Computer Vision
CS 6476 , Spring 2018

PS 5

Professor :
Cedric Pradalier

Author :
Melisande Zonta

April 6, 2018

Contents

1 Gaussian and Laplacian Pyramids

1.1 Reduce

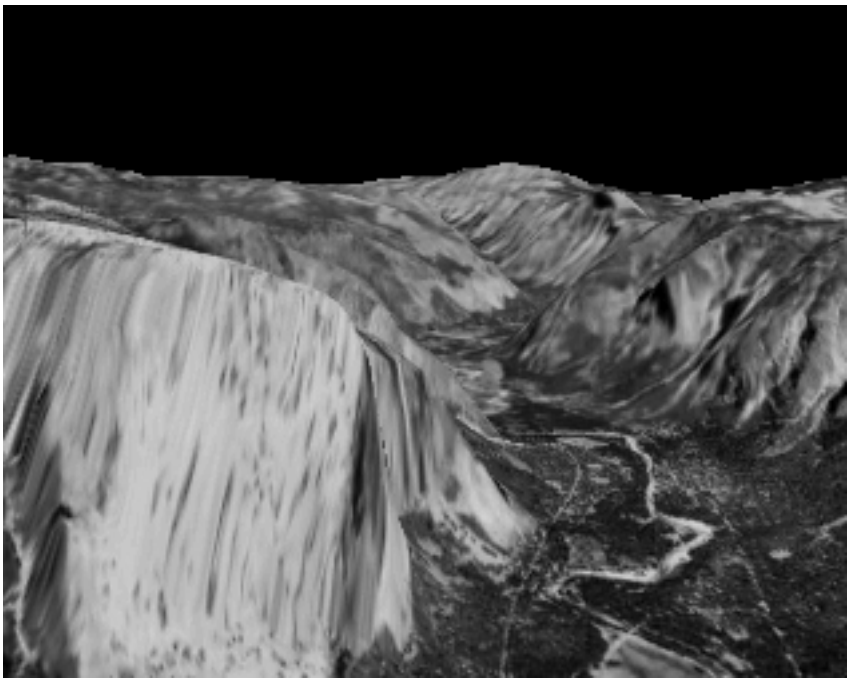
We start by implementing the reduce operator using the function below, which allows us to compute the Gaussian pyramid.

A first step of preprocessing was executed by creating a function to apply zero padding if the size of the image is not of a power of 2.

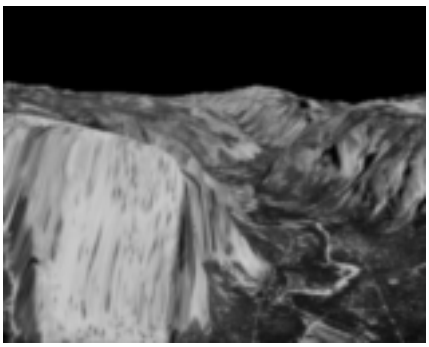
Then another function allows us to cut the resulting image in order to remove the zero padding. Finally the borders were taken in charge by using the built-in OpenCV function *copyMakeBorder* with the replication option.

```
def reduce(level0, a):

    # Kernel building
    w = np.array([[1/4-a/2, 1/4, a, 1/4, 1/4-a/2]])
    kernel = w * w.T

    m, n = level0.shape
    level1 = np.zeros([int(m/2), int(n/2)], dtype="float32")

    # Mirrors an image borders
    G = cv2.copyMakeBorder(level0, 2, 2, 2, 2, cv2.BORDER_REPLICATE)

    for r,i in zip(list(range(2, m+2, 2)), list(range(int(m/2)))):
        for c,j in zip(list(range(2, n+2, 2)), list(range(int(n/2)))):
            min_r = int(r - 2); max_r = int(r + 3)
            min_c = int(c - 2); max_c = int(c + 3)
            level1[i,j] = np.sum(np.multiply(kernel, G[min_r:max_r, min_c:max_c
    ]))

    return level1
```

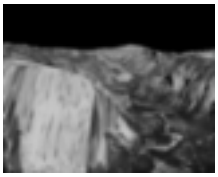
```
def remove_zero_padding(image, level, rows_diff, cols_diff):
    m, n = image.shape
    R = int(floor(rows_diff / pow(2, level)))
    C = int(floor(cols_diff / pow(2, level)))
    r_index = slice(0, m - R) if R > 0 else slice(0, m)
    c_index = slice(0, n - C) if C > 0 else slice(0, n)
    return image[r_index, c_index]

def Gaussian_Pyramid(image, n, cut=False):

    pyramid = []
    zero_padded_image, diff_row, diff_cols = zero_padding(image)
    pyramid.append(zero_padded_image)
    for i in range(n):
        level = reduce(pyramid[i], 0.4) # for a = 0.4 it is Gaussian-like
        pyramid.append(level)
        if cut:
```



a



b



c

d

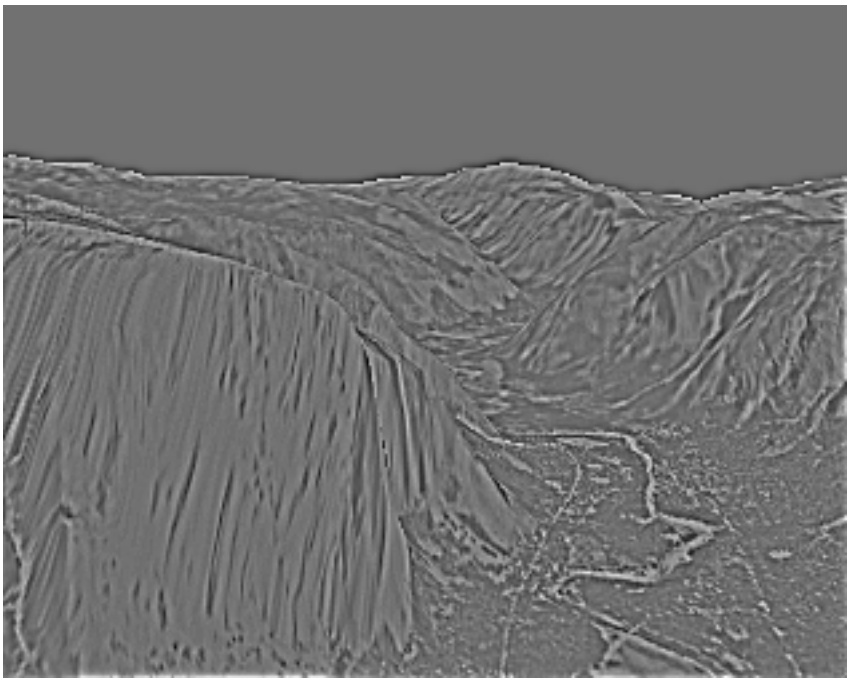
Figure 1: *a.* ps5-1-1-0.png. *b.* ps5-1-1-0.png. *c.* ps5-1-1-2.png. *d.* ps5-1-1-3.png.

The images presented in Figure 1 a, b, c and d show the levels 0 to 3 of the Gaussian pyramid for the first frame of DataSeq1.

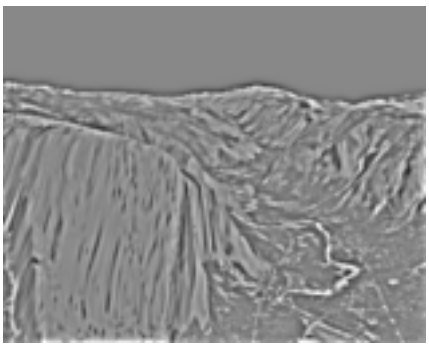
1.2 Expand

Now we write the expand function, that will allow us to build the hierarchical Lucas-Kanade akgorithm. We test it here by computing the Laplacian pyramid from the Gaussian pyramid of DataSeq1.

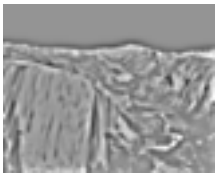
```
def expand(level0, a):
    # Kernel building
    w = np.array([[1/4-a/2, 1/4, a, 1/4, 1/4-a/2]])
    kernel = w * w.T

    m, n = level0.shape
    level1 = np.zeros([int(m*2), int(n*2)], dtype="float32")

    # Mirrors an image borders
    G = cv2.copyMakeBorder(level0, 2, 2, 2, 2, cv2.BORDER_REPLICATE)

    for i in range(int(m*2)):
        for j in range(int(n*2)):
            level1[i,j] = 0
            M = [-2,0,2] if i % 2 == 0 else [-1,1]
            N = [-2,0,2] if j % 2 == 0 else [-1,1]
            for m in M:
                for n in N:
                    r = int((i-m)/2); c = int((j-n)/2);
                    level1[i,j] += kernel[m+2,n+2] * G[r+2,c+2]
            level1[i,j] *= 4

    return level1
```



a



b



c

d

Figure 2: *a.* ps5-1-2-0.png. *b.* ps5-1-2-1.png. *c.* ps5-1-2-2.png. *d.* ps5-1-2-3.png.

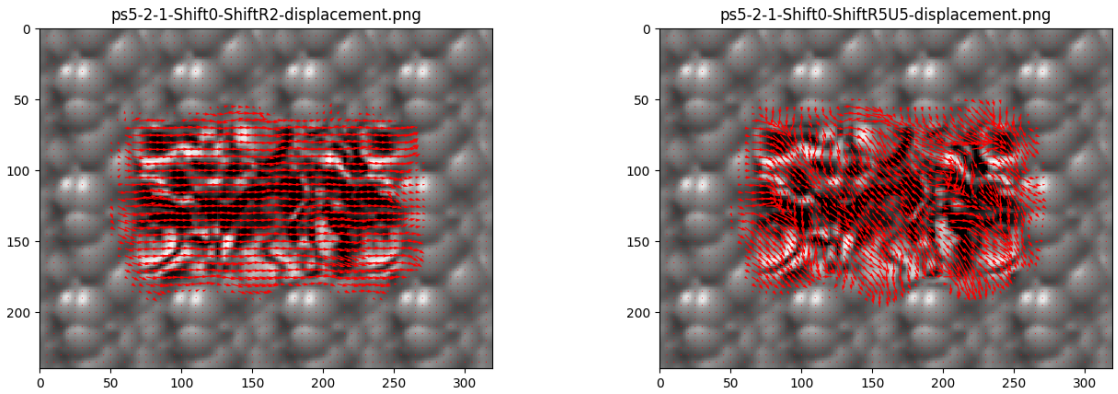
The resulting images (Figure 2 a, b, c and d) show the Laplacian pyramid level 0 to level 3 (the last level of the Gaussian pyramid).

2 Lucas-Kanade Optic Flow

2.1 Lucas-Kanade for Small Displacements

The next step is to implement the Lucas-Kanade computation. We first prefer to smooth the images and using a Gaussian kernel convolution. In order to remove outliers that are breaking the smooth motion assumption, a median filter is applied to the resulting displacement images. The parameters used for the following tests were a window size of 20 pixels and a sigma of 5.

```
def Lucas_Kanade(left, right, size, sigma):

    rows, cols = left.shape

    # Smooth images
    kernel = cv2.getGaussianKernel(size, sigma) * cv2.getGaussianKernel(size,
    sigma).T
    left = cv2.filter2D(left, -1, kernel)
    right = cv2.filter2D(right, -1, kernel)

    # Compute gradients
    g = np.array([[-1, 0, 1]])
    Ix1 = cv2.filter2D(left, cv2.CV_32F, g)
    Iy1 = cv2.filter2D(left, cv2.CV_32F, g.T)

    It = np.asarray(right, dtype="float32") - np.asarray(left, dtype="float32")

    Ixx = np.multiply(Ix1, Ix1)
    Iyy = np.multiply(Iy1, Iy1)
    Ixy = np.multiply(Ix1, Iy1)
    Ixt = np.multiply(Ix1, It)
    Iyt = np.multiply(Iy1, It)

    # Prepare weighting window
    w = np.ones([size, size])

    # Compute weighted sums
    Ixx = cv2.filter2D(Ixx, -1, w)
    Iyy = cv2.filter2D(Iyy, -1, w)
    Ixy = cv2.filter2D(Ixy, -1, w)
    Ixt = cv2.filter2D(Ixt, -1, w)
    Iyt = cv2.filter2D(Iyt, -1, w)

    U = np.zeros([rows, cols], dtype="float32")
    V = np.zeros([rows, cols], dtype="float32")
    for r in range(rows):
        for c in range(cols):
            L = np.array([[Ixx[r,c], Ixy[r,c]],[Ixy[r,c], Iyy[r,c]]])
            R = np.array([[-Ixt[r,c]], [-Iyt[r,c]]])
            if np.linalg.det(L) < 1e-8:
                U[r,c] = 0
                V[r,c] = 0
            else:
                X = np.linalg.inv(L).dot(R)
                U[r,c] = X[0,0]
                V[r,c] = X[1,0]
    return cv2.medianBlur(U, 5), cv2.medianBlur(V, 5)
```



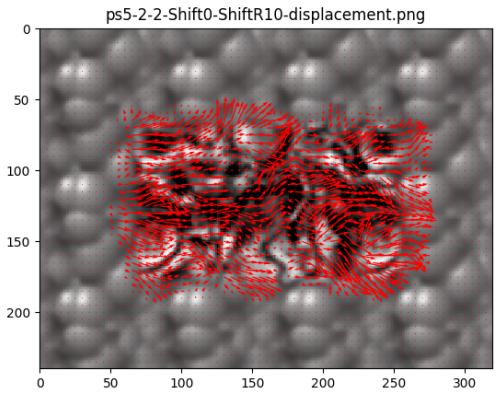
a b

Figure 3: *a*.ps5-2-1-Shift0-ShiftR2-displacement.png. *b*.ps5-2-1-Shift0-ShiftR5U5-displacement.png.

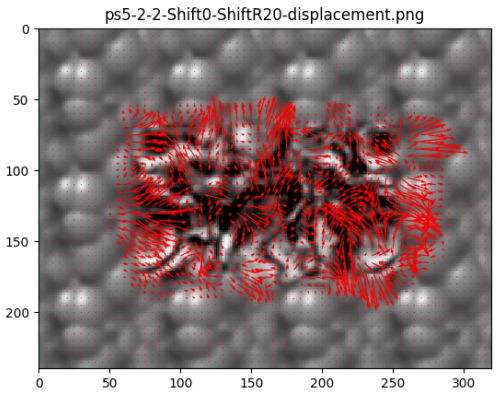
We test the process upon the textured images, the first one shifted by 2 pixels to the right (result in Figure 3 a) and the second one shifted by 5 pixels right and up (Figure 3 b).

2.2 Lucas-Kanade for Larger Displacements

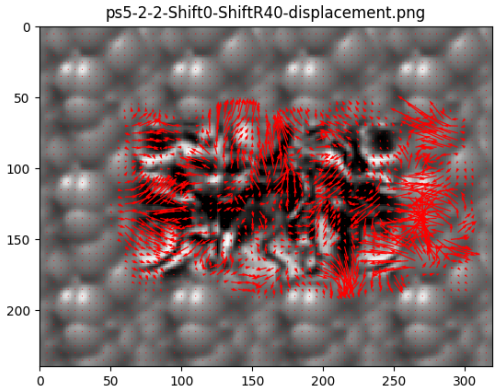
We now try to apply the same algorithm, using the same parameters, to images with larger motions (10, 20 and 40 pixels).



a



b



c

Figure 4: *a.* ps5-2-2-Shift0-ShiftR10-displacement.png. *b.* ps5-2-2-Shift0-ShiftR20-displacement.png. *c.* ps5-2-2-Shift0-ShiftR40-displacement.png.

The respective results are visible in Figure 4 a,b and c. We notice that although some kind of consistency in the displacement amplitude is visible for a shift of 10 pixels, the orientation is definitely wrong. For higher shifts, the resulting displacements are completely wrong. This is explained by the fact that the Lucas-Kanade algorithm, used at a fine level of the pyramid (here the full resolution image), works only for small motions (a few pixels), since it assumes a local consistency of motion.

2.3 Lucas-Kanade applied on more realistic images

We now try our algorithm on more realistic images (DataSeq1 and DataSeq2). We continue to use a single-level version of Lucas-Kanade, but allow ourselves to choose the level in the Gaussian pyramid that seems to work best. We then warp the first image to the second, using the recovered displacement images with the following code :

```
1 def warp(image, U, V):
2     m,n = image.shape
```

```
    X, Y = np.meshgrid(np.arange(n), np.arange(m), indexing="xy")
    Xmap = np.asarray(X - U, dtype="float32")
    Ymap = np.asarray(Y - V, dtype="float32")
    warped_nearest = cv2.remap(image, Xmap, Ymap, cv2.INTER_NEAREST)
    warped_linear = cv2.remap(image, Xmap, Ymap, cv2.INTER_LINEAR)
    indices = np.where(warped_linear == 0)
    warped_linear[indices] = warped_nearest[indices]
    return warped_linear
```

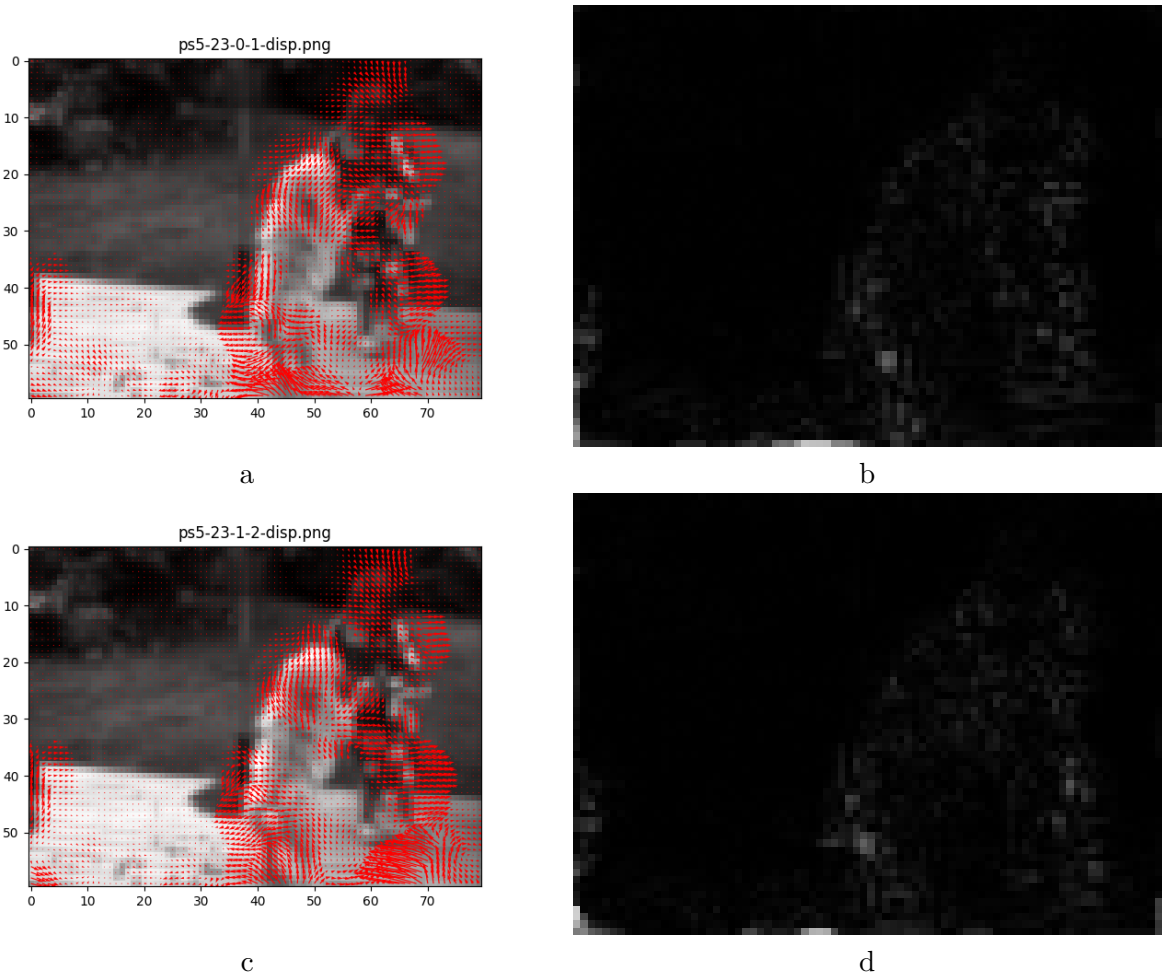
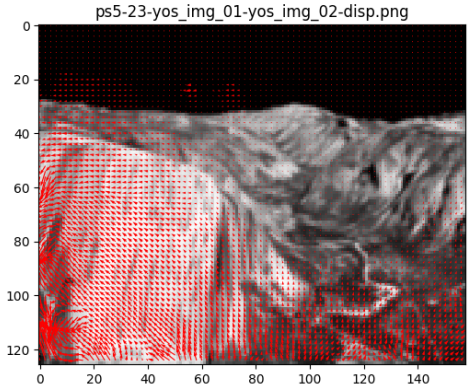
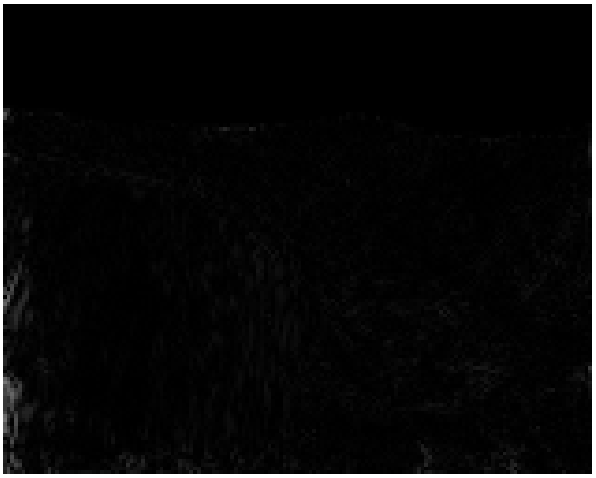


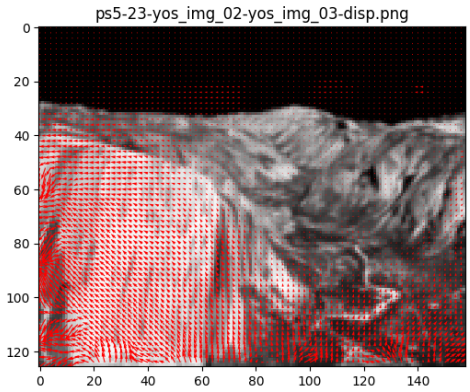
Figure 5: *a.* ps5-2-3-0-1-displacement.png. *b.* ps5-2-3-0-1-diff.png. *c.* ps5-2-3-1-2-displacement.png. *d.* ps5-2-3-1-2-diff.png.



a



b



c

d

Figure 6: *a.* ps5-2-3-yos-img-01-yos-img-02-displacement.png. *b.* ps5-2-3-yos-img-01-yos-img-02-diff.png. *c.* ps5-2-3-yos-img-02-yos-img-03-displacement.png. *d* ps5-2-3-yos-img-02-yos-img-03-diff.png.

Concerning DataSeq2, the level selected was the third level in the pyramid.
The figures 5a and b show the difference and displacement images when warping the first frame to the second one.
The figures 5c and d show the difference and displacement images when warping the second frame to the third one.

Then we study DataSeq1 and we observe that the best results are obtained for the second level of the pyramid.
The figures 6a and b show the difference and displacement images when warping the first frame to the second one.
The figures 6c and d show the difference and displacement images when warping the second frame to the third one.

We can state several problems and observations on the previous images :

- There is a problem on the edges of the images, which could be explained by a wrong handling of the borders of the images.
- The computations seems to work best for DataSeq1 than for DataSeq2 which could be explain by smaller motions spread on the entire image whereas the motions are stronger and more localized in some regions of the image. That's why a coarser level was used for DataSeq2 with a smaller window.

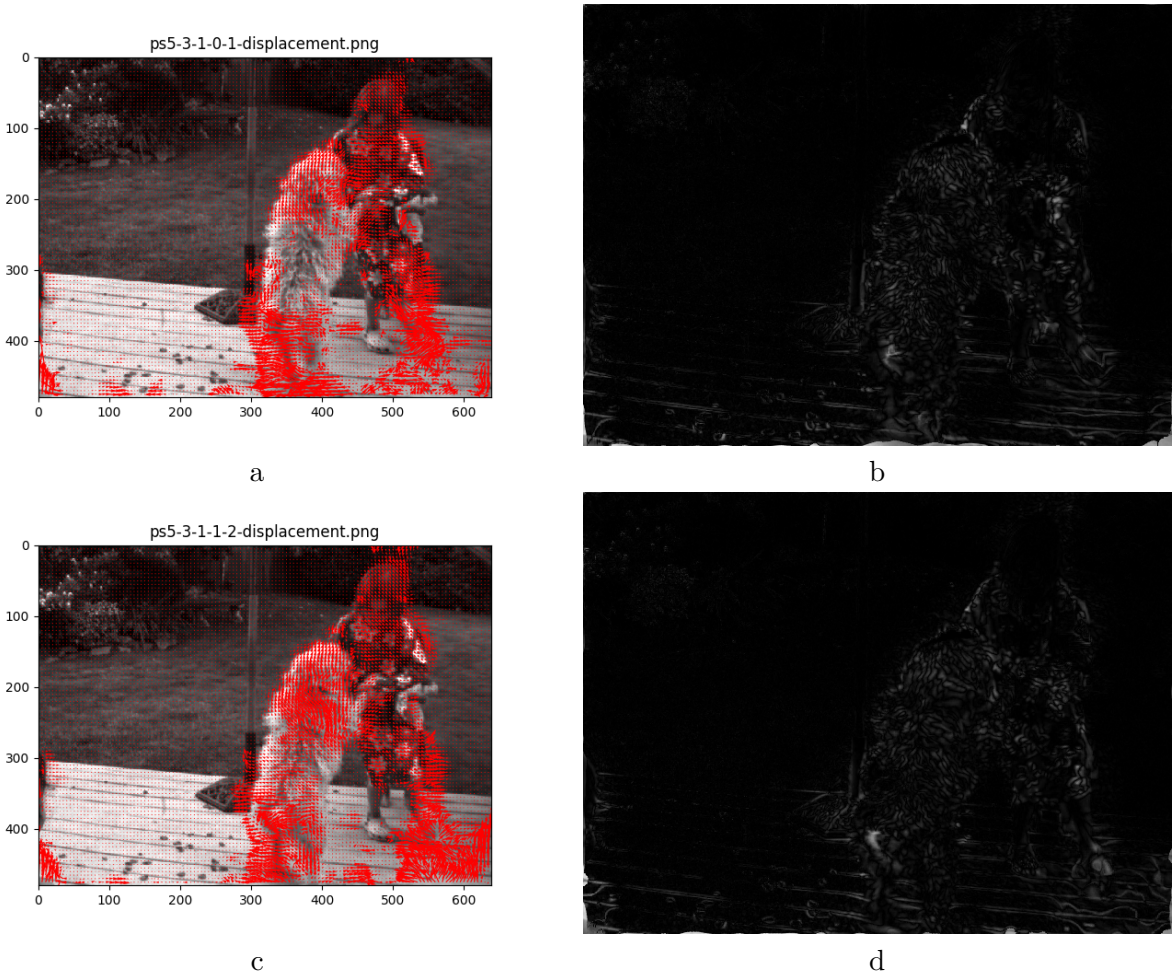
3 Hierarchical Lucas-Kanade

Now we are going to implement a multi-level of the Lucas-Kanade algorithm that is using Gaussian Pyramids, the warping and expand functions so we can initialize the next level with the previous level displacement image.

```
def Hierarchical_LK(left, right, n, a, size, sigma):

    # Initialize k = n where n is the max level
    pyr_left  = Gaussian_Pyramid(left, n, cut=True)
    pyr_right = Gaussian_Pyramid(right, n, cut=True)

    for k in range(n, -1, -1):

        # reduce both input images to level k
        Lk = pyr_left[k]
        Rk = pyr_right[k]

        # If k = n initialize U and V to be zero inages the size of Lk;
        if k == n:
            U = np.zeros(Lk.shape, dtype="float32")
            V = np.zeros(Lk.shape, dtype="float32")
        # otherwise expand the flow field and double to get the next level
        else:
            U = 2 * expand(U, a)
            V = 2 * expand(V, a)

        # Reduce U and V with zeros if size different than images
        if U.shape[0] != Lk.shape[0] or U.shape[1] != Lk.shape[1]:

            R = U.shape[0] != Lk.shape[0]
            C = U.shape[1] != Lk.shape[1]
            U = U[:-R, :-C]
            V = V[:-R, :-C]

        # Warp Lk using U and V to form Wk
        Wk = warp(Lk, U, V)

        # Perform LK on Wk and Rk to yield two incremental flow fields Dx and
    Dy
        Dx, Dy = Lucas_Kanade(np.float32(Wk), np.float32(Rk), size, sigma)

        # Add these to the original flow
        U = U + Dx
```

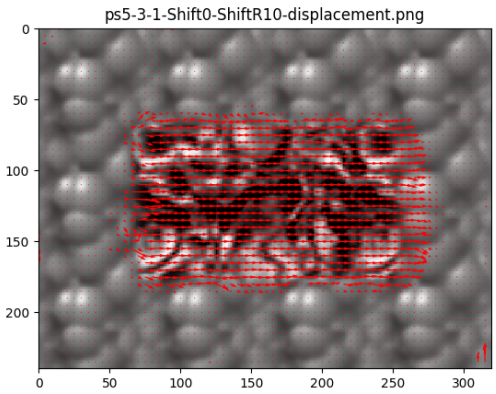
```
        V = V + Dy

    return U, V, Wk, Lk, Rk
```

Then we apply successively this algorithm on the TestSeq, DataSeq1 and DataSeq2.



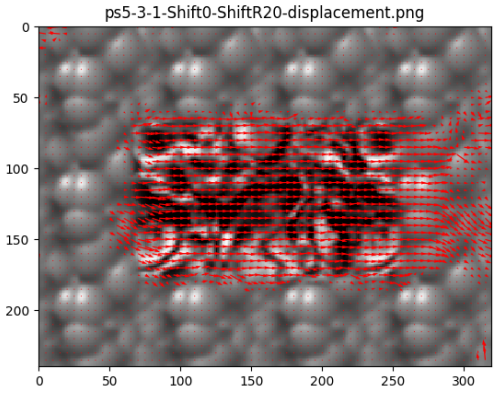
a b c d

Figure 7: *a.* ps5-3-1-0-1-displacement.png. *b.* ps5-3-1-0-1-diff.png. *c.* ps5-3-1-1-2-displacement.png. *d.* ps5-3-1-1-2-diff.png.



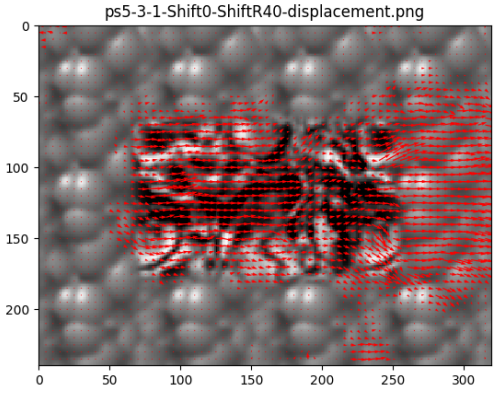
e

f



g

h



g

h

Figure 8: *a.* ps5-3-1-Shift0-ShiftR10-displacement.png. *b.* ps5-3-1-Shift0-ShiftR10-diff.png. *c.* ps5-3-1-Shift0-ShiftR20-displacement.png. *d* ps5-3-1-Shift0-ShiftR20-diff.png. *e* ps5-3-1-Shift0-ShiftR40-displacement.png *f* ps5-3-1-Shift0-ShiftR40-diff.png

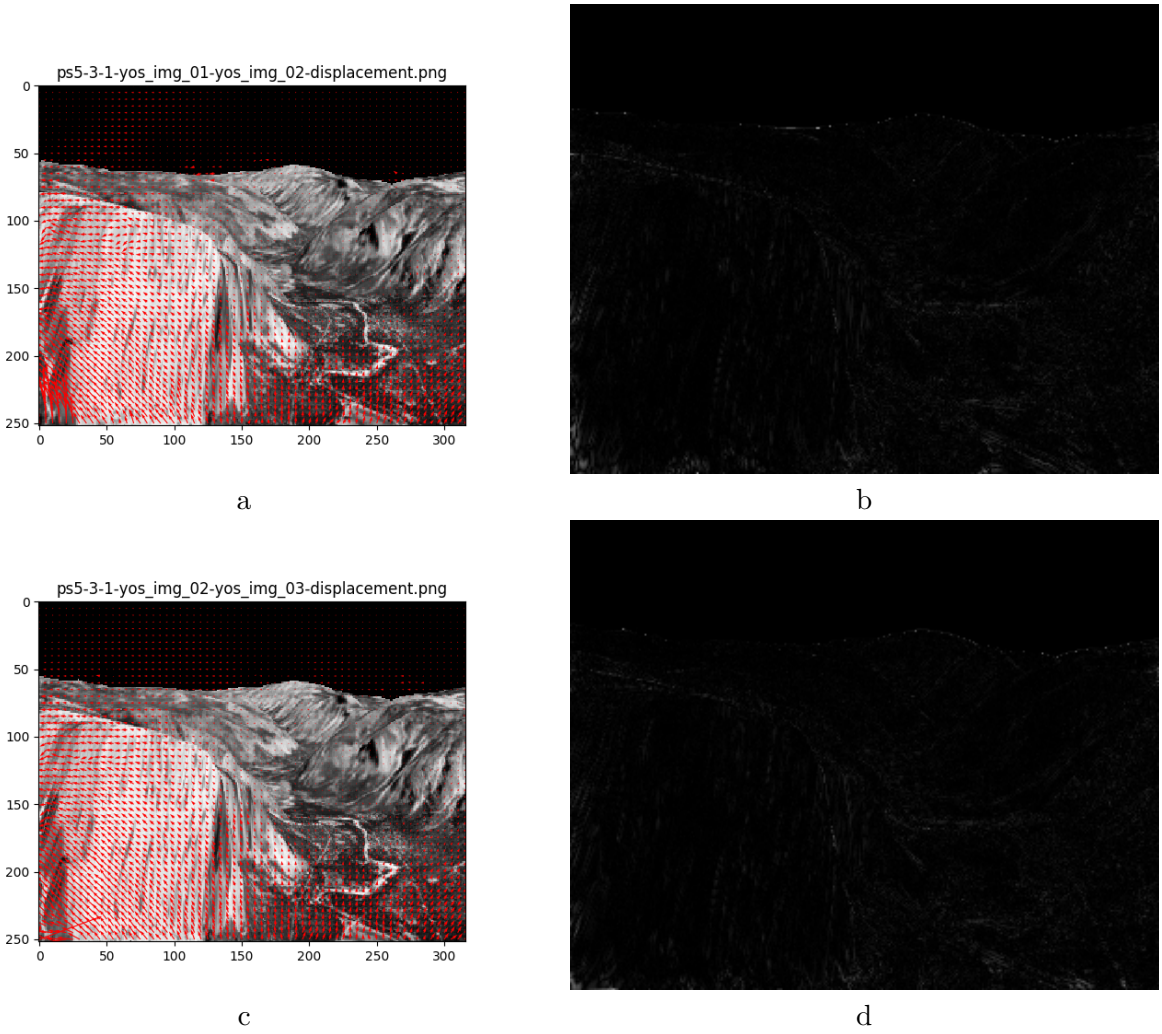


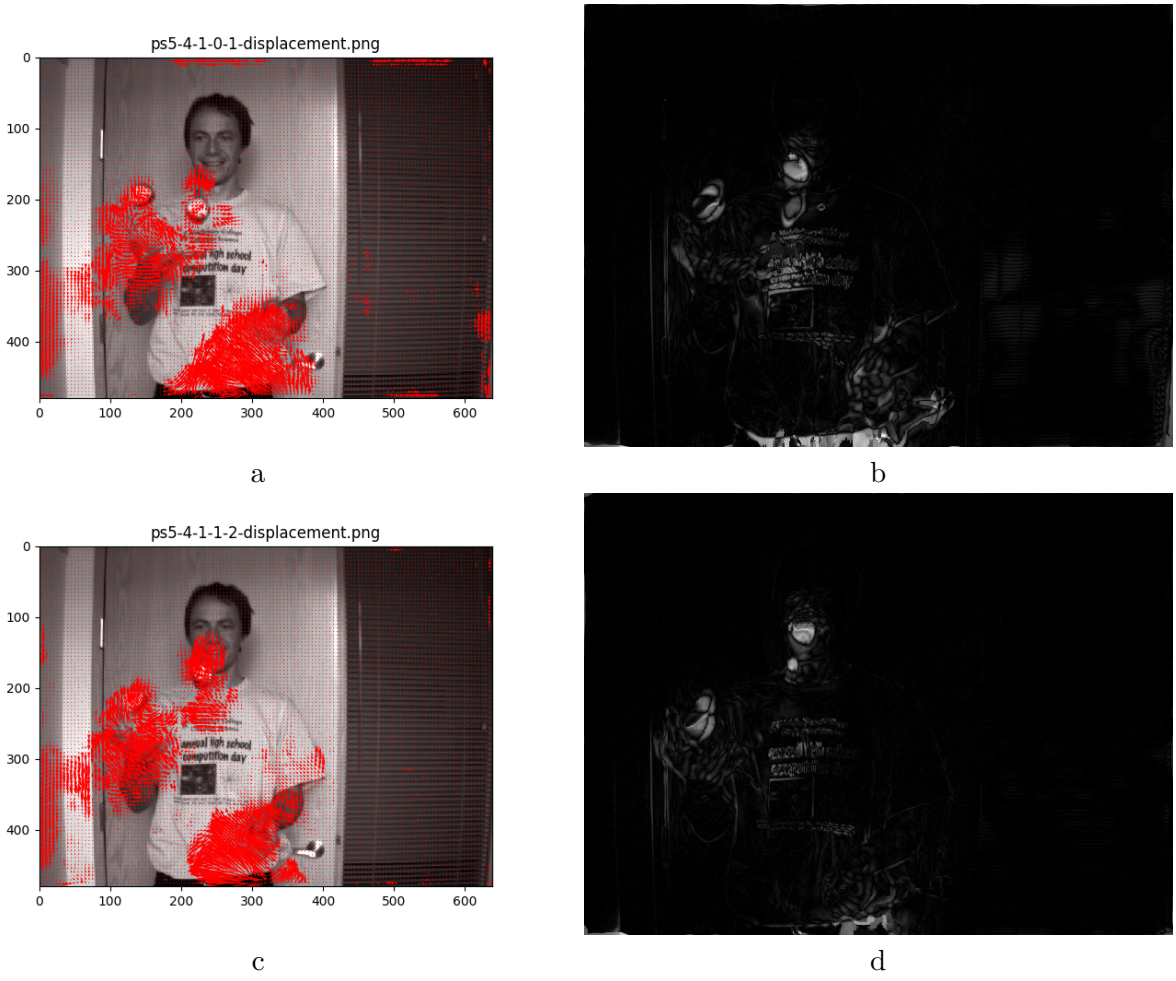
Figure 9: *a.* ps5-3-1-yos-img-01-yos-img-02-displacement.png. *b.* ps5-3-1-yos-img-01-yos-img-02-diff.png. *c.* ps5-3-1-yos-img-02-yos-img-03-displacement.png. *d.* ps5-3-1-yos-img-02-yos-img-03-diff.png.

We observe again several things in the previous results :

- We obtain better result for the TestSeq 10 and 20 pixels shifts than with a single-level LK.
- The process works also fine for DataSeq1 images but the results are not as good for DataSeq2 still because of the strong boundaries issues.
- For coarser levels, adding a large smoothing and weighting window will result in a very important averaging that might not be the best solution especially for very localized motions as in DataSeq2

4 The Juggle Sequence

Finally we test our Hierarchical Luka Kanade on frames of a juggler and we try to detect the motion of his arms and juggling balls.
But as for DataSeq2, the motions are very localized and quite larger so we expect our implementation not to perform well.



a b

c d

Figure 10: *a.* ps5-4-1-0-1-displacement.png. *b.* ps5-4-1-0-1-diff.png. *c.* ps5-4-1-1-2-displacement.png. *d.* ps5-4-1-1-2-diff.png.