Homework 1 for 20 spring EE569
Demosaicing, Brightness enhancement & denoising images
Chengyao Wang
69615991816
chengyao@usc.edu

Problem 1, Image demosaicing:

1. Motivation & Approaches:

Photosensitive element in modern digital cameras are usually made of CCD or CMOS, with each unit sensor being capable of capturing only one of the R / G / B color intensities. To reconstruct the colored image from the primal data recorded in CFA pattern which is H x W x 3 in dimensions, demosaicing techniques are used. Two widely used demosaicing methods used nowadays are "Bilinear Demosaicing" & "MHC Demosaicing". The former assumes linearity characteristics of the per-color pixel intensity values within a small 2D-vicinity (3 by 3) around our target location and approximate that value by the arithmetic mean of the values that belong to the same color we are interested. On the basis of former approach, MHC further introduced a quadratic calibration term based on cross-channel correlations. Detailed formulas are presented in the description of the problem and is omitted here.

Many pixel-wise operations under the scope of image processing can be expressed in "Filtering" or "Convolution" or "Sliding window", which is taking inner product in nature. The weights are often call "Filters" or "Kernels". Such expression is more related to programming and realizing such algorithms, and the ***normalized*** kernels used in the two techniques above are listed as follows:

Bilinear Interpolation

Center Color	Target Color				
		R (even / odd index)	G	B (even / odd index)	
	R	0, 0, 0 0, 1, 0 0, 0, 0	0, 1, 0 1, 0, 1 0, 1, 0	0.25, 0, 0.25 0, 0, 0 0.25, 0, 0.25	
	G	0, 0, 0 0.5, 0, 0.5 0, 0, 0	0, 0.5, 0 0, 0, 0 0, 0.5, 0	0, 0, 0 0, 1, 0 0, 0, 0	0, 0.5, 0 0.5, 0, 0.5 0, 0, 0 0, 0.5, 0
	B	0.25, 0, 0.25 0, 0, 0 0.25, 0, 0.25	0, 1, 0 1, 0, 1 0, 1, 0	0, 0, 0 0, 1, 0 0, 0, 0	

Malvar-He-Cutler (MHC) Interpolation (Coef: α, β, γ)

Target Color					
	R (even / odd index)		G	B (even / odd index)	
R	0, 0, 0, 0, 0 0, 0, 0, 0, 0 0, 0, 1, 0, 0 0, 0, 0, 0, 0 0, 0, 0, 0, 0		0, 0, $-\alpha/4$, 0, 0 0, 0, 0.25, 0, 0 $-\alpha/4$, 0.25, α, 0.25, $-\alpha/4$ 0, 0, 0.25, 0, 0 0, 0, $-\alpha/4$, 0, 0	0 0 $-\gamma/4$ 0 0 0 0.25 0 0.25 0 $-\gamma/4$ 0 γ 0 $-\gamma/4$ 0 0.25 0 0.25 0 0 0 $-\gamma/4$ 0 0	
G	0, 0, $-\beta/5$, 0, 0 0, $-\beta/5$, 0.5, $-\beta/5$, 0 $\beta/10$, 0, β, 0, $\beta/10$ 0, $-\beta/5$, 0.5, $-\beta/5$, 0 0, 0, $-\beta/5$, 0, 0	0, 0, $\beta/10$, 0, 0 0, $-\beta/5$, 0, $-\beta/5$, 0 $-\beta/5$, 0.5, β, 0.5, $-\beta/5$ 0, $-\beta/5$, 0, $-\beta/5$, 0 0, 0, $\beta/10$, 0, 0	0, 0, 0, 0, 0 0, 0, 0, 0, 0 0, 0, 1, 0, 0 0, 0, 0, 0, 0 0, 0, 0, 0, 0	0, 0, $-\beta/5$, 0, 0 0, $-\beta/5$, 0.5, $-\beta/5$, 0 $\beta/10$, 0, β, 0, $\beta/10$ 0, $-\beta/5$, 0.5, $-\beta/5$, 0 0, 0, $-\beta/5$, 0, 0	0, 0, $\beta/10$, 0, 0 0, $-\beta/5$, 0, $-\beta/5$, 0 $-\beta/5$, 0.5, β, 0.5, $-\beta/5$ 0, $-\beta/5$, 0, $-\beta/5$, 0 0, 0, $\beta/10$, 0, 0
B	0 0 $-\gamma/4$ 0 0 0 0.25 0 0.25 0 $-\gamma/4$ 0 γ 0 $-\gamma/4$ 0 0.25 0 0.25 0 0 0 $-\gamma/4$ 0 0		0 0 $-\alpha/4$ 0 0 0 0 0.25 0 0 $-\alpha/4$ 0.25 α 0.25 $-\alpha/4$ 0 0 0.25 0 0 0 0 $-\alpha/4$ 0 0	0, 0, 0, 0, 0 0, 0, 0, 0, 0 0, 0, 1, 0, 0 0, 0, 0, 0, 0 0, 0, 0, 0, 0	

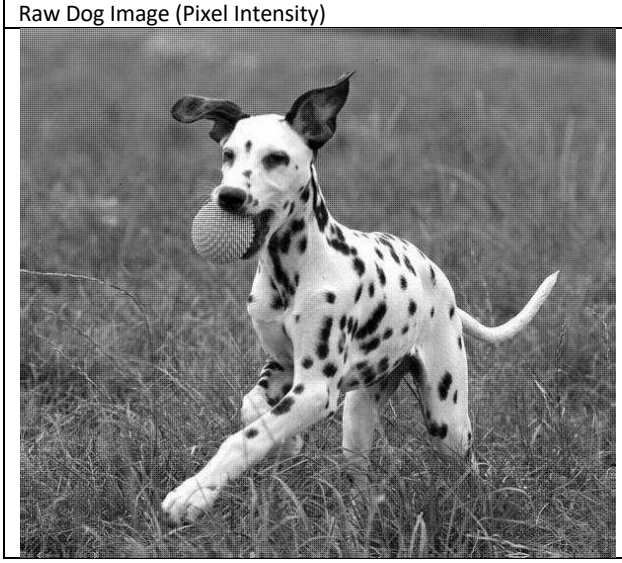
α, β, γ are coefficients controlling cross-channel correlations, are common choices are:

$$\alpha = \frac{1}{2}, \beta = \frac{5}{8}, \gamma = \frac{3}{4}$$

For corner cases of during the process of demosaicing, edge reflections are applied. Generally speaking, odd & even reflections are all accepted and feasible. But specifically, under the scenario of processing raw images, ***odd*** reflections are implemented here to conserve the pattern of Bayer array. The largest kernel used here are 5-by-5, thus only 2 rows / columns are needed to be extended. An illustration of odd reflections are as follows:

$$\begin{matrix} \grave{G}_{3.3} & \grave{R}_{3.2} & \grave{G}_{3.1} & \grave{R}_{3.2} & \grave{G}_{3.3} \\ \grave{B}_{2.3} & \grave{G}_{2.2} & \grave{B}_{2.1} & \grave{G}_{2.2} & \grave{B}_{2.3} \\ \grave{G}_{1.3} & \grave{R}_{1.2} & G_{1.1} & R_{1.2} & G_{1.3} \\ \grave{B}_{2.3} & \grave{G}_{2.2} & B_{2.1} & G_{2.2} & B_{2.3} \\ \grave{G}_{3.3} & \grave{R}_{3.2} & G_{3.1} & R_{3.2} & G_{3.3} \end{matrix}$$

2. Results & Analysis:

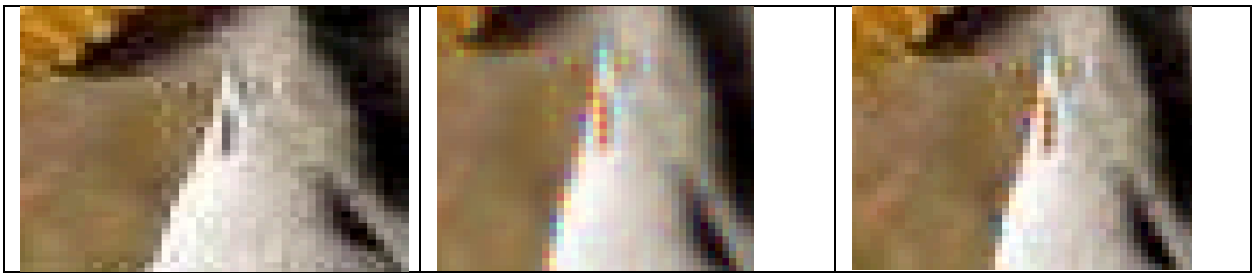
Raw Dog Image (Pixel Intensity)	Raw Dog Image (CFA colored)
	

Original Colored	Bilinear Interpolation	MHC Interpolation
		

(a.2) & (b.2):

From a larger scope, the two rather simple methods create similar results, and both bear considerable resemblance to the original image. Yet, when inspected carefully, there are still some noticeable artifacts from the two output images. Below is an example:

Original	Bilinear	MHC



This is the section of the image right under dog's chin. As can see, original image given have a vertical black (0, 0, 0) stripe with pixel-level width, while bilinear yields bright colors and MHC exploited colors too however somehow darker. This is largely resulted from large gradients in this area. For bilinear technique, the adjacent left & right columns have ~255 values thus we will never expect a bunch of ~255s averaging to ~0. From the "red" color of the stripes, one can even make arguments that the column index of the vertical stripe is likely to be odd, which is column for red pixels. This pattern is reverse "Bell" shaped in the row dimension, thus is out of the capabilities of bilinear interpolation. MHC can't also completely avoid exploiting wrong colors but is trying its best to drive the pixel to white / black using quadratic calibration introduced. It can be shown that there is further evidence for this:

Dog's Back neck

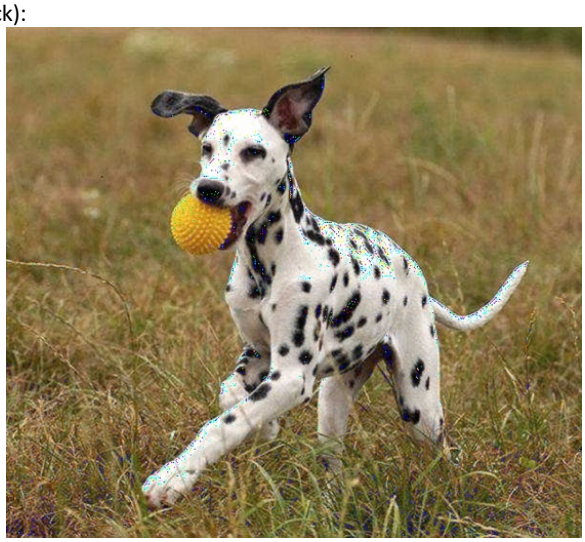
Original	Bilinear	MHC
		

Using larger non-linear kernels may solve the problem.

For MHC with its introduction to quadratic calibration term, the values may fall below 0 or beyond 255, because cross-channel strict correlation is just an assumption. Thus, we need an additional cropping step before casting floating point numbers to unsigned chars:

$$Value = \begin{cases} 0 & if\ value < 0 \\ (unsignedchar)value & if 0 \le value \le 255 \\ 255 & if\ value > 255 \end{cases}$$

Otherwise, directly casting negative floating points to unsigned char will result in 255 and outputting images like this (notice white regions on dog's back):



Problem 2, Histogram Manipulation & Brightness Enhancement:

1. Motivation & Approaches:

In gray scale images, 0 refers to black and 255 refers to white. The values of the pixel can thus be interpreted as "Pixel intensities". Single pixel doesn't make much difference, but their overall distribution characteristics will suffice to generate the notion of "brightness" or "Illuminance" of the whole picture. Colored images will more complex, but generally share the same intuition. One may argue that one image is "bright" when a considerable number of pixels processes the value over, say 200, and "dark" if majority below 50.

An image too bright or too dark can sometimes cause troubles. e.g. losing details. One post-process technique of brightness adjustment is histogram manipulation – by artificially manipulating the values or "intensities" of each individual pixel values to change to distribution from extremely biased white or black to another distribution that creates a better visual effect. Uniform distribution, as its name, is never a bad choice in most cases, and is also favored for its simplicity. There are two approaches for histogram manipulation: transfer-function-based & cumulative-probability-based histogram equalization method.

Transfer Function Based Method:

Transfer-function-based method is theoretically the same as the famous distribution sampling method – inverse CDF method. When we are changing from one random variable X_1 to another X_2, we simply utilize the following property:

$$if\ X \sim f(x)\ \Rightarrow\ F(X) \sim Uniform(0,1)$$

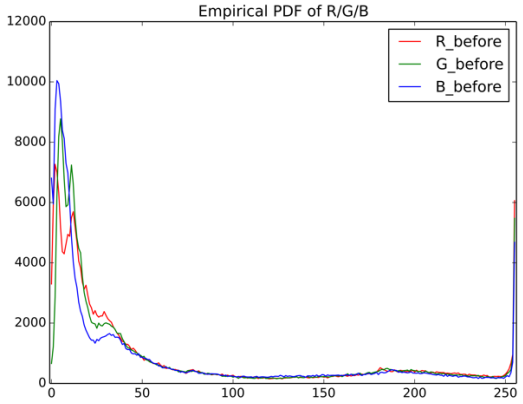
Thus:

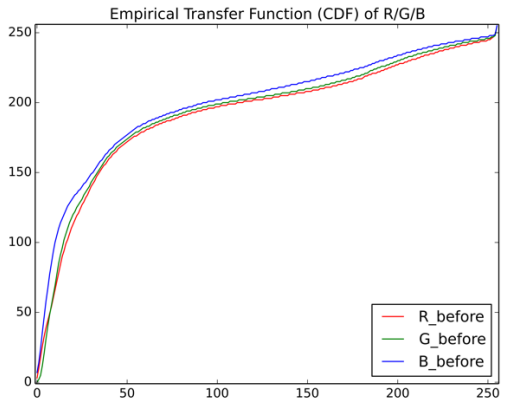
$$F_1(X_1) = F_2(X_2) = U_1 where\ U_1 \sim Uniform(0,1) \Rightarrow\ X_2 = F_2^{-1}(F_1(X_1))$$

A cumulative probability function is always invertible because its asymptoticity. We don't need the second part since our target function is uniform distribution. Empirical CDF for a discrete random variable can be calculated easily by:

$$F(X = n) = \sum_{i=MinValue}^{n} P(X = i) = \sum_{i=0}^{n} \frac{N_i}{\sum_{j=0}^{total} N_j} = \sum_{i=0}^{n} \frac{N_i}{Total\ Pixels}$$

Empirical pdf (histogram with bin width = 1) is calculated and written to a CSV file by C++ and visualized by Python. It's obvious that this image is extremely biased towards dark. With the calculation formula above, empirical CDF of per-channel pixel intensities can also be shown. Notice that the transfer function (or bijection mapping) have the same shape as CDF function, by linearly scaling the range of CDF from 0 ~ 1 to 0 ~ 255.





Cumulative Probability based method:

This method can intuitively describe as "Bucket Filling". We order the pixels by their intensity values in ascending order and create buckets with size $H \times W / 256$ and label them from 0 to 255. We "drop" pixel sequentially into the buckets, modifying its values to the bucket value and break ties randomly. In cases where the total number of pixels can't be divided by 256 because bucket sizes have to be integers, we have to use small tricks to address this issue during implementation, e.g. assigning the rest to black, make previous buckets size larger by 1 etc. But luckily, we don't have this concern with the given image.

Introduced data structures are fixed length arrays of vectors of pointers to simplify the whole process:

```
vector<int *> * myStack = new vector<int *> [256];
int * pnt = new int[2];
```

There is an array of 256 vectors, each will be storing dynamic amount of pointers which will point to a size-int[2] array. Indices of vectors in the array are its corresponding pixel intensities, and pointers in vectors will be pointing to coordinate

of a pixel of the same intensities. The use of vectors simplifies the randomization process, because std library provides API for shuffling the elements in the vector. The process is described as follows:

1. Traverse all pixels in the image, and do:

$$\boldsymbol{Mystack[Intensity[i][j]].push(p)\ where\ p -> [i,j]}$$

2. For every vector in Mystack, shuffle.

3. Initialize a counter for pointing to current bucket value, continuously do:

p=Mystack[pnt].pop()

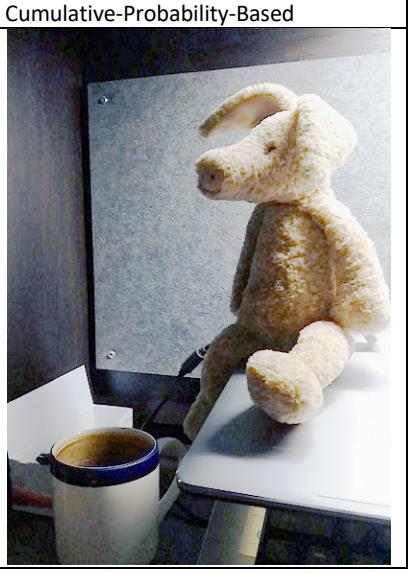
and assign:

Image_afterOp[p[0]][p[1]]=Current Bucket Value

If the Mystack[pnt] is empty during the process:

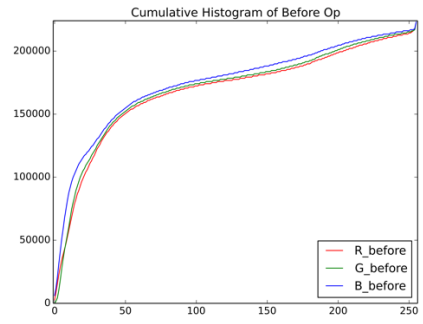
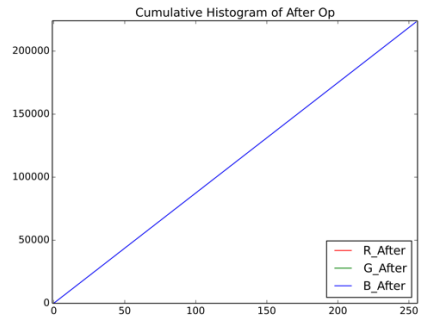
pnt +=1

2. Results & Analysis:

Original	Transfer-Function-Based	Cumulative-Probability-Based
		

Plots

Histograms of RGB channels from original image is shown above as in “empirical PDF” form. For cumulative-probability-based method, the cumulative histogram after enhancement is the same for 3 channels because we mandatorily ask a determined number of pixels for each pixel intensity values and histograms don’t care in detail how every particular pixel is assigned. Thus, cumulative histograms are shown as:

Before Op	After Op (Method B)
	

Notes:

1. One of the biggest reasons why brightness enhancement is more complex than gray scale images is that distribution of each channel is manipulated independently, **thus mix-up ratio for particular pixels is also changed which will lead to color shift.** Thanks to the image given here, empirical pdfs of RGB are quite similar thus this effect is not so obvious. Maybe looking for joint manipulation methods will be a good choice. A naïve proposal can be:

Perform dimension reduction by $I_{i,j} = max(R_{i,j}, G_{i,j}, B_{i,j})$, and redo the process above. Then, both two other channels are multiplied by the same expanding ratio of that max-channel. Idea is that the maximum intensity of among three channels of RGB may represents / dominates the brightness of that pixel.

There can be other proposals such as replacing the max with average, but it still runs into problem of color shifting when pixel intensities of some channel go beyond 255 where we need to crop down to 255. Inter-channel ratio will still be compromised.

2. **Uniform distribution is not always the best choice, or global enhancement is not always nice. The two things are not precisely the same but speaks quite the same for the following argument.** In this image, the quantity of dark pixels is too large such that those areas whose luminance is just appropriate are compelled to shift to near white. One typical example is the "leg" of the bear. It's well shown in the original picture with its correct color and appropriate luminance. But it turned white and over-lighted in the image after manipulation. In this case, we may want to maintain those areas and enhance the brightness in dark regions at the same time. Here, we can simply try out other target distributions other than uniform, or perform brightness enhancement in several local windows.

3. **Transfer function for discrete random variables are not always a surjection mapping**. This will cause problems when a huge number of pixels have their values at some particular number, usually ~0 or ~255 if are not artificially constructed. In cumulative-probability-based methods, they will be distributed randomly to "buckets" ranging from, say 50 ~ 100. In transfer-function-based methods, there could be cases where

$$F(i) = j \;\&\; F(i+1) = j + 20$$

And no pixel could take the value between $j + 1 \sim j + 19$ after manipulation. This may cause troubles in extremely over luminated or lack lamination images. Detailed effects needs further exploration.

Problem 3, Image denoising:

1. Motivation & Approaches:

The noise in "Corn_noisy" is additive gaussian white noise. Salt noise often appear to be saturated, single-colored (RGB, White, Black, Magenta, Cyan, Yellow) pixels, thus is not present in the images.

Noises are present in almost all images sampled from the environment, as a result of various reasons. Most of them can be generalized as gaussian noise, shot noise and salt noise. Gaussian noise is the most common noise and can be modeled as an additive fluctuation from the true values and that signed magnitude of perturbation follows a zero-mean, usually not standardized gaussian distribution. Approaches introduced and implemented here are rooted on the same methodology "Averaging", which from another perspective "low pass filtering". Detailed algorithms & approaches are all given in the problem statement, thus is not repeated here.

Denoising algorithms can often be generalized into "convolution operations" and "kernels", though they may not be linear, thus high-level way they're implemented and coded is generally the same as demosaicing part of the report. For evaluation metrics, since we have the noiseless image, PSNR is adopted and used for criterion of tuning parameters of included algorithms. Formula for computing PSNR is also explicitly given in the problem statement, thus is omitted here.

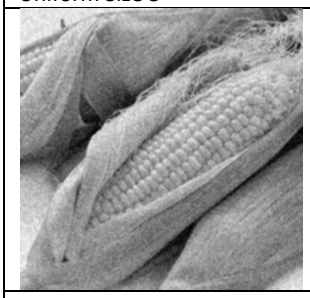
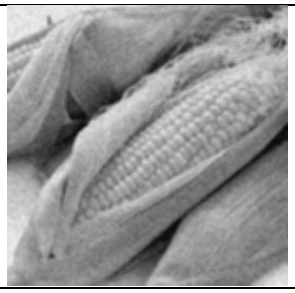
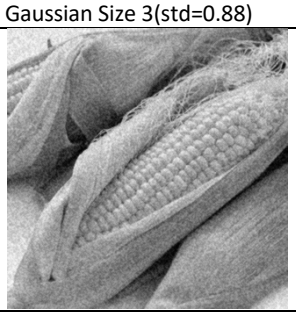
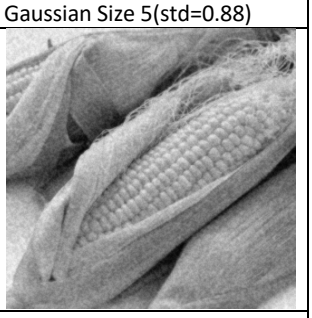
Grid search in combination with "Zooming In" is the technique implemented for tuning the parameters, but latter is done manually thus the tuning process actually took several executions of the program. Detailed process of different algorithms varies, but they can all be generalized as – iteratively do:

1. Exclusive brute force traverse the region in parameter space
2. Zoom in to the area where scores are high
3. Back to 1

2. Results & Analysis:

Original, noiseless images	Noisy Images

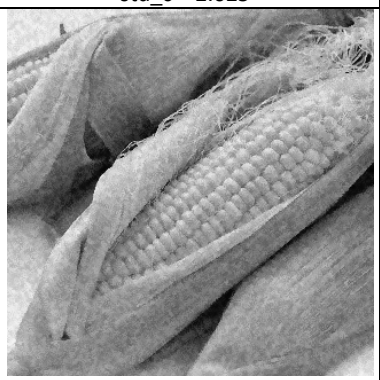
Linear Kernels

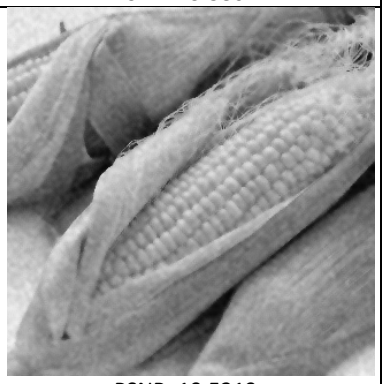
Uniform Size 3	Uniform Size 5	Gaussian Size 3(std=0.88)	Gaussian Size 5(std=0.88)
			
PSNR=19.3032	PSNR=19.0697	PSNR=19.3669	PSNR=19.4265

As the result shows, the two approaches both yield similar PSNR scores and made some improvements to the quality of the image. For uniform kernels, larger size creates more blurred image. Specifically, with the PSNR score and subjective visual impression of human eyes, we can say that size 5 kernels averages away detail information of the noisy image, thus performs worse than size 5. Also worth pointing out from this very first comparison, quantifies scores is not necessarily related to images' impression on the human eye. The two different sizes create similar PSNR scores, both in percentage and absolute value sense, yet they give a very different to humans.

Starting from gaussian kernels, correlationship between pixels are being considered and they contribute to the weight when averaging. Gaussian kernels only consider spacial correlation. The size of the kernels no longer matters that much as the uniform kernel does because the properties (shape) of gaussian distributions. Yet, we can still notice similar effects like it does in the uniform case.

It's difficult to tell the difference between uniform kernel 3 and gaussian kernel 5. But it seems that brightness of the image produced by gaussian kernel 5 is a little higher, and closer to the original image.

Bilateral Kernel 5 (Best: std_c = 2.625, std_s = 81)			
	std_c = 1.625	std_c = 2.625	std_c = 3.625
std_s = 51			
	PSNR=19.2818	PSNR=19.3866	PSNR=19.4049

std_s = 81	 PSNR=19.5656	 PSNR=19.5944	 PSNR=19.5862
std_s = 111	 PSNR=19.5744	 PSNR=19.5467	 PSNR=19.5219

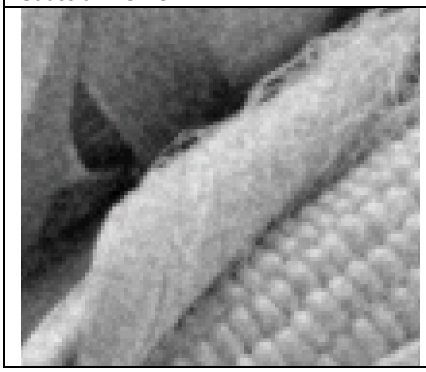
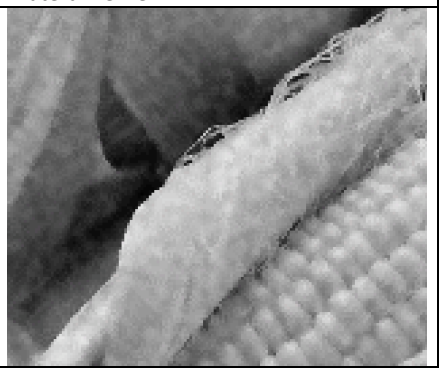
Bilateral kernels are built on top of gaussian kernels, by taking magnitude into the consideration of correlationship.

$$w(i,j,k,l) = exp\left(-\frac{(i-k)^2+(j-l)^2}{2\sigma_c^2} - \frac{\|I(i,j)-I(k,l)\|^2}{2\sigma_s^2}\right)$$

Pixels with closer intensity values are considered in a tighter relationship with the center pixel. This term is also modeled by gaussian metric. Interestingly that bilateral kernels consider spacial & magnitude independent, thus it considered uncoupled weight functions by directly multiplying them up. **The two parameters σ_c, σ_s controls the percentage of spacial & pixel intensity magnitude contributes to the overall correlationship, mostly by their relative sizes. And the overall denoising strength is controlled by their absolute sizes.** Reasons why best value pair of landed in 2s and 80s respectively is the difference in the range of the two numerators – numerator of σ_c is 0 ~ 8, and that of σ_s is $(\sim 50)^2$.

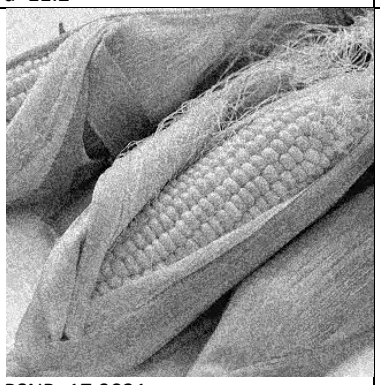
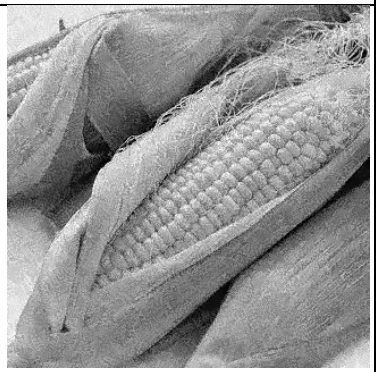
The best PSNR score of bilateral kernels is slightly better than linear kernels above, and it seemed to do better on edges, especially with two sides having large color differences. The fiber part of the corn's wrapper is a good example, sharper edges between the corn itself & the background, and more distinguishable fiber stripes. Again, with regard to the brightness of the picture, bilateral kernels are noticeably brighter that the two outputs above and closer to the original brightness.

Thanks to σ_s !

Gaussian Kernel	Bilateral Kernel
	

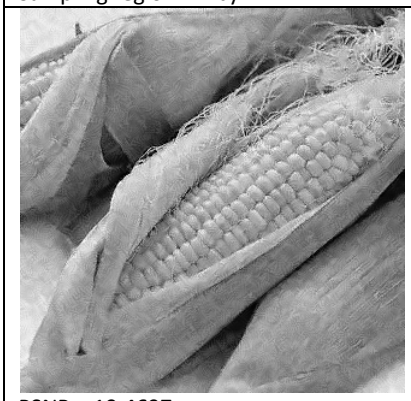
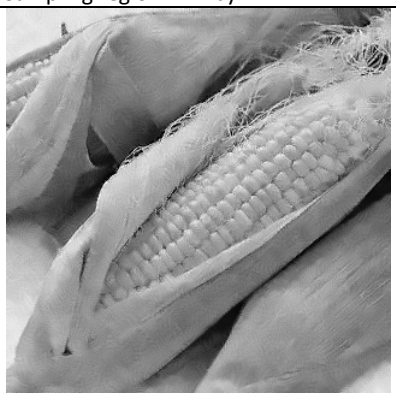
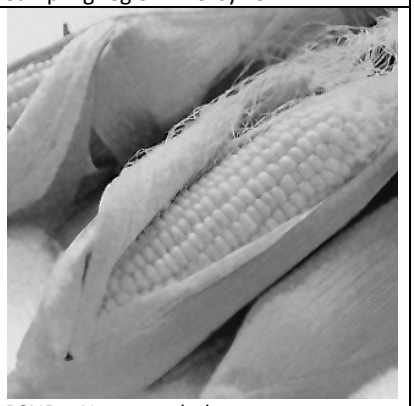
Non-local-means (NLM) extends the concept of correlationship / similarities between pixels. Gaussian & bilateral kernels only inspects the pixels themselves when evaluating their correlationship for weights, but NLM enlarges the scope to also their adjacent. When computing for a single pixel, pixels within a much wider range than commonly used 3 by 3 / 5 by 5 kernels are all considered. Algorithms for NLM is not presented here also for repeatedness. "h" explicitly controls the correlationship defined by $\left\|I\left(N_{i,j}\right)-I\left(N_{k,l}\right)\right\|_{2,a}^{2}$—smaller "h" is, more roles tighter correlated pixels play in the averaging. Notice that NLM now omits the spacial information of the two pixels of interest and has a different combination of spacial & magnitude information locally between pixels. Previously they nested in two gaussians in bilateral kernels, but in NLM, magnitude information took the form of squared difference. "a" controls the spread-ness of the gaussian weight – smaller "a" is, the more adjacent pixels contribute to the "characteristics" of the their "local", the more resemblance the two local regions have to be in order to get the center pixel of one region counted more towards the denoised result of the other center pixel.

Notice that NLM is really time consuming, thus we fixed the sampling region to 11 by 11 with cropping & no reflection extension on border pixels, and to grid search on "a" & "h" like the previous methods. And then, we change the sampling region size to see what role it plays and determine the best result of NLM. Kernel size is always set to 5. The two parameters in NLM are nested and are not equal in status like the bilateral kernel does, thus the optimal values they took this time will mean nothing.

Non-Local-Means 5 (Best: std_a = 16.2, std_h = 16.6)			
	a=11.2	a=16.2	a=21.2
h=11.6	 PSNR=17.9031	 PSNR=18.5492	 PSNR=19.1221
h=16.6	 PSNR=19.2828	 PSNR=19.7647	 PSNR=19.7506
h=21.6			

	PSNR=19.6856	PSNR=19.7425	PSNR=19.6677

On basis of the best result above

Sampling region = 7 by 7	Sampling region = 11 by 11	Sampling region = 49 by 49
		
PSNR = 19.4697	PSNR = 19.7647	PSNR = Not recorded

PSNR scores of the NLM outputs are again one step higher than bilateral kernels, and the results are visibly better than all previous methods. Though in theory, the role "h" & "a" plays are different, the effect they have on the output seems similar – increasing them increases the smoothness of the picture and erases the details left. The best one picked by PSNR also visually better than at least all the previous images. But notice that there are still some black & white patterns on the bottom part of the corn, and that gets alleviated by enlargement of the sampling region. Strictly speaking, it's the best to enlarge the sample region together with adjustments of "h" & "a" – the more candidate you're considering, the stricter you should become. And that seems to guarantee a better performance / result than our current best, but since the time complexity will increase quadratically with the region size and self-implemented algorithms is not as efficient as authorized libraries like OpenCV, this part should leave for further exploration.

Problem 2(d):

BM3D is an even better advanced denoising technique, and is state-of-the-art. It consists of two steps, which are roughly the same:

Step1: **1. Like NLM, define a sampling region and look for similar pixels, stack the local regions up.**
2. For each image, conduct 2-d transformation, like wavelet transform or DCT transform.
3. Aggregate all the regions to do a final step1 estimate.

Step2: **Now we have two images: original image & output from step 1. Perform step1 again, but this time, similar local regions are considered from both two images afore mentioned. Step 2 yields the final estimate.**

OpenCV library is used for a better run time efficiency and guarantee on the correctness of the algorithm implemented. But still BM3d is really time inefficient, which is one of its setbacks, and together with its bucket size of parameters makes it hard to implement grid search like precious algorithm does. Thus, only few pairs of parameters are chosen and tested. They are filter strength (in OpenCV API's notation "h") and stack size in mentioned in Step1.1 (in OpenCV API's notation "GroupSize"). They are roughly considered as most important parameters. Other arguments are set as follows:

```
cv::xphoto::bm3dDenoising(Image_noisy,
                          Image_afterOp,
                          20, 8, 256, 2500, 400, 16, 1, 2.0f,
                          cv::NORM_L2,
                          cv::xphoto::BM3D_STEPALL,
                          cv::xphoto::HAAR);
```

```
// @paras (default):
//      float   h = 1,
//      int     templateWindowSize = 4,
//      int     searchWindowSize = 16,
//      int     blockMatchingStep1 = 2500,
//      int     blockMatchingStep2 = 400,
//      int     groupSize = 8,
//      int     slidingStep = 1,
//      float   beta = 2.0f,
//      int     normType = cv::NORM_L2,
//      int     step = cv::xphoto::BM3D_STEPALL,
//      int     transformType = cv::xphoto::HAAR
```

Two pairs of parameter settings tried are the following:

BM3D Implementation para (h, groupSize)	
(30, 8)	(20, 16)

The right output is similar to NLM and is better than anyone else, which in some sense justifies it's title as "state-of-the-art".

Problem 2(e):

1. Salt & pepper (impulse) noise for the existence of the fully saturated, single-colored pixels. And additive gaussian noise as well.

2 & 3. For two noise types, yes. The existence of fully saturated points, in other words statistical outliers, will compromise the effect of common denoising algorithms for removing gaussian noises. A toy example is already shown in class. Except cases for whose unknown, advanced denoising algorithms that can handle the both noises one-shot, I would prefer cascading the filter in sequence by first handle Salt noises, then deal with gaussian noise. A simple, yet typical filter example for Salt noise can be median filters, which works pretty much the same as mean filters in high-level procedure.