mahotas Documentation

Release 1.0

Luis Pedro Coelho

December 16, 2013

Contents

Mahotas is a computer vision and image processing library for Python.

It includes many algorithms implemented in C++ for speed while operating in numpy arrays and with a very clean Python interface.

Notable algorithms:

- watershed.
- convex points calculations.
- hit & miss. thinning.
- Zernike & Haralick, LBP, and TAS features.
- freeimage based numpy image loading (requires freeimage libraries to be installed).
- Speeded-Up Robust Features (SURF), a form of local features.
- thresholding.
- convolution.
- Sobel edge detection.

Mahotas currently has over 100 functions for image processing and computer vision and it keeps growing.

The release schedule is roughly one release a month and each release brings new functionality and improved performance. The interface is very stable, though, and code written using a version of mahotas from years back will work just fine in the current version, except it will be faster (some interfaces are deprecated and will be removed after a few years, but in the meanwhile, you only get a warning). In a few unfortunate cases, there was a bug in the old code and your results will change for the better.

There is a manuscript about mahotas, which is forthcoming in the Journal of Open Research Software. Full citation (for the time being) is:

> Mahotas: Open source software for scriptable computer vision by Luis Pedro Coelho in Journal of Open Research Software (forthcoming).

CHAPTER 1

Examples

This is a simple example of loading a file (called *test.jpeg*) and calling *watershed* using above threshold regions as a seed (we use Otsu to define threshold).

```
import numpy as np
import mahotas
import pylab

img = mahotas.imread('test.jpeg')
T_otsu = mahotas.thresholding.otsu(img)
seeds,_ = mahotas.label(img > T_otsu)
labeled = mahotas.cwatershed(img.max() - img, seeds)

pylab.imshow(labeled)
```

Computing a distance transform is easy too:

```
import pylab as p
import numpy as np
import mahotas

f = np.ones((256,256), bool)
f[200:,240:] = False
f[128:144,32:48] = False
# f is basically True with the exception of two islands: one in the lower-right
# corner, another, middle-left

dmap = mahotas.distance(f)
p.imshow(dmap)
p.show()
```

CHAPTER 2

Full Documentation Contents

Jump to detailed API Documentation

2.1 How To Install Mahotas

2.1.1 From source

You can get the released version using your favorite Python package manager:

```
easy_install mahotas
```

or:

```
pip install mahotas
```

If you prefer, you can download the source from PyPI and run:

```
python setup.py install
```

You will need to have `numpy` and a `C++` compiler.

Bleeding Edge (Development)

Development happens on github. You can get the development source there. Watch out that *these versions are more likely to have problems*.

2.1.2 On Windows

On Windows, Christoph Gohlke does an excelent job maintaining binary packages of mahotas (and several other packages).

2.1.3 Packaged Versions

Python(x, y)

If you use Python(x, y), which is often a good solution, you can find mahotas in the Additional Plugins page.

FreeBSD

Mahotas is available for FreeBSD as graphics/mahotas.

MacPorts

For Macports, mahotas is available as py27-mahotas.

Frugalware Linux

Mahotas is available as `python-mahotas`.

2.2 Finding Wally

This was originally an answer on stackoverflow We can use it as a simple tutorial example.

The problem is to find Wally (who goes by Waldo in the US) in the following image:

```
from pylab import imshow, show
import mahotas
wally = mahotas.imread('../../mahotas/demos/data/DepartmentStore.jpg')
imshow(wally)
show()
```

Can you see him?

```
wfloat = wally.astype(float)
r,g,b = wfloat.transpose((2,0,1))
```

Split into red, green, and blue channels. It's better to use floating point arithmetic below, so we convert at the top.

```
w = wfloat.mean(2)
```

w is the white channel.

```
pattern = np.ones((24,16), float)
for i in xrange(2):
    pattern[i::4] = -1
```

Build up a pattern of +1,+1,-1,-1 on the vertical axis. This is Wally's shirt.

```
v = mahotas.convolve(r-w, pattern)
```

Convolve with red minus white. This will give a strong response where the shirt is.

```
mask = (v == v.max())
mask = mahotas.dilate(mask, np.ones((48,24)))
```

Look for the maximum value and dilate it to make it visible. Now, we tone down the whole image, except the region or interest:

```
wally -= .8*wally * ~mask[:,:,None]
```

And we get the following:

```
wfloat = wally.astype(float)
r,g,b = wfloat.transpose((2,0,1))
w = wfloat.mean(2)
pattern = np.ones((24,16), float)
for i in xrange(2):
    pattern[i::4] = -1
v = mahotas.convolve(r-w, pattern)
mask = (v == v.max())
mask = mahotas.dilate(mask, np.ones((48,24)))
wally -= .8*wally * ~mask[:,:,None]
imshow(wally)
show()
```

2.3 Labeled Image Functions

Labeled images are integer images where the values correspond to different regions. I.e., region 1 is all of the pixels which have value *1*, region two is the pixels with value 2, and so on. By convention, **region 0 is the background and often handled differently**.

2.3.1 Labeling Images

New in version 0.6.5. The first step is obtaining a labeled function from a binary function:

```
import mahotas as mh
import numpy as np
from pylab import imshow, show

regions = np.zeros((8,8), bool)

regions[:3,:3] = 1
regions[6:,6:] = 1
labeled, nr_objects = mh.label(regions)

imshow(labeled, interpolation='nearest')
show()
```

This results in an image with 3 values:

0. background, where the original image was 0
1. for the first region: (0:3, 0:3);
2. for the second region: (6:, 6:).

There is an extra argument to `label`: the structuring element, which defaults to a 3x3 cross (or, 4-neighbourhood). This defines what it means for two pixels to be in the same region. You can use 8-neighbourhoods by replacing it with a square:

```
labeled,nr_objects = mh.label(regions, np.ones((3,3), bool))
```

New in version 0.7: `labeled_size` and `labeled_sum` were added in version 0.7 We can now collect a few statistics on the labeled regions. For example, how big are they?

```
sizes = mh.labeled.labeled_size(labeled)
print 'Background size', sizes[0]
print 'Size of first region', sizes[1]
```

This size is measured simply as the number of pixels in each region. We can instead measure the total weight in each area:

```
array = np.random.random_sample(regions.shape)
sums = mh.labeled_sum(array, labeled)
print 'Sum of first region', sums[1]
```

2.3.2 Filtering Regions

New in version 0.9.6: remove_regions & relabel were only added in version 0.9.6 Here is a slightly more complex example. This is in `demos` directory as `nuclear.py`. We are going to use this image, a fluorescent microscopy image from a nuclear segmentation benchmark

First we perform a bit of Gaussian filtering and thresholding:

```
f = mh.gaussian_filter(f, 4)
f = (f> f.mean())
```

(Without the Gaussian filter, the resulting thresholded image has very noisy edges. You can get the image in the `demos/` directory and try it out.)

Labeling gets us all of the nuclei:

```
labeled, n_nucleus  = mh.label(f)
print('Found {} nuclei.'.format(n_nucleus))
```

`42` nuclei were found. None were missed, but, unfortunately, we also get some aggregates. In this case, we are going to assume that we wanted to perform some measurements on the real nuclei, but are willing to filter out anything that is not a complete nucleus or that is a lump on nuclei. So we measure sizes and filter:

```
sizes = mh.labeled.labeled_size(labeled)
too_big = np.where(sizes > 10000)
labeled = mh.labeled.remove_regions(labeled, too_big)
```

We can also remove the region touching the border:

```
labeled = mh.labeled.remove_bordering(labeled)
```

This array, `labeled` now has values in the range `0` to `n_nucleus`, but with some values missing (e.g., if region `7` was one of the ones touching the border, then `7` is not used in the labeling). We can `relabel` to get a cleaner version:

```
relabeled, n_left = mh.labeled.relabel(labeled)
print('After filtering and relabeling, there are {} nuclei left.'.format(n_left))
```

Now, we have `24` nuclei and `relabeled` goes from `0` (background) to `24`.

2.3.3 Borders

A border pixel is one where there is more than one region in its neighbourhood (one of those regions can be the background).

You can retrieve border pixels with either the `borders()` function, which gets all the borders or the `border()` (note the singular) which gets only the border between a single pair of regions. As usual, what neighbour means is defined by a structuring element, defaulting to a 3x3 cross.

2.3.4 API Documentation

The `mahotas.labeled` submodule contains the functions mentioned above. `label()` is also available as `mahotas.label`.

`mahotas.labeled.`**`borders`**(*labeled*, *Bc={3x3 cross}*, *out={np.zeros(labeled.shape, bool)}*)

Compute border pixels

A pixel is on a border if it has value *i* and a pixel in its neighbourhood (defined by *Bc*) has value *j*, with `i != j`.

Parameters **labeled** : ndarray of integer type

input labeled array

Bc : structure element, optional

out : ndarray of same shape as *labeled*, dtype=bool, optional

where to store the output. If `None`, a new array is allocated

mode : {'reflect', 'nearest', 'wrap', 'mirror', 'constant' [default], 'ignore'}

How to handle borders

Returns **border_img** : boolean ndarray

Pixels are True exactly where there is a border in *labeled*

`mahotas.labeled.`**`border`**(*labeled*, *i*, *j*, *Bc={3x3 cross}*, *out={np.zeros(labeled.shape, bool)}*, *always_return=True*)

Compute the border region between *i* and *j* regions.

A pixel is on the border if it has value *i* (or *j*) and a pixel in its neighbourhood (defined by *Bc*) has value *j* (or *i*).

Parameters **labeled** : ndarray of integer type

input labeled array

i : integer

j : integer

Bc : structure element, optional

out : ndarray of same shape as *labeled*, dtype=bool, optional

where to store the output. If `None`, a new array is allocated

always_return : bool, optional

if false, then, in the case where there is no pixel on the border, returns `None`. Otherwise (the default), it always returns an array even if it is empty.

Returns **border_img** : boolean ndarray

Pixels are True exactly where there is a border between *i* and *j* in *labeled*

`mahotas.labeled.`**`bwperim`**(*bw*, *n=4*)

Find the perimeter of objects in binary images.

A pixel is part of an object perimeter if its value is one and there is at least one zero-valued pixel in its neighborhood.

By default the neighborhood of a pixel is 4 nearest pixels, but if *n* is set to 8 the 8 nearest pixels will be considered.

Parameters **bw** : ndarray

A black-and-white image (any other image will be converted to black & white)

n : int, optional

Connectivity. Must be 4 or 8 (default: 4)

mode : {'reflect', 'nearest', 'wrap', 'mirror', 'constant' [default], 'ignore'}

How to handle borders

Returns **perim** : ndarray

A boolean image

See Also:

`borders` function This is a more generic function

`mahotas.labeled.`**`label`**(*array*, *Bc={3x3 cross}*, *output={new array}*)

Label the array, which is interpreted as a binary array

This is also called *connected component labeled*, where the connectivity is defined by the structuring element `Bc`.

See: http://en.wikipedia.org/wiki/Connected-component_labeling

Parameters **array** : ndarray

This will be interpreted as binary array

Bc : ndarray, optional

This is the structuring element to use

out : ndarray, optional

Output array. Must be a C-array, of type np.int32

Returns **labeled** : ndarray

Labeled result

nr_objects : int

Number of objects

`mahotas.labeled.`**`labeled_sum`**(*array*, *labeled*)

Labeled sum. sum will be an array of size `labeled.max() + 1`, where `sum[i]` is equal to `np.sum(array[labeled == i])`.

Parameters **array** : ndarray of any type

labeled : int ndarray

Label map. This is the same type as returned from `mahotas.label()`

Returns **sums** : 1-d ndarray of `array.dtype` `mahotas.labeled.`**`labeled_max`**`(`*array*, *labeled*`)`

Labeled minimum. `mins` will be an array of size `labeled.max() + 1`, where `mins[i]` is equal to `np.min(array[labeled == i])`.

Parameters **array** : ndarray of any type

labeled : int ndarray

Label map. This is the same type as returned from `mahotas.label()`

Returns **mins** : 1-d ndarray of `array.dtype`

`mahotas.labeled.`**`labeled_size`**`(`*labeled*`)`

Equivalent to:

```
for i in range(...):
    sizes[i] = np.sum(labeled == i)
```

but, naturally, much faster.

Parameters **labeled** : int ndarray

Returns **sizes** : 1-d ndarray of int

See Also:

`mahotas.fullhistogram` almost same function by another name (the only

`difference`

`mahotas.labeled.`**`relabel`**`(`*labeled*, *inplace=False*`)`

Relabeling ensures that `relabeled` is a labeled image such that every label from 1 to `relabeled.max()` is used (0 is reserved for the background and is passed through).

Example:

```
labeled,n = label(some_binary_map)
for region in xrange(n):
    if not good_region(labeled, region + 1):
        # This deletes the region:
        labeled[labeled == (region + 1)] = 0
relabel(labeled, inplace=True)
```

Parameters **relabeled** : ndarray of int

A labeled array

inplace : boolean, optional

Whether to perform relabeling inplace, erasing the values in `labeled` (default: False)

Returns **relabeled: ndarray** :

nr_objs : int

Number of objects

See Also:

`label` function

`mahotas.labeled.`**`is_same_labeling`**(*labeled0*, *labeled1*)

Checks whether `labeled0` and `labeled1` represent the same labeling (i.e., whether they are the same except for a possible change of label values).

Note that the background (value 0) is treated differently. Namely

is_same_labeling(a, b) implies np.all((a == 0) == (b == 0))

Parameters **labeled0** : ndarray of int

A labeled array

labeled1 : ndarray of int

A labeled array

Returns **same** : bool

Number of objects

See Also:

`label` function

`relabel` function

`mahotas.labeled.`**`remove_bordering`**(*labeled*, *rsize=1*, *out={np.empty_like(im)}*)

Remove objects that are touching the border.

Pass `im` as `out` to achieve in-place operation.

Parameters **labeled** : ndarray

Labeled array

rsize : int, optional

Minimum distance to the border (in Manhatan distance) to allow an object to survive.

out : ndarray, optional

If `im` is passed as `out`, then it operates inline.

Returns **slabeled** : ndarray

Subset of `labeled`

`mahotas.labeled.`**`remove_regions`**(*labeled*, *regions*, *inplace=False*)

removed = remove_regions(labeled, regions, inplace=False):

Removes the regions in `regions`. If an elementwise `in` operator existed, this would be equivalent to the following:

```
labeled[ labeled element-wise-in regions ] = 0
```

This function **does not** relabel its arguments. You can use the `relabel` function for that:

```
removed = relabel(remove_regions(labeled, regions))
```

Or, saving one image allocation:

```
removed = relabel(remove_regions(labeled, regions), inplace=True)
```

This is the same, but reuses the memory in the relabeling operation.

Parameters **relabeled** : ndarray of int

A labeled array

regions : sequence of int

These regions will be removed

inplace : boolean, optional

Whether to perform removal inplace, erasing the values in `labeled` (default: False)

Returns **removed** : ndarray

See Also:

`relabel` function After removing unecessary regions, it is often a good idea to relabel your label image.

2.4 Thresholding

The example in this section is present in the source under `mahotas/demos/thresholding.py`.

We start with an image, a grey-scale image:

```
luispedro_image = '../../mahotas/demos/data/luispedro.jpg'
photo = mahotas.imread(luispedro_image, as_grey=True)
photo = photo.astype(np.uint8)
```

The reason we convert to `np.uint8` is because `as_grey` returns floating point images (there are good reasons for this and good reasons against it, since it's easier to truncate than to go back, it returns `np.uint8`).

Thresholding functions have a trivial interface: they take an image and return a value. One of the most well-known thresholding methods is Otsu's method:

```
T_otsu = mahotas.otsu(photo)
print(T_otsu)
imshow(photo > T_otsu)
show()
```

prints `115`.

An alternative is the Riddler-Calvard method:

```
T_rc = mahotas.rc(photo)
print(T_rc)
imshow(photo > T_rc)
show()
```

In this image, it prints almost the same as Otsu: `115.68`. The thresholded image is exactly the same:

2.4.1 API Documentation

The `mahotas.thresholding` module contains the thresholding functions, but they are also available in the main `mahotas` namespace.

Thresholding Module

Thresholding functions:

otsu() Otsu method

rc() Riddler-Calvard's method

`mahotas.thresholding.`**`otsu`**(*img*, *ignore_zeros=False*)

Calculate a threshold according to the Otsu method.

Parameters **img** : an image as a numpy array.

This should be of an unsigned integer type.

ignore_zeros : Boolean

whether to ignore zero-valued pixels (default: False)

Returns **T** : integer

the threshold

`mahotas.thresholding.`**`rc`**(*img*, *ignore_zeros=False*)

Calculate a threshold according to the Riddler-Calvard method.

Parameters **img** : ndarray

Image of any type

ignore_zeros : boolean, optional

Whether to ignore zero valued pixels (default: False)

Returns **T** : float

threshold

`mahotas.thresholding.`**`soft_threshold`**(*f*, *tval*)

Soft threshold function

^ / | / | / | / | /

- – * ·

  ·

  ·

  ·

  ·

  ·

  ·

  ·

  ·

  ·

  ·

  ·

  ·

  ·->

  / |

  / |

  / |

/ |

/ |

/ |

Parameters **f** : ndarray

tval : scalar

Returns **thresholded** : ndarray

`mahotas.thresholding.bernsen(f, radius, contrast_threshold, gthresh={128})`

Bernsen local thresholding

Parameters **f** : ndarray
input image

radius : integer
radius of circle (to consider "local")

contrast_threshold : integer
contrast threshold

gthresh : numeric, optional
global threshold to fall back in low contrast regions

Returns **thresholded** : binary ndarray

See Also:

`gbernsen` function Generalised Bernsen thresholding

2.5 Wavelet Transforms

New in version 0.9.1: Wavelet functions were only added in version 0.9.1 We are going to use wavelets to transform an image so that most of its values are 0 (and otherwise small), but most of the signal is preserved.

The code for this tutorial is avalailable from the source distribution as `mahotas/demos/wavelet_compression.py`.

We start by importing and loading our input image

```
import numpy as np
import mahotas
from mahotas.thresholding import soft_threshold
from matplotlib import pyplot as plt
from os import path
luispedro_image = '../../mahotas/demos/data/luispedro.jpg'
f = mahotas.imread(luispedro_image, as_grey=True)
f = f[:256,:256]
plt.gray()
# Show the data:
print("Fraction of zeros in original image: {0}".format(np.mean(f==0)))
plt.imshow(f)
plt.show()
```

There are no zeros in the original image. We now try a baseline compression method: save every other pixel and only high-order bits.

```
direct = f[::2,::2].copy()
direct /= 8
direct = direct.astype(np.uint8)
```

```
print("Fraction of zeros in original image (after division by 8): {0}".format(np.mean(direct==0)))
plt.imshow(direct)
plt.show()
```

There are only a few zeros, though. We have, however, thrown away 75% of the values. Can we get a better image, using the same number of values, though?

We will transform the image using a Daubechies wavelet (D8) and then discard the high-order bits.

```
# Transform using D8 Wavelet to obtain transformed image t:
t = mahotas.daubechies(f,'D8')

# Discard low-order bits:
t /= 8
t = t.astype(np.int8)
print("Fraction of zeros in transform (after division by 8): {0}".format(np.mean(t==0)))
plt.imshow(t)
plt.show()
```

This has 60% zeros! What does the reconstructed image look like?

```
# Let us look at what this looks like
r = mahotas.idaubechies(t, 'D8')
plt.imshow(r)
plt.show()
```

This is a pretty good reduction without much quality loss. We can go further and discard small values in the transformed space. Also, let's make the remaining values even smaller in magnitude.

Now, this will be 77% of zeros, with the remaining being small values. This image would compress very well as a lossless image and we could reconstruct the full image after transmission. The quality is certainly higher than just keeping every fourth pixel and low-order bits.

```
tt = soft_threshold(t, 12)
print("Fraction of zeros in transform (after division by 8 & soft thresholding): {0}".format(np.mean
# Let us look again at what we have:
rt = mahotas.idaubechies(tt, 'D8')
plt.imshow(rt)
```

2.5.1 What About the Borders?

In this example, we can see some artifacts at the border. We can use `wavelet_center` and `wavelet_decenter` to handle borders to correctly:

```
fc = mahotas.wavelet_center(f)
t = mahotas.daubechies(fc, 'D8')
r = mahotas.idaubechies(fc, 'D8')
rd = mahotas.wavelet_decenter(r, fc.shape)
```

Now, `rd` is equal (except for rounding) to `fc` **without any border effects**.

2.5.2 API Documentation

A package for computer vision in Python.

Main Features

features Compute global and local features (several submodules, include SURF and Haralick features)

convolve Convolution and wavelets

morph Morphological features. Most are available at the mahotas level, include erode(), dilate()...

watershed Seeded watershed implementation

imread/imsave read/write image

Citation:

> Coelho, Luis Pedro, 2013. Mahotas: Open source software for scriptable computer vision. Journal of Open Research Software, 1:e3, DOI: http://dx.doi.org/10.5334/jors.ac

`mahotas.`**`haar`**(*f*, *preserve_energy=True*, *inline=False*)

Haar transform

Parameters **f** : 2-D ndarray

Input image

preserve_energy : bool, optional

Whether to normalise the result so that energy is preserved (the default).

inline : bool, optional

Whether to write the results to the input image. By default, a new image is returned. Integer images are always converted to floating point and copied.

See Also:

`ihaar` function Reverse Haar transform

`mahotas.`**`ihaar`**(*f*, *preserve_energy=True*, *inline=False*)

Reverse Haar transform

`ihaar(haar(f))` is more or less equal to `f` (equal, except for possible rounding issues).

Parameters **f** : 2-D ndarray

Input image. If it is an integer image, it is converted to floating point (double).

preserve_energy : bool, optional

Whether to normalise the result so that energy is preserved (the default).

inline : bool, optional

Whether to write the results to the input image. By default, a new image is returned. Integer images are always converted to floating point and copied.

Returns **f** : ndarray

See Also:

`haar` function Forward Haar transform

`mahotas.`**`daubechies`**(*f*, *code*, *inline=False*)

Daubechies wavelet transform

This function works best if the image sizes are powers of 2!

Parameters **f** : ndarray

2-D image

code : str

One of ‘D2’, ‘D4’, ... ‘D20’

inline : bool, optional

Whether to write the results to the input image. By default, a new image is returned. Integer images are always converted to floating point and copied.

See Also:

`haar` function Haar transform (equivalent to D2) `mahotas.`**`idaubechies`**(*f*, *code*, *inline=False*)

Daubechies wavelet inverse transform

Parameters **f** : ndarray

2-D image

code : str

One of 'D2', 'D4', ... 'D20'

inline : bool, optional

Whether to write the results to the input image. By default, a new image is returned. Integer images are always converted to floating point and copied.

See Also:

`haar` function Haar transform (equivalent to D2)

2.6 Distance Transform

The example in this section is present in the source under `mahotas/demos/distance.py`.

We start with an image, a black&white image that is mostly black except for two white spots:

```
import numpy as np
import mahotas

f = np.ones((256,256), bool)
f[200:,240:] = False
f[128:144,32:48] = False
```

There is a simple `distance()` function which computes the distance map:

```
import mahotas
dmap = mahotas.distance(f)
```

Now `dmap[y,x]` contains the squared euclidean distance of the pixel *(y,x)* to the nearest black pixel in `f`. If `f[y,x] == True`, then `dmap[y,x] == 0`.

```
from __future__ import print_function

import pylab as p
import numpy as np
import mahotas

f = np.ones((256,256), bool)
f[200:,240:] = False
f[128:144,32:48] = False
# f is basically True with the exception of two islands: one in the lower-right
# corner, another, middle-left

dmap = mahotas.distance(f)
p.imshow(dmap)
p.show()
```

2.6.1 Distance Transform and Watershed

The distance transform is often combined with the watershed for segmentation. Here is an example (which is available with the source in the `mahotas/demos/` directory as `nuclear_distance_watershed.py`).

The code is not very complex. Start by loading the image and preprocessing it with a Gaussian blur:

```
import mahotas
nuclear = mahotas.imread('mahotas/demos/data/nuclear.png')
nuclear = nuclear[:,:,0]
nuclear = mahotas.gaussian_filter(nuclear, 1.)
threshed  = (nuclear > nuclear.mean())
```

Now, we compute the distance transform:

```
distances = mahotas.stretch(mahotas.distance(threshed))
```

We find and label the regional maxima:

```
Bc = np.ones((9,9))
maxima = mahotas.morph.regmax(distances, Bc=Bc)
spots,n_spots = mahotas.label(maxima, Bc=Bc)
```

Finally, to obtain the image above, we invert the distance transform (because of the way that `cwatershed` is defined) and compute the watershed:

```
surface = (distances.max() - distances)
areas = mahotas.cwatershed(surface, spots)
areas *= threshed
```

We used a random colormap with a black background for the final image. This is achieved by:

```
import random
from matplotlib import colors as c
colors = map(cm.jet,range(0, 256, 4))
random.shuffle(colors)
colors[0] = (0.,0.,0.,1.)
rmap = c.ListedColormap(colors)
imshow(areas, cmap=rmap)
show()
```

2.6.2 API Documentation

A package for computer vision in Python.

Main Features

features Compute global and local features (several submodules, include SURF and Haralick features)

convolve Convolution and wavelets

morph Morphological features. Most are available at the mahotas level, include erode(), dilate()...

watershed Seeded watershed implementation

imread/imsave read/write image

Citation:

> Coelho, Luis Pedro, 2013. Mahotas: Open source software for scriptable computer vision. Journal of Open Research Software, 1:e3, DOI: http://dx.doi.org/10.5334/jors.ac

`mahotas.`**`distance`**(*bw*, *metric='euclidean2'*)

Computes the distance transform of image *bw*:

```
dmap[i,j] = min_{i', j'} { (i-i')**2 + (j-j')**2 | !bw[i', j'] }
```

That is, at each point, compute the distance to the background.

If there is no background, then a very high value will be returned in all pixels (this is a sort of infinity).

Parameters **bw** : 2d-ndarray

If boolean, `False` will denote the background and `True` the foreground. If not boolean, this will be interpreted as `bw != 0` (this way you can use labeled images without any problems).

metric : str, optional

one of 'euclidean2' (default) or 'euclidean'

Returns **dmap** : ndarray

distance map

2.7 Polygon Utitilities

2.7.1 Drawing

Mahotas is not a package to generate images, but there are a few simple functions to draw lines and polygons on an image (the target image is known as the *canvas* in this documentation).

The simplest function is `line`: Give it two points and it draws a line between them. The implementation is simple, and in Python, so it will be slow for many complex usage.

The main purpose of these utilities is to aid debugging and visualisation. If you need to generate fancy graphs, look for packages such as matplotlib.

2.7.2 Convex Hull

Convex hull functions are a more typical image processing feature. Mahotas has a simple one, called `convexhull`. Given a boolean image (or anything that will get interpreted as a boolean image), it finds the convex hull of all its on points.

The implementation is in C++, so it is fast.

A companion function `fill_convexhull` returns the convex hull as a binary image.

2.7.3 API Documentation

`mahotas.polygon.`**`line`**(*(y0, x0)*, *(y1, x1)*, *canvas*, *color=1*)

Draw a line

Parameters **p0** : pair of integers

first point

p1 : pair of integers

second point

canvas : ndarray

where to draw, will be modified in place

color : integer, optional

which value to store on the pixels (default: 1) `mahotas.polygon.`**`fill_polygon`**`(`[*(y0, x0), (y1, x1), ...*], *canvas*, *color=1*`)`

Draw a filled polygon in canvas

Parameters **polygon** : list of pairs

a list of (y,x) points

canvas : ndarray

where to draw, will be modified in place

color : integer, optional

which colour to use (default: 1)

`mahotas.polygon.`**`convexhull`**`(`*bwimg*`)`

Compute the convex hull as a polygon

Parameters **bwimg** : ndarray

input image (interpreted as boolean). Only 2D arrays are supported.

Returns **hull** : ndarray

Set of (y,x) coordinates of hull corners

`mahotas.polygon.`**`fill_convexhull`**`(`*bwimg*`)`

Compute the convex hull and return it as a binary mask

Parameters **bwimage** : input image (interpreted as boolean)

Returns **hull** : image of same size and dtype as *bwimg* with the hull filled in.

2.8 Features

By features we mean, basically, numerical functions of the image. That is, any method that gives me a number from the image, I can call it a *feature*. Ideally, these should be meaningful.

We can classify features into two types:

global These are a function of the whole image.

local These **have a position** and are a function of a local image region.

Mahotas supports both types.

The classification tutorial illustrates the usefulness of feature computation.

2.8.1 Global features

Haralick features

These are texture features, based on the adjancency matrix (the adjacency matrix stores in position *(i,j)* the number of times that a pixel takes the value *i* **next to** a pixel with the value *j*. Given different ways to define **next to**, you obtain slightly different variations of the features. Standard practice is to average them out across the directions to get some rotational invariance.

They can be computed for 2-D or 3-D images and are available in the `mahotas.features.haralick` module.

Only the first 13 features are implemented. The last (14th) feature is normally considered to be *unstable*, although it is not clear to me why this is. (See this unanswered question on Cross-validated).

Local Binary Patterns

Local binary patterns (LBP) are a more recent set of features. Each pixel is looked at individually. Its neighbourhood is analysed and summarised by a single numeric code. The normalised histogram across all the pixels in the image is the final set of features.

Again, this is an attempt at capturing texture. LBPs are insensitive to orientation and to illumination (scaling).

Threshold Adjancency Statistics

Threshold adjancency statistics (TAS) are a recent innovation too. In the original version, they have fixed parameters, but we have adapted them to *parameter-free* versions (see Structured Literature Image Finder: Extracting Information from Text and Images in Biomedical Literature by Coelho et al. for a reference). Mahotas supports both.

Zernike Moments

Zernike moments are **not** a texture feature, but rather a global measure of how the mass is distributed.

2.8.2 Local features

SURF: Speeded-Up Robust Features

Speeded-Up Robust Features (SURF) have both a *location* (pixel coordinates) and a *scale* (natural size) as well as a descriptor (the local features).

Read more about SURF.

2.9 Local Binary Patterns

New in version 0.7: LBPs are available before, but an important bug was fixed in 0.7. It is **highly recommended that you never use the older version**. Local binary patterns depend on the local region around each pixel. See the diagram below:

(Image reference: Wikipedia)

The reference pixel is in red, at the centre. A number of points are defined at a distance `r` from it. These are the green points. As you go from left to right, the number of green points increases.

The “pattern” in the name is the relationship of the value at the green points when compared to the central red point. We call it a binary pattern because all that is taken into account is whether the value at the green point is greater than the value at the red point.

As you can see, the green points do not necessarily fall exactly on another pixel, so we need to use interpolation to find a value for the green points.

2.9.1 API Documentation

The `mahotas.features.lb` module contains the `lbp` function which implements LBPs.

`mahotas.features.lbp.`**`lbp`**`(`*image*, *radius*, *points*, *ignore_zeros=False*`)`

Compute Linear Binary Patterns

The return value is a **histogram** of feature counts, where position `i` corresponds to the number of pixels that had code `i`. The codes are compressed so that impossible codes are not used. Therefore, this is the `i``th feature, not just the feature with binary code ``i`.

Parameters **image** : ndarray

input image (2-D numpy ndarray)

radius : number (integer or floating point) radius (in pixels)

points : integer
nr of points to consider

ignore_zeros : boolean, optional
whether to ignore zeros (default: False)

Returns **features** : 1-D numpy ndarray
histogram of features. See above for a caveat on the interpretation of these.

2.10 Speeded-Up Robust Features

New in version 0.6.1: SURF is only available starting in version 0.6.1, with an important bugfix in version 0.6.2.New in version 0.8: In version 0.8, some of the inner functions are now in mahotas.features.surf instead of mahotas.surf Speeded-Up Robust Features (SURF) are a recent innnovation in the *local features* family. There are two steps to this algorithm:

1. Detection of interest points.
2. Description of interest points.

The function `mahotas.features.surf.surf` combines the two steps:

```
import numpy as np
from mahotas.features import surf

f = ... # input image
spoints = surf.surf(f)
print "Nr points:", len(spoints)
```

Given the results, we can perform a simple clustering using, for example, milk (we could have used any other system, of course; having written milk, I am most familiar with it):

```
try:
    import milk

    # spoints includes both the detection information (such as the position
    # and the scale) as well as the descriptor (i.e., what the area around
    # the point looks like). We only want to use the descriptor for
    # clustering. The descriptor starts at position 5:
    descrs = spoints[:,5:]

    # We use 5 colours just because if it was much larger, then the colours
    # would look too similar in the output.
    k = 5
    values, _  = milk.kmeans(descrs, k)
    colors = np.array([(255-52*i,25+52*i,37**i % 101) for i in xrange(k)])
except:
    values = np.zeros(100)
    colors = [(255,0,0)]
```

So we are assigning different colours to each of the possible

The helper `surf.show_surf` draws coloured polygons around the interest points:

```
f2 = surf.show_surf(f, spoints[:100], values, colors)
imshow(f2)
show()
```

Running the above on a photo of luispedro, the author of mahotas yields:

```
from __future__ import print_function
import numpy as np
import mahotas
from mahotas.features import surf
from pylab import *

from os import path

try:
    luispedro_image = path.join(
                    path.dirname(path.abspath(__file__)),
                    'data',
                    'luispedro.jpg')
except NameError:
    luispedro_image = 'data/luispedro.jpg'

f = mahotas.imread(luispedro_image, as_grey=True)
f = f.astype(np.uint8)
spoints = surf.surf(f, 4, 6, 2)
print("Nr points:", len(spoints))

try:
    import milk
    descrs = spoints[:,5:]
    k = 5
    values, _  =milk.kmeans(descrs, k)
    colors = np.array([(255-52*i,25+52*i,37**i % 101) for i in range(k)])
except:
    values = np.zeros(100)
    colors = np.array([(255,0,0)])

f2 = surf.show_surf(f, spoints[:100], values, colors)
imshow(f2)
show()
```

2.10.1 API Documentation

The `mahotas.features.surf` module contains separate functions for all the steps in the SURF pipeline.

`mahotas.features.surf.`**`integral`**(*f*, *in_place=False*, *dtype=<type 'numpy.float64'>*)

fi = integral(f, in_place=False, dtype=np.double):

Compute integral image

Parameters **f** : ndarray

input image. Only 2-D images are supported.

in_place : bool, optional

Whether to overwrite *f* (default: False).

dtype : dtype, optional

dtype to use (default: double)

Returns **fi** : ndarray of *dtype* of same shape as *f*

The integral image

`mahotas.features.surf.`**`surf`**(*f*, *nr_octaves=4*, *nr_scales=6*, *initial_step_size=1*, *threshold=0.1*, *max_points=1024*, *descriptor_only=False*) points = surf(f, nr_octaves=4, nr_scales=6, initial_step_size=1, threshold=0.1, max_points=1024, descriptor_only=False):

Run SURF detection and descriptor computations

Speeded-Up Robust Features (SURF) are fast local features computed at automatically determined keypoints.

Parameters **f** : ndarray
input image

nr_octaves : integer, optional
Nr of octaves (default: 4)

nr_scales : integer, optional
Nr of scales (default: 6)

initial_step_size : integer, optional
Initial step size in pixels (default: 1)

threshold : float, optional
Threshold of the strength of the interest point (default: 0.1)

max_points : integer, optional
Maximum number of points to return. By default, return at most 1024 points. Note that the number may be smaller even in the case where there are that many points. This is a side-effect of the way the threshold is implemented: only `max_points` are considered, but some of those may be filtered out.

descriptor_only : boolean, optional
If `descriptor_only`, then returns only the 64-element descriptors

Returns **points** : ndarray of double, shape = (N, 6 + 64)
N is nr of points. Each point is represented as *(y,x,scale,score,laplacian,angle, D_0,...,D_63)* where *y,x,scale* is the position, *angle* the orientation, *score* and *laplacian* the score and sign of the detector; and *D_i* is the descriptor

If `descriptor_only`, then only the *D_i*s are returned and the array has shape (N, 64)!

2.11 Classification Using Mahotas

New in version 0.8: Before version 0.8, texture was under mahotas, not under mahotas.features Here is an example of using mahotas and milk for image classification (but most of the code can easily be adapted to use another machine learning package). I assume that there are three important directories: `positives/` and `negatives/` contain the manually labeled examples, and the rest of the data is in an `unlabeled/` directory.

Here is the simple algorithm:

1. Compute features for all of the images in positives and negatives
2. learn a classifier
3. use that classifier on the unlabeled images

In the code below I used jug to give you the possibility of running it on multiple processors, but the code also works if you remove every line which mentions `TaskGenerator`.

We start with a bunch of imports:

```
from glob import glob
import mahotas
import mahotas.features
import milk
from jug import TaskGenerator
```

Now, we define a function which computes features. In general, texture features are very fast and give very decent results:

```
@TaskGenerator
def features_for(imname):
    img = mahotas.imread(imname)
    return mahotas.features.haralick(img).mean(0)
```

`mahotas.features.haralick` returns features in 4 directions. We just take the mean (sometimes you use the spread `ptp()` too).

Now a pair of functions to learn a classifier and apply it. These are just `milk` functions:

```
@TaskGenerator
def learn_model(features, labels):
    learner = milk.defaultclassifier()
    return learner.train(features, labels)

@TaskGenerator
def classify(model, features):
     return model.apply(features)
```

We assume we have three pre-prepared directories with the images in jpeg format. This bit you will have to adapt for your own settings:

```
positives = glob('positives/*.jpg')
negatives = glob('negatives/*.jpg')
unlabeled = glob('unlabeled/*.jpg')
```

Finally, the actual computation. Get features for all training data and learn a model:

```
features = map(features_for, negatives + positives)
labels = [0] * len(negatives) + [1] * len(positives)

model = learn_model(features, labels)

labeled = [classify(model, features_for(u)) for u in unlabeled]
```

This uses texture features, which is probably good enough, but you can play with other features in `mahotas.features` if you'd like (or try `mahotas.surf`, but that gets more complicated).

(This was motivated by a question on Stackoverflow).

2.12 Morphological Operators

New in version 0.8: open() & close() were added in version 0.8 Morphological operators were the first operations in mahotas (back then, it was even, briefly, just a single C++ module called `morph`). Since then, mahotas has grown a lot. This module, too, has grown and acquired more morphological operators as well as being optimised for speed.

Let us first select an interesting image

2.12.1 Dilation & Erosion

Dilation and erosion are two very basic operators (mathematically, you only need one of them as you can define the erosion as dilation of the negative or vice-versa).

These operations are available in the `mahotas.morph` module:

```
mahotas.morph.dilate(eye)
```

Dilation is, intuitively, making positive areas "fatter":

```
mahotas.morph.erode(eye)
```

Erosion, by contrast, thins them out:

Mahotas supports greyscale erosion and dilation (depending on the `dtype` of the arguments) and you can specify any structuring element you wish (including non-flat ones). By default, a 1-cross is used:

```
# if no structure-element is passed, use a cross:
se = np.array([
        [0, 1, 0],
        [1, 1, 1],
        [0, 1, 0]], bool)
```

However, you can use whatever structuring element you want:

```
se = np.array([
    [1, 1, 0],
    [1, 1, 1],
    [0, 1, 1]], bool)
dilated = mahotas.morph.dilate(eye, se)
eroded = mahotas.morph.erode(eye, se)
```

Note that when you pass it a non-boolean array as the first argument, you will get *grescale erosion*. Mahotas supports full grescale erosion, including arbitrary, flat or non-flat, structuring elements).

2.12.2 Close & Open

Closing and opening are based on erosion and dilation. Again, they work in greyscale and can use an arbitrary structure element.

Here is closing:

```
mahotas.morph.close(eye)
```

And here is opening:

```
mahotas.morph.open(eye)
```

Both `close` and `open` take an optional structuring element as a second argument:

```
mahotas.morph.open(eye, se)
```

2.13 Input/Output with Mahotas

Mahotas does not have any builtin support for input/output. However, it wraps a few other libraries that do. The result is that you can do:

```
import mahotas as mh
image = mh.imread('file.png')
mh.imwrite('copy.png', image)
```

It can use the following backends (it tries them in the following order):

1. It prefers imread, if it is available. Imread is a native C++ library which reads images into Numpy arrays. It supports PNG, JPEG, TIFF, WEBP, BMP, and a few TIFF-based microscopy formats (LSM and STK).
2. It also looks for freeimage. Freeimage can read and write many formats. Unfortunately, it is harder to install and it is not as well-maintained as imread.
3. As a final fallback, it tries to use matplotlib, which has builtin PNG support and wraps PIL for other formats.

2.14 Frequently Asked Questions

2.14.1 Why did you not simply contribute to `scipy.ndimage` or `scikits.image`?

When I started this project (although it wasn't called mahotas and it was more of a collection of semi-organised routines than a project), there was no `scikits.image`.

In the meanwhile, all these projects have very different internal philosophies. `ndimage` is old-school scipy, in C, with macros. `scikits.image` uses Cython extensively, while `mahotas` uses C++ and templates. I don't want to use Cython as I find that it is not yet established enough and it cannot (I believe) be used to write functions that run on multiple types (like with C++ templates). The scipy community does not want to use C++.

I have, on the other hand, taken code from ndimage and ported it to C++ for use in mahotas. In the process, I feel it is much cleaner code (because you can use RAII, exceptions, and templates) and I want to keep it that way.

In any case, we all use the same data format: numpy arrays. It is very easy (trivial, really) to use all the packages together and take whatever functions you want from each. All the packages use function based interfaces which make it easy to mix-and-match.

2.14.2 What are the parameters to Local Binary Patterns?

Checkout the documentation on local binary patterns.

2.14.3 I am using mahotas in a scientific publication, is there a citation?

There is a manuscript about mahotas under review. In the meanwhile, only a pre-print is available at the arXiv. You can cite the preprint.

2.15 Mahotas Internals

This section is of interest if you are trying to understand how mahotas works in order to fix something, extend it (patches are always welcome), or use some of its technology in your projects.

2.15.1 Philosophy

Mahotas should not suck.

This is my main development goal and, if I achieve it, this alone should put mahotas in the top ten to one percent of software packages.

Mahotas should have no bugs. None. Ever.

Of course, some creep in. So, we settle for the next best thing: *Mahotas should have no **known bugs**.* Whenever a bug is discovered, the top priority is to squash it.

Read the principles of mahotas

2.15.2 C++/Python Division

Mahotas is written in C++, but almost always, you call a Python function which checks types and then calls the internal function. This is slightly slower, but it is easier to develop this way (and, for all but the smallest image, it will not matter).

So each `module.py` will have its associated `_module.cpp`.

2.15.3 C++ Templates

The main reason that mahotas is in C++ (and not in pure C) is to use templates. Almost all C++ functions are actually 2 functions:

1. A `py_function` which uses the Python C/API to get arguments, &c. This is almost always pure C.
2. A template `function<dtype>` which works for the `dtype` performing the actual operation.

So, for example, this is how *erode* is implemented. `py_erode` is generic:

```
PyObject* py_erode(PyObject* self, PyObject* args) {
    PyArrayObject* array;
    PyArrayObject* Bc;
    if (!PyArg_ParseTuple(args,"OO", &array, &Bc)) return NULL;
    PyArrayObject* res_a = (PyArrayObject*)PyArray_SimpleNew(array->nd,array->dimensions,PyArray_TYPE
    if (!res_a) return NULL;
    PyArray_FILLWBYTE(res_a, 0);
#define HANDLE(type) \
    erode<type>(numpy::aligned_array<type>(res_a), numpy::aligned_array<type>(array), numpy::aligned_

        SAFE_SWITCH_ON_INTEGER_TYPES_OF(array)
#undef HANDLE
    ...
```

These functions normally contain a lot of boiler-plate code: read the arguments, perform some sanity checks, perhaps a bit of initialisation, and then, the switch on the input type with the help of the `SAFE_SWITCH_ON_INTEGER_TYPES_OF()` and friends, which call the right specialisation of the template that does the actual work. In this example `erode` implements (binary) erosion:

```
template<typename T>
void erode(numpy::aligned_array<T> res, numpy::aligned_array<T> array, numpy::aligned_array<T> Bc) {
    gil_release nogil;
    const unsigned N = res.size();
    typename numpy::aligned_array<T>::iterator iter = array.begin();
    filter_iterator<T> filter(res.raw_array(), Bc.raw_array());
    const unsigned N2 = filter.size();
    T* rpos = res.data();

    for (int i = 0; i != N; ++i, ++rpos, filter.iterate_with(iter), ++iter) {
        for (int j = 0; j != N2; ++j) {
            T arr_val = false;
            filter.retrieve(iter, j, arr_val);
            if (filter[j] && !arr_val) goto skip_this_one;
        }
        *rpos = true;
        skip_this_one: continue;
    }
}
```

The template machinery is not that complicated and the functions using it are very simple and easy to read. The only downside is that there is some expansion of code size. Given the small size of these functions however, this is not a big issue.

In the snippet above, you can see some other C++ machinery:

`gil_release` This is a RAII object that release the GIL in its constructor and gets it back in its destructor. Normally, the template function will release the GIL after the Python-specific code is done.

`array` This is a thin wrapper around `PyArrayObject` that knows its type and has iterators.

`filter_iterator` This is taken from `scipy.ndimage` and it is useful to iterate over an image and use a centered filter around each pixel (it keeps track of all of the boundary conditions).

The inner loop is as direct an implementation of erosion as one would wish for: for each pixel in the image, look at its neighbours. If all are true, then set the corresponding output pixel to `true` (else, skip it as it has been initialised to zero).

Most of the functions follow this architecture.

2.16 The Why of mahotas

2.16.1 Principles of Mahotas

Here are the principles of mahotas, in decreasing order of importance:

1. Just work
2. Well documented
3. Fast code
4. Simple code
5. Minimal dependencies

2.16.2 Just work

The first principle is that things should *just work*. This means two things: (1) there should be no bugs, and (2) interfaces should be flexible or fail well.

To avoid bugs, tests are extensively used. Every reported bug leads to a new test case, so that it never happens again. New features should at least have a smoke test (test that runs the feature and verifies some basic properties of the output).

Interfaces are designed to be as flexible as possible. No specific types are required unless it is really needed or in performance-enhancing features (such as using `out` parameters).

The user should never be able to crash the Python interpreter with mahotas.

2.16.3 Well documented

No public function is without a complete docstring. In addition to that *hard documentation* (i.e., information with complete technical detail of every nook and cranny of the interface), there is also *soft documentation* (tutorial-like documentation with examples and higher level reasoning).

2.16.4 Fast code

Performance is a feature.

The code should be as fast as possible without sacrificing generality (see *just work* above). This is why C++ templates are used for type independent code.

2.16.5 Simple code

The code should be simple.

2.16.6 Minimal dependencies

Mahotas tries to avoid dependencies.

Right now, building mahotas depends on a C++ compiler, numpy. These are unlikely to ever change. To run mahotas, we need numpy. In order to read images, we need one of (1) imread, (2) FreeImage, or (3) matplotlib.

The imread/freeimage dependency is a soft dependency: everything, except for imread works without it. The code is written to ensure that `import`-ing mahotas without an IO backend will not trigger an error unless the `imread()` function is used.

Therefore, once mahotas is compiled, all you really need is numpy. This is unlikely to ever change.

2.17 Contributing

Development happens on github and the preferred contribution method is by forking the repository and issuing a pull request. Alternatively, just sending a patch to luis@luispedro.org will work just as well.

If you don't know git (or another distributed version control system), which is a fantastic tool in general, there are several good git and github tutorials. You can start with their official documentation.

If you want to work on the C++ code, you can read the chapter in the internals before you start. Also, read the principles declaration.

2.17.1 Debug Mode

If you compile mahotas in debug mode, then it will run slower but perform a lot of runtime checks. This is controlled by the `DEBUG` environment variable.

There are two levels:

1. `DEBUG=1` This turns on assertions. The code will run slower, but probably not noticeably slower, except for very large images.
2. `DEBUG=2` This turns on the assertions and additionally uses the debug version of the C++ library (this is probably only working if you are using GCC). Some of the internal code also picks up on this and adds even more sanity checking. The result will be code that runs **much slower** as all operations done through iterators into standard containers are now checked (including many inner loop operations).

The Makefile that comes with the source helps you:

```
make clean
make debug
make test
```

will rebuild in debug mode and run all tests. When you are done testing, use the `fast` Make target to get the non-debug build:

```
make clean
make fast
```

Using make will not change your environment. The `DEBUG` variable is set internally only.

If you don't know about it, check out ccache which is a great tool if you are developing in compiled languages (this is not specific to mahotas or even Python). It will allow you to quickly perform `make clean; make debug` and `make clean; make fast` so you never get your builds mixed up.

2.18 Possible Tasks

Here are a few ideas for improving mahotas.

2.18.1 New Features

- HOG
- BRISK
- Canny edge detection
- Hough Transform
- bilateral filtering
- Non Local Filtering
- Wiener filtering

2.18.2 Small Improvements

- something like the `overlay` function from pymorph (or even just copy it over and adapt it to mahotas style).
- H-maxima transform (again, pymorph can provide a basis)
- entropy thresholding

2.18.3 Internals

These can be very complex as they require an understanding of the inner workings of mahotas, but that does appeal to a certain personality.

- special case 1-D convolution on C-Arrays in C++. The idea is that you can

write a tight inner loop in one dimension:

```
void multiply(floating* r, const floating* f, const floating a, const int n, const int r_step, const
    for (int i = 0; i != n; ++i) {
        *r += a * *f;
        r += r_step;
        f += f_step;
    }
}
```

to implement:

```
r[row] += a* f[row+offset]
```

and you can call this with all the different values of `a` and `offset` that make up your filter. This would be useful for Guassian filtering.

2.18.4 Tutorials

Mahotas has very good API documentation, but not so many *start to finish* tutorials which touch several parts of it (and even other packages, the ability to seamlessly use other packages in Python is, of course, a good reason to use it).

2.19 History

2.19.1 1.0 (May 21 2013)

- Fix a few corner cases in texture analysis
- Integrate with travis
- Update citation (include DOI)

2.19.2 0.99 (May 4 2013)

- Make matplotlib a soft dependency
- Add demos.image_path() function
- Add citation() function

This version is **1.0 beta**.

2.19.3 0.9.8 (April 22 2013)

- Use matplotlib as IO backend (fallback only)
- Compute dense SURF features
- Fix sobel edge filtering (post-processing)
- Faster 1D convultions (including faster Gaussian filtering)
- Location independent tests (run mahotas.tests.run() anywhere)
- Add labeled.is_same_labeling function
- Post filter SLIC for smoother regions
- Fix compilation warnings on several platforms

2.19.4 0.9.7 (February 03 2013)

- Add `haralick_features` function
- Add `out` parameter to morph functions which were missing it
- Fix erode() & dilate() with empty structuring elements
- Special case binary erosion/dilation in C-Arrays
- Fix long-standing warning in TAS on zero inputs
- Add `verbose` argument to tests.run()
- Add `circle_se` to `morph`
- Allow `loc(max|min)` to take floating point inputs
- Add Bernsen local thresholding (`bernsen` and `gbernsen` functions)

2.19.5 0.9.6 (December 02 2012)

- Fix `distance()` of non-boolean images (issue #24 on github)
- Fix encoding issue on PY3 on Mac OS (issue #25 on github)
- Add `relabel()` function
- Add `remove_regions()` function in labeled module
- Fix `median_filter()` on the borders (respect the `mode` argument)
- Add `mahotas.color` module for conversion between colour spaces
- Add SLIC Superpixels
- Many improvements to the documentation

2.19.6 0.9.5 (November 05 2012)

- Fix compilation in older G++
- Faster Otsu thresholding
- Python 3 support without 2to3
- Add `cdilate` function
- Add `subm` function
- Add tophat transforms (functions `tophat_close` and `tophat_open`)
- Add `mode` argument to euler() (patch by Karol M. Langner)
- Add `mode` argument to bwperim() & borders() (patch by Karol M. Langner)

2.19.7 0.9.4 (October 10 2012)

- Fix compilation on 32-bit machines (Patch by Christoph Gohlke)

2.19.8 0.9.3 (October 9 2012)

- Fix interpolation (Report by Christoph Gohlke)
- Fix second interpolation bug (Report and patch by Christoph Gohlke)
- Update tests to newer numpy
- Enhanced debug mode (compile with DEBUG=2 in environment)
- Faster morph.dilate()
- Add labeled.labeled_max & labeled.labeled_min (This also led to a refactoring of the labeled_* code)
- Many documentation fixes

2.19.9 0.9.2 (September 1 2012)

- Fix compilation on Mac OS X 10.8 (reported by Davide Cittaro)
- Freeimage fixes on Windows by Christoph Gohlke
- Slightly faster _filter implementaiton

2.19.10 0.9.1 (August 28 2012)

- Python 3 support (you need to use `2to3`)
- Haar wavelets (forward and inverse transform)
- Daubechies wavelets (forward and inverse transform)
- Corner case fix in Otsu thresholding
- Add soft_threshold function
- Have polygon.convexhull return an ndarray (instead of a list)
- Memory usage improvements in regmin/regmax/close_holes (first reported as issue #9 by thanasi)

2.19.11 0.9 (July 16 2012)

- Auto-convert integer to double on gaussian_filter (previously, integer values would result in zero-valued outputs).
- Check for integer types in (reg|loc)(max|min)
- Use name *out* instead of *output* for output arguments. This matches Numpy better
- Switched to MIT License

2.19.12 0.8.1 (June 6 2012)

- Fix gaussian_filter bug when order argument was used (reported by John Mark

Agosta) - Add morph.cerode - Improve regmax() & regmin(). Rename previous implementations to locmax() & locmin() - Fix erode() on non-contiguous arrays

2.19.13 0.8 (May 7 2012)

- Move features to submodule
- Add morph.open function
- Add morph.regmax & morph.regmin functions
- Add morph.close function
- Fix morph.dilate crash

2.19.14 0.7.3 (March 14 2012)

- Fix installation of test data
- Greyscale erosion & dilation
- Use imread module (if available)
- Add output argument to erode() & dilate()
- Add 14th Haralick feature (patch by MattyG) — currently off by default
- Improved zernike interface (zernike_moments)
- Add remove_bordering to labeled
- Faster implementation of `bwperim`
- Add `roundness` shape feature

2.19.15 0.7.2 (February 13 2012)

There were two minor additions:

- Add as_rgb (especially useful for interactive use)
- Add Gaussian filtering (from scipy.ndimage)

And a few bugfixes:

- Fix type bug in 32 bit machines (Bug report by Lech Wiktor Piotrowski)
- Fix convolve1d
- Fix rank_filter

2.19.16 0.7.1 (January 6 2012)

The most important change fixed compilation on Mac OS X

Other changes:

- Add convolve1d
- Check that convolution arguments have right dimensions (instead of crashing)
- Add descriptor_only argument to surf.descriptors
- Specify all function signatures on freeimage.py

For version **0.7 (Dec 5 2011)**:

The big change was that the *dependency on scipy was removed*. As part of this process, the interpolate submodule was added. A few important bug fixes as well.

- Allow specification of centre in Zernike moment computation
- Fix Local Binary Patterns
- Remove dependency on scipy
- Add interpolate module (from scipy.ndimage)
- Add labeled_sum & labeled_sizes
- gvoronoi no longer depends on scipy
- mahotas is importable without scipy
- Fix bugs in 2D TAS (reported by Jenn Bakal)
- Support for 1-bit monochrome image loading with freeimage
- Fix GIL handling on errors (reported by Gareth McCaughan)
- Fix freeimage for 64-bit computers

For version **0.6.6 (August 8 2011)**: - Fix fill_polygon bug (fix by joferkington) - Fix Haralick feature 6 (fix by Rita Simões) - Implement `morph.get_structuring_element` for ndim > 2. This implies that functions such as `label()` now also work in multiple dimensions - Add median filter & `rank_filter` functions - Add template_match function - Refactor by use of mahotas.internal - Better error message for when the compiled modules cannot be loaded - Update contact email. All docs in numpydoc format now.

For version **0.6.5**: - Add `max_points` & `descriptor_only` arguments to mahotas.surf - Fix haralick for 3-D images (bug report by Rita Simões) - Better error messages - Fix hit&miss for non-boolean inputs - Add `label()` function

For version **0.6.4**:

- Fix bug in `cwatershed()` when using return_lines=1
- Fix bug in `cwatershed()` when using equivalent types for image and markers
- Move tests to mahotas.tests and include them in distribution
- Include ChangeLog in distribution

- Fix compilation on the Mac OS
- Fix compilation warnings on gcc

For version **0.6.3**:

- Improve `mahotas.stretch()` function
- Fix corner case in surf (when determinant was zero)
- `threshold` argument in mahotas.surf
- imreadfromblob() & imsavetoblob() functions
- `max_points` argument for mahotas.surf.interest_points()
- Add `mahotas.labeled.borders` function

For version **0.6.2**:

Bugfix release:

- Fix memory leak in _surf
- More robust searching for freeimage
- More functions in mahotas.surf() to retrieve intermediate results
- Improve compilation on Windows (patches by Christoph Gohlke)

For version **0.6.1**:

- Release the GIL in morphological functions
- Convolution
- just_filter option in edge.sobel()
- mahotas.labeled functions
- SURF local features

For version **0.6**:

- Improve Local Binary patterns (faster and better interface)
- Much faster erode() (10x faster)
- Faster dilate() (2x faster)
- TAS for 3D images
- Haralick for 3D images

Support

Website: http://luispedro.org/software/mahotas

API Docs: http://packages.python.org/mahotas/

Mailing List: Use the pythonvision mailing list for questions, bug submissions, etc.

2.20 Full API Documentation

A package for computer vision in Python.

2.20.1 Main Features

features Compute global and local features (several submodules, include SURF and Haralick features)

convolve Convolution and wavelets

morph Morphological features. Most are available at the mahotas level, include erode(), dilate()...

watershed Seeded watershed implementation

imread/imsave read/write image

Citation:

> Coelho, Luis Pedro, 2013. Mahotas: Open source software for scriptable computer vision. Journal of Open Research Software, 1:e3, DOI: http://dx.doi.org/10.5334/jors.ac

`mahotas.`**`as_rgb`**`(`*r*, *g*, *b*`)`

Returns an RGB image

If any of the channels is *None*, that channel is set to zero.

Parameters **r,g,b** : array-like, optional

The channels can be of any type or None. At least one must be not None and all must have the same shape.

Returns **rgb** : ndarray

RGB ndarray

`mahotas.`**`bbox`**`(`*img*`)`

Calculate the bounding box of image img.

Parameters **img** : ndarray

Any integer image type

Returns **min1,max1,min2,max2** : int,int,int,int

These are such that `img[min1:max1, min2:max2]` contains all non-zero pixels

`mahotas.`**`border`**`(`*labeled*, *i*, *j*, *Bc={3x3 cross}*, *out={np.zeros(labeled.shape, bool)}*, *always_return=True*`)`

Compute the border region between *i* and *j* regions.

A pixel is on the border if it has value *i* (or *j*) and a pixel in its neighbourhood (defined by *Bc*) has value *j* (or *i*).

Parameters **labeled** : ndarray of integer type

input labeled array

i : integer

j : integer

Bc : structure element, optional

out : ndarray of same shape as *labeled*, dtype=bool, optional

where to store the output. If `None`, a new array is allocated

always_return : bool, optional

if false, then, in the case where there is no pixel on the border, returns `None`. Otherwise (the default), it always returns an array even if it is empty.

Returns **border_img** : boolean ndarray

Pixels are True exactly where there is a border between *i* and *j* in *labeled*

`mahotas.`**`borders`**`(`*labeled*, *Bc={3x3 cross}*, *out={np.zeros(labeled.shape, bool)}*`)`

Compute border pixels

A pixel is on a border if it has value *i* and a pixel in its neighbourhood (defined by *Bc*) has value *j*, with `i != j`.

Parameters **labeled** : ndarray of integer type

input labeled array

Bc : structure element, optional

out : ndarray of same shape as *labeled*, dtype=bool, optional
where to store the output. If `None`, a new array is allocated

mode : {'reflect', 'nearest', 'wrap', 'mirror', 'constant' [default], 'ignore'}
How to handle borders

Returns **border_img** : boolean ndarray
Pixels are True exactly where there is a border in *labeled*

`mahotas.`**`bwperim`**(*bw*, *n=4*)

Find the perimeter of objects in binary images.

A pixel is part of an object perimeter if its value is one and there is at least one zero-valued pixel in its neighborhood.

By default the neighborhood of a pixel is 4 nearest pixels, but if *n* is set to 8 the 8 nearest pixels will be considered.

Parameters **bw** : ndarray
A black-and-white image (any other image will be converted to black & white)

n : int, optional
Connectivity. Must be 4 or 8 (default: 4)

mode : {'reflect', 'nearest', 'wrap', 'mirror', 'constant' [default], 'ignore'}
How to handle borders

Returns **perim** : ndarray
A boolean image

See Also:

`borders` function This is a more generic function

`mahotas.`**`cdilate`**(*f*, *g*, *Bc={3x3 cross}*, *n=1*)

Conditional dilation

cdilate creates the image *y* by dilating the image *f* by the structuring element *Bc* conditionally to the image *g*. This operator may be applied recursively *n* times.

Parameters **f** : Gray-scale (uint8 or uint16) or binary image.

g : Conditioning image. (Gray-scale or binary).

Bc : Structuring element (default: 3x3 cross)

n : Number of iterations (default: 1)

Returns **y** : Image

`mahotas.`**`center_of_mass`**(*img*, *labels=None) x0*, *x1*, *... = center_of_mass(img*, *labels=None*)

Returns the center of mass of img.

If *labels* is given, then it returns *L* centers of mass, one for each region identified by *labels* (including region 0).

Parameters **img** : ndarray

labels : ndarray
A labeled array

Returns **coords** : ndarray
if `not labels`, a 1-ndarray of coordinates (size = len(img.shape)), if `labels`, a 2-ndarray of coordinates (shape = (labels.max()+1) xlen(img.shape))

`mahotas.`**`cerode`**(*f*, *g*, *Bc={3x3 cross}*, *out={np.empty_as(A)}*)

Conditional morphological erosion.

The type of operation depends on the `dtype` of `A`! If boolean, then the erosion is binary, else it is greyscale erosion. In the case of greyscale erosion, the smallest value in the domain of `Bc` is interpreted as -Inf.

Parameters **f** : ndarray

input image

g : ndarray
conditional image

Bc : ndarray, optional
Structuring element. By default, use a cross (see `get_structuring_elem` for details on the default).

Returns **conditionally_eroded** : ndarray
eroded version of `f` conditioned on `g`

See Also:

`erode` function Unconditional version of this function

`dilate`

`mahotas.`**`close`** (*f*, *Bc={3x3 cross}*, *out={np.empty_like(f)}*)
Morphological closing.

close creates the image *y* by the morphological closing of the image *f* by the structuring element *Bc*. In the binary case, the closing by a structuring element *Bc* may be interpreted as the intersection of all the binary images that contain the image *f* and have a hole equal to a translation of *Bc*. In the gray-scale case, there is a similar interpretation taking the functions umbra.

Parameters **f** : ndarray
Gray-scale (uint8 or uint16) or binary image.

Bc : ndarray, optional
Structuring element. (Default: 3x3 elementary cross).

out : ndarray, optional
Output array

Returns **y** : ndarray

See Also:

`open` function

`mahotas.`**`close_holes`** (*ref*, *Bc=None*)
closed = close_holes(ref, Bc=None):

Close Holes

Parameters **ref** : ndarray
Reference image. This should be a binary image.

Bc : structuring element, optional
Default: 3x3 cross

Returns **closed** : ndarray
superset of *ref* (i.e. with closed holes)

`mahotas.`**`convolve`** (*f*, *weights*, *mode='reflect'*, *cval=0.0*, *out={new array}*)
Convolution of *f* and *weights*

Convolution is performed in *doubles* to avoid over/underflow, but the result is then cast to *f.dtype*.

Parameters **f** : ndarray
input. Any dimension is supported

weights : ndarray
weight filter. If not of the same dtype as *f*, it is cast

mode : {'reflect' [default], 'nearest', 'wrap', 'mirror', 'constant', 'ignore'}
How to handle borders

cval : double, optional
If *mode* is constant, which constant to use (default: 0.0)

out : ndarray, optional
Output array. Must have same shape and dtype as *f* as well as be C-contiguous.

Returns **convolved** : ndarray of same dtype as *f* `mahotas.`**`convolve1d`**(*f*, *weights*, *axis*, *mode='reflect'*, *cval=0.0*, *out={new array}*)

Convolution of *f* and *weights* along axis *axis*.

Convolution is performed in *doubles* to avoid over/underflow, but the result is then cast to *f.dtype*.

Parameters **f** : ndarray
: input. Any dimension is supported

weights : 1-D ndarray
: weight filter. If not of the same dtype as *f*, it is cast

axis : int
: Axis along which to convolve

mode : {'reflect' [default], 'nearest', 'wrap', 'mirror', 'constant', 'ignore'}
: How to handle borders

cval : double, optional
: If *mode* is constant, which constant to use (default: 0.0)

out : ndarray, optional
: Output array. Must have same shape and dtype as *f* as well as be C-contiguous.

Returns **convolved** : ndarray of same dtype as *f*

See Also:

`convolve` function generic convolution

`mahotas.`**`croptobbox`**(*img*, *border=0*)

Returns a version of img cropped to the image's bounding box

Parameters **img** : ndarray
: Integer image array

border : int, optional
: whether to add a border (default no border)

Returns **nimg** : ndarray
: A subimage of img.

`mahotas.`**`cwatershed`**(*surface*, *markers*, *Bc=None*, *return_lines=False) W*, *WL = cwatershed(surface*, *markers*, *Bc=None*, *return_lines=True*)

Seeded Watershed

Parameters **surface** : image

markers : image
: initial markers (must be a labeled image)

Bc : ndarray, optional
: structuring element (default: 3x3 cross)

return_lines : boolean, optional
: whether to return separating lines (in addition to regions)

Returns **W** : Regions image (i.e., W[i,j] == region for pixel (i,j))

WL : Lines image (*if return_lines==True*)

`mahotas.`**`daubechies`**(*f*, *code*, *inline=False*)

Daubechies wavelet transform

This function works best if the image sizes are powers of 2!

Parameters **f** : ndarray
: 2-D image

code : str
: One of 'D2', 'D4', ... 'D20'

inline : bool, optional
: Whether to write the results to the input image. By default, a new image is returned. Integer images are always converted to floating point and copied.

See Also:

`haar` function Haar transform (equivalent to D2) `mahotas.`**`dilate`**(*A*, *Bc={3x3 cross}*, *out={np.empty_like(A)}*)

Morphological dilation.

The type of operation depends on the `dtype` of `A`! If boolean, then the dilation is binary, else it is greyscale dilation. In the case of greyscale dilation, the smallest value in the domain of `Bc` is interpreted as +Inf.

Parameters **A** : ndarray of bools

input array

Bc : ndarray, optional

Structuring element. By default, use a cross (see `get_structuring_elem` for details on the default).

Returns **dilated** : ndarray

dilated version of `A`

See Also:

`erode`

`mahotas.`**`distance`**(*bw*, *metric='euclidean2'*)

Computes the distance transform of image *bw*:

```
dmap[i,j] = min_{i', j'} { (i-i')**2 + (j-j')**2 | !bw[i', j'] }
```

That is, at each point, compute the distance to the background.

If there is no background, then a very high value will be returned in all pixels (this is a sort of infinity).

Parameters **bw** : 2d-ndarray

If boolean, `False` will denote the background and `True` the foreground. If not boolean, this will be interpreted as `bw != 0` (this way you can use labeled images without any problems).

metric : str, optional

one of 'euclidean2' (default) or 'euclidean'

Returns **dmap** : ndarray

distance map

`mahotas.`**`erode`**(*A*, *Bc={3x3 cross}*, *out={np.empty_as(A)}*)

Morphological erosion.

The type of operation depends on the `dtype` of `A`! If boolean, then the erosion is binary, else it is greyscale erosion. In the case of greyscale erosion, the smallest value in the domain of `Bc` is interpreted as -Inf.

Parameters **A** : ndarray

input image

Bc : ndarray, optional

Structuring element. By default, use a cross (see `get_structuring_elem` for details on the default).

out : ndarray, optional

output array. If used, this must be a C-array of the same `dtype` as `A`. Otherwise, a new array is allocated.

Returns **erosion** : ndarray

eroded version of `A`

See Also:

`dilate`

`mahotas.`**`euler`**(*f*, *n=8*)

Compute the Euler number of image f

The Euler number is also known as the Euler characteristic given that many other mathematical objects are also known as Euler numbers.

Parameters **f** : ndarray

A 2-D binary image

n : int, optional

Connectivity, one of (4,8). default: 8

mode : {'reflect', 'nearest', 'wrap', 'mirror', 'constant' [default]}

How to handle borders

Returns **euler_nr** : int

Euler number

`mahotas.`**`fullhistogram`**`(`*img*`)`

Return a histogram with bins *0, 1, ..., ''img.max()''*.

After calling this function, it will be true that `hist[i] == (img == i).sum()`, for all `i`.

Parameters **img** : array-like of an unsigned type

input image.

Returns **hist** : an dnarray of type np.uint32

This will be of size `img.max() + 1`.

`mahotas.`**`gaussian_filter`**`(`*array*, *sigma*, *order=0*, *mode='reflect'*, *cval=0.*, *out={np.empty_like(array)}*`)`

Multi-dimensional Gaussian filter.

Parameters **array** : ndarray

input array, any dimension is supported. If the array is an integer array, it will be converted to a double array.

sigma : scalar or sequence of scalars

standard deviation for Gaussian kernel. The standard deviations of the Gaussian filter are given for each axis as a sequence, or as a single number, in which case it is equal for all axes.

order : {0, 1, 2, 3} or sequence from same set, optional

The order of the filter along each axis is given as a sequence of integers, or as a single number. An order of 0 corresponds to convolution with a Gaussian kernel. An order of 1, 2, or 3 corresponds to convolution with the first, second or third derivatives of a Gaussian. Higher order derivatives are not implemented

mode : {'reflect' [default], 'nearest', 'wrap', 'mirror', 'constant', 'ignore'}

How to handle borders

cval : double, optional

If *mode* is constant, which constant to use (default: 0.0)

out : ndarray, optional

Output array. Must have same shape as *array* as well as be C-contiguous. If *array* is an integer array, this must be a double array; otherwise, it must have the same type as *array*.

Returns **filtered** : ndarray

Filtered version of *array*

Notes

The multi-dimensional filter is implemented as a sequence of one-dimensional convolution filters. The intermediate arrays are stored in the same data type as the output. Therefore, for output types with a limited precision, the results may be imprecise because intermediate results may be stored with insufficient precision.

`mahotas.`**`gaussian_filter1d`**`(`*array*, *sigma*, *axis=-1*, *order=0*, *mode='reflect'*, *cval=0.*, *out={np.empty_like(array)}*`)`

One-dimensional Gaussian filter.

Parameters **array** : ndarray

input array of a floating-point type

sigma : float

standard deviation for Gaussian kernel (in pixel units)

axis : int, optional
: axis to operate on

order : {0, 1, 2, 3}, optional
: An order of 0 corresponds to convolution with a Gaussian kernel. An order of 1, 2, or 3 corresponds to convolution with the first, second or third derivatives of a Gaussian. Higher order derivatives are not implemented

mode : {'reflect' [default], 'nearest', 'wrap', 'mirror', 'constant', 'ignore'}
: How to handle borders

cval : double, optional
: If *mode* is constant, which constant to use (default: 0.0)

out : ndarray, optional
: Output array. Must have same shape and dtype as *array* as well as be C-contiguous.

Returns **filtered** : ndarray
: Filtered version of *array*

`mahotas.`**`get_structuring_elem`**(*A*, *Bc*)

Retrieve appropriate structuring element

Parameters **A** : ndarray
: array which will be operated on

Bc : None, int, or array-like

None Then Bc is taken to be 1

An integer

There are two associated semantics:

connectivity `Bc[y,x] = [[ is |y - 1| + |x - 1| <= Bc_i ]]`

count `Bc.sum() == Bc_i` This is the more traditional meaning (when one writes that "4-connected", this is what one has in mind).

Fortunately, the value itself allows one to distinguish between the two semantics and, if used correctly, no ambiguity should ever occur.

An array This should be of the same nr. of dimensions as A and will be passed through if of the right type. Otherwise, it will be cast.

Returns **Bc_out** : ndarray
: Structuring element. This array will be of the same type as A, C-contiguous.

`mahotas.`**`haar`**(*f*, *preserve_energy=True*, *inline=False*)

Haar transform

Parameters **f** : 2-D ndarray
Input image

preserve_energy : bool, optional
: Whether to normalise the result so that energy is preserved (the default).

inline : bool, optional
Whether to write the results to the input image. By default, a new image is returned. Integer images are always converted to floating point and copied.

See Also:

`ihaar` function Reverse Haar transform

`mahotas.`**`hitmiss`**(*input*, *Bc*, *out=np.zeros_like(input)*)

Hit & Miss transform

For a given pixel position, the hit&miss is `True` if, when `Bc` is overlaid on `input`, centered at that position, the `1` values line up with ``1``s, while the ``0``s line up with ``0``s (``2``s correspond to *don't care*).

Parameters **input** : input ndarray

This is interpreted as a binary array.

Bc : ndarray

hit & miss template, values must be one of (0, 1, 2)

out : output array

Returns **filtered** : ndarray

`mahotas.`**`idaubechies`**`(`*f*, *code*, *inline=False*`)`

Daubechies wavelet inverse transform

Parameters **f** : ndarray

2-D image

code : str

One of 'D2', 'D4', ... 'D20'

inline : bool, optional

Whether to write the results to the input image. By default, a new image is returned. Integer images are always converted to floating point and copied.

See Also:

`haar` function Haar transform (equivalent to D2)

`mahotas.`**`ihaar`**`(`*f*, *preserve_energy=True*, *inline=False*`)`

Reverse Haar transform

`ihaar(haar(f))` is more or less equal to `f` (equal, except for possible rounding issues).

Parameters **f** : 2-D ndarray

Input image. If it is an integer image, it is converted to floating point (double).

preserve_energy : bool, optional

Whether to normalise the result so that energy is preserved (the default).

inline : bool, optional

Whether to write the results to the input image. By default, a new image is returned. Integer images are always converted to floating point and copied.

Returns **f** : ndarray

See Also:

`haar` function Forward Haar transform

`mahotas.`**`imread`**`(`*filename*, *as_grey=False*`)`

Reads an image from file *filename*

Parameters **filename** : file name

as_grey : Whether to convert to grey scale image (default: no)

Returns **img** : ndarray

`mahotas.`**`imresize`**`(`*img*, *nsize*, *order=3*`)`

img' = imresize(img, nsize)

Resizes img

Parameters **img** : ndarray

nsize : float or tuple(float) or tuple(integers)

Size of return. Meaning depends on the type float: img'.shape[i] = nsize * img.shape[i] tuple of float: img'.shape[i] = nsize[i] * img.shape[i] tuple of int: img'.shape[i] = nsize[i]

order : integer, optional

Spline order to use (default: 3)

Returns **img'** : ndarray

See Also:

`scipy.ndimage.zoom` Similar function

`scipy.misc.pilutil.imresize` Similar function

`mahotas.`**`imsave`**`(`**args*, ***kwargs*`)`

Save an array as in image file.

The output formats available depend on the backend being used.

Arguments:

***fname*:** A string containing a path to a filename, or a Python file-like object. If *format* is *None* and *fname* is a string, the output format is deduced from the extension of the filename.

***arr*:** An MxN (luminance), MxNx3 (RGB) or MxNx4 (RGBA) array.

Keyword arguments:

***vmin*/*vmax*: [None | scalar]** *vmin* and *vmax* set the color scaling for the image by fixing the values that map to the colormap color limits. If either *vmin* or *vmax* is None, that limit is determined from the *arr* min/max value.

***cmap*:** cmap is a colors.Colormap instance, eg cm.jet. If None, default to the rc image.cmap value.

***format*:** One of the file extensions supported by the active backend. Most backends support png, pdf, ps, eps and svg.

origin ['upper' | 'lower'] Indicates where the [0,0] index of the array is in the upper left or lower left corner of the axes. Defaults to the rc image.origin value.

dpi The DPI to store in the metadata of the file. This does not affect the resolution of the output image.

`mahotas.`**`label`**`(`*array*, *Bc={3x3 cross}*, *output={new array}*`)`

Label the array, which is interpreted as a binary array

This is also called *connected component labeled*, where the connectivity is defined by the structuring element `Bc`.

See: http://en.wikipedia.org/wiki/Connected-component_labeling

Parameters **array** : ndarray

This will be interpreted as binary array

Bc : ndarray, optional

This is the structuring element to use

out : ndarray, optional

Output array. Must be a C-array, of type np.int32

Returns **labeled** : ndarray

Labeled result

nr_objects : int

Number of objects

`mahotas.`**`labeled_sum`**`(`*array*, *labeled*`)`

Labeled sum. sum will be an array of size `labeled.max() + 1`, where `sum[i]` is equal to `np.sum(array[labeled == i])`.

Parameters **array** : ndarray of any type

labeled : int ndarray

Label map. This is the same type as returned from `mahotas.label()`

Returns **sums** : 1-d ndarray of `array.dtype`

`mahotas.`**`majority_filter`**`(`*img*, *N=3*, *out={np.empty(img.shape, np.bool)}*`)`

Majority filter

filtered[y,x] is positive if the majority of pixels in the squared of size *N* centred on (y,x) are positive.

Parameters **img** : ndarray

input img (currently only 2-D images accepted)

N : int, optional

size of filter (must be odd integer), defaults to 3.

out : ndarray, optional

Used for output. Must be Boolean ndarray of same size as *img*

Returns **filtered** : ndarray
    boolean image of same size as img.

`mahotas.`**`median_filter`**(*f*, *Bc={square}*, *mode='reflect'*, *cval=0.0*, *out={np.empty(f.shape, f.dtype}*)

Median filter

Parameters **f** : ndarray
    input. Any dimension is supported
Bc : ndarray or int, optional
    Defines the neighbourhood, default is a square of side 3.
mode : {'reflect' [default], 'nearest', 'wrap', 'mirror', 'constant', 'ignore'}
    How to handle borders
cval : double, optional
    If *mode* is constant, which constant to use (default: 0.0)
out : ndarray, optional
    Output array. Must have same shape and dtype as *f* as well as be C-contiguous.

Returns **median** : ndarray of same type and shape as `f`
    median[i,j] is the median value of the points in f close to (i,j)

`mahotas.`**`moments`**(*img*, *p0*, *p1*, *cm=(0, 0)*, *convert_to_float=True*)

Returns the p0-p1 moment of image *img*

The formula computed is

sum_{ij} { img[i,j] (i - c0)**p0 (j - c1)**p1 }

where cm = (c0,c1). If *cm* is not given, then (0,0) is used.

If image is of an integer type, then it is internally converted to np.float64, unlesss *convert_to_float* is False. The reason is that, otherwise, overflow is likely except for small images. Since this conversion takes longer than the computation, you can turn it off in case you are sure that your images are small enough for overflow to be an issue. Note that no conversion is made if *img* is of any floating point type.

Parameters **img** : 2-ndarray
    An 2-d ndarray
p0 : float
    Power for first dimension
p1 : float
    Power for second dimension
cm : (int,int), optional
    center of mass (default: 0,0)
convert_to_float : boolean, optional
    whether to convert to floating point (default: True)

Returns **moment: float** :
    floating point number

`mahotas.`**`open`**(*f*, *Bc={3x3 cross}*, *out={np.empty_like(f)}*)

Morphological opening.

open creates the image y by the morphological opening of the image *f* by the structuring element *Bc*.

In the binary case, the opening by the structuring element *Bc* may be interpreted as the union of translations of *b* included in *f*. In the gray-scale case, there is a similar interpretation taking the functions umbra.

Parameters **f** : ndarray
    Gray-scale (uint8 or uint16) or binary image.
Bc : ndarray, optional
    Structuring element (default: 3x3 elementary cross).
out : ndarray, optional
    Output array

Returns **y** : ndarray

See Also:

`open` function

`mahotas.`**`otsu`**(*img*, *ignore_zeros=False*)

Calculate a threshold according to the Otsu method.

Parameters **img** : an image as a numpy array.

This should be of an unsigned integer type.

ignore_zeros : Boolean

whether to ignore zero-valued pixels (default: False)

Returns **T** : integer

the threshold

`mahotas.`**`rank_filter`**(*f*, *Bc*, *rank*, *mode='reflect'*, *cval=0.0*, *out=None*)

Rank filter. The value at `ranked[i,j]` will be the `rank``th largest in the neighbourhood defined by ``Bc`.

Parameters **f** : ndarray

input. Any dimension is supported

Bc : ndarray

Defines the neighbourhood. Must be explicitly passed, no default.

rank : integer

mode : {'reflect' [default], 'nearest', 'wrap', 'mirror', 'constant', 'ignore'}

How to handle borders

cval : double, optional

If *mode* is constant, which constant to use (default: 0.0)

out : ndarray, optional

Output array. Must have same shape and dtype as *f* as well as be C-contiguous.

Returns **ranked** : ndarray of same type and shape as `f`

ranked[i,j] is the ``rank``th value of the points in f close to (i,j)

See Also:

`median_filter` A special case of rank_filter

`mahotas.`**`rc`**(*img*, *ignore_zeros=False*)

Calculate a threshold according to the Riddler-Calvard method.

Parameters **img** : ndarray

Image of any type

ignore_zeros : boolean, optional

Whether to ignore zero valued pixels (default: False)

Returns **T** : float

threshold

`mahotas.`**`regmax`**(*f*, *Bc={3x3 cross}*, *out={np.empty(f.shape, bool)}*)

Regional maxima

Parameters **f** : ndarray

Bc : ndarray, optional

structuring element

out : ndarray, optional

Used for output. Must be Boolean ndarray of same size as *f*

Returns **filtered** : ndarray

boolean image of same size as f.

See Also:

`locmax` function Local maxima

`mahotas.`**`regmin`**(*f*, *Bc={3x3 cross}*, *out={np.empty(f.shape, bool)}*)

Regional minima

Parameters **f** : ndarray

Bc : ndarray, optional
structuring element

out : ndarray, optional
Used for output. Must be Boolean ndarray of same size as *f*

Returns **filtered** : ndarray
boolean image of same size as f.

See Also:

`locmin` function Local minima

`mahotas.`**`sobel`**(*img*, *just_filter=False*)

Compute edges using Sobel's algorithm

edges is a binary image of edges computed according to Sobel's algorithm.

This implementation is tuned to match MATLAB's implementation.

Parameters **img** : Any 2D-ndarray

just_filter : boolean, optional
If true, then return the result of filtering the image with the sobel filters, but do not threashold (default is False).

Returns **edges** : ndarray
Binary image of edges, unless *just_filter*, in which case it will be an array of floating point values.

`mahotas.`**`stretch`**(*img*, *arg0=None*, *arg1=None*, *dtype=<type 'numpy.uint8'>*)

img' = stretch(img, [dtype=np.uint8]) img' = stretch(img, max, [dtype=np.uint8]) img' = stretch(img, min, max, [dtype=np.uint8])

Contrast stretch the image to the range [0, max] (first form) or [min, max] (second form).

Parameters **img** : ndarray
input image. It is *not modified* by this function

min : integer, optional
minimum value for output [default: 0]

max : integer, optional
maximum value for output [default: 255]

dtype : dtype of output,optional
[default: np.uint8]

Returns **img': ndarray** :
resulting image. ndarray of same shape as *img* and type *dtype*.

`mahotas.`**`template_match`**(*f*, *template*, *mode='reflect'*, *cval=0.*, *out={np.empty_like(f)}*)

Match template.

The value at `match[i,j]` will be the difference (in squared euclidean terms), between *template* and a same sized window on *f* centered on that point.

Parameters **f** : ndarray
input. Any dimension is supported

template : ndarray
Template to match. Must be explicitly passed, no default.

mode : {'reflect' [default], 'nearest', 'wrap', 'mirror', 'constant', 'ignore'}
How to handle borders

cval : double, optional
If *mode* is constant, which constant to use (default: 0.0)

out : ndarray, optional
Output array. Must have same shape and dtype as *f* as well as be C-contiguous.

Returns **match** : ndarray of same type and shape as `f` match[i,j] is the squared euclidean distance between `f[i-s0:i+s0,j-s1:j+s1]` and `template` (for appropriately defined `s0` and `s1`).

`mahotas.`**`thin`**(*binimg*)

Skeletonisation by thinning

Parameters **binimg** : Binary input image

Returns **skel** : Skeletonised version of *binimg*

`mahotas.`**`wavelet_center`**(*f*, *border=0*, *dtype=float*, *cval=0.0*)

`fc` is a centered version of `f` with a shape that is composed of powers of 2.

Parameters **f** : ndarray

input image

border : int, optional

The border to use (default is no border)

dtype : type, optional

Type of `fc`

cval : float, optional

Which value to fill the border with (default is 0)

Returns **fc** : ndarray

See Also:

`wavelet_decenter` function Reverse function

`mahotas.`**`wavelet_decenter`**(*w*, *oshape*, *border=0*)

Undoes the effect of `wavelet_center`

Parameters **w** : ndarray

Wavelet array

oshape : tuple

Desired shape

border : int, optional

The desired border. This **must** be the same value as was used for `wavelet_center`

Returns **f** : ndarray

This will have shape `oshape`

See Also:

`wavelet_center` function Forward function

`mahotas.features.`**`haralick`**(*f*, *ignore_zeros=False*, *preserve_haralick_bug=False*, *compute_14th_feature=False*)

Compute Haralick texture features

Computes the Haralick texture features for the four 2-D directions or thirteen 3-D directions (depending on the dimensions of *f*).

Parameters **f** : ndarray of integer type

input image. 2-D and 3-D images are supported.

ignore_zeros : bool, optional

Whether to ignore zero pixels (default: False).

Returns **feats** : ndarray of np.double

A 4x13 or 4x14 feature vector (one row per direction) if *f* is 2D, 13x13 or 13x14 if it is 3D. The exact number of features depends on the value of `compute_14th_feature`

Other Parameters **preserve_haralick_bug** : bool, optional

whether to replicate Haralick's typo (default: False). You probably want to always set this to `False` unless you want to replicate someone else's wrong implementation.

compute_14th_feature : bool, optional whether to compute & return the 14-th feature

Notes

Haralick's paper has a typo in one of the equations. This function implements the correct feature unless *preserve_haralick_bug* is True. The only reason why you'd want the buggy behaviour is if you want to match another implementation.

References

Cite the following reference for these features:

```
@article{Haralick1973,
    author = {Haralick, Robert M. and Dinstein, Its'hak and Shanmugam, K.},
    journal = {Ieee Transactions On Systems Man And Cybernetics},
    number = {6},
    pages = {610--621},
    publisher = {IEEE},
    title = {Textural features for image classification},
    url = {http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4309314},
    volume = {3},
    year = {1973}
}
```

`mahotas.features.`**`lbp`**(*image*, *radius*, *points*, *ignore_zeros=False*)

Compute Linear Binary Patterns

The return value is a **histogram** of feature counts, where position `i` corresponds to the number of pixels that had code `i`. The codes are compressed so that impossible codes are not used. Therefore, this is the `i``th feature, not just the feature with binary code ``i`.

Parameters **image** : ndarray

input image (2-D numpy ndarray)

radius : number (integer or floating point)

radius (in pixels)

points : integer

nr of points to consider

ignore_zeros : boolean, optional

whether to ignore zeros (default: False)

Returns **features** : 1-D numpy ndarray

histogram of features. See above for a caveat on the interpretation of these.

`mahotas.features.`**`pftas`**(*img*, *T={mahotas.threshold.otsu(img)}*)

Compute parameter free Threshold Adjacency Statistics

TAS were presented by Hamilton et al. in "Fast automated cell phenotype image classification" (http://www.biomedcentral.com/1471-2105/8/110)

The current version is an adapted version which is free of parameters. The thresholding is done by using Otsu's algorithm (or can be pre-computed and passed in by setting *T*), the margin around the mean of pixels to be included is the standard deviation. This was first published by Coelho et al. in "Structured Literature Image Finder: Extracting Information from Text and Images in Biomedical Literature" (http://www.springerlink.com/content/60634778710577t0/)

Also returns a version computed on the negative of the binarisation defined by Hamilton et al.

Use tas() to get the original version of the features.

Parameters **img** : ndarray, 2D or 3D
input image
T : integer, optional
Threshold to use (default: compute with otsu)

Returns **values** : ndarray
A 1-D ndarray of feature values

`mahotas.features.`**`tas`**(*img*)

Compute Threshold Adjacency Statistics

TAS were presented by Hamilton et al. in “Fast automated cell phenotype image classification” (http://www.biomedcentral.com/1471-2105/8/110)

Also returns a version computed on the negative of the binarisation defined by Hamilton et al.

See also pftas() for a variation without any hardcoded parameters.

Parameters **img** : ndarray, 2D or 3D
input image

Returns **values** : ndarray
A 1-D ndarray of feature values

See Also:

`pftas` Parameter free TAS

`mahotas.features.`**`zernike`**(*im*, *degree*, *radius*, *cm={center_of_mass(im)}*)

`mahotas.features.`**`zernike_moments`**(*im*, *radius*, *degree=8*, *cm={center_of_mass(im)}*)

Zernike moments through `degree`

Returns a vector of absolute Zernike moments through `degree` for the image `im`.

Parameters **im** : 2-ndarray
input image
radius : integer
the maximum radius for the Zernike polynomials, in pixels
degree : integer, optional
Maximum degree to use (default: 8)
cm : pair of floats, optional
the centre of mass to use. By default, uses the image’s centre of mass.

Returns **zvalues** : 1-ndarray of floats
Zernike moments

CHAPTER 3

Indices and tables

- *genindex*
- *search*

Python Module Index

Python Module Index