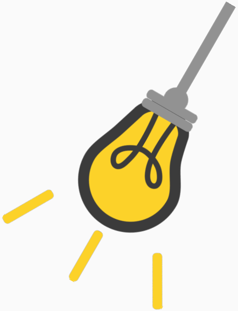
浙江大学爱丁堡大学联合学院
ZJU-UoE Institute

Basic image manipulation in Python

Nicola Romanò - nicola.romano@ed.ac.uk

Learning objectives

- Perform basic image manipulation (cropping, scaling, rotating, etc)
- Interpret and manipulate image histograms



Outline

A brief recap

Basic image operations

- Cropping
- Scaling
- Rotating

Last time we learnt how to open and display images using Python.

General plotting library

Specific functions for image manipulation

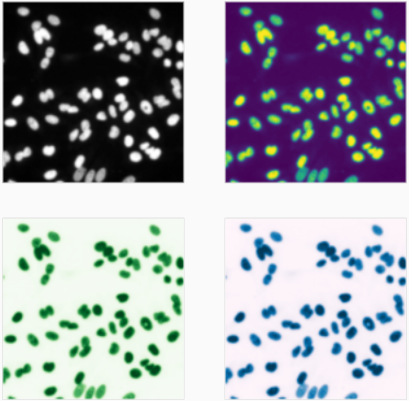
Library for matrix operations

Reading images as Numpy arrays

```
import matplotlib.pyplot as plt
img = plt.imread("cells.jpg")
plt.imshow(img)
plt.show()
```

```
from skimage import io
img = io.imread("cells.jpg")
io.imshow(img)
io.show()
```

```
fig, ax = plt.subplots(ncols=2, nrows=2)
ax[0, 0].imshow(img, "gray")
ax[0, 1].imshow(img, "viridis")
ax[1, 0].imshow(img, "Greens")
ax[1, 1].imshow(img, "PuBu")
```



Check out the Matplotlib website for a list of colourmaps!

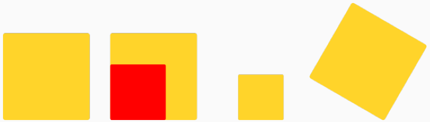
`https://matplotlib.org/stable/tutorials/colors/colormaps.html`

Outline

Basic image operations

Often you will need to crop, scale or rotate an image before further manipulation. There are many use cases for this, including

- Analysing only part of an image (cropping)
- Making sure all input images for a pipeline are the same size (scaling, cropping)
- Aligning multiple images (e.g. in video stabilization) (rotating)
- ...



Cropping

Cropping is as easy as subsetting the image matrix.

```
# Assume we loaded a 512 x 512 image in the img variable
# Take the top-left 100 x 100 pixels
img1 = img[0:100, 0:100] # shape (100, 100)
```

Cropping

Cropping is as easy as subsetting the image matrix.

```
# Assume we loaded a 512 x 512 image in the img variable
# Take the top-left 100 x 100 pixels
img1 = img[0:100, 0:100] # shape (100, 100)
# Take a 100 pixel wide strip on the left of the image...
img2 = img[:, 0:100] # shape (512, 100)
# Or on the right!
img3 = img[:, -100:]  # shape (512, 100)
```

Cropping in more than two dimensions

Since images are just tensors, we can crop them in any dimensions.

```
# We loaded a 512 x 512 RGB image in img
# img.shape is (3, 512, 512)
# Extract the green channel
img_green = img[1] # Shape (512, 512)
```

Cropping in more than two dimensions

Since images are just tensors, we can crop them in any dimensions.

```
# We loaded a 512 x 512 RGB image in img
# img.shape is (3, 512, 512)
# Extract the green channel
img_green = img[1] # Shape (512, 512)

# We have a 512 x 512 video of 300 frames
# Take the first 50 frames
# video.shape is (300, 512, 512)
video_crop = video[0:50]
```

Question time

We have a 100 frames video of a 512 x 512 z-stack with 60 planes loaded in `zstack`.
`zstack.shape` is `(100, 60, 512, 512)`

What does this code give you?

```
result = stack[50:70, :, 100:300]
```

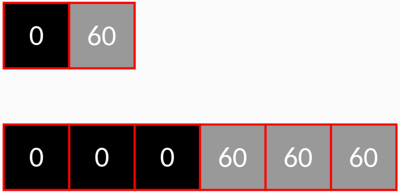
Two problems:

When **upscaling** we need to generate new information.

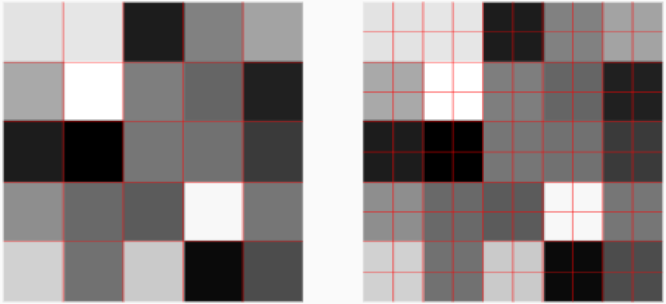
When **downscaling** we need to decide "how to lose" information.

Nearest-neighbor interpolation

The simplest way to resize an image is to use nearest neighbor interpolation. Each pixel of the scaled image will have the colour of the nearest pixel in the original image.
In this "1D" example, we resize a 1x2 image to 1x6

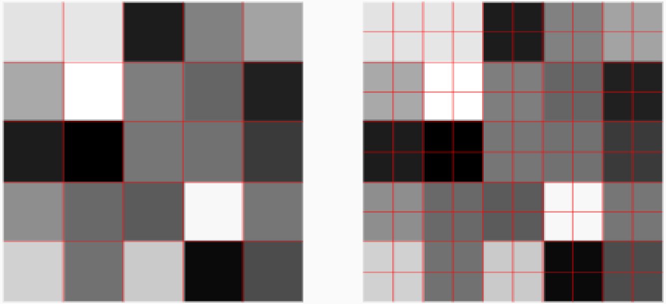


Nearest-neighbour interpolation



Upscaling of a 5x5 image by a factor of 2, to get a 10x10 image, with nearest neighbour interpolation

Nearest-neighbour interpolation



Upscaling of a 5x5 image by a factor of 2, to get a 10x10 image, with nearest neighbour interpolation

```
from skimage.transform import rescale
img_scaled = rescale(img, 2, order=0)
```

Scaling with interpolation

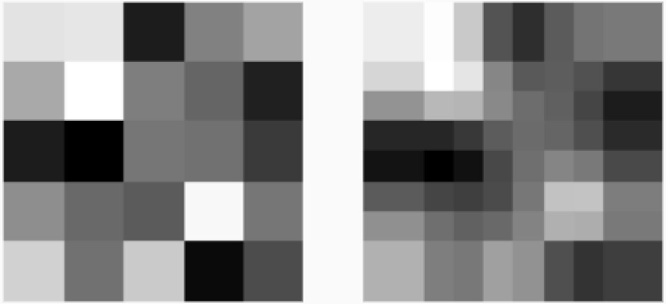
Better ways to scale an image involve changing the pixel values of the rescaled image based on their neighbourhood.

For example we could use linear interpolation

0	60

0	12	24	36	48	60

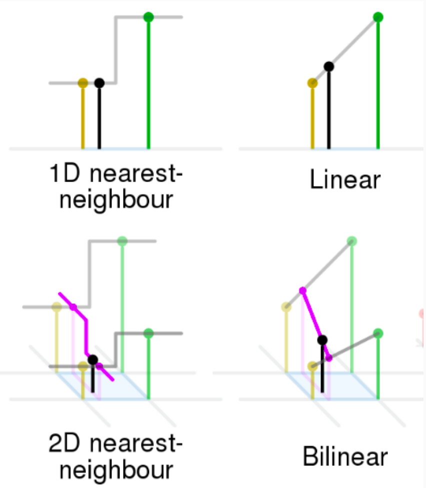
The same applies in 2D, although we need to take into accounts the values of both horizontal and vertical neighbours (bilinear interpolation).



Upscaling of a 5x5 image by a factor of 2, to get a 10x10 image, with bi-linear interpolation

```
from skimage.transform import rescale
img_scaled = rescale(img, 2, order=1)
```

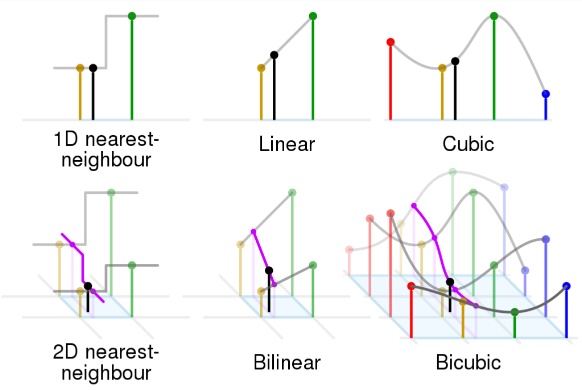
Nearest neighbor vs linear interpolation



Comparison of nearest neighbour and linear interpolation - Source: Wikipedia

Higher orders of interpolation

We can use higher orders of interpolation to produce smoother results.



Comparison of nearest neighbour and linear interpolation - Source: Wikipedia

Scikit Image supports values from 0 to 5 in the `order` parameter of the `rescale` function. 0 is nearest neighbor, 1 is bi-linear, 2 is bi-quadratic and so on.

Scaling to a target size

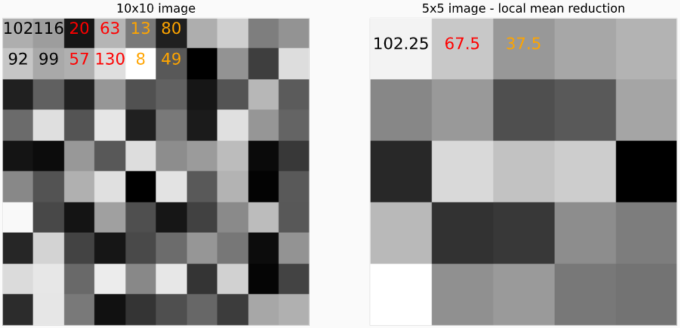
To scale to a target size, rather than by a specific factor, we can use `resize` instead of `rescale`.

```
import matplotlib.pyplot as plt
from skimage.transform import resize, rescale

img = plt.imread("cells.jpg")
img_scaled1 = rescale(img, 2)
img_scaled2 = resize(img, (150, 150))
```

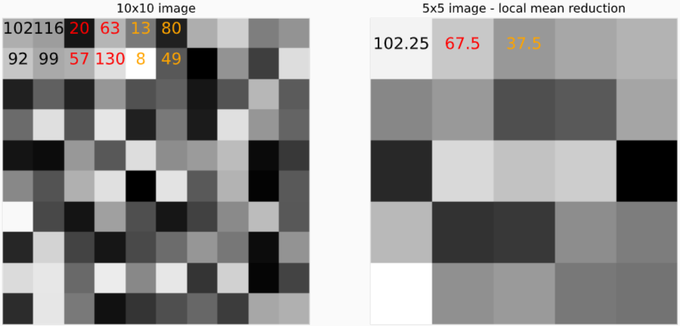
Local mean downscaling

Another simple solution for downscaling is calculating the local mean of each pixel



Local mean downscaling

Another simple solution for downscaling is calculating the local mean of each pixel




```
from skimage.transform import downscale_local_mean
img_small = downscale_local_mean(img, (2,2))
```

Image needs to be padded if the size is not a multiple of the downscaling factor. This method is fast but not good at keeping fine details.

Scaling summary

`skimage.transform.rescale` $\rightarrow$ scales an image by a specific factor (>1 upscaling; <1 downscaling.). Can specify a different scaling factor for each dimension of the image.

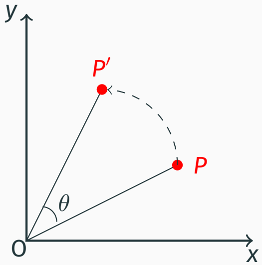
`skimage.transform.resize` $\rightarrow$ scales an image to a target size.

`skimage.transform.downscale_local_mean` $\rightarrow$ downscales the image by a specific factor (>1), using the local mean of each pixel.

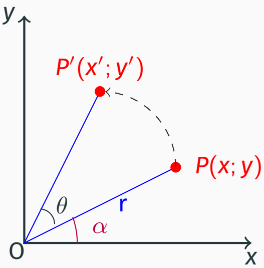
Outline

Rotating a point around the origin

We want to rotate a point $P(x, y)$ around the origin by an angle θ to get $P'(x', y')$.

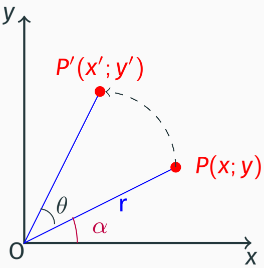


Rotating a point around the origin



$$P : \begin{cases} x = r\cos(\alpha) \\ y = r\sin(\alpha) \end{cases} \quad P' : \begin{cases} x' = r\cos(\alpha+\theta) \\ y' = r\sin(\alpha+\theta) \end{cases}$$

Rotating a point around the origin

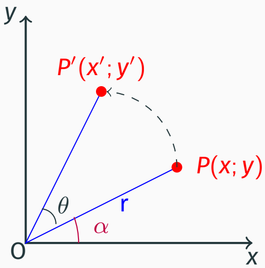


$$P : \begin{cases} x = r\cos(\alpha) \\ y = r\sin(\alpha) \end{cases} \quad P' : \begin{cases} x' = r\cos(\alpha+\theta) \\ y' = r\sin(\alpha+\theta) \end{cases}$$

$$P' : \begin{cases} x' = r\cos(\alpha)\cos(\theta) - r\sin(\alpha)\sin(\theta) \\ y' = r\sin(\alpha)\cos(\theta) + r\sin(\theta)\cos(\alpha) \end{cases}$$

Why? Wikipedia to the rescue!

Rotating a point around the origin



$$P : \begin{cases} x = r\cos(\alpha) \\ y = r\sin(\alpha) \end{cases} \quad P' : \begin{cases} x' = r\cos(\alpha+\theta) \\ y' = r\sin(\alpha+\theta) \end{cases}$$

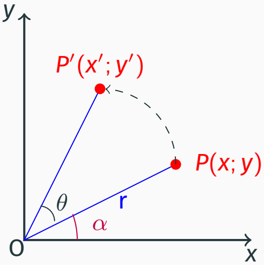
$$P' : \begin{cases} x' = r\cos(\alpha)\cos(\theta) - r\sin(\alpha)\sin(\theta) \\ y' = r\sin(\alpha)\cos(\theta) + r\sin(\theta)\cos(\alpha) \end{cases}$$

Why? Wikipedia to the rescue!

Thus:

$$P' : \begin{cases} x' = x\cos(\theta) - y\sin(\theta) \\ y' = y\cos(\theta) + x\sin(\theta) \end{cases}$$

Rotating a point around the origin



$$P : \begin{cases} x = r\cos(\alpha) \\ y = r\sin(\alpha) \end{cases} \quad P' : \begin{cases} x' = r\cos(\alpha+\theta) \\ y' = r\sin(\alpha+\theta) \end{cases}$$

$$P' : \begin{cases} x' = r\cos(\alpha)\cos(\theta) - r\sin(\alpha)\sin(\theta) \\ y' = r\sin(\alpha)\cos(\theta) + r\sin(\theta)\cos(\alpha) \end{cases}$$

Why? Wikipedia to the rescue!

Thus:

$$P' : \begin{cases} x' = x\cos(\theta) - y\sin(\theta) \\ y' = y\cos(\theta) + x\sin(\theta) \end{cases}$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Transformation (rotation) matrix

Rotating in Scikit Image

To rotate an image we will:

- Offset our image so that it is centered on the origin
- Generate a rotation matrix
- Multiply the coordinates of each pixel by the rotation matrix

Rotating in Scikit Image

To rotate an image we will:

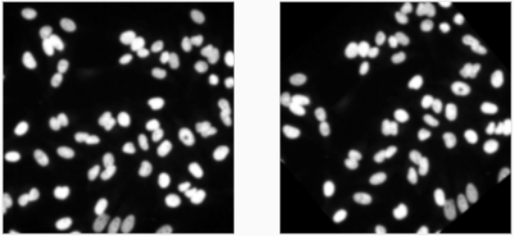
- Offset our image so that it is centered on the origin
- Generate a rotation matrix
- Multiply the coordinates of each pixel by the rotation matrix

Luckily, Scikit Image has a ready function for that, `skimage.transform.rotate`.

```
import matplotlib.pyplot as plt
from skimage.transform import rotate

img = plt.imread("cells.jpg")
img_rotated = rotate(img, 20)

plt.imshow(img_rotated, cmap="gray")
plt.show()
```



Note: we lost part of the image and we "gained" black pixels around it. By default, `rotate` performs bilinear interpolation.

Optional exercise - want to try by yourself?

It might be interesting to try to code image rotation by yourself.

Hints:

- You can generate transformation matrices using `skimage.transform.SimilarityTransform`
- You will need three matrices: one translation matrix to offset your image by (-xcenter, -ycenter); a rotation matrix to rotate your image; and finally another translation matrix to translate your image back to its original position.
- You can combine the matrices as `m1+m2+m3`
- You can use `skimage.transform.warp` to apply the transformation matrix to your image.

If you are stuck, try to look at the source code for rotate!