Computational Photography

Final Project

Thomas Melanson

CS6475 - Fall 2019

The Ultimate Thanksgiving Video: Interpolating the Whole Family

Using image metamorphosis to create a face interpolation application.

The Goal of Your Project

Original Project Scope:

The goal of this project was to generate automatic and seamless transition between faces, without needing to use hand drawn facial lines. The transition would be seamless in that each stage in the transformation could be a passable face. The automatic part is that each stage of the process, from finding the face to drawing the lines to creating the animation, would be done using computer vision / computational photography and not a person.

What motivated you to do this project?

I found the facial metamorphosis shown in the lecture slides to be fascinating because it created an animation between two disparate faces/images. I wanted to see if I could replicate that animation and apply this to images in the wild.

Scope Changes

- Did you run into issues that required you to change project scope from your proposal?

yes

- Give a detailed explanation of what changed

Originally, I was only supposed to compute the reprojections in the paper, and use some facial detection from OpenCV to create the lines. However, I also improved upon the indexing scheme from the paper by experimenting with padding and implementing bilinear interpolation, as well as extending the application to animating my face in different expressions

Showcase

Input
Left to Right: Input Source and Destination Images

Output
Left to Right: Source, Output, and Destination Sequences

Project Pipeline

Input: Two images, Source and Destination

For the sake of explaining the pipeline, only two images are taken as input. However, the main function does take in multiple images, looping over consecutive pairs

Step 1: Crop out the Face (Pre-Processing)

First, to instill some consistency between images and to not waste time warping the background, the images are cropped so only the faces are featured. The bounding box of the face was found using a Haar cascade classifier from OpenCV. After cropping the image to the bounding box, the image was rescaled to 200x200 (the original images were 1024p phone images, so 200 by 200 was approximately the size of the facial region)

Step 2: Generate Facial Landmarks / Lines

Once the faces were isolated, the lines required for the image metamorphosis were drawn on them. This was done by using the Facial Landmarks detector to generate 68 facial points, corresponding to the chin, eyebrows, eyes, nose, and mouth. Since these points always corresponded to the same points on the face, I could split the 68-vector into facial regions, and then convert the points into lines as (point1, point2), (point2, point3), etc.

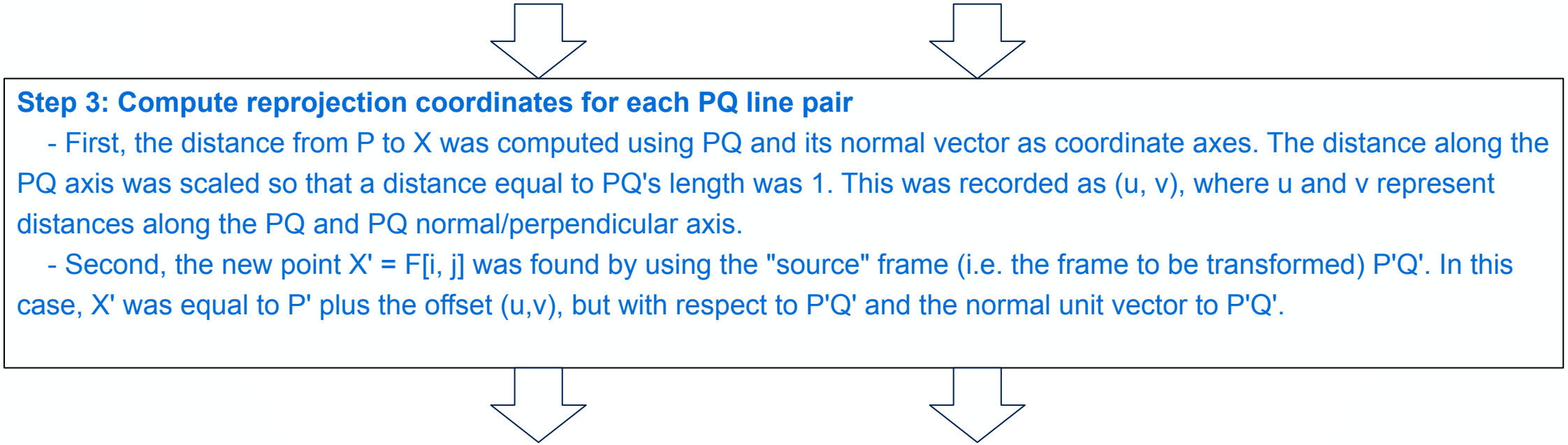
Project Pipeline (cont'd)

For each transition between source and destination image, I computed two discrete functions:

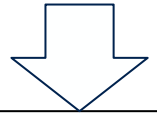
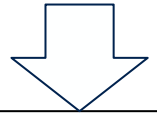
a. F[i_dst, j_dst] -> X_src

b. F[i_src, j_src] -> X_dst

Respectively, these functions compute the transformation of the source image onto the destination image reference frame and the transformation of the destination image onto the source image reference frame. This was done with the pseudo-code from the original paper, which consisted of doing two instances of Steps 3 and 4, one from src to dst and one from dst to src:



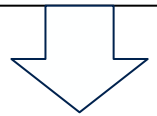
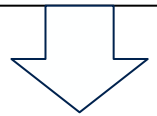
Project Pipeline (cont'd)

Step 4: Take the weighted average from each PQ line pair to find the weighted reprojected coordinates

- For each line segment (PQ, P'Q') in set:
- Compute X' using step 3
- Calculate displacement D from X to X'
- Calculate distance dist as the shortest distance from X to PQ
- Calculate weight
- Add D*weight to running sum DSUM
- Add weight to running weight sum weight_sum
- After for loop, compute X' as X + DSUM / weightsum

In this instance, having a prime (') means that the variable corresponds to the source image. PQ is the line segment for the destination image, and P'Q' is the line corresponding to the source image.

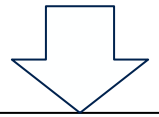
Project Pipeline (cont'd)

Step 5: Interpolate and Warp

The output is then split into n + 1 frames, where n is the number of transition frames specified by the user. For each value between 0 and n:

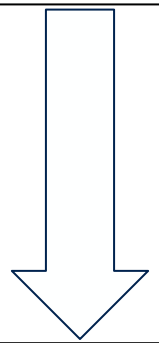
a. The blending alpha is determined by (i/(n+1))
b. For both src -> dst and dst -> src:
    i. Displacements are found as D = F[i, j] - X
    ii. For each pixel in the frame, the corresponding coordinate from the source and destination are found using the blending alpha multiplied by the DSUM/weightsum values found in step 3.
    iii. Since the coordinate values were usually decimal values, the pixel value of the coordinate was found using bilinear interpolation of the neighboring pixels.
c. The two images were then combined using (1-alpha)*(src->dst) + alpha*reverse(dst->src)
d. Because the black region beyond the edges were often visible in the transformation, any pixels where only one image had a pixel value would just be set to the nonzero pixel value.
e. Write the resulting mixed frame to the output directory

Project Pipeline (cont'd)



Step 6: Consolidate Output into Video (Post-processing)

Once these mixed frames were computed, FFMPEG was used to consolidate all n images into a single GIF. The exact command was taken from our Assignment 6 README.



Output: GIF video

Demonstration: Result Sets

Input: Images taken from a Samsung phone of my relatives behind a beige wall. No processing was done before the beginning of the pipeline. These images will be referred to, from left to right, as 1, 2, 3, and 4.

Demonstration: Result Sets (cont'd)

Output: Four sets of transition images, which consist of (a) the source image at each stage of transformation, (b) the destination image warped to each frame of the transformation, and (c). the combination of the source and destination image of each stage in the transformation. Each picture represents a frame mid-transformation. (a) is on the left side, (b) is on the right, and (c) is in the middle. This slide shows the transition from 1 to 2 and from 2 to 3.

Demonstration: Result Sets (cont'd)

Output: Four sets of transition images, which consist of (a) the source image at each stage of transformation, (b) the destination image warped to each frame of the transformation, and (c). the combination of the source and destination image of each stage in the transformation. (a) is on the left side, (b) is on the right, and (c) is in the middle. This slide shows the transition from 3 to 4 and from 4 back to 1.

Demonstration: Result Sets (cont'd)

GIF results are uploaded here:
https://drive.google.com/open?id=1o4x2ol_5228aUnM_3hqIMswA7cAISs4M

For reference, mom is image 1, dad image 2, granddad image 3, and grandmom image 4.

Final output: https://drive.google.com/open?id=1o2tHhiaxwq9vIR2Bd-T6F2veq7-l08um

output.gif rotates through images 1 through 4 and back.

Cropped output: https://drive.google.com/open?id=1HpY5gkM5E_4P51FdKtYCDAJtvgKkzjjX

In order to emphasize the part of the face that is interpolated, I cropped the sides of the GIF to remove the hair and clothing (although this was an earlier version before edge-padding replace zero-padding).

Project Development

- I first began by writing the main algorithm as described in the paper. Because I didn't yet have the infrastructure to test on real images, I used a 5x5 pixel binary F similar to the one shown in the paper for testing. Because the paper had a separate section dedicated to computing u,v in the single line case, I started there.
- Because I knew the warping would be a non-trivial process, even though it was a single line in the pseudocode, I decided to separate this function from the rest of the warping function. This became useful when I had to compute several warps of the image to create interpolation frames in the animation.
- Before I got my family to pose for a series of pictures, I initially chose some stock photos that I found online for integration testing. Although this proved simple because I didn't have to crop the images and I could choose faces making similar expressions, the amount of enhancements made on a stock photo (makeup, contrast, lighting, etc) made the interpolation process more difficult. In fact, the homemade images were much easier to interpolate than the stock photos once they were cropped.
- After passing my tests on the toy case, I moved on a single warp from one stock photo to another. For the first stock photo transformation, I decided to use just three lines: two for the width of each eye, and one for the approximate line of the mouth. This made the result of the warping clear: the eyes and mouth of the source image would be warped to to match the location of the eyes and mouth of the destination image. Starting with a simple model of 3 lines allowed me to identify a logic error (not taking into account relative location, as well as x-y and i-j confusion) that might not have been apparent with a more complicated set of lines.

Project Development (cont’d)

- Even when I was moving from exact tests to more visual sanity checks for the completed image, I made sure to write the tests before implementing them (i.e. test-driven development). This was to make the code development more intuitive and to minimize code re-writing and copy-paste towards the end of the project.
- In hindsight, I realize that I compute X' from the total displacement for the metamorphosis only to use it to extract the displacement back again so I can create a partial warping over n frames. In my opinion, I'm glad I did this, because it was more intuitive to test the X' computation in the early stages, and a numpy vectorized subtraction and addition don't take much time relative to the other processes.
- If there was one thing I wish I could improve, it would be to vectorize the indexing using in the warp function. Because numpy arrays cannot be directly used to index into an image, I had to loop over every pixel value for indexing. In Python, this is an extremely inefficient way of computing the image warp, as it doesn't rely on Numpy or OpenCV's built-in vectorization process.
- Because directly converting the warped coordinate values to integers resulted in a glitchy transformed value, I decided to use bilinear interpolation to smooth the image. Although this made the longest stage in the process considerably lengthier, I am happy with the results overall.
- In Assignment 6, the code created an output tree of directories and stored each corresponding frame in the directory. The instructions then gave us an ffmpeg library to use for writing the output directory to a GIF. I decided to use this format for the project output, which was useful when I wanted to retrieve specific images in between interpolation for the report.

Computation: Code Functional Description

- The top function shows the vectorized u,v computation stage of the project pipeline. It initializes X as a "coordinate grid", i.e. a matrix where every 2-D entry (i,j) contains the array [i,j]. It then uses the numpy dot product to determine the distance of PX both along PQ and perpendicular to PQ by using the PQ vector and normal vector (PQ_perp) respectively.
- The bottom function, find_reproj_coord, finds the coordinate grid X_prime for a single corresponding line pair, which for every entry (i,j) has the 2D vector for the source image (defined by (P_src, Q_src)). For each entry in the destination matrix, it computes (u,v) using the function above, then adds the (u,v) values (scaled by the PQ vector and the unit vector of the PQ perpendicular line, respectively) to the initial point of the line in the source frame (P_src).
- Both of these functions are entirely vectorized, meaning that it computes the reprojection coordinates for every index in the output matrix at once. This is done by taking advantage of numpy's broadcasting functionality (i.e. extending one value to an entire row or matrix).

```
'''
Given a the resulting image shape, as well as the line defined by initial point P and end point Q, return u, which
is the distance of X along PQ with respect to the length of PQ; and v, which is the distance of X from the line defined
by PQ.
'''
def compute_uv(img_shape, P, Q):
    X = createCoordinateGrid(img_shape)
    PQ_vector, PQ_perp, PQ_norm = computeAxesAndNorm(P, Q)
    PX_vector = X - P
    return np.dot(PX_vector, PQ_vector)/np.square(PQ_norm), \
           np.dot(PX_vector, PQ_perp)/PQ_norm



'''
Finds the coordinates used for projecting the source image into the destination reference frame.
param[in] dst_shape The size of the output projection coordinate matrix
param[in] P_dst, Q_dst The initial and end points of the facial line in the destination image.
param[in] P_src, Q_src The initial and end points of the corresponding facial line in the source image.
return A 3D-matrix, that, at each 2D index (i1, j1), specifies an (i2,j2) coordinate such that
output[i1, j1] = source[i2, j2].
'''
def find_reproj_coord(dst_shape, P_dst, Q_dst, P_src, Q_src):
    u, v = compute_uv(dst_shape, P_dst, Q_dst)
    u = np.atleast_3d(u)
    v = np.atleast_3d(v)
    P_src_broadcast = np.broadcast_to(np.array(P_src), (dst_shape[0], dst_shape[1], 2))
    PQ_src_vector, PQ_src_perp, PQ_src_norm = computeAxesAndNorm(P_src, Q_src)
    PQ_src_vector_broadcast = np.broadcast_to(np.array(PQ_src_vector), (dst_shape[0], dst_shape[1], 2))
    PQ_src_perp_broadcast = np.broadcast_to(np.array(PQ_src_perp), (dst_shape[0], dst_shape[1], 2))

    # Xp = Pp + u*(Qp-Pp) + v*Perp(Qp-Pp)/norm(Qp-Pp)
    X_prime = P_src_broadcast + u*PQ_src_vector_broadcast + v*PQ_src_perp_broadcast/PQ_src_norm
    return X_prime
```

Computation: Code Functional Description (cont'd)

```
'''
Computes the weighted average of the coordinate offset from two sets of (P,Q) pairs, then uses it to create the
reference coordinate matrix.
param[in] dst_shape The shape of the resulting output.
param[in] face1 The set of (P,Q) pairs corresponding to the first face.
param[in] face2 The set of (P,Q) pairs corresponding to the second face.
return[in] The reference coordinate matrix describing what source image pixels correspond to each destination image location.
'''
def compute_weighted_source_coordinates(dst_shape, face1, face2):
    D_sum = np.zeros(dst_shape + (2,))
    weight_sum = np.zeros(dst_shape)
    X = createCoordinateGrid(dst_shape)
    for feat1, feat2 in zip(face1, face2):
        P_src = np.array(feat1[:2])
        Q_src = np.array(feat1[2:])
        P_dst = np.array(feat2[:2])
        Q_dst = np.array(feat2[2:])

        length = np.linalg.norm(Q_dst - P_dst)
        u, v = compute_uv(dst_shape, P_dst, Q_dst)
        dist = computeDist(length, u, v)
        weight = computeWeight(length, dist, a=0.0001, p=1, b=2)
        X_src = find_reproj_coord(dst_shape, P_dst, Q_dst, P_src, Q_src)

        D = X_src.astype(np.float) - X.astype(np.float)
        D_sum += np.atleast_3d(weight) * D
        weight_sum += weight
    return X + D_sum / np.atleast_3d(weight_sum)
```

- compute_weighted_source_coordinates extends the functions in the previous slide by computing the weighted sum of the X_prime grids found by each line in a face (represented as a list of lines). For each line, three values are computed: length, which is the length of the destination PQ vector, dist is the distance each point is from the PQ line segment, and D is the offset from X (destination point) to X_prime (source point).
- A weight is computed in a helper function as weight = (length^p / (a + dist))^b , where p, a, and b are tunable variables set to p=1, a=0.0001, and b=2. p=1 means length is taken into account, a is close to 0 to create ideal rather than smoothed warping, and b=2 to minimize the impact of the features on each other, as well as on faraway facial features.

Computation: Code Functional Description (cont'd)

- This function is how the warping was done, given the source image and the mapping from source image coordinates to destination coordinates. Two things had to be accounted for when creating this image:
  - The source coordinates were not necessarily within the bounds of the original source image. To account for this, the source image was padded so that every coordinate would fall within the bounds of the image. Because there was padding to the left and above the image, thereby affecting the reference frame, the coordinates were also translated to this new reference frame. I found that padding with edge values provided a less jarring blend around the image edges
  - The source coordinates were also not integers, although the source image only had pixel values for integer coordinates. The initial solution was to use the floor of the coordinate values (i.e. round each value down). To make the resulting warp smoother, I employed bilinear interpolation. Essentially, this is a weighted sum of the four nearest pixels based on how close they are to the coordinate.

```
'''
Given a source image and a set of coordinates to be used to populate the destination image, populate the destination
image with the warped source image.
param[in] pixel_est_method How to extimate a pixel value when the exact value doesn't exist. Can be 'floor', where the
indices are truncated to be integer indices, or 'bilinear', where the pixel is computed as a blend of the four
neighboring indices.
param[in] pad How to pad the image. Right now supports 'zero' (pad with zeros), but could also pad with edge values.
return Source image warped to the destination reference frame.
'''
def warp_source(src, dst_coord, pixel_est_method='floor', pad='zero'):
```

```
if pad == 'zero' or pad == 'edge':
    pad_left = abs(int(np.minimum(0, np.floor(np.min(dst_coord[:,:,0])))))
    pad_right = int(np.maximum(0, np.max(dst_coord[:,:,0] - src.shape[0]+2)))
    pad_up = abs(int(np.minimum(0, np.floor(np.min(dst_coord[:,:,1])))))
    pad_down = int(np.maximum(0, np.max(dst_coord[:,:,1] - src.shape[1]+2)))
    # Set the value so it can be used with np pad
    if pad == 'zero':
        pad = 'constant'
    src = np.pad(src, ((pad_left, pad_right), (pad_up, pad_down), (0,0)), mode=pad)
    dst_coord[:,:,0] += pad_left
    dst_coord[:,:,1] += pad_up
```

```
if pixel_est_method == 'floor':
    dst_coord_est = np.floor(dst_coord).astype(np.uint16)
    for i in range(dst.shape[0]):
        for j in range(dst.shape[1]):
            try:
                dst[i, j] = src[tuple(dst_coord_est[i, j])]
            except IndexError as e:
                print("At index [{}, {}]".format(i, j))
                raise e
elif pixel_est_method == 'bilinear':
    dst_coord_topleft = np.floor(dst_coord).astype(np.uint16)
    dst_coord_bottomright = np.ceil(dst_coord).astype(np.uint16)
    dst_coord_topright = np.dstack([dst_coord_topleft[:, :, 0], dst_coord_bottomright[:, :, 1]])
    dst_coord_bottomleft = np.dstack([dst_coord_bottomright[:, :, 0], dst_coord_topleft[:, :, 1]])
    blend_alpha = dst_coord - dst_coord_topleft
    for i in range(dst.shape[0]):
        for j in range(dst.shape[1]):
            try:
                Q11 = src[tuple(dst_coord_topleft[i, j])]
                Q12 = src[tuple(dst_coord_topright[i, j])]
                Q21 = src[tuple(dst_coord_bottomleft[i, j])]
                Q22 = src[tuple(dst_coord_bottomright[i, j])]
                alphax = blend_alpha[i, j, 0]
                alphay = blend_alpha[i, j, 1]
                pixel = (1-alphay)*(1-alphax)*Q11 + (1-alphay)*alphax*Q21 + \
                        (alphay)*(1-alphax)*Q12 + alphay*alphax*Q22
                dst[i, j] = pixel
            except IndexError as e:
                print("At index [{}, {}]".format(i, j))
                raise e
```

Computation: Code Functional Description (cont'd)

- In addition to the main algorithm, I also created some external functions related to processing the image itself.
  - extract_feature_landmarks was code adapted from Joseph Rosenbrock's tutorial on facial recognition, the implementation of which was taken from a paper from Kazemi and Sullican (listed in the resources). The idea behind this paper was to use a series of 68 regression classifiers to detect specific regions of the face (which they called "landmarks"). I used these 68 points and created lines between points of similar facial features for use in the metamorphosis stage.
  - To save computational time, I created a function to detect a face in the image using OpenCV's Haar cascade classifier, crop the image to only the face, and use OpenCV's resize function to crop it to a specific size (in this case, 200x200).

```python
def extract_facial_landmarks(face_image, predictor_file=DEFAULT_SHAPE_PREDICTOR):
    # initialize dlib's face detector (HOG-based) and then create
    # the facial landmark predictor
    detector = dlib.get_frontal_face_detector()
    predictor = dlib.shape_predictor(predictor_file)

    # Convert face to grayscale
    gray = cv2.cvtColor(face_image, cv2.COLOR_BGR2GRAY)

    # detect faces in the grayscale image
    rects = detector(gray, 1)

    # Use the first rectangle detected for the face.
    rect = rects[0]

    # determine the facial landmarks for the face region, then
    # convert the facial landmark (x, y)-coordinates to a NumPy
    # array
    shapes = predictor(gray, rect)
    shapes = facial_utils.shape_to_np(shapes)

    # Return main face rectangle and list of (x, y)-coordinates
    return shapes, rect
```

```python
def crop_face(frame, dst_shape):
    frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    frame_gray = cv2.equalizeHist(frame_gray)
    #-- Detect faces
    face_cascade = cv2.CascadeClassifier()
    face_cascade.load(FACE_CASCADE_XML)
    facex, facey, face_width, face_height = face_cascade.detectMultiScale(frame_gray)[0]
    face = np.copy(frame[facey:(facey+face_height), facex:(facex+face_width)])
    return cv2.resize(face, dst_shape[::-1])
```

Computation: Code Functional Description (cont'd)

- For testing the projection code from the first two slides, I employed a unittest extension in which a binary F, as well as a 90 degree rotation and scale along the two horizontal lines was used. To give an example of some of the core early tests:
  - testLineDistance tested the UV matrix computation. Because the PQ vector in the source image is a perfectly vertical line, u was the same for each column, v for each row. The expected uv matrix was compared to the computed one
  - testReprojCoords takes two locations, one at the base of the F letter itself and another at its top left corner, and makes sure the reprojection function maps successfully maps the destination set to the source set.
  - testReproj is the final test, which simply used the variables created in setUp to project orig_image using (P_src, Q_src) to (P_dst, Q_dst), and made sure the result was identical to new_image.

```
class MetamorphosisAlgorithmTest(unittest.TestCase):

    def setUp(self):
        self.orig_image = np.array([
            [0, 0, 1, 1, 1],
            [0, 0, 1, 0, 0],
            [0, 0, 1, 1, 1],
            [0, 0, 1, 0, 0],
            [0, 0, 1, 0, 0]
            ], dtype=np.float)
        self.new_image = np.array([
            [0, 0, 1, 1, 1, 1, 1],
            [0, 0, 1, 1, 1, 1, 1],
            [0, 0, 0, 0, 1, 0, 1],
            [0, 0, 0, 0, 1, 0, 1],
            [0, 0, 0, 0, 1, 0, 1],
            [0, 0, 0, 0, 1, 0, 1],
            ], dtype=np.float)
        self.img_shape = self.orig_image.shape
        self.dst_shape = self.new_image.shape
        # Source line
        self.P_src = (0, 2)
        self.Q_src = (0, 4)
        # Destination line
        self.P_dst = (0, 6)
        self.Q_dst = (4, 6)

        self.face_1 = cv2.imread("face1.jpeg")
        self.face_2 = cv2.imread("face2.jpeg")
```

```
    # Test that the image finds the correct (u,v) pairs for a given line
    def testLineDistance(self):

# Test that a 90 degree rotation and doubling in size works
# Expects that uv computation works.
def testReprojCoords(self):

# Test that the reprojection (with floored source indices) behaves as expected
def testReproj(self):
```

Computation: Code Functional Description (cont'd)

- To put the pipeline between pre-processing (cropping out the face) and post-processing (combining the images sequences into GIFs), I used a main function that combined all of the dlib utils for extracting features from the face with the metamorphosis functions.
  - Because the result was an interpolation between the source and the destination image, both the source and the destination images had to be warped. As a result, both the forward and reverse transformations were computed
  - An interpolation, which consisted of dividing the reprojection coordinates into a series of smaller, incremental steps, converted the single matrix into a sequence
  - For each element in the sequence, the source warp and destination warps were computed, and mixed to create the expected result (although all three were recorded)
  - To make creating the GIF loop easier, the main function took in multiple images as a sequence, and looped over each consecutive pair in the sequence (going back to the start in the end).

```
def create_facial_interpolation(face_1, face_2, n_frames):
    points1, _ = dlib_util.extract_facial_landmarks(face_1)

    feat1 = dlib_util.convert_landmarks_to_lines(points1, facial_regions.get_index_pairs())
    points2, _ = dlib_util.extract_facial_landmarks(face_2)
    feat2 = dlib_util.convert_landmarks_to_lines(points2, facial_regions.get_index_pairs())

    # Determine f ( dest coordinates) => source coordinates
    X_src = metamorphosis.compute_weighted_source_coordinates(face_2.shape[:2], feat1, feat2)

    # Determine f ( src coordinates) => dest coordinates
    X_dst = metamorphosis.compute_weighted_source_coordinates(face_1.shape[:2], feat2, feat1)

    # Determine pixel coordinates along each intermediate frame
    interpolation_src = metamorphosis.create_interpolation(face_1.shape[:2], X_src, n_frames=n_frames)
    interpolation_dst = metamorphosis.create_interpolation(face_2.shape[:2], X_dst, n_frames=n_frames)

    src_warped = [metamorphosis.warp_source(face_1, X_src, pixel_est_method='bilinear', pad='edge')
                  for X_src in interpolation_src]
    dst_warped = [metamorphosis.warp_source(face_2, X_dst, pixel_est_method='bilinear', pad='edge')
                  for X_dst in interpolation_dst]

    mix_warped = [
                facial_io.mix_images(src_warped[i], dst_warped[n_frames-i], i / n_frames)
                for i in range(n_frames+1)
    ]

    return src_warped, dst_warped, mix_warped
```

```
    for i in range(len(images)):
        ip1 = (i+1) % len(images)
        image_1 = images[i]
        image_2 = images[ip1]
        name_1 = names[i]
        name_2 = names[ip1]

        src, dst, mix = create_facial_interpolation(image_1, image_2, args["frames"])

        output_dir = "{}_to_{}".format(name_1, name_2)
        write_video_directory("src", src, output_dir)
        write_video_directory("dst", dst, output_dir)
        write_video_directory("mix", mix, output_dir)
```

Computation: Code Functional Description (cont'd)

- Because I relied heavily on ffmpeg to process my code into GIFs, I created interpolate.sh (shown on the right) to automate the entire process. This bash script, after taking in the image directory name and number of frames as 1st and 2nd positional arguments, runs main.py with every image in the directory as input (the order depends on the order the ls command outputs).
- Once main.py is run, it will then run ffmpeg to convert all the image sequences created (namely the mixed ones) into GIFs using the ffmpeg command from assignment 6.
- Afterwards, it concatenates these gifs into a single output GIF, by putting the names into a list text file and running ffmpeg with the text file as input.

```
output_dir=output_$sequence_name
mkdir -p $output_dir
echo "Creating sequences, outputting to $output_dir ..."
python main.py --images `ls $sequence_name/*`  --frames $frames  --output-dir $output_dir

# List the sequences in the order they were created, to match the python sequence.
echo "Creating the gifs for $sequence_name ..."
for dir in `ls -ctr $output_dir`; do
    gif_name="${sequence_name}_$dir"
    ffmpeg -i ${output_dir}/$dir/mix/mix_%04d.png $gif_name.gif
done

# List the sequences in the order they were created, to match the python sequence.
echo "Combining the gifs into one ..."
rm mylist.txt
touch mylist.txt
for name in `ls -ctr ${sequence_name}_*.gif`; do
    echo "file '${name}'" >> mylist.txt
done

ffmpeg -f concat -i mylist.txt output_${sequence_name}.gif
```

Additional Details

I also ran an experiment on running my code end-to-end on a different set of images without any intervention. This time, I took a video of myself doing 4 facial expressions, then extracted 4 photos from the result. Using these photos as input, I created a gif animating my facial expressions and uploaded it **here: https://drive.google.com/open?id=1EZivAe40HUO8iQ-p_aswrXcS8GwN3siS**

Additional Details (cont'd)

Although most of the code dependencies come with the default cs6475 environment, the dlib library is a 3rd party dependency that will need to be installed.

Since it is not part of the standard conda packages, I will include the install command here:

conda install -c menpo dlib

In order to run the code, you run the bash script as follows:

./interpolate.sh <image directory name> <number of frames>

For my particular project, I ran the following:

./interpolate.sh family 50

(for the expressions side project, I changed the directory to **faces**)

Resources

Papers:

- T. Beier and S. Neely, "Feature Based Image Metamorphosis," 1992 ACM SIGGRAPH Computer Graphic, New York, NY, USA, July 1992, pp. 35-42.
- V. Kazemi and J. Sullivan, "One millisecond face alignment with an ensemble of regression trees," 2014 IEEE Conference on Computer Vision and Pattern Recognition, Columbus, OH, 2014, pp. 1867-1874.

Online Citations:

- Cascade Classifier https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html
- Haar cascade database: https://github.com/opencv/opencv/tree/master/data/haarcascades
- Drawing functions: https://docs.opencv.org/2.4/modules/core/doc/drawing_functions.html
- Haar cascade Parameter tuning: https://www.geeksforgeeks.org/python-smile-detection-using-opencv/
- Cascade classification: https://docs.opencv.org/2.4/modules/objdetect/doc/cascade_classification.html
- Rosenbrock tutorial for facial detection with dlib: https://www.pyimagesearch.com/2017/04/03/facial-landmarks-dlib-opencv-python/
- ffmpeg help from A6: https://github.gatech.edu/omscs6475/assignments/tree/master/A6-Video_Textures
- argparse help: https://docs.python.org/3/library/argparse.html

Resources (cont'd)

- Bilinear Interpolation: https://en.wikipedia.org/wiki/Bilinear_interpolation
- Bash help with arrays: https://unix.stackexchange.com/questions/328882/how-to-add-remove-an-element-to-from-the-array-in-bash
- Stack overflow posts:
  - https://stackoverflow.com/questions/678236/how-to-get-the-filename-without-the-extension-from-a-path-in-python
  - https://stackoverflow.com/questions/15589517/how-to-crop-an-image-in-opencv-using-python
  - https://stackoverflow.com/questions/28806816/use-ffmpeg-to-resize-image
  - https://stackoverflow.com/questions/15753701/how-can-i-pass-a-list-as-a-command-line-argument-with-argparse

Appendix: Your Code

Code Language: Python, Bash

List of code files:

- *main.py*
- *metamorphosis.py*
- *test-algorithm.py*
- *dlib_util.py*
- *facial_io.py*
- *facial_regions.py*
- *facial_utils.py*
- *interpolation.sh*

List of XML/DAT files for face detection:

- *facial-regions.txt*
- *shape_predictor_68_face_landmarks.dat*
- *haarcascade_frontalface_alt.xml*

Credits or Thanks

Thanks to my relatives for allowing their faces to be used for my assignment!