C:\opencv\build\x64\vc16\lib

opencv\_world4110d.lib

C:\opencv\build\include

D:/FresherXavisTech/Image/

<https://www.slideserve.com/eloise/canny-edge-detection>

<http://contents2.kocw.or.kr/KOCW/document/2018/handong/hwangsungsoo0228/15.pdf>

<https://docs.opencv.org/3.4/da/d5c/tutorial_canny_detector.html>

<https://www.cs.bilkent.edu.tr/~saksoy/courses/cs484-Spring2010/slides/cs484_edge.pdf>

<https://courses.cs.washington.edu/courses/cse576/16sp/Slides/4_Edges.pdf>

<https://homepages.inf.ed.ac.uk/rbf/AVINVERTED/STEREO/av5_edgess.pdf>

<https://justin-liang.com/tutorials/canny/#grayscale>

<https://www.youtube.com/watch?v=aCbfvxgYy8s>

**Slide 1:**

Now, I will present my results comparing the Gaussian filter and the Bilateral filter.

I applied both filters to a noisy image, and we can see the results here.  
On the left is the original image, in the middle is the result after applying the Gaussian filter, and on the right is the result from the Bilateral filter.

First, I fix the brightness sigma = 1.0, and I vary the spatial sigma, starting from 1.0

Let’s look at the Gaussian filter first.

* At σ = 1, the result is slightly blurred, and the noise is reduced quite well.
* When increasing σ to 2.0, 3.0, and 10.0, the image becomes increasingly blurred. While noise is removed almost completely, edges and small details are also blurred.
* At **σ = 2**, we already see that the edges are softened, the light gray region and the small gray dots become hard to see.
* At **σ = 3**, those details become even harder to recognize.
* And at **σ = 10**, we can no longer see the gray areas or the small dots — only the basic shape of the original image remains visible.

Now let’s go back to the **Bilateral filter**, with **range sigma = 1.**

This is a relatively small value, meaning that **only pixels with intensity values similar to the central pixel** will be included in the smoothing process.

Pixels that have a large intensity difference — such as noise or edges — are **excluded from the filtering effect.**

As a result, the output of the Bilateral filter remains **almost unchanged**, even when we increase **σs** to 2, 3, or 10.

In all four cases, the image looks nearly the same: noise is only slightly reduced, and **edges are very well preserved.**

**Slide 2:**

Now let’s consider the case where **σr is larger**, for example **σr = 15**.

In this case, when we increase **σs**, we can observe that image becomes smoother and the noise is effectively removed, but at the same time, edges and small details are well preserved.

The black details in the image become more clearly separated from the background, which is a major improvement compared to the results of the Gaussian filter.

This shows the strength of the Bilateral filter it can smooth the image while still maintaining important structural information.

**Slide 3:**

So far, we have kept σr constant.

Now let’s consider the opposite case: we fix **σs = 3.0** and **increase σr**

As we increase σr the filter begins to smooth over larger intensity differences between pixels. This results in a smoother image with better noise removal.

However, if σr becomes too large, the filter will start to blur edges and fine details, which defeats the purpose of the bilateral filter, and the filter behaves more like a Gaussian filter — blurring everything uniformly, including both noise and edges.

**Slide 4:**

We now apply the same process to a different image.

First, I keep sigma r constant and vary sigma s.

This zoomed-in view shows that when sigma r is small, regions with low intensity differences are effectively smoothed and noise is reduced.

And here is the result when we change sigma r. The results are similar to the previous example.

**Slide 21:**

Here is a performance comparison table between my function and OpenCV's built-in function. The results show that my function performs significantly worse than OpenCV's, taking much more time to execute. A comparison of the two outputs (one from my function and one from OpenCV) is shown below: we can see that the results are quite similar.

**Slide 22:**

Today, I will present the Canny edge detection algorithm.

Last week, I presented the Sobel operator, a simple method that uses gradient filters to calculate intensity changes in the horizontal and vertical directions. The result highlights edges where there is a significant change in brightness.

However, Sobel has no built-in noise suppression, the detected edges are usually thick, and false edges may appear in noisy regions.

Canny is a more complex method, which also uses gradient filters but is able to overcome these drawbacks. It detects edges more accurately, removes noise effectively, and produces edges that are much sharper compared to Sobel.

Slide 23 – 25:

Last week, I presented the different types of edges, explained why derivatives are used in edge detection, and discussed how noise affects derivatives, so we will skip those parts in this presentation.

**Slide 26:** This is steps for Canny edge detection

First, we smooth the input image using a Gaussian filter with a standard deviation sigma. This helps reduce image noise and small irrelevant details that could cause false edges.

Next, we compute the gradient magnitude and direction at each pixel of the smoothed image, typically using Sobel operators. This tells us how strong the edge is and in which direction it points.

In step 3, we perform non-maximum suppression. For each pixel, we look along the gradient direction and suppress it (set to zero) if it's not a local maximum compared to its neighbors. This step thins out the edges.

We then apply hysteresis thresholding with two thresholds: a high and a low. Strong edge pixels above the high threshold are kept, and weak ones between the two thresholds are only kept if they are connected to strong edges.

Finally, we keep only the pixels that belong to the strong edges traced through the hysteresis step. This removes small, isolated segments and retains meaningful contours.

**Slide 27:**

In Step 2, we typically use the Sobel filter to calculate the gradient magnitude and the direction of the gradient vector. Note that the gradient vector is always perpendicular to the edge.

**Slide 28:**

Sobel filtering usually produces edges that appear somewhat thick. The next step is to thin those ridges. One approach is to use *non-maximum suppression.*

On the left, we see two curves. The gray dashed line represents a sharp step edge, where the intensity jumps suddenly from 0 to 1. The curve shows the result of applying a Gaussian filter to this edge. Notice how the transition becomes smooth and gradual, instead of abrupt. This simulates how real-world images often behave after noise reduction.

On the right plot, we show the gradient of the smoothed edge – which is the first derivative of intensity. The curve peaks at the center of the edge, where the intensity changes most rapidly. This peak indicates the position of the strongest edge and is used in edge detection algorithms to identify edge locations.

To identify this point, we compare it with its neighboring points to determine if it is a local maximum. This example shows the situation for the two possible orientations of an edge

**Slide 30:**

Each pixel in the image is surrounded by a 3×3 matrix. The central pixel of this matrix is **w5**, which is the pixel we want to examine. The surrounding pixels, from **w1** to **w4** and from **w6** to **w9**, are the neighbors of **w5**. The diagram in the middle illustrates the relationship between an edge in the image and the **edge normal**, or gradient vector. The angle **α** represents the direction of this gradient vector. **α** is calculated using a specific formula.

However, in practical digital image processing, handling infinitely continuous gradient directions is inefficient. Therefore, we need to **quantize** (round off) these gradient directions. The circular diagram divides 360 degrees into 8 segments corresponding to the 8 neighboring pixels around **w5**. In this way, any gradient direction at pixel **w5** is rounded to one of these 8 standard directions (or grouped into 4 main categories).

Once we have the **quantized gradient direction** of pixel **w5**, we return to the 3×3 matrix to perform the final comparison step. **Non-Maximum Suppression (NMS)** will compare the gradient magnitude of the central pixel **w5** with the gradient magnitudes of the two neighboring pixels that lie exactly on the line in the gradient direction.

Finally, if the gradient magnitude of **w5** is the greatest among the three, **w5** is retained and considered a valid edge pixel. Otherwise, it is suppressed. This process is repeated for every pixel in the image, ensuring that the edges detected by the Canny algorithm are only **one pixel wide**, sharp, and clearly defined.

**Slide 31:**

After Non-Maximum Suppression, we obtain a map of 'potential edges' with varying strengths. However, we still cannot definitively distinguish true edges from noise. Thresholding helps address this issue. Using a single threshold is challenging, as a value too low may accept excessive noise, while a value too high risks breaking important but weaker edges. Hysteresis Thresholding employs two different intensity thresholds, a high and a low threshold, to resolve this.

**Slide 32:**

Hysteresis Thresholding main goal is to accurately separate true edges from noise. It does this by using two thresholds: a high threshold, called T-high, and a low threshold, called T-low.

First, for each pixel, we check the gradient magnitude:

If it's greater than T-high, we classify it as a Strong Edge Pixel – it’s definitely part of a real edge.

If it’s less than T-low, it’s considered not an edge, and we discard it.

If it lies between T-low and T-high, we marked it a Weak Edge Pixel – this needs further checking.

In the next step, we perform weak edge linking. For every Weak Edge Pixel, we check whether it's connected to any Strong Edge Pixel using 8-connected neighbors.

If it is connected, we keep it as part of the edge.

If it’s not connected, it’s likely caused by noise, so we discard it

Finally, only the Strong Edge Pixels and the connected Weak Edge Pixels are kept in the final edge map. This process helps ensure that real edges are continuous and smooth, while noise and false edges are removed."

**Slide 33:**

Here is a performance comparison table between the function I wrote and OpenCV's built-in function. The results show that my function performs significantly worse than OpenCV's, taking much more time to execute. Let take a look at the result images. Overall, the custom-built function was able to detect the main edges of the strawberry and the round shape of the plate. However, compared to the results from OpenCV, the edges produced by "My function" appear to be:

* Less continuous: There are many broken or incomplete boundary segments, especially in the small details of the strawberry.
* Noisier: Some small, irrelevant edges or noise artifacts from the original image appear more frequently than in the OpenCV result.

**Overall Conclusion:**

The custom-implemented Canny function successfully performed edge detection; however, the quality of the output has not yet achieved the level of refinement and consistency demonstrated by the standard Canny implementation in OpenCV.