

Gaussian Filters

A common tool for filtering images is the Gaussian Filter. This is a filter that uses a Gaussian function to create a two-dimensional filter kernel. But, instead of leaping right into a 2D filter, we'll start by examining the use of a Gaussian filter in only one dimension. The one-dimensional Gaussian is defined as:

$$h_i = \frac{1}{\sigma\sqrt{2\pi}} e^{-i^2/2\sigma^2} \quad 1$$

The parameter sigma determines the amount of smoothing the filter will perform. A larger value will cause greater smoothing. We can do a Gaussian filter on the rows of an image using a one-dimensional filter.

Example 1 – A $\sigma=2$ row filter

Suppose you want to create a Gaussian filter on the rows of an image with $\sigma=2$. We have to decide how big we want that filter to be. A good rule thumb is that σ should be about 70% of the size of the filter neighborhood, the range to the left and right of the pixel we are looking at. So, we want a neighborhood n such that $\sigma=0.7n$. $n=2.8$, but neighborhoods are always integers, and need to contain the range, so I'll round this number up to 3. This means the filter kernel size needs to be 7. One (1) for the pixel itself and three (3) on each size. I can compute the filter coefficients using Equation 1:

$$h_{-3} = 0.064, h_{-2} = 0.121, h_{-1} = 0.176, h_0 = 0.199, h_1 = 0.176, h_2 = 0.121, h_3 = 0.064$$

We can express this as a matrix: $H=[0.064 \ 0.121 \ 0.176 \ 0.199 \ 0.176 \ 0.121 \ 0.064]$. Now we can consider the indices into this matrix as -3, -2, -1, 0, 1, 2, 3, but that's not really what a compiler will like, so we'll consider the matrix to start at zero. So, h_3 is location 0. Any h_k will be location $k+3$ in the matrix. To compute the color at pixel r,c in our image, we compute the following equation:

$$y_{c,r} = \sum_{j=-\infty}^{\infty} x_{c-j,r} h_j \quad 2$$

Since h only ranges from -3 to 3, we can bound this equation to that range:

$$y_{c,r} = \sum_{j=-3}^3 x_{c-j,r} h_j \quad 3$$

Now we'll convert the h values to a location in the array:

$$y_{c,r} = \sum_{j=-3}^3 x_{c-j,r} H[j+3] \quad 4$$

In C++, this equation would be implemented as in Listing 1, assuming a monochrome image.

Listing 1 – Row Gaussian filtering of a monochrome image

```

for(int r=0; r<height; r++)
{
    for(int c=0; c<width; c++)
    {
        double sum = 0;
        for(int j=-range; j<=range; j++)
        {
            if(c + j < 0 || c + j >= width)
                continue;

            sum += H[j + range] * image1[r][c + j];
        }
        image2[r][c] = BYTE(sum);
    }
}

```

Figure 1 is an example image and Figure 2 is the same image after application of a row Gaussian filter of size 7 and $\sigma=2.0$.

A two-dimensional filter can be created from a one-dimensional filter by multiplying the filter as a vector by the transpose of the vector. This means we simply multiply a column vector by a row vector as illustrated in Equation 5.

$$\begin{bmatrix} h_{-2} \\ h_{-1} \\ h_0 \\ h_1 \\ h_2 \end{bmatrix} \begin{bmatrix} h_{-2} & h_{-1} & h_0 & h_1 & h_2 \end{bmatrix} = \begin{bmatrix} h_{-2}h_{-2} & h_{-2}h_{-1} & h_{-2}h_0 & h_{-2}h_1 & h_{-2}h_2 \\ h_{-1}h_{-2} & h_{-1}h_{-1} & h_{-1}h_0 & h_{-1}h_1 & h_{-1}h_2 \\ h_0h_{-2} & h_0h_{-1} & h_0h_0 & h_0h_1 & h_0h_2 \\ h_1h_{-2} & h_1h_{-1} & h_1h_0 & h_1h_1 & h_1h_2 \\ h_2h_{-2} & h_2h_{-1} & h_2h_0 & h_2h_1 & h_2h_2 \end{bmatrix} \quad 5$$

Each location in the 2D matrix is simply the product of two other locations. This leads to a simple way to implement the 2D Gaussian filter. Equation 6 computes a pixel using a two filter kernels for the rows and columns, where the rows and columns are assumed to be filtered the same way.

$$y_{c,r} = \sum_{k=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} x_{c-j,r-k} h_j h_k \quad 6$$

Example 2 – A $\sigma=2$ 2D Gaussian filter

Suppose you want to create a 2D Gaussian filter with $\sigma=2$. As in Example 1, we have to decide how big we want that filter to be. As before, this leads to a filter with a kernel size of 7. Again, we compute the filter coefficients using Equation 1:

$$h_{-3} = 0.064, h_{-2} = 0.121, h_{-1} = 0.176, h_0 = 0.199, h_1 = 0.176, h_2 = 0.121, h_3 = 0.064$$

We can express this as a matrix: $H=[0.064 \ 0.121 \ 0.176 \ 0.199 \ 0.176 \ 0.121 \ 0.064]$. Now we can consider the indices into this matrix as -3, -2, -1, 0, 1, 2, 3, but that's not really what a compiler will like, so we'll consider the matrix to start at zero. So, h_3 is location 0. Any h_k will be location $k+3$ in the matrix. To compute the color at pixel r,c in our image, we compute the following equation:

$$y_{c,r} = \sum_{k=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} x_{c-j,r-k} h_j h_k \quad 7$$

Since h only ranges from -3 to 3, we can bound this equation to that range:

$$y_{c,r} = \sum_{k=-3}^3 \sum_{j=-3}^3 x_{c-j,r-k} h_j h_k \quad 8$$

Now we'll convert the h values to a location in the array:

$$y_{c,r} = \sum_{k=-3}^3 \sum_{j=-3}^3 x_{c-j,r-k} H[j+3]H[k+3] \quad 9$$

In C++, this equation would be implemented as in Listing 2, assuming a monochrome image.

Listing 2 – 2D Gaussian filtering of a monochrome image

```
for(int r=0; r<height; r++)
{
    for(int c=0; c<width; c++)
    {
        double sum = 0;
        for(int k=-range; k<=range; k++)
        {
            if(r + k < 0 || r + k >= height)
                continue;

            for(int j=-range; j<=range; j++)
            {
                if(c + j < 0 || c + j >= width)
                    continue;

                sum += H[j + range] * H[k + range] *
                    image1[r + k][c + j];
            }
        }
    }
}
```

```

        image2[r][c] = BYTE(sum);
    }
}

```

Figure 3 is an example of applying a 2D size 7 $\sigma=2.0$ Gaussian filter to an image.

Normalizing a Gaussian Filter

A characteristic of the Gaussian function from Equation 1 is:

$$\sum_{i=-\infty}^{\infty} \frac{1}{\sigma\sqrt{2\pi}} e^{-i^2/2\sigma^2} = 1 \quad 10$$

This is an indication that the Gaussian filter will be intensity preserving. If applied to an image, the resulting image will have the same average intensity. However, in most cases this function is windowed, meaning only a limited extent of the range of the function is utilized. If sigma is 2.0, a 7 pixel wide filter might be used and:

$$\sum_{i=-3}^3 \frac{1}{\sigma\sqrt{2\pi}} e^{-i^2/2\sigma^2} < 1$$

The net effect of windowing the Gaussian function is that the image intensity is decreased. For a size 7 filter with $s=2.0$, the sum is 0.92, so the image intensity will be decreased by 8% if the filter is applied to just the rows. Applying the filter in 2D decreases the intensity by nearly 15%.

The solution to this problem is to *normalize* the filter. Normalizing a filter simply ensures that the sum of all coefficients used is 1. This is accomplished by dividing each coefficient by the sum of all of the coefficients:

$$h_i' = \frac{h_i}{\sum_{i=-r}^r h_i} \quad 11$$

In this equation, r is the range of the filter coefficients. For a size 7 filter, $r=3$. Listing 3 is an example of normalizing a filter.

Listing 3 – Normalizing a filter

```

for(int j=-range; j<=range; j++)
{
    sum += H[j + range];
}

for(int j=-range; j<=range; j++)

```

```
{  
    H[j + range] /= sum;  
}
```

Figure 4 is an example of applying a size 7 $\sigma=2.0$ normalized 2D Gaussian filter to an image.

Example Images



Figure 1 - Original image



Figure 2 - Image after size 7 $\sigma=2.0$ Gaussian filter on the rows only



Figure 3 - Image after size 7 $\sigma=2.0$ 2D Gaussian filter



Figure 4 - Image after size 7 $\sigma=2.0$ Normalized 2D Gaussian filter