

4. Basic Image processing: Filtering, scaling, thresholding

Almost all image processing pipelines start with some basic procedures like thresholding, scaling, or projecting a multi-dimensional image.

Let us import again all necessary packages:

```
In [1]: import numpy as np
import matplotlib.pyplot as plt
import skimage.io as io
from skimage.external.tifffile import TiffFile
```

Most filtering functions will come out from the filters module of scikit-image:

```
In [2]: import skimage.filters as skf
```

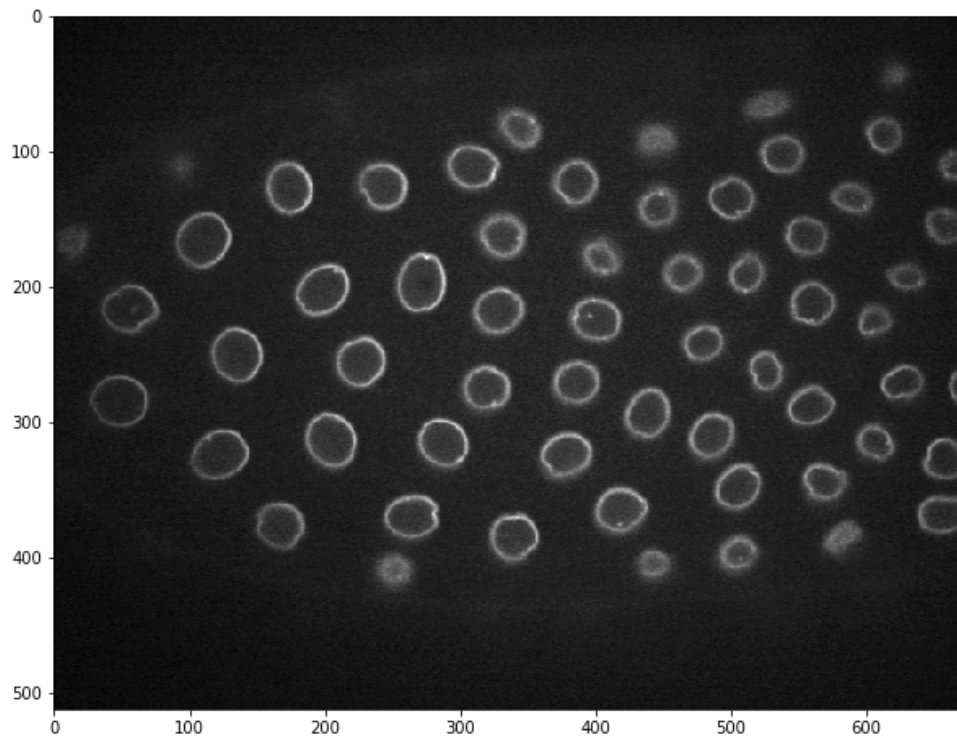
A specific region size/shape has often to be specified for filters. Those are defined in the morphology module:

```
In [3]: import skimage.morphology as skm
```

Additionally, this module offers a set of binary operators essential to operate on the masks resulting from segmentation.

We will start working on a single plane of the dataset seen in chapter [3 \(3-Image_import.ipynb\)](#)

```
In [4]: #load image
data = TiffFile('Data/30567/30567.tif')
image = data.pages[3].asarray()
#plot image
plt.figure(figsize=(10,10))
plt.imshow(image,cmap = 'gray');
```

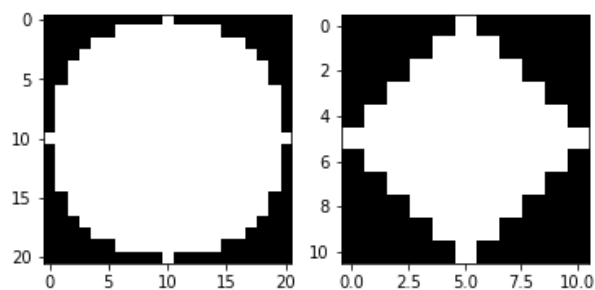


4.1 Filtering

A large set of filters are offered in scikit-image. Filtering is a local operation, where a value is calculated for each pixel and its surrounding region according to some function. For example a median filter of size 3, calculates for each pixel the median value of the 3x3 region around it.

Most filters take as input a specified region to consider for the calculation (e.g. 3x3 region). Those can be defined using the morphology module e.g.

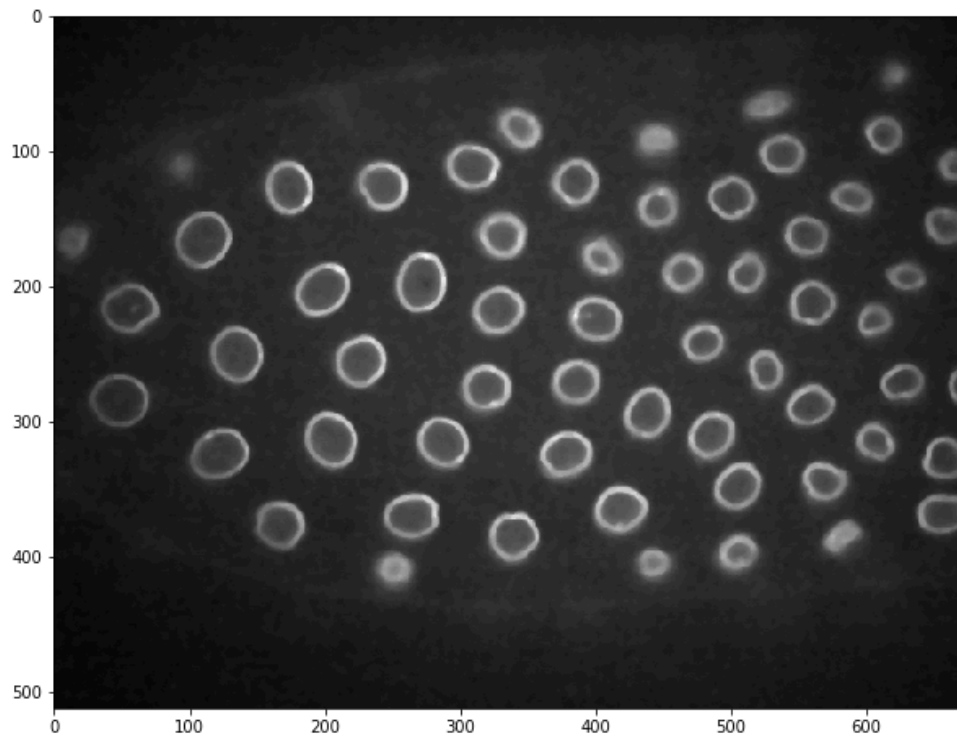
```
In [5]: disk = skm.disk(10)
diamond = skm.diamond(5)
plt.subplot(1,2,1)
plt.imshow(disk,cmap = 'gray')
plt.subplot(1,2,2)
plt.imshow(diamond,cmap = 'gray');
```



```
In [6]: image_mean = skf.median(image,selem=skm.disk(3)) # MZ: selem = selection element

/usr/local/lib/python3.5/dist-packages/skimage/filters/rank/generic.py:102: UserWarning: Bitdepth of 14 may result in bad rank filter performance due to large number of bins.
    "performance due to large number of bins." % bitdepth)
```

```
In [7]: plt.figure(figsize=(10,10))
plt.imshow(image_mean,cmap = 'gray');
```



Similar filters can be defined for a large range of operations: sum, min, max, mean etc.

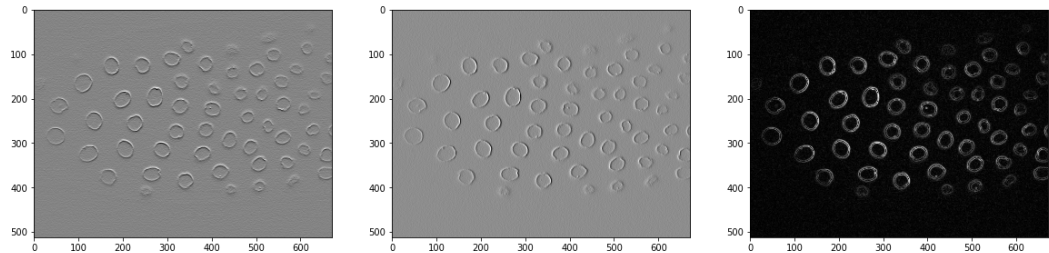
More specific filters are also provide in skimage. For example finding the gradient of intensity in an image can be done with a Sobel filter. Here for horizontal, vertical and their sum:

```
In [8]: image_gradienth = skf.sobel_h(image) # MZ: sobel filter, applied horizontally
image_gradientv = skf.sobel_v(image) # MZ: same filter, applied vertically
image_gradient = np.sqrt(image_gradientv**2+image_gradienth**2) # combine both
```

```
In [9]: plt.figure(figsize=(20,10))
plt.subplot(1,3,1)
plt.imshow(image_gradienth,cmap = 'gray')
plt.subplot(1,3,2)
plt.imshow(image_gradientv,cmap = 'gray')
plt.subplot(1,3,3)
plt.imshow(image_gradient,cmap = 'gray')

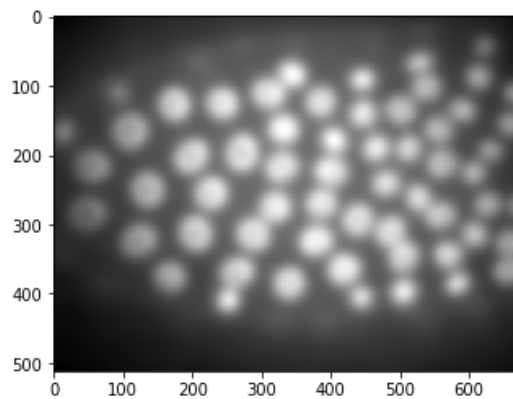
# MZ: highlight the edges horizontally (1) and vertically (2)
# (3) combined, the edges are highlighted
```

Out[9]: <matplotlib.image.AxesImage at 0x7f7f96116a90>



Finally, some functions can be used to filter the image, and one can pass function parameters to the filter. For example to filter with a Gaussian of large standard deviation $\sigma = 10$:

```
In [10]: image_gauss = skf.gaussian(image, sigma=10)#, preserve_range=True)
# MZ: Gaussian with really large sigma (e.g. highlight the nuclei)
# MZ: to just filter noise: use much smaller sigma
plt.imshow(image_gauss,cmap = 'gray');
# Gaussian filter automatically re-scales the image between the 0 and 1
```



A warning regarding filters: some filters can change the type and even the range of intensity of the image. Typically the gaussian filter used above rescales the image between 0 and 1:

```
In [11]: print(image.dtype)
print(image.max())
print(image.min())
```

```
uint16
20303
2827
```

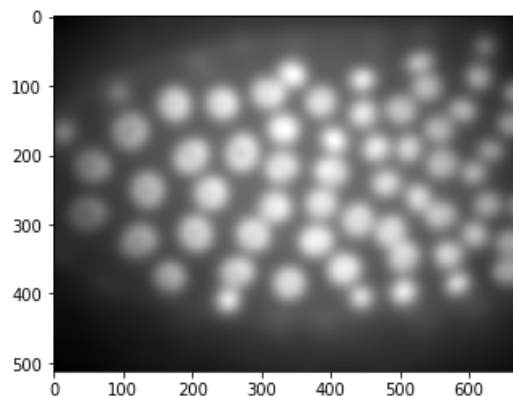
```
In [12]: print(image_gauss.dtype)
print(image_gauss.max())
print(image_gauss.min())

float64
0.12531917375072713
0.054386287321711344
```

In many cases, one can specify whether the original range should be preserved:

```
In [13]: image_gauss_preserve = skf.gaussian(image, sigma=10, preserve_range=True)
# MZ: use preserve_range, so that values are not re-scaled
plt.imshow(image_gauss_preserve, cmap = 'gray');
print(image_gauss_preserve.dtype)
print(image_gauss_preserve.max())
print(image_gauss_preserve.min())

float64
8212.792051753902
3564.2053396283527
```



4.2 Intensity re-scaling

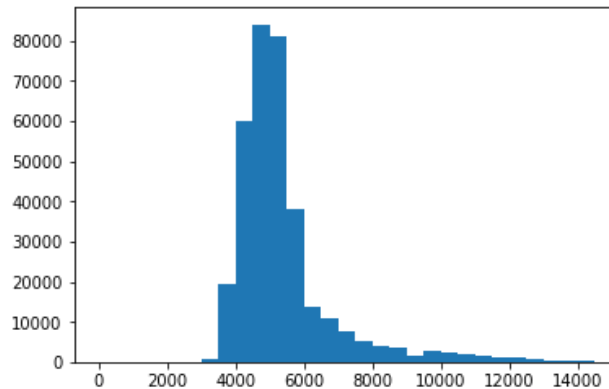
A very common operation to do in an image processing pipeline, is to rescale the intensity of images. The reason can be diverse: for example, one might want to remove an offset added to each pixel by the camera, or one might want to homogenize multiple images with slightly varying exposures.

The simplest thing to do is to rescale from min to max in the range 0-1. To create a histogram of the pixel values of an image, we first have to "flatten" the array, i.e. remove the dimensions, so that the plotting function doesn't believe we have a series of separate measurements.

```
In [14]: np.ravel(image).shape
# MZ convert 2D to 1D array -> flatten to have 1 big list of pixels
# (needed to draw one single histogram for all values)

Out[14]: (344064,)
```

```
In [15]: plt.hist(np.ravel(image), bins = np.arange(0,15000,500))
plt.show()
print("min val: "+ str(np.min(image)))
print("max val: "+ str(np.max(image)))
```



```
min val: 2827
max val: 20303
```

```
In [16]: image_minmax = (image-image.min())/(image.max()-image.min())
image_minmax[image_minmax>1] = 1
```

One problem that might emerge is that a few pixels might be affected by rare noise events that give them abnormal values. One way to remedy that is to use a small median filter in order to suppress those aberrant values:

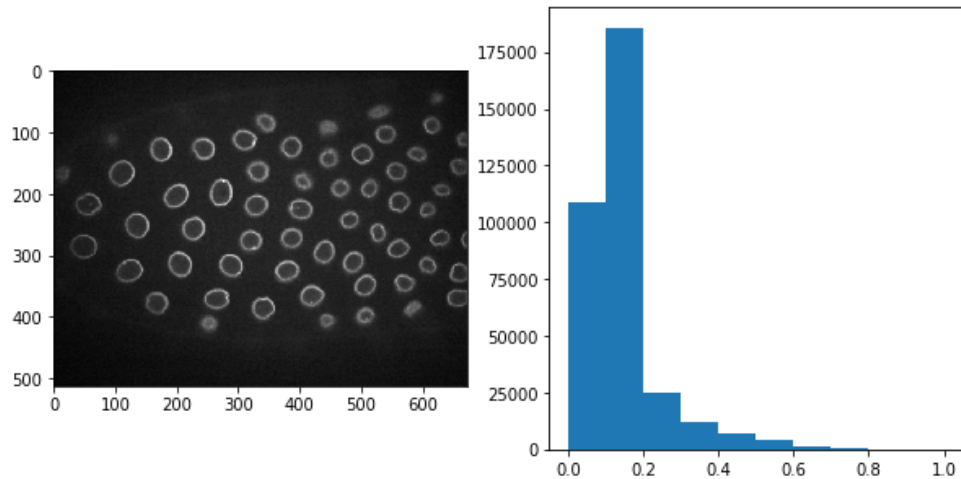
```
In [17]: image_median = skf.median(image,selem=np.ones((2,2)))
print("min val: "+ str(np.min(image_median)))
print("max val: "+ str(np.max(image_median)))

image_median_rescale = (image_median-image_median.min())/(image_median.m
ax()-image_median.min())
image_median_rescale[image_minmax>1] = 1
```

```
/usr/local/lib/python3.5/dist-packages/skimage/filters/rank/generic.py:10
2: UserWarning: Bitdepth of 14 may result in bad rank filter performance
due to large number of bins.
"performance due to large number of bins." % bitdepth)
```

```
min val: 3084
max val: 20046
```

```
In [18]: plt.figure(figsize=(10,5))
plt.subplot(1,2,1)
plt.imshow(image_median_rescale,cmap = 'gray')
plt.subplot(1,2,2)
plt.hist(np.ravel(image_median_rescale))#, bins = np.arange(0,15000,50
0))
plt.show()
```



Note that the `skimage.exposure` module offers several functions to adjust the image intensity distribution.

4.3 Thresholding

Another common operation is to try isolating regions of an image based on their intensity by using an intensity threshold: one can create a `maks` object where all values larger than a threshold are 1 and the other 0. It is usually better to use a smoothed version of the image (e.g. median or gaussian filtering) to avoid including noisy pixels in the `maks`.

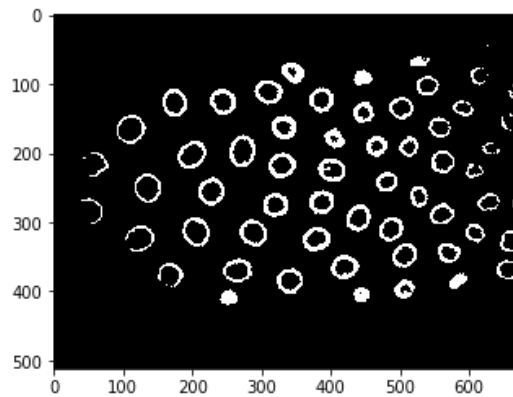
Let us imagine that we want to isolate the nuclei in our current image. To do that we can try to use their bright contour. Based on the intensity histogram, let's try to pick a threshold manually:

```
In [19]: # MZ: thresholded image to only keep values above a given threshold

threshold_manual = 8000

#create a mask using a logical operation
image_threshold = image_median > threshold_manual # MZ: create a boolean array

plt.imshow(image_threshold, cmap = 'gray')
plt.show()
```



Instead of picking manually the threshold, one can use one of the many automatic methods available in skimage,

```
In [20]: image_otsu_threshold = skf.threshold_otsu(image_median)
```

```
In [21]: image_otsu_threshold
```

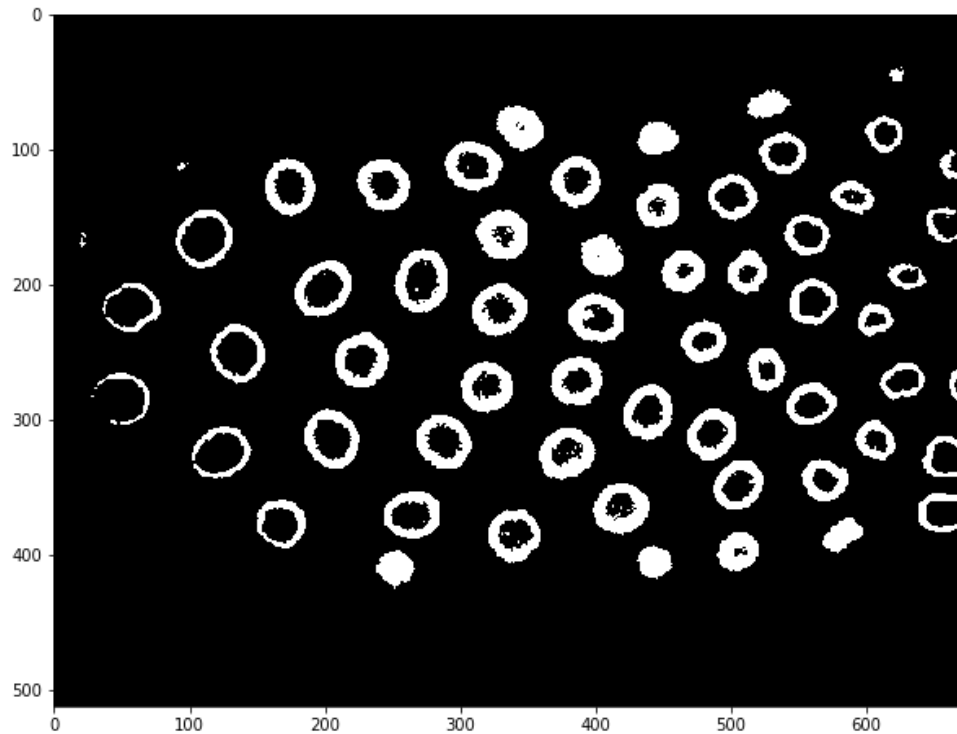
```
Out[21]: 7196
```

```
In [22]: image_otsu_threshold = skf.threshold_otsu(image_median)
print(image_otsu_threshold)
image_li_threshold = skf.threshold_li(image_median)
print(image_li_threshold)
```

```
7196
6416.599708799512
```

Knowing that threshold value we can create a binary image setting all pixels higher than the threshold to 1.


```
In [23]: image_otsu = image_median > image_otsu_threshold
plt.figure(figsize=(10,10))
plt.imshow(image_otsu, cmap = 'gray')
plt.show()
```



Since the illumination is uneven across the image, all standard thresholding methods are going to fail in some region of the image. What we could try to do instead is using a local thresholding, by repeating a standard thresholding method in sub-regions of the image:

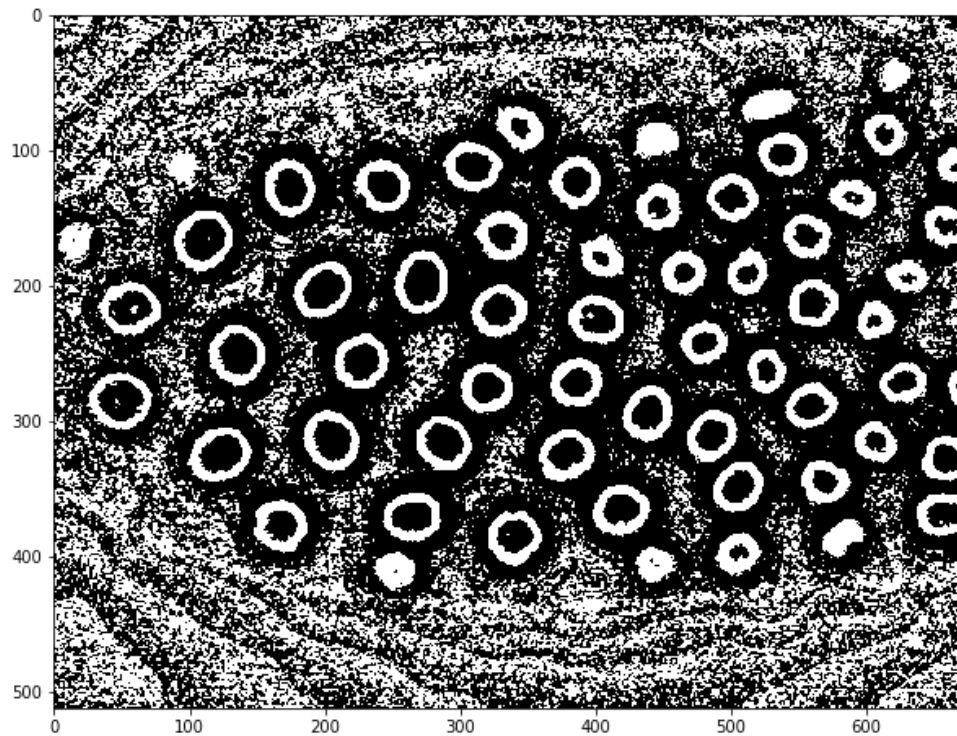
```
In [24]: image_local_threshold = skf.threshold_local(image_median, block_size=51)
```

```
In [25]: image_local_threshold.shape
```

```
Out[25]: (512, 672)
```

```
In [26]: image_local_threshold = skf.threshold_local(image_median, block_size=51)
image_local = image_median > image_local_threshold
```

```
In [27]: plt.figure(figsize=(10,10))  
plt.imshow(image_local, cmap = 'gray')  
plt.show()
```



We see that now each contour of the nuclei is recovered much better, however there is a lot of spurious background signal.

4.4 Note on higher-dimensional cases

Some functions of scikit-image are only designed for 2D images, and will generate an error message when used with 3D images. An alternative package to use in those cases is `scipy` and specifically `scipy.ndimage` and `scipy.filtering`