ITMO University

Image Processing:
Lab2

Prepared by

Bassel Alshawareb

April 11, 2023

1. Introduction

Image transformations refer to the process of modifying the appearance of an image through various techniques. These techniques can be applied to a wide range of image types, including photographs, digital art, and graphics. Image transformations can be used to correct image imperfections, enhance specific features. In this lab we are reviewing some of the most common transformations used in image processing.

2. Transformations

In this lab we will consider applying image transformations on the following images:

Figure 1 Images on which we will apply transformations. On the left is an image that contains coins, which have geometric shapes, and on the right is an image of woman in red dress..

Piecewise Linear Transformation:

Figure 2 coins after applying Piecewise linear transformation.

Projection:

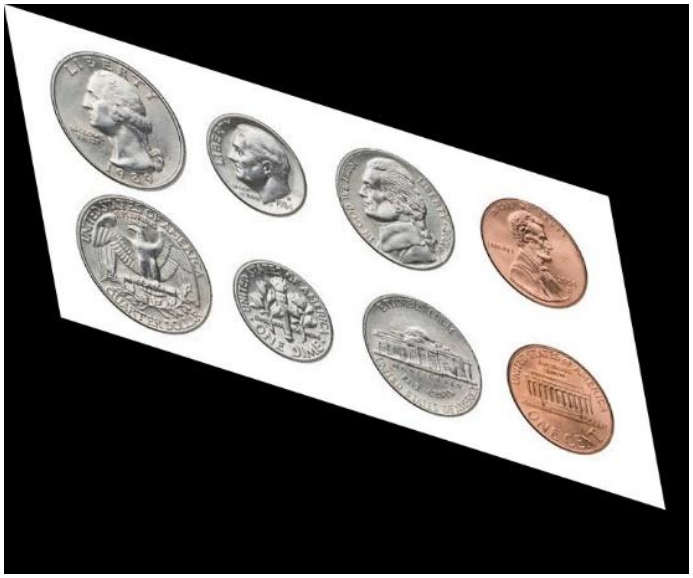


Figure 3 coins after projection.

Polynomial:

Figure 4 coins after applying polynomial transformation.

Sinusoidal:

Figure 5 Woman in the red dress after applying sinusoidal transformation

3. Removing Barrel and binocular distortions

We start by getting distorted images. To do that, we apply barrel and binocular transformations.

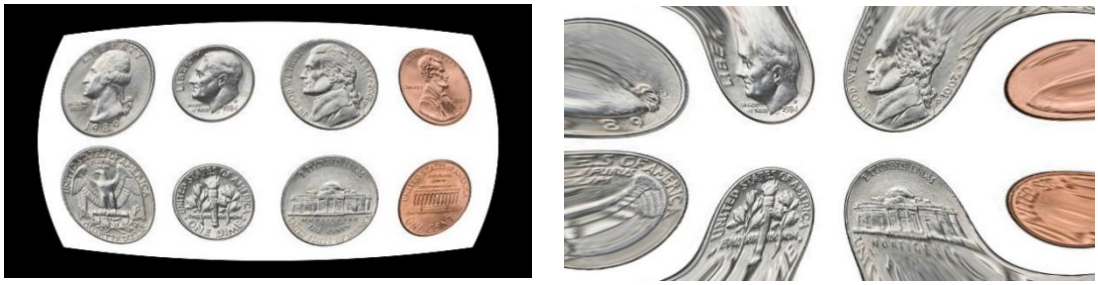


Figure 6 A) Coins after applying Barrel. B) Coins after applying Binocular.

Now we remove the Barrel by applying binocular transformation

Figure 7 The coins image after applying barrel followed by binocular distortions.

4. Stitching

Image stitching or photo stitching is the process of combining multiple photographic images with overlapping fields of view to produce a segmented panorama or high-resolution image.

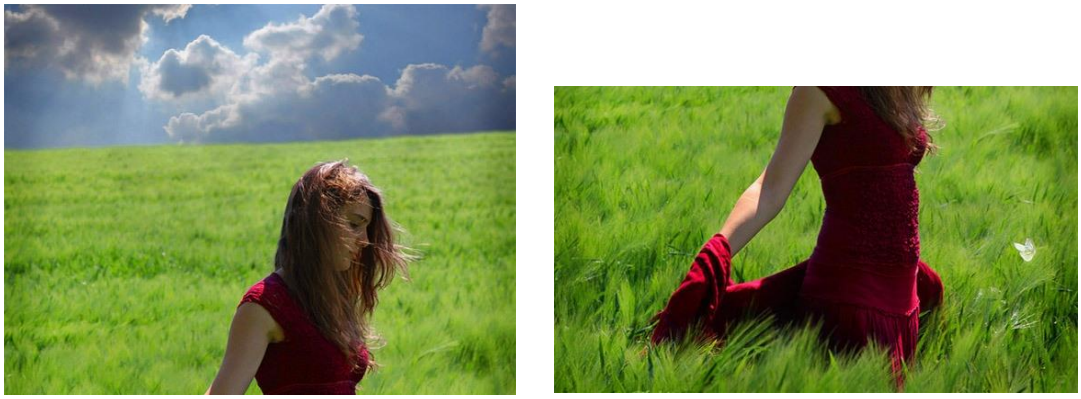
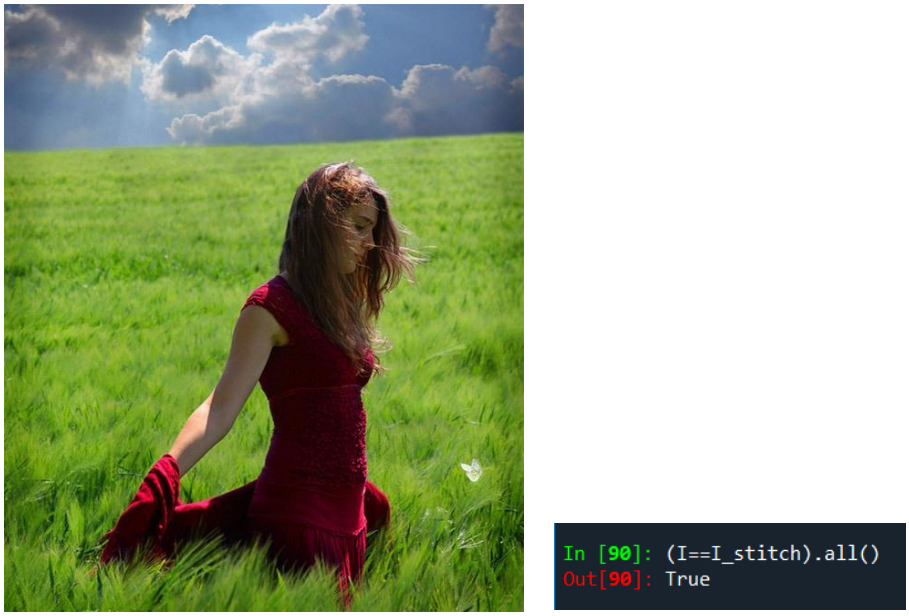


Figure 8 Two cropped parts from the same image.



```
In [90]: (I==I_stitch).all()
Out[90]: True
```

Figure 9 The result of stitching the 2 previous images. We can realize that the output image is identical to the original image.

Questions:

1. One way to rotate an image without using a rotation matrix is to perform a series of translations and shears on the image to achieve the desired rotation. Another way in the case of requiring special degrees of rotations, like for example, a rotation 180 degrees. In such cases, we can perform the rotation simply by reversing the coordinates of the pixels along x and/or y axes.

2. If the transformation order n = 4, at least four corresponding pairs of points must be specified on the original and distorted images. This is because a fourth-order transformation requires four control points to uniquely determine the transformation parameters.

3. The reason of having undefined values in the image after transformation, is because of the resulting float values from the transformation, or values outside of image boundaries. One way to deal with this problem is to take the median of the neighboring values. Another way is to average all neighboring pixels. In both cases, a parameter should be defined which is the size of the window, in which we will look for neighbors.

Code:

Code from file Histogram Processing

```
# -*- coding: utf-8 -*-
"""
Created on Sat Mar 25 13:21:23 2023

@author: Bassel
"""

import cv2
import numpy as np
import matplotlib.pyplot as plt
import os

class myHist:
    id_plot=0
    def __init__(self, img=None,histSize=256, histRange=(0,256),
CONFIG="BGR", EQUALIZE = True, pth = "default"):
        self.img_org = None
        self.history = []
        if pth == "default":
            path = os.getcwd()
        else:
            path = pth
        myHist.path_input = os.path.join(path,"inputs")
        myHist.path_output = os.path.join(path , "outputs")
        self.last_executed = ""
        self.histSize=256
        self.EQUALIZE = EQUALIZE
        if CONFIG=="BGR":
            a=[0,1,2]
        else:
            a=[2,1,0]
        self.order=a
        myHist.histSize=histSize
        myHist.histRange=histRange
        if img is not None:
            self.set_img(img)
        else:
            self.img=None

    def get_img(self):
        return self.img

    def set_img(self, img, text = None):
        if img is None:
            return
```

```
        if text is not None:
            print(text)
        if self.img_org is None:
            print("added origin")
            self.img_org = img
        self.img = img
        self.rows, self.cols = self.img.shape[0:2]
        self.history.append(self.img)
        self.calc()


    def calc(self,img=None):
        print("Calculating Histogram")
        if img is None:
            if self.img is None:
                print("error")
                return
            img=self.img
        img_s=cv2.split(img)
        bHist=cv2.calcHist(img_s,[0],None, [256], (0, 256))
        gHist=cv2.calcHist(img_s,[self.order[1]],None,
[self.histSize], (0, 256))
        rHist=cv2.calcHist(img_s,[self.order[2]],None,
[self.histSize], (0, 256))
        self.img=img
        self.bH=bHist
        self.gH=gHist
        self.rH=rHist
        if self.EQUALIZE:
            self.equalize()

    def equalize(self):
        if self.last_executed == "":
            self.last_executed = "equalized"
        print("Equalizing")
        self.bH_not_normalized = self.bH
        self.gH_not_normalized = self.gH
        self.rH_not_normalized = self.rH
        max_b = np.max(self.bH)
        max_g = np.max(self.gH)
        max_r = np.max(self.rH)
        self.bH = self.bH/max_b
        self.gH = self.gH/max_g
        self.rH = self.rH/max_r
        self.EQUALIZE = True




    def show(self, image = "current", name=None):
        if image == "org":
```

```
            I = self.img_org

        else:
            I = self.img

        if name is None:
            name="number"+str(myHist.id_plot)
        image_name = name + "_" + self.last_executed
        image_path = myHist.path_output + "/" + "images"
        hist_path = myHist.path_output + "/" + "Histograms"

        try:
            os.mkdir(image_path)
        except IOError:
            pass
        try:
            os.mkdir(hist_path)
        except IOError:
            pass

        myHist.id_plot=myHist.id_plot+1
        fig = plt.figure(myHist.id_plot)
        t=range(256)
        plt.plot(t,self.bH, color="blue")
        plt.plot(t,self.gH, color="green")
        plt.plot(t,self.rH, color="red")
        plt.suptitle(name)
        plt.savefig(hist_path + "/" + image_name  + "_Histogram" +
".png")
        plt.show()
        self.__show_img(I)
        plt.imsave(image_path + "/" + image_name + ".jpg", I)

    def __show_img(self, I):
        I = cv2.cvtColor(I, cv2.COLOR_BGR2RGB)
        myHist.id_plot=myHist.id_plot+1
        plt.figure(myHist.id_plot)
        plt.imshow(I)
        plt.show()

    def show_original(self):
        self.show(image = "org")

    def show_history(self):
        myHist.id_plot=myHist.id_plot+1
        for img in self.history:
            self.__show_img(img)
            plt.show()


def profile(img, x):
```

```
    return img[x,:]

def project_(img,xy):
    return np.sum(img,xy)/(img.shape[(xy+1)%2])



if __name__ == "__main__":
    pass
```

Code from file transformations

```
# -*- coding: utf-8 -*-
"""
Created on Thu Mar 30 16:44:51 2023

@author: Bassel
"""
import cv2
import numpy as np
import os
from scipy.optimize import fsolve
import Histogram_processing

class transformation(Histogram_processing.myHist):
    def __init__(self, img=None,histSize=256, histRange=(0,256),
CONFIG="BGR", EQUALIZE = True, pth = "default", on_origin = True ):
        super().__init__(img ,histSize, histRange, CONFIG, EQUALIZE,
pth)
        self.on_origin = on_origin
        print("origin bool", self.on_origin)
    def __copy_img(self):
        if self.on_origin == True:
            return self.img_org.copy()
        else:
            return self.img.copy()

    def shift(self, amount=50):
        I = self.__copy_img()
        if I is None:
            print("error")
            return
        I1=I

I1[:,:,0]=np.clip(I1[:,:,0].astype(np.int16)+amount,0,255).astype(np.
uint8)

I1[:,:,1]=np.clip(I1[:,:,1].astype(np.int16)+amount,0,255).astype(np.
uint8)
```

```
I1[:,:,2]=np.clip(I1[:,:,2].astype(np.int16)+amount,0,255).astype(np.
uint8)
        self.set_img(I1)

    def __filter_high_frequencies(self, H):
        thresholded = np.where(H<= 10**(-2),10, H) #Filtering out
small frequencies
        i_min = 0
        print("length of H ", thresholded.shape[0])
        for i in range(thresholded.shape[0]):
            if thresholded[i] !=10:
                i_min = i
                break
        i_max = 255
        for i in range(thresholded.shape[0]):
            if thresholded[-1-i] !=10:
                i_max = 255-i
                break
        return float(i_min)/255, float(i_max)/255

    def extend(self, alpha = 0.5, REMOVE_LOW_FREQUENCY = True):
        if REMOVE_LOW_FREQUENCY:
            self.last_executed = "Extended"
        else:
            s_alpha = str(alpha)
            list_s_alpha = s_alpha.split('.')
            alpha_for_writing = '_'.join(list_s_alpha)
            self.last_executed = "Extended_with_Alpha" +
alpha_for_writing
        self.alpha = alpha
        I_temp = self.__copy_img()
        I = I_temp.astype(np.float64)/255
        Ib = I[:,:,0]
        Ig = I[:,:,1]
        Ir = I[:,:,2]
        Iout = []
        if self.EQUALIZE:
            if REMOVE_LOW_FREQUENCY:
                #removing low frequicies
                Ib_min, Ib_max =
self.__filter_high_frequencies(self.bH)
                Ig_min, Ig_max =
self.__filter_high_frequencies(self.gH)
                Ir_min, Ir_max =
self.__filter_high_frequencies(self.rH)
            else:
                Ib_min, Ib_max = np.min(Ib), np.max(Ib)
                Ig_min, Ig_max = np.min(Ig), np.max(Ig)
                Ir_min, Ir_max = np.min(Ir), np.max(Ir)
```

```
            #Extend b
            Ib_extended = ( np.clip((255*((Ib-Ib_min)/(Ib_max -
Ib_min))**alpha),0,255) ).astype(np.uint8)
            Iout.append(Ib_extended)
            #Extend g
            Ig_extended = ( np.clip((255*((Ig-Ig_min)/(Ig_max -
Ig_min))**alpha),0,255) ).astype(np.uint8)
            Iout.append(Ig_extended)
            #Extend r
            Ir_extended = ( np.clip((255*((Ir-Ir_min)/(Ir_max -
Ir_min))**alpha),0,255) ).astype(np.uint8)
            Iout.append(Ir_extended)
            Iout = cv2.merge(Iout)
            self.set_img(Iout, text = "Extend is done")

    def rotate(self, theta = 90):
        I = self.__copy_img()
        phi = theta * np.pi / 180
        T1 = np.float32(
        [[1, 0, -(self.cols - 1) / 2.0],
        [0, 1, -(self.rows - 1) / 2.0],
        [0, 0, 1]])
        T2 = np.float32(
        [[np.cos(phi), -np.sin(phi), 0],
        [np.sin(phi), np.cos(phi), 0],
        [0, 0, 1]])
        T3 = np.float32(
        [[1, 0, (self.cols - 1) / 2.0],
        [0, 1, (self.rows - 1) / 2.0],
        [0, 0, 1]])
        T = np.matmul(T3, np.matmul(T2, T1))[0:2, :]
        I_rotate = cv2.warpAffine(I, T, (np.max(I.shape),
np.max(I.shape)))
        self.set_img(I_rotate, text = "rotated")

    def sinusoid(self):
        I = self.__copy_img()
        u, v = np. meshgrid (np. arange ( self.cols ), np. arange (
self.rows ))
        u = u + 20 * np.sin (2 * np.pi * v / 90)
        I_sinusoid = cv2 . remap (I, u. astype (np. float32 ), v.
astype (np. float32 ), cv2. INTER_LINEAR )
        self.set_img(I_sinusoid)

    def piecewise(self):
        I = self.__copy_img()
        stch=2
        T = np.float32([[stch, 0, 0], [0, 1, 0]])
        I_piecewiselinear = I.copy()
```

```
        I_piecewiselinear[:, int(self.cols/2):, :] =
cv2.warpAffine(I_piecewiselinear[:, int(self.cols/2):, :], T,
(self.cols - int(self.cols/2), self.rows))
        self.set_img(I_piecewiselinear)

    def projection(self):
        I = self.__copy_img()
        T = np. float32 ([[1.1 , 0.2 , 0.00075] ,[0.35 , 1.1 ,
0.0005] ,[0, 0, 1]])
        I_projective = cv2 . warpPerspective (I, T,(2*self.cols ,
2*self.rows ))
        self.set_img(I_projective)

    def barrel(self):
        I = self.__copy_img()
        xi , yi = np. meshgrid (np. arange ( self.cols ), np. arange
( self.rows ))
        midx=self.cols/2
        midy=self.rows/2
        xi=xi-midx
        yi=yi-midy

        r, theta = cv2.cartToPolar(xi/midx, yi/midy)
        F3 = 0.4
        F5 =0
        r = r + F3 * r**3 + F5 * r**5
        u, v = cv2.polarToCart(r, theta)
        u = u * midx + midx
        v = v * midy + midy
        I_barrel = cv2.remap(I, u.astype(np.float32),
v.astype(np.float32), cv2.INTER_LINEAR)
        self.set_img(I_barrel)

    def debarrel(self):
        I = self.__copy_img()
        xi , yi = np. meshgrid (np. arange ( self.cols ), np. arange
( self.rows ))
        midx=self.cols/2
        midy=self.rows/2
        xi=xi-midx
        yi=yi-midy
        r, theta = cv2.cartToPolar(xi/midx, yi/midy)
        F3 = 0.17
        F5 =0
        r = r -F3 * r**3 -F5 * r**5
        u, v = cv2.polarToCart(r, theta)
        u = u * midx + midx
        v = v * midy + midy
        I_debarrel = cv2.remap(I, u.astype(np.float32),
v.astype(np.float32), cv2.INTER_LINEAR)
        self.set_img(I_debarrel)
```

```
    def poly(self):
        I = self.__copy_img()
        T = np.array([[0, 0], [1, 0], [0, 1], [0.00001, 0], [0.002,
0], [0.001, 0]])
        I_polynomial = np.zeros(I.shape, I.dtype)
        x, y = np.meshgrid(np.arange(self.cols),
np.arange(self.rows))
        xnew = np.round(T[0, 0] + x * T[1, 0] + y * T[2, 0] + x * x *
T[3, 0] + x * y * T[4, 0] + y * y * T[5, 0]).astype(np.float32)
        ynew = np.round(T[0, 1] + x * T[1, 1] + y * T[2, 1] + x * x *
T[3, 1] + x * y * T[4, 1] + y * y * T[5, 1]).astype(np.float32)
        mask = np.logical_and(np.logical_and(xnew >= 0, xnew <
self.cols), np.logical_and(ynew >= 0, ynew < self.rows))
        if I.ndim == 2:
            I_polynomial[ynew[mask].astype(int),
xnew[mask].astype(int)] = I[y[mask], x[mask]]
        else:
            I_polynomial [ ynew [ mask ]. astype (int), xnew [ mask
]. astype (int ), :] =I [y[ mask ], x[ mask ], :]
        self.set_img(I_polynomial)

    def stitching(self):
        I = self.__copy_img()
        I_top=I[:int(self.rows/2)+100,:,:]
        self.set_img(I_top)
        I_bottom=I[int(self.rows/2):,:,:]
        self.set_img(I_bottom)
        templ_size = 10
        templ = I_top[-templ_size:, :, :]
        res = cv2.matchTemplate(I_bottom, templ, cv2.TM_CCOEFF)
        min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(res)
        I_stitch = np.zeros((I_top.shape[0] + I_bottom.shape[0] -
max_loc[1] - templ_size, I_top.shape[1], I_top.shape[2]),
dtype=np.uint8)
        I_stitch[0:I_top.shape[0], :, :] = I_top
        I_stitch[I_top.shape[0]:, :, :] = I_bottom[max_loc[1] +
templ_size:, :, :]
        self.set_img(I_stitch)


if __name__ == "__main__":
    #Demo

    path = os.getcwd()
    path_input = os.path.join(path,"inputs")
    path_output = os.path.join(path , "outputs")



    I=cv2.imread(path_input + '/dark_sky.jpg')
```

```
I=cv2.resize(I,(500,500))
img=I.copy()
img=cv2.resize(img,(500,500))

ob = transformation(on_origin = True, img = I)
ob.rotate(45)
ob.extend()
ob.shift()
ob.sinusoid()
ob.piecewise()
ob.poly()
ob.barrel()
ob.debarrel()
ob.stitching()
ob.show_history()
```