Python Exercise

Digital Systems M, Module 2
Matteo Poggi, Università di Bologna

Python Exercise

In this lecture, we will play with Python libraries studied so far.

The main goal is to understand how carefully thought design choices impact on **efficiency.** You can either try on your own workstation (desktop or laptop), or on a **real embedded device** (NVIDIA Jetson Nano)

Python Exercise

If you want to check how efficient your code is on the NVIDIA Jetson Nano:

1) import the time package in your script and wrap your code as follows:

```
import time
start = time.time()
# your code ...
end = time.time()
print("Time elapsed: %f"%(start-end))
```

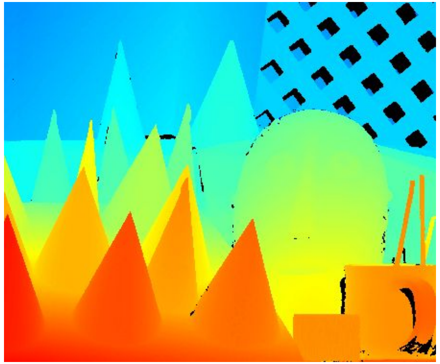
2) collect your code in a folder named “*YourSurname*” and send it over scp to:

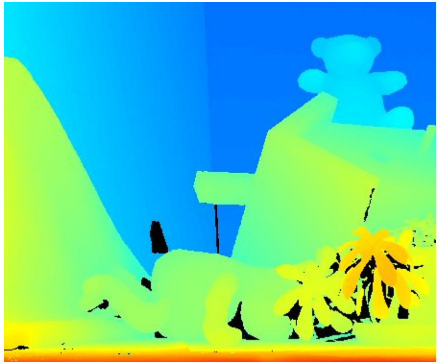
scp -r YourSurname exercise@sisdigitali.ddns.net:/home/exercise/
(password: sisdigitali2122)

(the board will be online only during this lecture)

Python Exercise

Case study: Stereo depth estimation





Python Exercise

Some basics:

1) Stereo depth estimation is based solving **pixel matching** across two images.

2) Given a left and a right image, any pixel in the former will appear **shifted on its left** in the latter. This shift is called **disparity.**

3) Depth and disparity are linked by inverse proportionality

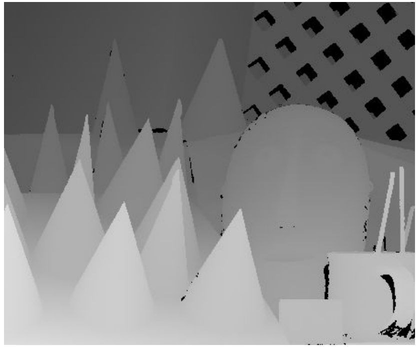
$$z \sim 1/d$$

Python Exercise

How to match pixels:

1) Two pixels on the two images match when they have the same **color** (to make it easier, we work on **grayscale** images)

2) For each pixel A on the left image, we look for D pixels on the right image and look for the **most similar one** B (basically, the one with the lowest color difference). The disparity of A is the difference between x coordinates of A and B.



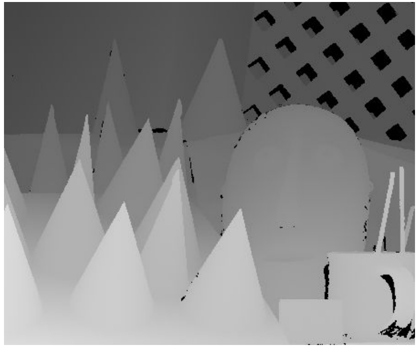
3) After this is carried out for any pixel, we can draw a **disparity map**

Python Exercise

How to match pixels:

1) Two pixels on the two images match when they have the same **color**

2) For each pixel A on the left image, we look for D pixels on the right image and look for the **most similar one** B (basically, the one with the lowest color difference). The disparity of A is the difference between x coordinates of A and B.

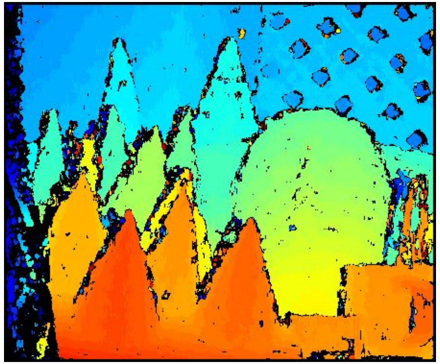


3) After this is carried out for any pixel, we can draw a **disparity map** (and apply a colormap, eventually)

Python Exercise

A basic stereo algorithm is provided by OpenCV and allows to obtain a quite nice disparity map with a few lines of code:

```
left = cv2.imread('cones0.png', cv2.IMREAD_GRAYSCALE)
right = cv2.imread('cones1.png', cv2.IMREAD_GRAYSCALE)
dmax = 64
block_size = 7
bm = cv2.StereoBM_create(dmax, block_size)
```



Let’s try to implement this with our custom code!

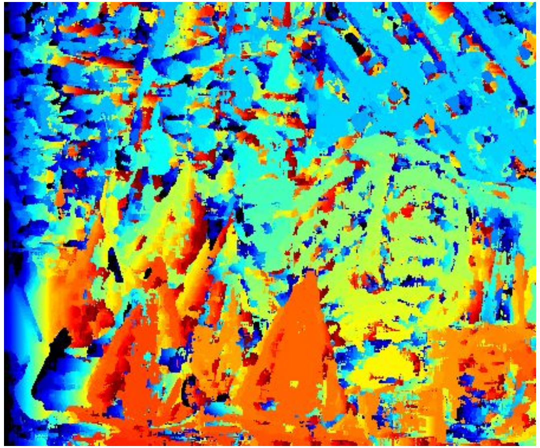
Problem:

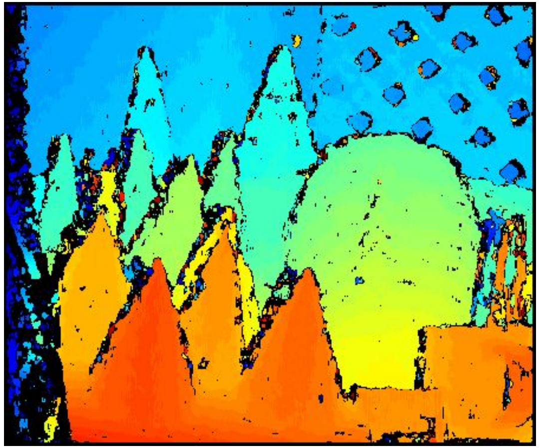
Many pixels on the right image might have the same color, leading to **ambiguity**

Solution (partial):

compare **image patches** rather than **single pixels** (block_size parameter in StereoBM is indeed the size of an image patch!)

If we compare 7x7 image patches, our final disparity map looks better. However, it is still far from the one obtained with OpenCV algorithms…





Maybe looking at pixels color is not enough?

Python Exercise

Census trasform:

Convert pixels color into a more meaningful information, better encoding the **local structure** of the image.

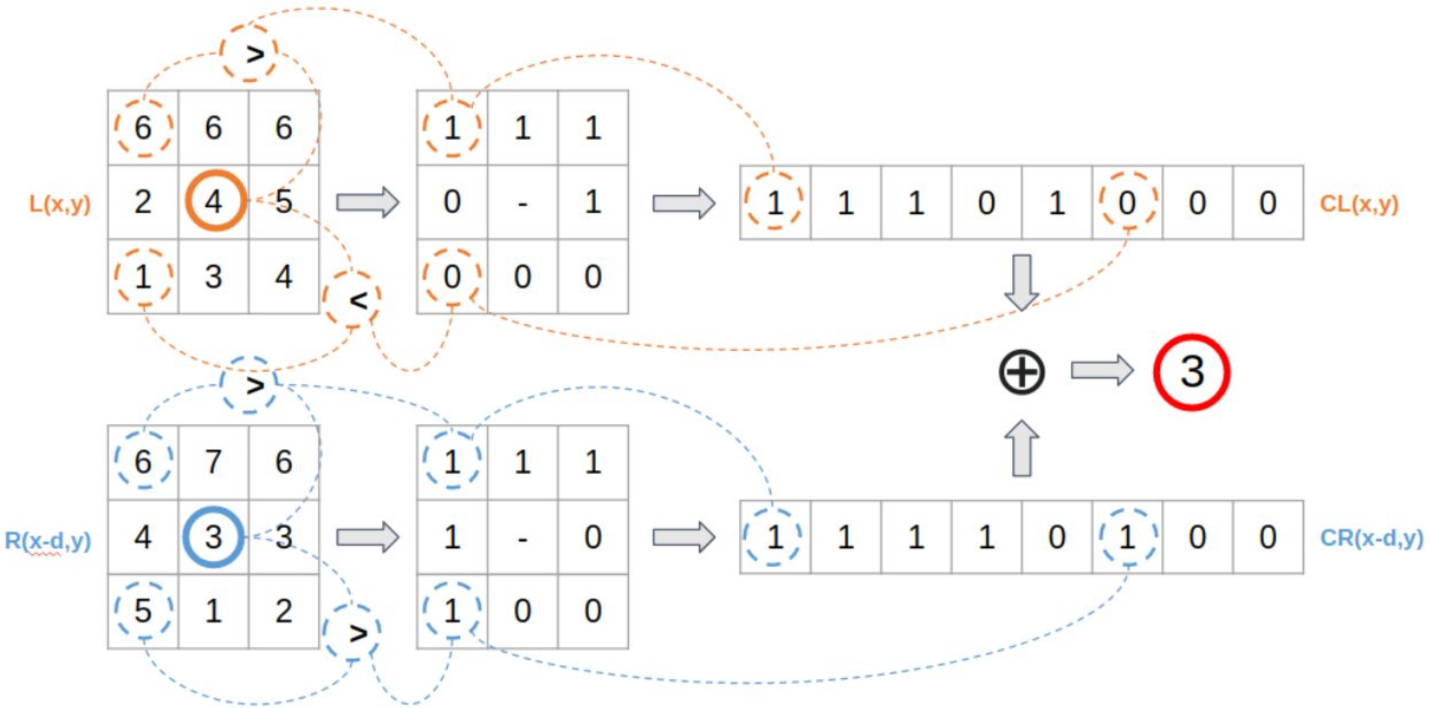
For a given pixel, we replace its color value with a **bit array**, obtained by comparing its intensity with the one of its neighbors (for instance, we assign 1 if the neighbor intensity is higher, 0 otherwise).

A string of **8 bits** (windows size - 1) is obtained.

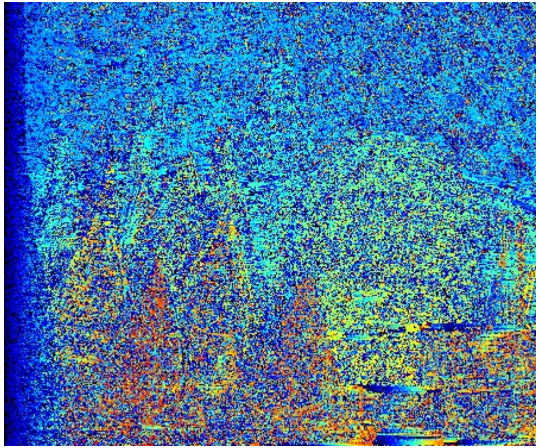
6	6	6
2	4	5
1	3	4

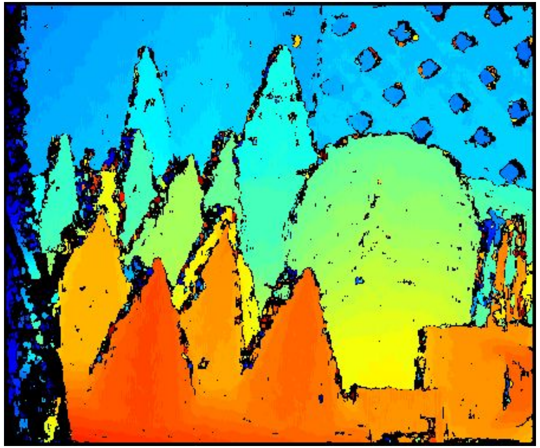
1	1	1	0	1	0	0	0

Given two patches, we can compare them by computing the census transform for the central pixel and then the **Hamming Distance** between the two bit arrays



Again, the result is quite disappointing…
Are we missing something?





What about comparing pixels bit arrays on a **local patch**?

Here we are!
This implementation looks very similar to OpenCV StereoBM.