Assignment 2

Status	Completed
Student Name	JURAKUZIEV DADAJON BOYKUZI UGLI
Student Number	201950853
Subject	Advanced Image Processing

1. Problem description

1. Write edge detection routines for Sobel Edge, Prewitt Edge and Canny Edge operators.

2. You need to generate intermediate images for each step to check how your algorithm works.

3. Try images uploaded to class page and your favorite images. For more images, refer

https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/bsds/BSDS300/html/dataset/images.html

You may calculate precision and recall for Berkeley data.
4. Report your experiments with analytic document.

---

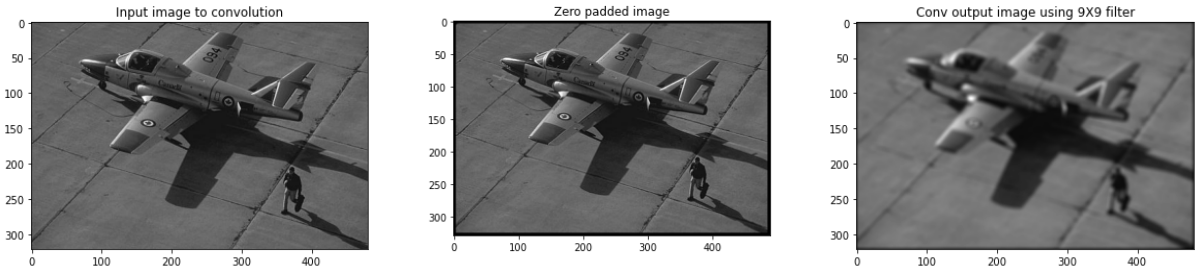
2. Source code

Convolution

```
def convolution(image, kernel, average=False):
    image_row, image_col = image.shape
    kernel_row, kernel_col = kernel.shape

    # zero padding
    output = np.zeros(image.shape)

    # calculate the size of the padding
    pad_height = int((kernel_row - 1) / 2)
    pad_width = int((kernel_col - 1) / 2)

    # create an empty numpy 2D array
    padded_image = np.zeros((image_row + (2 * pad_height), image_col + (2 * pad_width)))

    # copy the image to the proper location, apply padding
    padded_image[pad_height:padded_image.shape[0] - pad_height, pad_width:padded_image.shape[1] - pad_width] = ima
ge

    # convolution operation
    for row in range(image_row):
        for col in range(image_col):
            output[row, col] = np.sum(kernel * padded_image[row:row + kernel_row, col:col + kernel_col])

            # apply the smooth/blur effect
            if average:
                output[row, col] /= kernel.shape[0] * kernel.shape[1]

    return output
```

1. The function above has the `image` and `kernel` as the required parameters and `average` as the 3rd argument. The average argument will be used only for smoothing filter.

2. The output image should have the same dimension as the input image. In order to do so I used zero padding. I calculated the size of the padding using $\frac{k-1}{2}$ formula, where k is kernel size.
3. Then I created an empty 2D numpy array and copied the image to the proper location so that we can have the padding applied in the final output.
4. In order to apply the smooth/blur effect I divided the output pixel by the total number of pixel available in the kernel. This will be done only if the value of `average` is set `True`.
5. We are finally done with our simple convolution function. Here is the output image.



Gaussian blur

```
# calculate the density using the formula of normal distribution.
def dnorm(x, mu, sd):
    return 1 / (np.sqrt(2 * np.pi) * sd) * np.e ** (-np.power((x - mu) / sd, 2) / 2)

# generate Gaussian Kernel
def gaussian_kernel(size, sigma=1, verbose=False):
    kernel_1D = np.linspace(-(size // 2), size // 2, size)
    for i in range(size):
        kernel_1D[i] = dnorm(kernel_1D[i], 0, sigma)
    kernel_2D = np.outer(kernel_1D.T, kernel_1D.T)

    kernel_2D *= 1.0 / kernel_2D.max()

    return kernel_2D

# Apply Gaussian Blur effect to the image
def gaussian_blur(image, kernel_size, verbose=False):
    kernel = gaussian_kernel(kernel_size, sigma=np.sqrt(kernel_size), verbose=verbose)
    return convolution(image, kernel, average=True, verbose=verbose)
```

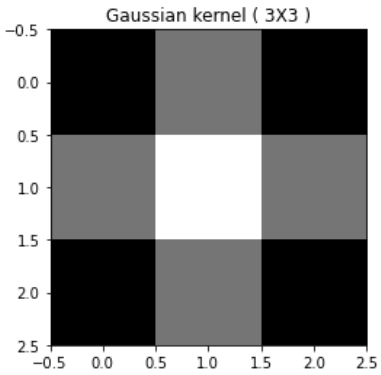
1. Create a function `gaussian_kernel()`. It takes mainly two parameters, the `kernel size` and the `standard deviation` (σ). This function creates a vector of equally spaced number using the size argument passed. When the size = 9, the kernel_1D will be like the following

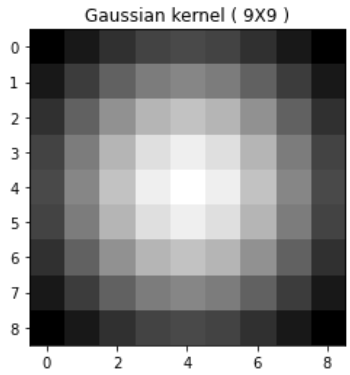
   ```
   [-4. -3. -2. -1.  0.  1.  2.  3.  4.]
   ```

2. Then call the `dnorm()` function which returns the density using the $\mu = 0$ and σ. Just calculated the density using the formula of `Univariate Normal Distribution` : $g(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$

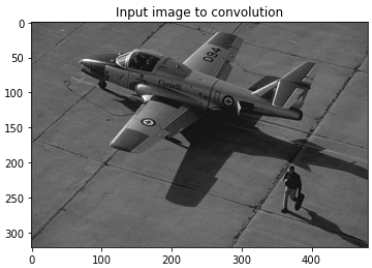
   ```
   [0.05467002 0.08065691 0.10648267 0.12579441 0.13298076 0.12579441
    0.10648267 0.08065691 0.05467002]
   ```

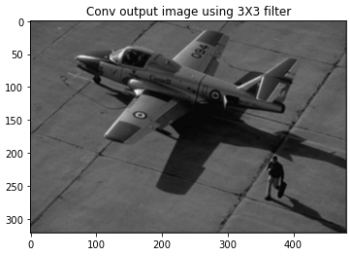
3. In order to set the sigma automatically, I used following equation: $\sigma = \sqrt{kernelsize}$
4. Here is the output of different kernel sizes.

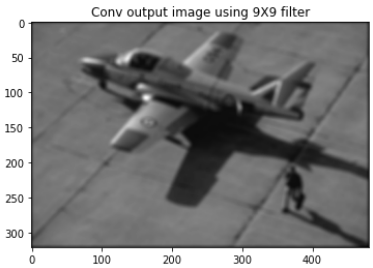




5. Final output images





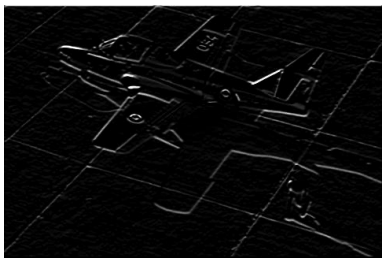


Sobel edge detection

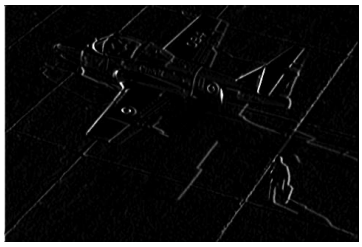
```
def sobel_edge_detection(image, convert_to_degree=False, verbose=False):
    vertical_filter = np.array([[-1, 0, 1],
                                [-2, 0, 2],
                                [-1, 0, 1]])
    # array([[-1, -2, -1],[ 0,  0,  0],[ 1,  2,  1]])
    horizontal_filter = vertical_filter.T

    image_x = convolution(image, vertical_filter, verbose=False)
    image_y = convolution(image, horizontal_filter, verbose=False)
    # calculate gradient magnitude
    # np.hypot() = sqrt(img_x**2 + img_2**2) -> hypotenuse
    G = np.hypot(image_x, image_y)
    G *= 255.0 / G.max()

    # calculate gradient direction
    G_theta = np.arctan2(image_y, image_x)

    if convert_to_degree:
        G_theta = np.rad2deg(G_theta)
        G_theta += 180

    return G, G_theta
```

1. `sobel_edge_detection()` function takes a required parameter, the `gaussian blurred image` and the optional parameters `convert_to_degree` to calculate gradient direction and `verbose` to print intermediate images.
2. I created the vertical mask $\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$ using `numpy` array. The horizontal mask will be derived from vertical mask $\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$
3. Then I called `convolution()` function using the vertical mask. Then apply the convolution using the horizontal mask. In order to combine both the vertical and horizontal edges I used the formula $G =$

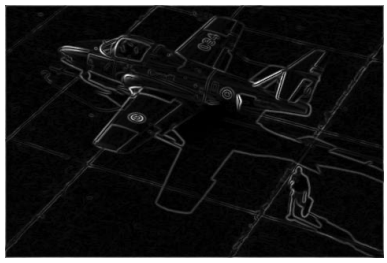
$\sqrt{image_x^2 + image_y^2}$ ⇒ `G = np.hypot(image_x, image_y)` and normalized the output to be between 0 and 255.



Sobel Vertical Edge



Sobel Horizontal Edge



Sobel

Prewitt edge detection

```
def prewitt_edge_detection(image):
    vertical_filter = np.array([[-1, 0, 1],
                                [-1, 0, 1],
                                [-1, 0, 1]])
    # array([[1, 1, 1], [0, 0, 0], [-1, -1, -1]])
    horizontal_filter = np.flip(vertical_filter.T, axis=0)

    prewitt_x = convolution(image, vertical_filter, verbose=False)
    prewitt_y = convolution(image, horizontal_filter, verbose=False)

    # calculate gradient magnitude
    prewitt = np.hypot(prewitt_x, prewitt_y)
    prewitt *= 255.0 / prewitt.max()

    return prewitt_x, prewitt_y, prewitt
```

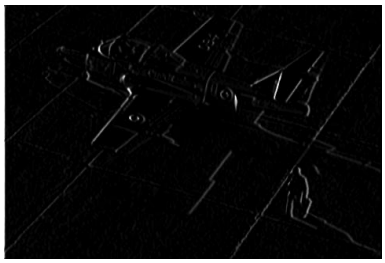
The idea behind Prewitt is the same with Sobel, but it uses different masks.

1. First, create 3D numpy array for vertical mask.

   vertical $\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$ and horizontal $\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$

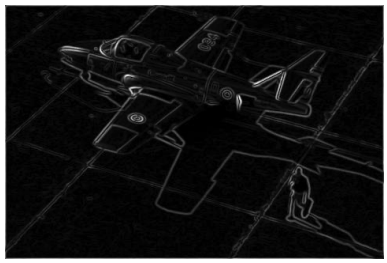
2. Then call `convolution()` function using the vertical mask. Then apply the convolution using the horizontal mask.

   In order to combine vertical and horizontal edges use the formula $G = \sqrt{image_x^2 + image_y^2}$ ⇒ `G = np.hypot(image_x, image_y)` and normalized the output to be between 0 and 255.



Prewitt Horizontal Edge

Prewitt Vertical Edge



Prewitt

Canny edge detection

```
def non_max_suppression(gradient_magnitude, gradient_direction):
    image_row, image_col = gradient_magnitude.shape

    output = np.zeros(gradient_magnitude.shape)

    PI = 180

    for row in range(1, image_row - 1):
        for col in range(1, image_col - 1):
            direction = gradient_direction[row, col]
            # angle 0
            if (0 <= direction < PI / 8) or (15 * PI / 8 <= direction <= 2 * PI):
                before_pixel = gradient_magnitude[row, col - 1]
                after_pixel = gradient_magnitude[row, col + 1]

            # angle 45
            elif (PI / 8 <= direction < 3 * PI / 8) or (9 * PI / 8 <= direction < 11 * PI / 8):
                before_pixel = gradient_magnitude[row + 1, col - 1]
                after_pixel = gradient_magnitude[row - 1, col + 1]

            # angle 90
            elif (3 * PI / 8 <= direction < 5 * PI / 8) or (11 * PI / 8 <= direction < 13 * PI / 8):
                before_pixel = gradient_magnitude[row - 1, col]
                after_pixel = gradient_magnitude[row + 1, col]

            # angle 135
            else:
                before_pixel = gradient_magnitude[row - 1, col - 1]
                after_pixel = gradient_magnitude[row + 1, col + 1]

            if gradient_magnitude[row, col] >= before_pixel and gradient_magnitude[row, col] >= after_pixel:
                output[row, col] = gradient_magnitude[row, col]

    return output


def threshold(image, low, high, weak):
    output = np.zeros(image.shape)

    strong = 255

    strong_row, strong_col = np.where(image >= high)
    weak_row, weak_col = np.where((image <= high) & (image >= low))

    output[strong_row, strong_col] = strong
    output[weak_row, weak_col] = weak

    return output


def hysteresis(image, weak):
    image_row, image_col = image.shape

    top_to_bottom = image.copy()

    for row in range(1, image_row):
        for col in range(1, image_col):
            if top_to_bottom[row, col] == weak:
                if top_to_bottom[row, col + 1] == 255 \
                        or top_to_bottom[row, col - 1] == 255 \
                        or top_to_bottom[row - 1, col] == 255 \
                        or top_to_bottom[row + 1, col] == 255 \
                        or top_to_bottom[row - 1, col - 1] == 255 \
                        or top_to_bottom[row + 1, col - 1] == 255 \
                        or top_to_bottom[row - 1, col + 1] == 255 \
                        or top_to_bottom[row + 1, col + 1] == 255:
                    top_to_bottom[row, col] = 255
                else:
                    top_to_bottom[row, col] = 0

    bottom_to_top = image.copy()

    for row in range(image_row - 1, 0, -1):
        for col in range(image_col - 1, 0, -1):
            if bottom_to_top[row, col] == weak:
                if bottom_to_top[row, col + 1] == 255 \
                        or bottom_to_top[row, col - 1] == 255 \
                        or bottom_to_top[row - 1, col] == 255 \
                        or bottom_to_top[row + 1, col] == 255 \
```

```
                        or bottom_to_top[row - 1, col - 1] == 255 \
                        or bottom_to_top[row + 1, col - 1] == 255 \
                        or bottom_to_top[row - 1, col + 1] == 255 \
                        or bottom_to_top[row + 1, col + 1] == 255:
                    bottom_to_top[row, col] = 255
                else:
                    bottom_to_top[row, col] = 0

    right_to_left = image.copy()

    for row in range(1, image_row):
        for col in range(image_col - 1, 0, -1):
            if right_to_left[row, col] == weak:
                if right_to_left[row, col + 1] == 255 \
                        or right_to_left[row, col - 1] == 255 \
                        or right_to_left[row - 1, col] == 255 \
                        or right_to_left[row + 1, col] == 255 \
                        or right_to_left[row - 1, col - 1] == 255 \
                        or right_to_left[row + 1, col - 1] == 255 \
                        or right_to_left[row - 1, col + 1] == 255 \
                        or right_to_left[row + 1, col + 1] == 255:
                    right_to_left[row, col] = 255
                else:
                    right_to_left[row, col] = 0

    left_to_right = image.copy()

    for row in range(image_row - 1, 0, -1):
        for col in range(1, image_col):
            if left_to_right[row, col] == weak:
                if left_to_right[row, col + 1] == 255 \
                        or left_to_right[row, col - 1] == 255 \
                        or left_to_right[row - 1, col] == 255 \
                        or left_to_right[row + 1, col] == 255 \
                        or left_to_right[row - 1, col - 1] == 255 \
                        or left_to_right[row + 1, col - 1] == 255 \
                        or left_to_right[row - 1, col + 1] == 255 \
                        or left_to_right[row + 1, col + 1] == 255:
                    left_to_right[row, col] = 255
                else:
                    left_to_right[row, col] = 0

    final_image = top_to_bottom + bottom_to_top + right_to_left + left_to_right

    final_image[final_image > 255] = 255

    return final_image


def canny_edge_detection(image):
    G, G_theta = sobel_edge_detection(image, convert_to_degree=True)
    nms_img = non_max_suppression(G, G_theta)
    threshold_img = threshold(nms_img, 50, 80, weak=400)
    canny_img = hysteresis(threshold_img, 400)

    return nms_img, threshold_img, canny_img
```

The Canny edge detection algorithm has 5 steps:

1. Noise reduction

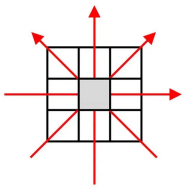
   I created a 1D Gaussian mask by applying `gaussian_blur()` function with `kernel_size=1` and $\sigma > 0$ to the input image.

2. Gradient calculation

   I used `sobel_edge_detection()` function to create a 1D mask for the first derivative of the Gaussian in the x and y directions and then convolve input image `img` with that mask along the rows and columns. This function returns gradient magnitude $G = \sqrt{image_x^2 + image_y^2}$ and gradient direction $G_\theta = \tan^{-1}(\frac{image_x}{image_y})$. The value of gradient direction is in radian. Thus I converted it to degree using numpy library `rad2deg()`. This returns degree between `-180 to 180`, which I converted from 0 to 360 by adding 180 to gradient direction.

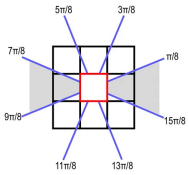
   Finally `sobel_edge_detection()` function returned both the magnitude and direction of gradient.

3. Non-maximum suppression

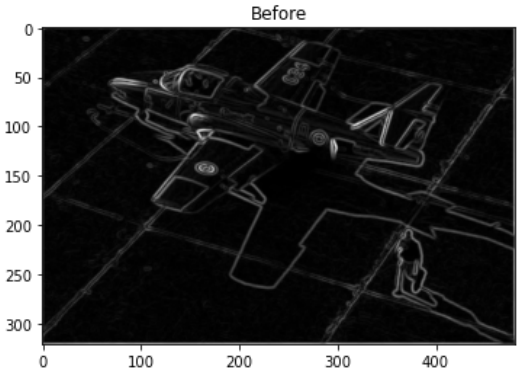


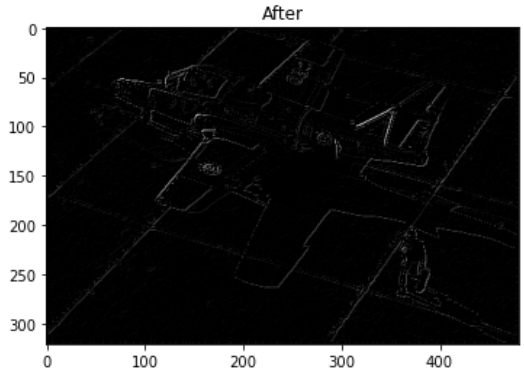
The most important step in Canny algorithm is `non_max_suppression()` function, which takes `gradient magnitude` and `gradient direction` as input parameters.

A pixel can have total 4 directions for the gradient (shown to the right) since there are total 8 neighboring pixels.



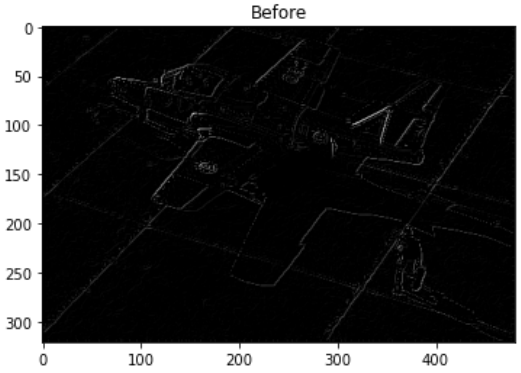
I looped through all the pixels in the gradient directions. Then based on the value of gradient direction I stored the gradient magnitude of the two neighboring pixel. At the end we I found out whether the selected/middle pixel has the highest gradient magnitude or not. If not I continued with the loop, otherwise update the output image for the given row and col with the value of the gradient magnitude.

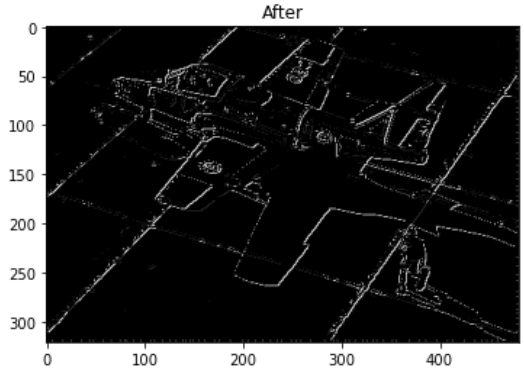




4. Double threshold

`threshold()` function if the value of any pixel is higher than the high value, then I set it to 255. I assumed these are proper edges. Next if the pixels are between low and high value then I set them to week value (passed as an argument). Remaining pixels will all the 0.

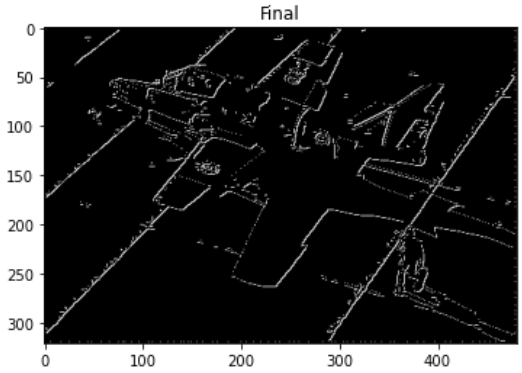




5. Edge Tracking by Hysteresis

I loop through each pixel in the image, if the value of the pixel is weak (only for weak pixels) and verify whether there are any neighboring pixel with value 255. If not then set the value of the pixel to 0. It is necessary to also scan the image from bottom-right to top-left corner, which will help to detect the right part of the edge. I did this total 4 times from all corners.

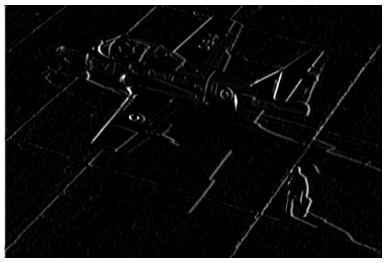
Sum all the pixels to create our final image. The white pixels will add up, hence to make sure there is no pixel value greater than 255, threshold them to 255.



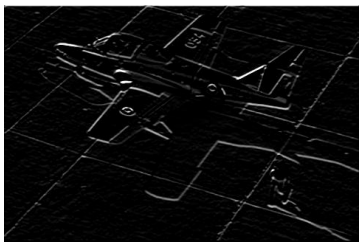
3. Results and Analysis

You can see all results in `output` folder of my project. I submitted it as `zip` file. Just unzip it and you can find source code, input and output images. Here are some results:

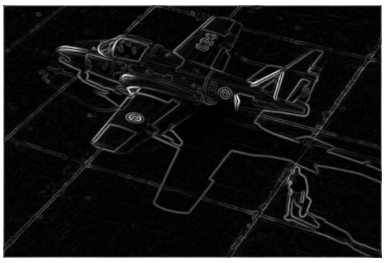
Sobel → plane.jpg



Sobel X

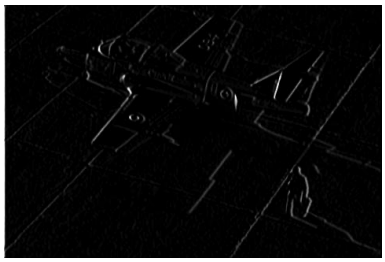


Sobel Y



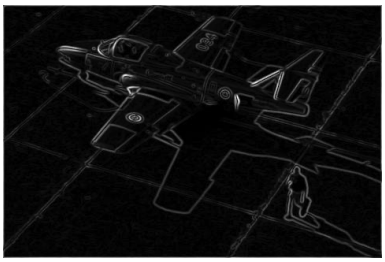
Sobel

Prewitt → plane.jpg



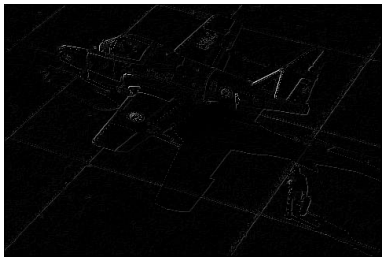
Prewitt X

Prewitt Y

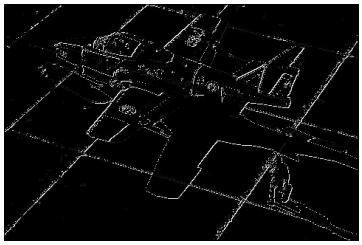


Prewitt

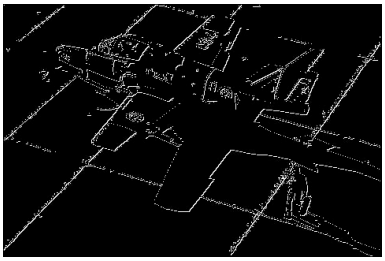
Canny → plane.jpg → kernel_size = 1, $\sigma = \sqrt{3}$



Non Maximum Suppression



Threshold



Canny

Sobel → zebra.jpg

Sobel X

Sobel Y

Sobel

Prewitt → zebra.jpg

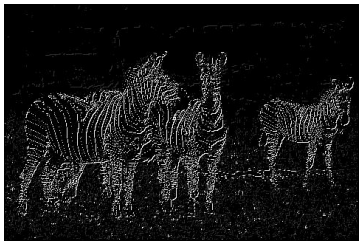
Prewitt X

Prewitt Y

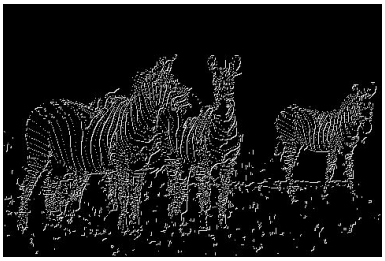
Prewitt

Canny → zebra.jpg → kernel_size = 1, σ=$\sqrt{3}$

Non Maximum Suppression



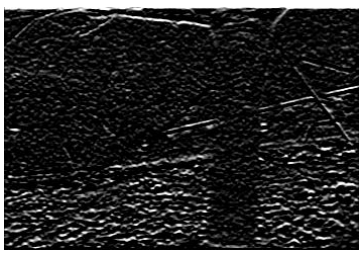
Threshold



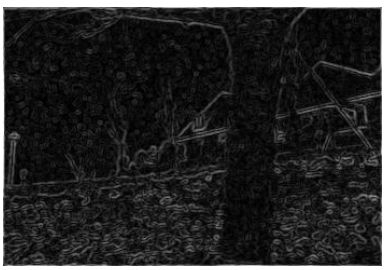
Canny

Sobel → Noisyimage1.jpg → kernel_size = 3, σ=$\sqrt{3}$

Sobel X

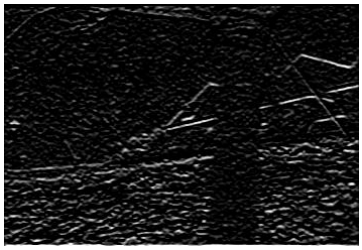


Sobel Y



Prewitt → Noisyimage1.jpg → kernel_size = 3, σ=$\sqrt{3}$

Prewitt X

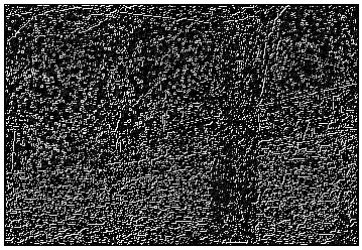


Prewitt Y

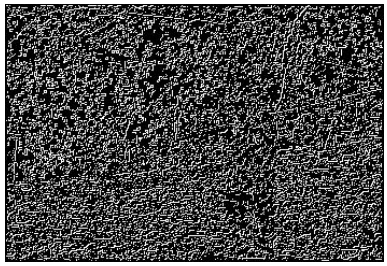
Prewitt

Canny → Noisyimage1.jpg → kernel_size = 1, σ=$\sqrt{3}$

Non Maximum Suppression



Threshold



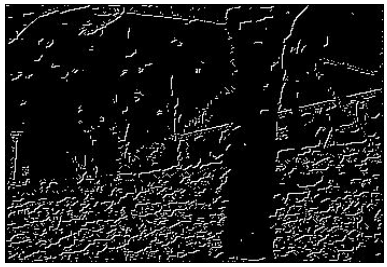
Canny

As you can see, when I used 1D Gaussian kernel with $\sigma = \sqrt{3}$ `canny_edge_detection()` performed really bad. You can see all noises in the image. So I tried 3D Gaussian kernel.

Canny → Noisyimage1.jpg → kernel_size = 3, σ=$\sqrt{3}$

Non Maximum Suppression

Threshold



Canny

Canny → Noisyimage1.jpg → kernel_size = 5, σ=$\sqrt{3}$, losing some lines

Non Maximum Suppression

Threshold

Canny

Sobel → face.jpg

Sobel X

Sobel Y

Sobel

Prewitt → face.jpg

Prewitt X

Prewitt Y

Prewitt

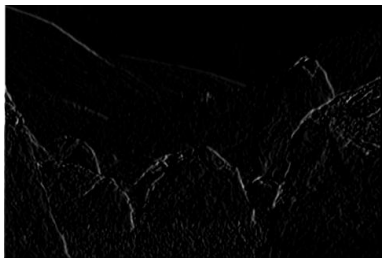
Canny → face.jpg → kernel_size = 1, $\sigma = \sqrt{3}$

Non Maximum Suppression

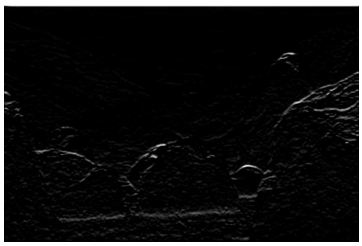
Threshold

Canny

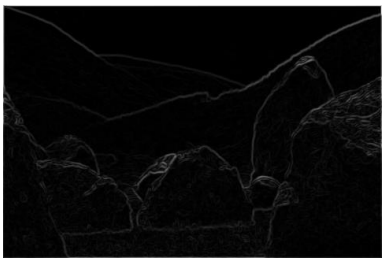
Sobel → mountain.jpg



Sobel X

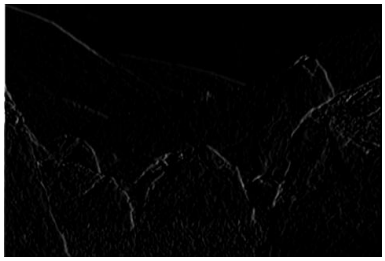


Sobel Y

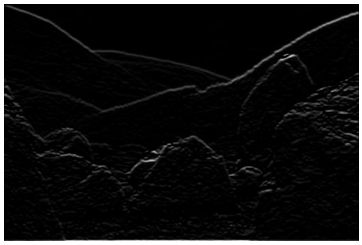


Sobel

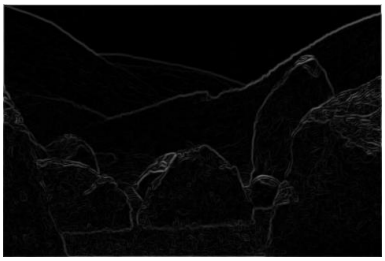
Prewitt → mountain.jpg



Prewitt X



Prewitt Y



Prewitt

Canny → mountain.jpg -> kernel_size = 1, $\sigma=\sqrt{3}$

Non Maximum Suppression Threshold Canny

Canny → mountain.jpg -> kernel_size = 3, $\sigma=\sqrt{3}$, low = 5, high=20, weak=50

Non Maximum Suppression

Threshold

Canny

4. References

1. OpenCV

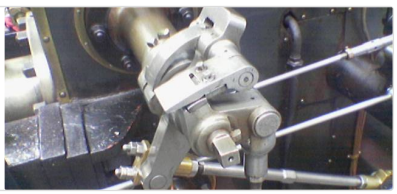
   OpenCV: Feature Detection
   Finds lines in a set of points using the standard Hough transform. The function finds lines in a set of points using a modification of the Hough transform. thetaMin, thetaMax, thetaStep);
   https://docs.opencv.org/4.4.0/dd/d1a/group__imgproc__feature.html#ga04723e007ed888ddf11d9ba04e2232de

2. Abhisek Jana's blog post

   Applying Gaussian Smoothing to an Image using Python from scratch - A Developer Diary
   Using Gaussian filter/kernel to smooth/blur an image is a very important tool in Computer Vision. You will find many algorithms using it before actually processing the image. Today we will be Applying Gaussian Smoothing to an image using Python from scratch and not using library like OpenCV.
   http://www.adeveloperdiary.com/data-science/computer-vision/applying-gaussian-smoothing-to-an-image-using-python-from-scratch/

3. Sobel operator → Wikipedia

   Sobel operator
   The Sobel operator, sometimes called the Sobel-Feldman operator or Sobel filter, is used in image processing and computer vision, particularly within edge detection algorithms where it creates an image emphasising edges. It is named after Irwin Sobel and Gary
   https://en.wikipedia.org/wiki/Sobel_operator



4. Prewitt operator → Wikipedia

   Prewitt operator
   The Prewitt operator is used in image processing, particularly within edge detection algorithms. Technically, it is a discrete differentiation operator, computing an approximation of the gradient of the image intensity function. At each point in the image,
   https://en.wikipedia.org/wiki/Prewitt_operator

5. Justin Liang's blog post

Canny Edge Detector

Convert the image to grayscale. In MATLAB the intensity values of the pixels are 8 bit and range from 0 to 255. Original Black and White Perform a Gaussian blur on the image. The blur removes some of the noise before further processing the image.

http://justin-liang.com/tutorials/canny/