There is a basic statistics principle named ‘Law of large number’: if there is a random variable X, then the average of n ‘xi’ will converge at the expectation of X when n is infinite, which implies the standard deviation of S\_n\_X/n is 0 when n is infinite. If n is finite, then the standard deviation will drop as n goes up.

The noisy image can be seen as the original image plus noise, which is a random variable. If we have N images which have the same information and iid noise, then we can denoise by adding them up and divide by N.

However, in most case we don't have a bunch of ‘identical images’. But if we tear the whole image into small parts, we can find many parts have similar construction. So we can add these (lets say n)similar parts and then divide by n to improve the quality of each small parts and the quality of the whole image is enhanced when we traverse all the parts in the noisy image. Consider the parts are just similar, not identical , we need add a weight to each part. In NLM, we use Gaussian weight function to do that:

IMG_256

Figure 3.b.1 Gaussian Weight Function (i copy this on wikipedia)

h stands for standard deviation, which is a parameter we need to adjust to get the best result(we will discuss it later). In my code, ***i use 2 \* h ^ 2 instead of h ^2.***And we also need to valuate how similar two parts are. I use Gaussian weighted Euclidean distance to do that. In some materials, they use ‘Box weighted’ Euclidean distance instead.

The NLM is a very time-consuming algorithm. In my experience ,it takes at least one hour to traverse all the parts for one pixel (comparing window @3X3) and at least 2 hours with comparing window @7X7, so i use fast\_NLM afterward. The idea of fast\_NLM is the same as NLM but instead of traversing all the parts in the image for one pixel, it only processes a small region named ‘searching window’. In practice ,using search window @23X23 and comparing window @7X7 is good enough to get a satisfying outcome.

In my code, there are three parameters which can be adjusted: h,a and size(‘a’ is the standard deviation of Gaussian Kernel used in Gaussian weighted Euclidean distance). Now i will use each of the R,G,B channel of pepper as examples to show the difference between different parameter setting.

**Core code#1：Gaussian distance**

double Gaussian\_distance\_weight(int array1[], int array2[], double weight[], double SD, int size)

{

double temp = 0.0;

for (int i = 0; i < size \* size; i++)

{

temp = temp + (double)((array1[i] - array2[i]) \* (array1[i] - array2[i])) \* weight[i];

}

temp = exp(-temp / (2 \* SD \* SD));

return temp;

}

**Core code#2: NLM Filter**

void Fast\_nlm\_7\_23(int input[], int output[], int width, int height, double SD, double gaussian\_k)

{

int temp[1000 \* 1000];

int temp2[1000 \* 1000];

int temp\_m\_in[1000][1000];

int temp\_m\_out[1000][1000];

int temp\_a\_out[1000 \* 1000];

double weight\_table[100\*100];

double adjusted\_weight\_table[100 \* 100];

int window\_data1[7 \* 7];

int window\_data2[7 \* 7];

double total\_weight = 0.0;

double filtered\_value;

double count = 0;

double gaussian\_core[49] = {0};

int index;

Create\_gaussian\_core(gaussian\_core, gaussian\_k, 7);

Border\_extend(input, temp, width, height); //1

Border\_extend(temp, temp2, width + 2, height + 2); //2

Border\_extend(temp2, temp, width + 4, height + 4); //3

Border\_extend(temp, temp2, width + 6, height + 6); //4

Border\_extend(temp2, temp, width + 8, height + 8); //5

Border\_extend(temp, temp2, width + 10, height + 10); //6

Border\_extend(temp2, temp, width + 12, height + 12); //7

Border\_extend(temp, temp2, width + 14, height + 14); //8

Border\_extend(temp2, temp, width + 16, height + 16); //9

Border\_extend(temp, temp2, width + 18, height + 18); //10

Border\_extend(temp2, temp, width + 20, height + 20); //11

for (int i = 0; i < height + 22; i++)

{

for (int j = 0; j < width + 22; j++)

{

temp\_m\_in[i][j] = temp[(width + 22) \* i + j];

}

}

for (int i = 11; i < height + 11; i++)

{

count++;

printf("\n Already finish %lf percent", count / 512.0\*100.0);

for (int j = 11; j < width + 11; j++)

{

window\_data1[0] = temp\_m\_in[i - 3][j - 3];

window\_data1[1] = temp\_m\_in[i - 3][j - 2];

window\_data1[2] = temp\_m\_in[i - 3][j - 1];

window\_data1[3] = temp\_m\_in[i - 3][j];

window\_data1[4] = temp\_m\_in[i - 3][j + 1];

window\_data1[5] = temp\_m\_in[i - 3][j + 2];

window\_data1[6] = temp\_m\_in[i - 3][j + 3];

window\_data1[7] = temp\_m\_in[i - 2][j - 3];

window\_data1[8] = temp\_m\_in[i - 2][j - 2];

window\_data1[9] = temp\_m\_in[i - 2][j - 1];

window\_data1[10] = temp\_m\_in[i - 2][j];

window\_data1[11] = temp\_m\_in[i - 2][j + 1];

window\_data1[12] = temp\_m\_in[i - 2][j + 2];

window\_data1[13] = temp\_m\_in[i - 2][j + 3];

window\_data1[14] = temp\_m\_in[i - 1][j - 3];

window\_data1[15] = temp\_m\_in[i - 1][j - 2];

window\_data1[16] = temp\_m\_in[i - 1][j - 1];

window\_data1[17] = temp\_m\_in[i - 1][j];

window\_data1[18] = temp\_m\_in[i - 1][j + 1];

window\_data1[19] = temp\_m\_in[i - 1][j + 2];

window\_data1[20] = temp\_m\_in[i - 1][j + 3];

window\_data1[21] = temp\_m\_in[i][j - 3];

window\_data1[22] = temp\_m\_in[i][j - 2];

window\_data1[23] = temp\_m\_in[i][j - 1];

window\_data1[24] = temp\_m\_in[i][j];

window\_data1[25] = temp\_m\_in[i][j + 1];

window\_data1[26] = temp\_m\_in[i][j + 2];

window\_data1[27] = temp\_m\_in[i][j + 3];

window\_data1[28] = temp\_m\_in[i + 1][j - 3];

window\_data1[29] = temp\_m\_in[i + 1][j - 2];

window\_data1[30] = temp\_m\_in[i + 1][j - 1];

window\_data1[31] = temp\_m\_in[i + 1][j];

window\_data1[32] = temp\_m\_in[i + 1][j + 1];

window\_data1[33] = temp\_m\_in[i + 1][j + 2];

window\_data1[34] = temp\_m\_in[i + 1][j + 3];

window\_data1[35] = temp\_m\_in[i + 2][j - 3];

window\_data1[36] = temp\_m\_in[i + 2][j - 2];

window\_data1[37] = temp\_m\_in[i + 2][j - 1];

window\_data1[38] = temp\_m\_in[i + 2][j];

window\_data1[39] = temp\_m\_in[i + 2][j + 1];

window\_data1[40] = temp\_m\_in[i + 2][j + 2];

window\_data1[41] = temp\_m\_in[i + 2][j + 3];

window\_data1[42] = temp\_m\_in[i + 3][j - 3];

window\_data1[43] = temp\_m\_in[i + 3][j - 2];

window\_data1[44] = temp\_m\_in[i + 3][j - 1];

window\_data1[45] = temp\_m\_in[i + 3][j];

window\_data1[46] = temp\_m\_in[i + 3][j + 1];

window\_data1[47] = temp\_m\_in[i + 3][j + 2];

window\_data1[48] = temp\_m\_in[i + 3][j + 3];

index = 0;

for (int y = i - 11 + 3; y <= i + 11 - 3; y++)

{

for (int x = j - 11 + 3; x <= j + 11 - 3; x++)

{

window\_data2[0] = temp\_m\_in[y - 3][x - 3];

window\_data2[1] = temp\_m\_in[y - 3][x - 2];

window\_data2[2] = temp\_m\_in[y - 3][x - 1];

window\_data2[3] = temp\_m\_in[y - 3][x];

window\_data2[4] = temp\_m\_in[y - 3][x + 1];

window\_data2[5] = temp\_m\_in[y - 3][x + 2];

window\_data2[6] = temp\_m\_in[y - 3][x + 3];

window\_data2[7] = temp\_m\_in[y - 2][x - 3];

window\_data2[8] = temp\_m\_in[y - 2][x - 2];

window\_data2[9] = temp\_m\_in[y - 2][x - 1];

window\_data2[10] = temp\_m\_in[y - 2][x];

window\_data2[11] = temp\_m\_in[y - 2][x + 1];

window\_data2[12] = temp\_m\_in[y - 2][x + 2];

window\_data2[13] = temp\_m\_in[y - 2][x + 3];

window\_data2[14] = temp\_m\_in[y - 1][x - 3];

window\_data2[15] = temp\_m\_in[y - 1][x - 2];

window\_data2[16] = temp\_m\_in[y - 1][x - 1];

window\_data2[17] = temp\_m\_in[y - 1][x];

window\_data2[18] = temp\_m\_in[y - 1][x + 1];

window\_data2[19] = temp\_m\_in[y - 1][x + 2];

window\_data2[20] = temp\_m\_in[y - 1][x + 3];

window\_data2[21] = temp\_m\_in[y][x - 3];

window\_data2[22] = temp\_m\_in[y][x - 2];

window\_data2[23] = temp\_m\_in[y][x - 1];

window\_data2[24] = temp\_m\_in[y][x];

window\_data2[25] = temp\_m\_in[y][x + 1];

window\_data2[26] = temp\_m\_in[y][x + 2];

window\_data2[27] = temp\_m\_in[y][x + 3];

window\_data2[28] = temp\_m\_in[y + 1][x - 3];

window\_data2[29] = temp\_m\_in[y + 1][x - 2];

window\_data2[30] = temp\_m\_in[y + 1][x - 1];

window\_data2[31] = temp\_m\_in[y + 1][x];

window\_data2[32] = temp\_m\_in[y + 1][x + 1];

window\_data2[33] = temp\_m\_in[y + 1][x + 2];

window\_data2[34] = temp\_m\_in[y + 1][x + 3];

window\_data2[35] = temp\_m\_in[y + 2][x - 3];

window\_data2[36] = temp\_m\_in[y + 2][x - 2];

window\_data2[37] = temp\_m\_in[y + 2][x - 1];

window\_data2[38] = temp\_m\_in[y + 2][x];

window\_data2[39] = temp\_m\_in[y + 2][x + 1];

window\_data2[40] = temp\_m\_in[y + 2][x + 2];

window\_data2[41] = temp\_m\_in[y + 2][x + 3];

window\_data2[42] = temp\_m\_in[y + 3][x - 3];

window\_data2[43] = temp\_m\_in[y + 3][x - 2];

window\_data2[44] = temp\_m\_in[y + 3][x - 1];

window\_data2[45] = temp\_m\_in[y + 3][x];

window\_data2[46] = temp\_m\_in[y + 3][x + 1];

window\_data2[47] = temp\_m\_in[y + 3][x + 2];

window\_data2[48] = temp\_m\_in[y + 3][x + 3];

index++;

weight\_table[index] = Gaussian\_distance\_weight(window\_data1, window\_data2, gaussian\_core, SD, 7);

}

}

total\_weight = 0.0;

for (int r = 0; r < 17 \* 17; r++)

{

total\_weight += weight\_table[r];

}

for (int k = 0; k < 17 \* 17; k++)

{

adjusted\_weight\_table[k] = weight\_table[k] / total\_weight;

}

filtered\_value = 0.0;

index = 0;

for (int y1 = i - 11 + 3; y1 <= i + 11 - 3; y1++)

{

for (int x1 = j - 11 + 3; x1 <= j + 11 - 3; x1++)

{

filtered\_value = filtered\_value + (double)temp\_m\_in[y1][x1] \* adjusted\_weight\_table[index];

index++;

}

}

output[width \* (i - 11) + j - 11] = (int)filtered\_value;

}

}

}

(a) Window Size (Pepper\_G)

The size of the window doesn't impact the result a lot. When i enlarged the comparing window @9X9 and search window@47X47, there wasn't a big improvement, the PSNR just increases 0.131466 and it takes much longer time (around 8 times as @7X7 23X23). Figure 3.b.2 show the effects of different window size.



Figure 3.b.2 PSRN of G\_7X7 23X23\_28.858878(left) / G\_9X9 47X47\_28.990344(right)

That also explain why fast\_NLM works. The improvement given by enlarging the window size is too limited, so we can just traverse a region instead of the whole image without too much quality dropping.

(b) The Gaussian Kernel\_a (Pepper\_R)

I find the filter is not sensitive to this parameter either. The range of a is from 0.01(almost impulse) to 10 (very large,usually the value is around 1) but the PSNR doesn't change a lot. If you observe the images extremely carefully like a crazy, you can find that the bigger a is, more blur (very little) the result is. I find a ‘peak value’ , but overall speaking Kernel\_a doesn't matter a lot.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **a** | 0.01 | 0.1 | 0.5 | 1 | 2 | 10 |
| **PSRN** | 27.39266 | 27.39266 | 28.083777 | 27.593478 | 27.294858 | 27.030316 |

Table 3.b.1 effect of Kernel\_a



Figure 3.b.3 a=0.01 Figure 3.b.4 a=0.1



Figure 3.b.5 a=0.5 Figure 3.b.6 a=1



Figure 3.b.7 a=2 Figure 3.b.8 a=10

(c) The standard deviation of weighted Gaussian ‘h’ (Pepper\_B)

This parameter have major impact on the performance. When it is too small, then the image looks like it wasn't filtered at all. When it is too big, the noise is depressed but the image will be very blur. The shape of the PSRN-h curve is ‘hill’.

\*Notice: i use 2 \* h ^ 2 as denominator

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **h** | 0.1 | 1 | 5 | 12 | 20 | 50 |
| **PSRN** | 22.176344 | 24.996821 | 26.959023 | 27.306335 | 27.052411 | 25.278358 |

Table 3.b.2 effect of ‘h’



Figure 3.b.9 h=0.1 Figure 3.b.10 h=1 

Figure 3.b.11 h=5 Figure 3.b.12 h=12



Figure 3.b.13 h=20 Figure 3.b.14 h=50

Now i use the best result of these three channels: R\_channel with h=25, G\_channel with h=15 and B\_channel h=12 to generate the final outcome.



Figure 3.b.15 NLM-filtered Figure 3.b.16 Common-filter-filtered

Figure 3.b.15 and Figure 3.b.16 show the different outcomes between NLM filter and Common filter. We can still see the ‘ghost’ of noise in common-filter-filtered image yet the noise basically disappear in NLM-filtered image. And the edge is more clear in NLM-filtered as well.

I think the advantage come from two points:

1. NLM use the whole image (in this case: searching window) instead of a few of pixels around the target pixels, hence it can use much more redundant information in the image, which i think is the reason why it can eliminate the noise better.
2. NLM use weighted mean instead of mean(used by Linear filter), so that when the pixel has oddly high or low value, it will not effect the outcome a lot. In addition, the method it uses to compute the weight is more comprehensive(compute the similarity first and use similarity to compute Gaussian weight) than that in common filter (like Gaussian filter). That’s why it can maintain more detail at edge and make the outcome less blur.

**//Repeating job: NLM\_filter toward sailboat\_image**

Channel\_R: a = 1.2 h = 17 Size = 9x9 47x47

Channel\_G: a = 0.5 h = 21 Size = 9x9 47x47

Channel\_B: a = 0.5 h = 25 Size = 9x9 47x47

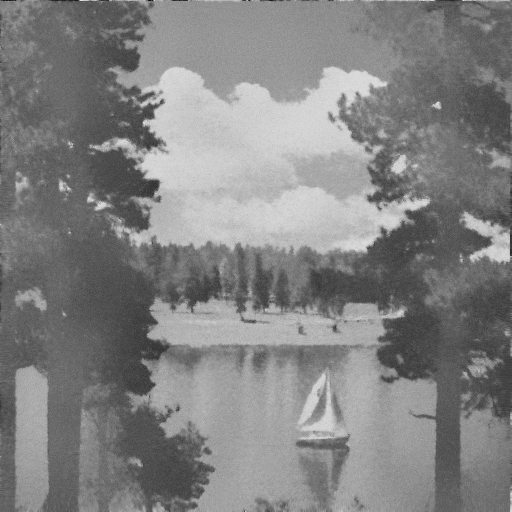


Figure 3.b.17 R\_noisy Figure 3.b.18 R\_filtered



Figure 3.b.19 G\_noisy Figure 3.b.20 G\_filtered



Figure 3.b.21 B\_noisy Figure 3.b.22 B\_filtered



Figure 3.b.23 noisy sailboat Figure 3.b.22 filtered sailboat

Although NLM has much better performance than common filters, it has two major weak point. The first one is that it can not handle impulse noise as good as Median filter. For instance ,there are still a few white points in the pepper\_B\_filtered. The second one is the complexity of the algorithm. Even to small image whose size is just 512 x 512, fast\_NLM still need at least 20 seconds to process using small searching and comparing window. Not to mention the original NLM who will traverse all the blocks in the image. (Actually the edge of filtered image will still get blur and is not as good as BM3D, but it is much better than common filters, so i won't call it a weak point)