Image Processing in Python

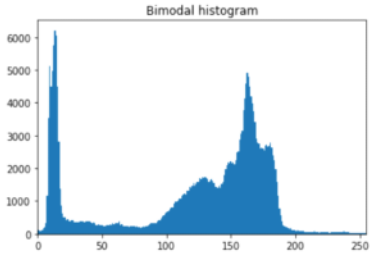
Friday, 21 August 2020 2:51 PM

Introducing Image Processing and scikit-image

What is image processing?

Operations to on images and videos to:

- Enhance an image
- Extract useful information
- Analyze it and make decisions



Applications

- Medical image analysis
- Artificial intelligence
- Image restoration and enhancement
- Geospatial computing
- Surveillance
- Robotic vision
- Automotive safety
- And many more...

Purposes

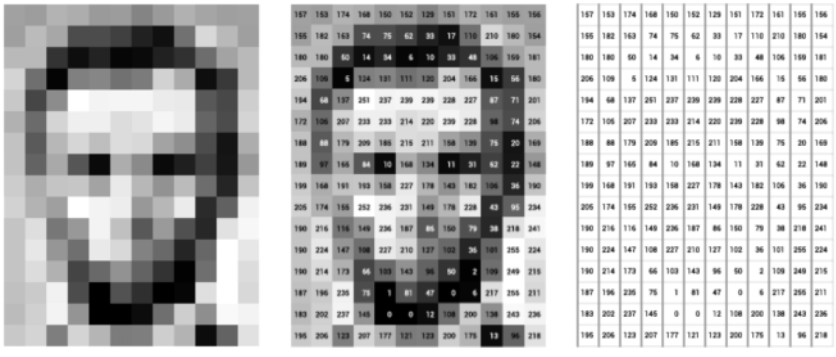
1. Visualization:
   - Objects that are not visible
2. Image sharpening and restoration
   - A better image
3. Image retrieval
   - Seek for the image of interest
4. Measurement of pattern
   - Measures various objects
5. Image Recognition
   - Distinguish objects in an image

Intro to scikit-image

- Easy to use
- Makes use of Machine Learning
- Out of the box complex algorithms

scikit-image
image processing in python

What is an image?

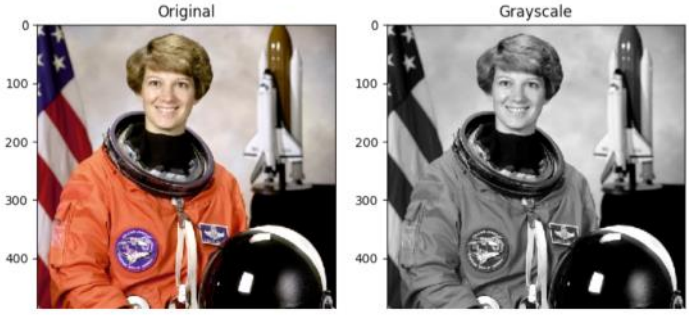


Images in scikit-image

```
from skimage import data

rocket_image = data.rocket()
```

RGB vs Grayscale

```
from skimage import color
grayscale = color.rgb2gray(original)

rgb = color.gray2rgb(grayscale)
```



Visualizing images in the course

Don't worry about Matplotlib!

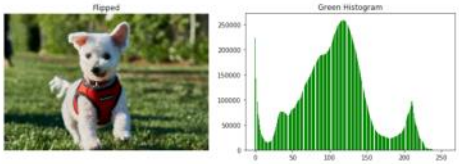
```
def show_image(image, title='Image', cmap_type='gray'):
    plt.imshow(image, cmap=cmap_type)
    plt.title(title)
    plt.axis('off')
    plt.show()
```

Visualizing images in the course

```
from skimage import color
grayscale = color.rgb2gray(original)

show_image(grayscale, "Grayscale")
```

NumPy for images

- Fundamentals of image processing techniques
  - Flipping
  - Extract and analyze features

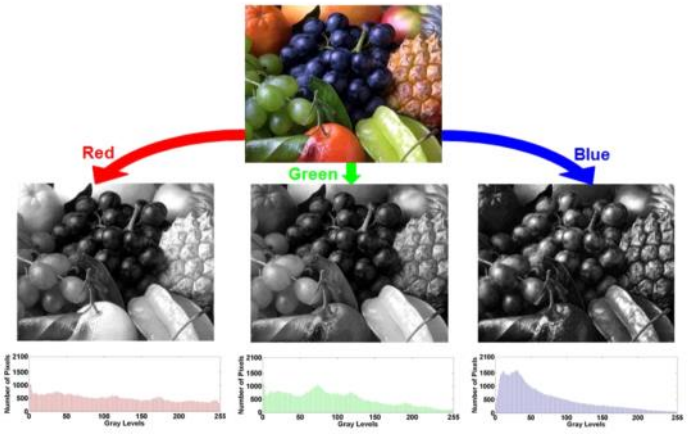


Images as NdArrays

```
# Loading the image using Matplotlib
madrid_image = plt.imread('/madrid.jpeg')

type(madrid_image)
```

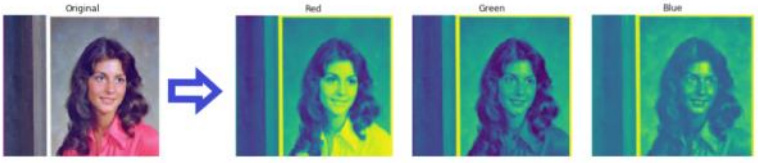
```
<class 'numpy.ndarray'>
```

Colors with NumPy

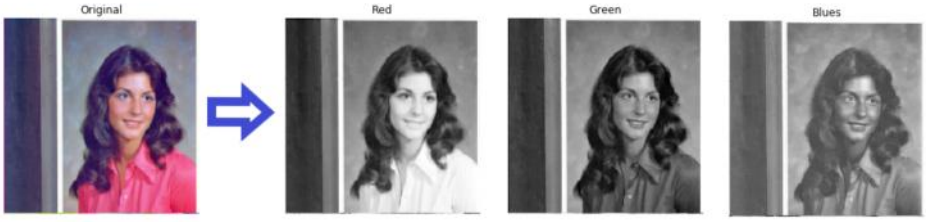


Colors with NumPy

```
# Obtaining the red values of the image
red = image[:, :, 0]

# Obtaining the green values of the image
green = image[:, :, 1]

# Obtaining the blue values of the image
blue = image[:, :, 2]
```



Colors with NumPy




```
plt.imshow(red, cmap="gray")
plt.title('Red')
plt.axis('off')
plt.show()
```

Shapes

```
# Accessing the shape of the image
madrid_image.shape
```

```
(426, 640, 3)
```

Sizes

```
# Accessing the shape of the image
madrid_image.size
```

```
817920
```

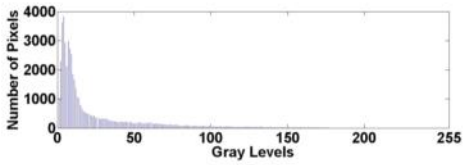
Flipping images: vertically

```
# Flip the image in up direction
vertically_flipped = np.flipud(madrid_image)

show_image(vertically_flipped, 'Vertically flipped image')
```

Flipping images: horizontally

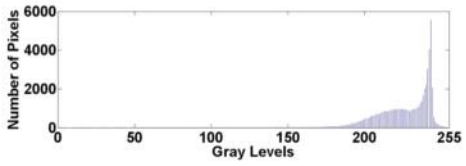
```
# Flip the image in left direction
horizontally_flipped = np.fliplr(madrid_image)

show_image(horizontally_flipped, 'Horizontally flipped image')
```

What is a histogram?



(a)

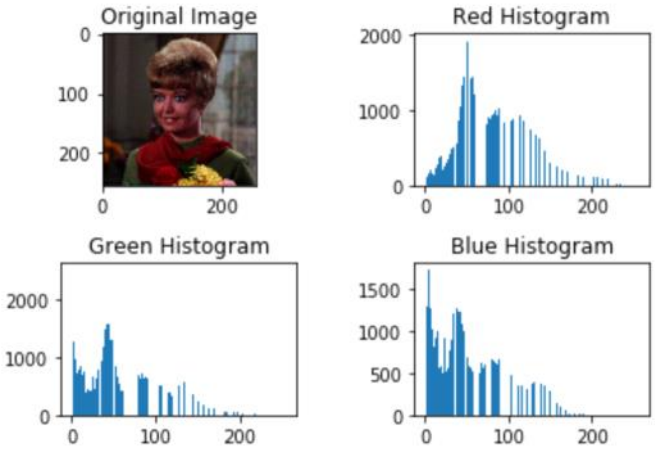
(b)



(a)

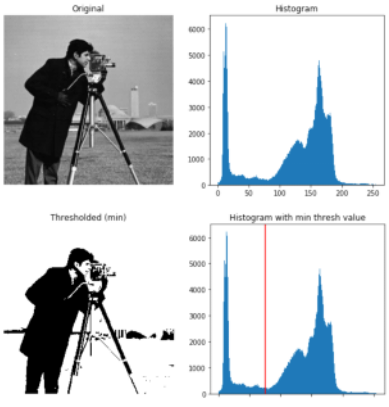
(b)

Color histograms

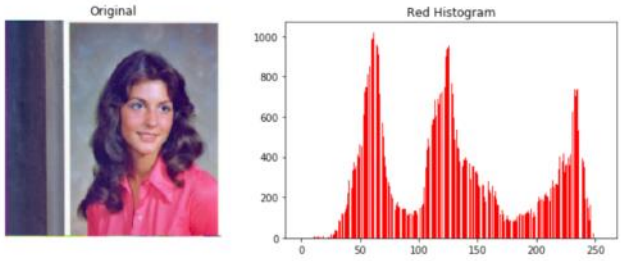


Applications of histograms

- Analysis
- Thresholding
- Brightness and contrast
- Equalize an image



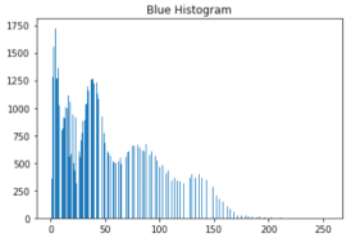
Histograms in Matplotlib




```
# Red color of the image
red = image[:, :, 0]

# Obtain the red histogram
plt.hist(red.ravel(), bins=256)
```

Visualizing histograms with Matplotlib

```
blue = image[:, :, 2]

plt.hist(blue.ravel(), bins=256)
plt.title('Blue Histogram')
plt.show()
```



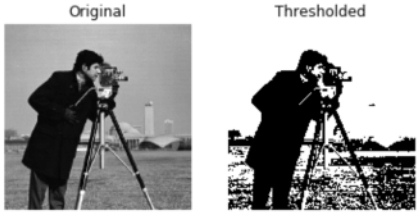
Thresholding

Partitioning an image into a foreground and background

By making it **black and white**

We do so by setting each pixel to:

- 255 (white) if pixel > thresh value
- 0 (black) if pixel < thresh value



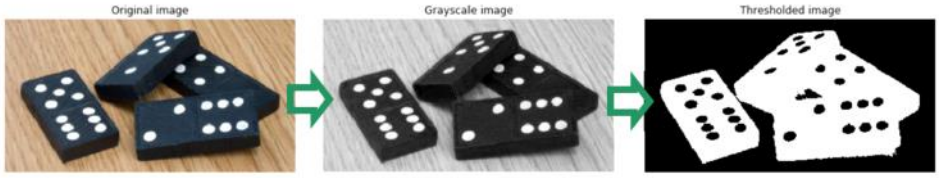
Thresholding

Simplest method of image segmentation

- Isolate objects
  - Object detection
  - Face detection
  - Etc.

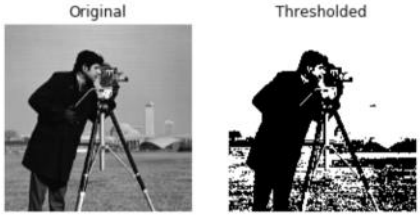
Thresholding

Only from **grayscale** images



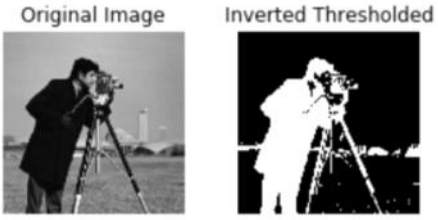
Apply it

```
# Obtain the optimal threshold value
thresh = 127

# Apply thresholding to the image
binary = image > thresh

# Show the original and thresholded
show_image(image, 'Original')
show_image(binary, 'Thresholded')
```



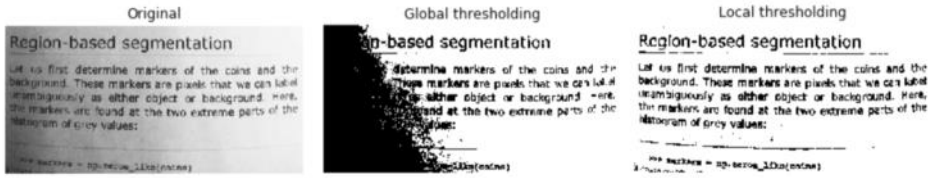
Inverted thresholding

```
# Obtain the optimal threshold value
thresh = 127

# Apply thresholding to the image
inverted_binary = image <= thresh

# Show the original and thresholded
show_image(image, 'Original')
show_image(inverted_binary,
           'Inverted thresholded')
```



Categories

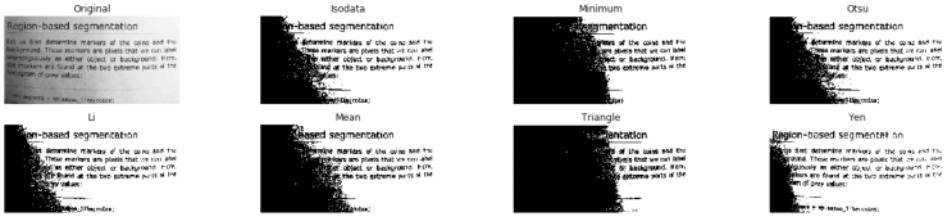
- **Global or histogram based:** good for uniform backgrounds
- **Local or adaptive:** for uneven background illumination



Try more thresholding algorithms

```
from skimage.filters import try_all_threshold

# Obtain all the resulting images
fig, ax = try_all_threshold(image, verbose=False)

# Showing resulting plots
show_plot(fig, ax)
```

Try more thresholding algorithms



Optimal thresh value

Global

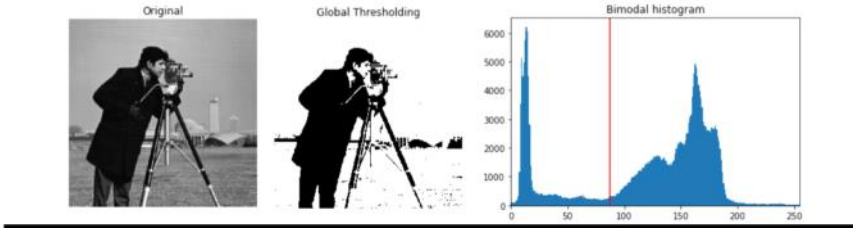
Uniform background

```
# Import the otsu threshold function
from skimage.filters import threshold_otsu

# Obtain the optimal threshold value
thresh = threshold_otsu(image)

# Apply thresholding to the image
binary_global = image > thresh
```

Optimal thresh value

Global

```
# Show the original and binarized image
show_image(image, 'Original')
show_image(binary_global, 'Global thresholding')
```



Optimal thresh value

Local

Uneven background

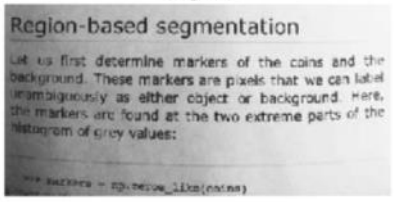
```
# Import the local threshold function
from skimage.filters import threshold_local

# Set the block size to 35
block_size = 35

# Obtain the optimal local thresholding
local_thresh = threshold_local(text_image, block_size, offset=10)

# Apply local thresholding and obtain the binary image
binary_local = text_image > local_thresh
```

Optimal thresh value

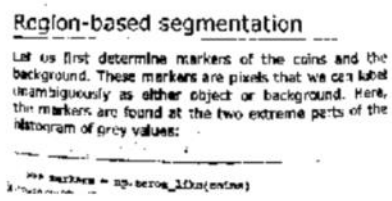
Local

```
# Show the original and binarized image
show_image(image, 'Original')
show_image(binary_local, 'Local thresholding')
```

Original



Local thresholding

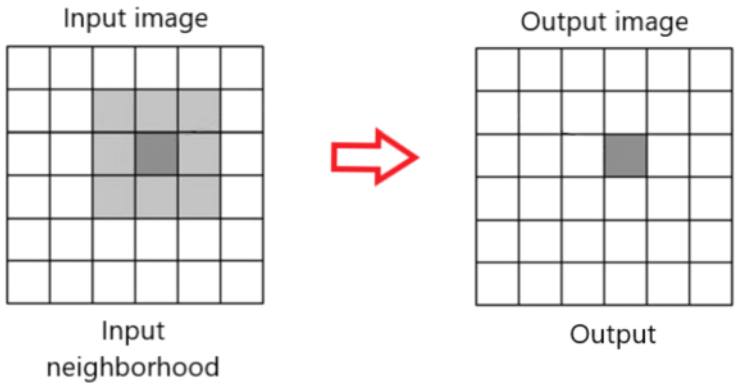


Filters, Contrast, Transformation and Morphology

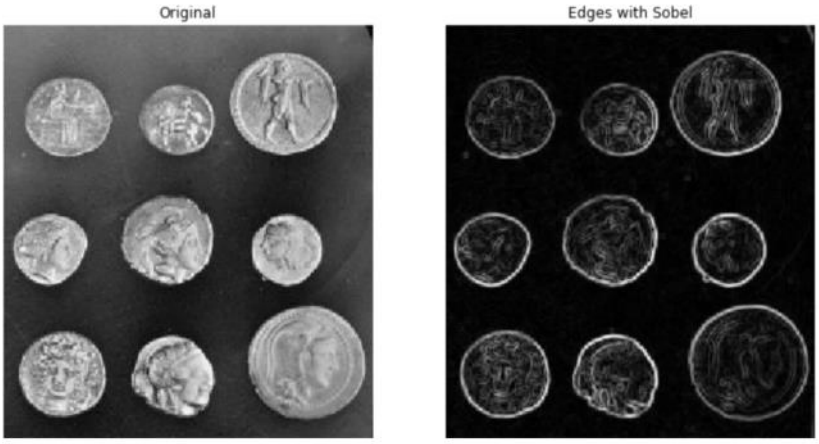
Filters

- Enhancing an image
- Emphasize or remove features
- Smoothing
- Sharpening
- Edge detection

Neighborhoods



Edge detection



Edge detection

Edge detection

Sobel

```
# Import module and function
from skimage.filters import sobel


# Apply edge detection filter
edge_sobel = sobel(image_coins)


# Show original and resulting image to compare
plot_comparison(image_coins, edge_sobel, "Edge with Sobel")
```

Comparing plots

```
def plot_comparison(original, filtered, title_filtered):

    fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(8, 6), sharex=True,
                                   sharey=True)
    ax1.imshow(original, cmap=plt.cm.gray)
    ax1.set_title('original')
    ax1.axis('off')
    ax2.imshow(filtered, cmap=plt.cm.gray)
    ax2.set_title(title_filtered)
    ax2.axis('off')
```

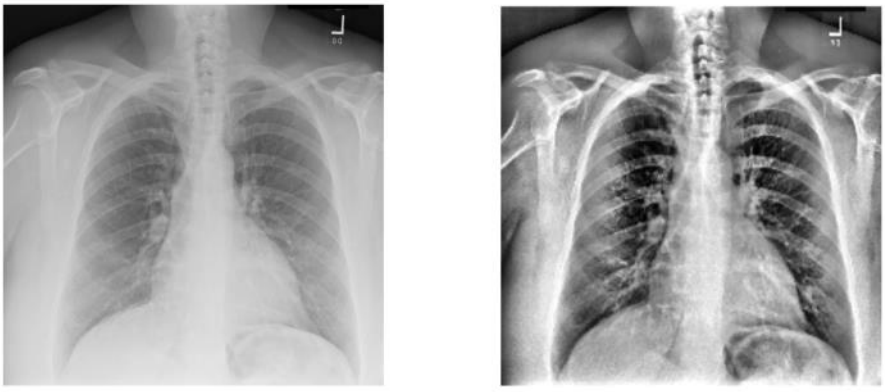
Gaussian smoothing

Gaussian smoothing

```
# Import the module and function
from skimage.filters import gaussian

# Apply edge detection filter
gaussian_image = gaussian(amsterdam_pic, multichannel=True)

# Show original and resulting image to compare
plot_comparison(amsterdam_pic, gaussian_image, "Blurred with Gaussian filter")
```

Gaussian smoothing

Contrast enhancement

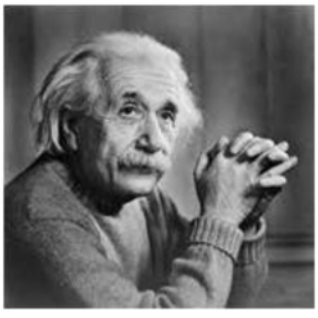


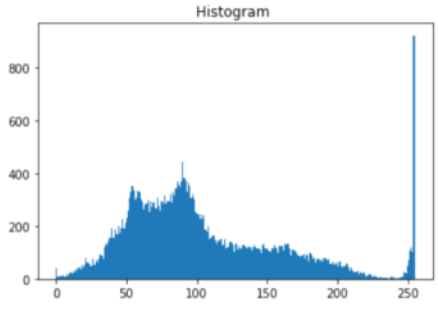
When we improve the contrast,the details become more visible.

The contrast of an image can be seen as the measure of its dynamic range, or the "spread" of its histogram.

Contrast

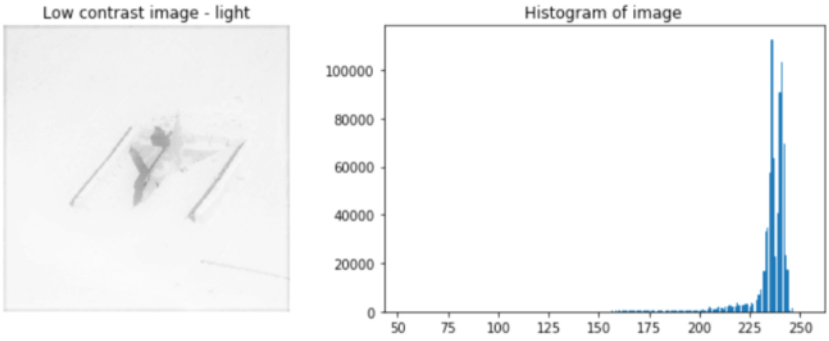
Histograms for contrast enhancement





The contrast is the difference between the maximum and minimum pixel intensity in the image.

Contrast



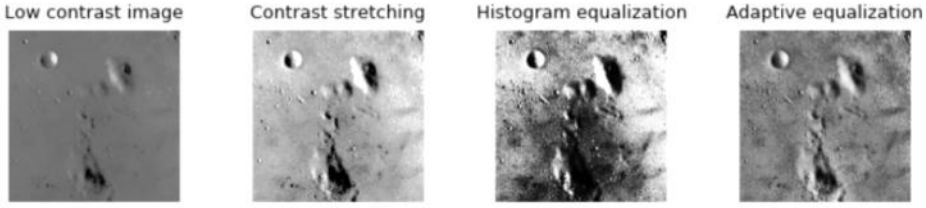
An image of low contrast has small difference between its dark and light pixel values.

Enhance contrast

- Contrast stretching
- Histogram equalization

Types

- Histogram equalization
- Adaptive histogram equalization
- Contrast Limited Adaptive Histogram Equalization (CLAHE)

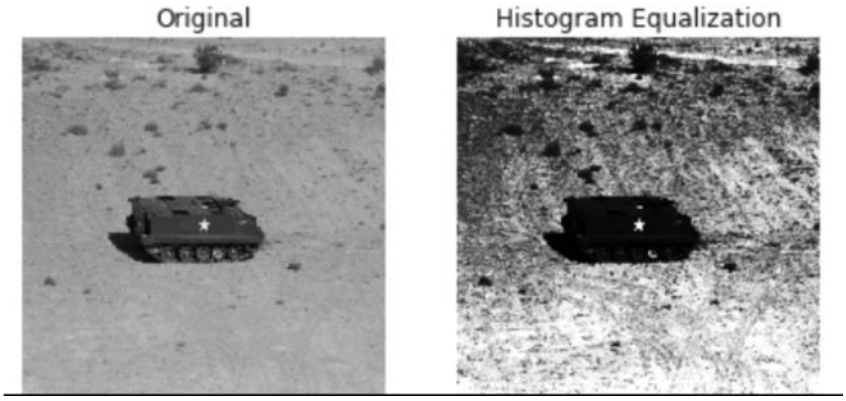


Histogram equalization

Histogram equalization

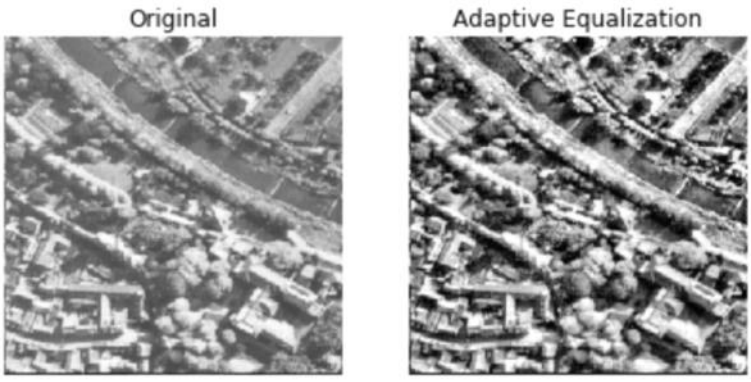
```
from skimage import exposure

# Obtain the equalized image
image_eq = exposure.equalize_hist(image)

# Show original and result
show_image(image, 'Original')
show_image(image_eq, 'Histogram equalized')
```

Histogram equalization



Adaptive Equalization

- Contrastive Limited Adaptive Histogram Equalization



Contrastive Limited Adaptive Equalization

CLAHE in scikit-image

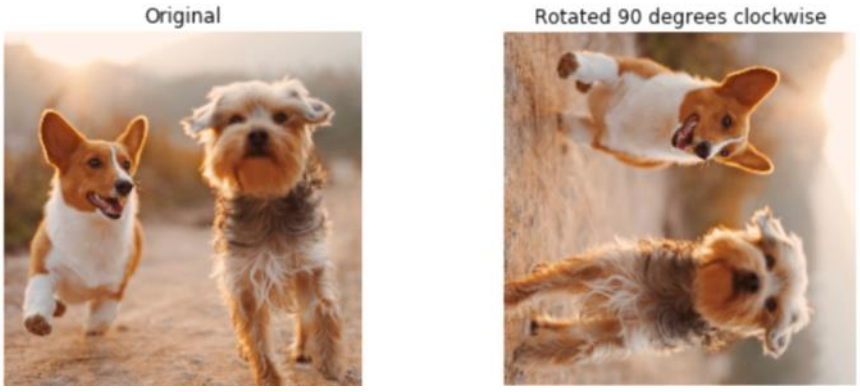
```
from skimage import exposure

# Apply adaptive Equalization
image_adapteq = exposure.equalize_adapthist(image, clip_limit=0.03)

# Show original and result
show_image(image, 'Original')
show_image(image_adapteq, 'Adaptive equalized')
```

This clipping limit, is normalized between 0 and 1 (higher values give more contrast)

Why transform images?

- Preparing images for classification Machine Learning models
- Optimization and compression of images
- Save images with same proportion

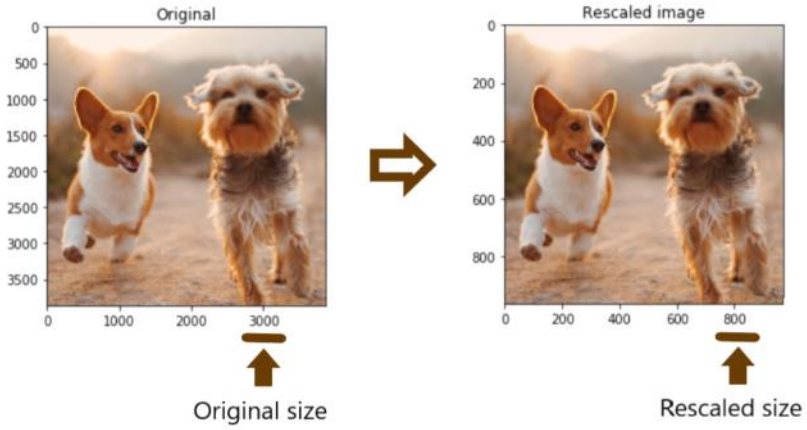
Rotating



Rotating clockwise

```
from skimage.transform import rotate

# Rotate the image 90 degrees clockwise
image_rotated = rotate(image, -90)

show_image(image_rotated, 'Original')
show_image(image_rotated, 'Rotated 90 degrees clockwise')
```

Rescaling

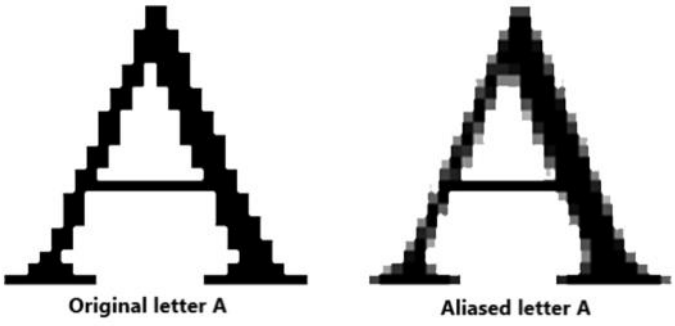


Rescaling

Downgrading

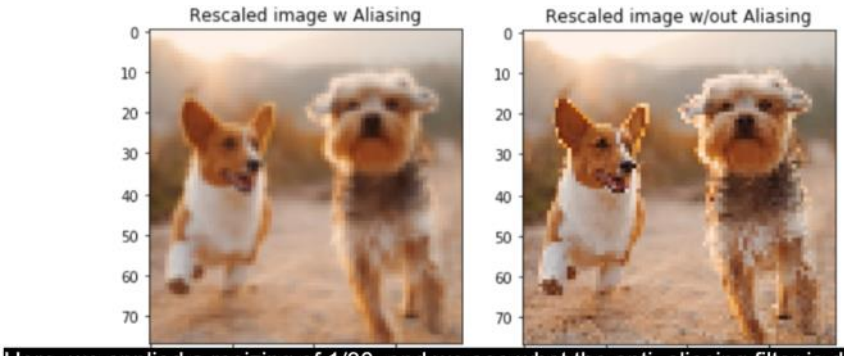
```
from skimage.transform import rescale

# Rescale the image to be 4 times smaller
image_rescaled = rescale(image, 1/4, anti_aliasing=True, multichannel=True)

show_image(image, 'Original image')
show_image(image_rescaled, 'Rescaled image')
```

Aliasing in digital images



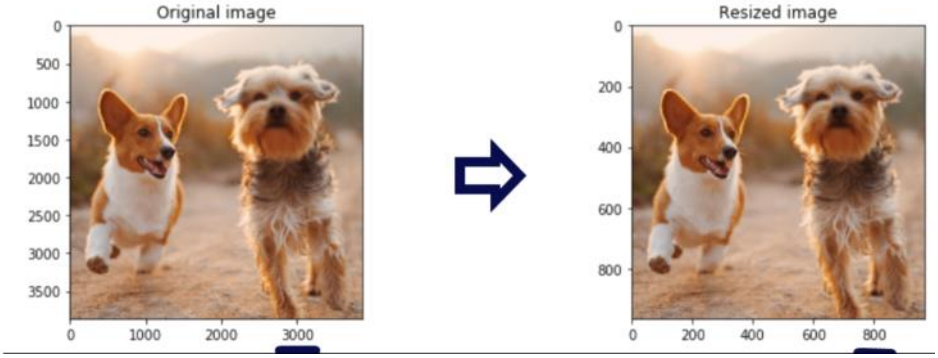
Aliasing makes the image look like it has waves or ripples radiating from a certain portion.

Aliasing in digital images



Here, we applied a resizing of 1/30, and we see what the anti_aliasing filter is doing to the image when is set.

Resizing

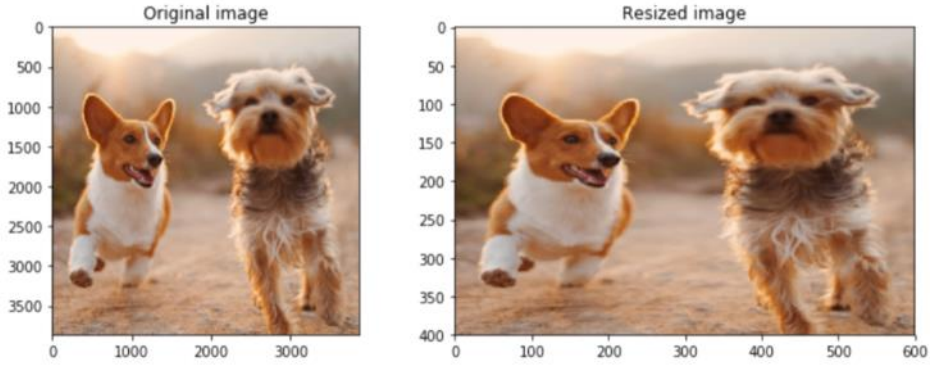


The same purpose as rescale, but allows to specify an output image shape instead of a scaling factor.

Resizing

```
from skimage.transform import resize

# Height and width to resize
height = 400
width = 500

# Resize image
image_resized = resize(image, (height, width), anti_aliasing=True)

# Show the original and resulting images
show_image(image, 'Original image')
show_image(image_resized, 'Resized image')
```

Resizing

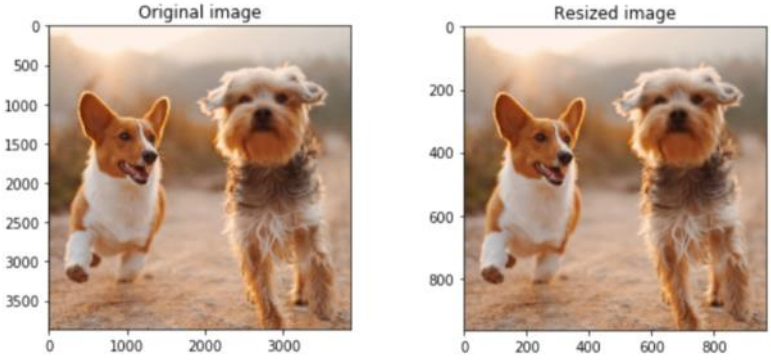


We can see how the image has been resized to a height of 400 and a width of 600.

Resizing proportionally

```
from skimage.transform import resize

# Set proportional height so its 4 times its size
height = image.shape[0] / 4
width = image.shape[1] / 4

# Resize image
image_resized = resize(image, (height, width), anti_aliasing=True)

show_image(image_resized, 'Resized image')
```

Resizing proportionally



We obtain a good looking and proportionally accurate resizing.

Binary images

original

Thresholded image

Binary regions produced by simple thresholding can be distorted by noise and texture, as we can see in the image.

Morphological filtering

- Better for binary images
- Can extend for grayscale

Binary image

Grayscale

Morphological operations

- Dilation
- Erosion

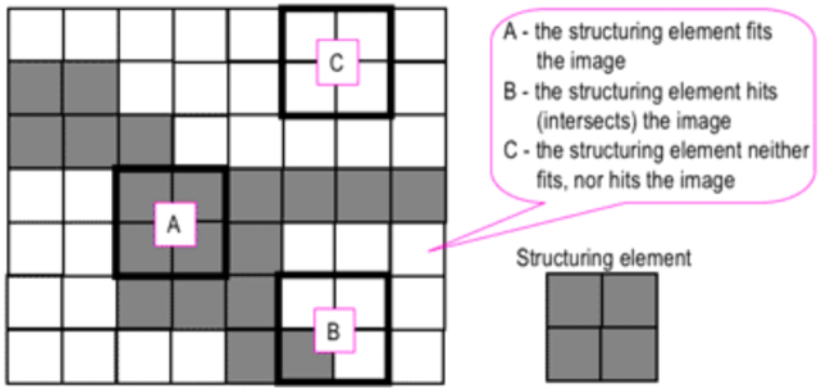
Original

Dilated

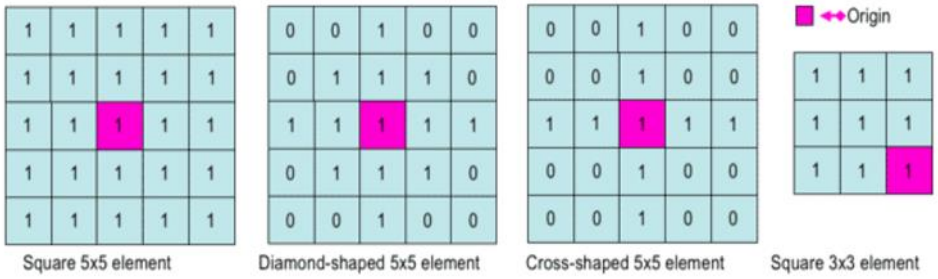
Original

Eroded

Structuring element



Structuring element



The pink cell is the center or origin of the structuring element.

Shapes in scikit-image

```
from skimage import morphology

square = morphology.square(4)
```

```
[[1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]]
```

```
rectangle = morphology.rectangle(4, 2)
```

```
[[1 1]
 [1 1]
 [1 1]
 [1 1]]
```

Erosion in scikit-image

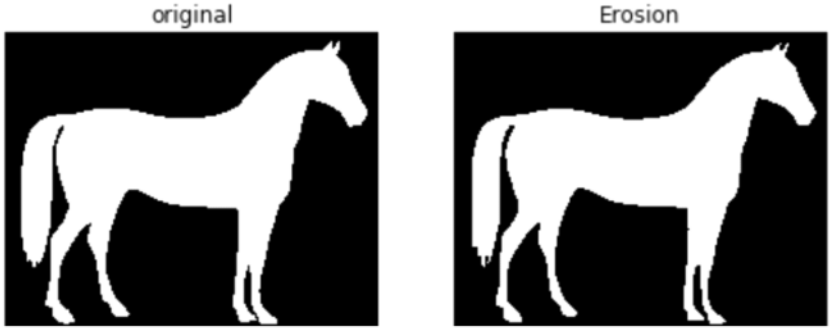
```
from skimage import morphology

# Set structuring element to the rectangular-shaped
selem = rectangle(12,6)

# Obtain the erosed image with binary erosion
eroded_image = morphology.binary_erosion(image_horse, selem=selem)
```

Erosion in scikit-image

```
# Show result
plot_comparison(image_horse, eroded_image, 'Erosion')
```

Binary erosion with default selem

```
# Binary erosion with default selem
eroded_image = morphology.binary_erosion(image_horse)
```



Dilation in scikit-image

```
from skimage import morphology

# Obtain dilated image, using binary dilation
dilated_image = morphology.binary_dilation(image_horse)

# See results
plot_comparison(image_horse, dilated_image, 'Erosion')
```

Dilation in scikit-image

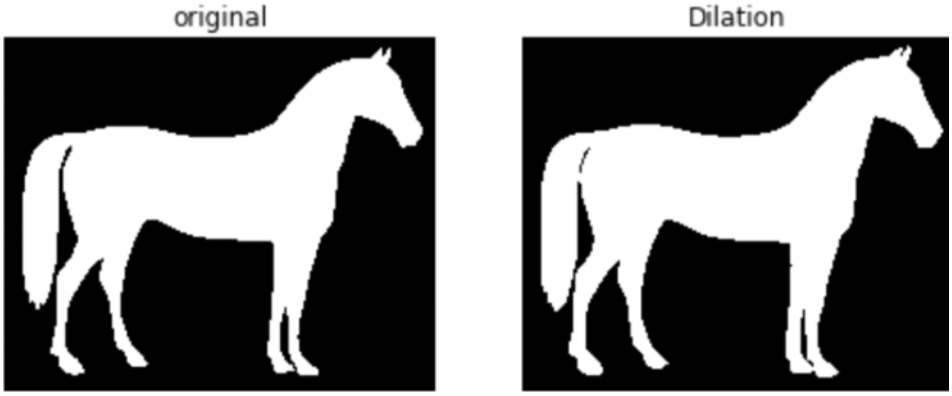


Image restoration, Noise, Segmentation and Contours

Restore an image

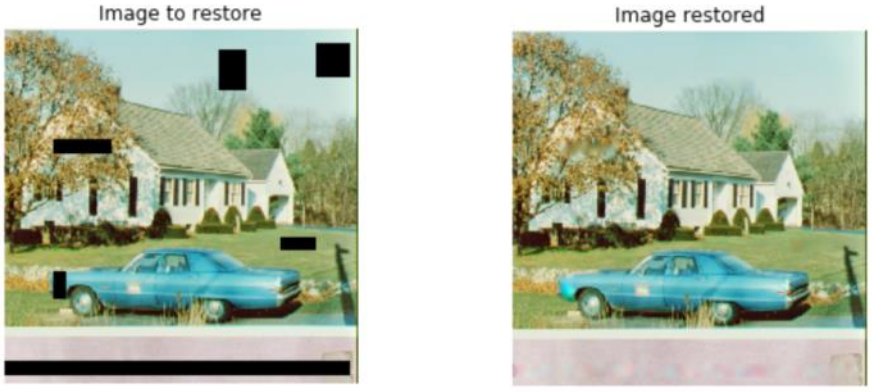


Image reconstruction

- Fixing damaged images
- Text removing
- Logo removing
- Object removing

Image reconstruction

Inpainting

- Reconstructing lost parts of images
- Looking at the non-damaged regions

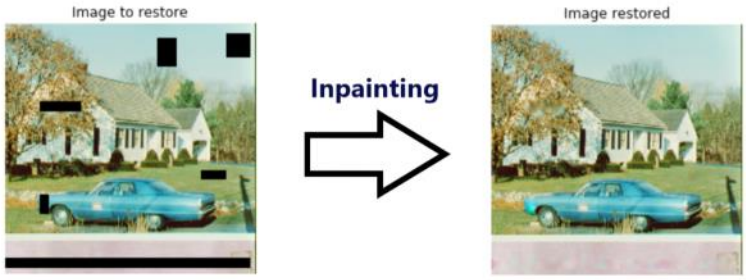


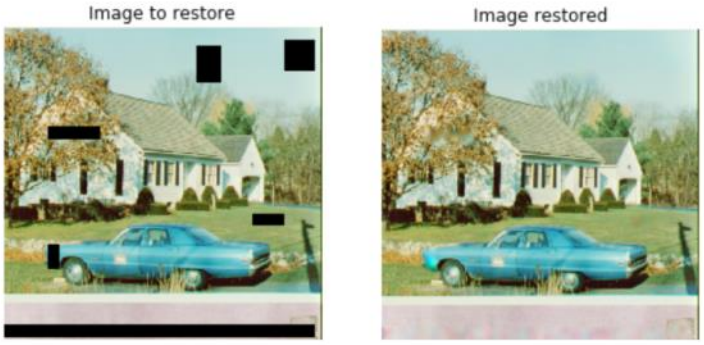
Image reconstruction

Image reconstruction in scikit-image

```
from skimage.restoration import inpaint

# Obtain the mask
mask = get_mask(defect_image)

# Apply inpainting to the damaged image using the mask
restored_image = inpaint.inpaint_biharmonic(defect_image,
                                            mask,
                                            multichannel=True)

# Show the resulting image
show_image(restored_image)
```

Image reconstruction in scikit-image

```
# Show the defect and resulting images
show_image(defect_image, 'Image to restore')
show_image(restored_image, 'Image restored')
```



Masks

Masks

```
def get_mask(image):
    ''' Creates mask with three defect regions '''
    mask = np.zeros(image.shape[:-1])

    mask[101:106, 0:240] = 1

    mask[152:154, 0:60] = 1
    mask[153:155, 60:100] = 1
    mask[154:156, 100:120] = 1
    mask[155:156, 120:140] = 1

    mask[212:217, 0:150] = 1
    mask[217:222, 150:256] = 1
    return mask
```

Unknown pixels have to be represented with 1 and known pixels with 0.

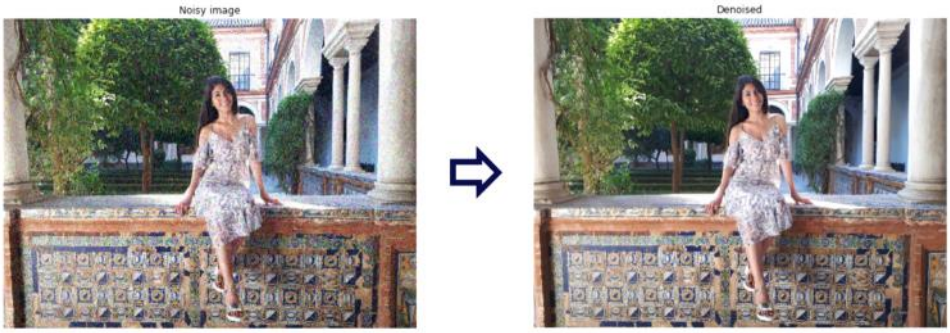
Noise

Apply noise in scikit-image

```
# Import the module and function
from skimage.util import random_noise

# Add noise to the image
noisy_image = random_noise(dog_image)

# Show original and resulting image
show_image(dog_image)
show_image(noisy_image, 'Noisy image')
```

Apply noise in scikit-image

Reducing noise



Denoising types

- Total variation (TV)
- Bilateral
- Wavelet denoising
- Non-local means denoising

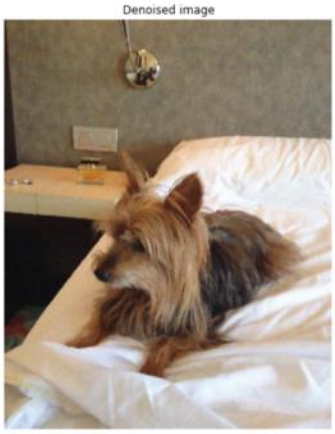
Denoising

Using total variation filter denoising

```
from skimage.restoration import denoise_tv_chambolle

# Apply total variation filter denoising
denoised_image = denoise_tv_chambolle(noisy_image,
                                      weight=0.1,
                                      multichannel=True)

# Show denoised image
show_image(noisy_image, 'Noisy image')
show_image(denoised_image, 'Denoised image')
```

Denoising

Total variation filter



Denoising

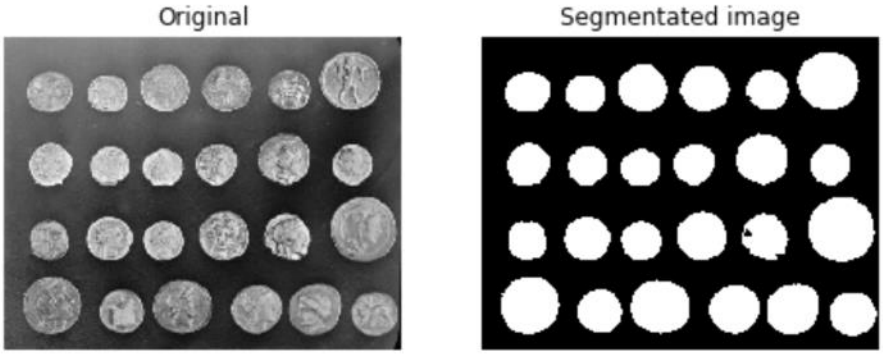
Bilateral filter

```
from skimage.restoration import denoise_bilateral

# Apply bilateral filter denoising
denoised_image = denoise_bilateral(noisy_image, multichannel=True)

# Show original and resulting images
show_image(noisy_image, 'Noisy image')
show_image(denoised_image, 'Denoised image')
```

Denoising

Bilateral filter

Segmentation



The goal is to partition images into regions, or segments, to simplify

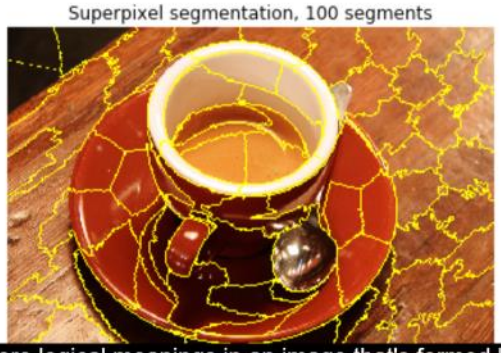
and/or change the representation into something more meaningful and easier to analyze.

Segmentation

Image representation

would you be able to tell that the pixel came from a puppy and that this single pixel holds a logical meaning?

Superpixels



So, we can explore more logical meanings in an image that's formed by bigger regions or grouped pixels.

A superpixel is a group of connected pixels with similar colors or gray levels.

Benefits of superpixels

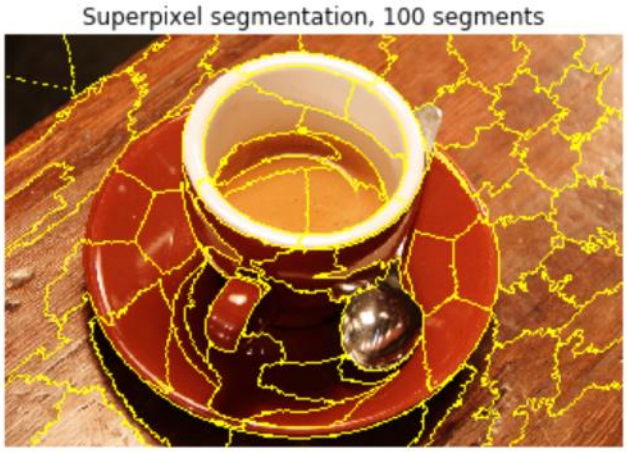
- More meaningful regions
- Computational efficiency

Segmentation

- Supervised
- Unsupervised

Unsupervised segmentation

Simple Linear Iterative Clustering (SLIC)



Unsupervised segmentation (SLIC)

```
# Import the modules
from skimage.segmentation import slic
from skimage.color import label2rgb

# Obtain the segments
segments = segmentation.slic(image)

# Put segments on top of original image to compare
segmented_image = label2rgb(segments, image, kind='avg')

show_image(image)
show_image(segmented_image, "Segmented image")
```

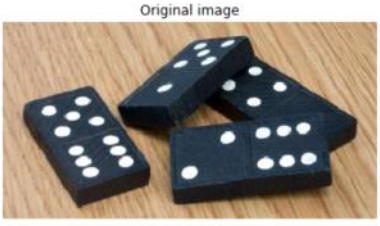
Unsupervised segmentation (SLIC)

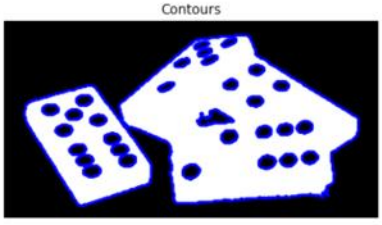
More segments

```
# Import the modules
from skimage.segmentation import slic
from skimage.color import label2rgb

# Obtain the segmentation with 300 regions
segments = slic(image, n_segments= 300)

# Put segments on top of original image to compare
segmented_image = label2rgb(segments, image, kind='avg')

show_image(segmented_image)
```

More segments

Finding contours

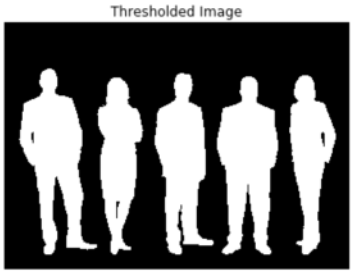


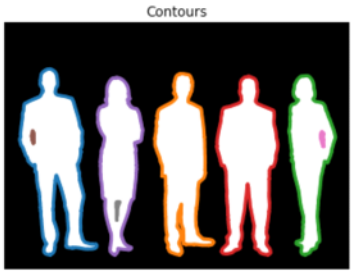


- Measure size
- Classify shapes
- Determine the number of objects

Total points in domino tokens: 35.

Binary images





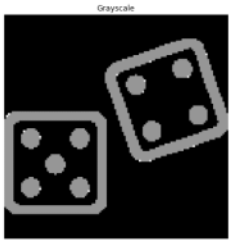
We can obtain a binary image applying **thresholding** or using **edge detection**

Find contours using scikit-image

PREPARING THE IMAGE

Transform the image to 2D grayscale.

```
# Make the image grayscale
image = color.rgb2gray(image)
```



Find contours using scikit-image

PREPARING THE IMAGE

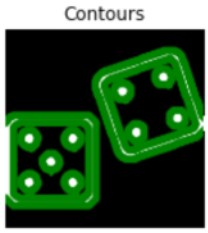
Binarize the image

```
# Obtain the thresh value
thresh = threshold_otsu(image)

# Apply thresholding
thresholded_image = image > thresh
```

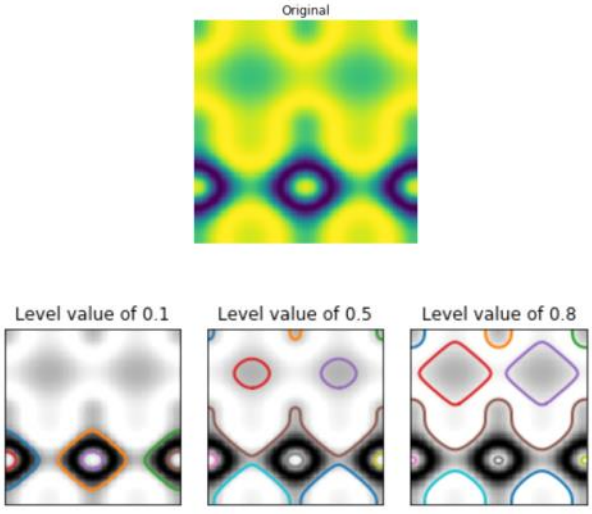
Find contours using scikit-image

And then use `find_contours()` .

```
# Import the measure module
from skimage import measure

# Find contours at a constant value of 0.8
contours = measure.find_contours(thresholded_image, 0.8)
```



Constant level value



The level value varies between 0 and 1, the closer to 1 the more sensitive the method is to detecting contours, so more complex contours will be detected.

We have to find the value that best detects the contours we care for.

The steps to spotting contours

```
from skimage import measure
from skimage.filters import threshold_otsu

# Make the image grayscale
image = color.rgb2gray(image)
# Obtain the optimal thresh value of the image
thresh = threshold_otsu(image)

# Apply thresholding and obtain binary image
thresholded_image = image > thresh

# Find contours at a constant value of 0.8
contours = measure.find_contours(thresholded_image, 0.8)
```

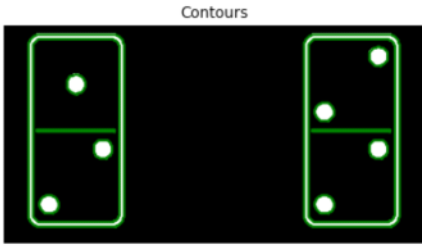
The steps to spotting contours

Resulting in

A contour's shape

```
for contour in contours:
    print(contour.shape)
```

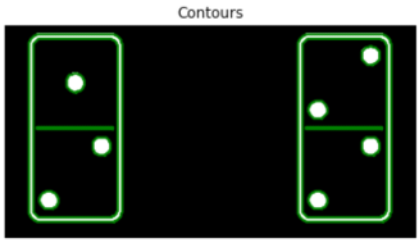
```
(433, 2)
(433, 2) --> Outer border
(401, 2)
(401, 2)
(123, 2)
(123, 2)
(59, 2)
(59, 2)
(59, 2)
(57, 2)
(57, 2)
(59, 2)
(59, 2)
```



A contour's shape

```
for contour in contours:
    print(contour.shape)
```

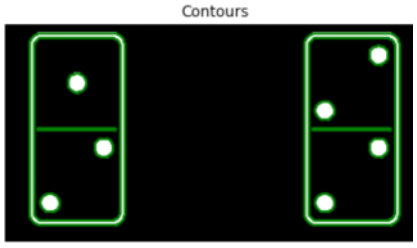
```
(433, 2)
(433, 2) --> Outer border
(401, 2)
(401, 2) --> Inner border
(123, 2)
(123, 2)
(59, 2)
(59, 2)
(59, 2)
(57, 2)
(57, 2)
(59, 2)
(59, 2)
```



A contour's shape

```
for contour in contours:
    print(contour.shape)
```

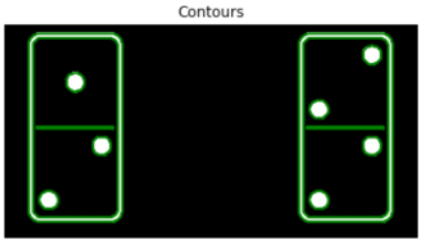
```
(433, 2)
(433, 2) --> Outer border
(401, 2)
(401, 2) --> Inner border
(123, 2)
(123, 2) --> Divisory line of tokens
(59, 2)
(59, 2)
(59, 2)
(57, 2)
(57, 2)
(59, 2)
(59, 2)
```



A contour's shape

```
for contour in contours:
    print(contour.shape)
```

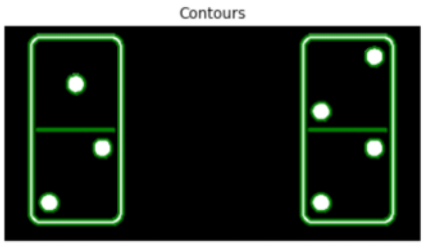
```
(433, 2)
(433, 2) --> Outer border
(401, 2)
(401, 2) --> Inner border
(123, 2)
(123, 2) --> Divisory line of tokens
(59, 2)
(59, 2)
(59, 2)
(57, 2)
(57, 2)
(59, 2)
(59, 2) --> Dots
```



A contour's shape

```
for contour in contours:
    print(contour.shape)
```

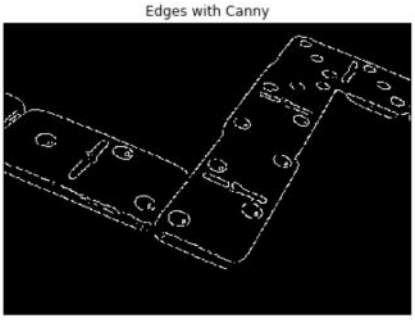
```
(433, 2)
(433, 2) --> Outer border
(401, 2)
(401, 2) --> Inner border
(123, 2)
(123, 2) --> Divisory line of tokens
(59, 2)
(59, 2)
(59, 2)
(57, 2)
(57, 2)
(59, 2)
(59, 2) --> Dots
```



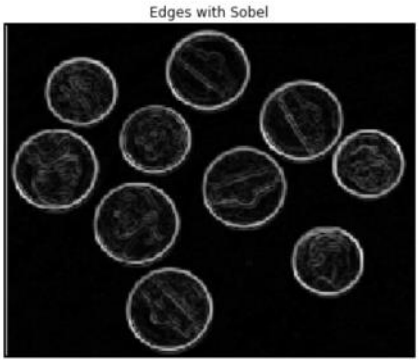
Number of dots: 7.

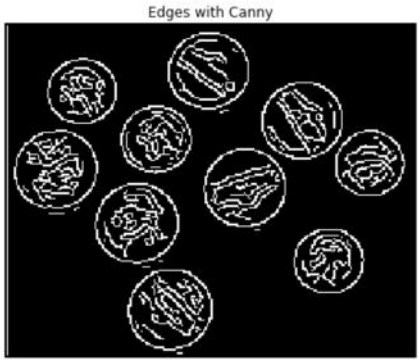
Advanced Operations, Detecting Faces and Features

Detecting edges



Edge detection

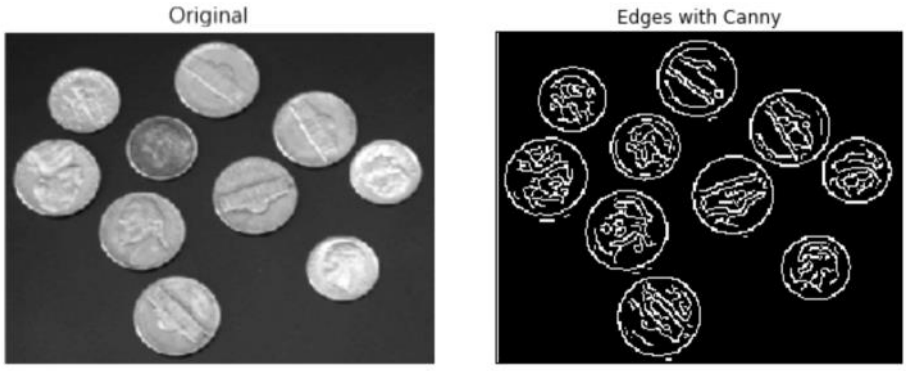




Edge detection

```
from skimage.feature import canny

# Convert image to grayscale
coins = color.rgb2gray(coins)

# Apply Canny detector
canny_edges = canny(coins)

# Show resulted image with edges
show_image(canny_edges, "Edges with Canny")
```

Edge detection



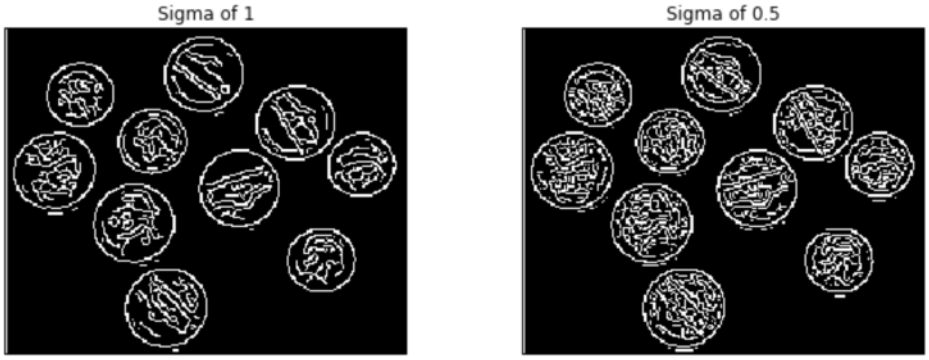
Canny edge detector

```
# Apply Canny detector with a sigma of 0.5
canny_edges_0_5 = canny(coins, sigma=0.5)

# Show resulted images with edges
show_image(canny_edges, "Sigma of 1")
show_image(canny_edges_0_5, "Sigma of 0.5")
```

The lower the value of this sigma, the less of gaussian filter effect is applied on the image, so it will spot more edges. On the other hand, if you set a higher value, more noise will be removed and the result is going to be a less edgy image.

The default value of this parameter is 1.

Canny edge detector



Corner detection

Edges are a type of feature in images.

Features are the points of interest which provide rich image content information.

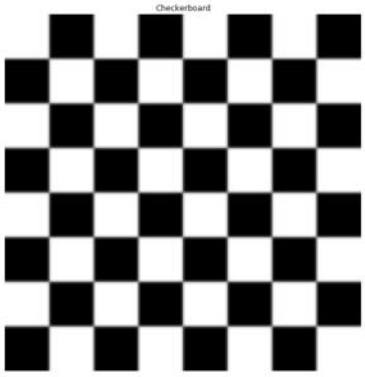
Points of interest

Points of interest are points in the image which are

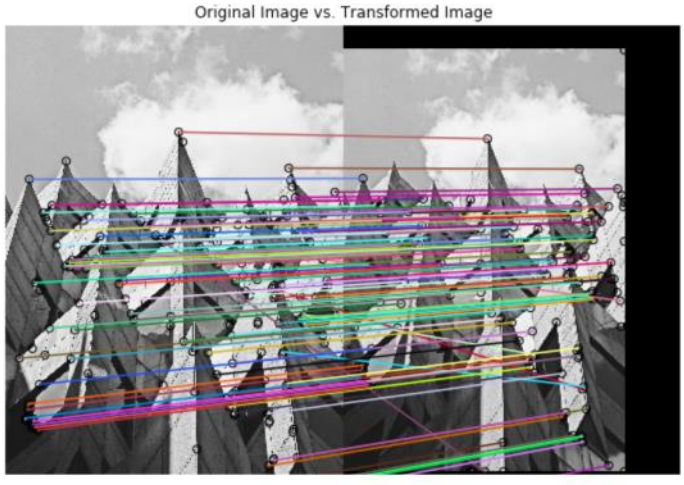
invariant to rotation, translation, intensity, and scale changes.

There are different interest points such as corners and edges.

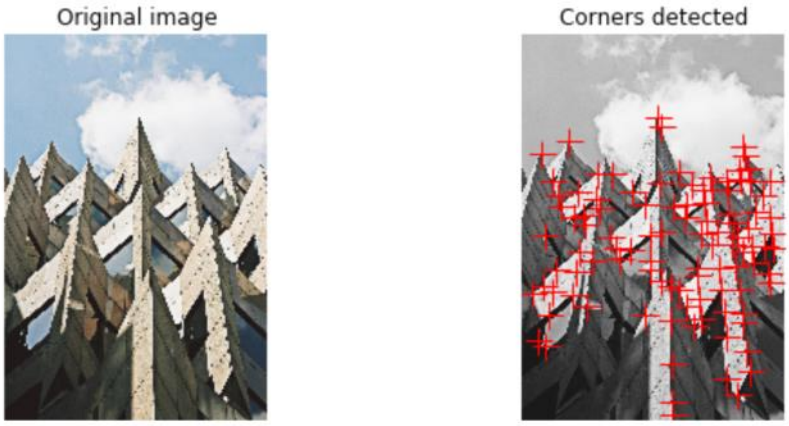
Corners



Matching corners



Harris corner detector



Harris corner detector

Harris corner detector

```
from skimage.feature import corner_harris

# Convert image to grayscale
image = rgb2gray(image)

# Apply the Harris corner detector on the image
measure_image = corner_harris(image)

# Show the Harris response image
show_image(measure_image)
```

Harris corner detector

Harris response image

Harris corner detector

```
# Finds the coordinates of the corners
coords = corner_peaks(corner_harris(image), min_distance=5)

print("A total of", len(coords), "corners were detected.")
```

```
A total of 122 corners were found from measure response image.
```

Corners detected

```
# Show image with marks in detected corners
show_image_with_detected_corners(image, coords)
```

Corners detected

Show image with contours

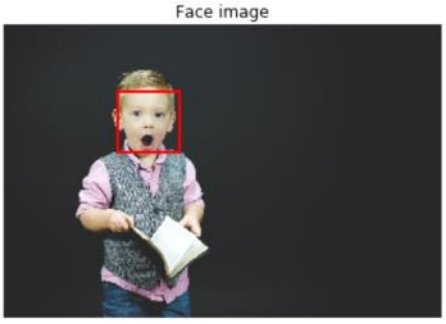
```
def show_image_with_corners(image, coords, title="Corners detected"):
    plt.imshow(image, interpolation='nearest', cmap='gray')
    plt.title(title)
    plt.plot(coords[:, 1], coords[:, 0], '+r', markersize=15)
    plt.axis('off')
    plt.show()
```

Face detection use cases

- Filters
- Auto focus
- Recommendations
- Blur for privacy protection
- To recognize emotions later on

Detecting faces with scikit-image

Face image

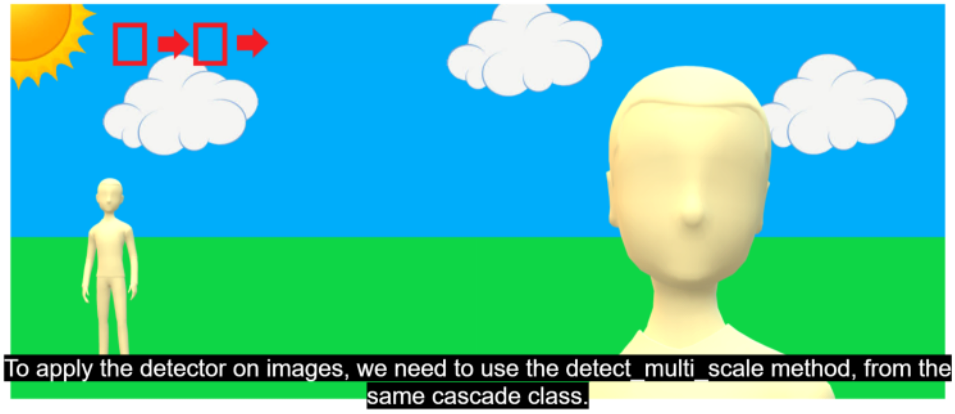


Face detected

Detecting faces with scikit-image

```
# Import the classifier class
from skimage.feature import Cascade

# Load the trained file from the module root.
trained_file = data.lbp_frontal_face_cascade_filename()

# Initialize the detector cascade.
detector = Cascade(trained_file)
```

Detecting faces

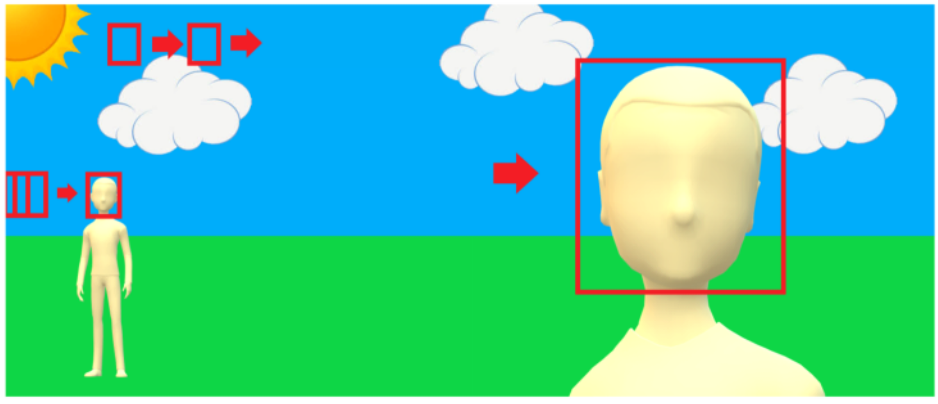


To apply the detector on images, we need to use the detect_multi_scale method, from the same cascade class.

This method searches for the object, in this case a face.

It creates a window that will be moving through the image until it finds something similar to a human face.

Detecting faces



Searching happens on multiple scales.

Detecting faces

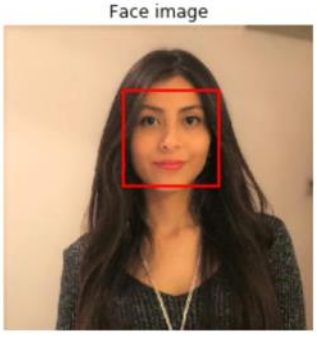
```
# Apply detector on the image
detected = detector.detect_multi_scale(img=image,
                                       scale_factor=1.2,
                                       step_ratio=1,
                                       min_size=(10, 10),
                                       max_size=(200, 200))
```

Detected faces

```
print(detected)
# Show image with detected face marked
show_detected_face(image, detected)
```

```
Detected face: [{'r': 115, 'c': 210, 'width': 167, 'height': 167}]
```

Face image

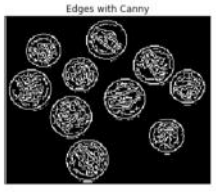


Show detected faces

```
def show_detected_face(result, detected, title="Face image"):
    plt.imshow(result)
    img_desc = plt.gca()
    plt.set_cmap('gray')
    plt.title(title)
    plt.axis('off')

    for patch in detected:
        img_desc.add_patch(
            patches.Rectangle(
                (patch['c'], patch['r']),
                patch['width'],
                patch['height'],
                fill=False,color='r',linewidth=2)
        )
    plt.show()
```

Applications

- Turning to grayscale before detecting edges/corners
- Reducing noise and restoring images
- Blurring faces detected
- Approximation of objects' sizes



Privacy protection

Privacy protection

```
# Import Cascade of classifiers and gaussian filter
from skimage.feature import Cascade
from skimage.filters import gaussian
```

Privacy protection

```
# Detect the faces
detected = detector.detect_multi_scale(img=image,
                                       scale_factor=1.2, step_ratio=1,
                                       min_size=(50, 50), max_size=(100, 100))
# For each detected face
for d in detected:
    # Obtain the face cropped from detected coordinates
    face = getFace(d)
```

Privacy protection

```
def getFace(d):
    ''' Extracts the face rectangle from the image using the
    coordinates of the detected.'''
    # X and Y starting points of the face rectangle
    x, y  = d['r'], d['c']

    # The width and height of the face rectangle
    width, height = d['r'] + d['width'],  d['c'] + d['height']

    # Extract the detected face
    face= image[x:width, y:height]
    return face
```

Privacy protection

```
# Detect the faces
detected = detector.detect_multi_scale(img=image,
                                       scale_factor=1.2, step_ratio=1,
                                       min_size=(50, 50), max_size=(100, 100))
# For each detected face
for d in detected:
    # Obtain the face cropped from detected coordinates
    face = getFace(d)

    # Apply gaussian filter to extracted face
    gaussian_face = gaussian(face, multichannel=True, sigma = 10)

    # Merge this blurry face to our final image and show it
    resulting_image = mergeBlurryFace(image, gaussian_face)
```

Privacy protection

```
def mergeBlurryFace(original, gaussian_image):
    # X and Y starting points of the face rectangle
    x, y  = d['r'], d['c']
    # The width and height of the face rectangle
    width, height = d['r'] + d['width'],  d['c'] + d['height']

    original[ x:width, y:height] =  gaussian_image
    return original
```

Privacy protection

Blurred faces