Computer vision – Final Report – View morphing & interpolation

The goal of this report is to describe my attempts at implementing a view morphing algorithm. Thus, I will explain all the steps I went through for this report.
The globality of my code will be written at the end of this report.

I decided to implement a view morphing algorithm because this was one of the most fascinating computer vision algorithms for me. I find the ability to artificially create a new image from two other images truly interesting, and that is why I wanted to challenge myself to do the same.
I chose to follow the exact outline of the exercise 11.11 in the book *Computer Vision: Algorithms and Applications*.

> **Ex 11.11: View morphing and interpolation.** Implement automatic view morphing, i.e., compute two-frame structure from motion and then use these results to generate a smooth animation from one image to the next (Section 11.3.3).
>
> 1. Decide how to represent your 3D scene, e.g., compute a Delaunay triangulation of the matched point and decide what to do with the triangles near the border. (Hint: try fitting a plane to the scene, e.g., behind most of the points.)
> 2. Compute your in-between camera positions and orientations.
> 3. Warp each triangle to its new location, preferably using the correct perspective projection (Szeliski and Shum 1997).

Unfortunately, the implementation of this exercise was not a success at the end for me, so I decided to slightly change my plans and tried to compute face morphing.

The initial idea: view morphing between 2 images of Paris buildings

My first idea was to compute the intermediate views between both images seen above.

1) Match feature points for the computation of the fundamental matrix

The first step to do this was to match look-a-like feature points of images 1 and 2 together, so that the algorithm could make the link between both images. At first, I had used cv2.SIFT_create() method to detect feature points on both images, and then cv2.BFMatcher() to put together feature points that looked alike the most. However, this method did not give good results. That Is why I finally chose to select manually all the matches: at the start of the program, the algorithm asks the user to click on the images to form the pairs.
This means that I used a sparse representation to construct my view morphing, and I think that this was a mistake.
With those matches, I could determine the fundamental matrix between two views with cv2.findFundamentalMat(), and then things became more complicated.

2) Determine the complete transformation for every feature point

As suggested by the outline of the exercise, the goal of the algorithm was to transform the points' position according to the epipolar geometry. However, the only info in both of my images was the pixels' position, which means that I did not have any info about the 3D world coordinates nor the camera intrinsic parameters. Thus, I came with a way to determine this information using only the fundamental matrix. I considered that all the feature points would belong to the same plane, and thus go throughout the same transformation. The approximations caused by this way of doing things may have been one of the reasons why my algorithm did not properly work.

The first thing to do was to determine the essential matrix from the fundamental matrix, which would help us determine the rotation and translation associated to the 3D real world coordinates of each feature point. However, the calibration matrices were needed to compute it, and I had a lot of trouble estimating them. Below is the formula of the calibration matrix.

$$\mathbf{K} = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

In order to determine K_1 and K_2 (respectively associated to the first and second image), I formulated the hypothesis that $f_x = f_y = f,\ s = 0$, as suggested by the book *Computer Vision: Algorithms and Applications*. (c_x, c_y) being roughly the center of the image, the only unknown parameter was f. However, it was very complex to estimate f. After reading various research articles, I decided to implement Kanatani and Matsugana's algorithm (2000)

Input: A fundamental matrix $\boldsymbol{F}$.
Output: Two focal lengths f and f'.
Procedure:
1. Compute the following quantities:

$$a = \frac{\|\boldsymbol{F}\boldsymbol{F}^\top \boldsymbol{k}\|^2}{\|\boldsymbol{F}^\top \boldsymbol{k}\|^2}, \qquad b = \frac{\|\boldsymbol{F}^\top \boldsymbol{F}\boldsymbol{k}\|^2}{\|\boldsymbol{F}\boldsymbol{k}\|^2},$$

$$c = \frac{(\boldsymbol{k}, \boldsymbol{F}\boldsymbol{k})^2}{\|\boldsymbol{F}^\top \boldsymbol{k}\|^2 \|\boldsymbol{F}\boldsymbol{k}\|^2}, \qquad d = \frac{(\boldsymbol{k}, \boldsymbol{F}\boldsymbol{F}^\top \boldsymbol{F}\boldsymbol{k})}{(\boldsymbol{k}, \boldsymbol{F}\boldsymbol{k})}, \quad (3)$$

$$A = \frac{1}{c} + a - 2d, \qquad B = \frac{1}{c} + b - 2d, \quad (4)$$

$$P = 2\left(\frac{1}{c} - 2d + \frac{1}{2}\|\boldsymbol{F}\|^2\right),$$

$$Q = -\frac{A+B}{c} + \frac{1}{2}\left(\|\boldsymbol{F}\boldsymbol{F}^\top\|^2 - \frac{1}{2}\|\boldsymbol{F}\|^4\right). \quad (5)$$

2. Solve the following quadratic equation in Z:

$$(1+cP)Z^2 - (cP^2 + 2P + 4cQ)Z + P^2 + 4cPQ + 12AB = 0. \quad (6)$$

3. Choose from among the two solutions the one for which the following is smaller (ideally zero):

$$\left| Z^3 - 3PZ^2 + 2(P^2 + 2Q)Z - 4\left(PQ + \frac{4AB}{c}\right) \right|. \quad (7)$$

4. Compute

$$X = -\frac{1}{c}\left(1 + \frac{2B}{Z-P}\right), \quad Y = -\frac{1}{c}\left(1 + \frac{2A}{Z-P}\right). \quad (8)$$

5. Return the following focal lengths f and f':

$$f = \frac{1}{\sqrt{1 + X/\|\boldsymbol{F}^\top \boldsymbol{k}\|^2}}, \quad f' = \frac{1}{\sqrt{1 + Y/\|\boldsymbol{F}\boldsymbol{k}\|^2}}. \quad (9)$$

It was a bit tedious to implement, but I eventually went through it, and obtained both focal lengths.

After that, I was able to calculate the essential matrix. From this essential matrix, I could extract the rotation matrix and translation vector. Those components can be fetched when computing the Singular Value Decomposition of the Essential Matrix. I will not go through the details of the extraction, but there are 4 possible values for the rotation matrix and 2 possible values for the translation matrix, and I could not discriminate the values, so I had to test manually which translation and rotation had the best view morphing effect (in the end the choice did not make much of a difference).

After obtaining the complete translation and rotation in 3D real world coordinates, I was ready to interpolate them smoothly to create intermediate images.
From those interpolated rotation, translation and calibration matrices, I would get the transformation matrix for each set of coordinates.

$$\tilde{x}_2 = \begin{pmatrix} x_2 \\ y_2 \\ 1 \\ d = 0 \end{pmatrix} = M\tilde{x}_1, \quad M = \tilde{P}_2\tilde{P}_1^{-1}, \quad \tilde{P}_i = \begin{bmatrix} K_i & 0 \\ 0^T & 1 \end{bmatrix}\begin{bmatrix} R_i & t_i \\ 0^T & 1 \end{bmatrix}$$

For convenience, I chose $R_1 = I$ and $t_1 = 0$.

3) Interpolate the rotation, translation and calibration matrices

After obtaining the complete transformation from one image to another, I was now able to compute the gradual transformation for every frame. I decided to create 23 intermediate views between both images, so I had to interpolate 23 times.
I chose linear interpolation for the sake of simplicity, but I could have tried smoother interpolation as well. As said above, I had to interpolate the calibration matrices as well: I linearly interpolated the focal lengths between f_1 and f_2, although it might not have been the right way to do it, which adds another potential source of failure for the final algorithm.
The interpolation of the rotation matrix was more complex than for translation and calibration : I had to switch the complete rotation matrix to quaternions and apply the SLERP (Spherical Linear interpolation) algorithm to it. I used the version of SLERP from the book *Computer Vision: Algorithms and Applications* and implemented it in Python.

procedure $slerp(\mathbf{q}_0, \mathbf{q}_1, \alpha)$:

1. $\mathbf{q}_r = \mathbf{q}_1/\mathbf{q}_0 = (\mathbf{v}_r, w_r)$
2. if $w_r < 0$ then $\mathbf{q}_r \leftarrow -\mathbf{q}_r$
3. $\theta_r = 2\tan^{-1}(\|\mathbf{v}_r\|/w_r)$
4. $\hat{\mathbf{n}}_r = \mathcal{N}(\mathbf{v}_r) = \mathbf{v}_r/\|\mathbf{v}_r\|$
5. $\theta_\alpha = \alpha\,\theta_r$
6. $\mathbf{q}_\alpha = (\sin\frac{\theta_\alpha}{2}\hat{\mathbf{n}}_r, \cos\frac{\theta_\alpha}{2})$
7. **return** $\mathbf{q}_2 = \mathbf{q}_\alpha\mathbf{q}_0$

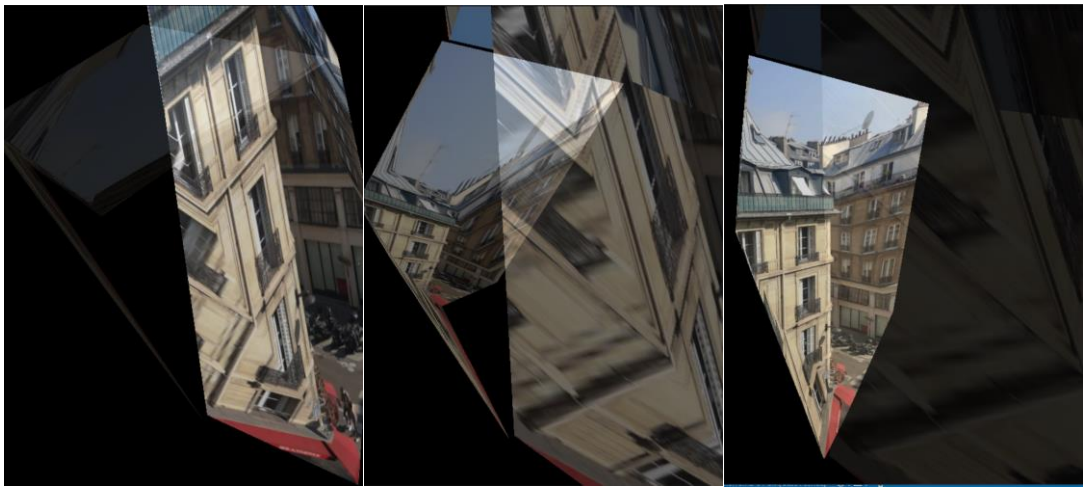
Once those matrices estimated, I could compute the interpolated transformation matrix and apply it to our set of feature points.
It should be noted here that I only described the computing of the transformation matrix from the first image to the second image, but the inverse transformation from the second image to the first image had to be computed as well because the final image is a blend of both transformations.

4) Warp the complete image with Delaunay's triangulation

At the end of step 3, we only had transformed the feature points' position. The last step enabled the algorithm to transform the complete image.
First, I computed a Delaunay triangulation of the feature points. This Delaunay triangulation would be applied to the feature points of the second image as well, following the matches done earlier on. The feature points that were located "the most" on the borders of the image did not undergo the same transformation as the other ones: they were just interpolated linearly from their initial position to their final position in the image.
Then, I warped individually each triangle obtained from the Delaunay triangulation with the function cv2.warpAffine(). I will not go into the details of how this was done, but I cropped the triangles and "pasted" them on the final image thanks to some OpenCV functions.

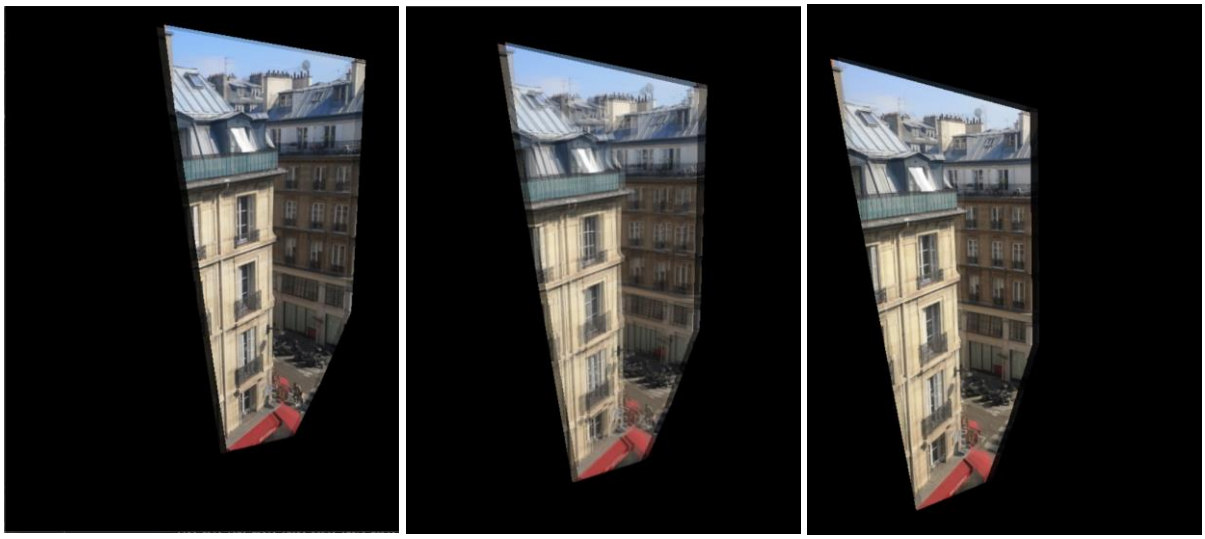
Finally, I blended both transformed images together to get the final output.

However, the result was not what I had expected, as you can see it below.



Frame 5 *Frame 13* *Frame 23*

The problem of the algorithm comes obviously from the determination of the rotation, translation and calibration parameters.
That is why I tried another running of the program with all the feature points being interpolated linearly, as I did it with the points located on the borders earlier on.
Without surprise, the results were a lot better, without being amazing:

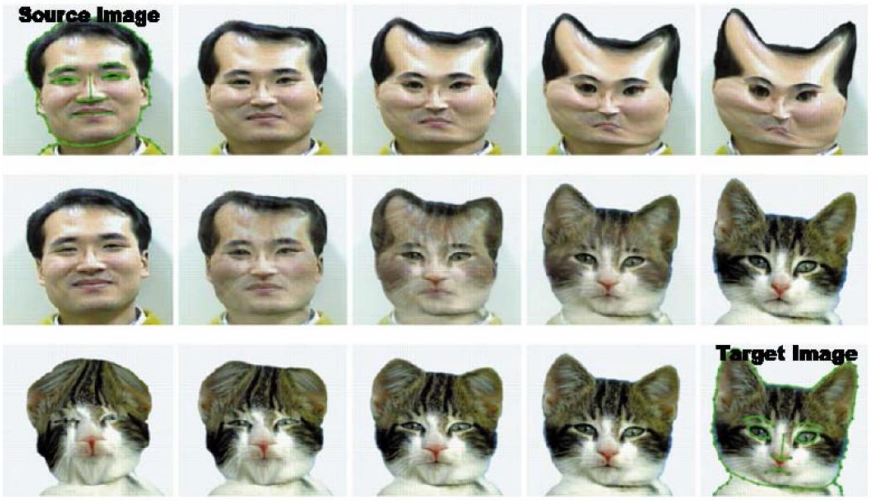


Frame 5 *Frame 13* *Frame 22*

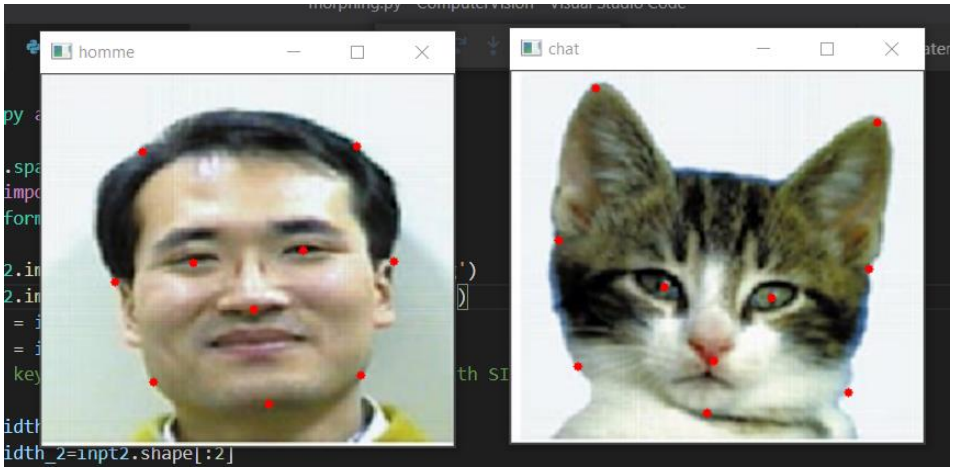
However, I was still not satisfied with having only this as the result of my work, so I decided to do the same with face morphing.

Change of plans: the face morphing

For the face morphing, I only used linear interpolation, like the second implementation of the view morphing algorithm. For my implementation, I chose two well-known pictures of the face of a man and the face of a cat. Those pictures were already used in Wolfberg's image morphing (1998).



I did the same manual selection of the feature points, but I added to the final list of points the corners of the image. I could do it because there was no translation intended between both pictures (unlike the previous image).
The results were pretty good.



Manual selection of the feature points

A first try

With an adequate choice of matching points, I was able to create a little animation, which is attached to this report.

<u>Conclusion: Overview and potential upgrades</u>

This project is for me only half a success, because one of the main parts of my algorithm (the rotation, translation and calibration determination) does not work. However, I was still able to generate some good-looking animation on face morphing, so I am still satisfied with its outcome. We could think of several upgrades to this program though. Well, the main upgrade would be to make the rotation, calibration and translation evaluation work, but there could be other upgrades such as smoother interpolation of feature points or a better interface with the user for the manual definition of the points.

Overall, I spent many hours on this project, but I think that it had a big impact on my OpenCV and Python skills, so I am happy to have worked this much on it.

CODE

Main

```
import numpy as np
import cv2
from scipy.spatial import Delaunay
from numpy import linalg as LA
from scipy.spatial.transform import Rotation as R
from math import *
from focalcompute import focal
from transform import *

inpt1 = cv2.imread('C:/Users/jules/Pictures/paris1.jpg')
inpt2 = cv2.imread('C:/Users/jules/Pictures/paris2.jpg')
inpt1_copy = inpt1.copy()
inpt2_copy = inpt2.copy()
# find the keypoints and descriptors of each image with SIFT

height_1,width_1=inpt1.shape[:2]
height_2,width_2=inpt2.shape[:2]

list_pt1=[]
list_pt2=[]

def click_event(event, x, y, flags, params):
    global counter
    if event == cv2.EVENT_LBUTTONDOWN:
        if counter%2==0:
            cv2.circle(inpt1_copy,(x,y),3,(0,0,255),thickness=-1)
            cv2.imshow('homme', inpt1_copy)
            list_pt1.append((x,y))
        else:
            cv2.circle(inpt2_copy,(x,y),3,(0,0,255),thickness=-1)
            cv2.imshow('chat', inpt2_copy)
            list_pt2.append((x,y))

for counter in range(24):
    print(counter)
    print(list_pt1)
    print(list_pt2)
    if counter%2==0:
        cv2.imshow('homme',inpt1_copy)
        cv2.setMouseCallback('homme', click_event)
        cv2.waitKey(0)
    if counter%2!=0:
        cv2.imshow('chat',inpt2_copy)
```

```
        cv2.setMouseCallback('chat', click_event)
        cv2.waitKey(0)

pt1=np.asarray(list_pt1)
pt2=np.asarray(list_pt2)

#list_pt1.extend([(0.0,0.0),(0.0,float(height_1)),(float(width_1),0.0),(float(
width_1),float(height_1))])
#list_pt2.extend([(0.0,0.0),(0.0,float(height_1)),(float(width_1),0.0),(float(
width_1),float(height_1))])

#compute Delaunay triangulation from 25 matches
pt1=np.asarray(list_pt1)
pt2=np.asarray(list_pt2)
tri=Delaunay(list_pt1)
simplices = tri.simplices

#first, we find F fundamental matrix

F,mask = cv2.findFundamentalMat(pt1,pt2,cv2.FM_RANSAC)

#then, we estimate the focal lengths associated to this fundamental matrix. We
 use Kanatani and Matsunaga's algorithm
f1,f2 = focal(F)
#once we get the focals, we can get the K1 and K2 calibration matrices, assumi
ng that the camera centers are the image centers
c_x_1=width_1/2
c_y_1=height_1/2
c_x_2=width_2/2
c_y_2=height_2/2
K1 = np.asarray([[f1,0,c_x_1],[0,f1,c_y_1],[0,0,1]])
K2 = np.asarray([[f2,0,c_x_2],[0,f2,c_y_2],[0,0,1]])

#then we get the essential matrix from the fundamental matrix, and from this e
ssential matrix we derive the translation and rotation between both images
E=np.transpose(K2)@F@K1
U,s,VT= np.linalg.svd(E)
t = U[:,2] #the translation vector is the 3rd singular vector, associated to t
he value 0

R_plus90 = np.asarray([[0,-1,0],[1,0,0],[0,0,1]])
R_moins90 = np.asarray([[0,1,0],[-1,0,0],[0,0,1]])
R_1 = U@R_plus90@VT
R_2 = U@R_moins90@VT
R_3 = -U@R_plus90@VT
R_4 = -
U@R_moins90@VT #we find the rotation matrix. There are 4 different rotation ma
trices that are possible
```

```
num_frames = 25
find=find_corners(pt1)
for frame_num in range(num_frames):

    frame_num_2=num_frames-1-frame_num
    transfo_pt1=np.zeros(pt1.shape)
    transfo_pt2=np.zeros(pt2.shape)

    R_id=np.identity(3)
    R_1inv=np.linalg.inv(R_1)
    t_0=0
    R_incr_1=rotation(R_id,R_1,num_frames, frame_num)
    R_incr_2=rotation(R_id,R_1inv,num_frames, frame_num_2)
    t_incr_1=translation(t_0,t,num_frames, frame_num)
    t_incr_2=translation(t_0,-t,num_frames, frame_num_2)
    K2_1,K2_2=intrinsic(f1,f2,num_frames, frame_num,c_x_2,c_y_2)

    for i in range(transfo_pt1.shape[0]):
        if i in find:
            transfo_pt1[i][0],transfo_pt1[i][1]=transform_corners(pt1[i][0],pt
1[i][1],pt2[i][0],pt2[i][1],num_frames, frame_num)
            transfo_pt2[i][0],transfo_pt2[i][1]=transform_corners(pt2[i][0],pt
2[i][1],pt1[i][0],pt1[i][1],num_frames, frame_num_2)
        else:
            transfo_pt1[i][0],transfo_pt1[i][1]=transform_normal(R_id,R_incr_1
,t_0,t_incr_1,pt1[i][0],pt1[i][1],K1,K2_1)
            transfo_pt2[i][0],transfo_pt2[i][1]=transform_normal(R_id,R_incr_2
,t_0,t_incr_2,pt2[i][0],pt2[i][1],K2,K2_2)



    output1 = np.zeros(inpt1.shape, dtype=inpt1.dtype)
    output2 = np.zeros(inpt1.shape, dtype=inpt1.dtype)
    for i in range(simplices.shape[0]) :
        src_1 = pt1[simplices][i]
        src_1 = np.float32(src_1.astype(int))
        transfo_1 = transfo_pt1[simplices][i]
        transfo_1 = np.float32(transfo_1.astype(int))
        src_2 = pt2[simplices][i]
        src_2 = np.float32(src_2.astype(int))
        transfo_2 = transfo_pt2[simplices][i]
        transfo_2 = np.float32(transfo_2.astype(int))
        output1=warp_tri(inpt1, src_1, transfo_1,output1)
        output2=warp_tri(inpt2, src_2, transfo_2,output2)

    output3 = cv2.addWeighted(output1, 1-
(frame_num/num_frames), output2, frame_num/num_frames, 0.0)
    cv2.imshow('image no %d' % frame_num, output3)
```

```
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

transform.py (for the interpolations, the computing of the transformation matrix and the warping)

```
import numpy as np
from math import *
from numpy import linalg as LA
from scipy.spatial.transform import Rotation as R
from quaternionop import quaternion_multiply, quaternion_divide
import cv2

def slerp(quat1, quat2, alpha):

    q_r=quaternion_divide(quat2,quat1)
    w_r=q_r[3]
    q_r= np.asarray(q_r)
    if w_r<0 :
        q_r= - q_r

    v_r=q_r[0:3]

    theta_r = 2*atan(LA.norm(v_r)/w_r)
    n_r = v_r/LA.norm(v_r)

    theta_alpha = alpha * theta_r

    theta_n_r = sin(theta_alpha/2)*n_r
    q_alpha =  [theta_n_r[0], theta_n_r[1], theta_n_r[2], cos(theta_alpha/2)]

    slerp1 = quaternion_multiply(q_alpha,quat1)

    return slerp1

def rotation(R_1,R_2,num_frames, frame_num):
    r1=R.from_dcm(R_1)
    q1=r1.as_quat()
    r2=R.from_dcm(R_2)
    q2=r2.as_quat()
    q_i=slerp(q1,q2,frame_num/num_frames)
    r = R.from_quat(q_i)
    R_i=r.as_dcm()
    return R_i

def translation(t_1,t_2,num_frames, frame_num):
    t=(t_2-t_1)*frame_num/num_frames+t_1
```

```
    return t

def intrinsic(f1,f2,num_frames, frame_num,c_x_2,c_y_2):
    f_nor=(f2-f1)*frame_num/num_frames+f1
    f_inv=(f1-f2)*(num_frames-frame_num-1)/num_frames+f2
    K2_nor = np.asarray([[f_nor,0,c_x_2],[0,f_nor,c_y_2],[0,0,1]])
    K2_inv = np.asarray([[f_inv,0,c_x_2],[0,f_inv,c_y_2],[0,0,1]])
    return K2_nor,K2_inv

def transform_normal(R_1,R_incr,t_1,t_incr,x,y,K1,K2):
    R_t_init_1=np.identity(4)
    R_t_init_1[0:3,0:3]=R_1
    R_t_init_1[0:3,3]=t_1
    K_ext_1=np.identity(4)
    K_ext_1[0:3,0:3]=K1
    P_1= K_ext_1@R_t_init_1

    R_t_init_2=np.identity(4)
    R_t_init_2[0:3,0:3]=R_incr
    R_t_init_2[0:3,3]=t_incr
    K_ext_2=np.identity(4)
    K_ext_2[0:3,0:3]=K2
    P_2= K_ext_2@R_t_init_2

    M=P_2@np.linalg.inv(P_1)
    x1=np.asarray([x,y,1,0]) #we consider that the depth d = 0
    x2=M@x1
    return x2[0], x2[1]

def transform_corners(x1,y1,x2,y2,num_frames, frame_num):
    x=(x2-x1)*frame_num/num_frames+x1
    y=(y2-y1)*frame_num/num_frames+y1
    return x,y

def find_corners(array):
    idx_max_x=0
    idx_max_y=0
    idx_min_x=0
    idx_min_y=0
    max_x=array[0][0]
    max_y=array[0][1]
    min_x=array[0][0]
    min_y=array[0][1]
    for i in range (array.shape[0]):
        if array[i][0]>max_x:
            idx_max_x=i
            max_x=array[i][0]
        if array[i][1]>max_y:
```

```
            idx_max_y=i
            max_y=array[i][1]
        if array[i][0]<min_x:
            idx_min_x=i
            min_x=array[i][0]
        if array[i][1]<min_y:
            idx_min_y=i
            min_y=array[i][1]
    return [idx_max_x,idx_max_y,idx_min_x,idx_min_y]

def warp_tri(inpt, tri1, tri2,output):
    height_1,width_1=inpt.shape[:2]

    r1 = cv2.boundingRect(tri1)
    r2 = cv2.boundingRect(tri2)

    tri1Cropped=[]
    tri2Cropped=[]
    for i in range(0, 3):
        tri1Cropped.append(((tri1[i][0] - r1[0]),(tri1[i][1] - r1[1])))
        tri2Cropped.append(((tri2[i][0] - r2[0]),(tri2[i][1] - r2[1])))

    img1Cropped = inpt[r1[1]:r1[1] + r1[3], r1[0]:r1[0] + r1[2]]
    warpMat = cv2.getAffineTransform(np.float32(tri1Cropped), np.float32(tri2C
ropped))
    img2Cropped = cv2.warpAffine( img1Cropped, warpMat, (r2[2], r2[3]), None,
flags=cv2.INTER_LINEAR, borderMode=cv2.BORDER_REFLECT_101 )

    mask = np.zeros((r2[3], r2[2], 3), dtype = np.float32)
    cv2.fillConvexPoly(mask, np.int32(tri2Cropped), (1.0, 1.0, 1.0), 16, 0)
    img2Cropped = img2Cropped * mask
    for j in range(r2[1],r2[1]+r2[3]):
        for i in range(r2[0],r2[0]+r2[2]):
            if 0<i <width_1:
                if 0<j<height_1:
                    if mask[j-r2[1],i-r2[0]][0]==1.0 and mask[j-r2[1],i-
r2[0]][1]==1.0 and mask[j-r2[1],i-r2[0]][2]==1.0 :
                        output[j, i] = img2Cropped[j-r2[1],i-r2[0]]
    return output
```

quaternionop.py (for defining operations on quaternions)

```
import numpy as np
from numpy import linalg as LA

def quaternion_multiply(quat1,quat2):
    v1,v2=np.asarray(quat1[0:3]),np.asarray(quat2[0:3])
    w1,w2=quat1[3],quat2[3]
```

```
    v = w1*v2
    v = np.cross(v1,v2) + w1*v2 + w2*v1
    m = [ v[0] , v[1] , v[2] , w1*w2 - np.dot(v1,v2)]
    return m

def quaternion_divide(quat1,quat2):
    v1,v2=np.asarray(quat1[0:3]),np.asarray(quat2[0:3])
    w1,w2=quat1[3],quat2[3]
    v = np.cross(v1,v2) + w1*v2 - w2*v1
    d = [ v[0] , v[1] , v[2] , w1*w2 - np.dot(v1,v2)]
    return d
```

focalcompute.py (implementation of Kanatani and Matsugana's algorithm)

```
import numpy as np
from numpy import linalg as LA

def focal(F):
    k=np.transpose(np.asarray([0,0,1]))
    F_t=np.transpose(F)

    # Step 1 : Compute a,b,c,d,A,B,P,Q
    a=(LA.norm(F@F_t@k))**2/(LA.norm(F_t@k))**2
    b=(LA.norm(F_t@F@k))**2/(LA.norm(F@k))**2
    c=np.dot(k,F@k)**2/((LA.norm(F_t@k))**2*(LA.norm(F@k))**2)
    d=np.dot(k,F@F_t@F@k)/np.dot(k,F@k)
    A=1/c+a-2*d
    B=1/c+b-2*d
    P=2*(1/c-2*d+0.5*LA.norm(F)**2)
    Q=-(A+B)/c+0.5*(LA.norm(F@F_t)**2-0.5*LA.norm(F)**4)

    # Step 2 : Solve a quadratic equation in z
    coeffs=[1+c*P,-(c*P**2+2*P+4*c*Q),P**2+4*c*P*Q+12*A*B]
    z=np.roots(coeffs)
    z1,z2=z[0],z[1]

    # Step 3 : Pick the solution for which inter_amount() is the smallest
    inter_amount= lambda x: abs(x**3 - 3*P*x**2 + 2*(P**2+2*Q)*x-
4*(P*Q+4*A*B/c))
    if inter_amount(z1)<inter_amount(z2):
        z_p=z1
    else:
        z_p=z2

    # Step 4 : Compute X & Y
    X=-1/c*(1+2*B/(z_p-P))
```

```
    Y=-1/c*(1+2*A/(z_p-P))

    # Step 5 : Return the focal lengths f1 and f2
    f1=1/abs(1+X/(LA.norm(F_t@k))**2)**0.5
    f2=1/abs(1+Y/(LA.norm(F@k))**2)**0.5

    return f1,f2
```