اصول پردازش تصاویر

نیم‌سال اول ۱۴۰۰-۱۴۰۱

مدرس: دکتر مصطفی کمالی تبریزی

دانشکده علوم ریاضی

گزارش تمرین سری سوم

نام و نام‌خانوادگی: -

شماره دانشجویی: -

کد های پاسخ درون فایل ZIP پاسخ به عنوان $q1.py$ ، $q2.py$ و $q3.py$ و نیز نتایج حاصل شده از آنها از $res1.jpg$ تا $res16.jpg$ ذخیره شده‌اند. در این سری از تمرین ها از محمدعلی علما و سروش آتشی مشورت های جزئی گرفته شد.

پرسش ۱ - Hough Transform

روند کلی حل این سوال، دقیقا مشابه توضیحات داده شده در داک تمرین سری ۳ است و تفاوت خاصی ندارد. پس در آغاز به سراغ توضیح کد می‌رویم. در بخش اصلی برنامه دو خط کد برای اندازه گیری زمان اجرا می‌بینیم. در بین این دو خط دو خط کد دیگر وجود دارد که خط اول تصویر ورودی را در ماتریس ۳ بعدی im ذخیره می‌کند و خط بعدی آنرا به تابع $q1$ می‌دهد تا در این تابع پردازش های لازم انجام گیرد و خروجی های لازم ذخیره گردد. در ادامه بخش های مختلف این تابع به ترتیب بررسی و شرح داده می‌شوند.

- در ابتدای این تابع با استفاده از $cv2.Canny$ لبه های تصویر به دست می‌اید و در ماتریس canny ذخیره می‌گردد. ترشهولد های ۱ و ۲ در این فیلتر بصورت پیش‌فرض هر دو برابر ۵۱۰ می‌باشد. در نهایت این ماتریس بصورت تصویر ذخیره می‌گردد.
- در ادامه مطابق خواسته ی صورت سوال، ماتریس انباشتگر مورد نظر سوال، در تابع maker_acc ساخته می‌شود و خروجی آن (یعنی ماتریس انباشتگر حاصل) در acc ذخیره می‌گردد. روشی که در این تابع استفاده می‌شود را شرح می‌دهیم. هنگام ورود به این تابع ابتدا یک ارتفاع برای ماتریس انباشتی تعریف کنیم که بصورت دو برابر فاصله نقطه ی بالا سمت چپ تصویر اصلی تا پایین سمت راست آن است. در ادامه یک ماتریس خالی با ارتفاع گفته شده برای r و عرض ind برای θ ایجاد می‌کنیم. ind) در اینجا برابر ۶۰۰ در نظر گرفته شده که مقدار بهینه ای است.)

حال با پیمایش روی طول و عرض تصویر canny ، هر درایه از این ماتریس هنگامی که ۰ نبود، یعنی گوشه تشخیص داده شده بود، یک حلقه ind تایی (برای هر θ) آغاز می‌شود هر θ در اینجا نمایانگر 0.3 یک درجه است(پس در مقیاس درجه بین ۰ تا ۱۸۰ درجه پیمایش می‌کنیم). پس برای بدست آوردن رادیان آن کافی است θ را در $\frac{\pi}{ind}$ ضرب نمود. حال با در دست داشتن θ و نیز x و y ، با استفاده از فرمول زیر خط مورد نظر در مختصات قطبی بدست می آید.

$$r = j\ cos(\frac{\theta\pi}{600}) + i\ sin(\frac{\theta\pi}{600})$$

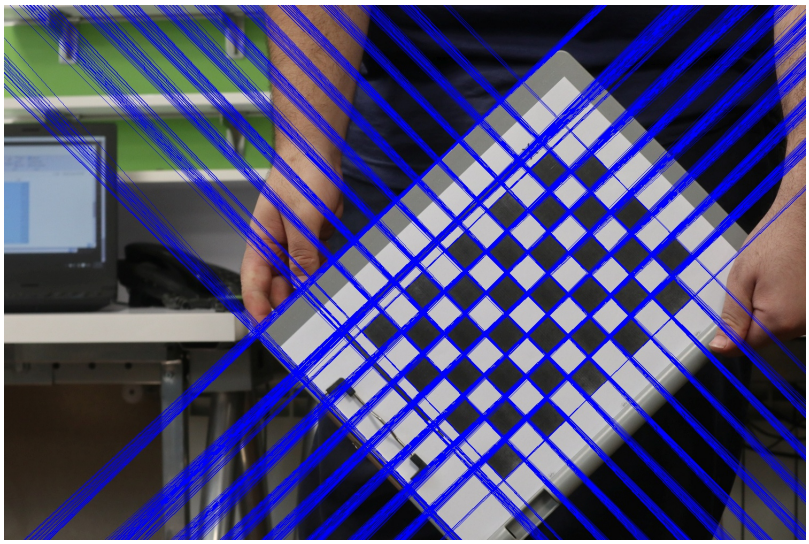
بعد از آن با گسسته سازی r ، درایه (r, θ) را به مقدار یک واحد افزایش می‌دهیم. اما قبل از آن چون مقدار r می‌تواند منفی باشد، ابتدا آنرا با نصف ارتفاع یعنی فاصله نقطه ی بالا سمت چپ تصویر اصلی تا پایین سمت راست تصویر اصلی جمع می‌کنیم. در بعد از آن هم یک خط کد برای بدست آوردن میزان پیمایش نوشته شده است و در پایان این دو حلقه نیز ماتریس انباشتی حاصل را به عنوان خروجی تابع برمی‌گردانیم. خروجی تابع نیز بصورت خواسته شده ذخیره می‌گردد. (توجه شود افزایش عرض تصویر که همان تتا هست، به دلیل نمایش بهتر و افزایش دقت بیشتر از ۱۸۰ اتخاذ شده است.)

- بعد از بدست آوردن ماتریس انباشتگر، از ۳ فیلتر برای از بین بردن خطوط کمرنگ و نیز خطوط اضافی در همسایگی هم استفاده می‌کنیم. این ۳ مورد را به ترتیب معرفی کرده و توضیح می‌دهیم.

۱. برای از بین بردن خطوط کمرنگ، کافی است از یک خط دستور بعد از بدست آوردن ماتریس انباشتگر استفاده کنیم. این یک خط در زبان پایتون بصورت زیر است:

$$acc[acc <= 100] = 0$$

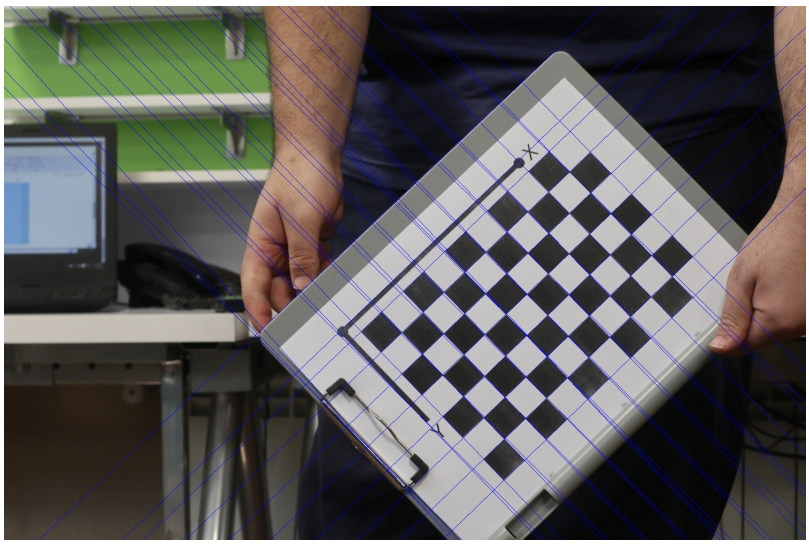
با این کار کل درایه های برابر ۱۰۰ و زیر ۱۰۰ ماتریس acc یا انباشتگر، برابر ۰ می‌شود که اکثر خطوط اضافه و غیر کاربردی حذف می‌گردند. یک نمونه نتیجه را پس از اعمال این فیلتر مشاهده می‌کنیم:



۲. سپس تابع فیلتر $filter1$ را اعمال می‌کنیم. acc را به این تابع می‌دهیم و خروجی را نیز در acc ذخیره می‌کنیم. حال تابع $filter1$ را توضیح می‌دهیم. این تابع کاری که می‌کند این است که یک فیلتر خاص به اندازه $n \times n$ روی کل درایه های تصویر اعمال می‌کند.این فیلتر بر روی هر ناحیه ای که قرار می‌گیرد تمام درایه های آن محدوده را بجز بزرگترین درایه برابر ۰ می‌کند. بنابر این به روشی بسیار کارامد همسایگی های نقاط مهم از بین میروند و نقاط مهم که احتمالا خط های مهمی هستند باقی می‌ماند. در این کد ابتدا یک حلقه بین ۰ تا ۵ داریم. اندازه فیلتر n توسط رابطه زیر بدست می‌آید:

$$n = \lfloor 2 + 1.7^i \rfloor$$

که i همان شماره حلقه است. محاسبه اندازه فیلتر با این رابطه هم سریع و هم نسبتا دقیق است و نقطه مهمی از قلم نخواهد افتاد، از طرفی سعی می‌شود با این رابطه، فیلتر روی نقاط تکراری اعمال نشود. سپس روی کل درایه های ماتریس انباشتگر این فیلتر را اعمال می‌کنیم. در هر مرحله با استفاده از تابع $cv2.minMaxLoc$ حداکثر مقدار و جایگاه آن را بدست می‌آوریم و بعد از ۰ کردن محدوده همان مقدار حداکثر را در همان نقطه حداکثر قرار می‌دهیم. توجه شود که خطوط کد اضافی برای افزایش سرعت برنامه و نمایش مقدار پیمایش آن نوشته شده اند و نقش تاثیر گذاری ندارند. در ادامه نتیجه این فیلتر و فیلتر قبلی را مشاهده می‌کنیم:



۳. سپس تابع فیلتر $filter2$ را اعمال می‌کنیم. acc را به این تابع می‌دهیم و خروجی را نیز در acc ذخیره می‌کنیم. حال تابع $filter2$ را توضیح می‌دهیم. این تابع کاری که می‌کند این است که یک فیلتر خاص به اندازه s روی کل درایه های تصویر اعمال می‌کند.این فیلتر مانند فیلتر قبلی بر روی هر ناحیه ای که قرار می‌گیرد تمام درایه های آن محدوده را بجز بزرگترین درایه برابر ۰ می‌کند. اما دو تفاوت اساسی با فیلتر قبلی دارد:

(آ) اندازه این فیلتر بسیار اندازه بزرگتری دارد و برابر $\frac{ind}{12}$ است. به اینصورت اندازه ی این فیلتر متناسب با عرض ماتریس انباشتگر است که در صورت تغییر آن، کد و بخصوص این فیلتر به درستی کار کند.

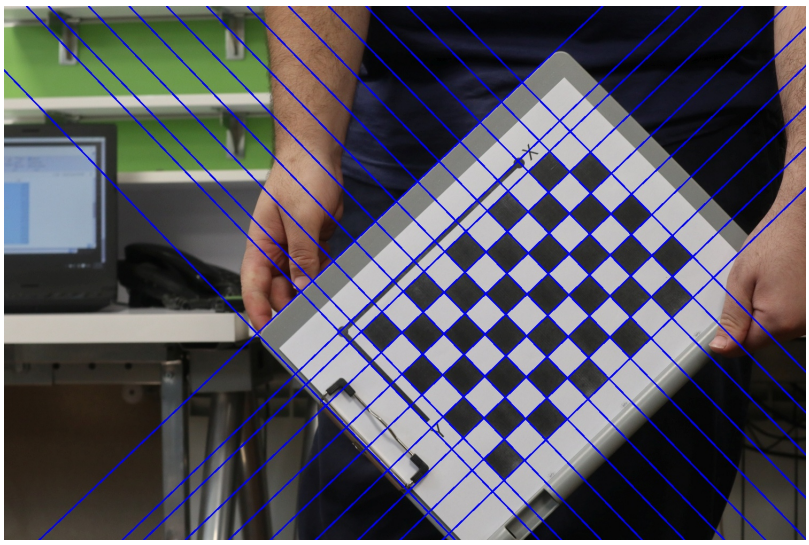
(ب) این فیلتر وقتی کار می‌کند که ۲ درایه از آن ناصفر باشند. پس برای حذف ۲ به ۲ یه خطوط نزدیک به هم اما تا حد خوبی ناموازی کاربرد دارد. پس حساسیت آن روی درایه هایی که در یک عرض قرار دارد بسیار کمتر است.

در ابتدای کد اندازه فیلتر s توسط رابطه زیر بدست می‌آید:

$$s = \lfloor \frac{ind}{12} \rfloor$$

سپس روی کل درایه های ماتریس انباشتگر این فیلتر را اعمال می‌کنیم. در هر مرحله با استفاده از تابع $np.nonzero$ درایه های ناصفر آن محدوده را بدست می‌آوریم. اگر تعداد آن ها برابر ۲ بود، عرض آنها را مقایسه می‌کند. از عرض یکی از آنها در همسایگی دیگری با شعاع $\frac{\sqrt{ind}}{8}$ نبود، فیلتر مشابه قبلی اعمال می‌شود.

از لحاظ آماری این کار بسیار مثمر ثمر واقع می‌شود؛ چراکه خطوط موازی با تقریب بسیار خوبی در همسایگی مذکور قرار می‌گیرند و اینکه دو خط در همسایگی هم قرار نداشته باشند یعنی زوایای مختلف نزدیک به هم هستند و در حقیقت دارند یک خط را به ما نشان می‌دهند و لازم است خط با مقدار کمتر حذف شود. توجه شود که خطوط کد اضافی برای افزایش سرعت برنامه و نمایش مقدار پیمایش آن نوشته شده اند و نقش تاثیر گذاری ندارند. در پایان نیز نتیجه ۳ فیلتر یاد شده را می‌بینیم:

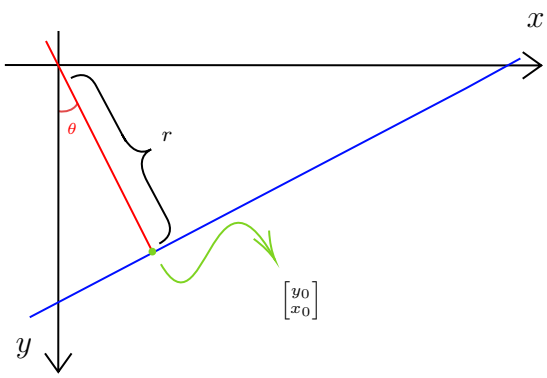


- حال که ۳ فیلتر مورد استفاده را مورد بررسی قرار دادیم، به سراغ رسم خطوط می‌رویم. در ابتدا یک کپی از تصویر را در $res56$ ذخیره می‌کنیم که محتوای اصلی آن از دست نرود. سپس ماتریس acc و $res56$ را به تابع drawline میدهیم که این خطوط را برای ما رسم کند، (حدودا مخالف کاری که در تابع maker_acc) انجام دادیم. حال به توضیح دقیق این تابع می‌پردازیم. بعد از ورود به تابع روی ماتریس انباشتگری که فیلتر ها بر رویش اعمال شده است پیمایش می‌کنیم. هرگاه درایه ی (r, θ) غیر صفر بود، وارد تابع transform می‌شود.

در این تابع به عنوان ورودی (r, θ, h) را می‌گیرد و a و b را در مختصات دکارتی خروجی می‌دهد.در این تابع ابتدا r را منهای نصف ارتفاع ماتریس انباشتگر می‌کنیم تا r واقعی که قبلا برای نمایش با این مقدار جمع کرده بودیم بدست بیاید. θ هم مطابق معمول به رادیان تبدیل می‌شود. در اینجا با استفاده از دو فرمول زیر $x0$ و $y0$ بدست می‌آید. (این فرمول متضاد کاری است که پیشتر انجام دادیم و صحت آن نیز با توجه به تعریف بدیهی است.)

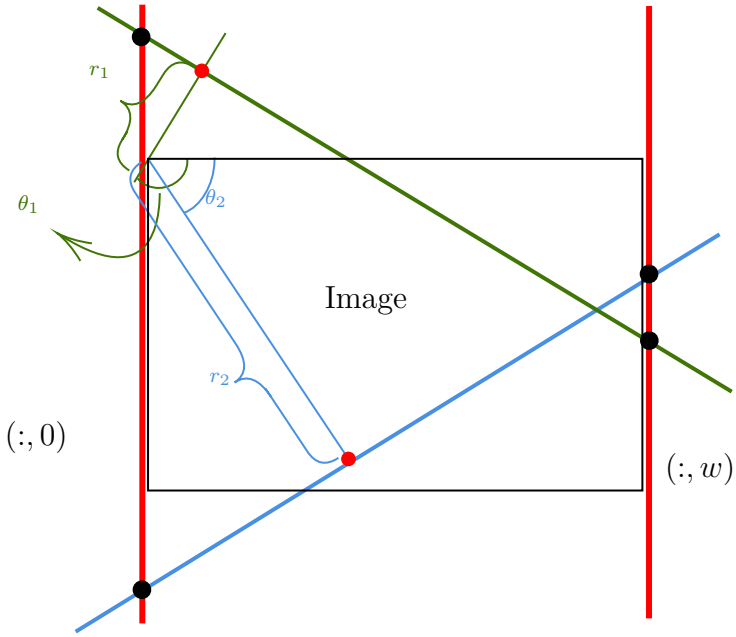
$$y0 = r\ cos(\theta)$$
$$x0 = r\ sin(\theta)$$

این مختصاتی است که خط مورد نظر ما از آن می‌گذرد و بر خطی که از مبدا رسم می‌شود عمود است:



پس ما این نقطه از خط مورد نظر را داریم، و همچنین با توجه به وارون بودن محور مختصات در اینجا نسبت به حالت عادی، شیب خط مورد نظر برای این مساله برابر $tan(-\theta)$ است. برای بدست آوردن معادله این خط بصورت $y = tan(-\theta)\ x + b$ تنها کافی است را نوشته و $x0$ و $y0$ را در آن جایگذاری کنیم. بدین ترتیب b نیز بدست

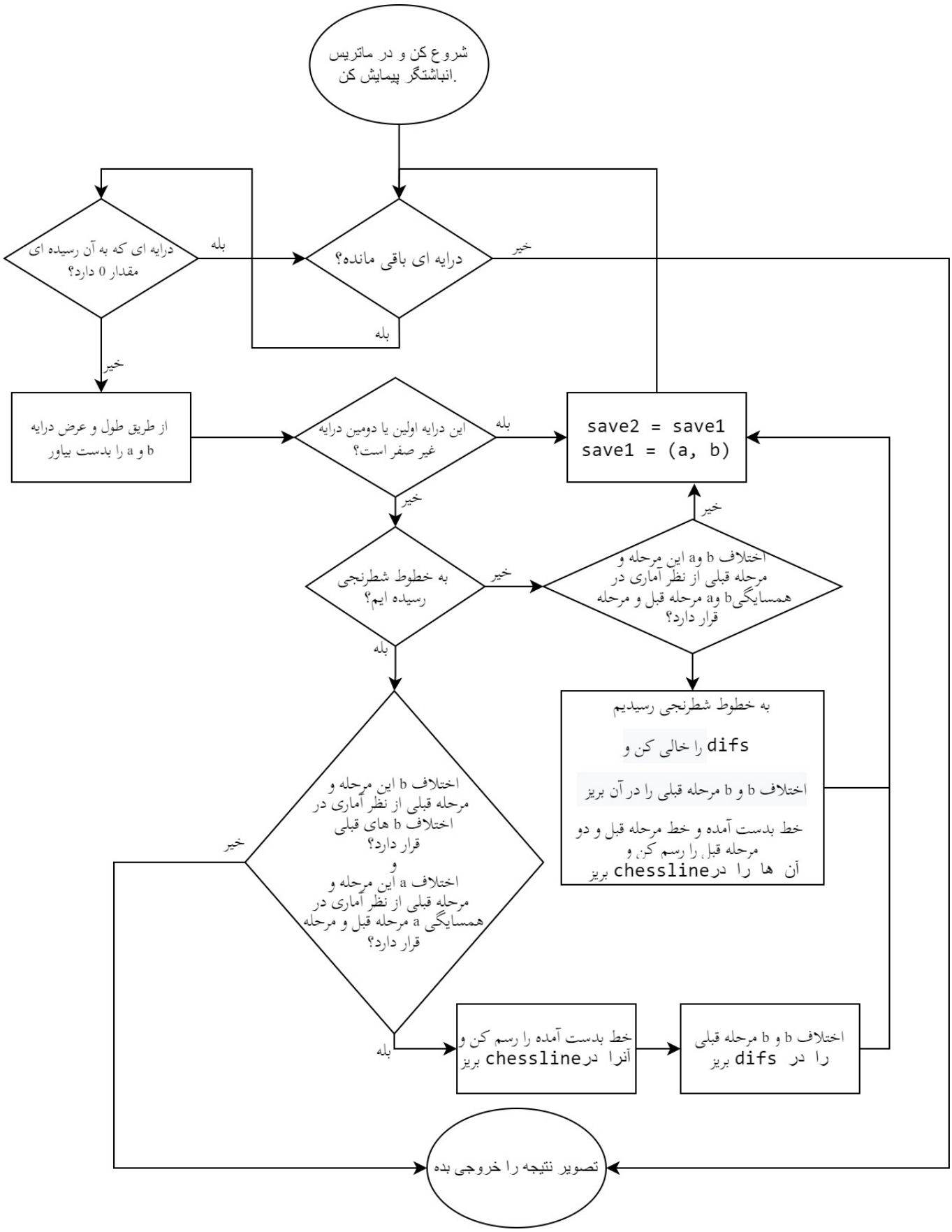
می آید. حال کافی است a، b و تصویر اصلی را به تابع $draw$ بدهیم. این تابع یک بار x را ۰ می‌دهد و یکبار w ، که w پهنای تصویر است. به این ترتیب دو خروجی به ما $y1$ و $y2$ را می‌دهد. حال با دادن دو نقطه $(y1, 0)$ و $(y2, w)$ ،(یعنی دو نقطه که نقاط تقاطع برخورد خط مورد نظر با سمت چپ و راست تصویر) و همچنین تصویر اصلی به تابع $cv2.line$ می‌توانیم کل خطوطی که در محدوده ی تصویر می افتد را رسم کنیم. (توجه کنید ک همین کار را می‌توانیم با خطوط بالا و پایین تصویر انجام بدهیم و تفاوتی ندارد). در نهایت خطی رسم می‌شود که می‌تواند در محدوده تصویر باشد یا نه. شکل زیر روش رسم دو نمونه خط را نشان می‌دهد:



حال اگر این مجموعه دستورات را برای تمام درایه های غیر ۰ ماتریس انباشتگر انجام بدهیم، کل خطوط فیلتر شده بدست می آیند. در پایان تابع نیز مقدار img ورودی تابع برمی‌گردد و در $res56$ ذخیره می‌شود. سپس در ادامه تابع $q1$ نتیجه بدست آمده ذخیره می‌گردد.

- در ادامه به توضیح روش پیدا کردن خطوط شطرنجی و حذف بقیه خطوط می‌پردازیم. در ابتدا یک لیست به نام chesslist باز می‌کنیم. در این لیست a و b خطوط شطرنج را برای استفاده ی بعدی ذخیره خواهیم کرد. در ادامه تابع findchess را دوبار صدا می‌زنیم. این تابع ۴ ورودی acc یا ماتریس انباشتگر، img یا تصویر اصلی، chessline یا لیست خطوط شطرنج و order یا ترتیب پیمایش هستند. در ابتدای تابع حدود و ترتیب و یک سری متغیر bloolean تعریف می‌کنیم که در ادامه به کاربرد آنها می‌پردازیم.

می‌خواهیم بر روی کل درایه های ماتریس acc پیمایش کنیم. با توجه به ورودی order تصمیم می‌گیریم از آخرین ردیف تا اولین ردیف برویم یا بالعکس. (ترتیب ستون ها ثابت از اولین ستون به آخرین ستون هستند.) هر درایه ای که ۰ نباشد، باعث می‌شود برنامه وارد یک پروسه بشود. کاری که اینجا دارد انجام می‌شود را بصورت یک فلوچارت نمایش می‌دهیم که درک بهتری از چگونگی انجام حذف سایر خطوط داشته باشیم:



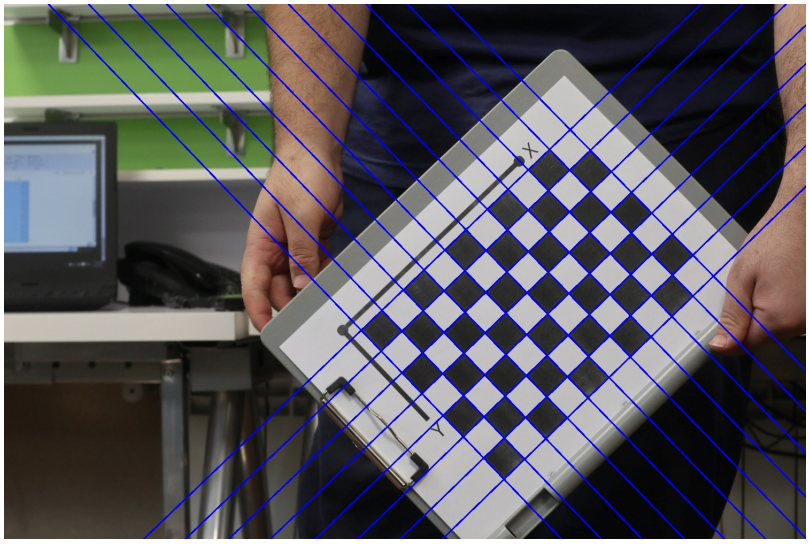
از آنجا که الگوریتم نسبتا واضح است، تنها متغیر ها و جزئیات را توضیح می‌دهیم. از نظر آماری در همسایگی بودن یعنی فاصله ی آن دو کمتر از $2\sqrt{m}$ باشد که m مقدار جامع تر ماست. مثلا فاصله دو خانه را با فاصله های خانه های قبلی را می‌خواهیم بسنجیم و ببینیم این دو در همسایگی هم هستند یا خیر، در اینجا میانگین فاصله های

قبلی را m در نظر گرفته و با رابطه $2\sqrt{m}$ شعاع همسایگی را بدست آورده و در همسایگی بودن این دو را بدست می‌آوریم. دقت کنید به دلیل اینکه a ها معیار خوبی برای تشخیص فاصله نیستند و مقادیرشان بر حسب تانژانت تتا تغییر می‌کند، در صورتی که کل داده های قبلی را جمع کنیم و مانند b متوالیا میانگین بگیریم و همسایگی نسبت به میانگین را حساب کنیم، احتمال بوجود آمدن خطاهای بزرگ زیاد می‌شود. بنابر این مقایسه همسایگی فواصل ۳ a متوالی تنها به هدف بدست آوردن تغییر جهت و موازی بودن حدودی خطوط می‌باشد.

لیست difs اختلاف عرض از مبدا (b) های خانه های متوالی شطرنج را در خود نگه می‌دارد. این خانه وقتی خانه های متوالی پیدا شدند خالی شده و از آن به بعد b ها در دو پله متوالی به آن اضافه می‌شود. chessline همانطور که پیشتر گفته شد a و b نقاط داده شده را برای استفاده ی بعدی ذخیره می‌کند. $save1$ که آخر هر حلقه برابر با a و b تنظیم می‌شود، در حین طی حلقه حاوی a و b مرحله قبل است و $save2$ نیز حاوی a و b دو مرحله قبل است.

متغیر start مشخص می‌کند که به حلقه رسیده ایم یا نه. این متغیر که در حالت عادی False است، در هنگامی که اولین ۳ خط شطرنج را پیدا می‌کنیم True می‌شود و باقی می‌ماند. حال دلیل اینکه از دو جهت در ارتفاع ماتریس acc پیمایش می‌کنیم این است که از یک سمت وقتی پیش می‌رویم در جایی به خانه های شطرنج برمی‌خوریم و خط بر روی آن رسم می‌شود. حال در پایان یا به خطوطی دیگر بخورد می‌کنیم که از تابع خارج می‌شویم یا جهت تغییر می‌کند که باعث تغییر ناگهانی در a می‌شود که اینجا هم خارج می‌شویم. و لزوما اگر از سمت دیگر پیمایش کنیم اول به جهت دیگر خطوط خانه های شطرنج می‌رسیم که آن ها را رسم می‌کنیم و به دلایل مشابهی از برنامه خارج می‌شویم.

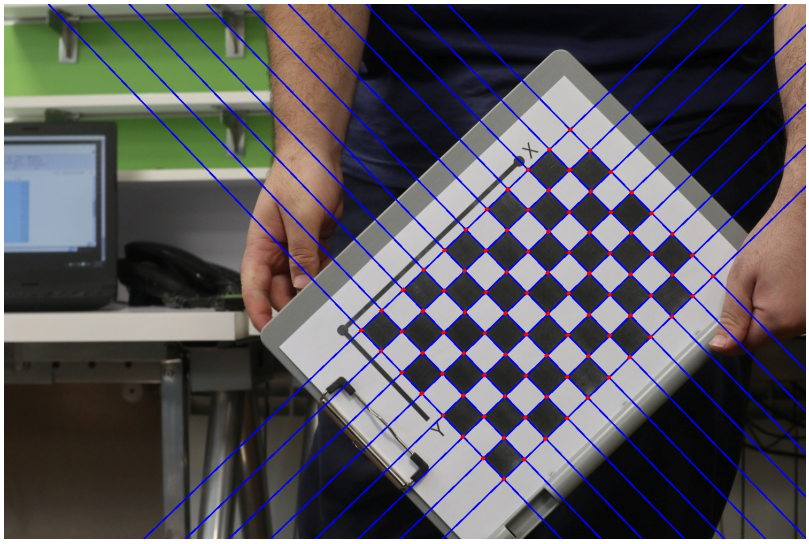
بنابراین با این روش می‌توان هم خطوط عمودی هم افقی صفحه شطرنج را رسم کرد. در این روش از ویژگی موازی بودن خطوط شطرنجی (در هر موقعیتی) استفاده کردیم. یعنی این خطوط باید شیب خط (a) بسیار نزدیک و اختلاف عرض از مبدا های نزدیک $(b-b')$ به صورتی که گفته شد استفاده می‌کنیم. نتیجه نمونه این بخش را در ادامه می‌بینیم:



بعد از اجرای این تابع از دو جهت و سپس ذخیره ی نتایج، به سراغ بخش آخر می‌رویم.

- در بخش آخر می‌خواهیم گوشه های خانه های شطرنج را پیدا کنیم. به این منظور لیست chessline را که در بخش قبل بدست آمد و تصویر اصلی که خطوط شطرنجی روی آن کشیده شده است را به تابع findcorners میدهیم. حال عملکرد این تابع را توضیح میدهیم. در این تابع یک حلقه وجود دارد که ستون به ستون پهنای تصویر را پیمایش می‌کند. در هر مرحله لیست ys خالی می‌شود. سپس در ادامه حلقه ای تعریف می‌کنیم و در آن بین خط های شطرنجی که بدست آمده اند پیمایش می‌کنیم. از آنجا که chessline شامل a و b هر خط هست، $y = ax + b$ که a و b از chessline می آید و x شماره ستون است (که در حال پیمایش آن هستیم) را حساب کرده و می‌بینیم که آیا در محدوده ی ۰ تا w قرار دارد یا خیر (قابل رویت است یا نه).. اگر در این بازه بود با یک حلقه در ys ها جستجو

می‌کنیم که یک y دیگر از بین خطوط در این ستون در همسایگی با شعاع $\frac{h}{500}$ وجود دارد یا خیر. اگر وجود داشت با دستور cv۲.circle یک دایره قرمز با شعاع ۲ در آن نقطه که ارتفاع $\lfloor y \rfloor$ دارد رسم می‌شود و سپس این y به لیست ys اضافه می‌گردد. به این ترتیب همه نقاط تقاطع خطوط بصورت دایره هایی قرمز نشان داده می‌شوند. نمونه ی این بخش هم بصورت زیر در می آید:



در پایان نیز تصویر نهایی بعنوان خروجی از تابع برگشته و ذخیره می‌گردد و تابع $q1$ به پایان می‌رسد.

در پایان نیز زمان اجرای برنامه به کاربر نشان داده شده و برنامه تمام می‌شود.

پرسش ۲ - Texture Synthesis

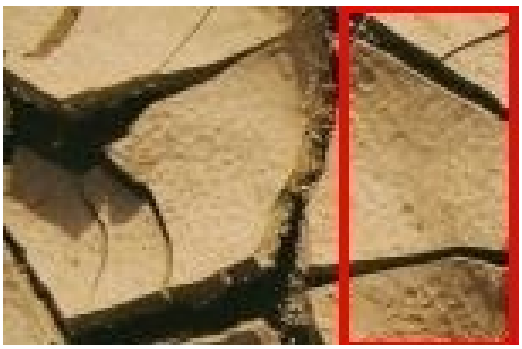
در این سوال از روش های گفته شده در کلاس استفاده شده است. روند کلی ساخت یک تصویر بزرگ از یک تکسچر کوچک در اینجا این است که ابتدا ردیف اول، سپس ستون اول و سپس بقیه تصویر ساخته می‌شود. حال به سراغ کد پاسخ سوال می‌رویم. در ابتدا در بخش اصلی کد، تکسچر ورودی در متغیر texture ذخیره می‌گردد و یک ماتریس ۲۵۰۰ در ۲۵۰۰ در ۳ به فرمت عددی $uint8$ ساخته می‌شود که تصویرمان در آن جای بگیرد. سپس main_patch را تعریف می‌کنیم که نشانگر ارتفاع patch ای است که پیدا می‌شود و به ادامه ی تصویر افزوده می‌شود. area_match هم داریم که بیانگر عرض پچ برای تمپلیت مچ می‌باشد. سپس limit را داریم که مشخص می‌کند که موقع تمپلیت مچینگ باید حداقل به چند جواب برسیم و از بین آنها رندوم انتخاب کنیم.

size را داری که با استفاده از روابط بین ۲۵۰۰ و متغیر های بالا می‌سنجد که چن بار باید طول و عرض را برای پر کردن تصویر طی کنیم. سپس gradient را داریم که می‌تواند True یا False باشد. با فعال بودن این گزینه وقتی یک پچ جدید پیدا می‌شود و جایگزین می‌گردد، تغییر بین ۲ تصویر به آرامی صورت می‌گیرد که این مورد برای وقتی که تکسچر متشکل از دانه ها یا تکه های مختلف است بسیار کارآمد می‌باشد. در صورت غیر فعال بودن gradient تصویر تیز بریده می‌شود که این حالت برای تصاویر طرح دار و بدون دانه و تکه بسیار مناسب است. سپس به تابع $q2$ تصویر تکسچر را می‌دهیم و تصویر خروجی کامل شده ۲۵۰۰ در ۲۵۰۰ را از آن دریافت می‌کنیم. حال این تابع را توضیح می‌دهیم.

در ابتدا لازم است بنابر اینکه گرادینت فعال یا غیر فعال است دو متغیر a و b را تعریف کنیم که بعدا به توضیح کارکرد آن می‌پردازیم. سپس قطعه بالا سمت چپ تکسچر را با عرض و طول main_patch را در همین گوشه با همین عرض و طول از تصویر اصلی بدست می آید. بعد از آن درون یک حلقه size بار پیمایش می‌کنیم و در هر مرحله متغیری به نام x تعریف می‌کنیم که نشان می‌دهد چقدر به جلو رفته‌ایم. x یک دنباله حسابی تشکیل می‌دهد که از $patch_main - 50$ شروع شده و $patch_main - 50$ عدد در هر پله به آن اضافه می‌گردد. سپس متغیر های main_patch و area_match همچنین limit و a و b و x به دو تابع row_first و clm_first داده می‌شود که به ترتیب ردیف و ستون اول را برای ما می‌سازند. این دو تابع عینا مانند هم هستند ولی با این تفاوت که یکی عمودی پیش می‌رود یکی افقی، این نشان می‌دهد جهت x و y آنها عوض شده است، یعنی مثلا اگر در تصویر تابع داشته باشیم $img[y, x]$ ، در تابع دوم خواهیم داشت: $img[x, y]$. پس توضیح یکی از آنها می‌تواند کفایت کند. پس به سراغ تابع row_first می‌رویم.

در ابتدای آن pic_match را تعریف می‌کنیم که همان پچ مورد استفاده از تصویر اصلی برای پیدا کردن قطعه بعدی است . برای مثال در قدم اول، این قطعه به شکل زیر در می‌آید:

بخش سیاه سمت چپ نیز بخشی است که تصویر جدید می‌خواهد در آن قرار بگیرد. سپس با استفاده از دستور cv۲.matchTemplate با روش NORMED_CCOEFF_TM و با دادن تصویری که در مستطیل قرمز رنگ قرار گرفته است و تصویر تکسچر (منهای ۵۰ ستون آخر)، تصویر نتایج جستجو در res ذخیره می‌گردد. بعد از آن یک حلقه به تعداد ۲۰۰ بار می‌گردد که در هر گردش از ترشهولدی که در ابتدا ۱ است، $\frac{1}{200}$ کم می‌کند تا به ۰ برسد. سپس با دستور np.where مکان نقاطی که بالا تر از ترشهولد قرار دارند را پیدا می‌کنیم و اگر این تعداد از limit بیشتر باشد، از حلقه خارج شده و با rnd.randrange بصورت تصادفی از بین موارد پیدا شده یکی را انتخاب کرده و طول و عرض آن را در متغیر coordinates می‌ریزد. به این ترتیب طول و عرض بهترین مقدار تصادفی بدست می‌آید. سپس کل تصویر بدست آمده در جایگاه به عرض و طول main_patch قرار می‌گیرد. در تصویر زیر تصویر پیدا شده در موقعیت مورد نظر قرار گرفته است و دور آن مستطیل قرمز رنگ رسم شده است:



در ادامه می‌خواهیم به روش برش مرزی مرز این دو را پیدا کرده و سمت چپ آن را به تصویر قبلی و سمت راست آن را به تصویر جدید بدهیم.

به این منظور تصویر به ارتفاع main_patch و عرض area_match (که با آن تمپلیت مچ کردیم) را از تصویر اصلی می‌گیریم و بصورت grayscale در $check1$ ریخته و تصویری با همین ابعاد را از پچی که پیدا کردیم انتخاب کنیم و بصورت grayscale در $check2$ قرار می‌دهیم. با کم کردن این دو از هم و بتوان ۲ رساندن آنها boundry_image را بدست می‌آوریم. به ابتدای boundry_image یک ردیف سیاه به عرض ۵۰ اضافه می‌کنیم و یک لیست خالی تعریف میکنیم. bestcost را عنوان حداکثر مقدار ممکن می‌گیریم که یک کران بالا برای تمام cost های مسیر ماست.

به این ترتیب بین a تا b یک حلقه تعریف می‌کنیم و مقدار را در start قرار دهد . و در نهایت یک ماتریس دیگر به شکل boundry_image می‌سازیم ولی تمام درایه های آن برابر bestcost باشد. سپس درایه با ردیف ۰ و ستونی که دارد روی آن پیمایش می‌شود را برابر ۱ در نظر می‌گیریم. به این ترتیب یک نقطه شروع برای مسیرمان داریم. سپس در ردیف ها بین ۱ تا boundry_image پیمایش می‌کنیم. در هر پیمایش نیز بین ستون a تا b پیمایش میشود. در اینجا میگوییم در هر ردیف به ردیف بالاتر و ستون یکی عقب تر، همان ستون و ستون یکی جلوتر نگاه کن و کمترین مقدار آن را با مقدار boundry_image جمع می‌کنیم و در همان خانه قرار می‌دهیم. توجه شود که اگر از حد ستون های a و b خارج شویم نمی‌توانیم به سمت چپ تر برای a و به سمت راست تر برای b نگاه کنیم و آن را لحاظ کنیم.

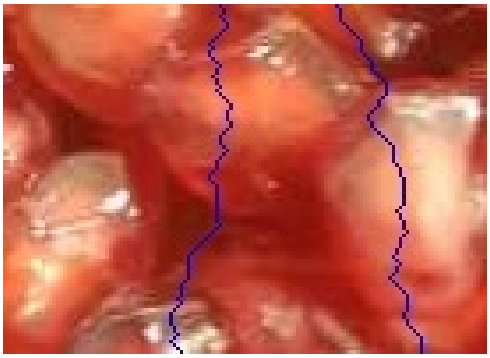
به این ترتیب cost هر خانه با شروع از خانه start بدست می‌آید. سپس وقتی که به آخرین ردیف می‌رسیم کمترین مقدار این ستون که هزینه ی نهایی با شروع از خانه start است در totalcost ذخیره می‌گردد. در مرحله بعدی بعد از خروج از حلقه این مقدار با bestcost مقایسه می‌شود که در حالت اول یک کران بالا برای totalcost است. اگر totalcost کوچکتر از bestcost بود، bestcost برابر با totalcost قرار می‌گیرد و costs_image نیز در costs_bestimage قرار میگیرد. آنقدر این چرخه صورت میگیرد که start به $b-1$ برسد و کل حالات شروع بررسی بشوند. و در نهایت پس از خروج از این حلقه بهترین cost ها در costs_bestimage قرار دارند.

بعد ازینکه costs_bestimage با تعریف مشخص بدست آمد، یک حلقه بین ردیف های آن ایجاد می‌کنیم که از ردیف آخر شروع می‌شود. در ابتدا در ردیف آخر کمترین مقدار را جست و جو می‌کنیم و پس از پیدا کردن آن موقعیت آن را در minx قرار داده و minx را نیز به عنوان درایه اول list به آن اضافه می‌کنیم. پس از آن در هر حلقه نگاه می‌کنیم که درایه اول لیستمان چیست، (در ابتدا برابر کمترین cost ردیف آخر است) در ردیف قبل ترش بین ستون های یکی قبل و یکی بعد تر به شرطی که بین a و b باشد می‌گردیم و کمترین cost را باز به عنوان درایه اول list به آن اضافه می‌کنیم. وقتی این کار را تا ردیف شماره ۱ ادامه دهیم، درون list شماره ستون های بهترین cost ذخیره می‌گردد. واضح است که شماره ی index آنها نیز نشانگر ردیف هستند.

حال می‌خواهیم آنها را جایگذاری کنیم برای این کار دو روش در اختیار داریم.

۱. بدون gradient: در این روش با سادگی در بین ردیف ها پیمایش می‌کنیم و هر چه در سمت چپ مقدار لیست در آن نقطه هست را برابر تصویر اولیه (که در حالت پیش‌فرض به همین صورت است) و هرچه در سمت راست آن است را به تصویر پیدا شده اختصاص می‌دهیم. به این صورت بهترین برس بدون gradient انجام می‌شود نمونه ی آن را در شکل زیر می‌توانیم ببینیم:

۲. در این حالت به صورت ۰ به ۱ مرز بندی نمی‌کنیم، بلکه از یک سمت با استفاده از یک حلقه دیگر، به آرامی وزن تصویر همان سمت را کم و کمتر می‌کنیم تا جایی که وقتی به مرز می‌رسیم هردو مقدار $\frac{1}{2}$ داشته باشند. سپس بعد از عبور از مرز بدست آمده متناسب با مقدار باقی مانده از مرز به آرامی ضریب آنرا کم می‌کنیم تا در نهایت برابر ۰ بشود و طرف مقابل برابر ۱ بشود. بعد از انجام این کار تصویری نرم خواهیم داشت که برای تصاویر دارای دانه و تکه بسیار مناسب تر می‌باشد. نمونه ی این روش را در تصویر انار می‌بینیم:

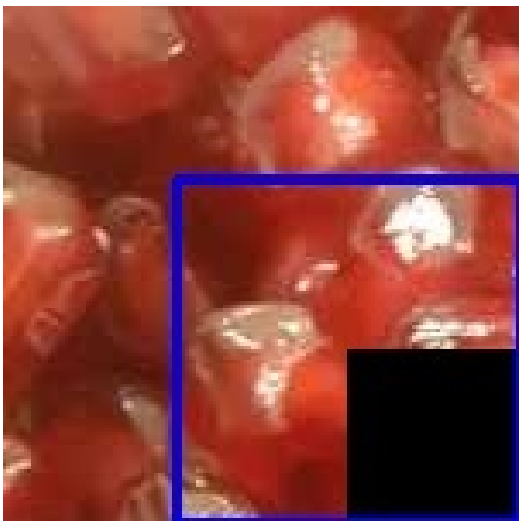


حال اینجا تاثیر a و b را می‌بینیم، گفتیم که a و b در حالت بدون gradient برابر ۰ و حداکثر پهنای پچ هستند اما در حالت با gradient برای ۱۰ و حداکثر پهنای پچ منهای ۱۰ هستند و کل محاسبات و حدود برش مرزی نیز به همین ترتیب انجام شد. این به دلیل این است که می‌خواهیم حد متعادلی از این به آرامی تبدیل شدن را نگه داریم. برای مثال اگر مرز تا ۰ تا پهنای برود، در یک طرف به سرعت تغییر تصویر انجام می‌شود و در یک طرف بسیار کند که این خود باعث کاهش شدید کیفیت تصویر می‌شود.

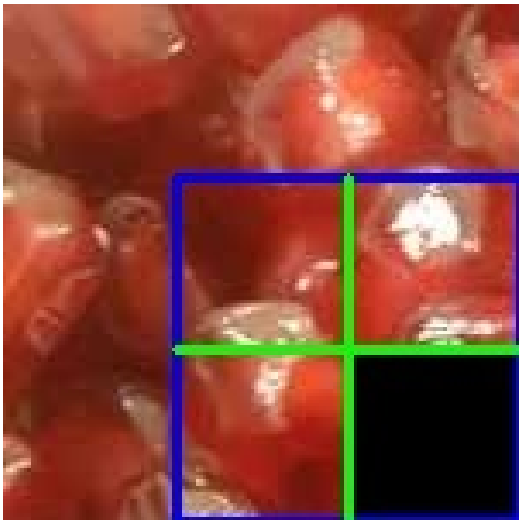
بعد از جایگذاری تابع به پایان می‌رسد و باز به حلقه ی درون تابع $q2$ بر می‌گردیم. این یعنی تصویر به ستون اول با موفقیت اضافه شده است و بعد از اضافه شدن این ستون، ردیف متناظر نیز اضافه میشود و همانطور که گفته شد دقیقا یک طرز کار دارد با این تفاوت که جای x و y عوض می‌شود و بجز این هیچ تغییر دیگری نداریم. (خطوط دیگر در این حلقه برای افزایش سرعت اجرا یا نمایش پیمایش انجام گرفته مورد استفاده قرار می‌گیرند.)

وقتی این حلقه تمام می‌شود، یعنی کن سطر و ستون های اول ساخته شده است و دو حلقه تو در توی دیگر برای پیمایش در بقیه تصویر می‌سازیم. حد هر کدام از ۰ تا size می‌باشد. در اینجا مانند گذشته یک r و یک x ساخته می‌شود که به ترتیب بیانگر طول و عرضی است که قرار است پچ جدید در آن قرار بگیرد. سپس با همان پارامتر های قبل وارد تابع main می‌شویم.

روند کار در اینجا کلا مانند قبل است، اما تفاوت هایی هم دارد که به توضیح آنها می‌پردازیم. در هنگام شروع تابع از آنجا که نمی‌توانیم تصویر L شکل به تابع تمپلیت مچ بدهیم، لازم است گوشه ی سیاه مستطیل را به گونه ای پر کنیم. برای اینکار مربع مورد نظر را به ۴ بخش تقسیم می‌کنیم و میانگین ۳ بخشی که تصویر سیاه ندارد را درون بخش سیاه آن قرار می‌دهیم و تمپلیت مچ انجام می‌دهیم. تصویر اولیه ی ما اگر بصورت زیر باشد:



به اینصورت آنرا به ۴ بخش تقسیم می‌کنیم:



و سپس در بخش خالی میانگین ۳ بخش دیگر را قرار می‌دهیم:

این کار به دلیل همسان سازی و پیدا کردن پچ های بهتر انجام می‌گیرد. حال دقیقا مانند قبل می‌گردیم تا بهترین تطابق تصویر بالا را با ترشهولد های متغیر مانند قبل بدست بیاوریم. اولین تفاوتی که این بخش با بخش های قبل می‌کند این است که در اینجا بر خلاف قبل، تصویر اولیه (تصویر بالایی) را در متغیر pic ذخیره می‌کنیم و سپس کل تصویر پیدا شده را به

شکل زیر درون جایگاه L شکل قرار می‌دهیم، به صورتی که حتی خود L را هم پوشش بدهد. سپس لازم است که ۲ برش مرزی بزنیم. یک برش برای مستطیل بالایی (سمت بالای (L و یک برش برای مستطیل سمت چپی (سمت چپ ،(L ما برای نمونه مستطیل بالایی را شرح می‌دهیم. پس از جدا کردن این تصویر و تصویر قبلی و تفریق به توان ۲ رساندن و ذخیره در boundary_image ، بجای پیمایش در کل ردیف ها برای بدست آوردن بهترین مسیر، بسته به فعال یا غیر فعال بودن gradient یکی از دو کار زیر را انجام می‌دهیم:

۱. اگر gradient غیر فعال باشد، از ردیف آخر ستون آخر مسیریابی را شروع می‌کنیم که به سمت چپ برسیم و عینا با روش قبلی بهترین مسیر را در لیست ذخیره می‌کنیم. نمونه ای از برش مرزی بدون gradient برای مستطیل بالایی را می‌توانیم ببینیم:

۲. اگر gradient فعال باشد حداکثر مقدار ردیف سمت راست که بین a و b باشد ۱ می‌شود، یعنی به عنوان نقطه شروع انتخاب می‌شود و سایر کار ها دقیقا مانند قبل انجام می‌شود. یعنی به سمت چپ مستطیل می‌رسیم و بهترین مسیر را در لیست ذخیره می‌کنیم. نمونه ای از این برش بصورت زیر مشخص است:

همینطور که می‌بینیم gradient در این مورد به کیفیت پچ اعمال شده کمک زیادی کرده است.

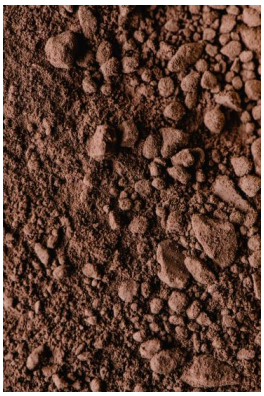
همانند قبل بین a و b بطور مداوم پیمایش می‌کنیم و در نهایت که لیست بدست آمد، در اینجا بجای قبل که از جایی که لیست نشان می‌دهد تا شروع تصویر بعد مقدار تصویر بعدی را قرار دهیم، از تصویر اصلی تا جایی که لیست قرار دارد تصویر اصلی قرار می‌گیرد، این کار برای عملی شدن برش مرزی مورد نیاز است. روش قرار گیری هم بجز ترتیب آن هیچ فرقی با سایر گذشته ندارد و هم gradient هم بدون آن عینا با یک روش پیمایش و جایگذاری عمل می‌کنند. عینا همین کار را برای مستطیل عمودی سمت چپ نیز انجام می‌دهیم فقط جای x و y طبق معمول فرق می‌کند و در بقیه موارد هیچ تفاوتی ندارد.

در انتها باز به تابع $q2$ برگشته و x در آن حلقه یک پله به جلو می‌رود که باعث می‌شود پچ L شکل در همان ردیف اما ستون بعدی بدست بیاید و اعمال شود، پس از پایان یک ردیف r یک واحد زیاد می‌شود که باعث ساخته شدن ردیف بعدی می‌شود. آنقدر این کار پیش می‌رود که در کل تصویر size بار پچ L شکل اعمال شود و به اینصورت تصویر کامل می‌شود. در پایان نیز از حلقه خارج شده و pic_new به عنوان خروجی به کد اصلی برگشته و در آنجا خروجی نهایی ذخیره می‌شود. در

پایان برنامه نیز زمان اجرا به کاربر نمایش داده می‌شود. (در تمام روند پردازش نیز درصد پیمایش به کاربر نشان داده می‌شود)

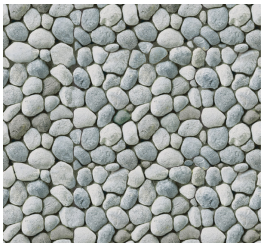
تکسچر های نمونه برای نتایج به کار رفته تکسچر طرح ۱۱ با gradient خاموش:

تکسچر کاکائو ۶ با gradient روشن:



تکسچر ترمه با gradient خاموش:

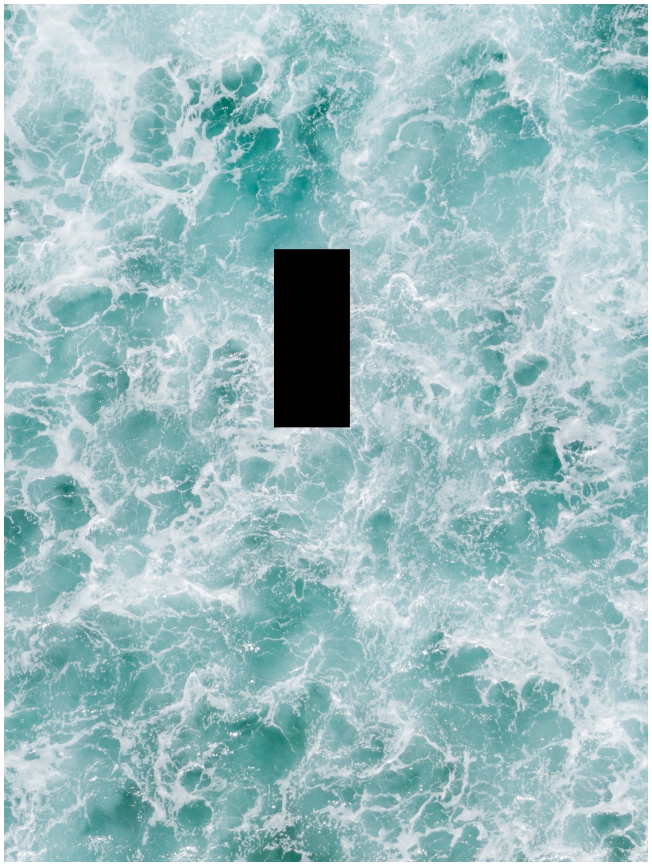
و تکسچر سنگ با gradient روشن :



هستند.

پرسش ۳ - Image Completion

راه حلی که برای این سوال استفاده شد شباهت بسیار زیادی به راه حل سوال قبلی دارد. ابتدا لازم است که تکه های شامل شناگر و پرندگان به صورتی از بین بروند. پس طبق صورت سوال، با نرم افزار paint مستطیل هایی سیاه در حدود سوژه ها رسم می‌کنیم و آنها را با فرمت png ذخیره می‌کنیم. در ادامه تصاویری که به برنامه داده‌ایم که نتایج از آن حاصل شده است را می‌بینیم (که البته برای نمایش و کاهش حجم تبدیل به jpg شده اند):



بعد از سیاه کردن نواحی گفته شده، لازم است برنامه ای بنویسیم که محدوده های سیاه را بتواند تشخیص دهد و مختصات آنها را ذخیره کند. به همین منظور به سراغ توضیح کد پاسخ می‌رویم.

پس از شروع شدن برنامه تصویری که نواحی را سیاه نشان می‌دهد با فرمت png دریافت می‌شود، دلیل اینکه تاکید بر روی png داریم، این است که تصویر اگر jpg ذخیره شود، مستطیل های سیاه کاملا سیاه نیستند و در گوشه ها حتی مقدار بسیار زیاد و دور از ۰ می‌گیرند. سپس تصویر را به تابع $q3$ می‌دهیم. در این تابع کل کار های اصلی برای اعمال مورد نظر انجام شده و خروجی آن را با نام res برمیگرداند.

در شروع این تابع ماتریس pic_new را تعریف می‌کنیم که کار آن این است که در هر درایه اش جمع ۳ کانال رنگی تصویر اصلی را نگهداری می‌کند. سپس این ماتریس را به تابع area_find می‌دهیم. این تابع ماتریس pic_new را می‌گیرد و لیست points را که یک لیست شامل مختصات نقطه بالا سمت چپ و نقطه پایین سمت راست برای هر محدوده سیاه است را برمی‌گرداند. با این ۴ نقطه می‌توانیم محل یک مستطیل سیاه متوجه بشویم. روش کار تابع تابع area_find را توضیح می‌دهیم.

- ابتدا لیست خالی points را تعریف می‌کنیم و سپس بر روی کل درایه های ماتریس پیمایش می‌کنیم و در هر پیمایش find را برابر با True قرار می‌دهیم، اگر به یک نقطه با مقدار ۰ خوردیم، یعنی رنگ آن پیکسل در تصویر اصلی تماما ۰ است. حال بین نقاط از قبل پیدا شده (points) می‌گردیم که آیا این نقطه در آن محدوده ها صدق می‌کند یا خیر (یعنی این نقطه در داخل مستطیل سیاه قرار دارد یا نه) اگر قرار داشت find برابر False شده و بدون انجام کاری دیگر به انتهای حلقه می‌رود. اما اگر نتوانست آن نقطه را در آن محدوده ها پیدا کند، یعنی نقطه ی سیاه جدیدی را پیدا کردیم. پس این نقطه را به همراه pic_new به تابع points_find می‌دهیم.

  روش کار این تابع بسیار آسان است. در همان ستونی که نقطه ورودی آن قرار دارد آنقدر پیش روی می‌کند که به جایی برسد که مقدار ماتریس آن دیگر ۰ نباشد. سپس در آخرین سطری که مقدار ماتریس ۰ بود، آنقدر در ستون ها پیشروی می‌کند که مقدار ماتریس باز هم غیر ۰ شود. این نقطه، نقطه ی گوشه ی سمت راست یک مستطیل است. سپس این تابع مختصات دو نقطه ی ورودی اش که مختصات نقطه ی گوشه ی سمت چپ مستطیل است (چون وقتی وارد یک مستطیل می‌شویم یه اولین نقطه ای که می‌رسیم، نقطه ی بالا سمت چپ آن مستطیل است) و نقطه ی پایین سمت راست مستطیل که بدست آمد را پشت سر هم برگردانده و آن را در تابع area_find درون متغیر found می‌ریزیم. سپس چک می‌شود که آیا ابعاد این مستطیل از ۵ در ۵ بزرگتر است یا خیر (کوچکتر بودن از این مقدار نشان دهنده ی نقاط سیاه پراکنده اند). اگر از این مقدار بزرگتر بود آنرا به لیست points اضافه کرده و باز حلقه را ادامه می‌دهیم. در پایان حلقه کل نواحی سیاه پیدا شده و درون لیست points قرار دارد. پس این لیست را برگردانده و در تابع $q3$ درون لیست points ذخیره می‌کنیم.

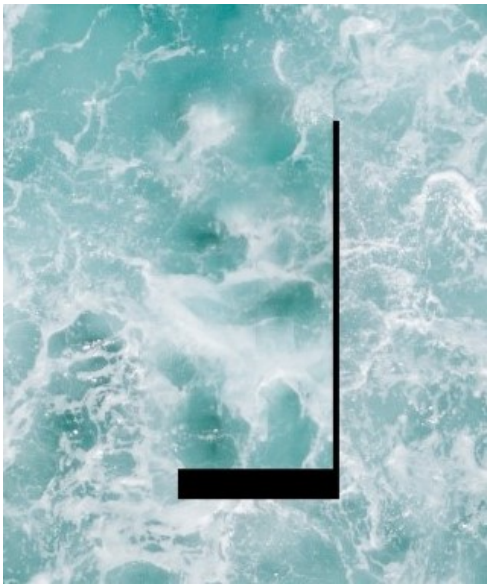
حال که نقاط را داریم، متغیر ها و ثابت هایی عینا مانند سوال ۲ تعریف می‌کنیم و سپس وارد یک حلقه بر روی points می‌شویم که تک به تک نواحی سیاه تصویر را پر کنیم. ما ۴ مرحله را برای رسیدن به هدف مورد نظر طی می‌کنیم که تک به تک آنها را توضیح می‌دهیم.

۱. در ابتدا به همان روش قبل با پچ های L شکل، اکثر مساحت ناحیه ی سیاه را می‌پوشانیم. این روش هیچ فرقی با روش L شکل سوال دو ندارد بجز اینکه برای شروع از ۵۰ ستون و ردیف قبل از ناحیه ی سیاه شروع می‌کنیم که کل ناحیه به درستی پوشانده شود، و نیز یک شرط مهم موقع templateMatching قرار می‌دهیم که این را در اینجا توضیح می‌دهیم. موقعی که تمپلیت مچ را انجام می‌دهیم، یک حلقه همواره درست می‌نویسم که ترشهولد را از ۱ به ۰ ببرد و اگر به ۰ رسید باز به ۱ برود و همین جریان ادامه داشته باشد.

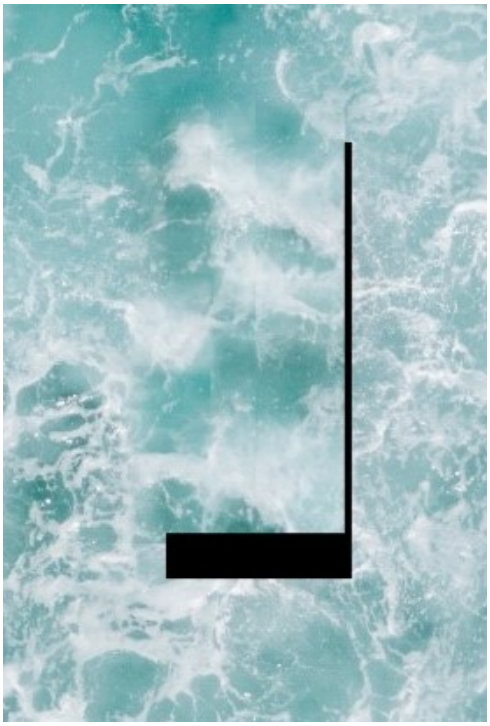
   درون این حلقه اگر تعداد قطعه های یافته شده به limit رسید، از بین آنها یکی به تصادف انتخاب می‌شود و در اینجا با توجه به عرض و طول قطعه، می‌بینیم که بخشی از قطعه درون یکی از نواحی سیاه قرار می‌گیرد یا خیر. اگر قرار می‌گرفت این قطعه از بین قطعه های یافته شده حذف می‌گردد و یک قطعه دیگر انتخاب می‌شود. این کار آنقدر ادامه پیدا می‌کند که یا به قطعه ای مناسب برسیم که باعث خروج از حلقه می‌شود یا تعداد قطعات تمام می‌شود که در اینصورت ابتدا به اندازه $\frac{9}{2000}$ به ترشهولد زیاد شده و سپس به ابتدای حلقه ی اولیه ی همواره صحیح برمی‌گردیم که ترشهولد را به اندازه $\frac{1}{200}$ کم می‌کند. پس یعنی $\frac{1}{2000}$ از ترشهولد کم می‌شود. اینکار برای این است که در این نقطه ی حساسی که یافته هایمان به اندازه است اما مناسب نیست بسیار آرام تر پیش برویم که دقت از دست نرود، زیرا اگر برای مثال ۵ قطعه پیدا کردیم و ناگهان $\frac{1}{200}$ باز از آن کم کنیم، ناگهان می‌بینیم ۴۰ قطعه پیدا کردیم که احتمال خطا

بسیار بالا می‌رود.

سپس همین کار تکرار می‌شود و آنقدر پیش می‌رود که یک قطعه ی مناسب پیدا شود و مختصاتش در coordinates ریخته شود . سپس عینا همان فرایند سوال ۲ برای این بخش طی می‌شود (منتها چون تعداد استفاده از ادامه ی فرایند زیاد است، این بخش را در تابع ریخته‌ایم). سپس با فرایند gradient سوال قبلی مرز ها به نرمی تغییر پیدا می‌کنند و این فرایند تا جایی که امکان دارد پیش می‌رود. پس اکثر محیط تصویر با این تابع پوشش داده می‌شود. برای مثال بعد از اجرای این تابع نتایج بصورت زیر خواهند بود:



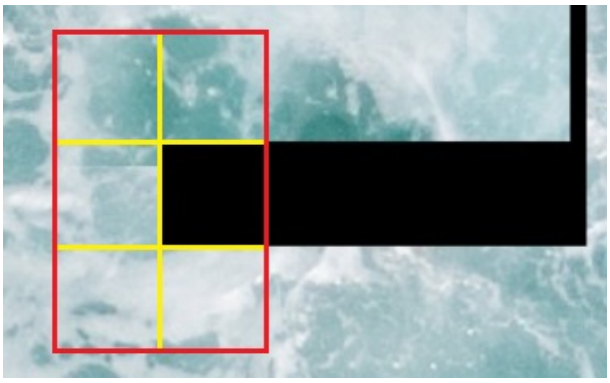
۲. در ادامه ستون آخر را به ارتفاع ۵۰ سیاه می‌کنیم که برای اجرای صحیح این مرحله آماده بشود:



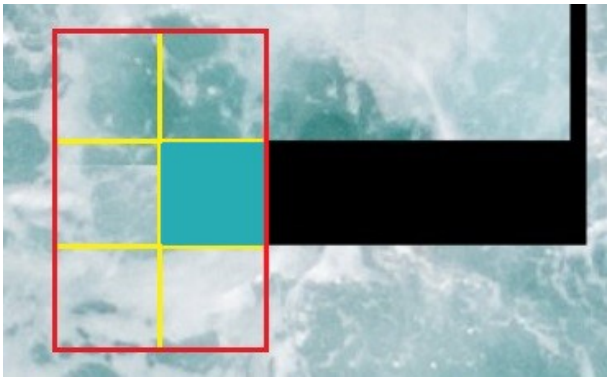
در این مرحله سطر آخر که در حال حاضر سیاه است را می‌سازیم. برای اینکار یک فور در تابع $q3$ ایجاد می‌کنیم که به اندازه ای که امکان دارد با استفاده از تابع row_last ردیف آخر را بسازد و در پهنای ناحیه سیاه خالی پیشروی کند. با این تفاوت که پچ این قسمت نعل اسبی می‌شود. برای اصلاح این بخش مربع بالای قسمت خالی، مستطیل

سمت چپ قسمت خالی و مربع زیر آن را میانگین وزن دار می‌گیریم و درون قسمت خالی می‌ریزیم (حدودا به روشی که در سوال ۲ پیاده شده است). یعنی اگر پچ بصورت زیر را داشته باشیم:

با بخش بندی بصورت زیر در می آید:

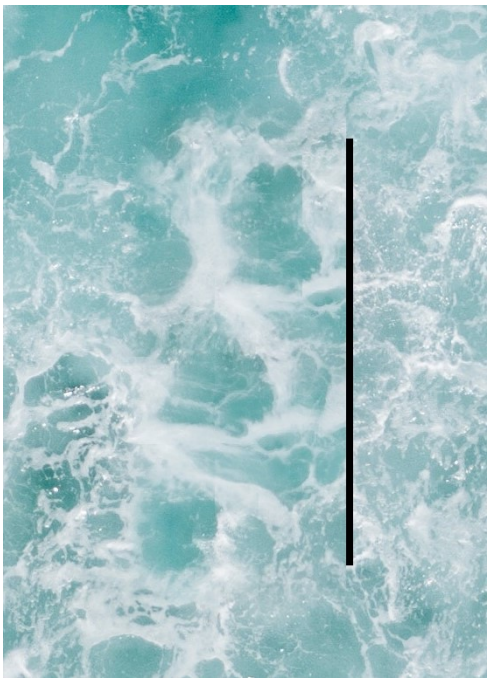


و سپس با میانگین گیری وزن دار و قرار دادن آن در بخش خالی بصورت زیر می‌شود:

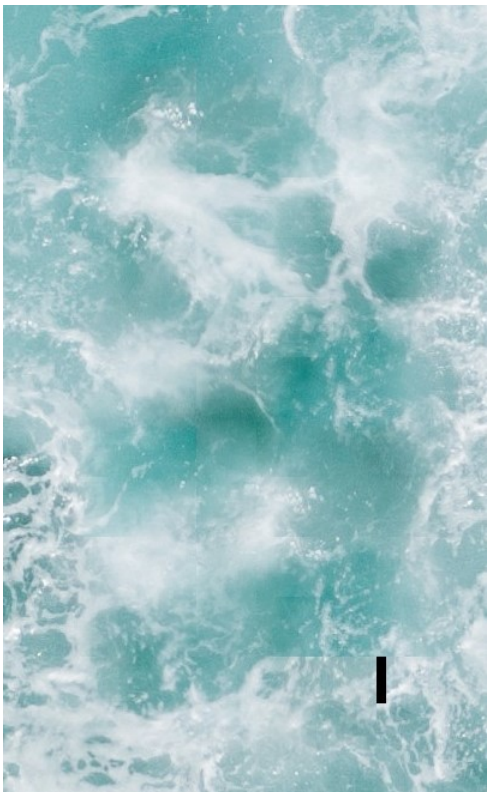


و در نهایت پچ زیر جستجو می‌شود:

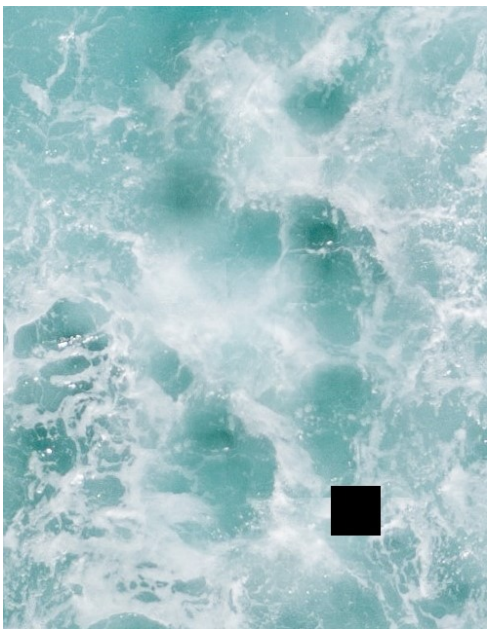
پس از جست و جو و پیدا کردن و انتخاب تصادفی عینا مانند بخش L شکل، از سمت بالا، پایین و سمت چپ با همان توابعی که توضیح داده شد برش مرزی خورده و به آرامی برش اعمال می‌شود که متوجه تغییر مرز نشویم. پس از این مرحله تصویر بصورت زیر در می‌آید:



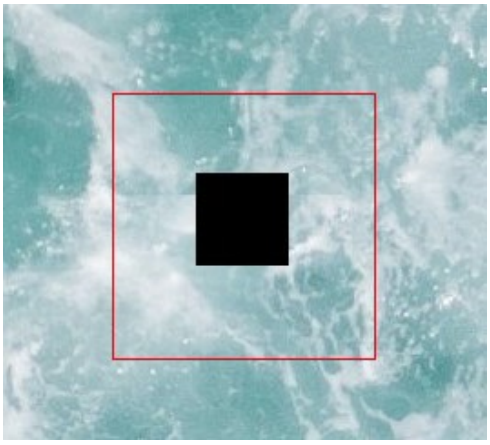
۳. سپس ستون آخر را به پهنای ۵۰ خالی می‌کنیم و دقیقا مانند قبل با همان روش، میانگین وزن دار می‌گیریم و پچ پیدا می‌کنیم و ادامه می‌دهیم تا حداکثر جایی که می‌توانیم ادامه بدهیم. در حقیقت تنها تفاوت این مرحله با مرحله قبل تغییر y و x است و با یک جابجایی ساده این دو مرحله تبدیل به هم می‌شوند. پس نیاز به توضیح مفصل برای این بخش نیست. در پایان این بخش تصویر نمونه بصورت زیر بدست می‌آید:



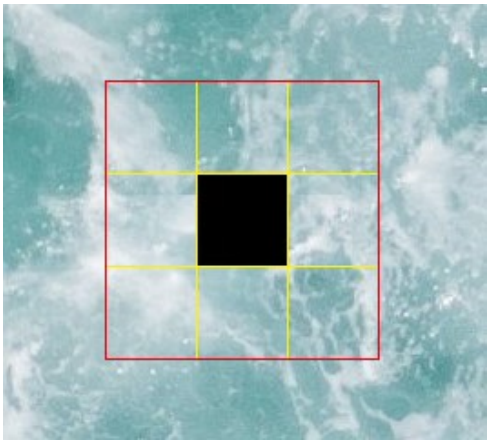
۴. در بخش آخر تکه ی کوچک باقی مانده را می‌خواهیم پر کنیم، پس ابتدا ۵۰ در ۵۰ را خالی می‌کنیم. تصویر خالی شده و آماده ی اجرای مرحله نهایی بصورت زیر است:



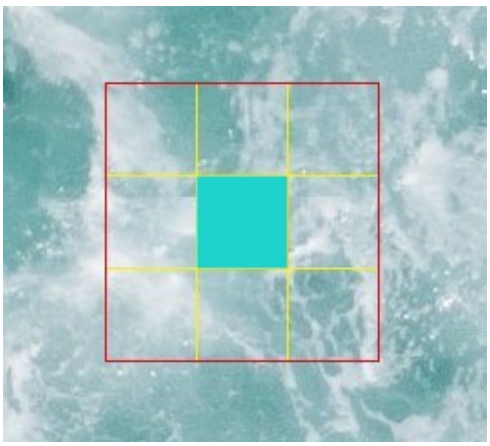
حال تنها یک تصویر که مربع ۵۰ در ۵۰ سیاه در مرکز آن قرار دارد را انتخاب کرده و میانگین ۸ مربع دورش را حساب کرده و مربع ۵۰ در ۵۰ وسطی را برابر با آن رنگ می‌کنیم. یعنی اگر مربع زیر را در نظر بگیریم:



آن را بصورت زیر بخش بنده می‌کنیم:



سپس مانند زیر میانگین مربع های کناری را در وسطی می‌ریزیم.



و پچ زیر، پچ مورد استفاده برای تمپلیت مچینگ می‌باشد.

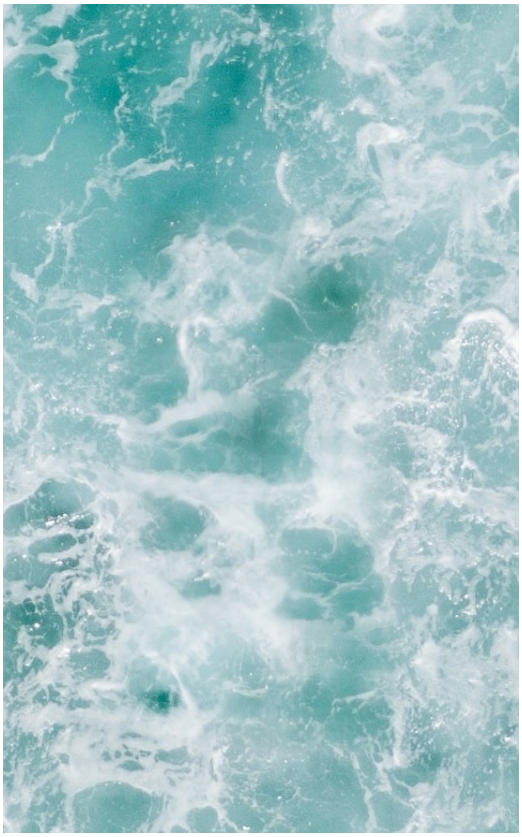
س از پیدا کردن این پچ با ۴ برش مرزی بالا، پایین، چپ و راست با توابع مشابهی که قبلا توضیح داده شد، تصویر حدودا آماده می‌شود:

پس از پایان این ۴ مرحله، اگر در تصویر بابا کمی دقت کنیم، وجود خطوط خیلی ریزی می‌شویم، این به دلیل اختلاف چند واحدی بین دو تصویر بوجود می آید که به سادگی با تابع smoothing قابل رفع است.

- این تابع تصویر را دریافت می‌کند و پس از نرم کردن نقاط برخورد آن را خروجی می‌دهد. این تابع شامل ۲ حلقه ی اصلی است، یکی برای خطوط عمودی و دیگری افقی؛ اول از همه برای خطوط عمودی ، به تعداد w_size که همان تعداد برش هایی است که قرار داده‌ایم حلقه اجرا می‌کنم و در هر کدام از حلقه ها ک حلقه ۳ تایی اجرا می‌کنیم. این حلقه باعث می‌شود ۳ ستون چپ و راست و خود شکاف رویشان یک فیلتر اثر کند.

  یک حلقه درون آن نیز از ۵۰ ردیف قبل از شروع ناحیه تا ۵۰ ردیف بعد از شروع ناحیه شروع به کار می‌کند و به هر درایه ای که می‌رسد یک فیلتر جعبه ای مناسب ۳ در ۳ روی آن اعمال می‌شود. این کار باعث نرم شدن نا ملموس نقاط برخورد می‌شود و کیفیت نتیجه را افزایش می‌دهد. بعد از پایان این حلقه، حلقه ی افقی شروع به کار می‌کند که این تنها با جابجا کردن x و y نیز ممکن است و نیاز به توضیح خاصی ندارد.

در نهایت نتیجه‌مان با استفاده از تابع بالا بصورت پایین در می‌آید.



در پایان نیز بلافاصله بعد از تابع smoothing نتیجه تصویر برگردانده شده و از تابع $q3$ خارج و در بخش اصلی کد نتیجه را خروجی گرفته و زمان اجرا را به کاربر نشان می‌دهیم.