اصول پردازش تصاویر

نیم‌سال اول ۱۴۰۰-۱۴۰۱

مدرس: دکتر مصطفی کمالی تبریزی

دانشکده علوم ریاضی

گزارش تمرین سری دوم

نام و نام‌خانوادگی: -

شماره دانشجویی: -

در حل این سوالات از زبان پایتون استفاده شده است که درون فایل های $q1.py$ تا $q4.py$ ذخیره شده اند. نتایج نیز از $res1.jpg$ تا $res31.jpg$ ذخیره شده اند. در حل این سوالات از هیچ فردی مشورتی هرچند جزئی گرفته نشد.

پرسش ۱ - Sharpening

در سوال ۱، ۴ روش مختلف شارپ کردن تصویر استفاده شد. در ابتدای کد بصورت معمول تصویر داده شده از فایلی که با کد تمرین در یک مسیر قرار دارد دریافت می‌شود. سپس با دستور astype ، دیتاتایپ آن را به int تغییر می‌دهیم و تصویری که بصورت int در آمده را در int_image ذخیره می‌کنیم. سپس به ترتیب وارد ۴ تابع میشویم که کلیات شارپ کردن به هر روش در هریک از این ۴ تابع قرار دارد. پس به تفصیل هرکدام از آنها را توضیح می‌دهیم.

۱. با استفاده از فیلتر گاوس : در ۴ مرحله ، فیلتر گاوس را می‌سازیم و با کانوالو کردن با عکس اصلی Blur شده ی تصویر اصلی را بدست می‌اوریم. سپس آنرا از تصویر اصلی کم کرده و ماسک آنشارپ بدست می‌اید و در نهایت با جمع ماسک آنشارپ و تصویر، تصویر نهایی شارپ شده را بدست می‌آوریم.

این پروسه در تابع gaussian_sharpening درون کد پاسخ قرار دارد.

- ساخت کرنل گاوسی: ابتدا اندازه ی این کرنل تعیین می‌شود. در اینجا این اندازه برابر ۱۳ و سیگمای آن نیز برابر ۲ است. یعنی کرنل نهایی گاوسی اندازه ای برابر 13×13 و $\sigma = 2$ خواهد داشت. سپس تابع get_gaussian_kernel را صدا می‌زنیم. این تابع دو آرگومان سایز و سیگما را دریافت می‌کند و ماتریس مورد نظر را به ما خروجی می‌دهد. روش کار این تابع به اینصورت است که ابتدا یک ماتریس تماما صفر با ابعاد داده شده ساخته می‌شود. دو حلقه از منفی رند شده ی نصف اندازه ی داده شده تا نصف اندازه داده شده اعمال می‌کند و نتیجه آنرا در ماتریس تماما صفر اولیه قرار می‌دهد. برای محاسبه هرکدام از درایه های از رابطه زیر استفاده شده است:

$$G_\sigma(x,y) = \frac{1}{2\pi\sigma^2}\, e^{-\frac{x^2+y^2}{2\sigma^2}}$$

و در نهایت نیز برای اینکه جمع کل درایه ها برابر ۱ شود، آنرا بر مجموع درایه هایش تقسیم می‌کنیم و نتیجه را به عنوان خروجی تابع خارج می‌کنیم. به این صورت کرنل گاوسی بدست آمده در gauss_kernel ذخیره می‌گردد. سپس کرنل گاوسی توسط تابع normalize بین مقادیر ۰ تا ۲۵۵ نرمالایز می‌شود. این تابع با استفاده از یک تابع خطی به راحتی کل مقادیر را بین ۰ تا ۲۵۵ نرمالایز می‌کند. یعنی حداقل مقدار تابع را برابر ۰ و حداکثر آنرا برابر ۲۵۵ قرار می‌دهد و به طور خطی بقیه ی مقادیر را نیز بین این دو بازه تقسیم می‌کند. به اینصورت مقادیر کرنل که همگی مقدار کمتری از ۱ دارند، تا حدی زیاد می‌شوند که بتوان به طور روشن فرم کلی این فیلتر را دید. در نهایت gauss_kernel را به تابع resize با ضریب $int(\frac{500}{kernel\_size})$ بزرگ می‌کنیم که به راحتی قابل نمایش باشد.

تابع resize با دریافت یک تصویر و یک مقدار، تصویر بزرگ شده را که به مقدار دریافتی بزرگ شده است بر می‌گرداند. برای مثال با مقدار ۱ تصویر دست نخورده باقی می‌ماند و با مقدار ۲، دو برابر و با ۰.۵ ، نصف مقدار اولیه را خروجی می‌دهد. روش کار این تابع بسیار آسان است و از تابع cv۲.resize استفاده می‌کند. در نهایت نیز خروجی این تابع در فایل گفته شده ذخیره می‌گردد. (توجه شود که دو تابع اعمال شده به مقادیر اصلی gauss_kernel کاری ندارند و صرفا برای نشان دادن آن استفاده می‌شوند. یعنی مقدار خود فیلتر برای مراحل بعدی ثابت می‌ماند)

- کانوالو کردن فیلتر گاوسی با عکس و ساختن یک تصویر Blur شده: این کار به راحتی با تابع cv۲.filter۲D انجام می‌گیرد. به این تابع تصویر ورودی، فیلتر و همچنین مقدار $ddepht = -1$ را می‌دهیم. سپس خروجی را در gauss_convolved ذخیره می‌کنیم. و در خط بعد آن را در فایل گفته شده ذخیره می‌کنیم.

- ساخت آنشارپ مسک از اختلاف تصویر اولیه و تصویر Blur شده: به راحتی با استفاده از np.substract تصویر اولیه که بصورت int در آمده را درایه به درایه منهای تصویر Blur شده ی بدست آمده را وارد تابع normalize می‌کنیم که پیشتر توضیح داده شد. به همین خاطر رنگ کلی تصویر خاکستری است. زیرا مقدار حول ۰ که در تصویر آنشارپ مسک حداکثر است، به میانه ی بازه روشنایی می‌رود و نقاط منفی تیره تر و نقاط مثبت روشن تر. خروجی تابع normalize را در فایل گفته شده ذخیره می‌کنیم اما از آن تنها برای نمایش استفاده می‌کنیم و برای استفاده در قسمت بعد همان آنشارپ مسک اصلی که مقادیر بسیار کوچک منفی و مثبت را دارد در نظر می‌گیریم.

- در نهایت از رابطه ی گفته شده $f + \alpha(f - f * g)$ سعی می‌کنیم نتیجه شارپ شده در این بخش را بدست بیاوریم. ابتدا آلفا که در صورت سوال نیز آمده است را تعریف می‌کنیم. در اینجا $\alpha = 3$ قرار می‌دهیم. سپس به صورت گفته شده با دستور np.add تصویر اصلی بصورت int را با آنشارپ مسک حاصل شده که مقادیرش در α جمع می‌کنیم. نتیجه را نیز به تابع clipping می‌دهیم. این تابع کل مقادیر ورودی را به ۰ تا ۲۵۵ محدود می‌کند و مقادیر منفی را برابر با ۰ و مقادیر بزرگتر از ۲۵۵ را برابر با ۲۵۵ قرار می‌دهد و دیتاتایپ را به $uint8$ در نهایت خروجی نهایی را در gauss_final ذخیره می‌کنیم و در فایل گفته شده به عنوان خروجی ذخیره می‌کنیم و به اینصورت تابع gaussian_sharpening تمام می‌شود.

۲. با استفاده از فیلتر لاپلاسین گاوس: در این بخش در ۳ مرحله تصویر شارپ شده مطابق با صورت سوال را میسازیم. ابتدا فیلتر لاپلاسین گاوس را میسازیم و نمایس میدهیم، سپس با کانوالو آن با تصویر، ماسک آنشارپ و در نهایت آنرا از تصویر کم کرده و تصویر نهایی را بدست میاوریم.

تابع این بخش در کد سوال، laplacian_sharpening نام دارد.

- دقیقا مشابه بخش قبلی برای ساخت کرنل گاوس عمل می‌کنیم. به اینصورت که اندازه کرنل در خط بعد 13×13 تعیین شده و سیگما نیز بصورت $\sigma = 1$ در نظر گرفته می‌شود. درون تابع همان ساختار برای ساخت تابع لاپلاسین gauss را داریم:

$$\nabla^2 G_\sigma(x, y) = \frac{x^2 + y^2 - 2\sigma^2}{2\pi\sigma^6} \, e^{-\frac{x^2+y^2}{2\sigma^2}}$$

فلذا در تابع get_laplacian_of_gaussian_kernel یک آرایه ی تماما ۰ به ابعاد داده شده در ورودی تابع ساخته می‌شود و در دو عدد حلقه رابطه ی بالا برای هر کدام از درایه ها اعمال می‌شود. در اینجا دیگر مجموع درایه ها نرمالایز نمی‌شود. در اینجا در مرکز کرنل مقدار منفی و در اطراف زیاد شود و به مثبت برسد و باز آنقدر کم شود که به ۰ برسد. در اینجا می‌توانیم حدس بزنیم که قرار است شاهد رفتاری معکوس برای تصویر

ماسک آنشارپ شاهد باشیم.

بعد از بدست آوردن کرنل لاپلاسین فوق، آنرا درون laplacian_kernel قرار می‌دهیم. تفاوت این بخش با بخش قبلی، داشتن مقادیر مثبت و منفی در کرنل فیلتر ساخته شده می‌باشد. به این منظور لازم است حداقل از دو رنگ برای نمایش فیلتر استفاده کنیم. روش کار به این صورت است. آرایه فیلتر laplacian_kernel را در show_laplace۱ می‌ریزیم. سپس با استفاده از روشی ساده، مقادیر منفی show_laplace۱ را برابر با ۰ قرار داده و تنها مقادیر مثبت را در اختیار داریم. سپس آنها را بین ۰ تا ۲۵۵ نرمالایز میکنیم و بصورت int برای show_laplace۲ کاری مشابه انجام می‌دهیم. تنها با این تفاوت که در ابتدا، کل درایه هایش را در -۱ ضرب می‌کنیم. به اینصورت مقادیر منفی مثبت میروند و بالعکس. با انجام پروسه ی بالا آرایه ای از قدر مطلق مقادیر منفی laplacian_kernel در اختیار خواهیم داشت.

سپس با استفاده از تابع cv۲.merge و قرار دادن show_laplace۱ و show_laplace۲ برای کانال های آبی (برای مقادیر مثبت) و قرمز (برای مقادیر منفی) و همچنین ساختن آرایه ای تمام ۰ با استفاده از np.zeros و قرار دادن آن به عنوان کانال سبز، final_laplacian_filter را بدست می آوریم. در پایان نیز با استفاده از تابع resize و ابعاد مشابه قسمت قبل، تصویر نسبتا بزرگی از فیلتر استفاده شده را در حافظه ذخیره می‌کنیم. (در نتیجه خروجی مقادیر منفی قرمز هستند و نتایج مثبت آبی. توجه شود به دلیل الگوریتم فشرده سازی jpg و به هم خوردن پیکسل های کناری هم منطقه ، لازم است تنها این بخش سوال بصورت png ذخیره شود که شکل درست فیلتر اعمالی را مشاهده کنیم)

- ساخت ماسک آنشارپ با استفاده از فیلتر لاپلاسین گاوس: در اینجا به سادگی با استفاده از کانوالو کردن عکس اولیه با دیتاتایپ int' و فیلتر لاپلاسین گاوس بدست آمده در مرحله قبل، $(f * \Delta g)$ را بدست می‌اوریم و همانند قبل، لازم است ابتدا نتیجه را با تابع normalize ، بین ۰ تا ۲۵۵ نرمال کنیم و نتیجه نرمال شده را نمایش بدهیم. (توجه شود از آنجا که این ماسک آنشارپ حدودا باید منهای ماسک آنشارپ در قسمت پیش باشد، توقع می‌رود جاهایی که در قسمت قبل روشن تر از خاکستری بود اینجا تیره تر باشد و بالعکس)

- قسمت نهایی این بخش نیز عینا مانند بخش پیشین است بجز اینکه در اینکه بجای جمع کردن مقدار f با ماسک آنشارپ، باید ماسک آنشارپ را از عکس اصلی کم کنیم که این ریشه در منفی بودن نتیجه ی ماسک آنشارپ دومی و اولی دارد . در اینجا κ را برابر ۵ در نظر گرفتیم که مقدار مناسبی بود. سپس با استفاده از تابع np.subtract و np.multiply ، عبارت $f - \kappa f * \Delta g$ را محاسبه کردیم. (دقت کنید که با توجه به محدودیت $uint$ برای مقادیر خارج از ۰ و ۲۵۵ باید این محاسبات بر پایه ی int انجام شود و در نهایت برای نمایش تبدیل به $uint8$ بشود. به همین دلیل از مقدار int فیلتر لاپلاسین نیز استفاده شده است.)

سپس مقدار بدست آمده با استفاده از تابع clipping بین ۰ تا ۲۵۵ محدود شده و در laplacian_final ذخیره شده است. در انتهای این روش نیز تصویر laplacian_final بصورت دیتاتایپ $uint8$ درآمده و ذخیره شده است. (البته با توجه به کارکرد تابع clipping نیازی به انجام این تغییر دیتاتایپ در عمل نبود و تنها برای استاندارد کردن اینکار صورت گرفت)

۳. با استفاده از تبدیل فوریه و فیلتر بالاگذر: برای این روش ابتدا تبدیل فوریه روی تابع اعمال کرده، فیلتر بالاگذر مناسبی می‌سازیم، آنرا در حوزه ی فرکانس برروی تصویر اعمال کرده و نتیجه را توسط معکوس تبدیل فوریه به حوزه مکان برمی‌گردانیم.

این پروسه در تابع fourier_highpass_sharpening درون کد پاسخ سوال قرار دارد.

- در ابتدا با استفاده از تابع fourier_transform_function تبدیل فوریه تصویر را می‌گیریم و در fourier_image ذخیره می‌کنیم. در این تابع از دو تابع بر اساس numpy استفاده می‌کنیم. اولی np.fft.fft۲ تصویر دریافتی

را در جهت های x و y تبدیل به تصویر تابع فوریه می‌کند. تابع بعدی (np.fft.fft۲) همانطور که در کلاس گفته شد، به گونه ی تصویر را تغییر می‌دهد که تمام فرکانس های پایینتر که در ۴ گوشه قرار داشتند، در مرکز قرار بگیرند و با دور شدن از مرکز فرکانس زیاد بشود.

بعد از آنکه نتیجه در fourier_image گرفت، برای نمایش بزرگی آن، از تابع show_magnitude_fourier استفاده می‌کنیم. این تابع تصویر را دریافت می‌کند، و پس از آنکه قدر مطلق مقادیر حقیقی اش را با ۱ جمع و لگاریتم آنرا حساب می‌کند، در n که ورودی تابع است و باید مقدار معقولی باشد ضرب کرده و این را با تابع normalize بین ۰ تا ۲۵۵ محدود و پخش کرده و در نهایت به فرمت $uint8$ آن را برمی‌گردانیم و نتیجه را مستقیما ذخیره می‌کنیم.

- در این مرحله برای ساخت تصویر فیلتر بالاگذر از تابع get_highpass_filter استفاده می‌کنیم. مقادیر ورودی این تابع شعاع، سایز کرنل و همچنین سیگما است. در این تابع یک تصویر که درونش دایره‌ای سیاه که مرکزش، مرکز تصویر است تولید می‌کنیم و با تابعی که در بخش ۱ گفته شد، یک فیلتر گاوسی با اندازه و سیگمای داده شده به تابع ساخته می‌شود. نتیجه ی کانوالو تصویر دایره سیاه با فیلتر گاوسی گفته شده، خروجی این تابع است. به این صورت فیلتر بالاگذر ساخته شده می‌شود و در قالب $uint8$ ذخیره می‌شود.
- در قسمت بعد، از طریق رابطه زیر، فیلتر بالاگذر مورد نظر را اعمال می‌کنیم:

$$(1 + \kappa H_{HP}).F$$

ابتدا κ در رابطه بالا را برابر 0.02 در نظر می‌گیریم. سپس با استفاده از دستورات ساده ی ضرب و جمع درایه به درایه ماتریسی، نتیجه را در fourier_highpass_filtered ذخیره می‌کنیم. سپس مانند قبل با استفاده از تابع show_magnitude_fourier بزرگی تصویر حاصل را نمایش می‌دهیم.

- حال باید رابطه زیر را برای تصویر محاسبه کنیم:

$$F^1\{(1 + \kappa H_{HP}).F\}$$

تصویر در حوزه ی فرکانسی را که حاصل شده بود، با تابع inverse_fourier_transform_function به حوزه مکان برمی‌گردانیم. این تابع تصویر حوزه فرکانس را دریافت می‌کند و پس از اعمال معکوس تبدیل های فوریه ای که در تابع fourier_transform_function انجام دادیم، مقدار قدر مطلق جز حقیقی تصویر را توسط تابع clipping بین ۰ تا ۲۵۵ محدود کرده و خروجی آن را برمی‌گردانیم. در پایان کار هم تصویر بدست آمده را در final_fourier_highpass_filter می‌ریزیم و سپس آنرا ذخیره می‌کنیم. به این صورت تصویر شارپ شده از طریق فیلتر بالاگذر ساخته می‌شود.

۴. با استفاده از لاپلاسین تصویر در حوزه فرکانس: این کار در ۳ مرحله انجام می‌شود. مرحله اول، با استفاده از تابع گفته شده در صورت سوال، فیلتر لاپلاسین را اعمال می‌کنیم، سپس آنرا به حوزه مکان برده و ماسک آنشارپ را تشکیل می‌دهیم؛ و در نهایت تصویر اصلی را با ضریب κ با ماسک آنشارپ بدست آمده جمع می‌کنیم و این نتیجه نهایی‌مان خواهد بود.

- در ابتدا عینا مانند قسمت قبل، بر روی تصویر در حوزه مکان تبدیل فوریه اعمال می‌کنیم و آنرا در image_fourier ذخیره می‌کنیم. در اینجا تصویر image_fourier را به تابع uv_main_transform می‌دهیم. در این تابع ابتدا یک تصویر با ابعاد یکسان با تصویر image_fourier با درایه های تماما صفر ایجاد می‌کنیم. با استفاده از دو حلقه که به ترتیب در عرض و طول تصویر پیمایش می‌کنند، تصویر جدید بصورت

$$4\pi^2(u^2 + v^2)$$

ایجاد می‌شود که u و v عرض و طول از مرکز هستند. یعنی عبارت (u^2+v^2) در داخل دو حلقه اعمال می‌شود و در نهایت نتیجه در $4\pi^2$ ضرب می‌گردد و به عنوان خروجی تابع برمی‌گردد. خروجی در reduced_uv_fourier_image

ذخیره می‌گردد. سپس با دادن این تصویر به تابع show_magnitude_fourier می‌توانیم بزرگی reduced_uv_fourier_image را بدست بیاوریم و در فایل گفته شده ذخیره کنیم.

- در مرحله بعدی، لازم است مانند قبل حاصل مرحله قبل را حوزه مکان برگردانیم. در اینجا برخلاف سایر دفعات نباید هیچگونه نرمالایز و کلیپ و همچنین از قدر مطلقی استفاده کنیم زیرا تصویر ماسک آنشارپ مقدار منفی و مثبت خارج از ۰ تا ۲۵۵ دارد و نباید این مقادیر را تغییر بدهیم. پس یک تابع جدید مشابه قبل می‌نویسیم که در پایان آن جایی که می‌خواهیم تصویر حوزه مکان حاصل را برگردانیم تنها یک $real$. قرار می‌دهیم که مقادیر حقیقی را به ما برگرداند.

  نام این تابع را non_normalized_inverse_fourier_transform_function می‌گذاریم و به ورودی آن reduced_uv_fourier_image می‌دهیم. خروجی تابع را برابر inverted_fourier_uv قرار می‌دهیم و بدون اینکه به مقادیر آن دست بزنیم، تابع normalize را بر روی آن پیاده سازی می‌کنیم و نتیجه را به صورتی که گفته شده است ذخیره می‌کنیم.

- در پایان نیز با استفاده از ضرب ضریب κ در ماسک آنشارپ بدست آمده و جمع آن با تصویر اولیه، تصویر شارپ شده را بدست می‌آوریم:

$$f + \kappa F^{-1}\{4\pi^2(u^2+v^2)F\}$$

  برای این کار ابتدا $\kappa = 2.1 \times 10^{-6}$ را تعریف می‌کنیم. سپس به صورتی که بالاتر گفته شد با استفاده از np.add و np.multiply محاسبه فوق را انجام می‌دهیم. سپس نتیجه کلی را بین ۰ تا ۲۵۵ کلیپ می‌کنیم و در fourier_uv_final ذخیره می‌کنیم. در پایان هم fourier_uv_final را با دیتاتایپ $uint8$ ذخیره می‌کنیم.

قسمت سوال	α	κ	σ
الف	3	-	2
ب	-	5	1
ج	-	2×10^{-2}	-
د	-	2.1×10^{-6}	-

جدول ۱: مقادیر ثابت استفاده شده در سوال ۱

در این سوال شارپ کردن تصاویر به ۴ روش گفته شده انجام شد و مراحل و توابع استفاده شده در کد پاسخ سوال نیز بطور کامل شرح داده شد. در نهایت هم جدول مقادیری که استفاده شد عنوان شد. برای بدست آوردن این مقادیر از آزمون و خطا استفاده شد که بهترین نتایج ممکن بدست بیاید، نه به قدری شارپ شود و از حالت طبیعی خود خارج بشود، نه تار باقی بماند. اگر مقادیر κ و α خیلی زیاد شوند، از آنجا که نتایج بین ۰ تا ۲۵۵ کلیپ شده‌اند، گوشه های اجسام به سفیدی میل می‌کند و این اتفاق خوشایندی نیست. بنابر این این مقادیر توانستند نتیجه ی خوبی را بدهند

پرسش ۲ - Template Matching

در شروع کد پاسخ سوال، ابتدا تصاویر کشتی یونانی بصورت رنگی و تصویر ستون (patch) در دریا بصورت سیاه و سفید وارد برنامه شده و در متغیر های ship و template ذخیره می‌گردند. سپس متغیری بنام scale_rate با مقدار پیش‌فرض ۱.۰ تعریف می‌شود. این به این معنی هست که تمپلیت مچینگ در مقیاس ۱.۰ حالت عادی انجام می‌گیرند. این تغییر در خطوط بعدی اعمال می‌شود. در دو خط بعدی ابتدا تصویر ،ship بصورت سیاه و سفید در می‌اید و تا ابعاد scale_rate اولیه کوچک می‌شود. سپس بصورت 'int' در image ذخیره می‌شود. سپس template نیز ابتدا وارد تابع cut_template می‌شود و سپس به همان اندازه scale_rate کوچک شده و سیاه و سفید می‌شود و بصورت 'int' در خودش ذخیره می‌شود. این کار ها به سه دلیل انجام می‌شود

۱. با اینکار ها محاسبات بسیار سریعتر انجام می‌گیرند، با کوچک کردن تصویر، پیکسل های مورد محاسبه حدود یک صدم حالت قبل می‌شود. با سیاه و سفید کردن هم یک سوم، پس انتظار می‌رود سرعت محاسبه تا ۳۰۰ برابر بالاتر برود.

۲. با پایین آوردن کیفیت، مقدار خطا بالاتر می‌رود. این در ظاهر می‌تواند خوب نباشد اما در عمل، در این سوال باعث می‌شود احتمال پیدا کردن قطعه های نزدیک به تمپلیت بالا می‌رود. و این دقیقا همان چیزی است که در صورت سوال عنوان شده است.

۳. با مقایسه ی سیاه سفید هم احتمال خطا بالا می‌رود و مشابه قبل احتمال پیدا کردن سوژه های مشابه بالاتر می‌رود و تنها محدود به سوژه با طیف رنگی یکسان با تمپلیت نیستیم

تابع cut_template که بر روی template اعمال می‌شود، هدفش به دست آوردن مقدار مناسبی از ستون درون عکس هست که کارایی در پیدا کردن سوژه های درون تصویر اصلی به حداکثر برسد. کاری که در این قسمت انجام می‌شود، کاملا مشابه برش حاشیه های تصویر در سوال ۳ تمرین سری ۱ است. این تابع یک تصویر را می‌گیرد و فیلتر canny را بر روی آن اعمال می‌کند. سپس از طریق ۴ حلقه ، در هر کدام یکی از مرز های اطراف تصویر را پیدا می‌کند. این حلقه ها یک ستون یا یک سطر در هر چرخه انتخاب می‌کند و از حاشیه ی تصویر تا میانه ی آن ادامه پیدا می‌کند.

هرجا که پیکسل های سفید از کسری از کل پیکسل ها بزرگ تر بود، حلقه متوقف می‌شود و مقدار آن با مقداری جمع یا کم می‌شود و ذخیره می‌شود. این جمع یا کم کردن به این دلیل است که در صورتی که تنها ستون در تصویر باشد، کل بدنه کشتی به عنوان میله در نظر گرفته می‌شود، از طرفی در صورت زیاد بودن حاشیه اطراف ستون ستون های نزدیک به هم نمی‌توانند تشخیص داده بشوند. بنابر این تصویر تمپلیت، از اطراف به مقدار مناسبی بریده خواهد شد و بریده شده آن به عنوان خروجی تابع بازگردانده می‌شود.

در خط بعدی طول و عرض تمپلیت و تصویر اصلی در ۴ متغیر متفاوت، پشت سر هم برای استفاده های مختلف ذخیره می‌شوند. بعد از این وارد تابع اصلی پیدا کردن تمپلیت های مشابه می‌شویم. این تابع find_template نام دارد. این تابع به عنوان آرگومان های ورودی، تصویر تمپلیت و تصویر اصلی که هردو کوچک و سیاه و سفید شده‌اند را دریافت می‌کند و دو لیست که هرکدام به ترتیب شامل x ها و y های پیدا شده هستند را به عنوان خروجی برمی‌گرداند. نحوه ی کار این تابع به این صورت است:

- بعد از دریافت کردن ۲ تصویر ورودی، دو عدد لیست خالی به نام های Xcords و Ycords ایجاد می‌کند. سپس یک آرایه با اندازه ی تمپلیت بوجود می‌آورد که تمام مقادیر آن برابر mean تمپلیت هستند. این کار توسط تابع np.mean(template) انجام می‌گیرد و به اندازه تمپلیت بزرگ می‌شود. سپس دو عدد حلقه for ایجاد می‌کنیم که در هرکدام به ترتیب به اندازه طول و عرض تصویر اصلی منهای عرض و طول تمپلیت پیمایش می‌شود. سپس قطعه ای به اندازه تصویر تمپلیت از تصویر اصلی بریده می‌شود و در cut_image ذخیره می‌گردد. در ادامه از رابطه زیر استفاده می‌شود:

$$total = \sum_{k,l}(g[k,l] - \overline{g})(f[m+k, n+l] - \overline{f_{m,n}})$$

پس در حلقه total محاسبه می‌شود و با توجه به ترش‌هولدی که تعیین می‌شود، نقاطی که مقادیر total شان بالاتر از حدی قرار دارند، مختصاتشان با استفاده از append درون Xcords و Ycords ریخته می‌شود. ترش‌هولد با آزمون و خطا در اینجا ۴۰۰۰۰ تعیین شده است که مقدار مناسبی است. در پایان دو حلقه نیز لیست های Xcords و Ycords بر می‌گردند.

بعد از دریافت مقادیر یافته شده، این دو لیست را به تابع arrange_founded_coords می‌دهیم. این تابع نقاط یافته شده را می‌گیرد و دو لیست دوبعدی برمی‌گرداند که مرتب شده ی این نقاط در آن قرار دارد. در بعد اول لیست ها، لیستی از لیست ها قرار دارد که در هر لیست نقاط نزدیک به هم قرار دارند. روش کار این تابع به اینصورت است:

- ابتدا درایه های اول Xcords و Ycords به ترتیب در درایه اول در لیست اول linked_points_x و linked_points_y می‌ریزیم. سپس از دو حلقه استفاده می‌کنیم. حلقه ی اول در درایه های Xcords پیمایش می‌کند. در شروع حلقه یک flag را True می‌کنیم. در حلقه ی دوم بین لیست های اول linked_points_y پیمایش می‌کند. هرکدام از لیست هایی که پیمایش می‌شود مین همه ی اعضای آن بدست می آید می‌شود و ، چک می‌شود درایه ای که در حلقه ی اول دارد پیمایش می‌شود، در بازه مین اعضای آن لیست بعلاوه منهای نصف مقدار عرض تمپلیت قرار می‌گیرد یا نه، اگر قرار گرفت، flag برابر False شده، سپس Xcords و Ycords در لیست متناظری که از قبل در linked_points_x و linked_points_y ایجاد داشت ریخته می‌شود. و شرط نهایی بعد از حلقه ی اول نیز اجرا نمیشود.

در صورتی که کل لیست ها پیمایش شد و مقدار x که در Xcords در حال بررسی است، در هیچکدام از بازه های تعریف شده برای لیست ها قرار نگرفت، flag درست باقی می‌ماند و در انتهای حلقه اول، لیستی جدید در linked_points_x و linked_points_y می سازیم و بعنوان اولین عضو، عضو Xcords را که در حال بررسی بود به آن میدهیم و باز به اول حلقه برمی‌گردیم و پیمایش درون آن را ادامه می‌دهیم. در پایان نیز دو لیست دو بعدی linked_points_x و linked_points_y را برمی‌گردانیم.

بنابر این تا اینجا linked_points_x و linked_points_y بدست آمد. بعد از آن این دو لیست دو بعدی را به تابع make_new_coords می‌دهیم. کار این تابع خروجی دادن یه لیست تک بعدی که هر کدام از اعضای آن، mean بعد دوم لیست های وارد شده هستند. عملکرد این تابع بسیار ساده است. هنگام ورود به تابع، new_cords_x و new_cords_y با عضو اول mean لیست اول linked_points_x و linked_points_y تشکیل می‌شود. سپس از ۱ تا اندازه تعداد لیست های درون linked_points_x درون linked_points_x و linked_points_y پیمایش می‌کنیم و در هر پیمایش، mean لیست n ام linked_points_x و linked_points_y را به عنوان عضو n ام new_cords_x و new_cords_y به آنها اضافه می‌کنیم و در پایان این حلقه، تابع دو لیست new_cords_x و new_cords_y را برمی‌گرداند.

در پایان نیز این دو لیست را به تابع draw_rectangle می‌دهیم. این تابع درون دو لیست دریافتی new_cords_x و new_cords_y با استفاده از حلقه پیمایش می‌کند و در هربار پیمایش، با دستور ساده ی ،cv۲.rectangle مستطیلی با گوشه های نقاط بدست آمده ضربدر ۱۰ و نقاط بدست آمده ضربدر ۱۰ بعلاوه ی طول و عرض ذخیره شده ی تمپلیت بعد از کوچک سازی آن ضربدر ۱۰، روی تصویر بزرگ اصلی رسم می‌شود. (ضربدر ۱۰ از ضریب scale_rate می‌آید. از آنجا که نقاط در ابعاد یک دهم اصلی بدست آمده‌اند، لازم است برای رسم درست آنها، ۱۰ برابر آنها را رسم کنیم)

در پایان کد هم تصویر نهایی که بر رویش مستطیل های مورد نظر کشیده شده اند، به عنوان فایلی که در صورت سوال ذکر شده بود، ذخیره می‌گردند.

پرسش ۳ - Homography and Image Warping

در مرحله اول کارکرد برنامه توضیح داده می‌شود

کارکرد برنامه به اینصورت است که کاربر ابتدا عکسی را به برنامه میدهد (در اینجا .('book.jpg' موقع شروع برنامه تصویر با ابعادی مناسب به کاربر نشان داده میشود. این نسبت در اینجا نصف ابعاد تصویر اصلی است. سپس کاربر باید گوشه ی بالا سمت چپ تصویری که می‌خواهد بصورت عمودی در تصویری جدا ذخیره شود را دابل کلیک کند، سپس در جهت پادساعت‌گرد ۳ نقطه دیگر را انتخاب کند. نقاط انتخاب شده با خطوط متصل کننده آنها به هم بعد از هر دابل کلیک رو تصویر مشخص می‌شود و به اینصورت می‌توان از انتخاب درست نقاط اطمینان حاصل نمود.

بعد از اینکه ۴ تا نقطه مشخص شدند و مستطیلی با رنگ آبی شکل گرفت، برنامه تصویر را پردازش می‌کند و درصد پیمایش را نیز در کنسول نمایش می‌دهد. پس از اتمام آن تصویر به درست آمده با ابعاد کامل نشان داده شده و ذخیره نیز می‌گردد. اگر پنجره ی باز شده جدید را ببندیم، می‌توانیم انتخاب ۴ نقطه جدید را ادامه دهیم و به تعداد نامحدود نیز اینکار را تا زمانی که کلید esc کیبورد را نفشرده‌ایم، ادامه دهیم.

حال به سراغ کد پاسخ سوال می‌رویم. در ابتدای شروع کد دو لیست خالی Xcoords و Ycoords تعریف می‌شود. سپس تصویر 'books.jpg' بصورت رنگی دریافت می‌شود و در org_image ذخیره می‌گردد. سپس یک کپی از آن با نصف ابعاد اصلی اش را در image می‌ریزیم. برای کوچک سازی از تابع resize استفاده می‌شود که با دریافت تصویر و ضریبی که می‌خواهیم کوچک شود، با استفاده از cv۲.resize ، آنرا کوچک(یا بزرگ) می‌کنیم. در ادامه یک پنجره به نام image می‌سازیم و برای callBack موس آن، تابع operate را تعریف می‌کنیم. به این صورت در هنگام کلیک بر روی آن، ۵ آرگومان را به operate می‌فرستد. ما با ۳ آرگومان آن کار داریم، event ، mouse of x ، mouse of y .

ما با دریافت و ذخیره این ۳ آرگومان کار های مختلفی که گفته شده را انجام می‌دهیم. در بعد از آن یک حلقه می‌بینیم که شرط آن همواره درست است. این حلقه باعث می‌شود همواره image درون پنجره ی باز شده آپدیت باشد و بعد از هر عملی نتیجه به روز شده در آن قرار بگیرد. در صورت کلیک کردن esc هم با استفاده از شرط درون حلقه، می‌توانیم از حلقه خارج شویم و برنامه تمام می‌شود. از آنجایی که تمام کارهای مدنظر در تابع operate انجام می‌شود، مفصلا بخش های مختلف این تابع توضیح داده می‌شود. این تابع از ۳ بخش اصلی تشکیل می‌شود که هرکدام زیر بخش هایی نیز می توانند داشته باشند.

۱. بخش رسم نقاط و خطوط: ابتدای ورود، بررسی می‌شود event ای که بر اساس آن وارد تابع شدیم، دابل کلیک می‌باشد یا نه. اگر دابل کلیک بود، مختصات نقاطی کلیک شده که از طریق آرگومان های تابع وارد شده است، به ترتیب در Xcoords و Ycoords ذخیره می‌گردد. سپس در صورتی که کلیک اول بود، تنها در آن نقطه یک دایره ی کوچک رسم می‌شود. اینجا شرط گذاشته‌ایم که اگر مقدار Xcoords بیشتر از ۰ بود یعنی کلیک اول نبود، از نقطه ی قبلی که رسم کردیم به نقطه ی جدید یک خط نیز رسم بشود.

   اگر اندازه Xcoords برابر ۴ شد، در ابتدا ۲ خط رسم شود، یک خط از نقطه قبلی به جدید یک خط از نقطه اولی به نقطه آخر. به اینصورت پس از کامل شدن ۴ نقطه یک مستطیل کامل تشکیل می‌شود. (توجه شود else آخر برای رسم دایره، برای این است که در هنگامی که ۴ کلیک می‌کنیم ابتدا دایره رسم شود و سپس اعمال محاسباتی دیگر انجام شود، و همچنین رسم این دایره دوبار تکرار نشود.)

۲. هنگامی که ۴ کلیک صورت می‌گیرد و مستطیل تشکیل می‌شود، دو برابر مقادیر هرکدام از Xcoords و Ycoords را به تابع make_matrix می‌دهیم. این دو برابر به دلیل نصف کردن تصویر اولیه برای نمایش درست و به تبع آن نصف بودن اعضای Xcoords و Ycoords نسبت به تصویر اصلی هستند. کار این تابع ساخت یک ماتریس هموگرافی به فرم زیر است:

$$H = \begin{bmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & 1 \end{bmatrix}$$

توجه شود که از این ماتریس نگاشت زیر ازx و y های موجود در تصویر به x' و y' های تصویر جدید بوجود می‌آید

$$\begin{bmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} u \\ v \\ w \end{bmatrix} \ , x' = \frac{u}{w}, y' = \frac{v}{w}$$

برای ساخت ماتریس H ، لازم است از ۴ نقطه ورودی و ۴ نقطه خروجی برای آن استفاده کرد. ۴ نقطه ورودی را که به عنوان آرگومان تابع make_matrix دریافت می‌کنیم. برای ۴ نقطه ی خروجی، ۴ نقطه ی $(0,0)$ ، $(h,0)$ ، $(0,w)$ و (h,w) را باید داشته باشیم. w و h به ترتیب عرض و طول تصویر نهایی هستند. برای بدست آوردن w و h ، از دو تابع find_w و find_h استفاده می‌کنیم.

این دو تابع ۴ نقطه ای که بدست آورده شده را می‌گیرد، برای h فاصله دو نقطه ی اول و دوم - سوم و چهارم از طریق رابطه فیثاغورث بدست می آیند و میانگین این دو مقدار به عنوان خروجی تابع برگردانده می‌شود. و برای w هم همین اتفاق با نقاط اول و چهارم - دوم و سوم رخ می‌دهد و نتیجه مشابهی را به ما می‌دهد. دو برابر خروجی توابع درون w و h ریخته می‌شود و حالا ۴ نقطه ی $(0,0)$ ، $(h,0)$ ، $(0,w)$ و (h,w) را داریم. در ادامه برای محاسبه h_1 تا h_8 ، از رابطه ی زیر استفاده می‌کنیم:

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x'_1x_1 & -x'_1y_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -y'_1x_1 & -y'_1y_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x'_2x_2 & -x'_2y_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -y'_2x_2 & -y'_2y_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x'_3x_3 & -x'_3y_3 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -y'_3x_3 & -y'_3y_3 \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x'_4x_4 & -x'_4y_4 \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -y'_4x_4 & -y'_4y_4 \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \\ h_6 \\ h_7 \\ h_8 \end{bmatrix} = \begin{bmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \\ x'_4 \\ y'_4 \end{bmatrix}$$

سپس بنابر رابطه ی فوق، رابطه خطی فوق را بصورت زیر می‌نویسیم:

$$Ah = b$$

با ساختن A (به صورت $float64$ برای ذخیره سازی مقادیر اعشاری)و b در خط های بعدی کد، از طریق دستور np.linalg.solve ، معادله را حل کرده و نتیجه ی h را در H_hat قرار می‌دهیم. در پایان نیز ماتریس H را بصورت زیر تعریف می‌کنیم:

$$\begin{bmatrix} \hat{H}[0] & \hat{H}[0] & \hat{H}[0] \\ \hat{H}[3] & \hat{H}[4] & \hat{H}[5] \\ \hat{H}[6] & \hat{H}[7] & 1 \end{bmatrix}$$

ماتریس فوق می‌تواند نگاشتی را اعمال کند که ۴ نقطه انتخابی تصویر اصلی را به ۴ نقطه تصویر نهایی map کند. اما ما نیاز داریم که این کار را برعکس انجام دهیم. یعنی به ازای هر پیکسل در تصویری که می‌خواهیم بسازیم، متناظرش در در تصویر اصلی پیدا کنیم. بنابراین لازم است که با دستور np.linalg.inv معکوس H را بدست بیاوریم. در پایان این تابع نیس معکوس H بدست آمده را به عنوان خروجی تابع بر می‌گردانیم.

۳. در بخش بعدی تابع mapping را داریم که خروجی ندارد. اما دو ورودی تصویر اصلی و H را دارد که ماتریس هموگرافی حاصل شده می‌باشد. در اینجا با دستور np.zeros یک ماتریس به اندازه ی h و w حاصل شده در قسمت قبل، و نیز عمق ۳ برای اینکه ۳ کانال رنگی داریم، ایجاد می‌کنیم. سپس ۳ حلقه for داریم که به ترتیب برای پیمایش در کانال رنگی، طول و عرض استفاده می‌شود. برای هر پیکسل از ماتریس ۰ ایجاد شده در یک کانال رنگی بخصوص، می‌خواهیم مقداری را اعمال کنیم. این مقدار از تابع find_value بدست می آید. ورودی آرگومان های این تابع ماتریس هموگرافی H ، بردار v، مقدار rbg که بیانگر کانال رنگی است که در حال پیمایش آن هستیم، و نیز تصویر اصلی هستند. بردار v بصورت زیر تعریف می‌شود:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

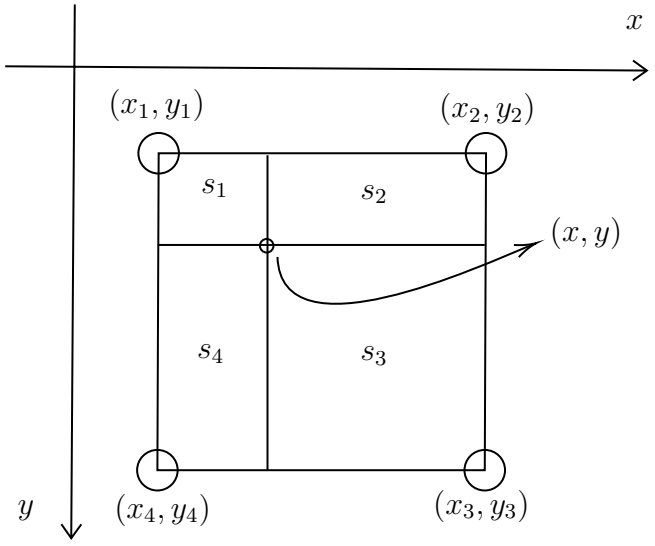
در حقیقت این همان چیزی است که باید هربار در ماتریس H ضرب شود تا مقادیر x' و y' بدست بیاید. اگر ماتریس ۰ اولیه را در نظر بگیریم، $mapped[j,i,rgb]$ در هربار اجرا شدن میان ۳ حلقه، مقدار آن برابر با int شده ی نتیجه ی تابع find_value می‌شود. حال تابع فوق را بررسی می‌کنیم.

- روش کار این تابع از لحاظ ریاضی بصورت زیر است:

$$\begin{bmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = Hv = \begin{bmatrix} u \\ v \\ w \end{bmatrix} \ , x' = \frac{u}{w}, y' = \frac{v}{w}$$

در شروع تابع، ضرب Hv فوق انجام میگیرد و نتیجه آن یعنی بردار (u,v,w) مذکور، در متغیر multiply ذخیره میگردد. حال با توجه به روابط لازم فوق برای بدست آوردن x' و y' ، دو تقسیم لازم را انجام می‌دهیم و نتیجه را در آرایه ی mapped_coords ذخیره می‌کنیم. با توجه به اینکه این مختصات بدست آمده صحیح نیستند و از طرفی مقادیر مورد انتظار ما برای مقدار پیکسل جدید باید صحیح باشد، لازم است برای بدست آوردن مقادیر پیکسل ها از تصویر قبلی، از درون‌یابی بین نقاط اطراف مختصات ناصحیح بدست آمده استفاده کنیم.

به این منظور ابتدا کوچکترین مقدار صحیح بعد از مختصات حاصل شده و بزرگترین مقدار صحیح کوچکتر از آن مختصات را با دستور int که مقدار اعشاری بدست آمده را قطع می‌کند و همان مقدار بعلاوه ی ۱ بدست می آوریم. نتایج بده دست آمده را برای جهت های x و y به ترتیب درون دو آرایه ی integer_x و integer_y ذخیره می‌کنیم. در اینجا آرایه ای به نام S ایجاد می‌کنیم که ۴ عضو دارد. هر عضو مساحت مستطیلی را ذخیره می‌کند که از برخورد خط های رسم شده نقطه ی اولیه با اضلاع مربع ۱ در ۱ ای که در آن قرار دارد (و هر راس یک پیکسل به حساب می آید) بوجود می‌آید.



در شکل بالا به روشنی می‌بینیم که آرایه S از چه عناصری تشکیل شده است. حال در ادامه از روشی استفاده می‌کنیم که از مقدار نقطه ی میانی (x,y) از جمع وزن دار نقاط اطرافش استفاده کند. در اینجا وزن هر پیکسل در راس مربع، متناسب با مساحت مستطیل طرف مقابلش است. یعنی مقدار نقطه ی میانی از رابطه زیر بدست می‌آید:

$$g(x_3, y_3) = f(x_3, y_3).s_1$$

$$g(x_4, y_4) = f(x_4, y_4).s_2$$
$$g(x_1, y_1) = f(x_1, y_1).s_3$$
$$g(x_2, y_2) = f(x_2, y_2).s_4$$

که مقدار میانی برابر می‌شود با:

$$f(x, y) = \sum_{i=1}^{4} g(x_i, y_i)$$

و این نتیجه ای است که تک تک برای هر پیکسل مختلف که map شده است، در هر ۳ کانال رنگی اعمال می‌گردد. توجه شود از آنجا که مساحت این مربع لزوما یک است، نیازی به نرمالایز کردن ندارد چون جمع s_1 تا s_4 همواره ۱ می‌شود. بنابراین تنها کافی است نتیجه را بعنوان خروجی تابع برگردانیم.

حال نتیجه را یک به یک درون ماتریس اولیه mapped قرار می‌دهیم. و آنقدر اینکار را تکرار می‌کنیم که تصویر جدید کاملا ساخته بشود. (در این حلقه قابلیت نمایش درصد پیمایش هم قرار داده شده است و با ساز و کاری ساده اینکار انجام می‌گیرد و نیازی به توضیح آن نیست.) بنابراین تا اینجا تصویر تغییر شکل یافته با مقیاس خوبی بدست آورده ایم و در mapped قرار داده‌ایم.

در انتها تصویر بدست آمده را در انتهای حلقه، وارد تابع show_image قرار می‌دهیم و تنها این تصویر با دیتاتایپ $'uint8'$ ، بصورت گفته شده ذخیره می‌گردد. قابل ذکر است که پس از اینکه تابع mapping به پایان رسید، در تابع operate در جایی که شرط برابر با ۴ بودن نقاط انتخاب شده چک می‌شود، با استفاده از دستور clear هر دو لیستی که مقادیر x و y انتخابی را در خود نگه می‌داشتند، پاک می‌شوند و برای کاربر امکان ادامه برنامه و ۴ نقطه دیگر و کلیه روند مذکور وجود دارد. از آنجا که این برنامه درون یک حلقه همواره درست قرار دارد، توجه شود تنها راه بستن برنامه فشردن دکه esc در کیبورد یا stop کردن کل برنامه است.

پرسش ۴ - Hybrid Images

در این سوال، سعی شد برای دو تصویر دور و نزدیک که خودمان باید انتخاب می‌کردیم، ایده ای متفاوت و جدید اجرا شود. ۴ نمونه تصویر متفاوت اعم از پرندگان مختلف، نقشه ی کشور ها و حیوانات و صورت انسان ها استفاده شد، اما نتایج معمولا رضایت بخش نبودند. در نهایت، دو تصویر انتخاب شده، تصاویر جغد (برای نزدیک) و هواپیمای B۲ (برای دور) می‌باشند. حدود یک سوم کد برای هم اندازه و مچ کردن این دو عکس بر روی هم است. در دو سوم دیگر از دو تصویر ساخته شده، نتایج خواسته شده را بدست می‌آوریم. به سراغ توضیح کد می‌رویم:

در ابتدای کد، دو تصویر بعنوان تصاویر اصلی وارد برنامه می‌شوند. هر دو تصویر در ابتدا با استفاده از نرم افزار خارجی، به صورت مناسبی align شده اند و طول و عرض یکسان پیدا کرده اند. بلافاصله تصاویر اصلی با نام های گفته شده در صورت سوال ذخیره می‌گردند. در ادامه تصاویر $image1$ و $image2$ بصورت سیاه و سفید در آرایه های $gray1$ و $gray2$ ذخیره می‌گردند. حال در ادامه لازم است تصاویر $image2$ ، $gray1$ و $gray2$ به تابع find_affine می‌دهیم. در ادامه روش کارکرد این تابع را توضیح میدهیم:

- در هنگام ورود به تابع، با ۳ تابع دیگر به نام های head_point ، end_point و top_point مواجه می‌شویم. به ترتیب کارکرد آنها را شرح می‌دهیم.

۱. تابع point_head : این تابع ابتدا فیلتر Canny را بر روی تصویر ورودی با ترش‌هولد بالا اعمال می‌کند. سپس از سمت راست تصویر،با استفاده از یک حلقه ستون به ستون چک می‌کند که آیا مجموع درایه های ستون غیر ۰ هست یا خیر. اگر غیر صفر بود، حلقه ی دیگری آغاز می‌شود که در این حلقه بررسی می‌گردد کدام سطر از این ستون دارای پیکسل غیر صفر بود. وقتی این پیکسل پیدا شد، طول و عرض مربوط به آن ذخیره می‌گردند، حلقه ها پایان می‌یابند و مختصات نقطه ی نوک هواپیما یا جغد به تابع find_affine باز می‌گردد.

۲. تابع point_end عملا هیچ فرقی با تابع point_head ندارد بجز اینکه ستون ها را از ابتدای تصویر بررسی میکند. با اینکار نقطه ی ابتدایی بدنه ی هواپیما یا بدن جغد پیدا میگردد.

۳. تابع point_top کمی با دو تابع بالا فرق دارد. این تابع همزمان تصویر اول و دومی را می‌گیرد، سپس کاری که برای دو تابع گفته شد را اینبار از بالا برای تصویر جغد اعمال می‌کند. یعنی از بالاترین سطر شروع می‌کند و هنگامی که به یک سطر با مجموع غیر صفر برخورد می‌کند، شماره ستونی که پیکسل در آن غیر صفر بوده است را ذخیره می‌کند. سپس در همان ستونی که اولین مقدار پیکسل غیر صفر را داشت، در تصویر هواپیما بررسی می‌شود در کدام سطر این ستون پیکسل غیر صفر می‌شود، یعنی به بدنه هواپیما برخورد می‌کنیم، و این نقطه را ذخیره می‌کنیم. سپس ۴ متغیر، شماره سطر و ستون جغد، و ستونی که مورد بررسی بود و شماره سطری که پیکسل در آن غیر صفر است به عنوان خروجی این تابع برمی‌گردد.

بعد از دریافت ۶ نقطه مذکور که متشکل از ۱۲ عدد است، می‌توانیم با استفاده از یک تبدیل ،affine بهترین تطابق جغد و هواپیما را پیدا و اعمال کنیم. برای اینکار از تابع getAffineTransform استفاده می‌کنیم. به اینصورت که ۳ نقطه ی سر، ته و بالای جغد و نقاط متناظر پیدا شده در هواپیما را به تابع می‌دهیم؛ یعنی به عنوان source نقاط، نقاط پیدا شده ی فوق برای جغد، و به عنوان ،dst نقاط پیدا شده برای هواپیما را می‌دهیم. این تابع یک آرایه ی ۲ در ۳ مناسب برای این تبدیل پیدا می‌کند.

سپس در خط بعدی، کاری که بصورت دستی برای سوال ۳ انجام داده ایم را با استفاده از تابع warpAffine انجام می‌دهیم. تابع مذکور تصویر $image2$ یا همان تصویر جغد را به همراه ماتریس affine یافت شده و نیز طول و عرض تصویر را می‌گیرد و یک تبدیل هندسی روی آن انجام می‌دهد. در نهایت نتیجه را باز درون $image2$ ذخیره می‌کند. در پایان نیز $image2$ یا همان تصویر جغد بصورت تبدیل یافته که بیشترین تطابق را با هواپیما دارد، به عنوان خروجی تابع find_affine بر می‌گردد.

بعد ازینکه تصویر جدید $image2$ از تابع خارج شد، هم $image2$ و هم $image1$ ، لازم است align شوند. به این منظور حدود ۱۰۰ پیکسل از اطراف هر کدام قطع می‌شود که به هم ریختن های احتمالی تصاویر در تبدیل affine و خارج شدن $image2$ از ابعاد تصویر، برطرف شود. بعد از مراحل بالا، تصاویر بدست آمده مطابق صورت سوال ذخیره می‌شوند.

سپس سه کانال تصویر ۱ ، بعنوان ۳ متغیر $r1$ ، $g1$ و $b1$ ذخیره می‌گردند. سپس هر سه وارد تابع fourier_transform می‌شوند. این تابع عینا تابع استفاده شده در سوال ۱ است که با یک تبدیل و یک shift ، تصویر را به طور مناسبی از حوزه مکان به حوزه فرکانس منتقل می‌کند. نتیجه ی این تابع برای هر سه کانال رنگی، در خودشان ذخیره می‌گردند. سپس هر سه تای آنها وارد تابع show_magnitude این تابع نیز به تفصیل توضیح داده شده است. آرگومان های ورودی آن تصویر و یک عدد هستند و پس از بدست آوردن بزرگی مقدار حقیقی تصویر توسط لگاریتم، برای نمایش مناسب در عدد n ضرب می‌شود و به ما خروجی می‌دهد.

در مرحله بعد، هر سه بزرگی بدست آمده در $image1\_show\_mag$ ذخیره می‌گردد. این متغیر در حقیقت یک آرایه ی عادی متشکل از ۳ کانال رنگی می‌باشد. سپس کارهای عینا یکسان برای $image2$ نیز انجام می‌گیرد و در پایان نتیجه ی بزرگی بدست آمده، با استفاده از تابع cv۲.merge و اعمال آن بر روی $image1\_show\_mag$ و $image2\_show\_mag$ ، کانال های ۳ تابع هماهنگ شده و بصورت تصویر مناسبی در می‌آید که بصورت خواسته شده ذخیره می‌گردد.

در ادامه لازم است یک فیلتر بالاگذر و یک فیلتر پایین گذر بسازیم و آنرا تک تک بر روی کانال های رنگی که بدست آورده‌ایم اعمال کنیم. برای این منظور از توابع lowpass و highpass استفاده می‌کنیم. در ادامه به توضیح این توابع می‌پردازیم:

- در تابع lowpass ، یک sigma و یک تصویر دریافت می‌گردد. طول و تصویر دریافتی در ابتدا ی آن، درون h و w ذخیره می‌گردد. سپس تابع get_gaussian_kernel اجرا می‌شود. این تابع بعد بزرگتر تصویر بعلاوه ی یک و همچنین سیگمای ورودی تابع را دریافت می‌کند. سپس یک تصویر مربعی با طول ضلع بعد بزرگتر تصویر بعلاوه ی یک با استفاده از دو عدد حلقه for می‌سازد. روش ساخت این حلقه بسیار آسان است، کما اینکه در سوال ۱ هم قدم به قدم این تابع ساخته شد، در اینجا تفاوت کوچکی دارد، اینکه در اینجا فیلتر گاوس حوزه ی فرکانس را می‌خواهیم بسازیم. این فیلتر با استفاده از فرمول زیر ساخته می‌شود.

$$H_\sigma(x,y) = e^{-\frac{x^2+y^2}{2\sigma^2}}$$

در رابطه بالا x و y طول و عرض از مرکز هستند. روش کار به این صورت است که ابتدا یک آرایه تماما صفر با طول و عرض داده شده ساخته می‌شود، سپس دو حلقه می‌سازیم که هر یک از منفی نصف طول تا مثبت نصف طول ادامه دارند. در اینجا به راحتی از رابطه بالا مقادیر مشخص برای تابع گاوس بدست می آیند و در آرایه raw جایگذاری می‌شود. در پایان نیز بدون هیچ نرمالایز کردنی، تابع برگردانده می‌شود.

در ادامه ی تابع highpass تصویر مربعی ساخته شده برای فیلتر گاوس در حوزه فرکانس، درون kernelgauss قرار می‌گیرد. سپس بلافاصله وارد تابع normalize_binary می‌گردد که در اینجا به سادگی تمام مقادیر این تصویر، با استفاده از درون‌یابی خطی درون بازه ۰ تا ۱ قرار می‌گیرد که این مطلوب ماست. سپس با استفاده از یک شرط، بررسی می‌شود که طول تصویر کمتر است یا عرض آن، هرکدام که کمتر بود، با الگوریتمی ساده از دو طرف آن به مقدار نصف فاصله شان بریده می‌شود؛ اگر هم مساوی بود دست نخورده باقی می‌ماند.

این کار بخاطر این است که تصویر فیلتر گاوس دقیقا اندازه برابری با تصویر اصلی $image1$ داشته باشد. در پایان نیز به صورتی که گفته شده است تصویر فیلتر بعد از نرمال و قابل نمایش شدن توسط تابع normalizer ، به صورتی گفته شده بود ذخیره می‌گردد. این تابع نیز قبلا چندین بار به دلیل استفاده های متعددش استفاده شده بود. این تابع با استفاده از درون‌یابی خطی، کل پیکسل های تصویر را بین ۰ تا ۲۵۵ نرمالایز می‌کند. در پایان نیز فیلتر عنوان شده، به عنوان خروجی تابع return می‌شود.

- تابع highpass عملکردی دقیقا مشابه lowpass دارد. یعنی تمام کار های گفته شده اعم از ساخته شدن یک فیلتر گاوسی در حوزه فرکانس با استفاده از تابع get_gaussian_kernel و یکسان سازی ابعاد و نرمال و ذخیره سازی تصویر در اینجا نیز انجام می‌گیرد. منتهی یک تفاوت کوچک دارد. اینکه بعد از نرمال شدن فیلتر بین ۰ و ۱ با

استفاده از تابع normalize_binary ، ۱ از کل درایه های فیلتر کم می‌شود. زیرا در این مرحله حداکثر مقدار فیلتر ۱ می‌باشد.

$$H'_\sigma(x,y) = 1 - H_\sigma(x,y)$$

و به اینصورت تابعی به عنوان تابع highpass ساخته می‌شود و در پایان نیز به عنوان خروجی تابع highpass برمی‌گردد.

بعد از ساخته شدن تابع lowpass با مقدار $\sigma = 25$ ، تک تک کانال های رنگی درون این فیلتر ضرب داخلی می‌شوند و در خودشان ریخته می‌شوند. عملی مشابه نیز برای بخش highpass مقدار $\sigma = 30$ انجام می‌گیرد. [بنابر مطالب گفته شده در مقاله ای که در داک سوال بود، لازم است فاصله ای هرچند کوتاه بین سیگمای بالاگذر و پایین گذر باشد، که با آزمون و خطا نیز، متوجه شدم اینگونه فاصله گذاشت، خیلی نتایج بهتری را به ما می‌دهد. سپس این دو مقدار که بهترین نتیجه را به ما می‌داد انتخاب شدند]

در ادامه بزرگی هر کانال رنگی کانال های ضرب شده در lowpass ، توسط تابع show_magnitude بدست آمده و دقیقا مشابه بخش نمایش بزرگی در حالت عادی قبل از فیلتر، درون متغیر های mag_1 ذخیره می‌گردند. این متغیر یک آرایه ی ۳ بعدی متشکل از کل درایه های هر ۳ کانال رنگی می‌باشد. سپس با استفاده از تابع cv.merge به هم پیوند خورده و آماده ی نمایش تصویر می‌شود. این تصویر مطابق خواسته ذخیره می‌گردد. دقیقا همین عمل نیز برای highpass نیز انجام می‌گردد. بزرگی ها در متغیر mag_2 ذخیره گشته و بعنوان بزرگی highpass شده ذخیره می‌گردد.

در ادامه تابع fourier_combine تعریف می‌گردد. این تابع ۴ آرگومان تصویر اول، تصویر دوم، ضریب تصویر اول و ضریب تصویر دوم دریافت می‌کند. سپس با استفاده از ضرب و جمع ساده، یک جمع وزن دار ساده بر روی آنها اعمال می‌کند و نتیجه ی این جمع را بعنوان خروجی بر می‌گرداند. به این صورت ما در ابتدا ۳ کانال رنگی را دو عکس را هرکدام جدا جدا با هم جمع می‌کنیم و درون ۳ متغیر fourier_sum ذخیره می‌کنیم. ضرایب بکار رفته در وزن highpass و lowpass هم درون دو متغیر highpass_coefficient و lowpass_coefficient ذخیره می‌گردند. در اینجا مقادیر بکار رفته به شرح زیر هستند.

$$lowpass\_weight = 1\ ,\ highpass\_weight = 0.28$$

در ادامه بزرگی ۳ مقداری که بدست آمده را مشابه قبل با استفاده از show_magnitude بدست می‌آوریم و در mag ذخیره کرده، سپس با استفاده از cv.merge آنرا نمایش می‌دهیم.

در انتها با استفاده از تابع inverse_fourier ، اعمالی را که برای تبدیل فوریه انجام دادیم را برعکس می‌کنیم. یعنی ابتدا شیفت مخالف می‌دهیم و با تبدیل وارون فوریه، تصویر آن در حوزه مکان را بدست می‌آوریم. سه کانال رنگی که وارد حوزه ی مکان شده اند را در b و g و r ذخیره می‌کنیم. سپس این سه تصویر را باز با استفاده از تابع cv.merge یکی می‌کنیم و یک تصویر نرمال بدست آورده و آنرا با فرمت 'uint۸' درون result ذخیره می‌کنیم. سپس تصویر اصلی در ابعاد اصلی را در فایل $'res30 - hybrid - near.jpg'$ و تصویر یک پنجم شده با استفاده از تابع resize (که تصویر اصلی و مقدار ضریب کوچک یا بزرگ شدن را می‌گیرد و با استفاده از cv.resize آنرا کوچک کرده و خروجی می‌دهد) ، را درون فایل $'res31 - hybrid - far.jpg'$ ذخیره می‌کنیم.