Face morphing and Active Appearance Models

Table of Contents

1. Introduction
2.1 Facemorphing: code overview
2.2 Mean face
2.3 Shape Models
3. Active Appearance Models overview
4. Eigenfaces: appearance modeling
5. Simulated annealing: optimization process
6. Conclusion
7. Usage

1. Introduction

The initial goal of this project was to do face morphing among at least three subjects, and it was suggested that I also look into Active Appearance Models. Now the result is that I have infrastructure to do shape modeling, appearance modeling, and explored a tentative optimization method of simulated annealing to fit an AAM to a new subject. The current system cannot actually fit the model because I did not implement parameterization or parameter-variation, which are the steps that an optimization process would take to iteratively fit models to face images.

2.1 Face morphing - code overview

To do face morphing, corresponding landmarks on each faces are manually chosen. The more points chosen the better to increase the resolution of transformations.

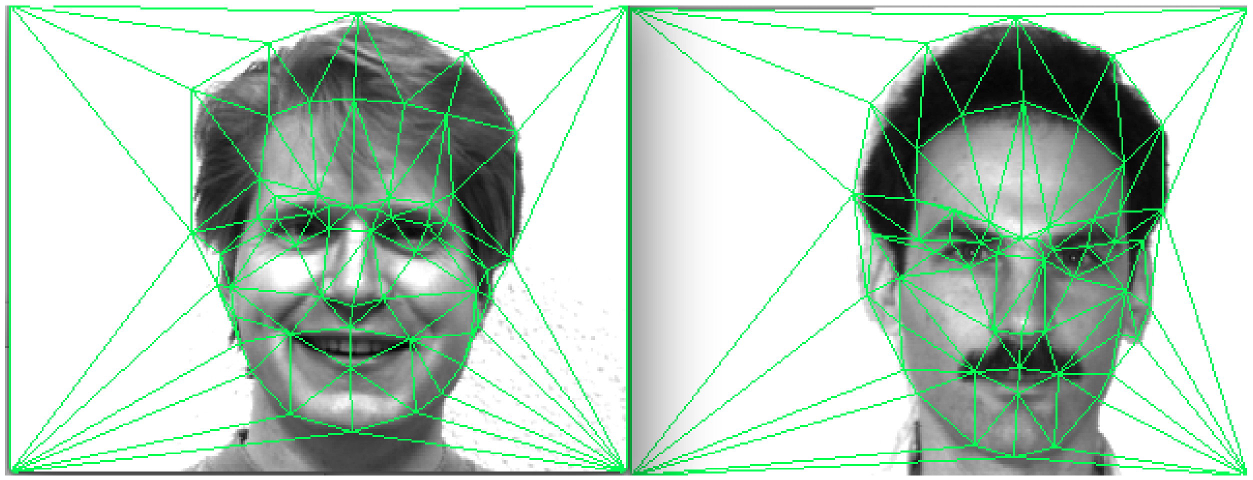


Figure. Delaunay triangulation from chosen landmarks

Then to find the transformation that describes the correspondence relationship, the landmarks are triangulated so that we can find a basic affine transformation (which can describe the transformation between any two triangles). With delaunay triangulation, it is guaranteed that we have the least number of triangles and maximum angles between edges. This also ensures that no intersecting or overlapping triangles are formed. The triangulation for both images must be the same, so in `delaunay2Faces()`, the delaunay triangulation is only done on one image, then the corresponding points on the other image are found with `getFPindex()` and triangulated at the same time in the same order.

After finding the affine transformation for every pair of correspondences, we can now warp one face to another. To get the color of each pixel of the new face, we do an inverse transformation to refer back to the original face. This is because the forward transformation could overlap other pixels while leaving others empty. Thus, iterating through each pixel in the new image, we find which triangle it was from by getting the index of the triangle in the vector set with getTriangleIndex, checking whether the point is in the current triangle with `isTriangle()`.

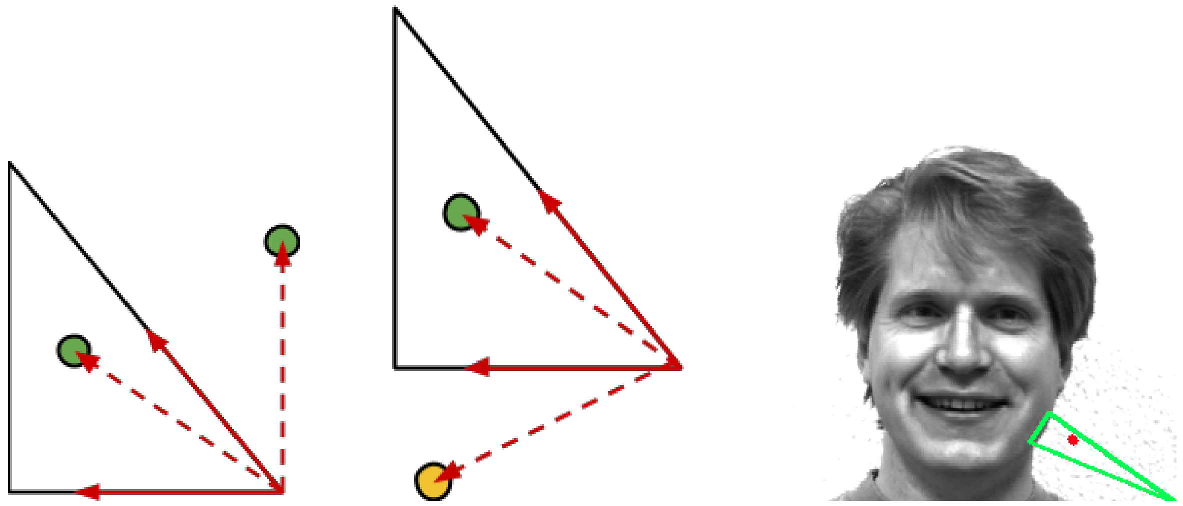


Figure. If green point is in the triangle (left), the cross product with one edge will have the same direction as two edges of the triangle around that vertex. This must be done for all three vertices for a complete check, else it would miss points like the yellow point (center). Point found in correct triangle (rightmost).

To determine if a point is in the triangle, I compared cross products around every vertex. For each vertex, I would compare the cross product between edges with the cross product between one edge and the point. If they are in opposite directions, then the point lies outside of the triangle. An alternative method to comparing just cross products would be to compare cross products with barycentric coordinates, since in a triangle the coefficients of the coordinates add up to 1 and we can use this comparison to determine whether a point is in a triangle.

By finding the triangle, we can find the affine transformation that warped that triangle, and thus calculate the inverse affine to get the correct color.

To step through the transformation, simply interpolate the triangles, which is done in `getAllTriangles()`, get the affine transformation, and repeat the warping process described above. All of this is done in `warpAllXLevels()`, which interpolates through X steps to get X warps with `warpImage()`.

The limitation is that it is tedious to pick 30 to 50 landmarks on the face in the correct order between every two images for morphing, and for every image in the mean face calculation.

As for how the face morphing works under more extreme conditions, the rmMorphFast.mp4 video shows that different head sizes, heads with/without hats, angles, poses, and genders could be reasonably smoothly interpolated between without further coding. One improvement that could be done might be to do a weighted interpolation to make the transition look smoother.

2.2 Mean face

To get the mean face between two images is trivial- just get the midpoint warp of the interpolation. For multiple images, we iteratively warp the mean face between two images to the new image, and get the midpoint warp for the new mean face. This is done in `getMeanIm()`.

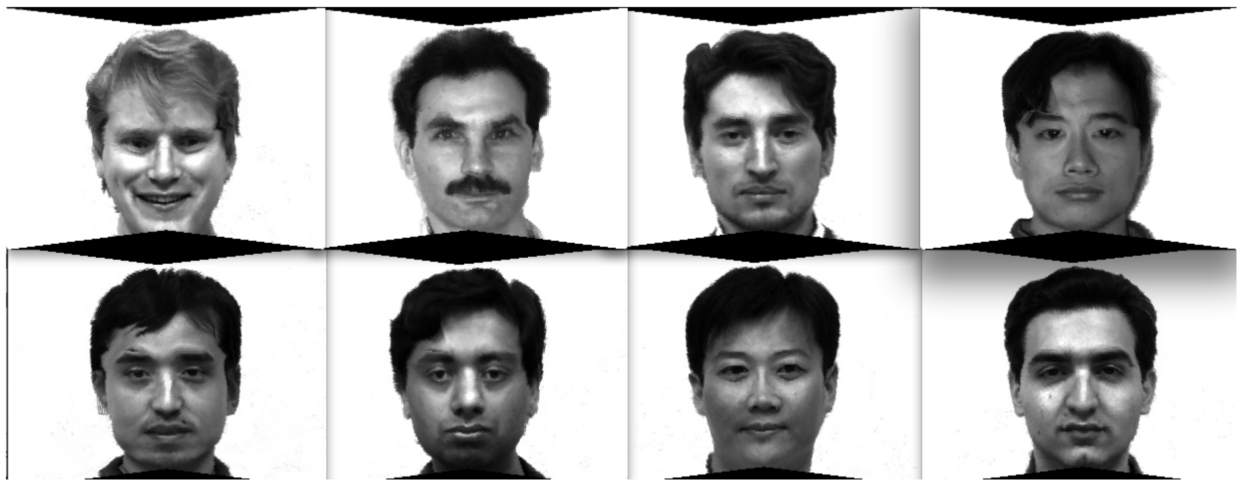


Figure. Individual images warped to mean image's shape model.

One interesting thing to note was that the mean face would be the most visually pleasing of the faces, and warping individual images to the mean face would sometimes look better.

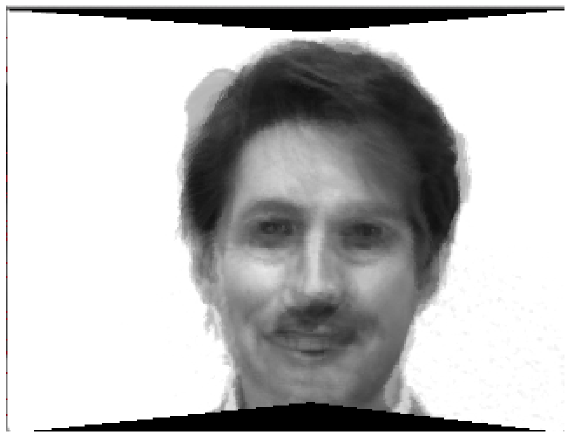
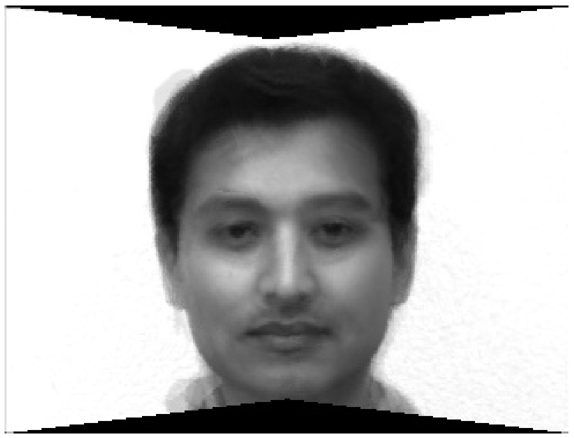



Figure. Mean of three images (left); mean of eight images (right)

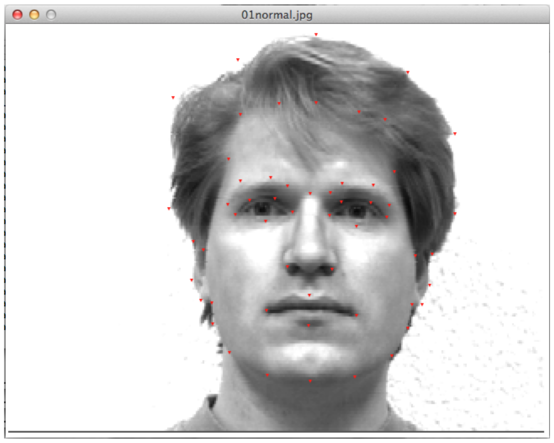
At three images, the features of the mean image were hard to distinguish. At eight, the mean image looked like a real face. For AAM's or more useful mean images, perhaps at least hundreds of faces are needed and even then racial or age-based considerations need to be accounted for (Caucasians typically have sharper noses that Asians, who usually have smaller eyes, and so on).

2.3 Shape models

With this, we are now able to construct other models by picking different sets of landmarks. For the Yale faces, the subjects were photographed in a regular manner and face-on. Their hair and even hats were taken into consideration during the warp as face morphing is about making the most visually appealing transformation. The landmarks around the eyes should be denser as making the eye transformation look natural is most important since humans by nature relate to eye contact. For an AAM shape model only the face would be considered, where the landmarks trace out the jaw and chin, the sides of the nose, the eyes, and the eyebrows.

With shape modeling, we are one step closer to making Active Appearance Models. Over a large (>100) training set, the mean shape model would be a good approximation to the general mean face, although there would be differences between male and female faces due to the bone and fat construction differences (men have more angles; women more curves), and between ages (a toddler's face is more squashed across the front of its head than an adult's face would.

With the points gathered, we can also computationally make new models by parameterizing the points, and calculate new triangles and thus new affine transformations.



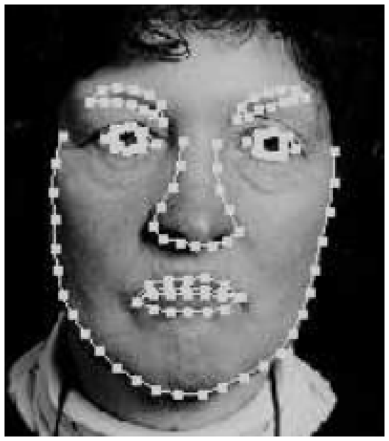


Figure a) Landmarks for face morphing b) From AAM landmarks from [1]

3. Active Appearance Models - overview

AAMs are general models good for describing organic subjects such as organs and faces. With only shape modeling, a new image can only be approximated to by considering its shape, disregarding its unique texture (Active Shape Models). An appearance model would take care of this, and can be constructed via Principal Component Analysis (PCA) to make eigenfaces, which is done in `getEigenFaces()` and described in the next section.

AAMs combine shape and appearance, then varies the parameters with small changes such as in size and position. Faster models would involve generating a model at different resolutions, or different Gaussian pyramid levels, and then changing the parameters of the smaller images while considering landmarks of the higher resolution images.

The limitations of AAMs is that the training set of images need to be face on, and have little variation in lighting. In fitting to models, it would probably work poorly with extreme-view models unless training sets from different angles (side, bottom, top) are taken, although in other papers it was shown that AAMs were robust enough to allow faces to deviate by small angles and still get matched to. Another issue is that the images are gray scaled first such that only the gray-level appearance is considered. In otherwords, most real life images, which have faces under varying lighting conditions and view points, occlusions, and even under varying natural conditions such as rain, sweat, facepaint or acne, would be hard to model as well.

4. Eigenfaces - appearance modeling

To describe one face, we would want to describe the deviations of that unique face from an average face. PCA finds the largest variances within a data set (finding the principal components), and the directions of the components are described by the eigenvectors of the covariance matrix of that data set. The first principal component would be the eigenvector with the largest eigenvalue, the second eigenvector has the second largest eigenvalue, and so on.

Each face that constitutes the mean face set can derive an eigenface. As these eigenfaces are orthornomal eigenvectors, we have a facial coordinate system, where different linear combinations of the basis would give us any vector within the range. The coefficients in this linear combination are usually known as 'weights'. So, we have a basis that from which we can (a) use to characterize faces since we can get patterns from various weight combinations of the faces; (b) construct new faces by varying the weights.

Since the eigenvectors help describe the variances, with the mean face as the origin any linear combination of the eigenvectors would return a desired face.

`getEigenFaces()` relies on `getMeanIm()` being executed first, so that we have a mean image to work relative to. To construct the covariance matrix, Cov = A A^T. To get A, column constitutes of: each individual face image (2d-matrix) is rolled out into one long 1d-vector, and this long vector is subtracted by the rolled-out mean image. The difference images show how similar an image is to the mean image, where the brighter images are more unique, and the darker images are similar to the average image.

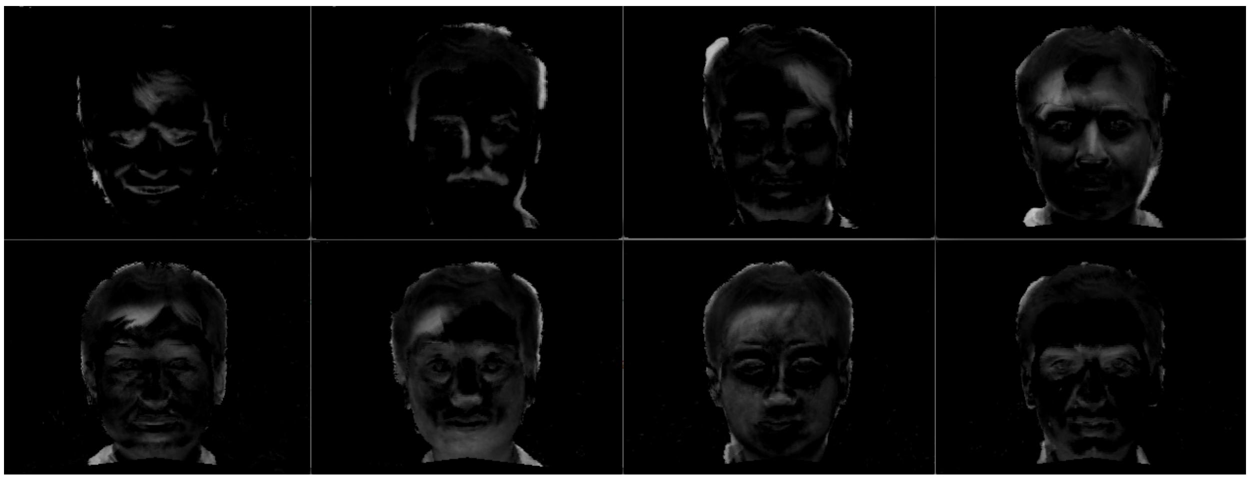


Figure. Individual images diff-ed with the mean image. Not a great deal of variation (white areas) as there are only 8 images to constitute the mean.

Then, by getting the eigenvectors of the covariance matrix (through SVD), we have the eigenfaces. One problem to note is how expensive it could be to do this, since a rolled-out 100x100-image would be 10^4 elements long, and the covariance matrix would thus have 10^8 elements. Instead we get Cov = A^TA to get a matrix only as large as the number of images in the set (8 images would return an 8x8-matrix). This is acceptable since the eigenfaces can be recovered by multiplying each vector by A.

These eigenvectors are known as eigenfaces in face systems because when the rolled-out vector is reconstructed into the image plane, the resulting matrix looks like a face.

For AAMs, the faces are first shape-normalized (warping image to mean) before extracting appearance information so that we don't consider shape variances. Making the two independent of each other is important for using the models in an expected, logical manner.

Non-shape-normalized eigenvectors show greater variations, whereas the shape-normalized ones looks more specific:

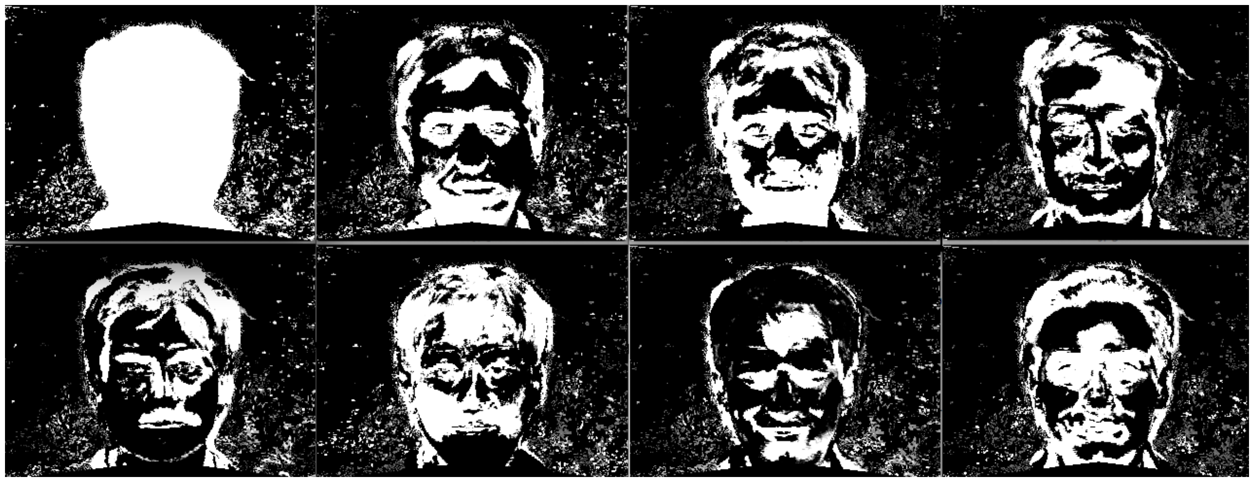


Figure. Eigenfaces after images were shape-normalized (warped to the mean image)

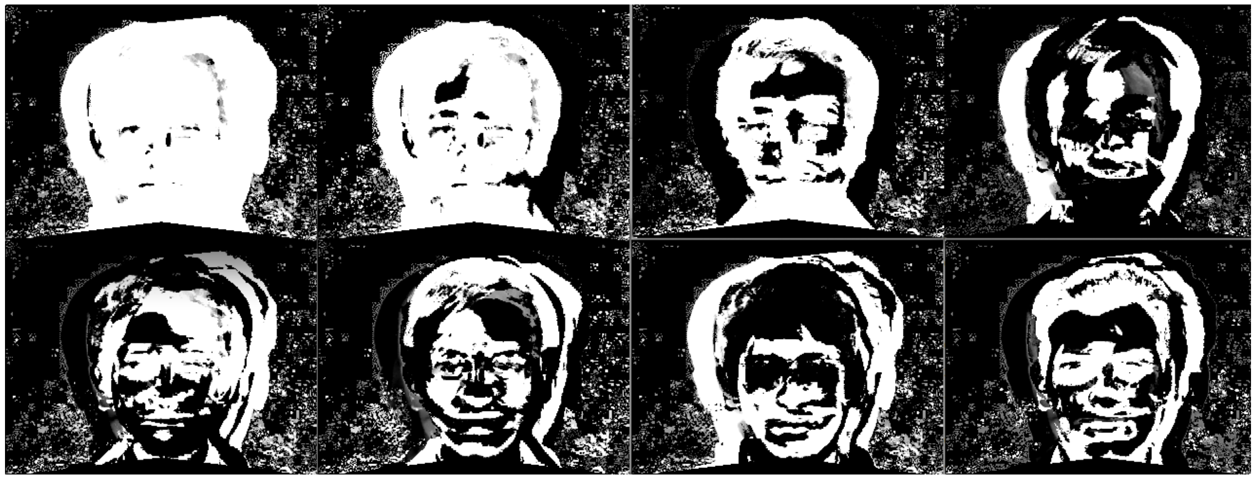


Figure: Eigenfaces without normalizing shapes.

5. Simulated Annealing - optimization process

Next, if we want to actually vary the models, such as fitting the models for face recognition, we need a process that directs hundreds of parameters which can go in any direction towards a certain goal in an efficient manner. I have not learned about optimization, so there should be better algorithms, but simulated annealing seemed to be reasonably smart but easy to check out. Mimicking the physical process of atoms returning to low-energy levels, simulated annealing seeks to decrease the entropy of a system, where the entropy is how far the system is from the goal state. It does this by accepting changes to the system that either decrease entropy, or increase it. It accepts higher-entropy changes only with a certain probability that is proportional to the entropy levels. In other words, simulated annealing is able to get out of local minima, unlike gradient descent, because it can travel up gradients with some probability. To make sure that it reaches the global minimum, however, the probability varies so that the lower the energy the state the less likely it is to except energy increases.

I have a prototype of this in Python in stochopt.py, where two arrows are placed randomly facing random directions. The goal state is for the arrows to find each other (arrow tips touching). Here, the simulated annealing process is directly observable because you can see the incremental steps the arrows take to find each other.

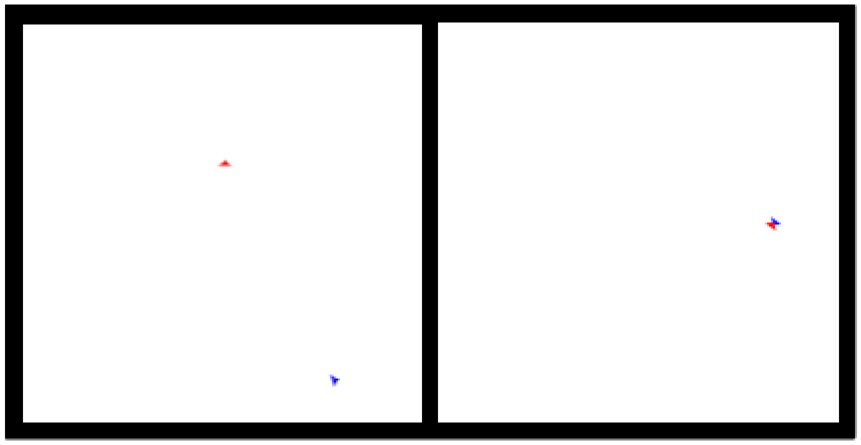


Figure. Arrows before starting (left). Arrows after (right).

If this process were generalized to modify the shape and appearance of an active appearance model, we could fit the model to new shapes with simulated annealing, which is flexible enough to get out of local minima. In terms of code, it proposes a step (random rotation or random translation) in `proposalCost()` and calculates the cost of that step with `costFunction()`. If it decreases energy, the step is automatically accepted. Else it accepts the cost with probability `tempAcceptProb()`, which is based on the temperature (and thus entropy) on the situation.

The disadvantage of simulated annealing is that it might be too slow for handling many parameters.

6. Conclusion

This project has described how to do face morphing, in which shape models had to be constructed.

Then, it went on to constructing eigenfaces with PCA to make appearance models, and explored one optimization process, simulated annealing. Active Appearance Models combines both shapes models and appearance models, and makes it possible to do useful things like face recognition by fitting an appearance model to the new image, and taking note of how we got there. The fitting should be done through optimization methods instead of brute force, which might take forever.

The main challenges of this project were implementation details and understanding how and why AAMs worked.

One way this project could go is to finish the AAM by combining shape and appearance parameters since no AAM is actually constructed- the shape models taken were only done for face morphing, although it would be trivial to simply choose different points; the shapes and appearances were not parameterized so that the optimization process could be applied across them.

From there, interesting applications like facial recognition or perhaps even emotion recognition be done could be done. By tracking specifically how a face morphs and comparing this with an extensive training database, we would not only be able to identify instataneous emotions, but also maybe even detect false emotions (fake smiles, joking indignance).

7. Usage

Detailed in README.txt.

The executable was built in Mac 10.7+, OpenCV 2.4.5.
Possible compilation issues (XCode settings):
Header Search Paths: /usr/local/include
Library Search Paths: /usr/local/lib
C++ Standard Libary: libstdc++ (GNU C++ standard library)

facepointChooser.py

```
$ python facepointChooser.py -l <log file name> -i <image0> [<image1>
…]
```

All chosen coordinates will be logged in one log file. Simply copy and paste each relevant section (headed by “ **<name$_i$>”) into separate log files, and put the files into the same directory as the faceMorpher build. Sample log files are included for reference/ testing use.

faceMorpher

For basic help information, just run `$ ./faceMorpher` for help information. Note: options can be input in any order.

- Facemorphing

```
$ ./faceMorpher -i0 <image0> -f0 <image0's landmarks> -i1 <image1>
-f1 <image1's landmarks>
```

- Mean face

```
$ ./faceMorpher -m ./<images to make mean>/ -mf ./<faces-for-mean's
landmarks directory>/
```

- Eigenfaces

```
$ ./faceMorpher -m ./<images to make mean>/ -mf ./<faces-for-mean's
landmarks directory>/  -e <1|0>

-e 1: eigenface with normalized shape (for AAMs)
-e 0: eigenface directly from images (normal)
```

Simulated annealing

```
$ python stochopt.py
```

Dependencies: Python 2.7+, Tkinter

Stops after 1000 iterations, usually finds a good solution within 200.

Note: code to draw arrows, in stochoptVisuals.py, was adapted from http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-00-introduction-to-computer-science-and-programming-fall-2008/assignments/ps11_visualize.py, which drew robots for something entirely different)

7.1 References

1. Face Recognition using Principal Component Analysis - Kyungnam Kim
http://www.umiacs.umd.edu/~knkim/KG_VISA/PCA/FaceRecog_PCA_Kim.pdf
2. Active Appearance Models - Cootes et al.
http://www.cs.cmu.edu/~efros/courses/AP06/Papers/cootes-eccv-98.pdf

Most face images are from the Yale Face Database (http://vision.ucsd.edu/content/yale-face-database) which had grayscale images from 1997 of different people with a variety of facial expressions.