Automated Face-Morph Generation Tool

Biometric Systems (02283)

Robert Scholes Lyck

June 2018

Abstract

This paper describes an automatic Face-Morph Generation software tool, implemented as a project for a course on Biometric Systems at the Technical University of Denmark (DTU). One form of attack on biometric systems employing face recognition, is to morph images of two contributing subjects, into a single image to fool the system into accepting both subjects, as seen in *The Magic Passport* paper [1]. This software tool is written in Python and comes with both a simply GUI and command-line interface. The tool implements two methods for face image morphing, both with open source code adopted from online tutorials on the topic. The software design, usage, morphing algorithms and underlying theory are described. The morphed image results are based on standard face databases and are evaluated against other work in this field.

Contents

1 Software tool **3**
1.1 GUI . . . 3
1.1.1 UI . . . 4
1.1.2 Design . . . 5
1.2 Command-line . . . 5
1.2.1 UI . . . 6
1.2.2 Design . . . 7

2 Morphing methods **7**
2.1 Full image . . . 8
2.2 Face swap morph . . . 10

3 Results **11**
3.1 FEI face database . . . 12
3.2 Comparison to related work . . . 13

4 Conclusion **16**

5 Further work **17**

References **17**

1 Software tool

This software tool is written in Python (v3.5). It is fairly easy to make Python projects into standalone executables with a library like PyInstaller [10], but the product of this project is a purely Python project, i.e. the Python environment and project specific dependencies has to be present on a computer in order to use these tools.
The primary packages used in this project are:

- OpenCV [7]
- dlib [8]
- wxPython [9]

Project folder structure and files

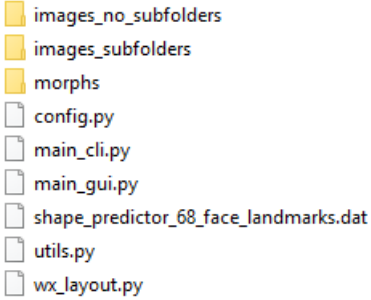

Figure 1: The folders are just for conveniences sake. The `shape_predictor_68_face_landmarks.dat` file has to be at the same level as the `main_cli.py` file. It is used for facial landmarks detection. `utils.py` contains all the morphing specific functions. `wx_layout.py` is an auto generated GUI file.

Once these requirements are met, there are two ways to interact with this software tool, that doesn't require editing the source code.

- GUI: `python main_gui.py`
- Command-line interface: `python main_cli.py [optional args]`

1.1 GUI

The GUI invokes all the same methods that the command-line tool does, which is why they will be explained in more detail in the command-line section.

1.1.1 UI

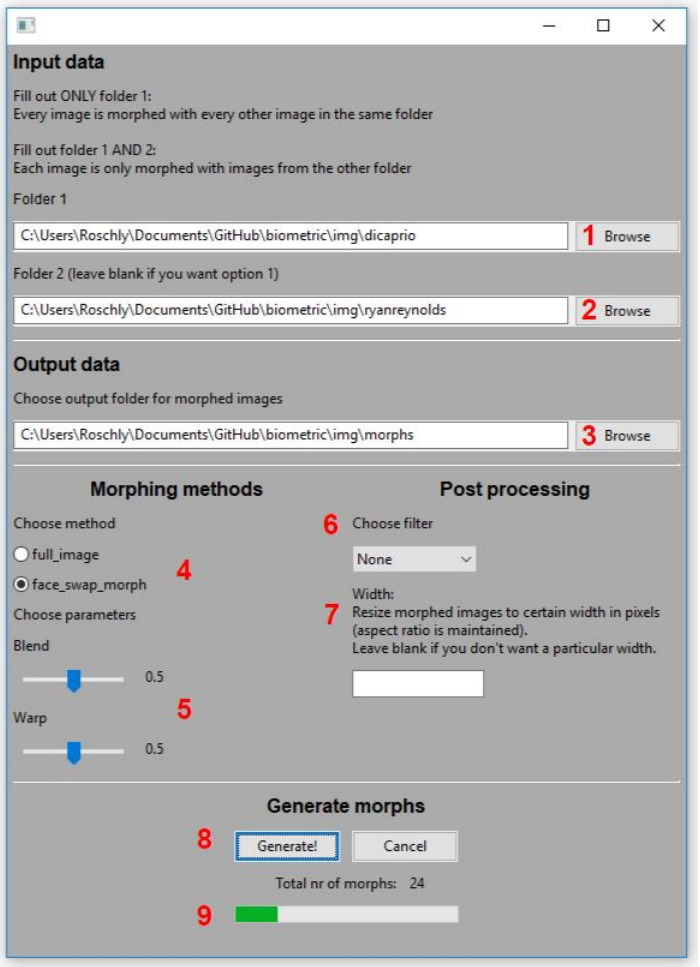

Figure 2: Overview of the Graphical User Interface:
1) Choose input directory of images
2) Choose an optional second input directory (changes behavior)
3) Choose which output directory to save the created morphs to
4) The two possible morphing methods
5) Blend and warp parameters for the chosen morphing method
6) Drop-down list of post processing filters to choose from
7) Specify width of morphed images (aspect ratio is kept)
8) Generate button to start the generation of morphs and a cancel button to terminate the process prematurely
9) Progress bar showing the percentage of completed morphs so far

1.1.2 Design

The GUI is created with a library called wxPython [9] which is a cross-platform GUI toolkit for the Python language. It was chosen because it allows for native user interfaces, with little or no modifications to run on multiple platforms. It has an accompanying program called wxFormBuilder [11] that allows for drag-and-drop GUI design. And both tools are open source.
All of which were ideal for this project.

wxPython, and so also wxFormBuilder, is designed in an object-oriented and event-driven manner. This means that every element in the GUI is an object (with its own class) and interactions with the elements trigger events. The output of the wxFormBuilder is a Python file, that is not meant to be altered manually, with hooks into the different events, e.g. pressing the generate button.

These hooks are simple function names, that are declared in the wxFormBuilder and attached to an event (e.g. onClickButton). These functions are defined in the main application file, that tie the GUI with the back-end morphing algorithms.

1.2 Command-line

The command-line tool can be used with the `config.py` file alone,
`main_cli.py [args]` alone or a combination. Any argument passed directly to `main_cli.py [arg]` will override its counterpart in the config file. This is for ease of use, so, for instance, cumbersome folder paths can be defined in the config file and the morphing methods can be easily changed for each execution.

1.2.1 UI

```
usage: main_cli.py [-h] [-i IN_DIR] [-i2 IN_DIR_2] [-o OUT_DIR] [-b BLEND]
                   [-w WARP] [-m MORPH_METHOD] [-f FILTER] [-mw MORPH_WIDTH]

optional arguments:
  -h, --help            show this help message and exit
  -i IN_DIR, --in_dir IN_DIR
                        path to input images
  -i2 IN_DIR_2, --in_dir_2 IN_DIR_2
                        path to second folder of input images
  -o OUT_DIR, --out_dir OUT_DIR
                        path to save morph images to
  -b BLEND, --blend BLEND
                        blend value
  -w WARP, --warp WARP  warp value
  -m MORPH_METHOD, --morph_method MORPH_METHOD
                        Morphing method: full_image OR face_swap_morph
  -f FILTER, --filter FILTER
                        post processing filter
  -mw MORPH_WIDTH, --morph_width MORPH_WIDTH
                        set width of morphed image
```

Figure 3: Command-line interface usage and arguments list

The following is a list describing the different arguments to the command-line program:

1. **INPUT_DIR**
   Path to input directory.

2. **INPUT_DIR_2**
   Path to second input directory. If this is set, the program only combines each image from one with all images on the second folder.
   If this is blank, all images in folder 1 are morphed with each other.

3. **OUTPUT_DIR**
   Path to output directory

4. **BLEND**
   Blending controls the share of pixel values from each image in the resulting morph. It is a real number in the range [0.0, 1.0].

5. **WARP**
   Warping controls the contribution of each subject's facial landmarks positions, to the resulting morph's landmark positions. It is a real number in the range [0.0, 1.0].

6. **MORPH_METHOD**
   Which morphing method to use.
   Two available: full_image and face_swap_morph

7. **FILTER**
   One of the following post processing filters can be selected:
   `none, sharpen, blur, smooth, smooth_more, detail, edge_enhance`
   These are standard image processing filters and implemented using the Pillow Image Module [12] (see reference for detailed descriptions)

8. **MORPH_WIDTH**
   Resizes the morphed image to a specific width (keeping aspect ratio). Resizes to any positive integer. If set to -1, the morphed image keeps it's original size.

The config file is a standard Python file. It defines the same arguments as the command-line. Being a Python file (instead of a text or json file) makes it possible to import Python libraries and built-in utilities to ease cross-platform folder paths and make arguments defined by external parameters if need be.

1.2.2 Design

The main function in the main_cli.py file, takes all the command-line/config file defined arguments and performs the following steps:

1. Finds all the image pairs and the total number of these pairs. The pairing method changes according to whether only the INPUT_DIR parameter has been set, or also the INPUT_DIR_2 parameter.

2. For each pair:

   (a) Morph them together, parameterized by blend and warp.

   (b) Combine original image file names into the morph filename.

   (c) Apply post processing filter, if other than "none" was chosen.

   (d) Resize morph if determined by parameter

   (e) Create output directory if it doesn't already exist.

   (f) Save morph to file.

   (g) Print progress.

   (h) If called from GUI, send progress to GUI and check if morph generation process should be terminated prematurely.

2 Morphing methods

Two morphing methods are implemented: full_image and face_swap_morph.

Both of the implemented methods have been adopted from source code presented on the following online tutorial sites:

- www.learnopencv.com [3, 4, 5]

- www.pyimagesearch.com [6]

A lot of retrofitting and refactoring of the code was necessary to fit it into this project.

2.1 Full image

The simplest way to "morph" two images is just to overlay them and let a blending parameter, α, control the ratio of each image's pixel value, in the final combined image. A natural choice for blending is $\alpha = 0.5$ as it equally mixes the two images.

Figure 4: Naive overlay approach to morphing

Unsurprisingly, this approach is too simple and will not produce genuine looking morphs. The obvious problem is that the images are misaligned. A better approach is to find corresponding pixels in each image and make sure they align when morphing.

Facial landmarks

Researchers have found 68 facial landmarks that are present in most if not all humans. These landmarks can be identified in each image and used as corresponding points for alignment.
In addition to the 68 facial landmarks, 8 points are added along the border of the image, to ensure alignment of the entire image and not just the face.

Delaunay triangulation and warping

Each image now has a total of 76 corresponding points. The morphed image will have 76 landmarks as well, that can be calculated by deciding where on the spectrum between the corresponding landmarks in the two original images, the landmarks should lie. A new warping parameter, β, controls this ratio. Setting

$\beta = 0.5$ ensures the morph landmarks will lie in the middle of the orignal images.

But the vast majority of the pixels in each image still don't have corresponding pixels. To remedy this, a technique know as delaunay triangulation [5] is performed on the landmarks. This technique combines the points into triangles with nice properties, specifically that triangle vertices are from the set of landmark points and that no triangle contains more than those three points.
With this, we have corresponding triangles instead of points and these triangles cover the entire image and therefore every pixel.

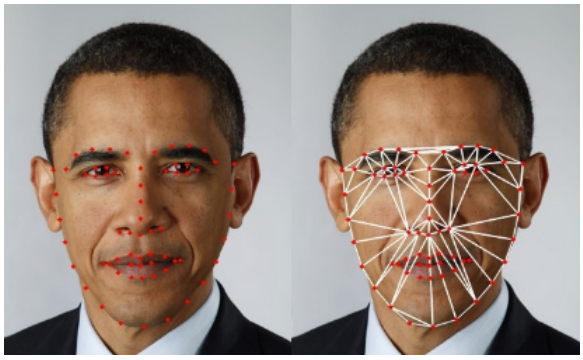

Figure 5: Facial landmarks and delaunay triangulation

Transforming corresponding triangles from the two original images into a median triangle, somewhere on the spectrum between them (controlled by the warp parameter), can be achieved by affine transformations. These are linear transformation that preserves points, straight lines and planes.

Results

The final morphed image is controlled by the blending and warping values. Natural choices for these parameters are the mean values of 0.5, but it is possible to prioritize one image over the other if need be.

Figure 6: full_image morphing with blend and warp at 0.5

Total number of morphs created

The order of the contributing subjects to a morph, with this method, is irrelevant. So the total number of morphs created, from a single folder where every image is morphed with every other, is: $\frac{N^2-N}{2}$

2.2 Face swap morph

Building on the previous method

A problem with the previous method is, that the highest resolution of landmarks is concentrated around the face. This leave the rest of the head, hair and general background without any directly corresponding points. This is why a lot of ghostly artefacts occur around the face with the previous method.

This problem has no easily automatable solution, since we cannot establish a matching 68 (or more) background or hairstyle landmarks, as could be done with the face. There is simply too much variation in both hairstyle and background (objects, landscape, shadows, color, etc).

One approach is to use the face-surrounding area from one of the two contributing images entirely, and only blending the facial areas. Assuming two realistic looking original images, the face-surrounding area from either will most likely also appear realistic and avoid the ghostly artefacts.

Morphed face swap

The way to achieve this, is as follows:

1. Full image morph:
   Use the previous method to achieve a full image morph, with given blend

and warp parameters.

2. Facial region:
   Extract the facial region from the morph. This region is defined by the convex hull around the morphed facial landmarks (so not including the 8 additional border points).

3. Face-surrounding image:
   Choose either of the original images as supplier of the face-surrounding area. Morph this image, but only using the (same) warp parameter (no blending). This ensures the face-surrounding area matches the morphed facial region, that will be overlain.

4. Seamless cloning:
   Simply overlaying the morphed facial region will look very artificial. Differences in skin tone and lighting and more, will be very clear. OpenCV has a `cv2.seamlessClone` method designed to mitigate this difference.

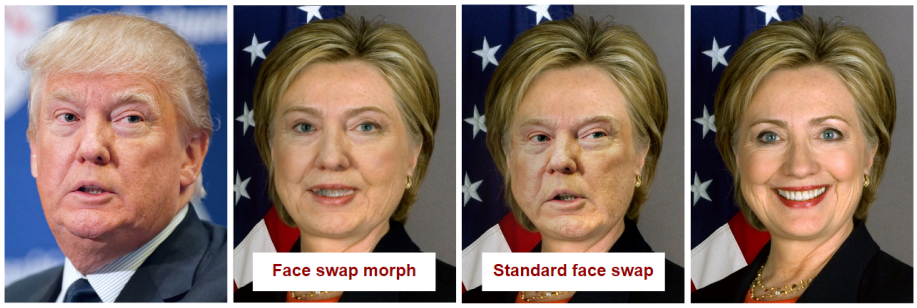

Figure 7: The difference between a standard face swap, with stretching of the face to match, and the face_swap_morph method.

Total number of morphs created

With this method, the order of the contributing subjects matter, since the first will supply the face-surrounding area. Total number of morhps: $N^2 - N$

3 Results

All morphs have been created with the "sharpen" post processing filter.

Lip artefacts: In morphs from both methods there appears artefacts on lips in several situations. There is no apparent reason for this. It is most likely a bug or the result of a rounding error in the code. Since they appear to happen randomly and have been described here, they will be ignored in the results and focus will be on the rest of the image.

3.1 FEI face database

A subset of the FEI face database is used [13]. The subset contains frontal face images that have been manually aligned, with a white homogeneous background. Furthermore, three female subjects with similar features have been used for these results. This constitutes a very ideal situation for creating a good morph.

Method: full_image (blend: 0.5, warp: 0.5)

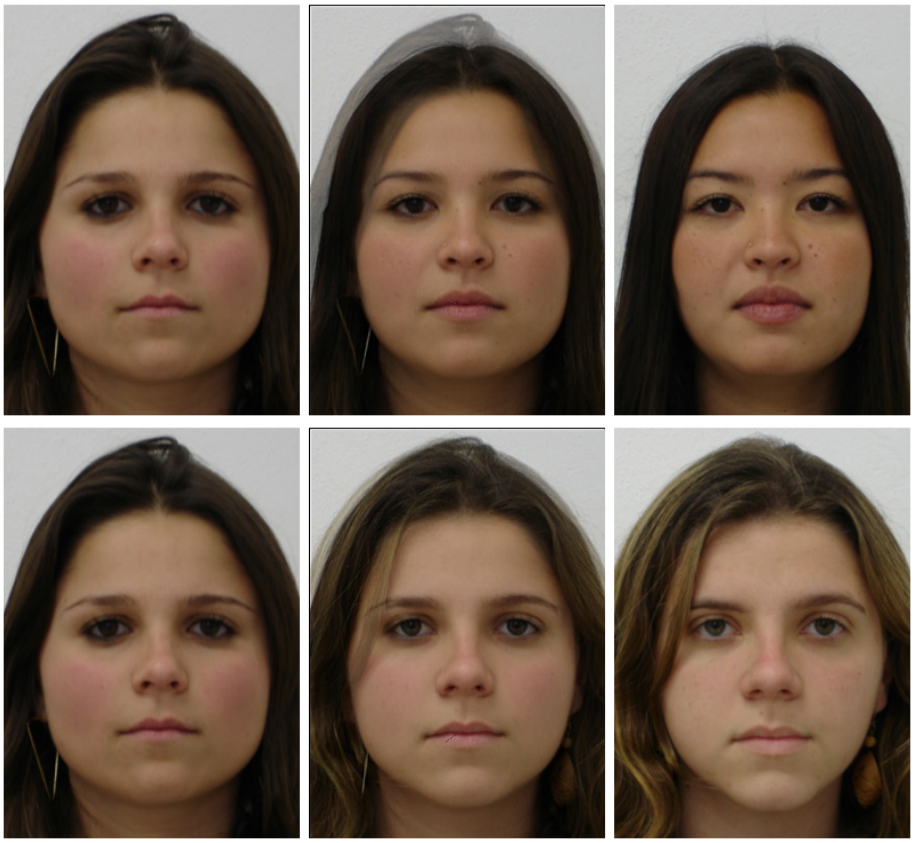

Figure 8: Under these very ideal conditions, the full_image morphs are quite good, except for the obvious ghostly hair. With similar haircuts, this could be mitigated.

Method: face_swap_morph(blend: 0.5, warp: 0.5)

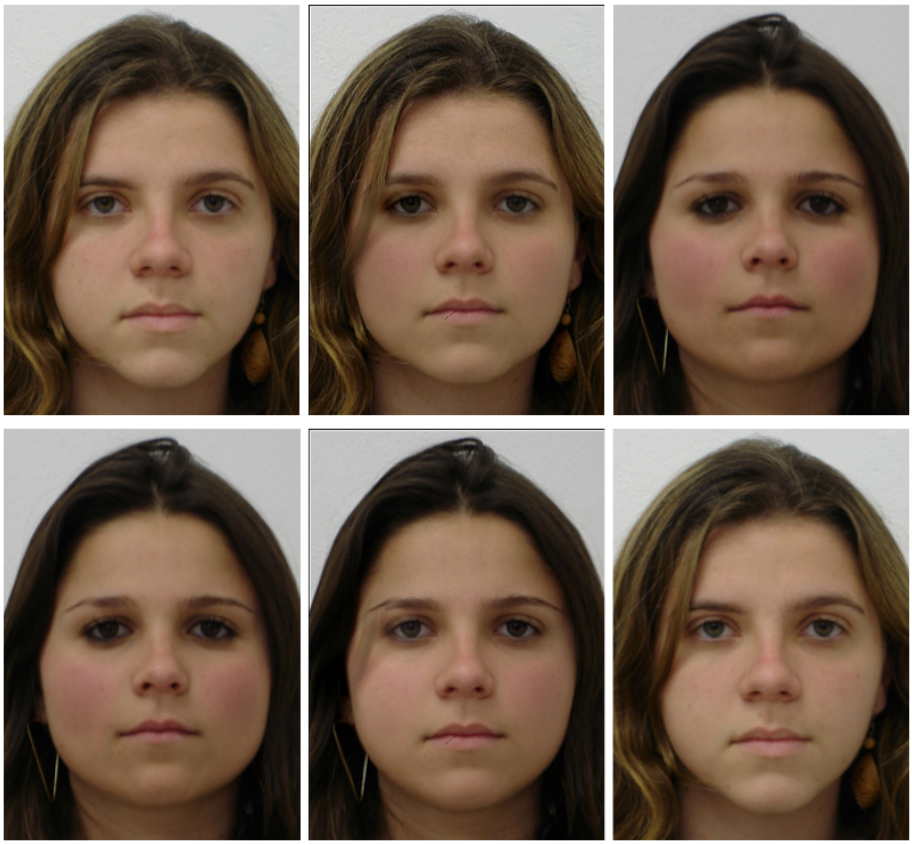

Figure 9: In the top row, the method morphs the hair partially covering the eyebrow, quite well. In the bottom row, it is the opposite case and an artefact is quite visible. But no ghostly hair and skin tones match.

3.2 Comparison to related work

In a related paper [2] on biometric systems under attack from morphing attacks, different morphing techniques are evaluated. Three morphs, from two contributing subjects, are created with different techniques and have different levels of quality.
Those two subjects and the resulting morphs are used in the following section to assess the quality of the morphing methods described in this paper.

Method: full_image (blend: 0.5, warp: 0.5)

Figure 10: More ghostly artefacts appear here than in the full_image morphs from the FEI database. The difference between images are larger here, with more background and clothing and greater skin tone difference as well.

Method: face_swap_morph(blend: 0.5, warp: 0.5)

Figure 11: The morph in the top row is noticeable better than the one in the bottom row. The large skin tone difference are visible in the bottom one.

Quality comparison

Figure 12: Top row: morphs from related paper [2] in decreasing order of quality. Left most is created manually, the other two automatically.
Bottom row: left and middle is from face_swap_morph, with the left of highest quality. Right is from full_image and very similar to the morph just above it.

In general, it is very difficult, if not impossible, to create realistic morphs if the contributing subjects differ too much in age, gender, skin tone and/or ethnicity.

4 Conclusion

This Face-Morph Generation tool has a simple and easy to use GUI and command-line interface. Building on cross-platform and open source native GUI libraries and tools, it is easy to extend this tool.

The two supported morphing methods are widely known and explained in great detail in online tutorials. They rely on very popular libraries such a

OpenCV and dlib, that makes it easy to extend and improve upon them.

The process of creating large number of morphs from a data set has been completely automated; e.g. no user input is required for facial landmarks detection. This was a stated goal of the project and makes it a very convenient tool, but also sacrifices quality in some situations.

It worth noting that machine learning tools are revolutionizing these kinds of image operations for both pictures and videos at this point in time. This tool contains no such ML algorithms. This tool could either serve as a way to generate a large initial data set, required to train machine learning algorithms like neural networks. Or could itself be extended with such algorithms, as an additional post processing step, to clear up ghostly artefacts.

5 Further work

There are several ways to improve and extend the current software tool. Some of the most obvious problems and improvement opportunities are listed below:

- Artefacts around the lips on many morphs from both methods could be from a bug or rounding error somewhere in the code.
- The GUI has very little error handling, type/bounds checking and does a poor job communicating these errors to the user.
- Artefacts around the eyes are present in both methods. These areas are high in detail, very distinct and very difficult to perfectly align. More landmark points in these regions could alleviate the problem.
- Finding similar looking images, to create better morphs could be implemented. Features such as age, gender, skin tone, hair color, etc. could be used.
- Many data sets have image feature information in the file name, e.g. emotion, lighting, angle. This could be used to improve morphing.
- The code could be more efficient. For example, facial landmarks are calculated for each image pair, instead of for each image. This means a lot of redundant facial landmark detection.
- Another way to optimize the code, would be to make sure the `face_swap_morph` method only creates one morph of each image. As of now, the method invokes the `full_image` method first and then creates another another morph (without any blending for face-surrounding purposes).
- Crop post processing functionality, to remove any leftover black or white borders, that occur on some morphs.

References

[1] M. Ferrara, A. Franco and D. Maltoni: *The Magic Passport*. In IEEE International Joint Conference on Biometrics IJCB), 2014.

[2] U. Scherhag, A. Nautsch, C. Rathgeb, M. Gomez-Barrero, R. Veldhuis, L. Spreeuwers, M. Schils, D. Maltoni, P. Grother, S. Marcel, R. Breithaupt, R. Raghavendra, C. Busch: *Biometric Systems under Morphing Attacks: Assessment of Morphing Techniques and Vulnerability Reporting*. Proceedings of the IEEE 16th International Conference of the Biometrics Special Interest Group (BIOSIG), Darmstadt, September 20-22, (2017).

[3] Learn OpenCV: Face Swap Tutorial
`https://www.learnopencv.com/face-swap-using-opencv-c-python/`

[4] Learn OpenCV: Face Morph Tutorial
`https://www.learnopencv.com/face-morph-using-opencv-cpp-python/`

[5] Learn OpenCV: Delaunay Triangulation Tutorial
`https://www.learnopencv.com/`
`delaunay-triangulation-and-voronoi-diagram-using-opencv-c-python/`

[6] PyImageSearch: Facial Landmarks Tutorial
`https://www.pyimagesearch.com/2017/04/17/`
`real-time-facial-landmark-detection-opencv-python-dlib/`

[7] OpenCV
`https://opencv.org/`

[8] dlib
`http://dlib.net/`

[9] wxPython
`https://www.wxpython.org/`

[10] PyInstaller
`https://www.pyinstaller.org/`

[11] wxFormBuilder
`https://github.com/wxFormBuilder/wxFormBuilder`

[12] Pillow ImageFilter
`http://pillow.readthedocs.io/en/5.1.x/reference/ImageFilter.html`

[13] FEI Face Database
`http://fei.edu.br/ cet/facedatabase.html`