 3-component integer vector

Default value: "[25, 25, 25]"

Radius of comparison region for coarse matching.

coarse binning [input]

Data type: integer

Default value: 2

Amount of resolution reduction in coarse matching phase.

fine comparison radius [input]

Data type: 3-component integer vector

Default value: "[10, 10, 10]"

Radius of comparison region for fine (full-resolution) matching.

fine binning [input]

Data type: integer

Default value: 2

Amount of resolution reduction in fine matching phase. Set to same value than coarse binning to skip fine matching phase.

normalize [input]

Data type: boolean

Default value: True

Indicates if the mean gray values of the two images should be made same in the overlapping region before matching.

subpixel accuracy [input]

Data type: string

Default value: centroid

Subpixel accuracy mode. Can be 'none', 'quadratic', or 'centroid'.

See also

blockmatch, *blockmatchmemsave*, *pullback*, *pointstodeformed*

3.8.15 box

There are 2 forms of this command.

`box(image, position, size, value, block origin)`

Draws a filled axis-aligned box into the image. The filling is performed such that the left-, top- and front-faces of the box are filled but the right-, back- and bottom-faces are not. Notice that this convention is different from what is used to draw a non-axis-aligned box with the version of this command that takes box orientation vectors as arguments.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

position [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

Position of the left-top corner of the box.

size [input]

Data type: 3-component integer vector

Default value: "[10, 10, 10]"

Size of the box.

value [input]

Data type: real

Default value: 1

Value for pixels inside the box.

block origin [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

Origin of current calculation block in coordinates of the full image. This argument is used internally in distributed processing. Set to zero in normal usage.

See also

line, *capsule*, *sphere*, *ellipsoid*, *set*, *get*, *ramp*, *ramp3*

`box(image, position, size, value, direction 1, direction 2, block origin)`

Draws a filled generic non-axis-aligned box into the image. The filling is performed such that pixels on the surface of the box are not filled. Notice that this convention is different than what is used in the version of the command that does not take box orientation vectors as arguments.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

position [input]

Data type: 3-component real vector

Default value: "[0, 0, 0]"

Position of the left-top corner of the box.

size [input]

Data type: 3-component real vector

Default value: "[10, 10, 10]"

Size of the box.

value [input]

Data type: real

Default value: 1

Value for pixels inside the box.

direction 1 [input]

Data type: 3-component real vector

Direction vector for the first axis of the box.

direction 2 [input]

Data type: 3-component real vector

Direction vector for the second axis of the box.

block origin [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

Origin of current calculation block in coordinates of the full image. This argument is used internally in distributed processing. Set to zero in normal usage.

See also

line, *capsule*, *sphere*, *ellipsoid*, *set*, *get*, *ramp*, *ramp3*

3.8.16 canny

Syntax: `canny(image, derivative sigma, lower threshold, upper threshold)`

Performs Canny edge detection. Skips the initial Gaussian blurring step, please perform it separately if you want to do it. Calculates image derivatives using convolution with derivative of Gaussian.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

derivative sigma [input]

Data type: real

Default value: 1

Scale parameter for derivative calculation. Set to the preferred scale of edges that should be detected. Derivatives are calculated using convolutions with derivative of Gaussian function, and this parameter defines the standard deviation of the Gaussian.

lower threshold [input]

Data type: real

Edges that have gradient magnitude below lower threshold value are discarded. Edges that have gradient magnitude between lower and upper thresholds are included in the result only if they touch some edge that has gradient magnitude above upper threshold.

upper threshold [input]

Data type: real

Edges that have gradient magnitude above upper threshold value are always included in the result. Edges that have gradient magnitude between lower and upper thresholds are included in the result only if they touch some edge that has gradient magnitude above upper threshold.

3.8.17 capsule

Syntax: `capsule(image, start, end, radius, value, block origin)`

Draws a capsule (with rounded ends) into the image.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

start [input]

Data type: 3-component real vector

Default value: "[0, 0, 0]"

Start position of the line.

end [input]

Data type: 3-component real vector

Default value: "[1, 1, 1]"

End position of the line.

radius [input]

Data type: real

Default value: 5

Radius of the capsule

value [input]

Data type: real

Default value: 1

Value for pixels inside the capsule.

block origin [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

Origin of current calculation block in coordinates of the full image. This argument is used internally in distributed processing. Set to zero in normal usage.

See also

line, *capsule*, *sphere*, *ellipsoid*, *set*, *get*, *ramp*, *ramp3*

3.8.18 ceil

Syntax: `ceil(image)`

Calculates ceiling of pixel values, i.e. the greatest integer less than or equal to the pixel value.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

3.8.19 chunksize

Syntax: `chunksize(chunk size)`

Sets the NN5 dataset chunk size used in distributed processing. This command overrides the value read from the configuration file.

This command can be used in the distributed processing mode, but it does not participate in distributed processing.

Arguments

chunk size [input]

Data type: 3-component integer vector

Default value: "[1536, 1536, 1536]"

New NN5 dataset chunk size.

See also

distribute, *delaying*, *maxmemory*, *maxjobs*, *chunksize*, *printscripts*

3.8.20 classifyskeleton

Syntax: `classifyskeleton(image)`

Classifies line skeleton to end points, curve points, branch points, junction points, internal points, and edge points according to Arcelli, From 3D Discrete Surface Skeletons to Curve Skeletons.End points are given value 2, curve points value 3, branch points value 4, junction points value 5, internal points value 6, and edge points value 7.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

See also

surfacethin, *surfaceskeleton*, *linethin*, *lineskeleton*, *tracelineskeleton*, *classifyskeleton*

3.8.21 cleanmaxima

Syntax: `cleanmaxima(image, maxima, radius multiplier)`

Removes all maxima that are smaller in radius than neighbouring maximum. Maximum is neighbour to another maximum if distance between them is less than radius of the larger maximum multiplied by radiusMultiplier. Removes all maxima m that satisfy $distance(m, n) < radiusMultiplier * radius(n)$ and $radius(n) > radius(m)$ for some n. The distance is measured between centroids of the maxima. The maxima are removed by combining them to the larger maxima.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

The image where the maxima have been extracted (by *localmaxima* command).

maxima [input & output]

Data type: int32 image

Image that contains the maxima. See output from *localmaxima* command.

radius multiplier [input]

Data type: real

Default value: 1

Maxima are enlarged by this multiplier.

See also

localmaxima, *cleanmaxima*, *labelmaxima*, *growlabels*, *grow*

3.8.22 cleanskeleton

Syntax: `cleanskeleton(vertices, edges, edge measurements, edge points)`

Removes straight-through and isolated nodes from a network traced from a skeleton (i.e. all nodes that have either 0 or 2 neighbours, i.e. all nodes whose degree is 0 or 2).

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

vertices [input & output]

Data type: float32 image

Image where vertex coordinates are stored. See *tracelineskeleton* command.

edges [input & output]

Data type: uint64 image

Image where vertex indices corresponding to each edge are stored. See *tracelineskeleton* command.

edge measurements [input & output]

Data type: float32 image

Image where properties of each edge are stored. See *tracelineskeleton* command.

edge points [input & output]

Data type: int32 image

Image that stores some points on each edge. See *tracelineskeleton* command.

See also

cleanskeleton, *pruneskeleton*, *removeedges*, *fillskeleton*, *getpointsandlines*

3.8.23 clear

Syntax: `clear(image name)`

Dispose image or value from the system (and free memory that the object consumes).

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image name [input]

Data type: string

Default value: “”

Name of image or value to erase. Specify empty name to clear everything.

See also

list

3.8.24 clearmeta

Syntax: `clearmeta(image)`

Clears metadata of an image

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to process.

See also

setmeta, *getmeta*, *writemeta*, *readmeta*, *clearmeta*, *listmeta*, *copymeta*, *metarowcount*, *metacolumncount*

3.8.25 closingfilter

Syntax: `closingfilter(image, radius, allow optimization, neighbourhood type, boundary condition)`

Closing filter. Closes gaps in bright objects. Has optimized implementation for rectangular neighbourhoods. Creates one temporary image of same size than input.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

radius [input]

Data type: 3-component integer vector

Default value: "[1, 1, 1]"

Radius of neighbourhood. Diameter will be $2r + 1$.

allow optimization [input]

Data type: boolean

Default value: True

Set to true to allow use of approximate decompositions of spherical structuring elements using periodic lines. As a result of the approximation processing is much faster but the true shape of the structuring element is not sphere but a regular polyhedron. See van Herk - A fast algorithm for local minimum and maximum filters on rectangular and octagonal kernels and Jones - Periodic lines Definition, cascades, and application to granulometries. The approximate filtering will give wrong results where distance from image edge is less than r. Consider enlarging the image by r

to all directions before processing. Enlarging in the z-direction is especially important for 2D images, and therefore approximate processing is not allowed if the image is 2-dimensional.

neighbourhood type [input]

Data type: neighbourhood type

Default value: Ellipsoidal

Type of neighbourhood. Can be Ellipsoidal for ellipsoidal or spherical neighbourhood; or Rectangular for rectangular neighbourhood.

boundary condition [input]

Data type: boundary condition

Default value: Nearest

Type of boundary condition. Zero indicates that values outside of image bounds are taken to be zero. Nearest indicates that the nearest value inside the image is to be used in place of values outside of image bounds.

See also

gaussfilter, *bilateralfilter*, *bilateralfilterapprox*, *vawefilter*, *openingfilter*, *closingfilter*, *minfilter*, *maxfilter*, *medianfilter*, *variancefilter*, *stddevfilter*, *bandpassfilter*, *highpassfilter*

3.8.26 conjugate

Syntax: `conjugate(image)`

Calculates complex conjugate of pixel values.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: complex32 image

Image to process.

3.8.27 convert

There are 2 forms of this command.

`convert(input image, output image, data type)`

Converts data type of input image.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [input]

Data type: string

Output image.

data type [input]

Data type: string

Data type of the output image. Can be uint8, uint16, uint32, uint64, int8, int16, int32, int64, float32, or complex32.

`convert(image, data type)`

Converts data type of an image.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image whose data type is to be converted.

data type [input]

Data type: string

Data type to convert to. Can be uint8, uint16, uint32, uint64, int8, int16, int32, int64, float32, or complex32.

3.8.28 copy

There are 2 forms of this command.

`copy(input image, output image)`

Copies input image to output image. If pixel data types are not the same, performs conversion.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image, complex32 image

Output image.

`copy(source image, target image, location)`

Copies pixel values from source image to target image to specified location. The size of the target image is not changed and out-of-bounds pixels are not copied. See also *set* command.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

source image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image that is copied to the target image.

target image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image whose values are set.

location [input]

Data type: 3-component integer vector

Location where the target image is placed in the source image.

See also

scale, *bin*, *maskedbin*, *scalelabels*

3.8.29 copymeta

Syntax: `copymeta(input image, output image)`

Copies metadata from one image to another.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image, complex32 image

Output image.

See also

setmeta, *getmeta*, *writemeta*, *readmeta*, *clearmeta*, *listmeta*, *copymeta*, *metarowcount*, *metacolumncount*

3.8.30 cos

Syntax: `cos(image)`

Calculates cosine of pixel values.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to process.

3.8.31 createfbpfilter

Syntax: `createfbpfilter(image, size, filter type, cut-off)`

Creates image containing values of filter function used in filtered backprojection. Can be used to visualize different filter functions.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image [output]

Data type: float32 image

Image where the filter is to be placed.

size [input]

Data type: positive integer

Default value: 100

Size of the filter. This corresponds to the padded image size in FBP.

filter type [input]

Data type: string

Default value: Ramp

Type of the filter. Supported values are Ideal ramp, Ramp, Shepp-Logan, Cosine, Hamming, Hann, Blackman, Parze.

cut-off [input]

Data type: real

Default value: 1

Filter cut-off frequency, 0 corresponds to DC and 1 corresponds to Nyquist.

See also

fbppreprocess, *fbp*, *deadpixelremoval*, *createfbpfilter*, *defaultrecsettings*

3.8.32 crop

Syntax: `crop(input image, output image, position, size)`

Crops the input image. If the size parameter is set to zero, crops to current size of the output image. NOTE: Input and output images must have the same data type. If not, output image is converted to the correct type.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Output image.

position [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

Position in input image where the top-left corner of the cropped image is placed.

size [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

Size of output image. Specify zeroes or nothing to crop to the current size of the output image.

See also

scale, *bin*, *maskedbin*, *scalelabels*

3.8.33 csa

There are 2 forms of this command.

`csa(original, energy, phi, theta, results, slice radius, slice count, random seed, slices, visualization, length, length slices)`

Analyzes cross-sections of cylindrical, tubular, or fibre-like objects. Use to e.g. measure cross-sectional area of fibres. Requires that the local orientation of the fibres has been determined using, e.g., *cylinderorientation* command.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

original [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Binary image containing the fibres as foreground.

energy [input]

Data type: float32 image

Image corresponding to the energy output image of cylinderorientation command. Non-zero energy signifies that local orientation is available at that location.

phi [input]

Data type: float32 image

The azimuthal angle of the local fibre orientation direction. The angle is given in radians and measured from positive x-axis towards positive y-axis and is given in range $[-\pi, \pi]$.

theta [input]

Data type: float32 image

The polar angle of the local fibre orientation direction. The angle is given in radians and measured from positive z-axis towards xy-plane. The values are in range $[0, \pi]$.

results [output]

Data type: float32 image

Image where analysis results are placed. This image will contain one row for each fibre cross-section analyzed. Use command *csaheaders* to retrieve meanings of columns.

slice radius [input]

Data type: positive integer

Default value: 40

Half width of one cross-sectional slice

slice count [input]

Data type: positive integer

Default value: 1000

Count of cross-sectional slices to analyze

random seed [input]

Data type: positive integer

Default value: 123

Seed for random number generator.

slices [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

The extracted slices are placed into this image.

visualization [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

The locations where the slices are extracted from are drawn into this image

length [input]

Data type: float32 image

Image containing local fibre length.

length slices [output]

Data type: float32 image

The extracted slices from the length image are placed into this image.

See also

analyzeparticles, *listanalyzers*, *headers*, *fillparticles*, *drawellipsoids*, *label*, *analyzelabels*, *regionremoval*, *greedycoloring*, *csa*, *cylinderorientation*

`csa(original, energy, phi, theta, results, slice radius, slice count, random seed, slices, visualization)`

Analyzes cross-sections of cylindrical, tubular, or fibre-like objects. Use to e.g. measure cross-sectional area of fibres. Requires that the local orientation of the fibres has been determined using, e.g., *cylinderorientation* command.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

original [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Binary image containing the fibres as foreground.

energy [input]

Data type: float32 image

Image corresponding to the energy output image of cylinderorientation command. Non-zero energy signifies that local orientation is available at that location.

phi [input]

Data type: float32 image

The azimuthal angle of the local fibre orientation direction. The angle is given in radians and measured from positive x-axis towards positive y-axis and is given in range $[-\pi, \pi]$.

theta [input]

Data type: float32 image

The polar angle of the local fibre orientation direction. The angle is given in radians and measured from positive z-axis towards xy-plane. The values are in range $[0, \pi]$.

results [output]

Data type: float32 image

Image where analysis results are placed. This image will contain one row for each fibre cross-section analyzed. Use command *csaheaders* to retrieve meanings of columns.

slice radius [input]

Data type: positive integer

Default value: 40

Half width of one cross-sectional slice

slice count [input]

Data type: positive integer

Default value: 1000

Count of cross-sectional slices to analyze

random seed [input]

Data type: positive integer

Default value: 123

Seed for random number generator.

slices [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

The extracted slices are placed into this image.

visualization [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

The locations where the slices are extracted from are drawn into this image

See also

analyzeparticles, *listanalyzers*, *headers*, *fillparticles*, *drawellipsoids*, *label*, *analyzelabels*, *regionremoval*, *greedycoloring*, *csa*, *cylinderorientation*

3.8.34 csaheaders

Syntax: `csaheaders()`

Retrieves information on the meaning of the columns in the result image of *csa* command.

This command can be used in the distributed processing mode, but it does not participate in distributed processing.

See also

csa

3.8.35 curvature

Syntax: `curvature(geometry, radius, kappa1, kappa2, boundary condition, non-surface value)`

Calculates curvature of surfaces in the image. Uses quadratic surface fitting algorithms in Petitjean - A Survey of Methods for Recovering Quadrics in Triangle Meshes. Pointwise surface normal is determined using principal component analysis of the covariance matrix of surface points near the center point. The surface normal orientation is chosen so that it points toward background voxels. Curvature is determined by transforming surface points near center point to a coordinate system where the z-direction is parallel to the surface normal, and then fitting a surface $f(x, y) = ax^2 + bxy + cy^2 + d$ to the tranformed points. The curvature values and directions are calculated from the coefficients a, b and c. Finally, directions are then transformed back to the original coordinates.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

geometry [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

The image containing the geometry. Non-zero pixels are assumed to be foreground.

radius [input]

Data type: real

Default value: 5

Geodesic radius (radius on the surface) of neighbourhood around surface point considered when determining curvature. Typically e.g. 5 gives good results.

kappa1 [output]

Data type: float32 image

Largest principal curvature will be placed to this image. Set image size to (1, 1, 1) to skip calculation of this quantity.

kappa2 [output]

Data type: float32 image

Smallest principal curvature will be placed to this image. Set image size to (1, 1, 1) to skip calculation of this quantity.

boundary condition [input]

Data type: boundary condition

Default value: Nearest

Type of boundary condition. Zero indicates that values outside of image bounds are taken to be zero. Nearest indicates that the nearest value inside the image is to be used in place of values outside of image bounds.

non-surface value [input]

Data type: real

Default value: nan

Value that is used to fill the non-surface points in the kappa1 and kappa2 images.

See also

meancurvature

3.8.36 cylinderorientation

Syntax: `cylinderorientation(geometry, phi, theta, derivative sigma, smoothing sigma)`

Estimates local orientation of cylindrical structures using the structure tensor method. See also B. Jähne, Practical handbook on image processing for scientific and technical applications. CRC Press, 2004.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

geometry [input & output]

Data type: float32 image

At input, an image that contains the geometry for which local orientation should be determined. At output, the orientation 'energy' will be stored in this image. It equals the sum of the eigenvalues of the structure tensor, and can be used to distinguish regions without any interfaces (i.e. no orientation, low energy value) from regions with interfaces (i.e. orientation available, high energy value).

phi [output]

Data type: float32 image

The azimuthal angle of orientation direction will be stored in this image. The angle is given in radians and measured from positive x-axis towards positive y-axis and is given in range $[-\pi, \pi]$.

theta [output]

Data type: float32 image

The polar angle of orientation direction will be stored in this image. The angle is given in radians and measured from positive z-axis towards xy-plane. The values are in range $[0, \pi]$.

derivative sigma [input]

Data type: real

Default value: 1

Scale parameter. Set to the preferred scale of edges that define the cylinders. Derivatives required in the stucture tensor are calculated using convolutions with derivative of a Gaussian function, and this parameter defines the standard deviation of the Gaussian.

smoothing sigma [input]

Data type: real

Default value: 1

The structure tensor is smoothed by convolution with a Gaussian. This parameter defines the standard deviation of the smoothing Gaussian.

See also

cylindricality, *cylinderorientation*, *plateorientation*, *mainorientationcolor*, *axelssoncolor*, *orientationdifference*

3.8.37 cylindricality

Syntax: `cylindricality(image, derivative sigma, smoothing sigma)`

Estimates likelihood of structures being cylinders, based on eigenvalues of the local structure tensor. The input image is replaced with the cylindricality values.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: float32 image

Image to process.

derivative sigma [input]

Data type: real

Default value: 1

Scale parameter. Set to the preferred scale of edges that define the cylinders. Derivatives required in the stucture tensor are calculated using convolutions with derivative of a Gaussian function, and this parameter defines the standard deviation of the Gaussian.

smoothing sigma [input]

Data type: real

Default value: 1

The structure tensor is smoothed by convolution with a Gaussian. This parameter defines the standard deviation of the smoothing Gaussian.

See also

cylindricality, *cylinderorientation*, *plateorientation*, *mainorientationcolor*, *axelssoncolor*, *orientationdifference*

3.8.38 danielsson2

Syntax: `danielsson2(input image, output image)`

Calculates an estimate of the set of centers of locally maximal spheres from squared Euclidean distance map. Uses Danielsson's algorithm. The output image can be interpreted as a medial axis or a distance ridge. Drawing a sphere on each nonzero pixel in the output image, with radius and color equal to square root of pixel value, results in a local thickness map of the structure. See e.g. Yaorong Ge and J. Michael Fitzpatrick - On the Generation of Skeletons from Discrete Euclidean Distance Maps, IEEE Transactions on Pattern Analysis and Machine Intelligence 18, 1996; and P.-E. Danielsson, Euclidean distance mapping, Computer Graphics and Image Processing 14(3), 1980.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Output image.

See also

dmap2, *dmap*, *tmap*

3.8.39 deadpixelremoval

Syntax: `deadpixelremoval(image, radius, magnitude)`

Removes dead pixels from projection dataset.Determines whether dead pixel removal algorithm should be applied to the projection images. In the algorithm, each flat-field corrected projection I is processed separately. Pixel at position x is classified as dead if its value $I(x)$ is NaN or it satisfies $|I(x) - m(x)| > M * std(|I - m|)$, where m is a median filtering of I with user-specified radius, M is a magnitude parameter, and std is standard deviation of whole image. If a pixel is dead, its value is replaced by $m(x)$, otherwise the value is left unchanged.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

radius [input]

Data type: positive integer

Default value: 1

Median filtering radius.

magnitude [input]

Data type: real

Default value: 10

Magnitude parameter M.

See also

fbppreprocess, *fbp*, *deadpixelremoval*, *createfbpfilter*, *defaultrecsettings*

3.8.40 defaultrecsettings

Syntax: `defaultrecsettings(image)`

Writes default reconstruction settings to image metadata.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image [input & output]

Data type: float32 image

Image to process.

See also

fbppreprocess, *fbp*, *deadpixelremoval*, *createfbpfilter*, *defaultrecsettings*

3.8.41 delaying

Syntax: `delaying(enable)`

Enables or disables possibility for combination of similar commands before sending them to compute processes/nodes in distributed processing (kind of lazy evaluation). Delaying decreases amount of disk space and I/O used for temporary images. When delaying is enabled, command run times (see echo command) are not accurate. Delaying is enabled by default.

This command can be used in the distributed processing mode, but it does not participate in distributed processing.

Arguments

enable [input]

Data type: boolean

Default value: True

Set to true to enable delaying.

See also

distribute, *delaying*, *maxmemory*, *maxjobs*, *chunksize*, *printscripts*

3.8.42 derivative

Syntax: `derivative(input image, output image, spatial sigma, dimension 1, dimension 2, boundary condition)`

Calculates Gaussian partial derivative of image, either $\partial f/\partial x_i$ or $\partial^2 f/(\partial x_i \partial x_j)$.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: float32 image

Output image.

spatial sigma [input]

Data type: 3-component real vector

Standard deviation of Gaussian kernel.

dimension 1 [input]

Data type: integer

Dimension where the first partial derivative should be calculated (index i in the example above).

dimension 2 [input]

Data type: integer

Default value: -1

Dimension where the second partial derivative should be calculated (index j in the example above). Pass negative value to calculate only the first derivative.

boundary condition [input]

Data type: boundary condition

Default value: Nearest

Type of boundary condition. Zero indicates that values outside of image bounds are taken to be zero. Nearest indicates that the nearest value inside the image is to be used in place of values outside of image bounds.

See also

gaussfilter, *gradient*, *gradientmagnitude*

3.8.43 distribute

Syntax: `distribute(workload manager system name)`

Enables or disables distributed processing of commands. Run this command before commands that you would like to run using distributed processing. Images used during distributed processing are not available for local processing and vice versa, unless they are loaded again from disk. All commands do not support distributed processing.

This command can be used in the distributed processing mode, but it does not participate in distributed processing.

Arguments

workload manager system name [input]

Data type: string

Default value: “”

Set to ‘SLURM’ to use SLURM workload manager; set to ‘LOCAL’ to process tasks sequentially using the local computer; set to empty string to disable distributed processing (default).

See also

distribute, *delaying*, *maxmemory*, *maxjobs*, *chunksize*, *printscripts*

3.8.44 divide

There are 2 forms of this command.

`divide(image, parameter image, allow broadcast)`

Divides two images. Output is placed to the first image. The operation is performed using saturation arithmetic.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to process.

parameter image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image, complex32 image

Parameter image.

allow broadcast [input]

Data type: boolean

Default value: False

Set to true to allow size of parameter image differ from size of input image. If there is a need to access pixel outside of parameter image, the nearest value inside the image is taken instead. If set to false, dimensions of input and parameter images must be equal. If set to true, the parameter image is always loaded in its entirety in distributed processing mode.

`divide(image, x)`

Divides an image by a constant. The operation is performed using saturation arithmetic.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

x [input]

Data type: real

Divider constant.

3.8.45 dmap

Syntax: `dmap(input image, output image, background value)`

Calculates distance map or distance transform of a binary image. In order to calculate squared distance map, use *dmap2* command. Uses algorithm from Maurer - A Linear Time Algorithm for Computing Exact Euclidean Distance Transforms of Binary Images in Arbitrary Dimensions.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image where background is marked with background value given by the third argument.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image

Output image (distance map) where pixel value equals Euclidean distance to the nearest background pixel. Input and output can be the same image if in-place transformation is preferred and input data type is suitable. Integer pixel data types result in approximate distance map only. If floating point pixel type is used, the results might contain artifacts for very large images due to floating point inaccuracy. If that is encountered, consider calculating squared distance map using integer data type and then converting to floating point format, followed by square root operation. That is, dmap2(img, img); convert(img, float32); squareroot(img); Notice that this sequence may require more memory than the standard operation of this command because of the explicit conversion step.

background value [input]

Data type: real

Default value: 0

Pixels belonging to the background are marked with this value in the input image.

See also

dmap, *dmap2*, *tmap*, *sdmap*

3.8.46 dmap2

Syntax: `dmap2(input image, output image, background value)`

Calculates squared distance map or squared distance transform of a binary image. In order to calculate (non-squared) distance map, use the *dmap* command or take a square root of the result using the *squareroot* command. Uses algorithm from Maurer - A Linear Time Algorithm for Computing Exact Euclidean Distance Transforms of Binary Images in Arbitrary Dimensions.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image where background is marked with background value given by the third argument.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image

Output image (squared distance map) where pixel value equals squared Euclidean distance to the nearest background pixel. Input and output can be the same image if in-place transformation is preferred and input data type is suitable.

For exact result, use integer type. Please note that squared values require relatively large pixel data type, e.g. int32, depending on the magnitude of distances in the image. If floating point pixel type is used, the results might contain artifacts for very large images due to floating point inaccuracy. If that is encountered, consider calculating squared distance map using integer data type and then converting to floating point format, possibly followed by square root operation. That is, dmap2(img, img); convert(img, float32); squareroot(img); Notice that this sequence may require more memory than the standard operation of this command because of the explicit conversion step.

background value [input]

Data type: real

Default value: 0

Pixels belonging to the background are marked with this value in the input image.

See also

dmap, *dmap2*, *tmap*, *sdmap*

3.8.47 doublethreshold

Syntax: `doublethreshold(image, first threshold, second threshold)`

Sets pixel to 0 if its value is less than the first threshold. Sets pixel to 1 if pixel its value is larger than or equal to the first threshold and less than the second threshold (if any). Sets pixel to 2 if pixel its value is larger than or equal to the second threshold.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

first threshold [input]

Data type: real

Pixels whose value is larger than or equal to this threshold and less than the second threshold are set to 1.

second threshold [input]

Data type: real

Pixels whose value is larger than or equal to this threshold are set to 2.

3.8.48 drawellipsoids

Syntax: `drawellipsoids(image, analyzers, results, fill color, ellipsoid type, block origin)`

Visualizes particles that have been analyzed with the *analyzeparticles* command by drawing a (scaled) principal component ellipsoid of each particle. The semi-axis directions of the ellipsoid are the principal directions of the particle, and semi-axis lengths are derived from the size of the particle in the principal directions as detailed in the 'ellipsoid type' parameter. If ellipsoid type parameter is set to 'BoundingSphere', the bounding sphere of the particle is drawn instead of ellipsoid.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

analyzers [input]

Data type: string

List of names of analyzers that have been used to analyze the particles in the *analyzeparticles* command. The analyzers argument must contain the 'pca' analyzer if the ellipsoid type argument is set to any type of ellipsoid. Additionally, 'volume' analyzer is required if the 'Volume' ellipsoid type is selected, and 'boundingsphere' analyzer is required if the 'BoundingSphere' ellipsoid type is selected.

results [input]

Data type: float32 image

Analysis results image.

fill color [input]

Data type: real

Fill color.

ellipsoid type [input]

Data type: string

Default value: principal

Type of ellipsoid to draw. 'Principal' denotes the principal axis ellipsoid without scaling, 'Bounding' results in an ellipsoid that covers all the points of the particle, and 'Volume' results in an ellipsoid whose volume equals the volume of the particle. 'BoundingSphere' denotes bounding sphere of the particle.

block origin [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

Shift that is to be applied to the image before filling the particles. This argument is used internally in distributed processing.

See also

analyzeparticles, *listanalyzers*, *headers*, *fillparticles*, *drawellipsoids*, *label*, *analyzelabels*, *regionremoval*, *greedycoloring*, *csa*

3.8.49 drawgraph

There are 2 forms of this command.

`drawgraph(image, vertices, edges, measurements, edge points, vertex radius, vertex color, edge color, use measured edge area, block origin)`

Draws a graph into the image. Vertices are drawn as spheres and edges as lines.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

vertices [input]

Data type: float32 image

Image where vertex coordinates are stored. The size of the image must be 3xN, where N is the number of vertices in the graph.

edges [input]

Data type: uint64 image

Image where vertex indices corresponding to each edge will be set. The size of the image must be 2xM where M is the number of edges. Each row of the image consists of a pair of indices to the vertex array.

measurements [input]

Data type: float32 image

Image that stores properties of each edge. See output from traceskeleton command.

edge points [input]

Data type: int32 image

Image that stores some points on each edge. See output from traceskeleton command.

vertex radius [input]

Data type: real

Default value: 2

Radius of the spheres corresponding to the vertices.

vertex color [input]

Data type: real(255), real(65535), real(4294967295), real(1.844674407370955e+19), real(127), real(32767), real(2147483647), real(9.223372036854776e+18), real(3.402823466385289e+38)

Default value: Shown along data types.

Value that is used to fill spheres corresponding to vertices.

edge color [input]

Data type: real(255), real(65535), real(4294967295), real(1.844674407370955e+19), real(127), real(32767), real(2147483647), real(9.223372036854776e+18), real(3.402823466385289e+38)

Default value: Shown along data types.

Value that is used to draw lines corresponding to edges.

use measured edge area [input]

Data type: boolean

Default value: True

Set to true to draw edges as capsules with cross-sectional area read from edge properties.

block origin [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

Origin of current block in coordinates of the full image. This argument is used internally in distributed processing to shift the vertices to correct locations when only a part of the image is processed. Set to zero in normal usage.

`drawgraph(image, vertices, edges, vertex radius, vertex color, edge color, block origin)`

Draws a graph into the image. Vertices are drawn as spheres and edges as lines.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

vertices [input]

Data type: float32 image

Image where vertex coordinates are stored. The size of the image must be 3xN, where N is the number of vertices in the graph.

edges [input]

Data type: uint64 image

Image where vertex indices corresponding to each edge will be set. The size of the image must be 2xM where M is the number of edges. Each row of the image consists of a pair of indices to the vertex array.

vertex radius [input]

Data type: real

Default value: 2

Radius of the spheres corresponding to the vertices.

vertex color [input]

Data type: real(255), real(65535), real(4294967295), real(1.844674407370955e+19), real(127), real(32767), real(2147483647), real(9.223372036854776e+18), real(3.402823466385289e+38)

Default value: Shown along data types.

Value that is used to fill spheres corresponding to vertices.

edge color [input]

Data type: real(255), real(65535), real(4294967295), real(1.844674407370955e+19), real(127), real(32767), real(2147483647), real(9.223372036854776e+18), real(3.402823466385289e+38)

Default value: Shown along data types.

Value that is used to draw lines corresponding to edges.

block origin [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

Origin of current block in coordinates of the full image. This argument is used internally in distributed processing to shift the vertices to correct locations when only a part of the image is processed. Set to zero in normal usage.

3.8.50 drawheightmap

Syntax: `drawheightmap(geometry, height map, visualization color)`

Draws a height map to an image.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

geometry [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image where whose pixels are to be set.

height map [input]

Data type: float32 image

Height map. The size of the height map must be $w \times h$ where w and h are the width and the height of the geometry image.

visualization color [input]

Data type: real

Color of the surface in the visualization.

See also

findsurface, *drawheightmap*, *setbeforeheightmap*, *setafterheightmap*, *shiftz*

3.8.51 drawspheres2

Syntax: `drawspheres2(input image, output image, save memory, block size)`

Draws a sphere centered in each non-zero point of the input image. Radius of the sphere is given by the square root of the value of the pixel. Color is given by the value of the pixel. This command is used to generate squared local radius map from squared distance map or output of *danielsson2* command.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Output image.

save memory [input]

Data type: boolean

Default value: False

For non-distributed processing: Set to true to use slower algorithm that uses less memory than the faster one. If set to true, the input and output images can be the same. For distributed processing: Only false is supported at the moment.

block size [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

Block size to use in distributed processing. Set to zero to use a default value calculated based on available memory. If calculation with default value fails, use smaller block size and report it to the authors. This argument has no effect when running in non-distributed mode.

See also

tmap

3.8.52 dualthreshold

Syntax: `dualthreshold(image, lower threshold, upper threshold)`

First sets all pixels with value over upper threshold to 1. Then sets all regions to 1 that have value over lower threshold and that are connected to region that has value over upper threshold.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

lower threshold [input]

Data type: real

Regions that have value below lower threshold value are discarded. Regions that have value between lower and upper thresholds are included in the result only if they touch some region that has value above upper threshold.

upper threshold [input]

Data type: real

Regions that have value above upper threshold value are always included in the result. Regions that have value between lower and upper thresholds are included in the result only if they touch some regoin that has value above upper threshold.

3.8.53 echo

Syntax: `echo(commands, timing)`

Enables or disables echoing of commands and timing information to output.

This command can be used in the distributed processing mode, but it does not participate in distributed processing.

Arguments

commands [input]

Data type: boolean

Default value: True

Set to true to show commands that have been run.

timing [input]

Data type: boolean

Default value: False

Set to true to show timing information on commands that have been run.

See also

help, *info*, *license*, *echo*, *print*, *waitreturn*, *hello*, *timing*, *savetiming*, *resettiming*

3.8.54 ellipsoid

Syntax: `ellipsoid(image, position, semi-axis lengths, value, direction 1, direction 2, block origin)`

Draws a filled ellipsoid into the image, given position, semi-axis lengths and the directions of the first two semi-axes. The direction of the third semi-axis is the cross-product of the first two semi-axis direction vectors. The command ensures that the direction of the second semi-axis is perpendicular to the first and to the third one.The lengths of the direction vectors can be anything. The filling is performed such that pixels on the surface of the ellipsoid are not filled.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

position [input]

Data type: 3-component real vector

Default value: "[0, 0, 0]"

Position of the center point of the ellipsoid.

semi-axis lengths [input]

Data type: 3-component real vector

Default value: "[10, 20, 30]"

Lengths of the semi-axes of the ellipsoid.

value [input]

Data type: real

Default value: 1

Value for pixels inside the ellipsoid.

direction 1 [input]

Data type: 3-component real vector

Default value: "[1, 0, 0]"

Direction vector for the first semi-axis.

direction 2 [input]

Data type: 3-component real vector

Default value: "[0, 1, 0]"

Direction vector for the second semi-axis.

block origin [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

Origin of current calculation block in coordinates of the full image. This argument is used internally in distributed processing. Set to zero in normal usage.

See also

line, *capsule*, *sphere*, *ellipsoid*, *set*, *get*, *ramp*, *ramp3*

3.8.55 ensuresize

There are 2 forms of this command.

`ensuresize(image, width, height, depth)`

Makes sure that the size of the parameter image equals dimensions given as an arguments. The parameter image is re-allocated only if its size must be changed.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to process.

width [input]

Data type: integer

Default value: 1

Desired width of the image.

height [input]

Data type: integer

Default value: 1

Desired height of the image.

depth [input]

Data type: integer

Default value: 1

Desired depth of the image.

See also

newimage

`ensuresize(image, size)`

Makes sure that the size of the parameter image equals dimensions given as an argument. The parameter image is re-allocated only if its size must be changed.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to process.

size [input]

Data type: 3-component integer vector

Desired image dimensions.

See also

newimage

3.8.56 eval

There are 3 forms of this command.

`eval(expression, image)`

Evaluates a mathematical expression on each pixel of one or more images. The expression is given as a string.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

expression [input]

Data type: string

The expression to evaluate on each pixel.

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process. The result will be assigned to this image. This image can be referenced by name x0 in the expression.

`eval(expression, image, argument image)`

Evaluates a mathematical expression on each pixel of one or more images. The expression is given as a string.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

expression [input]

Data type: string

The expression to evaluate on each pixel.

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process. The result will be assigned to this image. Pixel of this image can be referenced by name x0 in the expression.

argument image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image

Argument image. Pixel of this image can be referenced by name x1 in the expression.

`eval(expression, image, argument image, argument image 2)`

Evaluates a mathematical expression on each pixel of one or more images. The expression is given as a string.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

expression [input]

Data type: string

The expression to evaluate on each pixel.

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image

Image to process. The result will be assigned to this image. Pixel of this image can be referenced by name x0 in the expression.

argument image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image

Argument image. Pixel of this image can be referenced by name x1 in the expression.

argument image 2 [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image

Argument image. Pixel of this image can be referenced by name x2 in the expression.

3.8.57 exponentiate

Syntax: `exponentiate(image)`

Exponentates pixel values.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to process.

3.8.58 fbp

Syntax: `fbp(input image, output image, use GPU)`

Performs filtered backprojection of data for which fbppreprocess has been called. Reconstruction settings are read from the metadata of the input image. This command is experimental and may change in the near future.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

input image [input]

Data type: float32 image

Input image.

output image [output]

Data type: float32 image

Output image.

use GPU [input]

Data type: boolean

Default value: True

Set to true to allow processing on a GPU.

See also

fbppreprocess, *fbp*, *deadpixelremoval*, *createfbpfilter*, *defaultrecsettings*

3.8.59 fbppreprocess

Syntax: `fbppreprocess(input image, output image)`

Performs preprocessing of transmission projection data for filtered backprojection. Reconstruction settings are read from the metadata of the input image. This command is experimental and may change in the near future.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

input image [input]

Data type: float32 image

Input image.

output image [output]

Data type: float32 image

Output image.

See also

fbppreprocess, *fbp*, *deadpixelremoval*, *createfbpfilter*, *defaultrecsettings*

3.8.60 fft

Syntax: `fft(input image, output image)`

Calculates Fourier transform.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

input image [input]

Data type: float32 image

Input image.

output image [output]

Data type: complex32 image

Output image.

See also

fft, *ifft*, *bandpassfilter*, *highpassfilter*

3.8.61 fileinfo

There are 2 forms of this command.

`fileinfo(filename, info)`

Reads metadata of given image file.

This command can be used in the distributed processing mode, but it does not participate in distributed processing.

Arguments

filename [input]

Data type: string

Name of image file.

info [input]

Data type: uint32 image

At output, this image will contain four pixels corresponding to width, height, depth, and data type index of the image. If the image has only zero pixels, the given file is not an image that can be read into the pi system. The data type index one correspond to uint8, two to uint16, etc. in this sequence: uint8, uint16, uint32, uint64, float32, complex32, int8, int16, int32, int64.

`fileinfo(filename)`

Displays metadata of given image file.

This command can be used in the distributed processing mode, but it does not participate in distributed processing.

Arguments

filename [input]

Data type: string

Name of image file.

3.8.62 fillparticles

Syntax: `fillparticles(image, analyzers, results, fill color, connectivity, block origin)`

Fills particles that correspond to an entry in a list of particles with specified value. All other particles will be set to value 1. This command does not support cases where the particles have different colors.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

analyzers [input]

Data type: string

List of names of analyzers that have been used to analyze the particles in the *analyzeparticles* command. The analyzers must contain 'coordinates' analyzer; otherwise this command does not know where the particles are.

results [input]

Data type: float32 image

Particle analysis results image.

fill color [input]

Data type: real

Fill color.

connectivity [input]

Data type: connectivity

Default value: Nearest

Connectivity of the particles. Can be Nearest for connectivity to nearest neighbours only, or All for connectivity to all neighbours.

block origin [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

Shift that is to be applied to the image before filling the particles. This argument is used internally in distributed processing.

See also

analyzeparticles, *listanalyzers*, *headers*, *fillparticles*, *drawellipsoids*, *label*, *analyzelabels*, *regionremoval*, *greedycoloring*, *csa*

3.8.63 fillskeleton

Syntax: `fillskeleton(skeleton, vertices, edges, edge measurements, edge points, fill index, block origin)`

Fills a traced skeleton with one of values measured from the skeleton. E.g. fills each branch by its length.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

skeleton [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image containing the skeleton. Point belonging to the skeleton must be marked with non-zero value. The pixel data type should be able to store all values of the fill quantity, or otherwise clipping will occur. (Note that traceskeleton command clears the input image.)

vertices [input]

Data type: float32 image

Image where vertex coordinates are stored. See *tracelineskeleton* command.

edges [input]

Data type: uint64 image

Image where vertex indices corresponding to each edge are stored. See *tracelineskeleton* command.

edge measurements [input]

Data type: float32 image

Image that stores properties of each edge. See *tracelineskeleton* command

edge points [input]

Data type: int32 image

Image that stores some points on each edge. See *tracelineskeleton* command

fill index [input]

Data type: positive integer

Defines column in the measurement image where fill values are grabbed from. Zero corresponds to first column in edge measurements, etc.

block origin [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

If processing a block of original image, origin of the block in coordinates of the full image. This parameter is used internally in distributed processing and should usually be set to its default value.

See also

cleanskeleton, *pruneskeleton*, *removeedges*, *fillskeleton*, *getpointsandlines*

3.8.64 finalizetmap

Syntax: `finalizetmap(input image, output image)`

Converts squared local radius map to local thickness map.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image

Output image.

See also

tmap

3.8.65 findsurface

There are 2 forms of this command.

`findsurface(geometry, height map, stopping gray value, direction, surface tension, iterations, max step, visualization, visualization y, visualization color)`

Surface recognition algorithm 'Carpet' according to Turpeinen - Interface Detection Using a Quenched-Noise Version of the Edwards-Wilkinson Equation. The algorithm places a surface above (alternatively below) the image, and moves it towards larger (alternatively smaller) z values while controlling its dynamics according to the Edwards-Wilkinson equation. The movement of the surface stops when it encounters enough pixels with value above specific stopping gray level. The surface does not move through small holes in the object as it has controllable amount of surface tension.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

geometry [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Geometry image. This image does not need to be binary image (but it can be).

height map [input & output]

Data type: float32 image

Height map that defines the initial position of the surface, or an empty image. The height map gives z-position of the surface for each (x, y)-position of the geometry image. The final position of the surface is also saved into this height

map. If used as an input, the size of the height map must be $w \times h$ where w and h are the width and the height of the geometry image. If the size is not correct, the height map will be zeroed and set to the correct size.

stopping gray value [input]

Data type: real

The movement of the surface stops when it encounters pixels whose value is above this gray level.

direction [input]

Data type: string

Default value: Down

Direction where the surface moves. 'Down' corresponds to direction towards larger z values, and 'Up' corresponds to direction towards smaller z values.

surface tension [input]

Data type: real

Default value: 1

Value that indicates how smooth the surface will be. Larger value results in smoother surface.

iterations [input]

Data type: positive integer

Default value: 150

Count of iterations to perform.

max step [input]

Data type: real

Default value: 1

Maximum amount of movement allowed in one time step.

visualization [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

An image where a visualization of the evolution of the surface will be saved. The dimensions of the visualization will be set to $w \times d \times N$, where w and d are width and depth of the geometry image, and N is the count of iterations to perform.

visualization y [input]

Data type: positive integer

Indicates the y-coordinate of the xz-slice that will be visualized.

visualization color [input]

Data type: real

Color of the surface in the visualization. If set to zero, the color will be set to one above the maximum in geometry image.

See also

findsurface, *drawheightmap*, *setbeforeheightmap*, *setafterheightmap*, *shiftz*

`findsurface(geometry, height map, stopping gray value, direction, surface tension, iterations, max step)`

Surface recognition algorithm 'Carpet' according to Turpeinen - Interface Detection Using a Quenched-Noise Version of the Edwards-Wilkinson Equation. The algorithm places a surface above (alternatively below) the image, and moves it towards larger (alternatively smaller) z values while controlling its dynamics according to the Edwards-Wilkinson equation. The movement of the surface stops when it encounters enough pixels with value above specific stopping gray level. The surface does not move through small holes in the object as it has controllable amount of surface tension.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

geometry [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Geometry image. This image does not need to be binary image (but it can be).

height map [input & output]

Data type: float32 image

Height map that defines the initial position of the surface, or an empty image. The height map gives z-position of the surface for each (x, y)-position of the geometry image. The final position of the surface is also saved into this height map. The size of the height map must be $w \times h$ where w and h are the width and the height of the geometry image.

stopping gray value [input]

Data type: real

The movement of the surface stops when it encounters pixels whose value is above this gray level.

direction [input]

Data type: string

Default value: Down

Direction where the surface moves. 'Down' corresponds to direction towards larger z values, and 'Up' corresponds to direction towards smaller z values.

surface tension [input]

Data type: real

Default value: 1

Value that indicates how smooth the surface will be. Larger value results in smoother surface.

iterations [input]

Data type: positive integer

Default value: 150

Count of iterations to perform.

max step [input]

Data type: real

Default value: 1

Maximum amount of movement allowed in one time step.

See also

findsurface, *drawheightmap*, *setbeforeheightmap*, *setafterheightmap*, *shiftz*

3.8.66 flip

Syntax: `flip(image, dimension)`

Flips image in the given dimension, e.g. if dimension is zero, the left edge of the image becomes the right edge and vice versa.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to process.

dimension [input]

Data type: positive integer

The dimension to flip. Zero corresponds to x, one to y, etc.

See also

rot90cw, *rot90ccw*, *rotate*, *flip*, *reslice*, *crop*, *copy*, *scalelabels*

3.8.67 floodfill

There are 2 forms of this command.

`floodfill(image, start points, fill value, connectivity)`

Performs flood fill. Fills start point and all its neighbours and their neighbours etc. recursively as long as the color of the pixel to be filled equals color of the start point.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

start points [input]

Data type: float32 image

List of start points for the fill.

fill value [input]

Data type: real

Fill color.

connectivity [input]

Data type: connectivity

Default value: All

Connectivity of the region to fill. Can be Nearest for connectivity to nearest neighbours only, or All for connectivity to all neighbours.

See also

grow, *growlabels*, *floodfill*, *regionremoval*, *morphorec*

`floodfill(image, start point, fill value, connectivity)`

Performs flood fill. Fills start point and all its neighbours and their neighbours etc. recursively as long as the color of the pixel to be filled equals color of the start point.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

start point [input]

Data type: 3-component integer vector

Starting point for the fill.

fill value [input]

Data type: real

Fill color.

connectivity [input]

Data type: connectivity

Default value: All

Connectivity of the region to fill. Can be Nearest for connectivity to nearest neighbours only, or All for connectivity to all neighbours.

See also

grow, *growlabels*, *floodfill*, *regionremoval*, *morphorec*

3.8.68 floor

Syntax: `floor(image)`

Calculates floor of pixel values, i.e. the least integer greater than or equal to the pixel value.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

3.8.69 frangifilter

Syntax: `frangifilter(input image, output image, spatial sigma, output scale, c, gamma, alpha, beta)`

Calculates line-enhancing filter Vo according to Frangi - Multiscale vessel enhancement filtering.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Output image.

spatial sigma [input]

Data type: real

Standard deviation of Gaussian kernel, determines scale of structures that are probed.

output scale [input]

Data type: real

Default value: 0

Output values are scaled by this number. Pass in zero to scale output value 1 to maximum of the data type for integer data types and to 1 for floating point data types.

c [input]

Data type: real

Default value: 0.25

Sensitivity of the filter to deviation from background noise. Typical value is quarter of the value of the maximum intensity of the lines.

gamma [input]

Data type: real

Default value: 0

Scale-space scaling exponent. Set to zero to disable scaling.

alpha [input]

Data type: real

Default value: 0.5

Sensitivity of the filter to deviation from plate-like structures.

beta [input]

Data type: real

Default value: 0.5

Sensitivity of the filter to deviation from blob-like structures.

See also

satofilter

3.8.70 gaussfilter

Syntax: `gaussfilter(input image, output image, spatial sigma, boundary condition, allow optimization)`

Gaussian blurring. Removes imaging noise but makes images look unsharp. If optimization flag is set to true, processes integer images with more than 8 bits of resolution with separable convolution and floating point images with FFT filtering. If optimization flag is set to false, processes all integer images with normal convolution and floating point images with separable convolution.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Output image.

spatial sigma [input]

Data type: 3-component real vector

Standard deviation of Gaussian kernel.

boundary condition [input]

Data type: boundary condition

Default value: Nearest

Type of boundary condition. Zero indicates that values outside of image bounds are taken to be zero. Nearest indicates that the nearest value inside the image is to be used in place of values outside of image bounds.

allow optimization [input]

Data type: boolean

Default value: True

Set to true to allow optimized processing of images with high dynamic range data types. Might cause small deviations from true filtering result and edge artefacts.

See also

gaussfilter, *bilateralfilter*, *bilateralfilterapprox*, *vawefilter*, *openingfilter*, *closingfilter*, *minfilter*, *maxfilter*, *medianfilter*, *variancefilter*, *stddevfilter*, *bandpassfilter*, *highpassfilter*

3.8.71 generictransform

Syntax: `generictransform(image, transformed image, position, reference points, deformed points, exponent)`

Transforms image based on point-to-point correspondence data. Transformation between the points is interpolated from the point data using inverse distance interpolation.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image that will be transformed.

transformed image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

The result of the transformation is set to this image. Size of this image must be set before calling this command.

position [input]

Data type: 3-component integer vector

Position of the transformed image in coordinates of the original.

reference points [input]

Data type: float32 image

Points in the original image as 3xN image where each row contains (x, y, z)-coordinates of a single point, and there are N points in total.

deformed points [input]

Data type: float32 image

Locations of points in reference points image after the deformation has been applied. Encoded similarly to reference points image.

exponent [input]

Data type: real

Default value: 2.5

Smoothing exponent in the inverse distance interpolation. Smaller values smooth more.

3.8.72 get

Syntax: `get(image, output, positions, block origin)`

Reads multiple pixels from an input image. In distributed operating mode, the output and positions images must fit into the RAM.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image where the pixels are read from.

output [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Values of the pixels read from the input image are stored in this image. The size of the image will be set to the number of pixels read.

positions [input]

Data type: float32 image

Positions of pixels to read. Each row of this image contains (x, y, z) coordinates of a pixel to read from the input image. The size of the image must be 3xN where N is the count of pixels to read.

block origin [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

Origin of current calculation block in coordinates of the full image. This argument is used internally in distributed processing. Set to zero in normal usage.

See also

set, *getpointsandlines*

3.8.73 getmapfile

Syntax: `getmapfile(image, mapfile)`

Get a path to the file where the argument image has been memory-mapped to. Returns empty string if no mapping has been made for the argument image.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to process.

mapfile [output]

Data type: string

The name and path to the memory-mapped file.

See also

mapraw

3.8.74 getmeta

Syntax: `getmeta(image, key, value, column, row, default)`

Gets metadata item from an image.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to process.

key [input]

Data type: string

Name of the metadata item.

value [output]

Data type: string

Value of the metadata item is placed into this string.

column [input]

Data type: positive integer

Default value: 0

Column index of the item to retrieve from the data matrix.

row [input]

Data type: positive integer

Default value: 0

Row index of the item to retrieve from the data matrix.

default [input]

Data type: string

Default value: “”

This value is returned if the key is not found

See also

setmeta, *getmeta*, *writemeta*, *readmeta*, *clearmeta*, *listmeta*, *copymeta*, *metarowcount*, *metacolumncount*

3.8.75 getpointsandlines

Syntax: `getpointsandlines(vertices, edges, edge measurements, edge points, points, lines, smoothing sigma, max displacement)`

Converts a traced skeleton to points-and-lines format, where all points on the skeleton are stored in one array and all branches are stored as lists of indices of points that form the branch.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

vertices [input]

Data type: float32 image

Image where vertex coordinates are stored. See *tracelineskeleton* command.

edges [input]

Data type: uint64 image

Image where vertex indices corresponding to each edge are stored. See *tracelineskeleton* command.

edge measurements [input]

Data type: float32 image

Image where properties of each edge are stored. See *tracelineskeleton* command.

edge points [input]

Data type: int32 image

Image that stores some points on each edge. See *tracelineskeleton* command.

points [output]

Data type: float32 image

Image that stores coordinates of all points in the network. These include points on edges and centroids of intersection regions. Each row of image stores (x, y, z) coordinates of one point.

lines [output]

Data type: uint64 image

Image that stores a list of point indices for each edge. This image is stored in compressed format: [count of edges][count of points in 1st edge][index of point 1][index of point 2]...[count of points in 2nd edge][index of point 1][index of point 2]...

smoothing sigma [input]

Data type: real

Default value: 0

If smooth lines are requested instead of jagged lines (consisting of pixel locations), specify positive value here. The value is standard deviation of a Gaussian kernel used to smooth the lines in an anchored convolution smoothing. The end points of each line are not changed by smoothing. Values in range [0.5, 1.5] often give suitable amount of smoothing. The smoothing algorithm is described in Suhadolnik - An anchored discrete convolution algorithm for measuring length in digital images.

max displacement [input]

Data type: real

Default value: 0.5

Maximum displacement of points in anchored convolution smoothing of the lines.

See also

writevtk, *cleanskeleton*, *pruneskeleton*, *removeedges*, *fillskeleton*, *getpointsandlines*

3.8.76 gradient

Syntax: `gradient(f, spatial sigma, dfdx, dfdy, dfdz, gamma)`

Calculates Gaussian gradient $(\partial f/\partial x, \partial f/\partial y, \partial f/\partial z)$ of an image f. Each of the derivatives is calculated by convolving the image with the corresponding derivative of the Gaussian function.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

f [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image whose gradient is to be calculated.

spatial sigma [input]

Data type: real

Standard deviation of Gaussian kernel.

dfdx [output]

Data type: float32 image

Derivative in x-direction

dfdy [output]

Data type: float32 image

Derivative in y-direction

dfdz [output]

Data type: float32 image

Derivative in z-direction

gamma [input]

Data type: real

Default value: 0

Scale-space scaling exponent according to Lindeberg. Set to zero to disable scaling.

See also

derivative, *gradientmagnitude*

3.8.77 gradientmagnitude

Syntax: `gradientmagnitude(input image, output image, spatial sigma, gamma)`

Calculates norm of Gaussian gradient of image. The output image can be the same than the input image.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: float32 image

Output image.

spatial sigma [input]

Data type: real

Default value: 1

Standard deviation of Gaussian derivative kernel.

gamma [input]

Data type: real

Default value: 0

Scale-space scaling exponent. Set to zero to disable scaling.

3.8.78 greedycoloring

Syntax: `greedycoloring(image, connectivity)`

Perform greedy coloring of regions. Colors each region in image such that its neighbours are colored with different colors, and uses as little colors as possible. Uses greedy algorithm so the count of colors used might not be minimal. Assumes background to have value 0 and colors all non-zero regions.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

connectivity [input]

Data type: connectivity

Default value: All

Connectivity of the regions. Can be Nearest for connectivity to nearest neighbours only, or All for connectivity to all neighbours.

See also

analyzeparticles, *listanalyzers*, *headers*, *fillparticles*, *drawellipsoids*, *label*, *analyzelabels*, *regionremoval*, *greedycoloring*, *csa*

3.8.79 grow

There are 2 forms of this command.

`grow(image, source color, target color, connectivity)`

Grows regions with source color to regions with target color as much as possible.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

source color [input]

Data type: real

Color that defines regions that are going to be grown.

target color [input]

Data type: real

Color where the regions will grow.

connectivity [input]

Data type: connectivity

Default value: Nearest

Connectivity of the regions to grow. Can be Nearest for connectivity to nearest neighbours only, or All for connectivity to all neighbours.

See also

grow, *growlabels*, *floodfill*, *regionremoval*, *morphorec*

`grow(image, parameter image)`

Grows regions from seed points outwards. Seeds points are all nonzero pixels in the input image, pixel value defining region label. Each seed is grown towards surrounding zero pixels. Fill priority for each pixel is read from the corresponding pixel in the parameter image. Pixels for which priority is zero or negative are never filled. This process is equal to Meyer's watershed algorithm for given set of seeds, and watershed cuts are borders between filled regions in the output image.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

parameter image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image

Parameter image.

See also

grow, *growlabels*, *floodfill*, *regionremoval*, *morphorec*

3.8.80 growlabels

Syntax: `growlabels(image, allowed color, background color, connectivity)`

Grows all colored regions as much as possible into pixels that have a specific color. In practice, this command first finds all unique colors in the image, and uses each set of pixels having the same color as seed points for a flood fill that proceeds to pixels whose value is given in the 'allowed color' argument.

This growing method is suited only for situations where separate parts of the original structure are labelled and the labels must be grown back to the original structure. **If there are multiple labels in a connected component, non-labeled pixels are assigned the smallest label in the non-distributed version and (mostly) random label among all the possibilities in the distributed version.** Therefore, **this function is suited only for images containing separate blobs or particles**, where each particle contains seed point(s) of only single value.

An alternative to this command is *morphorec*. It works such that each pixel will get the label of the nearest labeled pixel.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

allowed color [input]

Data type: real

Color where other colors will be grown into.

background color [input]

Data type: real

Background color. Values of pixels having this color are not changed. Set to the same value than allowed color to fill to all pixels.

connectivity [input]

Data type: connectivity

Default value: Nearest

Connectivity of the regions to grow. Can be Nearest for connectivity to nearest neighbours only, or All for connectivity to all neighbours.

See also

grow, *growlabels*, *floodfill*, *regionremoval*, *morphorec*

3.8.81 headers

There are 2 forms of this command.

`headers(analyzers)`

Shows the column headers of a particle analysis result table.

This command can be used in the distributed processing mode, but it does not participate in distributed processing.

Arguments

analyzers [input]

Data type: string

List of names of analyzers that were used. Use the same value that was passed to *analyzeparticles* command. Separate the analyzer names with any non-alphanumeric character sequence.

See also

analyzeparticles, *listanalyzers*, *headers*, *fillparticles*, *drawellipsoids*, *label*, *analyzelabels*, *regionremoval*, *greedycoloring*, *csa*

`headers(analyzers, value)`

Gets a comma-separated list of the column headers of particle analysis result table.

This command can be used in the distributed processing mode, but it does not participate in distributed processing.

Arguments

analyzers [input]

Data type: string

List of names of analyzers that were used. Use the same value that was passed to *analyzeparticles* command. Separate the analyzer names with any non-alphanumeric character sequence.

value [output]

Data type: string

The headers are placed into this string as a comma-separated list.

See also

analyzeparticles, *listanalyzers*, *headers*, *fillparticles*, *drawellipsoids*, *label*, *analyzelabels*, *regionremoval*, *greedycoloring*, *csa*

3.8.82 hello

Syntax: `hello(name)`

Shows a greetings message. This command is used for testing things.

This command can be used in the distributed processing mode, but it does not participate in distributed processing.

Arguments

name [input]

Data type: string

Default value: there

Name of caller. Omit to output a generic greeting.

See also

help, *info*, *license*, *echo*, *print*, *waitreturn*, *hello*, *timing*, *savetiming*, *resettiming*

3.8.83 help

Syntax: `help(topic, format)`

Shows usage information.

This command can be used in the distributed processing mode, but it does not participate in distributed processing.

Arguments

topic [input]

Data type: string

Default value: “”

Name of command whose help is to be retrieved. Specify nothing to show general help.

format [input]

Data type: string

Default value: text

Output format. Can be text for textual output or rst for ReStructuredText output. The latter is more complex format.

See also

help, *info*, *license*, *echo*, *print*, *waitreturn*, *hello*, *timing*, *savetiming*, *resettiming*

3.8.84 highpassfilter

Syntax: `highpassfilter(input image, output image, spatial sigma, shift, boundary condition, allow optimization)`

Gaussian high-pass filtering. Use to remove smooth, large-scale gray-scale variations from the image.

Subtracts a Gaussian filtered version of input from itself. If optimization flag is set to true, processes integer images with more than 8 bits of resolution with separable convolution and floating point images with FFT filtering. If optimization flag is set to false, processes all integer images with normal convolution and floating point images with separable convolution.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Output image.

spatial sigma [input]

Data type: 3-component real vector

Standard deviation of Gaussian kernel.

shift [input]

Data type: real

Default value: 0

Constant to be added to the pixel values. Use to set the mean of the filtered image to a desired value. Useful especially when filtering unsigned images where non-shifted highpass filtering will lead to negative values that will be clipped to zero.

boundary condition [input]

Data type: boundary condition

Default value: Nearest

Type of boundary condition. Zero indicates that values outside of image bounds are taken to be zero. Nearest indicates that the nearest value inside the image is to be used in place of values outside of image bounds.

allow optimization [input]

Data type: boolean

Default value: True

Set to true to allow optimized processing of images with high dynamic range data types. Might cause small deviations from true filtration result, and edge artefacts.

See also

gaussfilter, *bilateralfilter*, *bilateralfilterapprox*, *vawefilter*, *openingfilter*, *closingfilter*, *minfilter*, *maxfilter*, *medianfilter*, *variancefilter*, *stddevfilter*, *bandpassfilter*, *highpassfilter*

3.8.85 hist

Syntax: `hist(input image, histogram, bin starts, histogram minimum, histogram maximum, bin count, output file, block index)`

Calculate histogram of an image.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image whose histogram will be calculated.

histogram [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image

Image that will contain the histogram on output. The number of pixels in this image will be set to match bin count. Please use pixel data type that can hold large enough values, e.g. float32 or uint64.

bin starts [output]

Data type: float32 image

Image that will contain the histogram bin start coordinates when the function finishes.

histogram minimum [input]

Data type: real

Default value: 0

Minimum value to be included in the histogram. Values less than minimum are included in the first bin.

histogram maximum [input]

Data type: real(255), real(65535), real(4294967295), real(1.844674407370955e+19), real(127), real(32767), real(2147483647), real(9.223372036854776e+18), real(3.402823466385289e+38)

Default value: Shown along data types.

The end of the last bin. Values greater than maximum are included in the last bin.

bin count [input]

Data type: integer

Default value: 256

Count of bins. The output image will be resized to contain this many pixels.

output file [input]

Data type: string

Default value: “”

Name of file where the histogram data is to be saved. Specify empty string to disable saving. This argument is used internally in distributed processing, but can be used to (conveniently?) save the histogram in .raw format.

block index [input]

Data type: integer

Default value: -1

Index of image block that we are currently processing. This argument is used internally in distributed processing and should normally be set to negative value. If positive, this number is appended to output file name.

3.8.86 hist2

Syntax: `hist2(first input image, min 1, max 1, bin count 1, second input image, min 2, max 2, bin count 2, histogram, bin starts 1, bin starts 2, output file, block index)`

Calculate bivariate histogram of two images. In the bivariate histogram position (i, j) counts number of locations where the value of the first input image is i and the value of the second input image is j (assuming minimum = 0, maximum = data type max, and bin size = 1). If image sizes are different, only the region available in both images is included in the histogram. In this case, a warning message is shown.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

first input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

First image whose histogram will be calculated.

min 1 [input]

Data type: real

Default value: 0

Minimum value of first image to be included in the histogram.

max 1 [input]

Data type: real(255), real(65535), real(4294967295), real(1.844674407370955e+19), real(127), real(32767), real(2147483647), real(9.223372036854776e+18), real(3.402823466385289e+38)

Default value: Shown along data types.

The end of the last bin. This is the smallest value above minimum that is not included in the histogram.

bin count 1 [input]

Data type: integer

Default value: 256

Count of bins. The first dimension of the output image will be resized to contain this many pixels.

second input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image

Second image whose histogram will be calculated.

min 2 [input]

Data type: real

Default value: 0

Minimum value of second image to be included in the histogram.

max 2 [input]

Data type: real(255), real(65535), real(4294967295), real(1.844674407370955e+19), real(3.402823466385289e+38), real(127), real(32767), real(2147483647), real(9.223372036854776e+18)

Default value: Shown along data types.

The end of the last bin. This is the smallest value above minimum that is not included in the histogram.

bin count 2 [input]

Data type: integer

Default value: 256

Count of bins. The second dimension of the output image will be resized to contain this many pixels.

histogram [output]

Data type: float32 image

Image that will contain the histogram on output.

bin starts 1 [output]

Data type: float32 image

Image that will contain the histogram bin start coordinates in the first dimension when the function finishes.

bin starts 2 [output]

Data type: float32 image

Image that will contain the histogram bin start coordinates in the second dimension when the function finishes.

output file [input]

Data type: string

Default value: “”

Name of file where the histogram data is to be saved. Specify empty string to disable saving. This argument is used internally in distributed processing, but can be used to (conveniently?) save the histogram in .raw format.

block index [input]

Data type: integer

Default value: -1

Index of image block that we are currently processing. This argument is used internally in distributed processing and should normally be set to negative value. If positive, this number is appended to output file name.

3.8.87 ifft

Syntax: `ifft(input image, output image)`

Calculates inverse Fourier transform. The size of the output image must be set to the size of the original data where the FFT was calculated from. The input image will be used as a temporary buffer and its contents are therefore destroyed.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

input image [input]

Data type: complex32 image

Input image.

output image [output]

Data type: float32 image

Output image.

See also

fft, *ifft*, *bandpassfilter*, *highpassfilter*

3.8.88 imag

Syntax: `imag(image)`

Calculates imaginary part of pixel values.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: complex32 image

Image to process.

3.8.89 info

Syntax: `info()`

Displays information about the computer and the PI system.

This command can be used in the distributed processing mode, but it does not participate in distributed processing.

See also

help, *license*

3.8.90 inpaintg

Syntax: `inpaintg(image, flag)`

Replaces pixels that have specific flag value by a value interpolated from the neighbouring pixels that do not have the specific flag value..

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

flag [input]

Data type: real

Default value: 0

Values of pixels that have this (flag) value are inpainted.

See also

inpaintn, *inpaintg*

3.8.91 inpaintn

Syntax: `inpaintn(image, flag)`

Replaces pixels that have specific flag value by the value of the nearest pixel that does not have the specific flag value.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to process.

flag [input]

Data type: real

Default value: 0

Values of pixels that have this (flag) value are inpainted.

See also

inpaintn, *inpaintg*

3.8.92 inv

Syntax: `inv(image)`

Calculates inverse (1/x) of pixel values.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to process.

3.8.93 invsubtract

There are 2 forms of this command.

```
invsubtract(image, parameter image, allow broadcast)
```

Subtracts two images in inverse order, and places output to the first image. In other words, first image = second image - first image. The operation is performed using saturation arithmetic.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to process.

parameter image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image, complex32 image

Parameter image.

allow broadcast [input]

Data type: boolean

Default value: False

Set to true to allow size of parameter image differ from size of input image. If there is a need to access pixel outside of parameter image, the nearest value inside the image is taken instead. If set to false, dimensions of input and parameter images must be equal. If set to true, the parameter image is always loaded in its entirety in distributed processing mode.

invsubtract(image, x)

Subtracts the first image from a constant, and places the result to the first image. The operation is performed using saturation arithmetic.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

x [input]

Data type: real

Constant from which the image is subtracted.

3.8.94 isimagefile

Syntax: `isimagefile(filename, result)`

Checks if a file with given name is an image file.

Note: In Python/pi2py2, the result parameter is not specified, but the test result is returned as a boolean return value.

This command can be used in the distributed processing mode, but it does not participate in distributed processing.

Arguments

filename [input]

Data type: string

Name of image file.

result [input]

Data type: uint8 image

Set to 1 if the given file name is a readable image file, and to 0 otherwise.

3.8.95 label

Syntax: `label(image, region color, connectivity)`

Labels distinct regions in the image with individual colors. The regions do not need to be separated by background. Execution is terminated in an error if the pixel data type does not support large enough values to label all the particles.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

region color [input]

Data type: real

Default value: 0

Current color of regions that should be labeled. Set to zero to label all non-zero regions.

connectivity [input]

Data type: connectivity

Default value: Nearest

Connectivity of the particles. Can be Nearest for connectivity to nearest neighbours only, or All for connectivity to all neighbours.

See also

analyzeparticles, *listanalyzers*, *headers*, *fillparticles*, *drawellipsoids*, *label*, *analyzelabels*, *regionremoval*, *greedycoloring*, *csa*

3.8.96 labelmaxima

Syntax: `labelmaxima(image, regions)`

Draws all regions in the given list, each with different color. If there are not enough colors available, an error is shown.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image where the maxima are to be drawn.

regions [input]

Data type: int32 image

Image that contains the regions. The format of this image is described in the *localmaxima*' command.

See also

localmaxima, *cleanmaxima*, *labelmaxima*, *growlabels*, *grow*

3.8.97 license

Syntax: `license()`

Displays license of this program and PI system.

This command can be used in the distributed processing mode, but it does not participate in distributed processing.

See also

help, *info*, *license*, *echo*, *print*, *waitreturn*, *hello*, *timing*, *savetiming*, *resettiming*

3.8.98 line

Syntax: `line(image, start, end, value, block origin)`

Draws a single-pixel wide line into the image.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

start [input]

Data type: 3-component real vector

Default value: "[0, 0, 0]"

Start position of the line.

end [input]

Data type: 3-component real vector

Default value: "[1, 1, 1]"

End position of the line.

value [input]

Data type: real

Default value: 1

Value for pixels of the line.

block origin [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

Origin of current calculation block in coordinates of the full image. This argument is used internally in distributed processing. Set to zero in normal usage.

See also

line, *capsule*, *sphere*, *ellipsoid*, *set*, *get*, *ramp*, *ramp3*

3.8.99 lineskeleton

Syntax: `lineskeleton(image)`

Calculates skeleton of the foreground of the given image. Positive pixels are assumed to belong to the foreground. The skeleton contains only lines (no plates). This command is not guaranteed to give the same result in both normal and distributed processing mode. Despite that, both modes should give a valid result.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

See also

surfacethin, *surfaceskeleton*, *linethin*, *lineskeleton*, *tracelineskeleton*, *classifyskeleton*

3.8.100 linethin

Syntax: `linethin(image)`

Thins one layer of pixels from the foreground of the image. Positive pixels are assumed to belong to the foreground. Run iteratively to calculate a line skeleton. This command is not guaranteed to give the same result in both normal and distributed processing mode. Despite that, both modes should give a valid result.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

See also

surfacethin, *surfaceskeleton*, *linethin*, *lineskeleton*, *tracelineskeleton*, *classifyskeleton*

3.8.101 linmap

Syntax: `linmap(image, input min, input max, output min, output max)`

Maps pixel values linearly from one range to another. Maps a value x in range [input min, input max] linearly to a value in range [output min, output max]. E.g. if x = 0.5 and [input min, input max] = [0, 1] and [output min, output max] = [3, 4], result will be 3 + (0.5 - 0) / (1 - 0) * (4 - 3) = 3.5.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

input min [input]

Data type: real

Minimum of the input range.

input max [input]

Data type: real

Maximum of the input range.

output min [input]

Data type: real

Minimum of the output range.

output max [input]

Data type: real

Maximum of the output range.

3.8.102 list

Syntax: `list()`

Displays information about images loaded to memory, or images prepared for distributed processing.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

See also

help, *info*, *license*, *echo*, *print*, *waitreturn*, *hello*, *timing*, *savetiming*, *resettiming*

3.8.103 listanalyzers

Syntax: `listanalyzers()`

Shows names of all available analysis methods that can be used in conjunction with *analyzeparticles* or *csa* commands. See also *headers* command.

This command can be used in the distributed processing mode, but it does not participate in distributed processing.

See also

analyzeparticles, *listanalyzers*, *headers*, *fillparticles*, *drawellipsoids*, *label*, *analyzelabels*, *regionremoval*, *greedycoloring*, *csa*

3.8.104 listmeta

Syntax: `listmeta(image, names)`

Builds a comma-separated list of the names of all the metadata items in an image.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to process.

names [output]

Data type: string

Names of metadata items will be stored in this string.

See also

setmeta, *getmeta*, *writemeta*, *readmeta*, *clearmeta*, *listmeta*, *copymeta*, *metarowcount*, *metacolumncount*

3.8.105 localmaxima

Syntax: `localmaxima(input image, output image, connectivity)`

Finds local maxima in the input image. Maxima migh be individual pixels or larger regions that have the same value and that are bordered by pixels of smaller value. The output image will be in format [count of regions][count of pixels in region 1][x1][y1][z1][zx2][y2][z2]...[count of items in region 2][x1][y1][z1][zx2][y2][z2]...

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: int32 image

Output image.

connectivity [input]

Data type: connectivity

Default value: All

Connectivity of the maxima regions. Can be Nearest for connectivity to nearest neighbours only, or All for connectivity to all neighbours.

See also

localmaxima, *cleanmaxima*, *labelmaxima*, *growlabels*, *grow*

3.8.106 localthreshold

Syntax: `localthreshold(input image, output image, radius, method, argument 1, argument 2, argument 3, argument 4, boundary condition)`

Local thresholding. Determines threshold value for a pixel based on its neighbourhood. The supported thresholding methods are

Otsu

Otsu's thresholding algorithm.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Described in N. Otsu, A Threshold Selection Method from Gray-Level Histograms, IEEE Transactions on Systems, Man, and Cybernetics 9(1), 1979.

Original C++ code by Jordan Bevik, ported to ImageJ plugin by Gabriel Landini, and finally ported to pi2.

Huang

Huang's fuzzy thresholding method.

Uses Shannon's entropy function (one can also use Yager's entropy function).

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Described in L.-K. Huang and M.-J.J. Wang, Image Thresholding by Minimizing the Measures of Fuzziness, Pattern Recognition 28(1), 1995.

Original code by M. Emre Celebi, ported to ImageJ plugin by G. Landini from E. Celebi's fourier_0.8 routines, then ported to pi2.

Intermodes

Assumes a bimodal histogram. The histogram needs is iteratively smoothed using a running average of size 3 until there are only two local maxima at j and k. The threshold t is (j+k)/2.

Images with histograms having extremely unequal peaks or a broad and flat valleys are unsuitable for this method.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Described in J. M. S. Prewitt and M. L. Mendelsohn, The analysis of cell images, Annals of the New York Academy of Sciences 128, 1966.

Ported to ImageJ plugin by Gabriel Landini from Antti Niemisto's Matlab code (GPL), and then to pi2.

IsoData

Iterative procedure based on the isodata algorithm described in T.W. Ridler and S. Calvard, Picture thresholding using an iterative selection method, IEEE Transactions on System, Man and Cybernetics, SMC-8, 1978.

The procedure divides the image into objects and background by taking an initial threshold, then the averages of the pixels at or below the threshold and pixels above are computed. The averages of those two values are computed, the threshold is incremented and the process is repeated until the threshold is larger than the composite average. That is, threshold = (average background + average objects) / 2

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

The code implementing this method is probably originally from NIH Image, then ported to ImageJ, and then to pi2.

Li

Li's Minimum Cross Entropy thresholding method.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

This implementation is based on the iterative version of the algorithm, described in C.H. Li and P.K.S. Tam, An Iterative Algorithm for Minimum Cross Entropy Thresholding, Pattern Recognition Letters 18(8), 1998.

Ported to ImageJ plugin by G.Landini from E Celebi's fourier_0.8 routines, and then to pi2.

MaxEntropy

Implements Kapur-Sahoo-Wong (Maximum Entropy) thresholding method described in J.N. Kapur, P.K. Sahoo, and A.K.C Wong, A New Method for Gray-Level Picture Thresholding Using the Entropy of the Histogram, Graphical Models and Image Processing 29(3), 1985.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Original code by M. Emre Celebi, ported to ImageJ plugin by G.Landini, then ported to pi2.

Mean

The threshold is shifted mean of the greyscale data, i.e. $t = \mu - c$, where t is the threshold, μ is the mean of the image or the neighbourhood, and c is the shift.

First argument is the shift value c.

Described in C. A. Glasbey, An analysis of histogram-based thresholding algorithms, CVGIP: Graphical Models and Image Processing 55, 1993.

MinError

Implements minimum error thresholding method described in J. Kittler and J. Illingworth, Minimum error thresholding, Pattern Recognition 19, 1986.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Code is originally from Antti Niemisto's Matlab code (GPL), then ported to ImageJ by Gabriel Landini, and then to pi2.

Minimum

Assumes a bimodal histogram. The histogram needs is iteratively smoothed (using a running average of size 3) until there are only two local maxima. Threshold t is such that $y_{t-1} > y_t$ and $y_t \leq y_{t+1}$. Images with histograms having extremely unequal peaks or a broad and flat valleys are unsuitable for this method.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Described in J. M. S. Prewitt and M. L. Mendelsohn, The analysis of cell images, Annals of the New York Academy of Sciences 128, 1966.

Original Matlab code by Antti Niemisto, ported to ImageJ by Gabriel Landini, then ported to pi2.

Moments

Attempts to preserve the moments of the original image in the thresholded result.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Described in W. Tsai, Moment-preserving thresholding: a new approach, Computer Vision, Graphics, and Image Processing 29, 1985.

Original code by M. Emre Celebi ported to ImageJ by Gabriel Landini, and then to pi2.

Percentile

Assumes the fraction of foreground pixels to be given value.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram. The fourth argument is the fraction of foreground pixels.

Described in W. Doyle, Operation useful for similarity-invariant pattern recognition, Journal of the Association for Computing Machinery 9, 1962.

Original code by Antti Niemisto, ported to ImageJ by Gabriel Landini, then to pi2.

RenyiEntropy

Similar to the MaxEntropy method, but using Renyi's entropy instead.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Described in J.N. Kapur, P.K. Sahoo, and A.K.C Wong, A New Method for Gray-Level Picture Thresholding Using the Entropy of the Histogram, Graphical Models and Image Processing 29(3), 1985.

Original code by M. Emre Celebi, ported to ImageJ plugin by G.Landini, then ported to pi2.

Shanbhag

Described in A.G. Shanhbag, Utilization of Information Measure as a Means of Image Thresholding, Graphical Models and Image Processing 56(5), 1994.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Original code by M. Emre Celebi, ported to ImageJ plugin by Gabriel Landini, then to pi2.

Triangle

The triangle algorithm assumes a peak near either end of the histogram, finds minimum near the other end, draws a line between the minimum and maximum, and sets threshold t to a value for which the point $(t, y(t))$ is furthest away from the line (where y is the histogram).

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Described in G.W. Zack, W.E. Rogers, and S.A. Latt, Automatic Measurement of Sister Chromatid Exchange Frequency, Journal of Histochemistry and Cytochemistry 25(7), 1977.

Original code by Johannes Schindelin, modified by Gabriel Landini, then ported to pi2.

Yen

Yen thresholding method described in J.C. Yen, F.K. Chang, and S. Chang, A New Criterion for Automatic Multilevel Thresholding, IEEE Transactions on Image Processing 4(3), 1995.

The threshold value is calculated from the histogram of the image. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram.

Original code by M. Emre Celebi, ported to ImageJ plugin by Gabriel Landini, then to pi2.

Median

The threshold is shifted median of the greyscale data, i.e. $t = M - c$, where t is the threshold, M is the median of the image, and c is the shift.

The median is calculated from image histogram, so its accuracy might degrade if too small bin count is used. The first and the second arguments are values corresponding to the minimum and maximum of the histogram. Out-of-range values will be placed to the first and the last bins, respectively. The third argument is the count of bins in the histogram. The fourth argument is the shift value c.

MidGrey

The threshold is $(m + M)/2 - c$, where m and M are the minimum and the maximum value of the image or neighbourhood, respectively. The value c is a user-specified shift.

The first argument is the shift value c.

Niblack

Niblack's thresholding method.

This method is mostly suited for local thresholding.

The threshold is $\mu+k*\sigma-c$, where μ and σ are the mean and the standard deviation of the image or the neighbourhood, respectively. The values k and c are user-specified scaling constant and shift.

The first argument is the scaling constant k. The second argument is the shift value c.

Described in W. Niblack, An introduction to Digital Image Processing, Prentice-Hall, 1986.

Phansalkar

Phansalkar's thresholding method.

This method is mostly suited for local thresholding.

Threshold is $\mu * (1.0 + p * \exp(-q * \mu) + k * (\sigma/r - 1))$ where μ and σ are the mean and the standard deviation of the image or the neighbourhood, respectively. The values k, r, p, and q are user-specified parameters of the method. The default values are $k = 0.25$, $r = 0.5$, $p = 2.0$, and $q = 10.0$.

The first four arguments are the parameters k, r, p, and q.

Described in N. Phansalskar, S. More, and A. Sabale, et al., Adaptive local thresholding for detection of nuclei in diversity stained cytology images, International Conference on Communications and Signal Processing (ICCSP), 2011.

Sauvola

Sauvola's thresholding method.

This method is mostly suited for local thresholding.

The threshold is $\mu * (1 + k * (\sigma/r - 1))$, where μ and σ are the mean and the standard deviation of the image or the neighbourhood, respectively. The values k and r are user-specified scaling constants.

The first argument is the scaling constant k. The second argument is the scaling constant r.

Described in Sauvola, J and Pietaksinen, M, Adaptive Document Image Binarization, Pattern Recognition 33(2), 2000.

Bernsen

Finds Bernsen's thresholding method.

This method is mostly suited for local thresholding.

The method uses a user-provided contrast threshold. If the local contrast (max - min) is above or equal to the contrast threshold, the threshold is set at the local midgrey value (the mean of the minimum and maximum grey values in the local window (or whole image in the case of global thresholding)). If the local contrast is below the contrast threshold the neighbourhood is considered to consist only of one class and the pixel is set to object or background depending on the value of the midgrey.

The first argument is the local contrast threshold.

Described in J. Bernsen, Dynamic Thresholding of Grey-Level Images, Proceedings of the 8th International Conference on Pattern Recognition, 1986.

The original code is written by Gabriel Landini, and it has been ported to pi2.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Output image.

radius [input]

Data type: 3-component integer vector

Default value: "[1, 1, 1]"

Radius of neighbourhood. Diameter will be $2r + 1$.

method [input]

Data type: string

Default value: Otsu

Thresholding method that will be applied.

argument 1 [input]

Data type: real

Default value: nan

Argument for the thresholding method. The purpose of this argument depends on the method, see the list above. Specify nan in order to use a method-specific default value.

argument 2 [input]

Data type: real

Default value: nan

Argument for the thresholding method. The purpose of this argument depends on the method, see the list above. Specify nan in order to use a method-specific default value.

argument 3 [input]

Data type: real

Default value: nan

Argument for the thresholding method. The purpose of this argument depends on the method, see the list above. Specify nan in order to use a method-specific default value.

argument 4 [input]

Data type: real

Default value: nan

Argument for the thresholding method. The purpose of this argument depends on the method, see the list above. Specify nan in order to use a method-specific default value.

boundary condition [input]

Data type: boundary condition

Default value: Nearest

Type of boundary condition. Zero indicates that values outside of image bounds are taken to be zero. Nearest indicates that the nearest value inside the image is to be used in place of values outside of image bounds.

See also

autothreshold, *threshold*

3.8.107 log

Syntax: `log(image)`

Calculates natural logarithm of pixel values.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to process.

3.8.108 log10

Syntax: `log10(image)`

Calculates base-10 logarithm of pixel values.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to process.

3.8.109 mainorientationcolor

Syntax: `mainorientationcolor(geometry, phi, theta, phim, thetam, r, g, b)`

Color codes orientation data according to deviation from a given main orientation. In the output, hue describes angle between the local orientation (phi and theta arguments) and the main orientation (phim and thetam arguments). Saturation is always 1 and value is the pixel value in the geometry image normalized such that maximum value of the geometry image is mapped to 1, and zero to 0.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

geometry [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

An image that contains the geometry for which local orientation has been determined.

phi [input]

Data type: float32 image

The azimuthal angle of the local orientation direction. The angle is given in radians and measured from positive x-axis towards positive y-axis and is given in range $[-\pi, \pi]$.

theta [input]

Data type: float32 image

The polar angle of the local orientation direction. The angle is given in radians and measured from positive z-axis towards xy-plane. The values are in range $[0, \pi]$.

phim [input]

Data type: real

The azimuthal angle of the main orientation direction in radians.

thetam [input]

Data type: real

The polar angle of the main orientation direction in radians.

r [output]

Data type: uint8 image

Red channel of the result will be stored into this image.

g [output]

Data type: uint8 image

Green channel of the result will be stored into this image.

b [output]

Data type: uint8 image

Blue channel of the result will be stored into this image.

See also

cylindricality, *cylinderorientation*, *plateorientation*, *mainorientationcolor*, *axelssoncolor*, *orientationdifference*

3.8.110 mapraw

There are 2 forms of this command.

`mapraw(image name, filename, data type, width, height, depth, read only)`

Maps a .raw image file to an image. If the image file does not exist, it is created. Changes made to the file are IMMEDIATELY reflected to disk, so there is no need to save or load the image. The image must be in the native byte order of the host computer. Dimensions of the .raw file do not need to be specified if the file name is in format name_WxHxD.raw, where [W, H, D] are the dimensions of the image. The system tries to guess the pixel data type, too, based on the file size and dimensions of the image as follows. If pixel size in bytes is 1, the system sets the pixel type to uint8. If pixel size in bytes is 2, the system sets the pixel type to uint16. If pixel size in bytes is 4, float32 pixel data is assumed (instead of e.g. int32 or uint32). If pixel size in bytes is 8, pixels are assumed to be of type uint64 (instead of e.g. int64 or complex32). If the guess is wrong, the pixel data type must be explicitly specified using the corresponding argument. Even in this case the dimensions can be read from the name of the file if the file name contains the dimensions.

Note: In Python/pi2py2, the image name parameter is not specified, and the function returns a newly created image mapped to the .raw image file.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image name [input]

Data type: string

Name of image in the system.

filename [input]

Data type: string

Name (and path) of file to map.

data type [input]

Data type: string

Data type of the image. Can be uint8, uint16, uint32, uint64, int8, int16, int32, int64, float32, or complex32. Specify empty value to infer data type from image dimensions.

width [input]

Data type: integer

Width of the image. Omit width, height and depth to infer dimensions from file name.

height [input]

Data type: integer

Height of the image. Omit width, height and depth to infer dimensions from file name.

depth [input]

Data type: integer

Depth of the image. Omit width, height and depth to infer dimensions from file name.

read only [input]

Data type: boolean

Default value: False

Set to true to do read-only mapping. This might be beneficial if the image file is accessed through a network share. WARNING: If set to true, writes to the image result in undefined behaviour, probably program crash to access violation or equivalent error.

See also

readrawblock, *readraw*, *getmapfile*

`mapraw(image name, filename, data type, dimensions, read only)`

Maps a .raw image file to an image. If the image file does not exist, it is created. Changes made to the file are IMMEDIATELY reflected to disk, so there is no need to save or load the image. The image must be in the native byte order of the host computer. Dimensions of the .raw file do not need to be specified if the file name is in format name_WxHxD.raw, where [W, H, D] are the dimensions of the image. The system tries to guess the pixel data type, too, based on the file size and dimensions of the image as follows. If pixel size in bytes is 1, the system sets the pixel type to uint8. If pixel size in bytes is 2, the system sets the pixel type to uint16. If pixel size in bytes is 4, float32 pixel data is assumed (instead of e.g. int32 or uint32). If pixel size in bytes is 8, pixels are assumed to be of type uint64 (instead of e.g. int64 or complex32). If the guess is wrong, the pixel data type must be explicitly specified using the corresponding argument. Even in this case the dimensions can be read from the name of the file if the file name contains the dimensions.

Note: In Python/pi2py2, the image name parameter is not specified, and the function returns a newly created image mapped to the .raw image file.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image name [input]

Data type: string

Name of image in the system.

filename [input]

Data type: string

Name (and path) of file to map.

data type [input]

Data type: string

Default value: “”

Data type of the image. Can be uint8, uint16, uint32, uint64, int8, int16, int32, int64, float32, or complex32. Specify empty value to infer data type from image dimensions.

dimensions [input]

Data type: 3-component integer vector

Default value: “[0, 0, 0]”

Dimensions of the image. Set to zero to infer dimensions from file name.

read only [input]

Data type: boolean

Default value: False

Set to true to do read-only mapping. This might be beneficial if the image file is accessed through a network share. WARNING: If set to true, writes to the image result in undefined behaviour, probably program crash to access violation or equivalent error.

See also

readrawblock, *readraw*, *getmapfile*

3.8.111 maskedbin

Syntax: `maskedbin(input image, output image, factor, bad value, undefined value)`

Reduces size of input image by given integer factor. Each output pixel corresponds to factor^dimensionality block of pixels in the input image. Supports treating one value in the input image as ‘bad’ such that pixels having that value do not contribute to the output at all.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Output image.

factor [input]

Data type: positive integer

Binning factor in each coordinate direction. Value 2 makes the output image dimension half of the input image dimension, 3 makes them one third etc.

bad value [input]

Data type: real

Default value: 0

Value that should not be considered in the averaging calculations.

undefined value [input]

Data type: real

Default value: 0

Value that is placed to those pixels of the output image that do not correspond to any valid pixels in the input image.

See also

scale, *bin*, *maskedbin*, *scalelabels*

3.8.112 maskedmean

Syntax: `maskedmean(input image, output image, ignored value, print to log, square root)`

Calculates mean of all pixels in the input image, but skips specific value in the calculation. The output is a 1x1x1 image.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: float32 image

Output image.

ignored value [input]

Data type: real

This value is ignored in the calculation.

print to log [input]

Data type: boolean

Default value: False

Set to true to print the results to the log. This is a hack required because there is currently no good way to return single/simple values in distributed processing.

square root [input]

Data type: boolean

Default value: False

Set to true to take square root of the pixel values before calculating mean. This is a hack currently needed in distributed thickness map calculation. If set to true, the input image will contain its square root at output, except in distributed processing where it will retain its old value.

3.8.113 max

There are 2 forms of this command.

```
max(image, parameter image, allow broadcast)
```

Calculates pixelwise maximum of two images. Output is placed to the first image.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

parameter image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image

Parameter image.

allow broadcast [input]

Data type: boolean

Default value: False

Set to true to allow size of parameter image differ from size of input image. If there is a need to access pixel outside of parameter image, the nearest value inside the image is taken instead. If set to false, dimensions of input and parameter images must be equal. If set to true, the parameter image is always loaded in its entirety in distributed processing mode.

`max(image, x)`

Calculates maximum of an image and a constant.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

x [input]

Data type: real

Constant value.

3.8.114 maxfilter

Syntax: `maxfilter(input image, output image, radius, allow optimization, neighbourhood type, boundary condition)`

Maximum filter. Replaces pixel by maximum of pixels in its neighbourhood. For binary images, this equals dilation of foreground (non-zero) pixels.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Output image.

radius [input]

Data type: 3-component integer vector

Default value: "[1, 1, 1]"

Radius of neighbourhood. Diameter will be $2r + 1$.

allow optimization [input]

Data type: boolean

Default value: True

Set to true to allow use of approximate decompositions of spherical structuring elements using periodic lines. As a result of the approximation processing is much faster but the true shape of the structuring element is not sphere but a regular polyhedron. See van Herk - A fast algorithm for local minimum and maximum filters on rectangular and octagonal kernels and Jones - Periodic lines Definition, cascades, and application to granulometries. The approximate filtering will give wrong results where distance from image edge is less than r. Consider enlarging the image by r to all directions before processing. Enlarging in the z-direction is especially important for 2D images, and therefore approximate processing is not allowed if the image is 2-dimensional.

neighbourhood type [input]

Data type: neighbourhood type

Default value: Ellipsoidal

Type of neighbourhood. Can be Ellipsoidal for ellipsoidal or spherical neighbourhood; or Rectangular for rectangular neighbourhood.

boundary condition [input]

Data type: boundary condition

Default value: Nearest

Type of boundary condition. Zero indicates that values outside of image bounds are taken to be zero. Nearest indicates that the nearest value inside the image is to be used in place of values outside of image bounds.

See also

gaussfilter, *bilateralfilter*, *bilateralfilterapprox*, *vawefilter*, *openingfilter*, *closingfilter*, *minfilter*, *maxfilter*, *medianfilter*, *variancefilter*, *stddevfilter*, *bandpassfilter*, *highpassfilter*

3.8.115 maxjobs

Syntax: `maxjobs(max number of jobs)`

Sets the maximum number of jobs that will be submitted in parallel in distributed mode. If there are more jobs to be submitted at once, the jobs are combined into larger jobs until the total number of jobs is below or equal to the speficied maximum. This command overrides max_parallel_submit_count value read from the distributor configuration file.

This command can be used in the distributed processing mode, but it does not participate in distributed processing.

Arguments

max number of jobs [input]

Data type: positive integer

Default value: 0

Maximum number of jobs to submit in parallel. If the analysis task requires more jobs than specified, the jobs are combined until only maximum number of jobs are left. Specify zero for unlimited number of jobs.

See also

distribute, *delaying*, *maxmemory*, *maxjobs*, *chunksize*, *printscripts*

3.8.116 maxmemory

Syntax: `maxmemory(maximum memory)`

Sets the maximum memory setting used in distributed processing. This command overrides the value read from the configuration file. The maximum memory is the amount of memory that can be used either on the local computer (Local distribution mode) or in a compute node (Slurm distribution mode).

This command can be used in the distributed processing mode, but it does not participate in distributed processing.

Arguments

maximum memory [input]

Data type: real

Default value: 0

Maximum amount of memory to use, in megabytes. Specify zero to calculate the value automatically.

See also

distribute, *delaying*, *maxmemory*, *maxjobs*, *chunksize*, *printscripts*

3.8.117 maxproject

There are 2 forms of this command.

`maxproject(input image, output image, dimension)`

Calculates projection of the input image. The dimensionality of the output image is the dimensionality of the input image subtracted by one.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Output image.

dimension [input]

Data type: positive integer

Default value: 2

Dimension to project over, zero corresponding to x, one corresponding to y, and 2 corresponding to z.

`maxproject(input image, output image, dimension, input value image, output value image)`

Calculates projection of the input image. The dimensionality of the output image is the dimensionality of the input image subtracted by one. Constructs another image from values of second input image taken at location of the extrema.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Output image.

dimension [input]

Data type: positive integer

Default value: 2

Dimension to project over, zero corresponding to x, one corresponding to y, and 2 corresponding to z.

input value image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image

Image where values of the extra output are taken.

output value image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image

Value of this image will equal value of 'input value image' at location of the extrema (whose value is stored in the corresponding output image).

3.8.118 maxval

Syntax: `maxval(input image, output image, print to log)`

Calculates max of all pixels in the input image. The output is a 1x1x1 image.

Note: In Python/pi2py2, the output image is not specified, and the result value is returned by the function.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image

Output image.

print to log [input]

Data type: boolean

Default value: False

Set to true to print the results to the log.

3.8.119 mean

Syntax: `mean(input image, output image, print to log)`

Calculates mean of all pixels in the input image. The output is a 1x1x1 image.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: float32 image

Output image.

print to log [input]

Data type: boolean

Default value: False

Set to true to print the results to the log.

3.8.120 meancurvature

Syntax: `meancurvature(geometry, mean curvature, sigma)`

Calculates mean curvature of surfaces defined by $f(x, y, z) = 0$, multiplied by the norm of the gradient of the image f. Without multiplication by $||\nabla f||$, the curvature of f is very hard to interpret as the values are only valid near the zero contour of f (at least from the image processing point-of-view). This command calculates curvature as divergence of unit normal: $-0.5\nabla \cdot (\nabla f/||\nabla f||)||\nabla f||$. See also https://en.wikipedia.org/wiki/Mean_curvature#Implicit_form_of_mean_curvature for formulas that can be used to determine the mean curvature. (The corresponding 2D formula is found at https://en.wikipedia.org/wiki/Implicit_curve#Curvature.) Note that this function might produce invalid results for surfaces of structures whose thickness is similar to the value of the sigma parameter. See *curvature* command for a version that does not suffer from this deficiency and that can be used to calculate the principal curvatures of surfaces in binary images.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

geometry [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

The image containing the geometry. Non-zero pixels are assumed to be foreground. The image does not need to be a binary image.

mean curvature [output]

Data type: float32 image

The mean curvature will be placed into this image.

sigma [input]

Data type: real

Default value: 1

Scale parameter for derivative calculation. Set to the preferred scale of edges that should be detected. Derivatives are calculated using convolutions with derivative of Gaussian function, and this parameter defines the standard deviation of the Gaussian.

See also

curvature, *derivative*

3.8.121 meanproject

Syntax: `meanproject(input image, output image, dimension)`

Calculates projection of the input image. The dimensionality of the output image is the dimensionality of the input image subtracted by one.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: float32 image

Output image.

dimension [input]

Data type: positive integer

Default value: 2

Dimension to project over, zero corresponding to x, one corresponding to y, and 2 corresponding to z.

3.8.122 medianfilter

Syntax: `medianfilter(input image, output image, radius, neighbourhood type, boundary condition)`

Median filtering. Replaces pixel by median of pixels in its neighbourhood. Removes noise from the image while preserving sharp edges.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Output image.

radius [input]

Data type: 3-component integer vector

Default value: "[1, 1, 1]"

Radius of neighbourhood. Diameter will be $2r + 1$.

neighbourhood type [input]

Data type: neighbourhood type

Default value: Ellipsoidal

Type of neighbourhood. Can be Ellipsoidal for ellipsoidal or spherical neighbourhood; or Rectangular for rectangular neighbourhood.

boundary condition [input]

Data type: boundary condition

Default value: Nearest

Type of boundary condition. Zero indicates that values outside of image bounds are taken to be zero. Nearest indicates that the nearest value inside the image is to be used in place of values outside of image bounds.

See also

gaussfilter, *bilateralfilter*, *bilateralfilterapprox*, *vawefilter*, *openingfilter*, *closingfilter*, *minfilter*, *maxfilter*, *medianfilter*, *variancefilter*, *stddevfilter*, *bandpassfilter*, *highpassfilter*

3.8.123 metacolumncount

Syntax: `metacolumncount(image, key, row index, count)`

Retrieves count of columns in a specific row of a metadata item.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to process.

key [input]

Data type: string

Name of the metadata item.

row index [input]

Data type: integer

Index of the row whose column count is to be returned.

count [output]

Data type: integer

Item count.

See also

setmeta, *getmeta*, *writemeta*, *readmeta*, *clearmeta*, *listmeta*, *copymeta*, *metarowcount*, *metacolumncount*

3.8.124 metarowcount

Syntax: `metarowcount(image, key, count)`

Retrieves count of rows in a metadata item.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to process.

key [input]

Data type: string

Name of the metadata item.

count [output]

Data type: integer

Item count.

See also

setmeta, *getmeta*, *writemeta*, *readmeta*, *clearmeta*, *listmeta*, *copymeta*, *metarowcount*, *metacolumncount*

3.8.125 min

There are 2 forms of this command.

`min(image, parameter image, allow broadcast)`

Calculates pixelwise minimum of two images. Output is placed to the first image.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

parameter image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image

Parameter image.

allow broadcast [input]

Data type: boolean

Default value: False

Set to true to allow size of parameter image differ from size of input image. If there is a need to access pixel outside of parameter image, the nearest value inside the image is taken instead. If set to false, dimensions of input and parameter images must be equal. If set to true, the parameter image is always loaded in its entirety in distributed processing mode.

`min(image, x)`

Calculates minimum of an image and a constant.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

x [input]

Data type: real

Constant value.

3.8.126 minfilter

Syntax: `minfilter(input image, output image, radius, allow optimization, neighbourhood type, boundary condition)`

Minimum filter. Replaces pixel by minimum of pixels in its neighbourhood. For binary images this equals erosion of foreground (non-zero) pixels.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Output image.

radius [input]

Data type: 3-component integer vector

Default value: "[1, 1, 1]"

Radius of neighbourhood. Diameter will be $2r + 1$.

allow optimization [input]

Data type: boolean

Default value: True

Set to true to allow use of approximate decompositions of spherical structuring elements using periodic lines. As a result of the approximation processing is much faster but the true shape of the structuring element is not sphere but a regular polyhedron. See van Herk - A fast algorithm for local minimum and maximum filters on rectangular and octagonal kernels and Jones - Periodic lines Definition, cascades, and application to granulometries. The approximate filtering will give wrong results where distance from image edge is less than r. Consider enlarging the image by r to all directions before processing. Enlarging in the z-direction is especially important for 2D images, and therefore approximate processing is not allowed if the image is 2-dimensional.

neighbourhood type [input]

Data type: neighbourhood type

Default value: Ellipsoidal

Type of neighbourhood. Can be Ellipsoidal for ellipsoidal or spherical neighbourhood; or Rectangular for rectangular neighbourhood.

boundary condition [input]

Data type: boundary condition

Default value: Nearest

Type of boundary condition. Zero indicates that values outside of image bounds are taken to be zero. Nearest indicates that the nearest value inside the image is to be used in place of values outside of image bounds.

See also

gaussfilter, *bilateralfilter*, *bilateralfilterapprox*, *vawefilter*, *openingfilter*, *closingfilter*, *minfilter*, *maxfilter*, *medianfilter*, *variancefilter*, *stddevfilter*, *bandpassfilter*, *highpassfilter*

3.8.127 minproject

There are 2 forms of this command.

minproject(input image, output image, dimension)

Calculates projection of the input image. The dimensionality of the output image is the dimensionality of the input image subtracted by one.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Output image.

dimension [input]

Data type: positive integer

Default value: 2

Dimension to project over, zero corresponding to x, one corresponding to y, and 2 corresponding to z.

minproject(input image, output image, dimension, input value image, output value image)

Calculates projection of the input image. The dimensionality of the output image is the dimensionality of the input image subtracted by one. Constructs another image from values of second input image taken at location of the extrema.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Output image.

dimension [input]

Data type: positive integer

Default value: 2

Dimension to project over, zero corresponding to x, one corresponding to y, and 2 corresponding to z.

input value image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image

Image where values of the extra output are taken.

output value image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image

Value of this image will equal value of 'input value image' at location of the extrema (whose value is stored in the corresponding output image).

3.8.128 minval

Syntax: `minval(input image, output image, print to log)`

Calculates min of all pixels in the input image. The output is a 1x1x1 image.

Note: In Python/pi2py2, the output image is not specified, and the result value is returned by the function.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image

Output image.

print to log [input]

Data type: boolean

Default value: False

Set to true to print the results to the log.

3.8.129 montage

Syntax: `montage(input image, output image, columns, rows, scale, first slice, last slice, step, border width, border color)`

Makes a 2D montage of a 3D image.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Output image.

columns [input]

Data type: positive integer

Number of 2D slices in the montage in the horizontal direction.

rows [input]

Data type: positive integer

Number of 2D slices in the montage in the vertical direction.

scale [input]

Data type: real

Default value: 1

Scaling factor between slices in the original image and the slices in the montage.

first slice [input]

Data type: positive integer

Default value: 0

First slice to include in the montage.

last slice [input]

Data type: positive integer

Default value: 18446744073709551615

Last slice to include in the montage. Note that the columns and rows parameters define the maximum number of slices that will fit into the montage.

step [input]

Data type: positive integer

Default value: 0

Step between slices to include in the montage. Specify zero to set the step to a value that accommodates approximately all the stack slices in the montage.

border width [input]

Data type: positive integer

Default value: 0

Width of borders between the slices in the montage.

border color [input]

Data type: real

Default value: 0

Color of borders between slices in the montage.

3.8.130 morphorec

Syntax: `morphorec(image, parameter image)`

Morphological reconstruction. Dilates input image (seed image) and after each dilation constraints changes to nonzero pixels of parameter image (mask image). Continues until the image does not change anymore.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

parameter image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image

Parameter image.

3.8.131 multiply

There are 2 forms of this command.

`multiply(image, parameter image, allow broadcast)`

Multiplies two images. Output is placed to the first image. The operation is performed using saturation arithmetic.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to process.

parameter image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image, complex32 image

Parameter image.

allow broadcast [input]

Data type: boolean

Default value: False

Set to true to allow size of parameter image differ from size of input image. If there is a need to access pixel outside of parameter image, the nearest value inside the image is taken instead. If set to false, dimensions of input and parameter images must be equal. If set to true, the parameter image is always loaded in its entirety in distributed processing mode.

`multiply(image, x)`

Multiplies an image by a constant. The operation is performed using saturation arithmetic.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

x [input]

Data type: real

Multiplier constant.

3.8.132 negate

Syntax: `negate(image)`

Negates pixel values.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to process.

3.8.133 newimage

There are 2 forms of this command.

`newimage(image name, data type, width, height, depth)`

Creates a new, empty image.

Note: In Python/pi2py2, the image name parameter is not specified, and the return value is a Pi2Image object that can be passed to any command expecting an image name as an argument.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image name [input]

Data type: string

Name of the image in the system.

data type [input]

Data type: string

Default value: uint8

Data type of the image. Can be uint8, uint16, uint32, uint64, int8, int16, int32, int64, float32, or complex32.

width [input]

Data type: integer

Default value: 1

Width of the image.

height [input]

Data type: integer

Default value: 1

Height of the image.

depth [input]

Data type: integer

Default value: 1

Depth of the image.

See also

ensuresize, *newlike*

`newimage(image name, data type, dimensions)`

Creates a new, empty image.

Note: In Python/pi2py2, the image name parameter is not specified, and the return value is a Pi2Image object that can be passed to any command expecting an image name as an argument.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image name [input]

Data type: string

Name of the image in the system.

data type [input]

Data type: string

Data type of the image. Can be uint8, uint16, uint32, uint64, int8, int16, int32, int64, float32, or complex32.

dimensions [input]

Data type: 3-component integer vector

Dimensions of the image.

See also

ensuresize, *newlike*

3.8.134 newlike

There are 2 forms of this command.

`newlike(image name, template image, data type, width, height, depth)`

Creates a new, empty image that has properties (dimensions, data type) similar to another image.

Note: In Python/pi2py2, the image name parameter is not specified, and the return value is the newly created image object that can be passed to other functions instead of an image name.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image name [input]

Data type: string

Name of the new image in the system.

template image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Name of existing image where dimensions and data type will be copied from.

data type [input]

Data type: string

Default value: “”

Data type of the image. Can be uint8, uint16, uint32, uint64, int8, int16, int32, int64, float32, or complex32. Leave empty or set to Unknown to copy the value from the template image.

width [input]

Data type: integer

Default value: 0

Width of the image. Set to zero to copy the value from the template image.

height [input]

Data type: integer

Default value: 0

Height of the image. Set to zero to copy the value from the template image.

depth [input]

Data type: integer

Default value: 0

Depth of the image. Set to zero to copy the value from the template image.

See also

newlikefile, *newimage*, *ensuresize*

`newlike(image name, template image, data type, dimensions)`

Creates a new, empty image that has properties (dimensions, data type) similar to another image.

Note: In Python/pi2py2, the image name parameter is not specified, and the return value is the newly created image object that can be passed to other functions instead of an image name.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image name [input]

Data type: string

Name of the new image in the system.

template image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Name of existing image where dimensions and data type will be copied from.

data type [input]

Data type: string

Default value: “”

Data type of the image. Can be uint8, uint16, uint32, uint64, int8, int16, int32, int64, float32, or complex32. Leave empty or set to Unknown to copy the value from the template image.

dimensions [input]

Data type: 3-component integer vector

Dimensions of the image. Set any component to zero to copy the value from the template image.

See also

newlikefile, *newimage*, *ensuresize*

3.8.135 newlikefile

There are 2 forms of this command.

```
newlikefile(image name, template filename, data type, width, height, depth)
```

Creates a new, empty image that has properties (dimensions, data type) similar to another image that has been saved to disk.

Note: In Python/pi2py2, the image name parameter is not specified, and the return value is the created image that can be passed to other functions instead of an image name.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image name [input]

Data type: string

Name of the new image in the system.

template filename [input]

Data type: string

Name of existing image file where dimensions and data type will be copied from.

data type [input]

Data type: string

Default value: “”

Data type of the image. Can be uint8, uint16, uint32, uint64, int8, int16, int32, int64, float32, or complex32. Leave empty or set to Unknown to copy the value from the template image.

width [input]

Data type: integer

Default value: 0

Width of the image. Set to zero to copy the value from the template image.

height [input]

Data type: integer

Default value: 0

Height of the image. Set to zero to copy the value from the template image.

depth [input]

Data type: integer

Default value: 0

Depth of the image. Set to zero to copy the value from the template image.

See also

newlikefile, *newimage*, *ensuresize*

`newlikefile(image name, template filename, data type, dimensions)`

Creates a new, empty image that has properties (dimensions, data type) similar to another image that has been saved to disk.

Note: In Python/pi2py2, the image name parameter is not specified, and the return value is the created image that can be passed to other functions instead of an image name.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image name [input]

Data type: string

Name of the new image in the system.

template filename [input]

Data type: string

Name of existing image file where dimensions and data type will be copied from.

data type [input]

Data type: string

Default value: “”

Data type of the image. Can be uint8, uint16, uint32, uint64, int8, int16, int32, int64, float32, or complex32. Leave empty or set to Unknown to copy the value from the template image.

dimensions [input]

Data type: 3-component integer vector

Default value: “[0, 0, 0]”

Dimensions of the image. Set any component to zero to copy that from the template image.

See also

newlikefile, *newimage*, *ensuresize*

3.8.136 newvalue

Syntax: `newvalue(name, type, value)`

Creates a new variable.

Note: In Python/pi2py2, one should use the newstring command.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

name [input]

Data type: string

Name of the variable in the system.

type [input]

Data type: string

Default value: string

Data type of the variable. Can be 'string', 'int', 'real', or 'bool'.

value [input]

Data type: string

Default value: ""

Initial value of the variable

See also

set, *clear*

3.8.137 noise

Syntax: `noise(image, mean, standard deviation, seed)`

Adds additive Gaussian noise to the image.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

mean [input]

Data type: real

Default value: 0

Mean value of the noise to add.

standard deviation [input]

Data type: real

Default value: 0

Standard deviation of the noise to add. Specify zero to select standard deviation based on typical maximum value range of the pixel data type.

seed [input]

Data type: integer

Default value: 0

Seed value. Set to zero to use time-based seed.

3.8.138 normalize

Syntax: `normalize(image)`

Makes norm of each pixel one.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: complex32 image

Image to process.

3.8.139 normalizez

Syntax: `normalizez(image, target mean)`

Makes sure that all z-slices of the image have the same mean value.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

target mean [input]

Data type: real

Default value: nan

Global mean that the image should have after normalization. Specify nothing or nan to retain global mean of the image.

3.8.140 normsquared

Syntax: `normsquared(image)`

Calculates squared norm of pixel values.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: complex32 image

Image to process.

3.8.141 openingfilter

Syntax: `openingfilter(image, radius, allow optimization, neighbourhood type, boundary condition)`

Opening filter. Widens gaps in bright objects and removes objects smaller than neighbourhood size. Has optimized implementation for rectangular neighbourhoods. Creates one temporary image of same size than input.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

radius [input]

Data type: 3-component integer vector

Default value: "[1, 1, 1]"

Radius of neighbourhood. Diameter will be $2r + 1$.

allow optimization [input]

Data type: boolean

Default value: True

Set to true to allow use of approximate decompositions of spherical structuring elements using periodic lines. As a result of the approximation processing is much faster but the true shape of the structuring element is not sphere but a regular polyhedron. See van Herk - A fast algorithm for local minimum and maximum filters on rectangular and octagonal kernels and Jones - Periodic lines Definition, cascades, and application to granulometries. The approximate filtering will give wrong results where distance from image edge is less than r. Consider enlarging the image by r to all directions before processing. Enlarging in the z-direction is especially important for 2D images, and therefore approximate processing is not allowed if the image is 2-dimensional.

neighbourhood type [input]

Data type: neighbourhood type

Default value: Ellipsoidal

Type of neighbourhood. Can be Ellipsoidal for ellipsoidal or spherical neighbourhood; or Rectangular for rectangular neighbourhood.

boundary condition [input]

Data type: boundary condition

Default value: Nearest

Type of boundary condition. Zero indicates that values outside of image bounds are taken to be zero. Nearest indicates that the nearest value inside the image is to be used in place of values outside of image bounds.

See also

gaussfilter, *bilateralfilter*, *bilateralfilterapprox*, *vawefilter*, *openingfilter*, *closingfilter*, *minfilter*, *maxfilter*, *medianfilter*, *variancefilter*, *stddevfilter*, *bandpassfilter*, *highpassfilter*

3.8.142 orientationdifference

Syntax: `orientationdifference(phi, theta, alpha, phim, thetam)`

For each pixel x, calculates angle between $(\phi(x), \theta(x))$ and (ϕ_m, θ_m) and assigns that to the output image.This command does the same calculation than what *mainorientationcolor* command does, but outputs the angular difference values instead of color map, and does not weight the output values by the original geometry in any way.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

phi [input]

Data type: float32 image

The azimuthal angle of the local orientation direction. The angle is given in radians and measured from positive x-axis towards positive y-axis and is given in range $[-\pi, \pi]$.

theta [input]

Data type: float32 image

The polar angle of the local orientation direction. The angle is given in radians and measured from positive z-axis towards xy-plane. The values are in range $[0, \pi]$.

alpha [output]

Data type: float32 image

The output image. The values are in range $[0, \pi/2]$.

phim [input]

Data type: real

The azimuthal angle of the main orientation direction in radians.

thetam [input]

Data type: real

The polar angle of the main orientation direction in radians.

See also

cylindricality, *cylinderorientation*, *plateorientation*, *mainorientationcolor*, *axelssoncolor*, *orientationdifference*

3.8.143 pathlength

Syntax: `pathlength(input image, output image)`

Replaces value of each pixel by the length of the longest constrained path that goes through that pixel. Works only with binary input images. NOTE: This command creates temporary files to the current directory. NOTE: This command is highly work-in-progress.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: float32 image

Output image.

3.8.144 plateorientation

Syntax: `plateorientation(geometry, phi, theta, derivative sigma, smoothing sigma)`

Estimates local orientation of planar structures (normal of the plane) using the structure tensor method. See also B. Jähne, Practical handbook on image processing for scientific and technical applications. CRC Press, 2004.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

geometry [input & output]

Data type: float32 image

At input, an image that contains the geometry for which local orientation should be determined. At output, the orientation 'energy' will be stored in this image. It equals the sum of the eigenvalues of the structure tensor, and can be used to distinguish regions without any interfaces (i.e. no orientation, low energy value) from regions with interfaces (i.e. orientation available, high energy value).

phi [output]

Data type: float32 image

The azimuthal angle of orientation direction will be stored in this image. The angle is given in radians and measured from positive x-axis towards positive y-axis and is given in range $[-\pi, \pi]$.

theta [output]

Data type: float32 image

The polar angle of orientation direction will be stored in this image. The angle is given in radians and measured from positive z-axis towards xy-plane. The values are in range $[0, \pi]$.

derivative sigma [input]

Data type: real

Default value: 1

Scale parameter. Set to the preferred scale of edges that define the cylinders. Derivatives required in the stucture tensor are calculated using convolutions with derivative of a Gaussian function, and this parameter defines the standard deviation of the Gaussian.

smoothing sigma [input]

Data type: real

Default value: 1

The structure tensor is smoothed by convolution with a Gaussian. This parameter defines the standard deviation of the smoothing Gaussian.

See also

cylindricality, *cylinderorientation*, *plateorientation*, *mainorientationcolor*, *axelssoncolor*, *orientationdifference*

3.8.145 pointstodeformed

Syntax: `pointstodeformed(points, file name prefix)`

Projects points from reference configuration to deformed configuration, using a transformation determined with the *blockmatch* command.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

points [input]

Data type: float32 image

Image that contains the points that will be transformed. The size of the image must be 3xN where N is the number of points to transform.

file name prefix [input]

Data type: string

File name prefix (and path) passed to blockmatch command.

See also

blockmatch, *blockmatchmemsave*, *pullback*, *pointstodeformed*

3.8.146 polysdmap

Syntax: `polysdmap(seeds, geometry, output, max segment length)`

Calculates seeded distance map of a binary image. For each pixel in the image, finds the shortest segmented polygonal path to nearest seed region such that the path does not pass through regions that are marked as obstacles, and ensuring that each segment that makes up the path is shorter than specified in 'max segment length' argument. This command is highly experimental.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

seeds [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Seed image that contains the set of pixels where the distance is zero. The set is marked with nonzero values, i.e. the distance map will propagate to pixels that have zero value in this image. This image is not modified.

geometry [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image containing the geometry in which to calculate the distances. Regions marked with zero pixel value are 'unpassable obstacles'. The distance transform will only proceed to pixels whose color in this image matches the color of the seed point (in this image). This image is not modified.

output [output]

Data type: float32 image

Output image that will contain the distance to the nearest seed region, not passing through zero regions in the geometry image.

max segment length [input]

Data type: real

Maximum length of segments that make up the path to the nearest seed point.

See also

dmap, *dmap2*, *tmap*, *sdmap*

3.8.147 print

Syntax: `print(string)`

Prints a message to the output. This is mainly used internally in distributed processing.

This command can be used in the distributed processing mode, but it does not participate in distributed processing.

Arguments

string [input]

Data type: string

String to print

See also

help, *info*, *license*, *echo*, *print*, *waitreturn*, *hello*, *timing*, *savetiming*, *resettiming*

3.8.148 printscripts

Syntax: `printscripts(enable)`

Enables or disables printing of pi2 script run for each task in compute node/process. Has effect only if distributed processing is enabled.

This command can be used in the distributed processing mode, but it does not participate in distributed processing.

Arguments

enable [input]

Data type: boolean

Default value: True

Set to true to enable printing.

See also

distribute, *delaying*, *maxmemory*, *maxjobs*, *chunksize*, *printscripts*

3.8.149 pruneskeleton

Syntax: `pruneskeleton(vertices, edges, edge measurements, edge points, maximum length, disconnect straight-through nodes, remove isolated nodes)`

Prunes a traced skeleton. Removes all edges that end in a node with no other branches connected to it (degree = 1) and whose length is less than specified value.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

vertices [input & output]

Data type: float32 image

Image where vertex coordinates are stored. See *tracelineskeleton* command.

edges [input & output]

Data type: uint64 image

Image where vertex indices corresponding to each edge are stored. See *tracelineskeleton* command.

edge measurements [input & output]

Data type: float32 image

Image where properties of each edge are stored. See *tracelineskeleton* command.

edge points [input & output]

Data type: int32 image

Image that stores some points on each edge. See *tracelineskeleton* command.

maximum length [input]

Data type: real

Edges shorter than this are pruned if they are not connected to other edges in both ends.

disconnect straight-through nodes [input]

Data type: boolean

If set to true, all straight-through nodes (nodes with degree = 2) are removed after pruning. This operation might change the network even if no edges are pruned.

remove isolated nodes [input]

Data type: boolean

If set to true, all isolated nodes (nodes with degree = 0) are removed after pruning. This operation might change the network even if no edges are pruned.

See also

cleanskeleton, *pruneskeleton*, *removeedges*, *fillskeleton*, *getpointsandlines*

3.8.150 pullback

There are 2 forms of this command.

`pullback(image, pullback image, file name prefix, interpolation mode)`

Applies reverse of a deformation (calculated using blockmatch command) to image. In other words, performs pull-back operation. Makes output image the same size than the input image.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image that will be pulled back.

pullback image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Will store the result of the pullback operation.

file name prefix [input]

Data type: string

File name prefix (and path) passed to blockmatch command.

interpolation mode [input]

Data type: interpolation mode

Default value: Cubic

Interpolation mode. Can be Nearest for nearest neighbour interpolation, Linear for linear interpolation, or Cubic for cubic interpolation.

See also

blockmatch, *blockmatchmemsave*, *pullback*, *pointstodeformed*

`pullback(image, pullback image, grid start, grid step, grid max, x, y, z, interpolation mode, pullback position, pullback size)`

Applies reverse of a deformation (calculated, e.g., using the blockmatch command) to image. In other words, performs pull-back operation.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image that will be pulled back, i.e. the deformed image.

pullback image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Will store the result of the pullback operation, i.e. the deformed image transformed to coordinates of the reference image.

grid start [input]

Data type: 3-component integer vector

Start of reference point grid in the coordinates of the reference image.

grid step [input]

Data type: 3-component integer vector

Grid step in each coordinate direction.

grid max [input]

Data type: 3-component integer vector

End of reference point grid in the coordinates of the reference image. The grid will contain floor((max - start) / step) + 1 points in each coordinate direction. Difference between maximum and minimum does not need to be divisible by step.

x [input]

Data type: float32 image

X-coordinate of each reference grid point in the coordinates of the deformed image. Dimensions of this image must equal point counts in the reference grid.

y [input]

Data type: float32 image

Y-coordinate of each reference grid point in the coordinates of the deformed image. Dimensions of this image must equal point counts in the reference grid.

z [input]

Data type: float32 image

Z-coordinate of each reference grid point in the coordinates of the deformed image. Dimensions of this image must equal point counts in the reference grid.

interpolation mode [input]

Data type: interpolation mode

Default value: Cubic

Interpolation mode. Can be Nearest for nearest neighbour interpolation, Linear for linear interpolation, or Cubic for cubic interpolation.

pullback position [input]

Data type: 3-component real vector

Default value: "[0, 0, 0]"

Position of region to be pulled back in reference image coordinates.

pullback size [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

Size of the region to be pulled back. Specify zeroes to default to the size of the deformed image.

See also

blockmatch, *blockmatchmemsave*, *pullback*, *pointstodeformed*

3.8.151 ramp

Syntax: `ramp(image, dimension, block origin)`

Fills image with ramp in given dimension, i.e. performs img[r] = r[dimension].

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

dimension [input]

Data type: integer

Default value: 0

Dimension of the ramp.

block origin [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

Origin of current calculation block in coordinates of the full image. This argument is used internally in distributed processing. Set to zero in normal usage.

See also

line, *capsule*, *sphere*, *ellipsoid*, *set*, *get*, *ramp*, *ramp3*

3.8.152 ramp3

Syntax: `ramp3(image, block origin)`

Fills image with ramp in all three dimensions, i.e. performs img[r] = ||r||_1.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

block origin [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

Origin of current calculation block in coordinates of the full image. This argument is used internally in distributed processing. Set to zero in normal usage.

See also

line, *capsule*, *sphere*, *ellipsoid*, *set*, *get*, *ramp*, *ramp3*

3.8.153 rawinfo

Syntax: `rawinfo(filename)`

Displays metadata of .raw file.

This command can be used in the distributed processing mode, but it does not participate in distributed processing.

Arguments

filename [input]

Data type: string

Name of .raw file.

3.8.154 read

Syntax: `read(image name, filename, data type)`

Reads an image or image sequence from disk. Determines type of file automatically.

Note: In Python/pi2py2, the image name parameter is not specified, and the function returns the newly created image read from the disk.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image name [input]

Data type: string

Name of image in the system.

filename [input]

Data type: string

Name (and path) of file to read or a sequence definition. If a directory name is given, all files in the directory will be read. If the a file name is given, it can contain wildcards $*$, ? and @. Wildcard $*$ corresponds to any sequence of characters, wildcard ? corresponds to any character, and wildcard @ corresponds to sequence of numerical digits. For example, sequence containing files xyz_000.png, xyz_001.png, xyz_002.png, etc. could be read with template xyz_@.png.

data type [input]

Data type: string

Default value: “”

Data type of the image. Can be uint8, uint16, uint32, uint64, int8, int16, int32, int64, float32, or complex32. Specify empty value to infer data type from file content.

3.8.155 readblock

There are 2 forms of this command.

readblock(image name, filename, x, y, z, block width, block height, block depth, data type)

Reads a block of an image from a file. If the file is a .raw file, special conditions apply: Dimensions of the .raw file do not need to be specified if the file name is in format name_WxHxD.raw, where [W, H, D] are the dimensions of the image. The system tries to guess the pixel data type, too, based on the file size and dimensions of the image as follows. If pixel size in bytes is 1, the system sets the pixel type to uint8. If pixel size in bytes is 2, the system sets the pixel type to uint16. If pixel size in bytes is 4, float32 pixel data is assumed (instead of e.g. int32 or uint32). If pixel size in bytes is 8, pixels are assumed to be of type uint64 (instead of e.g. int64 or complex32). If the guess is wrong, the pixel data type must be explicitly specified using the corresponding argument. Even in this case the dimensions can be read from the name of the file if the file name contains the dimensions.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image name [input]

Data type: string

Name of image in the system.

filename [input]

Data type: string

Name (and path) of file to read.

x [input]

Data type: integer

X-coordinate of the first pixel to read.

y [input]

Data type: integer

Y-coordinate of the first pixel to read.

z [input]

Data type: integer

Z-coordinate of the first pixel to read.

block width [input]

Data type: integer

Width of block to read.

block height [input]

Data type: integer

Height of block to read.

block depth [input]

Data type: integer

Depth of block to read.

data type [input]

Data type: string

Default value: “”

Data type of the image. Used only if reading .raw images. Leave empty to guess data type based on file size.

`readblock(image name, filename, position, block size, data type)`

Reads a block of an image from a file. If the file is a .raw file, special conditions apply: Dimensions of the .raw file do not need to be specified if the file name is in format name_WxHxD.raw, where [W, H, D] are the dimensions of the image. The system tries to guess the pixel data type, too, based on the file size and dimensions of the image as follows. If pixel size in bytes is 1, the system sets the pixel type to uint8. If pixel size in bytes is 2, the system sets the pixel type to uint16. If pixel size in bytes is 4, float32 pixel data is assumed (instead of e.g. int32 or uint32). If pixel size in bytes is 8, pixels are assumed to be of type uint64 (instead of e.g. int64 or complex32). If the guess is wrong, the pixel data type must be explicitly specified using the corresponding argument. Even in this case the dimensions can be read from the name of the file if the file name contains the dimensions.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image name [input]

Data type: string

Name of image in the system.

filename [input]

Data type: string

Name (and path) of file to read.

position [input]

Data type: 3-component integer vector

Coordinates of the first pixel to read.

block size [input]

Data type: 3-component integer vector

Dimensions of the block to read.

data type [input]

Data type: string

Default value: “”

Data type of the image. Used only if reading .raw images. Leave empty to guess data type based on file size.

3.8.156 readmeta

Syntax: `readmeta(image, filename)`

Reads image metadata from disk.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to process.

filename [input]

Data type: string

Name and path of file to read from.

See also

setmeta, *getmeta*, *writemeta*, *readmeta*, *clearmeta*, *listmeta*, *copymeta*, *metarowcount*, *metacolumncount*

3.8.157 readnn5block

There are 2 forms of this command.

`readnn5block(image name, filename, x, y, z, block width, block height, block depth)`

Reads a block of an NN5 dataset.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image name [input]

Data type: string

Name of image in the system.

filename [input]

Data type: string

Name (and path) of the dataset to read.

x [input]

Data type: integer

X-coordinate of the first pixel to read.

y [input]

Data type: integer

Y-coordinate of the first pixel to read.

z [input]

Data type: integer

Z-coordinate of the first pixel to read.

block width [input]

Data type: integer

Width of block to read.

block height [input]

Data type: integer

Height of block to read.

block depth [input]

Data type: integer

Depth of block to read.

readnn5block(image name, filename, position, block size)

Reads a block of an NN5 dataset.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image name [input]

Data type: string

Name of image in the system.

filename [input]

Data type: string

Name (and path) of the dataset to read.

position [input]

Data type: 3-component integer vector

Coordinates of the first pixel to read.

block size [input]

Data type: 3-component integer vector

The size of the block to read.

3.8.158 readraw

There are 2 forms of this command.

readraw(image name, filename, data type, width, height, depth)

Reads a .raw image from a file. If the image is not in the native byte order of the host computer, the byte order may be changed using *swapbyteorder* command. Dimensions of the .raw file do not need to be specified if the file name is in format name_WxHxD.raw, where [W, H, D] are the dimensions of the image. The system tries to guess the pixel data type, too, based on the file size and dimensions of the image as follows. If pixel size in bytes is 1, the system sets the pixel type to uint8. If pixel size in bytes is 2, the system sets the pixel type to uint16. If pixel size in bytes is 4, float32 pixel data is assumed (instead of e.g. int32 or uint32). If pixel size in bytes is 8, pixels are assumed to be of type uint64 (instead of e.g. int64 or complex32). If the guess is wrong, the pixel data type must be explicitly specified using the corresponding argument. Even in this case the dimensions can be read from the name of the file if the file name contains the dimensions.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image name [input]

Data type: string

Name of image to create.

filename [input]

Data type: string

Name (and path) of file to read.

data type [input]

Data type: string

Data type of the image. Can be uint8, uint16, uint32, uint64, int8, int16, int32, int64, float32, or complex32. Specify empty value to infer data type from file size.

width [input]

Data type: integer

Width of the image. Omit width, height and depth to infer dimensions from file name.

height [input]

Data type: integer

Height of the image. Omit width, height and depth to infer dimensions from file name.

depth [input]

Data type: integer

Depth of the image. Omit width, height and depth to infer dimensions from file name.

See also

mapraw, *readrawblock*

`readraw(image name, filename, data type, dimensions)`

Reads a .raw image from a file. If the image is not in the native byte order of the host computer, the byte order may be changed using *swapbyteorder* command. Dimensions of the .raw file do not need to be specified if the file name is in format name_WxHxD.raw, where [W, H, D] are the dimensions of the image. The system tries to guess the pixel data type, too, based on the file size and dimensions of the image as follows. If pixel size in bytes is 1, the system sets the pixel type to uint8. If pixel size in bytes is 2, the system sets the pixel type to uint16. If pixel size in bytes is 4,

float32 pixel data is assumed (instead of e.g. int32 or uint32). If pixel size in bytes is 8, pixels are assumed to be of type uint64 (instead of e.g. int64 or complex32). If the guess is wrong, the pixel data type must be explicitly specified using the corresponding argument. Even in this case the dimensions can be read from the name of the file if the file name contains the dimensions.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image name [input]

Data type: string

Name of image to create.

filename [input]

Data type: string

Name (and path) of file to read.

data type [input]

Data type: string

Default value: “”

Data type of the image. Can be uint8, uint16, uint32, uint64, int8, int16, int32, int64, float32, or complex32. Specify empty value to infer data type from file size.

dimensions [input]

Data type: 3-component integer vector

Default value: “[0, 0, 0]”

Size of the image. Set to zero to infer dimensions from file name.

See also

mapraw, *readrawblock*

3.8.159 readrawblock

There are 2 forms of this command.

readrawblock(image name, filename, x, y, z, block width, block height, block depth, data type, total width, total height, total depth)

Reads a block of a .raw image file. Dimensions of the .raw file do not need to be specified if the file name is in format name_WxHxD.raw, where [W, H, D] are the dimensions of the image. The system tries to guess the pixel data type, too, based on the file size and dimensions of the image as follows. If pixel size in bytes is 1, the system sets the pixel type to uint8. If pixel size in bytes is 2, the system sets the pixel type to uint16. If pixel size in bytes is 4, float32 pixel data is assumed (instead of e.g. int32 or uint32). If pixel size in bytes is 8, pixels are assumed to be of type uint64 (instead of e.g. int64 or complex32). If the guess is wrong, the pixel data type must be explicitly specified using the corresponding argument. Even in this case the dimensions can be read from the name of the file if the file name contains the dimensions.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image name [input]

Data type: string

Name of image in the system.

filename [input]

Data type: string

Name (and path) of file to read.

x [input]

Data type: integer

X-coordinate of the first pixel to read.

y [input]

Data type: integer

Y-coordinate of the first pixel to read.

z [input]

Data type: integer

Z-coordinate of the first pixel to read.

block width [input]

Data type: integer

Width of block to read.

block height [input]

Data type: integer

Height of block to read.

block depth [input]

Data type: integer

Depth of block to read.

data type [input]

Data type: string

Default value: “”

Data type of the image. Can be uint8, uint16, uint32, uint64, int8, int16, int32, int64, float32, or complex32. Specify empty value to infer data type from image dimensions

total width [input]

Data type: integer

Default value: 0

Width of the image. Omit width, height and depth to infer dimensions from file name.

total height [input]

Data type: integer

Default value: 0

Height of the image. Omit width, height and depth to infer dimensions from file name.

total depth [input]

Data type: integer

Default value: 0

Depth of the image. Omit width, height and depth to infer dimensions from file name.

See also

mapraw, *readraw*

`readrawblock(image name, filename, position, block size, data type, image size)`

Reads a block of a .raw image file. Dimensions of the .raw file do not need to be specified if the file name is in format name_WxHxD.raw, where [W, H, D] are the dimensions of the image. The system tries to guess the pixel data type, too, based on the file size and dimensions of the image as follows. If pixel size in bytes is 1, the system sets the pixel type to uint8. If pixel size in bytes is 2, the system sets the pixel type to uint16. If pixel size in bytes is 4, float32 pixel data is assumed (instead of e.g. int32 or uint32). If pixel size in bytes is 8, pixels are assumed to be of type uint64 (instead of e.g. int64 or complex32). If the guess is wrong, the pixel data type must be explicitly specified using the corresponding argument. Even in this case the dimensions can be read from the name of the file if the file name contains the dimensions.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image name [input]

Data type: string

Name of image in the system.

filename [input]

Data type: string

Name (and path) of file to read.

position [input]

Data type: 3-component integer vector

Coordinates of the first pixel to read.

block size [input]

Data type: 3-component integer vector

The size of the block to read.

data type [input]

Data type: string

Default value: “”

Data type of the image. Can be uint8, uint16, uint32, uint64, int8, int16, int32, int64, float32, or complex32. Specify empty value to infer data type from image dimensions

image size [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

Dimensions of the full image. Set to zero to infer dimensions from file name.

See also

mapraw, *readraw*

3.8.160 readsequence

Syntax: `readsequence(image name, filename template)`

Reads an image sequence from disk. Supports any image formats supported by the back end (at least .png).

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image name [input]

Data type: string

Name of image in the system.

filename template [input]

Data type: string

Name (and path) template of the sequence to be read. If a directory name is given, all files in the directory will be read. If the a file name is given, it can contain wildcards $*$, ? and @. Wildcard $*$ corresponds to any sequence of characters, wildcard ? corresponds to any character, and wildcard @ corresponds to sequence of numerical digits. For example, sequence containing files xyz_000.png, xyz_001.png, xyz_002.png, etc. could be read with template xyz_@.png.

3.8.161 readsequenceblock

There are 2 forms of this command.

`readsequenceblock(image name, filename, x, y, z, block width, block height, block depth)`

Reads a block of an image sequence.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image name [input]

Data type: string

Name of image in the system.

filename [input]

Data type: string

Name (and path) of file to read.

x [input]

Data type: integer

X-coordinate of the first pixel to read.

y [input]

Data type: integer

Y-coordinate of the first pixel to read.

z [input]

Data type: integer

Z-coordinate of the first pixel to read.

block width [input]

Data type: integer

Width of block to read.

block height [input]

Data type: integer

Height of block to read.

block depth [input]

Data type: integer

Depth of block to read.

readsequenceblock(image name, filename, position, block size)

Reads a block of an image sequence.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image name [input]

Data type: string

Name of image in the system.

filename [input]

Data type: string

Name (and path) of file to read.

position [input]

Data type: 3-component integer vector

Coordinates of the first pixel to read.

block size [input]

Data type: 3-component integer vector

The size of the block to read.

3.8.162 readvol

Syntax: `readvol(image name, filename)`

Reads a .vol image from a file.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image name [input]

Data type: string

Name of image in the system.

filename [input]

Data type: string

Name (and path) of file to read.

3.8.163 real

Syntax: `real(image)`

Calculates real part of pixel values.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: complex32 image

Image to process.

3.8.164 regionremoval

Syntax: `regionremoval(image, volume threshold, connectivity, allow multi-threading)`

Removes nonzero foreground regions smaller than given threshold. This command supports only binary images (where regions are separated by backround).

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

volume threshold [input]

Data type: positive integer

Default value: 600

All nonzero regions consisting of less than this many pixels are removed.

connectivity [input]

Data type: connectivity

Default value: Nearest

Connectivity of the particles. Can be Nearest for connectivity to nearest neighbours only, or All for connectivity to all neighbours.

allow multi-threading [input]

Data type: boolean

Default value: True

Set to true to allow multi-threaded processing. Set to false to use single-threaded processing. Single-threaded processing is often faster if it is known in advance that there are only a few particles or if the image is small. This argument has no effect in the distributed processing mode. There, the processing is always multi-threaded.

See also

openingfilter, *closingfilter*, *analyzeparticles*

3.8.165 removeedges

Syntax: `removeedges(vertices, edges, edge measurements, edge points, flags, disconnect straight-through nodes, remove isolated nodes)`

Prunes the network by removing user-selected edges from it. The edges to be removed are specified in an image, having one pixel for each edge, where non-zero value specifies that the corresponding edge is to be removed.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

vertices [input & output]

Data type: float32 image

Image where vertex coordinates are stored. See *tracelineskeleton* command.

edges [input & output]

Data type: uint64 image

Image where vertex indices corresponding to each edge are stored. See *tracelineskeleton* command.

edge measurements [input & output]

Data type: float32 image

Image where properties of each edge are stored. ee *tracelineskeleton* command.

edge points [input & output]

Data type: int32 image

Image that stores some points on each edge. See *tracelineskeleton* command.

flags [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image that has as many pixels as there are edges. Edges corresponding to non-zero pixels are removed.

disconnect straight-through nodes [input]

Data type: boolean

If set to true, all straight-through nodes (nodes with degree = 2) are removed after pruning. This operation might change the network even if no edges are pruned.

remove isolated nodes [input]

Data type: boolean

If set to true, all isolated nodes (nodes with degree = 0) are removed after pruning. This operation might change the network even if no edges are pruned.

See also

cleanskeleton, *pruneskeleton*, *removeedges*, *fillskeleton*, *getpointsandlines*

3.8.166 replace

Syntax: `replace(image, a, b)`

Finds pixels that have value a and sets their values to b.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

a [input]

Data type: real

Value to be replaced by b.

b [input]

Data type: real

Value that replaces a.

3.8.167 resettiming

Syntax: `resettiming()`

Zeroes all existing timing data.

This command can be used in the distributed processing mode, but it does not participate in distributed processing.

See also

help, *info*, *license*, *echo*, *print*, *waitreturn*, *hello*, *timing*, *savetiming*, *resettiming*

3.8.168 reslice

Syntax: `reslice(input image, output image, direction)`

Rotates the input image like a 3D cube such that the face of the cube defined by 'direction' argument will be the first slice in the output image.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Output image.

direction [input]

Data type: string

The reslice direction. Can be Top, Bottom, Left, or Right.

See also

rot90cw, *rot90ccw*, *rotate*, *flip*, *reslice*, *crop*, *copy*, *scalelabels*

3.8.169 rot90ccw

Syntax: `rot90ccw(input image, output image)`

Rotates input image counterclockwise 90 degrees around the z-axis.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Output image.

See also

rot90cw, *rot90ccw*, *rotate*, *flip*, *reslice*, *crop*, *copy*, *scalelabels*

3.8.170 rot90cw

Syntax: `rot90cw(input image, output image)`

Rotates input image clockwise 90 degrees around the z-axis.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Output image.

See also

rot90cw, *rot90ccw*, *rotate*, *flip*, *reslice*, *crop*, *copy*, *scalelabels*

3.8.171 rotate

There are 2 forms of this command.

```
rotate(input image, output image, angle, axis, input center, output center,
interpolation mode, boundary condition, block origin, full input dimensions)
```

Rotates input image around given axis. NOTE: This command does not set the size of the output image automatically. Please set the size of the output to the desired value before calling this command.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Output image.

angle [input]

Data type: real

Rotation angle in radians.

axis [input]

Data type: 3-component real vector

Default value: "[0, 0, 1]"

Rotation axis. Does not need to be unit vector.

input center [input]

Data type: 3-component real vector

Rotation center in the input image.

output center [input]

Data type: 3-component real vector

The rotation center in the input image is mapped to this point in the output image.

interpolation mode [input]

Data type: interpolation mode

Default value: Linear

Interpolation mode. Can be Nearest for nearest neighbour interpolation, Linear for linear interpolation, or Cubic for cubic interpolation.

boundary condition [input]

Data type: boundary condition

Default value: Zero

Boundary condition. Zero indicates that values outside of image bounds are taken to be zero. Nearest indicates that the nearest value inside the image is to be used in place of values outside of image bounds.

block origin [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

This argument is used internally in distributed processing. It is assigned the origin of the current calculation block. In normal operation it should be assigned to zero vector.

full input dimensions [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

This argument is used internally in distributed processing. It is assigned the full dimensions of the input image. In normal operation it should be assigned to zero vector.

See also

rot90cw, *rot90ccw*, *rotate*, *flip*, *reslice*, *crop*, *copy*, *scalelabels*

`rotate(input image, output image, angle, axis, interpolation mode, boundary condition, block origin, full input dimensions)`

Rotates input image around given axis. Rotation center is in the center of the input image, and it is mapped to the center of the output image. NOTE: This command does not set the size of the output image automatically. Please set the size of the output to the desired value before calling this command.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Output image.

angle [input]

Data type: real

Rotation angle in radians.

axis [input]

Data type: 3-component real vector

Default value: "[0, 0, 1]"

Rotation axis. Does not need to be unit vector.

interpolation mode [input]

Data type: interpolation mode

Default value: Linear

Interpolation mode. Can be Nearest for nearest neighbour interpolation, Linear for linear interpolation, or Cubic for cubic interpolation.

boundary condition [input]

Data type: boundary condition

Default value: Zero

Boundary condition. Zero indicates that values outside of image bounds are taken to be zero. Nearest indicates that the nearest value inside the image is to be used in place of values outside of image bounds.

block origin [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

This argument is used internally in distributed processing. It is assigned the origin of the current calculation block. In normal operation it should be assigned to zero vector.

full input dimensions [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

This argument is used internally in distributed processing. It is assigned the full dimensions of the input image. In normal operation it should be assigned to zero vector.

See also

rot90cw, *rot90ccw*, *rotate*, *flip*, *reslice*, *crop*, *copy*, *scalelabels*

3.8.172 round

Syntax: `round(image)`

Rounds pixel values.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

3.8.173 rounddistanceridge2

Syntax: `rounddistanceridge2(image)`

Rounds distance values to nearest integer. Inputs and outputs squared distance values.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

See also

tmap

3.8.174 satofilter

Syntax: `satofilter(input image, output image, spatial sigma, scale, gamma, gamma23, gamma12, alpha)`

Calculates line-enhancing filter lambda123 according to Sato - Three-dimensional multi-scale line filter for segmentation and visualization of curvilinear structures in medical images.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Output image.

spatial sigma [input]

Data type: real

Standard deviation of Gaussian kernel, determines scale of structures that are probed.

scale [input]

Data type: real

Default value: 0

Output values are scaled by this number. Pass in zero to scale output value 1 to maximum of the data type for integer data types and to 1 for floating point data types.

gamma [input]

Data type: real

Default value: 0

Scale-space scaling exponent. Set to zero to disable scaling.

gamma23 [input]

Data type: real

Default value: 1

γ_{23} parameter; controls the sharpness of the selectivity for the cross-section isotropy. Only non-negative values are valid.

gamma12 [input]

Data type: real

Default value: 1

γ_{12} parameter. Non-negative values are valid.

alpha [input]

Data type: real

Default value: 0.5

Alpha parameter, must be between 0 and 1.

See also

frangifilter

3.8.175 savetiming

Syntax: `savetiming(file name)`

Saves timing information to a file. Running this command causes all delayed commands to be executed. The output includes the following time classes.

Overhead

General overhead, e.g. parsing inputs, finding correct commands to run etc. This includes the total overhead time spent in the main process, and in possible cluster job processes.

I/O

Time spent in I/O-bound processing. This is the time when the disk I/O is the bottleneck. This includes the total I/O time spent in the main process, and in possible cluster job processes. Time spent in output data compression is counted to this time class.

Computation

Time spent in CPU/GPU-bound processing. This is the time when the CPU/GPU is the bottleneck. This includes the total computation time spent in the main process, and in possible cluster job processes. This is the default mode for all commands.

Job execution

Total distributed job execution time. This value includes Overhead+IO+Computation of all jobs, plus workload manager node reservation, process starting, etc. overhead. This value does not include time spent in workload manager queue.

Job queuing

Total distributed job queuing time. This is the total time all jobs have spent in the workload manager queue, waiting to be executed.

Total job waiting

Total time from submitting the first distributed job until all of them are found to be finished. This is the total time spent in the job execution process, from submission to jobs until all of them are done.

Write preparation

Time spent in preparing for writing output images (e.g. NN5 write preparation). This is the total time spent in the submitting process while preparing writing of output images.

Total write finalization waiting

Time spent in write finalization jobs, including queuing (e.g. NN5 write finalization jobs). This is the total time spent in the submitting process, from submission of write finalization jobs until all of them are done.

This command can be used in the distributed processing mode, but it does not participate in distributed processing.

Arguments

file name [input]

Data type: string

The name and path of the file where the information is to be saved.

See also

help, *info*, *license*, *echo*, *print*, *waitreturn*, *hello*, *timing*, *savetiming*, *resettiming*

3.8.176 scale

Syntax: `scale(input image, output image, scaling factor, average when downsizing, interpolation mode, boundary condition, block origin)`

Scales input image and places the result into the output image. Set size of output image before calling this command or specify scaling factor as an argument. Does not suppress aliasing artifacts when downscaling unless average when downsizing-parameter is set to true.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Output image.

scaling factor [input]

Data type: 3-component real vector

Default value: "[0, 0, 0]"

Scaling factor in each coordinate direction. If zero in some dimension, the factor is calculated from current size of the output image and the input image in that dimension.

average when downsizing [input]

Data type: boolean

Default value: False

Set to true to average when downsizing.

interpolation mode [input]

Data type: interpolation mode

Default value: Linear

Interpolation mode. Can be Nearest for nearest neighbour interpolation, Linear for linear interpolation, or Cubic for cubic interpolation.

boundary condition [input]

Data type: boundary condition

Default value: Nearest

Type of boundary condition. Zero indicates that values outside of image bounds are taken to be zero. Nearest indicates that the nearest value inside the image is to be used in place of values outside of image bounds.

block origin [input]

Data type: 3-component integer vector

Default value: "[-1, -1, -1]"

Specifies origin of current calculation block. This parameter is used internally in distributed processing and should be set to (-1, -1, -1) in normal use.

See also

scale, *bin*, *maskedbin*, *scalelabels*

3.8.177 scalelabels

Syntax: `scalelabels(input image, output image, scaling factor, block origin)`

Scales input image and places the result into the output image. Assumes that input image is a binary image or contains label values, and performs scaling such that edges of regions do not become jagged. Supports only upscaling by integer scaling factor. Set size of output image before calling this command or specify scaling factor as an argument.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Output image.

scaling factor [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

Scaling factor in each coordinate direction. If zero in some dimension, the factor is calculated from current size of the output image and the input image in that dimension.

block origin [input]

Data type: 3-component integer vector

Default value: "[-1, -1, -1]"

Specifies origin of current calculation block. This parameter is used internally in distributed processing and should be set to (-1, -1, -1) in normal use.

See also

scale, *bin*, *maskedbin*, *scalelabels*

3.8.178 sdmap

Syntax: `sdmap(seeds, geometry, output, connectivity)`

Calculates seeded distance map of a binary image. For each pixel in the image, finds the shortest pixel-by-pixel path to nearest seed region such that the path does not pass through regions that are marked as obstacles. The output is the length of the shortest path, calculated as sum of pixel-to-pixel steps that must be taken to follow the path. This is in contrast to the *dmap* command where the distance to the seed region is direct Euclidean distance and no separate obstacles are supported. Please note that comparing the output of *dmap* and *sdmap* commands might lead to surprising results due to the different definition of distance.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

seeds [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Seed image that contains the set of pixels where the distance is zero. The set is marked with nonzero values, i.e. the distance map will propagate to pixels that have zero value in this image. This image is not modified.

geometry [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image containing the geometry in which to calculate the distances. Regions marked with zero pixel value are 'unpassable obstacles'. The distance transform will only proceed to pixels whose color in this image matches the color of the seed point (in this image). This image is not modified.

output [output]

Data type: float32 image

Output image that will contain the distance to the nearest seed region, not passing through zero regions in the geometry image.

connectivity [input]

Data type: connectivity

Default value: All

Connectivity of the distance map. Can be Nearest for connectivity to nearest neighbours only, or All for connectivity to all neighbours.

See also

dmap, *dmap2*, *tmap*, *sdmap*

3.8.179 sequenceinfo

Syntax: `sequenceinfo(filename template)`

Displays metadata of image sequence.

This command can be used in the distributed processing mode, but it does not participate in distributed processing.

Arguments

filename template [input]

Data type: string

Filename template corresponding to the sequence, as described in readsequence command documentation.

3.8.180 set

There are 8 forms of this command.

`set(name, value)`

Sets value of a string variable

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

name [output]

Data type: string

Variable to set.

value [input]

Data type: string

New value.

See also

set, *clear*

`set(name, value)`

Sets value of an integer variable

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

name [output]

Data type: integer

Variable to set.

value [input]

Data type: integer

New value.

See also

set, *clear*

`set(name, value)`

Sets value of an real number variable

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

name [output]

Data type: real

Variable to set.

value [input]

Data type: real

New value.

See also

set, *clear*

`set(name, value)`

Sets value of an boolean variable

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

name [output]

Data type: boolean

Variable to set.

value [input]

Data type: boolean

New value.

See also

set, *clear*

`set(target image, source image)`

Copies pixel values from the source image to the target image. Sets the size and data type of the target image to those of the source image. See also *copy* command.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

target image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image whose values are set.

source image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image that is copied to the target image.

`set(image, x)`

Sets all pixels to the same value.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

x [input]

Data type: real

Pixel value.

`set(image, position, value, block origin)`

Sets a pixel in the image.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

position [input]

Data type: 3-component integer vector

Position to set.

value [input]

Data type: real

Value that the pixel at given position should be set to.

block origin [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

Origin of current calculation block in coordinates of the full image. This argument is used internally in distributed processing. Set to zero in normal usage.

See also

line, *capsule*, *sphere*, *ellipsoid*, *set*, *get*, *ramp*, *ramp3*

`set(image, positions, values)`

Sets values of multiple pixels in an image. Points outside of the image are not set.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image where the pixels are set.

positions [input]

Data type: float32 image

Positions of pixels to set. Each row of this image contains (x, y, z) coordinates of a pixel to be set. The size of the image must be 3xN where N is the count of pixels to write. Floating point values are rounded to the nearest integer.

values [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Values of the pixels to be set. The size of the image must be the number of pixels to set.

See also

set, *set*, *getpointsandlines*

3.8.181 setafterheightmap

Syntax: `setafterheightmap(geometry, height map, visualization color)`

Sets values of all pixels located after (below) the given height map.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

geometry [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image where whose pixels are to be set.

height map [input]

Data type: float32 image

Height map. The size of the height map must be $w \times h$ where w and h are the width and the height of the geometry image.

visualization color [input]

Data type: real

Color of the surface in the visualization.

See also

findsurface, *drawheightmap*, *setbeforeheightmap*, *setafterheightmap*, *shiftz*

3.8.182 setbeforeheightmap

Syntax: `setbeforeheightmap(geometry, height map, visualization color)`

Sets values of all pixel located before (above) the given height map.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

geometry [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image where whose pixels are to be set.

height map [input]

Data type: float32 image

Height map. The size of the height map must be $w \times h$ where w and h are the width and the height of the geometry image.

visualization color [input]

Data type: real

Color of the surface in the visualization.

See also

findsurface, *drawheightmap*, *setbeforeheightmap*, *setafterheightmap*, *shiftz*

3.8.183 setedges

Syntax: `setedges(image, edge value, radius)`

Set edges of the image to specified value.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

edge value [input]

Data type: real

Default value: 0

The edges are set to this value.

radius [input]

Data type: integer

Default value: 1

Pixels whose distance to the image edge is less than or equal to this value are set.

3.8.184 setmeta

Syntax: `setmeta(image, key, value, column, row)`

Sets metadata item of an image.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to process.

key [input]

Data type: string

Name of the metadata item.

value [input]

Data type: string

Value of the metadata item.

column [input]

Data type: positive integer

Default value: 0

Column index of the item to set in the data matrix.

row [input]

Data type: positive integer

Default value: 0

Row index of the item to set in the data matrix.

See also

setmeta, *getmeta*, *writemeta*, *readmeta*, *clearmeta*, *listmeta*, *copymeta*, *metarowcount*, *metacolumncount*

3.8.185 shiftz

Syntax: `shiftz(image, shift map, subtract mean, interpolation mode, boundary condition)`

Shift each z-directional column of input image by amount given in the shift map.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image whose z-directional columns are to be shifted.

shift map [input]

Data type: float32 image

Shift map that gives the amount of shift to apply in each z-directional column. The size of the shift map must be $w \times h$ where w and h are the width and the height of the geometry image.

subtract mean [input]

Data type: boolean

Set to true to automatically subtract the average value of the shift map from each shift. This is useful if the shift map is negation of a surface map found using the findsurface command, and the intent is to make the surface straight.

interpolation mode [input]

Data type: interpolation mode

Default value: Linear

Interpolation mode. Can be Nearest for nearest neighbour interpolation, Linear for linear interpolation, or Cubic for cubic interpolation.

boundary condition [input]

Data type: boundary condition

Default value: Zero

Boundary condition. Zero indicates that values outside of image bounds are taken to be zero. Nearest indicates that the nearest value inside the image is to be used in place of values outside of image bounds.

See also

findsurface, *drawheightmap*, *setbeforeheightmap*, *setafterheightmap*, *shiftz*

3.8.186 sin

Syntax: `sin(image)`

Calculates sine of pixel values.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to process.

3.8.187 sphere

Syntax: `sphere(image, position, radius, value, block origin)`

Draws a filled sphere into the image.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

position [input]

Data type: 3-component real vector

Default value: "[0, 0, 0]"

Position of the center point of the sphere.

radius [input]

Data type: real

Default value: 10

Radius of the sphere.

value [input]

Data type: real

Default value: 1

Value for pixels inside the sphere.

block origin [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

Origin of current calculation block in coordinates of the full image. This argument is used internally in distributed processing. Set to zero in normal usage.

See also

line, *capsule*, *sphere*, *ellipsoid*, *set*, *get*, *ramp*, *ramp3*

3.8.188 square

Syntax: `square(image)`

Calculates square of pixel values.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to process.

3.8.189 squareroot

Syntax: `squareroot(image)`

Calculates square root of pixel values.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to process.

3.8.190 squaresum

Syntax: `squaresum(input image, output image, print to log)`

Calculates squareSum of all pixels in the input image. The output is a 1x1x1 image.

Note: In Python/pi2py2, the output image is not specified, and the result value is returned by the function.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image

Output image.

print to log [input]

Data type: boolean

Default value: False

Set to true to print the results to the log.

3.8.191 stddev

Syntax: `stddev(input image, output image, print to log)`

Calculates standard deviation of all pixels in the input image. The output is a 1x1x1 image.

Note: In Python/pi2py2, the output image is not specified, and the result value is returned by the function.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: float32 image

Output image.

print to log [input]

Data type: boolean

Default value: False

Set to true to print the results to the log.

3.8.192 stddevfilter

Syntax: `stddevfilter(input image, output image, radius, neighbourhood type, boundary condition)`

Standard deviation filter. Replaces pixel by standard deviation of pixels in its neighbourhood. For accurate results, use on images with floating point data type. Has optimized implementation for rectangular neighbourhoods.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Output image.

radius [input]

Data type: 3-component integer vector

Default value: "[1, 1, 1]"

Radius of neighbourhood. Diameter will be $2r + 1$.

neighbourhood type [input]

Data type: neighbourhood type

Default value: Ellipsoidal

Type of neighbourhood. Can be Ellipsoidal for ellipsoidal or spherical neighbourhood; or Rectangular for rectangular neighbourhood.

boundary condition [input]

Data type: boundary condition

Default value: Nearest

Type of boundary condition. Zero indicates that values outside of image bounds are taken to be zero. Nearest indicates that the nearest value inside the image is to be used in place of values outside of image bounds.

See also

gaussfilter, *bilateralfilter*, *bilateralfilterapprox*, *vawefilter*, *openingfilter*, *closingfilter*, *minfilter*, *maxfilter*, *medianfilter*, *variancefilter*, *stddevfilter*, *bandpassfilter*, *highpassfilter*

3.8.193 submitjob

Syntax: `submitjob(script, job type)`

Submits a pi2 script job to the active cluster system, or runs it locally if distributed processing is not active as if the commands in the parameter script would be run instead of this command. Does not wait for job completion. Typically you would not use this method of job submission, but just call *distribute* function and let the system deal with submitting jobs.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

script [input]

Data type: string

Pi2 script to run.

job type [input]

Data type: string

Default value: normal

Job type, either fast, normal, or slow.

See also

submitjob, *waitforjobs*

3.8.194 subtract

There are 2 forms of this command.

`subtract(image, parameter image, allow broadcast)`

Subtracts two images. Output is placed to the first image. The operation is performed using saturation arithmetic.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to process.

parameter image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image, complex32 image

Parameter image.

allow broadcast [input]

Data type: boolean

Default value: False

Set to true to allow size of parameter image differ from size of input image. If there is a need to access pixel outside of parameter image, the nearest value inside the image is taken instead. If set to false, dimensions of input and parameter images must be equal. If set to true, the parameter image is always loaded in its entirety in distributed processing mode.

`subtract(image, x)`

Subtracts a constant from the image. The operation is performed using saturation arithmetic.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

x [input]

Data type: real

Constant to subtract from the image.

3.8.195 sum

Syntax: `sum(input image, output image, print to log)`

Calculates sum of all pixels in the input image. The output is a 1x1x1 image.

Note: In Python/pi2py2, the output image is not specified, and the result value is returned by the function.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image

Output image.

print to log [input]

Data type: boolean

Default value: False

Set to true to print the results to the log.

3.8.196 sumproject

Syntax: `sumproject(input image, output image, dimension)`

Calculates projection of the input image. The dimensionality of the output image is the dimensionality of the input image subtracted by one.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: float32 image

Output image.

dimension [input]

Data type: positive integer

Default value: 2

Dimension to project over, zero corresponding to x, one corresponding to y, and 2 corresponding to z.

3.8.197 surfacearea

Syntax: `surfacearea(geometry, surface area, isovalue)`

Calculates the total surface area between foreground and background in an image.Foreground and background are defined by a
R"(Uses Marching Cubes algorithm.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

geometry [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

An image containing the input geometry. If using a non-binary image, please specify isovalue parameter, too.

surface area [output]

Data type: real

The total surface area in square pixels will be returned in this value.

isovalue [input]

Data type: real

Default value: 1

Threshold value that separates foreground and background.

See also

curvature, *derivative*

3.8.198 surfaceskeleton

Syntax: `surfaceskeleton(image, retain surfaces)`

Calculates skeleton of the foreground of the given image. Positive pixels are assumed to belong to the foreground. The skeleton may contain both lines and plates. This command is not guaranteed to give the same result in both normal and distributed processing mode. Despite that, both modes should give a valid result.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

retain surfaces [input]

Data type: boolean

Default value: True

Set to false to allow thinning of surfaces to lines if the surface does not surround a cavity.

See also

surfacethin, *surfaceskeleton*, *linethin*, *lineskeleton*, *tracelineskeleton*, *classifyskeleton*

3.8.199 surfacethin

Syntax: `surfacethin(image, retain surfaces)`

Thins one layer of pixels from the foreground of the image. Positive pixels are assumed to belong to the foreground. Run iteratively to calculate a surface skeleton. This command is not guaranteed to give the same result in both normal and distributed processing mode. Despite that, both modes should give a valid result.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

retain surfaces [input]

Data type: boolean

Default value: True

Set to false to allow thinning of surfaces to lines if the surface does not surround a cavity.

See also

surfacethin, *surfaceskeleton*, *linethin*, *lineskeleton*, *tracelineskeleton*, *classifyskeleton*

3.8.200 swapbyteorder

Syntax: `swapbyteorder(image)`

Swaps byte order of each pixel value. Use this command to convert images read in wrong endianness to the correct one, or before saving an image if it should be saved in non-native byte order.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to process.

3.8.201 tan

Syntax: `tan(image)`

Calculates tangent of pixel values.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to process.

3.8.202 threshold

There are 2 forms of this command.

`threshold(image, parameter image, allow broadcast)`

Thresholds left image, taking threshold values from right image. Sets pixel to 1 if pixel value > threshold and to 0 otherwise.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

parameter image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image

Parameter image.

allow broadcast [input]

Data type: boolean

Default value: False

Set to true to allow size of parameter image differ from size of input image. If there is a need to access pixel outside of parameter image, the nearest value inside the image is taken instead. If set to false, dimensions of input and parameter images must be equal. If set to true, the parameter image is always loaded in its entirety in distributed processing mode.

`threshold(image, x)`

Thresholds image. Sets pixel to 1 if pixel value > threshold and to 0 otherwise.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

x [input]

Data type: real

Threshold value.

3.8.203 thresholdperiodic

Syntax: `thresholdperiodic(image, period start, period end, min, max)`

Thresholds a range from the image where pixel values are from periodic range (e.g. angle). Sets to one all pixels whose value is in]min, max] mod period, and sets all other pixels to zero. If threshold range is larger than period, sets all pixels to 1. If threshold range start is larger than or equal to threshold range end, sets all pixels to 0.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

period start [input]

Data type: real

Minimum of the period of pixel values.

period end [input]

Data type: real

Maximum of the period of pixel values.

min [input]

Data type: real

Minimum of the threshold range.

max [input]

Data type: real

Maximum of the threshold range.

3.8.204 thresholdrange

Syntax: `thresholdrange(image, min, max)`

Thresholds a range from the image. Sets to one all pixels whose value is in]min, max], and sets all other pixels to zero.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

image [input & output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image to process.

min [input]

Data type: real

Minimum of the threshold range.

max [input]

Data type: real

Maximum of the threshold range.

3.8.205 timing

Syntax: `timing()`

Prints information about wall-clock time taken by various sub-processes. Running this command causes all delayed commands to be executed. The output includes the following time classes.

Overhead

General overhead, e.g. parsing inputs, finding correct commands to run etc. This includes the total overhead time spent in the main process, and in possible cluster job processes.

I/O

Time spent in I/O-bound processing. This is the time when the disk I/O is the bottleneck. This includes the total I/O time spent in the main process, and in possible cluster job processes. Time spent in output data compression is counted to this time class.

Computation

Time spent in CPU/GPU-bound processing. This is the time when the CPU/GPU is the bottleneck. This includes the total computation time spent in the main process, and in possible cluster job processes. This is the default mode for all commands.

Job execution

Total distributed job execution time. This value includes Overhead+IO+Computation of all jobs, plus workload manager node reservation, process starting, etc. overhead. This value does not include time spent in workload manager queue.

Job queuing

Total distributed job queuing time. This is the total time all jobs have spent in the workload manager queue, waiting to be executed.

Total job waiting

Total time from submitting the first distributed job until all of them are found to be finished. This is the total time spent in the job execution process, from submission to jobs until all of them are done.

Write preparation

Time spent in preparing for writing output images (e.g. NN5 write preparation). This is the total time spent in the submitting process while preparing writing of output images.

Total write finalization waiting

Time spent in write finalization jobs, including queuing (e.g. NN5 write finalization jobs). This is the total time spent in the submitting process, from submission of write finalization jobs until all of them are done.

This command can be used in the distributed processing mode, but it does not participate in distributed processing.

See also

help, *info*, *license*, *echo*, *print*, *waitreturn*, *hello*, *timing*, *savetiming*, *resettiming*

3.8.206 tmap

Syntax: `tmap(input image, output image, background value, round distance values, save memory, block size)`

Calculates local thickness map (i.e. opening transform) of a binary image. Input image is used as temporary storage space. The pixel data type must be able to contain large enough values; usually uint8 is too small. Uint32 or uint64 are recommended.

The algorithm selection depends on the value of the 'save memory' argument. If 'save memory' is true, the algorithm introduced in Hildebrand - A New Method for the Model-Independent Assessment of Thickness in Three-Dimensional Images is used.If 'save memory' is false, the separable algorithm in Lovric - Separable distributed local thickness algorithm for efficient morphological characterization of terabyte-scale volume images is applied. The separable algorithm is usually much faster than the Hildebrand algorithm, but requires much more RAM (normal mode) or temporary disk space (distributed mode).

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image where background is marked with background value given by the third argument.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image

Output image (thickness map) where pixel value equals diameter of object at that location, i.e. diameter of the largest sphere that fits in foreground points and contains the object.

background value [input]

Data type: real

Default value: 0

Pixels belonging to the background are marked with this value in the input image.

round distance values [input]

Data type: boolean

Default value: False

Set to true to generate approximate distance map by rounding distance values to nearest integers. Calculation of rounded distance map is faster, but the result is only approximation both in distance values and shape of structures.

save memory [input]

Data type: boolean

Default value: False

For non-distributed processing: Set to true to use slower algorithm (Hildebrand) that uses less memory than the faster one (separable). If set to true, the input and output images can be the same. For distributed processing: Only false is supported at the moment.

block size [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

Block size to use in distributed processing in sphere plotting phase. Set to zero to use a default value calculated based on available memory. If calculation with default value fails, use smaller block size and report it to the authors. This argument has no effect when running in non-distributed mode.

See also

dmap2, *dmap*

3.8.207 tracelineskeleton

There are 2 forms of this command.

tracelineskeleton(skeleton, original, vertices, edges, edge measurements, edge points, store all edge points, thread count, smoothing sigma, max displacement)

Traces a line skeleton into a graph structure. Each branch intersection point becomes a vertex in the graph and each branch becomes an edge.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

skeleton [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image containing the skeleton. The pixels of the image will be set to zero.

original [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Original image from which the skeleton has been calculated. This image is used for branch shape measurements.

vertices [output]

Data type: float32 image

Image where vertex coordinates are stored. The size of the image is set to 3xN during processing, where N is the number of vertices in the graph.

edges [output]

Data type: uint64 image

Image where vertex indices corresponding to each edge will be set. The size of the image is set to 2xM where M is the number of edges. Each row of the image consists of a pair of indices to the vertex array.

edge measurements [output]

Data type: float32 image

Image that stores (pointCount, length, cross-sectional area) for each edge. The size of the image is set to 3xN during processing, where N is the number of edges in the graph. Each row contains properties of edge at corresponding row in the edges image. Area measurements are approximations if distributed processing is allowed.

edge points [output]

Data type: int32 image

Image that stores some points on each edge. The points are required for skeleton filling commands to work correctly even if the skeleton is pruned. The points are stored such that the first edgeCount pixels of the image store count of points for each edge. The remaining pixels store (x, y, z) coordinates of each point and each edge sequentially. For example, the format for two edges that have 1 and 2 points is therefore '1 2 x11 y11 z11 x21 y21 z21 x22 y22 z22', where Aij is A-component of j:th point of edge i.

store all edge points [input]

Data type: boolean

Default value: False

Set to true to store all points of each edge to edge points image. If set to false, only single point on each edge is stored. This argument is required to be set to true if the graph will be converted to points and lines format later.

thread count [input]

Data type: positive integer

Default value: 0

Count of threads to use in the tracing process. Set to zero to determine count of threads automatically. Set to one to use single-threaded processing.

smoothing sigma [input]

Data type: real

Default value: 1

Standard deviation value to be used in smoothing of the edges (using anchored convolution) before length measurements.

max displacement [input]

Data type: real

Default value: 0.5

Maximum displacement of points in anchored convolution done before length measurements.

See also

surfaceskeleton, *lineskeleton*, *cleanskeleton*, *pruneskeleton*, *removeedges*, *fillskeleton*, *getpointsandlines*

`tracelineskeleton(skeleton, vertices, edges, edge measurements, edge points, store all edge points, thread count, smoothing sigma, max displacement)`

Traces a line skeleton into a graph structure. Each branch intersection point becomes a vertex in the graph and each branch becomes an edge.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

skeleton [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image containing the skeleton. The pixels of the image will be set to zero.

vertices [output]

Data type: float32 image

Image where vertex coordinates are stored. The size of the image is set to 3xN during processing, where N is the number of vertices in the graph.

edges [output]

Data type: uint64 image

Image where vertex indices corresponding to each edge will be set. The size of the image is set to 2xM where M is the number of edges. Each row of the image consists of a pair of indices to the vertex array.

edge measurements [output]

Data type: float32 image

Image that stores (pointCount, length, cross-sectional area) for each edge. The size of the image is set to 3xN during processing, where N is the number of edges in the graph. Each row contains properties of edge at corresponding row in the edges image.

edge points [output]

Data type: int32 image

Image that stores some points on each edge. The points are required for skeleton filling commands to work correctly even if the skeleton is pruned. The points are stored such that the first edgeCount pixels of the image store count of points for each edge. The remaining pixels store (x, y, z) coordinates of each point and each edge sequentially. For example, the format for two edges that have 1 and 2 points is therefore '1 2 x11 y11 z11 x21 y21 z21 x22 y22 z22', where Aij is A-component of j:th point of edge i.

store all edge points [input]

Data type: boolean

Default value: False

Set to true to store all points of each edge to edge points image. If set to false, only single point on each edge is stored. This argument is required to be set to true if the graph will be converted to points and lines format later.

thread count [input]

Data type: positive integer

Default value: 0

Count of threads to use in the tracing process. Set to zero to determine count of threads automatically. Set to one to use single-threaded processing.

smoothing sigma [input]

Data type: real

Default value: 1

Standard deviation value to be used in smoothing of the edges (using anchored convolution) before length measurements.

max displacement [input]

Data type: real

Default value: 0.5

Maximum displacement of points in anchored convolution done before length measurements.

See also

surfaceskeleton, *lineskeleton*, *cleanskeleton*, *pruneskeleton*, *removeedges*, *fillskeleton*, *getpointsandlines*

3.8.208 translate

Syntax: `translate(input image, output image, shift)`

Translates input image by specified amount.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Output image.

shift [input]

Data type: 3-component real vector

Translation that will be applied to the input image.

See also

rot90cw, *rot90ccw*, *rotate*, *flip*, *reslice*, *crop*, *copy*, *scalelabels*

3.8.209 variancefilter

Syntax: `variancefilter(input image, output image, radius, neighbourhood type, boundary condition)`

Variance filter. Replaces pixel by variance of pixels in its neighbourhood. For accurate results, use on images with floating point data type. Has optimized implementation for rectangular neighbourhoods.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Output image.

radius [input]

Data type: 3-component integer vector

Default value: "[1, 1, 1]"

Radius of neighbourhood. Diameter will be $2r + 1$.

neighbourhood type [input]

Data type: neighbourhood type

Default value: Ellipsoidal

Type of neighbourhood. Can be Ellipsoidal for ellipsoidal or spherical neighbourhood; or Rectangular for rectangular neighbourhood.

boundary condition [input]

Data type: boundary condition

Default value: Nearest

Type of boundary condition. Zero indicates that values outside of image bounds are taken to be zero. Nearest indicates that the nearest value inside the image is to be used in place of values outside of image bounds.

See also

gaussfilter, *bilateralfilter*, *bilateralfilterapprox*, *vawefilter*, *openingfilter*, *closingfilter*, *minfilter*, *maxfilter*, *medianfilter*, *variancefilter*, *stddevfilter*, *bandpassfilter*, *highpassfilter*

3.8.210 vawefilter

Syntax: `vawefilter(input image, output image, radius, noise standard deviation, neighbourhood type, boundary condition)`

Variance Weighted mean filtering. Removes noise from the image while trying to preserve edges. Edge preservation is achieved by filtering less aggressively in regions where local variance of pixel values is high. The filter thus assumes that high local variance corresponds to details of interest, and low local variance corresponds to regions containing solely noise.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Input image.

output image [output]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Output image.

radius [input]

Data type: 3-component integer vector

Default value: "[1, 1, 1]"

Radius of neighbourhood. Diameter will be $2r + 1$.

noise standard deviation [input]

Data type: real

Standard deviation of noise. For a rough order of magnitude estimate, measure standard deviation from a region that does not contain any features.

neighbourhood type [input]

Data type: neighbourhood type

Default value: Ellipsoidal

Type of neighbourhood. Can be Ellipsoidal for ellipsoidal or spherical neighbourhood; or Rectangular for rectangular neighbourhood.

boundary condition [input]

Data type: boundary condition

Default value: Nearest

Type of boundary condition. Zero indicates that values outside of image bounds are taken to be zero. Nearest indicates that the nearest value inside the image is to be used in place of values outside of image bounds.

See also

gaussfilter, *bilateralfilter*, *bilateralfilterapprox*, *vawefilter*, *openingfilter*, *closingfilter*, *minfilter*, *maxfilter*, *medianfilter*, *variancefilter*, *stddevfilter*, *bandpassfilter*, *highpassfilter*

3.8.211 waitforjobs

Syntax: `waitforjobs()`

Waits until all running cluster jobs are complete. Use this command to wait for jobs submitted using the submitjob command.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

See also

submitjob, *waitforjobs*

3.8.212 waitreturn

Syntax: `waitreturn()`

Waits until the user presses return key. Does not show any prompts on screen.

This command can be used in the distributed processing mode, but it does not participate in distributed processing.

See also

help, *info*, *license*, *echo*, *print*, *waitreturn*, *hello*, *timing*, *savetiming*, *resettiming*

3.8.213 whist

Syntax: `whist(input image, weight, histogram, bin starts, histogram minimum, histogram maximum, bin count, output file, block index)`

Calculate weighted histogram of an image.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

Image whose histogram will be calculated.

weight [input]

Data type: float32 image

Weight image. This image must contain weight value for each pixel in the input images

histogram [output]

Data type: float32 image

Image that will contain the histogram on output. The number of pixels in this image will be set to match bin count.

bin starts [output]

Data type: float32 image

Image that will contain the histogram bin start coordinates when the function finishes.

histogram minimum [input]

Data type: real

Default value: 0

Minimum value to be included in the histogram. Values less than minimum are included in the first bin.

histogram maximum [input]

Data type: real(255), real(65535), real(4294967295), real(1.844674407370955e+19), real(127), real(32767), real(2147483647), real(9.223372036854776e+18), real(3.402823466385289e+38)

Default value: Shown along data types.

The end of the last bin. Values greater than maximum are included in the last bin.

bin count [input]

Data type: integer

Default value: 256

Count of bins. The output image will be resized to contain this many pixels.

output file [input]

Data type: string

Default value: “”

Name of file where the histogram data is to be saved. Specify empty string to disable saving. This argument is used internally in distributed processing, but can be used to (conveniently?) save the histogram in .raw format.

block index [input]

Data type: integer

Default value: -1

Index of image block that we are currently processing. This argument is used internally in distributed processing and should normally be set to negative value. If positive, this number is appended to output file name.

3.8.214 whist2

Syntax: `whist2(first input image, min 1, max 1, bin count 1, second input image, min 2, max 2, bin count 2, weight, histogram, bin starts 1, bin starts 2, output file, block index)`

Calculate weighted bivariate histogram of two images. In the bivariate histogram position (i, j) counts total weight of locations where the value of the first input image is i and the value of the second input image is j (assuming minimum = 0, maximum = data type max, and bin size = 1). If image sizes are different, only the region available in both images is included in the histogram. In this case, a warning message is shown.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

first input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image

First image whose histogram will be calculated.

min 1 [input]

Data type: real

Default value: 0

Minimum value of first image to be included in the histogram.

max 1 [input]

Data type: real(255), real(65535), real(4294967295), real(1.844674407370955e+19), real(127), real(32767), real(2147483647), real(9.223372036854776e+18), real(3.402823466385289e+38)

Default value: Shown along data types.

The end of the last bin. This is the smallest value above minimum that is not included in the histogram.

bin count 1 [input]

Data type: integer

Default value: 256

Count of bins. The first dimension of the output image will be resized to contain this many pixels.

second input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, float32 image, int8 image, int16 image, int32 image, int64 image

Second image whose histogram will be calculated.

min 2 [input]

Data type: real

Default value: 0

Minimum value of second image to be included in the histogram.

max 2 [input]

Data type: real(255), real(65535), real(4294967295), real(1.844674407370955e+19), real(3.402823466385289e+38), real(127), real(32767), real(2147483647), real(9.223372036854776e+18)

Default value: Shown along data types.

The end of the last bin. This is the smallest value above minimum that is not included in the histogram.

bin count 2 [input]

Data type: integer

Default value: 256

Count of bins. The second dimension of the output image will be resized to contain this many pixels.

weight [input]

Data type: float32 image

Weight image. This image must contain weight value for each pixel in the input images

histogram [output]

Data type: float32 image

Image that will contain the histogram on output.

bin starts 1 [output]

Data type: float32 image

Image that will contain the histogram bin start coordinates in the first dimension when the function finishes.

bin starts 2 [output]

Data type: float32 image

Image that will contain the histogram bin start coordinates in the second dimension when the function finishes.

output file [input]

Data type: string

Default value: “”

Name of file where the histogram data is to be saved. Specify empty string to disable saving. This argument is used internally in distributed processing, but can be used to (conveniently?) save the histogram in .raw format.

block index [input]

Data type: integer

Default value: -1

Index of image block that we are currently processing. This argument is used internally in distributed processing and should normally be set to negative value. If positive, this number is appended to output file name.

3.8.215 writelz4

Syntax: `writelz4(input image, filename)`

Write an image to an .lz4raw file.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to save.

filename [input]

Data type: string

Name (and path) of the file to write. If the file exists, its current contents are erased. Extension .lz4raw is automatically appended to the name of the file.

3.8.216 writemeta

Syntax: `writemeta(image, filename)`

Writes image metadata to disk.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to process.

filename [input]

Data type: string

Name and path of file to write. The file will be replaced.

See also

setmeta, *getmeta*, *writemeta*, *readmeta*, *clearmeta*, *listmeta*, *copymeta*, *metarowcount*, *metacolumncount*

3.8.217 writenn5

Syntax: `writenn5(input image, path, chunk size)`

Write an image to an .nn5 dataset.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to save.

path [input]

Data type: string

Name (and path) of the dataset to write. If the dataset exists, its current contents are erased.

chunk size [input]

Data type: 3-component integer vector

Default value: "[1536, 1536, 1536]"

Chunk size for the NN5 dataset to be written.

3.8.218 writenn5block

Syntax: `writenn5block(input image, filename, position, file dimensions, source position, source block size, chunk size)`

Write an image to a specified position in an NN5 dataset. Optionally can write only a block of the source image.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to save.

filename [input]

Data type: string

Name (and path) of the dataset to write.

position [input]

Data type: 3-component integer vector

Position of the image in the target file.

file dimensions [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

Dimensions of the output file. Specify zero to parse dimensions from the file. In this case it must exist.

source position [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

Position of the block of the source image to write.

source block size [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

Size of the block to write. Specify zero to write the whole source image.

chunk size [input]

Data type: 3-component integer vector

Default value: "[1536, 1536, 1536]"

Size of chunks in the NN5 dataset.

3.8.219 writenrrd

Syntax: `writenrrd(input image, filename)`

Write an image to an .nrrd file.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to save.

filename [input]

Data type: string

Name (and path) of the file to write. If the file exists, its current contents are erased. Extension .nrrd is automatically appended to the name of the file.

3.8.220 writepng

Syntax: `writepng(input image, filename)`

Write an image to a .png file. This command supports 1- and 2-dimensional images only.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to save.

filename [input]

Data type: string

Name (and path) of the file to write. If the file exists, its current contents are erased. Extension .png is automatically appended to the name of the file.

3.8.221 writeraw

There are 2 forms of this command.

`writeraw(input image, filename, append)`

Write an image to .raw file. The dimensions of the image are automatically appended to the file name. If distributed processing is enabled, uses optimized implementation that does not need to copy or write any image data if the image is saved to a temporary image storage location and that location is on the same partition than the file being written.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to save.

filename [input]

Data type: string

Name (and path) of the file to write. If the file exists, its current contents are erased (unless append parameter is set to true).

append [input]

Data type: boolean

Default value: False

Set to true to append to existing .raw file. This parameter must be set to false in distributed mode.

`writeraw(r, g, b, filename, append)`

Writes an RGB image to a .raw file. The dimensions of the image are automatically appended to the file name.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

r [input]

Data type: uint8 image

Red component image.

g [input]

Data type: uint8 image

Green component image.

b [input]

Data type: uint8 image

Blue component image.

filename [input]

Data type: string

Name (and path) of the file to write. If the file exists, its current contents are erased (unless append parameter is set to true).

append [input]

Data type: boolean

Default value: False

Set to true to append to existing .raw file.

3.8.222 writerawblock

Syntax: `writerawblock(input image, filename, position, file dimensions, source position, source block size)`

Write an image to a specified position in a .raw file. Optionally can write only a block of the source image. Dimensions of the .raw file do not need to be specified if the file name is in format name_WxHxD.raw, where [W, H, D] are the dimensions of the image. The system tries to guess the pixel data type, too, based on the file size and dimensions of the image as follows. If pixel size in bytes is 1, the system sets the pixel type to uint8. If pixel size in bytes is 2, the system sets the pixel type to uint16. If pixel size in bytes is 4, float32 pixel data is assumed (instead of e.g. int32 or uint32). If pixel size in bytes is 8, pixels are assumed to be of type uint64 (instead of e.g. int64 or complex32). If the guess is wrong, the pixel data type must be explicitly specified using the corresponding argument. Even in this case the dimensions can be read from the name of the file if the file name contains the dimensions.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to save.

filename [input]

Data type: string

Name (and path) of file to write.

position [input]

Data type: 3-component integer vector

Position of the image in the target file.

file dimensions [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

Dimensions of the output file. Specify zero to parse dimensions from the file name.

source position [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

Position of the block of the source image to write.

source block size [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

Size of the block to write. Specify zero to write the whole source image.

3.8.223 writesequence

Syntax: `writesequence(input image, filename)`

Write an image sequence to disk.

This command can be used in the distributed processing mode. Use *distribute* command to change processing mode from local to distributed.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to save.

filename [input]

Data type: string

Name (and path) of file to write. Existing files are overwritten. Specify file name as a template where @ signifies the location where the slice number should be added. Use @(n) to specify field width, e.g., if n = 4, slice number 7 would be labeled 0007. Specify @(-) to select suitable field width automatically. Specify empty file name to save using default template '@.png' that corresponds to file names 0.png, 1.png, 2.png etc.

3.8.224 writesequenceblock

Syntax: `writesequenceblock(input image, filename, position, file dimensions, source position, source block size)`

Write an image to a specified position in an image sequence. Optionally can write only a block of the source image.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to save.

filename [input]

Data type: string

Name (and path) of the file to write.

position [input]

Data type: 3-component integer vector

Position of the image in the target file.

file dimensions [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

Dimensions of the output file. Specify zero to parse dimensions from the file. In this case it must exist.

source position [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

Position of the block of the source image to write.

source block size [input]

Data type: 3-component integer vector

Default value: "[0, 0, 0]"

Size of the block to write. Specify zero to write the whole source image.

3.8.225 writetif

Syntax: `writetif(input image, filename)`

Write an image to a .tif file.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

input image [input]

Data type: uint8 image, uint16 image, uint32 image, uint64 image, int8 image, int16 image, int32 image, int64 image, float32 image, complex32 image

Image to save.

filename [input]

Data type: string

Name (and path) of the file to write. If the file exists, its current contents are erased. Extension .tif is automatically appended to the name of the file.

3.8.226 writevtk

There are 3 forms of this command.

`writevtk(points, lines, filename, point data name, point data, line data name, cell data)`

Writes network in points-and-lines format to a .vtk file.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

points [input]

Data type: float32 image

Image that stores coordinates of all points in the network. See *getpointsandlines* command.

lines [input]

Data type: uint64 image

Image that stores a list of point indices for each edge. See *getpointsandlines* command.

filename [input]

Data type: string

Name of file to write. Suffix .vtk will be added to the file name.

point data name [input]

Data type: string

Default value: “”

Comma-separated list of names of data fields for each point.

point data [input]

Data type: float32 image

Image that has one row for each point. The data in columns will be saved in the .vtk file using names given in argument ‘point data name’.

line data name [input]

Data type: string

Default value: “”

Comma-separated list of names of data fields for each line.

cell data [input]

Data type: float32 image

Image that has one row for each line. The data in columns will be saved in the .vtk file using names given in argument ‘line data name’.

See also

getpointsandlines

`writevtk(points, lines, filename)`

Writes network in points-and-lines format to a .vtk file.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

points [input]

Data type: float32 image

Image that stores coordinates of all points in the network. See *getpointsandlines* command.

lines [input]

Data type: uint64 image

Image that stores a list of point indices for each edge. See *getpointsandlines* command.

filename [input]

Data type: string

Name of file to write. Suffix .vtk will be added to the file name.

`writevtk(points, lines, filename, point data name, point data)`

Writes network in points-and-lines format to a .vtk file.

This command cannot be used in the distributed processing mode. If you need it, please contact the authors.

Arguments

points [input]

Data type: float32 image

Image that stores coordinates of all points in the network. See *getpointsandlines* command.

lines [input]

Data type: uint64 image

Image that stores a list of point indices for each edge. See *getpointsandlines* command.

filename [input]

Data type: string

Name of file to write. Suffix .vtk will be added to the file name.

point data name [input]

Data type: string

Default value: “”

Comma-separated list of names of data fields for each point.

point data [input]

Data type: float32 image

Image that has one row for each point. The data in columns will be saved in the .vtk file using names given in argument ‘point data name’.

3.9 Supported file formats

Pi2 supports a limited selection of useful file formats as indicated in the table below. Pi2 can either read full files or small blocks of files, but both operation modes are not supported for all file formats. If you need to read a format not listed here, please contact the authors to check if support for the format could be added.

Type	Reading	Writing	Reading blocks	Writing blocks
.raw	Yes	Yes	Yes	Yes
.tif	Yes	Yes	Yes	Yes
.nrrd	Yes	Yes	No	No
.vol/.longvol	Yes	No	Yes	No
.pcr	Yes	No	Yes	No
.dcm (DICOM)	Yes	No	No	No
NN5	Yes	Yes	Yes	Yes
.tif sequence	Yes	Yes	Yes	Yes
.png sequence	Yes	Yes	Yes	Yes
.jpg sequence	Yes	No	Yes	No
.dcm (DICOM) sequence	Yes	No	Yes	No
.tif (2D)	Yes	Yes	Yes	Yes
.png (2D)	Yes	Yes	Yes	Yes
.jpg (2D)	Yes	No	No	No
.dcm (2D DICOM)	Yes	No	No	No

All the file formats can be read using the generic *read* command. Some of the file formats can be read also using specific `readX` command, where `X` is the file extension.

Files are written using the `writeX` command, where `X` is the file extension. Sequences can be written with the *writesequence* command.

3.9.1 .raw file format

The pi2 .raw files are equal to those used in Fiji/ImageJ. Basically the .raw file contains only the pixel values in such an order that the fastest changing coordinate is the first image dimension. Pi2 assumes that pixels are represented in the native byte order of the host computer. If this is not the case, byte order can be changed after reading the .raw file by using the *swapbyteorder* command.

In addition, pi2 supports parsing dimensions of the .raw files from the file name, as long as the file name is in format `name_WxHxD.raw` or `name_WxH.raw`, where `W`, `H`, and `D` are the width, height, and depth of the image in pixels.

Pi2 uses .raw format extensively in distributed processing mode as multiple compute nodes can easily write to the same .raw file at the same time (but of course not to the same locations).

3.9.2 Memory-mapped .raw files

An advantage of .raw files is that they can be easily memory-mapped. Memory-mapping means that the file on hard disk is correlated one-to-one to RAM that pi2 uses. If the file is larger than available RAM, the operating system manages reading and writing relevant portions of the file to and from RAM.

Memory-mapping enables relatively efficient processing of files larger than RAM, but it should not be confused with *distributed computing possibilities* offered by pi2.

Raw files can be memory-mapped with command *mapraw* that is a (almost) drop-in replacement for the *readraw* command. That is, after you *mapraw* an image, it can be used just like any other image but any changes made to it are immediately reflected to disk. Memory-mapped images do not have to be saved using the *writeraw* command. Any changes are automatically written to the file.

Note that memory-mapping files stored on network shares does not usually work as expected!

3.10 Articles where pi2 has been used

This is an incomplete list of articles where this software or some of its previous versions has been used. If you want to add your paper here, please email arttu.i.miettinen@jyu.fi.

- Truong M. et al., Sub-micrometer morphology of human atherosclerotic plaque revealed by synchrotron radiation-based μCT—A comparison with histology. PLoS ONE 17(4), e0265598. https://doi.org/10.1371/journal.pone.0265598
- Blaskovic A. et al., Early life exposure to nicotine modifies lung gene response after elastase-induced emphysema. Respiratory Research 23, 2022. https://doi.org/10.1186/s12931-022-01956-4
- Romano M., et al., X-Ray Phase Contrast 3D Virtual Histology:Evaluation of Lung Alterations After Microbeam Irradiation. Radiation Oncology, Biology, Physics, 2022. https://doi.org/10.1016/j.ijrobp.2021.10.009
- Eckermann M. et al., Three-dimensional virtual histology of the human hippocampus based on phase-contrast computed tomography, PNAS 118(48), 2021.
- Wälchli T. et al., Hierarchical imaging and computational analysis of three-dimensional vascular network architecture in the entire postnatal and adult mouse brain. Nature Protocols 16, 2021. https://doi.org/10.1038/s41596-021-00587-1
- Bosch C. et al., Functional and multiscale 3D structural investigation of brain tissue through correlative in vivo physiology, synchrotron micro-tomography and volume electron microscopy. Pre-print available at https://www.biorxiv.org/content/10.1101/2021.01.13.426503v1.full.
- Miettinen A. et al., Micrometer-resolution reconstruction and analysis of whole mouse brain vasculature by synchrotron-based phase-contrast tomographic microscopy. In press, preprint available at https://www.biorxiv.org/content/10.1101/2021.03.16.435616v1.abstract.
- Karakoç A. et al., Effective elastic properties of biocomposites using 3D computational homogenization and X-ray microcomputed tomography, Composite Structures 273, 2021.
- Longo E. et al., X-ray Zernike phase contrast tomography: 3D ROI visualization of mm-sized mice organ tissues down to sub-cellular components, Biomedical Optics Express 11(10), 2020.

- Eckermann M. et al., 3d virtual pathohistology of lung tissue from COVID-19 patients based on phase contrast x-ray tomography, eLife 2020;9:e60408, 2020.
- Borisova E. et al., Micrometer-resolution X-ray tomographic full-volume reconstruction of an intact post mortem juvenile rat lung, Histochemistry and Cell Biology, 2020.
- Anschuetz L. et al., Synchrotron radiation imaging revealing the sub-micron structure of the auditory ossicles, Hearing Research 383, 2019.
- Miettinen A. et al., NRStitcher: Non-rigid stitching of terapixel-scale volumetric images, Bioinformatics 35(24), 2019.
- Jairan Nafar Dastgerdi, Behnam Anbarlooie, Arttu Miettinen, Hossein Hosseini-Toudeshky, Heikki Remes. Effects of particle clustering on the plastic deformation and damage initiation of particulate reinforced composite utilizing X-ray CT data and finite element modeling, Composites Part B 153(15), 2018.
- Sayab M. et al., Orthogonal switching of AMS axes during type-2 fold interference: Insights from integrated X-ray computed tomography, AMS and 3D petrography, Journal of Structural Geology 103, 2017.
- Hajlane A. et al., Use of micro-tomography for validation of method to identify interfacial shear strength from tensile tests of short regenerated cellulose fibre composites, IOP Conference Series: Materials Science and Engineering 139, 2016.
- Miettinen A. et al., Non-destructive automatic determination of aspect ratio and cross-sectional properties of fibres, Composites Part A 77, 2015.
- Miettinen A. et al., Identification of true microstructure of composites based on various flax fibre assemblies by means of three-dimensional tomography, 20th International Conference on Composite Materials, Copenhagen, 2015.
- Joffre J. et al., X-ray micro-computed tomography investigation of fibre length degradation during the processing steps of short-fibre composites, Composites Science and Technology 105, 2014.
- Miettinen A. et al., A non-destructive X-ray microtomography approach for measuring fibre length in short-fibre composites, Composites Science and Technology 72(15), 2012.

3.11 Spherical coordinates

Orientation-related commands (e.g. *cylinderorientation*) use spherical coordinates defined in the figure below. Namely, ϕ refers to the azimuthal angle and θ to the polar angle. The azimuthal angle is measured in xy-plane from the positive x-axis towards the positive y-axis, and its values are always in the range $[-\pi, \pi]$. The polar angle is measured from the positive z-axis towards the xy-plane, and its values are always in the range $[0, \pi]$.

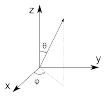

Fig. 47: Spherical coordinate system used in pi2. Here, ϕ is the azimuthal angle and θ is the polar angle.

Conversion from the cartesian coordinates to the polar coordinates is done with

$$
\begin{aligned}
r &= \sqrt{x^2 + y^2 + z^2} \\
\phi &= \arctan2(y, x) \\
\theta &= \arccos(z/r)
\end{aligned}
$$

Conversion from the polar coordinates to the cartesian coordinates is done with

$$x = r\cos(\phi)sin(\theta)$$
$$y = r\sin(\phi)sin(\theta)$$
$$z = r\cos(\theta)$$

Most orientation-related commands return only angles for which the cartesian x-component is positive. This is because the orientations are symmetrical, i.e. directions $-\vec{r}$ and $\vec{r}$ describe the same orientation, and therefore half of the possible directions are redundant. Additionally, in orientation-related commands $r = 1$ most of the time.

3.12 License

The itl2 library and the pi2 program are licensed under the GNU General Public License version 3, shown below. The pi2 system uses third-party components that are licensed under their respective open source or free licenses, also shown below.

```
This program is licensed under GNU General Public License Version 3:

 GNU GENERAL PUBLIC LICENSE
                       Version 3, 29 June 2007

 Copyright (C) 2007 Free Software Foundation, Inc. <https://fsf.org/>
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

                            Preamble

  The GNU General Public License is a free, copyleft license for
software and other kinds of works.

  The licenses for most software and other practical works are designed
to take away your freedom to share and change the works.  By contrast,
the GNU General Public License is intended to guarantee your freedom to
share and change all versions of a program--to make sure it remains free
software for all its users.  We, the Free Software Foundation, use the
GNU General Public License for most of our software; it applies also to
any other work released this way by its authors.  You can apply it to
your programs, too.

  When we speak of free software, we are referring to freedom, not
price.  Our General Public Licenses are designed to make sure that you
have the freedom to distribute copies of free software (and charge for
them if you wish), that you receive source code or can get it if you
want it, that you can change the software or use pieces of it in new
free programs, and that you know you can do these things.

  To protect your rights, we need to prevent others from denying you
these rights or asking you to surrender the rights.  Therefore, you have
certain responsibilities if you distribute copies of the software, or if
you modify it: responsibilities to respect the freedom of others.

  For example, if you distribute copies of such a program, whether
gratis or for a fee, you must pass on to the recipients the same
```

freedoms that you received. You must make sure that they, too, receive
or can get the source code. And you must show them these terms so they
know their rights.

  Developers that use the GNU GPL protect your rights with two steps:
(1) assert copyright on the software, and (2) offer you this License
giving you legal permission to copy, distribute and/or modify it.

  For the developers' and authors' protection, the GPL clearly explains
that there is no warranty for this free software. For both users' and
authors' sake, the GPL requires that modified versions be marked as
changed, so that their problems will not be attributed erroneously to
authors of previous versions.

  Some devices are designed to deny users access to install or run
modified versions of the software inside them, although the manufacturer
can do so. This is fundamentally incompatible with the aim of
protecting users' freedom to change the software. The systematic
pattern of such abuse occurs in the area of products for individuals to
use, which is precisely where it is most unacceptable. Therefore, we
have designed this version of the GPL to prohibit the practice for those
products. If such problems arise substantially in other domains, we
stand ready to extend this provision to those domains in future versions
of the GPL, as needed to protect the freedom of users.

  Finally, every program is threatened constantly by software patents.
States should not allow patents to restrict development and use of
software on general-purpose computers, but in those that do, we wish to
avoid the special danger that patents applied to a free program could
make it effectively proprietary. To prevent this, the GPL assures that
patents cannot be used to render the program non-free.

  The precise terms and conditions for copying, distribution and
modification follow.

                       TERMS AND CONDITIONS

  0. Definitions.

  "This License" refers to version 3 of the GNU General Public License.

  "Copyright" also means copyright-like laws that apply to other kinds of
works, such as semiconductor masks.

  "The Program" refers to any copyrightable work licensed under this
License. Each licensee is addressed as "you". "Licensees" and
"recipients" may be individuals or organizations.

  To "modify" a work means to copy from or adapt all or part of the work
in a fashion requiring copyright permission, other than the making of an
exact copy. The resulting work is called a "modified version" of the
earlier work or a work "based on" the earlier work.

  A "covered work" means either the unmodified Program or a work based

on the Program.

  To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

  To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

  An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

  1. Source Code.

  The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

  A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

  The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

  The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require,

```
such as by intimate data communication or control flow between those
subprograms and other parts of the work.

  The Corresponding Source need not include anything that users
can regenerate automatically from other parts of the Corresponding
Source.

  The Corresponding Source for a work in source code form is that
same work.

  2. Basic Permissions.

  All rights granted under this License are granted for the term of
copyright on the Program, and are irrevocable provided the stated
conditions are met.  This License explicitly affirms your unlimited
permission to run the unmodified Program.  The output from running a
covered work is covered by this License only if the output, given its
content, constitutes a covered work.  This License acknowledges your
rights of fair use or other equivalent, as provided by copyright law.

  You may make, run and propagate covered works that you do not
convey, without conditions so long as your license otherwise remains
in force.  You may convey covered works to others for the sole purpose
of having them make modifications exclusively for you, or provide you
with facilities for running those works, provided that you comply with
the terms of this License in conveying all material for which you do
not control copyright.  Those thus making or running the covered works
for you must do so exclusively on your behalf, under your direction
and control, on terms that prohibit them from making any copies of
your copyrighted material outside their relationship with you.

  Conveying under any other circumstances is permitted solely under
the conditions stated below.  Sublicensing is not allowed; section 10
makes it unnecessary.

  3. Protecting Users' Legal Rights From Anti-Circumvention Law.

  No covered work shall be deemed part of an effective technological
measure under any applicable law fulfilling obligations under article
11 of the WIPO copyright treaty adopted on 20 December 1996, or
similar laws prohibiting or restricting circumvention of such
measures.

  When you convey a covered work, you waive any legal power to forbid
circumvention of technological measures to the extent such circumvention
is effected by exercising rights under this License with respect to
the covered work, and you disclaim any intention to limit operation or
modification of the work as a means of enforcing, against the work's
users, your or third parties' legal rights to forbid circumvention of
technological measures.

  4. Conveying Verbatim Copies.

  You may convey verbatim copies of the Program's source code as you
```

```
receive it, in any medium, provided that you conspicuously and
appropriately publish on each copy an appropriate copyright notice;
keep intact all notices stating that this License and any
non-permissive terms added in accord with section 7 apply to the code;
keep intact all notices of the absence of any warranty; and give all
recipients a copy of this License along with the Program.

  You may charge any price or no price for each copy that you convey,
and you may offer support or warranty protection for a fee.

  5. Conveying Modified Source Versions.

  You may convey a work based on the Program, or the modifications to
produce it from the Program, in the form of source code under the
terms of section 4, provided that you also meet all of these conditions:

    a) The work must carry prominent notices stating that you modified
    it, and giving a relevant date.

    b) The work must carry prominent notices stating that it is
    released under this License and any conditions added under section
    7.  This requirement modifies the requirement in section 4 to
    "keep intact all notices".

    c) You must license the entire work, as a whole, under this
    License to anyone who comes into possession of a copy.  This
    License will therefore apply, along with any applicable section 7
    additional terms, to the whole of the work, and all its parts,
    regardless of how they are packaged.  This License gives no
    permission to license the work in any other way, but it does not
    invalidate such permission if you have separately received it.

    d) If the work has interactive user interfaces, each must display
    Appropriate Legal Notices; however, if the Program has interactive
    interfaces that do not display Appropriate Legal Notices, your
    work need not make them do so.

  A compilation of a covered work with other separate and independent
works, which are not by their nature extensions of the covered work,
and which are not combined with it such as to form a larger program,
in or on a volume of a storage or distribution medium, is called an
"aggregate" if the compilation and its resulting copyright are not
used to limit the access or legal rights of the compilation's users
beyond what the individual works permit.  Inclusion of a covered work
in an aggregate does not cause this License to apply to the other
parts of the aggregate.

  6. Conveying Non-Source Forms.

  You may convey a covered work in object code form under the terms
of sections 4 and 5, provided that you also convey the
machine-readable Corresponding Source under the terms of this License,
in one of these ways:
```

a) Convey the object code in, or embodied in, a physical product
(including a physical distribution medium), accompanied by the
Corresponding Source fixed on a durable physical medium
customarily used for software interchange.

b) Convey the object code in, or embodied in, a physical product
(including a physical distribution medium), accompanied by a
written offer, valid for at least three years and valid for as
long as you offer spare parts or customer support for that product
model, to give anyone who possesses the object code either (1) a
copy of the Corresponding Source for all the software in the
product that is covered by this License, on a durable physical
medium customarily used for software interchange, for a price no
more than your reasonable cost of physically performing this
conveying of source, or (2) access to copy the
Corresponding Source from a network server at no charge.

c) Convey individual copies of the object code with a copy of the
written offer to provide the Corresponding Source. This
alternative is allowed only occasionally and noncommercially, and
only if you received the object code with such an offer, in accord
with subsection 6b.

d) Convey the object code by offering access from a designated
place (gratis or for a charge), and offer equivalent access to the
Corresponding Source in the same way through the same place at no
further charge. You need not require recipients to copy the
Corresponding Source along with the object code. If the place to
copy the object code is a network server, the Corresponding Source
may be on a different server (operated by you or a third party)
that supports equivalent copying facilities, provided you maintain
clear directions next to the object code saying where to find the
Corresponding Source. Regardless of what server hosts the
Corresponding Source, you remain obligated to ensure that it is
available for as long as needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided
you inform other peers where the object code and Corresponding
Source of the work are being offered to the general public at no
charge under subsection 6d.

A separable portion of the object code, whose source code is excluded
from the Corresponding Source as a System Library, need not be
included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any
tangible personal property which is normally used for personal, family,
or household purposes, or (2) anything designed or sold for incorporation
into a dwelling. In determining whether a product is a consumer product,
doubtful cases shall be resolved in favor of coverage. For a particular
product received by a particular user, "normally used" refers to a
typical or common use of that class of product, regardless of the status
of the particular user or of the way in which the particular user
actually uses, or expects or is expected to use, the product. A product

is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

```
  Notwithstanding any other provision of this License, for material you
add to a covered work, you may (if authorized by the copyright holders of
that material) supplement the terms of this License with terms:

    a) Disclaiming warranty or limiting liability differently from the
    terms of sections 15 and 16 of this License; or

    b) Requiring preservation of specified reasonable legal notices or
    author attributions in that material or in the Appropriate Legal
    Notices displayed by works containing it; or

    c) Prohibiting misrepresentation of the origin of that material, or
    requiring that modified versions of such material be marked in
    reasonable ways as different from the original version; or

    d) Limiting the use for publicity purposes of names of licensors or
    authors of the material; or

    e) Declining to grant rights under trademark law for use of some
    trade names, trademarks, or service marks; or

    f) Requiring indemnification of licensors and authors of that
    material by anyone who conveys the material (or modified versions of
    it) with contractual assumptions of liability to the recipient, for
    any liability that these contractual assumptions directly impose on
    those licensors and authors.

  All other non-permissive additional terms are considered "further
restrictions" within the meaning of section 10.  If the Program as you
received it, or any part of it, contains a notice stating that it is
governed by this License along with a term that is a further
restriction, you may remove that term.  If a license document contains
a further restriction but permits relicensing or conveying under this
License, you may add to a covered work material governed by the terms
of that license document, provided that the further restriction does
not survive such relicensing or conveying.

  If you add terms to a covered work in accord with this section, you
must place, in the relevant source files, a statement of the
additional terms that apply to those files, or a notice indicating
where to find the applicable terms.

  Additional terms, permissive or non-permissive, may be stated in the
form of a separately written license, or stated as exceptions;
the above requirements apply either way.

  8. Termination.

  You may not propagate or modify a covered work except as expressly
provided under this License.  Any attempt otherwise to propagate or
modify it is void, and will automatically terminate your rights under
this License (including any patent licenses granted under the third
paragraph of section 11).
```

```
  However, if you cease all violation of this License, then your
license from a particular copyright holder is reinstated (a)
provisionally, unless and until the copyright holder explicitly and
finally terminates your license, and (b) permanently, if the copyright
holder fails to notify you of the violation by some reasonable means
prior to 60 days after the cessation.

  Moreover, your license from a particular copyright holder is
reinstated permanently if the copyright holder notifies you of the
violation by some reasonable means, this is the first time you have
received notice of violation of this License (for any work) from that
copyright holder, and you cure the violation prior to 30 days after
your receipt of the notice.

  Termination of your rights under this section does not terminate the
licenses of parties who have received copies or rights from you under
this License.  If your rights have been terminated and not permanently
reinstated, you do not qualify to receive new licenses for the same
material under section 10.

  9. Acceptance Not Required for Having Copies.

  You are not required to accept this License in order to receive or
run a copy of the Program.  Ancillary propagation of a covered work
occurring solely as a consequence of using peer-to-peer transmission
to receive a copy likewise does not require acceptance.  However,
nothing other than this License grants you permission to propagate or
modify any covered work.  These actions infringe copyright if you do
not accept this License.  Therefore, by modifying or propagating a
covered work, you indicate your acceptance of this License to do so.

  10. Automatic Licensing of Downstream Recipients.

  Each time you convey a covered work, the recipient automatically
receives a license from the original licensors, to run, modify and
propagate that work, subject to this License.  You are not responsible
for enforcing compliance by third parties with this License.

  An "entity transaction" is a transaction transferring control of an
organization, or substantially all assets of one, or subdividing an
organization, or merging organizations.  If propagation of a covered
work results from an entity transaction, each party to that
transaction who receives a copy of the work also receives whatever
licenses to the work the party's predecessor in interest had or could
give under the previous paragraph, plus a right to possession of the
Corresponding Source of the work from the predecessor in interest, if
the predecessor has it or can get it with reasonable efforts.

  You may not impose any further restrictions on the exercise of the
rights granted or affirmed under this License.  For example, you may
not impose a license fee, royalty, or other charge for exercise of
rights granted under this License, and you may not initiate litigation
(including a cross-claim or counterclaim in a lawsuit) alleging that
```

any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within

the scope of its coverage, prohibits the exercise of, or is
conditioned on the non-exercise of one or more of the rights that are
specifically granted under this License. You may not convey a covered
work if you are a party to an arrangement with a third party that is
in the business of distributing software, under which you make payment
to the third party based on the extent of your activity of conveying
the work, and under which the third party grants, to any of the
parties who would receive the covered work from you, a discriminatory
patent license (a) in connection with copies of the covered work
conveyed by you (or copies made from those copies), or (b) primarily
for and in connection with specific products or compilations that
contain the covered work, unless you entered into that arrangement,
or that patent license was granted, prior to 28 March 2007.

  Nothing in this License shall be construed as excluding or limiting
any implied license or other defenses to infringement that may
otherwise be available to you under applicable patent law.

  12. No Surrender of Others' Freedom.

  If conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License. If you cannot convey a
covered work so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you may
not convey it at all. For example, if you agree to terms that obligate you
to collect a royalty for further conveying from those to whom you convey
the Program, the only way you could satisfy both those terms and this
License would be to refrain entirely from conveying the Program.

  13. Use with the GNU Affero General Public License.

  Notwithstanding any other provision of this License, you have
permission to link or combine any covered work with a work licensed
under version 3 of the GNU Affero General Public License into a single
combined work, and to convey the resulting work. The terms of this
License will continue to apply to the part which is the covered work,
but the special requirements of the GNU Affero General Public License,
section 13, concerning interaction through a network will apply to the
combination as such.

  14. Revised Versions of this License.

  The Free Software Foundation may publish revised and/or new versions of
the GNU General Public License from time to time. Such new versions will
be similar in spirit to the present version, but may differ in detail to
address new problems or concerns.

  Each version is given a distinguishing version number. If the
Program specifies that a certain numbered version of the GNU General
Public License "or any later version" applies to it, you have the
option of following the terms and conditions either of that numbered
version or of any later version published by the Free Software
Foundation. If the Program does not specify a version number of the

GNU General Public License, you may choose any version ever published
by the Free Software Foundation.

  If the Program specifies that a proxy can decide which future
versions of the GNU General Public License can be used, that proxy's
public statement of acceptance of a version permanently authorizes you
to choose that version for the Program.

  Later license versions may give you additional or different
permissions. However, no additional obligations are imposed on any
author or copyright holder as a result of your choosing to follow a
later version.

  15. Disclaimer of Warranty.

  THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY
APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT
HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY
OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM
IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF
ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

  16. Limitation of Liability.

  IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING
WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS
THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY
GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE
USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF
DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD
PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS),
EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF
SUCH DAMAGES.

  17. Interpretation of Sections 15 and 16.

  If the disclaimer of warranty and limitation of liability provided
above cannot be given local legal effect according to their terms,
reviewing courts shall apply local law that most closely approximates
an absolute waiver of all civil liability in connection with the
Program, unless a warranty or assumption of liability accompanies a
copy of the Program in return for a fee.

                     END OF TERMS AND CONDITIONS

            How to Apply These Terms to Your New Programs

  If you develop a new program, and you want it to be of the greatest
possible use to the public, the best way to achieve this is to make it
free software which everyone can redistribute and change under these terms.

  To do so, attach the following notices to the program. It is safest

```
to attach them to the start of each source file to most effectively
state the exclusion of warranty; and each file should have at least
the "copyright" line and a pointer to where the full notice is found.

    <one line to give the program's name and a brief idea of what it does.>
    Copyright (C) <year>  <name of author>

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program.  If not, see <https://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

  If the program does terminal interaction, make it output a short
notice like this when it starts in an interactive mode:

    <program>  Copyright (C) <year>  <name of author>
    This program comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
    This is free software, and you are welcome to redistribute it
    under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate
parts of the General Public License.  Of course, your program's commands
might be different; for a GUI interface, you would use an "about box".

  You should also get your employer (if you work as a programmer) or school,
if any, to sign a "copyright disclaimer" for the program, if necessary.
For more information on this, and how to apply and follow the GNU GPL, see
<https://www.gnu.org/licenses/>.

  The GNU General Public License does not permit incorporating your program
into proprietary programs.  If your program is a subroutine library, you
may consider it more useful to permit linking proprietary applications with
the library.  If this is what you want to do, use the GNU Lesser General
Public License instead of this License.  But first, please read
<https://www.gnu.org/licenses/why-not-lgpl.html>.


------------------------------------------------------------------------------
↪-----------

This software is based in part on the work of the Independent JPEG Group.
```

```
------------------------------------------------------------------------------
 ↪-----------

This program contains code from the LZ4 library, licensed
under the license shown below.
See also https://github.com/lz4/lz4

LZ4 Library
Copyright (c) 2011-2020, Yann Collet
All rights reserved.

Redistribution and use in source and binary forms, with or without␣
 ↪modification,
are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this
  list of conditions and the following disclaimer.

* 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 COPYRIGHT HOLDER 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.




------------------------------------------------------------------------------
 ↪-----------

This program contains code from the JSON for Modern C++ library, licensed
under the MIT license available at http://www.opensource.org/licenses/MIT.
See also https://github.com/nlohmann/json

JSON for Modern C++
version 3.10.5
https://github.com/nlohmann/json

Licensed under the MIT License < http ://opensource.org/licenses/MIT>.
SPDX - License - Identifier : MIT
Copyright(c) 2013 - 2022 Niels Lohmann < http ://nlohmann.me>.
```

```
------------------------------------------------------------------------------
↪-----------

This program contains code from PStreams library, licensed under the␣
↪following license:

PStreams - POSIX Process I/O for C++

Copyright (C) 2001 - 2017 Jonathan Wakely
Distributed under the Boost Software License, Version 1.0.

Boost Software License - Version 1.0 - August 17th, 2003

Permission is hereby granted, free of charge, to any person or organization
obtaining a copy of the software and accompanying documentation covered by
this license(the "Software") to use, reproduce, display, distribute,
execute, and transmit the Software, and to prepare derivative works of the
Software, and to permit third - parties to whom the Software is furnished to
do so, all subject to the following :

The copyright notices in the Software and this entire statement, including
the above license grant, this restriction and the following disclaimer,
must be included in all copies of the Software, in whole or in part, and
all derivative works of the Software, unless such copies or derivative
works are solely in the form of machine - executable object code generated by
a source language processor.

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, TITLE AND NON - INFRINGEMENT. IN NO EVENT
SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE
FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE,
ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.




------------------------------------------------------------------------------
↪-----------

This program contains code ported from public-domain reference implementation␣
↪of JAMA : A Java Matrix Package.
The original code is available at https://math.nist.gov/javanumerics/jama/



------------------------------------------------------------------------------
↪-----------

This program contains code ported from a public-domain library TNT : Template␣
↪Numerical Toolkit.
The original code is available at https://math.nist.gov/tnt/
```

```
------------------------------------------------------------------------------
 ↪-----------

This program contains optimized median calculation code by N. Devillard.
The original code is downloadable from http://ndevilla.free.fr/median/ and is␣
 ↪licensed under the following statement:

The following routines have been built from knowledge gathered
around the Web. I am not aware of any copyright problem with
them, so use it as you want.
N. Devillard - 1998




------------------------------------------------------------------------------
 ↪-----------

This program contains 'whereami' library by Gregory Pakosz, licened under the␣
 ↪MIT license:

Copyright Gregory Pakosz

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.




------------------------------------------------------------------------------
 ↪-----------

Windows versions of this program contain code from libtiff library licensed␣
```

```
↪under the libtiff license:

Copyright(c) 1988 - 1997 Sam Leffler
Copyright(c) 1991 - 1997 Silicon Graphics, Inc.

Permission to use, copy, modify, distribute, and sell this software and
its documentation for any purpose is hereby granted without fee, provided
that(i) the above copyright notices and this permission notice appear in
all copies of the software and related documentation, and (ii)the names of
Sam Leffler and Silicon Graphics may not be used in any advertising or
publicity relating to the software without the specific, prior written
permission of Sam Leffler and Silicon Graphics.

THE SOFTWARE IS PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND,
EXPRESS, IMPLIED OR OTHERWISE, INCLUDING WITHOUT LIMITATION, ANY
WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

IN NO EVENT SHALL SAM LEFFLER OR SILICON GRAPHICS BE LIABLE FOR
ANY SPECIAL, INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES OF ANY KIND,
R ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER OR NOT ADVISED OF THE POSSIBILITY OF DAMAGE, AND ON ANY THEORY OF
LIABILITY, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.




-------------------------------------------------------------------------
↪-----------

This program contains code owned by Joachim Kopp, used for some matrix␣
↪operations and licensed
under the terms of the GNU Lesser General Public License(LGPL), see
https://www.gnu.org/licenses/lgpl.txt for details. The original source code␣
↪can be
found at https://www.mpi-hd.mpg.de/personalhomes/globes/3x3/.
The functionality of the code is further described in article
Joachim Kopp - Efficient numerical diagonalization of hermitian 3x3 matrices
Int. J. Mod. Phys. C 19 (2008) 523-548
arXiv.org: physics/0610206



-------------------------------------------------------------------------
↪-----------

This program contains code from https://github.com/ITHare/util with the␣
↪following license:

Copyright(c) 2018, ITHare.com
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met :
```

```
* Redistributions of source code must retain the above copyright notice, this
list of conditions and the following disclaimer.

* 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.

* Neither the name of the copyright holder 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 COPYRIGHT HOLDER 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.



--------------------------------------------------------------------------
↪-----------

This program contains code from http://www.davekoelle.com/files/alphanum.hpp␣
↪licensed under the MIT license:

Released under the MIT License - https://opensource.org/licenses/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.



--------------------------------------------------------------------------
```

```
↪-----------

Windows versions of this program contain code from libpng library licensed␣
↪under the libpng license:

COPYRIGHT NOTICE, DISCLAIMER, and LICENSE:

If you modify libpng you may insert additional notices immediately following
this sentence.

This code is released under the libpng license.

libpng versions 1.0.7, July 1, 2000 through 1.6.34, September 29, 2017 are
Copyright (c) 2000-2002, 2004, 2006-2017 Glenn Randers-Pehrson, are
derived from libpng-1.0.6, and are distributed according to the same
disclaimer and license as libpng-1.0.6 with the following individuals
added to the list of Contributing Authors:

   Simon-Pierre Cadieux
   Eric S. Raymond
   Mans Rullgard
   Cosmin Truta
   Gilles Vollant
   James Yu
   Mandar Sahastrabuddhe
   Google Inc.
   Vadim Barkov

and with the following additions to the disclaimer:

   There is no warranty against interference with your enjoyment of the
   library or against infringement.  There is no warranty that our
   efforts or the library will fulfill any of your particular purposes
   or needs.  This library is provided with all faults, and the entire
   risk of satisfactory quality, performance, accuracy, and effort is with
   the user.

Some files in the "contrib" directory and some configure-generated
files that are distributed with libpng have other copyright owners and
are released under other open source licenses.

libpng versions 0.97, January 1998, through 1.0.6, March 20, 2000, are
Copyright (c) 1998-2000 Glenn Randers-Pehrson, are derived from
libpng-0.96, and are distributed according to the same disclaimer and
license as libpng-0.96, with the following individuals added to the list
of Contributing Authors:

   Tom Lane
   Glenn Randers-Pehrson
   Willem van Schaik

libpng versions 0.89, June 1996, through 0.96, May 1997, are
Copyright (c) 1996-1997 Andreas Dilger, are derived from libpng-0.88,
and are distributed according to the same disclaimer and license as
```

```
libpng-0.88, with the following individuals added to the list of
Contributing Authors:

   John Bowler
   Kevin Bracey
   Sam Bushell
   Magnus Holmgren
   Greg Roelofs
   Tom Tanner

Some files in the "scripts" directory have other copyright owners
but are released under this license.

libpng versions 0.5, May 1995, through 0.88, January 1996, are
Copyright (c) 1995-1996 Guy Eric Schalnat, Group 42, Inc.

For the purposes of this copyright and license, "Contributing Authors"
is defined as the following set of individuals:

   Andreas Dilger
   Dave Martindale
   Guy Eric Schalnat
   Paul Schmidt
   Tim Wegner

The PNG Reference Library is supplied "AS IS".  The Contributing Authors
and Group 42, Inc. disclaim all warranties, expressed or implied,
including, without limitation, the warranties of merchantability and of
fitness for any purpose.  The Contributing Authors and Group 42, Inc.
assume no liability for direct, indirect, incidental, special, exemplary,
or consequential damages, which may result from the use of the PNG
Reference Library, even if advised of the possibility of such damage.

Permission is hereby granted to use, copy, modify, and distribute this
source code, or portions hereof, for any purpose, without fee, subject
to the following restrictions:

  1. The origin of this source code must not be misrepresented.

  2. Altered versions must be plainly marked as such and must not
     be misrepresented as being the original source.

  3. This Copyright notice may not be removed or altered from any
     source or altered source distribution.

The Contributing Authors and Group 42, Inc. specifically permit, without
fee, and encourage the use of this source code as a component to
supporting the PNG file format in commercial products.  If you use this
source code in a product, acknowledgment is not required but would be
appreciated.

END OF COPYRIGHT NOTICE, DISCLAIMER, and LICENSE.

TRADEMARK:
```

```
The name "libpng" has not been registered by the Copyright owner
as a trademark in any jurisdiction.  However, because libpng has
been distributed and maintained world-wide, continually since 1995,
the Copyright owner claims "common-law trademark protection" in any
jurisdiction where common-law trademark is recognized.

OSI CERTIFICATION:

Libpng is OSI Certified Open Source Software.  OSI Certified Open Source is
a certification mark of the Open Source Initiative. OSI has not addressed
the additional disclaimers inserted at version 1.0.7.

EXPORT CONTROL:

The Copyright owner believes that the Export Control Classification
Number (ECCN) for libpng is EAR99, which means not subject to export
controls or International Traffic in Arms Regulations (ITAR) because
it is open source, publicly available software, that does not contain
any encryption software.  See the EAR, paragraphs 734.3(b)(3) and
734.7(b).

Glenn Randers-Pehrson
glennrp at users.sourceforge.net
September 29, 2017



------------------------------------------------------------------------------
 ↪-----------

Windows versions of this program contain code from zlib library licensed with␣
 ↪the following notice:

(C)1995 - 2017 Jean - loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty. In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions :

1. The origin of this software must not be misrepresented; you must not
claim that you wrote the original software.If you use this software
in a product, an acknowledgment in the product documentation would be
appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be
misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

```
If you use the zlib library in a product, we would appreciate *not* receiving
lengthy legal documents to sign. The sources are provided for free but without
warranty of any kind. The library has been entirely written by Jean-loup
Gailly and Mark Adler; it does not include third-party code.

If you redistribute modified sources, we would appreciate that you include in
the file ChangeLog history information documenting your changes. Please read
the FAQ for more information on the distribution of modified source versions.
```

```
------------------------------------------------------------------------------
↪-----------
```

```
Windows versions of this program are shipped with FFTW3 library licensed
under the GNU General Public License Version 2.
Please see https://www.gnu.org/licenses/gpl-2.0.html for full license text.
```

3.13 About

pi2 = Process Image 2

Contact: arttu.i.miettinen@jyu.fi

Based on work done at

- Complex materials research group, Department of Physics, University of Jyvaskyla, Finland
- X-ray tomography research group, TOMCAT beamline, Swiss Light Source (SLS), Paul Scherrer Institute (PSI), Switzerland
- Institute for Biomedical Engineering, Swiss Federal Institute of Technology Zurich (ETH Zurich), Switzerland
- Centre d'Imagerie BioMedicale (CIBM), Ecole Polytechnique Federale de Lausanne (EPFL), Switzerland