Optional Project

Ali Ghavampour 97102293

Farhad Fallah 97102214

Sharif University of Technology

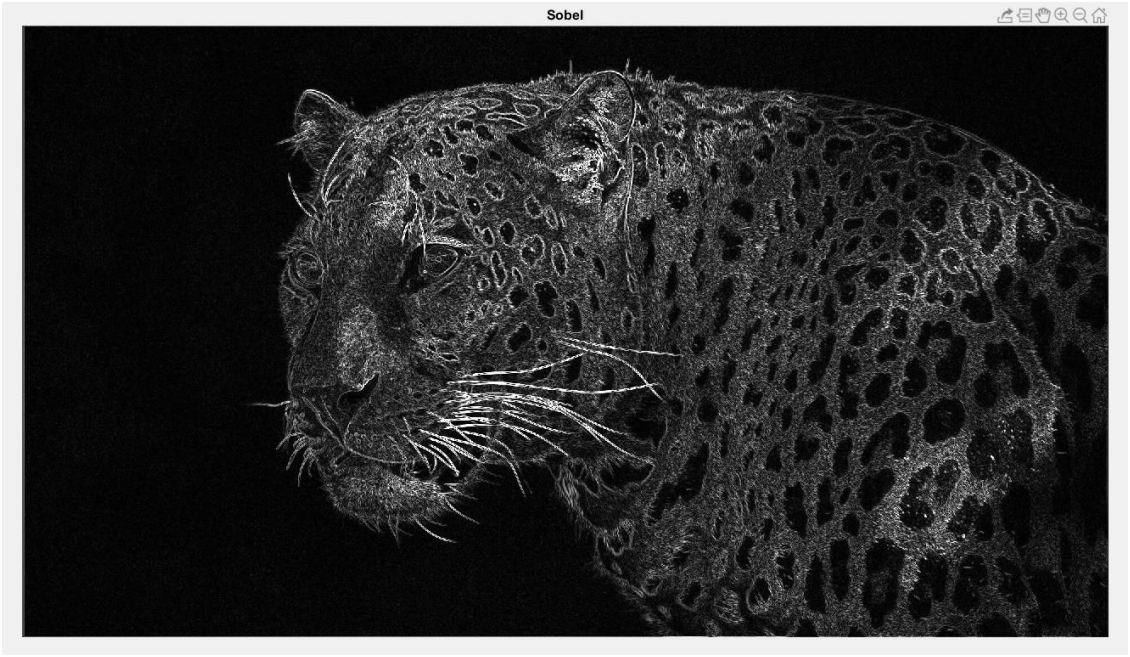
Signal and Systems

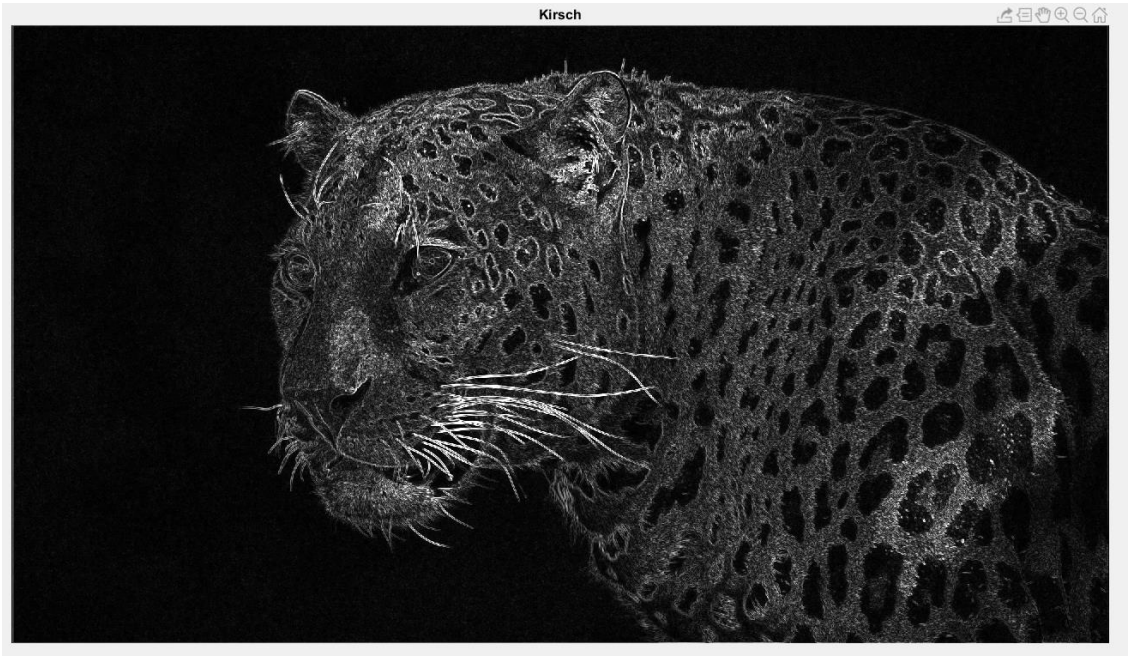
Dr. Hamid K. Aghajan

1 یافتن لبه ها در تصاویر دیجیتال

کد این بخش در فایل Question01.m قرار دارد. جفت الگوریتم ها در تابع های SobelFeldman() و Kirsch() زده شده است. از آنجا که خروجی این توابع خیلی روشن بود و تقریبا چیزی دیده نمی‌شد، نیاز به نرمالایز کردن روشنایی ها بود که برای اینکار، میانگین همه المان های ماتریس تصاویر محاسبه می‌شود و 8 برابر می‌شود (عدد 8 به صورت چشمی انتخاب شده است) و ماتریس تصاویر بر آن تقسیم می‌شود.

خروجی برای تصویر Jaguar:





همانطور که تقریبا از تصاویر مشخص است، این دو الگوریتم عملکرد خیلی مشابهی دارند و حتی برای بررسی این مورد، corr2() بین دو خروجی اندازه گیری شد که برابر با 0.9755 است، که شباهت خیلی زیاد لبه یابی بین این دو الگوریتم را نشان می‌دهد.

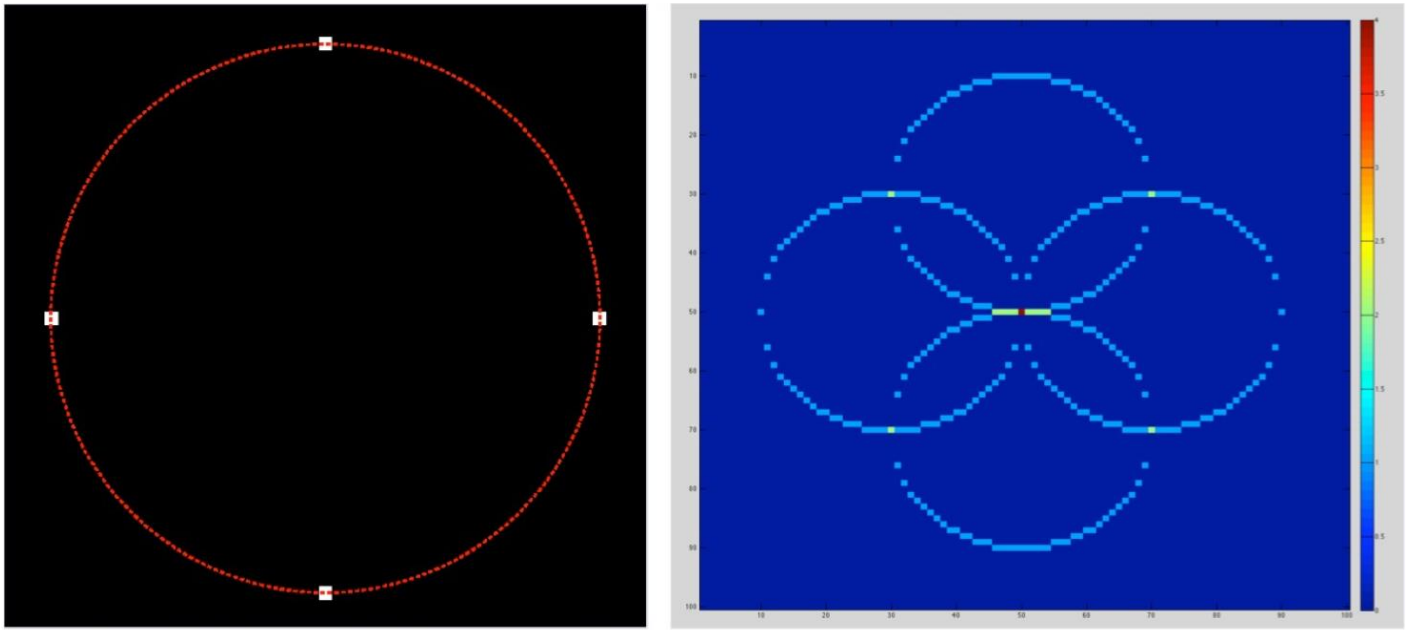
مدت زمان ران شدن الگوریتم Sobel حدودا 0.0529 و الگوریتم Kirsch حدودا 0.3969 می‌باشد. (نسبت این دو عدد 7.5 می‌باشد که نشانگر تفاوت تعداد conv2() گرفتن در هر الگوریتم است.)

2 الگوریتمی برای شمارش دایره در تصویر

الف)

توجه: در کد این بخش از تابع max به صورت max(A,[],'all') استفاده شده که گویا تنها از متلب 2018b در دسترس است!

فرض کنیم که یک دایره در صفحه داریم و می‌خواهیم مرکز آن را پیدا کنیم. همچنین این فرض را می‌کنیم که شعاع دایره را داریم و برابر با r است. یک نقطه روی محیط دایره انتخاب می‌کنیم و یک دایره به مرکزیت آن نقطه و شعاع r رسم می‌کنیم. اگر برای 3 نقطه متمایز دیگر همین کار را تکرار کنیم، محل برخورد چهار دایره، برابر با مرکز دایره اصلی می‌باشد. یک مثال را در شکل زیر می‌بینیم:

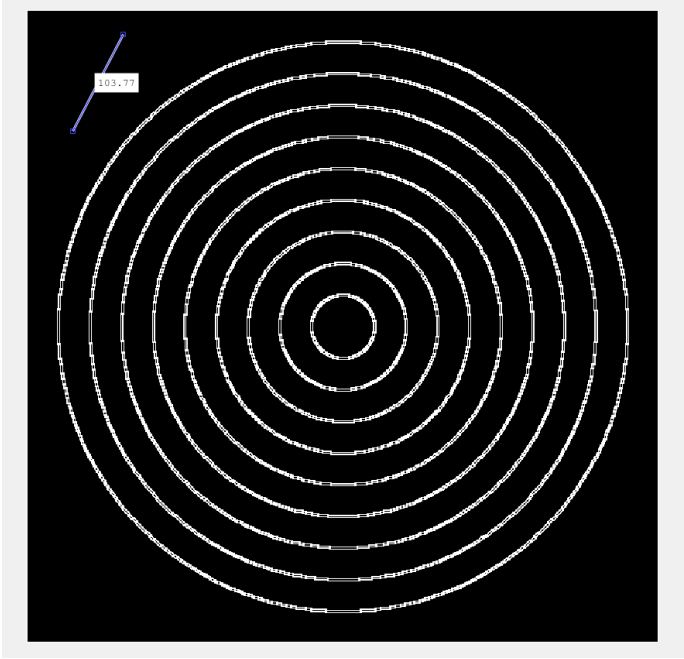


شکل سمت چپ، نقاط انتخاب شده را روی دایره اصلی نشان می‌دهد و شکل سمت راست، دایره های رسم شده به مرکزیت این نقاط و محل برخوردشان که با رنگ قرمز مشخص شده است.

پایه ریاضی الگوریتم پیاده سازی شده در این بخش نیز همین موضوع می‌باشد. در ابتدا با استفاده از فیلتر پیدا کننده لبه در بخش قبل (Sobel)، لبه های شکل را جدا می‌کنیم. سپس برای اینکه نقاط مزاحم به حداقل برسند، مقدار همه نقاط غیر لبه را در ماتریس تصویر، صفر می‌کنیم. حالا تصویر ما

برای پیاده سازی الگوریتم بالا آماده می‌باشد. در نهایت کاری که ما روی این تصویر لبه دار شده، انجام می‌دهیم، این است که روی همه محیط های دایره حرکت می‌کنیم و مطابق شعاع تقریبی دایره ها، یک دایره به مرکزیت هر نقطه می‌کشیم. وقتی همه این دایره ها را رسم کردیم، محل مرکز های دایره های اصلی، نقاطی خواهند بود که روشنی زیادی دارند. این مورد آخر را می‌توان در شکل صفحه بعد مشاهده کرد:

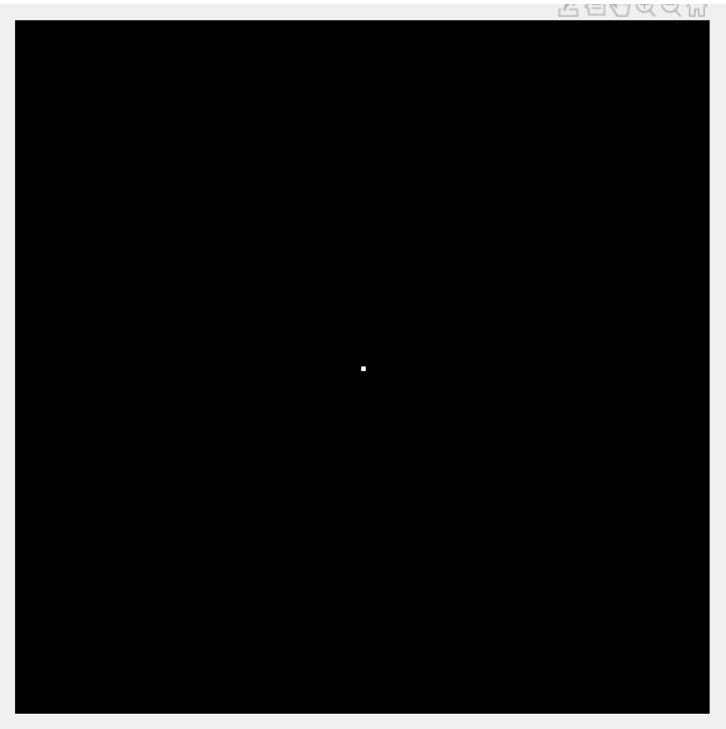
تصویر لبه دار شده:



تصویر ساخته شده از برخورد دایره ها:

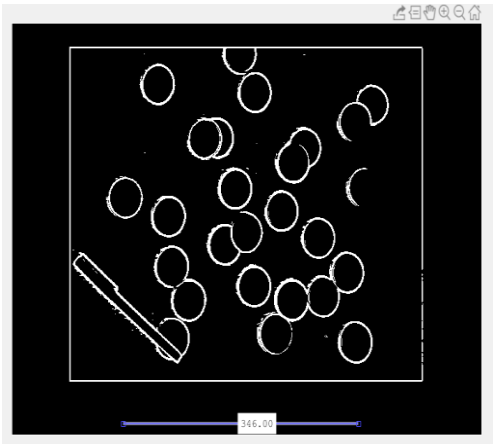
از آنجا که دایره ها هم مرکز هستند، فقط یک نقطه نورانی داریم.

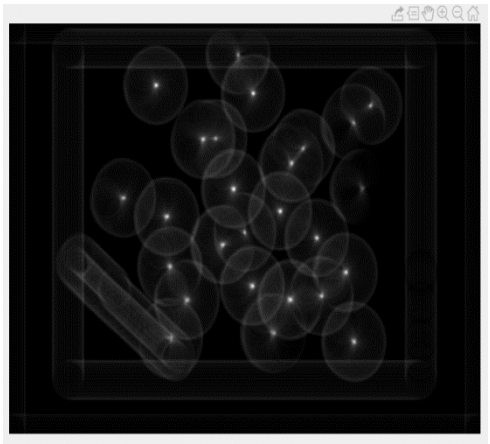
تصویر نهایی مراکز دایره های اصلی:

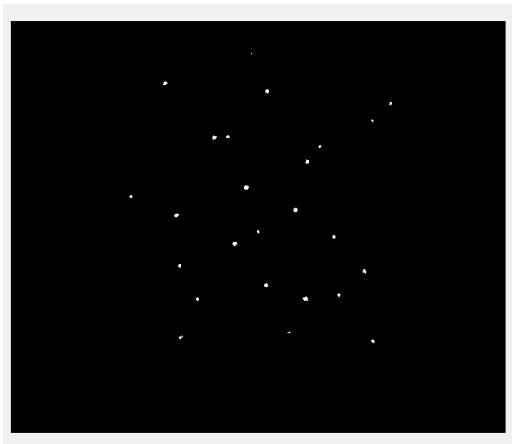


در این تصویر، مرکز دایره ها با رنگ سفید مشخص است.

همچنین به طور خلاصه، همه موارد بالا را برای تصویر circles.jpg نیز در زیر آورده ایم:







چند نکته مهم راجب کار کردن این کد وجود دارد:

نکته اول: باید حدود شعاع دایره های تصویر را بدانیم، که برای اینکار از تابع imdistline استفاده می‌کنیم.

نکته دوم: باید برای هر تصویر، مطابق یک تحلیل ساده که به صورت ذهنی و کمی آزمون خطا انجام می‌شود، دو مقدار threshold را طوری ست کنیم که خروجی بهترین حالت ممکن باشد. این دو مقدار با کامنت سبز به صورت زیر در کد مشخص شده اند:

```
if (temp_max > 10) % Another important threshold to set ---------------<<<<
    finded_centers{cnt} = find_center(G_temp, radii, 0.6); %-------- Here you can set threshold-------<<<<<
```

نکته سوم: برای زدن این کد، توابع زیر نوشته شده اند و در قسمت function کد موجود هستند.

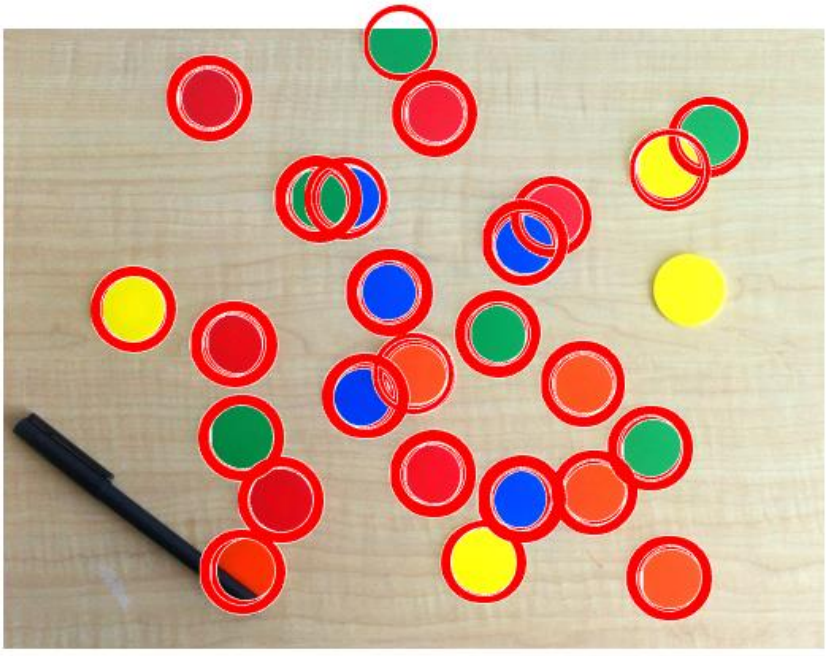
finded_centers()

circle_mat()

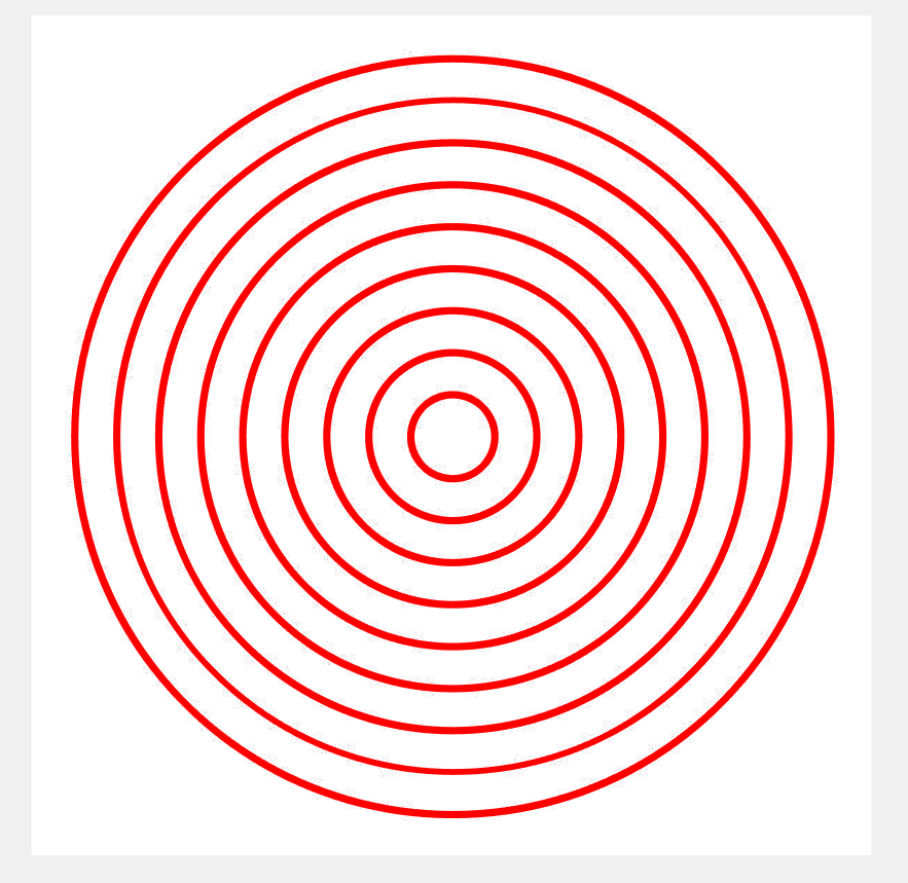
البته توجه کنید منظور finded، founded بوده است، اما به دلیلی که در ابتدای کار این اشتباه لغوی رخ داد و در انتها متوجه شدیم، تغییر آن بسیار سخت بود.

خروجی کد برای تصویر circles.jpg:

(مدت زمان ران شدن کد با threshold های در نظر گرفته شده، حدود 37 ثانیه است)



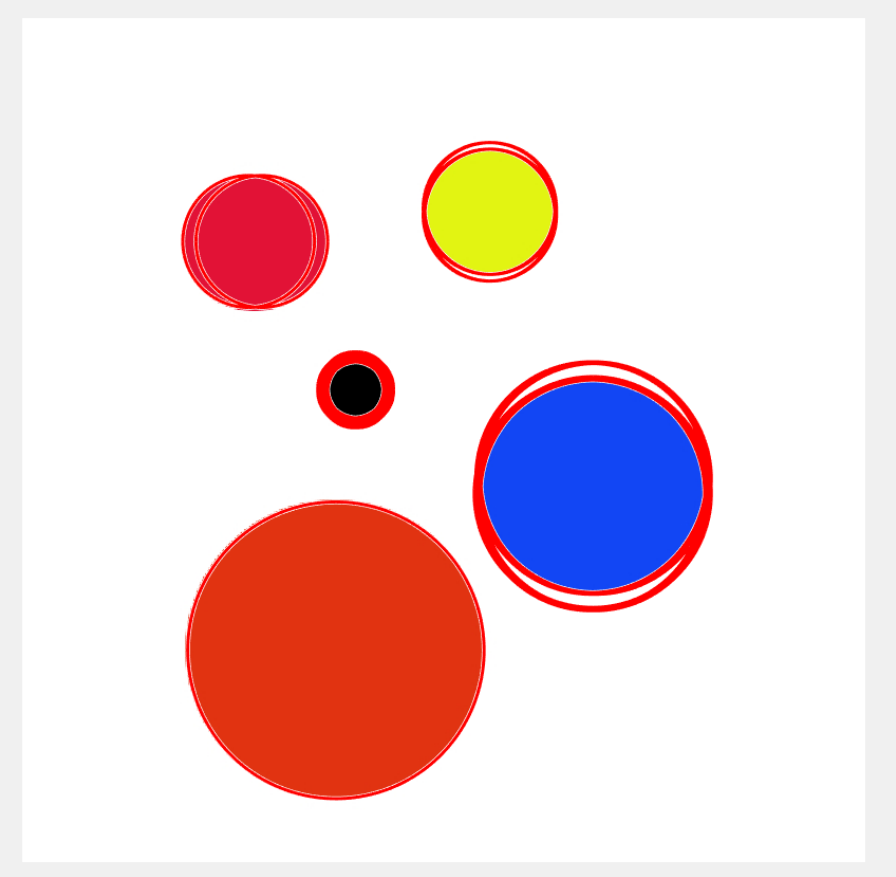
خروجی برای تصویر circles2.jpg:



دایره های قرمز رنگ با استفاده از تابع viscircles() تولید شده اند.

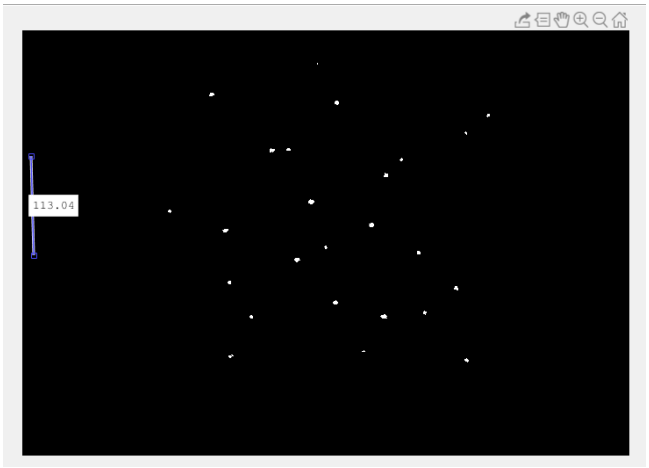
(زمان ران این قسمت با توجه به threshold های کمی سختگیرانه 201.6 ثانیه بود.)

خروجی برای تصویر circles3.jpg:



(مدت زمان ران شدن، 31 ثانیه)

در ادامه برای اینکه برنامه بتواند تعداد دایره ها را به ما بدهد، یک الگوریتم شبه کلاسترینگ (فقط شمارش می‌کنیم) بر اساس مراکز پیدا شده، می‌نویسیم. تابع این قسمت با نام clustering در قسمت function ها قرار دارد. همانطور که گفتیم، مطابق شکل مراکز خوشه بندی را انجام می‌دهیم. یعنی شکل های زیر: (این شکل ها به ترتیب از راست به چپ برای circle.jpg و circles3.jpg رسم شده اند.)



خروجی تعداد دایره ها برای تصویر circles3.jpg:

```
number of circles =
     5
```

خروجی تعداد دایره ها برای تصویر circles.jpg:

```
number of circles =
    25
```

به علت کنتراست کم قسمت راست دایره سمت راستی با پشت صحنه، این الگوریتم در حالت کلی نمی‌تواند این دایره را تشخصی دهد، برای همین تعداد را 25 نشان می‌دهد.

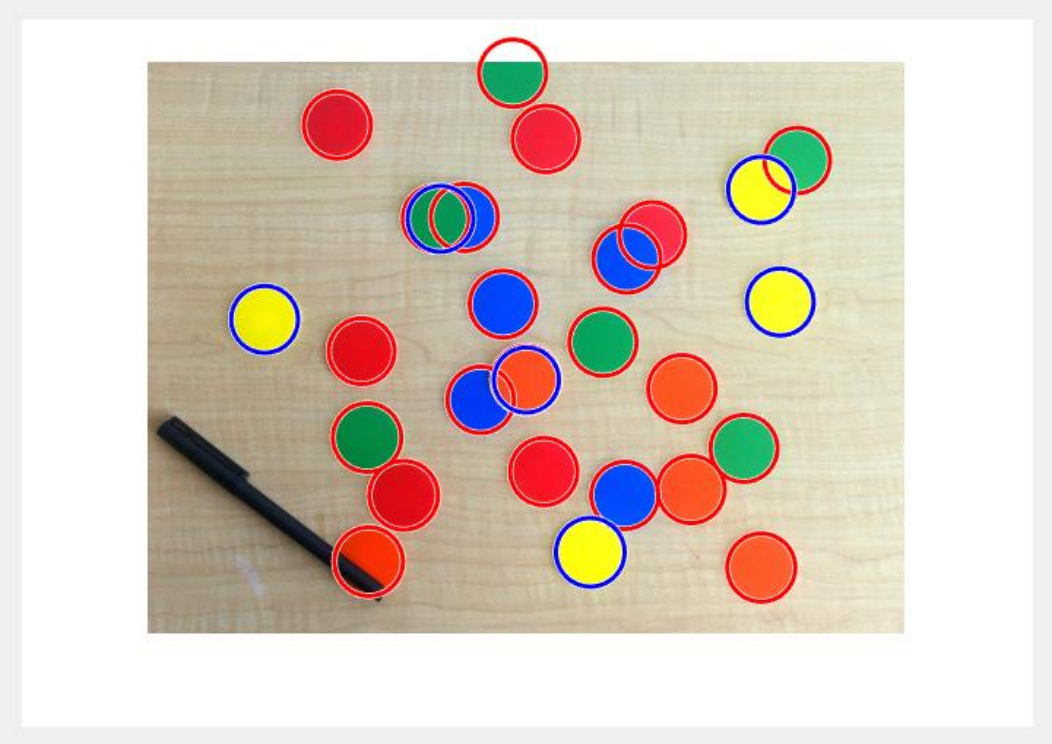
ب)

در این قسمت از تابع imfindcircles() متلب استفاده می‌کنیم. این تابع یک المان ورودی به صورت dark و bright دارد که اگر dark بزنیم، دایره های با رنگ dark تر را تشخیص می‌دهد و برای bright هم به همین ترتیب.

برای استفاده در این بخش، هر دو نوع دایره bright و dark را استخراج کردیم.

دایره های bright با رنگ آبی و دایره های dark با رنگ قرمز مشخص شده اند.

خروجی برای circles.jpg:



4 Image Segmentation

4.1 K-Means

- **به صورت شهودی، بیان کنید که چرا علاقه به کمینه کردن عبارت مشخص شده داریم.**

فرض کنیم k خوشه داریم. ما می‌خواهیم نقاط مختلف صفحه ها را طوری به خوشه ها نسبت دهیم که هر کدام کمترین فاصله را تا خوشه مربوطه داشته باشند. در یک فضای n بعدی، می‌توان فاصله اقلیدسی 2 نقطه را به صورت نورم تفریق دو نقطه نشان داد. حالا ما برای هر خوشه، یک نقطه مرکزی تعریف که می‌کنیم که این نقطه برابر با میانگین همه نقاط مربوط به آن خوشه می‌باشد. پس فاصله ما با هر خوشه، برابر با نورم تفریق هر نقطه از دیتاست ما تا مرکز خوشه است. پس رابطه نوشته کاملا توجیه می‌شود. در واقع این رابطه کل انگیزه ما را در یک عبارت ریاضی خلاصه کرده است.

- **با جست وجو، یک روش معروف برای به دست آوردن یک جواب معقول برای این مسئله را معرفی کنید. روش را به طور کامل توضیح دهید. توجه داشته باشید که نحوه ی انتخاب اولیه ی مرکز خوشه ها و همچنین شرط اتمام الگوریتم را نیز در گزارش خود ذکر کنید. همچنین، بیان کنید که چگونه می توان از K-Means برای حل مسئله ی مطلوب ما استفاده کرد.**

فرض کنیم یک تصویر با سایز $x \times y$ داریم. هر پیکسل از این تصویر را به صورت $p(x,y)$ نشان می‌دهیم.

حالا به ترتیب می‌خواهیم مراحل زیر را طی کنیم:

مرحله 1) ابتدا مقدار کلاستر ها (k) را تعیین می‌کنیم و برای هر کلاستر یک مرکز در نظر می‌گیریم که آن را با c_k نمایش می‌دهیم. در این پروژه مرکز ها را به صورت رندوم مشخص می‌کنیم. یعنی از آنجا که k=2 است، دو نقطه را به صورت رندوم به عنوان مراکز اولیه انتخاب می‌کنیم.

مرحله 2) برای هر پیکسل، فاصله تا مرکز را با استفاده از رابطه زیر، محاسبه می‌کنیم:

$$d = ||p(x,y) - c_k||$$

مرحله 3) بر اساس فاصله های محاسبه شده، هر پیکسل را به یک کلاستر نسبت می‌دهیم. طبیعتا کلاستری را انتخاب می‌کنیم که کمترین فاصله را با آن داریم.

مرحله 4) بعد از نسبت دادن همه پیکسل ها، باید محل جدید مراکز کلاستر ها را محاسبه کنیم. این مراکز برابر با میانگین نقاط نسبت داده شده به هر خوشه می‌باشد. رابطه مراکز جدید به صورت زیر است:

$$c_k = \frac{1}{k} \sum_{y \in c_k} \sum_{x \in c_k} p(x,y)$$

مرحله 5) آنقدر مرحله 1 تا 4 را تکرار می‌کنیم که به شرط اتمام الگوریتم برسیم.

شرط اتمام را به این صورت می‌گذاریم که اگر مرکز خوشه ها دیگر تغییر نکرد یا به یک میزان iteration خاصی رسیدیم، برنامه متوقف شود.

در اینجا به طور خاص به iteration بسنده می‌کنیم.

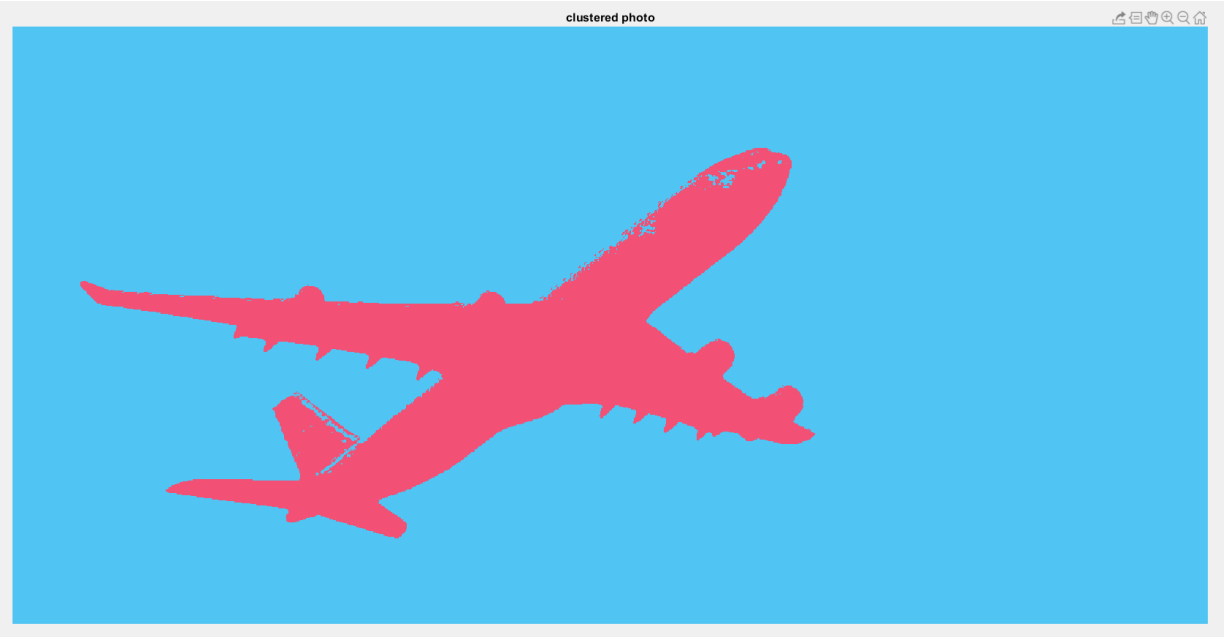
تابع نوشته شده برای این قسمت، k_means() نام دارد که در قسمت Functions موجود است.

همچنین در ابتدا کمی modification روی تصویر اصلی انجام می‌دهیم تا به علت شباهت های رنگی دچار مشکل نشویم. از آنجا که کلا قرار است تصویر را به دو بخش تقسیم کنیم، این کار بسیار ساده است. برای مثال در تصویر img2.jpeg غالب تصویر آبی است و کل قسمت آبی باید یک کلاستر شود. حالا به علت وجود رنگ های تیره در هواپیما ممکن است، این بخش ها جزوی از آسمان تشخیص داده شوند. به همین دلیل، ما یک شرط می‌گذاریم که اگر در تصویر اصلی رنگ های تیره وجود داشت، قسمت آبی آن را حذف کند.

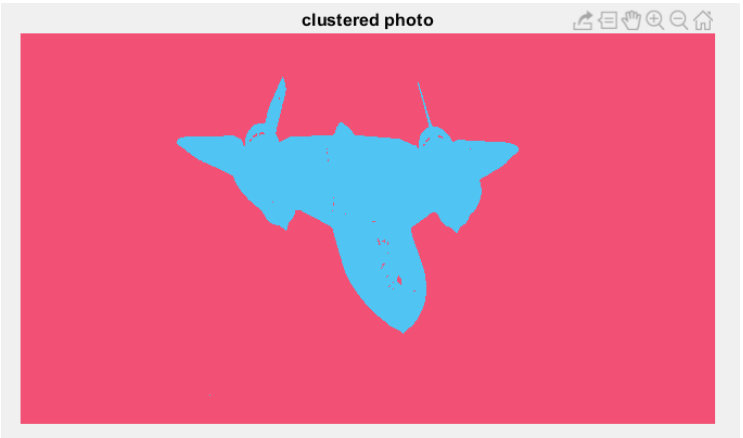
یک approach کلی تر برای این نوع modification می‌تواند این باشد که رنگ غالب تصویر را پیدا کنیم (مثلا با میانگین گیری) و سپس آن را از بخش های خاص عکس که رنگ متفاوتی دارند، حذف کنیم. برای مناظر طبیعی مثلا آسمان و دشت، این رنگ غالب معمولا فقط مربوط به یک کلاستر می‌شوند و همه پیکسل هایی که این رنگ ها را دارند، اصولا در یک کلاستر قرار می‌گیرند.

خروجی برای تصویر img2.jpeg:

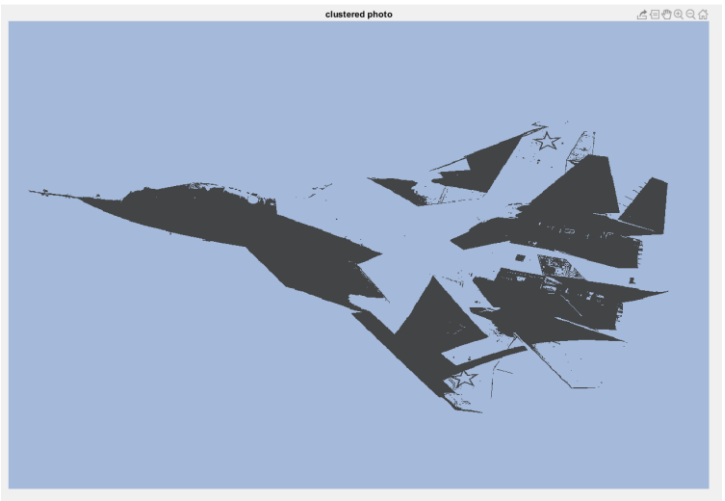
همانطور که مشاهده می‌کنید، در تصویر سمت راست، طیف آبی از رنگ های تیره داخل هواپیما حذف شده است.



خروجی برای تصویر Airplane6.jpg:



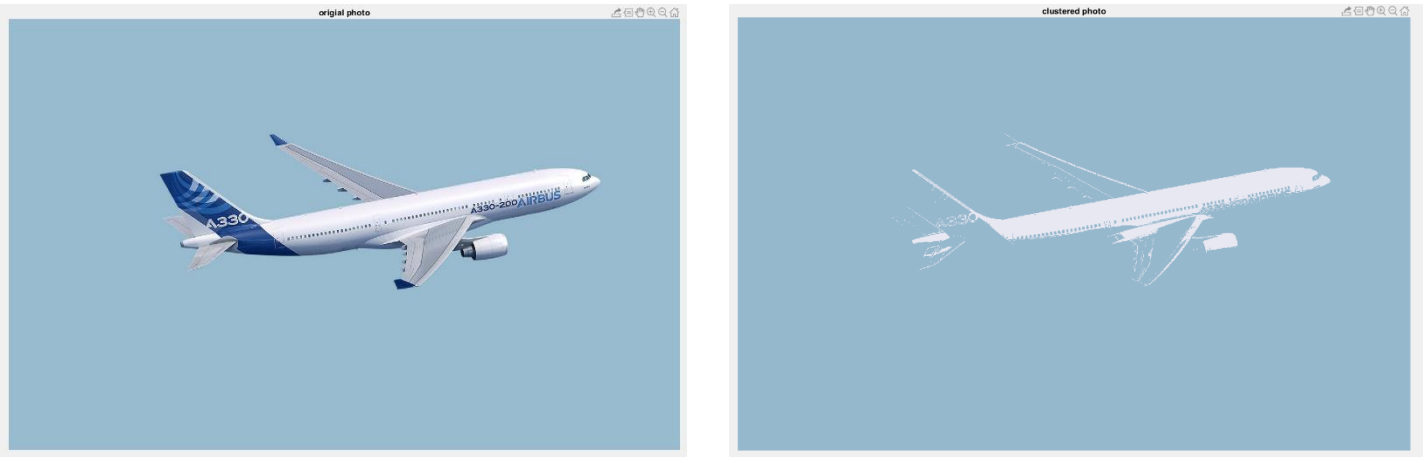
خروجی برای تصویر Airplane4.jpg:



خروجی برای تصویر Airplane3.jpg:

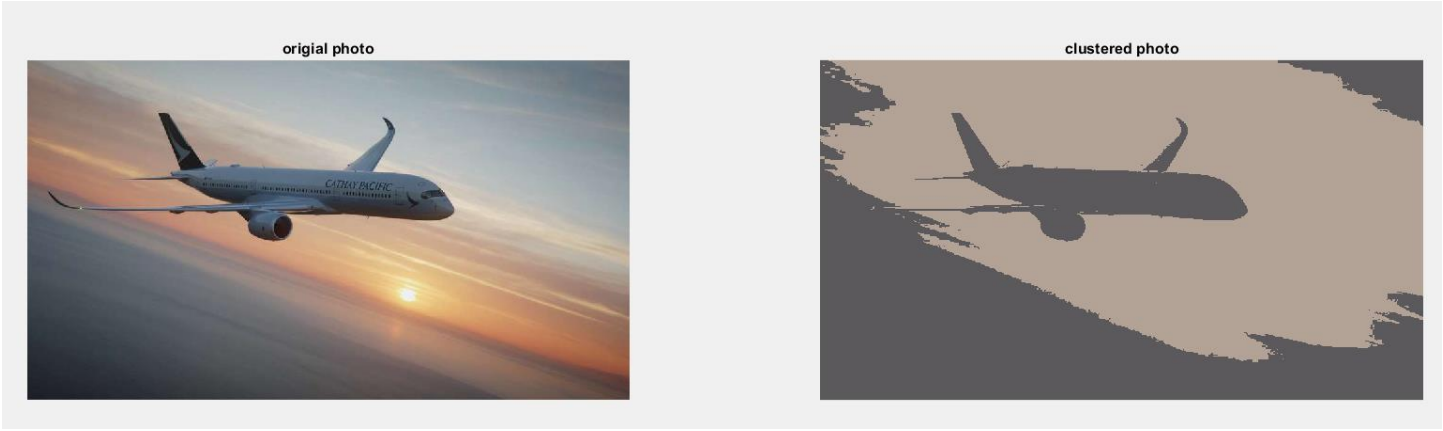
در تمام خروجی های بالا iteration برابر با 5 قرار داده شد و مراکز اولیه رندوم بود.

خروجی برای تصویر Airplane2.jpg:



در این تصویر، اگر مرکز های اولیه به صورت رندوم انتخاب می‌شد، تقریبا در هر بار ران کردن، فقط آسمان آبی باقی می‌ماند. برای رفع این مشکل، یکی از نقاط مرکزی اولیه را روی هواپیما گرفتیم.

خروجی برای تصویر img1.jpg:



به علت همرنگ بودن تقریبی، قسمت هایی از آسمان جزو هواپیما حساب شده اند.

4.2 Otsu Algorithm

فرض کنیم یک threshold برای intensity پیکس های هر تصویر در نظر بگیریم و پیکسل ها بالای این آستانه را تا ماکسیمم intensity بالا ببریم و پیکسل های کمتر از این آستانه را خاموش کنیم. پایه عملکرد الگوریتم Otsu نیز همین موضوع می‌باشد.

در روش Otsu برای دو class، تلاش می‌کنیم که این مقدار threshold را طوری تعیین کنیم تا مقدار واریانس بین دو class ماکسیمم شود. این واریانس را به صورت زیر محاسبه می‌کنیم.

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

در این رابطه w ها برابر با احتمال وقوع کلاس ها هستند که از تعداد پیکسل های تصویر و intensity آنها، به صورت زیر قابل محاسبه است:

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

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

همچنین مقادیر u میانگین کلاس ها هستند.

برای پیاده سازی کد ابتدا intensity را از تصویر جدا می‌کنیم. سپس روی threshold های از 1 تا maximum intensity حرکت می‌کنیم و در هر مرحله، u و w و واریانس بین دو کلاس را محاسبه می‌کنیم. در نهایت با توجه به واریانس ماکسیمم، threshold را انتخاب می‌کنیم و تصویر نهایی را می‌سازیم.

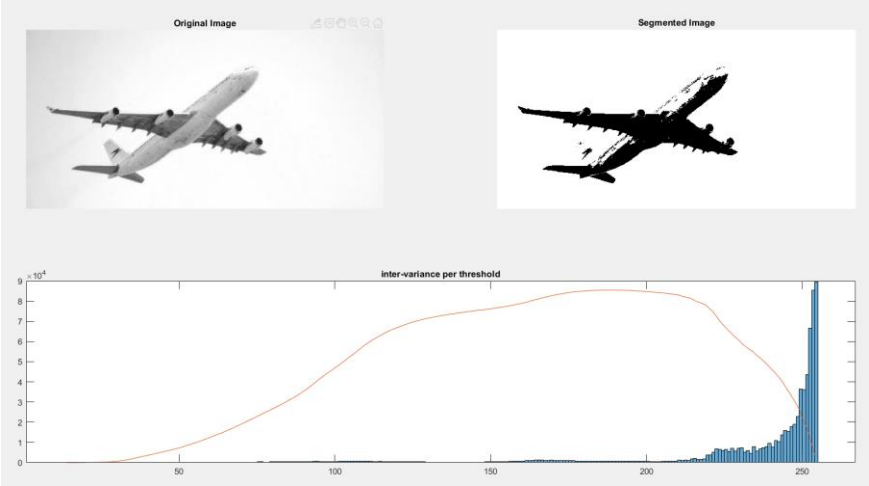
تابع این قسمت با نام Otsu() در بخش توابع کد موجود است.

در کنار تصاویر خروجی، نمودار هیستوگرام intensity پیکسل ها و نمودار واریانس بین دو class روی آن (با رنگ قرمز) رسم شده است.

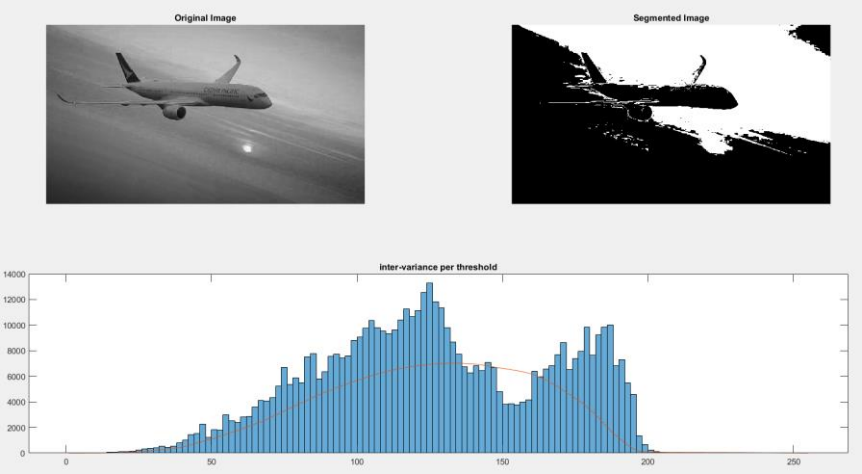
برای بررسی صحت عملکرد کد، مقادیر threshold محاسبه شده توسط این کد، با مقادیر به دست آمده از یک تابع مشابه موجود در ویکیپدیا (که تابعی صحیح و مرجع تلقی می‌شود) مقایسه شد و عملا تفاوتی بین threshold ها دیده نشد.

مدت زمان ران شدن برای همه تصاویر صفحه های بعد، حدود 0.4 ثانیه می‌باشد.

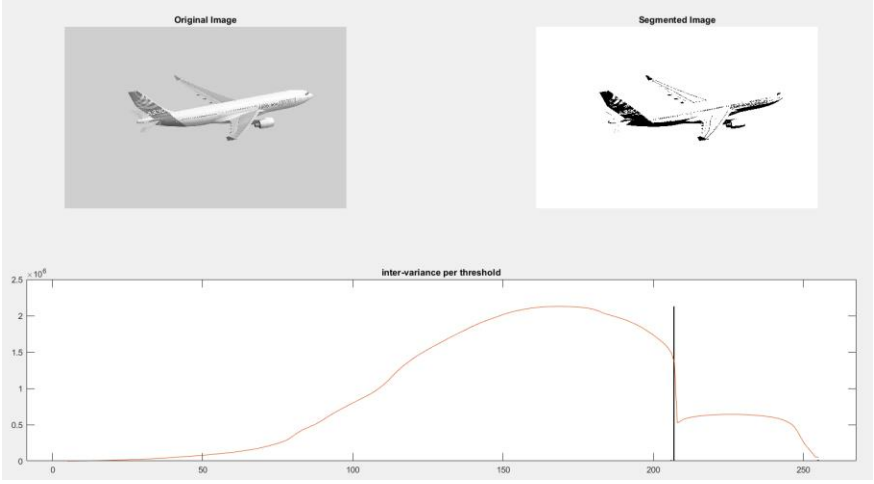
خروجی برای تصویر img2.jpeg:



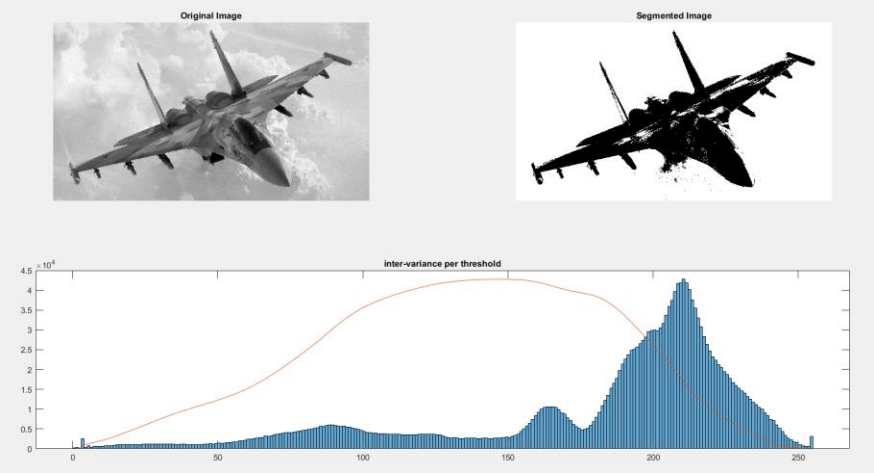
خروجی برای تصویر img1.jpg:



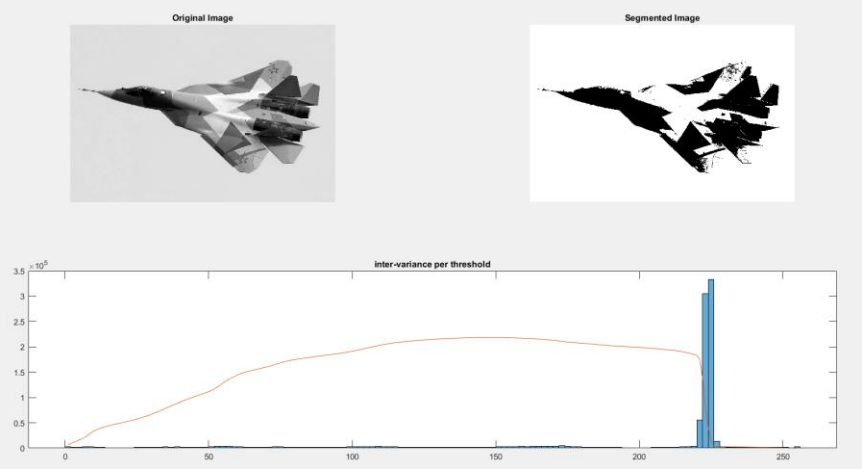
خروجی برای تصویر Airplane2.jpg:



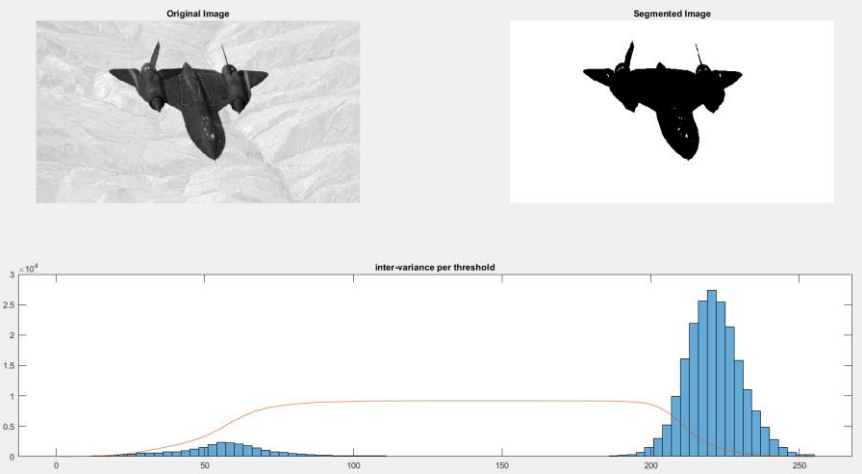
خروجی برای تصویر Airplane3.jpg:



خروجی برای تصویر Airplane4.jpg:



خروجی برای تصویر Airplane6.jpg:



4.3 Comparison & Conclusion

برای مقایسه این دو الگوریتم طراحی شده، چند معیار را انتخاب می‌کنیم و برای هر تصویر از تصاویر تست شده، مقایسه بین روش ها را انجام می‌دهیم.

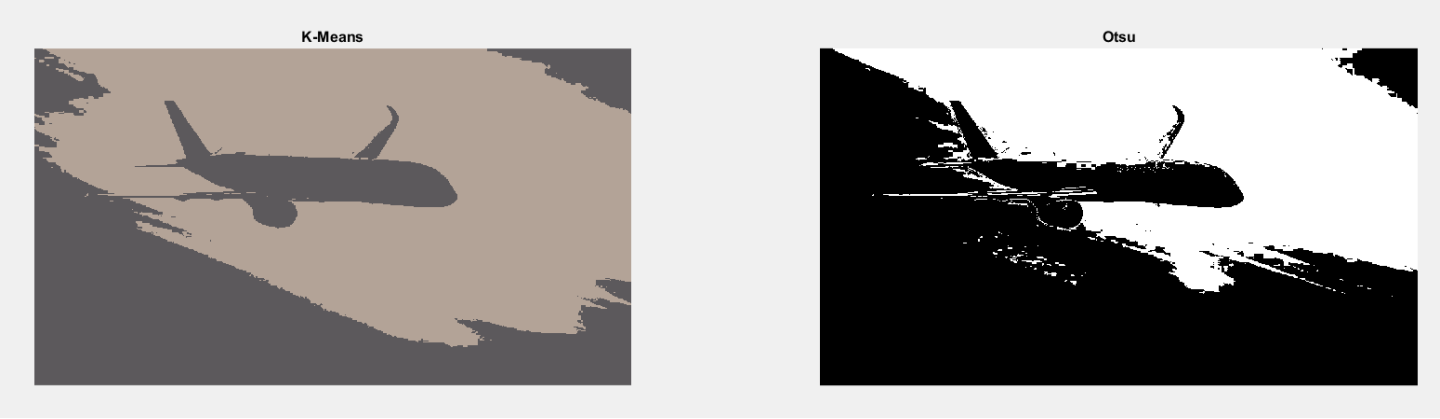
یک معیار باید **سرعت ران شدن کد** باشد. معیار دیگر **خروجی الگوریتم ها** است که چقدر با انتظاری که داشتیم شباهت دارد. همچنین معیار دیگر **سادگی پیاده سازی** است که در این معیار هر دو الگوریتم در حالت پایه به یک میزان ساده هستند.

توجه کنید که هر کدام از این الگوریتم ها به شکل های پیچیده تر و حتی سریع تر و چه بسا بهتر، می‌توانند پیاده سازی شوند. مثلا برای الگوریتم قسمت بندی K-Means راه حل های خیلی زیادی وجود دارد و ما تنها از یکی از راه های معروف و به نسبت ساده استفاده کرده ایم.

اکنون به مقایسه جزیی دو الگوریتم برای هر تصویر می‌پردازیم.

در تمام تست ها، iteration برای K-Means برابر با 5 قرار داده شده.

برای تصویر img1.jpg:



مدت زمان ران شدن به ترتیب، 0.078 و 5.58 ثانیه.

همانطور که می‌بینید در این تصویر، الگوریتم K-Means بهتر عمل کرده و لااقل قسمت کمتری از پشت هواپیما بخشی از هواپیما حساب شده است. ضعف روش Otsu در اینجا می‌تواند تداخل زیاد طیف intensity بک گراند و خود هواپیما باشد. اگر به نمودار variance per T که در قسمت قبل (وسط صفحه 14) برای این تصویر رسم شده نگاه کنید، این تداخل را به وضوح می‌توانید ببینید.

برای تصویر img2.jpeg:

مدت زمان ران شدن به ترتیب 0.084 و 7.78 ثانیه.

در این تصویر نیز K-Means بهتر عمل کرده است. در اینجا باز هم مرز intensity برای Otsu کمی محو می‌باشد. اما برای K-Means تقریبا همه شرایط آماده است چون بک گراند یک رنگ مشخص دارد و هیچ تصویر دیگر مزاحمت ایجاد نمی‌کند.

برای تصویر Airplane2.jpg:

مدت زمان ران شدن به ترتیب، 0.131 و 37.35 ثانیه.

اگر یادتان باشد در صفحه 12 توضیح دادیم که برای این تصویر به علت شباهت رنگی زیاد اجزا، باید یک مرکز اولیه روش K-Means را خودمان روی هواپیما انتخاب کنیم وگرنه با روش رندوم تقریبا احتمالش کم است که نقطه اولیه روی هواپیما بیوفتد و در نتیجه تنها یک سگمنت مربوط به آسمان خواهیم داشت. اگر نقطه اولیه را دستی انتخاب کنیم، عملکرد این دو روش تقریبا یکسان است و فقط کمی در روش Otsu جزییات بیشتری وجود دارد. البته مدت زمان ران شدن کد، به مراتب متفاوت است و K-Means خیلی بیشتر زمان برده است.

برای تصویر Airplane3.jpg:

مدت زمان ران شدن به ترتیب، 0.129 و 21.59 می‌باشد.

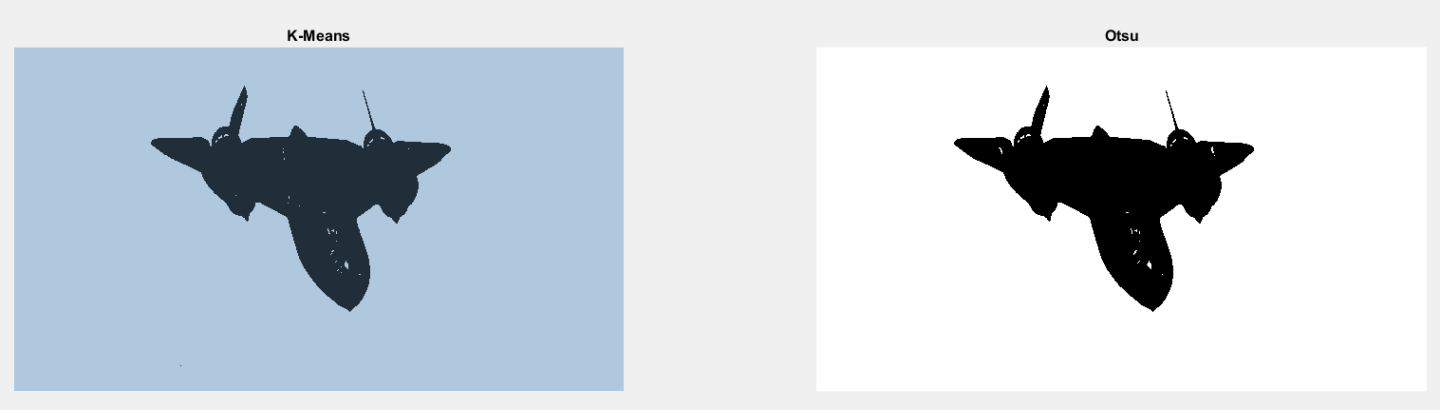
تقریبا می‌توان گفت که عملکرد Otsu بهتر است چون مقدار کمتری از صحنه پشت هواپیما را جزو هواپیما حساب کرده. البته تنها این مورد نباید گولمان بزند زیرا که بخش هایی از درون هواپیما در روش Otsu جزو آسمان در نظر گرفته شده اند و در روش K-Means این اتفاق کمتر افتاده است.

برای تصویر Airplane4.jpg:

زمان ران شدن به ترتیب 0.089 و 10.387 می‌باشد.

می‌توان تا حدی در این تصویر دید که Otsu مقدار بیشتری از هواپیما را گرفته. البته به مقدار جزیی این برتری وجود دارد.

برای تصویر Airplane6.jpg:



زمان ران شدن به ترتیب، 0.072 و 2.47 ثانیه.

همانطور که می‌بینید در این تصویر هر دو الگوریتم عملکرد خیلی خوبی داشته اند و در واقع Otsu کمی بهتر عمل کرده است.

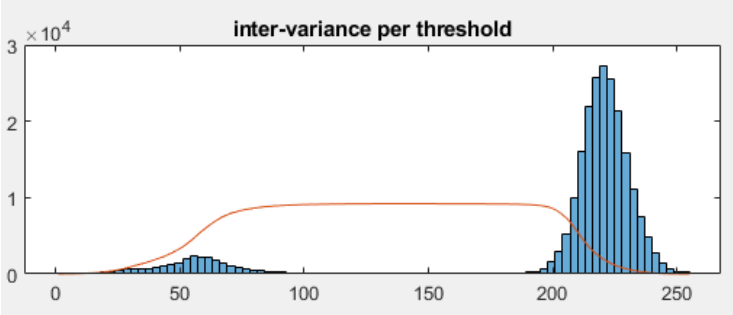
در نهایت کمی به مقایسه کلی این دو روش می‌پردازیم و مزایا و معایب هر یک را بررسی می‌کنیم.

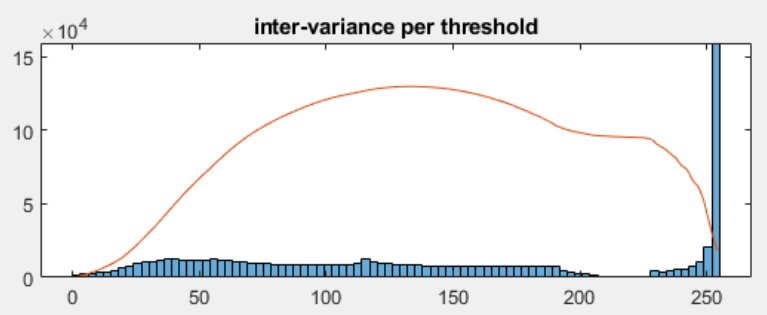
در کل دیدیم که در تصاویر، سرعت الگوریتم Otsu بسیار بالا تر از الگوریتم K-Means می‌باشد. البته برای K-Means روش های پیاده سازی بهتر و سریع تری نیز وجود دارد اما به هر حال الگوریتم Otsu را می‌توان در کل سریع تر به شمار آورد. در رابطه با خروجی نیز اگر طیف intensity تصاویر تداخل زیادی نداشته باشد، روش Otsu در کل بهتر عمل می‌کند. (با توجه به اینکه سرعت آن نیز خیلی بیشتر است)

مزایا و معایب روش Otsu:

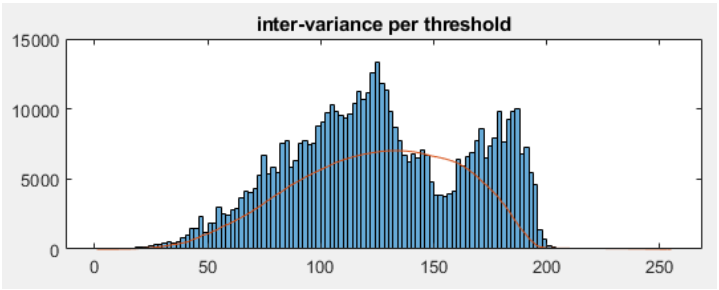
از مزایای این روش سادگی و سرعت زیاد آن است. همچنین این روش با یک پشتوانه ریاضی بالا آمده است و مطابق قواعد آمار و احتمال پیاده سازی می‌شود.

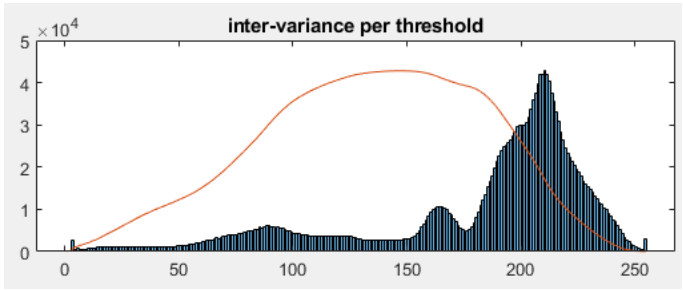
یکی از ضعف های این روش در تصاویری است که تداخل طیف intensity در آنها زیاد است. منظورمان را با چند مثال نشان می‌دهیم.





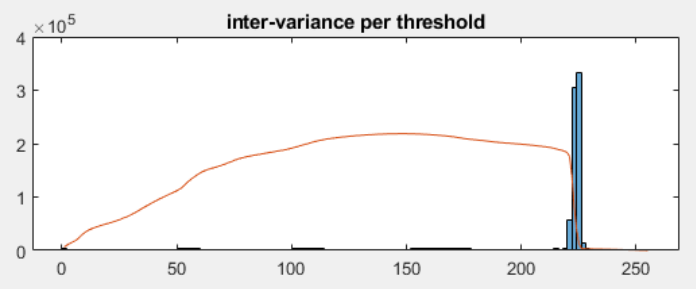
این دو نمودار نمونه های خیلی خوب پخش شدن intensity در تصویر می‌باشند. نمودار سمت راست مربوط به تصویر OtsuCheck.jpg و نمودار سمت چپ مربوط به تصویر Airplane6.jpg می‌باشد. البته در تصویر OtsuCheck ما در واقع دو جسم نداریم و تصویر بزرگتر و گستره تری است. اما اگر با همین نمودار یک تصویر دو جسمی (مانند هواپیما و آسمان) داشتیم، به احتمال خیلی خوبی، به راحتی از هم جدا می‌شوند.

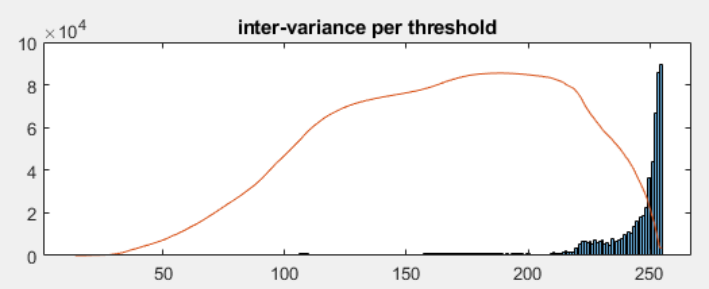




نمودار بالا سمت راست، مربوط به تصویر Airplane6.jpg می‌باشد که یک نمونه ضعیف تر از نمونه های قبلی که تداخل نداشتند حساب می‌شود. در واقع اینجا تداخل وجود دارد اما خیلی کمتر است و در نتیجه خروجی ما به نسبت مناسب است.

نمودار سمت چپ نیز برای تصویر img2.jpeg است که این تداخل بسیار زیاد شده و در نهایت نیز هواپیما به خوبی از پس زمینه جدا نشده است.





یک نمونه دیگر از نمودار ها، نمودار های مانند سهمی هستند که در این نمودار ها هم احتمالا خروجی آنچنان با کیفیتی نخواهیم داشت چون اصولا بین دو جسم، تداخل intensity وجود دارد و کم پیش می‌آید که این طیف تنها مربوط به یک جسم باشد.

نمودار بالا راست مربوط به img2.jpeg و نمودار چپ مربوط به Airplane4.jpg می‌باشد.

مزایا و معایب K-Means:

از مزایای این روش مفهوم قابل درک آن است و در واقع هدف ما کاملا مشخص است و تنها باید برای رسیدن به این هدف یک مسیری را طی کنیم. (مثلا روش های تعیین فاصله مختلفی برای clustering هستند. ما از روش فاصله اقلیدسی استفاده کرده ایم)

از مزایای دیگر آن توانایی جداسازی اجسام مطابق رنگ می‌باشد. طبیعتا در بسیاری از موارد می‌تواند کاربرد زیادی داشته باشد.

یک مورد که بستگی به نوع کار ما دارد و می‌تواند مزیت و یا عیب این روش باشد، از قبل معلوم بودن تعداد خوشه ها می‌باشد. در حالت کلی ما نمی‌دانیم که در تصویر چه خبر است، بنابراین نمی‌دانیم باید انتظار چند نوع جسم را داشته باشیم. پس در صورت استفاده از K-Means باید این آگاهی را از قبل داشته باشیم.

از معایب دیگر آن زمان گیر بودن آن است. به طوری که ما با iteration=5 به مراتب سرعت کند تری نسبت به Otsu داشته ایم. البته قبلا هم گفتیم که روش های پیاده سازی سریع تر هم وجود دارند.

3 حذف نویز

عکس اصلی که در این بخش تحت نویز ها و فیلتر های مختلف قرار میگیرد به صورت زیر است:

الف)

Salt & pepper :

به عنوان نویز ضربه نیز شناخته میشود و در تصویر به صورت یک پیسکل سفید یا سیاه دیده میشود.

و از علل آن میتوان به تداخل سیگنال های با دامنه زیاد اشاره کرد.

از انجایی که این نویز تنها در پیکسل ها ی خاص ایجاد میشود به کمک فیلتر های میانه گیر و میانگین به راحتی قابل برطرف شدن است.

https://dsp.stackexchange.com/questions/28920/how-does-salt-and-pepper-noise-occurs-in-an-image

ب)

Gaussian:

این نویز معمولا هنگام دریافت تصاویر از گیرنده ها نظیر سنسور ها در شرایط خاص (برای سنسور روشنایی :نور کم-حرارتی:دمای بالا) ایجاد می‌شود و با داده گیری های زیاد مشاهده می‌شود که نمودار این نویز ها به صورت توزیع نرمال در می آید و به کمک فیلتر هایی مانند فیلتر Gaussian نیز تا حد زیادی بر طرف میشود اما امکان دارد کمی از کیفیت تصویر اصلی پایین تر باشد.

https://dsp.stackexchange.com/questions/29475/why-is-gaussian-noise-called-so/29476

https://en.wikipedia.org/wiki/Gaussian_noise#:~:text=Principal%20sources%20of%20Gaussian%20noise,transmission%20e.g.%20electronic%20circuit%20noise

د)

Speckle :

در تصویر تقریبا به صورت نقطه نقطه دیده میشود . و معمولا ناشی از تداخل سیگنال برگشتی در دیافراگم مبدل است.

در کار های راداری و تصویر برداری سونو گرافی رایج است.

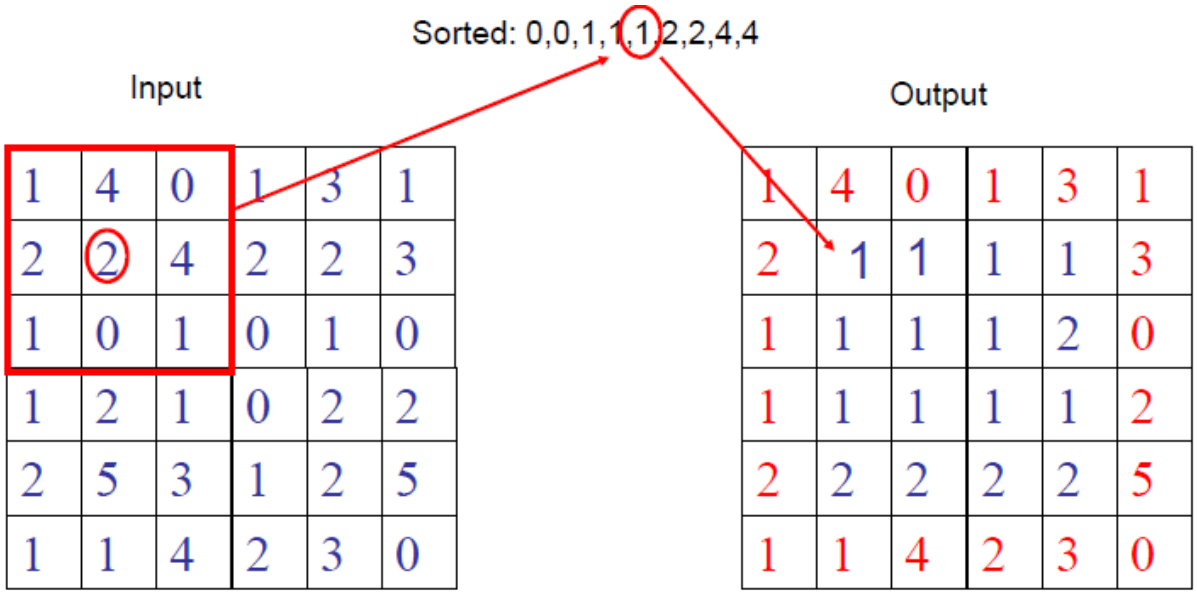
برای از بین بردن آن میتوان از یک فیلتر adaptive (وزن متفاوت در بخش های مختلف که به سطح نویز بستگی دارد) وnon adaptive که دارای وزن یکسان در تمام بخش ها است استفاده کرد.

https://en.wikipedia.org/wiki/Speckle_(interference)

ب)

Median Filter:

در این نوع فیلتر یک پنجره روی ماتریس جا بجا میشود و درایه مرکزی برابر میانه ی تمام داده های درون آن پنجره میشود.



1	4	0	1	3	1
2	2	4	2	2	3
1	0	1	0	1	0
1	2	1	0	2	2
2	5	3	1	2	5
1	1	4	2	3	0

1	4	0	1	3	1
2	1	1	1	1	3
1	1	1	1	2	0
1	1	1	1	1	2
2	2	2	2	2	5
1	1	4	2	3	0

این فیلتر یک فیلتر غیر خطی است و خروجی آن بر اساسی ضابطه ی دقیقی از ورودی ها تعیین نمیشود و به همین علت نمیتوان از کانولوشن یک کرنل در این ماتریس به نتیجه رسید و باید این پنجره را به کمک دو حلقه ی تو در تو روی ماتریس حرکت داد و عضو ها را بررسی کرد.

Gaussian Filter:

این فیلتر از ضرب یک کرنل که درایه های آن توزیع نرمال دوبعدی هستند درست شده است و به کمک کانولوشن کردن در ماتریس نویز دار، نویز آن را کاهش میدهد.

برای مثال کرنل یک فیلتر گوسی با انحراف معیار 5.5 به این صورت محاسبه میشود.

$$\begin{pmatrix} \frac{1}{2\pi(5.5)^2}e^{-\frac{1^2+1^2}{2(5.5)^2}} & \frac{1}{2\pi(5.5)^2}e^{-\frac{0^2+1}{2(5.5)^2}} & \frac{1}{2\pi(5.5)^2}e^{-\frac{1^2+1^2}{2(5.5)^2}} \\ \frac{1}{2\pi(5.5)^2}e^{-\frac{1^2+0^2}{2(5.5)^2}} & \frac{1}{2\pi(5.5)^2}e^{-\frac{0^2+0^2}{2(5.5)^2}} & \frac{1}{2\pi(5.5)^2}e^{-\frac{1^2+0^2}{2(5.5)^2}} \\ \frac{1}{2\pi(5.5)^2}e^{-\frac{1^2+1^2}{2(5.5)^2}} & \frac{1}{2\pi(5.5)^2}e^{-\frac{0^2+1^2}{2(5.5)^2}} & \frac{1}{2\pi(5.5)^2}e^{-\frac{1^2+1^2}{2(5.5)^2}} \end{pmatrix}$$

ج)

-3

در بخش های 1و2 توابع خواسته شده نوشته شد و اکنون هر یک از عکس های آلوده به نویز را ازهردو فیلتر عبور میدهیم:

Salt & pepper

به کمک MedianFilter و n=3:

چون این نویز مستقل از روابط پیچیده است با sampling window کوچک هم میتوان به نتیجه مطلوب دست یافت.

Gausian Filter:

برای این تصویر از فیلتر با سایز کرنل 13 و انحراف معیار 3 استفاده شده است.

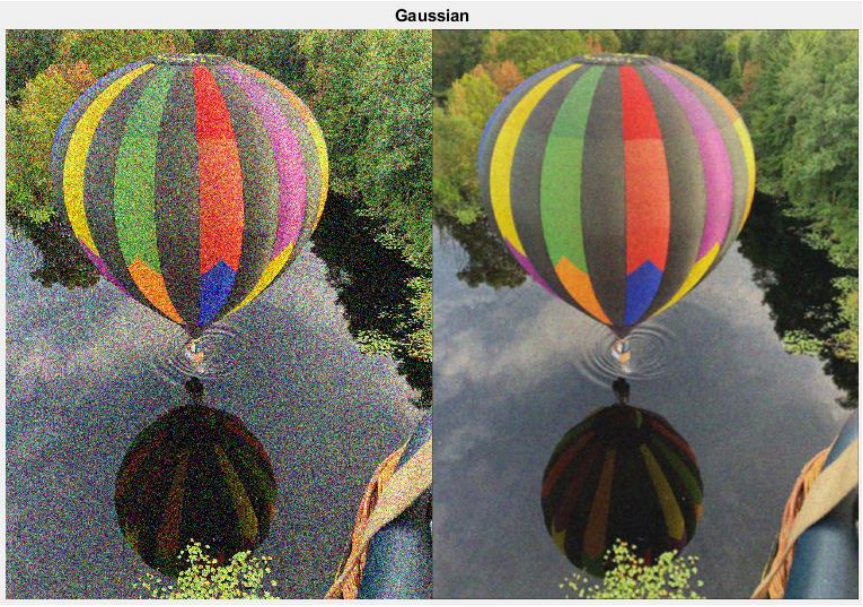
اگر اندازه کرنل کوچک تر از این مقدار باشد نویز ها به خوبی بر طرف نشده و اگر خیلی بیشتر باشد تصویر به اصطلاح دچار Bluring شده و مات میشود.

اگر انحراف معیار نیز کم لحاظ شود نیز نویز ها به خوبی بر طرف نمیشوند.

Gaussian:

Medin Filter:

Gaussian Filter:



Poisson:

Median:

Gaussian:

Speckle :

Median Filter:

Gaussian Filter: