Face Edge Detection

Visual Coursework-3

Samayak Malhotra M00848138

Importing all the essential libraries

- Flask: A web application framework for Python.
- render_template: Renders templates created using Jinja2 syntax.
- request: An object that contains the incoming request data from the client.
- redirect: A function that redirects the client to a new URL.
- url_for: A function that generates a URL to a specific function.
- PIL: A Python Imaging Library for image processing.
- flash: A message flashing system for Flask.
- numpy: A library for mathematical operations.
- cv2: OpenCV library for image and video processing.
- urllib.request: A library for opening URLs.
- os: A library for interacting with the operating system.
- secure_filename: Function for generating a secure filename.
- FileStorage: A class for working with file uploads.
- werkzeug.utils: A library for HTTP requests and responses.
- werkzeug.datastructures: A library for working with Data structures.

Edge Detection Techniques

Used to identify the boundaries of objects within an image.

Here are some Edge detection techniques implemented in this coursework:

1. **Sobel Filter:** simple and widely used edge detection filter. It uses a small kernel to convolve with the image and compute the gradient in the X and Y directions. The magnitude of the gradient is then used to identify edges in an image.

2. **Prewitt Filter:** similar to the Sobel filter but uses a different kernel to compute the gradients. It is also a popular edge detection filter.

3. **Laplacian Filter:** The Laplacian filter is a second-order derivative filter that is used to identify edges by detecting changes in the second derivative of the image intensity. It can detect edges in any direction, but it is more sensitive to noise than the Sobel and Prewitt filters.

4. **Canny Edge Detector:** The Canny edge detector is a more advanced technique that is widely used in computer vision applications. It involves blurring the image, computing gradients, suppressing non-maximum edges, and thresholding to identify strong edges.

5. **Roberts Cross:** simple edge detection operator that uses two 2x2 kernels to compute the gradient in the X and Y directions. The gradient magnitude is then used to identify edges.

6. **Mean Thresholding:** type of image thresholding technique that is used to convert a grayscale image into a binary image. In this technique, the image is divided into small segments and the threshold value for each segment is calculated based on the mean intensity of the pixels in that segment. This threshold value is then used to convert the pixels in that segment into either black or white based on their intensity values. This process is repeated for all segments of the image to obtain a binary image.

7. **Binary Mask:** A threshold value is chosen and all pixels with intensity values above this threshold are set to 255 (white) and all pixels with intensity values below this threshold are set to 0 (black).

Sobel Filter

The Sobel operator is a spatial filter that calculates the gradient of an image intensity function at every pixel.

In my code, the cv2.Sobel() function was used to apply Sobel edge detection to the grayscale version of the uploaded image. The function takes the following arguments:

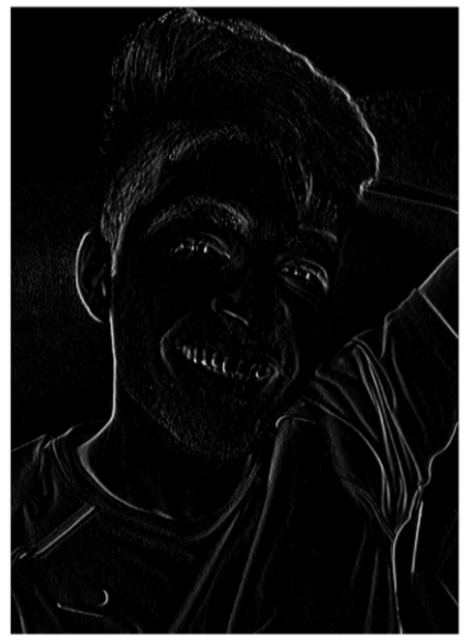
- image: the input image on which to apply the filter
- -1: the depth of the output image; -1 indicates the same depth as the input image
- 1: the order of the derivative in the x direction
- 0: the order of the derivative in the y direction

Later, I stored the output of the Sobel filter in the x direction in the gradients_sobelx variable, while the output of 'y' direction in the gradients_sobely variable. These two outputs are then combined to form the final edge detection output using the cv2.addWeighted() function:

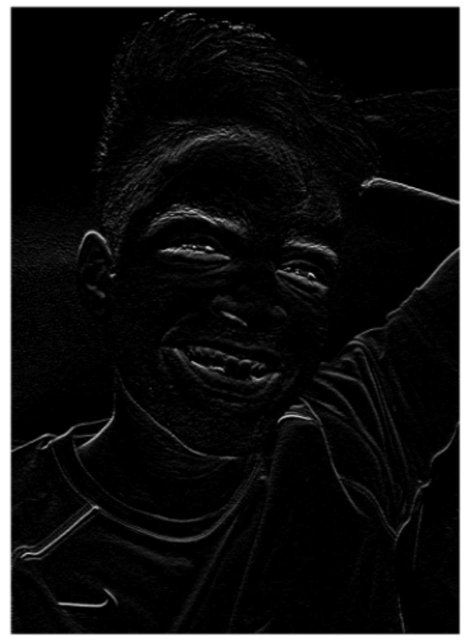
>>gradients_sobelxy = cv2.addWeighted(gradients_sobelx, 0.5, gradients_sobely, 0.5, 0).

The Output is then saved using Cv2.imwrite() function.

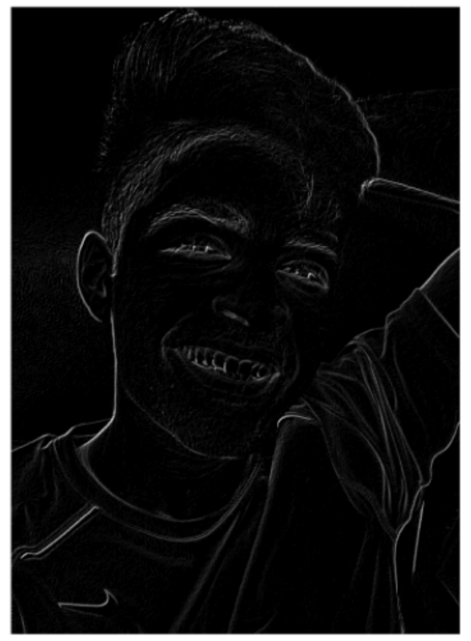
Sobel X Filter



SobelY Transformation



SobelXY Edge Detection



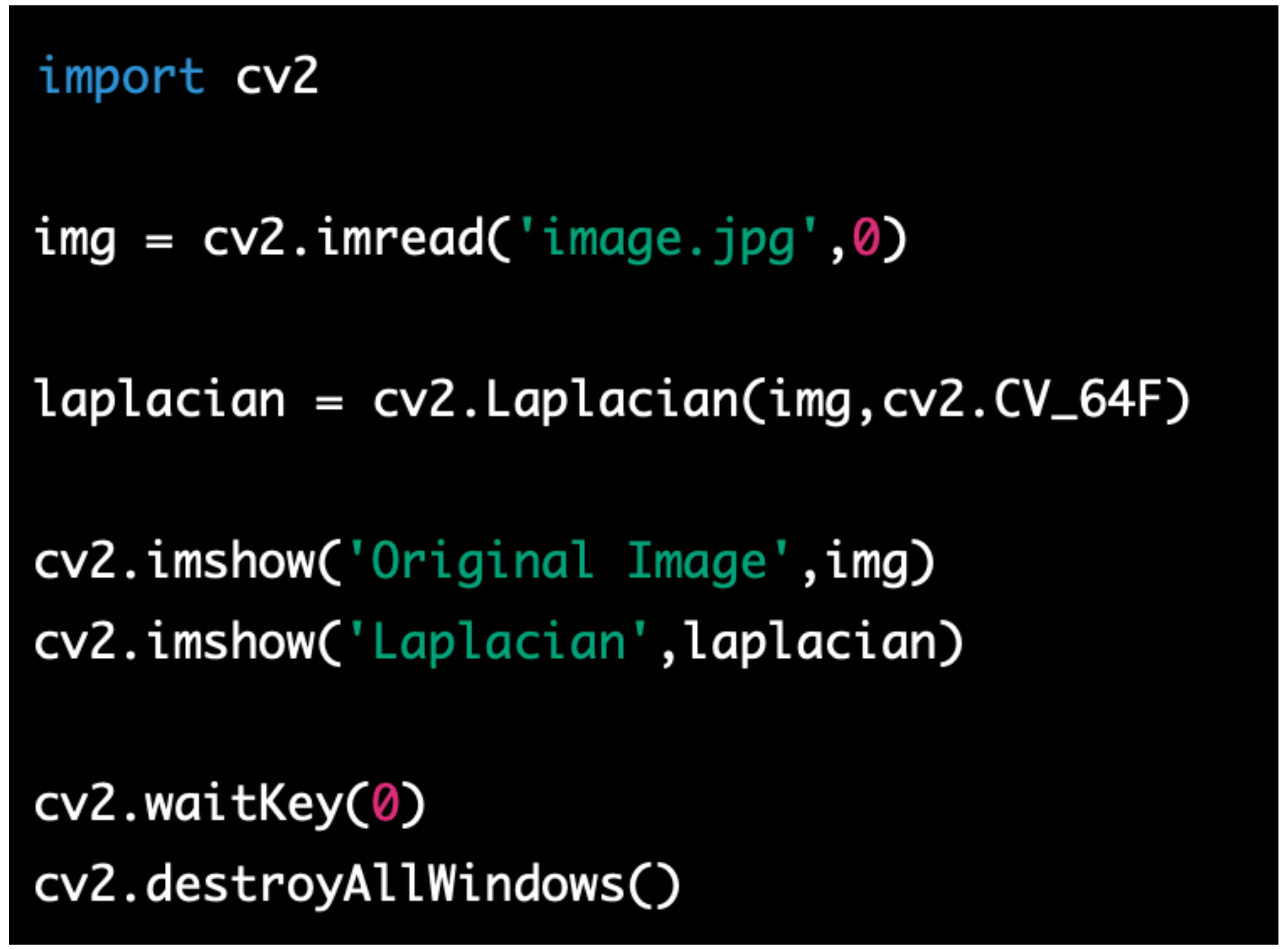
```
image = cv2.imread(os.path.join(app.config['UPLOAD_FOLDER'], filename))
    image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    gradients_sobelx = cv2.Sobel(image, -1, 1, 0)
    gradients_sobely = cv2.Sobel(image, -1, 0, 1)
    gradients_sobelxy = cv2.addWeighted(gradients_sobelx, 0.5, gradients_sobely, 0.5, 0)
```

Laplacian Edge Detection

In this code, I applied the Laplacian edge detection algorithm using the cv2.Laplacian() function, which takes in the input image as ‘image’ and the data type of the output image, which in this case is cv2.CV_64F.

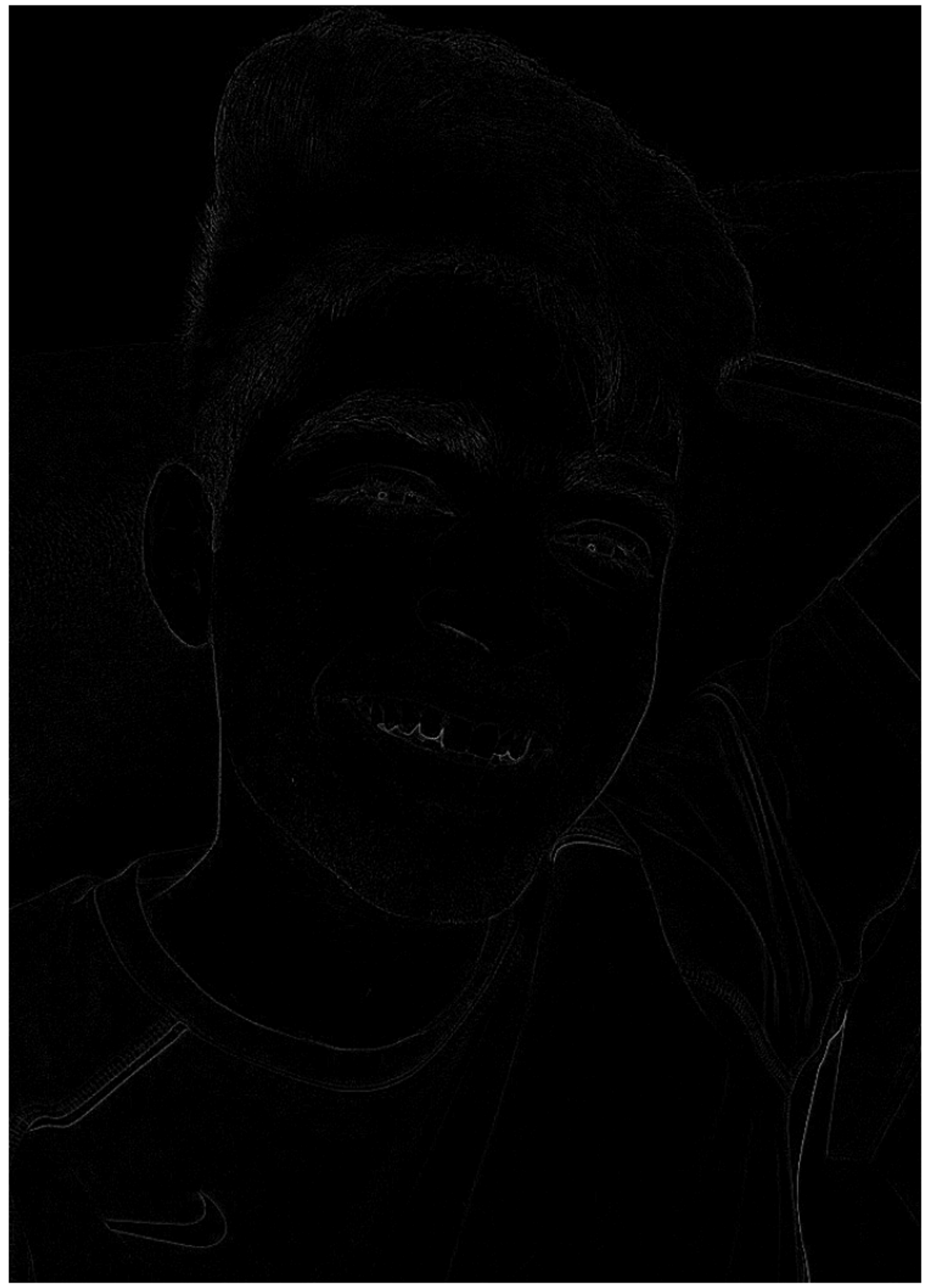
The Laplacian edge detection algorithm calculates the second derivative of the input image, highlights the edges and also detects the zero-crossings.

Finally, I displayed the original image and the Laplacian output using the cv2.imshow() function.



```
import cv2

img = cv2.imread('image.jpg',0)

laplacian = cv2.Laplacian(img,cv2.CV_64F)

cv2.imshow('Original Image',img)
cv2.imshow('Laplacian',laplacian)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

OUTPUT



Canny Edge

The Canny Edge Detection algorithm is used to detect these edges by applying a series of steps to the image:

1. Gaussian Blur: The image is first blurred to reduce noise and smooth out any irregularities in the image.

2. Gradient Calculation: The gradient of the image is then calculated to identify areas of rapid change in intensity.

3. Non-Maximum Suppression: The algorithm then identifies the local maximums in the gradient and suppresses all other values.

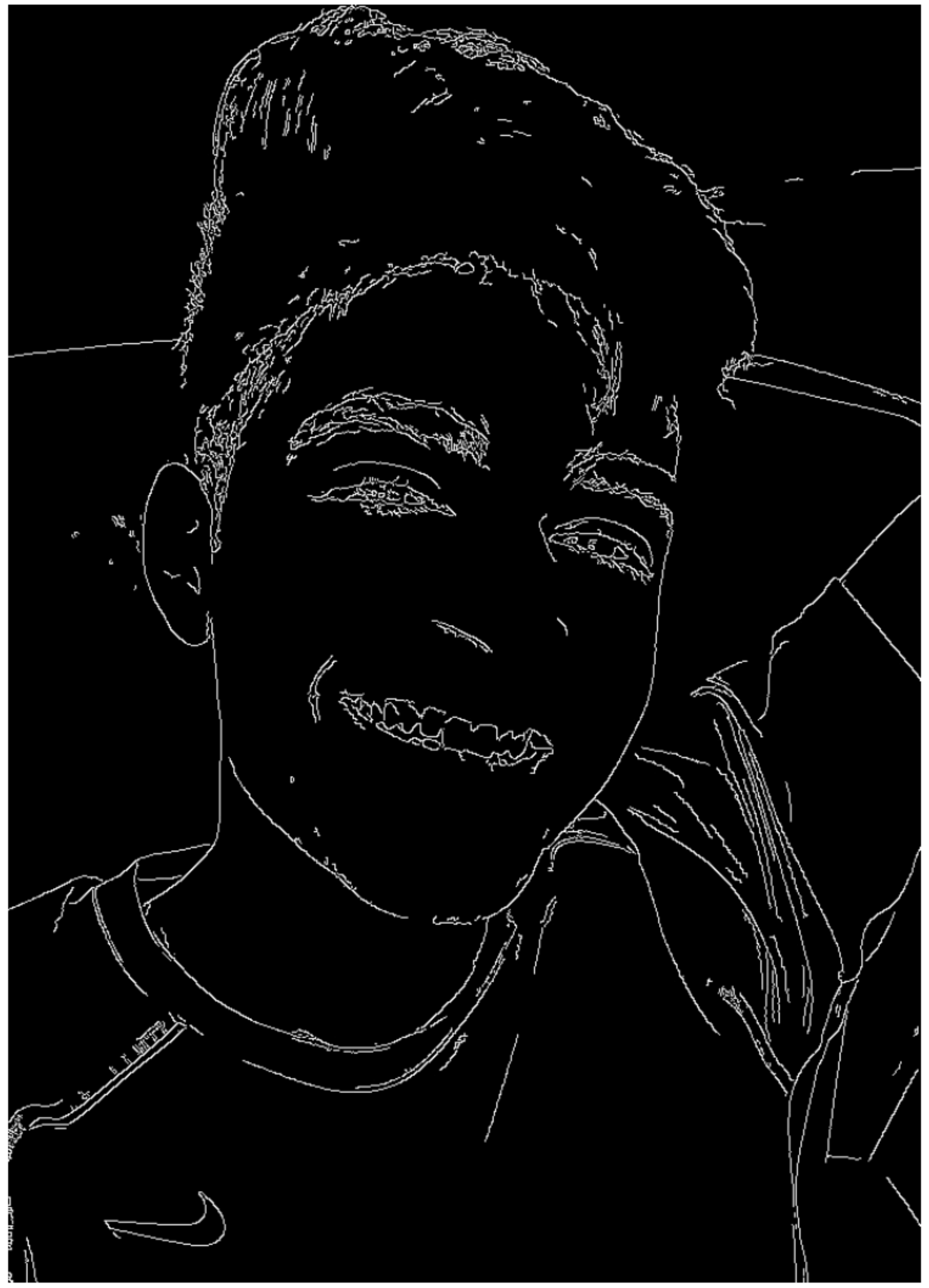
4. Double Thresholding: The image is then thresholded to identify strong edges and weak edges.

5. Edge Tracking: Finally, the algorithm tracks edges by linking strong edges with adjacent weak edges.

The result of this process is a binary image where the edges are highlighted in white and the non-edges are black.

Canny Edge Detection is a useful tool as it can also be used to enhance images, detect features, and perform object recognition tasks.

OUTPUT

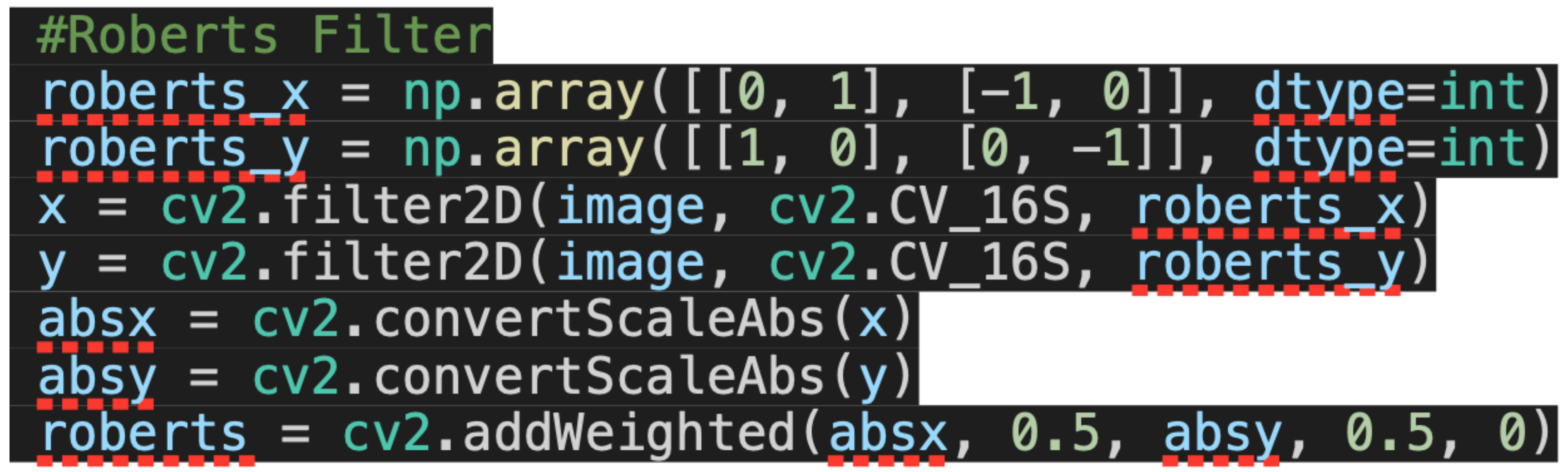


Roberts Cross

An Edge Detection filter

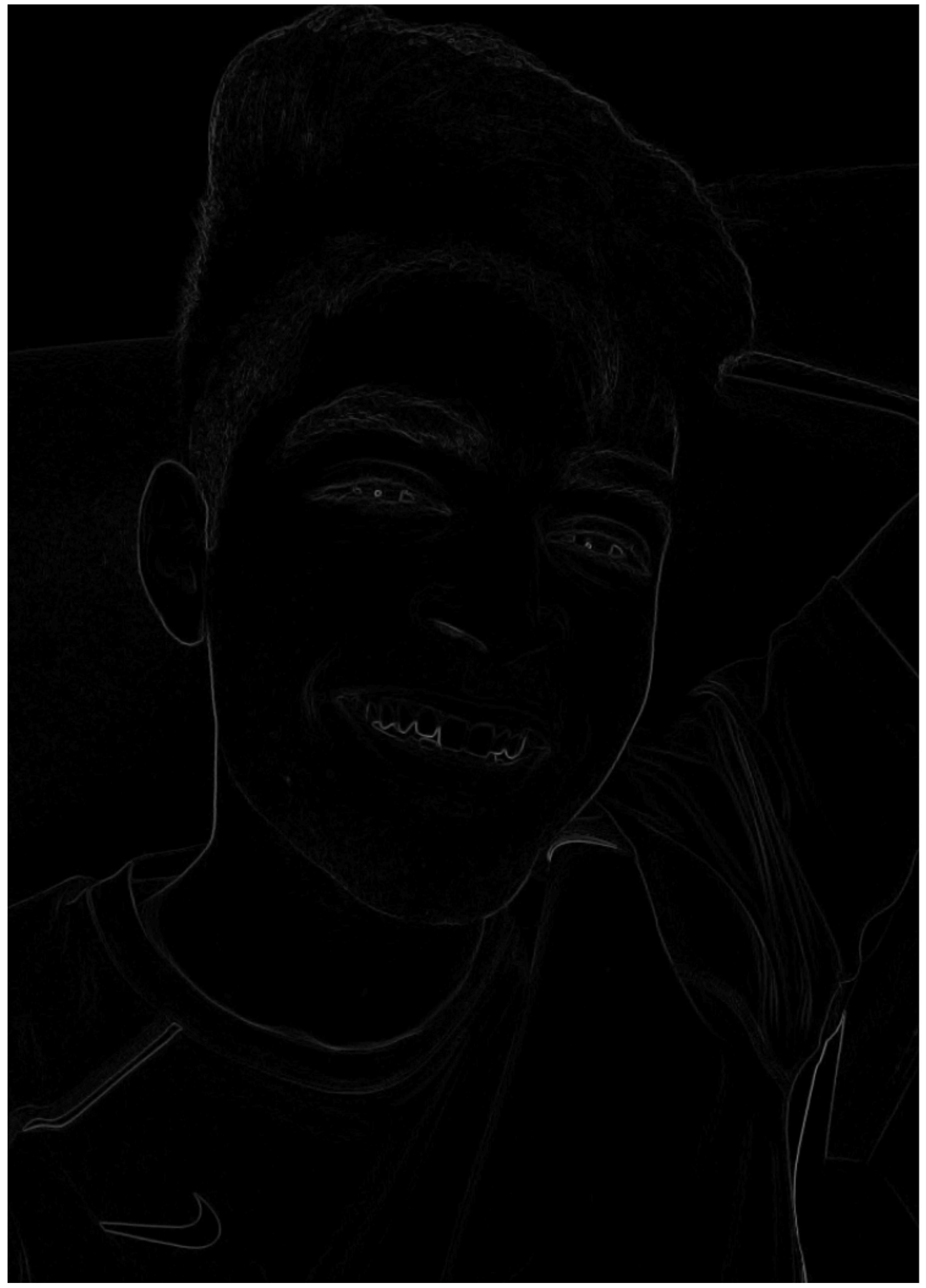
The Roberts cross filter is a simple edge detection filter that is used in image processing to detect edges in an image. It works by approximating the gradient of the image at each pixel using a 2x2 matrix.

Potential issue with this filter: it is sensitive to noise in the image. In noisy images, the filter may produce false edges or miss real edges.



```
#Roberts Filter
roberts_x = np.array([[0, 1], [-1, 0]], dtype=int)
roberts_y = np.array([[1, 0], [0, -1]], dtype=int)
x = cv2.filter2D(image, cv2.CV_16S, roberts_x)
y = cv2.filter2D(image, cv2.CV_16S, roberts_y)
absx = cv2.convertScaleAbs(x)
absy = cv2.convertScaleAbs(y)
roberts = cv2.addWeighted(absx, 0.5, absy, 0.5, 0)
```

OUTPUT

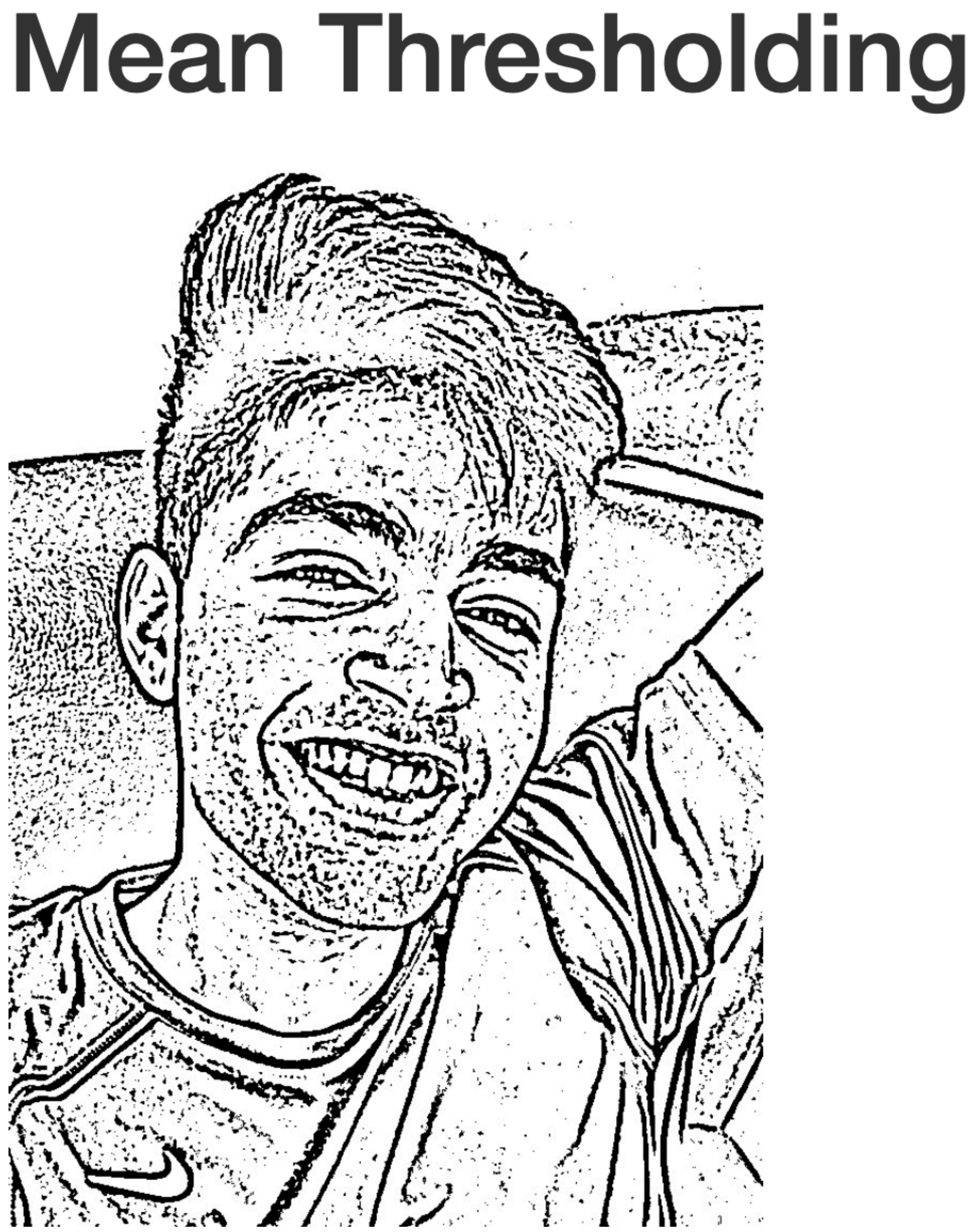


Mean thresholding

Mean thresholding is a commonly used image processing technique for thresholding an image. In Flask, you can use Python's image processing library, Pillow (PIL), to perform mean thresholding on an image.

Mean thresholding works by computing the mean pixel value of the image and using it as a threshold value. All pixels with values below the threshold are set to 0 (black), and all pixels with values above the threshold are set to 255 (white).

```
#Applying Mean Thresholding

mean_threshold = cv2.adaptiveThreshold(blurred, 255,
cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, 11, 2)
```



SVM and K-Means Clustering

Support Vector Machines (SVM) was used in the coursework for edge detection as its useful in classifying pixels in an image as either edge or non-edge pixels.

The SVM model uses a set of features that are indicative of edge pixels, such as changes in intensity or color, to classify pixels in an image. These features are extracted from a neighborhood around each pixel and used as input to the SVM model.

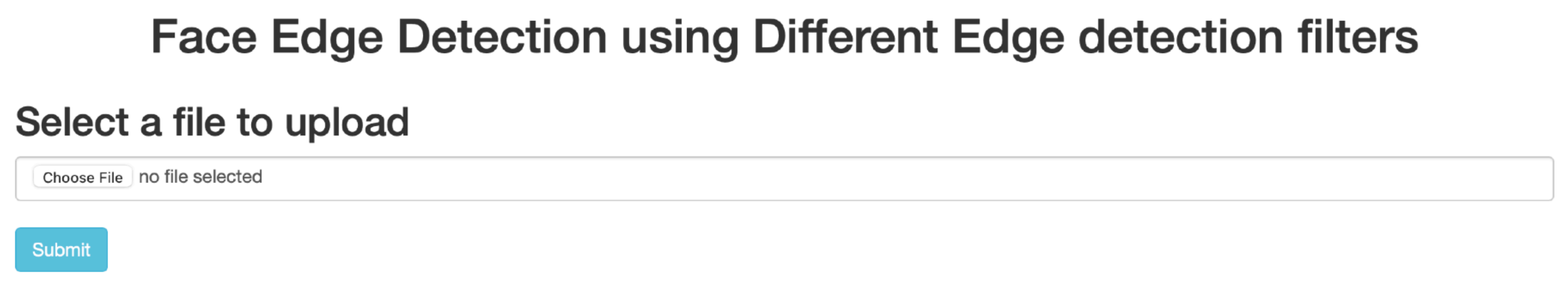
Once the SVM model has classified the pixels in an image as edge or non-edge pixels, the edges can be extracted by applying a threshold to the output of the SVM. Pixels classified as edges are then connected to form a continuous edge.

```
# Define the face segmentation route
@app.route('/segment', methods=['POST'])
def segment_face():
    # Get the uploaded image
    image = cv2.imdecode(np.fromstring(request.files['image'].read(), np.uint8),
cv2.IMREAD_UNCHANGED)
    # Preprocess the image
    features = preprocess_image(image)
    # Use the trained SVM model to predict if the image contains a face or not
    proba = svm_model.predict_proba(features)
    if proba[0][1] > 0.5:
        # If the SVM predicts that the image contains a face, segment the face from the
background
        face_cascade = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
        gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
        faces = face_cascade.detectMultiScale(gray, 1.3, 5)
        for (x, y, w, h) in faces:
            # Draw a rectangle around the face
            cv2.rectangle(image, (x, y), (x + w, y + h), (255, 0, 0), 2)
            # Crop the face region
            face = image[y:y + h, x:x + w]
        # Convert the segmented face to PNG format
        _, encoded_image = cv2.imencode('.png', face)
        output = encoded_image.tobytes()
        return jsonify({'result': 'success', 'image': output})
    else:
        # If the SVM predicts that the image does not contain a face, return an error
message
        return jsonify({'result': 'error', 'message': 'No face detected in the image.'})
```

K-means implementation

```
# Define the face segmentation route
@app.route('/segment', methods=['POST'])
def segment_face():
    # Get the uploaded image
    image = cv2.imdecode(np.fromstring(request.files['image'].read(), np.uint8), 
cv2.IMREAD_UNCHANGED)
    # Preprocess the image
    pixels = preprocess_image(image)
    # Use K-Means Clustering to segment the face from the background
    kmeans = KMeans(n_clusters=2)
    kmeans.fit(pixels)
    labels = kmeans.labels_.reshape(image.shape[:2])
    # Segment the face from the background
    mask = np.uint8(labels == 0)
    face = cv2.bitwise_and(image, image, mask=mask)
    # Convert the segmented face to PNG format
    _, encoded_image = cv2.imencode('.png', face)
    output = encoded_image.tobytes()
    return jsonify({'result': 'success', 'image': output}
```

OUTPUT

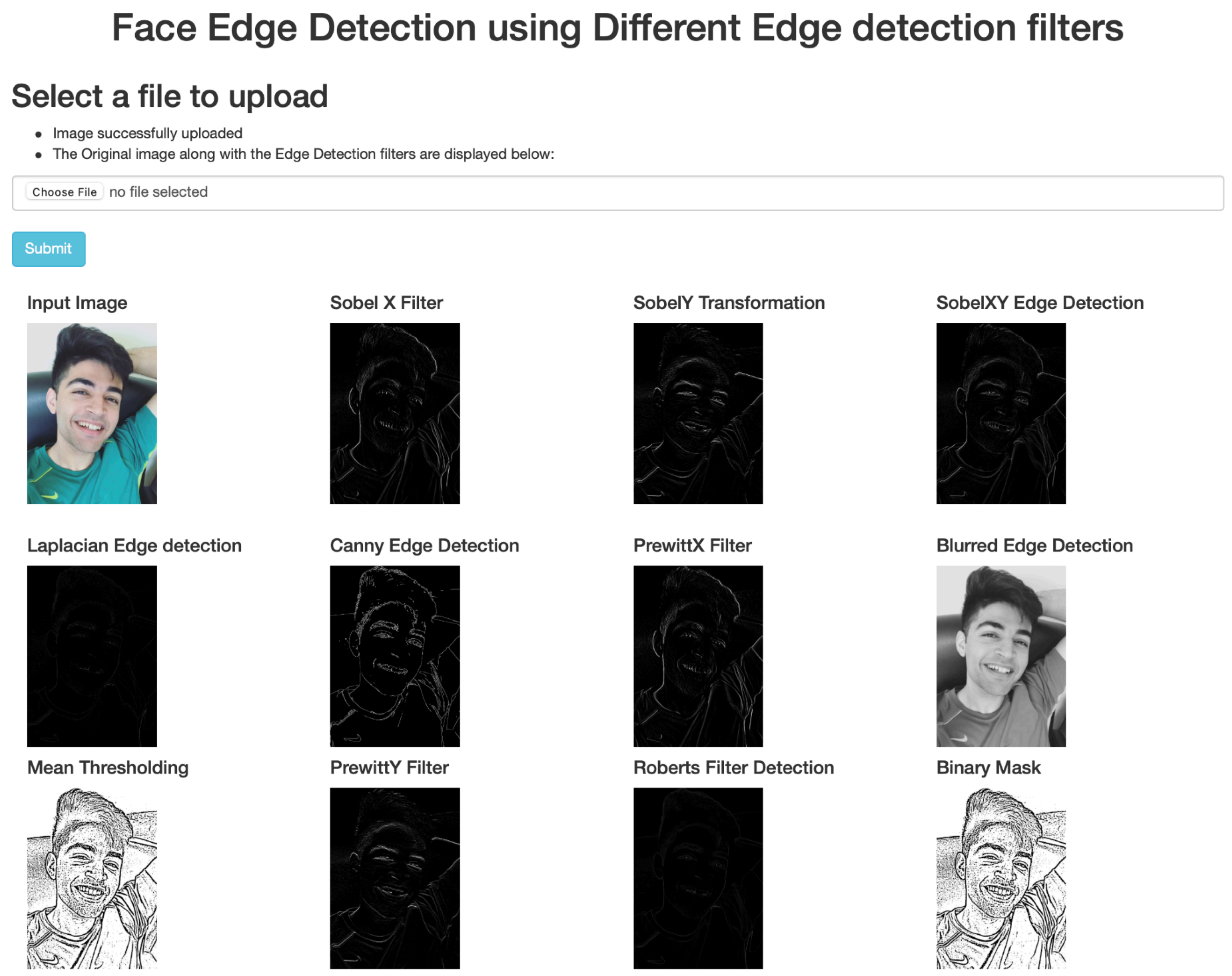


Face Edge Detection using Different Edge detection filters

Select a file to upload

Choose File no file selected

Submit



Face Edge Detection using Different Edge detection filters

Select a file to upload

- Image successfully uploaded
- The Original image along with the Edge Detection filters are displayed below:

Choose File no file selected

Submit

Input Image
Sobel X Filter
SobelY Transformation
SobelXY Edge Detection

Laplacian Edge detection
Canny Edge Detection
PrewittX Filter
Blurred Edge Detection

Mean Thresholding
PrewittY Filter
Roberts Filter Detection
Binary Mask

- User is able to upload any image from the PC in specific formats(jpg, png, gif, jpeg)
- The output is then displayed on the flask web app with the help of an HTML file (index.html).
- Various edge detection filters are applied to the image, the output is generated and displayed in a grid layout

Conclusion

Would consider using Canny edge detection for detecting edges in an image, particularly in the context of face detection. This is because the Canny edge detection algorithm provides high accuracy, low error rates, and robustness to noise (as seen in the results).

Canny algorithm involves smoothing the image with a Gaussian filter, calculating the gradient magnitude and orientation of the image, applying non-maximum suppression to thin out the edges, and finally, applying thresholding to link the remaining edges.

The use of a Gaussian filter in Canny edge detection reduces the impact of noise on the edges, which is particularly important in face detection, where images can contain a lot of noise.

Moreover, Median thresholding allows for more robust edge detection by setting high and low thresholds to ensure that only edges that are part of a continuous contour are detected.

The End