Exploring image processing pipelines with scikit-image, joblib, ipywidgets and dash

A bag of tricks for processing images faster

Emmanuelle Gouillart

joint Unit CNRS/Saint-Gobain SVI
and the scikit-image team

@EGouillart
From images to science
A typical pipeline

- How to discover & select the different algorithms?
- How to iterate quickly towards a satisfying result?
- How to verify processing results?
A typical pipeline

- How to discover & select the different algorithms?
- How to iterate quickly towards a satisfying result?
- How to verify processing results?
Introducing scikit-image

A NumPy-ic image processing library for science

```python
>>> from skimage import io, filters
>>> camera_array = io.imread('camera_image.png')
>>> type(camera_array)
<type 'numpy.ndarray'>
>>> camera_array.dtype
dtype('uint8')
>>> filtered_array = filters.gaussian(camera_array, sigma=5)
>>> type(filtered_array)
<type 'numpy.ndarray'>
```

Submodules correspond to different tasks: I/O, filtering, segmentation...

Compatible with 2D and 3D images
Documentation at a glance: galleries of examples

General examples
General-purpose and introductory examples for the scikit.

Blob Detection
BRIEF binary descriptor
Canny edge detector

CENSURE feature detector
Circular and Elliptical Hough Transforms
Contour finding

Convex Hull
Corner detection
Dense DAISY feature

Navigation
Documentation Home
Previous topic
License
Next topic
Blob Detection
Contents
General examples
Longer examples and demonstrations
Versions
skimage dev
skimage 0.10.x
skimage 0.9.x
skimage 0.8.0
skimage 0.7.0
skimage 0.6
skimage 0.5
skimage 0.4
skimage 0.3
Getting started: finding documentation

Label image regions
This example shows how to segment an image with image labelling. The following steps are applied:

1. Thresholding with automatic Otsu method
2. Close small holes with binary closing
3. Remove artifacts touching image border
4. Measure image regions to filter small objects

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches

from skimage import data
from skimage.filter import threshold_otsu
from skimage.segmentation import clear_border
from skimage.morphology import label, closing, square
from skimage.measure import regionprops
from skimage.color import label2rgb

image = data.coins()[150:50, 50:50]

# apply threshold
thresh = threshold_otsu(image)
bw = closing(image > thresh, square(3))

# remove artifacts connected to image border
cleared = bw.copy()
clear_border(cleared)

# label image regions
label_image = label(cleared)
borders = np.logical_xor(bw, cleared)
label_image[borders] = -1
image_label_overlay = label2rgb(label_image, image=image)

fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(6, 6))
ax.imshow(image_label_overlay)

for region in regionprops(label_image):
    # skip small images
    if region.area < 100:
        continue

    # draw rectangle around segmented coins
    minr, minc, maxr, maxc = region.bbox
    rect = mpatches.Rectangle((minr, minc), maxr - minr, maxc - minc, 
                              fill=False, edgecolor='red', linewidth=2)
    ax.add_patch(rect)

plt.show()
```
Welcome to Sphinx-Gallery's documentation!

Sphinx extension for automatic generation of an example gallery. It is extracted from the scikit-learn project and aims to be an independent general purpose extension.
threshold_otsu

`skimage.filters.threshold_otsu(image, nbins=256)`

Return threshold value based on Otsu's method.

**Parameters:**
- `image` : (N, M) ndarray
  Grayscale input image.
- `nbins` : int, optional
  Number of bins used to calculate histogram. This value is ignored for integer arrays.

**Returns:**
- `threshold` : float
  Upper threshold value. All pixels with an intensity higher than this value are assumed to be foreground.

**Raises:**
- `ValueError`
  If image only contains a single grayscale value.

**Notes**
The input image must be grayscale.

**References**

**Examples**
```python
from skimage.data import camera
image = camera()
thresh = threshold_otsu(image)
binary = image <= thresh
```
Auto documenting your API with links to examples

noisy image  Canny filter, $\sigma = 1$  Canny filter, $\sigma = 3$

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

from skimage import feature

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

im = ndi.rotate(im, 15, mode='constant')
im = ndi.gaussian_filter(im, 4)
im += 0.2 * np.random.random(im.shape)

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

# display results
fig, (ax1, ax2, ax3) = plt.subplots(nrows=1, ncols=3, figsize=(8, 3), sharex=True, sharey=True)
```
Learning by yourself

filters.try_all_threshold

Original

Li

Minimum

Triangle

Isodata

Mean

Otsu

Yen
Convenience functions: Numpy operations as one-liners

```python
labels = measure.label(im)
sizes = np.bincount(labels.ravel())
sizes[0] = 0
keep_only_large = (sizes > 1000)[labels]
```
Convenience functions: Numpy operations as one-liners

```python
labels = measure.label(im)
sizes = np.bincount(labels.ravel())
sizes[0] = 0
keep_only_large = (sizes > 1000)[labels]
morphology.remove_small_objects(im))
```

clear_border, relabel_sequential, find_boundaries, ↔
join_segmentations
More interaction for faster discovery: widgets

```python
from ipywidgets import widgets

@widgets.interact(t=(50, 240))
def threshold(t):
    show(img > t)
```
More interaction for faster discovery: web applications made easy

https://dash.plot.ly/
More interaction for faster discovery: web applications made easy

```python
@app.callback(
    dash.dependencies.Output('image-seg', 'figure'),
    [dash.dependencies.Input('slider_min', 'value'),
     dash.dependencies.Input('slider_max', 'value')])

def update_figure(v_min, v_max):
    mask = np.zeros(img.shape, dtype=np.uint8)
    mask[img < v_min] = 1
    mask[img > v_max] = 2
    seg = segmentation.random_walker(img, mask, mode='←cg_mg')
    return {'data': [
        go.Heatmap(
            z=img, colorscale='Greys') ,
        go.Contour(
            z=seg, ncontours=1,
            contours=dict(start=1.5, end=1.5,
                           coloring='lines'),
            line=dict(width=3)
        )
    ]}
```
from joblib import Memory
memory = Memory(cachedir='./cachedir', verbose=0)

@memory.cache
def mem_label(x):
    return measure.label(x)

@memory.cache
def mem_threshold_otsu(x):
    return filters.threshold_otsu(x)

[...]
val = mem_threshold_otsu(dat)
objects = dat > val

median_dat = mem_median_filter(dat, 3)
val2 = mem_threshold_otsu(median_dat[objects])
liquid = median_dat > val2
segmentation_result = np.copy(objects).astype(np.uint8)
segmentation_result[liquid] = 2

aggregates = mem_binary_fill_holes(objects)
aggregates_ds = np.copy(aggregates[::4, ::4, ::4])
cores = mem_binary_erosion(aggregates_ds, np.ones((10, 10, 10)))
import numpy as np
from sklearn.externals.joblib import Parallel, delayed

def apply_parallel(func, data, *args, chunk=100, overlap=10, n_jobs=4,
                   **kwargs):
    """
    Apply a function in parallel to overlapping chunks of an array. joblib is used for parallel processing.
    """
    sh0 = data.shape[0]
    nb_chunks = sh0 // chunk
    end_chunk = sh0 % chunk
    arg_list = [data[max(0, i*chunk - overlap):
                   min((i+1)*chunk + overlap, sh0)]
                for i in range(0, nb_chunks)]

    if end_chunk > 0:
        arg_list.append(data[-end_chunk - overlap:]

    res_list = Parallel(n_jobs=n_jobs)(delayed(func)(sub_im, *args, **kwargs)
                                         for sub_im in arg_list)
    output_dtype = res_list[0].dtype
    out_data = np.empty(data.shape, dtype=output_dtype)

    for i in range(1, nb_chunks):
        out_data[i*chunk:(i+1)*chunk] = res_list[i][overlap:overlap+chunk]
    out_data[:chunk] = res_list[0][:-overlap]

    if end_chunk > 0:
        out_data[-end_chunk:] = res_list[-1][overlap:]

    return out_data
Experimental chunking and parallelization

Library comparison:
- `joblib`
- `dask`

Function names:
- `img_as_ubyte`
- `img_as_float32`
- `img_as_uint`
- `rescale_intensity`
- `binary_opening`
- `binary_closing`
- `black_tophat`
- `white_tophat`
- `top_hat`
- `morphological_gradient`
- `math Huff`
- `loosely_(binary erosion)`
- `loosely_(binary dilation)`

Graph illustrating apply_parallel timings with speed ratio (parallel library is x times faster) for various functions.
Synchronized matplotlib subplots

```python
fig, ax = plt.subplots(1, 3, sharex=True, sharey=True)
```
Synchronizing *mayavi* visualization modules

```python
mayavi_module.sync_trait('trait', other_module)
```
Conclusions

- Explore as much as possible
  Take advantage of documentation (maybe improve it!)

- Keep the pipeline interactive

- Check what you’re doing,
  use meaningful visualizations