南開大學
Nankai University

# Computer Vision

**Project Topic: Image Segmentation and Superimposition using Thresholding in Computer Vision**

---


## Group Members:

BARRIE AMADU WURIE ID: 2120236035

JALLOH EJATU ID: 2120236036

KAMARA FATMATA N. ID: 2120236040

NYALLAY SHEKU ID: 2120236041

BANGURA EMMANUEL ID: 2120236038


Date: 21th December 2023

## Introduction

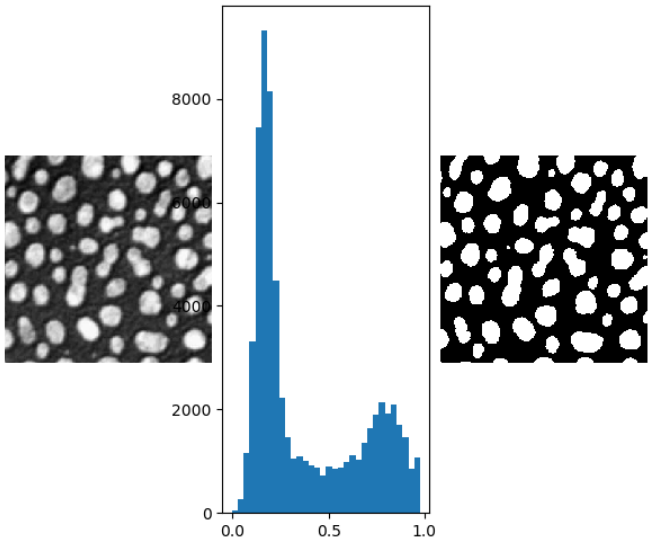
In digital image processing, thresholding is the simplest method of segmenting images. It plays a crucial role in image processing as it allows for the segmentation and extraction of important information from an image. By dividing an image into distinct regions based on pixel intensity or pixel value, thresholding helps distinguish objects or features of interest from the background. This technique is widely used in various applications such as object detection, image segmentation, and character recognition, enabling efficient analysis and interpretation of digital images. Additionally, image thresholding can enhance image quality by reducing noise and improving overall visual clarity.

Thresholding methods encompass a variety of approaches, such as global thresholding, where a single threshold value is applied to the entire image, or adaptive thresholding, which adjusts the threshold value for different regions of an image to handle variations in lighting or contrast.
The utilization of thresholding in image segmentation holds significant relevance across various applications. It aids in object extraction, noise reduction, and feature enhancement within images. By separating regions based on intensity or other attributes, thresholding facilitates the isolation of specific objects or areas of interest, laying the groundwork for further analysis, recognition, or processing.

### What is Image Thresholding?

Image thresholding involves dividing an image into two or more regions based on intensity levels, allowing for easy analysis and extraction of desired features. By setting a threshold value, pixels with intensities above or below the threshold can be classified accordingly This technique aids in tasks such as object detection, segmentation, and image enhancement.

Image thresholding is a technique that simplifies a grayscale image into a binary image by classifying each pixel value as either black or white based on its intensity level or gray-level compared to the threshold value. This technique reduces the image to only two levels of intensity, making it easier to identify and isolate objects of interest.




In the above image showing particles embedded in a substrate, the particles are brighter (more white) than the background. This is also clearly visible in the histogram. On the right in the above figure the thresholded image is shown. The threshold value is manually selected at ***t* =0.5**

**Applications of Image Thresholding**

Image thresholding is a technique used in computer vision that has a variety of applications, including image segmentation, object detection, and character recognition. By separating objects from their background in an image, image thresholding makes it easier to analyze and extract relevant information. Optical character recognition (OCR) systems, for example, use image thresholding to distinguish between foreground (text) and background pixels in scanned documents, making them editable.

## Real-world applications

**Object Detection:** By setting a threshold value, objects can be separated from the background, allowing for more accurate and efficient object detection.

**Medical Images**: Image thresholding can be used to segment different structures or abnormalities for diagnosis and analysis in medical imaging.

**Quality Control**: Image thresholding plays a crucial role in quality control processes, such as inspecting manufactured products for defects or ensuring consistency in color and texture of a color image.

**Object Segmentation**: Image thresholding is also commonly used in computer vision tasks such as object segmentation, where it helps to separate foreground objects from the background. This enables more accurate and efficient detection of objects within an image.

**Noise Reduction**: Thresholding can be utilized for noise reduction, as it can help to eliminate unwanted artifacts or disturbances in an image.

**Edge Detection**: Image thresholding aids in identifying and highlighting the boundaries between different objects or regions within an image with edge detection algorithms.

However, in this paper, we are going to implement image segmentation and Superimposition using Thresholding. In this section, we will use the image files named cat.jpg and monastery.jpg

cat.jpg

monastery.jpg

First, we crop the cat image to a smaller one to remove some of the background. We can start a new REPL session for this project:

```
from PIL import Image
filename_cat = "cat.jpg"
with Image.open(filename_cat) as img_cat:
    img_cat.load()
img_cat = img_cat.crop((800, 0, 1650, 1281))
img_cat.show()
```

The cropped image contains the cat and some of the background that's too close to the cat for you to crop it:

Each pixel in a color image is represented digitally by three numbers corresponding to the red, green, and blue values of that pixel. Thresholding is the process of converting all the pixels to either the maximum or minimum value depending on whether they're higher or lower than a certain number. It's easier to do this on a grayscale image:

```
img_cat_gray = img_cat.convert("L")
img_cat_gray.show()

threshold = 100
img_cat_threshold = img_cat_gray.point(
    lambda x: 255 if x > threshold else 0)
img_cat_threshold.show()
```

We achieve thresholding by calling **.point()** to convert each pixel in the grayscale image into either 255 or 0. The conversion depends on whether the value in the grayscale image is greater or smaller than the threshold value. The threshold value in this example is 100.

The figure below shows the grayscale image and the result from the thresholding process:

In this example, all the points in the grayscale image that had a pixel value greater than 100 are converted to white, and all other pixels are changed to black. You can change the sensitivity of the thresholding process by varying the threshold value.

Thresholding can be used to segment images when the object to segment is distinct from the background. You can achieve better results with versions of the original image that have higher contrast. In this example, you can achieve higher contrast by thresholding the blue channel of the original image rather than the grayscale image, because the dominant colors in the background are brown and green colors, which have a weak blue component.

We can extract the red, green, and blue channels from the color image as you did earlier:

```
red, green, blue = img_cat.split()
red.show()
green.show()
blue.show()
```

The **red, green**, and **blue** channels are shown below, from left to right. All three are displayed as grayscale images.

The blue channel has a higher contrast between the pixels representing the cat and those representing the background. We can use the blue channel image to threshold:

```
threshold = 57
img_cat_threshold = blue.point(lambda x: 255 if x > threshold else 0)
img_cat_threshold = img_cat_threshold.convert("1")
img_cat_threshold.show()
```

We use a threshold value of 57 in this example. We also convert the image into a binary mode using "1" as an argument to **.convert().** The pixels in a binary image can only have the values of 0 or 1.

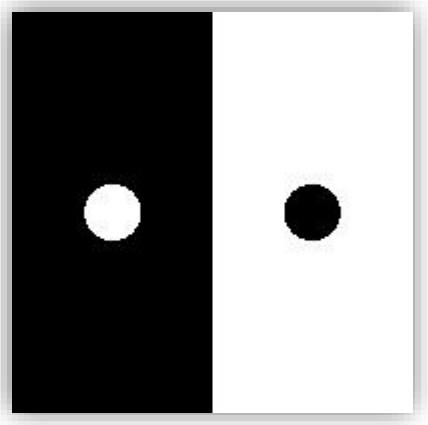
## The result of thresholding is the following:

We can identify the cat in this black-and-white image. However, we'd like to have an image in which all the pixels that correspond to the cat are white and all other pixels are black. In this image, we still have black regions in the area which corresponds to the cat, such as where the eyes, nose and mouth are, and we also still have white pixels elsewhere in the image.

We can use the image processing techniques called erosion and dilation to create a better mask that represents the cat. You'll learn about these two techniques in the next section.

## Erosion and Dilation

We can look at the image file called ***dot_and_hole.jpg***



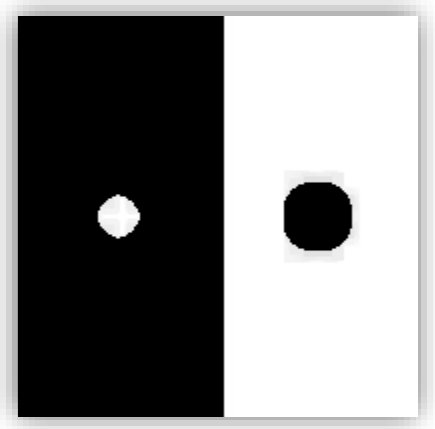
The left-hand side of this binary image shows a white dot on a black background, while the right-hand side shows a black hole in a solid white section.

**Erosion** is the process of removing white pixels from the boundaries in an image. We can achieve this in a binary image by using **ImageFilter.MinFilter(3)** as an argument for the **.filter()** method. This filter replaces the value of a pixel with the minimum value of the nine pixels in the 3x3 array centered around the pixel. In a binary image, this means that a pixel will have the value of zero if any of its neighboring pixels are zero.

We can see the effect of erosion by applying **ImageFilter.MinFilter(3)** several times to the **dot_and_hole.jpg** image. We should continue with the same REPL session as in the previous section:

```
from PIL import ImageFilter
filename = "dot_and_hole.jpg"
with Image.open(filename) as img:
    img.load()
for _ in range(3):
    img = img.filter(ImageFilter.MinFilter(3))
img.show()
```

We've applied the filter three times using a for loop. This code gives the following output:



The dot has shrunk but the hole has grown as a result of erosion.

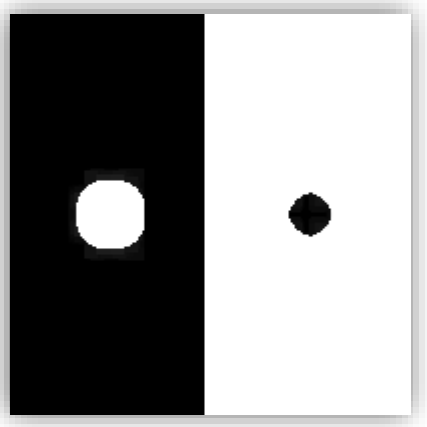
**Dilation** is the opposite process to erosion. White pixels are added to the boundaries in a binary image. You can achieve dilation by using ImageFilter.MaxFilter(3), which converts a pixel to white if any of its neighbors are white.

We can apply dilation to the same image containing a dot and a hole, which you can open and load again:

```
with Image.open(filename) as img:
    img.load()

for _ in range(3):
    img = img.filter(ImageFilter.MaxFilter(3))
img.show()
```

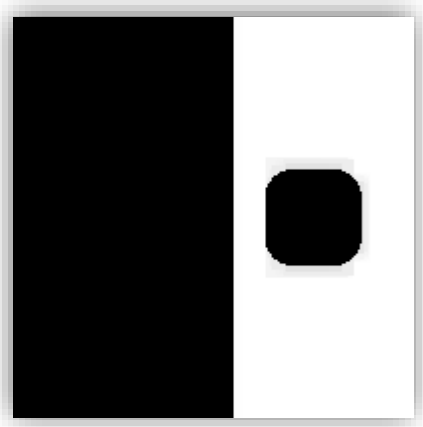
**The dot has now grown bigger, and the hole has shrunk:**



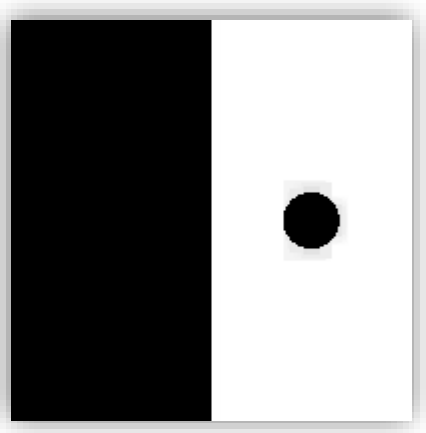
We can use erosion and dilation together to fill in holes and remove small objects from a binary image. Using the image with a dot and hole, you can perform ten erosion cycles to remove the dot, followed by ten dilation cycles to restore the hole to its original size:

```
with Image.open(filename) as img:
    img.load()
for _ in range(10):
    img = img.filter(ImageFilter.MinFilter(3))
img.show()
for _ in range(10):
    img = img.filter(ImageFilter.MaxFilter(3))
img.show()
```

We performed ten erosion cycles with the first for loop. The image at this stage is the following:



The dot has disappeared, and the hole is larger than it was in the original image. The second for loop performs ten dilation cycles, which return the hole to its original size:



However, the dot is no longer present in the image. The erosions and dilations have modified the image to keep the hole but remove the dot. The number of erosions and dilations needed depends on the image and what you want to achieve. Often, you'll need to find the right combination through trial and error.

We can define functions to perform several cycles of erosion and dilation:

```
def erode(cycles, image):
    for _ in range(cycles):
        image = image.filter(ImageFilter.MinFilter(3))
    return image

def dilate(cycles, image):
    for _ in range(cycles):
        image = image.filter(ImageFilter.MaxFilter(3))
    return image
```

These functions make it easier to experiment with erosion and dilation for an image. We'll use these functions in the next section as you continue working on placing the cat into the monastery.

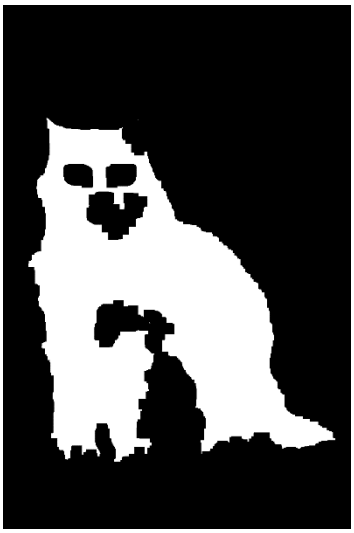
**Image Segmentation Using Thresholding**

We can use a sequence of erosions and dilations on the threshold image that you obtained earlier to remove parts of the mask that don't represent the cat and to fill in any gaps in the region containing the cat. Once we've experimented with erosion and dilation, we'll be able to use educated guesses in a trial-and-error process to find the best combination of erosions and dilations to achieve the ideal mask.

Starting with the image img_cat_threshold, which we obtained earlier, we can start with a series of erosions to remove the white pixels that represent the background in the original image. We should continue working in the same REPL session as in the previous sections.

```
step_1 = erode(12, img_cat_threshold)
step_1.show()
```

The eroded threshold image no longer contains white pixels representing the background of the image:



However, the remaining mask is smaller than the overall outline of the cat and has holes and gaps within it. We can perform dilations to fill the gaps:

```
step_2 = dilate(58, step_1)
step_2.show()
```

The fifty-eight cycles of dilation filled all the holes in the mask to give the following image:

However, this mask is too big. We can therefore finish the process with a series of erosions:

```
cat_mask = erode(45, step_2)
cat_mask.show()
```

The result is a mask that we can use to segment the image of the cat:

We can avoid the sharp edges of a binary mask by blurring this mask. We'll have to convert it from a binary image into a grayscale image first:

```
cat_mask = cat_mask.convert("L")
cat_mask = cat_mask.filter(ImageFilter.BoxBlur(20))
cat_mask.show()
```

The BoxBlur() filter returns the following mask:

The mask now looks like a cat! Now we're ready to extract the image of the cat from its background:

```
blank = img_cat.point(lambda _: 0)
cat_segmented = Image.composite(img_cat, blank, cat_mask)
cat_segmented.show()
```

First, we create a blank image with the same size as img_cat. We create a new Image object from img_cat by using **.point()** and setting all values to zero. Next, you use the composite() function in PIL.Image to create an image made up from both img_cat and blank using cat_mask to determine which parts of each image are used. The composite image is shown below:

We've segmented the image of the cat and extracted the cat from its background.

**Superimposition of Images Using Image.paste()**

We can go a step further and paste the segmented image of the cat into the image of the monastery.

```
filename_monastery = "monastery.jpg"
with Image.open(filename_monastery) as img_monastery:
    img_monastery.load()

img_monastery.paste(
    img_cat.resize((img_cat.width // 5, img_cat.height // 5)),
    (1300, 750),
    cat_mask.resize((cat_mask.width // 5, cat_mask.height // 5)),
)

img_monastery.show()
```

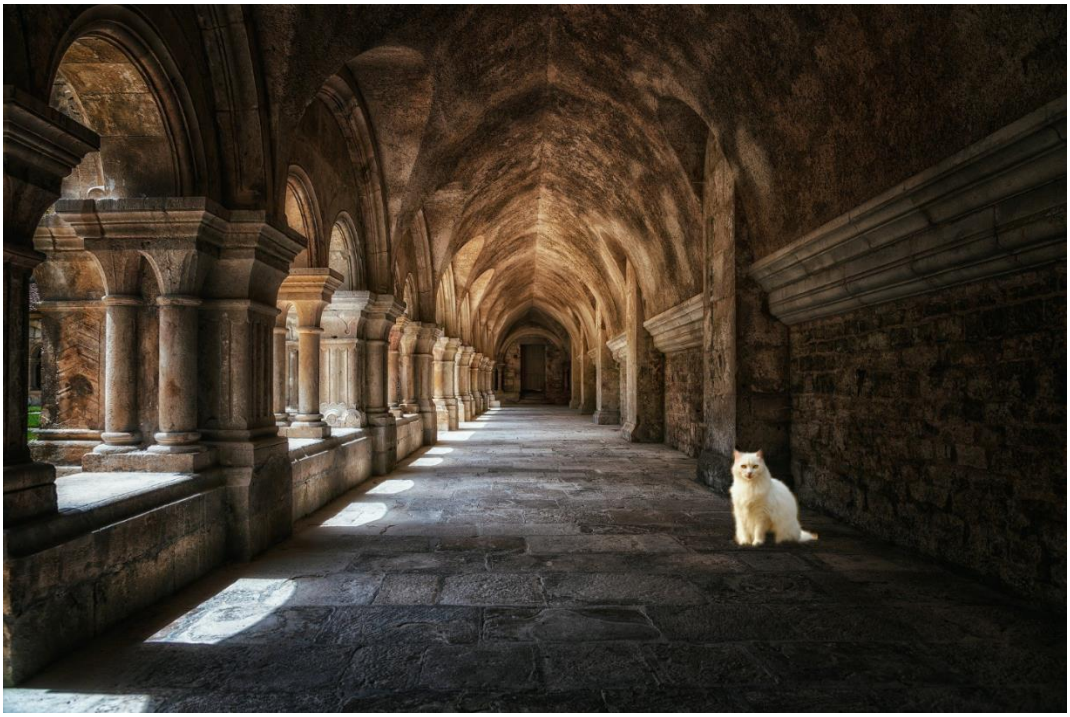
We've used .paste() to paste an image onto another one. This method can be used with three arguments:

The first argument is the **image** that we want to paste in. You're resizing the image to one-fifth of its size using the integer division operator (//).

The second argument is the **location** in the main image where we want to paste the second picture. The tuple includes the coordinates within the main image where we want to place the top-left corner of the image that we're pasting in.

The third argument provides the **mask** that we wish to use if we don't want to paste the entire image.

We've used the mask that you obtained from the process of thresholding, erosion, and dilation to paste the cat without its background. The output is the following image:



We've segmented the cat from one image and placed it into another image to show the cat sitting quietly in the monastery courtyard rather than in the field where it was sitting in the original image.

## Code References

```
from PIL import Image
filename_cat = "cat.jpg"
with Image.open(filename_cat) as img_cat:
    img_cat.load()

img_cat = img_cat.crop((800, 0, 1650, 1281))
img_cat.show()
img_cat_gray = img_cat.convert("L")
img_cat_gray.show()

threshold = 100
img_cat_threshold = img_cat_gray.point(
    lambda x: 255 if x > threshold else 0)
img_cat_threshold.show()

red, green, blue = img_cat.split()
red.show()
green.show()
blue.show()

threshold = 57
img_cat_threshold = blue.point(lambda x: 255 if x > threshold else 0)
img_cat_threshold = img_cat_threshold.convert("1")
img_cat_threshold.show()

from PIL import ImageFilter
filename = "dot_and_hole.jpg"

with Image.open(filename) as img:
    img.load()
for _ in range(3):
    img = img.filter(ImageFilter.MinFilter(3))
img.show()

with Image.open(filename) as img:
    img.load()

for _ in range(3):
    img = img.filter(ImageFilter.MaxFilter(3))

img.show()

with Image.open(filename) as img:
    img.load()
```

```
for _ in range(10):
    img = img.filter(ImageFilter.MinFilter(3))
img.show()

for _ in range(10):
    img = img.filter(ImageFilter.MaxFilter(3))
img.show()

def erode(cycles, image):
    for _ in range(cycles):
         image = image.filter(ImageFilter.MinFilter(3))
    return image

def dilate(cycles, image):
    for _ in range(cycles):
         image = image.filter(ImageFilter.MaxFilter(3))
    return image

step_1 = erode(12, img_cat_threshold)
step_1.show()

step_2 = dilate(58, step_1)
step_2.show()

cat_mask = erode(45, step_2)
cat_mask.show()

cat_mask = cat_mask.convert("L")
cat_mask = cat_mask.filter(ImageFilter.BoxBlur(20))
cat_mask.show()

blank = img_cat.point(lambda _: 0)
cat_segmented = Image.composite(img_cat, blank, cat_mask)
cat_segmented.show()

filename_monastery = "monastery.jpg"
with Image.open(filename_monastery) as img_monastery:
    img_monastery.load()
img_monastery.paste(
    img_cat.resize((img_cat.width // 5, img_cat.height // 5)),
    (1300, 750),
    cat_mask.resize((cat_mask.width // 5, cat_mask.height // 5)),
)
img_monastery.show()
```