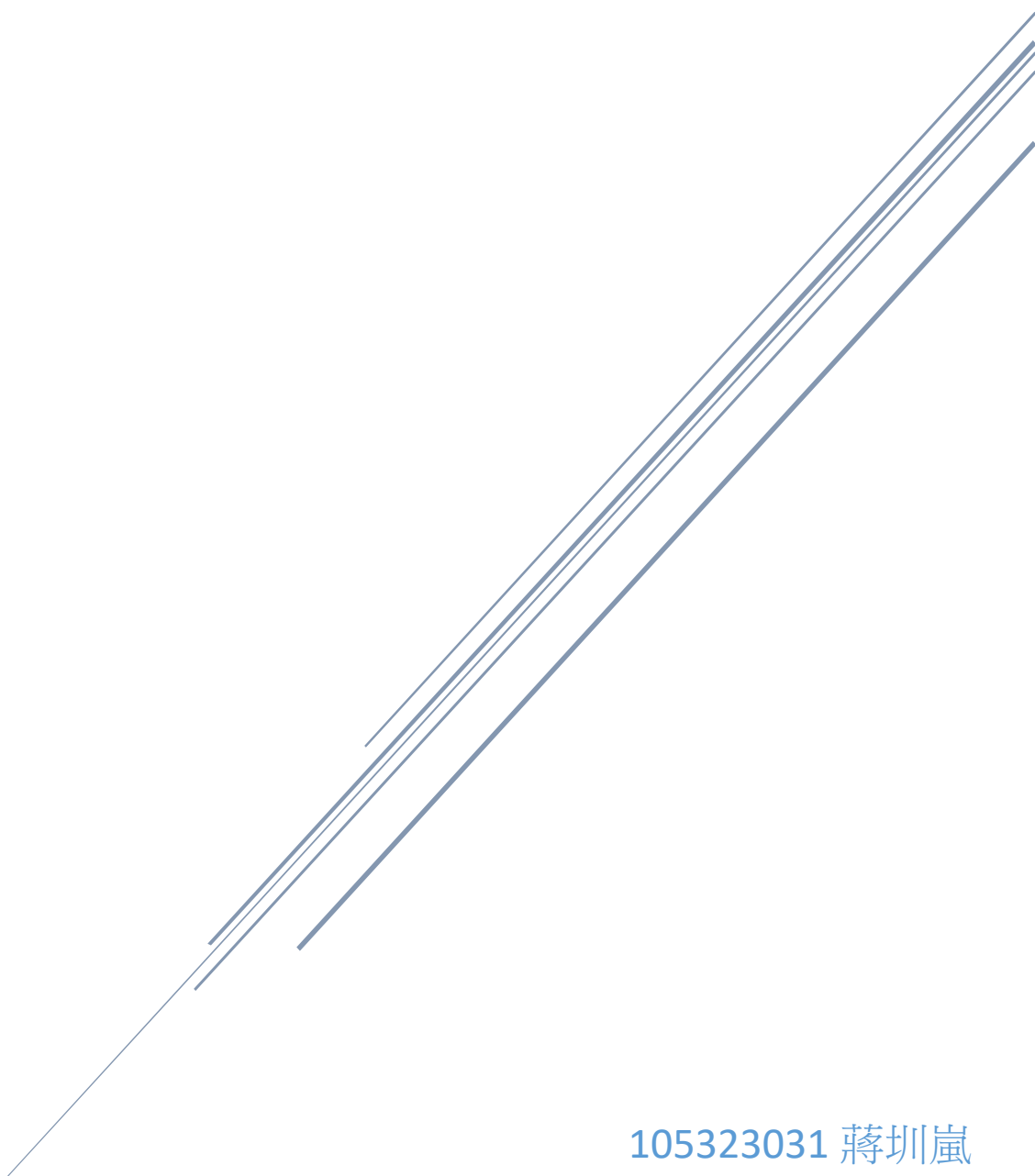


# 數位影像處理

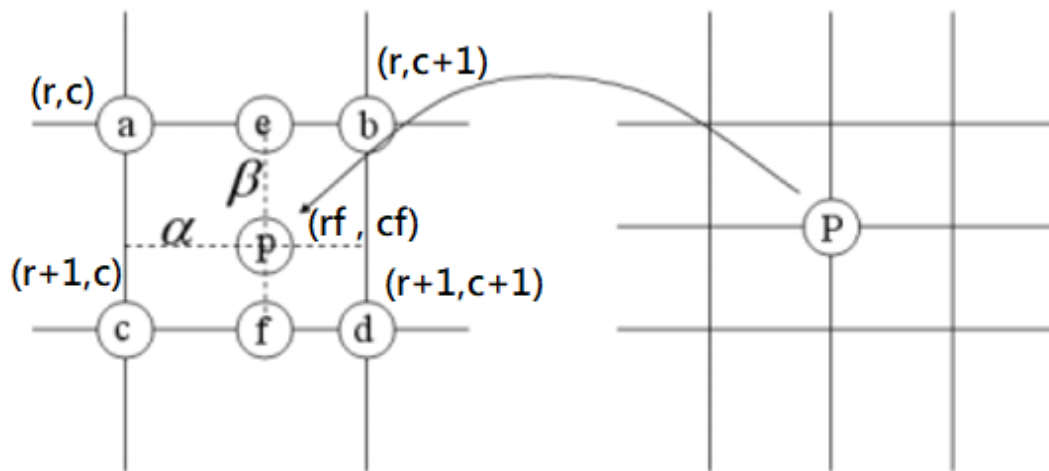


105323031 蔣圳嵐

2019/03/17

# 原理:

1. Downsample:使用 Matlab 內建的 downsample 函數實現。
2. Upsample:利用上課時老師所介紹的 Bilinear interpolation 之方法來實現。  
Bilinear interpolation 即是假設要求得 p 點(rf,cf)的數值，並且令點 a、b 之間的距離為 1，就要取得鄰近四點(a,b,c,d)的數值再乘上 x 軸之權重( $\alpha$ )得到式(2-9)和式(2-10)也就是 e 點和 f 點的數值，最後再利用 e 點和 f 點乘上 y 軸之權重( $\beta$ )即可得到 P 點的數值。



▲原理示意圖

$$e = (1 - \alpha)a + \alpha b \quad (2-9)$$

$$f = (1 - \alpha)c + \alpha d \quad (2-10)$$

$$P = (1 - \beta)e + \beta f \quad (2-11)$$

# 程式說明:

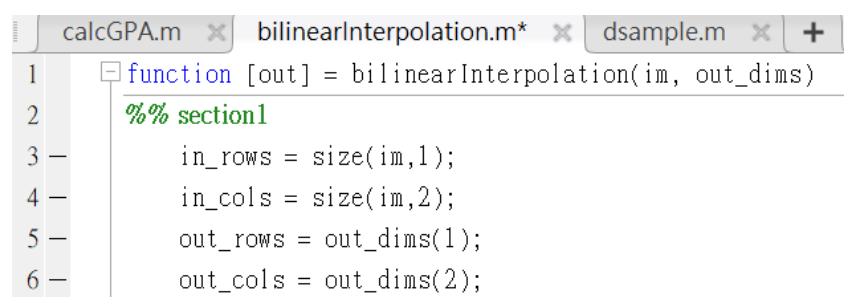
## 1. Downsample:



```
Editor - dsample.m*
calcGPA.m x bilinearInterpolation.m x dsample.m* x +
1 function [out] = dsample(im , factor)
2
3     im_half = downsample(im,factor);
4     out_turn = downsample(permute(im_half,[2 1 3]),factor);
5
6     out = permute(out_turn,[2 1 3]);
7 end
```

先將函式取名為 **dsample** 輸入為 **im** 降低倍數為 **factor** 輸出為 **out**，注意取名不得與 Matlab 內建的函式名稱重複。當我使用 **downsample** 函式時發現程式只會處理 **columns** 的值，因此將 **im** 的 **columns** 做 **downsample** 取得新矩陣 **im\_half**，之後把 **im\_half** 的 **row** 和 **column** 對調再進行一次 **downsample**，最後要將 **row** 和 **column** 對調回來即可完成 **downsample**。補充：因為 **im** 為一個三維的矩陣因此無法使用' (也就是 Matlab 內建的算術運算 **transpose**) 來對調 **column** 和 **row**，所以要使用 **permute** 函式來解決 **permute**(欲對調矩陣，[依次擺放維度的順序，1 為 **row** 依此類推])。

## 2. Upsample:



```
calcGPA.m x bilinearInterpolation.m* x dsample.m x +
1 function [out] = bilinearInterpolation(im, out_dims)
2 %% section1
3     in_rows = size(im,1);
4     in_cols = size(im,2);
5     out_rows = out_dims(1);
6     out_cols = out_dims(2);
```

首先先設定函式 **bilinearInterpolation** 的輸入圖片為 **im** 欲輸出像素 **out\_dims** 輸出為 **out**。之後在 **section1** 裡面設定一些基本數值，輸入和輸出個別 **column** 和 **row** 的數量。

```

8      %% section2
9 -      S_R = in_rows / out_rows;
10 -     S_C = in_cols / out_cols;
11
12 -     [cf, rf] = meshgrid(1 : out_cols, 1 : out_rows);
13
14 -     rf = rf * S_R;
15 -     cf = cf * S_C;

```

在 section2 中要計算出 **rf** 和 **cf** 的值，因此先利用輸入和輸出各自 **rows** 和 **columns** 的比值去計算出 **S\_R** 和 **S\_C**，之後創造一個矩陣 **rf** 代表 **row** 的值從 1 到輸出要求 **row** 的像素值，最後把 **rf** 和 **S\_R** 相乘即可得到我們真正要的 **rf**。而 **cf** 也是同樣道理，即可求得真正的 **cf**。

```

16     %% section3
17 -     r = floor(rf);
18 -     c = floor(cf);
19
20 -     r(r < 1) = 1;
21 -     c(c < 1) = 1;
22 -     r(r > in_rows - 1) = in_rows - 1;
23 -     c(c > in_cols - 1) = in_cols - 1;

```

在 section3 裡面，首先取得 **rf** 鄰近的整數值取名為 **r**。為了之後取值的時候不要出現錯誤因此下面加了兩個判斷式分別式當 **r** 小於 1 的時候值為 1 和 **r** 大於輸入 **row** 像素-1 值為輸入 **row** 像素-1。c 也是使用同樣道理。

```

24     %% section4
25 -     beta = rf - r;
26 -     alpha = cf - c;

```

計算出  $\alpha$  和  $\beta$  值。

```

27     %% section5
28 -     point_a = sub2ind([in_rows, in_cols], r, c);
29 -     point_b = sub2ind([in_rows, in_cols], r+1,c);
30 -     point_c = sub2ind([in_rows, in_cols], r, c+1);
31 -     point_d = sub2ind([in_rows, in_cols], r+1, c+1);

```

在 section5 是取得點 **a,b,c,d** 在矩陣內的儲存位置。

```

32     %% section6
33 -     out = zeros(out_rows, out_cols, size(im, 3));
34 -     out = cast(out, class(im));

```

section6 則是設定輸出矩陣 **out** 的各項設定，例如：大小、class 等等。

```

35      %% section7
36  —  for idx = 1 : size(im, 3)
37  —      value = double(im(:, :, idx));
38  —      e = (1-alpha).*value(point_a) + alpha.*value(point_b);
39  —      f = (1-alpha).*value(point_c) + alpha.*value(point_d);
40  —      P = (1-beta).*e + beta.*f;
41  —      out(:, :, idx) = cast(P, class(im));
42  —  end

```

最後 section7 就是使用一個 for 迴圈，分次取得 RGB 的值。迴圈跑第一次時事取得 R 層的值之後丟到 value 裡面，再帶入從 section5 中取得的儲存位置即可求得各點 R 層的數值，再利用式(2-9)、式(2-10)、式(2-11)最後求出 P，再將 P 的值給予矩陣 out 的 R 層。之後再跑 G、B 層就可以得到輸出圖片 out 了。

## 結論:

這次的程式讓我重視到了 `class` 的重要性，因為當兩個不同 `class` 的數值是無法做運算的，因此在寫程式的時候要注意。我也藉著這個實驗更了解圖片是如何在電腦內儲存的。然後也觀察到只要經過影像處理之後就會有圖片變模糊的情況發生，因此選擇一個好的處理方法相當重要。

所以當我發現 `Bilinear interpolation` 也可以去實現 `downsample` 時，我把同樣的一張圖片分別使用兩種方法都 `downsample` 2 倍，左圖為 `Bilinear interpolation`，右圖為 `Matlab` 內建的 `downsample`。觀察發現右圖帽子頂端的地方都快和背景融為一體，但是左圖的還是可以清晰判斷分界點。鼻子的地方左圖也較沒有顆粒感，因此左圖和右圖相比左圖使用的方法是一個較好的方法。




▲原圖



▲左圖為 `Bilinear interpolation`，右圖為 `Matlab` 內建的 `downsample`

再從資料大小上面觀察也能發現左圖的大小比右圖大，這樣也可以驗證我所觀察到的，因此結論是：左圖使用的方法 **Bilinear interpolation** 是一個比右圖使用的方法 **Matlab** 內建的 **downsample** 函式效能還要好。



lenna\_bili\_down

---

檔案類型:      JPG 檔案 (.jpg)

開啟檔案:      相片 

變更(C)...

---

位置:            C:\Users\ASUS\Desktop\things\visual hw

大小:            5.66 KB (5,800 位元組)

磁碟大小:       8.00 KB (8,192 位元組)

▲左圖資料大小為 5800 位元組



lenna\_down

---

檔案類型:      JPG 檔案 (.jpg)

開啟檔案:      相片 

變更(C)...

---

位置:            C:\Users\ASUS\Desktop\things\visual hw

大小:            5.61 KB (5,752 位元組)

磁碟大小:       8.00 KB (8,192 位元組)

▲右圖資料大小 5752 位元組