|  |  |  |  |
| --- | --- | --- | --- |
| Optimized_image_813b5ec2.png | | **دانشگاه تهران**  **پردیس دانشکده­های فنی**  **دانشکده مهندسی برق و کامپیوتر** | Images__Logo_FE.gif |
|  | | | |
|  | | | |
| پردازش تصاویر دیجیتال  **گزارش تمرین سری اول** | | | |
| **دانشجو**  سید محمد جواد موسوی | | | |
| **استاد کلاس**  دکتر حمید سلطانیان­زاده | | | |

# سوال 1

## 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);

10. title('Lenna rgb image');

بخش دوم

برای تبدیل تصویر رنگی به تصویر خاکستری، از تابع rgb2gray 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 255 (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 (Red) → 0.2989
* سبز (Green) → 0.5870
* آبی (Blue) → 0.1140

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:

Gray Value=0.2989×R+0.5870×G+0.1140×B

در MATLAB، برای تبدیل تصویر RGB به خاکستری از تابع rgb2gray 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.

1. %convert the image to grayscale

2. gray\_lenna\_rgb = rgb2gray(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;

10. title('Grayscale lenna image');

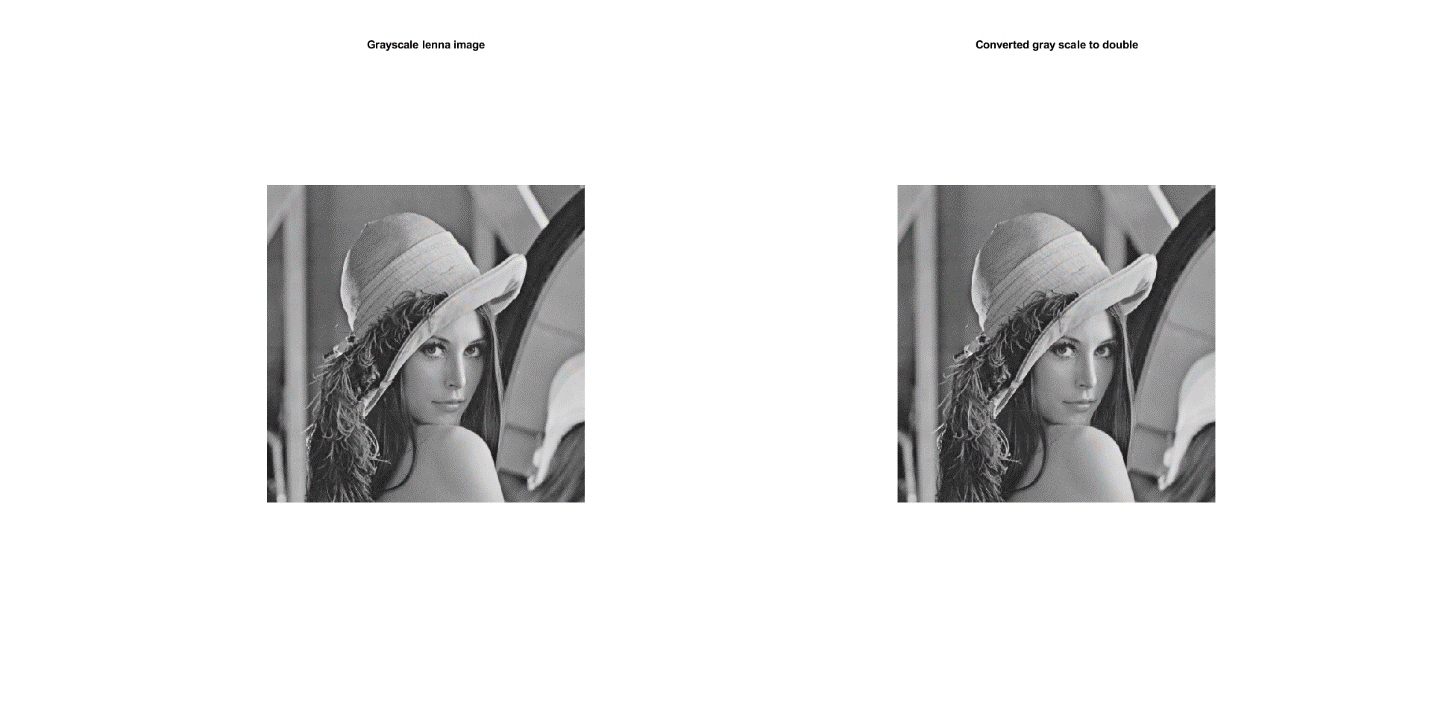




بخش چهارم

کد double\_lenna\_rgb = im2double(gray\_lenna\_rgb); تصویر خاکستری gray\_lenna\_rgb را به نوع داده double تبدیل می‌کند. در ابتدا باید بدانیم که تصاویر در MATLAB معمولاً به صورت ماتریس‌هایی از اعداد در نوع داده‌های مختلف مانند uint8 (8 بیتی بدون علامت) ذخیره می‌شوند. در نوع داده uint8، مقادیر پیکسل‌ها در بازه [0, 255] قرار دارند، که 0 نمایانگر رنگ سیاه و 255 نمایانگر سفید است. اما با استفاده از تابع im2double، مقادیر پیکسل‌ها به نوع داده double تبدیل می‌شوند و مقادیر پیکسل‌ها به بازه [0, 1] تغییر می‌کنند. به عبارت دیگر، مقدار هر پیکسل به 255 تقسیم می‌شود تا به بازه [0, 1] برسد. این تبدیل دقت بیشتری را در پردازش‌های ریاضیاتی فراهم می‌آورد، زیرا نوع داده double اجازه می‌دهد تا عملیات‌های پیچیده‌تری مانند ضرب ماتریس یا محاسبات دقیق‌تر انجام شوند. همچنین، این تبدیل محدودیت‌های عددی نوع uint8 را 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 double تبدیل شوند تا به درستی روی آن‌ها اعمال شوند. در نتیجه، تبدیل تصویر خاکستری به double باعث می‌شود که پردازش‌های تصویر به دقت و کارایی بیشتری انجام شوند.

وقتی مقدار روشنایی به doubleتبدیل می­شود میتوان محاسبات را با دقت اعشار تا 15 رقم انجام داد که به نسبت حالت uint8 که مقادیر کی از اعداد صفر تا 255 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);

10. imshow(double\_lenna\_rgb);

11. axis equal;

12. 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.

1. Original Grayscale (uint8): Min = 25, Max = 245

2. Converted Grayscale (double): Min = 0.10, Max = 0.96

3. 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 imresize در MATLAB انجام می‌شود. در این بخش از پروژه، تصویر خاکستری gray\_lenna\_rgb که در بخش‌های قبلی تبدیل شده است، در مقیاس‌های مختلف تغییر اندازه داده می‌شود و نتایج برای تحلیل به نمایش درمی‌آید.

1. %% Part f:

2. scale\_factor = [5, 1/2, 1/4];

3.

4. figure;

5. subplot(2, 2, 1);

6. imshow(gray\_lenna\_rgb);

7. axis equal;

8. title('Origin Grayscale Image');

9.

10. for i = 1:length(scale\_factor)

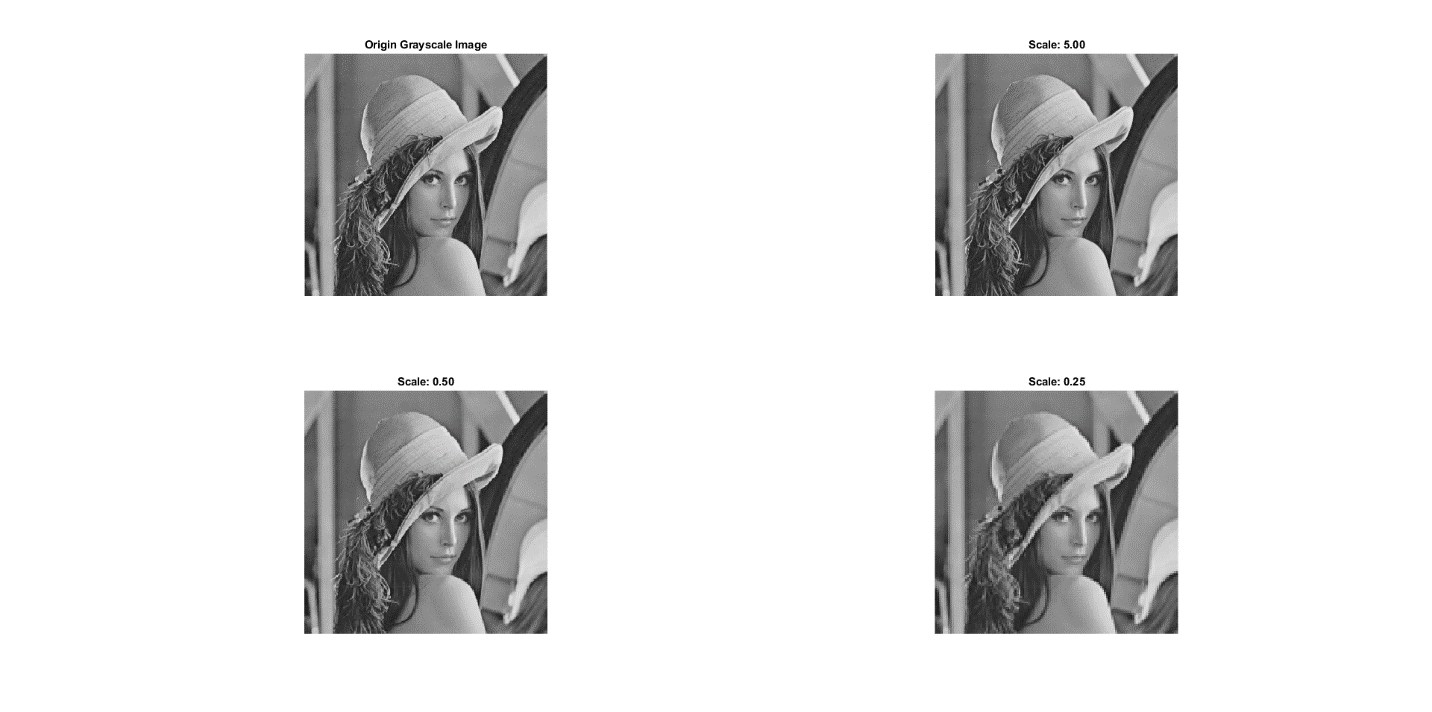
11. resized\_image = imresize(gray\_lenna\_rgb, scale\_factor(i));

12. subplot(2, 2, i+1);

13. imshow(resized\_image);

14. title(sprintf('Scale: %.2f', scale\_factor(i)));

15. 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 5، ابعاد تصویر 5 برابر شده و با توجه بهروش درونیابی مورد استفاده تصویر وضوح بهتری نسبت به تصویر اصلی دارد. البته که تغییر ابعاد باعث ایجاد تعداد پیکسلهای جدید و در نتیجه نیاز به حافظه بیشتر می­شود . زمانی که تصویر به نصف اندازه اصلی 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 بیشتری 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\_number تعریف می‌شود که شماره خطی از تصویر را نشان می‌دهد که قرار است بررسی شود. در این مثال، شماره خط 150 است. با استفاده از این متغیر، یک خط افقی از تصویر gray\_lenna\_rgb انتخاب می‌شود. این کار با دستور scan\_line = gray\_lenna\_rgb(line\_number, :); انجام می‌شود که همه پیکسل‌های موجود در ردیف 150 (به‌صورت افقی) را از تصویر استخراج می‌کند. سپس با استفاده از تابع plot(scan\_line)، مقادیر شدت روشنایی پیکسل‌ها در خط 150 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, ylabel برای 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.

1. line\_number = 150;

2.

3. scan\_line = gray\_lenna\_rgb(line\_number, :);

4.

5. figure;

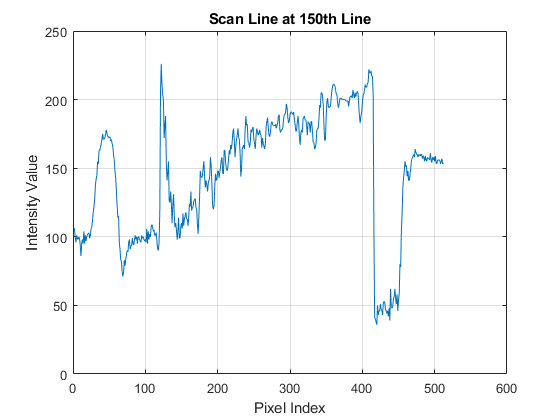
6. plot(scan\_line);

7. title('Scan Line at 150th Line');

8. xlabel('Pixel Index');

9. ylabel('Intensity Value');

10. grid on;



سوال 2

در این سوال قرار است تا سطوح شدت روشنایی تصویر skull.tif را به سطوح 64 و 16 و 4 و 2 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 64 و 16 و 4 و 2 را با تابع reduce\_level انجام می­دهیم. برای کاهش سطح در از رابطه زیر استفاده می­کنیم:

در رابطه با عبارت بالا ذکر چند نکته الزامی است:

* شدت روشنایی تصویر اولیه به صورت 8 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 255 باشد برای اینکه سطح روشنایی را به 4 سطح کاهش دهیم لازم است تا بازه بین صفر تا 255 به 4 قسمت تقسیم شود:

بازه (0 تا 63) را باید به شدت روشنایی صفر نسبت دهیم. بازه (64 تا 127) را باید به شدت روشنایی 85 نسبت دهیم. بازه (128 تا 191) را به شدت روشنایی 170 نسبت دهیم و بازه (192 تا 255) را نیز به شدت روشنایی 255 نسبت دهیم.

بنابراین برای رابطه بالا از تابع زیر استفاده شده است:

1. % Define a function to reduce intensity levels

2. reduce\_level = @(img, levels) uint8(floor(double(img) / (256 / levels)) \* (256 / (levels - 1)));

دلیل استفاده از عبارت uint8 این است که این تابع برای مقادیر شدت روشنایی کمتر از صفر مقدار صفر و برای مقادیر شدت روشنایی بزرگتر از 255 مقدار 255 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

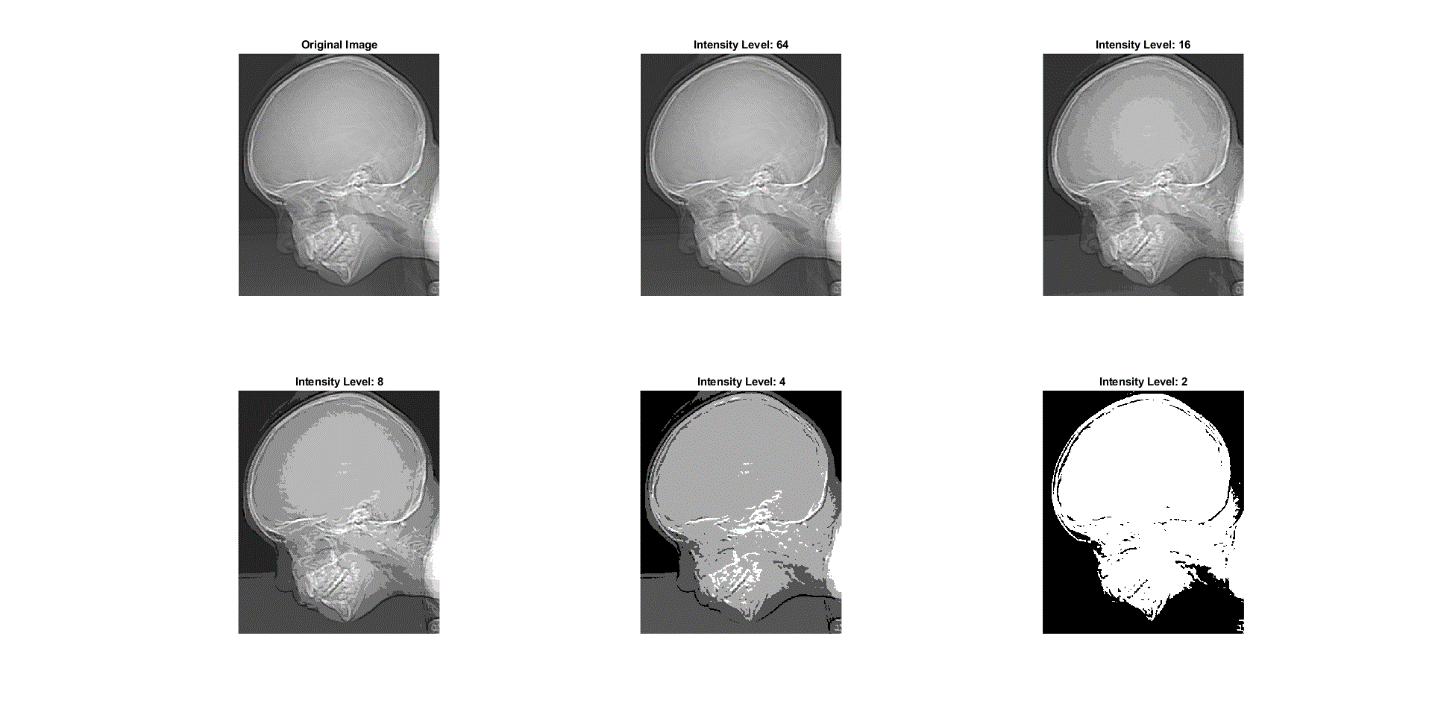
8. subplot(2, 3, i + 1);

9. imshow(img\_reduced);

10. title(sprintf('Intensity Level: %d', levels(i)));

11. end

خروجی در شکل زیر آمده است:



با توجه به نتایج تصاویری با سطوح 256 و 64 از نظر بینایی برای اهداف عملی یکسان هستند و تفاوت قابل ملاحظه­ای دیده نمی­شود. در سطوح 16 و 8 و 4 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 3

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:

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

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:

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

1. sx = 0.7;

2. sy = 1.2;

3.

4. % Define scaling matrix

5. T\_scale = [sx 0 0; 0 sy 0; 0 0 1];

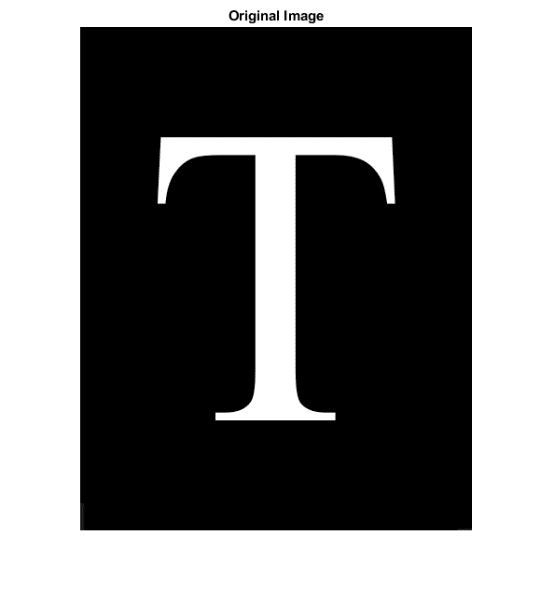
تابع**affine2d** در متلب برای ایجاد یک آبجکت تبدیل آفین (**Affine Transformation Object**) استفاده می‌شود. این تابع یک ماتریس تبدیل آفین که معمولاً یک ماتریس ۳x۳ است را می‌گیرد و آن را به یک آبجکت تبدیل تبدیل می‌کند که می‌توان از آن در توابع دیگر متلب، مانند **imwarp**، برای اعمال تبدیل‌های هندسی استفاده 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**affine2d** این امکان را می‌دهد که یک ماتریس آفین را به یک آبجکت تبدیل کنید که سپس می‌تواند 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:

1. tform\_scale = affine2d(T\_scale);

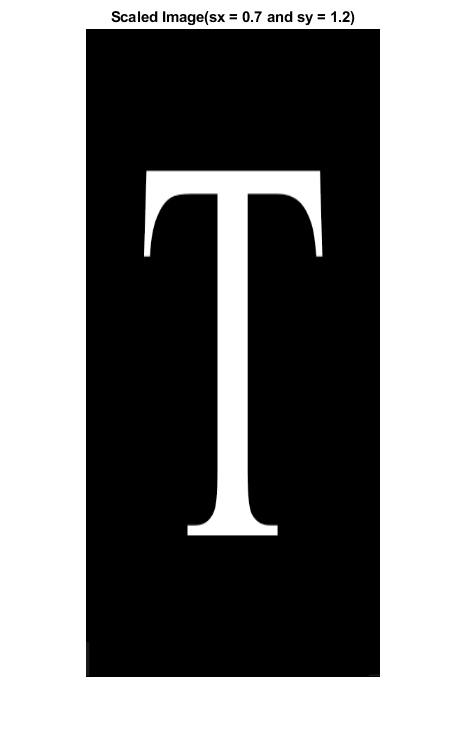
تابع**imwarp** در متلب برای اعمال تبدیل‌های هندسی بر روی تصاویر استفاده می‌شود و این امکان را می‌دهد که از یک آبجکت تبدیل آفین (که توسط تابع**affine2d** ایجاد می‌شود) برای تغییر شکل تصویر استفاده کنید. نحوه عملکرد آن به این صورت است که ابتدا ماتریس تبدیل (که معمولاً توسط تابع**affine2d** ایجاد می‌شود) 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:

1. 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:

ابتدا مقدار چرخش در 30 دره تنظیم می­شود. سپس آنرا با دستور deg2rad به رادیان تبدیل می­کنیم.

1. theta = 30; % Rotation angle in degrees

2. theta\_rad = deg2rad(theta); % Convert to radians

ماتریس تبدیل به صورت زیر تنظیم شده است:

1. % Define the rotation matrix

2. T\_rotate = [cos(theta\_rad), -sin(theta\_rad), 0;

3. sin(theta\_rad), cos(theta\_rad), 0;

4. 0, 0, 1];

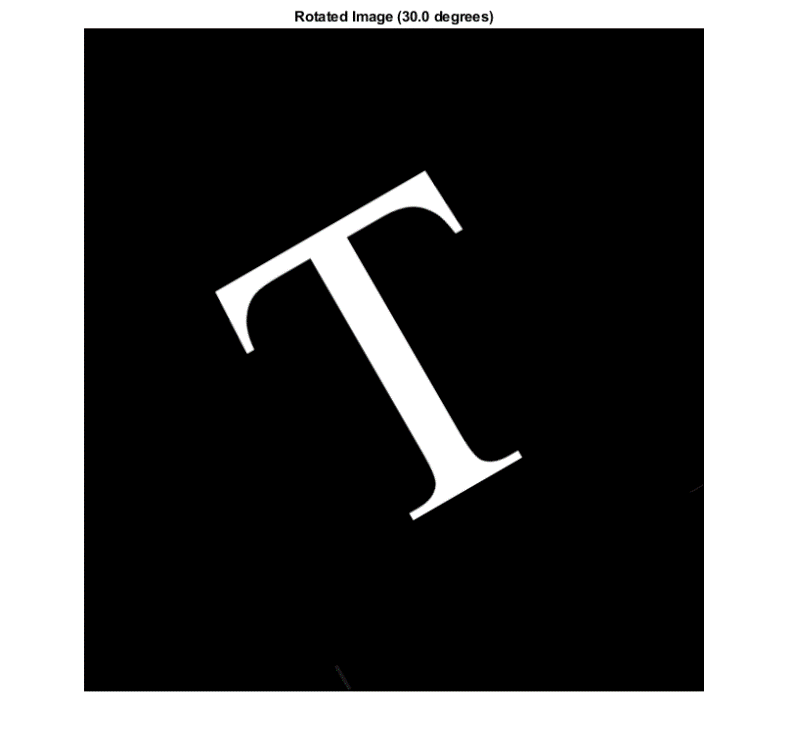
در نهایت مشابه با بخش قبل تصویر خروجی ساخته شده است:

1. % Apply rotation using imwarp

2. tform\_rotate = affine2d(T\_rotate);

3. 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:

1. function translatedImage = translationFunction(inputImage, tx, ty)

2. % Get the dimensions of the input image

3. [rows, cols, channels] = size(inputImage);

4.

5. % Create a new image with the same dimensions as the input image

6. translatedImage = zeros(rows, cols, channels, 'like', inputImage);

7.

8. % Loop through each pixel of the input image

9. for m = 1:rows

10. for n = 1:cols

11. x = m + tx;

12. y = n + ty;

13.

14. % Ensure the new coordinates are within the image bounds

15. if x > 0 && x <= rows && y > 0 && y <= cols

16. translatedImage(x, y, :) = inputImage(m, n, :);

17. end

18. end

19. end

20. 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:

1. %% Translation Transformation

2. tx = 50;

3. ty = 50;

4.

5. translatedImage = translationFunction(T\_img, tx, ty);

6. % Display the translated image

7. figure;

8. imshow(translatedImage);

9. 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:

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

1. % Define the shear factors

2. Shx = 0.3; % Shear in X direction

3. Shy = 0.2; % Shear in Y direction

4.

5. % Define the shear matrix

6. T\_shear = [1 Shx 0; Shy 1 0; 0 0 1];

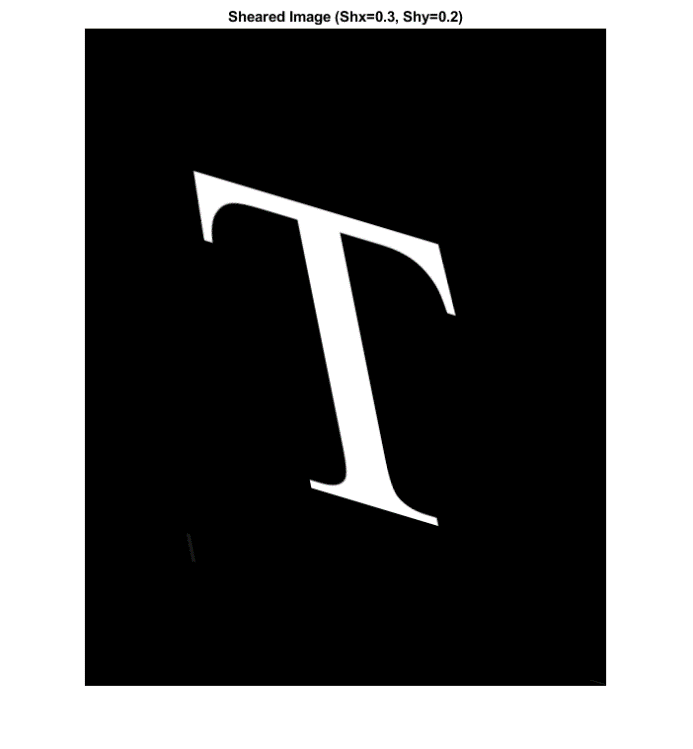
سپس مشابه بخش­های قبل ماتریس آفین به صورت زیر تنظیم شده است:

1. % Apply shear using imwarp

2. tform\_shear = affine2d(T\_shear);

3. sheared\_img = imwarp(T\_img, tform\_shear);

نتیجه در شکل زیر نمایش داده شده است:



سوال 4

عملیات منطقی در پردازش تصویر و محاسبات دیجیتال، به کارگیری **عملیات‌های پایه‌ای منطقی** برای تحلیل داده‌ها است. در این عملیات‌ها، معمولاً از مقادیر **باینری (0 و 1)** استفاده می‌شود. در اینجا، به معرفی و توضیح برخی از این عملیات‌های منطقی پرداخته می‌شود که در کد استفاده کرده‌ایم:

1. **عملیاتNOT (نقیض)**

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

|  |  |
| --- | --- |
| NOT(A) | A |
| 1 | 0 |
| 0 | 1 |

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

1. % Function to perform NOT operation (voxel by voxel)

2. function result = logical\_not(A)

3. [rows, cols] = size(A);

4. result = zeros(rows, cols); % Initialize result matrix with zeros

5.

6. for x = 1:rows

7. for y = 1:cols

8. if A(x, y) == 1

9. result(x, y) = 0; % If A is 1, set to 0

10. else

11. result(x, y) = 1; % If A is 0, set to 1

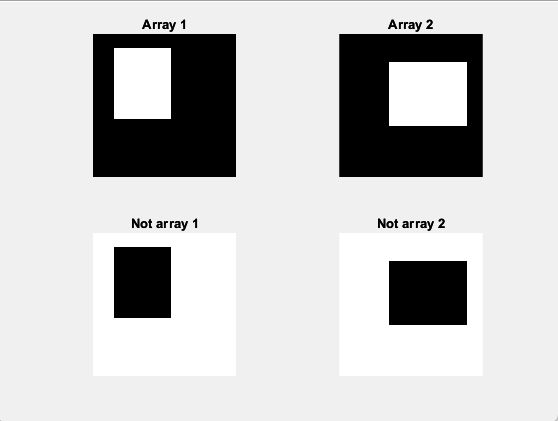
12. end

13. end

14. end

15. end

تابعlogical\_not که عملیات نقیض (NOT) را به صورت دستی روی یک ماتریس باینری انجام می‌دهد، ابتدا یک ماتریس ورودی A می‌گیرد که حاوی مقادیر 0 و 1 است. 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 1 باشد، در ماتریس result مقدار آن به 0 تغییر می‌کند و اگر مقدار پیکسل برابر 0 باشد، مقدار آن در ماتریس result به 1 تغییر می‌یابد. در نهایت، پس از انجام این عملیات روی تمام پیکسل‌ها، ماتریس result که 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 MATLAB ندارد.



1. عملیات AND

عملیات **AND** بر اساس این قانون عمل می‌کند: اگر هر دو ورودی برابر با 1 باشند، نتیجه 1 خواهد بود؛ در غیر این صورت، نتیجه 0 خواهد بود.

|  |  |  |
| --- | --- | --- |
| A and B | B | A |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 1 |
| 1 | 1 | 1 |

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

1. % AND operation

2. function result = logical\_and(A, B)

3. [rows, cols] = size(A);

4. result = zeros(rows, cols);

5. for x = 1:rows

6. for y = 1:cols

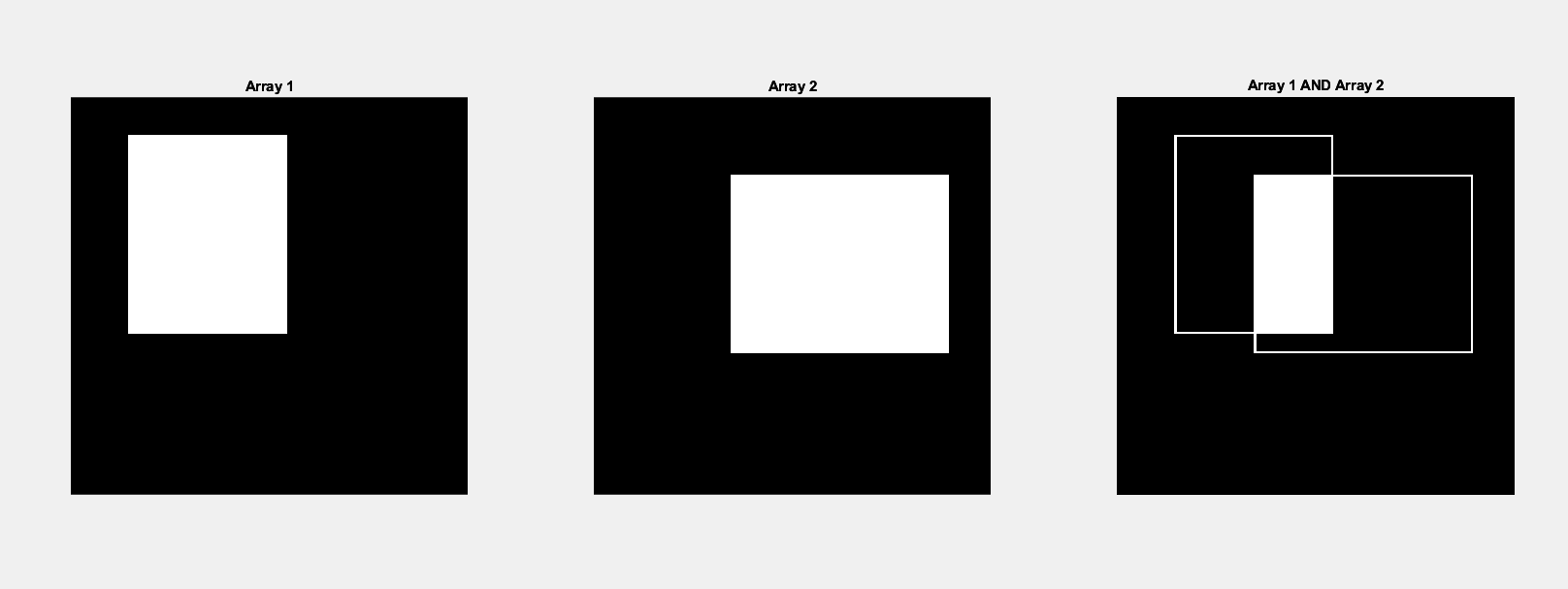
7. result(x, y) = A(x, y) \* B(x, y);

8. end

9. end

10. end

تابع logical\_and برای انجام عملیات منطقی AND بر روی دو ماتریس باینری طراحی شده است. این تابع دو ورودی A و B می‌گیرد که هرکدام از آن‌ها ماتریس‌هایی از مقادیر 0 و 1 هستند. ابتدا ابعاد هر دو ماتریس با استفاده از دستور [rows, cols] = size(A) محاسبه می‌شود، که در آن rows تعداد ردیف‌ها و cols تعداد 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 = 1:rows و for y = 1:cols) ) برای پیمایش هر عنصر از ماتریس‌ها A و B نوشته 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 A و B انجام می‌شود. این به این معناست که اگر هر دو پیکسل برابر 1 باشند، نتیجه برابر 1 خواهد شد و در غیر این صورت نتیجه 0 خواهد بود. در نهایت، بعد از تکمیل این عملیات برای تمام پیکسل‌ها، ماتریس result که شامل نتایج نهایی عملیات AND است، به عنوان خروجی تابع برمی‌گردد.



1. عملیات OR

عملیات **OR** بر اساس این قانون عمل می‌کند: اگر هر یک از ورودی‌ها 1 باشند، نتیجه 1 خواهد بود. در غیر این صورت، نتیجه 0 خواهد بود.

|  |  |  |
| --- | --- | --- |
| A OR B | B | A |
| 0 | 0 | 0 |
| 1 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

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:

1. % OR operation

2. function result = logical\_or(A, B)

3. [rows, cols] = size(A);

4. result = zeros(rows, cols);

5. for x = 1:rows

6. for y = 1:cols

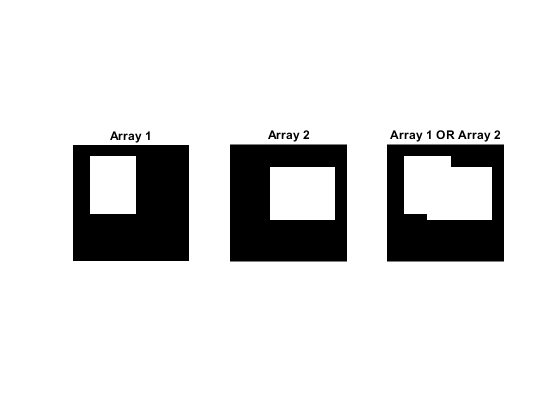
7. result(x, y) = A(x, y) + B(x, y);

8. end

9. end

10. end

تابع logical\_or برای انجام عملیات منطقی OR بر روی دو ماتریس باینری طراحی شده است. این تابع دو ورودی A و B می‌گیرد که هرکدام از آن‌ها ماتریس‌هایی از مقادیر 0 و 1 هستند. ابتدا ابعاد هر دو ماتریس با استفاده از دستور [rows, cols] = size(A) محاسبه می‌شود، که در آن rows تعداد ردیف‌ها و cols تعداد 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 = 1:rows و for y = 1:cols) ) برای پیمایش هر عنصر از ماتریس‌ها A و B نوشته 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 1 باشند، نتیجه برابر 1 خواهد بود (چرا که در جمع مقادیر 1 + 0 یا 0 + 1 برابر 1 و در 1 + 1 نیز نتیجه 1 خواهد بود). تنها زمانی که هر دو پیکسل برابر 0 باشند، نتیجه جمع برابر 0 خواهد شد. در نهایت، بعد از تکمیل این عملیات برای تمام پیکسل‌ها، ماتریس result که شامل نتایج نهایی عملیات OR است، به عنوان خروجی تابع برمی‌گردد.



1. 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 AND-NOT B | B | A |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |

برای اجرای این عملگر از کد زیر استفاده شده است:

1. % AND-NOT operation

2. function result = logical\_and\_not(A, B)

3. [rows, cols] = size(A);

4. result = zeros(rows, cols);

5. for x = 1:rows

6. for y = 1:cols

7. result(x, y) = A(x, y) \* ~B(x, y);

8. end

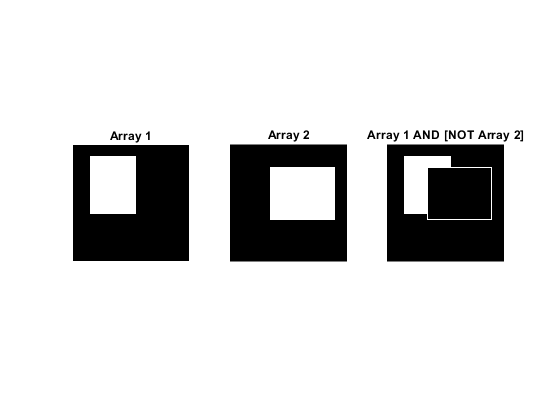
9. end

10. end

تابع logical\_and\_not به منظور انجام عملیات منطقی AND-NOT بین دو ماتریس باینری طراحی شده است. ورودی‌های این تابع دو ماتریس A و B هستند که هرکدام مقادیر 0 و 1 را در خود دارند. هدف از این تابع انجام عملیات منطقی AND بین ماتریس A و معکوس ماتریس B (که همان NOT ماتریس B است) می‌باشد.

در ابتدا، ابعاد ماتریس‌های ورودی A و B با 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 result با 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 = 1:rows و for y = 1:cols))، هر یک از پیکسل‌های ماتریس‌های A و B بررسی می‌شود. در داخل این حلقه‌ها، برای هر پیکسل از ماتریس‌ها، عمل منطقی AND-NOT انجام می‌شود. این کار با ضرب ماتریس A در معکوس ماتریس B (که با استفاده از عملگر ~ انجام می‌شود) صورت می‌گیرد. در واقع، معکوس ماتریس B به این معناست که مقادیر 1 تبدیل به 0 و مقادیر 0 تبدیل به 1 می‌شوند. نتیجه نهایی ضرب این دو مقدار به طور طبیعی همان نتیجه عملیات AND-NOT است. در این عملیات، اگر پیکسل در A برابر 1 باشد و پیکسل متناظر در B برابر 0 باشد، نتیجه نهایی 1 خواهد بود. در غیر این صورت، نتیجه برابر 0 خواهد بود. در نهایت، پس از تکمیل عملیات برای تمام پیکسل‌ها، ماتریس result که 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.



1. عملیات XOR

عملیات **XOR** (Exclusive OR) بر اساس این قانون عمل می‌کند: اگر دو ورودی برابر باشند، نتیجه 0 خواهد بود؛ در غیر این صورت، نتیجه 1 خواهد بود.

|  |  |  |
| --- | --- | --- |
| A XOR B | B | A |
| 0 | 0 | 0 |
| 1 | 1 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |

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

10.

11. result = zeros(rowsA, colsA);

12.

13. for y = 1:rowsA

14. for x = 1:colsA

15. if A(y, x) ~= B(y, x)

16. result(y, x) = 1;

17. else

18. result(y, x) = 0;

19. end

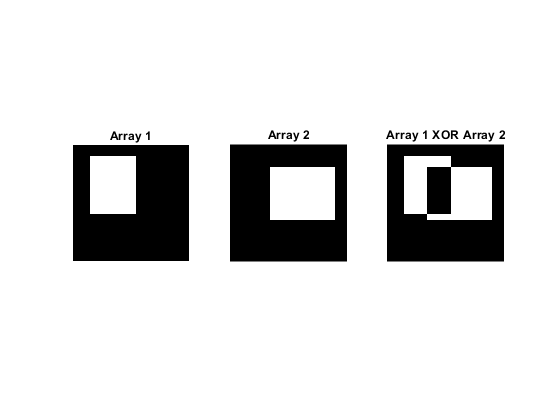
20. end

21. end

22. 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 0 و 1 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 = 1:rowsA و for x = 1: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 1 و دیگری 0 باشد)، نتیجه در ماتریس result برابر 1 خواهد بود. در غیر این صورت، اگر مقادیر برابر باشند، نتیجه برابر 0 خواهد بود. این منطق دقیقاً همان عملکرد XOR است که در آن دو مقدار متفاوت نتیجه 1 می‌دهد و دو مقدار مشابه نتیجه 0 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(200, 200), دو آرایه array1 و array2 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 array1 قرار گیرد، شامل موقعیت شروع مستطیل و ابعاد آن می‌شود. موقعیت شروع مستطیل از سلول (20, 30) 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 array1 اختصاص داده می‌شود. مشابه مرحله قبل، پارامترهای مستطیل دوم برای آرایه array2 تعریف می‌شود. این مستطیل از نقطه شروع (40, 70) 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 array2 اختصاص داده می‌شود.

1. %% In this section, we want to implement some important logical operations.

2. % Create two 200x200 arrays filled with zeros

3. array1 = zeros(200, 200);

4. array2 = zeros(200, 200);

5.

6. % Define the rectangle parameters in array1

7. x\_start1 = 20;

8. y\_start1 = 30;

9. rect\_length1 = 80;

10. rect\_height1 = 100;

11.

12. % Add the rectangle (set the region to 1)

13. array1(x\_start1:x\_start1+rect\_height1-1, y\_start1:y\_start1+rect\_length1-1) = 1;

14.

15. % Define the rectangle parameters in array1

16. x\_start2 = 40;

17. y\_start2 = 70;

18. rect\_length2 = 110;

19. rect\_height2 = 90;

20.

21. % Add the rectangle (set the region to 1)

22. array2(x\_start2:x\_start2+rect\_height2-1, y\_start2:y\_start2+rect\_length2-1) = 1;

در این بخش از کد، عملیات منطقی "NOT" بر روی دو آرایه‌ی array1 و array2 که پیش‌تر ایجاد شده‌اند، 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 بررسی تفاوت‌ها و تغییرات در داده‌های باینری استفاده می‌شود. در ابتدا با استفاده از تابع imshow و درون یک پنجره گرافیکی، دو آرایه array1 و array2 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 1" و "Array 2" در بالای هر تصویر نمایش داده می‌شود. در این مرحله، از تابع logical\_not برای انجام عملیات "NOT" روی هر دو آرایه استفاده می‌شود. این تابع که به صورت دستی پیاده‌سازی شده است، مقادیر ۱ را به ۰ و مقادیر ۰ را به ۱ تبدیل می‌کند. این تغییرات برای آرایه‌های array1 و array2 به ترتیب در متغیرهای image\_not\_array1 و image\_not\_array2 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 دست آمده در دو زیر پنجره دیگر به نمایش در می‌آیند. اولین پنجره نتایج تغییرات بر روی array1 را نشان می‌دهد که در آن مستطیل‌ها به صورت معکوس نمایش داده می‌شوند. مشابه همین روند برای array2 انجام می‌شود و نتیجه آن در پنجره بعدی به نمایش در می‌آید. عنوان‌های "Not array 1" و "Not array 2" در بالای تصاویر نمایش داده شده به‌طور واضح نتایج اعمال عملیات "NOT" را مشخص می‌کنند.

1. %% NOT

2. % Display the arrays to verify the rectangle placement

3. figure(1);

4. subplot(2,2,1);

5. imshow(array1);

6. title('Array 1');

7.

8. subplot(2,2,2);

9. imshow(array2);

10. title('Array 2');

11.

12. image\_not\_array1 = logical\_not(array1);

13. image\_not\_array2 = logical\_not(array2);

14.

15. subplot(2,2,3);

16. imshow(image\_not\_array1);

17. title('Not array 1');

18.

19. subplot(2,2,4);

20. imshow(image\_not\_array2);

21. title('Not array 2');

در این بخش از کد، عملیات منطقی "AND" بر روی دو آرایه‌ی باینری array1 و array2 انجام می‌شود. عملیات "AND" در منطق باینری یک عملیات پایه‌ای است که دو ورودی را گرفته و تنها زمانی خروجی آن برابر ۱ خواهد شد که هر دو ورودی به طور همزمان برابر با ۱ باشند. این عملیات برای ترکیب دو آرایه باینری و مشاهده تداخل و اشتراک آن‌ها در مقادیر ۱ استفاده می‌شود. ابتدا از تابع logical\_and برای اعمال عملیات "AND" بر روی دو آرایه‌ی array1 و array2 استفاده می‌شود. این تابع به‌طور عنصر به عنصر دو آرایه را با هم ترکیب می‌کند و نتیجه‌ی آن در متغیر image\_and ذخیره می‌شود. مقادیر ۱ در این متغیر تنها در جاهایی که هر دو آرایه در آن موقعیت مقدار ۱ دارند، ذخیره خواهند شد. به عبارت دیگر، این عمل به‌طور دقیق تداخل و اشتراک 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 ۱ اضافه می‌شود. این کار با استفاده از ایندکس‌گذاری مستقیم انجام می‌شود، که در آن برای هر مستطیل (هم در array1 و هم در array2)، مقدار ۱ در موقعیت‌های خاص مرزی تنظیم می‌شود. در نهایت، مرزهای مستطیل‌ها در تصویر image\_and با رنگ سفید (مقدار ۱) به نمایش در می‌آید. .در نهایت، نتایج به‌دست آمده به‌طور گرافیکی نمایش داده می‌شود

1. %% AND

2. % Find the border of the rectangles (the edges of the 1's in the arrays)

3.

4. image\_and = logical\_and(array1, array2);

5.

6. % For array1

7. image\_and(x\_start1,y\_start1:y\_start1+rect\_length1-1) = 1;

8. image\_and(x\_start1+rect\_height1-1,y\_start1:y\_start1+rect\_length1-1) = 1;

9. image\_and(x\_start1:x\_start1+rect\_height1-1, y\_start1) = 1;

10. image\_and(x\_start1:x\_start1+rect\_height1-1, y\_start1+rect\_length1-1) = 1;

11.

12. % For array2

13. image\_and(x\_start2,y\_start2:y\_start2+rect\_length2-1) = 1;

14. image\_and(x\_start2+rect\_height2-1,y\_start2:y\_start2+rect\_length2-1) = 1;

15. image\_and(x\_start2:x\_start2+rect\_height2-1, y\_start2) = 1;

16. image\_and(x\_start2:x\_start2+rect\_height2-1, y\_start2+rect\_length2-1) = 1;

17.

18. figure(2);

19. subplot(1,3,1);

20. imshow(array1);

21. title('Array 1');

22. subplot(1,3,2);

23. imshow(array2);

24. title('Array 2');

25. subplot(1,3,3);

26. imshow(image\_and);

27. title('Array 1 AND Array 2');

در این بخش از کد، عملیات منطقی "OR" بر روی دو آرایه‌ی باینری array1 و array2 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\_or برای اعمال عملیات "OR" بر روی دو آرایه‌ی array1 و array2 استفاده می‌شود. این تابع به‌طور عنصر به عنصر دو آرایه را با هم ترکیب می‌کند و نتیجه‌ی آن در متغیر image\_or ذخیره 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.

10. subplot(1,3,2);

11. imshow(array2);

12. title('Array 2');

13.

14. subplot(1,3,3);

15. imshow(image\_or);

16. title('Array 1 OR Array 2');

در این بخش از کد، عملیات منطقی "AND-NOT" بر روی دو آرایه باینری array1 و array2 انجام می‌شود. عملیات "AND-NOT" ترکیبی از دو عمل منطقی "AND" و "NOT" است. ابتدا، عملیات "NOT" روی آرایه دوم (array2) اعمال می‌شود که باعث معکوس کردن مقادیر ۰ و ۱ می‌شود. سپس، عملیات "AND" روی آرایه اول (array1) و آرایه دوم معکوس‌شده انجام می‌شود. به عبارت ساده‌تر، این عملیات تنها زمانی مقدار ۱ در خروجی می‌گذارد که مقدار متناظر در array1 برابر با ۱ باشد و در array2 مقدار معکوس شده آن برابر با ۱ باشد (یعنی در array2 مقدار ۰). بنابراین، در نتیجه، تنها نواحی که در array1 دارای مقدار ۱ هستند و در array2 دارای مقدار ۰ هستند، در تصویر نهایی نمایش داده می‌شوند. در این کد، ابتدا آرایه‌های array1 و array2 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\_not نمایش داده می‌شود.

1. image\_and\_not = logical\_and\_not(array1, array2);

2.

3. figure(4);

4. subplot(1,3,1);

5. imshow(array1);

6. title('Array 1');

7.

8. subplot(1,3,2);

9. imshow(array2);

10. title('Array 2');

11.

12. subplot(1,3,3);

13.

14. % For array2

15. image\_and\_not(x\_start2,y\_start2:y\_start2+rect\_length2-1) = 1;

16. image\_and\_not(x\_start2+rect\_height2-1,y\_start2:y\_start2+rect\_length2-1) = 1;

17. image\_and\_not(x\_start2:x\_start2+rect\_height2-1, y\_start2) = 1;

18. image\_and\_not(x\_start2:x\_start2+rect\_height2-1, y\_start2+rect\_length2-1) = 1;

19.

20. imshow(image\_and\_not);

21. title('Array 1 AND [NOT Array 2]');

در این بخش از کد، عملیات XOR یا Exclusive OR روی دو آرایه array1 و array2 انجام 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 باشد)، نتیجه ۱ خواهد بود، اما اگر مقادیر مشابه باشند (یعنی هر دو ۰ یا هر دو ۱)، نتیجه ۰ خواهد بود. این به این معناست که فقط در مکان‌هایی که دو آرایه مقادیر مختلف دارند، نتیجه عملیات ۱ می‌شود. در ابتدا، دو آرایه array1 و array2 که قبلاً مستطیل‌هایی از ۱ دارند، به تابع manual\_xor داده می‌شوند. این تابع به طور دستی و با بررسی هر خانه از این دو آرایه، مقدار XOR را محاسبه می‌کند. به این صورت که برای هر خانه از آرایه‌ها، اگر مقادیر آن‌ها متفاوت باشند، در آرایه نتیجه، ۱ قرار می‌دهد و اگر مشابه باشند، ۰ قرار می‌دهد.

1. %% XOR

2.

3. figure(5);

4. image\_xor = manual\_xor(array1, array2);

5.

6. subplot(1,3,1);

7. imshow(array1);

8. title('Array 1');

9.

10. subplot(1,3,2);

11. imshow(array2);

12. title('Array 2');

13.

14. subplot(1,3,3);

15. imshow(image\_xor);

16. title('Array 1 XOR Array 2');

سوال 5

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:

1. % Define a 3\*3 avraging filter

2. kernel\_3 = ones(3, 3) / 9;

3. kernel\_7 = ones(7, 7) / 49;

4. kernel\_11 = ones(11, 11) / 121;

5. kernel\_23 = ones(23, 23) / 529;

نحوه عملکرد کرنل برای حالتیکه ما از یک کرنل 3 در 3 استفاده می­کنیم در زیر آورده شده است.

ابتدا یک آرایه 3 در 3 با مقادیر یکسان واحد تعریف شده و همگی درایه ها بر 9 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 9) تقسیم شود. لذا نمایش این کرنل به صورت ماتریسی به قرار زیر است:

نحوه محاسبه کرنل برای کرنل با ابعاد بزرگتر نیز مشابه مثال بالاست. سپس بعد از تعریف این کرنلها، با دستور conv2 این کرنلها را به تصویر اصلی اعمال کردیم.

1. % Apply the filter using convolution

2. smooth\_lenna\_kernel3 = conv2(double(lenna\_img), kernel\_3, 'same');

3. smooth\_lenna\_kernel7 = conv2(double(lenna\_img), kernel\_7, 'same');

4. smooth\_lenna\_kernel11 = conv2(double(lenna\_img), kernel\_11, 'same');

5. smooth\_lenna\_kernel23 = conv2(double(lenna\_img), kernel\_23, 'same');

6.

7. % Convert to uint8 format

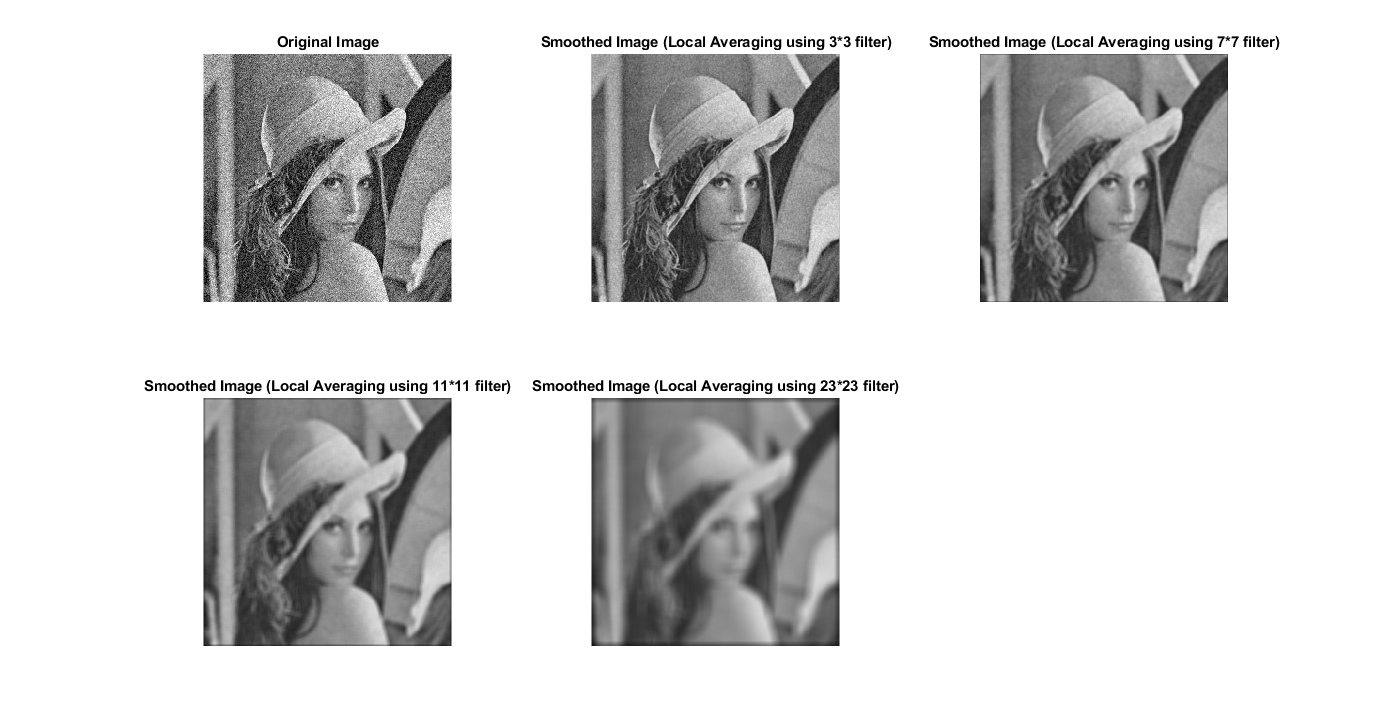
8. smooth\_lenna\_kernel3 = uint8(smooth\_lenna\_kernel3);

9. smooth\_lenna\_kernel7 = uint8(smooth\_lenna\_kernel7);

10. smooth\_lenna\_kernel11 = uint8(smooth\_lenna\_kernel11);

11. smooth\_lenna\_kernel23 = uint8(smooth\_lenna\_kernel23);

در این بخش از کد، ابتدا فیلترهای میانگین‌گیری با اندازه‌های مختلف بر روی تصویر اعمال می‌شوند. این کار با استفاده از تابع conv2 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 uint8 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\_kernel3، smooth\_lenna\_kernel7، smooth\_lenna\_kernel11 و smooth\_lenna\_kernel23 ذخیره می‌شوند. پس از انجام کانولوشن، خروجی‌ها که در فرمت double هستند، برای نمایش در متلب به uint8 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 3 در 3 بهترین انتخاب است.

سوال 6

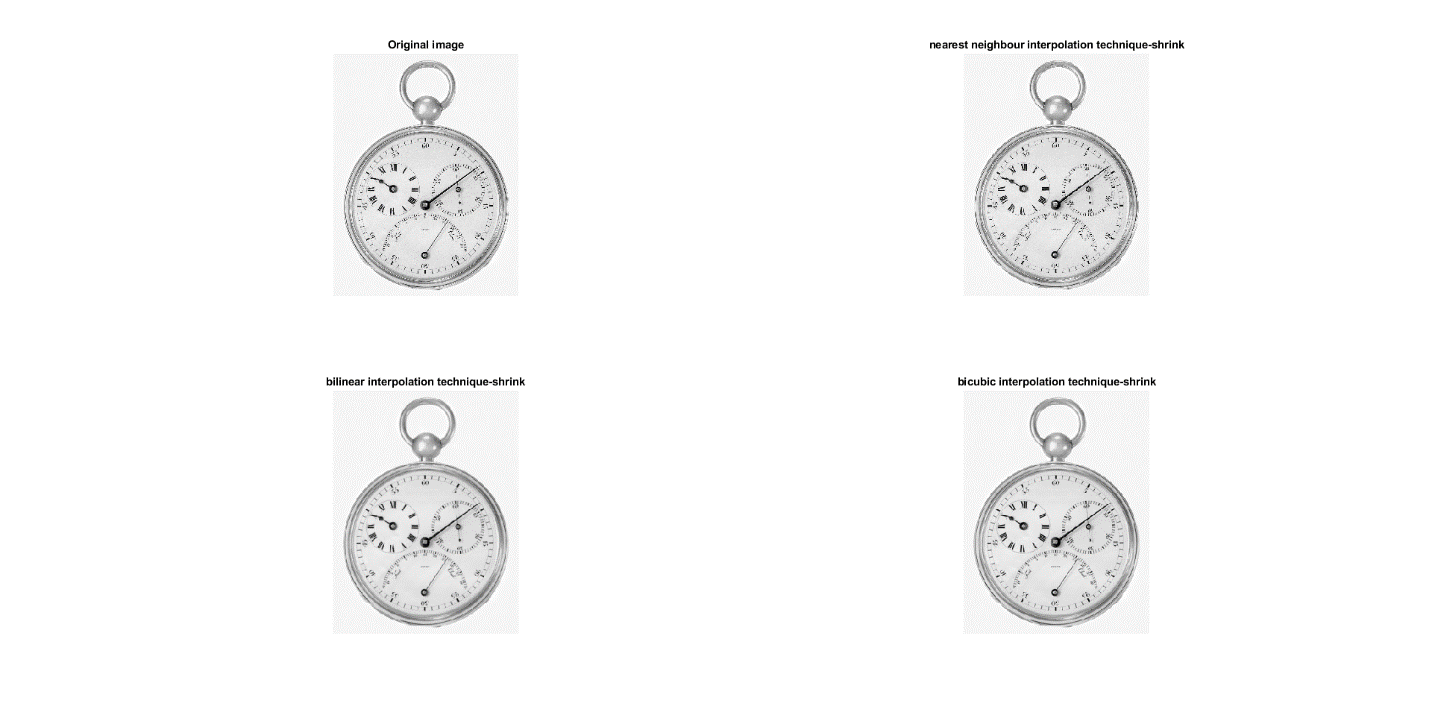
در این پروژه، سه تکنیک مختلف میان‌یابی (interpolation) برای تغییر اندازه یک تصویر مورد استفاده قرار گرفته‌اند: **Nearest Neighbour Interpolation**، **Bilinear Interpolation** و **Bicubic Interpolation**. هدف این است که تفاوت‌های این روش‌ها در کاهش و افزایش اندازه تصویر بررسی شوند. برای این منظور، ابتدا یک تصویر ساعت به نام watch.tif بارگذاری 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:

1. %shrinking the watch image

2. imageA\_shr=imresize(watch\_image,1/5,'nearest');

3. imageB\_shr=imresize(watch\_image,1/5,'bilinear');

4. imageC\_shr=imresize(watch\_image,1/5,'bicubic');



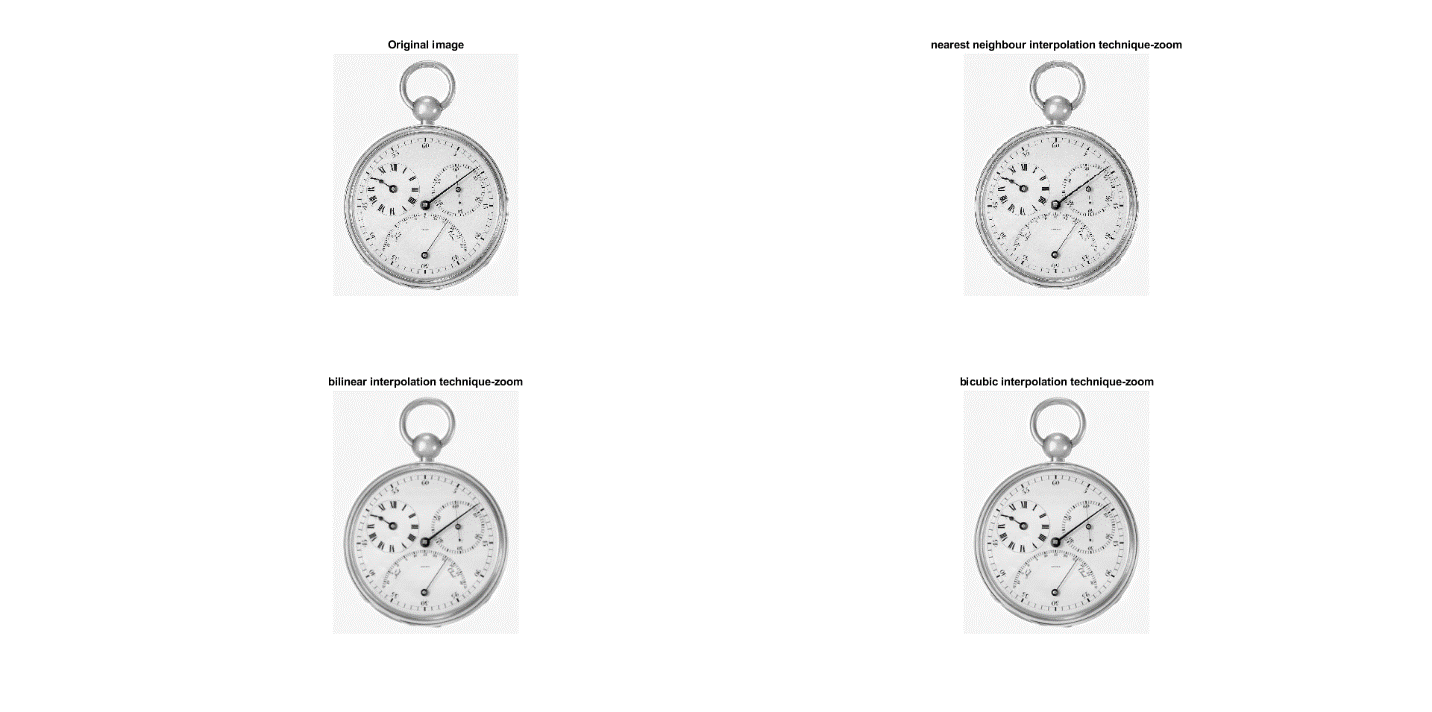
بدیهی است که ابعاد تصویر در هر روش به یک پنجم مقدار اولیه تبدیل می­شود. برای اینکه تفاوت بین روشهای مختلف را مشاهده کنیم لازم است تا ابعاد تصویر به اندازه مقدار اولیه بازگردند. برای این منظور از کد زیر استفاده شده است:

1. %zooming the watch image

2. imageA\_zoom=imresize(imageA\_shr,5,'nearest');

3. imageB\_zoom=imresize(imageB\_shr,5,'bilinear');

4. imageC\_zoom=imresize(imageC\_shr,5,'bicubic');



همانطور که در نتایج مشخص است روش دوم و سوم نتایج واضح تری را ایجاد کرده اند. روش **Nearest Neighbour** ساده‌ترین و سریع‌ترین روش میان‌یابی است، اما هنگام بزرگ‌سازی تصویر معمولاً نتیجه به صورت پیکسلی و خشن به نظر می‌رسد. این روش در کاهش اندازه به خوبی عمل می‌کند، اما در بزرگ‌سازی کیفیت تصویر به مراتب پایین‌تر است. در مقابل، روش **Bilinear** کیفیت بهتری نسبت به **Nearest Neighbour** ارائه می‌دهد. در کاهش اندازه، جزئیات بیشتری حفظ می‌شود و در بزرگ‌سازی نیز نسبت به روش **Nearest Neighbour** نتیجه بهتری ایجاد می‌کند، اما هنوز تصویر به اندازه کافی نرم و صاف نیست. در نهایت، روش **Bicubic** دقیق‌ترین و بهترین روش میان‌یابی است که در کاهش اندازه تصویر، جزئیات دقیق‌تری حفظ می‌شود و در بزرگ‌سازی نیز تصویری نرم‌تر و با جزئیات بیشتر به نمایش گذاشته می‌شود. این روش معمولاً در کاربردهایی که نیاز به کیفیت بالا دارند، ترجیح داده می‌شود.

سوال 7

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.

1. % Load the image

2. img = imread('Images/WashingtonDC.tif');

3. whos img

خروجی کد بالا شامل ابعاد تصویر و تعداد بیتهای هر پیکسل(8 بیت) به صورت زیر است:

1. Name Size Bytes Class Attributes

2.

3. img 564x564 318096 uint8

برای اینکه بیت ششم از هر پیکسل برابر صفر شود لازم است تا اطلاعات هر پیکسل در عبارت 11011111 ضرب شود. عبارت بالا از تفریق عدد 11111111 و 00100000 حاصل می­شود. برای ایجاد تصویری که هر پیکسل آن برابر با مقدار 11011111 باشد از دستور uint8(255 - 32) 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.

1. img\_bits = uint8(img); % Ensure the image is in uint8 format (0-255)

2. img\_modified = bitand(img\_bits, uint8(255 - 32));

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

1. difference = abs(double(img) - double(img\_modified)); % Compute absolute difference

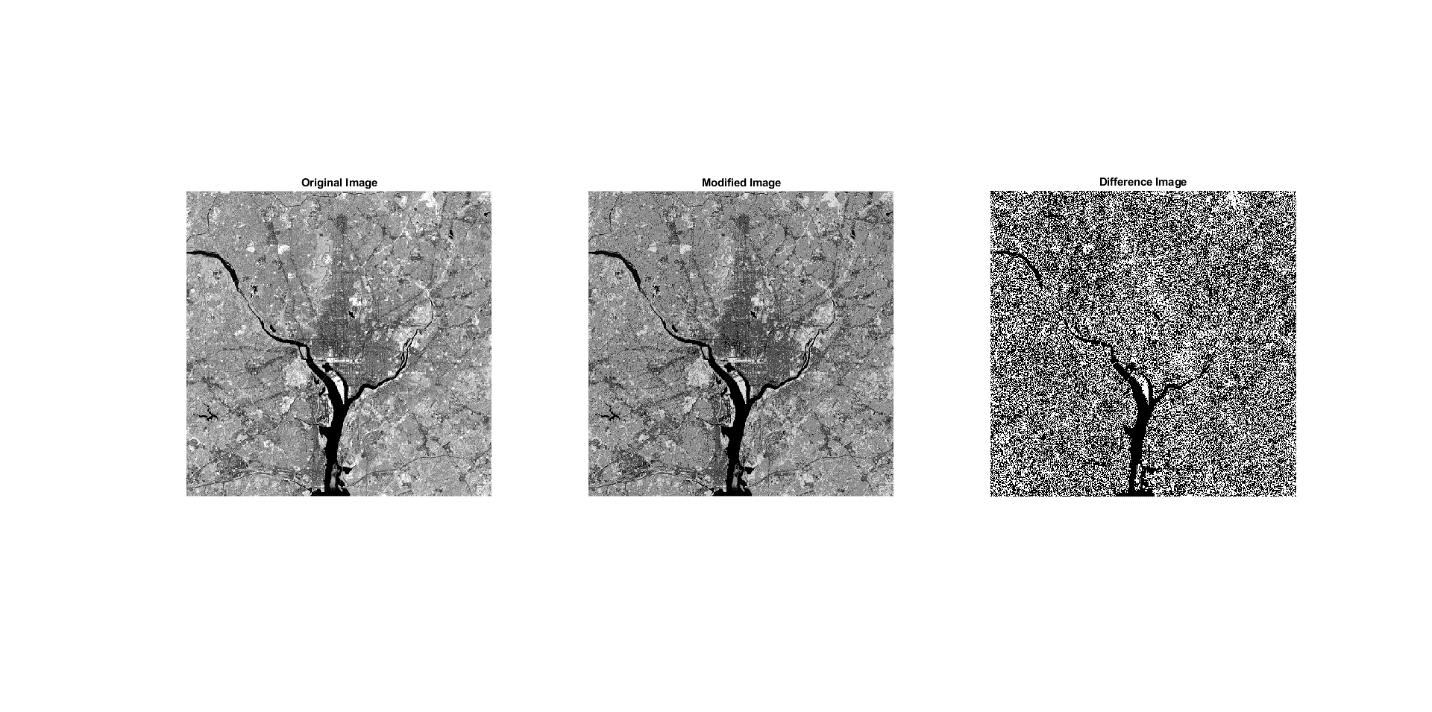
با توجه به اینکه بازه تغییرات شدت روشنایی بین 0 تا 255 باید باشد لذا از تابع abs استفاده شده است. در نهایت برای محاسبه میزان اختلاف ابتدا تمامی مقادیر اختلاف برای تمامی پیکسلها را جمع کرده و بر تعداد پیکسلها تقسیم شده است.

1. 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 14.93 به این معناست که پس از اصلاح بیت ششم در هر پیکسل، به‌طور متوسط 14.93 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 32 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 imread در 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 double تبدیل 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 255 - uint8(blood\_vessels) انجام می‌شود که باعث معکوس شدن رنگ‌ها و واضح‌تر شدن رگ‌ها در تصویر می‌شود.

1. mask\_image = imread('Images/angiography\_mask.tif');

2. live\_image = imread('Images/angiography\_live.tif');

3.

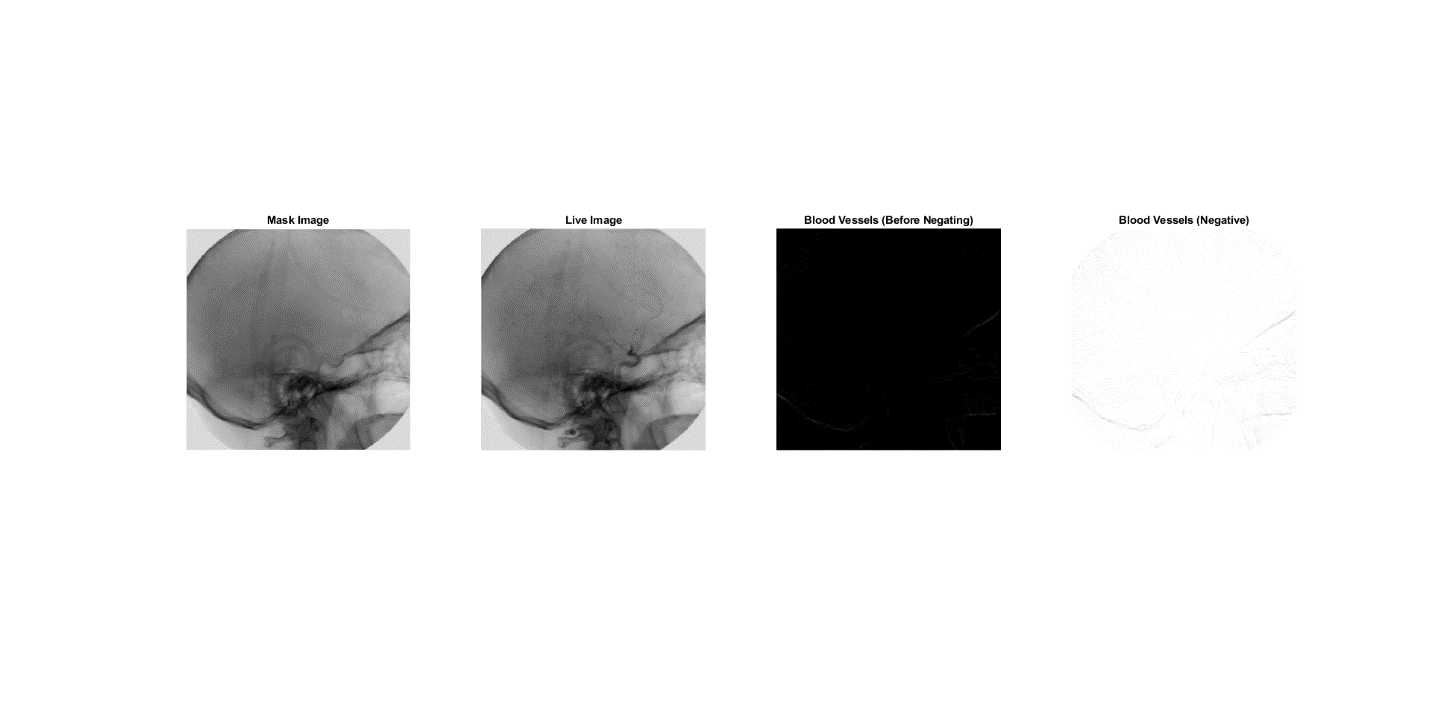
4. % Step 2: Perform subtraction (highlight the blood vessels)

5. blood\_vessels = double(live\_image) - double(mask\_image); % Convert to double for subtraction

6.

7. % Step 3: Enhance visualization by displaying the negative of the result

8. blood\_vessels\_negative = 255 - uint8(blood\_vessels); % Make sure it's in the 0-255 range

در نهایت، تصاویر مختلف در یک پنجره گرافیکی با استفاده از دستور subplot به 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.