Report of the first practical Work

Computer Vision

DAMMOU Meryem

meryem.dammou@univ-rouen.fr

Table of content

List of figures

The goal of this lab is to implement and evaluate the algorithm of depth estimation using a couple of stereo images. In the following parts, we will see how we managed to upload and work with each function in order to obtain the results.

1. Depth map:

We need to implement an algorithm that will estimate the depth map, which is an image or an image channel that contains information relating to the distance of the surfaces of scene objects from a viewpoint. For this, we will use the algorithm SSD. But first, what is SSD?

For two images $p_1(u_1,v_1)$ and $p_2(u_2,v_2)$, the SSD (Sum of squared Distances) is defined as follow:

$$SSD(\mathrm{p1(u1,v1),p2(u2,v2)}) = \sum_{i=-N}^{N} \sum_{j=-P}^{P} |I_1(u_1+i, v_1+j) - I_2(u_2+i, v_2+j)|$$

I_1 and I_2 are the intensities of the images.

The packages we will work with through this whole work are:

- Matplotlib.pyplot to visualize the images
- Skimage to upload the images
- Numpy to work with matrixes

The first step we need to do is indeed to upload the two images from the file. For that, we use *io* available on the package Skimage.

```
img1 = io.imread('TP 1 Depth 18_19/synthetic/syntheticd.pgm')
img2 = io.imread('TP 1 Depth 18_19/synthetic/syntheticg.pgm')

N,M = img1.shape


plt.figure()
plt.imshow(img1, cmap=plt.cm.gray)
plt.axis('off')
plt.title('Image1')
plt.figure()

plt.figure()
plt.imshow(img2, cmap=plt.cm.gray)
plt.axis('off')
plt.title('Image2')
```

Figure 1 Uploading the images

Note that we use, inside *imshow,* the function *cmap=plt.cm.gray* in order to have an image in grayscale

The output looks like this:

Figure 2 The images

1.1. Sliding window size 1:

We need to compute the minimum distance value of the two images, based on the sliding window (pixel per pixel).

```
img_res = np.zeros((N,M))

for i in range (N):
    for j in range (M):
        d = []
        ssd = 0

        for k in range (N):
            ssd+=np.abs((int(img1[i,j])-int(img2[i,k]))**2)
            d.append(ssd)


        img_res[i,j] = np.min(d)



plt.figure()
plt.imshow(img_res, cmap=plt.cm.gray)
plt.axis('off')
plt.savefig('Window = 1')
plt.title('Image resultat ')
```

We initialize the matrix the will contain our minimum distances

The distance is a list, ssd is a number. The two values will be changed in each iteration

For each line, we compute the value of ssd and we add it into the d list

We return the min value of d that will be put into the image

Plotting the final image and saving it

Figure 3 Sliding window code

The output image looks like this:

Figure 4 The output image for a sliding window sized 1

1.2. Sliding window sized 3:

As for the first method, we need to compute at each iteration the value of ssd and minimum distance. However, the sliding window's size needs to change. Here, it is size 3*3.

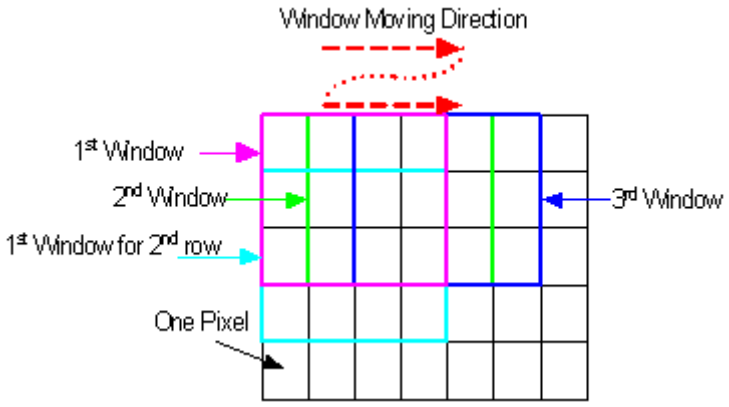


Figure 5 The philosophy of the sliding window

The code for 3*3 sliding window is below. The idea behind it is the same:

- We initialize the sliding window, here the 1-size square
- For each value of the image, we memorize the ssd value
- we return the min value the distance for each iteration.

The output is a match

```
def Init_img3(img):
    N,M = img.shape
    imgr = np.zeros ((N+2,M+2))

    for i in range (N):
        for j in range (M):
            imgr[i+1,j+1] = img[i,j]
    return imgr

imgr1 = Init_img3(img1)
imgr2 = Init_img3(img2)

windows = [(-1,-1),(-1,0),(-1,1),(0,1),(0,-1),(1,1),(1,-1),(1,0),(-1,1)]
img_result = np.zeros((N+1,M+1))
for i in range (1,N+1):
    for j in range (1,M+1):
        d = []
        for k in range (1,M+1):
            ssd = 0
            for w in windows:
                ssd += (imgr1[i+w[0],j+w[1]]-imgr2[i+w[0],k+w[1]])**2
            d.append(ssd)
        img_result[i][j] = int(min(d))

plt.figure()
plt.imshow(img_result, cmap=plt.cm.gray)
plt.axis('off')
plt.title('Image resultat ')
```

Here, the process is the same as the previous one, but, the only difference is that we add more pixels to the image so that the sliding of the window goes through the whole image. We prevent errors.

*Figure 6 Code for the 3*3 sliding window*

The output of the code is the image *img_ result,* plotted in gray levels looks like this:

*Figure 7 Output for 3*3*

We can see that the image is visibly smoother but lost some details.

1.3. Sliding window 5*5:

Here, we need to see the effect of increasing the sliding window's size. The code remains the same, but again, we need to increase the size of the image so that the sliding window will be able to go through the whole image input.

```python
def Init_img5(img):
    N,M = img.shape
    imgr = np.zeros ((N+4,M+4))

    for i in range (N):
        for j in range (M):
            imgr[i+2,j+2] = img[i,j]
    return imgr


imgr1 = Init_img5(img1)
imgr2 = Init_img5(img2)

windows = [(i,j) for i in range(-2,3) for j in range(-2,3)]
img_result = np.zeros((N+4,M+4))

for i in range (2,N+2):
    for j in range (2,M+2):
        d = []
        for k in range (2,M+2):
            ssd = 0
            for w in windows:
                ssd += (imgr1[i+w[0],j+w[1]]-imgr2[i+w[0],k+w[1]])**2
            d.append(ssd)
        img_result[i][j] = int(min(d))

plt.figure()
plt.imshow(img_result, cmap=plt.cm.gray)
plt.axis('off')
plt.title('Image resultat ')
```

Adapting the size of the image to the size of the window

In each iteration, we compute ssd, and we append its minimum value to the final image.

*Figure 8 Code for 5*5 sliding window*

The output is the image below:

*Figure 9 Output for 5*5*

We can clearly see that, compared to the previous outputs, the image has certainly less details but looks smoother.

1.4. Sliding window 7*7:

We define a new function that does the same functions as the previous ones. Once again,

```
def Init_img7(img):
    N,M = img.shape
    imgr = np.zeros ((N+6,M+6))

    for i in range (N):
        for j in range (M):
            imgr[i+3,j+3] = img[i,j]
    return imgr


imgr1 = Init_img7(img1)
imgr2 = Init_img7(img2)

windows = [(i,j) for i in range(-3,4) for j in range(-3,4)]
img_result = np.zeros((N+6,M+6))

for i in range (3,N+3):
    for j in range (3,M+3):
        d = []
        for k in range (3,M+3):
            ssd = 0
            for w in windows:
                ssd += (imgr1[i+w[0],j+w[1]]-imgr2[i+w[0],k+w[1]])**2
            d.append(ssd)
        img_result[i][j] = int(min(d))

plt.figure()
plt.imshow(img_result, cmap=plt.cm.gray)
plt.axis('off')
plt.title('Image resultat ')
```

*Figure 10 Code for 7*7 window*

The output is the image below:

*Figure 11 output image for 7*7*

1.5. Generic Function SSD:

Now that we've seen all of these steps, we can combine them into one function, that takes as an input the image and the size of the sliding window and gives to the user the final disparity map.

```
def SSD_Function (img1, img2, Twindows):
    L,C = img1.shape
    L2,C2 = img2.shape
    imgP1 = np.zeros ((L+Twindows-1,C+Twindows-1))
    imgP2 = np.zeros ((L2+Twindows-1,C2+Twindows-1))
    imgR = np.zeros ((L2+Twindows-1,C2+Twindows-1))

    #Prolongation matrice
    b = int((Twindows-1)/2)
    imgP1 [b:L+b, b:C+b] = img1
    imgP2 [b:L2+b, b:C2+b] = img2

    if(Twindows % 2 == 0):
        print("choose another windows ...!")
    else :
        for i in range (b,L):
            for j in range(b,C):
                v = []
                for k in range(b,C2):
                    v.append(np.sum(np.power((imgP1[i-b:i+b+1, j-b:j+b+1] - imgP2[i-b:i+b+1, k-b:k+b+1]),2)))
                imgR[i,j]=min(v)
    return imgR
plt.figure()
plt.imshow(SSD_Function(img1,img2,3), cmap=plt.cm.gray)
plt.axis('off')
plt.title('Image resultat ')
```

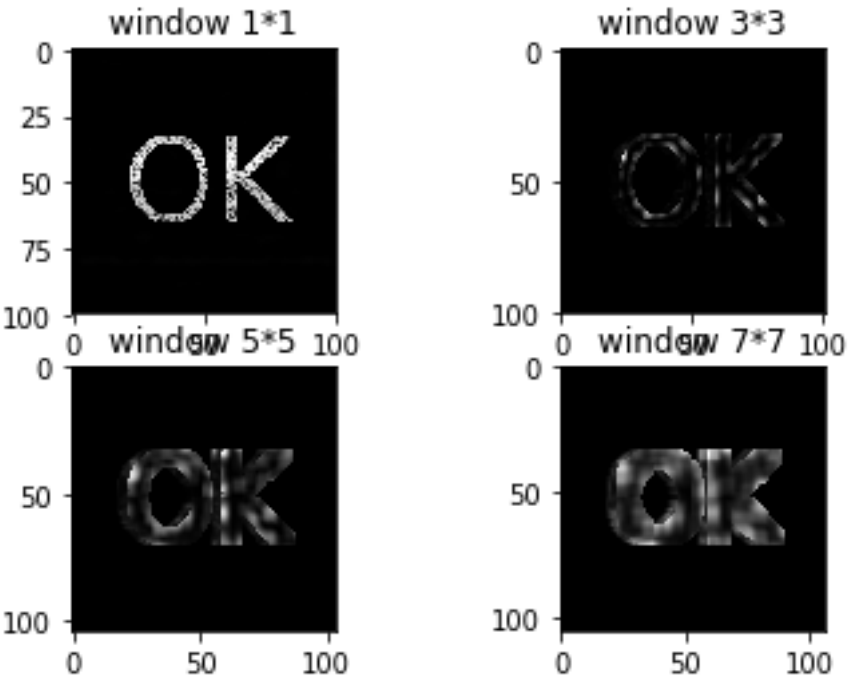
Figure 12 Code of the SSD Function

In this function, we combined all of results listed above. All we have to do is to insert the arguments of the function (the images and the size of the window) in order to visualize the result.

Here is an example of what we obtain, given the same two images and a sliding window 3*3

*Figure 13 SSD function on 3*3*

1.6. Summary:



We plotted all of figures with different windows to see the major differences that occur. We can clearly see that smaller window size (1*1, 3*3) seem to have more details and the OK is much visible. However, there is so much noise in it, and the disparity map is not that clear. The more we increase the size of the window (5*5, 7*7), the more we can see that the details of the figure are lost. But, the disparity map is more clear and smooth. To assert the results, we plot the disparity map for 11*11, listed below, where we can clearly see that.

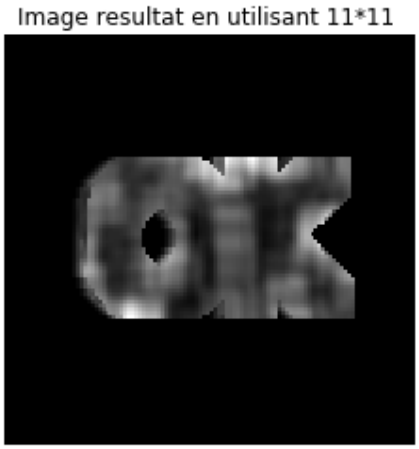


*Figure 14 Disparity Map 11*11*

2. Testing on other images:

We test our algorithm on other photos, cone and teddy. Unfortunately, the computing time for more that 1*1 window is huge, due to the size of the photos. We could resize them, but

the output would loose so much of its pixels. That's why, we decided to keep the image as follow and only see what it would give with a simple algorithm.

Figure 15 Teddy images

Figure 16 Cones images

Here, we can clearly see that the image is so noisy, and has many details to it. That's what makes the depth map nearly undetectable.

In order to visualize the output for bigger windows, we decide to resize the input images. We chose we give the size 100*100. Please notice that the image will loose so much of its characteristics, but we do this in order to give a result.

*Figure 17 Disparity Map 7*7*

*Figure 18 Disparity Map 3*3*

We can see that the depth map contains so much noise. Comparing it to the data provided, the one we have has clearly lost so much information.