Daniel van Eijk-Bos, V2A, 14-Feb-2020

Goal:

I will be implementing an edge detection algorithm as part of a facial recognition project. This project will be used in airports to compare people's faces to the picture in their passports. This would aid in shortening waiting times before boarding planes, as the passports wouldn't have to be compared to the people manually by employees. The circumstances in which the facial recognition, and thus the edge detection, will be used are a well-lit environment in which the person being analysed stares straight into the camera from a short distance away.

In the part of the program that will be developed using this implementation plan, I will be programming an algorithm that returns a version of the provided image in which the edges are highlighted. In other projects this image will be used to pinpoint the eyes, mouth and nose of the person and compare these with the face on the passport.

To develop the best edge detection algorithm, I will research different algorithms that are already in use and compare and measure them on the following characteristics: speed, memory efficiency, accuracy and robustness when analysing different facial structures.

Methods:

The more complicated terms mentioned below will be explained further down the document.

There are 2 main methods of detecting edges: *the first order derivative edge detection* and *the second order derivative edge detection*.

These methods are sometimes also named *search-based or Gradient* and *zero-crossing based or Laplacian*, respectively, as their category. When one of these terms is mentioned in this implementation plan, this can also be interpreted as a reference to the other names of this category of algorithms.

The first order derivative edge detection algorithms consist of simple algorithms that first convolve the image with a chosen kernel, then register the intensity of the pixels, and lastly detect peaks within the first derivative of the function of this gradient. These peaks are compared to a set threshold and, if exceeded, the algorithm detects an edge. The edge-pixels are all drawn white, while the rest of the image is black, regardless of the precise intensity of the pixel. Sadly, these methods are very sensitive to noise and produce thicker edges.

The second order derivative edge detection algorithms consist of more sophisticated methods aimed towards automatized edge detection systems; these algorithms are, however, still quite sensitive to noise. The method just mentioned, instead of using the first derivative of intensity within the picture, uses the second derivative. The second derivative is a derivative of the first derivative that was used in the previously mentioned method. When using the second derivative, the algorithm looks for places where the second derivative hits 0 and marks these spots as edges. This however, as mentioned previously, is also very sensitive to noise, which is why these methods are often used in conjunction with smoothing algorithms that remove noise.

Explanations:

Convolving is the process of adding each element of the image to its local neighbours, weighted by the kernel. This kernel varies from algorithm to algorithm based on what the creator thought would work best.

$$\left(\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \right) [2,2] = (i \cdot 1) + (h \cdot 2) + (g \cdot 3) + (f \cdot 4) + (e \cdot 5) + (d \cdot 6) + (c \cdot 7) + (b \cdot 8) + (a \cdot 9).$$

Figure 1: Example of convolving a matrix (or kernel, a-i) with part of an image (1-9)

A derivative is a function that returns how rapidly the output of the desired function changes.
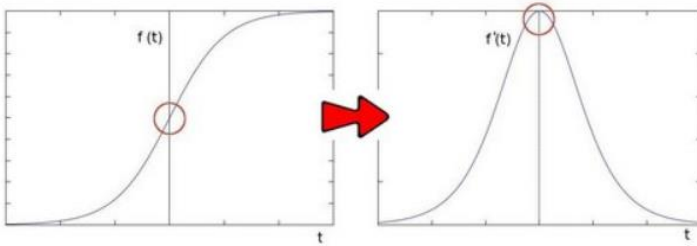
Figure 2: A regular function and its first derivative counterpart

A second derivative would be a function that returns how rapidly the output of the desired derivative function changes.
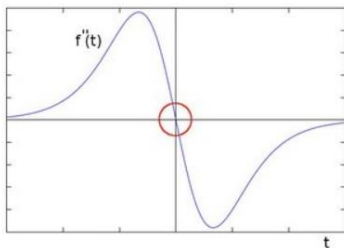

Figure 3: A second derivative, the derivative of the first derivative shown in Figure 2

The following algorithms are candidates for the new edge detection algorithm and thus will be extensively researched until the best fit has been found. The conclusions from the research will be stated below each of the algorithms and summarised further down the document.

Sobel:

The Sobel operator is a first order derivative edge detection algorithm that functions exactly like described above. The kernels used by the Sobel operator consist of a kernel for vertical and a kernel for horizontal iterations through the image. The Sobel kernel provides smoothing and places emphasis on the pixels closest to the pixel currently being convolved. Sobel is one of the most commonly used edge detectors because of the relative inexpensiveness and speed of its computations. This does come at a cost to the quality of the image output, but even then, those outputs are not bad.

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Figure 4: The Kernels (also called matrices or masks) used by Sobel for convolving images as shown in Figure 1

Prewitt:

The Prewitt operator is a first order derivative edge detection algorithm that functions much like the Sobel operator does. As in it detects edges horizontal and vertical edges using different kernels for horizontal/vertical iterations. The only difference between Prewitt and Sobel is that Prewitt doesn't place any emphasis on pixels that are closer to the pixel currently being convolved and doesn't provide smoothing, making it even more sensitive to noise.

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \quad G_y = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

Figure 5: The Kernels used by Prewitt

Laplacian:

The Laplacian operator is a second order derivative edge detection algorithm that works like described above. Because the kernel used by the Laplacian operator approximates the second derivative of the image, the data gained from this is very sensitive to noise. Because of this, the Laplacian operator is often used in conjunction with the Gaussian algorithm to smooth the image out.

It is also possible to convolve the Gaussian kernel with the Laplacian kernel and apply both in one go. If desired, a 5x5 or even larger kernel could be used for better performance. If the combined Gaussian-Laplacian kernel is used instead of applying the kernels one after another, then this algorithm is the fastest of all. In this case it applies only one kernel to the image instead of the 2 that Prewitt and Sobel apply on each pixel. The memory usage is a bit worse than Prewitt and Sobel because of the Gaussian blur that has to be applied, but not by much and the images it creates are so-so.

Figure 6: The Kernels used by Laplacian; the right kernel includes diagonal edges but is more sensitive

Canny:
The Canny algorithm is a much more complex algorithm than the previously mentioned ones and makes use of the first order derivative edge detection. The first step in this algorithm is to smooth out the image using the Gaussian smooth to reduce noise. Afterwards the gradient of the pixels in the image is calculated using either Sobel or Prewitt.

Step 3 is to use these gradients to calculate the edge direction that can be traced using the previously calculated gradients. This is done by comparing the selected pixel with the pixels next to it, if the value of the surrounding pixels is lower than the one currently selected, they are suppressed.

The last step, step 4, is to link the lines broken in the previous step together. This is done by iterating through the non-supressed edge pixels and check their surrounding area for more edges, if they have the same direction as the currently selected edge, horizontal/vertical, then the pixel is marked as an edge. The edges also differentiate between being weak edges and strong edges, to this end, two thresholds need to be selected. If the edge is lower than the lower threshold, the pixel intensity is set to 0, if the edge is higher than the lower threshold but lower than the higher threshold, it's marked as a weak edge, and if it's higher than both of the thresholds, it's marked as a strong edge.
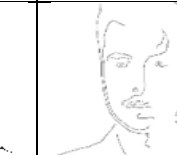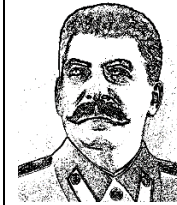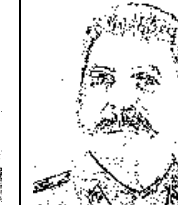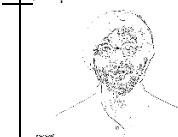
 The strong edges will be displayed in the end image no matter what, but the fate of the weak edges can vary depending on the use of the algorithm. The weak edges can either be completely removed, they can all be kept, or they can all be analysed to check whether they've got a neighbour that's a strong edge, and only be kept if this is true.

This algorithm provides what are probably the second most useful edges for a basic facial recognition algorithm, because it ignores small and useless edges, sadly it also takes a lot longer to analyse images, it uses more memory than the other algorithms and the pictures it outputs do lack details for more advanced facial algorithms.

Difference of Gaussian:

The Difference of Gaussian method of edge detection is very different than the other methods mentioned. This method doesn't use any kernel and doesn't even really search for edges. The Difference of Gaussian method relies on creating two new versions of an image, both created by applying the Gaussian smoothing to them, one a little more than the other. When these two images are created, the algorithm subtracts them from each other leaving behind enhanced edges and other details in the image. This method ignores noise because of the application of the Gaussian smoothing, which means the noise isn't something to worry about anymore. However, by applying the Gaussian smoothing to both images, it doesn't just remove the noise, the operation returns a lower contrast image than the one originally provided. Not only that, but this method is also more costly when it comes to memory efficiency considering the 2 extra images being created, its speed isn't that different from something like Prewitt or Sobel. The blurry returned image won't matter when the input image is of high quality and would in that case most likely be around their level.

	Sobel	Prewitt	Laplace	Canny	Difference of Gaussian
Speed	★★★★	★★★★	★★★★★	★	★★★★
Explanation	2 kernels are decomposed and applied to create 2 new images. These images are combined to create a final image.	Read: Sobel Speed Explanation.	Gaussian and Laplacian kernels are be convolved, decomposed and applied.	Runs a lot of different, complex mini algorithms.	Creates 2 new blurred images and then compares them.
Memory Efficiency	★★★	★★★	★★★★	★	★★★
Explanation	Saves 3 new images when all are stored.	Saves 3 new images when all are stored.	Saves 2 new images when all are stored.	Saves a lot of new images when all are stored, because of the large number of steps.	Saves 3 new images when all are stored.
Accuracy	★★★★	★★★	★★★	★★★★	★★★
Explanation	Ignores noise for the most part and provides a lot of clear lines.	Handles noise poorly but provides some more details than Sobel.	On its own it suffers from a lot of noise, combined with a Gaussian blur this effect is diminished but still provides a so-so image with grainy edges.	Ignores noise. Very minimalistic, highlights only the important features of a face.	Deals very well with noise. Provides a lower quality image, which, doesn't display clear edges.

Robustness	★★★★	★★★	★★	★★★★★	★
Explanation	Deals well with any kind of face.	Deals well with any kind of face but draws a bit darker lines than Sobel.	Deals poorly with faces without clear lines.	Draws clear lines for all important parts of the face regardless of the kind of face.	Depending on the kind of face, this algorithm either displays thin and grainy edges or thick and black lines. The inconsistency is a problem.
Usefulness	★★★★	★★★	★★	★★★★★	★★
Explanation	Shows clear lines but shows too many of them.	Same thing as with Sobel, but with worse noise cancellation.	Doesn't provide clear enough lines in some cases.	Regardless of face type, displays clear lines for all important parts of the face.	Inconsistent clarity of the lines makes this algorithm near useless.
Examples:					

Edge Thinning:
Edge thinning is a technique used to make thick edges thinner. If applied the right way, this algorithm can remove unwanted pixels along edges and turn thick edges into edges merely 1 pixel thick. This would probably improve some of the edge detection results from above. I could not find a way to create such pictures without programming this algorithm myself, because of this I didn't consider this method when judging the algorithms in the table above. Testing with this algorithm could be considered later on in the project.

Gaussian smoothing:
Gaussian smoothing is the process of applying a mask to an image to blur it a little bit. Each time this is done, background noise becomes less and less defined and easier for algorithms to ignore.

Choice:
I've chosen to implement the Canny algorithm. This is because, while it takes a longer time to run than the other algorithms, the results of this algorithm are the best. The longer time it takes to run doesn't even matter that much, it may take longer than the other algorithms but not long enough to diminish the effect of using the algorithm instead of human employees. The best thresholds for the Canny algorithm will be determined by testing different ones later on in the project.
I will also try to test what kind of results will be when implementing the Sobel and Difference of Gaussian together with an edge thinning algorithm. This will only be thoroughly tested if the first few tests show promise.
The Canny algorithm is the most complicated out of the analysed algorithms but shouldn't be impossible for me to program. The tests I plan to run should all be possible to complete within the available timespan, there should also be enough time to apply the results from the tests.

Implementation:

This algorithm will be implemented within the already existing program for facial recognition. The specific place where this will be implemented, is the stepEdgeDetection function from the StudentPreProcessing class.

Activity diagram with that explain how the Canny algorithm will be programmed:
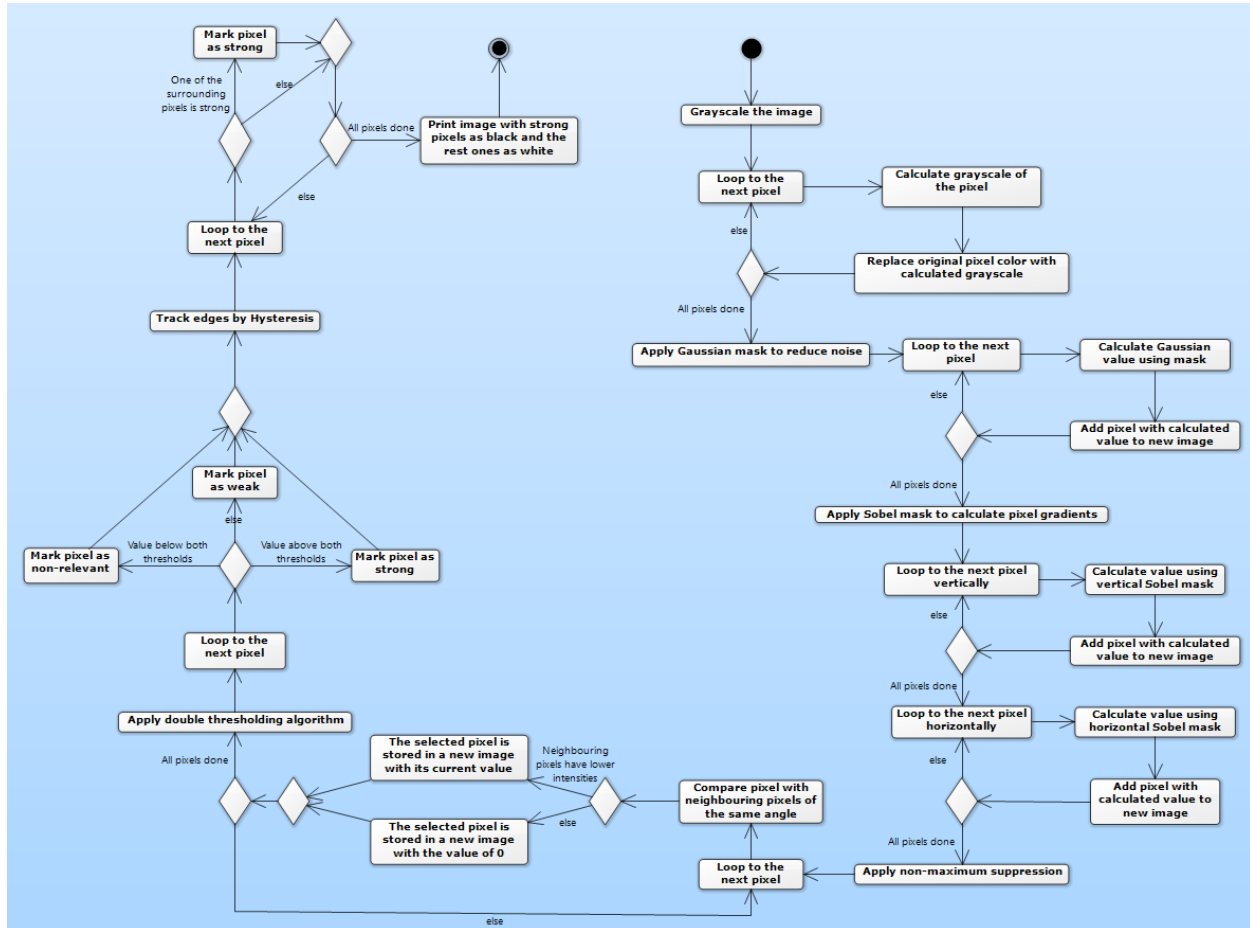
I expect this algorithm to be much more accurate than the existing one, although a bit slower. The tests I am going to run aren't expected to overshadow the results from the Canny algorithm but rather to shed some light on how effective the other algorithms can truly be.

Evaluation:

All to be tested algorithms will be tested for accuracy and robustness. The other traits previously researched, such as speed and memory efficiency, are, while useful in their own right, not very important compared to the accuracy and robustness of the algorithms.

The accuracy and robustness will be tested by running the algorithms using different faces and having someone pick the best results each time. The algorithm with the best results will be chosen.

Personally, I predict the Canny algorithm to come out victorious because of the few but accurate lines it produces.

Sources:

Image editing online:

https://www.diffchecker.com/image-diff

https://pinetools.com/search

http://bigwww.epfl.ch/demo/ip/demos/edgeDetector/

Information:

https://medium.com/@nikatsanka/comparing-edge-detection-methods-638a2919476e

https://en.wikipedia.org/wiki/Difference_of_Gaussians

https://en.wikipedia.org/wiki/Canny_edge_detector

https://www.aishack.in/tutorials/sobel-laplacian-edge-detectors/

https://www.cs.auckland.ac.nz/courses/compsci373s1c/PatricesLectures/Gaussian%20Filtering_1up.pdf

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.402.1860&rep=rep1&type=pdf

https://pdfs.semanticscholar.org/0c00/f88ed5afc9f5d11bd584418088d0451f5afa.pdf

https://pdfs.semanticscholar.org/6bca/fdf33445585966ee6fb3371dd1ce15241a62.pdf

https://en.wikipedia.org/wiki/Sobel_operator

https://en.wikipedia.org/wiki/Kernel_(image_processing)#Convolution

https://www.researchgate.net/post/What_are_the_differences_in_first_order_derivative_edge_detection_algorithms_and_second_order_edge_detection_algorithms

https://docs.opencv.org/3.4/d5/db5/tutorial_laplace_operator.html

https://en.wikipedia.org/wiki/Edge_detection