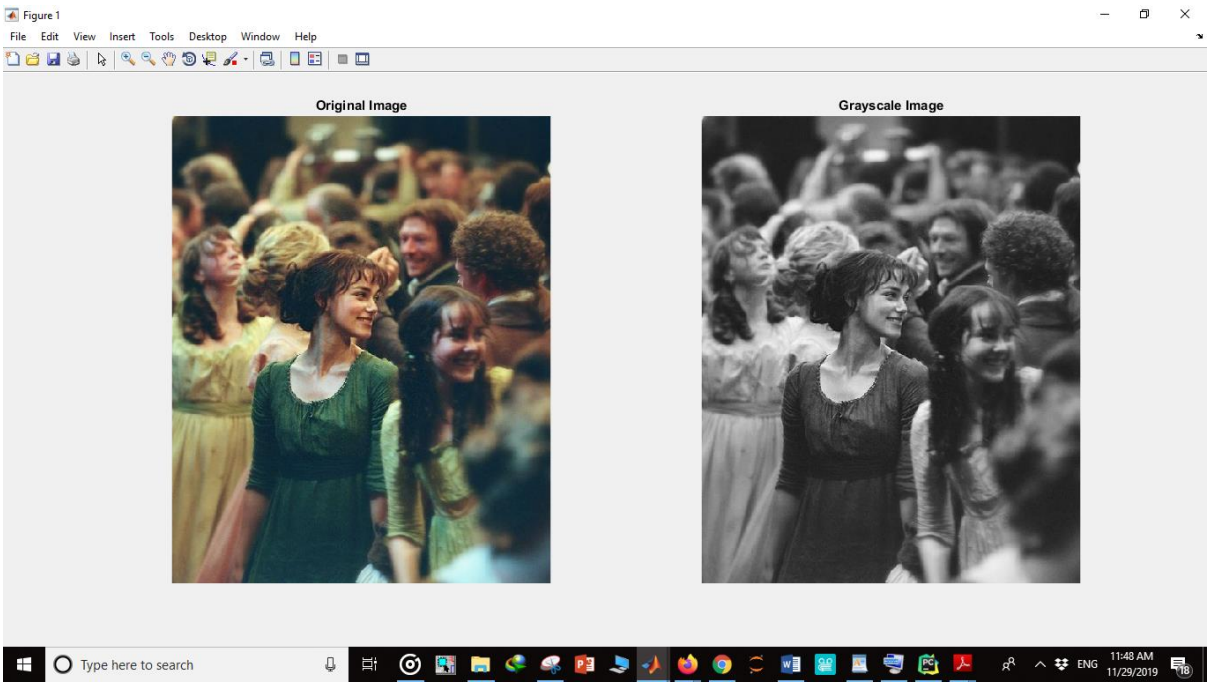
Yasaman Mirmohammad / 9431022

سیستم‌های چندرسانه‌ای

Fundamentals of multimedia سوالات فصل سه کتاب

۱. یک عکس ۶۴۰*۴۸۰ را به اختیار خود انتخاب کنید. به وسیله نرم‌افزار Matlab عملیات‌های زیر را انجام دهید.

۱,۱. تصویر را به 8 bit grey level image تغییر دهید.



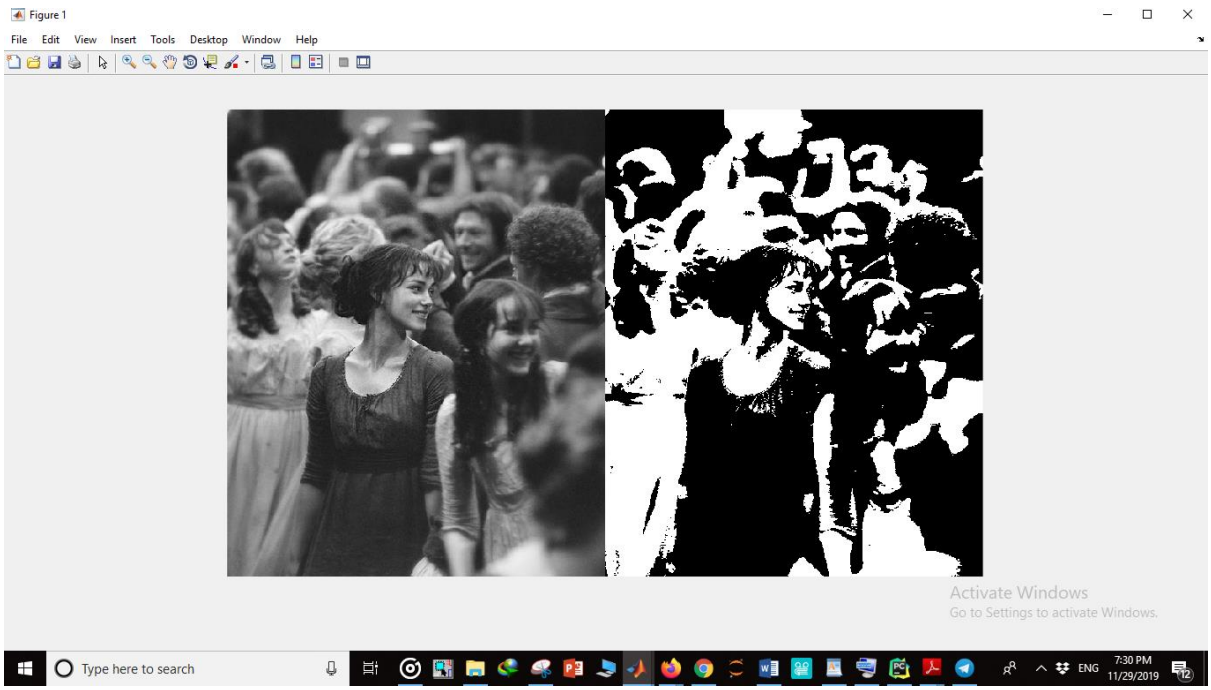

```
function returnedImage = To_gray_image_8(image)
        i = image;
        R = i(:, :, 1);
        G = i(:, :, 2);
        B = i(:, :, 3);
        newImage = zeros(size(i,1), size(i,2), 'uint8');

        for x=1:size(i,1)
           for y=1:size(i,2)
               newImage(x,y) = (R(x,y)*.3)+(G(x,y)*.6)+(B(x,y)*.1);
           end
        end

        returnedImage = newImage;
        imwrite(returnedImage, 'gray_image.jpg');
end
```

۱,۲. تصویر را به 1 bit image تغییر دهید.

Binary Image(threshold):

تصویری را که به خاکستری تبدیل کرده بودیم اکنون تبدیل به تصویر سیاه و سفید میکنیم:

```
I = imread('photo.jpg'););
BW = imbinarize(image);
figure
imshowpair(I,BW,'montage')
returnedImage=BW;
```



۲. با توجه به الگوریتم فلوید استینبرگ به سوالات زیر پاسخ دهید.

۲,۱. Dithering چیست؟

Dithering takes advantage of the human eye's tendency to "mix" two colors in close proximity to one another. It is used for making the gray theme in human vision.

For printing on a 1-bit printer, dithering is used to calculate larger patterns of dots, such that values from 0 to 255 correspond to pleasing patterns that correctly represent darker and brighter pixel values. The main strategy is to replace a pixel value by a larger pattern, say 2×2 or 4×4, such that the number of printed dots approximates the varying-sized disks of ink used in *halftone printing*. Half-tone printing is an analog process that uses smaller or larger filled circles of black ink to represent shading, for newspaper printing, say.
If instead we use an $n \times n$ matrix of on–off 1-bit dots, we can represent $n^2 + 1$ levels of intensity resolution—since, for example, three dots filled in any way counts

as one intensity level. The dot patterns are created heuristically. For example, if we use a 2 × 2 “dither matrix”:

we can first remap image values in 0 .. 255 into the new range 0 .. 4 by (integer) dividing by 256/5. Then, for example, if the pixel value is 0, we print nothing in a 2 × 2 area of printer output. But if the pixel value is 4, we print all four dots. So the rule is:
If the intensity is greater than the dither matrix entry, print an `on` dot at that entry location: replace each pixel by an *n* × *n* matrix of such `on` or `off` dots.

Floyd–Steinberg dithering is an image dithering algorithm first published in 1976 by Robert W. Floyd and Louis Steinberg. It is commonly used by image manipulation software, for example when an image is converted into GIF format that is restricted to a maximum of 256 colors.

$$\begin{bmatrix} & & * & \frac{7}{16} & \cdots \\ \cdots & \frac{3}{16} & \frac{5}{16} & \frac{1}{16} & \cdots \end{bmatrix}$$

(*) indicates the pixel currently being scanned, and the blank pixels are the previously-scanned pixels. The algorithm scans the image from left to right, top to bottom, quantizing pixel values one by one. Each time the quantization error is transferred to the neighboring pixels, while not affecting the pixels that already have been quantized. Hence, if a number of pixels have been rounded downwards, it becomes more likely that the next pixel is rounded upwards, such that on average, the quantization error is close to zero.

```
for each y from top to bottom
   for each x from left to right
      oldpixel  := pixel[x][y]
      newpixel  := find_closest_palette_color(oldpixel)
      pixel[x][y]  := newpixel
      quant_error  := oldpixel - newpixel
      pixel[x + 1][y    ] := pixel[x + 1][y    ] + quant_error * 7 / 16
      pixel[x - 1][y + 1] := pixel[x - 1][y + 1] + quant_error * 3 / 16
      pixel[x    ][y + 1] := pixel[x    ][y + 1] + quant_error * 5 / 16
      pixel[x + 1][y + 1] := pixel[x + 1][y + 1] + quant_error * 1 / 16
```

۲,۲. فرض کنید میخواهیم دامنه رنگی یک عکس سیاه سفید را از ۲۵۶ رنگ به ۲ رنگ کاهش دهیم، ماتریس زیر بیانگر ارزش رنگی در هر پیکسل است، با استفاده از الگوریتم فلوید استینبرگ عکس را dither کنید. و ماتریس نهایی را تشکیل دهید.

105	**255**	**15**
0	**215**	**55**
95	**50**	**0**

ماتریس ما ۳*۳ است. از ماتریس دیترینگ ۲*۲ استفاده میکنیم:

$$\begin{pmatrix} 0 & 2 \\ 3 & 1 \end{pmatrix}$$

L=256/5=51.02

مقدار هر پیکسل را به l تقسیم میکنیم. حاصل را با هر یک از اعضای ماتریس دیترینگ مقایسه میکنیم، اگر بزرگتر بود سیاه چاپ میکنیم.(برای هر پیکسل عکس اولیه، باید چهار درایه چاپ کنیم)

105/51.02=2.05➔ 2 ➔ wbbw

255/51.02=4.99➔ 4 ➔ bbbb

15/51.02=0.2➔ bwww

215/51.02=4.21➔bbbb

55/51.02=1.07➔ wbbb

95/51.02=1.86➔ wbbb

50/51.02=0.98➔bwww

۲,۳.

با استفاده از متلب کد dither کردن تصویر سوال قبل را یک بار با محاسبه‌ی خطا و یکبار بدون محاسبه‌ی خطا پیاده‌سازی کنید و نتایج را با هم مقایسه کنید. (دامنه رنگی را برای هر رنگ قرمز، آبی و سبز از ۲۵۶ به ۴ کاهش دهید)

- Ordered dithering

Defined two dither matrix (*22 and 44*) for ordered dithering.

- Floyd-Steinberg algorithm

The most famous error diffusion method.

بدون محاسبه ی خطا:

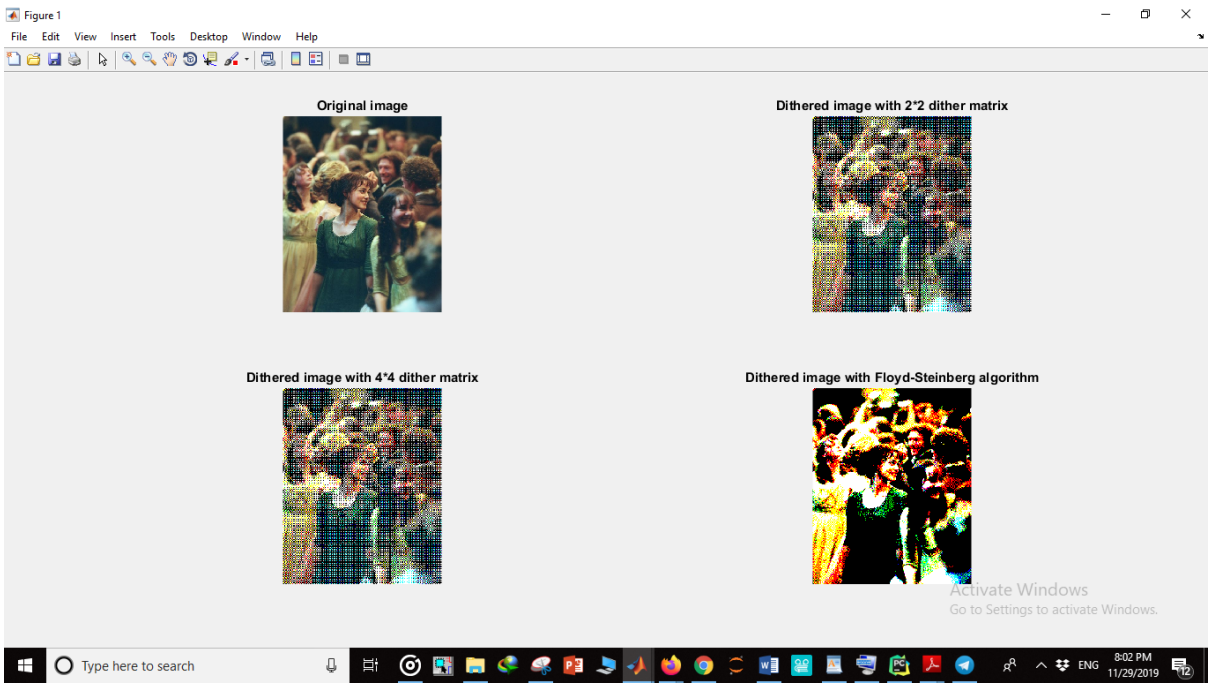
```
%% Ordered dithering
function dithered_img = ordered_dithering(img, matrix_size)

if matrix_size == 4
    dither_matrix = [0 8 2 10; 12 4 14 6; 3 11 1 9; 15 7 13 5] * 16;
else
    matrix_size = 2;
```

```
    dither_matrix = [0 128 ;192 64];
end

[row, col] = size(img);
dithered_img = zeros([row, col]);

for x = 1:row
    for y = 1:col
        i = mod(x, matrix_size) + 1;
        j = mod(y, matrix_size) + 1;
        if img(x,y) > dither_matrix(i,j)
            dithered_img(x,y) = 1;
        end
    end
end
end
```

با محاسبه ی خطا:

```
%% Floyd-Steinberg image dithering
function dithered_img = fs_dithering(img)

[row, col] = size(img);
error_matrix = zeros([row, col]);
dithered_img = zeros([row, col]);

for y = 1:col
    for x = 1:row
        if (img(x,y) + error_matrix(x,y) < 128)
            dithered_img(x,y) = 0;
        else
            dithered_img(x,y) = 255;
        end
        diff = img(x,y) + error_matrix(x,y) - dithered_img(x,y);

        if y < col
            error_matrix(x,y+1) = error_matrix(x,y+1) + diff * 7 / 16;
        end
        if x < row
            error_matrix(x+1,y) = error_matrix(x+1,y) + diff * 5 / 16;
            if y > 1
                error_matrix(x+1,y-1) = error_matrix(x+1,y-1) + diff * 3 /
16;
            end
            if y < col
                error_matrix(x+1,y+1) = error_matrix(x+1,y+1) + diff * 1 /
16;
            end
        end
    end
end
end
```

نتیجه:



روش Floyd-Steinberg، نتیجه ی خیلی بهتری به ما میدهد. در رو.ش oredered، هر چقدر تلاش کنیم باز هم به این نتیجه نمیرسیم.

۲,۴. چرا در این الگوریتم خطای هر پیکسل را با پیکسل های مجاور جمع میکنیم؟

برای کاهش error. در این حالت انگار تاثیر خطای هر پیکسل را در پیکسل های مجاورش پخش میکنیم.

In 1975, Floyd and Steinberg introduced *error diffusion*, a process where the error made at one pixel, when it is assigned a 0 or a 255, is passed on to its neighbours. Much like a real diffusion process, the error is spread so that it can be compensated for later on. The image is scanned row by row once. Each pixel visited is assigned a value, and the difference between that value and the input value is split among the neighbours not yet visited.

Boiled down to its simplest form, dithering is fundamentally about ***error diffusion*.**

Error diffusion works as follows: let's pretend to reduce a grayscale photograph to black and white, so we can print it on a printer that only supports pure black (ink) or pure white (no ink). The first pixel in the image is dark gray, with a value of 96 on a scale from 0 to 255, with zero being pure black and 255 being pure white.

Unfortunately, error diffusion dithering has problems of its own. For better or worse, dithering always leads to a spotted or stippled appearance. This is an inevitable side-effect of

working with a small number of available colors – those colors are going to be repeated over and over again, because there are only so many of them.

When converting such a pixel to black or white, we use a simple formula – is the color value closer to 0 (black) or 255 (white)? 96 is closer to 0 than to 255, so we make the pixel black.

At this point, a standard approach would simply move to the next pixel and perform the same comparison. But a problem arises if we have a bunch of "96 gray" pixels – they all get turned to black, and we're left with a huge chunk of empty black pixels, which doesn't represent the original gray color very well at all.

In the simple error diffusion example above, another problem is evident – if you have a block of very similar colors, and you only push the error to the right, all the "dots" end up in the same place! This leads to funny lines of dots, which is nearly as distracting as the original, non-dithered version.

The problem is that we're only using a *one-dimensional* error diffusion. By only pushing the error in one direction (right), we don't distribute it very well. Since an image has two dimensions – horizontal and vertical – why not push the error in multiple directions? This will spread it out more evenly, which in turn will avoid the funny "lines of speckles" seen in the error diffusion example above.

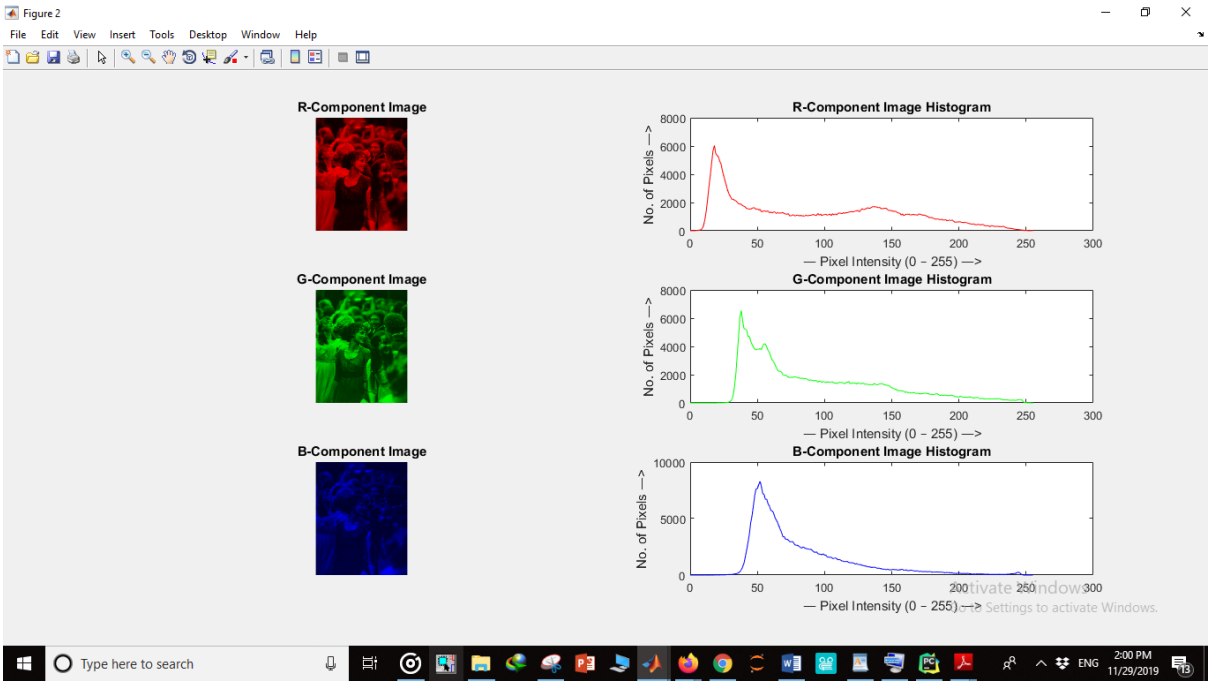
There are many ways to diffuse an error in two dimensions. For example, we can spread the error to one or more pixels on the right, one or more pixels on the left, one or more pixels up, and one or more pixels down.

For simplicity of computation, all standard dithering formulas push the error *forward*, never backward. If you loop through an image one pixel at a time, starting at the top-left and moving right, you never want to push errors *backward* (e.g. left and/or up). The reason for this is obvious – if you push the error backward, you have to revisit pixels you've already processed, which leads to more errors being pushed backward, and you end up with an infinite cycle of error diffusion.

So for standard loop behavior (starting at the top-left of the image and moving right), we only want to push pixels *right* and *down*.

۳. یک عکس ۶۴۰*۴۸۰ را به اختیار خود انتخاب کنید. برای گرفتن نتیجه بهتر توصیه میشود از عکس raw استفاده کنید یعنی عکسی که هیچ فشرده سازی‌ای روی آن اتفاق نیفتاده باشد.

۳,۱. هیستوگرام این عکس را جداگانه برای رنگ های قرمز، آبی و سبز رسم کنید



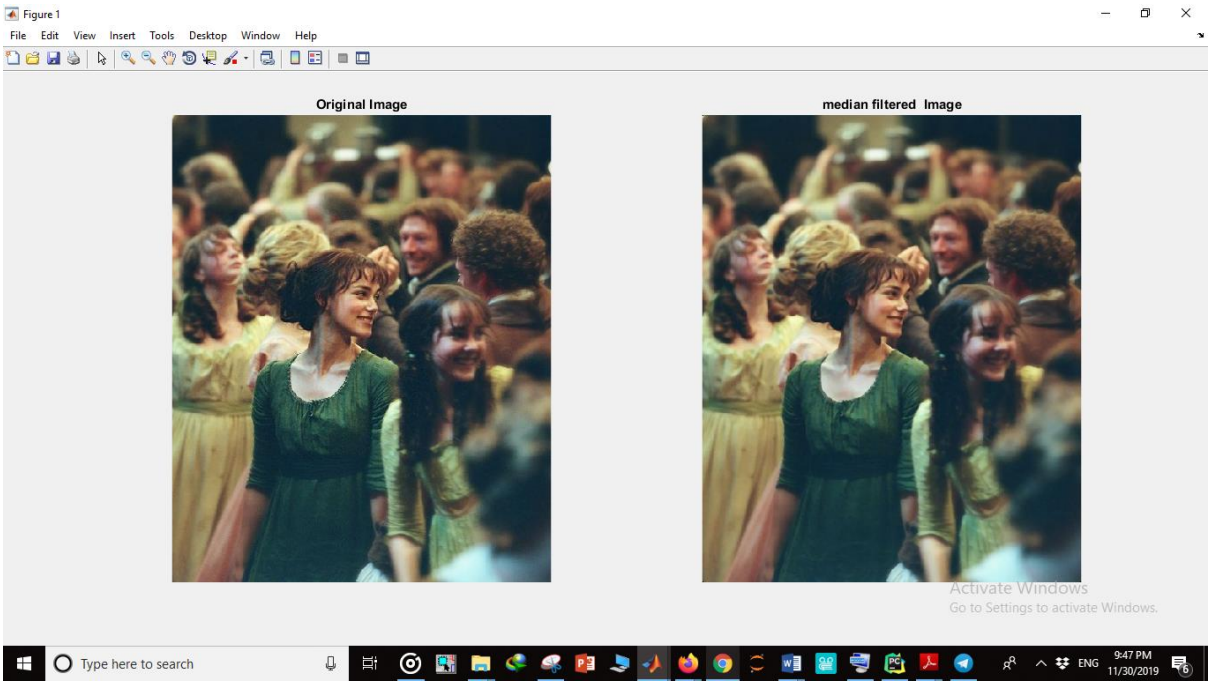
۳,۲. با استفاده از الگوریتم **Median cut** تعداد بیت های قرمز و سبز را به سه و آبی را به دو کاهش دهید. برای اینکار ابتدا میانه های هر رنگ و نماینده دسته را با استفاده از میانگین بیابید. سپس از تابع **quantiz** در متلب استفاده کنید. تا بتوان هر کد رنگ را به یک رنگ بخصوص تبدیل کرد.

Median cut is an algorithm to sort data of an arbitrary number of dimensions into series of sets by recursively cutting each set of data at the median point along the longest dimension. Median cut is typically used for color quantization. For example, to reduce a 64k-colour image to 256 colours, median cut is used to find 256 colours that match the original data well.

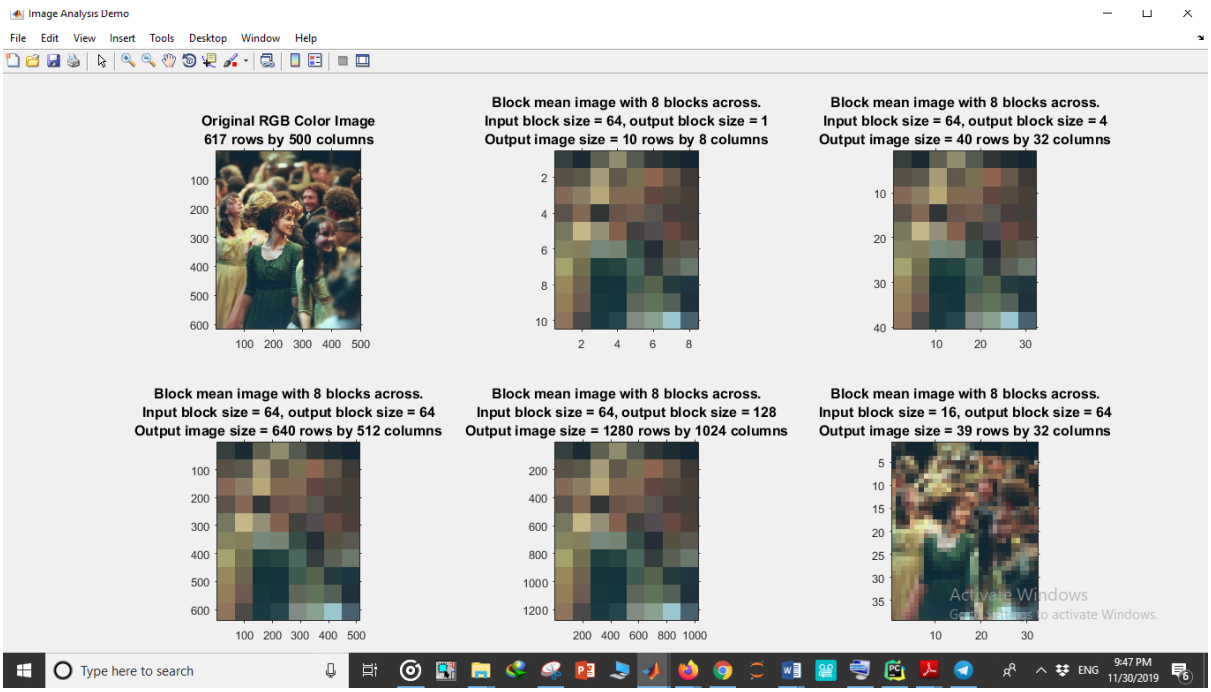
`rgb2ind` uses *colormap mapping* (instead of quantization) to find the colors in the specified colormap that best match the colors in the RGB image. This method is useful if you need to create images that use a fixed colormap. For example, if you want to display multiple indexed images on an 8-bit display, you can avoid color problems by mapping them all to the same colormap. Colormap mapping produces a good approximation if the specified colormap has similar colors to those in the RGB image. If the colormap does not have similar colors to those in the RGB image, this method produces poor results.

۳,۳. تصویر حاصل را نمایش داده و با تصویر اصلی مقایسه کنید.

Median Cut filtering:



Mean cut filtering:



مشاهده میکنیم که روش median cut دقت خیلی بالاتری از روش mean cut دارد.

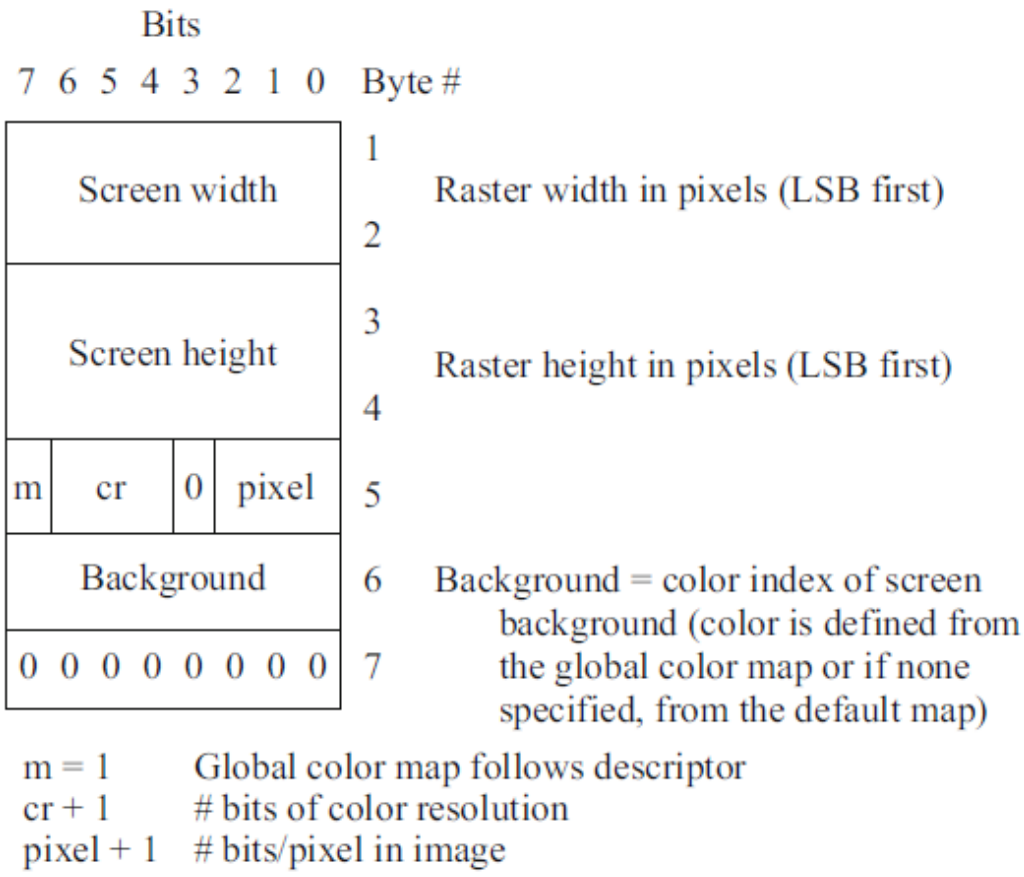
۴. از میان فرمت های png, gif, jpeg, tiff, ptm دو عدد را به دلخواه توضیح دهید.(پاسخ برای هر مورد در حد چند خط).

Tiff:

Tagged Image File Format (TIFF) is another popular image file format.
Developed by the Aldus Corporation in the 1980s, it was later supported by Microsoft. Its support for attachment of additional information (referred to as “tags”) provides a great deal of flexibility.
The most important tag is a format signifier: what type of compression, etc., is in use in the stored image. For example, TIFF can store many different types of images: 1-bit, grayscale, 8-bit, 24-bit RGB, and so on.
TIFF was originally a lossless format, but an added tag allows you to opt for JPEG, JBIG, and even JPEG-2000compressions.
Since TIFF is not as user-controllable as JPEG, it does not provide any major advantages over the latter for lossy compression.
It is quite common to useTIFF files to store *uncompressed* data.
TIFF files are divided into sections, each of which can store a bitmap image, a vector-based or stroke-based image or other types of data.
Each section’s data type is specified in its tag.

GIF:

Graphics Interchange Format (GIF) was devised by UNISYS Corporation and Compuserve, initially for transmitting graphical images over phone lines via modems. The GIF standard uses the Lempel-Ziv-Welch algorithm (a form of compression—see Chap. 7), modified slightly for image scanline packets to use the line grouping of pixels effectively.
The GIF standard is limited to 8-bit (256) color images only. While this produces acceptable color, it is best suited for images with fewdistinctive colors (e.g., graphics or drawing).
The GIF image format has a few interesting features, notwithstanding the fact that it has been largely supplanted. The standard supports *interlacing*—the successive display of pixels in widely spaced rows by a four-pass display process.



PTM

PTM (Polynomial Texture Mapping) is a technique for storing a representation of a camera scene that contains information about a set of images taken under a set of lights that each have the same spectrum, but with each placed at a different direction from the scene [7].

Suppose we have acquired *n* images of a scene, taken with a fixed-position camera but with lighting from $i = 1 .. n$ different lighting directions $\boldsymbol{e}_i = (u_i, v_i, w_i)^T$.

For example, a hemispherical lighting frame could be used with, say, 40 or 50 lights, one at each vertex of a geodesic dome. The objective of PTM is in part to find out the surface properties of the object being imaged—this has been used for imaging museum artifacts and paintings, for instance. But the main task for PTM is being able to interpolate the lighting directions, so as to generate new images not seen before. The file size for PTM image collections is kept small by packing multiple interpolation coefficients into integer data.

PNG:

Portable Network Graphics (PNG).

This standard is meant to supersede the GIF standard and extends it in important ways. The motivation for a new standard was in part the patent held by UNISYS and Compuserve on the LZW compression method. (Interestingly, the patent covers only compression, not decompression: this is why the Unix `gunzip` utility can decompress LZW-compressed files).

Special features ofPNGfiles include support for up to 16 bits per pixel in each color channel, i.e., 48-bit color—alarge increase. Files may also contain gamma-correction information. for correct display of color images and α-channel information (up to 16 bits) for such uses as control of transparency.

Instead of a progressive display based on row-interlacing as in GIF images, the display progressively

displays pixels in a two-dimensional interlacing over seven passes through each 8 × 8 block of an image.
It supports both lossless and lossy compression with performance better than GIF.
PNG is widely supported by various web browsers and imaging software.

JPEG:
The most important current standard for image compression is JPEG . This standard was created by a working group of the International Organization for Standardization (ISO) that was informally called the Joint Photographic Experts Group and is therefore so named.
The human vision system has some specific limitations, which JPEG takes advantage of to achieve high rates of compression. The eye–brain system cannot see extremely fine detail. If many changes occur within a few pixels, we refer to that image segment as having *high spatial frequency*—that is, a great deal of change in (*x*, *y*) space. This limitation is even more conspicuous for color vision than for grayscale (black and white). Therefore, color information in JPEG is *decimated* (partially dropped, or averaged) and then small blocks of an image are represented in the spatial frequency domain (*u*, v), rather than in (*x*, *y*). That is, the speed of changes in *x* and *y* is evaluated, from low to high, and a new "image" is formed by grouping the coefficients or weights of these speeds.
Weights that correspond to slow changes are then favored, using a simple trick: values are divided by some large integer and truncated. In this way, small values are zeroed out. Then a scheme for representing long runs of zeros efficiently is applied, and *voila!*—the image is greatly compressed.
Since we effectively throwaway a lot of information by the division and truncation step, this compression scheme is "lossy" (although a lossless mode exists). What is more, since it is straightforward to allow the user to choose how large a denominator to use and hence how much information to discard, JPEG allows the user to set a desired level of quality, or compression ratio (input divided by output).

This image is a mere 1.5% of the original size. In comparison, a JPEG image with $Q = 75$ yields an image size 5.6% of the original, whereas a GIF version of this image compresses down to 23.0% of the uncompressed image size.

توجه: الگوریتم دیترینگ در سایت ویکی پدیا خیلی خوب توضیح داده شده
https://en.wikipedia.org/wiki/Floyd%E2%80%93Steinberg_dithering
یک ویدیو از پیاده سازی الگوریتم:
https://youtu.be/0L2n8Tg2FwI