

بنام خدا

گزارش تمرین ۴ مطلب

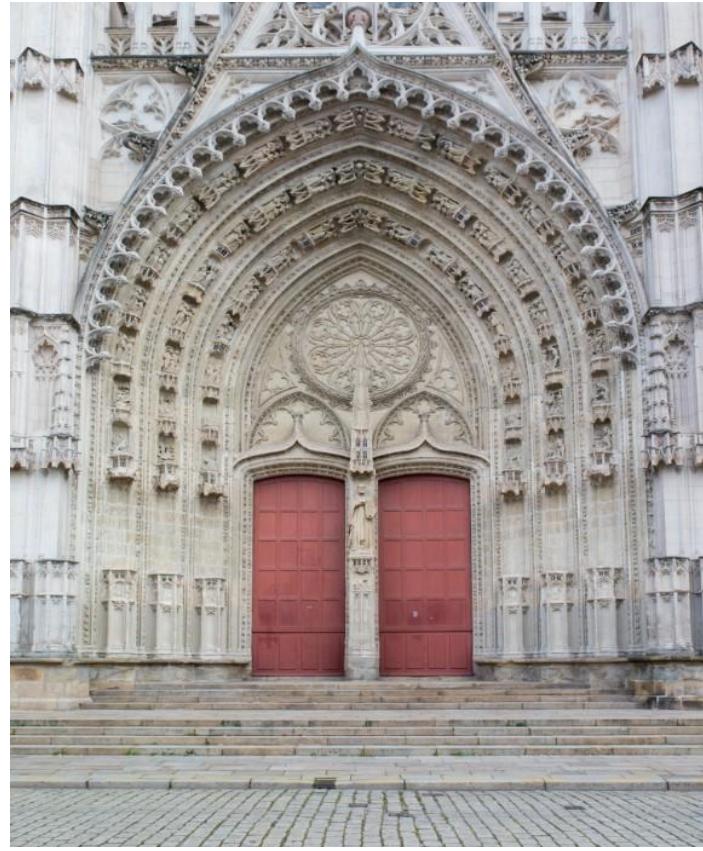
آرشمام لولوهری ۹۹۱۰۲۱۵۶

محمد رضا سید نژاد ۹۹۱۰۱۷۵۱

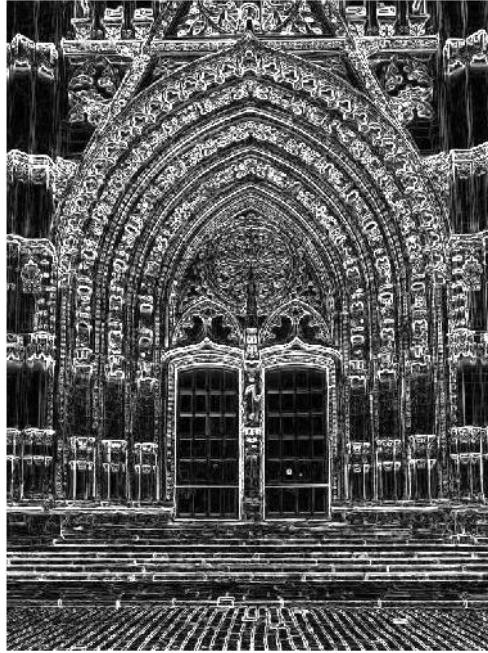
## ۱.۱

در روش sobel با استفاده از کرنل های مذکور در صورت سوال، و کانولو کردن آنها با تصویر سیاه و سفید ورودی، مشتق شدت نور در هر نقطه از تصویر، به ترتیب در راستا های  $X$  و  $Y$  محاسبه میشود. این دو مقدار در ماتریس های  $G_x$  و  $G_y$  ریخته میشوند و اندازه مشتق هم از رابطه مذکور در سوال، بدست می آید و در  $G$  ریخته میشود. برای این بخش، تابع sobel در فایل Q1 ساخته شده است که با گرفتن نام فایل عکس (که در همان محل فایل m. قرار دارد)، اینکار را انجام میدهد. ابتدا کرنل ها در  $M_x, M_y$  ریخته شده اند. فایل عکس با imread خوانده شده و در img ریخته میشود و در صورتی که رنگی باشد، با rgb2gray سیاه سفید میشود. سپس ماتریس های گرادیان در جهات افقی و عمودی را با استفاده از کانولوشن دو بعدی conv2، تشکیل داده و در  $G_x, G_y$  میریزیم. همانطور که ذکر شد، اندازه مشتق را بدست آورده و در  $G$  میریزیم و این ماتریس، تصویر نهایی ماست.

علت اینکه این روش کار مورد نظر را انجام میدهد این است که به وضوح، در لبه های هر جسم یا شی در عکس، تغییرات نور و رنگ خیلی شدیدتر است. در نتیجه در لبه ها مشتق بزرگتری داریم. ماتریس  $G$  این مشتق ها را مدل میکند و در لبه ها که مشتق بزرگتری داریم، عدد بزرگتری قرار میدهد و در نتیجه آن لبه ها را به رنگ سفید تر و روشن تر نشان میدهد. نتیجه را به ازای تصویر انتخاب شده *edgePic.jpg*، در زیر میبینیم (تصویر اول، تصویر اصلی است):

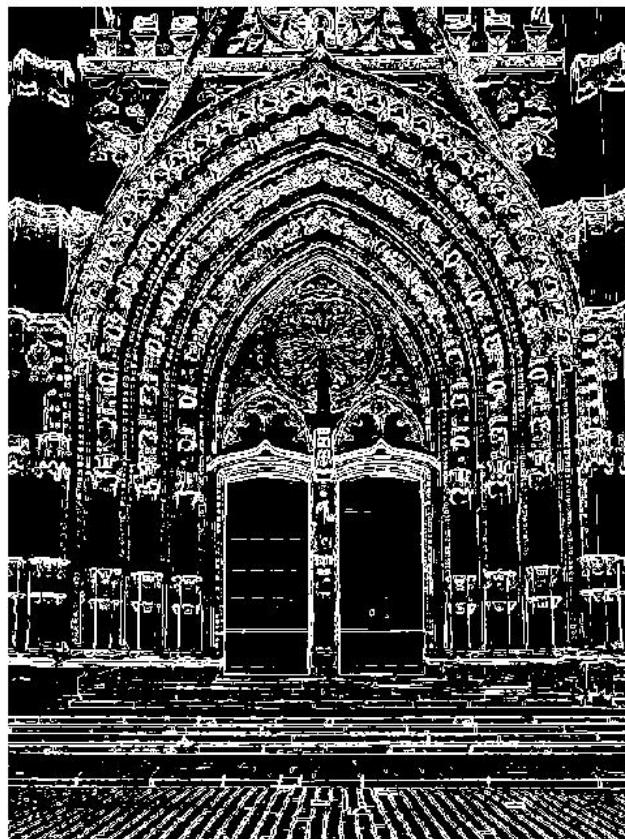


**edge-detected image using sobel**



میتوان با استفاده از `imbinarize` و دادن ماتریس تصویر نهایی به آن، حد آستانه (threshold) ای برای عدد متناظر با هر پیکس در نظر گرفت. در تصویر بالا میبینیم که جاهایی که مرز نیستند، تیره تر اند(عدد کوچکتری دارند) و یا شدت سفیدی کمتری دارند. با استفاده از این تابع، جاهایی که مرز نیستند و سفیدی کوچکی دارند، فیلتر شده و حذف میشوند (کاملاً تیره میشوند) اما مرز ها که عدد پیکسل به اندازه کافی بزرگ است، از فیلتر عبور کرده و تقویت نیز میشوند(کاملاً سفید میشوند). به بیان دیگر، مقدار پیکسل ها به صورت صفر و یکی تنظیم میشود و بدین ترتیب به صورت زیر، مرز ها واضح تر میشوند (البته اینکار گاها موجب حذف برخی لبه های جزیی تر میشود اما لبه های اصلی را خیلی بهتر نشان میدهد. در این تصویر، نتیجه به وضوح بهتر میشود):

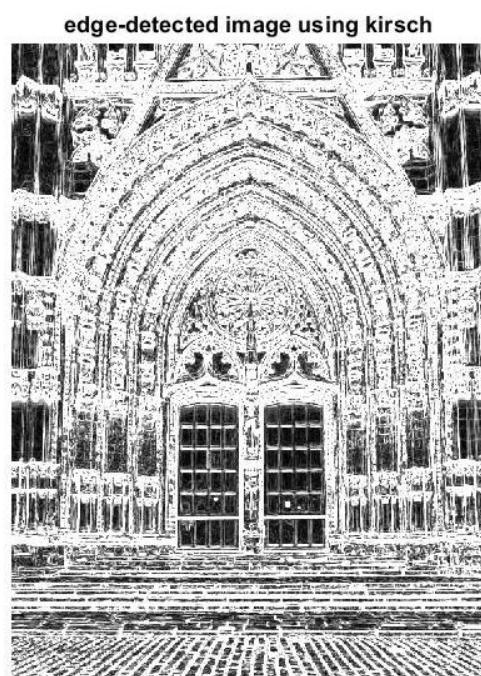
edge-detected binarized image using sobel



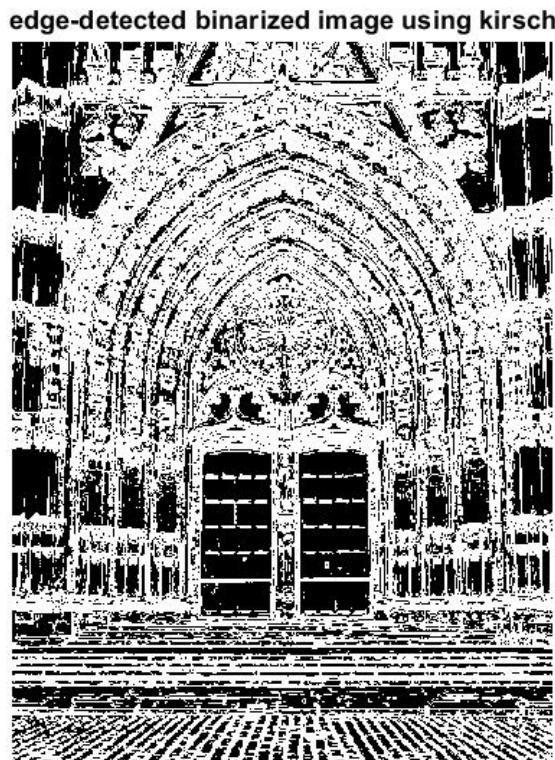
## :1.2

این بار فرایند به گونه ایست که با استفاده از کرنل های تعریف شده کانولوشن، مشتق شدت نور بجای صرفا راستای  $\nabla_x$ ، در راستای  $\nabla_y$  جهت اصلی گرفته میشود. هریک از این کانولوشن های دو بعدی، تغییر شدت نور در جهت متناظر با خود را میدهد. سپس ماتریس نهایی را ماکزیمم این مشتق ها میگیریم. یعنی برای هر پیکسل، مشتقی انتخاب میشود که بیشتر از سایر جهات است.

کرنل ها در  $g_{1-8}$  و با استفاده از تابع چرخش  $90^\circ$  درجه ای  $rot90$  تعریف شده اند. فایل را به طرز مشابه قبل خوانده، از حالت رنگی در آورده و سپس کانولوشن های مختلف را در  $h_{1-8}$  ریخته ایم. سپس در ادامه، خروجی را در هر پیکسل، ماکزیمم مقادیر  $h_1$  تا  $h_8$  گرفته ایم (با استفاده از  $\max$ ). بدین ترتیب تابع  $kirsch$  کامل است و در سکشن Q1.2 به طرز مشابه  $sobel$  آن را صدا زده ایم:



## پس از اعمال `:imbinarize`



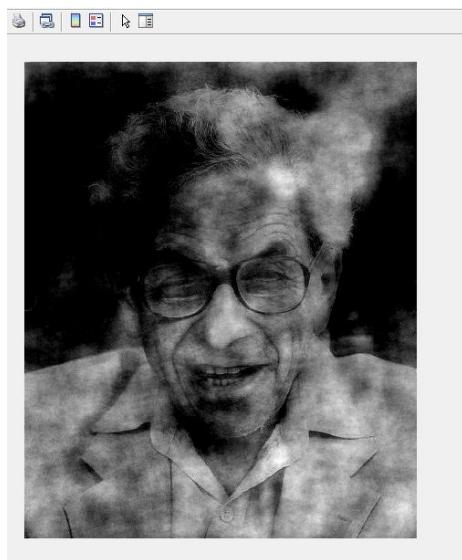
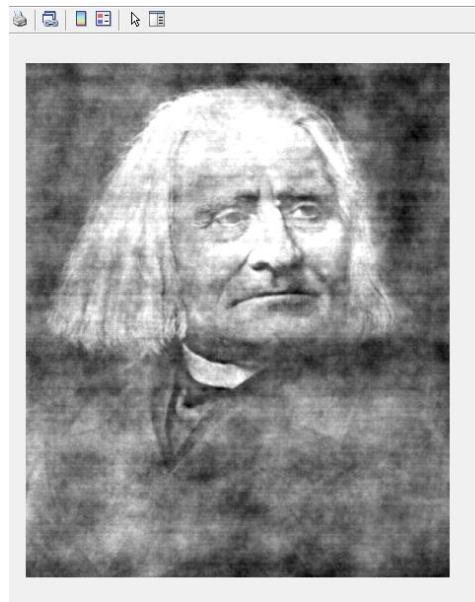
این تصویر بدلیل داشتن لبه های مختلف، و در جهات مختلف، برای مقایسه این دو عملگر آورده شده است. در بالا دیدیم که عملگر `kirsch` تا حدی بهتر عمل کرد. مخصوصاً اگر به لبه های موجود در ستون های بالای تصویر، در دو سمت چپ و راست دقیق کنیم، انتظار داریم بخاطر وجود این لبه ها، خط های سفیدی ایجاد شود. در روش `sobel`، چه بدون فیلتر و چه با فیلتر `binarize`، این خطوط چندان دیده نمیشوند. اما در روش `kirsch` این خطوط به وضوح دیده میشوند. در واقع در `sobel` ما فقط مشتق را در جهت افقی و عمودی میبینیم که در همان جهت عمودی هم (بدلیل عمودی بودن ستون ها)، مشتق بسیار کوچکی داریم. در نتیجه اعداد حاصله برای این لبه ها بسیار کوچک میشود اما در روش `kirsch` از ۸ جهت تغییرات را بررسی میکنیم و ماکزیمم آنها را لحاظ میکنیم. طبیعتاً انتظار هم میرود که عدد بیشتری برای این

پیکسل ها لحاظ شود. نمونه دیگر تفاوت این دو روش هم در لبه های موجود روی درب هاست.

اما از لحاظ زمان اجرا، طبیعتا بدلیل اینکه در روش kirsch ما باید ۸ بار کانولوشن بگیریم و سپس تک تک ماتریس های حاصله را با هم مقایسه کنیم، و نیز اینکه ابعاد ماتریس های مربوط به تصاویر، معمولاً بزرگ است، زمان بیشتری طول میکشد تا فرایند پیاده و اجرا شود. در sobel فقط دوبار کانوالو صورت میگیرد و یکی از دلایلی که همچنان از این روش استفاده میشود، همین سرعت بالای این فرایند است.

## سوال ۲

خروجی های سوال به شکل زیر است :



همانطور که مشاهده می شود تصویر اول که فاز تصویر دوم را دارد بسیار شبیه به تصویر دوم است و تصویر دوم که فاز تصویر اول را دارد بسیار شبیه به تصویر اول است . این مسئله به خاطر آن است که فاز تصویر محل وقوع جزئیات را بر عهده دارد در حالی

که اندازه تبدیل فوریه سایر ویژگی ها را بر عهده دارد . این مسئله باعث می شود که جزئیات تصویر توسط فاز تعیین شود در حالی که رنگ بندی تصاویر ایجاد شده به علت اینکه فاز تصویر اصلی را ندارند چنان صحیح نباشد و تصویر مانند یک تصویر نویزی شود .

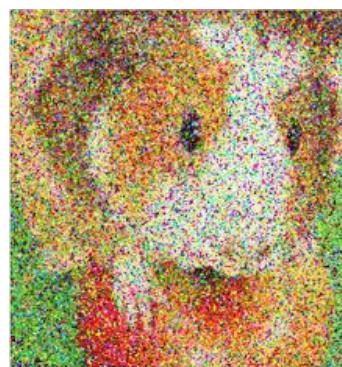
:۳

## الف)

۱. نویز salt & pepper ، تنها برخی از پیکسل های تصویر را دچا نویز میکند، اما این پیکسل ها دچار نویز شدیدی میشوند. در واقع برخی از پیکسل های تصویر به صورت پراکنده، رنگ کاملا سفید یا کاملا سیاه میگیرند که نام این نویز نیز بخاره همین رنگ هاست. نمونه ای از آن به صورت زیر است:



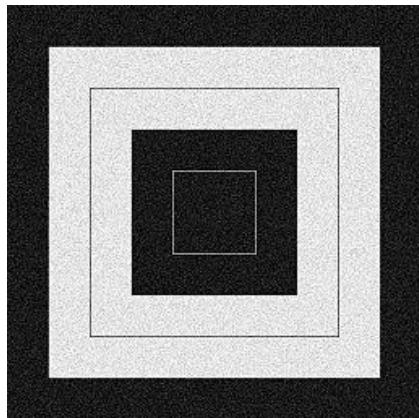
البته در تصاویر رنگی، چون rgb داریم، این نویز ها میتوانند به رنگ های کاملا قرمز، کاملا آبی یا کاملا سبز باشند:



۲. نویز گاوی، نوعی نویز است که pdf ای با توزیع نرمال یا گوسی دارد. در واقع میزان تیرگی و روشنی رنگ هر پیکسل در این نویز، از یک توزیع نرمال تبعیت میکند:

$$P_G(z) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(z-\mu)^2}{2\sigma^2}}$$

که در آن،  $z$  میزان grey level (تیرگی) پیکسل،  $\mu$  میانگین و  $\sigma$  انحراف معیار این کمیت است. به این نویز نویز الکترونیک هم گفته میشود چرا که در تقویت کننده ها بیشتر دیده میشود. مرجع آن معمولاً جنبش اتم ها و تشتشع امواج از سوی اجسام گرم است. نمونه ای از این نویز در زیر قابل مشاهده است:



۳. برای نویز پواسون، باید توجه کنیم که نور شامل تعداد بسیار زیادی فوتون است. بنابراین تصاویر دیجیتالی که ما میبینیم، توسط این فوتون ها قابل رویت میشوند. هرچه تعداد فوتون هایی که به کار گرفته میشوند تا یک پیکسل از تصویر را نمایش دهند، بیشتر باشد، چیزی که ما از آن پیکسل میبینیم، به تصویر واقعی اش شباهت بیشتری دارد. حال در نویز پواسون، تعداد فوتون بکار رفته برای نمایش هر پیکسل از توزیع احتمالاتی پواسون تعییت میکند:

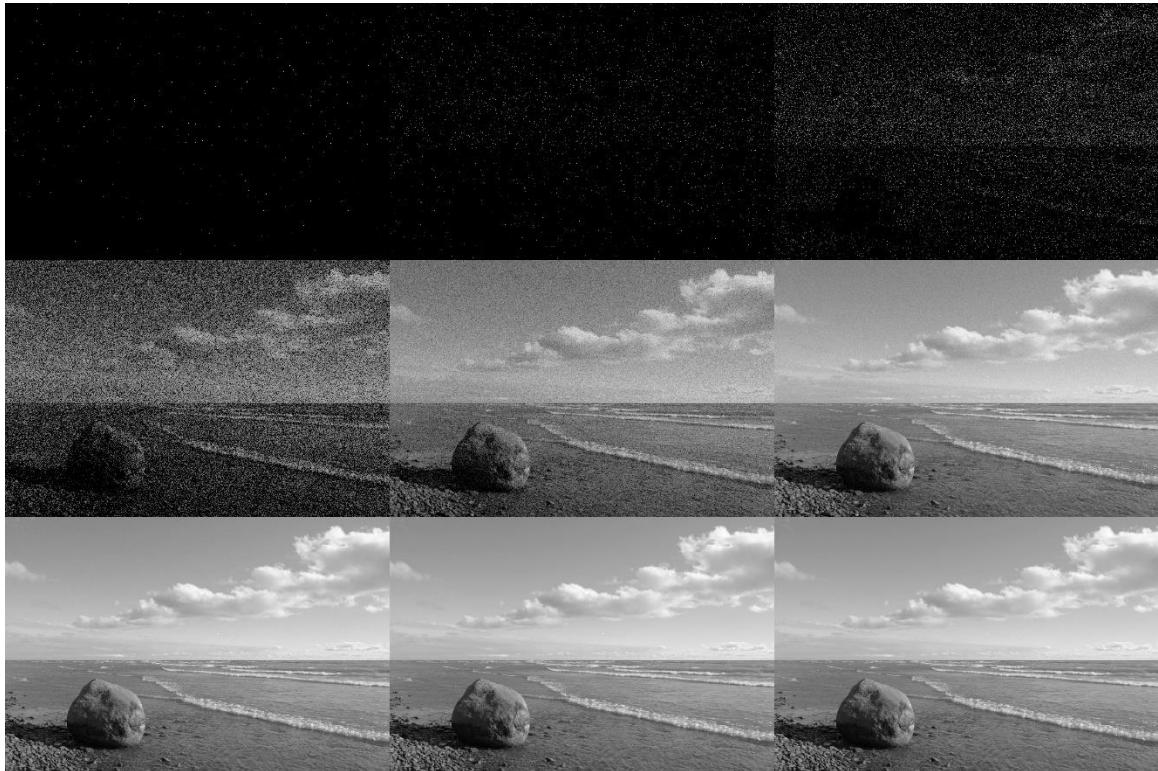
**Poisson Probability Distribution**

$$P(x) = \frac{\lambda^x e^{-\lambda}}{x!}$$

$\lambda$ - mean number of successes over a given interval  
 $Var(X) = \lambda$

در مبحث ما، لاندا میانگین تعداد فوتون هایی است که برای هر پیکسل استفاده میشود. حال هرچه تعداد فوتون های یک پیکسل کمتر باشد، رنگ آن پیکسل به سمت تیرگی (کم نور شدن) میرود و از رنگ واقعی اش فاصله میگیرد. بدیهی است هرچه لاندا یا میانگین تعداد فوتون ها بیشتر باشد، نویز کمتری دیده خواهد شد. در تصاویر زیر نمونه

هایی از این نویز را میبینیم که از بالا به پایین و از چپ به راست، لاندای توزیع تعداد فوتون ها بیشتر میشود و در نتیجه نویز کاهش می یابد:



۴. نویز speckle بیشتر به خود تصویر برمیگردد و یک نویز ذاتی است که در درون تصویر وجود دارد. عکس هایی که ما میگیریم، نور های بازتاب شده از سطوح را تشخیص میدهند. این نور ها از هر سطحی، به طور پراکنده بازتاب میشوند و به دلیل رزولوشن محدود عکاسی ما، همواره مقداری از این امواج را دریافت میکنیم. سیگنال های دریافت شده توسط چشم یا دوربین ما، به طور سازنده(برابر بودن فاز امواج و در نتیجه تشديدي دامنه امواج) و یا ويرانگر(داشتن فاز های مخالف و رساندن دامنه امواج به حداقل) بر هم اثر كرده و با هم جمع میشوند و در نتیجه در پیکسل هایی که تداخل سازنده رخ داده، ممکن است رنگ پیکسل روشن تر از حالت عادي شود و در تداخل های ويرانگر، با کاهش دامنه موج نور، رنگ پیکسل به سمت تيرگي ميرود. اين پدیده را نویز speckle مینامند.

در تصاویر زیر به ترتیب بالا، نویز ها را با استفاده از `imnoise` وارد کردن نام نویز و برخی مشخصات احتمالی، تولید کرده ایم:







پارامتر های قابل تنظیم مانند میانگین و واریانس و... به گونه ای تنظیم شده اند که هم تصویر و هم نویز قابل مشاهده باشند. در مورد پواسون لازم به ذکر است که میانگین نویز های پواسون بسته به **type** اعداد موجود در تصویر ما، به طور خودکار توسط متلب تنظیم میشوند و چیزی در دست ما نیست. اما به هر حال مشاهده میکنیم که تیرگی هایی در پیکسل های مختلف تصویر ایجاد شده که کمبود فوتون های نمایش دهنده را مدل میکند. این تیرگی ها نسبت به عکس اصلی، به ویژه در قسمت آسمان تصویر کاملاً واضح اند.

(ب)

• **Median Filter**: یک فیلتر غیر خطی است که برای کاهش نویز سیگنال یا تصاویر بکار میرود. بطور خلاصه، کلیت کار این فیلتر این است که میانه‌ی داده های همسایه‌ی هر داده را با خودش جایگزین میکند. برای مثال اگر سیگنال تک بعدی  $x = (2, 3, 80, 6, 2, 3)$  را در نظر بگیریم، همسایه عدد ۲، ۳ و ۸۰ و خود

۳ است که میانه آنها برابر با ۳ است. اما برای داده ۸۰، میانه بین ۸۰ و ۳ و ۶ است و بجای ۸۰، ۶ قرار میگیرد. میبینیم بدین ترتیب، با محاسبه میانه، داده‌ی بسیار بزرگ و پرت که غالباً نقش نویز دارد، از سیگنال حذف شده و داده‌ای منطقی‌تر جایگزینش میشود. با ادامه این فرایند، بدون در نظر گرفتن مقادیر مرز، به خروجی  $(3, 6, 6, 3) = 6$  میرسیم. در سیگنال‌های دو بعدی نیز، مانند تصاویر، همسایه‌ها، همان پیکسل‌های مجاور هر پیکسل هستند. اگر در یک پیکسل نویزی داشته باشیم، مقدار عددی این پیکسل با پیکسل‌های اطرافش تفاوتی معمولاً زیاد دارد و در نتیجه با این روش، این مقدار به عددی معقول‌تر متغیر میشود و نویز کاهش می‌یابد. روش‌هایی برای مرز‌های سیگنال، این است که مقدار خودشان را دست نخورده نگه داریم، آنها را پس از عبور از فیلتر حذف کنیم یا اینکه مثلاً آنها را هم فقط با همسایه‌های محدود خود، میانه بگیریم. هر یک از روش‌ها ممکن است استفاده شود. مزیت این روش این است که لبه‌های تصاویر را همچنان حفظ کرده و نشان میدهد. فرایند زیر نمونه‌ای از اثر این فیلتر است:



عملکرد این فیلتر، به ویژه روی نویز‌های گاوی، رندوم و salt & pepper خوب است. این فیلتر‌ها انواع مختلف دارند برای مثال اگر نوع آن  $2^*2$  باشد، خطوطی با ضخامت ۲ پیکسل را حذف میکنند. اگر  $3^*3$  باشند خطوط با ضخامت ۳ پیکسل و ... . این اتفاقات ناشی از این است که هر بار، ماتریس همسایه‌ما سایز متفاوتی پیدا میکند و داده‌های بیشتری را میانه میگیریم.

• Gaussian Filter در توزیع گاوی دو بعدی، با فرض صفر بودن میانگین داریم:

In 1D:

$$g_\sigma(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{x^2}{2\sigma^2}\right)$$

In 2D:

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

که از ضرب دو توزیع گاوی در جهات  $x$  و  $y$  میتوان به توزیع دو بعدی رسید. فیلتر گاوی میگوید برای رفع نویز، از میانگین گیری موزون با پیکسل های همسایه استفاده کنیم. سایز کرنل به ما میگوید که ماتریس همسایه ها را با چه ابعادی انتخاب کنیم و ضرایب میانگین گیری، با دور شدن از پیکسل اصلی و حرکت به سمت همسایه های دورتر، با فرمول بالا کوچک شود. این عمل موجب میشود نویز هایی که مقادیر پرت در برخی سیگنال ها ایجاد کرده اند، با میانگین گرفتن با همسایه ها، به مقادیر معقول تری برسند و نویز کاهش یابد. هرچند زیاد کردن اثر فیلتر (با تنظیم انحراف معیار و سایز کرنل)، مانند فیلتر median میتواند تصویر را تار تر کند. در متلب این عمل میانگین گیری را میتوان با استفاده از کانولوشن دو بعدی انجام داد. کرنل ما در کانولوشن باید ماتریس ضرایب میانگین گیری باشد، به گونه ای که پیکسل(های) وسط باید بیشترین ضریب را داشته باشند و در درایه های گوشه تر، ضرایب کمتری طبق فرمول بالا بدست آیند.

(ج)

(۱) تابع این بخش، gaussianFilter است که نام فایل عکس (که باید در محل اجرای کد باشد) و سایز کرنل یا همان ماتریس همسایه و نیز انحراف معیار فرمول توزیع گاوی را میگیرد و فیلتر مورد نظر را اعمال میکند.

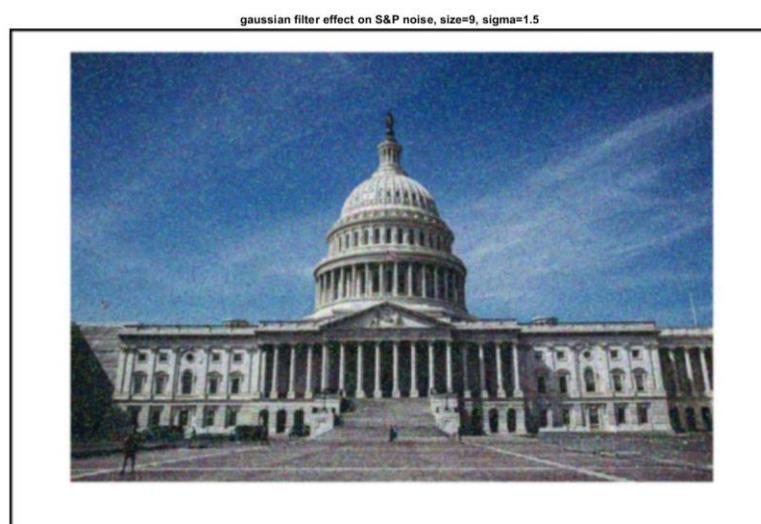
ابتدا فایل خوانده شده و به دابل تبدیل میشود. سپس برای رفع مشکل میانگین گیری در لبه ها، صفر هایی را به آنها اضافه میکنیم. اضافه کردن این صفر ها موجب میشود که حین کانولوشن و میانگین گیری، همسایه هایی که وجود ندارند، صفر باشند. حال ماتریس کرنل را میسازیم. اگر سایز ماتریس فرد بود، یک درایه مرکزی داریم که جایگاهش را center نامیده ایم. سپس از آنجا که ماکریمم ضریب در مرکز ماتریس قرار دارد،  $u, x$ ,  $i$ -center,  $j$ -center تعريف میکنیم تا  $(i, j)$  به درستی و بر اساس فرمول مذکور در بخش ب، برای هر درایه ساخته شود. اگر هم سایز کرنل زوج بود، یک چهارم از ماتریس کرنل را با استفاده از firstQuart و با فرمول مشابه قبل، میسازیم. سپس با قرینه کردن این ماتریس در جهات مختلف، ضرایب سه بخش دیگر کرنل نیز بدست می آید (در این حالت، چهار درایه ای مرکزی، بیشترین ضرایب را دارند). سپس در mean\_kernel، کرنل را بر مجموع ضرایب تقسیم میکنیم تا میانگین گیری به درستی انجام شود. سپس لایه های قرمز و سبز و آبی تصویر را جدا میکنیم و عمل conv2 با mean\_kernel ساخته شده را روی هریک پیاده میکنیم. با یک حلقه تو در تو، مقادیر جدید حاصل از کانوالو را در تصویر جدید میریزیم و سپس به newIm، بخشی از ماتریس ها را که میانگین گیری شده اند (بخشی که شامل صفر های اضافه شده نیست) میدهیم.

(۲) تابع این بخش، medianFilter است که نام فایل عکس (که باید در محل اجرای کد باشد) و سایز کرنل یا همان ماتریس همسایه را میگیرد و فرایнд را روی عکس انجام میدهد. ابتدا عکس را در img ریخته و دابل میکنیم. سپس تعداد سطر ها و ستونهای عکس، و ماتریس های منتظر با رنگ های قرمز، سبز و آبی اش را بدست آورده ایم. توجه داریم که عملگر را روی هر سه ای این ماتریس ها باید اعمال کنیم تا در هر رنگی، میانه همسایه های هر پیکسل بدست آید. با مثال kernelSize=3 بقیه فرایند را توضیح

میدهیم. در این شرایط، پیکسل اصلی در وسط یک ماتریس  $3 \times 3$  قرار میگیرد و در نتیجه در گوشه هایی که سطر یا ستون آنها، دو تای اول یا آخر است، ماتریس ما کاملا درست نمیشود و در نتیجه فرایند را برای سایر پیکسل ها انجام خواهیم داد. بدین ترتیب رنج تغییر  $z$ ، در حلقه های فور تعیین میشود. سپس باید در هر مرحله، برای هر سه ماتریس قرمز و سبز و آبی جدیدمان، درایه  $a_{ij}$  را مشخص کنیم. ابتدا از تصویر اصلی، درایه  $a_{ij}$  را در نظر میگیریم. ماتریس همسایه متناظر با آن، ماتریسی است که اولین و آخرین درایه آن به ترتیب  $a_{(j-1)(i-1)}$  و  $a_{(j+1)(i+1)}$  هستند. این ماتریس را در داخل حلقه های تودرتو برای هر سه رنگ درست میکنیم. برای محاسبه میانه های این ماتریس، ابتدا این ماتریس را با `reshape` به فرم برداری تبدیل کرده و سپس از `median` استفاده میکنیم. با تکرار این مراحل، تمام درایه های هر سه ماتریس جدید، به روش `median filter` ساخته میشود و تصویر نهایی برگردانده میشود.

(۳)

برای gaussian، چهار عکسی که در بخش های قبلتر با ۴ نوع نویز ایجاد شده بودند را با این فیلتر میدهیم و انحراف معیار و سایز کرنل را برای هریک، در مقدار مناسب تنظیم میکنیم:



gaussian filter effect on gaussian noise, size=7, sigma=2.2



gaussian filter effect on poisson noise, size=5, sigma=1.2



برای واضحتر شدن اثر روی پواسون، تصویر زیر را هم بررسی میکنیم:



gaussian filter effect on poisson noise, size=5, sigma=1

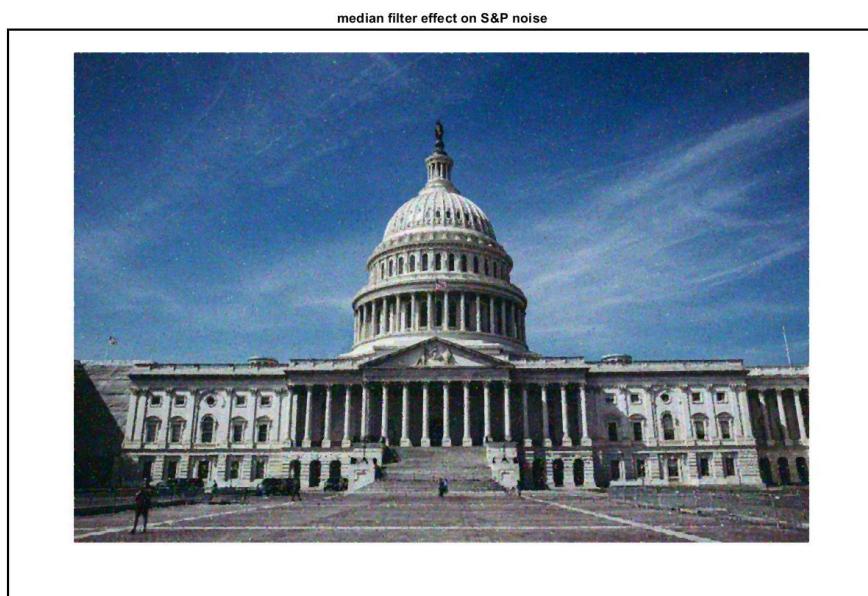


gaussian filter effect on speckle noise, size=7, sigma=1.2



توجه داریم چون در عکس انتخابی ما حاشیه سفید وجود داشت، مشکل گوشه ها در خود عکس دیده نشد اما برای اینکه برای هر تصویری (با یا بدون حاشیه) بتوان میانگین گیری را انجام داد، حاشیه ی صفر را در تابع به عکسمان اضافه کردیم. همین حاشیه صفر در برخی عکس ها باعث کاهش میانگین در گوشه های سفید رنگ، و ایجاد تیرگی های خیلی باریک شده است.

برای median نیز چهار عکسی که بالاتر با ۴ نوع نویز ایجاد شده بودند را به ورودی تابع میدهیم و خروجی را بررسی میکنیم:



median filter effect on gaussian noise



median filter effect on poisson noise



median filter effect on speckle noise



برای بهتر دیده شدن اثر فیلتر روی پواسون، تصویر زیر را هم بررسی میکنیم:





کرنل برای S&P و تصویر پواسون اول روی ۳، و برای بقیه روی ۵ تنظیم شده است. میبینیم هرچه سایز کرنل را بیشتر کنیم، با اینکه نویز ها کمتر میشوند، اما تصاویر به سمت تاری و ناواضحی پیش میروند. بر این اساس بهترین سایز برای هر تصویر انتخاب شده است. از مقایسه این عکس ها با نسخه نویزی شان به وضوح میتوان بهبود و کاهش نویز را مشاهده کرد.

۴) محاسبات این بخش در سکشن Q3 part c\_4 انجام شده است. هر خط کد در کامنت، توضیح داده شده است. به صورت کلی، تصاویری که در بخش های قبل بدست آمده بود را دوباره از فایل خوانده، و هریک را همراه با تصویر اصلی بدون نویز به تابع  $\text{SNR}_{\text{cal}}$  میدهیم. این تابع، فرمول ذکر شده برای  $\text{SNR}$  را برای تک تک درایه های تصاویر اصلی و فیلتر شده(یا نویزی) اعمال کرده و  $\text{snr}$  محاسبه شده را برمیگرداند. نتایج به صورت زیر است که میبینیم  $\text{SNR}$  ها پس از عبور از فیلتر، نسبت به حالت نویزی خود، بیشتر شده اند.

برای تصاویر نویزدار:

Noisy image	SNR
Salt & pepper	<b>7.66</b>
Gaussian	<b>6.68</b>
Poisson	<b>11.64</b>
Speckle	<b>7.43</b>

برای تصاویر فیلتر شده:

Filtered image	SNR_median filt	SNR_gaussian filt
Salt & pepper	<b>9.59</b>	<b>8.52</b>
Gaussian	<b>8.13</b>	<b>8.34</b>
Poisson	<b>11.84</b>	<b>10.13</b>
Speckle	<b>9.03</b>	<b>8.43</b>

(۵) یک روش دیگر حذف نویز، که البته یک فیلتر خطی است، mean filter است. ساختار کلی فیلتر مشابه Gaussian filter است با این تفاوت که در فیلتر گاوسی، ضرایب از بیشترین ضریب در پیکسل مورد نظر، تا کمترین ضریب در پیکسل های دورتر، متغیر بود. اما در این روش، سایز کرنل تعیین میشود و سپس تمام پیکسل های همسایه، ضریبی برابر با خود پیکسل دارند. به بیان دیگر، بجای میانگین موزون، میانگین ساده داریم. این فیلتر را با تبدیل کردن ماتریس کرنل موجود در فیلتر گاوسی، به یک ماتریس تمام ۱، میتوان ساخت. در سشکن توابع، تابع meanFilt، همین کار را انجام داده است و در آخرین سکشن (قبل از توابع) نیز این تابع اجرا شده است. یک نتیجه‌ی این فیلتر را با انتخاب سایز کرنل مناسب (۵) بر روی تصویر نویزدار salt&pepper میبینیم:



روش جالب دیگر برای رفع نویز، **conservative filter** است. قاعده کلی در این فیلتر این است که مقدار هر پیکسل، با پیکسل های همسایه خود (بسته به سایز کرنل) سنجیده میشود. این همسایه ها، مقدار مینیمم و ماکزیمم دارند و نحوه کار این فیلتر بدین شکل است که اگر خود پیکسل، بین مینیمم و ماکزیمم همسایه هایش بود، احتمالاً دچار نویز و تغییر نشده و در نتیجه مقدار پیکسل تغییر نمیکند. اما اگر مقدار پیکسل، از ماکزیمم بالاتر بود یا از مینیمم پایین تر بود، آن پیکسل را نویزی فرض کرده و مقدار پیکسل را به ترتیب در ماکزیمم و مینیمم تعیین شده قرار میدهد. از آنجا که این روش، میانگین گیری ای بین پیکسل های اطراف انجام نمیدهد، لبه های تصویر را به خوبی حفظ میکند. این روش مخصوصاً برای نویز **salt&pepper** بسیار مناسب است چون در این نوع نویز، پیکسل های نویزی به شدت تغییر رنگ میدهند (مقادیر پیکسل، یا در ماکزیمم قرار میگیرد یا در مینیمم) و در نتیجه این فیلتر میتواند تا حد خوبی، خروجی پیکسل های نویزی از ماکزیمم یا مینیمم مشان را تشخیص دهد. تابع این فیلتر در سکشن توابع با نام **cosFilter** نوشته شده و توضیحاتش در کامنت نوشته شده است. یک فیلتر جدید **salt&pepper** برای این فیلتر ساخته ایم و سپس با کرنل با سایز  $3 \times 3$ ، رفع فیلتر کرده ایم. در زیر تصاویر نویزدار و فیلتر شده را میبینیم (به ویژه در قسمت آسمان تصویر، رفع نویز مشهود است):



conservative filter effect on S&P noise, size=3



#### سوال 4

4.1: هدف از این الگوریتم پیاده کردن خوش‌هایی است که اعضای آن به هم مشابه باشند یا به عبارتی تفاوت یا فاصله بین اعضای خوش‌های مینیمم باشد. پس اگر بتوانیم حاصل جمع فاصله اعضای خوش‌های مرکز دسته را مینیمم کنیم به هدف خود رسیده‌ایم.

4.2: روش کلی پیاده سازی الگوریتم به این صورت است که در ابتدا تعداد خوش‌هایی که می‌خواهیم تصویر یا به طور کلی ماتریس به آن تقسیم شوند را تعیین می‌کنیم. سپس به آن تعداد مرکز خوش‌های تعیین می‌کنیم. تعیین مرکز خوش‌ها می‌تواند به شکل رندم صورت بگیرد. به طور کلی اگر تعیین مرکز خوش‌ها به طوری نباشد که این مراکز به شدت به هم شبیه باشند خوش‌بندی مشکلی نداشته و نهایتاً الگوریتم در زمان کمتر یا بیشتر اجرا می‌شود. اما اگر تعداد مرکز خوش‌ها نسبت به تعداد اعضاء زیاد باشد و مرکز خوش‌ها به هم نزدیک باشند این امکان وجود دارد که الگوریتم صحیح کار نکند. در ادامه الگوریتم به این صورت است که فاصله نقاط را تا مرکز خوش‌ها به دست می‌آوریم و با مقایسه فاصله‌ها خانه‌ها را به خوش‌های اختصاص می‌دهیم که فاصله مینیمم دارد. به این ترتیب خوش‌بندی انجام می‌شود. سپس میانگین خوش‌های را حساب می‌کنیم و به عنوان مرکز خوش‌های جدید ارائه می‌کنیم. الگوریتم را آنقدر تکرار می‌کنیم تا مرکز خوش‌های جدید با مرکز خوش‌های قبلی برابر شود.

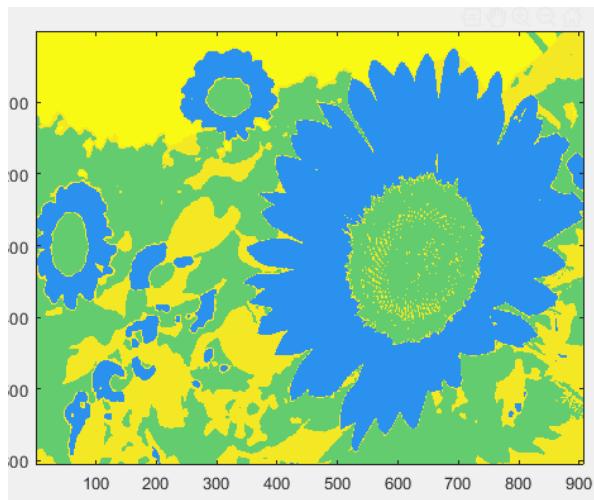
4.3: این تابع تحت عنوان cluster\_1 نوشته شده که کار خواسته شده را انجام می دهد تابع دیگری که تحت عنوان cluster موجود است نقاط هم خوش را با رنک مربوط به مرکز خوش جایگزین می کند بنابراین خروجی آن یک عکس است و سه بعدی است .

توضیحات مربوط به این دو تابع به شرح زیر است :

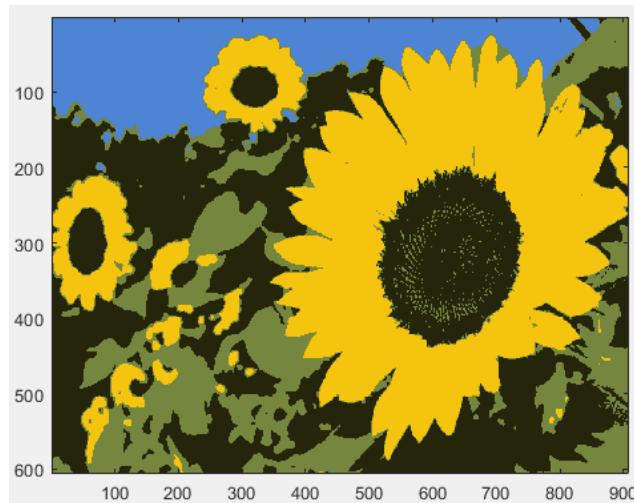
تمام عملیات الگوریتم در داخل while تعریف می شود . این while تا زمانی که دو مرکز خوش تکراری تولید نشده ادامه پیدا می کند و این مسئله توسط متغیر check بررسی می شود . در یک for تک تک مرکز خوش ها را بررسی می کنیم . در بخش اول for بررسی می کنیم که آیا این اولین بار اجرا الگوریتم است یا نه . اگر اولین بار است لازم است مرکز خوش توسط بردار های  $a, b, c$  تعیین شود و در غیر این صورت تعیین مرکز خوش توسط میانگین گیری بین اعضای خوش تعیین می شود . مرکز خوش فعلی در centers تعیین می شود و همه مرکز خوش ها در تجمعیع می شوند . پس از تعیین مرکز خوش نوبت به محاسبه فاصله ها می رسد . مجذور فاصله ها قرار است در متغیر d ذخیره شود . الگوریتم محاسبه فاصله به این صورت است که  $a, b, c$  قرار است به ترتیب مجذور فاصله مولفه قرمز سبز و آبی را از مولفه های مشابه در تصویر به دست آورد . جمع این سه فاصله را به ما می دهد . در نهایت متغیری تحت عنوان Min تعریف می کنیم که کم ترین فاصله ها را تشخیص می دهد . سپس این کم ترین فاصله ها را با فاصله از تک تک مرکز خوش ها انطباق می دهیم تا مشخص شود هر نقطه متعلق به کدام خوش است . به این ترتیب خوش بندی

در ماتریس سه بعدی `clus` ثبت می شود. در بخش آخر شرط توقف الگوریتم را چک می کنیم . `centers_1` , `centers_2` به ترتیب نمایانگر مرکز خوشة قدیم و جدید هستند . برابری آن ها با دستور `isequal` چک می شود .

4: تصاویر خروجی دوتابع ایجاد شده به شکل زیر است



خروجی تابع دوم که مورد نظر سوال است



خروجی تابع اصلی که از جایگزاری رنگ مرکز خوشه به جای اعضای آن به دست می آید.

۴.۵: این الگوریتم یکی از الگوریتم‌های ساده‌ای است که در بحث بینایی ماشین . کاربرد این الگوریتم در جدا سازی بک گراند است و در ساده‌ترین کاربرد (که مورد تحقیق این سوال هم است) تصویر را به دو بخش سیاه و سفید تقسیم می‌کند . هر چند انواع دیگری از آن هم موجود است . عملکرد این الگوریتم این چنین است که سعی می‌کند واریانس بین کلاسی را ماکزیمم کند و یا به عبارتی واریانس درون کلاس را مینیمم کند . رابطه این واریانس به شکل زیر است .

$$\sigma_w^2(t) = \omega_0(t)\sigma_0^2(t) + \omega_1(t)\sigma_1^2(t)$$

شهود این رابطه واضح هست . ما می‌خواهیم داده‌هایی را که تقریباً یک نواخت هستند یا به عبارتی واریانس کم ایجاد می‌کنند در دو دسته قرار دهیم . یعنی داده‌ها را به نحوی طبقه‌بندی کنیم که که هر دسته واریانس مینیمم داشته باشد . از طرفی بخشی از تصویر که وزن بیشتری دارد باید حساسیت بیشتری نسبت به واریانس داشته باشد . این فرمول چندان قابل پیاده کردن نیست . برای اینکه آن را در کد پیده کنیم از داده‌های زیر استفاده می‌کنیم .

$$\begin{aligned}\sigma_b^2(t) &= \sigma^2 - \sigma_w^2(t) = \omega_0(t)(\mu_0 - \mu_T)^2 + \omega_1(t)(\mu_1 - \mu_T)^2 \\ &= \omega_0(t)\omega_1(t)[\mu_0(t) - \mu_1(t)]^2\end{aligned}$$

این رابطه ایست که در کد از آن استفاده شده است  $w$  ها بیانگر وزن دسته هستند که آن را می‌توان برابر با احتمال انتخاب دسته دانست .  $\mu$  ها هم بیانگر میانگین داده‌های دسته هستند . با توجه به سیاه و سفید بودن عکس ۱۲۳ دسته بندی متفاوت را می‌توانیم

امتحان کنیم و باید  $\max$  عبارت بالا که واریانس بین کلاسی است را بیابیم زیرا واریانس کل مقدار ثابتی است و اگر این مقدار  $\max$  شود به این معناست که عبارت اولیه مینیمم شده است.

$$\omega_0(t) = \sum_{i=0}^{t-1} p(i)$$

$$\omega_1(t) = \sum_{i=t}^{L-1} p(i)$$

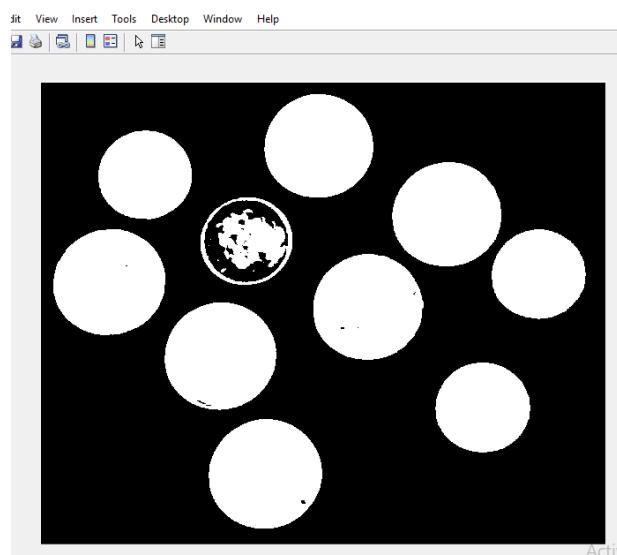
$$\mu_0(t) = \frac{\sum_{i=0}^{t-1} ip(i)}{\omega_0(t)}$$

$$\mu_1(t) = \frac{\sum_{i=t}^{L-1} ip(i)}{\omega_1(t)}$$

۱۴.۶ کد به اندازه خوبی واضح نوشته شده. هدف بررسی ۱۲۳ مقدار برای واریانس بین کلاس هاست و مقادیر وزن ها و میانگین ها در توابع مربوطه ایجاد می شود.



تصویر اولیه



تصویر ثانویه پس از فیلتر

4.7: الگوریتم otsu یک تصویر سیاه و سفید را به عنوان ورودی دریافت می کند و خروجی آن هم به این کل است در حالی که الگوریتم kmean یک تصویر معمولی را به عنوان ورودی می گیرد و همانطور که مشخص شد می تواند یک تصویر مشابه تصویر اولیه در خروجی ایجاد کند . همینطور الگوریتم kmean بیشتر برای چند خوش کردن تصاویر استفاده می شود در حالی که کاربرد اصلی otsu در تشخیص یک گراند تصاویر است . یکی از نقاط ضعف otsu در مواردی است که یک عامل نویز مانند در میان است . تصور منید صفحه ای نوشته داریم که می خواهیم بخش نوشته را از مابقی کاغذ آن جدا کنیم . حال اگر منبع نوری (مثل نور چراغ مطالعه) به آن تابیده باشد در حاشیه های صفحه سایه می افتد . اگر این الگوریتم را بر روی صفحه اجرا کنیم این سایه ها هم به عنوان نوشته تشخیص داده می شوند . یک راه حل این مشکل این است که تصویر را به چند بخش تقسیم کنیم و الگوریتم را جدا جدابر هر بخش اعمال کنیم . یک راه دیگر حل این مشکل می تواند

انتخاب یک پنجره با ابعاد مشخص و کتრک باشد که صفحه را پیمایش کند و الگوریتم را در آن اجرا کند و از بخش های تبدیل یافته مشترک میانگین بگیرد . در بخش ۴/۲ اشاره ای به یکی از نقاط ضعف kmean شده است .

## سوال ۵

ایده استفاده شده برای حل این سوال از نوع زدن حلقه for است . در این حلقه for مقادیر شیفت در راستای  $y$  ,  $x$  به همراه زاویه دوران تعیین می شود . سپس corr2 برروی تصویر جدید اعمال می شود تا میزان همبستگی جدید حساب شود . فقط باید توجه شود که corr2 برروی ماتریس های هم اندازه اعمال می شود و تابع مربوط به دوران ابعاد تصویر را عوض می کند بنابراین تصویر اولیه را که ثابت است به ابعاد تصویر سابق در می آوریم به این صورت که مربع ۷۵ در ۷۵ اولیه را ثابت گرفته و به جای سایر خانه ها صفر می گذاریم . سپس در یک if چک می کنیم که آیا مقدار جدید هم بستگی از قبلی بیشتر است یا نه و اگر بود مقدار آن در cor ذخیره می شود و تصویر مربوط به این مرحله هم در خروجی ذخیره می شود . در نهایت تصاویر را نمایش می دهیم . توجه شود که عیب این روش زمانبری آن است . اگر اطلاعاتی درباره میزان انحراف تصاویر داشته باشیم می توانیم for را بهینه کنیم .