CANNY EDGE DETECTOR

AMIT SRIVASTAVA
2017CSB1189

Indian Institue of Technology
Ropar - Punjab,
India

Abstract

Canny edge detection is an image processing method used to detect edges in an image while suppressing noise.It extracts useful structural information from different vision objects and dramatically reduce the amount of data to be processed. It has been widely applied in various computer vision systems. Canny has found that the requirements for the application of edge detection on diverse vision systems are relatively similar.

It involves following steps :
1. Noise reduction
2. Gradient calculation
3. Non-maximum suppression
4. Double threshold
5. Edge Tracking by Hysteresis

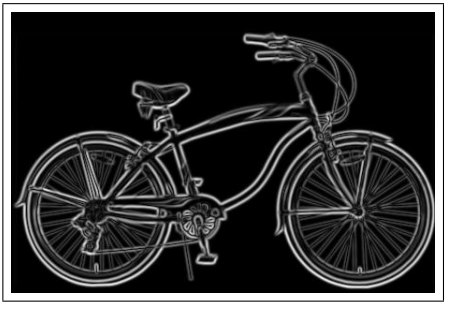
It is also known as the optimal edge detector :
1. The first criterion should have low error rate and filter out unwanted information while the useful information preserve.
2. The second criterion is to keep the lower variation as possible between the original image and the processed image.
3. Third criterion removes multiple responses to an edge.

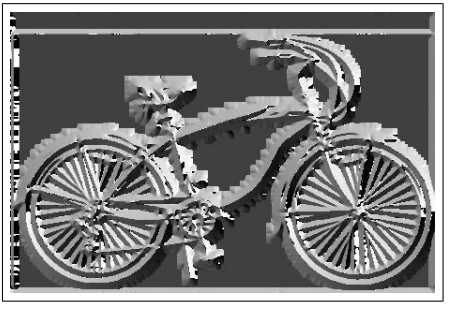
First of all convert the image to gray-scale. Perform a Gaussian blur on the image.The gradients can be determined by using a Sobel filter. The image magnitude produced results in thick edges. Ideally, the final image should have thin edges. Thus, we must perform non maximum suppression to thin out the edges.After that I performed an edge tracking algorithm to remove weak edges. Weak edges that are connected to strong edges will be actual/real edges. Weak edges that are not connected to strong edges will be removed. I performed this using DFS technique.

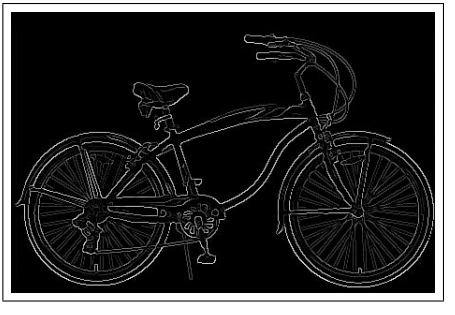
Output and Intermediate Image format :

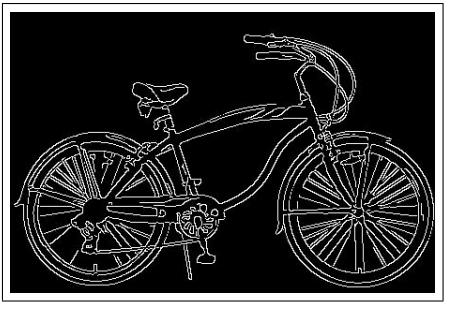
1. Gradient Magnitue
2. Gradient Direction
3. Thinned Image
4. Canny Output

1 Intermediate and Final Images of Bicycle

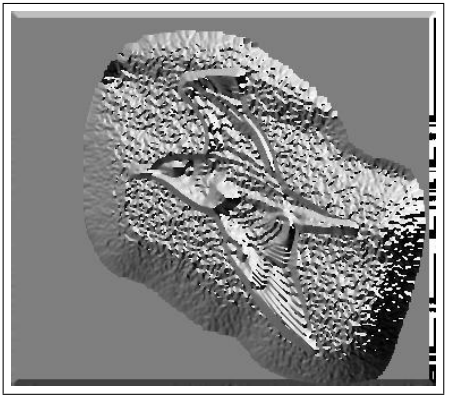


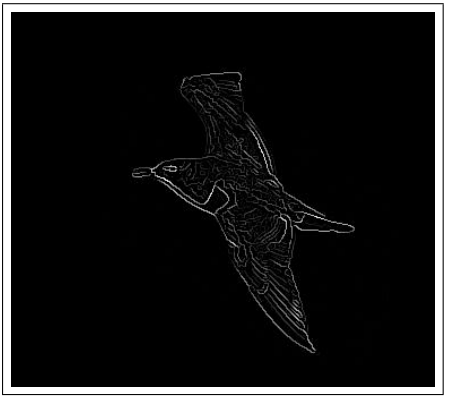


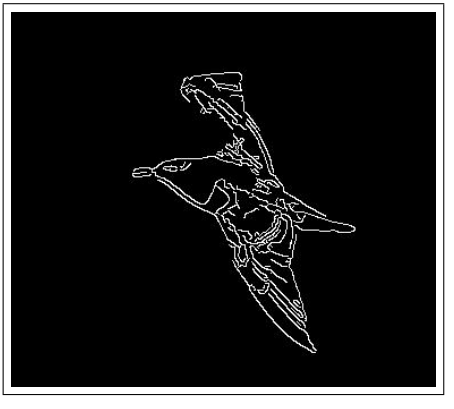




2 Intermediate and Final Images of Bird

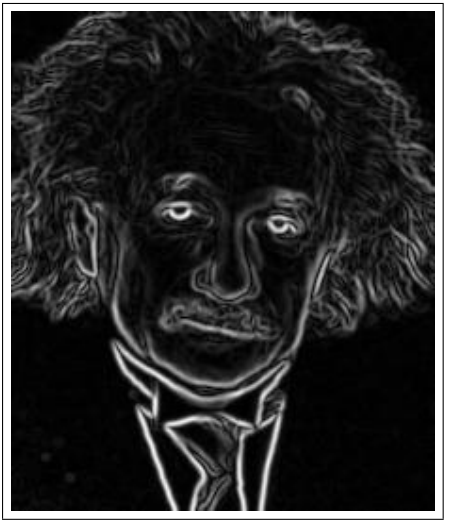


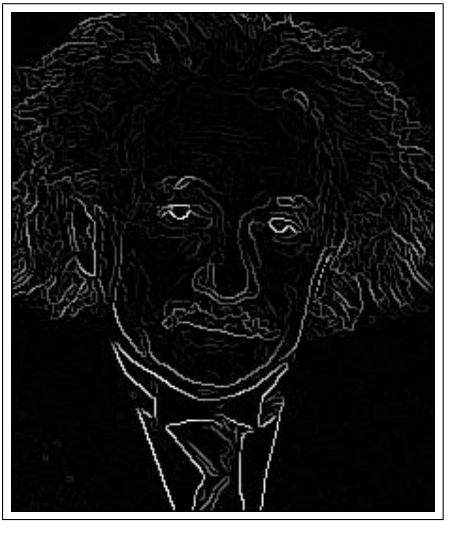


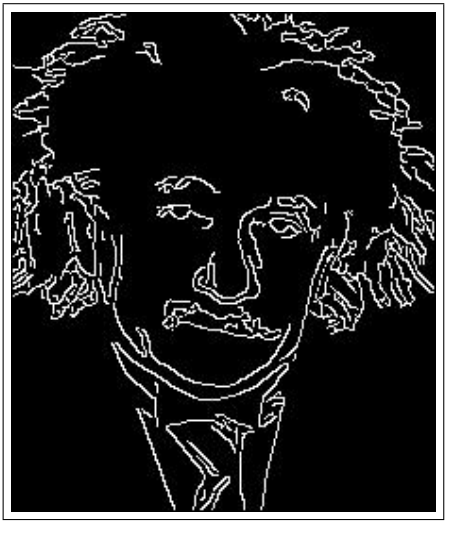


3 Intermediate and Final Images of Dog

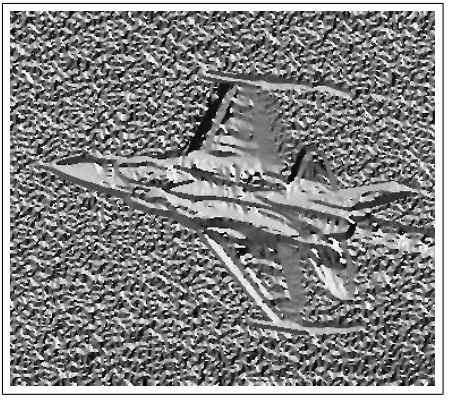
4 Intermediate and Final Images of Einstein

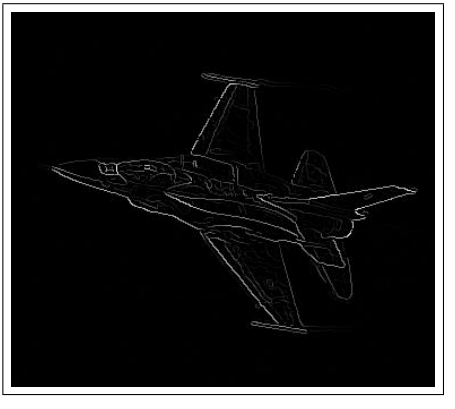


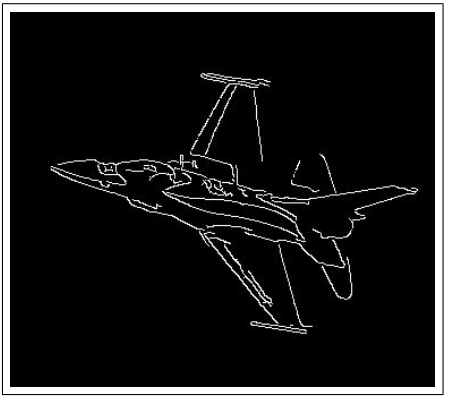




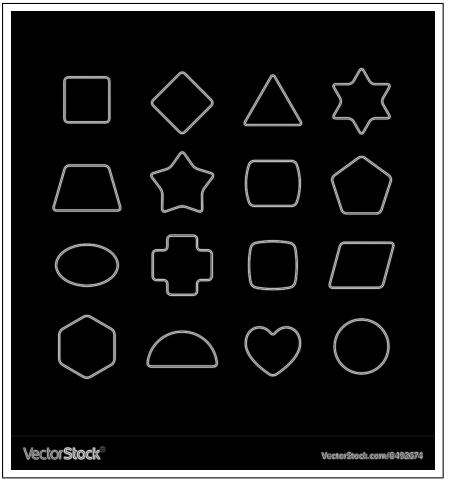
5 Intermediate and Final Images of Plane

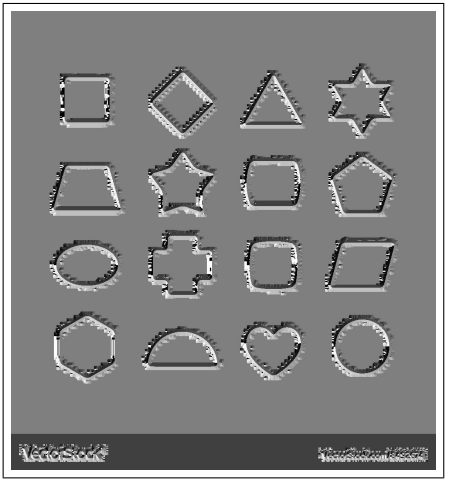


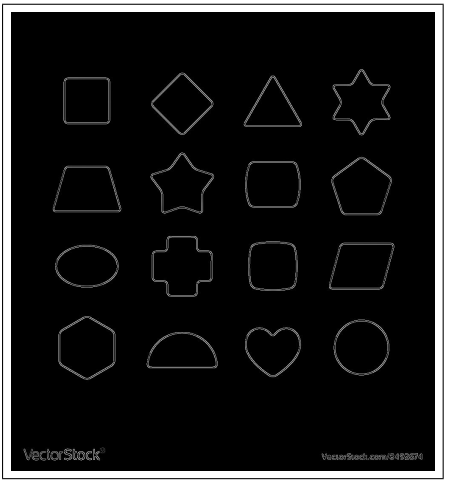


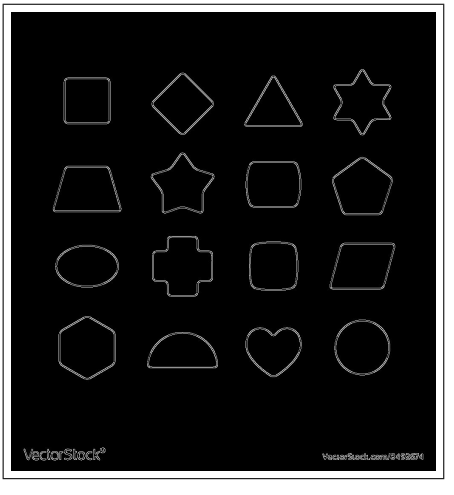


6 Intermediate and Final Images of Toy Image









7 Algorithms

Algorithm 1 closest-direction-function(grad-dir)

$closest-dir-arr = np.zeros(grad-dir.shape)$

while $i < height$ **do**
  while $j < width$ **do**

    if $(grad-dir[i,j] > -22.5 \; and \; grad-dir[i,j] <= 22.5) \; or \; (grad-dir[i,j] <= -157.5 \; and \; grad-dir[i,j] > 157.5)$ **then**
      closest-dir-arr[i, j] = 0
    end if

    if $(grad-dir[i,j] > 22.5 \; and \; grad-dir[i,j] <= 67.5) \; or \; (grad-dir[i,j] <= -112.5 \; and \; grad-dir[i,j] > -157.5)$ **then**
      closest-dir-arr[i, j] = 45
    end if

    if $(grad-dir[i,j] > 67.5 \; and \; grad-dir[i,j] <= 112.5) \; or \; (grad-dir[i,j] <= -67.5 \; and \; grad-dir[i,j] > -112.5)$ **then**
      closest-dir-arr[i, j] = 90
    end if

    if $(grad-dir[i,j] > 112.5 \; and \; grad-dir[i,j] <= 157.5) \; or \; (grad-dir[i,j] <= -157.5 \; and \; grad-dir[i,j] > 157.5)$ **then**
      closest-dir-arr[i, j] = 135
    end if

  end while
end while
Return closest-dir-arr

Algorithm 2 Non-Maximal-Suppressor(grad-mag, grad-dir)

thinned-output = np.zeros(grad-mag.shape)

while $i < height$ **do**
  while $j < width$ **do**

    if $closest-dir[i,j] == 0$ **then**
      if $(grad-mag[i,j] > grad-mag[i,j+1])$ *and* $(grad-mag[i,j] > grad-mag[i,j-1])$ **then**
        $thinned-output[i,j] = grad-mag[i,j]$
      else
        $thinned-output[i,j] = 0$
      end if
    end if

    if $closest-dir[i,j] == 45$ **then**
      if $(grad-mag[i,j] > grad-mag[i+1,j+1])$ *and* $(grad-mag[i,j] > grad-mag[i-1,j-1])$ **then**
        $thinned-output[i,j] = grad-mag[i,j]$
      else
        $thinned-output[i,j] = 0$
      end if
    end if

    if $closest-dir[i,j] == 90$ **then**
      if $(grad-mag[i,j] > grad-mag[i+1,j])$ *and* $(grad-mag[i,j] > grad-mag[i-1,j])$ **then**
        $thinned-output[i,j] = grad-mag[i,j]$
      else
        $thinned-output[i,j] = 0$
      end if
    end if

    if $closest-dir[i,j] == 135$ **then**
      if $(grad-mag[i,j] > grad-mag[i+1,j-1])$ *and* $(grad-mag[i,j] > grad-mag[i-1,j+1])$ **then**
        $thinned-output[i,j] = grad-mag[i,j]$
      else
        $thinned-output[i,j] = 0$
      end if
    end if

  end while
end while
Return thinned-output

Algorithm 3 hyst-threshold(img)

```
low − ratio = 0.10
high − ratio = 0.30
diff = np.max(img) - np.min(img)
t-low = np.min(img) + low-ratio * diff
t-high = np.min(img) + high-ratio * diff
temp-img is copy of original img

while i < height do
    while j < width do
        if img[i, j] > t − high then
            temp-img[i,j] = 2
        end if
        if img[i, j] < t − low then
            temp-img[i,j] = 0
        else
            temp-img[i,j] = 2
        end if

        Apply DFS to find those weak edges that are connected to strong edges
        Include these pixels in final output
    end while
end while
Return temp-img
```

8 Conclusion

The output of the canny edge detector were nearly accurate. The output heavily dependent on the threshold value. This threshold is different per image so I had to vary the values. First I found different threshold values for each image that give best result. But it is quite a tedious task and for new image we have to choose new threshold value. In my implementation I found it helpful to choose a threshold ratio instead of a specific value and multiple that by the difference of max pixel value and minimum pixel value in the image.

I set threshold ratios to be 0.10 and 0.30 to get the best output. Lower threshold value is low-ratio * diff and Higher threshold value is hgh-ratio * diff. Doing this allowed me to successfully use approximately the same ratios for other images to successfully detect edges.

The output also depend on the edge tracking algorithm used. The one I implemented gives the best result. Weak edges that are connected to strong edges will be actual/real edges. Weak edges that are not connected to strong edges will be removed. To speed up this process, my algorithm keeps track of the weak and strong edges that way I can recursively iterate through the strong edges and see if there are connected weak edges instead of having to iterate through every pixel in the image.

When I used the Depth-First-Search Technique for edge tracking than the output was not as accurate as the one that I finally implemented. The one I implemented is a little bit slower than the normal Depth-First-Search Technique but since it gives better result, in my final code I implemented it.