

گزارش تشریحی تمرین سری اول

درس پردازش تصاویر دیجیتال

استاد: دکتر آذرنوش

دانشجو: محمدجواد زلّقی

شماره دانشجویی: ۹۸۱۲۶۰۷۹

تاریخ: ۱۳۹۹/۸/۱

سوال اول: الف)

ابتدا تصویر را با دستور زیر خواندیم و در متغیر `img0` بصورت آرایه ذخیره کردیم:

```
img0 = cv2.imread("mandrill.jpg")
```

برای خواندن ابعاد تصویر از ویژگی `img0.shape` که ابعاد آرایه را می‌خواند، استفاده کردیم. ابعاد تصویر را در متغیر `dim0` ذخیره کردیم. ابعاد تصویر به شرح زیر بود:

```
(512, 512, 3)
```

در واقع یک تصویر با طول و عرض ۵۱۲ و سه سطح رنگی RGB داریم.

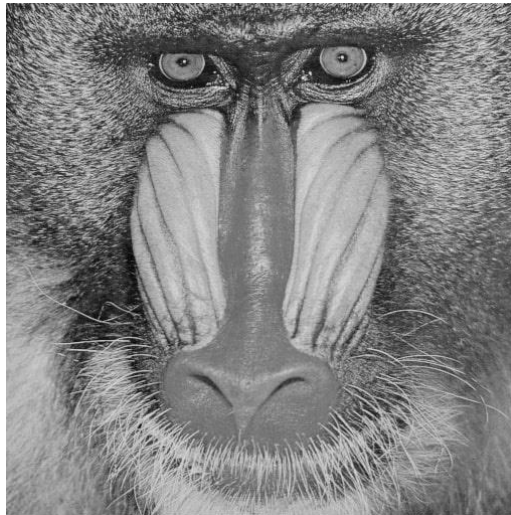
برای بررسی نوع داده‌ای پیکسل‌ها از ویژگی `img0.dtype` استفاده کردیم که در خروجی نوع داده‌ای آرایه را می‌دهد. ما این نوع را در متغیر `dt0` ذخیره کردیم. نوع داده‌ای هر پیکسل تصویر به شرح زیر بود:

```
dtype('uint8')
```

که توضیح آن به معنای این است که تمامی پیکسل‌ها دارای عدد بدون علامت ۰ تا ۲۵۵ هستند. بعبارتی تصویر ۸ بیتی است.

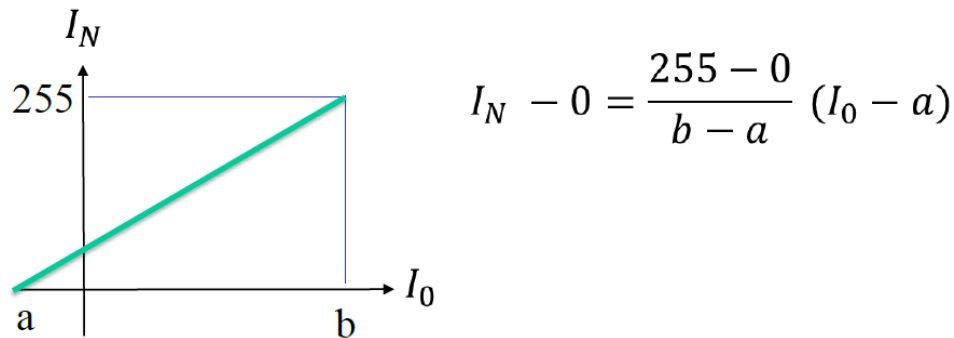
سوال اول: ب)

با تابع `cv2.cvtColor(img0, cv2.COLOR_BGR2GRAY)` تصویر خوانده شده در قسمت الف را به فضای خاکستری نگاشت می‌کنیم و در متغیر `img1` ذخیره می‌کنیم. همچنین با تابع `cv2.imwrite("mandrill_gray.jpg", img1)` آن را در فایل آرگومان ذخیره می‌کنیم و با تابع `cv2.imshow("gray mandrill:", img1)` تصویر خاکستری را نمایش می‌دهیم که به شکل پایین است:

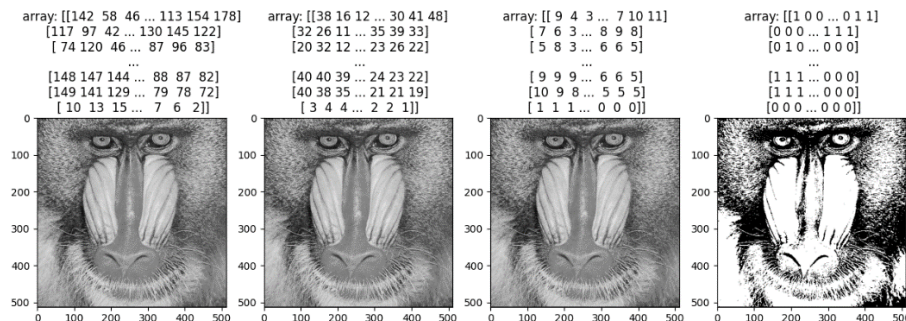


سوال اول: ج)

ابتدا تابع `contrastStretching(img, wantedRange)` را تعریف کردیم که ورودی آن تصویر با گری لول‌های مشخص است (با توابع مین و ماکس پیدا می‌شوند) و خروجی این تابع در واقع تصویری است که شدت پیکسل‌های آن تحت تبدیل خطی CONTRAST STRETCHING به شرح زیر (از لکچر نت‌های استاد) قرار گرفته است:



سپس برای بازه‌های خواسته شده یعنی ۰-۶۳، ۱۵-۰ و ۰-۱ تصاویر تحت این تبدیل قرار گرفته و به همراه مختصر آرایه به شکل زیر نمایش داده می‌شوند:



مشخص است رنج سطوح روشنایی از تصویر اصلی که ۰-۲۵۵ بود از چپ به راست کم می‌شود تا نهایتاً به تصویر باینری آخر می‌رسیم.

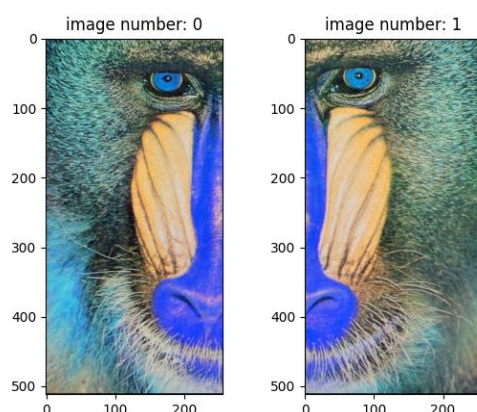
اثر تعداد سطوح: هر چه تعداد سطوح کم‌تر می‌شود، تصویر شارپ‌تر می‌شود. بعبارتی شدت‌های میانی رفته رفته حذف می‌شوند. توجه داریم اگر کنتراست یک تصویر پایین باشد و این بخاطر توزیع محدود پیکسل‌ها در هیستوگرام باشد، با افزایش تعداد سطوح می‌توان (عکس کاری که کردیم) کنتراست آن را بهتر کرد. اما در اینجا ما تصویر را از چپ به راست شارپ کردیم.

سوال اول: (د)

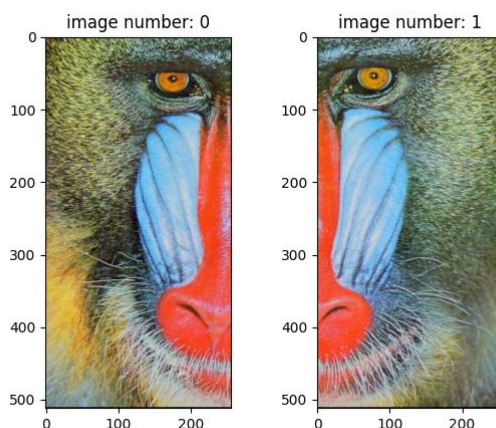
بدین منظور تابع `imgCrop(img, cropRange)` را تعریف کردیم که در ورودی یک تصویر و مختصات دو گوشه یک مستطیل را می‌گیرد تا از تصویر، آن مستطیل را جدا کرده و در خروجی بعنوان تصویری جدید بدهد.

عملکرد تابع نیز به این شکل است که یک آرایه خالی هم ابعاد با ناحیه‌ای که در ورودی داده می‌شود، ایجاد می‌کند و سپس با کمک حلقه مقدار تک تک پیکسل‌ها را متناظر با تصویر ورودی تابع تعیین می‌کند.

نهایتاً بر روی تصویر اصلی با اعمال این تابع و مشخص کردن مختصات مدنظر بر اساس طول و عرض تصویر اصلی و نصف آن‌ها، دو تصویر راست و چپ متقارن از تصویر اول ایجاد کردیم و در خروجی نمایش دادیم. خروجی تصویری به شکل زیر شد:

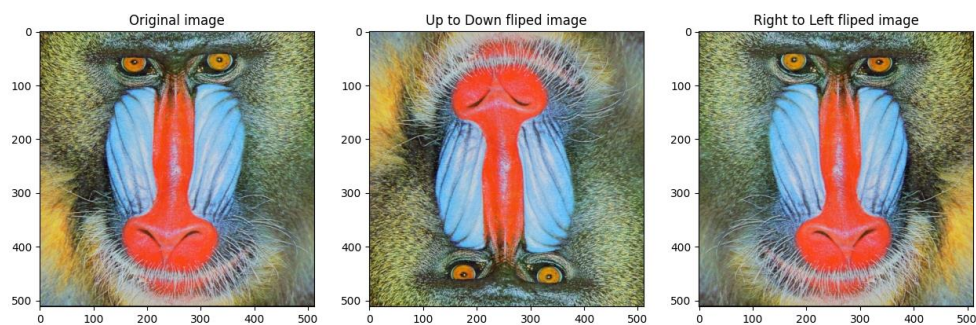


نکته: دلیل تغییر رنگ‌های تصویر نسبت به تصویر اصلی مرجع، تفاوت در قرارداد RGB و BGR است که در openCv تصویر بصورت قرارداد دوم نمایش داده می‌شود. البته براحتی می‌توان به همان RGB با دستور `cvtColor()` تغییر داد:



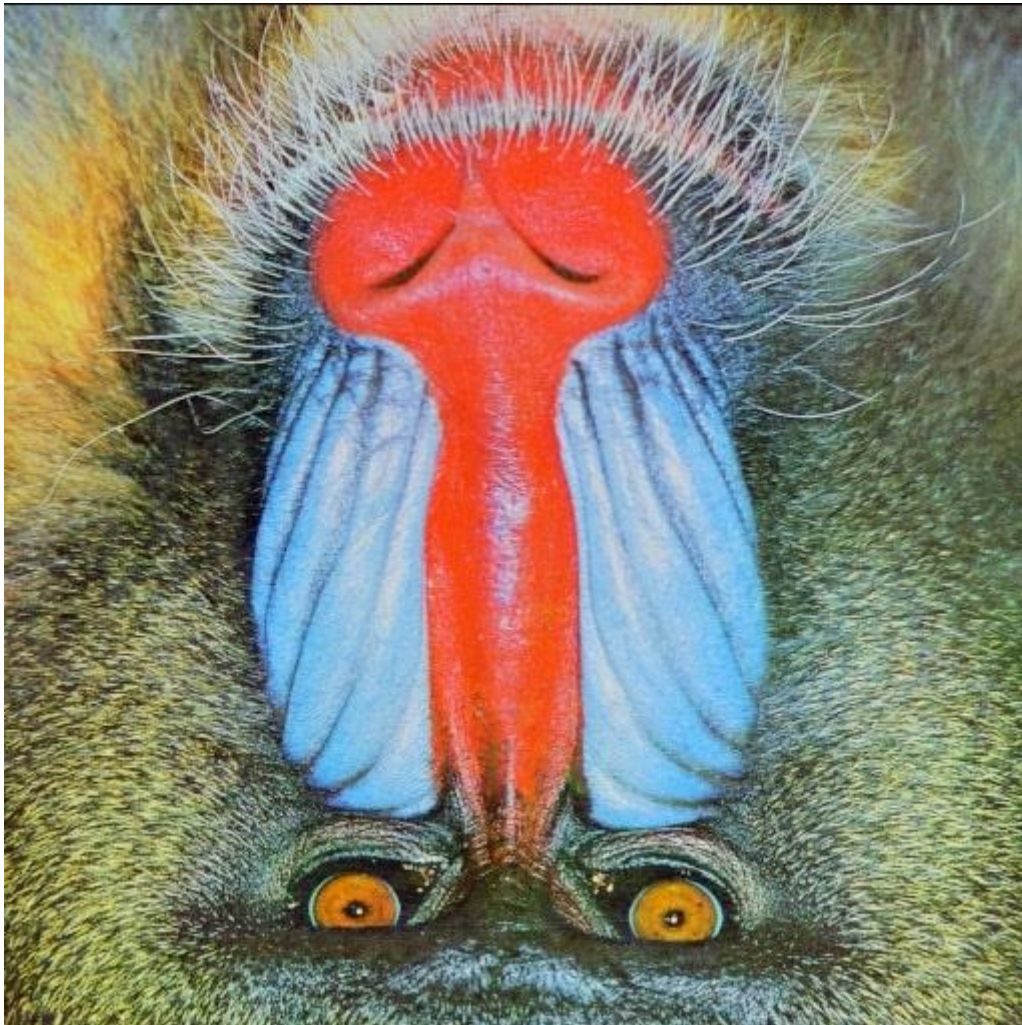
سوال اول: ه)

برای واردن کردن تصویر از دستور `cv2.flip()` استفاده شده است. این تابع تصویر مبدأ و نوع واردين شدن را با يك كد بعنوان ورودی می گیرد و در خروجی تصویر وارون شده را می دهد. توجه داریم كد ۰ وارون عمودی و كد ۱ وارون افقی را انجام می دهد. نهایتاً تصویر اصلی و دو تصویر وارون شده را نمایش دادیم. این پنجره به شكل زیر است:



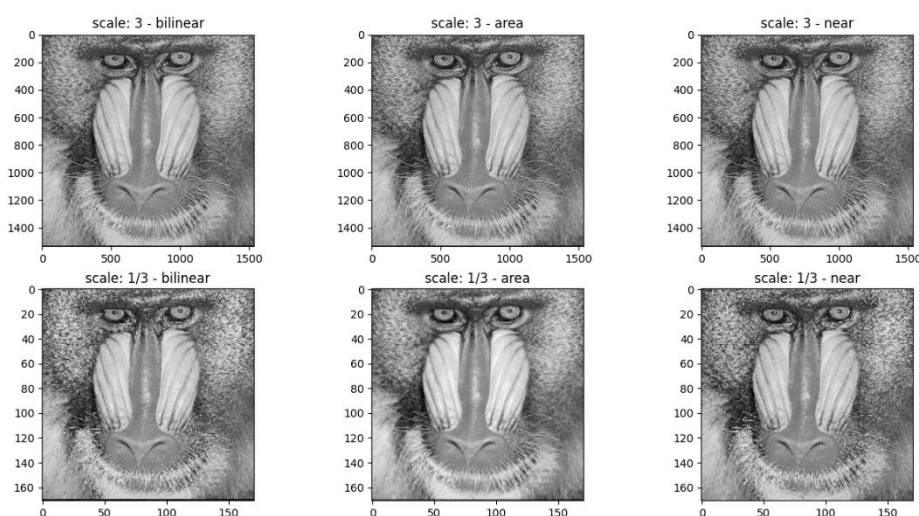
سوال اول: (و)

برای ذخیره کردن تصاویر با کمک ماژول openCV از تابع `v2.imwrite("Up_to_Down_fliped_image.png", cv2.cvtColor(img0_fliped_u2d, cv2.COLOR_BGR2RGB))` استفاده می‌کنیم. آرگومان اول نام و فرمت فایل ذخیره شده در دایرکتوری و آرگومان دوم تصویری است که قصد ذخیره کردن آن را داریم. نهایتاً فایل با اجرای برنامه در دایرکتوری سوال اول ذخیره می‌شود. فایل ذخیره شده بصورت زیر است:



سوال اول: ی)

برای تغییر رزولوشن از تابع `cv2.resize()` استفاده می‌کنیم که در ورودی تصویر مبدأ، اسکیل تغییر تعداد پیکسل‌ها در راستای افقی و عمودی و نحوه درونیابی برای تعیین مقدار پیکسل‌ها در تصویر اسکیل شده را می‌گیرد و در خروجی تصویر اسکیل شده را می‌دهد. برای روش‌های درونیابی خواسته شده در `cv2` از متدهای `cv2.INTER_NEAREST`، `cv2.INTER_LINEAR` و `cv2.INTER_NEAREST` استفاده می‌کنیم. سپس تصاویر را بصورت زیر نمایش می‌دهیم:



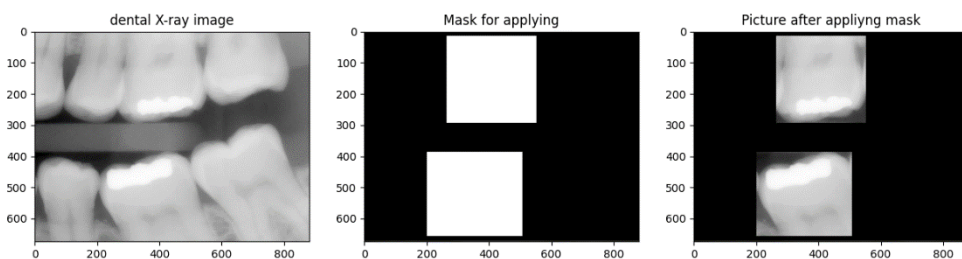
بحث: در ابتدا باید ذکر شود که با توجه به ابعاد تصاویر اسکیل شدن آنها کاملاً مشهود است. همچنین بصورت کلی برداشت می‌شود در افزایش رزولوشن با هر روش درونیابی تصویر دچار شارپنینگ نمی‌شود اما در کاهش رزولوشن تصویر شارپ‌تر می‌شود و پله‌های تصویر مشهودتر هستند. در کاهش رزولوشن با روش تکرار پیکسل تصویر اسموزتر (یا بلرتر) می‌شود. پس از روش پیکسل نزدیک یا خطی می‌تواند بهتر باشد. همچنین در افزایش رزولوشن روش خطی در زوم روی تصویر دارای پیوستگی بهتری از دو روش دیگر هست.

می‌توان خلاصه کرد: در کاهش رزولوشن روش خطی و نزدیک‌ترین همسایه تصویر را شارپ می‌کنند اما روش تکرار پیکسل تصویر را اسموز (یا بلر می‌کند). در افزایش رزولوشن هر سه روش خروجی تقریباً خوبی دارند، اما روش درونیابی خطی تصویر پیوسته‌تری خصوصاً در لبه‌ها می‌توان داشت.

همچنین در یک بحث در سایت استک آور فلو ([لینک بحث](#)) درباره این روش‌ها با توضیحات ما بحث شده است. همچنین درباره سرعت هر یک از این الگوریتم‌ها نیز می‌توان بحث کرد که بستگی به بار محاسباتی دارند. بطور مختصر تکرار پیکسل، خطی و نزدیک‌ترین الگوریتم‌ها قرار می‌گیرند.

سوال دوم: الف)

ابتدا تصاویر را می‌خوانیم. توجه داریم که پایتون طبق قرارداد BGR تصاویر را می‌خواند اما تصاویر واقعی خاکستری هستند. برای همین در ابتدا آن‌ها را با تبدیل به فضای خاکستری می‌بریم. سپس یک تصویر با ابعاد تصویر ورودی با مقدار پیکسل‌های صفر ایجاد می‌کنیم. سپس با حلقه چک می‌کنیم اگر پیکسلی از ماسک غیر از صفر بود، آن وقت مقدار شدت متناظر با آن موقعیت را از تصویر اصلی در تصویر خواسته شده قرار می‌دهیم. توجه داریم یک راه دیگر استفاده از ضرب درایه به درایه است که البته باید با یک نگاشت مقادیر را به همان ۰-۲۵۵ برگردانیم. با این حال ما از روش اول استفاده کردیم. نتایج بصورت زیر در یک تصویر نمایش داده می‌شود:



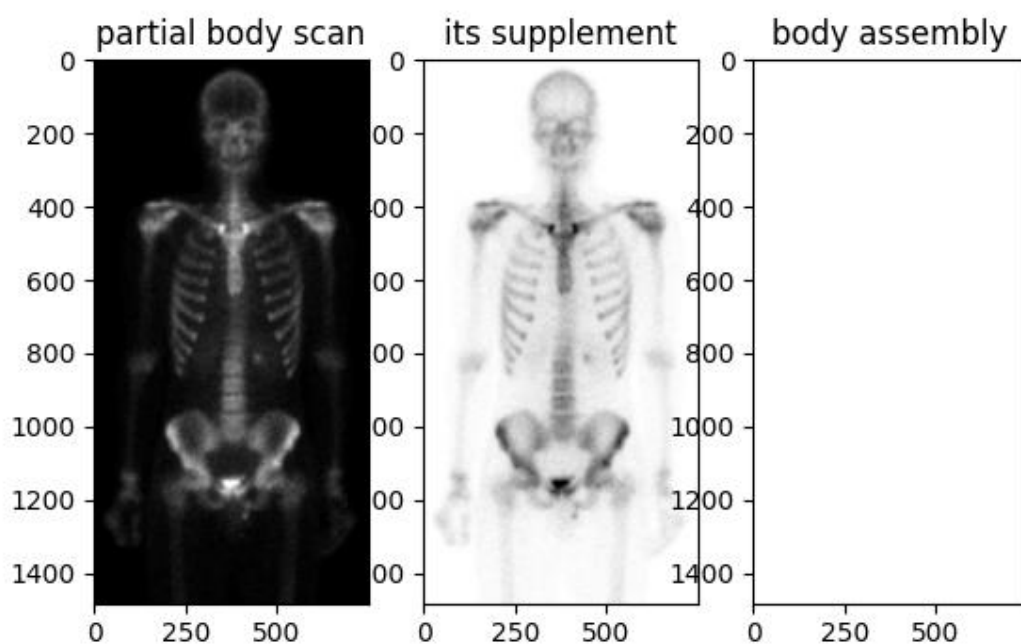
همچنین لازم به توضیح است که همین عملیات را می‌توان با تعریف یک تابع انجام داد تا بصورت عمومی بشود از آن استفاده کرد.

سوال دوم: ب)

ابتدا تصویر بدن را بصورت سیاه و سفید (خاکستری) می خوانیم. سپس تابع `imgSupplement(img)` را تعریف می کنیم که یک تصویر را بعنوان ورودی می گیرد و برای خروجی با حلقه مقدار مکمل هر پیکسل را با توجه به اینکه از تفاضل ۲۵۵ و مقدار پیکسل تصویر ورودی محاسبه می شود، در پیکسل متناظر تصویری که خروجی می دهد، قرار می دهد.

سپس این تابع را بر روی تصویر ورودی اعمال کردیم تا تصویر مدنظر سوال (تصویر مکمل) را ایجاد کنیم. همچنین برای ایجاد تصویر اجتماع بدیهی است که باید هر دو تصویر اصلی و مکمل را که ایجاد کردیم، با هم جمع کنیم.

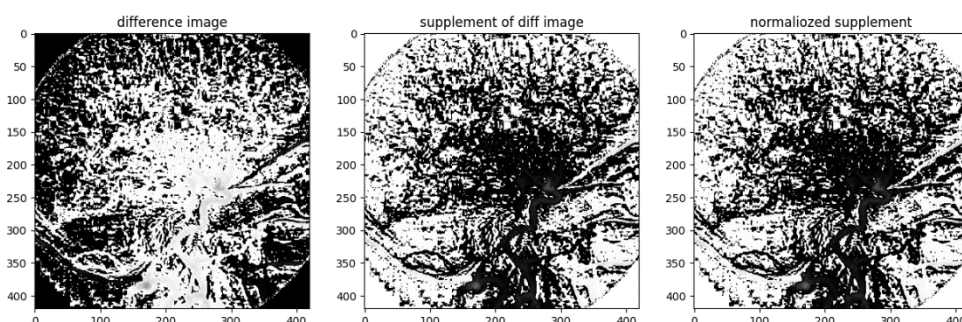
نهایتاً نمایش تصاویر کنار هم بصورت زیر شد:



بحث: مکانیزم مکمل یابی نقاط تیره را روشن و روشن را تاریک می کند. براحتی این اتفاق را می توان در تصاویر دید. همچنین اجماع منجر به آرایه ای با اعضای کامل ۲۵۵ می شود و این یعنی تصویر یکدست سفید. توجه داریم تصویر سوم با `cmap = "gray_r"` نمایش داده می شود زیرا در غیر این صورت با اینکه تمام پیکسل ها مقدار ۲۵۵ دارند، توسط `cmap = "gray"` بصورت سیاه نشان داده می شود!

سوال دوم: ج)

طبق خواست سوال، تصاویر مربوط به آنژیوگرافی خوانده شد. سپس اختلاف آن‌ها بسادگی محاسبه شد. سپس با تابع `imgSupplement(angiography_diff)` که برای بخشی قبلی تعریف شد، در ورودی تابع تصویر اختلاف داده می‌شود تا مکمل آن محاسبه شود. سپس با تابع `cv2.normalize()` تصویر مکمل، سعی شد تا توزیع هیستوگرام با متد `cv2.NORM_MINMAX` بهتر شود. حال سه عملیات فوق را کنار هم نمایش می‌دهیم:



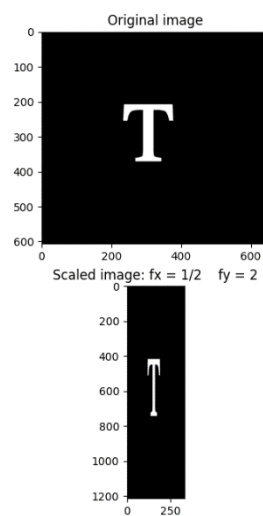
تحلیل: توجه داریم که کم کردن دو تصویر از هم در این خصوص، تغییرات تصویر را به ما نشان می‌دهد. اما باید توجه کنیم که در تصویر اختلاف، پیکسل‌های غیرمشکی بیانگر تغییر و مشکی‌ها همچون زمینه (خارج از دایره) بیانگر عدم تغییر هستند. بهمین منظور، تلاش می‌شود تا مکمل آن استخراج شود. در تصویر مکمل مشکی بیانگر چیزهایی است که قبلاً وجود نداشته و تازه پدید آمده‌اند. با توجه به ظاهر، بنظر می‌تواند رگ‌های خونی روی قلب باشند که خون در آن‌ها جریان کرده است. همچنین در نرمال کردن قصد توزیع خطی و استفاده بیشتر از تمام بیت‌های موجود است (۰ تا ۲۵۵) که در این تصویر چون هم روی ۰ و هم روی ۲۵۵ پیکسل داریم (در توزیع هیستوگرام)، عملاً تفاوتی بین تصویر دوم و سوم مشهود نیست. البته می‌توان حال که از ۰-۲۵۵ استفاده شده است، از بازه‌های کوچکتر نیز استفاده کرد که منطقی نیست. یا اینکه از توزیع یکنواختی نیز می‌توان بهره برد.

سوال سوم)

- Scaling

برای اسکیلینگ از تابع `cv2.resize(T, None, fx = 1/2 , fy = 2)` استفاده شده است. در این تابع ورودی تصویر مبدا و ابعاد یا ضرایب اسکیل شدن در راستای هر دو محور است. همچنین نوع درون یابی را نیز می توان تعیین کرد که در اینجا از نوع پیشفرض یعنی خطی استفاده شده است.

توجه داریم در این تبدیل ما در راستای محور افقی تصویر را به نصف فشرده و در راستای محور عمودی آن را دو برابر بزرگ کرده ایم. خروجی تصویر نمایش داده شده به شکل زیر است:



سوال سوم

• Translation

برای انجام عملیات امکان استفاده از تابع آماده داشتیم، اما ترجیح داده شد تا یک تابع برای پیاده سازی این تبدیل هندسی نوشته شود. برای این کار، تابع `imgTrans(img, dx, dy)` نوشته شد که در ورودی تصویر مبدأ، میزان جابجایی در راستای محور عمودی و نهایتاً میزان جابجایی در راستای محور افقی گرفته می‌شود و بعنوان خروجی تصویر انتقال یافته بازگشت داده می‌شود.

تابع تعریف شده ابتدا یک تصویر هم اندازه با تصویر ورودی با پیکسل‌های مشکی (مقدار ۰) ایجاد می‌کند. سپس با ماتریس تبدیل همگن `mat = np.array([[1, 0, dx], [0, 1, dy], [0, 0, 1]])` موقعیت پیکسل‌های تصویر خروجی که از نظر شدتی باید متناظر با پیکسل تصویر ورودی باشد (تا تصویر انتقال یافته ساخته شود) را توسط حلقه پیدا می‌کند. بعبارتی اگر تصویر خروجی پیکسل‌های (p, q) را دارد و تصویر ورودی پیکسل‌های (m, n) را دارد، رابطه‌ی زیر برای موقعیت یابی شدت‌ها در تصویر خروجی اعمال می‌شود:

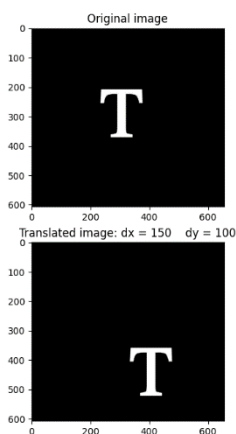
```
p = np.dot(mat[0, :], np.array([m, n, 1]))
q = np.dot(mat[1, :], np.array([m, n, 1]))
```

که طبق تئوری تبدیل همگن است:

$$(p, q) = T\{(m, n)\} \quad \text{Translation} \quad \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{array}{c} \rightarrow \\ T \end{array}$$

البته یک شرط در تعریف تابع قرار داده شده است که مختصات (p, q) باید درون چارچوب تصویر تعریف شده قرار بگیرد، در غیر این صورت مقدار آن همان ۰ (مشکی) باقی می‌ماند.

سپس برای تصویر خواسته شده یک انتقال در راستای محور عمودی با ۱۵۰ پیکسل و محور افقی با ۱۰۰ پیکسل اعمال کردیم و تصویر را نمایش دادیم. تصویر به شکل زیر شد:



سوال سوم

• Horizontal Shear

برای انجام عملیات امکان استفاده از تابع آماده داشتیم، اما ترجیح داده شد تا یک تابع برای پیاده سازی این تبدیل هندسی نوشته شود. برای این کار، تابع `imgHShear(img, s_h)` نوشته شد که در ورودی تصویر مبدا و پارامتر تنظیم شدت برش گرفته می شود و بعنوان خروجی تصویر برش یافته بازگشت داده می شود.

تابع تعریف شده ابتدا یک تصویر هم اندازه با تصویر ورودی با پیکسل های مشکی (مقدار ۰) ایجاد می کند. سپس با ماتریس تبدیل همگن `mat = np.array([[1, 0, 0], [s_h, 1, 0], [0, 0, 1]])` پیکسل های تصویر خروجی که از نظر شدتی باید متناظر با پیکسل تصویر ورودی باشد (تا تصویر برش یافته ساخته شود) را توسط حلقه پیدا می کند. عبارتی اگر تصویر خروجی پیکسل های (p, q) را دارد و تصویر ورودی پیکسل های (m, n) را دارد، رابطه ی زیر برای موقعیت یابی شدت ها در تصویر خروجی اعمال می شود:

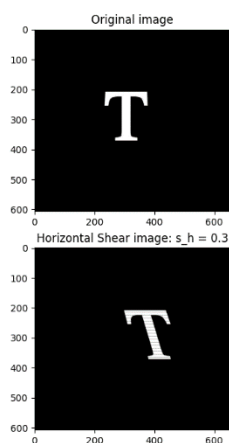
```
p = np.dot(mat[0, :], np.array([m, n, 1]))
q = np.dot(mat[1, :], np.array([m, n, 1]))
```

که طبق تئوری تبدیل همگن است:

$$(p, q) = T\{(m, n)\} \quad \text{Shear (horizontal)} \quad \begin{bmatrix} 1 & 0 & 0 \\ s_h & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \mathcal{T} \rightarrow$$

البته یک شرط در تعریف تابع قرار داده شده است که مختصات (p, q) باید درون چارچوب تصویر تعریف شده قرار بگیرد، در غیر این صورت مقدار آن همان ۰ (مشکی) باقی می ماند. همچنین با توجه به اینکه (p, q) می توانند غیر صحیح شوند، مقدار صحیح آن ها استفاده شده است که یکی از مشکلات تبدیل فروارد است.

سپس برای تصویر خواسته شده یک برش با مقدار پارامتر 0.3 اعمال کردیم و تصویر را نمایش دادیم. تصویر به شکل زیر شد:



سوال سوم)

- Vertical Shear

برای انجام عملیات امکان استفاده از تابع آماده داشتیم، اما ترجیح داده شد تا یک تابع برای پیاده سازی این تبدیل هندسی نوشته شود. برای این کار، تابع `imgvShear(img, s_v)` نوشته شد که در ورودی تصویر مبدا و پارامتر تنظیم شدت برش گرفته می شود و بعنوان خروجی تصویر برش یافته بازگشت داده می شود.

تابع تعریف شده ابتدا یک تصویر هم اندازه با تصویر ورودی با پیکسل‌های مشکی (مقدار ۰) ایجاد می‌کند. سپس با ماتریس تبدیل همگن `mat = np.array([[1,s_v,0],[0,1,0],[0,0,1]])` پیکسل‌های تصویر خروجی که از نظر شدتی باید متناظر با پیکسل تصویر ورودی باشد (تا تصویر برش یافته ساخته شود) را توسط حلقه پیدا می‌کند. عبارتی اگر تصویر خروجی پیکسل‌های (p, q) را دارد و تصویر ورودی پیکسل‌های (m, n) را دارد، رابطه‌ی زیر برای موقعیت پایی شدت‌ها در تصویر خروجی اعمال می‌شود:

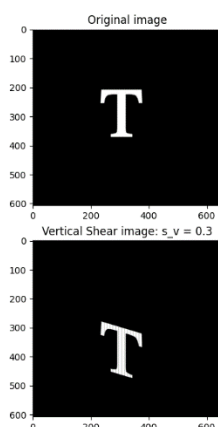
```
p = np.dot(mat[0, :], np.array([m, n, 1]))
q = np.dot(mat[1, :], np.array([m, n, 1]))
```

که طبق تئوری تبدیل همگن است:

$$(p, q) = T\{(m, n)\} \quad \text{Shear (vertical)} \quad \begin{bmatrix} 1 & s_v & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{array}{c} \text{I} \\ \downarrow \end{array}$$

البته یک شرط در تعریف تابع قرار داده شده است که مختصات (p, q) باید درون چارچوب تصویر تعریف شده قرار بگیرد، در غیر این صورت مقدار آن همان \cdot (مشکی) باقی می‌ماند. همچنین با توجه به اینکه (p, q) می‌توانند غیر صحیح شوند، مقدار صحیح آن‌ها استفاده شده است که یکی از مشکلات تبدیل فروارد است.

سپس برای تصویر خواسته شده یک برش با مقدار پارامتر 0.3 اعمال کردیم و تصویر را نمایش دادیم. تصویر به شکل زیر شد:




سوال سوم

• Forward and Inverse Rotation

برای این کار، تابع `imgRot(img, theta, operationType)` نوشته شد که در ورودی تصویر مبدأ، میزان دوران به رادیان و پارامتر تنظیم نوع عملیات که می‌تواند مستقیم `operationType == "Forward"` یا معکوس `operationType == "Inverse"` باشد، بعنوان ورودی گرفته می‌شود. در خروجی نیز تصویر دوران یافته تحت نوع عملیاتی که در ورودی تنظیم شده است، داده می‌شود.

در عملیات مستقیم بر اساس تبدیل هندسی دوران به ازای تک تک پیکسل‌های تصویر ورودی، موقعیت متناظر پیکسل‌های تصویر خروجی پیدا می‌شود. اما در عملیات معکوس، به ازای موقعیت تک تک پیکسل‌های تصویر خروجی، پیکسل متناظر از نظر مقدار شدت در تصویر ورودی یافته می‌شود.

طبق تئوری، برای تبدیل مستقیم داریم:

$$(p, q) = T\{(m, n)\} \quad \text{Rotation} \quad \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$


که ماتریس دوران به شکل زیر پیدا می‌شود:

```
mat = np.array([[np.cos(theta),
np.sin(theta), 0], [np.sin(theta), np.cos(theta), 0], [0, 0, 1]])
```

و پیکسل خروجی متناظر با پیکسل تصویر ورودی به شکل زیر پیدا می‌شود: (ماتریس، ماتریس دوران است).

```
p = int(round(np.dot(mat[0, :], np.array([m, n, 1]))))
q = int(round(np.dot(mat[1, :], np.array([m, n, 1]))))
```

همچنین طبق تئوری برای تبدیل معکوس داریم:

$$(m, n) = T^{-1}\{(p, q)\}$$

کتابخانه numpy پیدا می‌شود. برای پیکسل‌های متناظر با پیکسل‌های تصویر خروجی در تصویر ورودی داریم:

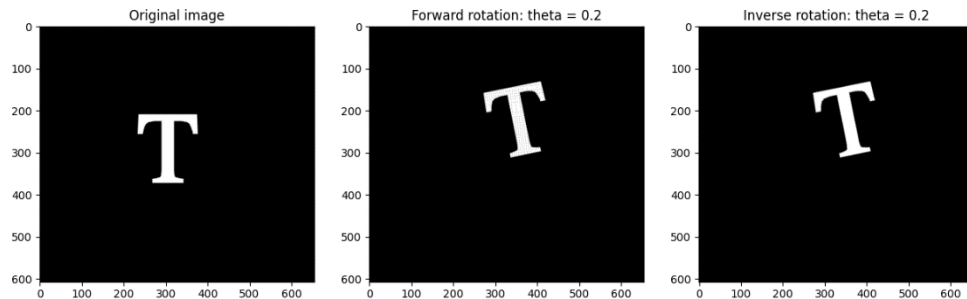
```
m = int(round(np.dot(mat[0, :], np.array([p, q, 1]))))
n = int(round(np.dot(mat[1, :], np.array([p, q, 1]))))
```

طبق توضیح برای پیدا کردن ماتریس دوران داریم:

```
mat = np.linalg.inv(mat)
```

همچنین در برنامه یک شرط قرار داده شده است که چنانچه موقعیت پیکسل‌ها درون قاب تعریف شده در ابتدا قرار گرفت، عملیات نگاشت اعمال شود.

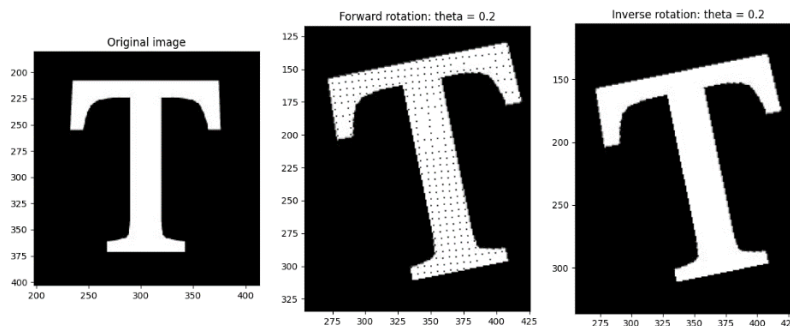
در ادامه نتایج اعمال تبدیل‌های دوران مستقیم و معکوس بر روی تصاویر را داریم:



تحت دوران با مقدار 0.2 رادیان خروجی‌های تابع تعریف شده با متدهای مستقیم و معکوس مشخص است.

تفاوت دو روش دوران مستقیم و معکوس:

قبل از بحث، مقداری روی تصویر خروجی زوم می‌کنیم:



به وضوح در خروجی با روش مستقیم، وجود نقاط سیاه در مکان‌هایی که در تصویر اصلی کاملاً سفید بوده‌اند، مشخص است. دلیل نیز واضح است. در این تبدیل برخی نقاط تصویر ورودی به خاطر ویژگی گسستگی تصویر به یک نقطه در خروجی ورودی نگاشت می‌شوند و برخی نقاط تصویر خروجی عملاً تحت نگاشت قرار نمی‌گیرند. اما در روش معکوس چون نگاشت روی تمام پیکسل‌های تصویر خروجی صورت می‌گیرد، هیچ نقطه‌ای از آن بدون پوشش باقی نمی‌ماند. پس در دوران، روش معکوس به روش مستقیم برتری دارد زیرا خروجی دقیق و کامل‌تری ارائه می‌دهد.