Canny Edge Detection

Osama Al-Wardi

March 28, 2019

Table of Contents

1 Introduction

This is a coding challenge given to me by FARO to assess my skills and my problem solving abilities. I had the choice of using many programming languages such as (C/C++, python 3.x.x or Java/Javascript). I decided to work with python for several reasons:

- I personally prefer it when dealing with time limited projects.
- There's also a large number of online resources that are available.
- This implemetation can be easily modified to support multiple systems and platforms.
- Python has many GUI modules that are very easy to use.

I have to also mention that I used a variety of online resources to get to learn how to implement the algorithm and create the user interface. All the links are listed in the references below.

2 Project Requirements

The challenge can be put into five steps:

Step 1) The program should have a GUI where it is possible for the user, to choose a video from a folder and import it, also choose the folder to export the processed video.

Step 2) The program should convert the video frames to grayscale.

Step 3) The program should do a Canny Edge Detection on each frame.

Step 4) The computed frame should be put together to a stream and be saved as a video to the chosen folder.

Optional: Step 5) A real time edge detection and play the original and processed video, side by side on your GUI.

3 Environment Setup & Libraries Used

I created this project and tested it in a Ubuntu linux environment and this code can work on any Unix like environment given that all libraries and dependencies are present. Modifications to the code have to be done in order to run it in Windows environment.

The libraries used are listed below.

- opencv2 → used to handles frame operations
- numpy → used for matrix operations
- scipy → used for the convolution operation
- subprocess → used to handels shell commands
- tkinter → used for the graphical user interface

All the links for installing these libraries are listed in the references. Another tool that is used is ffmpeg. It is used for assembling the computed frames back into a stream. You can install it using this command.

```
me@The-Hijacker:~$ sudo apt-get install ffmpeg
```

After all the libraries are installed now you can run the program. You need to be in the src directory and run ui.py.

```
me@The-Hijacker:~/Desktop/canny/src/ python3 ui.py
```

4 Project Structure

I logically seperated my code into 3 files:

- canny.py → runs a canny edge detection on an image object.
- framer.py → creates a working enviromnent, extracts frames, assembles the computed frames and cleans up after.
- ui.py → handels all user interactions and provides buttons, status bar and progress bar.

These files are under the src directory. There are other directories such as doc where this file lies and maybe later a testing directory ..etc.

5 Code Documentation

5.1 Algorithm Implementation

5.1.1 Preprocessing

To start we need to remove random noise. One way to get rid of noise on an image, is by applying a "Gaussian Blur". This is an image convolution technique that is applied with a Gaussian Kernel of size 5x5. The kernel size results in a different blurring effect. I used a 5x5 Gaussian filter as it was mentioned in tutorial that I was given. Below is the formula I followed.

$$H_{ij} = \frac{1}{2\pi\sigma} exp(-\frac{(i-(k+1))^2 + (j-(k+1))^2}{2\sigma^2})$$

```
# STEP1 :: Perprocessing :: Applying Gaussian Blur
def gaussian_kernel(size, sigma=1):
    size = int(size) // 2
    x, y = np.mgrid[-size:size + 1, -size:size + 1]
    normal = 1 / (2.0 * np.pi * sigma**2)
    g = np.exp(-((x**2 + y**2) / (2.0 * sigma**2))) * normal
    return g
```

5.1.2 Calculating Gradients

The Gradient calculation step detects the edge intensity and direction by calculating the gradient of the image using edge detection operators. Edges correspond to a change of pixels intensity. To detect it, the easiest way is to apply filters that highlight this intensity change in both directions: horizontal (x) and vertical (y).

```
#STEP2 :: Calculating Gradients :: Sobel Edge Detector
def sobel_filters(img):
    Kx = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], np.float32)
    Ky = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]], np.float32)
    Ix = ndimage.filters.convolve(img, Kx)
    Iy = ndimage.filters.convolve(img, Ky)
    G = np.hypot(Ix, Iy)
    G = G / G.max() * 255
    theta = np.arctan2(Iy, Ix)
    return (G, theta)
```

5.1.3 Non-Maximum Suppression

We must perform non-maximum suppression to thin out the edges. So If a pixel is not a maximum, it is suppressed. To do this, we iterate over all the points on the gradient intensity matrix and find the pixels with the maximum value in the edge directions.

```
#STEP3 :: Non-Maximum Suppression :: Reducing the Edges
def non_max_suppression(img, D):
    M, N = img.shape
    Z = np.zeros((M,N), dtype=np.int32)
    angle = D * 180. / np.pi
    angle[angle < 0] += 180
    for i in range(1,M-1):
        for j in range(1,N-1):
            try:
                q = 255
                r = 255
               #angle 0
                if (0 <= angle[i,j] < 22.5) or (157.5 <= angle[i,j] <=
                    180):
                    q = img[i, j+1]
                    r = img[i, j-1]
                #angle 45
                elif (22.5 <= angle[i,j] < 67.5):
                    q = img[i+1, j-1]
                    r = img[i-1, j+1]
                #angle 90
                elif (67.5 <= angle[i,j] < 112.5):
                    q = img[i+1, j]
                    r = img[i-1, j]
                #angle 135
                elif (112.5 <= angle[i,j] < 157.5):
                    q = img[i-1, j-1]
                    r = img[i+1, j+1]
                if (img[i,j] >= q) and (img[i,j] >= r):
                    Z[i,j] = img[i,j]
                else:
                    Z[i,j] = 0
            except IndexError as e:
                pass
    return Z
```

5.1.4 Thresholding

We must identify strong, weak, and non-relevant pixels. Strong pixels are pixels that have an intensity so high that we are sure they contribute to the final edge. Weak pixels are pixels that have an intensity value that is not enough to be considered as strong ones, but yet not small enough to be considered as non-relevant for the edge detection. Other pixels are considered as non-relevant for the edge.

In general we need to set a high threshold that is used to identify the strong pixels and a low threshold is used to identify the non-relevant pixels. In my

implementation I used 0.05 as a low threshold and 0.15 as a high threshold.

```
#STEP4 :: Thresholding :: Identifying Pixels
def threshold(img, lowThresholdRatio=0.05, highThresholdRatio=0.15):
    highThreshold = img.max() * highThresholdRatio;
    lowThreshold = highThreshold * lowThresholdRatio;
    M, N = img.shape
    res = np.zeros((M,N), dtype=np.int32)
    weak = np.int32(25)
    strong = np.int32(255)
    strong_i, strong_j = np.where(img >= highThreshold)
    zeros_i, zeros_j = np.where(img < lowThreshold)
    weak_i, weak_j = np.where((img <= highThreshold) & (img >=
        lowThreshold))
    res[strong_i, strong_j] = strong
    res[weak_i, weak_j] = weak
    return (res, weak, strong)
```

5.1.5 Hysteresis

Based on the threshold results, the hysteresis consists of transforming weak pixels into strong ones, if and only if at least one of the pixels around the one being processed is a strong one.

```
def hysteresis(img, weak=75, strong=255):
    M, N = img.shape
    for i in range(1, M-1):
        for j in range(1, N-1):
            if (img[i,j] == weak):
                try:
                    if ((img[i+1, j-1] == strong) or (img[i+1, j] ==
                         strong) or (img[i+1, j+1] == strong)
                        or (img[i, j-1] == strong) or (img[i, j+1] ==
                             strong)
                        or (img[i-1, j-1] == strong) or (img[i-1, j] ==
                             strong) or (img[i-1, j+1] == strong)):
                        img[i, j] = strong
                    else:
                        img[i, j] = 0
                except IndexError as e:
                    pass
    return img
```

5.2 Complexity Analysis

Steps (1), (2), (3), and (4) are all implemented in terms of convolutions of the image with kernels of a fixed size. In scipy convolutions are implemented in time O(n log n), where n is the dimension of the image. If the image has dimensions i j, the time complexity will be O(ij log ij) for these steps.

The final step works by postprocessing the image to remove all the high and low values, then dropping all other pixels that aren't near other pixels. This can be done in time O(n).

Therefore, the overall time complexity is O(n log n).

With regards to the 4K video provided, it had 38072 frames and each frame takes about 20 seconds to compute. According to my rough calcuations this will take 761440 seconds which is around 211 hours which is about 9 days to complete. Since I was only given 7 days to complete this taks, it is virtually impossible to finish the whole video, however I included a portion of the video with the submittion.

6 References

- "The Canny Edge Detector. An in-depth exploration" [Click here to visit the link]
- "The Sobel and Laplacian Edge Detectors" [Click here to visit the link]
- "Basic Implementation of Canny Edge" [Click here to visit the link]
- "How to install opencv2" [Click here to visit the link]
- "ffmpeg a complete, cross-platform solution to record, convert and stream audio and video" [Click here to visit the link]
- "Installing scipy & NumPy" [Click here to visit the link]
- "Installing Tkinter" [Click here to visit the link]