

高等影像處理

作業#2

姓名：_____蘇柏凱_____

學號：_____111c71007_____

指導老師：_____蔣欣翰、李曉祺_____

1.1

Figure

Lena256_512_out.raw



Result

```
1.1_MSE :48.1183  
1.1_PSNR :31.3077
```

Disgussion

MSE 為 48.1183,PSNR 為 31.3077，MSE 與 PSNR 的計算僅用同一對應位置的像素點，而缺乏考量了空間關係，因此在部分時候會忽略掉一些視覺上的特徵，有可能在我們看來十分相似的圖片卻因為 shift 而導致 MSE、PSNR 表現很差，而一般來說 PSNR 在 30~40 表示圖片質量還是可以接受。

1.2

Figure

Lena512_128_blur_out.raw



Lena512_128_out.raw



Disgussion

可以在實作中發現有經過模糊處理後再下採樣的圖片整體較為自然，而沒有做過模糊處理直接下採樣的圖片在一些邊角的地方顯得較為銳利。



1.3	
Figure	
Name	lena128_256_a_out.raw
Image Neighbor (256*256)	
Name	lena128_256_b_out.raw
Image Bilinear (256*256)	
Name	lena128_256_c_out.raw

Image Bicubic (256*256)	
Name	lena128_512_a_out.raw
Image Neighbor (512*512)	
Name	lena128_256_c_out.raw

Image Bilinear (512*512)	
Name	lena128_256_c_out.raw
Image Bicubic (512*512)	

1.3 256 Neighbor 運算所花費的時間 : 0.011 S

1.3 512 Neighbor 運算所花費的時間 : 0.046 S

1.3_256_neighbor_MSE :101.649

1.3_256_neighbor_PSNR :28.0598

1.3_512_neighbor_MSE :172.788

1.3_512_neighbor_PSNR :25.7557

1.3 256 Bilinear 運算所花費的時間 : 0.019 S

1.3 512 Bilinear 運算所花費的時間 : 0.068 S

1.3_256_bilinear_MSE :87.1461

1.3_256_bilinear_PSNR :28.7283

1.3_512_bilinear_MSE :123.265

1.3_512_bilinear_PSNR :27.2224

1.3 256 Bicubic 運算所花費的時間 : 0.059 S

1.3 256 Bicubic 運算所花費的時間 : 0.227 S

1.3_256_bicubic_MSE :72.8098

1.3_256_bicubic_PSNR :29.5089

1.3_512_bicubic_MSE :113.568

1.3_512_bicubic_PSNR :27.5783

Discussion

利用 Neighbor 128x128->256x256 的圖片 MSE 為 101.649,PSNR 為 28.0598,運算時間為 0.011 秒

利用 Neighbor 128x128->512x512 的圖片 MSE 為 172.788,PSNR 為 25.7557,運算時間為 0.046 秒

利用 Bilinear 128x128->256x256 的圖片 MSE 為 87.1461,PSNR 為 28.7283,運算時間為 0.019 秒

利用 Bilinear 128x128->512x512 的圖片 MSE 為 123.265,PSNR 為 27.2224,運算時間為 0.068 秒

利用 Bicubic 128x128->256x256 的圖片 MSE 為 72.8098,PSNR 為 29.5089,運算時間為 0.059 秒



利用 Bicubic 128x128->512x512 的圖片 MSE 為 113.568,PSNR 為 27.783,運算時間為 0.227 秒

可以發現 bicubic 的效果>bilinear 的效果>neighbor 的效果，不論是在 MSE 或是在 PSNR 上的指標都可以看出這個特性，但是也同時可以發現在肉眼看起來 bilinear 會有格子點的暈邊現象，就肉眼看起來的效果並不如 neighbor，這些演算法都有自己的侷限性，當圖片被放大到一定程度後三個演算法的差異

會被縮小，而同時也可以發現 **bicubic** 視效果最好、計算量最大的演算法，在計算時間上 **bicubic>bilinear>neighbor** 的計算時間。



1.4

Figure



Name	Lena512_384_a_out.raw	
Image Neighbor		
Name	Lena512_384_b_out.raw	
Image Bilinear		

Name	Lena512_384_c_out.raw	
Image Bicubic		
<div>1.4 Neighbor運算所花費的時間：0.023 S 1.4 Bilinear運算所花費的時間：0.04 S 1.4 Bicubic運算所花費的時間：0.137 S</div>		
Disgussion		
<div>1.4 Neighbor 運算所花費的時間：0.023 S 1.4 Bilinear 運算所花費的時間：0.04 S 1.4 Bicubic 運算所花費的時間：0.137 S</div> <p>從肉眼看圖片的品質可以發現 bicubic 的效果與 bilinear 的效果相近都優於 neighbor 的效果，但是也同時可以發現這些演算法都有自己的侷限性，以 bilinear 跟 bicubic 為例可以發現在邊緣的處理上採用不同方式會有不同效果，如果是採用補 0 的話圖片邊緣會顯得較為暗淡，而我採用的是依據離邊緣最近的 block 計算，可以發現在邊緣會有一些輕微鋸齒狀，而在計算的時間也可以發現 bicubic>bilinear>neighbor 的計算時間。。</p>		

1.5	
Figure	
	Neighbor
Name	lena256_576_384_a_out
Image (↑ 2.25 ↓ 1.5) Neighbor	
Name	lena256_128_384_a_out
Image (↓ 1.5 ↑ 2.25) Neighbor	

Name	lena256_384_a_out
Image (↑ 1.5) Neighbor	
	Bilinear
Name	lena256_576_384_b_out
Image (↑ 2.25 ↓ 1.5) Bilinear	

Name	lena256_128_384_b_out
<p>Image (↓ 1.5 ↑ 2.25)</p> <p>Bilinear</p>	
Name	lena256_384_b_out
<p>Image (↑ 1.5)</p> <p>Bilinear</p>	

	Bicubic
Name	lena256_576_384_c_out
Image (↑ 2.25 ↓ 1.5) Bicubic	
Name	lena256_128_384_c_out
Image (↓ 1.5 ↑ 2.25) Bicubic	

Name	lena256_384_c_out
Image (↑ 1.5) Bicubic	
<p>1.5 上2.25下1.5 Neighbor 運算所花費的時間：0.082 S</p> <p>1.5 上2.25下1.5 Bilinear 運算所花費的時間：0.12 S</p> <p>1.5 上2.25下1.5 Bicubic 運算所花費的時間：0.415 S</p> <p>1.5 下1.5上2.25 Neighbor 運算所花費的時間：0.028 S</p> <p>1.5 下1.5上2.25 Bilinear 運算所花費的時間：0.047 S</p> <p>1.5 下1.5上2.25 Bicubic 運算所花費的時間：0.153 S</p> <p>1.5 上1.5 Neighbor 運算所花費的時間：0.023 S</p> <p>1.5 上1.5 Bilinear 運算所花費的時間：0.037 S</p> <p>1.5 上1.5 Bicubic 運算所花費的時間：0.128 S</p>	
Disgussion	
<p>1.5 ↑ 2.25 ↓ 1.5 Neighbor 運算所花費的時間：0.082 S</p> <p>1.5 ↑ 2.25 ↓ 1.5 Bilinear 運算所花費的時間：0.12 S</p> <p>1.5 ↑ 2.25 ↓ 1.5 Bicubic 運算所花費的時間：0.415 S</p> <p>1.5 ↓ 1.5 ↑ 2.25 Neighbor 運算所花費的時間：0.028 S</p>	

1.5 ↓ 1.5 ↑ 2.25 Bilinear 運算所花費的時間：0.047 S

1.5 ↓ 1.5 ↑ 2.25 Bicubic 運算所花費的時間：0.153 S

1.5 ↑ 1.5 Neighbor 運算所花費的時間：0.023 S

1.5 ↑ 1.5 Bilinear 運算所花費的時間：0.037 S




1.5 ↑ 1.5 Bicubic 運算所花費的時間：0.128 S

可以發現先縮小再放大的圖片在細節上損失最多，畫面整體也較模糊，可能是因為資料像下降維的壓縮破壞掉一部分資訊，而再放大時這些受到破壞的資訊就會被凸顯出來，而先放大再縮小的圖片在細節上較為銳利，畫面整體比起直接放大的模糊一些，可能是因為在縮小時並沒有先作模糊處理，而細部在放大時被凸顯導致，而在計算的時間也可以發現

bicubic>bilinear>neighbor 的計算時間。

2.1

Figure

	D4	D8	Dm
72			
總步數	39	30	39

Traveled path:
 (0, 0) → (0, 1) → (0, 2) → (0, 3) → (0, 4) → (0, 5) → (0, 6) → (0, 7) → (0, 8) → (1, 8) → (2, 8) → (3, 8) → (4, 8) → (4, 7) → (4, 6) → (3, 6) → (2, 6) → (2, 5) → (2, 4) → (3, 4) → (4, 4) → (4, 3) → (5, 3) → (6, 3) → (6, 2) → (6, 1) → (6, 0) → (7, 0) → (8, 0) → (9, 0) → (9, 1) → (9, 2) → (9, 3) → (9, 4) → (9, 5) → (9, 6) → (9, 7) → (9, 8) → (9, 9)
 總步數 :39
 Traveled path:
 (0, 0) → (0, 1) → (0, 2) → (0, 3) → (0, 4) → (0, 5) → (0, 6) → (0, 7) → (1, 8) → (2, 8) → (3, 8) → (4, 7) → (3, 6) → (2, 5) → (3, 4) → (4, 3) → (5, 3) → (6, 2) → (6, 1) → (7, 0) → (8, 0) → (9, 1) → (9, 2) → (9, 3) → (9, 4) → (9, 5) → (9, 6) → (9, 7) → (9, 8) → (9, 9)
 總步數 :30
 Traveled path:
 (0, 0) → (0, 1) → (0, 2) → (0, 3) → (0, 4) → (0, 5) → (0, 6) → (0, 7) → (0, 8) → (1, 8) → (2, 8) → (3, 8) → (4, 8) → (4, 7) → (4, 6) → (3, 6) → (2, 6) → (2, 5) → (2, 4) → (3, 4) → (4, 4) → (4, 3) → (5, 3) → (6, 3) → (6, 2) → (6, 1) → (6, 0) → (7, 0) → (8, 0) → (9, 0) → (9, 1) → (9, 2) → (9, 3) → (9, 4) → (9, 5) → (9, 6) → (9, 7) → (9, 8) → (9, 9)
 總步數 :39

Disgussion

D4 Traveled path:

(0, 0) → (0, 1) → (0, 2) → (0, 3) → (0, 4) → (0, 5) → (0, 6) → (0, 7) → (0, 8) → (1, 8) → (2, 8) → (3, 8) → (4, 8) → (4, 7) → (4, 6) → (3, 6) → (2, 6) → (2, 5) → (2, 4) → (3, 4) → (4, 4) → (4, 3) → (5, 3) → (6, 3) → (6, 2) → (6, 1) → (6, 0) → (7, 0) → (8, 0) → (9, 0) → (9, 1) → (9, 2) → (9, 3) → (9, 4) → (9, 5) → (9, 6) → (9, 7) → (9, 8) → (9, 9)

總步數 :39

D8 Traveled path:

(0, 0) → (0, 1) → (0, 2) → (0, 3) → (0, 4) → (0, 5) → (0, 6) → (0, 7) → (1, 8) → (2, 8) → (3, 8) → (4, 7) → (3, 6) → (2, 5) → (3, 4) → (4, 3) → (5, 3) → (6, 2) → (6, 1) → (7, 0) → (8, 0) → (9, 1) → (9, 2) → (9, 3) → (9, 4) → (9, 5) → (9, 6) → (9, 7) → (9, 8) → (9, 9)

總步數 :30

Dm Traveled path:

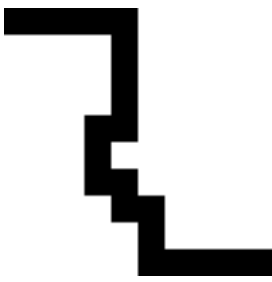

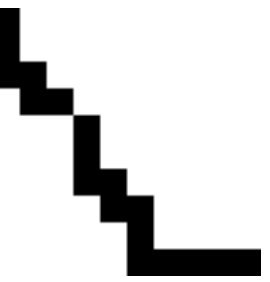
(0, 0) → (0, 1) → (0, 2) → (0, 3) → (0, 4) → (0, 5) → (0, 6) → (0, 7) → (0, 8) → (1, 8) → (2, 8) → (3, 8) → (4, 8) → (4, 7) → (4, 6) → (3, 6) → (2, 6) → (2, 5) → (2, 4) → (3, 4) → (4, 4) → (4, 3) → (5, 3) → (6, 3) → (6, 2) → (6, 1) → (6, 0) → (7, 0) → (8, 0) → (9, 0) → (9, 1) → (9, 2) → (9, 3) → (9, 4) → (9, 5) → (9, 6) → (9, 7) → (9, 8) → (9, 9)

總步數 :39

這是一個標準的路徑規劃題目，起初採用 Dynamic Programming 的方式，採用 Dynamic Programming 的方式可以很輕鬆地找到最短路徑長度，但是卻較難還原出行走路徑，於是改成採用 BFS(寬度優先演算法)把每一個路徑 push 進 vector 中並同時記錄已經走過的路徑直到走到終點為止，路徑則依據走過的路徑步數進行回推，找到步數-1 且符合 D4/D8/Dm 規則的路徑，並進行記錄，圖片部分用 xnview 的 NN 放大為 100*100 並儲存。

2.2

Figure

	D4	D8	Dm
{72,145}			
總步數	21	12	18

```

Traveled path:
(0, 0) -> (0, 1) -> (0, 2) -> (0, 3) -> (0, 4) -> (1, 4) -> (2, 4) -> (3, 4) -> (4, 4) -> (4, 3) -> (5, 3) -> (6, 3) ->
(6, 4) -> (7, 4) -> (7, 5) -> (8, 5) -> (9, 5) -> (9, 6) -> (9, 7) -> (9, 8) -> (9, 9)
總步數 :21
Traveled path:
(0, 0) -> (1, 0) -> (2, 1) -> (3, 2) -> (4, 3) -> (5, 3) -> (6, 4) -> (7, 5) -> (8, 6) -> (9, 7) -> (9, 8) -> (9, 9)
總步數 :12
Traveled path:
(0, 0) -> (1, 0) -> (2, 0) -> (2, 1) -> (3, 1) -> (3, 2) -> (4, 3) -> (5, 3) -> (6, 3) -> (6, 4) -> (7, 4) -> (7, 5) ->
(8, 5) -> (9, 5) -> (9, 6) -> (9, 7) -> (9, 8) -> (9, 9)
總步數 :18

```

Disgussion

D4 Traveled path:

(0, 0) -> (0, 1) -> (0, 2) -> (0, 3) -> (0, 4) -> (1, 4) -> (2, 4) -> (3, 4) -> (4, 4) -> (4, 3) -> (5, 3) -> (6, 3) -> (6, 4) -> (7, 4) -> (7, 5) -> (8, 5) -> (9, 5) -> (9, 6) -> (9, 7) -> (9, 8) -> (9, 9)

總步數 :21

D8 Traveled path:

(0, 0) -> (1, 0) -> (2, 1) -> (3, 2) -> (4, 3) -> (5, 3) -> (6, 4) -> (7, 5) -> (8, 6) -> (9, 7) -> (9, 8) -> (9, 9)

總步數 :12

Dm Traveled path:

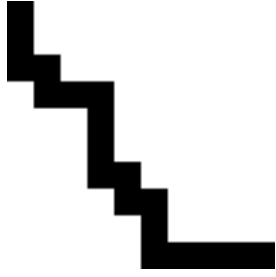
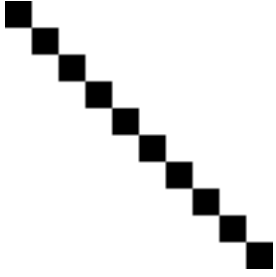

(0, 0) -> (1, 0) -> (2, 0) -> (2, 1) -> (3, 1) -> (3, 2) -> (3, 3) -> (4, 3) -> (5, 3) -> (6, 3) -> (6, 4) -> (7, 4) -> (7, 5) -> (8, 5) -> (9, 5) -> (9, 6) -> (9, 7) -> (9, 8) -> (9, 9)

總步數 :18

這是一個標準的路徑規劃題目，改成採用 BFS(寬度優先演算法)把每一個路徑 push 進 vector 中並同時記錄已經走過的路徑直到走到終點為止，路徑則依據走過的路徑步數進行回推，找到步數-1 且符合 D4/D8/Dm 規則的路徑，並進行記錄，圖片部分用 xview 的 NN 放大為 100*100 並儲存。

2.3

Figure

	D4	D8	Dm
{72,145,218 }			
總步數	19	10	16

Traveled path:
(0, 0) -> (1, 0) -> (2, 0) -> (2, 1) -> (3, 1) -> (3, 2) -> (3, 3) -> (4, 3) -> (5, 3) -> (6, 3) -> (6, 4) -> (7, 4) -> (7, 5) -> (8, 5) -> (9, 5) -> (9, 6) -> (9, 7) -> (9, 8) -> (9, 9)
總步數 :19
Traveled path:
(0, 0) -> (1, 1) -> (2, 2) -> (3, 3) -> (4, 4) -> (5, 5) -> (6, 6) -> (7, 7) -> (8, 8) -> (9, 9)
總步數 :10
Traveled path:
(0, 0) -> (1, 0) -> (2, 0) -> (2, 1) -> (3, 1) -> (3, 2) -> (3, 3) -> (4, 3) -> (5, 3) -> (5, 4) -> (5, 5) -> (6, 6) -> (7, 7) -> (8, 8) -> (9, 8) -> (9, 9)
總步數 :16

Disgussion

D4 Traveled path:

(0, 0) -> (1, 0) -> (2, 0) -> (2, 1) -> (3, 1) -> (3, 2) -> (3, 3) -> (4, 3) -> (5, 3) -> (6, 3) -> (6, 4) -> (7, 4) -> (7, 5) -> (8, 5) -> (9, 5) -> (9, 6) -> (9, 7) -> (9, 8) -> (9, 9)

總步數 :19

D8 Traveled path:

(0, 0) -> (1, 1) -> (2, 2) -> (3, 3) -> (4, 4) -> (5, 5) -> (6, 6) -> (7, 7) -> (8, 8) -> (9, 9)

總步數 :10

Dm Traveled path:

(0, 0) -> (1, 0) -> (2, 0) -> (2, 1) -> (3, 1) -> (3, 2) -> (3, 3) -> (4, 3) -> (5, 3) -> (5, 4) -> (5, 5) -> (6, 6) -> (7, 7) -> (8, 8) -> (9, 8) -> (9, 9)

總步數 :16

這是一個標準的路徑規劃題目，是採用 BFS(寬度優先演算法)把每一個路徑 push 進 vector 中並同時記錄已經走過的路徑直到走到終點為止，路徑則依據走過的路徑步數進行回推，找到步數-1 且符合 D4/D8/Dm 規則的路徑，並進行記錄，圖片部分用 xview 的 NN 放大為 100*100 並儲存。

Figure

Lena256
Shift 0 bit




lena_gray_0bits.raw

```
3_img_lena_0_bits_MSE :0  
3_img_lena_0_bits_PSNR :inf
```

Lena256
Shift 1 bit



lena_gray_1bits.raw

	<pre>3_img_lena_1_bits_MSE :0.506226 3_img_lena_1_bits_PSNR :51.0874</pre>
Lena256 Shift 2 bit	 <pre>lena_gray_2bits.raw 3_img_lena_2_bits_MSE :1.66493 3_img_lena_2_bits_PSNR :45.9168</pre>
Lena256 Shift 3 bit	 <pre>lena_gray_3bits.raw 3_img_lena_3_bits_MSE :7.63368 3_img_lena_3_bits_PSNR :39.3035</pre>

Lena256
Shift 4 bit



lena_gray_4bits.raw

```
3_img_lena_4_bits_MSE :33.871  
3_img_lena_4_bits_PSNR :32.8325
```

Lena256
Shift 5 bit



lena_gray_5bits.raw

```
3_img_lena_5_bits_MSE :152.171  
3_img_lena_5_bits_PSNR :26.3075
```

Lena256
Shift 6 bit



lena_gray_6bits.raw

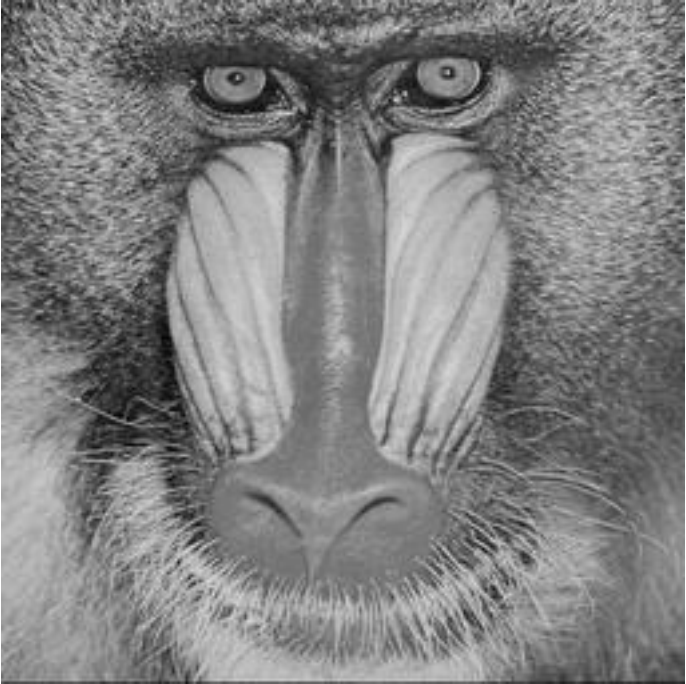
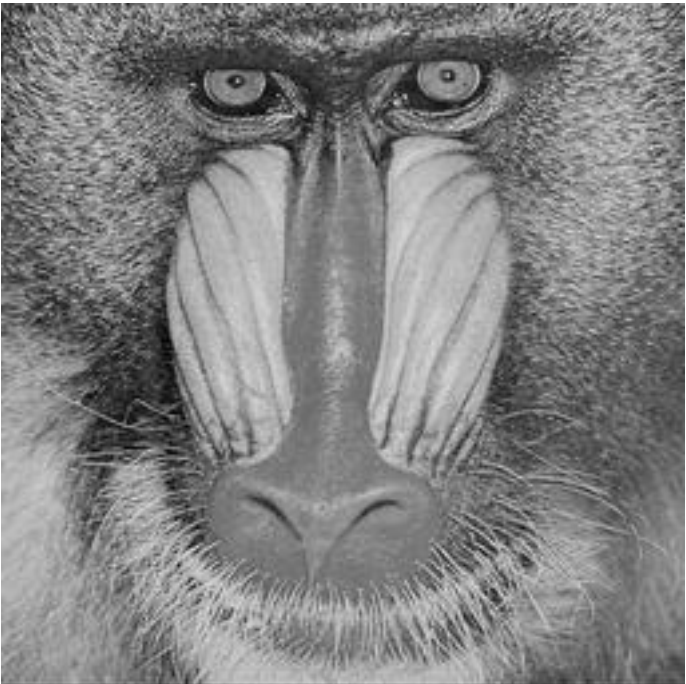
```
3_img_lena_6_bits_MSE :1038.92  
3_img_lena_6_bits_PSNR :17.965
```

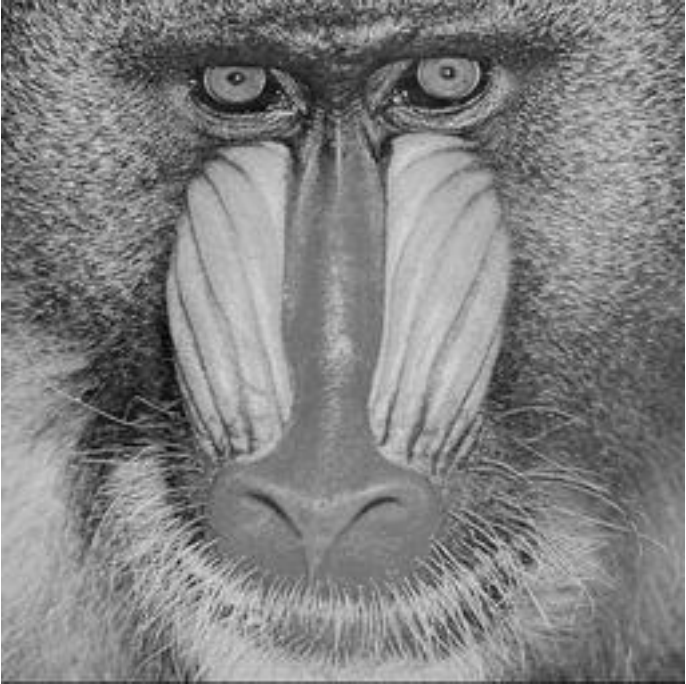
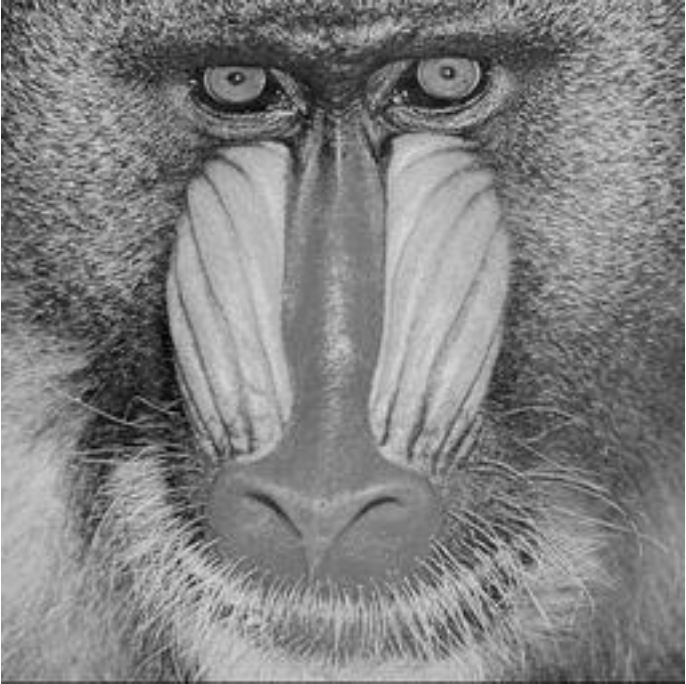
Lena256
Shift 7 bit

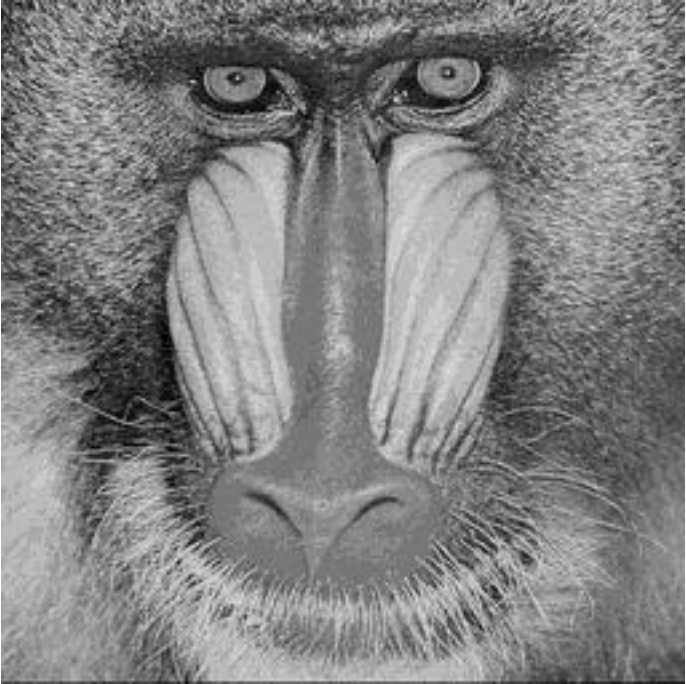
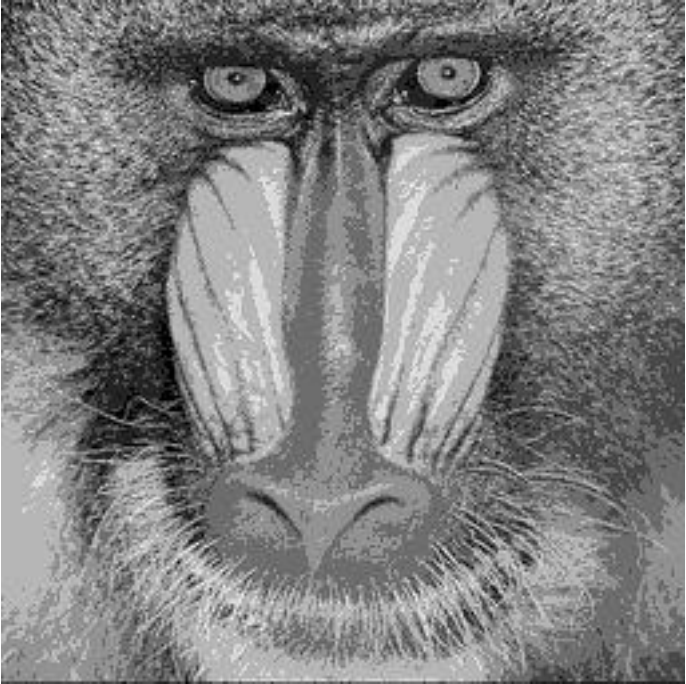


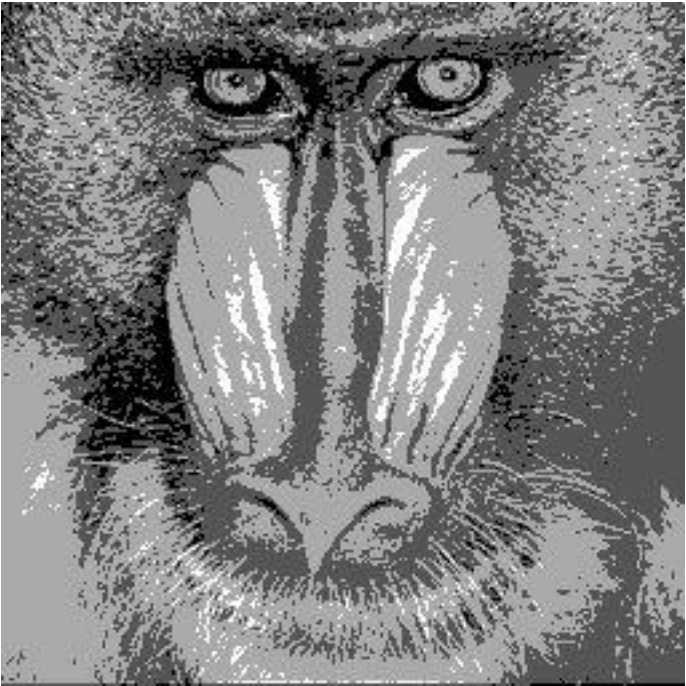
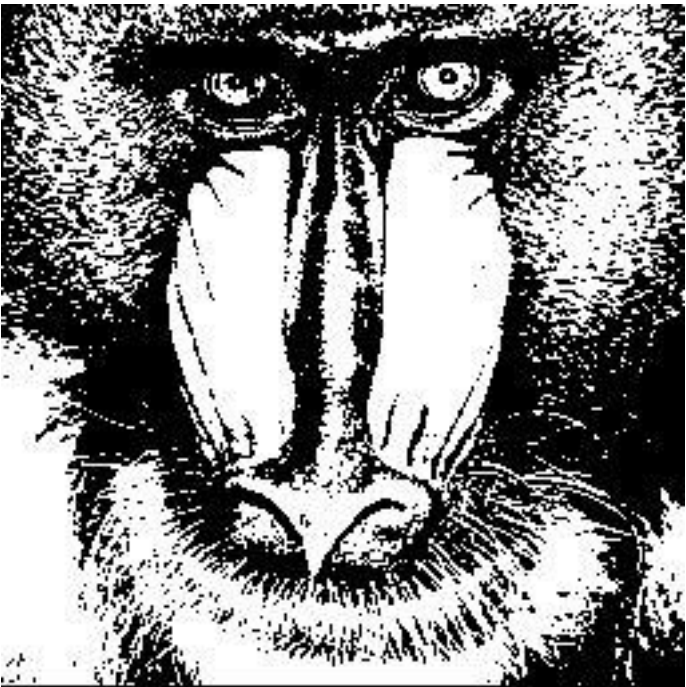
lena_gray_7bits.raw

```
3_img_lena_7_bits_MSE :8493.91  
3_img_lena_7_bits_PSNR :8.83973
```

<p>Baboon256 Shift 0 bit</p>	 <p data-bbox="799 920 1110 954">baboon_gray_0bits.raw</p> <pre data-bbox="667 965 1246 1048">3_img_baboon_0_bits_MSE :0 3_img_baboon_0_bits_PSNR :inf</pre>
<p>Baboon256 Shift 1 bit</p>	 <p data-bbox="799 1780 1110 1814">baboon_gray_1bits.raw</p> <pre data-bbox="627 1825 1286 1908">3_img_baboon_1_bits_MSE :0.507782 3_img_baboon_1_bits_PSNR :51.074</pre>

<p>Baboon256 Shift 2 bits</p>	 <p data-bbox="802 920 1110 954">baboon_gray_2bits.raw</p> <pre data-bbox="620 967 1294 1048">3_img_baboon_2_bits_MSE :1.56209 3_img_baboon_2_bits_PSNR :46.1937</pre>
<p>Baboon256 Shift 3 bits</p>	 <p data-bbox="802 1783 1110 1816">baboon_gray_3bits.raw</p> <pre data-bbox="620 1830 1294 1910">3_img_baboon_3_bits_MSE :6.8869 3_img_baboon_3_bits_PSNR :39.7506</pre>

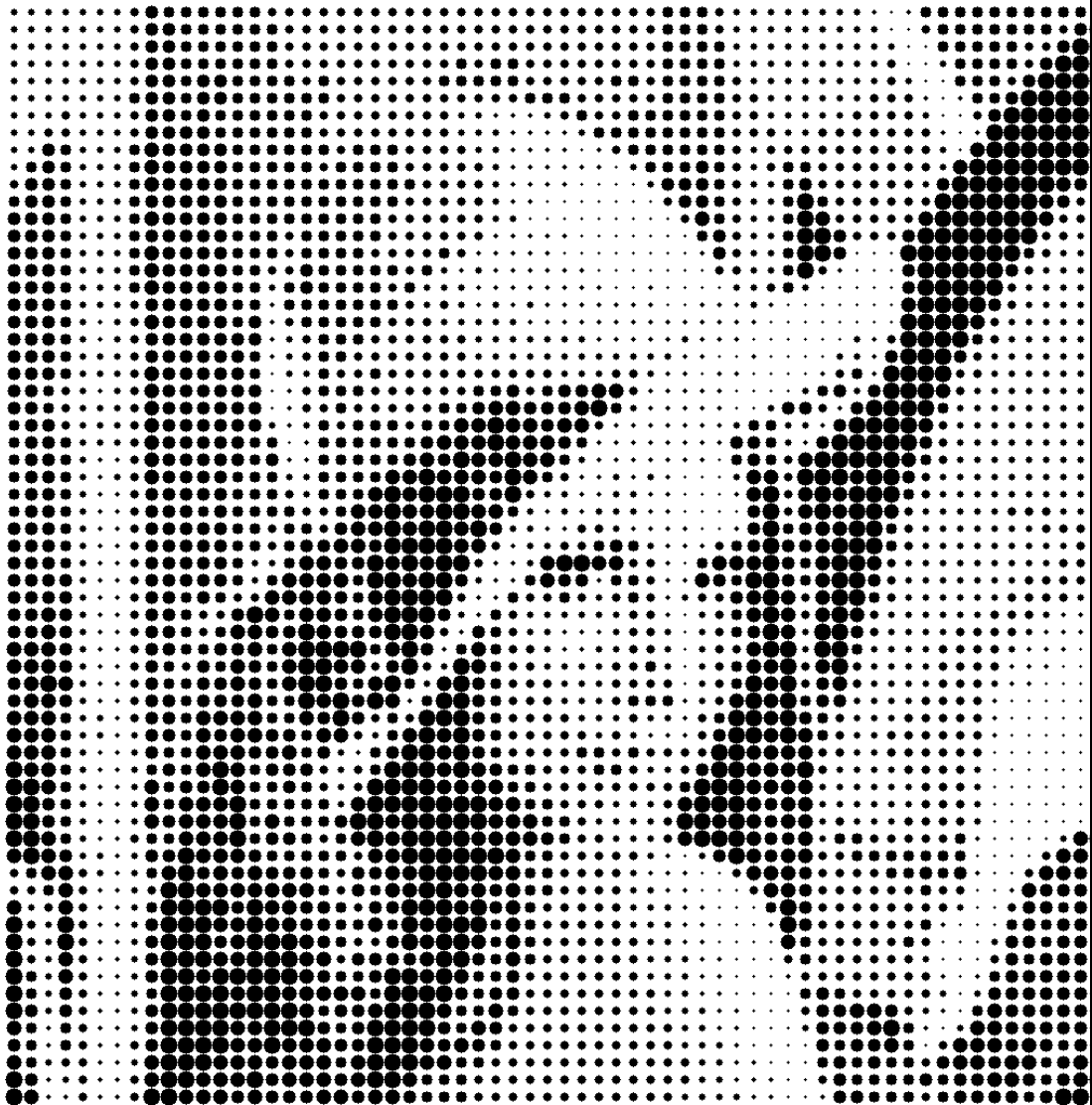
<p>Baboon256 Shift 4 bits</p>	 <p>baboon_gray_4bits.raw</p> <pre>3_img_baboon_4_bits_MSE :29.4662 3_img_baboon_4_bits_PSNR :33.4376</pre>
<p>Baboon256 Shift 5 bits</p>	 <p>baboon_gray_5bits.raw</p> <pre>3_img_baboon_5_bits_MSE :139.663 3_img_baboon_5_bits_PSNR :26.68</pre>

<p>Baboon256 Shift 6 bits</p>	 <p>baboon_gray_6bits.raw</p> <pre>3_img_baboon_6_bits_MSE :759.58 3_img_baboon_6_bits_PSNR :19.3251</pre>
<p>Baboon256 Shift 7 bits</p>	 <p>baboon_gray_7bits.raw</p> <pre>3_img_baboon_7_bits_MSE :9678.37 3_img_baboon_7_bits_PSNR :8.27278</pre>
<p>Disgussion</p>	

img_lena 保留 8Bits MSE :0
img_lena 保留 8Bits PSNR :inf
img_baboon 保留 8Bits MSE :0.
img_baboon 保留 8Bits PSNR :inf
img_lena 保留 7Bits MSE :0.506226
img_lena 保留 7Bits PSNR :51.0874
img_baboon 保留 7Bits MSE :0.507782
img_baboon 保留 7Bits PSNR :51.074
img_lena 保留 6Bits MSE :1.66493
img_lena 保留 6BitsPSNR :45.9168
img_baboon_保留 6BitsMSE :1.56209
img_baboon 保留 6BitsPSNR :46.1937
img_lena 保留 5BitsMSE :7.63368
img_lena 保留 5BitsPSNR :39.3035
img_baboon 保留 5BitsMSE :6.8869
img_baboon 保留 5BitsPSNR :39.7506
img_lena 保留 4BitsMSE :33.871
img_lena 保留 4BitsPSNR :32.8325
img_baboon 保留 4BitsMSE :29.4662
img_baboon 保留 4BitsPSNR :33.4376
img_lena 保留 3BitsMSE :152.171
img_lena 保留 3BitsPSNR :26.3075
img_baboon 保留 3BitsMSE :139.663
img_baboon 保留 3BitsPSNR :26.68
img_lena 保留 2BitsMSE :1038.92
img_lena 保留 2BitsPSNR :17.965
img_baboon 保留 2BitsMSE :759.58
img_baboon 保留 2BitsPSNR :19.3251
img_lena 保留 1BitMSE :8493.91
img_lena 保留 1BitPSNR :8.83973
img_baboon 保留 1BitMSE :9678.37
img_baboon 保留 1BitPSNR :8.27278

這一題的儲存上因為常見的儲存形式都是用 8-bits 而 xview 也只能夠見到 8-bits 的影像，在影像儲存理論上每少一個 bit 儲存空間也會少一半，但為了可視化 raw 檔案在儲存中實際上都還是採用 8-bits，而為了可視化，在右移後我再根據數字將其還原在 0-255 中應該有的數字大小，以右移 7bits 為例只剩下 0、1，則變成 0、255，而 shift 0 bit 表示保留 8bits，與原圖無異。

Figure



lena1024_cir_out.raw

Disgussion

這一題把 1024*1024 的圖像變成 32*32 個 block，每一個 block 中有 16*16 個 pixels，本題使用 open-cv 協助畫圓，並依據每一個 block 中 gray-level 的平均分成 8 種不同半徑，並在最後把結果存回 raw 檔中，可以發現在把圖像縮小的時候看起來與原圖更為接近，透過不同圓形大小達到視覺上看起來不同的灰階是一個十分有趣的視覺效果。