A Comprehensive Guide To Canny Edge Detection In Python

Tyler Jones - tyler.jones@byu.edu

December 26, 2019

Abstract

Canny Edge Detection is an algorithm to minimize noise and identify edges in a picture. Through this process one can reduce the amount of data in an image, while still maintaining its general shape and form. Edge detection is often just one of many filters used in image processing, and this paper will look to guide users in implementing their own Canny Edge Detection in Python.

Introduction & Prerequisites

Canny Edge Detection utilizes convolutions and intermediate level programming, as well as basic concepts in linear algebra. It is highly recommended that users are familiar with the Python Libraries *Numpy*, *Pillow*, and *Matplotlib*.

There are 5 primary steps to implement this algorithm: smoothing, estimating gradients, non-maximum suppression, double thresholding, and edge tracking. Each step will be described with visuals in detail below, along with code samples.

For those unfamiliar, a brief example of a convolution K being applied to a matrix A is given below. Note that with convolutions the resulting image is slightly smaller, just by definition of the convolution. Mathematically this would be written as:

$$A * K = A'$$

$$\underbrace{\begin{pmatrix} 0 & 1 & 1 & 1_{x1} & 0_{x0} & 0_{x1} & 0 \\ 0 & 0 & 1 & 1_{x0} & 1_{x1} & 0_{x0} & 0 \\ 0 & 0 & 0 & 1_{x1} & 1_{x0} & 1_{x1} & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}}_{A} * \underbrace{\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}}_{K} = \underbrace{\begin{pmatrix} 1 & 4 & 3 & 4 & 1 \\ 1 & 2 & 4 & 3 & 3 \\ 1 & 2 & 3 & 4 & 1 \\ 1 & 3 & 3 & 1 & 1 \\ 3 & 3 & 1 & 1 & 0 \end{pmatrix}}_{A'}$$

Step 1 - Smoothing

The motivation for this first step is to remove small amounts of noise from our image, which should ensure that we only pick up on legitimate edges. To smooth or slightly blur the image we use a $5x5$ Gaussian Kernel.

$$S = \frac{1}{159}\begin{pmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{pmatrix}$$

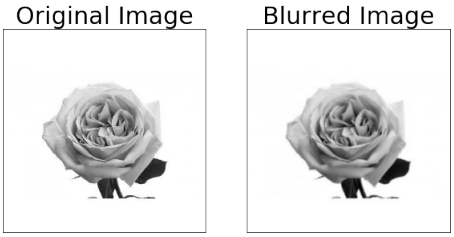

Figure 1: An example of a Gaussian Kernel is found on the left, and its implementation on a simple graphic is shown on the right.

```python
def Gaussian_Smoothing(K, A):
    """ Given a grayscale matrix A and kernel K, apply
        and return the kernel.
    """

    # Get bounds of image and check kernel
    n = K.shape[0]
    assert n % 2 == 1

    # Find bounds for our kernel
    offset_ = int(np.floor(n/2))
    lim_x, lim_y = A.shape
    B = np.copy(A)

    # Apply our kernel across our image
    for i in range(offset_, lim_x-offset_):
        for j in range(offset_, lim_y-offset_):
            # Get the bounds of our convolution
            x_l, x_r = i-offset_,i+offset_+1
            y_l, y_r = j-offset_,j+offset_+1
            A_ = A[x_l:x_r,y_l:y_r]
            # Apply and store our convolution
            L = np.multiply(A_, K)
            B[i,j] = np.sum(L.flatten())

    return B
```

Listing 1: Applying a Gaussian Kernel

Utilizing the example code above and any gray-scale image, the resulting image will be slightly more softened. Note that edges are smoothed, yet the general structure of the image still remains unchanged. This process can be done multiple times to further soften the image if needed.

Step 2 - Estimating Gradients

Next we look at estimating the strength of edges among each pixel. We do this by again using convolutions, but this time we use the *Sobel-operator*. At each pixel of our now smoothed image, use each respective kernel to estimate the gradient or edge strength in the x and y direction.

$$G_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \qquad G_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

Figure 2: Kernels used to find the estimated gradients in the x and y direction

At each pixel *i,j*, calculate $G_x^{i,j}$ and $G_y^{i,j}$, and then define the total gradient or strength of said pixel by the following equation:

$$G^{i,j} = min(255, \sqrt{(G_x^{i,j})^2 + (G_y^{i,j})^2})$$

In addition to estimating the strength of the gradient at each pixel, we also store the direction in a separate matrix T. To do this we use the *atan2* function, and then round it to the nearest $\frac{\pi}{4}$ for simplicity. Thus the direction of the gradient at pixel *i, j* is defined as:

$$T_{i,j} = atan2(\frac{G_y^{i,j}}{G_x^{i,j}})$$

Notice from the figures below that some noise may be present in both the pixel gradients or the gradient direction. This is to be expected, and can be accounted for in our next steps.

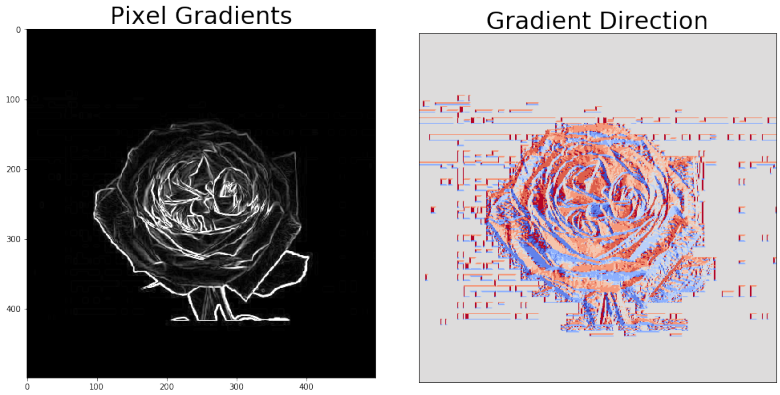

Figure 3: On the left we have our matrix G of gradient strengths, and on the right we see our matrix T of the direction of the gradient in radians.

```
def gradient_magnitudes(K_X, K_Y, A):
    """ Given a matrix A and two kernels K_X, K_Y,
        run the kernels on our matrix A.
    """

    # Get bounds of image and check kernel
    n = K_X.shape[0]
    assert n % 2 == 1

    offset_ = int(np.floor(n/2))
    lim_x, lim_y = A.shape

    # Hold edge strengths and edge direction
    G = np.zeros((A.shape[0]-offset_, A.shape[1]-offset_))
    T = np.zeros((A.shape[0]-offset_, A.shape[1]-offset_))

    # Get our 8 directions that our gradient can be going
    neighborhood = [0, np.pi/4, np.pi/2, 3*np.pi/4, np.pi,
                    -np.pi/4, -np.pi/2, -3*np.pi/4, -np.pi]

    # Go through our whole image
    for i in range(offset_, lim_x-offset_):
        for j in range(offset_, lim_y-offset_):

            # Get our kernel indices
            x_l, x_r = i-offset_,i+offset_+1
            y_l, y_r = j-offset_,j+offset_+1
            A_ = A[x_l:x_r,y_l:y_r]

            # Calculate our kernel and respective values
            X_res = np.multiply(A_, K_X)
            Y_res = np.multiply(A_, K_Y)
            G_y = np.sum(Y_res.flatten())
            G_x = np.sum(X_res.flatten())

            # Calculate our edge strength
            G[i,j] = min(np.sqrt(G_y**2+G_x**2), 255)

            # Use atan2 to calculate our gradient direction
            direc_ = math.atan2(G_y, G_x)

            # Fit edge direction to closest pi/4 angle
            closest = neighborhood[np.argmin([np.abs(direc_-d)
                      for d in neighborhood])]

            T[i, j] = closest

    return G, T
```

Listing 2: Example code to find the gradient magnitudes and gradient directions for each pixel in an image.

Step 3 - Non-Maximum Suppression

In order to reduce noise and keep only the strongest edges, we apply non-maximum suppression to our entire image. Given our gradient magnitude matrix G, and gradient directional matrix T we apply this step pixel by pixel to G.

Recall that $T_{i,j}$ contains the direction of the gradient of the i,j'th pixel of G, rounded to the nearest $\frac{\pi}{4}$. To perform non-maximum suppression on the (i, j) pixel in G, follow the following steps:

1. Find the direction of the gradient of said pixel, given in $T_{i,j}$.
2. Compare the value of $G_{i,j}$ to the values of G immediately parallel along the direction of the gradient.
3. If $G_{i,j}$ is not the largest value, then it is not the strongest edge and can be discarded, set $G_{i,j} = 0$. Otherwise if it is the largest value, retain it.

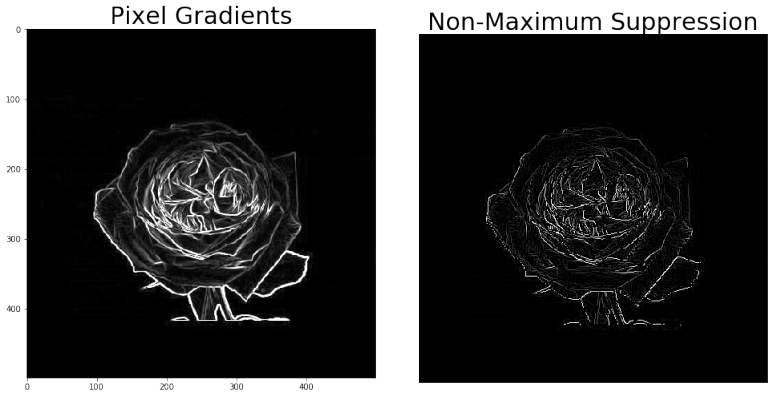

Figure 4: On the left we again have our matrix G of gradient strengths, and on the right we see our modified matrix G' where only the strongest edges are kept This greatly reduces noisy edges and keeps only the strongest edge signals.

```
def non_max_supression(G, T):
    """ Given an image matrix A, along with a matrix of it's
        magnitude direction we identify and keep only the
        strongest edges.
    """

    assert G.shape == T.shape

    # Iterate through our whole image
    for i in range (1,G.shape[0]-1):
        for j in range(1,G.shape[1]-1):
```

```
            # Get direction of gradient in specific cell
            direc_ = np.abs(T[i,j])
            val_ = G[i,j]

            if direc_ == np.pi/4:
                """ Look in the upper right and bottom left
                    direction
                """
                if val_ > G[i-1, j+1] and val_ > G[i+1, j-1]:
                    pass
                else:
                    G[i,j] = 0

            elif direc_ == np.pi/2:
                """ Look in the up/down direction """
                if val_ > G[i-1, j] and val_ > G[i+1, j]:
                    pass
                else:
                    G[i,j] = 0

            elif direc_ == 3*np.pi/4:
                """ Look in the top left and bottom right
                    direction
                """
                if val_ > G[i-1, j-1] and val_ > G[i+1, j+1]:
                    pass
                else:
                    G[i,j] = 0

            elif direc_ == np.pi or direc_ == 0:
                """ Look in the left and right directions """
                if val_ > G[i, j-1] and val_ > G[i, j+1]:
                    pass
                else:
                    G[i,j] = 0

    return G
```

Listing 3: Code to perform Non-Maximum Suppression on our target image G.

Step 4 - Double Thresholding

Now we have an array G where only the strongest edges are retained. However the strength of each of those pixels has a wide distribution. To remedy this we classify each pixel as either *strong, weak* or *noise*. This step is aptly named as to do this we simply use a double threshold with values θ_l and θ_h where $\theta_l < \theta_h$. To determine the value of $G_{i,j}$:

$$G_{i,j} = \begin{cases} 0 & G_{i,j} < \theta_l \\ 128 & \theta_l \leq G_{i,j} \leq \theta_h \\ 255 & \theta_h < G_{i,j} \end{cases}$$

The choices for our threshold θ_l and θ_h are hyper-parameters that must be chosen wisely for the image on hand. It is recommended to try many different values and identify what works best. Images with a lot of noise may require higher filters than average in order to best discern actual edges.

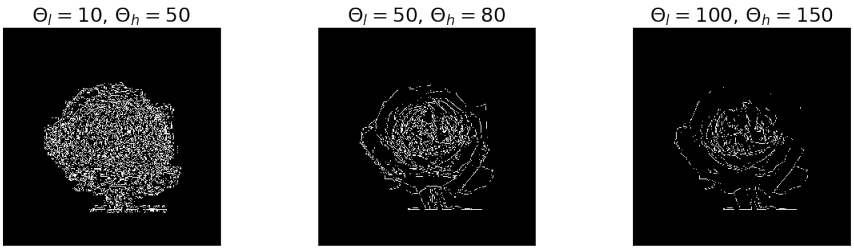

Figure 5: Example of how different threshold values alter the output of our image. Note that the lower θ_l is, the more noise is classified as an actual edge and kept in our image.

```
def double_thresh(H, L, G):
    """ Given a matrix G, and hyper-parameters H and L,
        performs double threshold on G.
    """
    A = np.copy(G)

    A[A > H] = 255
    A[A < L] = 0
    A[(A > L) & (A < H)] = 128

    return A
```

Listing 4: Code to perform a double threshold on our target image G.

Step 5 - Edge Tracking

For our final step we look to discern which weak edges to keep, and which to discard. To do this we retain all of the strong edges, and then look at every weak edge. If the weak edge is connected to a strong edge in its direct neighborhood, then we retain it. Otherwise we discard it as noise.

```
def edge_track(G):
    """ Given a matrix G, remove all weak edges that are not
        connected to a strong edge.
    """

    for i in range (1,G.shape[0]-1):
        for j in range(1,G.shape[1]-1):
            if G[i, j] == 128:
                # Examine the neighborhood of a weak edge
```

```
                N = (G[i-1:i+1, j-1:j+1]).flatten()

                if 255 not in N:
                    G[i,j] = 0
            else:
                pass
```

Listing 5: Code to implement edge tracking on our G.

Conclusion

Through the process of smoothing, estimating gradients, non-maximum suppression, double thresholding, and edge tracking, you are able to reduce the amount of data in an image and identify edges. Canny Edge Detection is just one of many edge detection methods, but understanding how to implement it is a great way to learn more about computer vision.

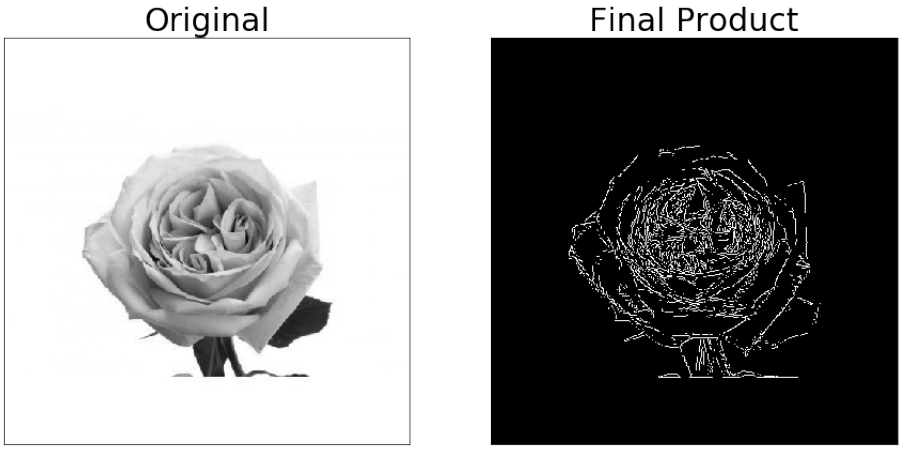

Figure 6: The full implementation of our Canny Edge Detection Algorithm