دانشگاه تهران
پردیس دانشکده‌های فنی
دانشکده مهندسی برق و کامپیوتر

پردازش تصاویر دیجیتال

گزارش تمرین سری اول

دانشجو
سید محمد جواد موسوی

استاد کلاس
دکتر حمید سلطانیان‌زاده

سوال ۱

بخش اول

هدف آشنایی با پردازش تصویر دیجیتال در MATLAB از طریق بارگذاری و تبدیل تصاویر است. در این مثال، ابتدا تصویر رنگی `lenna_rgb.png` که در فرمت PNG قرار دارد، با استفاده از دستور `imread` بارگذاری می‌شود. سپس با استفاده از دستور `imshow`، تصویر اصلی به نمایش درمی‌آید.

```
1. %% Question 1 - Here, we have an example to get familiar with MATLAB for Digital Image
Processing- Part a to c
2. file_path = 'images/lenna_rgb.png';
3.
4. % To read the image using imread
5. lenna_rgb = imread(file_path);
6.
7. % To display the image using imshow
8. figure;
9. imshow(lenna_rgb);
١٠. title('Lenna rgb image');
```

بخش دوم

برای تبدیل تصویر رنگی به تصویر خاکستری، از تابع `rgb۲gray` استفاده می‌شود که تصویر رنگی را به مقیاس خاکستری تبدیل می‌کند. این تابع هر سه کانال رنگی (قرمز، سبز و آبی) را به یک کانال خاکستری واحد ترکیب می‌کند تا تصویر نهایی ایجاد شود. تصویر **خاکستری** یک تصویر دو بعدی است که فقط شامل شدت روشنایی یا نوری است و هیچ رنگی ندارد. در تصاویر خاکستری، همه پیکسل‌ها یک شدت نوری دارند که معمولاً از صفر (سیاه) تا ۲۵۵ (سفید) متغیر است. به عبارت دیگر، در تصاویر خاکستری، همه پیکسل‌ها از یک طیف نوری (نه رنگی) برخوردارند. تصاویر رنگی معمولاً از سه کانال رنگی **قرمز(Red)** ، **سبز (Green)** و **آبی (Blue)** تشکیل شده‌اند که برای هر پیکسل مقدارهایی از این سه رنگ وجود دارد. در تبدیل به خاکستری، باید این سه رنگ را به یک مقدار نوری واحد ترکیب کرد. در حالت استاندارد، برای تبدیل RGB به خاکستری از یک وزن‌دهی خاص برای هر کانال رنگی استفاده می‌شود. زیرا چشم انسان به رنگ‌ها به طور متفاوتی حساس است. معمولاً وزن‌هایی که برای هر کانال در نظر گرفته می‌شود به شکل زیر است:

- قرمز ۰,۲۹۸۹ → (Red)
- سبز ۰,۵۸۷۰ → (Green)
- آبی ۰,۱۱٤۰ → (Blue)

این وزن‌ها به این دلیل انتخاب شده‌اند که چشم انسان حساسیت بیشتری به رنگ سبز دارد و حساسیت کمتری به رنگ آبی.

فرمول تبدیل به خاکستری به صورت زیر است:

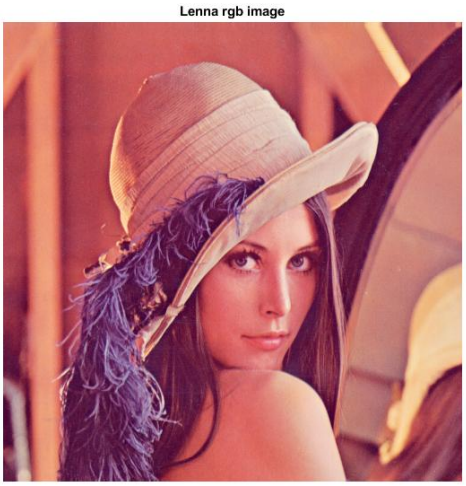
$$\text{Gray Value}=۰,۲۹۸۹\times R+۰,۵۸۷۰\times G+۰,۱۱۴۰\times B$$

درMATLAB ، برای تبدیل تصویر RGB به خاکستری از تابع rgb۲gray استفاده می‌شود. این تابع به طور پیش‌فرض از فرمول ذکر شده برای تبدیل استفاده می‌کند. در نتیجه، رنگ‌های تصویر به یک سطح شدت نوری واحد تبدیل می‌شوند.

```
۱. %convert the image to grayscale
۲. gray_lenna_rgb = rgb۲gray(lenna_rgb);
```

نتایج کد زیر در ادامه نمایش داده شده است:

```
 1. figure;
 2. subplot(1,2, 1);
 3. imshow(lenna_rgb)
 4. axis equal;
 5. title('Lenna rgb image');
 6.
 7. subplot(1, 2, 2);
 8. imshow(gray_lenna_rgb);
 9. axis equal;
۱۰. title('Grayscale lenna image');
```



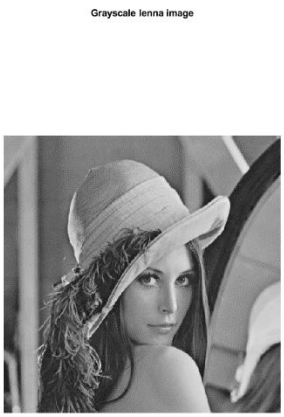
Lenna rgb image

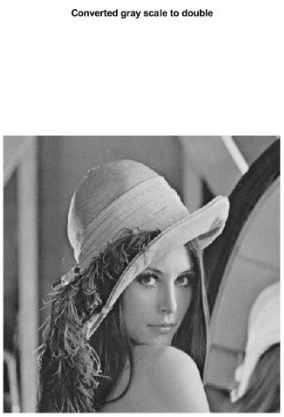
Grayscale lenna image

بخش چهارم

کد ;(gray_lenna_rgb)im۲double = double_lenna_rgb تصویر خاکستری gray_lenna_rgbرا به نوع داده doubleتبدیل می‌کند. در ابتدا باید بدانیم که تصاویر در MATLAB معمولاً به صورت ماتریس‌هایی از اعداد در نوع داده‌های مختلف مانند uint۸(۸ بیتی بدون علامت) ذخیره می‌شوند. در نوع داده uint۸، مقادیر پیکسل‌ها در بازه [۰, ۲۵۵] قرار دارند، که ۰ نمایانگر رنگ سیاه و ۲۵۵ نمایانگر سفید است. اما با استفاده از تابع im۲double، مقادیر پیکسل‌ها به نوع داده doubleتبدیل می‌شوند و مقادیر پیکسل‌ها به بازه [۰, ۱] تغییر می‌کنند. به عبارت دیگر، مقدار هر پیکسل به ۲۵۵ تقسیم می‌شود تا به بازه [۰, ۱] برسد. این تبدیل دقت بیشتری را در پردازش‌های ریاضیاتی فراهم می‌آورد، زیرا نوع داده doubleاجازه می‌دهد تا عملیات‌های پیچیده‌تری مانند ضرب ماتریس یا محاسبات دقیق‌تر انجام شوند. همچنین، این تبدیل محدودیت‌های عددی نوع uint۸را از بین می‌برد و امکان انجام عملیات‌هایی که نیاز به مقادیر دقیق‌تری دارند، مانند تقسیم یا جذر، را فراهم می‌کند. علاوه بر این، بسیاری از الگوریتم‌های پردازش تصویر و یادگیری ماشین نیاز دارند که تصاویر به نوع داده doubleتبدیل شوند تا به درستی روی آن‌ها اعمال شوند. در نتیجه، تبدیل تصویر خاکستری به doubleباعث می‌شود که پردازش‌های تصویر به دقت و کارایی بیشتری انجام شوند.

وقتی مقدار روشنایی به doubleتبدیل می‌شود میتوان محاسبات را با دقت اعشار تا ۱۵ رقم انجام داد که به نسبت حالت uint۸ که مقادیر کی از اعداد صفر تا ۲۵۵ هستن تفاوت زیادی است.






```
1. double_lenna_rgb = im2double(gray_lenna_rgb);
2.
3. figure;
4. subplot(1, 2, 1);
5. imshow(gray_lenna_rgb);
6. axis equal;
7. title('Grayscale lenna image');
8.
9. subplot(1, 2, 2);
١٠. imshow(double_lenna_rgb);
١١. axis equal;
١٢. title('Converted gray scale to double');
```

در نهایت بعد از اعمال تغییرات تصویر به صورت زیر ذخیره شد و کمترین و بیشترین مقدار سطح روشنایی در تصویر نیز نمایش داده شده است.

```
١. Original Grayscale (uint٨): Min = ٢٥, Max = ٢٤٥
٢. Converted Grayscale (double): Min = ٠,١٠, Max = ٠,٩٦
٣. Image saved successfully as lenna_rgb_double.jpg
```

بخش پنجم

هدف اصلی مقایسه تأثیر تغییر مقیاس (resizing) تصویر خاکستری بر کیفیت و جزئیات تصویر است. این کار با استفاده از تابع imresizeدر MATLAB انجام می‌شود. در این بخش از پروژه، تصویر خاکستری

gray_lenna_rgbکه در بخش‌های قبلی تبدیل شـده اسـت، در مقیاس‌های مختلف تغییر اندازه داده می‌شـود و نتایج برای تحلیل به نمایش درمی‌آید.

```
%% Part f:
scale_factor = [5, 1/2, 1/4];

figure;
subplot(2, 2, 1);
imshow(gray_lenna_rgb);
axis equal;
title('Origin Grayscale Image');

for i = 1:length(scale_factor)
    resized_image = imresize(gray_lenna_rgb, scale_factor(i));
    subplot(2, 2, i+1);
    imshow(resized_image);
    title(sprintf('Scale: %.2f', scale_factor(i)));
end
```

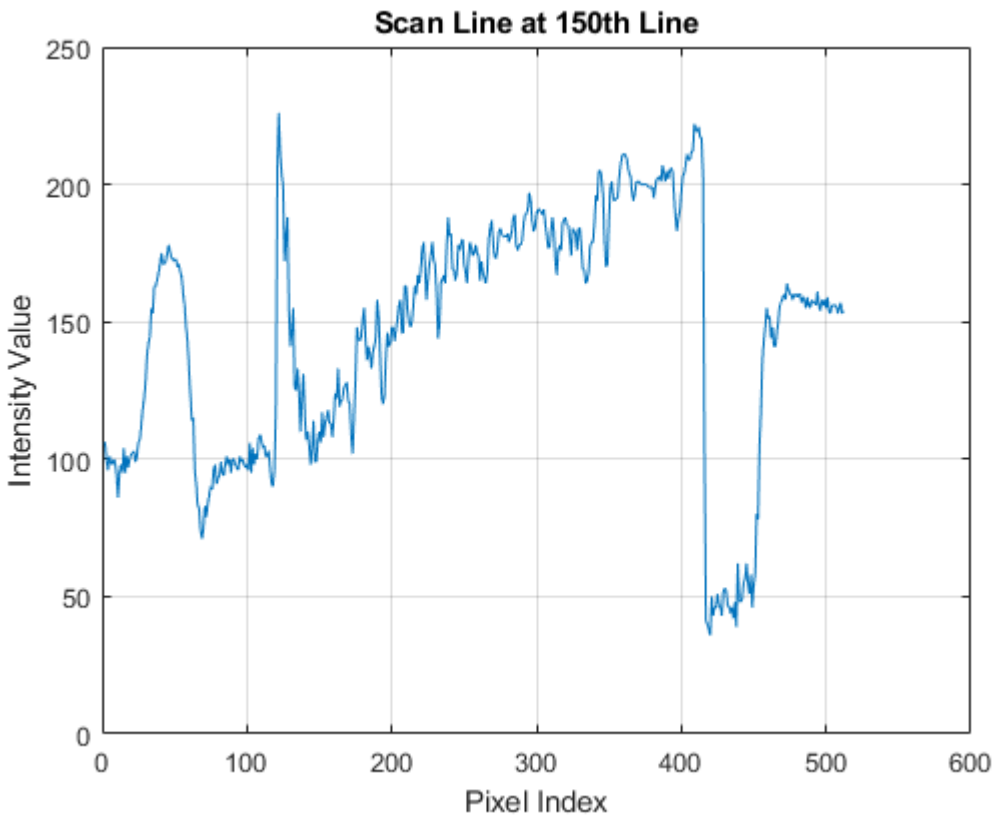
تغییر اندازه تصاویر می‌تواند اطلاعات تصویری را از دست بدهد، به‌ویژه زمانی که اندازه تصویر کوچک‌تر می‌شود. نتایج نشان می‌دهند که در تصویر با مقیاس ۵، ابعاد تصویر ۵ برابر شده و با توجه بهروش درونیابی مورد استفاده تصـویر وضـوح بهتری نسـبت به تصـویر اصـلی دارد. البته که تغییر ابعاد باعث ایجاد تعداد پیکسـلهای جدید و در نتیجه نیاز به حافظه بیشـتر می‌شـود . زمانی که تصـویر به نصـف اندازه اصـلی کاهش می‌یابد، هنوز جزئیات خوبی حفظ شده، اما برخی از جزئیات ریز ممکن است از دست بروند. زمانی که تصویر به یک‌چهارم اندازه اصلی کاهش می‌یابد، جزئیات بیشـتری از دست رفته و تصویر به صورت پیکسل به پیکسل می‌شود. این تغییرات نشان می‌دهد که کاهش اندازه تصویر، به ویژه زمانی که مقیاس به مقادیر کوچک‌تر می‌رسد، منجر به کاهش کیفیت و از دست

رفتن جزئیات می‌شود. لازم به ذکر است که به طور پیش‌فرض MATLAB از روش Bilinearبرای تغییر اندازه استفاده می‌کند. در این روش، مقدار هر پیکسل جدید با استفاده از یک میانگین وزنی از چهار پیکسل مجاور (در جهت‌های عمودی و افقی) محاسبه می‌شود. این روش معمولاً برای تغییر اندازه‌های کوچک‌تر یا بزرگ‌تر مناسب است و تصویر نرم‌تری تولید می‌کند.

بخش ششم

هدف اصلی استخراج یک خط از تصویر و نمایش تغییرات شدت روشنایی (intensity) در این خط است. این کار به‌ویژه برای تحلیل ویژگی‌های خاصی از تصویر، مانند شدت رنگ یا تغییرات در جزئیات تصویر، مفید است. در ابتدا، یک متغیر به نام line_numberتعریف می‌شود که شماره خطی از تصویر را نشان می‌دهد که قرار است بررسی شود. در این مثال، شماره خط ۱۵۰ است. با استفاده از این متغیر، یک خط افقی از تصویر gray_lenna_rgbانتخاب می‌شود. این کار با دستور scan_line = gray_lenna_rgb(line_number, :); انجام می‌شود که همه پیکسل‌های موجود در ردیف ۱۵۰ (به‌صورت افقی) را از تصویر استخراج می‌کند. سپس با استفاده از تابع plot(scan_line)، مقادیر شدت روشنایی پیکسل‌ها در خط ۱۵۰ ترسیم می‌شود. در اینجا، محور افقی نمایانگر شاخص پیکسل‌ها در آن خط و محور عمودی نمایانگر شدت روشنایی یا مقدار پیکسل‌ها است. برای بهبود نمای گراف، از دستورات title, xlabel, ylabelبرای اضافه کردن عنوان و برچسب به محورهای نمودار استفاده می‌شود. همچنین با استفاده از دستور grid on، خطوط شبکه برای تسهیل مشاهده تغییرات در گراف اضافه می‌شود. هدف از این کد استخراج و نمایش تغییرات شدت روشنایی در یک خط خاص از تصویر است. این روش معمولاً برای آنالیز ویژگی‌های خاص تصاویر، مانند لبه‌ها، الگوهای روشنایی یا بررسی نواحی خاص تصویر استفاده می‌شود. بررسی شدت روشنایی در یک خط خاص می‌تواند به تشخیص تغییرات یا جزئیات ظریف تصویر کمک کند.

```
line_number = 150;

scan_line = gray_lenna_rgb(line_number, :);

figure;
plot(scan_line);
title('Scan Line at 150th Line');
xlabel('Pixel Index');
ylabel('Intensity Value');
grid on;
```



سوال ۲

در این سوال قرار است تا سطوح شدت روشنایی تصویر skull.tif را به سطوح ۶۴ و ۱۶ و ۴ و ۲ کاهش دهیم البته لازم است تا اندازه تصویر تغییر نکند و تعداد پیکسل‌ها ثابت بماند. در ابتدا بعد از فراخوانی تصویر، تصویر اصلی را با دستور imshow به نمایش می گذاریم. سپس برای کاهش تصویر از دو تابع استفاده می‌کنیم. عملیات کاهش شدت به مقادیر ۶۴ و ۱۶ و ۴ و ۲ را با تابع reduce_level انجام می‌دهیم. برای کاهش سطح در از رابطه زیر استفاده می‌کنیم:

$$image_{new} = floor(\frac{double(image_{old}) \times level_{new}}{۲۵۶}) \times \frac{۲۵۶}{level_{new} - ۱}$$

در رابطه با عبارت بالا ذکر چند نکته الزامی است:

- شدت روشنایی تصویر اولیه به صورت ۸ بیتی است. در ابتدا برای اینکه محاسبات به صورت اعشاری انجام شود و اطلاعات تقلیل پیدا نکند لازم است تا سطح روشنایی را با تابع double به حالت اعشاری در بیاوریم.
- تابع floor در MATLAB (و بسیاری از زبان‌های برنامه‌نویسی دیگر) مقدار یک عدد را به **بزرگ‌ترین عدد صحیح کوچکتر یا مساوی آن** گرد می‌کند. به عبارت دیگر، همیشه مقدار اعشاری را **به سمت پایین** گرد می‌کند.

برای روشن شدن نحوه عملکرد رابطه بالا در ادامه یک مثال عددی آورده شده است:

اگر بازه شدت روشنایی بین صفر تا ۲۵۵ باشد برای اینکه سطح روشنایی را به ۴ سطح کاهش دهیم لازم است تا بازه بین صفر تا ۲۵۵ به ۴ قسمت تقسیم شود:

بازه (۰ تا ۶۳) را باید به شدت روشنایی صفر نسبت دهیم. بازه (۶۴ تا ۱۲۷) را باید به شدت روشنایی ۸۵ نسبت دهیم. بازه (۱۲۸ تا ۱۹۱) را به شدت روشنایی ۱۷۰ نسبت دهیم و بازه (۱۹۲ تا ۲۵۵) را نیز به شدت روشنایی ۲۵۵ نسبت دهیم.

بنابراین برای رابطه بالا از تابع زیر استفاده شده است:

```
۱. % Define a function to reduce intensity levels
۲. reduce_level = @(img, levels) uint۸(floor(double(img) / (۲۵٦ / levels)) * (۲۵٦ / (levels
- 1)));
```

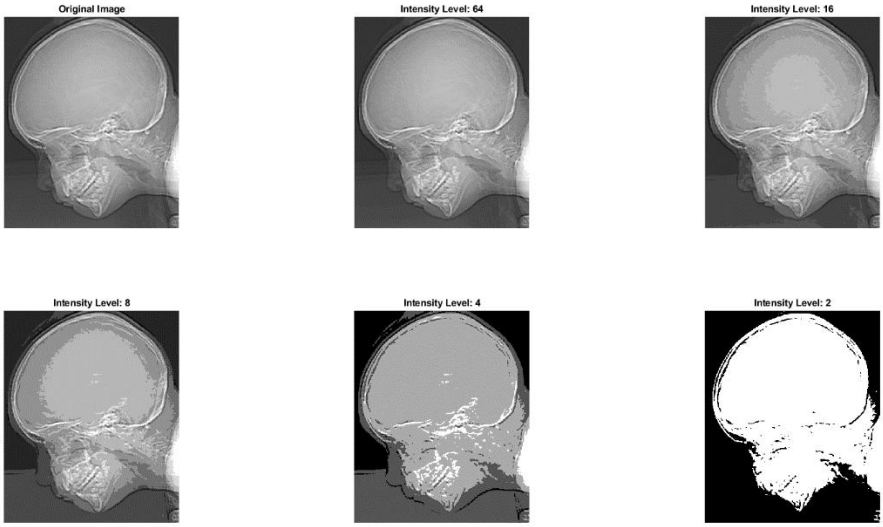
دلیل استفاده از عبارت uint۸ این است که این تابع برای مقادیر شدت روشنایی کمتر از صفر مقدار صفر و برای مقادیر شدت روشنایی بزرگتر از ۲۵۵ مقدار ۲۵۵ را بر می‌گرداند.

در انتها برای نمایش خروجی در سطوح مختلف از یک حلقه به صورت زیر استفاده شده است:

```
1. levels = [64, 16, 8, 4, 2];
2.
3. for i = 1:length(levels)
4.
5.     img_reduced = reduce_level(skull_img, levels(i));
6.
7.     % Display images with different intensity levels
```

```
    subplot(2, 3, i + 1);
    imshow(img_reduced);
    title(sprintf('Intensity Level: %d', levels(i)));
end
```

خروجی در شکل زیر آمده است:



با توجه به نتایج تصاویری با سطوح ۲۵۶ و ۶۴ از نظر بینایی برای اهداف عملی یکسان هستند و تفاوت قابل ملاحظه‌ای دیده نمی‌شود. در سطوح ۱۶ و ۸ و ۴ بدلیل بکار گیری تعداد ناکافی از سطوح شدت روشنایی، اثر لبه کاذب رخ داده است. در واقع در این تصاویر مرزها مشابه خطور تراز توپوگرافی در نقشه هستند. تصویر مربوط به سطح روشنایی تک بیتی نیز بدلیل باینری بودن اطلاعات کمی در اختیار بیننده قرار می‌دهد.

سوال ۳

بخش اول – تغییر مقیاس

تبدیلات هندسی روابط مکانی بین پیکسل‌ها را در یک تصویر اصلاح می‌کند. این تبدیلات را اغلب تبدیلات rubber sheet می‌نامند. تبدیلات هندسی شامل دو عملیات اصلی هستند:

تبدیل مکانی مختصات

درونیابی شدت

به طور کلی تبدیل مختصات به صورت زیر بیان می‌شود:

$$(x, y) = T\{(v, w)\}$$

که (v, w) مختصات پیکسل در تصویر اصلی و (x, y) مختصات پیکسل در تصویر تبدیل یافته است. یکی از متداول ترین تبدیلات مختصات مکانی، تبدیل مستوی (Affine transformation) استکه به صورت زیر تعریف می‌شود:

$$[x, y, ۱] = [v, w, ۱]T$$

که در رابطه بالا با توجه به ماتریس T میتوان تبدیلات مقیاس، دوران، انتقال یا تغییر جهت را انجام داد.

برای تغییر مقیاس بعد از فراخوانی تصویر اصلی از ماتریس تبدیل زیر استفاده شده است:

$$\begin{bmatrix} s_x & ۰ & ۰ \\ ۰ & s_y & ۰ \\ ۰ & ۰ & ۱ \end{bmatrix}$$

که با تنظیم مقادیر s_x و s_y می‌توان ابعاد تصویر را بزرگتر یا کوچکتر یا به صورت همانی تغییر داد. برای مثال این مقادیر را به صورت زیر تنظیم کردیم:

```
۱. sx = ۰,۷;
۲. sy = ۱,۲;
۳.
٤. % Define scaling matrix
٥. T_scale = [sx ۰ ۰; ۰ sy ۰; ۰ ۰ ۱];
```

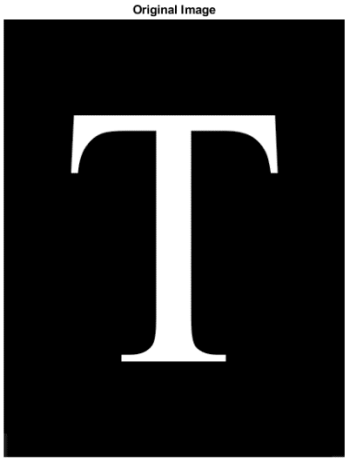
تابع **affine۲d** در متلب برای ایجاد یک آبجکت تبدیل آفین (**Affine Transformation Object**) استفاده می‌شود. این تابع یک ماتریس تبدیل آفین که معمولاً یک ماتریس ۳x۳ است را می‌گیرد و آن را به یک آبجکت تبدیل تبدیل می‌کند که می‌توان از آن در توابع دیگر متلب، مانند **imwarp**، برای اعمال تبدیل‌های هندسی استفاده کرد. تبدیل‌های آفین مجموعه‌ای از تغییرات خطی هستند که می‌توانند شامل مقیاس‌بندی، چرخش، انتقال، برش و انعطاف باشند. این تغییرات به‌صورت خطی عمل می‌کنند و ویژگی‌هایی مانند موازی بودن خطوط و نسبت ثابت فاصله‌ها را حفظ می‌کنند. تابع **affine۲d** این امکان را می‌دهد که یک ماتریس آفین را به یک آبجکت تبدیل کنید که سپس می‌تولند در توابع مختلفی مانند **imwarp** برای اعمال تغییرات به تصویر یا داده‌های دوبعدی دیگر مورد استفاده قرار گیرد. بر این اساس برای تشکیل این شی از کد زیر استفاده شده است:

```
۱. tform_scale = affine۲d(T_scale);
```

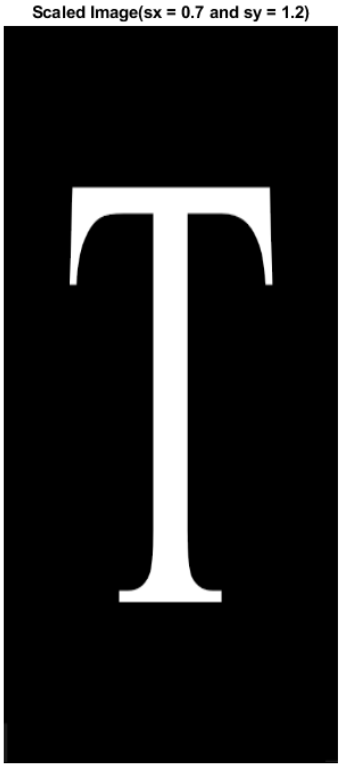
تابع **imwarp** در متلب برای اعمال تبدیل‌های هندسی بر روی تصاویر استفاده می‌شود و این امکان را می‌دهد که از یک آبجکت تبدیل آفین (که توسط تابع **affine۲d** ایجاد می‌شود) برای تغییر شکل تصویر استفاده کنید. نحوه عملکرد آن به این صورت است که ابتدا ماتریس تبدیل (که معمولاً توسط تابع **affine۲d** ایجاد می‌شود) به عنوان ورودی گرفته می‌شود. سپس تبدیل بر روی مختصات پیکسل‌های تصویر اعمال می‌شود و هر پیکسل در تصویر ورودی به مختصات جدیدی که توسط ماتریس تبدیل مشخص شده است، نقشه‌برداری می‌شود. در نهایت، مقدار پیکسل‌ها به مختصات جدید در تصویر خروجی اختصاص داده می‌شود و در نتیجه تصویر خروجی با تغییرات هندسی مورد نظر به‌وجود می‌آید. به همین منظور با کد زیر تصویر جدید ساخته شده است:

```
۱. scaled_image = imwarp(T_img, tform_scale);
```

نتایج حاصل در زیر آورده شده است:



تصویر بالا تصویر اصلی است که بعد از اعمال تغییر مقیاس به صورت تصویر صفحه بعد در آمده است:



بخش دوم – دوران

برای دوران تصویر اصلی از ماتریس تبدیل زیر استفاده شده است:

$$\begin{bmatrix} \cos\theta & \sin\theta & ۰ \\ -\sin\theta & \cos\theta & ۰ \\ ۰ & ۰ & ۱ \end{bmatrix}$$

ابتدا مقدار چرخش در ۳۰ دره تنظیم می‌شود. سپس آنرا با دستور deg۲rad به رادیان تبدیل می‌کنیم.

```
۱. theta = ۳۰;  % Rotation angle in degrees
۲. theta_rad = deg۲rad(theta);  % Convert to radians
```

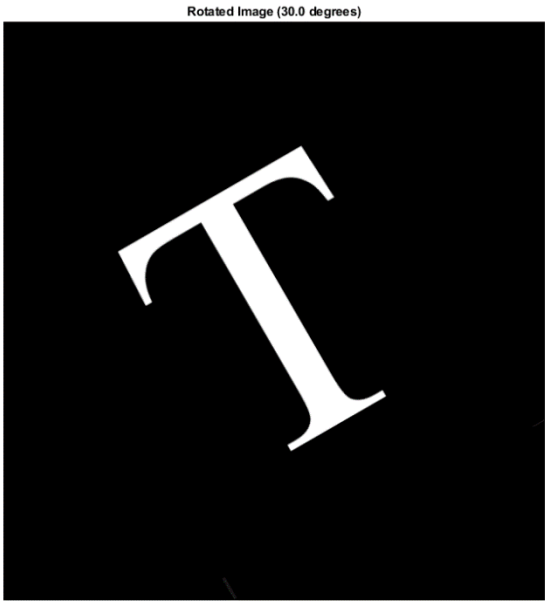
ماتریس تبدیل به صورت زیر تنظیم شده است:

```
۱. % Define the rotation matrix
۲. T_rotate = [cos(theta_rad), -sin(theta_rad), ۰;
۳.             sin(theta_rad), cos(theta_rad), ۰;
۴.             ۰, ۰, ۱];
```

در نهایت مشابه با بخش قبل تصویر خروجی ساخته شده است:

```
% Apply rotation using imwarp
tform_rotate = affine2d(T_rotate);
rotated_img = imwarp(T_img, tform_rotate);
```

نتیجه در شکل زیر آورده شده است:



بخش سوم – انتقال

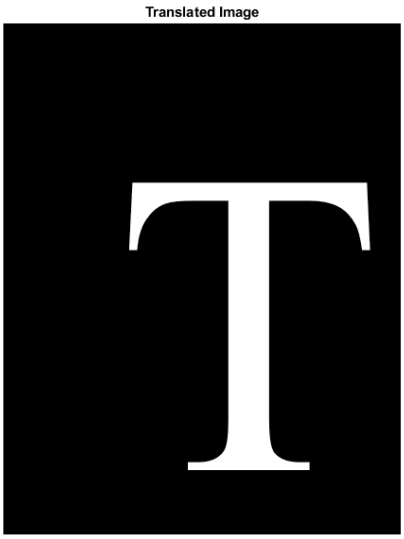
برای محاسبه این بخش از تابع translationFunction.m استفاده شده که تبدیلات را به صورت دستی انجام می‌دهد. تابع به صورت زیر تعریف شده است:

```
function translatedImage = translationFunction(inputImage, tx, ty)
    % Get the dimensions of the input image
    [rows, cols, channels] = size(inputImage);

    % Create a new image with the same dimensions as the input image
    translatedImage = zeros(rows, cols, channels, 'like', inputImage);

    % Loop through each pixel of the input image
    for m = 1:rows
        for n = 1:cols
            x = m + tx;
            y = n + ty;

            % Ensure the new coordinates are within the image bounds
            if x > 0 && x <= rows && y > 0 && y <= cols
                translatedImage(x, y, :) = inputImage(m, n, :);
```

```
            end
        end
    end
end
```

تابع سه مقدار میزان انتقال در جهت افقی و در جهت عمودی و تصویر ورودی را می‌گیرد. سپس برای ثابت ماند تصویر بعد از تبدیل لازم است تا با استفاده از ابعاد تصویر اولیه به تولید یک تصویر با مقادیر صفر و ابعاد مشابه تصویر اولیه پرداخته شود. در نهایت با استفاده از دو حلقه توو در توو کان پیکسلها تغییر می‌کند و در پایان مقدار شدت روشنایی پیکسلهای اولیه به نقاط انتقال یافته اختصاص پیدا می‌کند. خروجی برای مقادیر تنظیمی در کد زیر در ادامه آورده شده است:

```
%% Translation Transformation
tx = 50;
ty = 50;

translatedImage = translationFunction(T_img, tx, ty);
% Display the translated image
figure;
imshow(translatedImage);
title('Translated Image');
```



بخش چهارم – تغییر جهت عمودی و افقی

برای تغییر جهت افقی و عمودی تصویر اصلی از ماتریس تبدیل زیر استفاده شده است:

$$\begin{bmatrix} ۱ & Shx & ۰ \\ Shy & ۱ & ۰ \\ ۰ & ۰ & ۱ \end{bmatrix}$$

مقادیر تغییر جهت در جهت محوری افقی و عمودی به صورت زیر تنظیم شده است:

```
۱. % Define the shear factors
۲. Shx = ۰,۳;  % Shear in X direction
۳. Shy = ۰,۲;  % Shear in Y direction
٤.
٥. % Define the shear matrix
٦. T_shear = [۱ Shx ۰; Shy ۱ ۰; ۰ ۰ ۱];
```

سپس مشابه بخش‌های قبل ماتریس آفین به صورت زیر تنظیم شده است:

```
۱. % Apply shear using imwarp
۲. tform_shear = affine۲d(T_shear);
۳. sheared_img = imwarp(T_img, tform_shear);
```

نتیجه در شکل زیر نمایش داده شده است:

سوال ۴

عملیات منطقی در پردازش تصویر و محاسبات دیجیتال، به کارگیری عملیات‌های پایه‌ای منطقی برای تحلیل داده‌ها است. در این عملیات‌ها، معمولاً از مقادیر باینری (۰ و ۱) استفاده می‌شود. در اینجا، به معرفی و توضیح برخی از این عملیات‌های منطقی پرداخته می‌شود که در کد استفاده کرده‌ایم:

۱. عملیات NOT (نقیض)

عملیات **NOT**(یا نقیض) ساده‌ترین عملیات منطقی است که هر مقدار باینری را معکوس می‌کند:

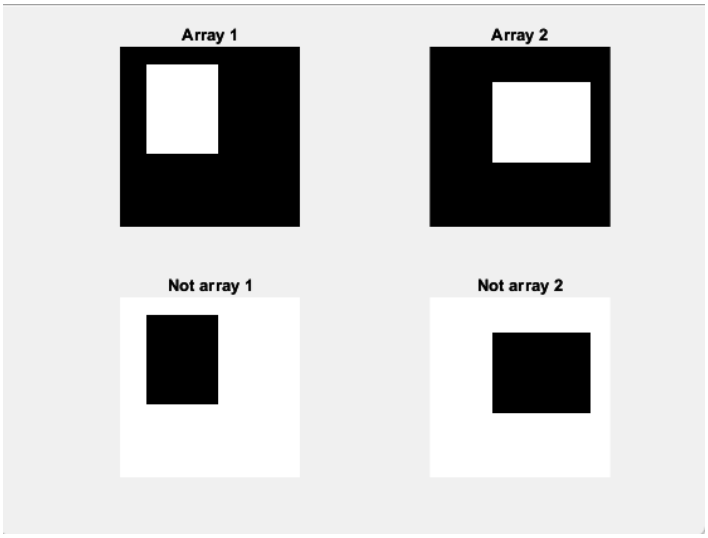
A	NOT(A)
۰	۱
۱	۰

برای پیاده سازی این عملگر به صورت پیکسل به پیکسل کد زیر نوشته شده است:

```
% Function to perform NOT operation (voxel by voxel)
function result = logical_not(A)
    [rows, cols] = size(A);
    result = zeros(rows, cols);  % Initialize result matrix with zeros

    for x = 1:rows
        for y = 1:cols
            if A(x, y) == 1
                result(x, y) = 0;  % If A is 1, set to 0
             else
                 result(x, y) = ١;  % If A is ٠, set to ١
             end
         end
     end
end
```

تابع logical_not که عملیات نقیض (NOT) را به صورت دستی روی یک ماتریس باینری انجام می‌دهد، ابتدا یک ماتریس ورودی Aمی‌گیرد که حاوی مقادیر ۰و ۱است. این ماتریس می‌تواند نمایانگر یک تصویر سیاه و سفید باشد. سپس، یک ماتریس جدید به نام result با ابعاد مشابه ماتریس ورودی ایجاد می‌کند که به طور پیش‌فرض

تمام مقادیر آن صفر است. بعد از آن، با استفاده از دو حلقه for، به صورت پیکسل به پیکسل، تمام عناصر ماتریس ورودی پیمایش می‌شود. در داخل این حلقه‌ها، اگر مقدار پیکسل برابر ۱باشد، در ماتریس result مقدار آن به ۰ تغییر می‌کند و اگر مقدار پیکسل برابر ۰باشد، مقدار آن در ماتریس resultبه ۱تغییر می‌یابد. در نهایت، پس از انجام این عملیات روی تمام پیکسل‌ها، ماتریس resultکه حالا نسخه معکوس‌شده ماتریس ورودی است، به عنوان خروجی برگردانده می‌شود. این کد به صورت دستی عملیات نقیض را انجام می‌دهد و نیازی به توابع داخلی MATLABندارد.



۲. عملیات AND

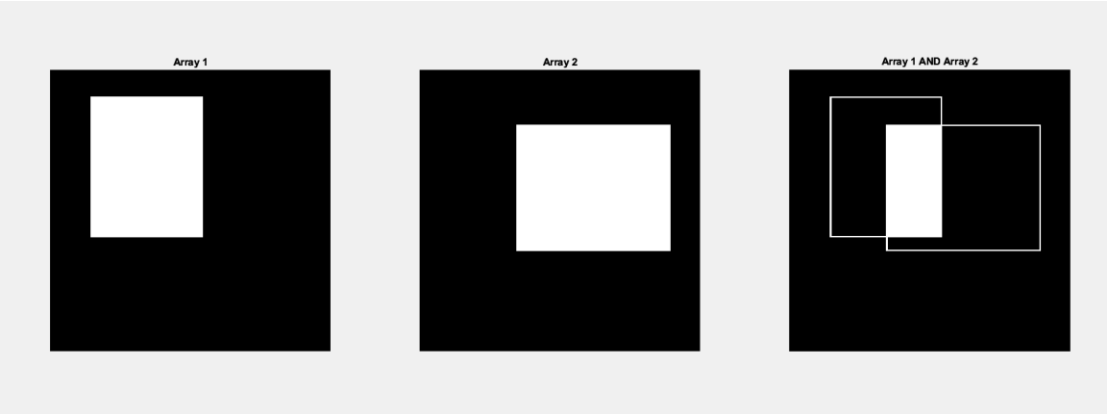
عملیات **AND**بر اساس این قانون عمل می‌کند: اگر هر دو ورودی برابر با ۱ باشند، نتیجه ۱ خواهد بود؛ در غیر این صورت، نتیجه ۰ خواهد بود.

A	B	A and B
۰	۰	۰
۰	۱	۰
۱	۰	۰
۱	۱	۱

این عملیات معمولاً برای **تشخیص نواحی مشترک** در دو تصویر استفاده می‌شود. برای اجرای این عملیات از تابع زیر استفاده شده است:

```
% AND operation
function result = logical_and(A, B)
    [rows, cols] = size(A);
    result = zeros(rows, cols);
    for x = 1:rows
        for y = 1:cols
            result(x, y) = A(x, y) * B(x, y);
        end
    end
end
```

تابع logical_and برای انجام عملیات منطقی AND بر روی دو ماتریس باینری طراحی شده است. این تابع دو ورودی Aو Bمی‌گیرد که هرکدام از آن‌ها ماتریس‌هایی از مقادیر ۰و ۱هستند. ابتدا ابعاد هر دو ماتریس با استفاده از دستور [rows, cols] = size(A)محاسبه می‌شود، که در آن rowsتعداد ردیف‌ها و colsتعداد ستون‌ها را نشان می‌دهد. سپس، یک ماتریس جدید resultبا همان ابعاد و با مقادیر پیش‌فرض صفر ایجاد می‌شود تا نتایج عملیات منطقی در آن ذخیره شود. در ادامه، دو حلقه تو در تو(for x = ۱:rows) و (for y = ۱:cols) برای پیمایش هر عنصر از ماتریس‌ها Aو Bنوشته شده‌اند. این حلقه‌ها به طور جداگانه تمام ردیف‌ها و ستون‌های ماتریس‌ها را مرور می‌کنند. در داخل این حلقه‌ها، برای هر پیکسل، عملیات AND به صورت ضرب مقادیر معادل از ماتریس‌های Aو Bانجام می‌شود. این به این معناست که اگر هر دو پیکسل برابر ۱باشند، نتیجه برابر ۱خواهد شد و در غیر این صورت نتیجه ۰خواهد بود. در نهایت، بعد از تکمیل این عملیات برای تمام پیکسل‌ها، ماتریس resultکه شامل نتایج نهایی عملیات AND است، به عنوان خروجی تابع برمی‌گردد.



۳. عملیات OR

عملیات **OR** بر اساس این قانون عمل می‌کند: اگر هر یک از ورودی‌ها ۱ باشند، نتیجه ۱ خواهد بود. در غیر این صورت، نتیجه ۰ خواهد بود.

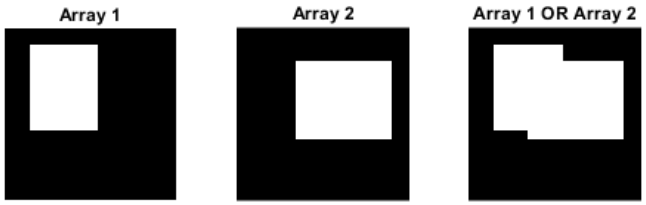
A	B	A OR B
۰	۰	۰
۰	۱	۱
۱	۰	۱
۱	۱	۱

این عملیات معمولاً برای **تشخیص نواحی موجود در یکی از تصاویر** یا در هر دو تصویر استفاده می‌شود.برای پیاده سازی این عملگر از کد زیر استفاده شده است:

```
% OR operation
function result = logical_or(A, B)
    [rows, cols] = size(A);
    result = zeros(rows, cols);
    for x = 1:rows
        for y = 1:cols
            result(x, y) = A(x, y) + B(x, y);
        end
    end
end
```

تابع logical_or برای انجام عملیات منطقی OR بر روی دو ماتریس باینری طراحی شده است. این تابع دو ورودی Aو Bمی‌گیرد که هرکدام از آن‌ها ماتریس‌هایی از مقادیر ۰و ۱هستند. ابتدا ابعاد هر دو ماتریس با استفاده از دستور [rows, cols] = size(A)محاسبه می‌شود، که در آن rowsتعداد ردیف‌ها و colsتعداد ستون‌ها را نشان می‌دهد. سپس، یک ماتریس جدید result با همان ابعاد و با مقادیر پیش‌فرض صفر ایجاد می‌شود تا نتایج عملیات منطقی در آن ذخیره شود. در ادامه، دو حلقه تو در تو (for x = ۱:rows) و (for y = ۱:cols) برای پیمایش هر عنصر از ماتریس‌ها Aو Bنوشته شده‌اند. این حلقه‌ها به طور جداگانه تمام ردیف‌ها و ستون‌های ماتریس‌ها را مرور

می‌کنند. در داخل این حلقه‌ها، برای هر پیکسل، عملیات OR با استفاده از عمل جمع انجام می‌شود. در واقع، اگر هر کدام از پیکسل‌ها برابر ۱باشند، نتیجه برابر ۱خواهد بود (چرا که در جمع مقادیر ۰ + ۱یا ۱ + ۰برابر ۱و در ۱ + ۱نیز نتیجه ۱خواهد بود). تنها زمانی که هر دو پیکسل برابر ۰باشند، نتیجه جمع برابر ۰خواهد شد. در نهایت، بعد از تکمیل این عملیات برای تمام پیکسل‌ها، ماتریس resultکه شامل نتایج نهایی عملیات OR است، به عنوان خروجی تابع برمی‌گردد.



٤. عملیات AND-NOT

عملیات **AND-NOT** ترکیبی از عملیات AND و NOT است. در این عملیات، ابتدا **NOT** روی تصویر دوم اعمال می‌شود، سپس **AND** بین تصویر اول و تصویر معکوس‌شده انجام می‌شود. این عملیات معمولاً برای حذف نواحی مشترک در دو تصویر استفاده می‌شود. در **A AND NOT B** فقط نواحی که در A وجود دارند و در B نیستند به نتیجه می‌روند.

A	B	A AND-NOT B
۰	۰	۰
۰	۱	۰
۱	۰	۱

۰	۱	۱

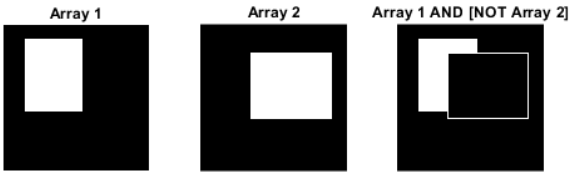
برای اجرای این عملگر از کد زیر استفاده شده است:

```
% AND-NOT operation
function result = logical_and_not(A, B)
    [rows, cols] = size(A);
    result = zeros(rows, cols);
    for x = 1:rows
        for y = 1:cols
            result(x, y) = A(x, y) * ~B(x, y);
        end
    end
end
```

تابع logical_and_not به منظور انجام عملیات منطقی AND-NOT بین دو ماتریس باینری طراحی شده است. ورودی‌های این تابع دو ماتریس Aو Bهستند که هرکدام مقادیر ۰و ۱را در خود دارند. هدف از این تابع انجام عملیات منطقی AND بین ماتریس Aو معکوس ماتریس) Bکه همان NOT ماتریس Bاست) می‌باشد.

در ابتدا، ابعاد ماتریس‌های ورودی Aو Bبا استفاده از دستور [rows, cols] = size(A)استخراج می‌شود، که تعداد ردیف‌ها و ستون‌های هر دو ماتریس را مشخص می‌کند. سپس یک ماتریس جدید به نام resultبا ابعاد مشابه ماتریس‌های ورودی ایجاد می‌شود که تمام مقادیر آن برابر با صفر است. این ماتریس برای ذخیره نتایج عملیات منطقی AND-NOT استفاده خواهد شد.

سپس درون دو حلقه تو در تو (for x = ۱:rows)و (for y = ۱:cols)، هر یک از پیکسل‌های ماتریس‌های Aو B بررسی می‌شود. در داخل این حلقه‌ها، برای هر پیکسل از ماتریس‌ها، عمل منطقی AND-NOT انجام می‌شود. این کار با ضرب ماتریس Aدر معکوس ماتریس) Bکه با استفاده از عملگر ~انجام می‌شود) صورت می‌گیرد. در واقع، معکوس ماتریس Bبه این معناست که مقادیر ۱تبدیل به ۰و مقادیر ۰تبدیل به ۱می‌شوند. نتیجه نهایی ضرب این دو مقدار به طور طبیعی همان نتیجه عملیات AND-NOT است. در این عملیات، اگر پیکسل در A برابر ۱باشد و پیکسل متناظر در Bبرابر ۰باشد، نتیجه نهایی ۱خواهد بود. در غیر این صورت، نتیجه برابر ۰ خواهد بود. در نهایت، پس از تکمیل عملیات برای تمام پیکسل‌ها، ماتریس resultکه شامل نتایج نهایی عملیات منطقی AND-NOT است، به عنوان خروجی تابع برگردانده می‌شود. این تابع به صورت دستی و بدون استفاده از توابع داخلی MATLAB عملیات منطقی AND-NOT را بین دو ماتریس انجام می‌دهد و نتیجه را در قالب یک ماتریس باینری ارائه می‌دهد.



۵. عملیات XOR

عملیات (Exclusive OR) **XOR** بر اساس این قانون عمل می‌کند: اگر دو ورودی برابر باشند، نتیجه ۰ خواهد بود؛ در غیر این صورت، نتیجه ۱ خواهد بود.

A	B	A XOR B
۰	۰	۰
۰	۱	۱
۱	۰	۱
۱	۱	۰

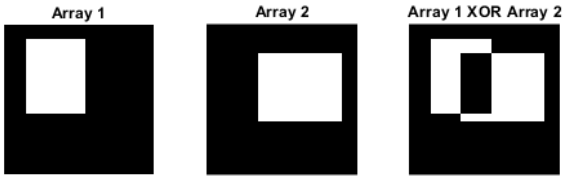
این عملیات برای **تشخیص تفاوت‌ها** در دو تصویر استفاده می‌شود.برای پیاده سازی این عملگر از کد زیر استفاده شده است:

```
1. % XOR operation
2. function result = manual_xor(A, B)
3. 
4.     [rowsA, colsA] = size(A);
5.     [rowsB, colsB] = size(B);
6. 
7.     if rowsA ~= rowsB || colsA ~= colsB
8.         error('Matrices must have the same size!');
9.     end
١٠.
```

```
    result = zeros(rowsA, colsA);

    for y = ١:rowsA
        for x = ١:colsA
            if A(y, x) ~= B(y, x)
                result(y, x) = ١;
            else
                result(y, x) = ٠;
            end
        end
    end
end
```

تابع manual_xor به منظور انجام عملیات منطقی XOR بین دو ماتریس باینری طراحی شده است. ورودی‌های این تابع دو ماتریس A و B هستند که مقادیر باینری ۰ و ۱ را در خود دارند. هدف این تابع اعمال عملیات XOR بین مقادیر متناظر در ماتریس‌های A و B است. در ابتدا، ابعاد ماتریس‌های ورودی A و B با استفاده از دستور [rowsA, colsA] = size(A) و [rowsB, colsB] = size(B) استخراج می‌شود که تعداد ردیف‌ها و ستون‌های هر دو ماتریس را مشخص می‌کند. سپس با استفاده از دستور شرطی، بررسی می‌شود که آیا ابعاد دو ماتریس برابر هستند یا خیر. اگر ابعاد متفاوت باشند، یک پیام خطا به کاربر نمایش داده می‌شود تا اطلاع دهد که ماتریس‌ها باید اندازه یکسانی داشته باشند. این امر برای اطمینان از درستی عملیات منطقی است، چرا که عملیات XOR تنها بین ماتریس‌هایی با ابعاد یکسان قابل انجام است. پس از تایید تطابق ابعاد، یک ماتریس جدید به نام result ایجاد می‌شود که ابعاد آن مشابه ماتریس‌های ورودی A و B است و تمام مقادیر آن برابر با صفر قرار می‌گیرد. این ماتریس برای ذخیره نتایج عملیات XOR استفاده خواهد شد.

سپس درون دو حلقه تو در تو (for y = ١:rowsA) و (for x = ١:colsA)، هر یک از پیکسل‌های ماتریس‌های A و B بررسی می‌شود. در داخل این حلقه‌ها، عمل منطقی XOR انجام می‌شود. این کار با مقایسه مقادیر متناظر در ماتریس‌های A و B صورت می‌گیرد. اگر مقادیر پیکسل‌ها در A و B برابر نباشند (یعنی یکی از آن‌ها ۱ و دیگری ۰ باشد)، نتیجه در ماتریس result برابر ۱ خواهد بود. در غیر این صورت، اگر مقادیر برابر باشند، نتیجه برابر ۰ خواهد بود. این منطق دقیقاً همان عملکرد XOR است که در آن دو مقدار متفاوت نتیجه ۱ می‌دهد و دو مقدار مشابه نتیجه ۰ می‌دهد. در نهایت، پس از تکمیل عملیات برای تمام پیکسل‌ها، ماتریس result که شامل نتایج نهایی عملیات منطقی XOR است، به عنوان خروجی تابع برگردانده می‌شود.



در ادامه به توضیح کد اصلی برای پیاده سازی این عملیات بر روی دو تصویر می‌پردازیم:

در این بخش از کد، ابتدا دو آرایه ۲۰۰x۲۰۰ که با صفر پر شده‌اند، ایجاد می‌شوند. این آرایه‌ها به‌طور پایه‌ایی برای ذخیره‌سازی داده‌های منطقی (صفر و یک) طراحی شده‌اند. هدف از این بخش اضافه کردن دو مستطیل در این آرایه‌ها است که به صورت منطقی با استفاده از مقادیر ۱ و ۰ مشخص می‌شوند. در ابتدا با استفاده از تابع zeros(۲۰۰, ۲۰۰), دو آرایه array۱ و array۲ به ابعاد ۲۰۰x۲۰۰ ساخته می‌شوند. تمام مقادیر اولیه این آرایه‌ها صفر هستند. این آرایه‌ها در ادامه برای افزودن مستطیل‌ها به آن‌ها استفاده خواهند شد. پارامترهای مربوط به مستطیل اول که قرار است در آرایه array۱ قرار گیرد، شامل موقعیت شروع مستطیل و ابعاد آن می‌شود. موقعیت شروع مستطیل از سلول (۲۰, ۳۰) در آرایه آغاز می‌شود و ابعاد مستطیل به طول ۸۰ و ارتفاع ۱۰۰ تعریف شده است. سپس با استفاده از این پارامترها، مقادیر ۱ به درون منطقه مستطیل در آرایه array۱ اختصاص داده می‌شود. مشابه مرحله قبل، پارامترهای مستطیل دوم برای آرایه array۲ تعریف می‌شود. این مستطیل از نقطه شروع (۴۰, ۷۰) شروع می‌شود و ابعاد آن ۱۱۰ به طول و ۹۰ به ارتفاع است. سپس، با استفاده از این پارامترها، مقادیر ۱ به منطقه مستطیل در آرایه array۲ اختصاص داده می‌شود.

```
%% In this section, we want to implement some important logical operations.
% Create two 200x200 arrays filled with zeros
array1 = zeros(200, 200);
array2 = zeros(200, 200);

% Define the rectangle parameters in array1
x_start1 = 20;
y_start1 = 30;
rect_length1 = 80;
rect_height١ = ١٠٠;

% Add the rectangle (set the region to ١)
```

```
array١(x_start١:x_start١+rect_height١-١, y_start١:y_start١+rect_length١-١) = ١;

% Define the rectangle parameters in array١
x_start٢ = ٤٠;
y_start٢ = ٧٠;
rect_length٢ = ١١٠;
rect_height٢ = ٩٠;

% Add the rectangle (set the region to ١)
array٢(x_start٢:x_start٢+rect_height٢-١, y_start٢:y_start٢+rect_length٢-١) = ١;
```

در این بخش از کد، عملیات منطقی "NOT" بر روی دو آرایه‌ی array۱ و array۲ که پیش‌تر ایجاد شـــده‌اند، انجام می‌شود. عملیات "NOT" یک عملیات منطقی است که مقادیر ۰ را به ۱ و مقادیر ۱ را به ۰ تبدیل می‌کند. این عملیات برای بررســی تفاوت‌ها و تغییرات در داده‌های باینری اســتفاده می‌شــود. در ابتدا با اســتفاده از تابع imshow درون یک پنجره گرافیکی، دو آرایه array۱ و array۲ که شامل مستطیل‌های باینری هستند، نمایش داده می‌شوند. این نمایش برای اطمینان از صحت قرارگیری مستطیل‌ها در آرایه‌ها صورت می‌گیرد. در این بخش از کد، دو مستطیل که قبلاً در آرایه‌ها قرار داده شده‌اند، به طور جداگانه و در دو زیر پنجره (subplot) به نمایش در می‌آیند. این بخش از کد با عنوان‌های "Array ۱" و "Array ۲" در بالای هر تصویر نمایش داده می‌شود. در این مرحله، از تابع logical_not برای انجام عملیات "NOT" روی هر دو آرایه اســتفاده می‌شــود. این تابع که به صــورت دســتی پیاده‌ســازی شــده اســت، مقادیر ۱ را به ۰ و مقادیر ۰ را به ۱ تبدیل می‌کند. این تغییرات برای آرایــه‌هــای array۱ و array۲ بــه ترتیــب در متغیرهــای image_not_array۱ و image_not_array۲ ذخیره می‌شوند. نتیجه‌ی این تغییرات به این صورت است که مستطیل‌های موجود در این آرایه‌ها از حالت ۱ به ۰ و از حالت ۰ به ۱ تغییر خواهند کرد. پس از انجام عملیات "NOT" ، نتایج به دســـت آمده در دو زیر پنجره دیگر به نمایش در می‌آیند. اولین پنجره نتایج تغییرات بر روی array۱ را نشـان می‌دهد که در آن مسـتطیل‌ها به صـورت معکوس نمایش داده می‌شــوند. مشــابه همین روند برای array۲ انجام می‌شــود و نتیجه آن در پنجره بعدی به نمایش در می‌آید. عنوان‌های "Not array ۱" و "Not array ۲" در بالای تصاویر نمایش داده شـده به‌طور واضـح نتایج اعمال عملیات "NOT" را مشخص می‌کنند.

```
%% NOT
% Display the arrays to verify the rectangle placement
figure(1);
subplot(2,2,1);
imshow(array1);
title('Array 1');

subplot(2,2,2);
imshow(array2);
title('Array ٢');

image_not_array١ = logical_not(array١);
```

```
image_not_array٢ = logical_not(array٢);

subplot(٢,٢,٣);
imshow(image_not_array١);
title('Not array ١');

subplot(٢,٢,٤);
imshow(image_not_array٢);
title('Not array ٢');
```

در این بخش از کد، عملیات منطقی "AND" بر روی دو آرایه‌ی باینری array١ وarray٢ انجام می‌شود. عملیات "AND"در منطق باینری یک عملیات پایه‌ای اســـت که دو ورودی را گرفته و تنها زمانی خروجی آن برابر ١ خواهد شد که هر دو ورودی به طور همزمان برابر با ١ باشند. این عملیات برای ترکیب دو آرایه باینری و مشاهده تداخل و اشـتراک آن‌ها در مقادیر ١ اسـتفاده می‌شـود. ابتدا از تابع logical_and برای اعمال عملیات "AND" بر روی دو آرایه‌ی array١ وarray٢ اســتفاده می‌شــود. این تابع به‌طور عنصــر به عنصــر دو آرایه را با هم ترکیب می‌کند و نتیجه‌ی آن در متغیر image_and ذخیره می‌شـــود. مقادیر ١ در این متغیر تنها در جاهایی که هر دو آرایه در آن موقعیت مقدار ١ دارند، ذخیره خواهند شـد. به عبارت دیگر، این عمل به‌طور دقیق تداخل و اشـتراک دو مستطیل باینری را نشان می‌دهد. پس از اعمال عملیات"AND" ، برای شبیه‌سازی مرز مستطیل‌ها در نتایج، در مرزهای هر دو مستطیل که در آرایه‌ها قرار دارند، مقادیر ١ اضافه می‌شود. این کار با استفاده از ایندکس‌گذاری مسـتقیم انجام می‌شـود، که در آن برای هر مسـتطیل (هم در array١ وهم در array٢)، مقدار ١ در موقعیت‌های خاص مرزی تنظیم می‌شـود. در نهایت، مرزهای مسـتطیل‌ها در تصـویر image_and با رنگ سـفید (مقدار ١) به نمایش در می‌آید. .در نهایت، نتایج به‌دست آمده به‌طور گرافیکی نمایش داده می‌شود

```
%% AND
% Find the border of the rectangles (the edges of the 1's in the arrays)

image_and = logical_and(array1, array2);

% For array1
image_and(x_start1,y_start1:y_start1+rect_length1-1) = 1;
image_and(x_start1+rect_height1-1,y_start1:y_start1+rect_length1-1) = 1;
image_and(x_start1:x_start1+rect_height1-1, y_start1) = 1;
image_and(x_start١:x_start١+rect_height١-١, y_start١+rect_length١-١) = ١;

% For array٢
image_and(x_start٢,y_start٢:y_start٢+rect_length٢-١) = ١;
image_and(x_start٢+rect_height٢-١,y_start٢:y_start٢+rect_length٢-١) = ١;
image_and(x_start٢:x_start٢+rect_height٢-١, y_start٢) = ١;
image_and(x_start٢:x_start٢+rect_height٢-١, y_start٢+rect_length٢-١) = ١;

figure(٢);
subplot(١,٣,١);
imshow(array١);
title('Array ١');
subplot(١,٣,٢);
imshow(array٢);
```

```
٢٤. title('Array ٢');
٢٥. subplot(١,٣,٣);
٢٦. imshow(image_and);
٢٧. title('Array ١ AND Array ٢');
```

در این بخش از کد، عملیات منطقی "OR" بر روی دو آرایه‌ی باینری array۱ وarray۲ انجام می‌شود. عملیات "OR"در منطق باینری یک عملیات پایه‌ای است که دو ورودی را گرفته و تنها زمانی خروجی آن برابر با ۰ خواهد شد که هر دو ورودی برابر ۰ باشند. در تمامی دیگر حالت‌ها، خروجی ۱ خواهد بود. این عملیات برای ترکیب دو آرایه باینری و نمایش نواحی‌ای که در هر کدام از آرایه‌ها مقدار ۱ دارند، به کار می‌رود. ابتدا از تابع logical_orبرای اعمال عملیات "OR" بر روی دو آرایه‌ی array۱ وarray۲ استفاده می‌شود. این تابع به‌طور عنصر به عنصر دو آرایه را با هم ترکیب می‌کند و نتیجه‌ی آن در متغیر image_orذخیره می‌شود. مقادیر ۱ در این متغیر در جاهایی قرار می‌گیرند که حداقل یکی از دو آرایه در آن موقعیت مقدار ۱ داشته باش. در عملیات "OR"، برای هر موقعیت از آرایه‌ها، اگر در هر کدام از آرایه‌ها مقدار ۱ باشد، نتیجه‌ی عملیات در آن موقعیت نیز ۱ خواهد شد. این بدان معناست که هر ناحیه‌ای که در یکی از مستطیل‌ها قرار داشته باشد، در تصویر نهایی به‌عنوان ۱ نمایش داده می‌شود. این عملیات به‌طور کلی تمامی نواحی را که در یکی از دو مستطیل قرار دارند، پوشش می‌دهد. پس از انجام عملیات"OR" ، نتایج به‌طور گرافیکی نمایش داده می‌شوند.

```
1. %% OR
2.
3. image_or = logical_or(array1, array2);
4.
5. figure(3);
6. subplot(1,3,1);
7. imshow(array1);
8. title('Array 1');
9.
١٠. subplot(١,٣,٢);
١١. imshow(array٢);
١٢. title('Array ٢');
١٣.
١٤. subplot(١,٣,٣);
١٥. imshow(image_or);
١٦. title('Array ١ OR Array ٢');
```

در این بخش از کد، عملیات منطقی "AND-NOT" بر روی دو آرایه باینری array۱ وarray۲ انجام می‌شود. عملیات "AND-NOT" ترکیبی از دو عمل منطقی "AND" و "NOT" است. ابتدا، عملیات "NOT" روی آرایه دوم (array۲) اعمال می‌شود که باعث معکوس کردن مقادیر ۰ و ۱ می‌شود. سپس، عملیات "AND" روی آرایه اول (array۱) و آرایه دوم معکوس‌شده انجام می‌شود. به عبارت ساده‌تر، این عملیات تنها زمانی مقدار ۱ در

خروجی می‌گذارد که مقدار متناظر در array۱برابر با ۱ باشـد و در array۲مقدار معکوس شـده آن برابر با ۱ باشـد (یعنی در array۲مقدار ۰). بنابراین، در نتیجه، تنها نواحی که در array۱دارای مقدار ۱ هسـتند و در array۲دارای مقدار ۰ هسـتند، در تصـویر نهایی نمایش داده می‌شـوند. در این کد، ابتدا آرایه‌های array۱و array۲ با مستطیل‌هایی که مقادیر آن‌ها ۱ است، پر می‌شوند. سپس عملیات "AND-NOT" روی این دو آرایه انجام می‌شود و نتایج آن در تصویر image_and_notنمایش داده می‌شود.

```
image_and_not = logical_and_not(array1, array2);

figure(4);
subplot(1,3,1);
imshow(array1);
title('Array 1');

subplot(1,3,2);
imshow(array2);
title('Array ۲');

subplot(۱,۳,۳);

% For array۲
image_and_not(x_start۲,y_start۲:y_start۲+rect_length۲-۱) = ۱;
image_and_not(x_start۲+rect_height۲-۱,y_start۲:y_start۲+rect_length۲-۱) = ۱;
image_and_not(x_start۲:x_start۲+rect_height۲-۱, y_start۲) = ۱;
image_and_not(x_start۲:x_start۲+rect_height۲-۱, y_start۲+rect_length۲-۱) = ۱;

imshow(image_and_not);
title('Array ۱ AND [NOT Array ۲]');
```

در این بخش از کد، عملیات XOR یا Exclusive OR روی دو آرایه array۱و array۲انجام می‌شود. این عملیات منطقی به این صورت عمل می‌کند که برای هر جفت مقدار در دو آرایه، اگر مقادیر آن‌ها برابر نباشند (یعنی یکی ۰ و دیگری ۱ باشـد)، نتیجه ۱ خواهد بود، اما اگر مقادیر مشـابه باشـند (یعنی هر دو ۰ یا هر دو ۱)، نتیجه ۰ خواهد بود. این به این معناسـت که فقط در مکان‌هایی که دو آرایه مقادیر مختلف دارند، نتیجه عملیات ۱ می‌شـود. در ابتدا، دو آرایه array۱و array۲که قبلاً مسـتطیل‌هایی از ۱ دارند، به تابع manual_xorداده می‌شـوند. این تابع به طور دسـتی و با بررسـی هر خانه از این دو آرایه، مقدار XOR را محاسـبه می‌کند. به این صـورت که برای هر خانه از آرایه‌ها، اگر مقادیر آن‌ها متفاوت باشـند، در آرایه نتیجه، ۱ قرار می‌دهد و اگر مشـابه باشند، ۰ قرار می‌دهد.

```
%% XOR

figure(5);
image_xor = manual_xor(array1, array2);

subplot(1,3,1);
imshow(array1);
```

```
 8. title('Array 1');
 9.
١٠. subplot(١,٣,٢);
١١. imshow(array٢);
١٢. title('Array ٢');
١٣.
١٤. subplot(١,٣,٣);
١٥. imshow(image_xor);
١٦. title('Array ١ XOR Array ٢');
```

سوال ۵

هموارسازی تصویر یکی از مراحل مهم پیش‌پردازش در پردازش تصویر است که برای کاهش نویز و بهبود کیفیت تصویر استفاده می‌شود. یکی از روش‌های رایج برای این کار، **میانگین‌گیری محلی (فیلتر میانگین)** است که در آن مقدار هر پیکسل با میانگین مقادیر پیکسل‌های همسایه جایگزین می‌شود. در این کد متلب، فیلتر میانگین‌گیری با چهار اندازه‌ی مختلف **۳×۳، ۷×۷، ۱۱×۱۱ و ۲۳×۲۳** روی تصویر اعمال شده است.در ادامه به توضیح کد و بررسی نتایج می‌پردازیم:

بعد از فراخوانی تصویر در ابتدا به تعریف کرنل‌ها پرداخته شده است:

```
١. % Define a ٣*٣ avraging filter
٢. kernel_٣ = ones(٣, ٣) / ٩;
٣. kernel_٧ = ones(٧, ٧) / ٤٩;
٤. kernel_١١ = ones(١١, ١١) / ١٢١;
٥. kernel_٢٣ = ones(٢٣, ٢٣) / ٥٢٩;
```

نحوه عملکرد کرنل برای حالتیکه ما از یک کرنل ۳ در ۳ استفاده می‌کنیم در زیر آورده شده است.

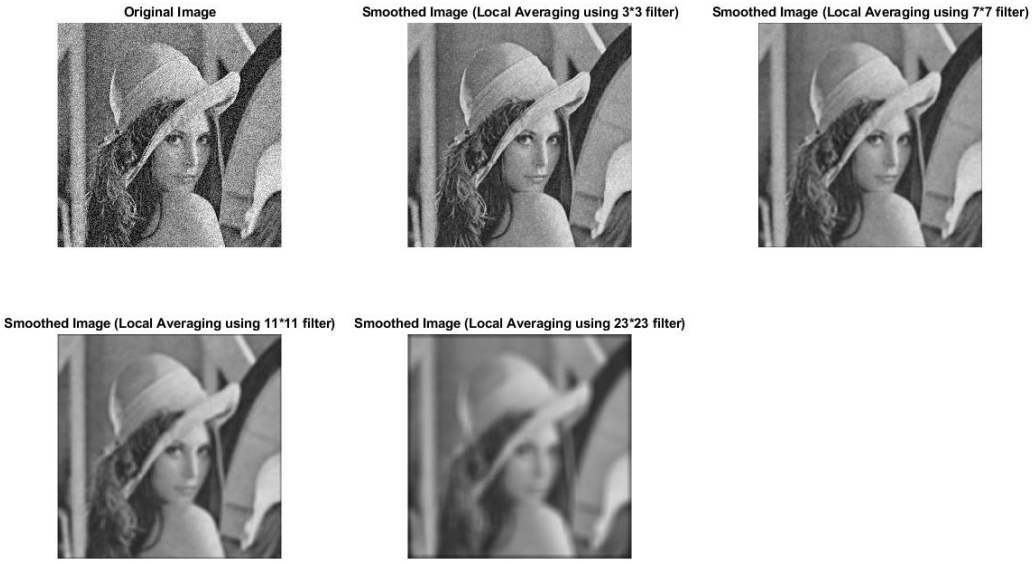
ابتدا یک آرایه ۳ در ۳ با مقادیر یکسان واحد تعریف شده و همگی درایه ها بر ۹ تقسیم می‌شوند. دلیل این امر این است که چون ما قرار است از میانگین واکسل های اطراف استفاده کنیم. ابتدا لازم است که درایه های موجود در کرنل نظیر به نظیر در تصویر مورد نظر ضرب شده سپس بر تعداد کل واکسلهای موجود در کرنل(برای این مثال ۹) تقسیم شود. لذا نمایش این کرنل به صورت ماتریسی به قرار زیر است:

$$\begin{bmatrix} ١ & ١ & ١ \\ ١ & ١ & ١ \\ ١ & ١ & ١ \end{bmatrix} * \frac{١}{٩}$$

نحوه محاسبه کرنل برای کرنل با ابعاد بزرگتر نیز مشابه مثال بالاست. سپس بعد از تعریف این کرنلها، با دستور conv۲ این کرنلها را به تصویر اصلی اعمال کردیم.

```
% Apply the filter using convolution
smooth_lenna_kernel3 = conv2(double(lenna_img), kernel_3, 'same');
smooth_lenna_kernel7 = conv2(double(lenna_img), kernel_7, 'same');
smooth_lenna_kernel11 = conv2(double(lenna_img), kernel_11, 'same');
smooth_lenna_kernel23 = conv2(double(lenna_img), kernel_23, 'same');

% Convert to uint8 format
smooth_lenna_kernel3 = uint8(smooth_lenna_kernel3);
smooth_lenna_kernel7 = uint8(smooth_lenna_kernel7);
smooth_lenna_kernel۱۱ = uint۸(smooth_lenna_kernel۱۱);
smooth_lenna_kernel۲۳ = uint۸(smooth_lenna_kernel۲۳);
```

در این بخش از کد، ابتدا فیلترهای میانگین‌گیری با اندازه‌های مختلف بر روی تصویر اعمال می‌شوند. این کار با استفاده از تابع conv۲ انجام می‌شود که کانولوشن دوبعدی را برای محاسبه مقدار جدید هر پیکسل بر اساس مقادیر همسایگان آن انجام می‌دهد. در اینجا، double(lenna_img) برای تبدیل تصویر از فرمت uint۸ به double استفاده شده است تا محاسبات دقیق‌تری انجام شود. آرگومان 'same' نیز باعث می‌شود که خروجی کانولوشن هم‌اندازه‌ی تصویر اصلی باشد. سپس، نتایج حاصل از اعمال فیلترهای ۳×۳، ۷×۷، ۱۱×۱۱ و ۲۳×۲۳ در متغیرهای smooth_lenna_kernel۳، smooth_lenna_kernel۷، smooth_lenna_kernel۱۱ و smooth_lenna_kernel۲۳ ذخیره می‌شوند. پس از انجام کانولوشن، خروجی‌ها که در فرمت double هستند، برای نمایش در متلب به uint۸ تبدیل می‌شوند تا تصویر پردازش‌شده قابل مشاهده باشد. در نهایت خروجی برای کرنلهای مختلف به همراه تصویر اصلی رسم شده است.

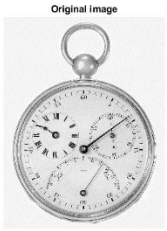


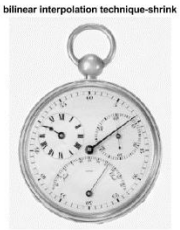
فیلتر میانگین‌گیری یک روش ساده و کارآمد برای هموارسازی تصویر است. انتخاب اندازه‌ی مناسب فیلتر از اهمیت بالایی برخوردار است—**فیلترهای کوچک** جزئیات تصویر را حفظ کرده و نویز کمی را کاهش می‌دهند، درحالی‌که **فیلترهای بزرگ‌تر** نویز بیشتری را حذف کرده ولی باعث کاهش وضوح و از بین رفتن جزئیات تصویر می‌شوند. نتایج این آزمایش نشان می‌دهد که افزایش اندازه‌ی فیلتر باعث افزایش هموارسازی و کاهش جزئیات تصویر می‌شود، بنابراین باید بسته به نیاز پردازش تصویر، اندازه‌ی فیلتر به‌درستی انتخاب شود. با افزایش اندازه‌ی فیلتر میانگین‌گیری، میزان تاری تصویر بیشتر می‌شود، زیرا تعداد بیشتری از پیکسل‌های همسایه در محاسبه‌ی مقدار جدید هر پیکسل نقش دارند و در نتیجه، تصویر صاف‌تر و تارتر به نظر می‌رسد. این موضوع باعث می‌شود که لبه‌ها و جزئیات تصویر کم‌رنگ‌تر و محوتر شوند. از سوی دیگر، افزایش اندازه‌ی فیلتر به کاهش نویز تصویر کمک می‌کند، زیرا مقدار هر پیکسل با میانگین مقادیر بیشتری از همسایگان جایگزین شده و نوسانات کوچک ناشی از نویز کاهش می‌یابد، اما اگر فیلتر بیش‌ازحد بزرگ باشد، نه‌تنها نویز بلکه بافت‌های مهم تصویر نیز از بین می‌روند. در نهایت، افزایش بیش‌ازحد اندازه‌ی فیلتر باعث کاهش وضوح و جزئیات تصویر می‌شود، به‌طوری‌که ساختارهای کوچک و اطلاعات دقیق تصویر ناپدید شده و تصویر به‌طور کلی نرم و محو به نظر می‌رسد. به‌عنوان مثال، در یک فیلتر ۲۳×۲۳، تصویر ممکن است تا حد زیادی صاف و بدون جزئیات شود، شبیه به یک نسخه‌ی کاملاً محو شده از تصویر اصلی. بنابراین در این سوال کرنل ۳ در ۳ بهترین انتخاب است.

سوال ۶

در این پروژه، سه تکنیک مختلف میان‌یابی (interpolation) برای تغییر اندازه یک تصویر مورد استفاده قرار گرفته‌اند**Nearest Neighbour Interpolation** ،. **Bilinear Interpolation**و .**Bicubic Interpolation** هدف این است که تفاوت‌های این روش‌ها در کاهش و افزایش اندازه تصویر بررسی شوند. برای این منظور، ابتدا یک تصویر ساعت به نام watch.tifبارگذاری می‌شود و سپس این تصویر با استفاده از هر یک از این روش‌ها کاهش (shrink) و سپس مجدداً بزرگ‌سازی (zoom) می‌شود. در نهایت، نتایج به صورت گرافیکی در دو شکل نمایش داده می‌شوند تا مقایسه‌ای از کارایی هر یک از روش‌ها ارائه دهند. بعد از بارگزاری تصویر با استفاده از کد زیر سه روش درونیابی را انجام می‌دهیم:

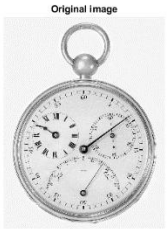
```
۱. %shrinking the watch image
۲. imageA_shr=imresize(watch_image,۱/۵,'nearest');
۳. imageB_shr=imresize(watch_image,۱/۵,'bilinear');
٤. imageC_shr=imresize(watch_image,۱/۵,'bicubic');
```

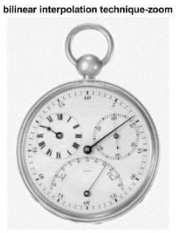




بدیهی است که ابعاد تصویر در هر روش به یک پنجم مقدار اولیه تبدیل می‌شود. برای اینکه تفاوت بین روشهای مختلف را مشـــاهده کنیم لازم اســـت تا ابعاد تصـــویر به اندازه مقدار اولیه بازگردند. برای این منظور از کد زیر استفاده شده است:

```
١. %zooming the watch image
٢. imageA_zoom=imresize(imageA_shr,٥,'nearest');
٣. imageB_zoom=imresize(imageB_shr,٥,'bilinear');
٤. imageC_zoom=imresize(imageC_shr,٥,'bicubic');
```





همانطور که در نتایج مشـــخص اســـت روش دوم و ســـوم نتایج واضـــح تری را ایجاد کرده اند. روش **Nearest Neighbour** ساده‌ترین و سریع‌ترین روش میان‌یابی است، اما هنگام بزرگ‌سازی تصویر معمولاً نتیجه به صورت پیکسـلی و خشـن به نظر می‌رسـد. این روش در کاهش اندازه به خوبی عمل می‌کند، اما در بزرگ‌سـازی کیفیت تصـویر به مراتب پایین‌تر اسـت. در مقابل، روش **Bilinear** کیفیت بهتری نسـبت به **Nearest Neighbour** ارائه می‌دهد. در کاهش اندازه، جزئیات بیشـــتری حفظ می‌شـــود و در بزرگ‌ســـازی نیز نســـبت به روش **Nearest Neighbour** نتیجه بهتری ایجاد می‌کند، اما هنوز تصـــویر به اندازه کافی نرم و صـــاف نیســـت. در نهایت، روش **Bicubic** دقیق‌ترین و بهترین روش میان‌یابی است که در کاهش اندازه تصویر، جزئیات دقیق‌تری حفظ می‌شود و در بزرگ‌ســـازی نیز تصـــویری نرم‌تر و با جزئیات بیشـــتر به نمایش گذاشـــته می‌شـــود. این روش معمولاً در کاربردهایی که نیاز به کیفیت بالا دارند، ترجیح داده می‌شود.

سوال ۷

بخش اول

هدف این تمرین اصلاح بیت ششم از هر پیکسل تصویر به صفر، مقایسه تصویر اصلاح‌شده با تصویر اصلی، نمایش تفاوت‌ها و محاسبه میزان اطلاعات از دست رفته در نتیجه‌ی این تغییر است. در ابتدا تصویر WashingtonDC.tif بارگزاری شده و تعداد بیتهای هر پیکسل آن بررسی می‌شود.

```
% Load the image
img = imread('Images/WashingtonDC.tif');
whos img
```

خروجی کد بالا شامل ابعاد تصویر و تعداد بیتهای هر پیکسل(۸ بیت) به صورت زیر است:

```
  Name         Size                Bytes  Class     Attributes

  img          ٥٦٤x٥٦٤             ٣١٨٠٩٦  uint٨
```

برای اینکه بیت ششم از هر پیکسل برابر صفر شود لازم است تا اطلاعات هر پیکسل در عبارت ۱۱۰۱۱۱۱۱ ضرب شود. عبارت بالا از تفریق عدد ۱۱۱۱۱۱۱۱ و ۰۰۱۰۰۰۰۰ حاصل می‌شود. برای ایجاد تصویری که هر پیکسل آن برابر با مقدار ۱۱۰۱۱۱۱۱ باشد از دستور (۳۲ - ۲۵۵)uint۸ استفاده شده است. در نهایت برای حذف بیت ششم در تمام تصویر لازم است تا این تصویر به صورت بیت به بیت با تصویر اصلی ضرب شود.

```
img_bits = uint٨(img); % Ensure the image is in uint٨ format (٠-٢٥٥)
img_modified = bitand(img_bits, uint٨(٢٥٥ - ٣٢));
```

در نهایت اختلاف تصویر اصلی با تصویری که بیت ششم آن حذف شده است از طریق زیر محاسبه شده است:

```
difference = abs(double(img) - double(img_modified));  % Compute absolute difference
```

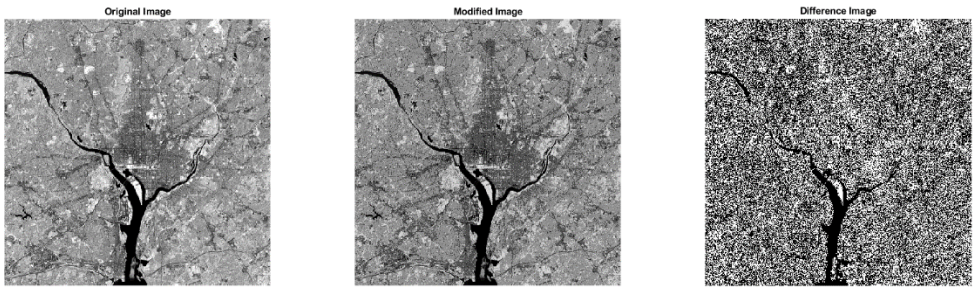
با توجه به اینکه بازه تغییرات شدت روشنایی بین ۰ تا ۲۵۵ باید باشد لذا از تابع abs استفاده شده است. در نهایت برای محاسبه میزان اختلاف ابتدا تمامی مقادیر اختلاف برای تمامی پیکسلها را جمع کرده و بر تعداد پیکسلها تقسیم شده است.

```
info_loss = sum(difference(:)) / numel(difference); % Sum of the pixel differences
```

مقدار اختلاف در زیر نمایش داده شده است:

```
Information lost (mean pixel 1. difference): 14.926211
```

این عدد نشان‌دهنده میانگین تفاوت پیکسل‌ها بین تصویر اصلی و تصویر اصلاح‌شده است. به عبارت دیگر، مقدار ۱۴.۹۳ به این معناست که پس از اصلاح بیت ششم در هر پیکسل، به‌طور متوسط ۱۴.۹۳ واحد از مقدار پیکسل‌ها کاهش یافته است.



مقدار **اطلاعات از دست رفته** به‌عنوان تفاوت میانگین پیکسل‌ها (Mean Pixel Difference) محاسبه شده است. این عدد نشان می‌دهد که پس از اصلاح بیت ششم (که مقدار آن ۳۲ است)، تغییرات قابل توجهی در مقادیر پیکسل‌ها رخ داده است، اما این تغییرات به اندازه‌ای نیستند که تصویر به‌طور کلی از بین برود یا به شدت تغییر کند.در واقع، اطلاعات از دست رفته ناشی از صفر کردن تنها یک بیت از پیکسل‌ها (بیت ششم) مقدار قابل توجهی است، اما هنوز تصویر اصلی و اصلاح‌شده شباهت‌های زیادی دارند. البته هر چه شماره بیت بیشتر شود(مثلا بیت هفتم یا هشتم) اطلاعات غنی تر و حذف آنها باعث تغییرات بیشتری می‌شود.

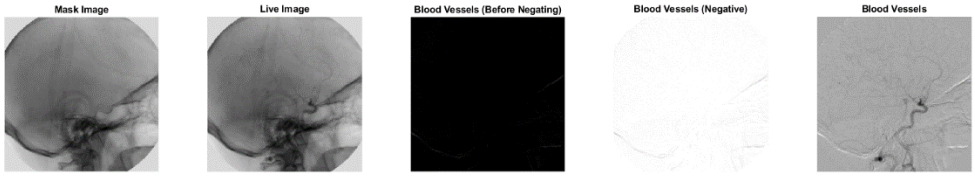
بخش دوم

در این بخش، از تصاویر رادیوگرافی (X-ray) استفاده می‌کنیم که شامل یک تصویر **ماسک** (تصویر قبل از تزریق ید در جریان خون) و یک تصویر **زنده** (تصویر پس از تزریق) است. هدف این است که با استفاده از **تفاضل** این دو تصویر، **رگ‌های خونی** که در تصویر زنده ظاهر شده‌اند، برجسته شوند. برای تقویت نمایش و تجزیه و تحلیل بهتر، تصویر تفاضل شده به شکل منفی نیز نمایش داده می‌شود تا این رگ‌ها واضح‌تر دیده شوند.

بتدا تصاویر مورد نظر با استفاده از دستور imreadدر MATLAB بارگذاری می‌شوند. تصویر اول، **تصویر ماسک** است که نمایی از سر بیمار قبل از تزریق ید را نشان می‌دهد و تصویر دوم، **تصویر زنده** است که نمایی از سر بیمار پس از تزریق ید به بدن است. این تصاویر به صورت آرایه‌های ماتریسی در MATLAB ذخیره می‌شوند. برای برجسته‌سازی رگ‌های خونی، باید تفاوت پیکسل‌های تصویر **زنده** و **ماسک** محاسبه شود. به این منظور، هر دو تصویر به نوع داده doubleتبدیل می‌شوند تا امکان انجام عملیات ریاضی روی مقادیر پیکسل‌ها فراهم شود. پس از تبدیل به double، عملیات تفاضل بین دو تصویر انجام می‌شود. نتیجه این عملیات، تصویر جدیدی است که تفاوت‌های بین تصویر زنده و تصویر ماسک را نشان می‌دهد و به طور معمول رگ‌های خونی در این تفاضل برجسته می‌شوند. برای تقویت دید رگ‌های خونی و افزایش وضوح آنها، تصویر تفاضل شده به صورت منفی نمایش داده می‌شود. این کار با استفاده از دستور ۲۵۵ - uint۸(blood_vessels) انجام می‌شود که باعث معکوس شدن رنگ‌ها و واضح‌تر شدن رگ‌ها در تصویر می‌شود.

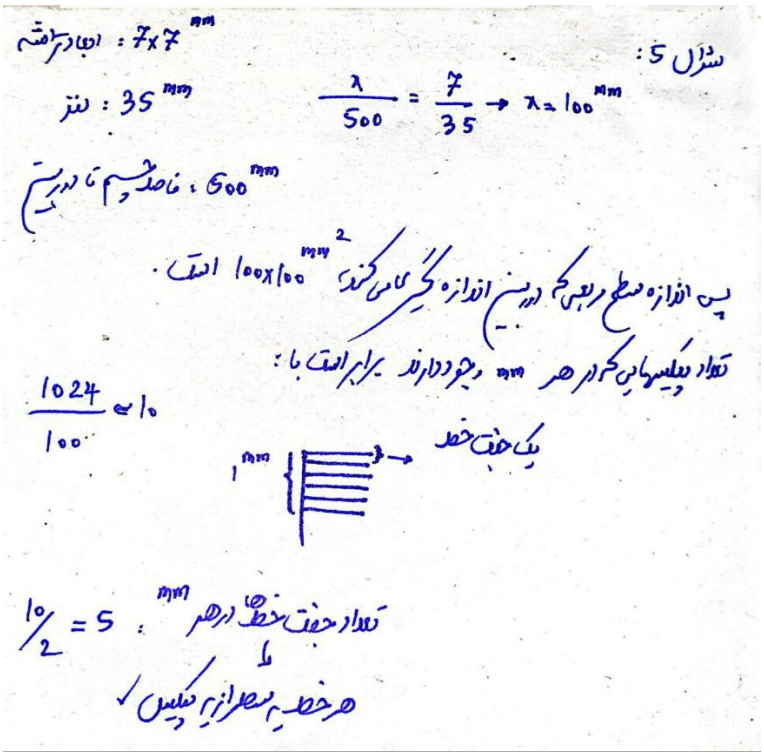
```
١. mask_image = imread('Images/angiography_mask.tif');
٢. live_image = imread('Images/angiography_live.tif');
٣.
٤. % Step ٢: Perform subtraction (highlight the blood vessels)
٥. blood_vessels = double(live_image) - double(mask_image);  % Convert to double for
subtraction
٦.
٧. % Step ٣: Enhance visualization by displaying the negative of the result
٨. blood_vessels_negative = ٢٥٥ - uint٨(blood_vessels);  % Make sure it's in the ٠-٢٥٥
range
```

در نهایت، تصاویر مختلف در یک پنجره گرافیکی با استفاده از دستور subplotبه نمایش در می‌آید. در این پنجره، ابتدا تصویر ماسک و سپس تصویر زنده نمایش داده می‌شود. پس از آن، تصویر تفاضل شده به نمایش درمی‌آید، که رگ‌های خونی را بدون اعمال معکوس نشان می‌دهد. و در نهایت، تصویر تفاضل شده پس از اعمال معکوس (منفی) نمایش داده می‌شود که کمک می‌کند رگ‌های خونی بهتر دیده شوند .

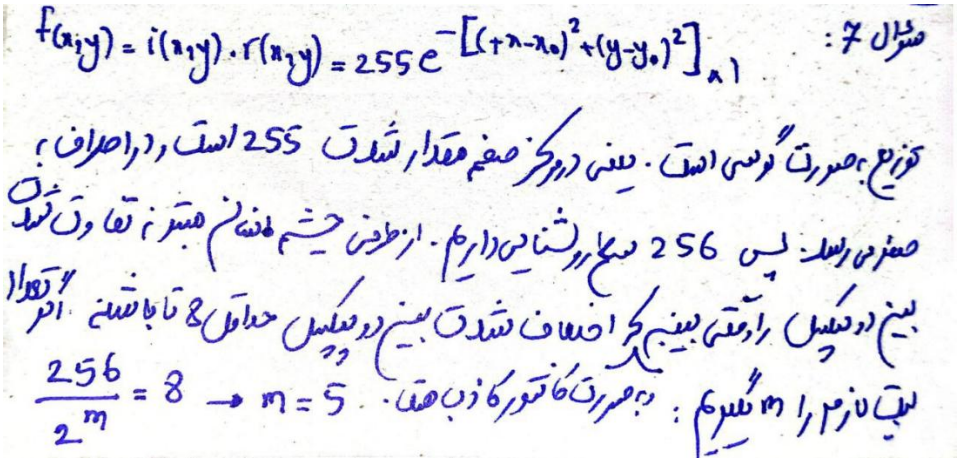


با استفاده از این روش، تفاضل پیکسل به پیکسل بین تصویر **ماسک** و **تصویر زنده** انجام شد و رگ‌های خونی که در تصویر زنده به دلیل تزریق ید به جریان خون ظاهر شده‌اند، برجسته گردیدند. با استفاده از تصویر منفی، وضوح این رگ‌ها بیشتر شد و آنها به وضوح در برابر پس‌زمینه روشن‌تر و قابل مشاهده‌تر شدند.

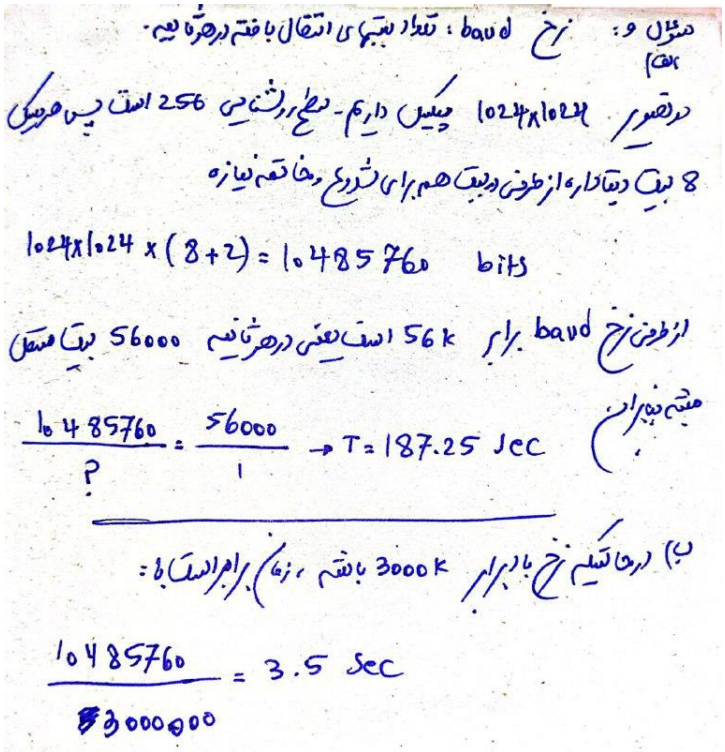
سوال ۵ تشریحی



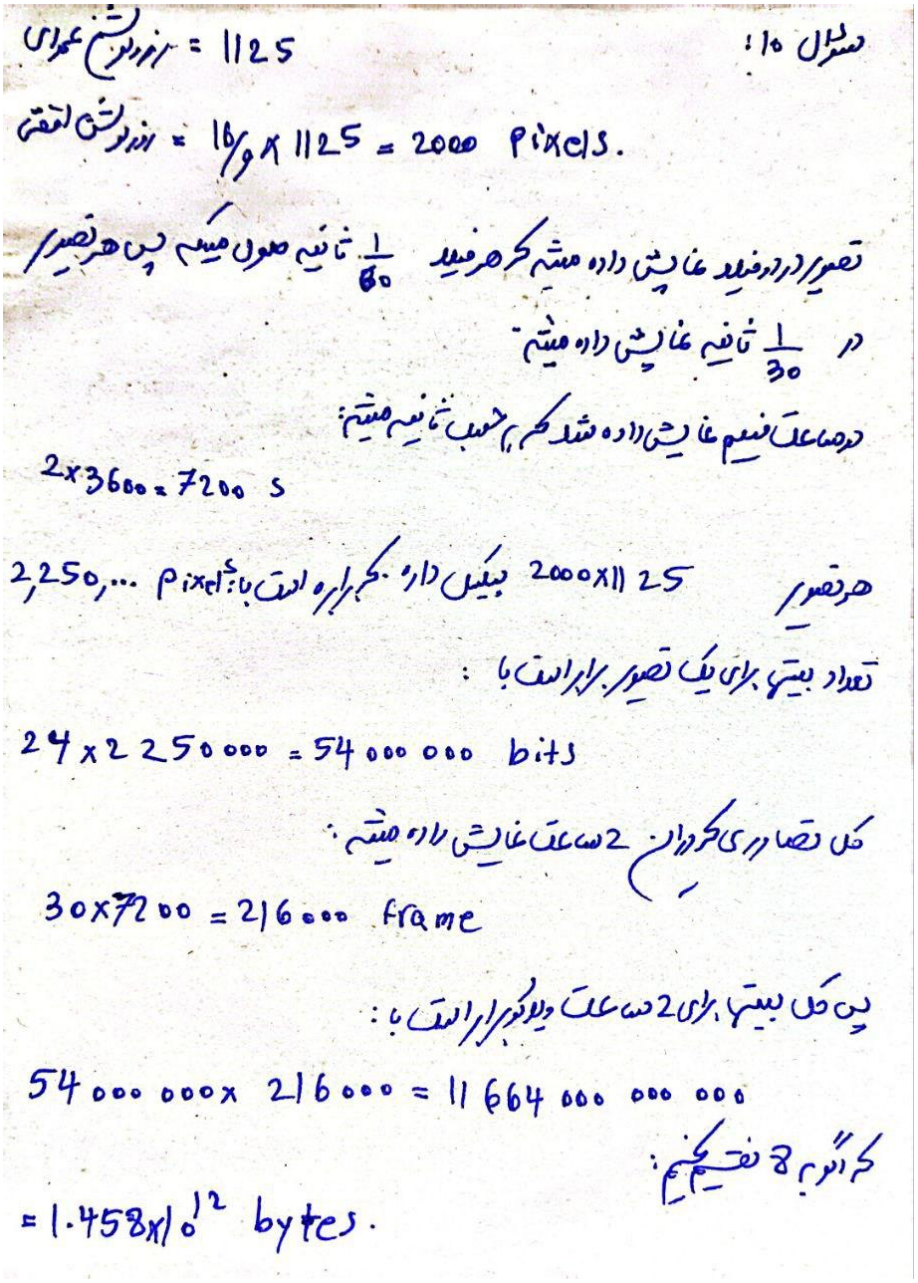
سوال ۷


سوال 7 :
$f(x,y) = i(x,y) \cdot r(x,y) = 255 e^{-[(x-x_0)^2+(y-y_0)^2]}$
توزیع به صورت گوسی است . یعنی در مرکز عضو مقدار شدت 255 است و در اطراف
عضو می رسد . پس 256 سطح روشنایی داریم . از طرفی چشم انسان میتواند تفاوت شدت
بین دو پیکسل را وقتی ببیند که اختلاف شدت بین دو پیکسل حداقل 8 تا باشد . اگر تعداد
$\frac{256}{2^m} = 8 \rightarrow m = 5$: به صورت کانتور کاذب هست .

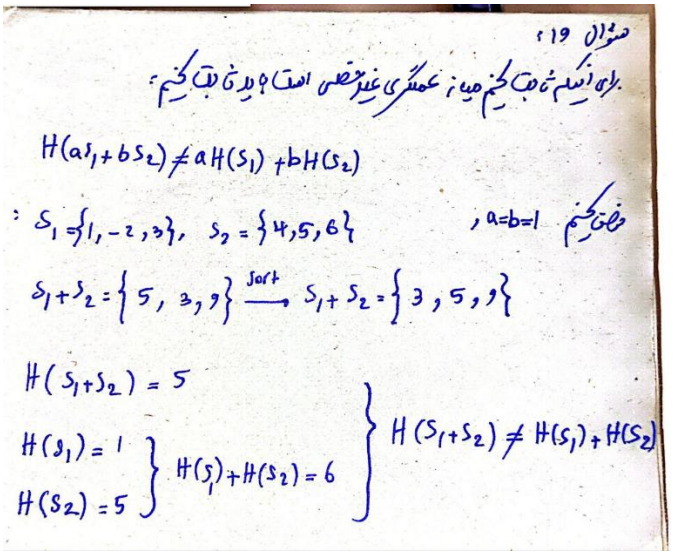
سوال ۹


سوال ۹ : نرخ baud : تعداد بیت های انتقال یافته در هر ثانیه .
در تصویر 1024x1024 پیکسل داریم . سطح روشنایی 256 است پس هر پیکسل
8 بیت دیتا دارد . از طرفی دو بیت هم برای شروع و خاتمه نیاز است .
$1024 \times 1024 \times (8+2) = 10485760$ bits
از طرفی نرخ baud برابر 56k است یعنی در هر ثانیه 56000 بیت منتقل
میشود بنابراین
$\frac{10485760}{?} = \frac{56000}{1} \rightarrow T = 187.25$ sec
ب) در حالتی که نرخ باد برابر 3000k باشد ، زمان برابر است با :
$\frac{10485760}{3000000} = 3.5$ sec

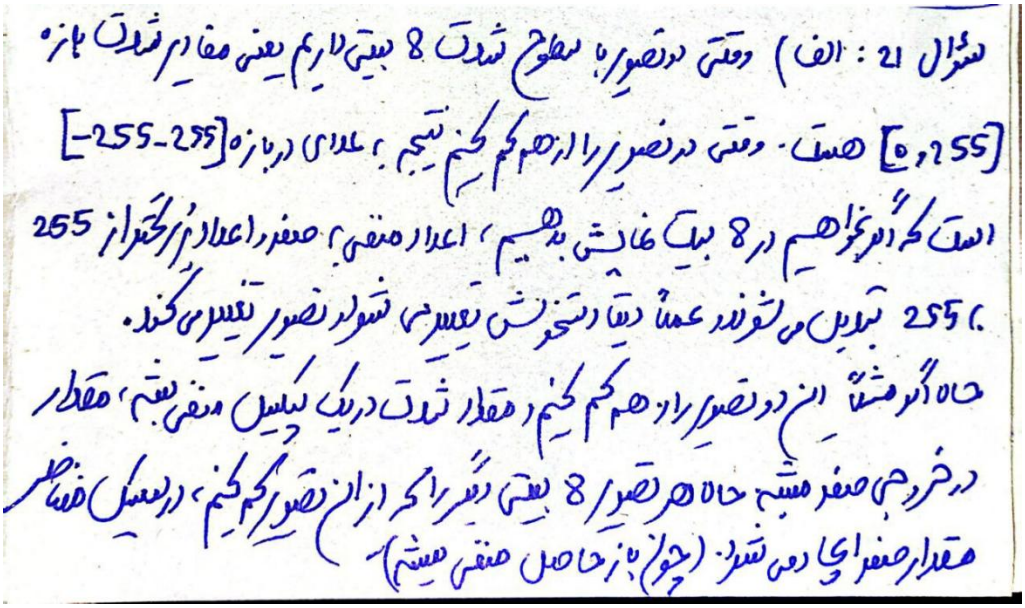
سوال ۱۰


سوال ۱۰:
تعداد خطوط عمودی = 1125
تعداد پیکسل در هر خط = 16/9 × 1125 = 2000 pixels.
تصویر در هر ثانیه 30 بار نمایش داده میشه که هر فریم $\frac{1}{30}$ ثانیه طول میکشه پس هر تصویر در $\frac{1}{30}$ ثانیه نمایش داده میشه.
دو ساعت فیلم نمایش داده شده که بر حسب ثانیه میشه:
2x3600 = 7200 s
هر تصویر 2000x1125 پیکسل داره که در کل 2,250,... pixel با:
تعداد بیت برای یک تصویر برابر است با:
24 x 2250000 = 54 000 000 bits
کل تصاویری که در این 2 ساعت نمایش داده میشه:
30x7200 = 216000 frame
پس کل بیت برای 2 ساعت فیلم برابر است با:
54 000 000x 216000 = 11 664 000 000 000
که اگر بر 8 تقسیم کنیم:
= 1.458x10^12 bytes.

سوال ۱۹



سوال ۲۱



سوال 21) ب: همانطور که در ترتیب تصاویر نیز مشخص است. فرض کنید

تعداد سلول a برابر 128 با تعداد سلول b برابر صفر

$a - b = 128$ → تعداد روشنایی برابر 128 میشه.

$b - a = -128$ → تعداد روشنایی در 8 بیت با صفر تبدیل میشه

سوال ۲۳