CONSTRUCTOR UNIVERSITY

Data Structures and Processing

Dr. Carlos Henrique Brandt
cbrandt@constructor.university

Image Processing

References

- “Image Processing with Python”, The Data Carpentries, https://datacarpentry.org/image-processing

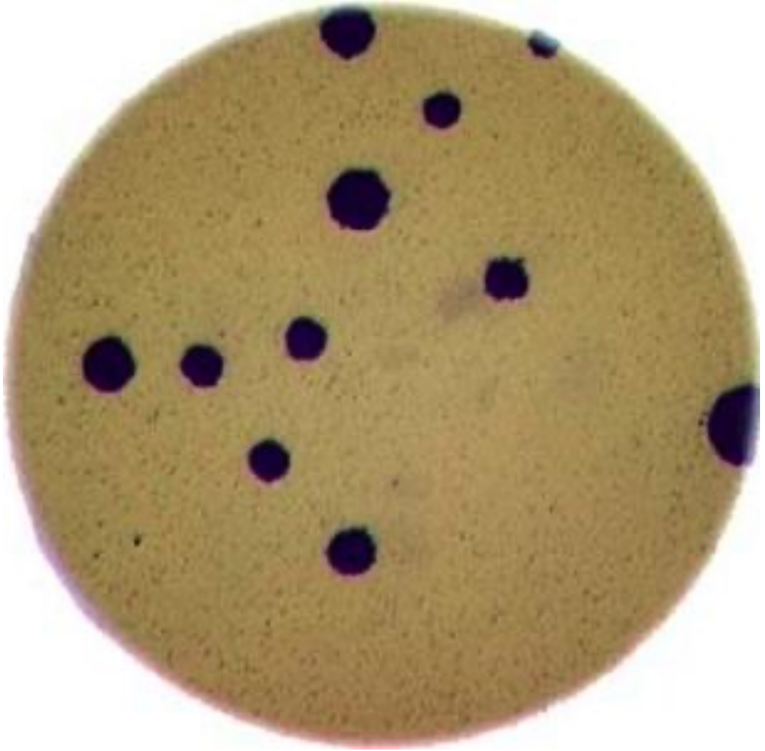
Introduction

Images are an important source of information, humans have always relied on photography to registry life.

- Image processing is used for
  - detecting astronomical objects
  - detecting diseases
  - estimating populations
  - driving cars

Introduction

- Morphometrics: counting and measuring objects (sizes, shapes)



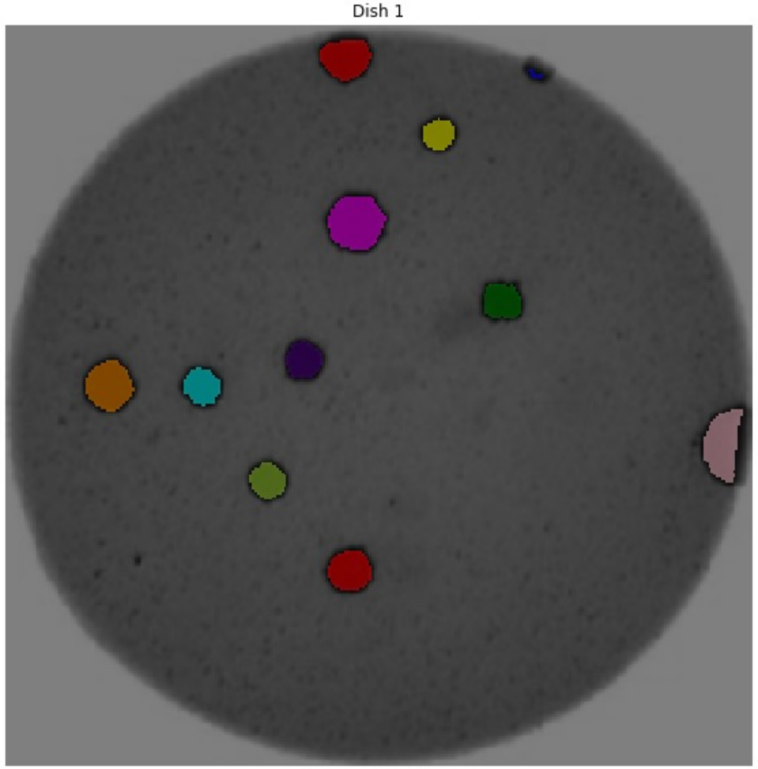


Image Basics

How are images represented in digital formats?

- Pixels and arrays
  - bit and bytes
  - Monochromatic and RGB
- Image formats
  - BMP, JPEG, TIFF
- Compression
  - lossy and lossless
- Metadata

Image Basics

Pixels and arrays

Images are stored as rectangular arrays of discrete "picture elements", otherwise known as ***pixels***.

Each pixel can be thought of as a single (square) point of colour.

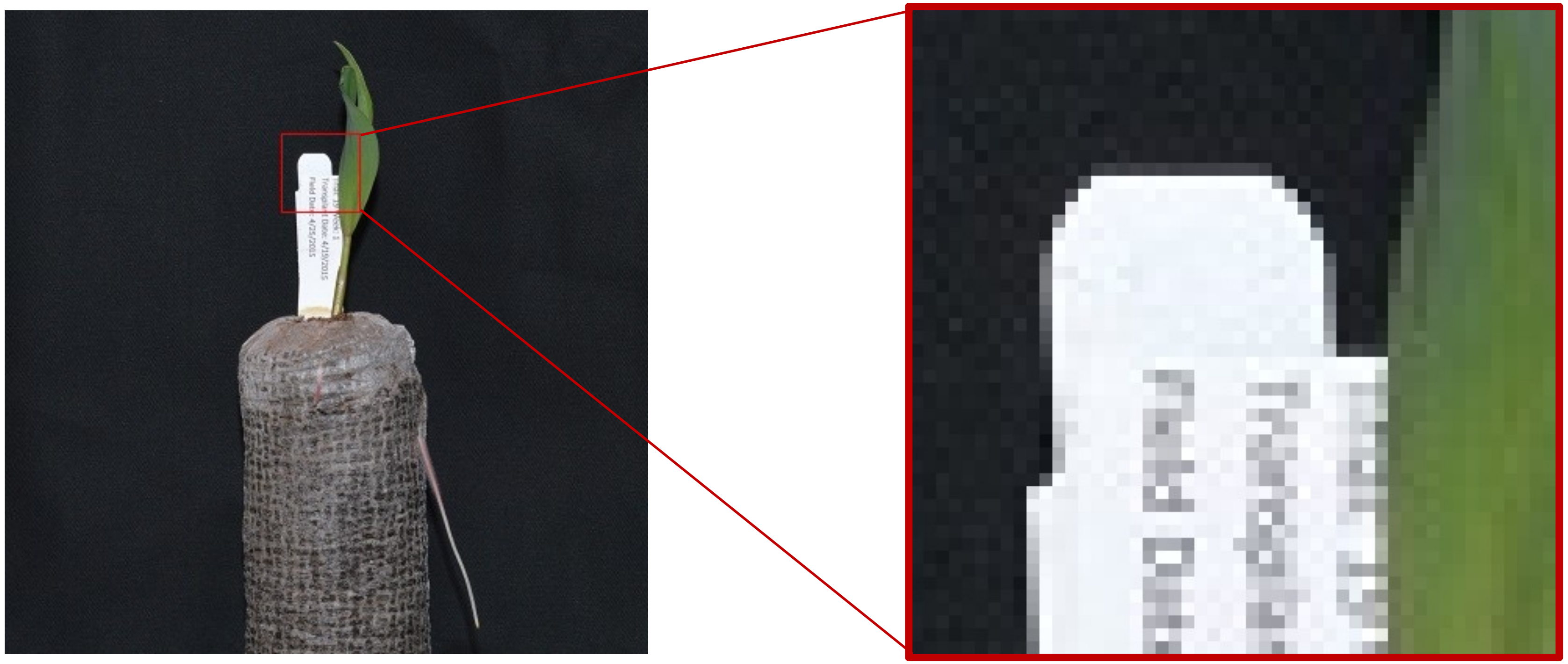


Image Basics

Pixels and arrays

Here is an image of an “8” in 15 pixels (5,3):

This image is represented by array:

```
[[0. 0. 0.]
 [0. 1. 0.]
 [0. 0. 0.]
 [0. 1. 0.]
 [0. 0. 0.]]
```

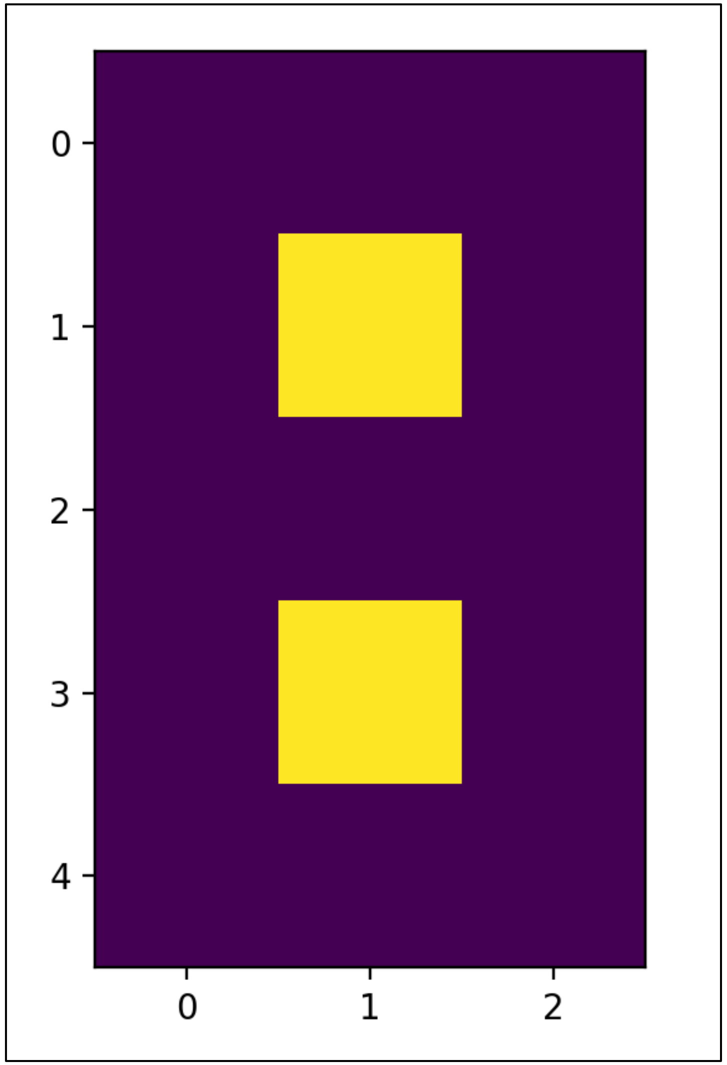


Image Basics

Pixels and arrays

Here is the “8” transformed into “0”:

```
image[2,1] = 1
```

This image is represented by array:

```
[[0. 0. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 0. 0.]]
```

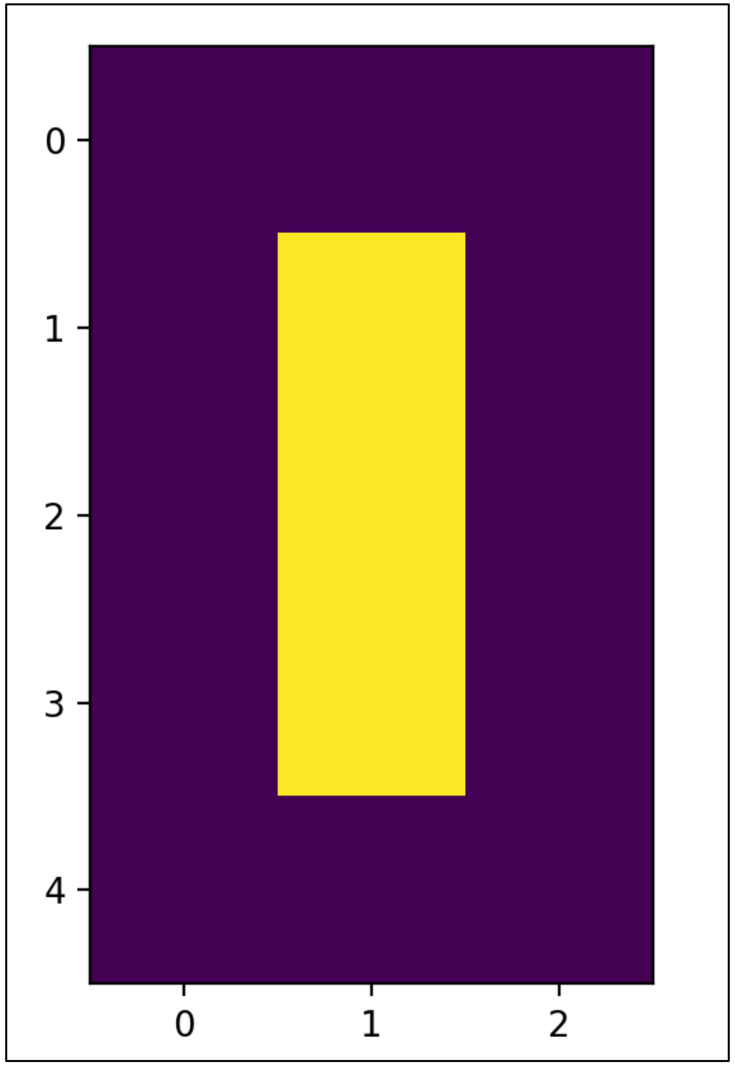


Image Basics

Pixels and arrays

There are a couple of aspects to notice from these simple examples:

- The colours associated to “0” and “1” were arbitrary.
  - Visualization tools will often decide for a colour palette if none is explicitly defined.
- The (axes) coordinates system start from the top-left corner.
  - X-coordinates increase to the right
  - Y-coordinates increase downwards

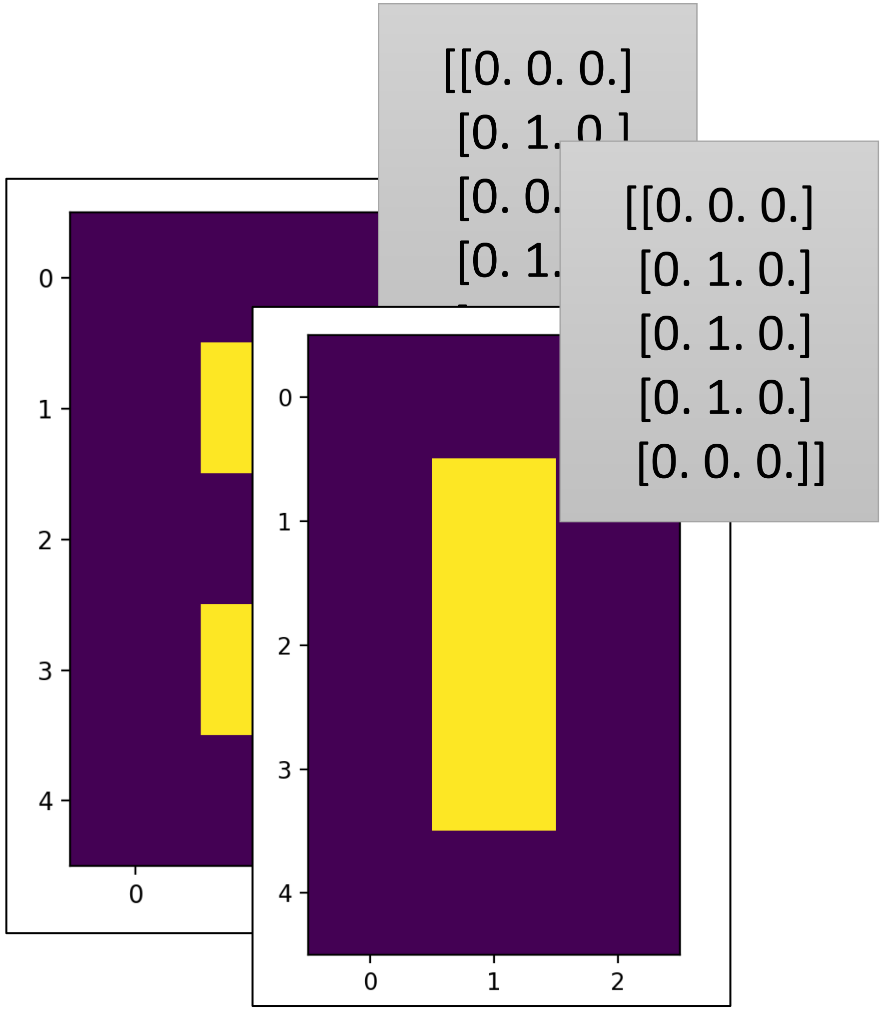


Image Basics

Pixels and arrays

There are a couple of aspects to notice from these simple examples:

- The colours associated to “0” and “1” were arbitrary.
    - Visualization tools will often decide for a colour palette if none is explicitly defined.
- The (axes) coordinates system start from the top-left corner.
    - X-coordinates increase to the right
    - Y-coordinates increase downwards

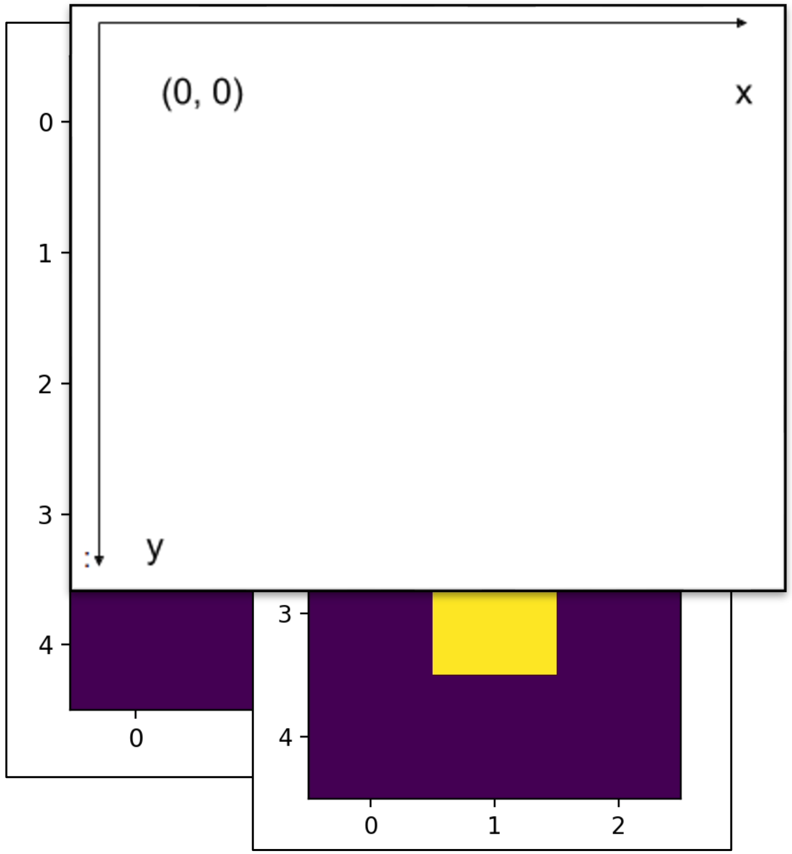
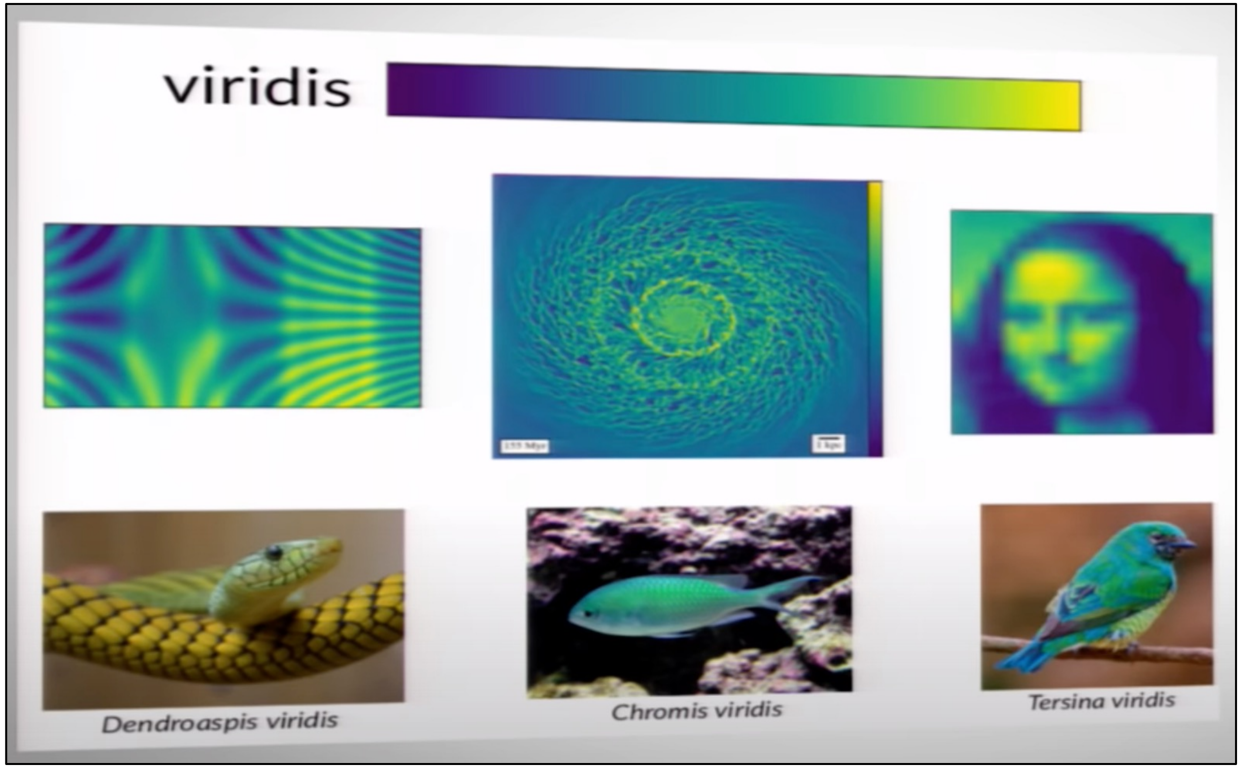


Image Basics

Colours

The tool we're using to visualize these images is Matplotlib, which use the Viridis colour palette by default, and normalize the values in the array to span the colour range.



See https://bids.github.io/colormap/

Image Basics

Colours

Let's see what Matplotlib does if we add more non-zero values to our (5,3) array, and stretch the values from 0 to 255:

```
> image *= 128
> image[2, :] = 255
```

```
[[0.   0.   0.  ]
 [0.   128. 0.  ]
 [255. 255. 255.]
 [0.   128. 0.  ]
 [0.   0.   0.  ]]
```

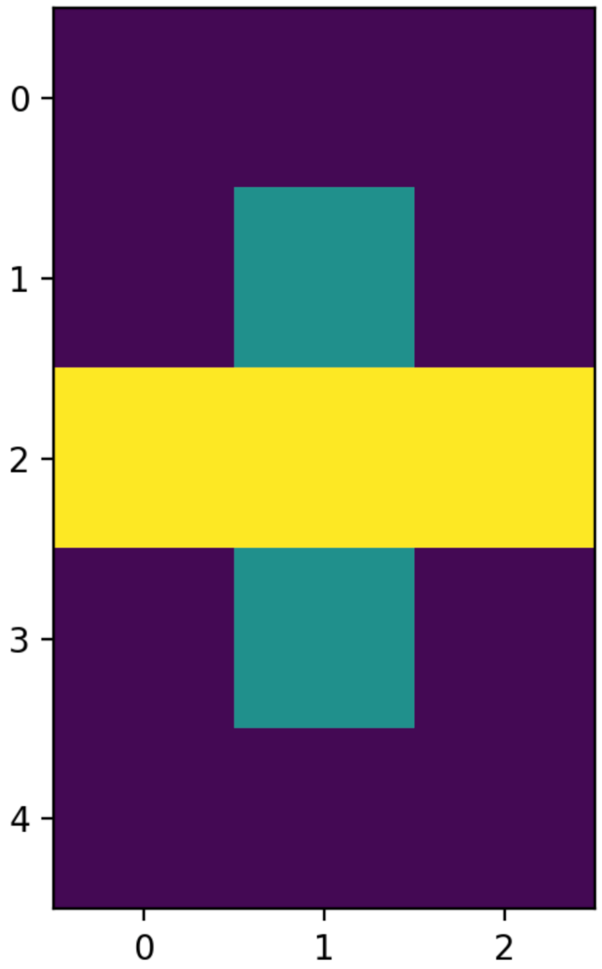


Image Basics

Colours

…the same image, through a grayscale colormap:

```
> plt.imshow(image, cmap=plt.cm.gray)
```

```
[[0.   0.   0.  ]
 [0.   128. 0.  ]
 [255. 255. 255.]
 [0.   128. 0.  ]
 [0.   0.   0.  ]]
```

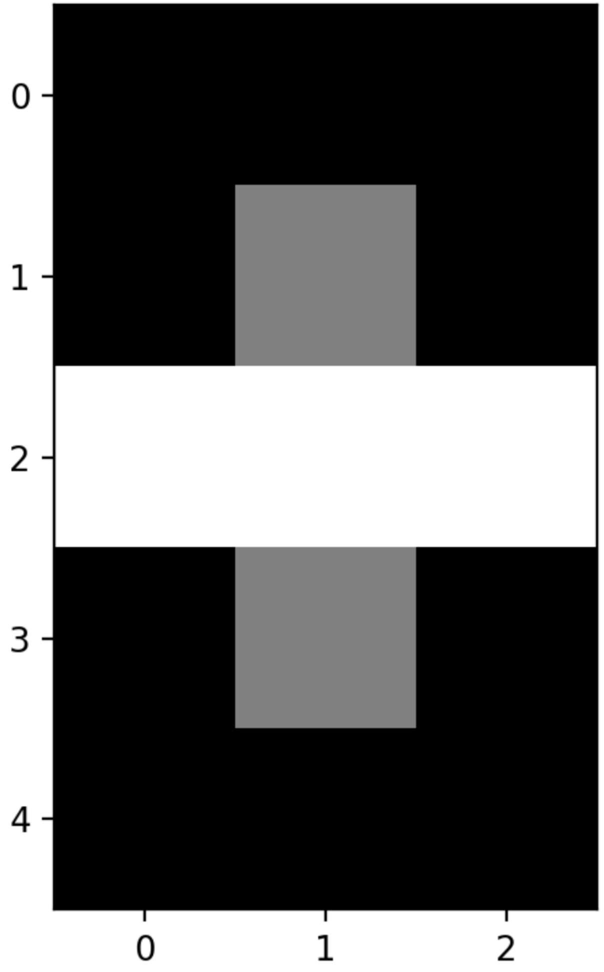


Image Basics

Colours and arrays

Images represented in 2-dimensional are called *monochromatic* images.

The colours associated to those values (by Matplotlib, for instance) are fake colours given by a colormap. There are many palettes one can use, see:

- matplotlib.org/gallery/color/colormap_reference

True colour images are represented by 3-dimensional arrays, where each colour channel is represented by a 2D array.

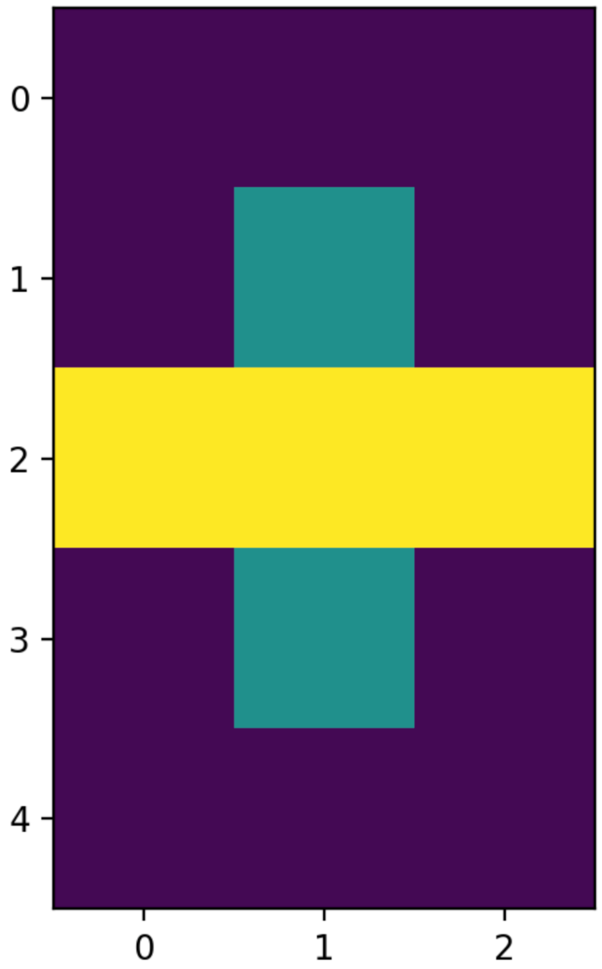


Image Basics

Colours and arrays

The most common standard for colour images is the additive RGB where each pixel is represented by three values: Red, Green, Blue

The following (Numpy) array will create the image we see here:

```
> rgen = np.random.RandomState(2021)
> img = rgen.randint(0,255,size=(4,4,3))
> plt.imshow(img)
```

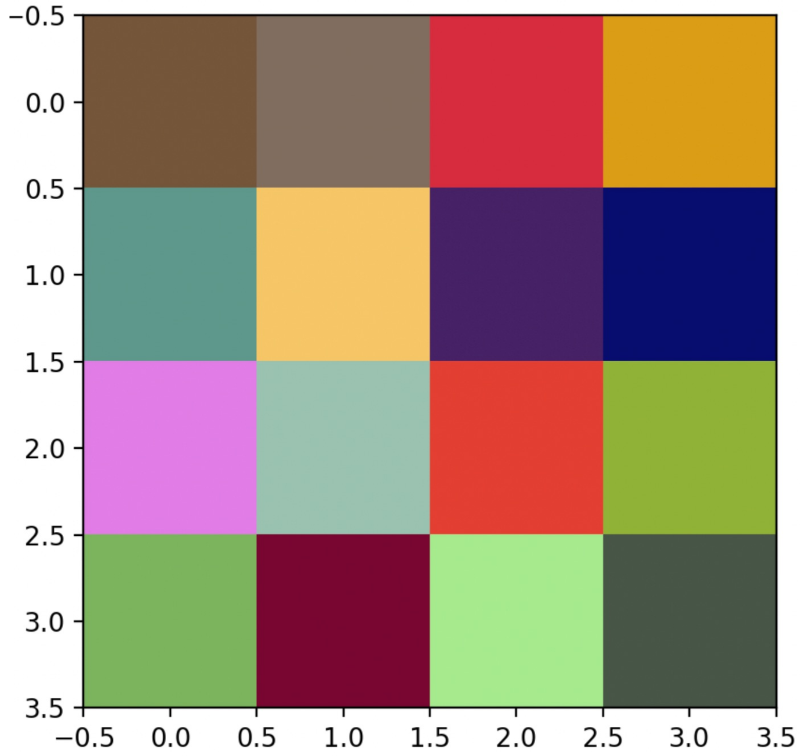
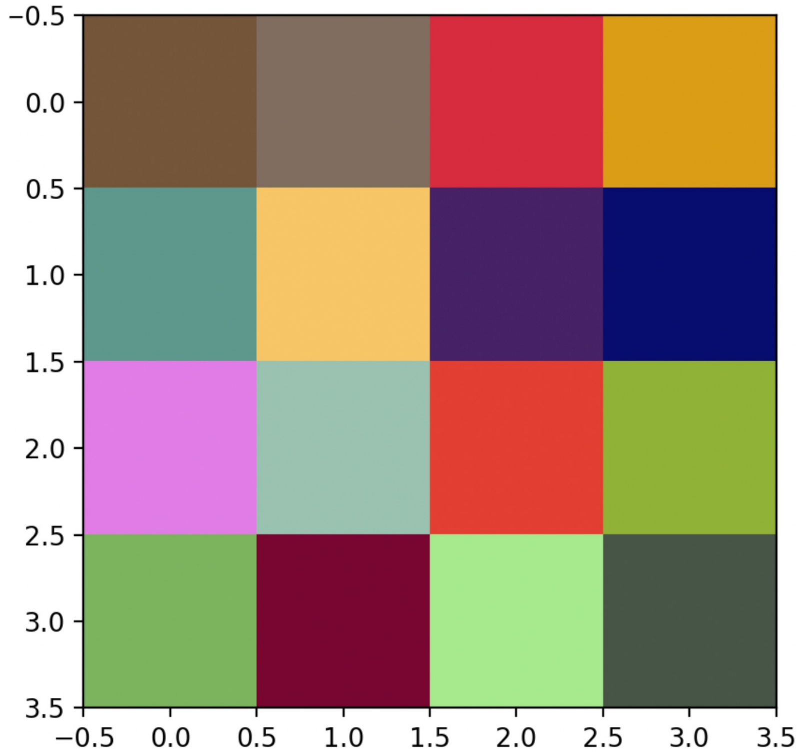


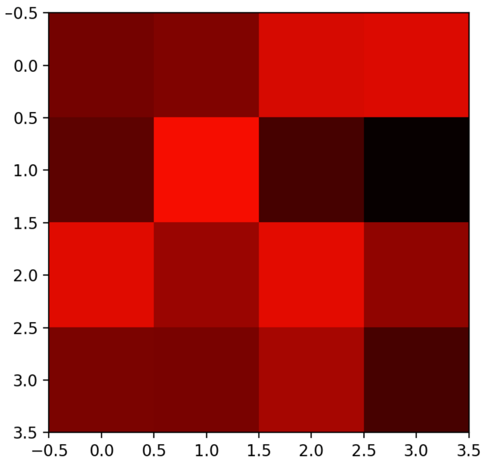
Image Basics

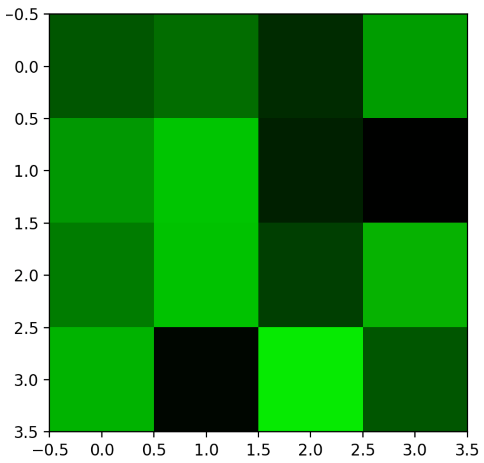
Colours and arrays

Each of those colours are the result of combining the values of each (R,G,B) channel:

```
> red_channel = img * [1, 0, 0]
> green_channel = img * [0, 1, 0]
> blue_channel = img * [0, 0, 1]
```







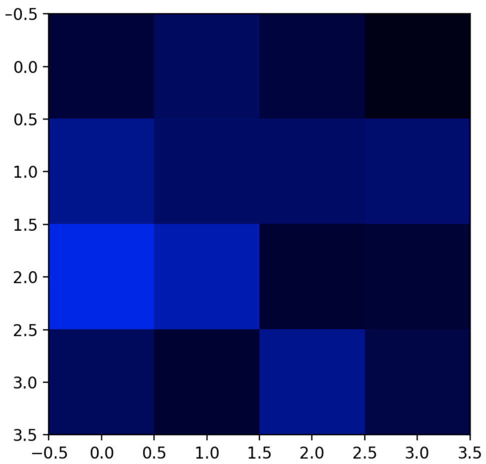


Image Basics

Colours, bits and bytes

RGB images are also called 24-bit RGB images, each channel is represented by an 8-bit – or 1 byte – number (from 0-255).

A byte – 8 bits – can store 256 integer values. The amount of bits used to represent an image is called the depth.

A monochromatic (integer) image is an 8-bit image, meaning the pixels can store 256 values.

Pixels in a (24-bit) RGB images can represent 16 million colours.

Image Basics

Colours, bits and bytes

RGB colour table:

Color name	RGB triplet	Color
Red	(255, 0, 0)	
Lime	(0, 255, 0)	
Blue	(0, 0, 255)	
White	(255, 255, 255)	
Black	(0, 0, 0)	
Gray	(128, 128, 128)	
Fuchsia	(255, 0, 255)	
Yellow	(255, 255, 0)	
Aqua	(0, 255, 255)	
Silver	(192, 192, 192)	
Maroon	(128, 0, 0)	
Olive	(128, 128, 0)	
Green	(0, 128, 0)	
Teal	(0, 128, 128)	
Navy	(0, 0, 128)	
Purple	(128, 0, 128)	

Image Basics

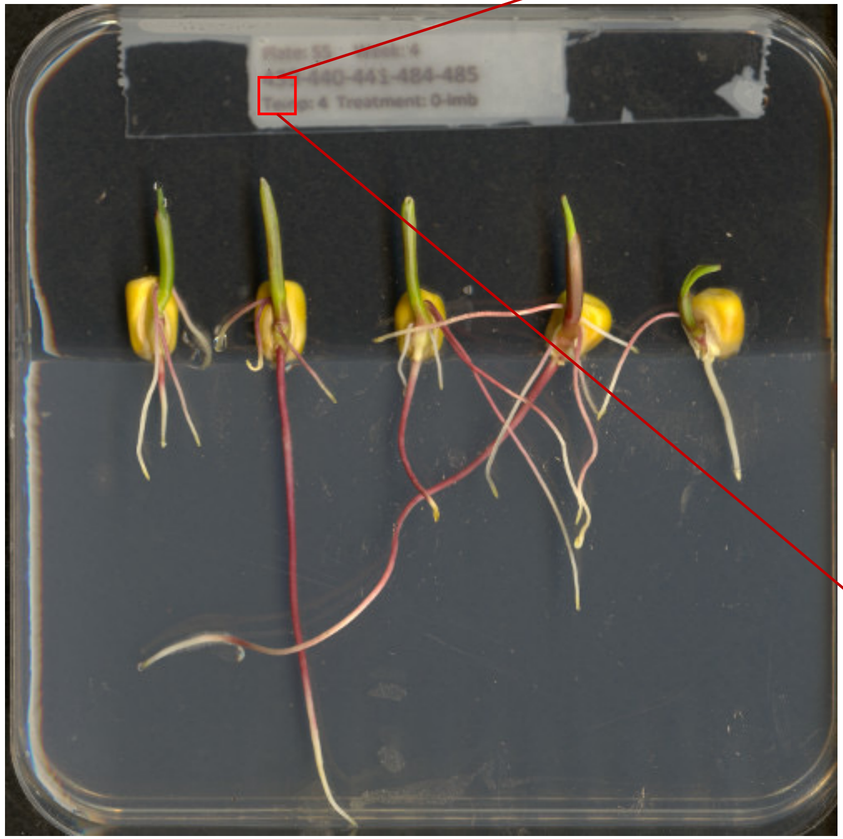
Image formats

- BMP: *Device-Independent Bitmap* are non-compressed file formats capable of storing 8-bit, 16-bit, or 24-bit (colour depth) images.
- JPEG: *Joint Photographic Experts Group* file is a compressed lossy format capable of storing 24-bit RGB images. The amount of compression when storing an image can be adjusted.
- PNG: *Portable Network Graphics* file supports lossless compression and can store 8-bit (gray), 24-bit (RGB), and 32-bit (RGBA) images.
- TIFF: *Tag Image File Format* can be uncompressed, or compressed using either lossy or lossless. TIFs can stored more than one image.

Image Basics

Image formats

Effects of compression in an image:



Uncompressed TIF

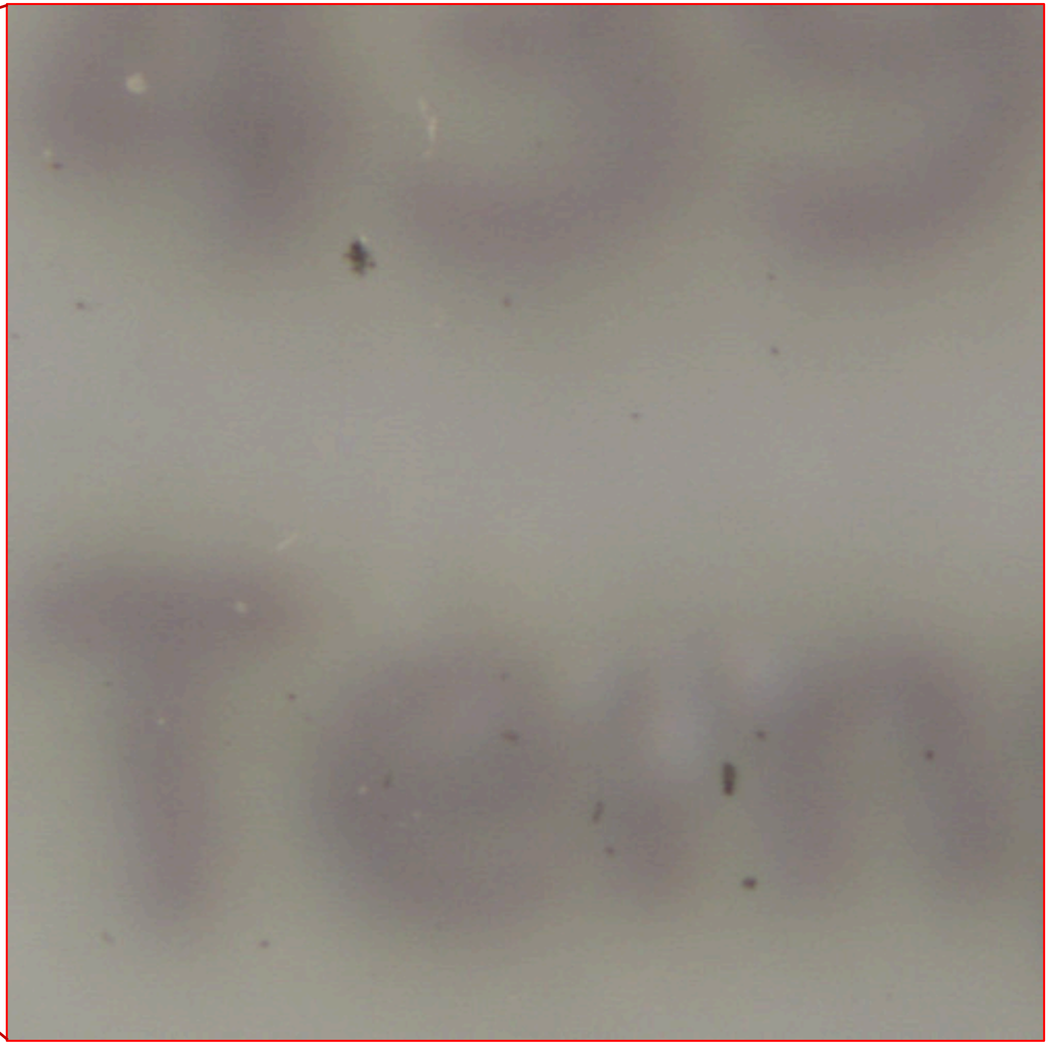
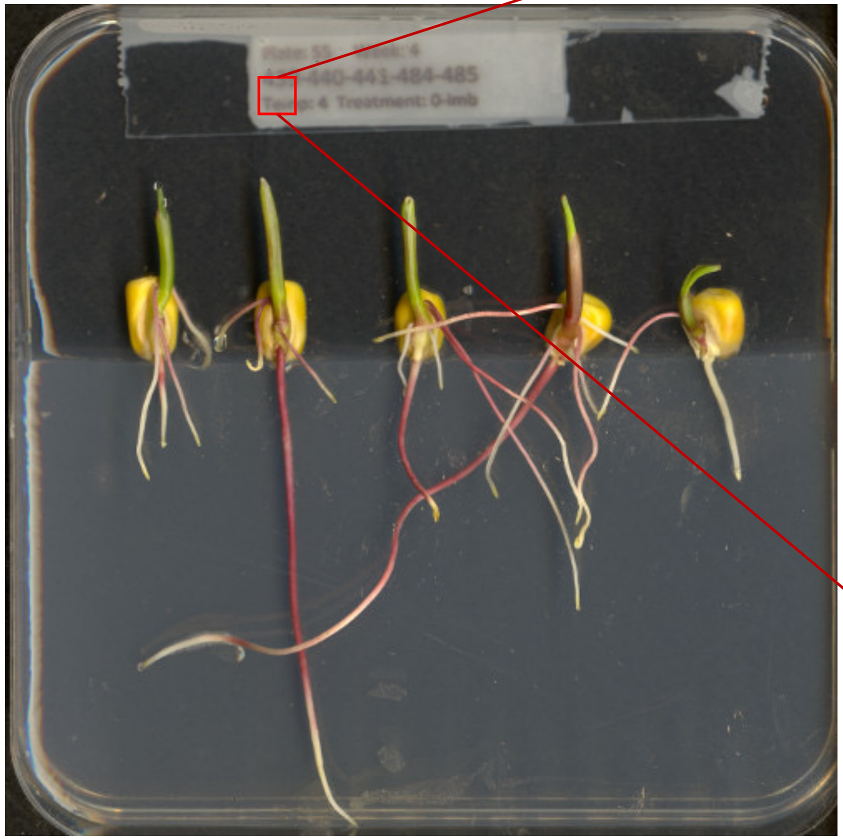


Image Basics

Image formats

Effects of compression in an image:



Compressed JPG

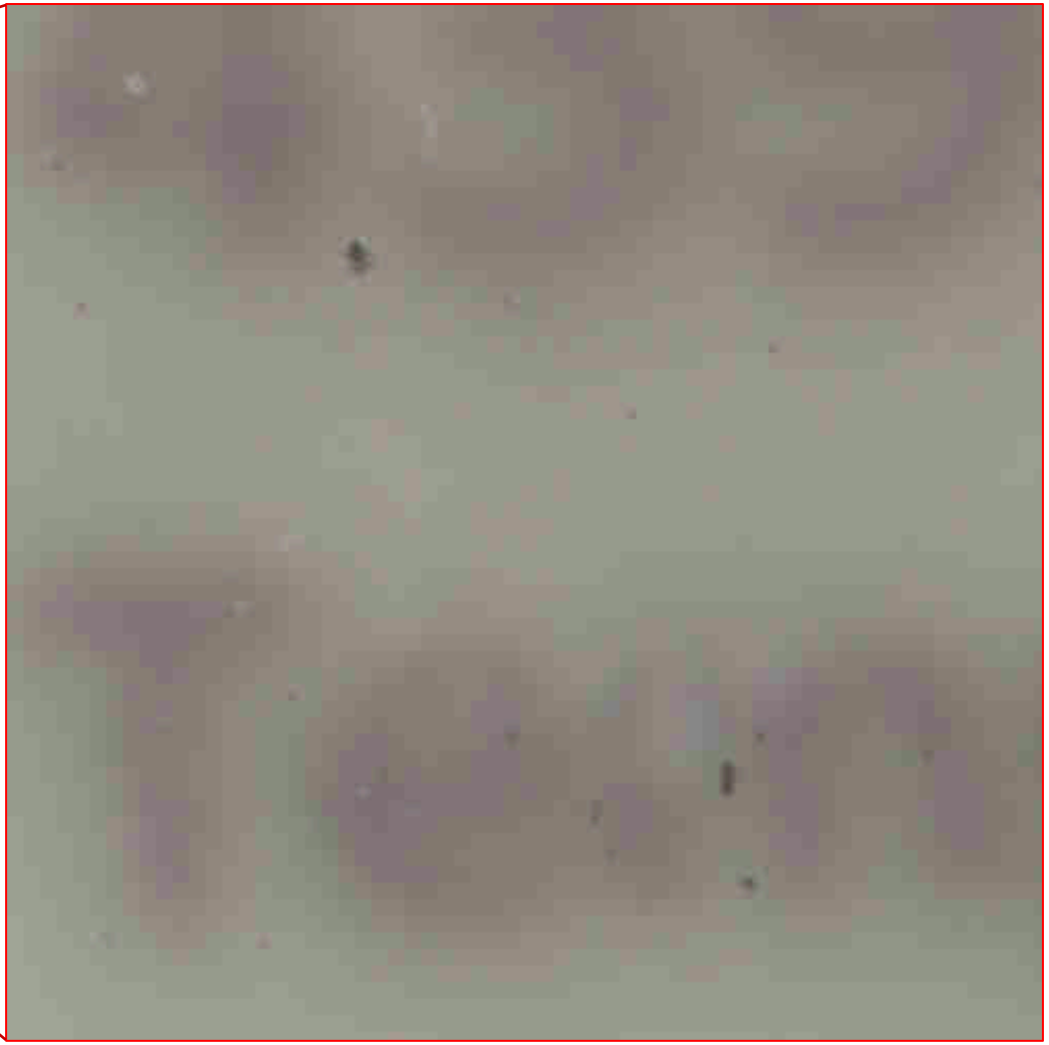


Image Basics

Image metadata

Some image formats (like JPEG and TIFF) support the inclusion of metadata (along with the image data).

Metadata is textual information about the image itself, such as time when the image was created, location, technical information about the camera taking the picture, etc.

Working with Numpy

- Image I/O
  - read image
  - write image
- Matplotlib
  - show image
- Numpy
  - image as array

Working with Numpy

Image I/O

To read and write images we will use the *imageio* Python library, and visualize it using Matplotlib.

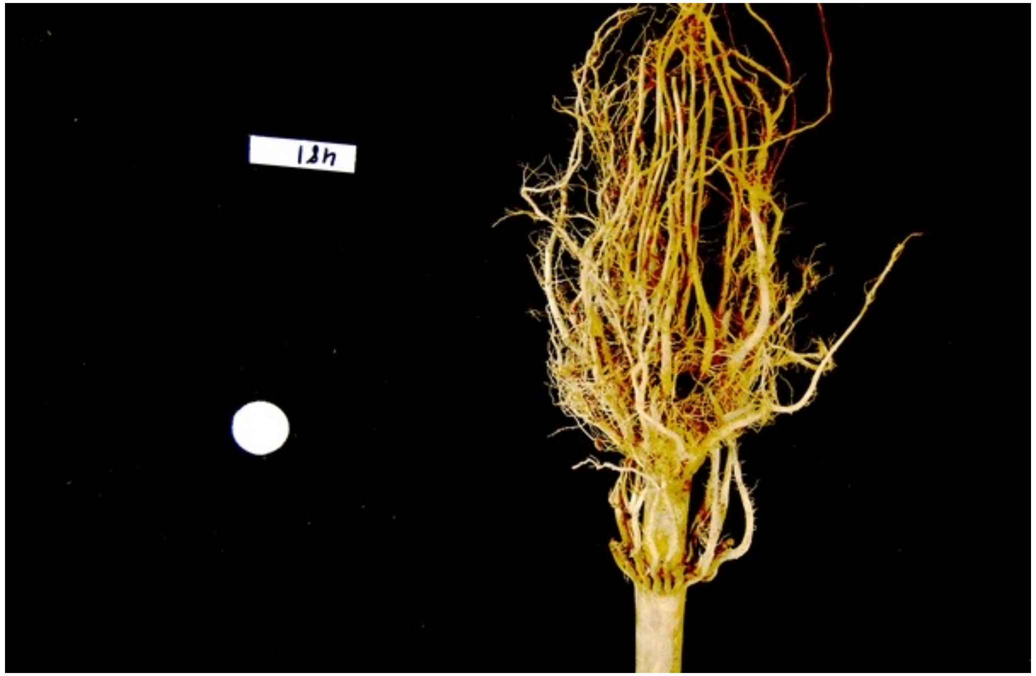
```
> import imageio.v3 as iio
> import matplotlib.pyplot as plt
>
> filename = "maize-root-cluster.jpg"
> image = iio.imread(uri=filename)
> plt.imshow(image)
>
> # save as a TIF file
> iio.imwrite("roots.tif", image)
```

Working with Numpy

Manipulating colours

Once we have the image in memory (as an array) we can manipulate the pixels. For example, let's convert all (dark) background into black.

```
> import imageio.v3 as iio
> import matplotlib.pyplot as plt
>
> filename = "maize-root-cluster.jpg"
> image = iio.imread(uri=filename)
>
> image[image < 128] = 0
>
> plt.imshow(image)
```

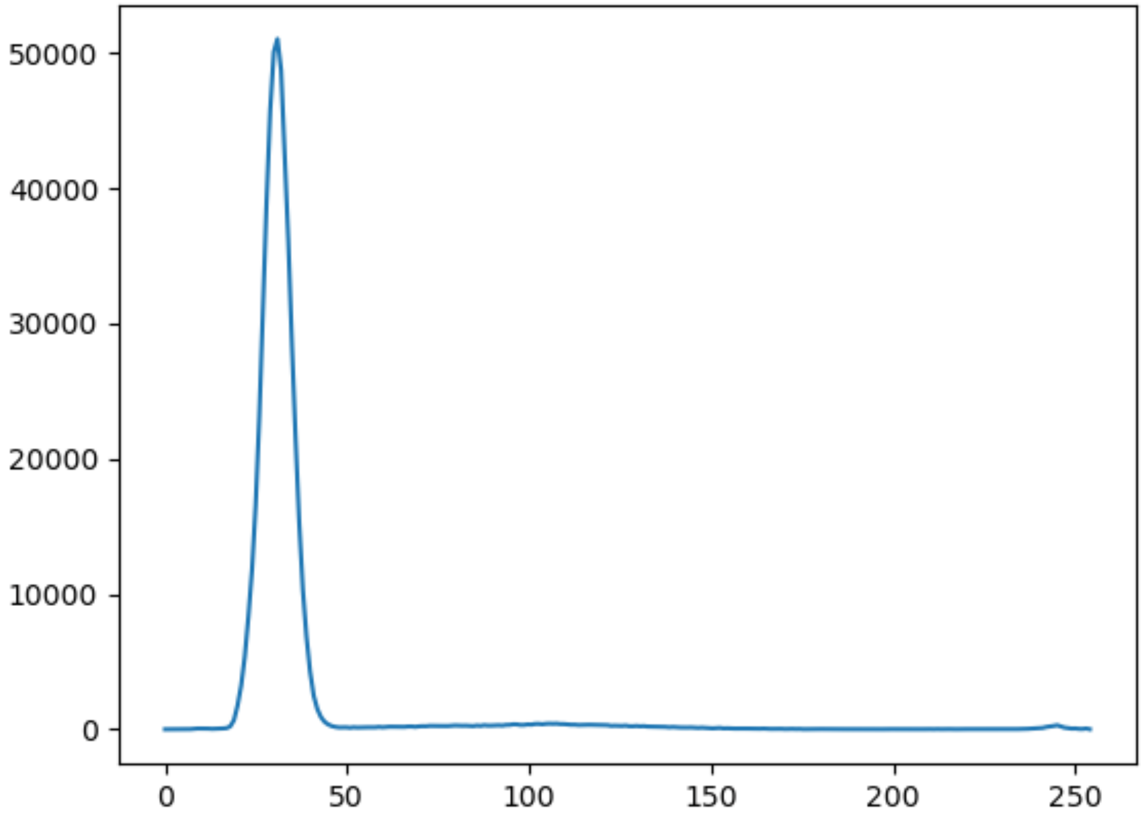


Working with Numpy

Histograms

Intensity/Colour histograms play an important role in image processing, giving a deeper understanding of image through their light distribution.

```
> filename = "plant-seedling.jpg"
> image = iio.imread(uri=filename,
                         mode="L")
>
> counts, edges = np.histogram(
      image.flatten(),
      bins=256,
      range=(0,256))
>
> plt.plot(edges[:-1], counts)
```

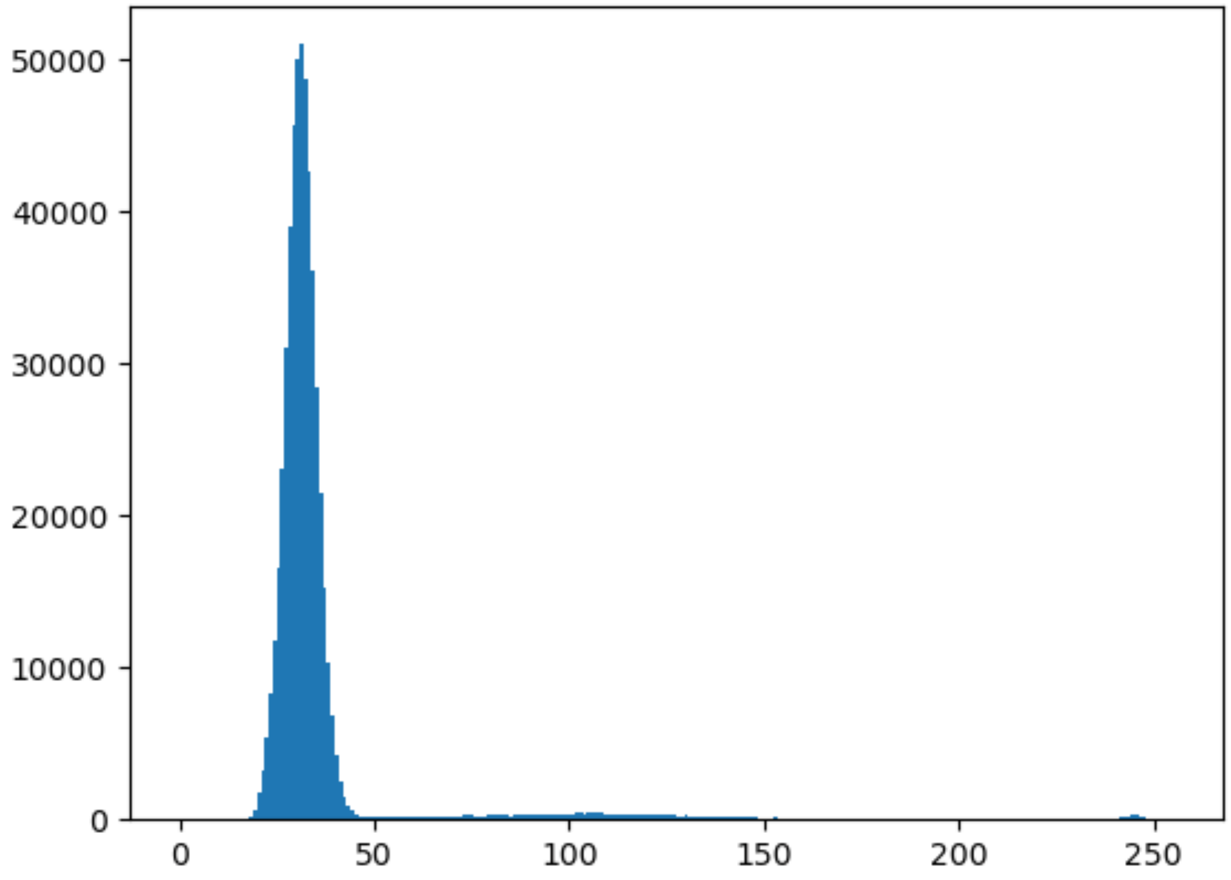


Working with ~~Numpy~~ Matplotlib

Histograms

… we can also use Matplotlib (directly) to plot our image’s histogram:

```
> filename = "plant-seedling.jpg"
> image = iio.imread(uri=filename,
                     mode="L")
>
> _ = plt.hist(
      image.flatten(),
      bins=256,
      range=(0,256)
      )
```

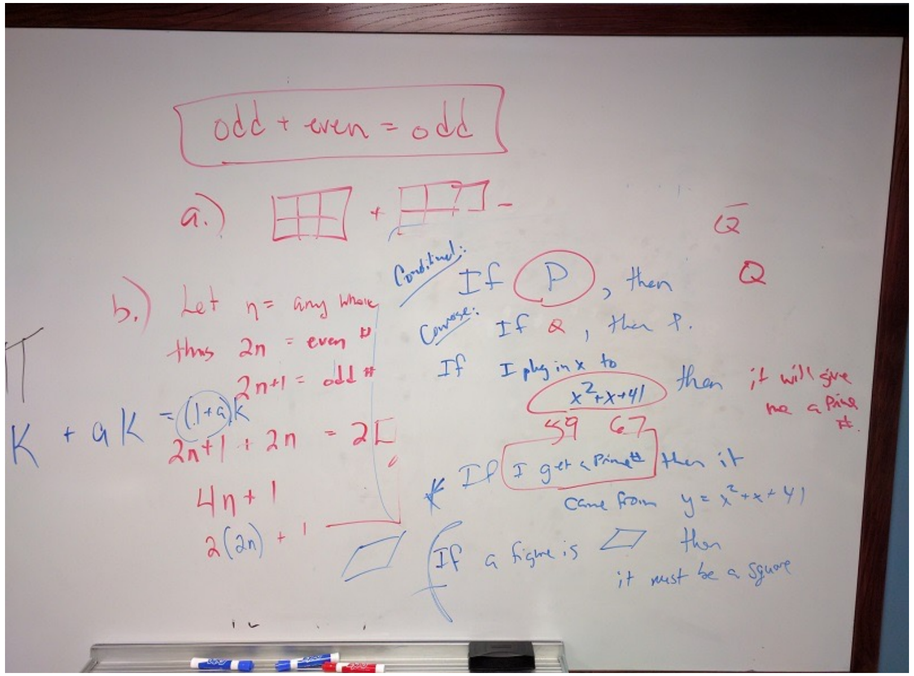


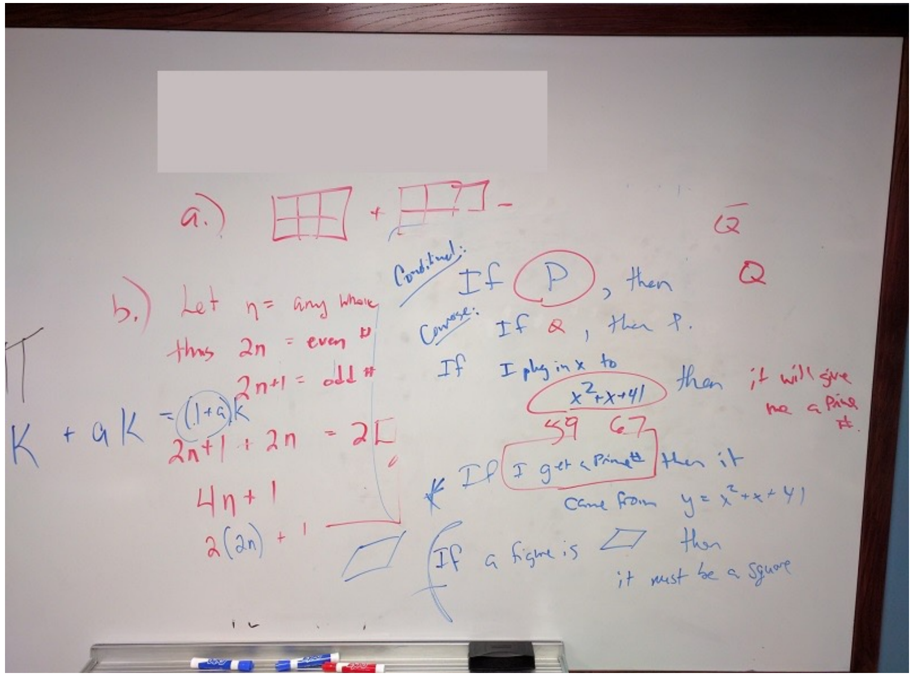
Working with Numpy

Manipulating colours

We can also clip parts of an image:

```
> filename = "board.jpg"
> image = iio.imread(uri=filename)
> fig, ax = plt.subplots()
> ax.imshow(image)
>
> clip = image[60:151, 135:481, :]
> color = image[330, 90]
>
> image[60:151, 135:481] = color
> fig, ax = plt.subplots()
> ax.imshow(image)
```





Working with Skimage

Manipulating colours

We can use another library, scikit-image, to do image transformation. For example, to convert a colour image into grayscale.

```
> from skimage.color import rgb2gray
>
> filename = "maize-root-cluster.jpg"
> image = iio.imread(uri=filename)
>
> gray_img = rgb2gray(image)
>
> plt.imshow(gray_img, cmap="gray")
```

Working with Skimage

Resizing images

Skimage provides tools for image transformations such as resizing and conversion between datatypes (eg, from float to unsigned integer).

```
> from skimage.transform import resize
> from skimage.util import img_as_ubyte
>
> new_shape = (
      image.shape[0] // 2,
      image.shape[1] // 2,
      image.shape[2]
)
> small = resize(image, new_shape)
> small = img_as_ubyte(small)
```

Working with Skimage

Masking

In many situations in image processing we want to retain or manipulate only some parts of the image. We do that by using *masks*.

Masks are Boolean arrays where the “pixels” in the regions (or patches) of interest are valued True (False) whereas the rest are False (True).

The way the pixels (or array elements) of interest are defined varies according to the problem, you can, for instance, define the elements through colour/value selection (like we did when we turned all dark pixels into black “0”), or you can use a pre-defined list of [y,x] positions.

Working with Skimage

Masking

If we want to mask a certain region of the image and we want to create a list of the [y,x] elements in that area we can use skimage drawing tools to accomplish that.

Suppose we want to retain only the roots of the seeds in the following image:

```
> filename = "maize-seedlings.tif"
> image = iio.imread(uri=filename)
>
> fig, ax = plt.subplots()
> ax.imshow(image)
```

Working with Skimage

Masking

Suppose we want to retain only the roots of the seeds in the following image, we can define the rectangle covering the region through a mask and manipulate the image as follows:

```
> from skimage.draw import rectangle
>
> mask = np.ones(image.shape[0:2],
                 dtype=bool)
>
> rr, cc = rectangle(start=(357, 44),
                     end=(740, 720))
>
> mask[rr,cc] = False
> image[mask] = 0
```

Segmentation

Segmentation is the process of dividing an image in regions of interest, it is central to identification of objects/content in images. The whole segmentation process is composed by the following stages:

- pre-processing: prepare the image (eg, smoothing, normalization);
- processing: apply strategy to split image into regions or layers;
- post-processing: filter and identify objects of interest.

Segmentation

Blurring images

The blurring of images is usually a necessary step as it smooths differences (eg, object border) between neighbour pixels, and distribute imperfections (eg, noise) among nearby pixels.

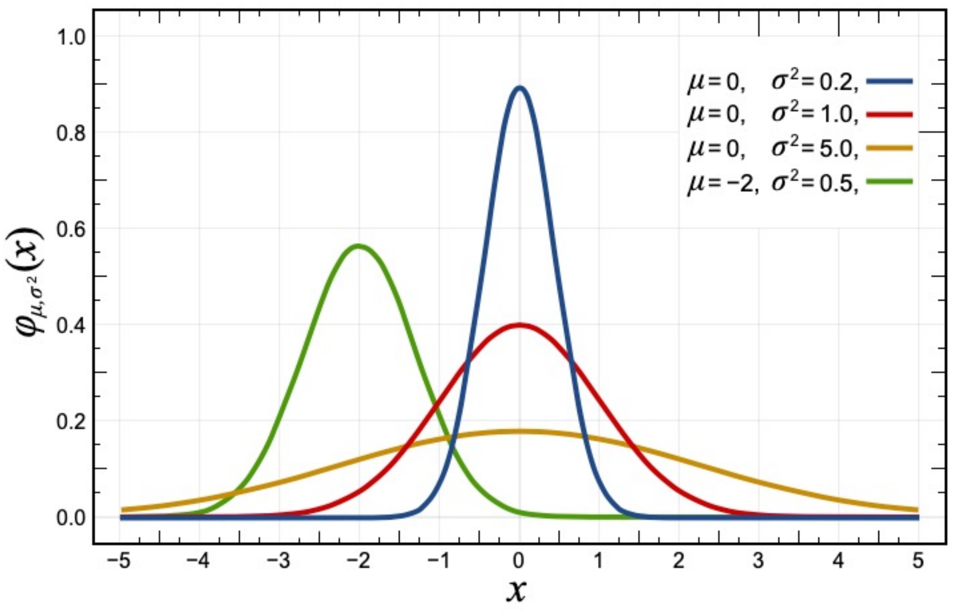
In signal processing, “blurring” is called a *low-pass filter*. On the other hand, *high-pass filters* are used to *enhance* differences (eg, borders) between nearby pixels.

When we talk about “blurring” an image, we usually mean the use of a *Gaussian filter*. A Gaussian filter is very similar (effect) to a simple *mean filter*, but the gaussian preserves better the image information.

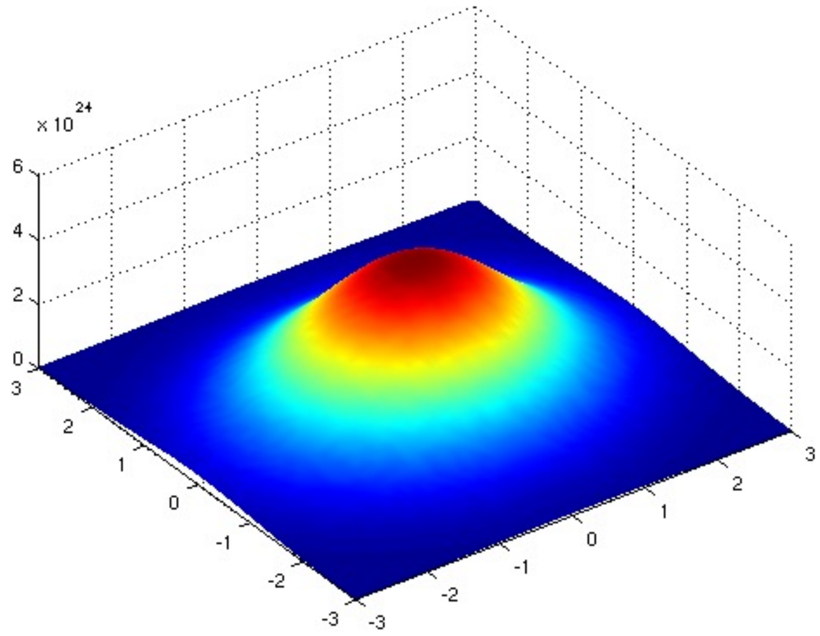
Segmentation

Gaussian blur

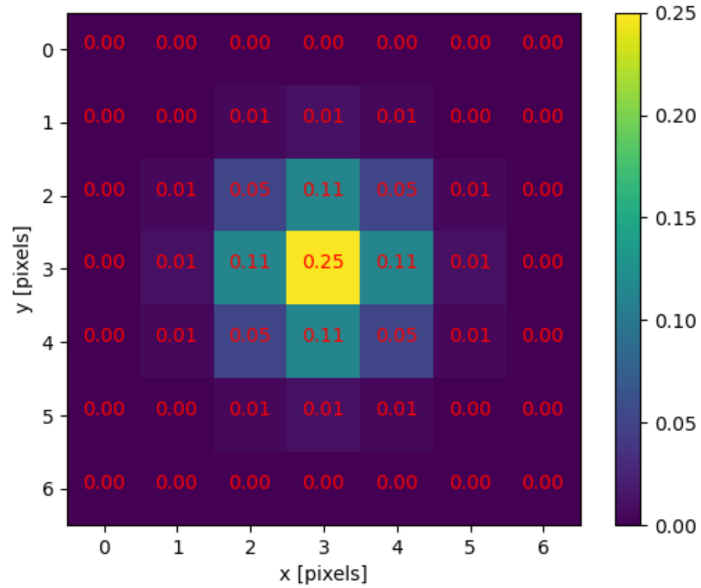
To refresh our memory, how Gaussian curves/filter look like:



Curves with varied mean/std-dev values

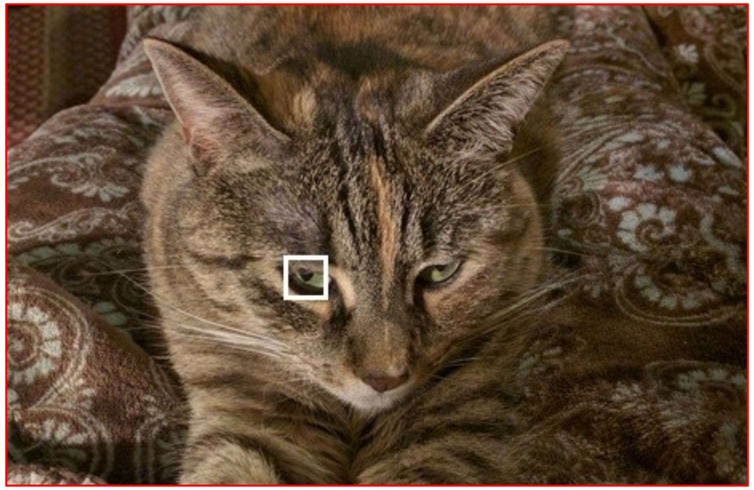


3-dimensional surface



7x7 array kernel

Segmentation
Gaussian blur



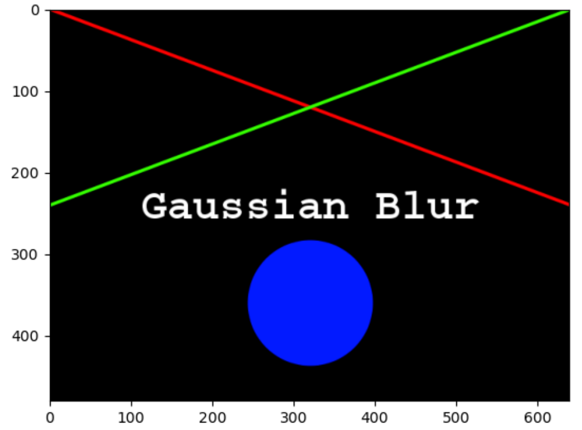
Animations of blurring kernel being applied:

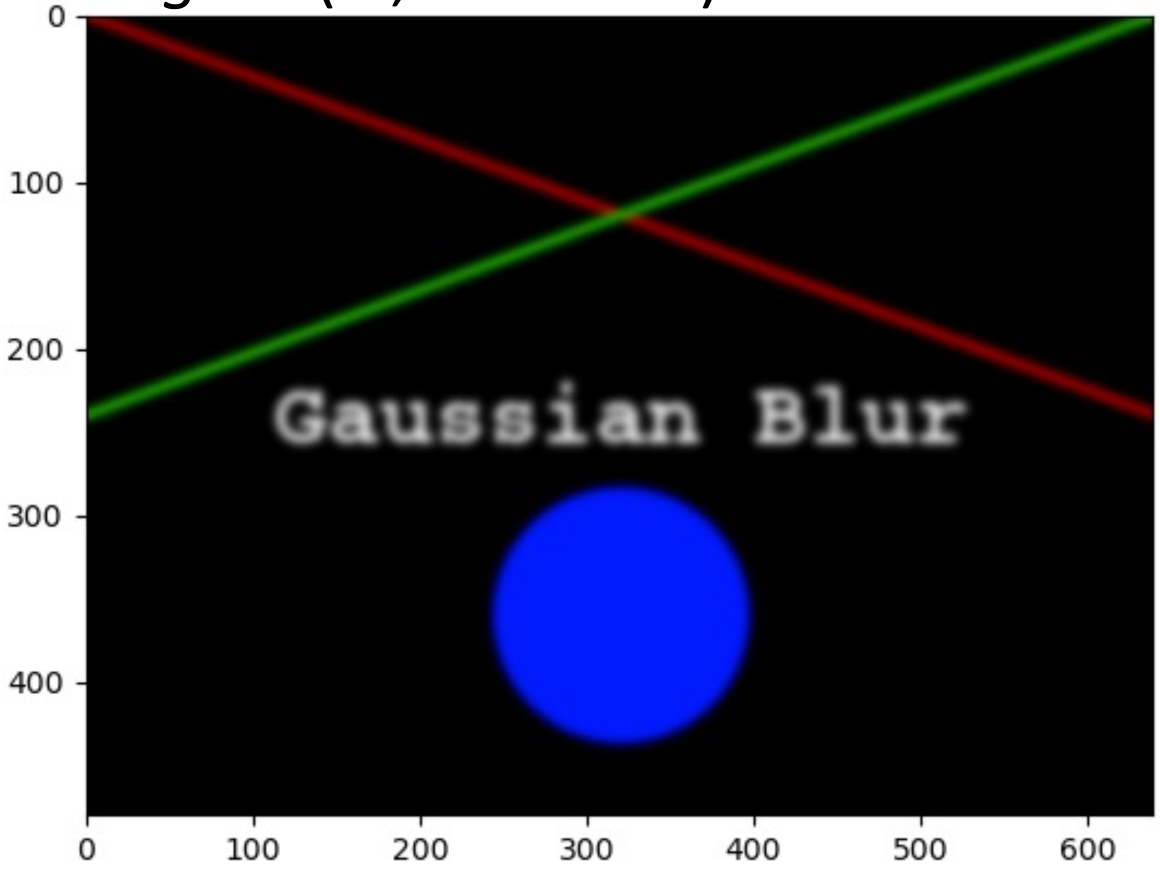
Segmentation

Gaussian blur - Skimage

Skimage has a whole module dedicated to filters, *gaussian* is one. See how results change according to different *sigma* (ie, filter size) values.

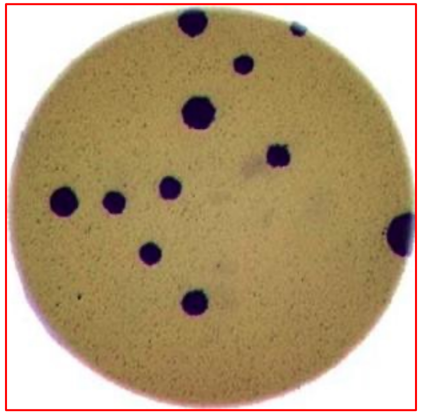
```
> from skimage.filters import gaussian
>
> image = iio.imread("gaussian.png")
> blur = gaussian(image, sigma=3)
```



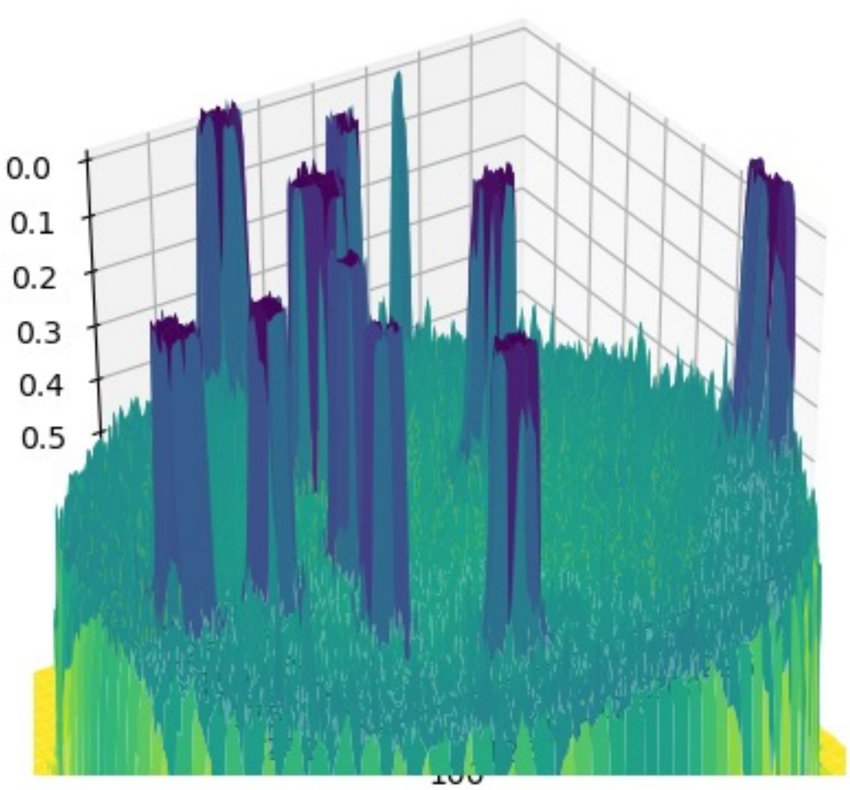


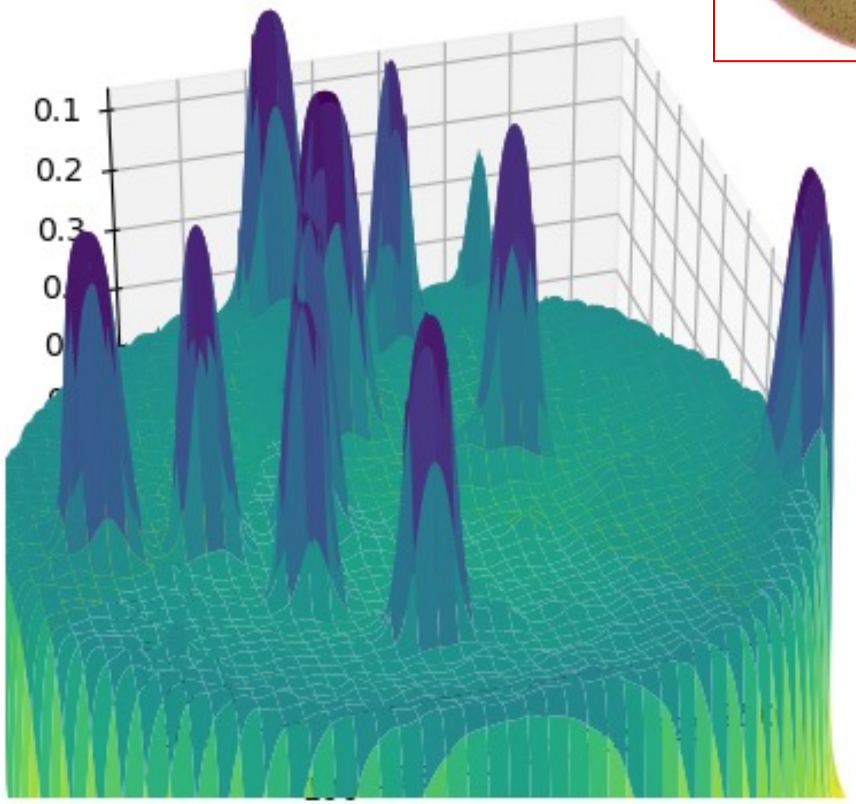
Segmentation

Gaussian blur – 3d View



Just so we have a better view of the effect of blurring, a 3D view of the “petri dish” image (after a blur):





Segmentation

Thresholding

The simplest – yet powerful – way to split an image's content is through the use of threshold method.

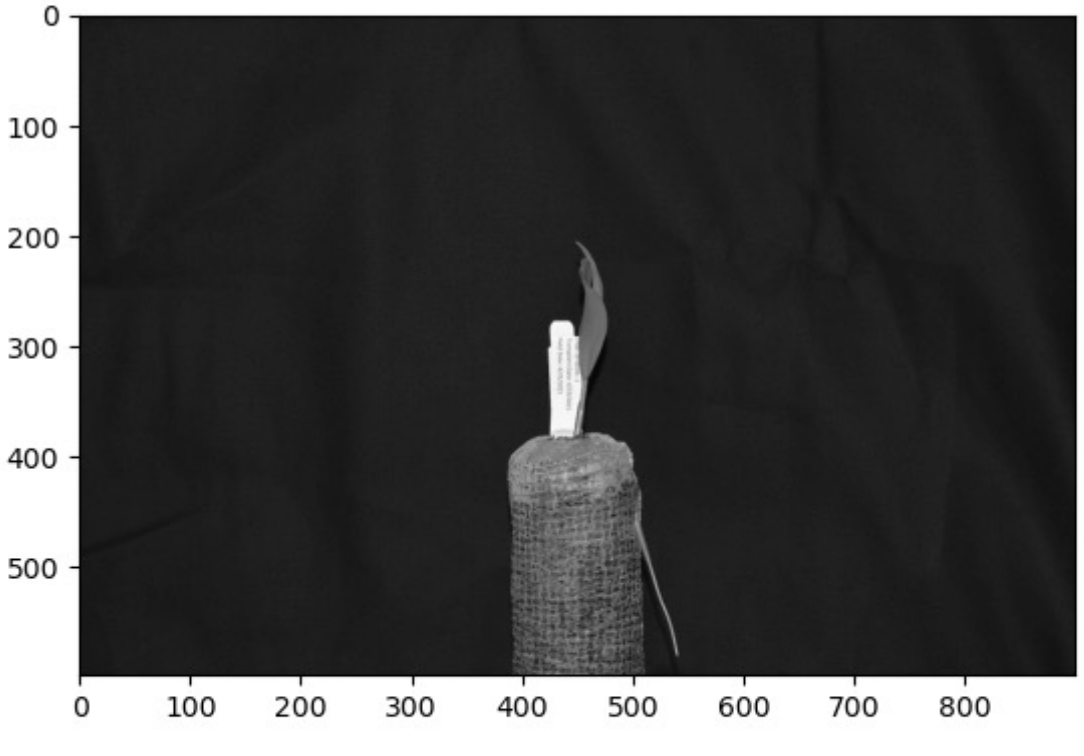
In this method, a value "t" in image's intensity range is used to mark the pixels (with intensity) "above V" and the pixels "below t".

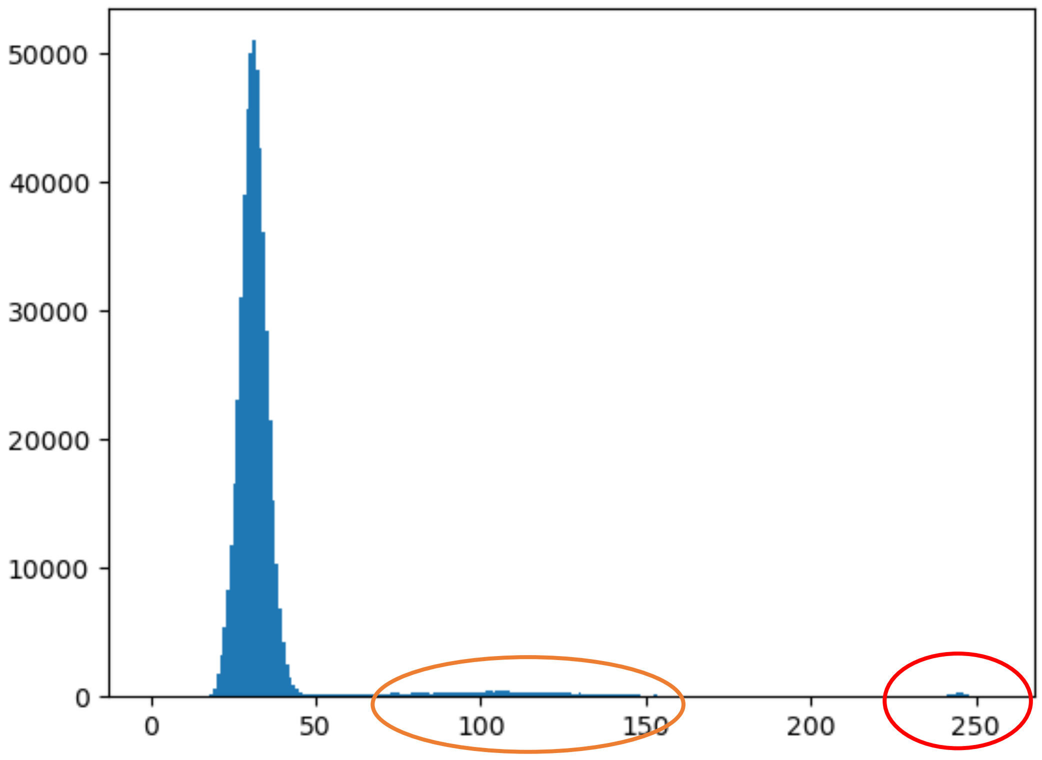
Such value "t" may be chosen through different methods and is usually niche specific. In most of the cases/images, though, the method known as *Otsu's method* is usually a good first-guess.

Segmentation

Thresholding

For example, consider the histogram section of this presentation, we used the “seedling” image to draw the following histogram:



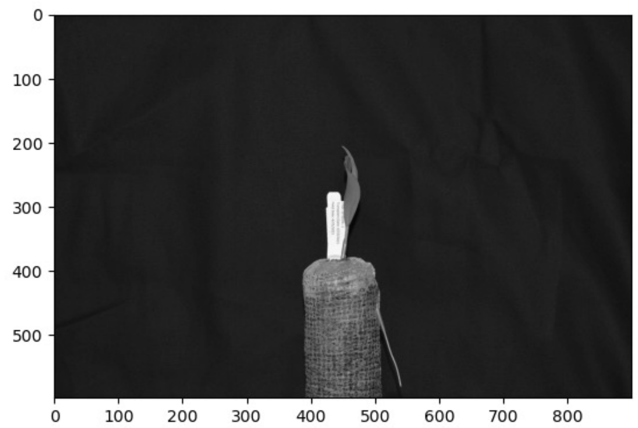


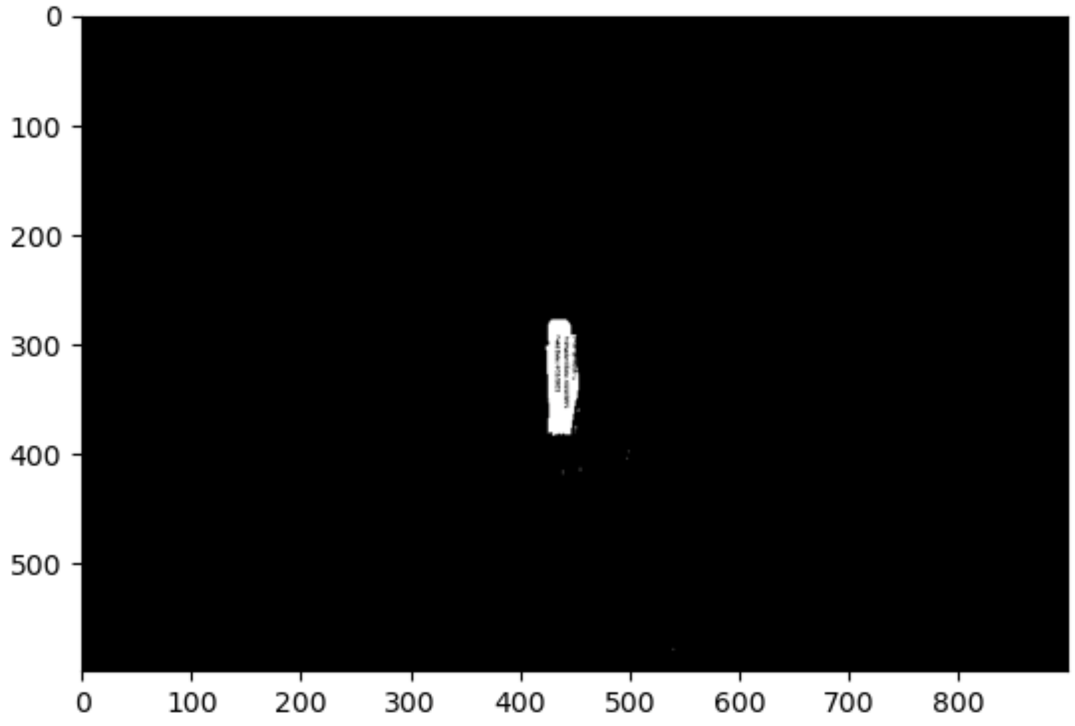
Segmentation

Thresholding

Let’s say that every pixel with intensity below t=200 should be turned black and every pixel above, white (in reality, False/True)

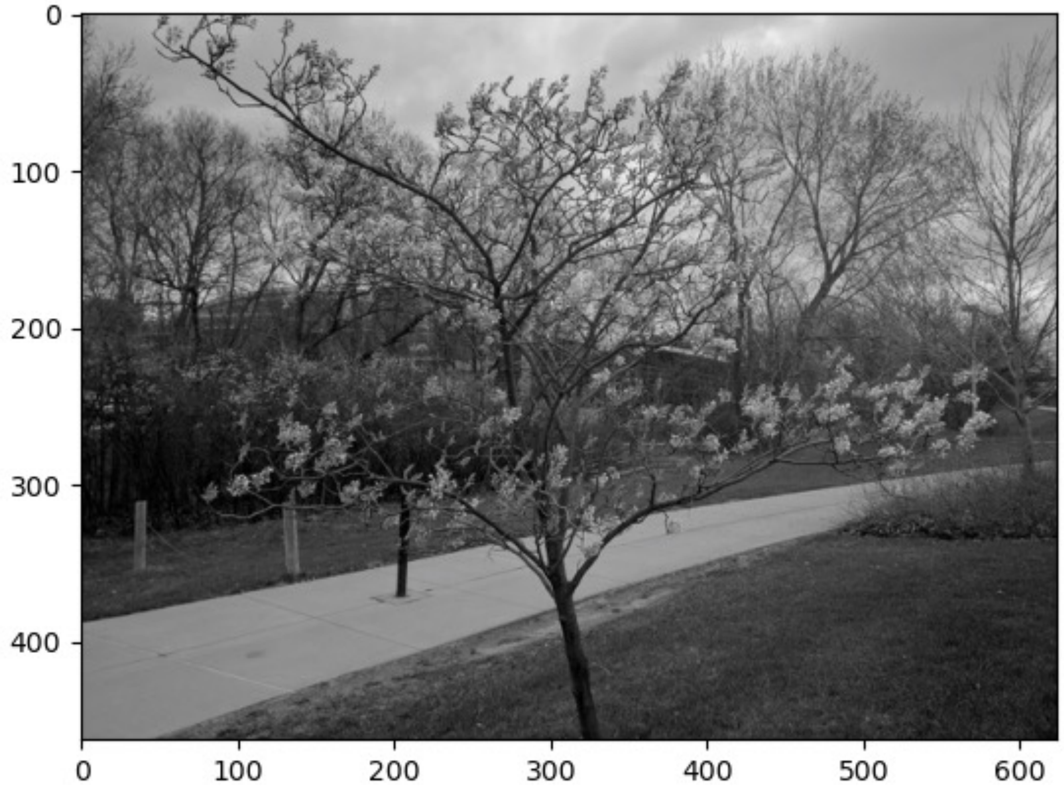
```
> image = iio.imread("seedling.jpg",
                     mode="L")
> t = 200
> mask = image > t
```

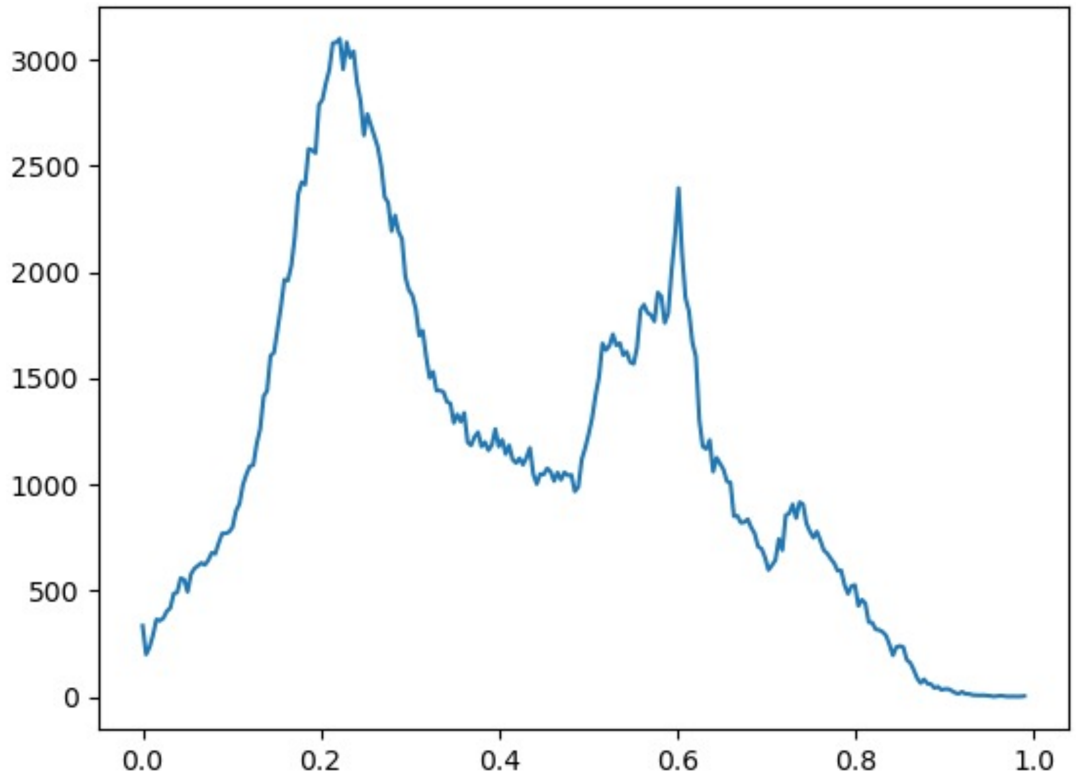




Segmentation
Otsu's thresholding

In some cases, may not be that simple to define the threshold value "t".



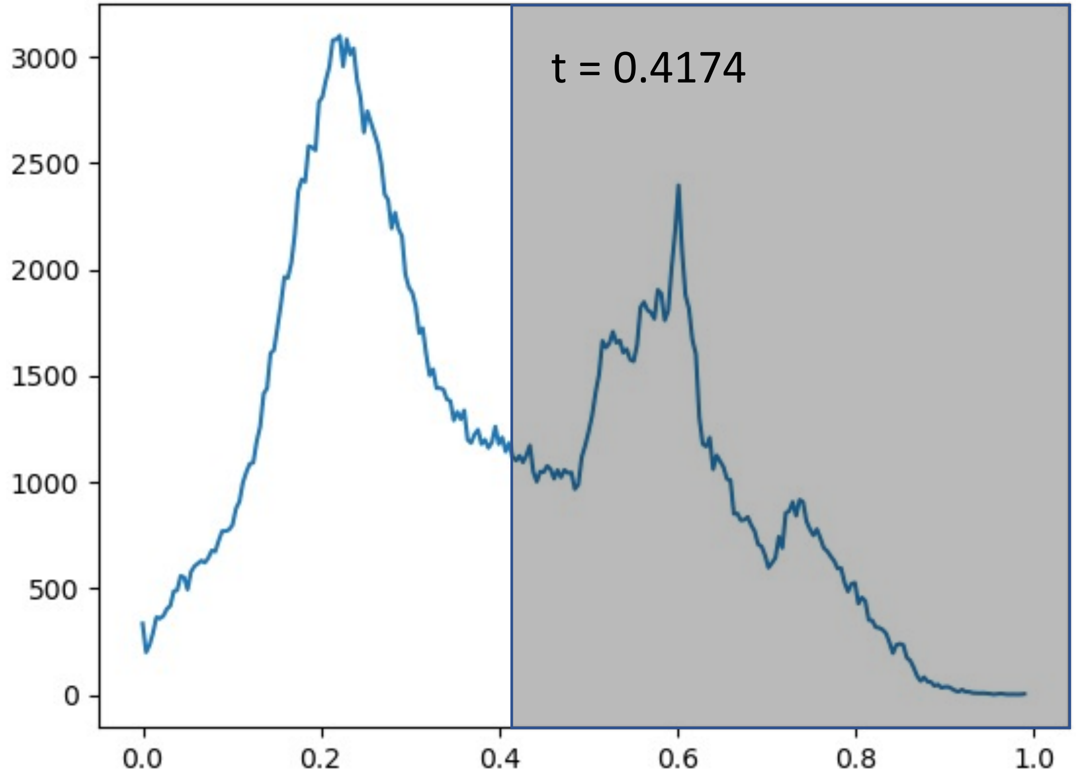


Segmentation

Otsu's thresholding

The Otsu's method automate the process of defining "t" by looking for the (intensity) value that maximizes the information on each class.

```
> from skimage.filters import (
      threshold_otsu
      )
>
> image = iio.imread("tree.jpg",
                     mode="L")
>
> t = threshold_otsu(image)
>
> mask = image > t
> plt.imshow(mask)
```

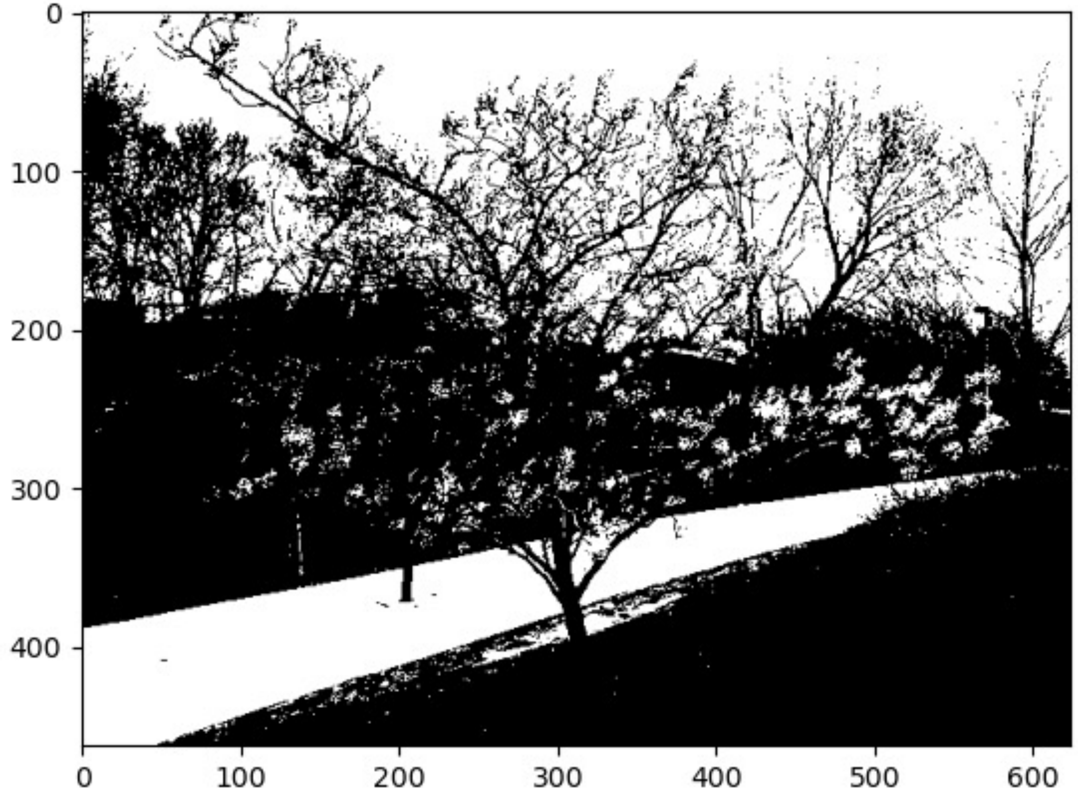


Segmentation

Otsu’s thresholding

The Otsu’s method automate the process of defining “t” by looking for the (intensity) value that maximizes the information on each class.

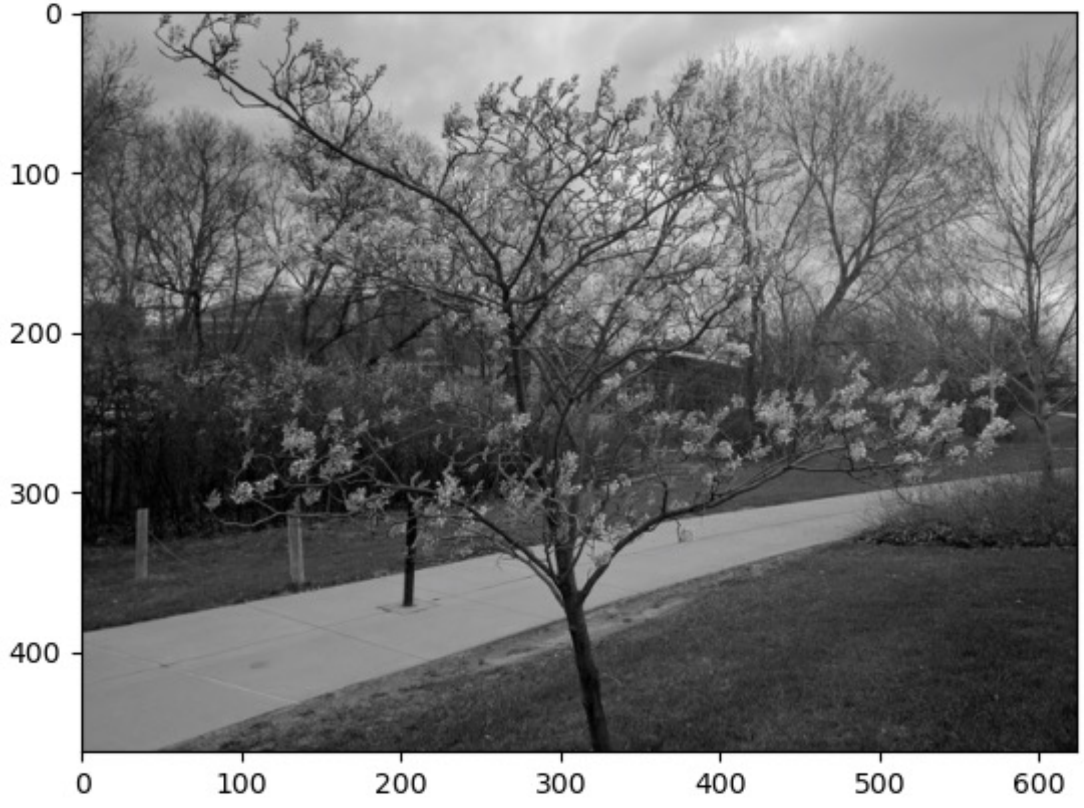
```
> from skimage.filters import (
      threshold_otsu
      )
>
> image = iio.imread("tree.jpg",
                     mode="L")
>
> t = threshold_otsu(image)
>
> mask = image > t
> plt.imshow(mask)
```

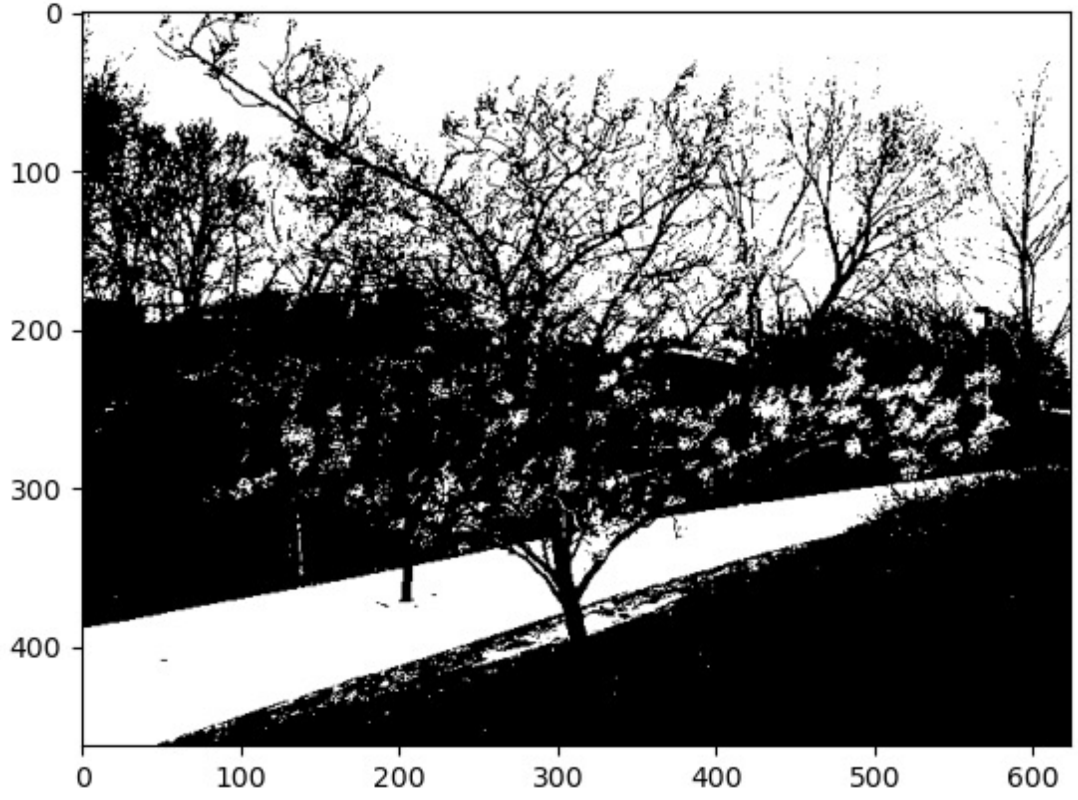


Segmentation

Otsu's thresholding

The [Otsu's method]() automate the process of defining "t" by looking for the (intensity) value that maximizes the information on each class.

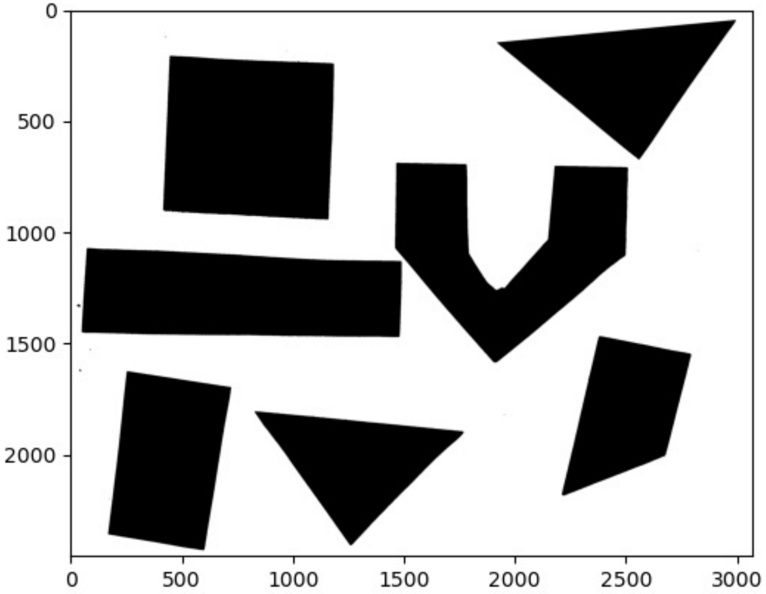


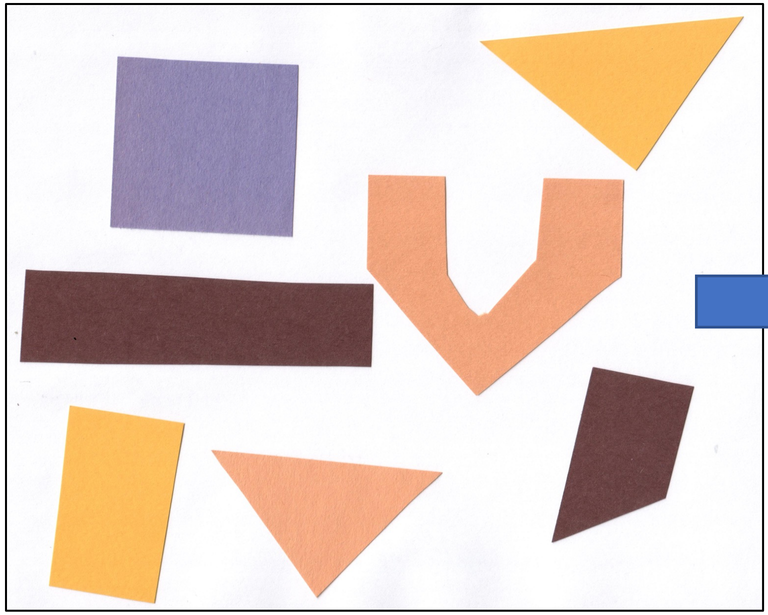


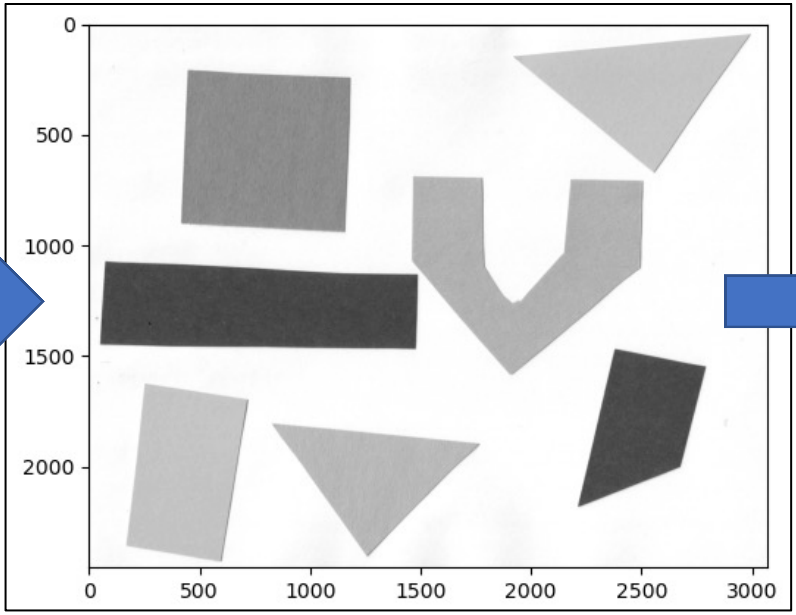
Segmentation

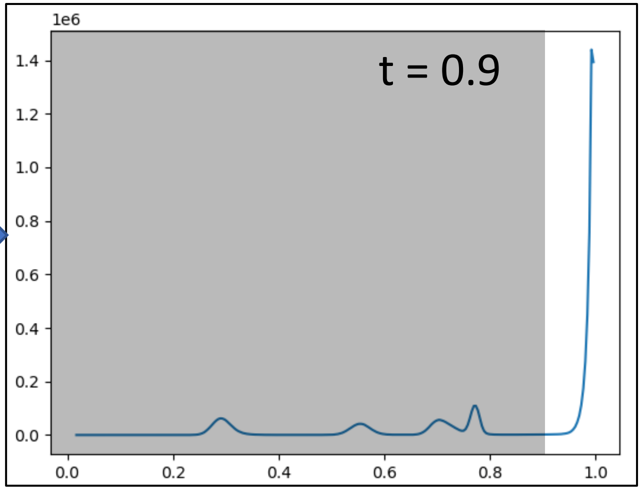
Object identification

We learned separate foreground (objects) from background pixels:









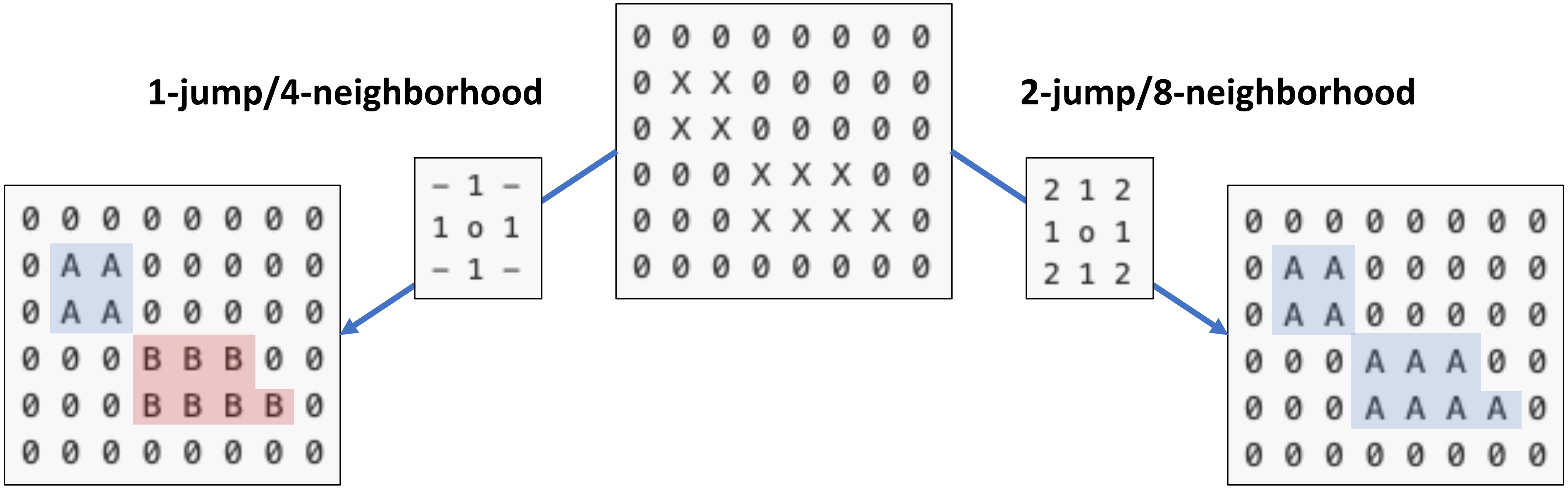
Segmentation

Object identification

We learned separate foreground (objects) from background pixels.

The question now is how to identify each one (‘A’, ‘B’, ‘C’, …).

To do that we use a technique called Connected Component Analysis.

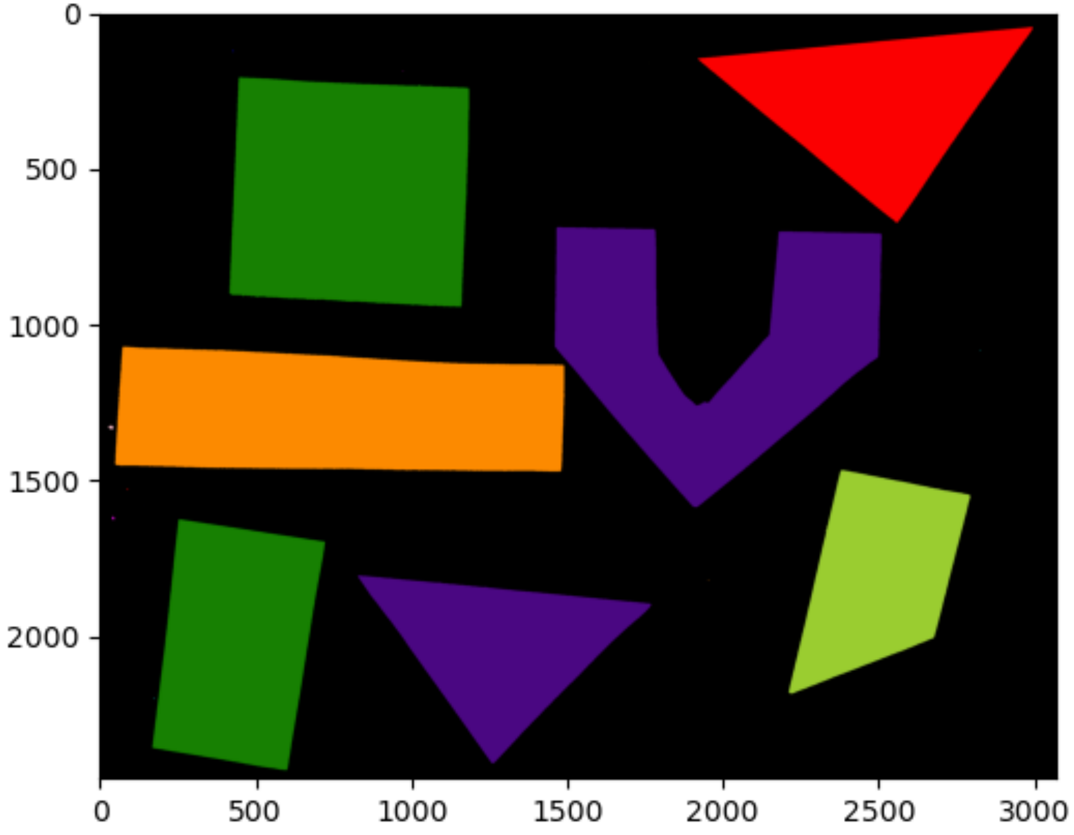


Segmentation

Object identification

Skimage provides identification of objects that we can use to *label* a binary image into individual objects.

```
> from skimage.measure import label
> from skimage.color import label2rgb
>
> lbl_image = label(binary_mask,
                     connectivity=2)
>
> rgb_lbl_image = label2rgb(lbl_image)
>
> plt.imshow(rgb_lbl_image)
```



Segmentation

Morphometrics

Finally, we get to measure each of the identified objects. To do so, we use the image return from the labelling function (`lbl_image`) so we proceed to analyse and filter and select the objects we want.

```
> from skimage.measure import regionprops
>
> obj_features = regionprops(lbl_image)
>
> obj_lbl_area = [(o["label"],o["area"]) for o in obj_features]
>
> print(obj_lbl_area)
[(1, 317791.0), (2, 15.0), (3, 1.0), (4, 5.0), (5, 521173.0), (6, 494815.0), (7,
514227.0), (8, 6.0), (9, 137.0), (10, 254455.0), (11, 16.0), (12, 1.0), (13, 1.0),
(14, 57.0), (15, 338003.0), (16, 264905.0), (17, 8.0), (18, 8.0)]
```

Homework

Homework

- The data files used in the following exercises are provided at this week's course material as `data.zip` (Moodle/Teams), or directly from Figshare:
  - https://figshare.com/articles/dataset/Data_Carpentry_Image_Processing_Data_beta_/19260677

- The following (Python) libraries are necessary:
  - Scikit-Image
  - Matplotlib
  - Numpy

See https://datacarpentry.org/image-processing/setup/ for details.

Homework

Basics

1. Load “8” image, file “data/eight.tif”, and simply change the value of pixels so you have what looks like a 5 instead of an 8.
   Display the image and print out the matrix (ie, array) as well.
2. Suppose we represent colours as triplets (R,G,B) of integer values between [0:255], what are the following colours:
   a. (255, 0, 0)
   b. (0, 255, 0)
   c. (0, 0, 255)
   d. (255, 255, 255)
   e. (0, 0, 0)
   f. (128, 128, 128)
3. How many colours can be represented in a 24-bit RGB system?

Homework

Working with Numpy

1. Turn the (white) background of image “data/sudoku.png” into gray.
2. Consider image “data/maize-root-cluster.jpg”, clip the roots/plant from the (larger) image, display it, and save it as “data/clip.tif”.
3. Make the colour histogram (ie, counts for R, G, and B, in the same) of image “data/tree.jpg”.

Homework

Working with Numpy

1. Turn the (white) background of image “data/sudoku.png” into gray.
2. Consider image “data/maize-root-cluster.jpg”, clip the roots/plant from the (larger) image, display it, and save it as “data/clip.tif”.
3. Make the colour histogram (ie, counts for R, G, and B, in the same) of image “data/tree.jpg”.

Homework

Blurring and Thresholding

1. Try the Gaussian filter function from Skimage. See the (visual) effects of different combinations of ‘sigma’, ‘truncate’, ‘multichannel’ parameters using “gaussian-original.png” image.
2. Create a black/white image (aka, *binary image*) using thresholding method to separate the objects (*True*) from the background (*False*) in image “data/shapes-02.jpg”.
3. Create a black/white image (aka, *mask*) to separate (foreground) bacteria colonies from the background in image “colonies-1.tif”:
   a. Using a manually defined (through histogram) “t” value;
   b. Using an automatically defined “t” value through Otsu’s method.

Homework

Connected Component Analysis

1. How many objects/regions would be defined in the following image/array, using
   a. 4-neighborhood (1-jump) strategy?
   b. 8-neighborhood (2-jumps) strategy?

```
0 0 0 0 0 0 0 0
0 X 0 0 0 X X 0
0 0 X 0 0 0 0 0
0 X 0 X X X 0 0
0 X 0 X X 0 0 0
0 0 0 0 0 0 0 0
```

2. Re-apply the segmentation and labelling steps we saw in the “Segmentation/Object Identification” slides but, now, *before transforming the labelled image into RGB* (‘label2rgb’), <u>re-label the objects in (a copy of) the image using their respective areas.</u>

3. Also, does the number of objects detected match your expectations? Why, and how to fix it?

Homework

- Solve the problem – Capstone Challenge – at:
  - https://datacarpentry.org/image-processing/09-challenges/index.html

You will apply all the steps we went through in this lesson to count the number of bacteria colonies in petri-dish images

References

- “Image Processing with Python”, The Data Carpentries, https://datacarpentry.org/image-processing