Angel U. Ortega
Computer Vision – Lab 3
Version 1.0
10/20/2016

Table of Contents

1. Introduction

1.1. Lab Overview

The lab is composed of five problems, P1 – P5. The problem statement is as follows:

P1. Write a function to perform real-valued indexing on an image. Your function should receive an image I and the (real-valued) pixel coordinates < r, c > and return the RGB values of obtained by performing a weighted average of the pixels, where the weights are proportional to the area of overlap of the pixels in the original image and a hypothetical pixel centered at < r, c >. Use this function and regular rounding to evaluate the results of each of the methods from the following questions.
P2. Write a program to extract the image of a rectangular object seen under perspective in an image and display it without perspective effects. You can do this using the homography-based algorithm described in the textbook, or the line-based algorithm described in class.
P3. Write a program to perform the inverse process from the previous question, that is, it should insert a rectangular region into an image with perspective effects.
P4. Write a program to implement k-nearest neighbor warping, as explained in class. Your program should allow the user to input source and destination points and generate a sequence of images illustrating a smooth transition from the source to the destination image.
P5. Write a program to morph a face image in frontal view into another. Use a variation of your program from the previous question to align the faces, then apply cross-dissolve.

1.2. References

[1] http://www.pyimagesearch.com/2015/03/09/capturing-mouse-click-events-with-python-and-opencv/
[2] http://stackoverflow.com/questions/28650721/cv2-python-image-blending-fade-transition
[3] https://drive.google.com/open?id=0B_kWRxLZdmeJSUlHMm9EaFFSWlU

2. Proposed Solution Design and Implementation

2.1. P1

For this problem, I identified a region composed of the immediate pixels to the left, right, top, and bottom of the float point value (nearest neighbors). This gave me a region of four pixels, as shown in Figure 1. This method retrieved RGB pixels. Then I calculated the corresponding percentage value of each pixel derived from the distance to float value (nearest neighbor distance). I then calculated the value of each pixel channel (A-R, A-G, A-B, …, D-R, D-G, D-B). Finally, I just added the four weighted values per channel, and returned the three sums.

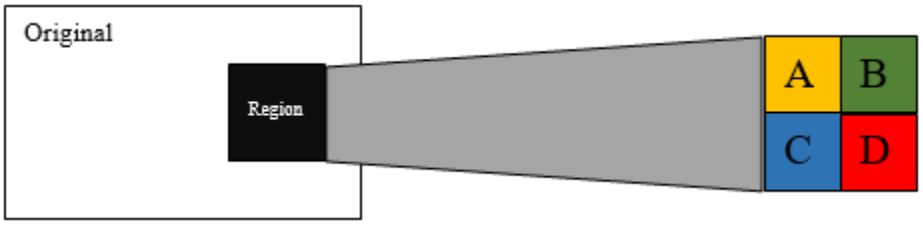


Figure 1: Diagram depicting nearest neighbor selection.

2.2. P2

For this problem, I decided to follow a line based algorithm. Through a deep understanding of trigonometry, I was able to come up with a formula that would derive all the float point numbers in a line between two matching edge points. I began by constructing two linear arrays, one for the left edge, and one for the right edge. I had to have these arrays be the same length if I wanted my extracted image to be rectangular. I implemented my trigonometric formula in a method called getValues(pointA, pointB, index, partitions). This method, basically, returns a point value that lies exactly between pointA and pointB. This retval also lies at a derived distance from either point, depending on the index. This method can be seen in the Appendix. After I had developed this method, I implemented another one that would fill in the rest of the coordinate array, an array with corresponding points to each index. I decided to take this step in the middle so implementation for P3 would be easier. After I had derived the coordinate array that matched to the selected region, I then called extractImage(coordinateArray, image). The extractImage method compiled an image of the extracted region using the coordinate array and the pixel color average method I implemented in P1. The image returned by extractImage is the rectangular version of the image selected by the four points. Results can be seen in the next section.

2.3. P3

For this problem, I took advantage of my implementation in P2. I reused most of the methods from P2, except for the method extractImage(coordinateArray, image), which in turn used the pixel color average method of P1. Instead, I implemented two methods to replace a warped image with a planar one. I had to make sure the source image had the correct proportions to replace, or actually cover-up, the region selected. I used the getCoordinateArray method from P2. I also implemented a resize method that returned a resized version of the source image. The dimensions of this scaled image had to match exactly the dimensions of the coordinate array. I then simply parsed through the coordinate array, and replaced the values of the destination image, with the pixel values of the source image. The results can be seen in the next section.

2.4. P4

This problem proved to be harder to implement than originally expected. The complexity of the iterations, points of interest and number of variables made this program more complicated than the others. I began by designing the user interface I would be needing. I found a tutorial online which I was able to modify to fit my specific needs. The tutorial can be found under references [1]. The tutorial covers how to display an image, then click and drag over the image to draw a rectangle or interest (ROI). The program then crops and displays this ROI. I modified the tutorial so that the program drew a line instead of a rectangle. The program did not need to crop the image, it did, however, need to store the two points clicked. I used cv2 to listen to pressed keys to determine continuity of the program or ending. The user simply needs to click ‘N’ to continue warping, or ‘C’ to complete the warping. This implementation made it easier to do a series of modifications like those showcased by Dr. Fuentes during CS 5354 lecture. For documentation purposes, the program saves the warp selection and warped images to the working directory.

2.5. P5

I implemented this program by modifying my P4 program. I started by concatenating the two images to be morphed into one figure for display purposes. In order to do this, I had to make sure the images were exactly the same size, if not, then I resized on of them to the size of the other. Through ginput, I collected two points, one from each image. After I collected the two points of interest (POs), I evaluated them to make sure they were within range of the images; the x value of one of these points was greater than the image width, so I subtracted the image width from this x value. I then morphed each image using these POs. Using cv2, like P4, I listened to key press to determine continuity or ending of program. Once I had the morphed images, I ran a cross dissolve method of the original images and the morphed images. My cross dissolve method is a modification from a fade transition from stack overflow [2].

3. Experimental Results and Conclusions

All images used for this lab can be accessed at:
https://drive.google.com/open?id=0B_kWRxLZdmeJSTFoZ2syVmcwTXc [3]
Below are just a few examples from the images used and the images derived.

3.1. P1

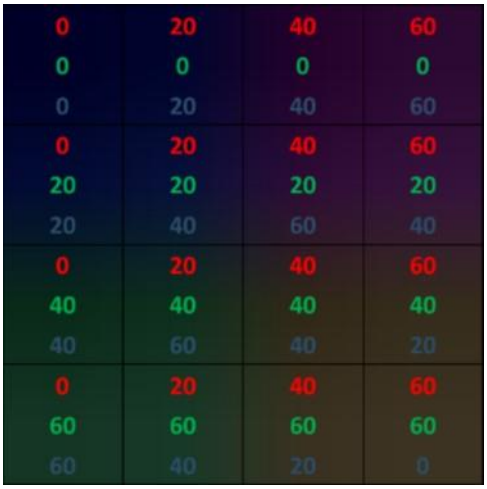


Figure 2: Image created to test correctness of pixelcoloraverage method.

As it can be seen in Figure 2, I created a test image, using the values overlaid to test the correctness of the pixel color average method. Figure 2 also shows the results, which were the ones expected.

3.2. P2

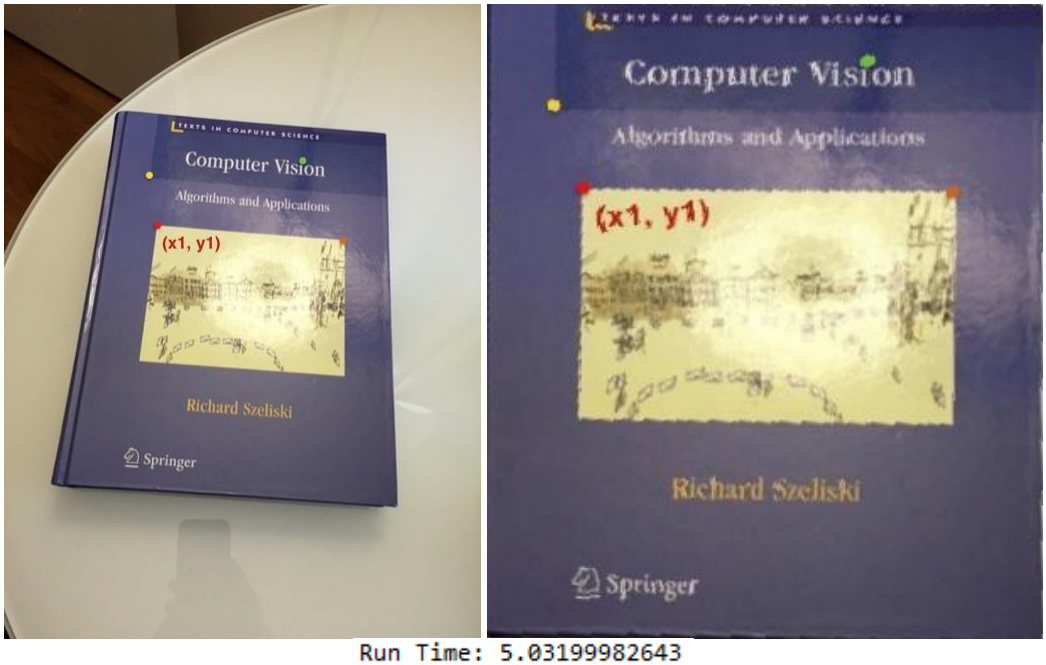


Figure 3: Original and extracted image of book, along with program run time.

Figure 3 shows an image I got off the internet that shows a book on a table. Figure 3 also shows the derived image from selecting the four corners of the book as point of interest. As stated in the previous section, I implemented this program using the pixelcoloraverage method. The resulting image looks granular, which could be a result of poorly chosen corners, or a small image, which has a smaller pixel count per area of interest, like a letter. The original image is of size 477 x 636 pixels. As is can be seen in Figure 3, the resulting image seems off-aspect. This can be easily resolved by resizing image to desired ratio.

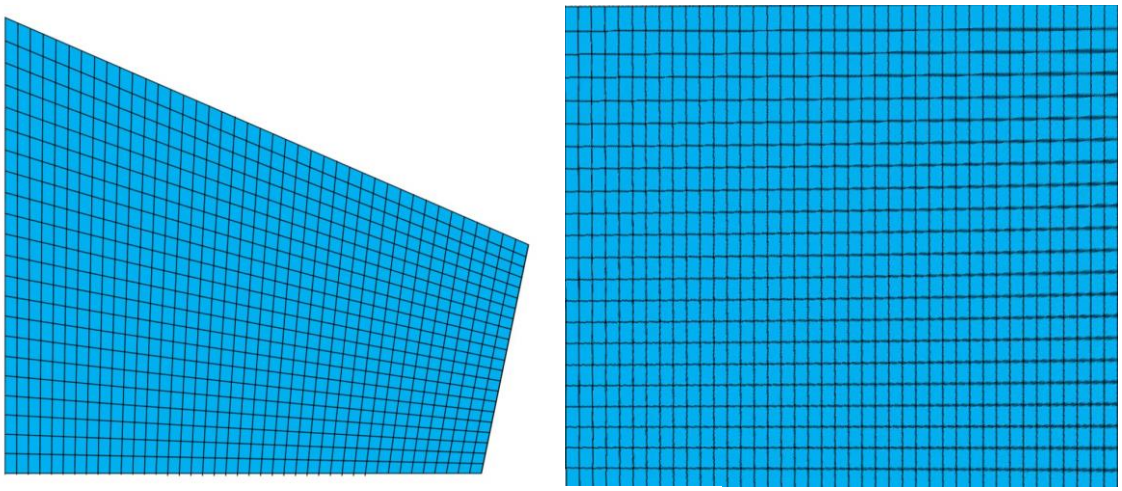


Figure 4: Original and extracted image of skewed grid, along with program run time.

Figure 4 shows an image I created of a skewed grid to test the correctness of the program. The resulting image, like the previous test, looks granular and off-aspect. The program, however, does result in a very convincing extraction. The original image is of size 1010 x 861 pixels. Like stated before, the granularity could be a result of poorly-selected corners or the original image size. The runtime of the program with the grid image was about three times slower than that of the book image.

```
Run Time: 119.638000011
```

Figure 5: Original and extracted image of Maiko lithograph, along with program run time.

Figure 5 shows a picture I took of a lithograph by Chaplan. The lithograph showcases a Maiko in kimono. I chose this image because of the minimalist approach of the print and the soft colors which allows the viewer to focus on the lines. The original picture is of size 3000 x 4000 pixels. Figure 5 also shows the extracted image of the lithograph. Although off-aspect, the lines that can be appreciated in the picture are very consistent with the lines of the extracted image. As a result, I would try to run larger images of the first two tests to see if size of original image processed has an effect on quality of image extracted. My hypothesis would be that the size allows the program to extract a more accurate average of the pixels of a particular region of interest, thus resulting in a clearer extraction.

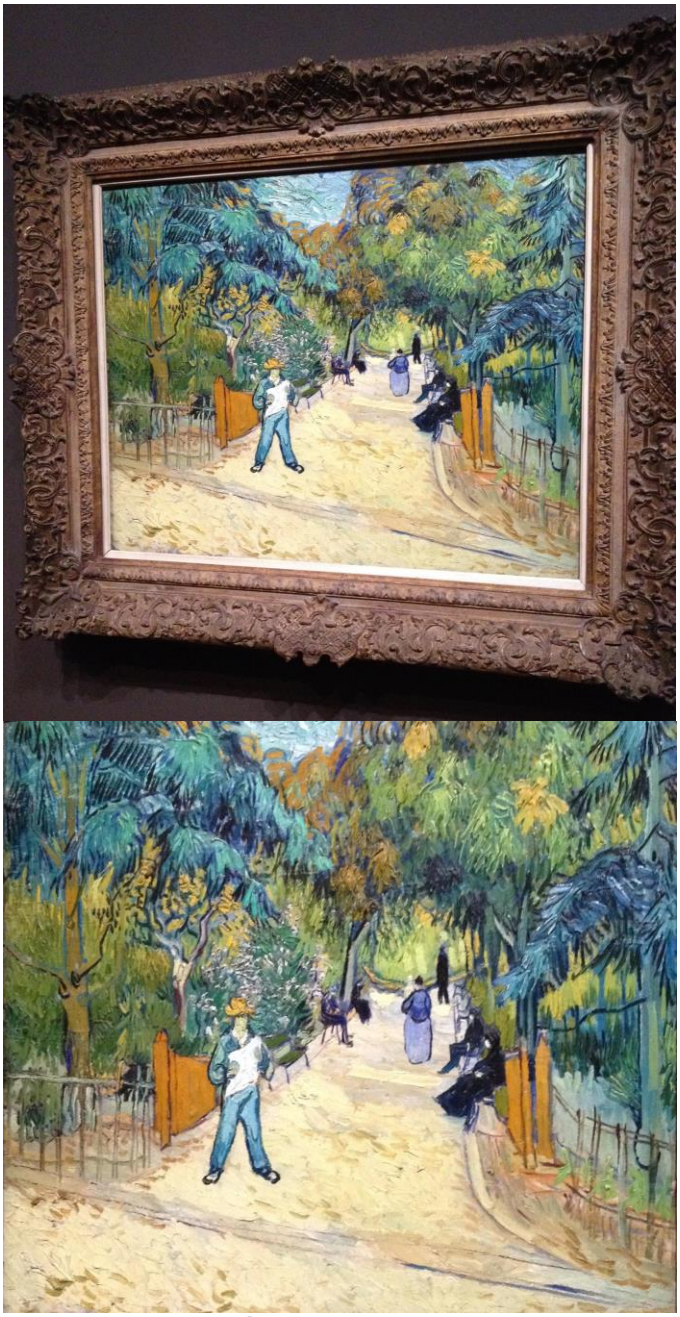


Figure 6: Original and extracted image of van Gogh painting, along with program run time.

Figure 6 shows a picture of *Entrance to the Public Garden in Arles* by Vincent van Gogh. The original picture size is 2448 x 2609 pixels. I ran the program with this original image with a number of pixels larger than two thousand per axis. Due to the complexity in colors, shapes, and proportion of extraction, this test took only three seconds less to run that the Maiko test, which ran in about 119.63 seconds.

3.3. P3

Figure 7: Destination image, source image, resized source image, and result, along with program run time.

Figure 7 shows the same image of the book I utilized to test program 2. I also retrieved the book cover of Harry Potter and the Half-Blood Prince by J.K. Rowling. I chose this image to replace the book cover due to the colors of the two books. The book cover image is of size 400 x 610 pixels. The original image is of size 477 x 636 pixels. Both images were of roughly the same size, which meant that the cover image had to be shrunk to fit inside the original image. As stated in the previous section, I utilized most of my methods from the image extraction program. This image replacement program, however, is much quicker because it does not need to extract pixel color averages like image extraction has to. The runtime of this program was 0.42 seconds, compared to 5.03 seconds of the image extraction program with the same original image and corner points.

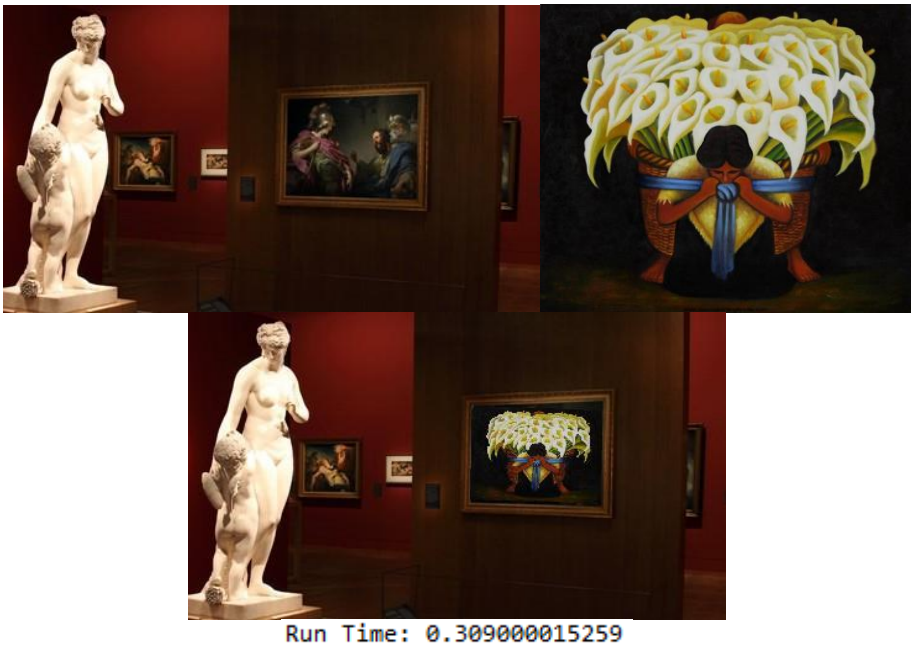


Figure 8: Destination image, source image, and result, along with program run time.

Figure 8 shows a photograph of the Louvre Museum and Diego Rivera's *Vendedor de Alcatraces*. The size of the original image is 435 x 250 pixels. The size of the replacement is 564 x 469 pixels. As expected from test shown in Figure 7, the runtime of P3 was even faster with smaller image selections. This same phenomenon can be seen in Figure 9. Figure 9 shows a picture of the Metropolitan Museum of Art in New York City. The replacement image is a Huichol painting of unknown artist. The size of the original image is 1741 x 1365 pixels. The size of the rerplacement image is 259 x 194 pixels. Although the original image size is ci=onsiderably larger than the replacement image, the region selection is still small, making the runtime very fast.

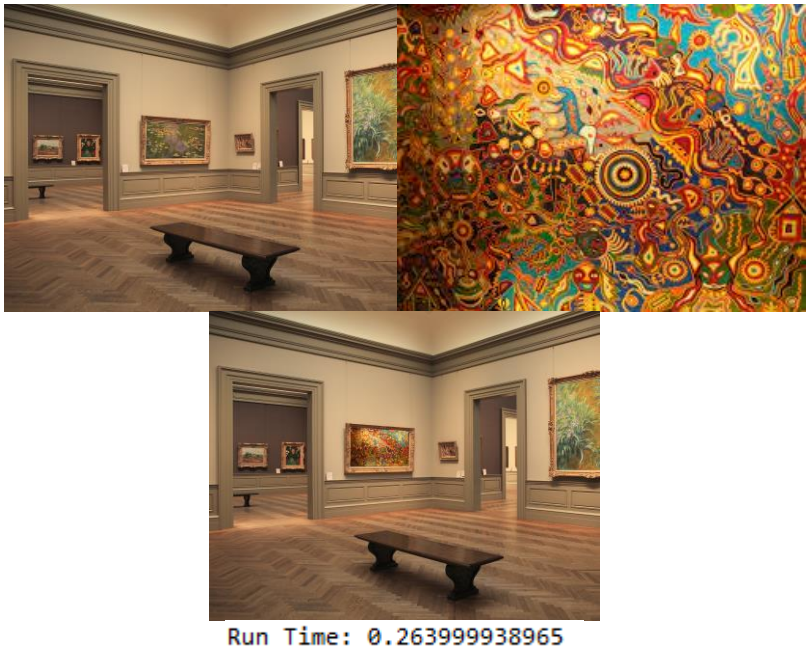


Figure 9: Destination image, source image, and result, along with program run time.

Figure 10: Destination image, source image, resized source image, and result, along with program run time.

Figure 10 shows the same image I used to test P2 in Figure 5. This time I replaced the region selected with a picture of Ziyi Zhang in kimono. The resulting image can be seen in Figure 10. Compared to the 119.63 seconds it took P2 to run with the same image and selection, P3 is faster by a factor of approximately 9.58.

3.4. P4

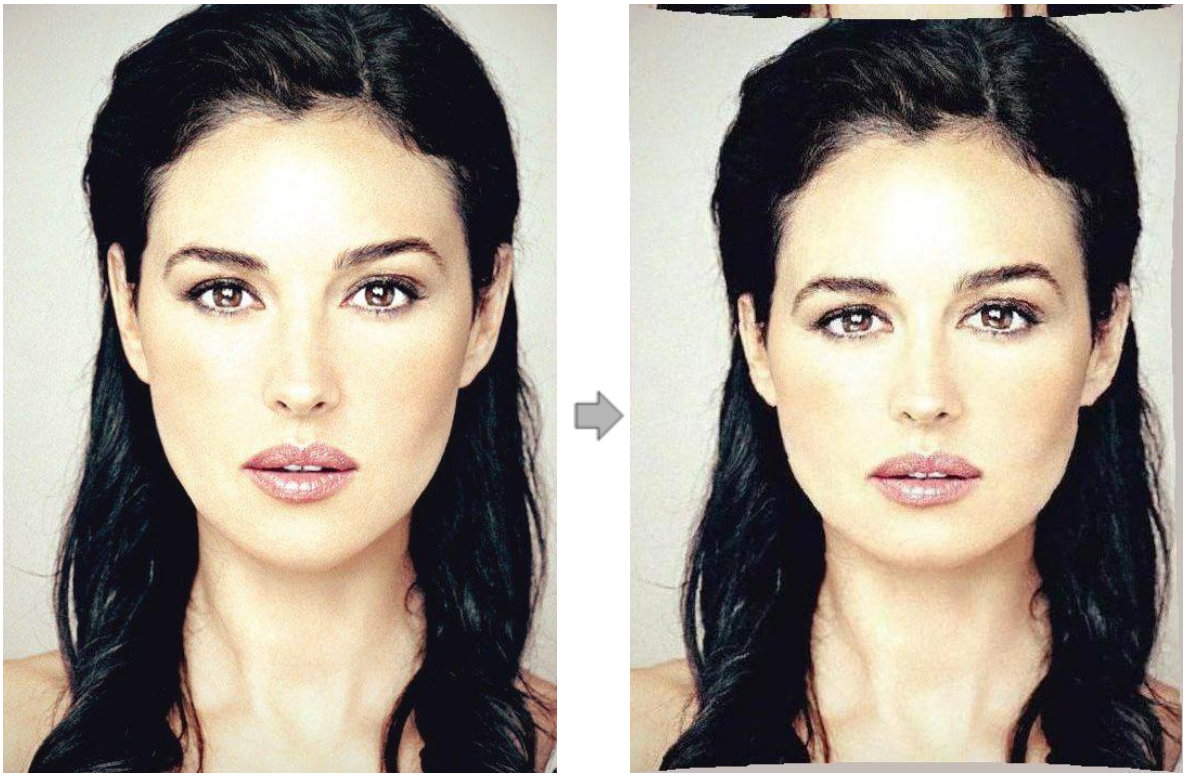
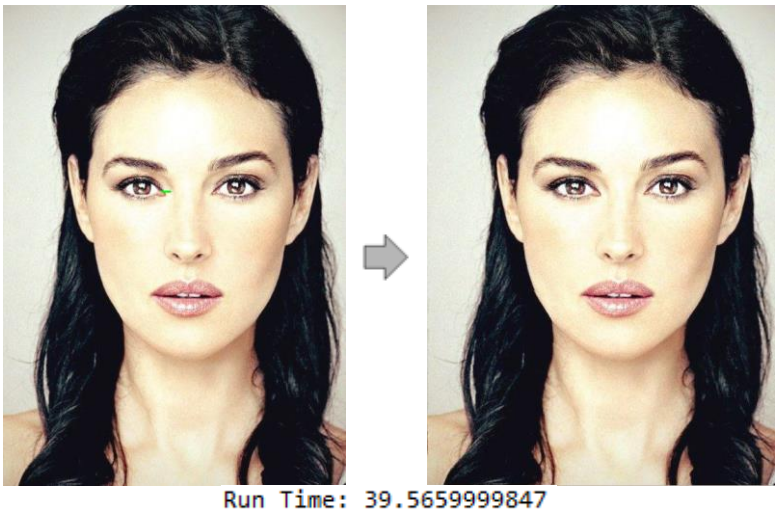
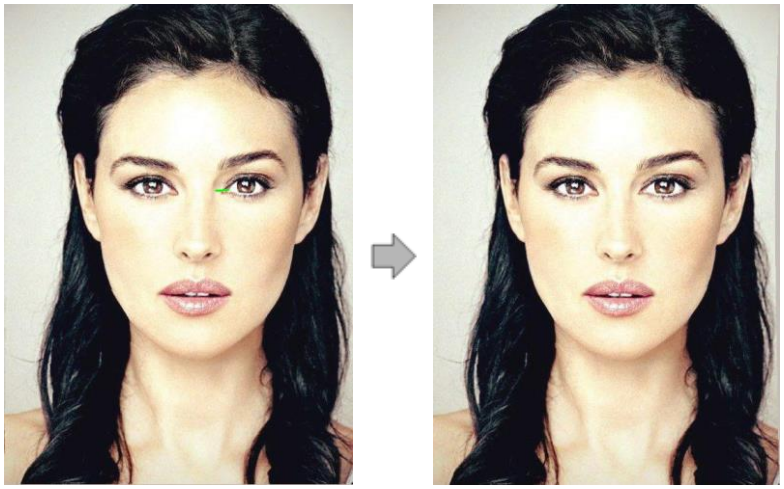


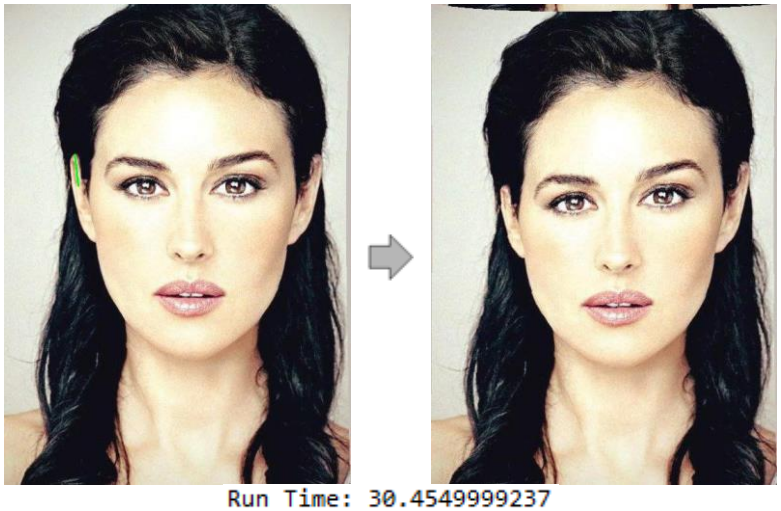
Figure 11.1: Original image and final result



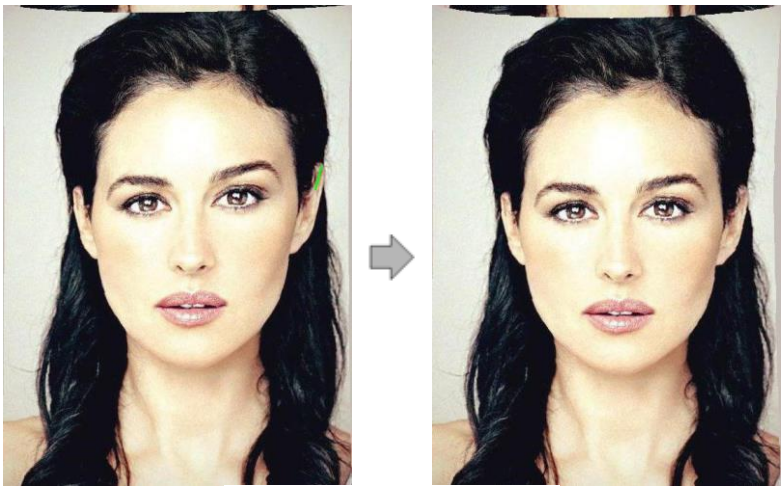
Run Time: 39.5659999847



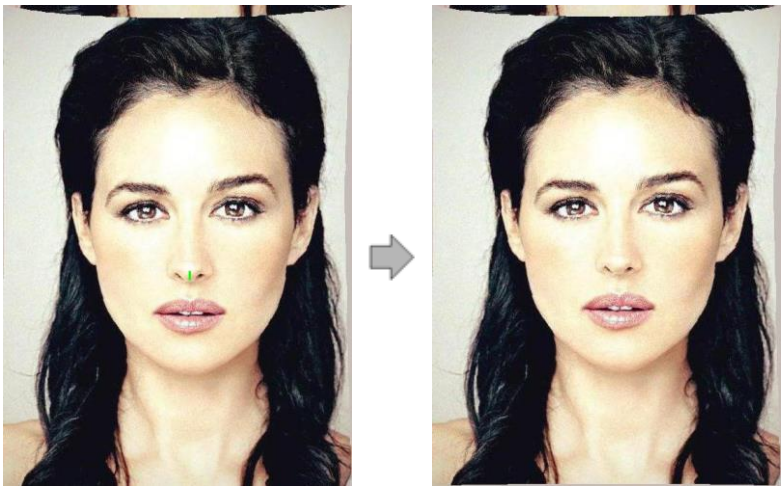
Run Time: 40.6519999504



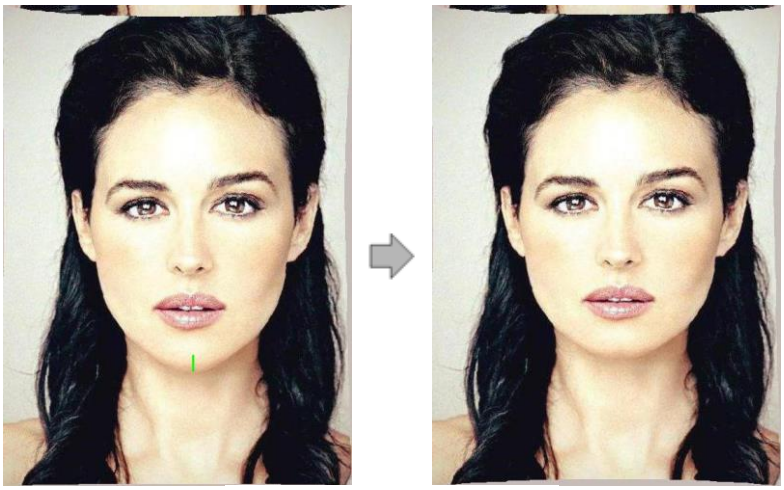
Run Time: 30.4549999237



Run Time: 30.0770001411



Run Time: 30.381000042



Run Time: 30.5450000763

Figure 11.2: From top to bottom – Image with morph selection and resulting warped image, with corresponding runtime. This figure shows a chain of continuous warpings of the same image. Figure 11.1 shows final result.

Figures 11.1 and 11.2 show warpings of an image of Monica Bellucci. I did a few warpings to adjust eye size, ear length, and nose and chin position. Figure 11.1 shows original and final image. Figure 11.2 shows the warp selection in green and the corresponding result and runtime of each warp.

3.5. P5

Figure 12.1: From top to bottom and left to right: A – Original image of Donald Trump; B – Morphed image of Donald Trump and Hillary Clinton Originals; C – Morphed image of Hillary Clinton and Donald Trump Originals; Original image of Hillary Clinton.

Figure 12.1 shows original images of Donald Trump and Hillary Clinton. Figure 12.1 also shows morphing of each original to the other original. These morphed images were obtained by a series of complementary warps of the original images with the use of points of interest (PIs). These images used twenty (20) PIs. The PIs were distributed in the eyes, nose, mouth, and contour. Figure 12.2 shows the cross dissolve of these images. The morphed images in Figure 12.1 seem very odd, but once they are cross dissolved, they work very nicely by morphing the original images to the matching PIs in the other original image.

Figure 12.2: Cross dissolve of images from Figure 12.1

4. Appendix

4.1. P1 Code

```
# -*- coding: utf-8 -*-
"""
Created on Wed Sep 28 15:51:15 2016

@author: auort

Write a function to perform real-valued indexing on an image. Your function
should receive an image I and the (real-valued) pixel coordinates < r, c > and
return the RGB values of obtained by performing a weighted average of the
pixels, where the weights are proportional to the area of overlap of the
pixels in the original image and a hypothetical pixel centered at < r, c >.
Use this function and regular rounding to evaluate the results of each of
the methods from the following questions.

"""

from PIL import Image
import scipy.misc
import cv2
```

```
import numpy as np
import time

def pixelcoloraverage(primes, image):
    #print "Primes: %s" % primes
    #image is an array with 3 channels
    cprime = primes[1]
    rprime = primes[0]
    c = int (cprime)#python always downcasts
    r = int (rprime)

    #distances
    dc2 = cprime - c
    dc1 = 1 - dc2
    dr2 = rprime - r
    dr1 = 1 - dr2

    vector1 = np.array([[dr1],[dr2]])
    vector2 = np.array([dc1,dc2])

    mask = vector1*vector2

    #print "Mask:"
    #print mask

    region = (image[c:c+2,:][:,r:r+2])

    sumred = 0
    sumgreen = 0
    sumblue = 0

    try:
        sumred = sumred + region[0][0][0] * mask[0][0]
        sumred = sumred + region[0][1][0] * mask[0][1]
        sumred = sumred + region[1][0][0] * mask[1][0]
        sumred = sumred + region[1][1][0] * mask[1][1]
        #print 'Sumred:%s' %sumred
        sumgreen = sumgreen + region[0][0][1] * mask[0][0]
        sumgreen = sumgreen + region[0][1][1] * mask[0][1]
        sumgreen = sumgreen + region[1][0][1] * mask[1][0]
        sumgreen = sumgreen + region[1][1][1] * mask[1][1]
        #print 'Sumgreen:%s' %sumgreen
        sumblue = sumblue + region[0][0][2] * mask[0][0]
        sumblue = sumblue + region[0][1][2] * mask[0][1]
        sumblue = sumblue + region[1][0][2] * mask[1][0]
        sumblue = sumblue + region[1][1][2] * mask[1][1]
        #print 'Sumblue:%s' %sumblue

    except IndexError:
        #print "Cprime: %s" % cprime
        #print "Rprime: %s" % rprime
        '''
        sumred = image[c][r][0]
        sumgreen = image[c][r][1]
        sumblue = image[c][r][2]
        '''
    return [int(sumred), int(sumgreen), int(sumblue)]

image = np.array(Image.open('C:\\Users\\auort\\Desktop\\CV_L2\\14.jpg'))
image = (image[0:4,:][:,0:4])
scipy.misc.imsave('C:\\Users\\auort\\Desktop\\CV_E6\\Selection.jpg', image)
print len(image)
print len(image[0])
```

```
image[0][0] = [0,0,0]
image[0][1] = [20,0,20]
image[0][2] = [40,0,40]
image[0][3] = [60,0,60]

image[1][0] = [0,20,20]
image[1][1] = [20,20,40]
image[1][2] = [40,20,60]
image[1][3] = [60,20,40]

image[2][0] = [0,40,40]
image[2][1] = [20,40,60]
image[2][2] = [40,40,40]
image[2][3] = [60,40,20]

image[3][0] = [0,60,60]
image[3][1] = [20,60,40]
image[3][2] = [40,60,20]
image[3][3] = [60,60,0]

scipy.misc.imsave('C:\\Users\\auort\\Desktop\\CV_L3\\ImageCreated.jpg', image)
print 'Image created:'
print image
print ''
start_time = time.time()
avg1 = pixelcoloraverage([1.25, 1.25], image)
end_time = time.time()
runtime = end_time - start_time
print 'Averages 1.25, 1.25:'
print avg1
print 'Runtime: %s' % runtime
print ' '

start_time = time.time()
avg1 = pixelcoloraverage([1.75, 1.75], image)
end_time = time.time()
runtime = end_time - start_time
print 'Averages 1.75, 1.75:'
print avg1
print 'Runtime: %s' % runtime
print ' '


start_time = time.time()
avg1 = pixelcoloraverage([1.50, 1.50], image)
end_time = time.time()
runtime = end_time - start_time
print 'Averages 1.50, 1.50:'
print avg1
print 'Runtime: %s' % runtime
print ' '

start_time = time.time()
avg1 = pixelcoloraverage([1.25, 1.75], image)
end_time = time.time()
runtime = end_time - start_time
print 'Averages 1.25, 1.75:'
print avg1
print 'Runtime: %s' % runtime
print ' '

start_time = time.time()
avg1 = pixelcoloraverage([2, 2], image)
```

```
end_time = time.time()
runtime = end_time - start_time
print 'Averages 2, 2:'
print avg1
print 'Runtime: %s' % runtime
print ' '
```

4.2. P2 Code

```
# -*- coding: utf-8 -*-
"""
Created on Sun Oct 2 15:58:58 2016

@author: auort

Write a program to extract the image of a rectangular object seen under
perspective in an image and display it without perspective effects. You can
do this using the homography-based algorithm described in the textbook, or
the line-based algorithm described in class.

"""

import numpy as np
from PIL import Image
import scipy.misc
import time
from pylab import *
from numpy import *



def getMaxWidthAndHeight(Ax, Ay, Bx, By, Cx, Cy, Dx, Dy):
    width1 = abs(Bx-Ax)
    width2 = abs(Cx-Dx)
    height1 = abs(Dy-Ay)
    height2 = abs(Cy-By)
    return int(max(width1, width2)), int(max(height1, height2))

def getValues(A, B, index, partitions):
    Ax = A[0]
    Ay = A[1]
    Bx = B[0]
    By = B[1]
    '''
    print "Ax: %d, Ay: %d, Bx: %d, By: %d " % (Ax, Ay, Bx, By)
    '''
    width = abs(Bx-Ax)
    height = abs(By-Ay)
    stepup = float(height)/partitions
    stepover = float(width)/partitions
    '''
    print "Width: %s" % width
    print "Height: %s" % height
    '''
    if(Ax > Bx):
        stepover = stepover *-1
    ''''
    print "Stepup: %s" % stepup
    print "Stepover: %s" % stepover
    print "Partitions: %s" % partitions
    '''
```

```
    xvalue = Ax + (stepover*index)
    yvalue = Ay + (stepup*index)
    return [xvalue,yvalue]


def getCoordinateArray(Ax, Ay, Bx, By, Cx, Cy, Dx, Dy):
    width, height =  getMaxWidthAndHeight(Ax, Ay, Bx, By, Cx, Cy, Dx, Dy)
    array = []
    leftcolumn = []
    rightcolumn = []
    '''
    print Ax
    print Ay
    print Dx
    print Dy
    print height
    '''
    #print "-----------------------------building columns: "
    leftcolumn = [getValues([Ax, Ay], [Dx, Dy], 0, height)]
    #print leftcolumn
    rightcolumn = [getValues([Bx, By], [Cx, Cy], 0, height)]
    #print rightcolumn
    for i in range(height-1):
        #print i
        leftcolumn.append(getValues([Ax, Ay], [Dx, Dy], i+1, height))
        rightcolumn.append(getValues([Bx, By], [Cx, Cy], i+1, height))
        i=i
    '''
    print "-----------------------------printing leftcolumn: "
    print leftcolumn
    print "-----------------------------printing rightcolumn: "
    print rightcolumn

    print "-----------------------------buiding array: "
    '''
    coordinatearray = []
    row = []
    for y in range(height):
        for x in range(width):
            value = getValues(leftcolumn[y], rightcolumn[y], x, width)
            row.append(value)
            '''
            print " "
            print "leftcolumn[y]:"
            print leftcolumn[y]
            print "rightcolumn[y]:"
            print rightcolumn[y]
            print "value:"
            print value
            '''
        coordinatearray.append(row)
        row = []

    return coordinatearray

def extractImage(coordinateArray, image):
    print "Extract image called"
    width = len(coordinateArray)
    height = len(coordinateArray[0])
    extractedImage = []
    row = []
    for y in range(height):
        for x in range(width):
```

```
            row.append(pixelcoloraverage(coordinateArray[x][y], image))
        extractedImage.insert(0, row)
        row = []

    return extractedImage

def getCorners(image):
    imshow(image)
    refPt = ginput(4)
    Ax = refPt[0][0]
    Ay = refPt[0][1]
    Bx = refPt[1][0]
    By = refPt[1][1]
    Cx = refPt[2][0]
    Cy = refPt[2][1]
    Dx = refPt[3][0]
    Dy = refPt[3][1]
    close()
    return Ax, Ay, Bx, By, Cx, Cy, Dx, Dy

def runImageExtraction(Path, Path2):
    image = np.array(Image.open(Path))
    Ax,Ay,Bx,By,Cx,Cy,Dx,Dy = getCorners(image)
    start_time = time.time()

    coordinateArray = getCoordinateArray(Ax,Ay,Bx,By,Cx,Cy,Dx,Dy)
    h = len(coordinateArray[0])
    w = len(coordinateArray)
    '''
    print "Height: %s" % h
    print "Width: %s" % w
    print "A: %d, %d" %(coordinateArray[0][0][0], coordinateArray[0][0][1])
    print "B: %d, %d" %(coordinateArray[w-1][0][0], coordinateArray[w-1][0][1])
    print "C: %d, %d" %(coordinateArray[w-1][h-1][0], coordinateArray[w-1][h-1][1])
    print "D: %d, %d" %(coordinateArray[0][h-1][0], coordinateArray[0][h-1][1])
    '''
    #print "--------------------------Coordinate Array:"
    #print coordinateArray
    imageExtracted = extractImage(coordinateArray, image)
    scipy.misc.imsave(Path2, imageExtracted)
    imageExtracted = np.rot90(imageExtracted,3)
    scipy.misc.imsave(Path2, imageExtracted)

    print "image has been saved"
    end_time = time.time()
    print "Run Time: %s" % (end_time - start_time)

runImageExtraction('maiko2.jpg','maikoextracted.jpg')
```

4.3. P3 Code

```
# -*- coding: utf-8 -*-
"""
Created on Sun Oct 16 23:05:13 2016

@author: auort

Write a program to perform the inverse process from the previous question,
that is, it should insert a rectangular region into an image with perspective
effects.
```

```
"""
from PIL import Image
import numpy as np
import scipy.misc
import time
from pylab import *
from numpy import *

def getMaxWidthAndHeight(Ax, Ay, Bx, By, Cx, Cy, Dx, Dy):
    width1 = abs(Bx-Ax)
    width2 = abs(Cx-Dx)
    height1 = abs(Dy-Ay)
    height2 = abs(Cy-By)
    return int(max(width1, width2)), int(max(height1, height2))

def getValues(A, B, index, partitions):
    Ax = A[0]
    Ay = A[1]
    Bx = B[0]
    By = B[1]
    '''
    print "Ax: %d, Ay: %d, Bx: %d, By: %d " % (Ax, Ay, Bx, By)
    '''
    width = abs(Bx-Ax)
    height = abs(By-Ay)
    stepup = float(height)/partitions
    stepover = float(width)/partitions
    '''
    print "Width: %s" % width
    print "Height: %s" % height
    '''
    if(Ax > Bx):
        stepover = stepover *-1
    ''''
    print "Stepup: %s" % stepup
    print "Stepover: %s" % stepover
    print "Partitions: %s" % partitions
    '''
    xvalue = Ax + (stepover*index)
    yvalue = Ay + (stepup*index)
    return [xvalue,yvalue]

def getCoordinateArray(Ax, Ay, Bx, By, Cx, Cy, Dx, Dy):
    width, height =  getMaxWidthAndHeight(Ax, Ay, Bx, By, Cx, Cy, Dx, Dy)
    array = []
    leftcolumn = []
    rightcolumn = []
    '''
    print Ax
    print Ay
    print Dx
    print Dy
    print height
    '''
    #print "-----------------------------building columns: "
    leftcolumn = [getValues([Ax, Ay], [Dx, Dy], 0, height)]
    #print leftcolumn
    rightcolumn = [getValues([Bx, By], [Cx, Cy], 0, height)]
    #print rightcolumn
    for i in range(height-1):
        #print i
        leftcolumn.append(getValues([Ax, Ay], [Dx, Dy], i+1, height))
```

```
        rightcolumn.append(getValues([Bx, By], [Cx, Cy], i+1, height))
        i=i
    '''
    print "-----------------------------printing leftcolumn: "
    print leftcolumn
    print "-----------------------------printing rightcolumn: "
    print rightcolumn

    print "-----------------------------buiding array: "
    '''
    coordinatearray = []
    row = []
    for y in range(height):
        for x in range(width):
            value = getValues(leftcolumn[y], rightcolumn[y], x, width)
            row.append(value)
            '''
            print " "
            print "leftcolumn[y]:"
            print leftcolumn[y]
            print "rightcolumn[y]:"
            print rightcolumn[y]
            print "value:"
            print value
            '''
        coordinatearray.append(row)
        row = []

    return coordinatearray

def replaceImage(coordinateArray, sourceimage, image):
    height = len(coordinateArray[0])
    width = len(coordinateArray)
    for y in range(height):
        for x in range(width):
            '''
            print "x: %d, y: %d" % (x,y)
            print "width: %d, height: %d" % (width,height)
            '''
            xcoor = int(coordinateArray[x][y][0])
            ycoor = int(coordinateArray[x][y][1])
            try:
                image[ycoor][xcoor] = sourceimage[x][y]
            except IndexError:
                print "Index Error caught"
    return image

def resizeimagetoarray(image, width, length):
    image = Image.open(image)
    image = image.resize((width,length))
    return np.array(image)

def getCorners(image):
    imshow(image)
    refPt = ginput(4)
    Ax = refPt[0][0]
    Ay = refPt[0][1]
    Bx = refPt[1][0]
    By = refPt[1][1]
    Cx = refPt[2][0]
    Cy = refPt[2][1]
    Dx = refPt[3][0]
    Dy = refPt[3][1]
```

```
    close()
    return Ax, Ay, Bx, By, Cx, Cy, Dx, Dy

def runImageReplace(Source, Destination):
    image = np.array(Image.open(Source))
    dest = np.array(Image.open(Destination))
    Ax,Ay,Bx,By,Cx,Cy,Dx,Dy = getCorners(dest)
    start_time = time.time()
    print "Getting Coordinate Array"
    coordinateArray = getCoordinateArray(Ax,Ay,Bx,By,Cx,Cy,Dx,Dy)
    width, length = getMaxWidthAndHeight(Ax,Ay,Bx,By,Cx,Cy,Dx,Dy)
    print "Resizing source image"
    sourceimage = resizeimagetoarray(Source, width, length)
    scipy.misc.imsave('resizedsource.jpg', sourceimage)
    print "replacing image"
    imageReplaced = replaceImage(coordinateArray, sourceimage, dest)
    print type(imageReplaced)
    #imageReplaced.save('replacedcover.jpg')
    scipy.misc.imsave('replacedcover2.jpg', imageReplaced)


    print "image has been saved"
    end_time = time.time()
    print "Run Time: %s" % (end_time - start_time)

runImageReplace('geisha.jpg','maiko2.jpg')
```

4.4. P4 Code

```
# -*- coding: utf-8 -*-
"""
Created on Sat Oct 8 12:34:18 2016

@author: auort

The image point selection part of this program was derived from Adrian
Rosebrock's Capturing mouse click events with Python and OpenCV, available at
http://www.pyimagesearch.com/2015/03/09/capturing-mouse-click-events-with-
python-and-opencv/

Write a program to implement k-nearest neighbor warping, as explained in class.
Your program should allow the user to input source and destination points and
generate a sequence of images illustrating a smooth transition from the source
to the destination image.

"""
import cv2
import numpy as np
import math
import pylab
import copy
import time

# initialize the list of reference points and boolean indicating
# whether selecting is being performed or not
refPt = []
selecting = False

def click_and_save(event, x, y, flags, param):
       # grab references to the global variables
```

```
        global refPt, selecting
        # if the left mouse button was clicked, record the starting
        # (x, y) coordinates and indicate that cropping is being
        # performed
        if event == cv2.EVENT_LBUTTONDOWN:
          refPt = [(x, y)]
          selecting = True

        # check to see if the left mouse button was released
        elif event == cv2.EVENT_LBUTTONUP:
          refPt.append((x, y))

          selecting = False
          # draw a line between two points
          display = cv2.imread('warpedimage.jpg')
          original = cv2.imread('warpedimage.jpg')
          #warp(display, refPt)
          cv2.line(display, refPt[0], refPt[1], (0, 255, 0), 2)
          cv2.imshow("image selection", display)
          cv2.imwrite('warpedimageselection.jpg', display)
          warp(refPt)
          display = cv2.imread('warpedimage.jpg')
          cv2.imshow("warped image", display)

def warp(refPt):
    #print "warping image"
    start_time = time.time()
    source = []
    destination = []
    im = array(Image.open('warpedimage.jpg'))
    newim = zeros(im.shape)

    #corner points
    source.append([0,0])
    source.append([0,len(im[0])])
    source.append([len(im),0])
    source.append([len(im),len(im[0])])
    destination.append([0,0])
    destination.append([0,len(im[0])])
    destination.append([len(im),0])
    destination.append([len(im),len(im[0])])

    inX = refPt[0][1]
    inY = refPt[0][0]
    source.append([inX,inY])
    inX2 = refPt[1][1]
    inY2 = refPt[1][0]
    destination.append([inX2,inY2])

    disp = subtract(source, destination)
    weight = []

    for r in range(len(im)):
       for c in range(len(im[0])):
           distance = zeros(len(destination))
           for dest in range(len(destination)):
                  distance[dest]    =    (sqrt(pow(r-destination[dest][0],2)    +    pow(c-
destination[dest][1],2)))
                  weight = 1/(distance + 0.0000001)
                  weight = weight/sum(weight)
                  newx = r + dot(weight, disp)[0]
                  newy = c +dot(weight, disp)[1]
                  if (newx < len(im)-1 and newy < len(im[0])-1):
```

```
                        newim[r][c] = im[int(newx)][int(newy)]
                    else:
                        newim[r][c] = im[0][0]
    scipy.misc.imsave('warpedimage.jpg', newim)
    end_time = time.time()
    print "Run Time: %s" % (end_time - start_time)

def runSelectiveWarping(Path):
    image = cv2.imread(Path)
    cv2.imwrite('warpedimage.jpg', image)
    cv2.namedWindow("image")
    cv2.setMouseCallback("image", click_and_save)

    # keep looping until the 'c' key is pressed
    while True:
        # display the image and wait for a keypress
        cv2.imshow("image", image)
        key = cv2.waitKey(1) & 0xFF
        # if the 'c' key is pressed, break from the loop
        if key == ord("c"):
            break
        elif key == ord("n"):
            runSelectiveWarping('warpedimage.jpg')
    # close all open windows
    cv2.destroyAllWindows()

Path = 'monica1.jpg'
runSelectiveWarping(Path)
```

4.5. P5 Code

```
# -*- coding: utf-8 -*-
"""
Created on Sun Oct 16 17:00:38 2016

@author: auort

The cross dissolve method is a modification from stack overflows's cv2 Python
image blending "Fade" transition accessed at
http://stackoverflow.com/questions/28650721/cv2-python-image-blending-fade-transition

Write a program to morph a face image in frontal view into another. Use a
variation of your program from the previous question to align the faces,
then apply cross-dissolve.

"""

from PIL import Image
from pylab import *
from numpy import *
import scipy.misc
import cv2
from shutil import copyfile
import time


def warpimage(im, refPt, name):
    newim = zeros(im.shape)
    source = []
    destination = []
    #corner points
```

```
    source.append([0,0])
    source.append([0,len(im[0])])
    source.append([len(im),0])
    source.append([len(im),len(im[0])])

    destination.append([0,0])
    destination.append([0,len(im[0])])
    destination.append([len(im),0])
    destination.append([len(im),len(im[0])])

    inX = refPt[0][1]
    inY = refPt[0][0]
    source.append([inX,inY])
    inX2 = refPt[0][3]
    inY2 = refPt[0][2]
    destination.append([inX2,inY2])

    disp = subtract(source, destination)
    weight = []

    for r in range(len(im)):
       for c in range(len(im[0])):
           distance = zeros(len(destination))
           for dest in range(len(destination)):
                  distance[dest]    =    (sqrt(pow(r-destination[dest][0],2)    +    pow(c-
destination[dest][1],2)))
                  weight = 1/(distance + 0.0000001)
                  weight = weight/sum(weight)
                  newx = r + dot(weight, disp)[0]
                  newy = c +dot(weight, disp)[1]
                  if (newx < len(im)-1 and newy < len(im[0])-1):
                      newim[int(r)][int(c)] = im[int(newx)][int(newy)]
                  else:
                      newim[r][c] = im[0][0]
    scipy.misc.imsave('warpedimage.jpg', newim)
    scipy.misc.imsave(name, newim)

    #imshow(newim)

def morphImages():
    imname1 = 'morphim1.jpg'
    imname2 = 'morphim2.jpg'

    img1 = Image.open(imname1)
    img2 = Image.open(imname2)
    img2 = img2.resize(img1.size, Image.ANTIALIAS)
    #img2 = resizeimage.resize_thumbnail(img2, img1.size)
    imagewidth = img1.size[0]

    im1 = array(img1)
    im2 = array(img2)

    sbsimages = np.concatenate((im1, im2), axis=1)
    imshow(sbsimages)

    #imshow(im)
    refPt = ginput(2)
    refPt1 = []
    refPt2 = []

    xim1 = refPt[0][0]
    yim1 = refPt[0][1]
    xim2 = refPt[1][0]
```

```
    yim2 = refPt[1][1]


    if (xim1 > imagewidth):
        xim1 = xim1 - imagewidth
    else:
        xim2 = xim2 - imagewidth
    refPt1.append(([xim1,yim1,xim2,yim2]))
    refPt2.append(([xim2,yim2,xim1,yim1]))

    print "morph images called"
    start_time = time.time()
    print "morphing image 1"
    warpimage(im1, refPt1, imname1)
    print "morphing image 2"
    warpimage(im2, refPt2, imname2)
    print "morphing complete"
    end_time = time.time()
    run_time = end_time - start_time
    print run_time

def crossDissolve (im1, im2):
    img1 = array(Image.open(im1).convert('L'))
    img2 = array(Image.open(im2).convert('L'))
    scipy.misc.imsave('crossdisolveCheck1.jpg', img1)
    scipy.misc.imsave('crossdisolveCheck2.jpg', img2)
    #while True:
    for IN in range(0,10):
            fadein = IN/10.0
            dst = cv2.addWeighted( img1, 1-fadein, img2, fadein, 0)#linear $
            cv2.imwrite('crossdisolve'+str(IN)+'.jpg', dst)
            cv2.imshow('window', dst)
            cv2.waitKey(1)
            print fadein
            time.sleep(0.05)
            if fadein == 1.0: #blendmode mover
                    fadein = 1.0
    return # exit function

def runMorphing(name1, name2):

    copyfile(name1, 'morphim1.jpg')
    copyfile(name2, 'morphim2.jpg')
    blank = cv2.imread('morphBlank.jpg')
    instructions = cv2.imread('morphInstructions.jpg')

    #morphImages()
    morphImages()


    # keep looping until the 'c' key is pressed
    while True:
        # display the image and wait for a keypress
        cv2.imshow("instructions", instructions)
        key = cv2.waitKey(1) & 0xFF

        # if the 'c' key is pressed, break from the loop
        if key == ord("c"):
            break
        elif key == ord("m"):
            cv2.imshow("instructions", blank)
            morphImages()
```

```
    # close all open windows
    cv2.destroyAllWindows()

runMorphing('donaldsmall.jpg', 'hillarysmall.jpg')
crossDissolve ('hillarysmall.jpg','morphim1.jpg')
crossDissolve ('morphim1.jpg','donaldsmall.jpg')
crossDissolve ('donaldsmall.jpg','morphim2.jpg')
crossDissolve ('morphim2.jpg','hillarysmall.jpg')
```

End of Document

End of Document