



OpenCV

پردازش تصویر در

بِسْمِ اللّٰهِ الرَّحْمٰنِ الرَّحِيْمِ

پردازش تصویر در OpenCV

نویسنده‌گان:

علیرضا داودی

رضا غریبی

فهرست

۱	فصل اول - معرفی	۱
۷	معرفی آسی وی	۱,۱
۹	بارگذاری، نمایش و ذخیره تصویر	۱,۲
۱۱	کلاس MAT	۱,۳
۱۹	ذخیره سازی و بازیابی اطلاعات	۱,۴
۲۷	فصل دوم - حوزه مکان	۲
۲۷	شکل‌های هندسی ساده	۲,۱
۳۱	ترکیب خطی دو تصویر	۲,۲
۳۳	فیلتر خطی	۲,۳
۳۵	تبدیل آفاین	۲,۴
۳۹	نگاشت	۲,۵
۴۲	تغییر کنترast و روشنایی تصویر	۲,۶
۴۵	هموار سازی تصویر	۲,۷
۵۱	اضافه کردن قاب به تصویر	۲,۸
۵۳	مولد عدد تصادفی و ترسیم متن	۲,۹
۶۲	بزرگ‌نمایی و کوچک‌نمایی تصویر	۲,۱۰
۶۵	آستانه‌گذاری	۲,۱۱
۷۱	بافت‌نگار تصویر	۲,۱۲
۷۶	برابرسازی بافت‌نگار	۲,۱۳
۷۹	عملگر سابل	۲,۱۴
۸۲	عملگر لایپلاس	۲,۱۵
۸۵	عملگر کَنی	۲,۱۶
۸۸	کانتورهای تصویر	۲,۱۷
۹۱	پوش محدب	۲,۱۸
۹۴	قاب‌های مستطیلی و دایره‌ای برای کانتورها	۲,۱۹
۹۶	قاب‌های چرخیده مستطیلی و بیضوی برای کانتورها	۲,۲۰
۹۹	فصل سوم - حوزه زمان	۳
۹۹	تبدیل فوریه‌گسسته	۳,۱
۱۰۵	فصل چهارم - مورفولوژی	۴
۱۰۵	فرسایش و گشايش	۴,۱

۱۰۹	تبديل‌های مورفولوژی پیچیده.....	۴,۲
۱۱۵	فصل پنجم-شناسایی الگو.....	۵
۱۱۵	مقایسه بافت‌نگارها	۵,۱
۱۱۸	بک پروجکشن	۵,۲
۱۲۲	تطبیق الگو	۵,۳
۱۲۷	تبديل خط هُو	۵,۴
۱۳۲	تبديل دایره هُو	۵,۵

۱ فصل اول - معرفی

۱،۱ معرفی اُسی وی

اُسی وی^۱ یک کتابخانه از توابع برنامه‌نویسی است و هدف اصلی آن بینایی کامپیوتر بی‌درنگ است. این کتابخانه توسط شرکت اینتل (مرکز تحقیقات روسیه) توسعه داده شد و در حال حاضر توسط شرکت‌های Itseez و Willow Garage پشتیبانی می‌شود. استفاده از این کتابخانه تحت مجوز متن باز بی‌اس دی، آزاد (و مجانی) است. بیشتر تمرکز این کتابخانه روی پردازش تصاویر به صورت بی‌درنگ است. اگر روی سیستمان برنامه Integrated Performance Primitives^۲ را نصب داشته باشید، اُسی وی می‌تواند از آن جهت بهبود عملکرد خودش استفاده کند و سرعت و کار کردن را بهبود بخشد.

۱،۱،۱ تاریخچه

پروژه اُسی وی به صورت رسمی در سال ۱۹۹۹ به منظور حل مشکل کاربردهای شدیداً پردازندۀای^۳ به عنوان قسمتی از سری پروژه‌های شامل رדיابی اشعه بی‌درنگ و دیوارهای نمایشگر سه بعدی، شروع شد. مهم‌ترین شرکت کنندگان در این پروژه شامل تعدادی متخصصان بهینه سازی در اینتل روسیه و همچنین تیم کتابخانه کارایی^۴ اینتل بودند. در روزهای ابتدایی، هدف پروژه به صورت زیر تشریح شده بود:

- پیشرفت تحقیقات بینایی کامپیوتر به واسطه ارائه نه تنها یک کد آزاد، بلکه یک کد بهینه برای کارهای ساده زیربنایی در بینایی کامپیوتر.
- انتشار علم بینایی کامپیوتر به واسطه ارائه یک زیربنایی مشترک که توسعه دهنده‌گان بتوانند روی آن برنامه بسازند و بنابراین قابلیت خوانایی و انتقال دهی کد افزایش یابد.
- ارتقا کاربردهای تجاری مبتنی بر بینایی کامپیوتر به واسطه ساخت کدهای آزاد، قابل حمل و بهینه شده.

اولین نسخه آلفای اُسی وی در کنفرانس بینایی کامپیوتر و کشف الگوی IEEE در سال ۲۰۰۰ به عموم معرفی شد و ۵ نسخه بتا بین سال‌های ۲۰۰۱ و ۲۰۰۵ منتشر شد. نسخه ۱،۰ در ۲۰۰۶ انتشار یافت. در اواسط ۲۰۰۸، Willow Garage از اُسی وی حمایت کرد و هم اکنون در حال توسعه فعال است. یک نسخه ۱،۱ به عنوان "پیش انتشار" در اکتبر ۲۰۰۸ منتشر شد.

نسخه دوم اُسی وی در اکتبر ۲۰۰۹ منتشر شد. اُسی وی ۲ تغییرهای چشم گیری در رابط C++ داشت که عموماً به هدف ارائه الگوهای آسانتر و نوع امنتر^۵ و توابع جدید و پیاده سازی بهتر توابع قبلی برای کارایی بهتر (مخصوصاً روی پردازندۀای چند هسته‌ای) بود. در حال حاضر نسخه‌های رسمی هر شش ماه یک بار منتشر می‌شود و توسعه این کتابخانه توسط یک تیم مستقل روسی که از سوی شرکت‌های تجاری حمایت می‌شود، انجام می‌گیرد.

در آگوست ۲۰۱۲ حمایت از اُسی وی از طریق یک سازمان غیر انتفاعی به نام OpenCV.org شکل گرفت. این سازمان شامل یک سایت برای کاربران و یک سایت برای توسعه دهنده‌گان است.

OpenCV^۱

^۲ برنامه‌ای جهت استفاده بهینه‌تر از توانایی پردازندۀای شرکت اینتل.

^۳ CPU Intensive

^۴ Performance Library

^۵ Type safe^۶

۱۱،۲ کاربردها

زمینه‌های کاربرد اُسی وی شامل موارد زیر است:

- تخمین^۶ Egomotion
- سیستم تشخیص چهره^۷
- تشخیص اشاره^۸
- تعامل انسان کامپیوتر (HCI)^۹
- رباتیک موبایل^{۱۰}
- درک حرکت^{۱۱}
- شناسایی شیء^{۱۲}
- تقسیم بندی و تشخیص^{۱۳}
- چشم انداز عمق استریو^{۱۴}: ادراک عمق از دو دوربین
- ساختار از حرکت (SFM)^{۱۵}
- ردیابی حرکت^{۱۶}
- واقعیت افزوده^{۱۷}

برای پشتیبانی از برخی از موارد بالا، اُسی وی یک کتابخانه یادگیری ماشین آماری، که شامل موارد زیر است را در بر دارد:

- بوستینگ^{۱۸} (الگوریتم متا)
- یادگیری درخت تصمیم گیری^{۱۹}
- درختان افزایش گرادیان^{۲۰}
- الگوریتم بیشینه سازی امید ریاضی^{۲۱}
- الگوریتم K نزدیکترین همسایه^{۲۲}
- دسته بند کننده نایو بیز^{۲۳}

Egomotion estimation ^۶
Facial recognition system ^۷
Gesture recognition ^۸
Human-computer interaction ^۹
Mobile robotics ^{۱۰}
Motion understanding ^{۱۱}
Object identification ^{۱۲}
Segmentation and Recognition ^{۱۳}
Stereopsis Stereo vision ^{۱۴}
Structure From Motion ^{۱۵}
Motion tracking ^{۱۶}
Augmented reality ^{۱۷}
Boosting ^{۱۸}
Decision tree ^{۱۹}
Gradient boosting trees ^{۲۰}
Expectation-maximization algorithm ^{۲۱}
K nearest neighbor ^{۲۲}
Naïve Bayes classifier ^{۲۳}

- شبکه‌های عصبی مصنوعی^{۲۴}
- جنگل تصادفی^{۲۵}
- ماشین برداری پشتیبان (SVM)^{۲۶}

۱,۱,۳ زبان برنامه‌نویسی

اُسی وی ۲ با C++ نوشته شده است و رابط اصلی آن همین C++ است ولی همچنان می‌توانید از رابط C باقی مانده (از نسخه اول) در آن استفاده کنید. در حال حاضر رابطه‌ای کاملی برای زبان‌های MATLAB/OCTAVE و Python و Java وجود دارد. API‌های این رابطها را می‌توانید در مستندات آنلاین پیدا کنید. به منظور تشویق توسعه دهنده‌گان زبان‌های دیگر از جمله Ruby و Ch، C# و به استفاده از این کتابخانه، رابطه‌ای برای آنها نیز توسعه داده شده است.

در حال حاضر تمامی توسعه‌ها و الگوریتم‌های جدید در C++ توسعه می‌یابند.

۱,۱,۴ سیستم‌عامل‌ها

تا این لحظه اُسی وی روی سیستم‌عامل‌های BlackBerry، iOS، OpenBSD، FreeBSD، Maemo، Android، Windows، SourceForge و OS X قابل اجرا است. کاربر می‌تواند برای استفاده در هر کدام از این سیستم‌عامل‌ها، نسخه‌های رسمی را از دریافت کند.

۱,۲ بارگذاری، نمایش و ذخیره تصویر

در این بخش می‌خواهیم شما را با سه عمل بنیادی آشنا کنیم. تقریباً در تمام کاربردهای پردازش تصویری به سه عمل بارگذاری، نمایش و ذخیره تصویر نیاز داریم. در ادامه با نحوه انجام این کارها در اُسی وی آشنا می‌شویم.

۱,۲,۱ کد

```

1 #include <opencv2/core/core.hpp>
2 #include <opencv2/highgui/highgui.hpp>
3 #include <opencv2/imgproc/imgproc.hpp>
4
5 int main( int argc, char** argv )
6 {
7     char* imageName = argv[1];
8
9     cv::Mat image;
10    image = cv::imread( imageName, CV_LOAD_IMAGE_COLOR );
11
12    if(!image.data )
13    {
14        printf( " No image data \n " );
15        return -1;
16    }
17
18    cv::Mat gray_image;
19    cv::cvtColor( image, gray_image, CV_BGR2GRAY );
20
21    cv::imwrite( argv[2], gray_image );
22
23    cv::namedWindow( imageName, CV_WINDOW_AUTOSIZE );

```

```

24     cv::namedWindow( "Gray image", CV_WINDOW_AUTOSIZE );
25
26     cv::imshow( imageName, image );
27     cv::imshow( "Gray image", gray_image );
28
29     cv::waitKey(0);
30
31     return 0;
32 }

```

۱,۲,۲ توضیح

در خطوط ۱ تا ۳ فایل‌های سرایند مورد نیاز را وارد برنامه می‌کنیم. فایل سرایند core.hpp مربوط به ماژول Core در آسی وی است که توابع و متغیرهای عمومی در آن قرار دارند. کلاس Mat در این ماژول تعریف شده است.

فایل سرایند highgui.hpp هم مربوط به ماژولی با همین نام در آسی وی است. در این فایل تابع و کلاس‌های مربوط به رابط گرافیکی آسی وی قرار دارد. توابع imread، imshow و waitKey در این ماژول تعریف شده‌اند.

فایل سرایند imgproc.hpp مربوط به ماژول imgProc است. در این ماژول تابع و الگوریتم‌های پردازش تصویر قرار گرفته‌اند. تابع cvtColor و پرچم‌های آن در این ماژول تعریف شده‌اند.

در خط ۷ آدرس تصویر ورودی را از آرگومان ورودی برنامه می‌گیریم و در متغیر imageName قرار می‌دهیم.

در خط ۹ یک شیء از کلاس Mat درست می‌کنیم. قرار است تصویر ورودی در این شیء قرار گیرد.

در خط ۱۰ با استفاده از تابع imread تصویر ورودی را می‌خوانیم. این تابع دو آرگومان دارد. آرگومان اول آدرس تصویر است و آرگومان دوم نحوه بارگذاری تصویر است. در اینجا مشخص کردایم که تصویر به صورت رنگی بارگذاری شود. اگر می‌خواستیم تصویر به صورت سیاه و سفید بارگذاری شود، باید از پرچم CV_LOAD_IMAGE_GRAYSCALE استفاده می‌کردیم.

در خط ۱۲ چک می‌کنیم که آیا تصویر درست بارگذاری شده است یا نه. اگر درست بارگذاری نشده باشد یک پیغام خط‌نمایش می‌دهیم و از برنامه خارج می‌شویم.

در خط ۱۹ با استفاده از تابع cvtColor فضای رنگی تصویر ورودی را از رنگی به سیاه و سفید می‌بریم. آرگومان اول این تابع تصویر ورودی و آرگومان دوم تصویر خروجی است. آرگومان سوم مشخص می‌کند که از چه فضای رنگی به چه فضای رنگی می‌خواهیم تبدیل کنیم.

در خط ۲۱ با استفاده از تابع imwrite تصویر سیاه و سفید را ذخیره می‌کنیم. آرگومان اول این تابع آدرس تصویر خروجی است و آرگومان دوم هم تصویری است که می‌خواهیم ذخیره کنیم. دقت کنید که پسوند نام تصویر خروجی تعیین می‌کند که تصویر خروجی از چه نوعی باشد. مثلاً اگر نام تصویر خروجی jpg باشد، تصویر خروجی به فرمت jpg نوشته می‌شود.

در خطوط ۲۳ و ۲۴ دو پنجره برای نمایش تصویر ورودی و تصویر خروجی درست می‌کنیم. آرگومان اول این تابع نام پنجره و آرگومان دوم تنظیمات پنجره است.

در خطوط ۲۶ و ۲۷ با استفاده از تابع imshow تصویر ورودی و تصویر خروجی را در پنجره‌هایی که درست کردیم، نمایش می‌دهیم. آرگومان اول این تابع نام پنجره و آرگومان دوم تصویری است که می‌خواهیم نمایش دهیم.

در خط ۲۹ با استفاده از تابع waitKey کاری می‌کنیم که برنامه ما تا زمانی که کاربر کلیدی فشار نداده است همچنان در حال اجرا باشد.

۱,۲,۳ خروجی

برنامه را به صورت زیر اجرا می‌کنیم:

```
SimpleProgram.exe "D:/input.jpg" "D:/output.jpg"
```



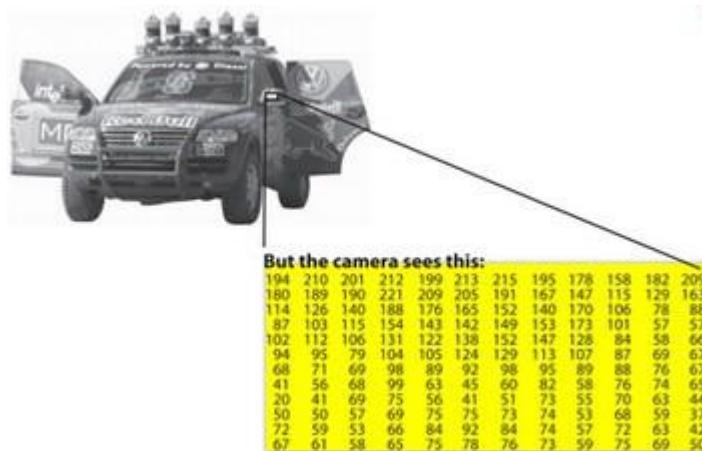
تصویر خروجی



تصویر ورودی

MAT ۱,۳ کلاس

راههای متفاوتی برای به دست آوردن عکس‌های دیجیتال از دنیای بیرون وجود دارد: دوربین‌های دیجیتال، اسکنرهای سی‌تی اسکن یا عکس‌برداری رزونانس مغناطیسی، تنها چند مورد از این راه‌ها هستند. در هر صورت چیزی که ما (انسان‌ها) می‌بینیم، تنها یک عکس است. البته چیزی که دستگاه‌های دیجیتال ما ذخیره می‌کنند، تعدادی عدد به ازای هر نقطه در تصویر است.



مثلاً در تصویر بالا می‌بینید که آینه خودرو تنها یک ماتریس عددی است؛ هر کدام از این عددها نمایان‌گر میزان روشنایی^{۷۷} یک نقطه عکس هستند. در نهایت همه عکس‌های موجود در یک کامپیوتر، تبدیل به ماتریس‌های عددی به علاوه اطلاعات شرح دهنده آن‌ها می‌شوند.

در ابتدای توزیع OpenCV، تنها از رابط C پشتیبانی می‌شد. در آن زمان برای ذخیره عکس‌ها در حافظه از داده ساختار `IplImage` استفاده می‌شد. این روش تمام جنبه‌های بد زبان C را روی کار می‌آورد. اصلی‌ترین مشکل، مدیریت دستی حافظه بود که برنامه نویس را مسئول اختصاص و باز پس گیری حافظه می‌کرد. البته ممکن است این موضوع در برنامه‌های کوچک خود نمایی نکند، ولی زمانی که برنامه‌ها بزرگ و بزرگ‌تر می‌شوند، دست و پنجه نرم کردن با این قضیه بسیار طاقت فرسا خواهد شد به گونه‌ای که شما را از هدف اصلیتان دور می‌کند و به یک برنامه نویس سیستم مدیریت حافظه تبدیل می‌کند!

خوشنختانه با روی کار آمدن C++ و معرفی مفهوم شی‌گرایی، راهی جدید به روی برنامه نویسان باز شد: مدیریت خودکار حافظه (البته کم و بیش). کاملاً با C سازگار بوده و هیچ نگرانی در مورد برنامه‌های قدیمی وجود ندارد. بنابراین در نسخه دوم OpenCV جدید C++ با بهره‌گیری از خواص شی‌گرایی C++ ارائه شد که راهی جدید برای انجام کارها به برنامه نویس پیشنهاد می‌داد. راهی که

برنامه نویس را کمتر در گیر مدیریت حافظه می کرد و کدها را شفاف تر می ساخت. قانونی نانوشته که می گفت «با کمتر نوشت، چیزهای بیشتری به دست بیاور».

اولین چیزی که باید درباره **Mat** بدانید این است که دیگر نیازی به اختصاص دستی حافظه و آزاد کردن آن پس از اتمام کار ندارید. بیشتر تابع های OpenCV داده های خروجی شان را به صورت خودکار در مکانی از حافظه قرار می دهند (البته می توان این کار را به صورت دستی هم انجام داد). می توان یک شیء **Mat** ساخته شده را به تابعی ارسال کرد تا اگر اندازه شیء ارسال شده فضای مورد نیاز را ارضا کرد، تابع از آن شیء برای ذخیره داده ها استفاده کند. با این ویژگی می توانید در تمام برنامه تنها از اندازه های از حافظه استفاده کنید که برای انجام کار تان احتیاج دارید؛ نه بیشتر و نه کمتر.

Mat در واقع از دو قسمت تشکیل شده است:

۱. هدر که حاوی اطلاعاتی از قبیل اندازه ماتریس، شیوه ذخیره سازی، آدرس محل ذخیره ماتریس و... است.
۲. یک اشاره گر به ماتریسی که مقادیر پیکسل ها در آن ذخیره شده است و بسته به شیوه ذخیره سازی، ممکن است یک یا چند بعد داشته باشد.

اندازه هدر برای همه ماتریس ها ثابت است ولی اندازه ماتریس داده ممکن است از یک عکس تا عکس دیگر متفاوت باشد. بنابراین وقتی در قسمتی از برنامه می خواهید عکسی را کپی کنید، بیشتر زمان صرف کپی کردن قسمت داده ای ماتریس می شود.

با وجود آن که یک ماتریس داده می تواند بین چند شیء **Mat** مشترک باشد، زمانی که نیازی به آن نیست، آخرین شیئی که از آن استفاده کرده، آن را پاک خواهد کرد. بدین منظور از یک مکانیسم شمارش مرجع استفاده می شود. وقتی هدر یک شیء **Mat** کپی می شود، یک واحد به شمارنده درون ماتریس کپی شده اضافه شده و وقتی یک هدر پاک شود، این شمارنده کاهش می یابد؛ وقتی شمارنده به صفر رسید ماتریس آزاد می شود.

۱,۳,۱ شیوه ذخیره سازی داده ها در کلاس **Mat**

از این کلاس می توان برای ذخیره سازی یک ماتریس عددی n بعدی یک کاناله یا چند کاناله استفاده کرد. می توانید عکس های سیاه و سفید یا رنگی، بردارها و ماتریس های عدد مختلط، نقاط، هیستوگرامها و ... را در آن ذخیره کرد. چیدمان داده ها در ماتریس M به وسیله آرایه $M.step[]$ مشخص می شود؛ آدرس عناصر این ماتریس که به صورت $(i_0, i_1, \dots, i_{M.dims-1})$ است از طریق فرمول زیر محاسبه می شود:

$$addr(M_{i_0, \dots, i_{M.dims-1}}) = M.data + M.step[0] * i_0 + M.step[1] * i_1 + \dots + M.step[M.dims - 1] * i_{M.dims-1}$$

برای آرایه های دو بعدی فرمول بالا به صورت زیر کاهش می یابد:

$$addr(M_{i,j}) = M.data + M.step[0] * i + M.step[1] * j$$

توجه کنید که رابطه $M.step[i] \geq M.step[i+1] * M.size[i+1]$ همواره برقرار است. این یعنی ماتریس های دو بعدی به صورت سطر و ماتریس های سه بعدی به صورت صفحه به صفحه و به همین صورت برای ماتریس های با ابعاد بالاتر، شیوه ذخیره سازی ادامه می یابد. $M.step[M.dims - 1]$ کوچک ترین است و همواره برابر با $M.elemSize()$ است.

۱,۳,۲ ساخت شیء **Mat**

راه های خیلی زیادی برای ساخت یک شیء **Mat** وجود دارد. محبوب ترین راه ها به صورت زیر هستند:

• سازنده ^{۲۸}: **Mat**

```
cv::Mat M(2,2, CV_8UC3, cv::Scalar(0,0,255));
```

```
std::cout << "M = " << std::endl << " " << M << std::endl;
```

M =

$$\begin{bmatrix} 0, 0, 255, 0, 0, 255 \\ 0, 0, 255, 0, 0, 255 \end{bmatrix}$$

برای عکس‌های دو بعدی و چند کanalه ابتدا تعداد ردیف‌ها و ستون‌ها را تعریف می‌کنیم. سپس نوع داده‌ای که برای ذخیره عناصر استفاده می‌کنیم و همچنین تعداد کanalهای ماتریس به ازای هر پیکسل را مشخص کرد. برای مشخص کردن نوع داده‌ها و تعداد کanalها می‌توان از روش قراردادی زیر استفاده کرد:

[تعداد کanalها]C[پیشوند نوع][علامت دار یا بی علامت][تعداد بیت‌ها برای هر آیتم]CV_Item

مثال CV_8UC3 یعنی نوع داده ۸ بیتی بی علامت سه کanalه است. این مقادیر برای حداکثر ۴ کanal از پیش تعریف شده‌اند. اگر به بیشتر از ۴ کanal نیاز دارید، باید با استفاده از ماکرو زیر داده مورد نظر را تعریف کنید:

(تعداد کanalها)C[پیشوند نوع][علامت دار یا بی علامت][تعداد بیت‌ها برای هر آیتم]

مثال CV_8UC(5) یک نوع ۸ بیتی بدون علامت ۵ کanalه درست می‌کند.

Scalar هم یک بردار چهار عنصری است و با استفاده از آن می‌توان در تمام پیکسل‌های ماتریس تعریف شده، مقدار مشخصی را قرار داد.

- استفاده از آرایه برای مشخص کردن ابعاد:

```
int sz[3] = {2,2,2};  
cv::Mat I(3,sz, CV_8UC(1), cv::Scalar::all(0));
```

مثال بالا طریقہ ساخت یک ماتریس با بیش از دو بعد را نشان می‌دهد. ابتدا با استفاده از یک آرایه ابعاد آن را مشخص می‌کنیم و سپس اشاره گر آن آرایه را به سازنده Mat ارسال می‌کنیم.

• تابع :create()

```
M.create(4,4, CV_8UC(2));  
std::cout << "M = " << std::endl << " " << M << std::endl;
```

M =

$$\begin{bmatrix} 205, 205 \end{bmatrix}$$

در این روش نمی‌توان ماتریس را مقدار دهی اولیه کرد. این تابع فقط در صورتی که ماتریس قدیمی به اندازه کافی فضا برای ماتریس جدید نداشته باشد، آن را مجدداً مقدار دهی می‌کند.

• MATLAB شیوه: با استفاده از توابع zeros() و ones() می‌توان مطابق با روش متلب، ماتریس‌های مختلفی ساخت:

```

cv::Mat E = cv::Mat::eye(4, 4, CV_64F);
std::cout << "E = " << std::endl << " " << E << std::endl;

cv::Mat O = cv::Mat::ones(2, 2, CV_32F);
std::cout << "O = " << std::endl << " " << O << std::endl;

cv::Mat Z = cv::Mat::zeros(3,3, CV_8UC1);
std::cout << "Z = " << std::endl << " " << Z << std::endl;

```

E =

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
Z =

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$
O =

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

- برای ماتریس‌های کوچک می‌توان از روش زیر استفاده کرد:

```

cv::Mat C = (cv::Mat<double>(3,3) << 0, -1, 0, -1, 5, -1, 0, -1, 0);
std::cout << "C = " << std::endl << " " << C << std::endl;

```

C =

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

در اینجا `Mat` یک کلاس الگو برای `Mat` است که در بخش‌های بعد به طور مفصل توضیح داده شده است.

- کردن قسمتی از یک ماتریس `copyTo` یا `clone`

```

cv::Mat RowClone = C.row(1).clone();
std::cout << "RowClone = " << std::endl << " " << RowClone;

```

در این روش تمام ماتریس (یعنی هم هدر و ماتریس داده) کپی می‌شوند.

RowClone =

$$\begin{bmatrix} -1 & 5 & -1 \end{bmatrix}$$

۱,۳,۳ دسترسی به عناصر درون‌شیء `Mat`

برای دسترسی به عناصر درون یک شیء `Mat` ۴ راه وجود دارد که در زیر هر یک را به صورت جداگانه بررسی می‌کنیم.

برای نشان دادن این روش‌ها و همچنین مقایسه آنها با هم یک کاربرد واحد را در نظر می‌گیریم. هدف این کاربرد جایگزینی مقادیر یک ماتریس بر اساس یک جدول است.

۱,۳,۳,۱ روش بهینه

وقتی بحث کارایی پیش می‌آید، نمی‌توان در مقابل شیوه قدیمی زبان C برای دسترسی به عناصر حافظه، یعنی عملگر `[]` (اشاره‌گر) ایستاد. بنابراین بهینه‌ترین روشی که می‌توان پیشنهاد کرد روش زیر است:

```

1   cv::Mat& ScanImageAndReduceC(cv::Mat& I, const uchar* const table)
2   {
3       // accept only char type matrices
4       CV_Assert(I.depth() != sizeof(uchar));
5       int channels = I.channels();

```

```

6     int nRows = I.rows;
7     int nCols = I.cols * channels;
8     if (I.isContinuous())
9     {
10         nCols *= nRows;
11         nRows = 1;
12     }
13     int i,j;
14     uchar* p;
15     for( i = 0; i < nRows; ++i)
16     {
17         p = I.ptr<uchar>(i);
18         for( j = 0; j < nCols; ++j)
19         {
20             p[j] = table[p[j]];
21         }
22     }
23     return I;
24 }
```

در اینجا به صورت ساده فقط یک اشاره گر به ابتدای هر سطر به دست آورده و سپس آن را تا انتهای ادامه می‌دهیم. در حالت خاصی که ماتریس تنها در یک سطر ذخیره شده، کافی است تنها یک بار اشاره گر سطر را درخواست کنیم و سپس تا انتهای آن رفته تا به تمام پیکسل‌های ماتریس دسترسی داشته باشیم. باید مراقب عکس‌های رنگی بود؛ از آنجایی که در این عکس‌ها سه کانال داریم، پس لازم است روی سه برابر آیتم بیشتر در هر سطر حرکت کنیم.

راه دیگری هم برای این کار وجود دارد. متغیر `data` در شیء `Mat`، اشاره گری به اولین سطر و ستون است. اگر این اشاره گر پوچ باشد، داده معتبری در آن شیء وجود نخواهد داشت. بررسی کردن این اشاره گر ساده‌ترین راه برای مطمئن شدن از بارگذاری صحیح عکس است. در حالتی که ذخیره سازی به صورت پشت سر هم باشد، می‌توان از این روش برای دسترسی به پیکسل‌ها استفاده کرد. در مورد عکس سیاه و سفید به صورت زیر عمل می‌کنیم:

```

uchar* p = I.data;
for( unsigned int i = 0; i < ncol*nrows; ++i)
    *p++ = table[*p];
```

نتیجه این کد مطابق با نتیجه کد قبل است با این تفاوت که در اینجا با کدی پیچیده‌تر روبرو هستیم که خوانایی کمتری هم دارد. جالب است بدانید که این کد در اجرا عملکرد مشابه ای با کد قبل دارد.

۱.۳.۳.۲ روش/من

در روش بهینه برنامه نویس باید مراقب حفره‌های احتمالی بین سطراها باشد و تعداد فیلدهای مورد بررسی را کنترل کند (تا بیشتر یا کمتر از مقدار موجود نشود). روش امن از آن جهت که این نگرانی‌ها را ندارد، روش امن‌تری است. تنها کاری که باید انجام داد این است که `iterator` شروع و پایان ماتریس را درخواست کرده و سپس فقط `iterator` شروع را افزایش داده تا به `iterator` پایان برسد. برای به دست آوردن مقدار اشاره شده توسط `iterator`، از عملگر `*` استفاده می‌کنیم (قبل از آن قرار می‌دهیم).

```

1   cv::Mat& ScanImageAndReduceIterator(cv::Mat& I, const uchar* const
2   table)
3   {
4       // accept only char type matrices
5       CV_Assert(I.depth() != sizeof(uchar));
6       const int channels = I.channels();
7       switch(channels)
8       {
```

```
9  case 1:
10 {
11     cv::MatIterator<uchar> it, end;
12     for( it = I.begin<uchar>(), end = I.end<uchar>(); it != end;
13         ++it)
14         *it = table[*it];
15     break;
16 }
17 case 3:
18 {
19     cv::MatIterator<Vec3b> it, end;
20     for( it = I.begin<Vec3b>(), end = I.end<Vec3b>(); it != end;
21         ++it)
22     {
23         (*it)[0] = table[(*it)[0]];
24         (*it)[1] = table[(*it)[1]];
25         (*it)[2] = table[(*it)[2]];
26     }
27 }
28 }
29 return I;
30 }
```

در عکس‌های رنگی سه آیتم `uchar` در هر ستون وجود دارد و می‌توان آنها را به صورت یک بردار کوچک از آیتم‌های `uchar` در نظر گرفت که در `OpenCV` با نام `Vec3b` شناخته می‌شوند. بنابراین اگر از یک `uchar iterator` نوع `iterator` استفاده کنید، تنها به مقادیر کanal آبی دسترسی خواهد داشت. برای دسترسی به `n` امین زیر-ستون از عملگر ساده¹⁰ استفاده می‌کنیم. به خاطر بسیاری‌که `iterator`‌های `OpenCV` روی ستون‌ها حرکت می‌کنند و به صورت خودکار به سطر بعدی می‌روند.

روش بی‌درنگ ۱,۳,۳,۳

استفاده از این روش برای مرور تصاویر پیشنهاد نمی‌شود. از این روش باید زمانی که تعداد دسترسی‌های مiman کم است استفاده کرد. برای استفاده از روش بی‌درنگ، باید سطر و ستون پیکسل مورد نظر و نوع داده را مشخص کیم. برای عکس‌های سیاه و سفید به صورت زیر

```
1 cv::Mat& ScanImageAndReduceRandomAccess(cv::Mat& I, const uchar* const
2 table)
3 {
4     // accept only char type matrices
5     CV_Assert(I.depth() != sizeof(uchar));
6     const int channels = I.channels();
7     switch(channels)
8     {
9         case 1:
10    {
11        for( int i = 0; i < I.rows; ++i)
12            for( int j = 0; j < I.cols; ++j )
13                I.at<uchar>(i,j) = table[I.at<uchar>(i,j)];
14        break;
15    }
16    case 3:
17    {
18        cv::Mat_<Vec3b> _I = I;
19        for( int i = 0; i < I.rows; ++i)
20            for( int j = 0; j < I.cols; ++j )
21            {
22                I(i,j)[0] = table[ _I(i,j)[0]];

```

```

23             _I(i,j)[1] = table[_I(i,j)[1]];
24             _I(i,j)[2] = table[_I(i,j)[2]];
25         }
26     I = _I;
27     break;
28 }
29 }
30 return I;
31 }
```

تابع (`at()`) نوع داده را به صورت الگو از ورودی می‌گیرد و آدرس آیتم درخواست شده را به صورت بی‌درنگ محاسبه می‌کند و سپس یک اشاره گر به آن بر می‌گردد.

در خط ۵ بررسی می‌کنیم که ماتریس `I` از نوع کاراکتری باشد (یعنی یک بایتی باشد).

اگر می‌خواهید به صورت مکرر به منظور دسترسی به پیکسل‌های یک عکس از این روش استفاده کنید، نوشتن نوع داده و کلید واژه `at` برای هر دسترسی می‌تواند در دسر ساز و وقت گیر باشد. برای حل این مشکل، `OpenCV` یک نوع داده به نام `_Mat` ارائه می‌دهد که کاملاً مشابه `Mat` است و فقط باید هنگام تعریف آن، نوع داده را مشخص کرد. در این داده ساختار می‌توان از عملگر `()` برای دسترسی سریع به عناصر ماتریس استفاده کرد. همچنین می‌توانید داده ساختارهای `Mat` و `Vec3b` را به راحتی به هم تبدیل کنید. در خط ۱۸ کد بالا مشاهده می‌کنید که ابتدا ماتریس `I` را به یک شیء از نوع `_Mat` با نوع داده `Vec3b` تبدیل می‌کنیم، سپس در خطوط ۲۴ تا ۲۶ با استفاده از عملگر `()` به داده‌های هر عنصر ماتریسی دسترسی پیدا می‌کنیم. به هر حال سرعت این روش در مقایسه با روش اول (یعنی استفاده از تابع `(at())` هیچ فرقی نمی‌کند).

۱.۳.۳.۴ روش جدول جستجو

در این روش از جدول جستجو برای تغییر مقدار پیکسل‌های یک عکس استفاده می‌کنیم. جایگزین کردن مقدار پیکسل‌های یک عکس با مقدارهای جدید یک کار معمول در پردازش تصویر است. به همین خاطر `OpenCV` تابعی ارائه داده که این کار را به صورت خودکار برای ما انجام می‌دهد و دیگر نیازی نیست که خودمان روی همه پیکسل‌های تصویر حرکت کنیم و مقدار پیکسل‌های را با مقدارهای جدید جایگزین کنیم. این تابع (`LUT()`) است. برای شروع باید یک جدول جستجو بسازیم. برای این کار از یک شیء `Mat` استفاده می‌کنیم:

```

cv::Mat lookUpTable(1, 256, CV_8U);
uchar* p = lookUpTable.data;
for( int i = 0; i < 256; ++i)
    p[i] = table[i];
```

سپس تابع `LUT` را روی عکس ورودی صدا می‌زنیم (اً عکس ورودی و ل هم عکس خروجی است):

```
cv::LUT(I, lookUpTable, J);
```

۱.۳.۳.۵ مقایسه روش‌ها

برای اینکه بهتر بتوان تفاوت کارایی را دید، از یک عکس رنگی بسیار بزرگ (2560 X 1600 پیکسلی) استفاده شده و به منظور حصول دقیق‌تر، از زمان‌های به دست آمده در ۱۰۰ بار اجرای متوالی برای هر روش، میانگین گرفته شده است.

روش	زمان اجرا (میلی ثانیه)
بهمنه	۷۹,۴۷۱۷
امن	۸۳,۷۲۰۱
بی‌درنگ	۹۳,۷۸۷۸
جدول جستجو	۳۲,۵۷۵۹

از این جدول چند نتیجه می‌توان گرفت:

۱. تا حد امکان از تابع‌های که OpenCV ارائه داده است استفاده کنید زیرا کتابخانه OpenCV از چند نخی پشتیبانی می‌کند.
۲. روش امن با وجود اینکه امن‌ترین روش است اما بسیار کند عمل می‌کند.
۳. پژوهش‌نامه‌ترین روش، روش بی‌درنگ است. البته این در حالت اجرای خطایابی است و ممکن است در حالت انتشار بهتر از روش امن عمل کند ولی مطمئناً در زمینه امنیت از روش امن عقب‌تر است.

۱.۳.۴ عملگرهای ریاضی

در لیست زیر عملگرهای ریاضی پشتیبانی شده توسط کلاس Mat را می‌بینید. می‌توانید هر کدام از این عملگرهای را برای ایجاد عبارت‌های پیچیده با هم ترکیب کرد. در اینجا A و B ماتریس‌هایی از نوع Mat هستند و s یک نوع Scalar است و alpha هم یک مقدار حقیقی اسکالر است.

- جمع، تفریق و نفیض: $A + B, A - B, A + s, A - s, s + A, s - A, -A$
- مقیاس گذاری $\alpha \cdot A$
- ضرب و تقسیم درایه‌ای: $A \cdot \text{mul}(B), A/B, \text{alpha}/A$
- ضرب ماتریسی: $A * B$
- ترانهاده: $A.t()$
- معکوس و شبه معکوس ماتریس، حل سیستم‌های خطی و مسائل کمترین مربعات:

$$A.\text{inv}([\text{method}]) (\sim A^{-1}), A.\text{inv}([\text{method}]) * B (\sim X: AX = B)$$

- مقایسه: عبارت‌های $A < B, A > B, A == B, A != B$ می‌تواند یکی از عملگرهای مقدار ۲۵۵ و در صورت برقرار نبودن مقدار صفر دارند.
- عملگرهای بیتی: عبارت‌های $A \& B, A \mid B, A \sim B, A \text{ logicop } s, s \text{ logicop } A, \sim A$ می‌تواند یکی از عملگرهای \wedge, \mid, \sim باشد.
- کمینه و بیشینه درایه‌ای: $\min(A, B), \max(A, B), \min(A, \text{alpha}), \max(A, \text{alpha})$
- قدر مطلق درایه‌ای: $\text{abs}(A)$
- ضرب داخلی $\text{dot}(A)$ و خارجی $\text{cross}(A)$
- توابعی مثل $\text{repeat}, \text{determinant}, \text{trace}, \text{countNonZero}, \text{sum}, \text{mean}, \text{norm}$ که یک ماتریس یا یک مقدار اسکالار بر می‌گردانند.
- مقدار دهنده‌های ماتریس (مثل $\text{Mat::eye}()$ و $\text{Mat::zeros}()$ و $\text{Mat::ones}()$)، مقدار دهنده‌های به وسیله کاما، سازنده‌های ماتریس و عملگرهای که یک زیر ماتریس را استخراج می‌کنند.
- سازنده‌های به شکل $\text{Mat}_{\langle \text{destination_type} \rangle}()$ که ماتریس را به نوع مناسب تبدیل می‌کنند.

۱.۴ ذخیره سازی و بازیابی اطلاعات

در این بخش می خواهیم شما را با شیوه استاندارد ذخیره سازی و بازیابی داده ساختارهای موجود در آسی وی آشنا کنیم. البته با این روش می توان داده ساختارهایی که به نوعی از داده ساختارهای آسی وی درست شده اند را نیز ذخیره بازیابی کرد.

کد ۱.۴.۱

```

1 #include <opencv2/core/core.hpp>
2 #include <iostream>
3 #include <string>
4
5 using namespace cv;
6 using namespace std;
7
8 static void help(char** av)
9 {
10     cout << endl
11         << av[0] << " shows the usage of the OpenCV serialization
12 functionality." << endl << "usage: " << endl
13         << av[0] << " outputfile.yml.gz" << endl
14         << "The output file may be either XML (xml) or YAML
15 (yml/yaml). You can even compress it by "
16         << "specifying this in its extension like xml.gz yaml.gz
17 etc..." << endl
18         << "With FileStorage you can serialize objects in OpenCV by
19 using the << and >> operators" << endl
20         << "For example: - create a class and have it serialized"
21 << endl
22         << "                                - use it to read and write matrices."
23 << endl;
24 }
25
26 class MyData
27 {
28 public:
29     MyData() : A(0), X(0), id()
30     {}
31     explicit MyData(int) : A(97), X(CV_PI), id("mydata1234") // //Write
32 explicit to avoid implicit conversion
33     {}
34     void write(FileStorage& fs) const //Write
35 serialization for this class
36     {
37         fs << "{" << "A" << A << "X" << X << "id" << id << "}";
38     }
39     void read(const FileNode& node) //Read
40 serialization for this class
41     {
42         A = (int)node["A"];
43         X = (double)node["X"];
44         id = (string)node["id"];
45     }
46 public: // Data Members
47     int A;
48     double X;
49     string id;
50 };
51
52

```

```

53 //These write and read functions must be defined for the
54 serialization in FileStorage to work
55 static void write(FileStorage& fs, const std::string&, const MyData&
56 x)
57 {
58     x.write(fs);
59 }
60 static void read(const FileNode& node, MyData& x, const MyData&
61 default_value = MyData()) {
62     if(node.empty())
63         x = default_value;
64     else
65         x.read(node);
66 }
67
68 // This function will print our custom class to the console
69 static ostream& operator<<(ostream& out, const MyData& m)
70 {
71     out << "{ id = " << m.id << ", ";
72     out << "X = " << m.X << ", ";
73     out << "A = " << m.A << "}";
74     return out;
75 }
76
77 int main(int ac, char** av)
78 {
79     if (ac != 2)
80     {
81         help(av);
82         return 1;
83     }
84
85     string filename = av[1];
86     { //write
87         Mat R = Mat<uchar>::eye(3, 3),
88             T = Mat<double>::zeros(3, 1);
89         MyData m(1);
90
91         FileStorage fs(filename, FileStorage::WRITE);
92
93         fs << "iterationNr" << 100;
94         fs << "strings" << "["; // text - string sequence
95         fs << "image1.jpg" << "Awesomeness" << "baboon.jpg";
96         fs << "]"; // close sequence
97
98         fs << "Mapping"; // text - mapping
99         fs << "{" << "One" << 1;
100        fs << "Two" << 2 << "}";
101
102        fs << "R" << R; // cv::Mat
103        fs << "T" << T;
104
105        fs << "MyData" << m; // your own data structures
106
107        fs.release(); // explicit close
108        cout << "Write Done." << endl;
109    }
110
111    { //read
112        cout << endl << "Reading: " << endl;
113        FileStorage fs;

```

```

114         fs.open(filename, FileStorage::READ);
115
116         int itNr;
117         //fs["iterationNr"] >> itNr;
118         itNr = (int) fs["iterationNr"];
119         cout << itNr;
120         if (!fs.isOpened())
121     {
122             cerr << "Failed to open " << filename << endl;
123             help(av);
124             return 1;
125         }
126
127         FileNode n = fs["strings"]; // Read string sequence - Get
128     node
129         if (n.type() != FileNode::SEQ)
130     {
131             cerr << "strings is not a sequence! FAIL" << endl;
132             return 1;
133         }
134
135         FileNodeIterator it = n.begin(), it_end = n.end(); // Go
136     through the node
137         for (; it != it_end; ++it)
138             cout << (*string)*it << endl;
139
140
141         n = fs["Mapping"]; // Read mappings from a sequence
142         cout << "Two " << (int)(n["Two"]) << ";" ;
143         cout << "One " << (int)(n["One"]) << endl << endl;
144
145
146         MyData m;
147         Mat R, T;
148
149         fs["R"] >> R; // Read cv::Mat
150         fs["T"] >> T;
151         fs["MyData"] >> m; // Read your own structure_
152
153         cout << endl
154             << "R = " << R << endl;
155         cout << "T = " << T << endl << endl;
156         cout << "MyData = " << endl << m << endl << endl;
157
158         //Show default behavior for non existing nodes
159         cout << "Attempt to read NonExisting (should initialize the
160     data structure with its default).";
161         fs["NonExisting"] >> m;
162         cout << endl << "NonExisting = " << endl << m << endl;
163     }
164
165         cout << endl
166             << "Tip: Open up " << filename << " with a text editor to
167     see the serialized data." << endl;
168
169         return 0;
}

```

۱۴۲ توضیح

در این بخش فقط در مورد فایل‌های XML و YAML صحبت می‌کنیم. فایل ورودی/خروجی شما ممکن است فقط یکی از این دو فرمت را داشته باشد.

به طور کلی دو نوع داده ساختار قابل سریال‌سازی^{۳۲} وجود دارد:

- نگاشت‌ها^{۳۳} (مثل map در STL)
- دنبالهٔ عناصر^{۳۴} (مثل vector در STL)

تفاوت این دو در این است که در نگاشت‌ها هر عنصر یک کلید یکتا دارد و می‌توان در صورت وجود مستقیماً به آن دسترسی پیدا کرد، ولی در دنباله برای پیدا کردن یک عنصر باید کل دنباله را جستجو کرد.

در ادامه با نحوه ذخیره سازی و بازیابی هر دو نوع داده ساختار بالا را بررسی می‌کنیم.

۱. بازبسته کردن فایل: قبل از اینکه در یک فایل بنویسید، لازم است آن را باز کرده و سپس در انتهای آن را به بنید. در اسی وی برای کار با فایل‌های XML و YAML از کلاس FileStorage استفاده می‌شود. برای باز کردن یک فایل با استفاده از این کلاس می‌توان از سازنده آن یا تابع open آن استفاده کرد:

```
string filename = "I.xml";
FileStorage fs(filename, FileStorage::READ);
// OR ...
FileStorage fs;
fs.open(filename, FileStorage::READ);
```

آرگومان دوم یک ثابت است که مشخص کننده عملیاتی است که می‌خواهید روی آن فایل انجام دهید: WRITE (نوشتن)، READ (خواندن) یا APPEND (اضافه کردن). همچنین پسوندی که در اسم فایل مشخص شده است، تعیین کننده نوع فایل خروجی است. جالب اینکه با اضافه کردن .gz به انتهای اسم فایل، آن فایل فشرده خواهد شد. وقتی یک شیء FileStorage از بین برود، فایل مربوط به آن به صورت خودکار بسته می‌شود. البته می‌توان این کار را به صورت دستی با تابع release انجام داد:

```
fs.release();
```

۲. ورودی/خروجی عددی و متغیر: کلاس FileStorage مشابه کتابخانه STL برای خروجی از عملگر <> استفاده می‌کند. برای چاپ هر ساختار داده‌ای لازم است که ابتدا نام آن مشخص شود. این کار به سادگی با چاپ کردن اسم آن انجام می‌شود. برای داده‌های اولیه می‌توان به روش زیر، هم اسم و هم مقدار آن را چاپ کرد:

```
fs << "iterationNr" << 100;
```

برای خواندن مقدار iterationNr باید با استفاده از عملگر [] نام گره‌ای که قصد خواندن آن را داریم (که در اینجا است)، مشخص کنیم. سپس می‌توان به یکی از دو روش زیر به مقدار آن دسترسی پیدا کرد:

```
int itNr;
```

```
fs["iterationNr"] >> itNr;
// OR ...
itNr = (int) fs["iterationNr"];
```

۳. ورودی/خروجی ساختار داده‌های اُسی وی: این کار دقیقاً مشابه نوشتمن نوع‌های اولیه است:

```
Mat R = Mat<uchar>::eye(3, 3),
T = Mat<double>::zeros(3, 1);

fs << "R" << R; // Write cv::Mat
fs << "T" << T;

fs["R"] >> R; // Read cv::Mat
fs["T"] >> T;
```

۴. ورودی/خروجی آرایه‌های ساده و انجمانی: همان‌طور که قبلاً گفته شد، می‌توان نگاشتها و دنباله‌ها (آرایه‌ها و بردارها) را با استفاده از کلاس `FileStorage` چاپ کرد. روند چاپ اینگونه داده ساختارها به این صورت است که ابتدا نام متغیر را می‌آوریم و سپس مشخص می‌کنیم که خروجی یک دنباله یا یک نگاشت است.
برای دنباله‌ها، قبل از آیتم اول کاراکتر [و بعد از آیتم آخر نیز کاراکتر] را قرار می‌دهیم:

```
fs << "strings" << "[";
fs << "image1.jpg" << "Awesomeness" << "baboon.jpg";
fs << "]"; // close sequence
```

در مورد نگاشتها هم همین کار را می‌کنیم، با این تفاوت که از کاراکترهای { و } استفاده می‌کنیم:

```
fs << "Mapping"; // text - mapping
fs << "{" << "One" << 1;
fs << "Two" << 2 << "}";
```

برای خواندن هم از ساختار داده‌های `FileNode` و `FileNodeIterator` استفاده می‌کنیم. عملگر [] مربوط به کلاس `FileNode` یک شیء `FileNodeIterator` را برمی‌گرداند. اگر نود بازگشته یک دنباله بود، از `FileNodeStorage` برای خواندن آیتم‌ها استفاده می‌کنیم:

```
FileNode n = fs["strings"]; // Read string sequence - Get node
if (n.type() != FileNode::SEQ)
{
    cerr << "strings is not a sequence! FAIL" << endl;
    return 1;
}

FileNodeIterator it = n.begin(), it_end = n.end(); // Go through the node
for (; it != it_end; ++it)
    cout << (*string)*it << endl;
```

در مورد نگاشتها می‌توانیم از همان عملگر [] برای دسترسی به آیتم‌ها استفاده کنیم:

```
n = fs["Mapping"]; // Read mappings from a sequence
cout << "Two " << (int)(n["Two"]) << ";" ;
cout << "One " << (int)(n["One"]) << endl << endl;
```

۵. ورودی/خروجی ساختمان داده‌های ساختگی: فرض کنید یک ساختار داده به شکل زیر داریم:

```
class MyData
{
public:
    MyData() : A(0), X(0), id() {}
public: // Data Members
    int A;
    double X;
    string id;
};
```

می‌توان این کلاس را به وسیله رابط ورودی/خروجی YAML/XML سریال کرد (درست مثل ساختار داده‌های اُسی وی). برای این کار باید یک تابع `read` و یک تابع `write`، داخل و بیرون کلاس مورد نظر اضافه شود. به عنوان مثال قسمت داخلی کلاس بالا را به شکل زیر تعریف می‌کنیم:

```
//Write serialization for this class
void write(FileStorage& fs) const
{
    fs << "{" << "A" << A << "X" << X << "id" << id << "}";
}
//Read serialization for this class
void read(const FileNode& node)
{
    A = (int)node["A"];
    X = (double)node["X"];
    id = (string)node["id"];
}
```

سپس قسمت خارجی را به صورت زیر تعریف می‌کنیم:

```
//These write and read functions must be defined for the serialization in
FileStorage to work
static void write(FileStorage& fs, const std::string&, const MyData& x)
{
    x.write(fs);
}
static void read(const FileNode& node, MyData& x, const MyData&
default_value = MyData()){
    if(node.empty())
        x = default_value;
    else
        x.read(node);
}
```

در قسمت `read` مشخص کردہ‌ایم که اگر نود درخواستی وجود نداشته باشد از یک نود پیش فرض استفاده شود (یک شی خام از کلاس `MyData`). البته بهتر است که در برنامه‌های واقعی یک خطای پرتاب کنیم.

بعد از اضافه کردن این چهار تابع از عملگر `<<` برای خواندن و از عملگر `>>` برای نوشتمن استفاده می‌کنیم:

```
MyData m(1);

fs << "MyData" << m; // Write your own data structures
fs["MyData"] >> m; // Read your own structure_
```

۱،۴،۳ خروجی

برنامه را به صورت زیر اجرا می‌کنیم:

FileRW.exe "1.xml"

محتوای فایل ۱.xml به صورت زیر است:

```
<?xml version="1.0"?>
<opencv_storage>
<iterationNr>100</iterationNr>
<strings>
    image1.jpg Awesomeness baboon.jpg</strings>
<Mapping>
    <One>1</One>
    <Two>2</Two></Mapping>
<R type_id="opencv-matrix">
    <rows>3</rows>
    <cols>3</cols>
    <dt>u</dt>
    <data>
        1 0 0 0 1 0 0 0 1</data></R>
<T type_id="opencv-matrix">
    <rows>3</rows>
    <cols>1</cols>
    <dt>d</dt>
    <data>
        0. 0. 0.</data></T>
<MyData>
    <A>97</A>
    <X>3.1415926535897931e+000</X>
    <id>mydata1234</id></MyData>
</opencv_storage>
```

برنامه را به صورت زیر اجرا می‌کنیم:

FileRW.exe "2.yml"

محتوای فایل ۲.yml به صورت زیر است:

```
%YAML:1.0
iterationNr: 100
strings:
    - "image1.jpg"
    - Awesomeness
    - "baboon.jpg"
Mapping:
    One: 1
    Two: 2
R: !!opencv-matrix
    rows: 3
    cols: 3
    dt: u
    data: [ 1, 0, 0, 0, 1, 0, 0, 0, 1 ]
T: !!opencv-matrix
    rows: 3
    cols: 1
    dt: d
    data: [ 0., 0., 0. ]
MyData:
    A: 97
```

x: 3.1415926535897931e+000

id: mydata1234

۲ فصل دوم - حوزه مکان

۲,۱ شکل‌های هندسی ساده

در این بخش می‌خواهیم طریقه ترسیم شکل‌های هندسی ساده را آموزش دهیم. این شکل‌ها عبارتند از خط، دایره، بیضی، مستطیل، چند ضلعی.

۲,۱,۱ کد

```

1 #include <opencv2/core/core.hpp>
2 #include <opencv2/highgui/highgui.hpp>
3 #include <iostream>
4 #define w 400
5
6 using namespace cv;
7
8 /// Function headers
9 void MyEllipse( Mat img, double angle );
10 void MyFilledCircle( Mat img, Point center );
11 void MyPolygon( Mat img );
12 void MyLine( Mat img, Point start, Point end );
13
14 /**
15 * @function main
16 * @brief Main function
17 */
18 int main( void ){
19
20     /// Windows names
21     char atom_window[] = "Drawing 1: Atom";
22     char rook_window[] = "Drawing 2: Rook";
23
24     /// Create black empty images
25     Mat atom_image = Mat::zeros( w, w, CV_8UC3 );
26     Mat rook_image = Mat::zeros( w, w, CV_8UC3 );
27
28     /// 1. Draw a simple atom:
29     /// -----
30
31     /// 1.a. Creating ellipses
32     MyEllipse( atom_image, 90 );
33     MyEllipse( atom_image, 0 );
34     MyEllipse( atom_image, 45 );
35     MyEllipse( atom_image, -45 );
36
37     /// 1.b. Creating circles
38     MyFilledCircle( atom_image, Point( w/2, w/2 ) );
39
40     /// 2. Draw a rook
41     /// -----
42
43     /// 2.a. Create a convex polygon

```

```

44     MyPolygon( rook_image );
45
46     /// 2.b. Creating rectangles
47     rectangle(rook_image, Point( 0, 7*w/8 ), Point( w, w ), Scalar(
48     0, 255, 255 ), -1, 8 );
49
50     /// 2.c. Create a few lines
51     MyLine( rook_image, Point( 0, 15*w/16 ), Point( w, 15*w/16 ) );
52     MyLine( rook_image, Point( w/4, 7*w/8 ), Point( w/4, w ) );
53     MyLine( rook_image, Point( w/2, 7*w/8 ), Point( w/2, w ) );
54     MyLine( rook_image, Point( 3*w/4, 7*w/8 ), Point( 3*w/4, w ) );
55
56     /// 3. Display your stuff!
57     imshow( atom_window, atom_image );
58     moveWindow( atom_window, 0, 200 );
59     imshow( rook_window, rook_image );
60     moveWindow( rook_window, w, 200 );
61
62     waitKey( 0 );
63     return(0);
64 }
65
66 /// Function Declaration
67
68 /**
69  * @function MyEllipse
70  * @brief Draw a fixed-size ellipse with different angles
71  */
72 void MyEllipse( Mat img, double angle )
73 {
74     int thickness = 2;
75     int lineType = 8;
76
77     ellipse( img, Point( w/2, w/2 ), Size( w/4, w/16 ), angle, 0,
78     360, Scalar( 255, 0, 0 ), thickness, lineType );
79 }
80
81 /**
82  * @function MyFilledCircle
83  * @brief Draw a fixed-size filled circle
84  */
85 void MyFilledCircle( Mat img, Point center )
86 {
87     int thickness = -1;
88     int lineType = 8;
89
90     circle(img, center, w/32, Scalar( 0, 0, 255 ), thickness,
91     lineType);
92 }
93
94 /**
95  * @function MyPolygon
96  * @function Draw a simple concave polygon (rook)
97  */
98 void MyPolygon( Mat img )
99 {
100     int lineType = 8;
101
102     /** Create some points */
103     Point rook_points[1][20];
104     rook_points[0][0] = Point(      w/4,      7*w/8 );

```

```

105     rook_points[0][1] = Point( 3*w/4, 7*w/8 );
106     rook_points[0][2] = Point( 3*w/4, 13*w/16 );
107     rook_points[0][3] = Point( 11*w/16, 13*w/16 );
108     rook_points[0][4] = Point( 19*w/32, 3*w/8 );
109     rook_points[0][5] = Point( 3*w/4, 3*w/8 );
110     rook_points[0][6] = Point( 3*w/4, w/8 );
111     rook_points[0][7] = Point( 26*w/40, w/8 );
112     rook_points[0][8] = Point( 26*w/40, w/4 );
113     rook_points[0][9] = Point( 22*w/40, w/4 );
114     rook_points[0][10] = Point( 22*w/40, w/8 );
115     rook_points[0][11] = Point( 18*w/40, w/8 );
116     rook_points[0][12] = Point( 18*w/40, w/4 );
117     rook_points[0][13] = Point( 14*w/40, w/4 );
118     rook_points[0][14] = Point( 14*w/40, w/8 );
119     rook_points[0][15] = Point( w/4, w/8 );
120     rook_points[0][16] = Point( w/4, 3*w/8 );
121     rook_points[0][17] = Point( 13*w/32, 3*w/8 );
122     rook_points[0][18] = Point( 5*w/16, 13*w/16 );
123     rook_points[0][19] = Point( w/4, 13*w/16 );
124
125     const Point* ppt[1] = { rook_points[0] };
126     int npt[] = { 20 };
127
128     fillPoly(img, ppt, npt, 1, Scalar( 255, 255, 255 ), lineType);
129 }
130
131 /**
132 * @function MyLine
133 * @brief Draw a simple line
134 */
135 void MyLine( Mat img, Point start, Point end )
136 {
137     int thickness = 2;
138     int lineType = 8;
139     line(img, start, end, Scalar( 0, 0, 0 ), thickness, lineType);
140 }

```

۲,۱,۲ توضیح

از آنجایی که می‌خواهیم دو شکل رسم کنیم (یک اتم و یک قلعه)، باید دو ماتریس به عنوان صفحهٔ ترسیم و دو پنجره برای نمایش آنها درست کنیم (خطوط ۲۵ و ۲۶).

برای کشیدن هر شکل یک تابع جداگانه درست کرده‌ایم. به عنوان مثال برای کشیدن اتم از توابع `MyFilledCircle` و `MyEllipse` استفاده کرده‌ایم (خطوط ۲۸ تا ۳۹).

برای کشیدن قلعه از توابع `MyLine`، `MyPolygon` و یک `rectangle` استفاده کرده‌ایم (خطوط ۴۰ تا ۵۴).

بهتر است ببینیم درون این توابع چه می‌گذرد:

• تابع `MyLine` (خطوط ۱۳۵ تا ۱۴۰)

همان طور که می‌بینید این تابع فقط به روش زیر چندین بار از تابع `line` (که در مژول `Core` موجود است) استفاده می‌کند:

- یک خط از نقطه `start` به نقطه `end` می‌کشد.
- خط بالا در تصویر `img` رسم می‌کند.

رنگ خط با یک اسکالر به فرم $\text{Scalar}(0,0,0)$ است، به تابع `line` فرستاده شده.

- میزان نازکی خط با متغیر `thickness` مشخص شده است.
- این خط از نوع متصل است و این مورد با متغیر `lineType` که مقدار ۲ دارد مشخص شده است.

• تابع `MyEllipse` (خطوط ۷۲ تا ۷۹)

این تابع به شکل زیر یک بیضی می‌کشد:

- بیضی در عکس `img` ترسیم می‌شود.
- مرکز بیضی نقطه $(w/2, w/2)$ است و در یک مستطیل به اندازه $(w/4, w/16)$ محدود شده.
- بیضی به اندازه `angle` درجه چرخانده می‌شود.
- بیضی به صورت یک قطاع از زاویه 0 تا 360 است.
- رنگ بیضی با $\text{Scalar}(255,0,0)$ که در RGB یعنی آبی، مشخص شده است.
- نازکی و نوع خط بیضی با متغیرهای `thickness` و `lineType` مشخص شده‌اند.

• تابع `MyFilledCircle` (خطوط ۸۵ تا ۹۲)

آرگومان‌های تابع `circle` به شرح زیر است:

- `img` عکسی است که دایره در آن کشیده می‌شود.
- مرکز دایره با نقطه `center` مشخص شده است.
- شعاع دایره $W/32$ است.
- رنگ دایره با $\text{Scalar}(0,0,255)$ که در RGB به معنی قرمز است، مشخص شده است.
- به خاطر اینکه مقدار `thickness` عدد -1 قرار داده شده، دایره توپر کشیده می‌شود.

• تابع `MyPolygon` (خطوط ۹۸ تا ۱۲۹)

برای کشیدن یک چند ضلعی توپر، از تابع `fillPoly` که در ماژول `Core` موجود است استفاده می‌کنیم. قابل ذکر است که:

- چندضلعی روی عکس `img` کشیده می‌شود.
- رأس‌های چند ضلعی همان مجموعه نقاطی هستند که در `ppt` قرار داده شده‌اند.
- تعداد رأس‌هایی که باید رسم شوند در متغیر `npt` قرار داده شده است.
- تعداد چند ضلعی‌هایی که باید رسم شوند، 1 عدد است.
- رنگ چند ضلعی با $\text{Scalar}(255,255,255)$ که در RGB به معنای سفید است، مشخص شده است.

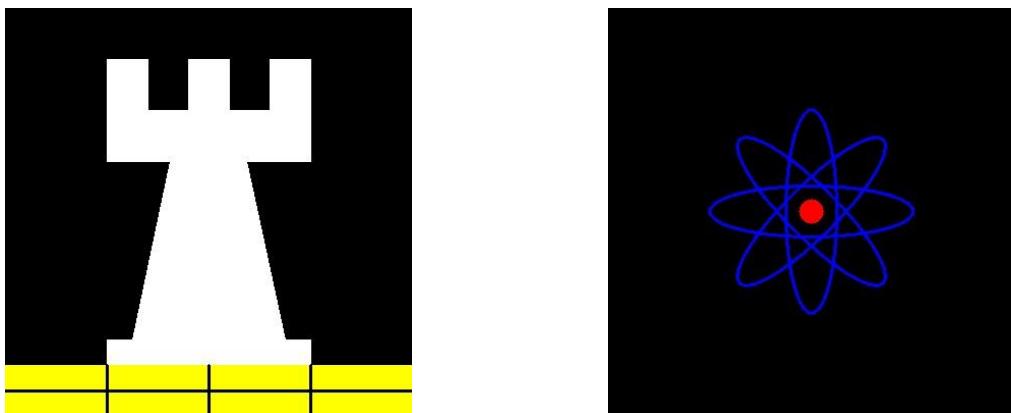
• تابع `rectangle` (خط ۴۷)

در آخر تابع `rectangle` را معرفی می‌کنیم. در این تابع:

- مستطیل روی عکس `rook_image` کشیده می‌شود.
- دو رأس مخالف مستطیل با $\text{Point}(0,7*w/8.0)$ و $\text{Point}(w,w)$ مشخص شده‌اند.
- رنگ مستطیل با $\text{Scalar}(0,255,255)$ که نشان دهنده رنگ زرد در سیستم RGB است، مشخص شده است.
- به خاطر اینکه مقدار `thickness` عدد -1 قرار داده شده است، مستطیل توپر خواهد بود.

۲,۱,۳ خروجی

خروجی برنامه به شکل زیر است:



۲،۲ ترکیب خطی دو تصویر

دو تصویر را می‌توان با استفاده از یک ترکیب خطی از مقادیر پیکسل‌های آنان، جمع کرد. فرم کلی یک ترکیب خطی به صورت زیر است:

$$g(x) = \alpha f_0(x) + \beta f_1(x) + \gamma : \alpha, \beta, \gamma \in \mathbb{R}$$

قبل‌با عملگرهای ریاضی مجازی که می‌توانستیم روی کلاس Mat استفاده کنیم آشنا شدیم. می‌دانیم که می‌توان به راحتی دو عکس (با ابعاد یکسان) را با هم جمع کرد و عکس دیگری با همان ابعاد به وجود آورد. اما در این بخش می‌خواهیم شما را با تابع addWeighted آشنا کنیم. این تابع دقیقاً مشابه معادله بالا عمل می‌کند؛ یعنی با دریافت دو ماتریس دو عدد α و β و سه عدد α ، β و γ یک ترکیب خطی از دو ماتریس ورودی درست می‌کند و به عنوان خروجی بر می‌گرداند.

۲،۲،۱ کد

```

1 #include <opencv2/core/core.hpp>
2 #include <opencv2/highgui/highgui.hpp>
3 #include <iostream>
4
5 int main( int argc, char** argv )
6 {
7     double alpha = 0.5; double beta; double input;
8     cv::Mat src1, src2, dst;
9
10    // Ask the user enter alpha
11    std::cout << " Simple Linear Blender " << std::endl;
12    std::cout << "-----" << std::endl;
13    std::cout << "* Enter alpha [0-1]: ";
14    std::cin >> input;
15
16    // We use the alpha provided by the user iff it is between 0 and
17    if( alpha >= 0 && alpha <= 1 )
18        alpha = input;
19
20    // Read image ( same size, same type )
21    src1 = cv::imread("PICTURES/1.jpg");
22    src2 = cv::imread("PICTURES/2.jpg");
23    if( !src1.data )
24    {
25        std::cout << "Error loading src1" << std::endl;
26        return -1;
27    }
28 }
```

```

29     if( !src2.data )
30     {
31         std::cout << "Error loading src2" << std::endl;
32         return -1;
33     }
34
35 // Create Windows
36 beta = ( 1.0 - alpha );
37 cv::addWeighted( src1, alpha, src2, beta, 0.0, dst );
38
39 cv::imshow( "Linear Blend", dst );
40 cv::waitKey(0);
41
42     return 0;
43 }
```

۲,۲,۲ توضیح

در این برنامه ابتدا از کاربر می‌خواهیم که یک مقدار بین 0 و 1 برای α انتخاب کند. اگر کاربر عدد درستی را وارد کرده باشد، متغیر α را برابر با آن عدد قرار می‌دهیم (خط ۱۹)؛ در غیر این صورت از همان مقدار پیشفرض که 0.5 است، استفاده می‌شود.

در ادامه دو تصویر را از حافظه می‌خوانیم و در $src1$ و $src2$ قرار می‌دهیم. همچنین مطمئن می‌شویم که این دو تصویر به درستی بارگذاری شده‌اند.

در خط ۳۶ مقدار β را برابر $\alpha - 1$ قرار می‌دهیم. این یعنی می‌خواهیم تصویر نهایی به نسبت $src1$ از تصویر $src2$ و به نسبت $\alpha - 1$ از تصویر $src2$ تشکیل شده باشد.

برای اعمال ترکیب خطی در خط ۳۷ تابع `addWeighted` را صدا می‌زنیم. توجه کنید که در اینجا مقدار γ را صفر قرار داده‌ایم.

۲,۲,۳ خروجی

برنامه را به صورت زیر اجرا می‌کنیم:

```
LinearBlend.exe
> 0.3
```

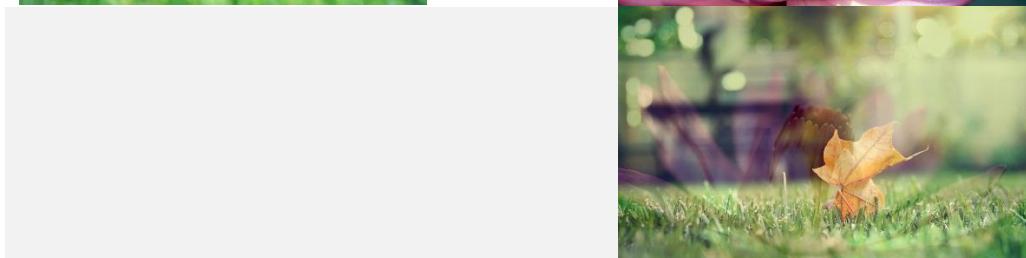
خروجی به صورت زیر است:



تصویر
ورودی دوم



تصویر ورودی
اول



تصویر
خروجی

۲,۳ فیلتر خطی

در این بخش نحوه اعمال فیلتر خطی روی تصاویر را بررسی می‌کنیم. برای اعمال فیلتر خطی باید یک کرنل داشته باشیم؛ سپس آن کرنل را در تصویر کانولوشن می‌کنیم.

به طور کلی، به عملی که بین یک کرنل و همه قسمت‌های یک تصویر انجام می‌شود کانولوشن می‌گویند. همچنین کرنل آرایه‌ای از ضرایب ثابت است و یک لنگر دارد که معمولاً در مرکز آرایه قرار دارد. لنگر همان درایه‌ای از کرنل است که روی پیکسل مورد بررسی قرار می‌گیرد.

1	-2	1
2		2
1	-2	1

یک کرنل نمونه که لنگر در مرکز آن قرار دارد

برای هر پیکسل (x, y) در تصویر I ، اگر K ماتریس کرنل با ابعاد $N * M$ و (a, b) مختصات لنگر در کرنل باشد، آنگاه $H(x, y)$ ، یعنی مقدار پیکسل (x, y) در تصویر فیلتر شده، به صورت زیر حساب می‌شود:

$$H(x, y) = \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} I(x + i - a, y + j - b) K(i, j)$$

برای اعمال فیلتر خطی لازم نیست خودمان فرمول بالا را پیاده سازی کنیم چون تابع filter2D در اسی وی دقیقاً همین کار را انجام می‌دهد.

۲,۳,۱ کد

```

1 #include "opencv2/imgproc/imgproc.hpp"
2 #include "opencv2/highgui/highgui.hpp"
3
4 using namespace cv;
5
6 /** @function main */
7 int main ( int argc, char** argv )
8 {
9     /// Declare variables
10    Mat src, dst;
11
12    Mat kernel;
13    Point anchor;
14    double delta;
15    int ddepth;
16    int kernel_size;
17    char* window_name = "filter2D Demo";
18
19    int c;
20
21    /// Load an image
22    src = imread( argv[1] );
23
24    if( !src.data )
25    { return -1; }
26

```

```

27     /// Create window
28     namedWindow( window_name, CV_WINDOW_AUTOSIZE );
29
30     /// Initialize arguments for the filter
31     anchor = Point( -1, -1 );
32     delta = 0;
33     ddepth = -1;
34
35     /// Loop - Will filter the image with different kernel sizes each
36     0.5 seconds
37     int ind = 0;
38     while( true )
39     {
40         c = waitKey(500);
41         /// Press 'ESC' to exit the program
42         if( (char)c == 27 )
43             { break; }
44
45         /// Update kernel size for a normalized box filter
46         kernel_size = 3 + 2*( ind%5 );
47         kernel = Mat::ones( kernel_size, kernel_size, CV_32F )/
48         (float)(kernel_size*kernel_size);
49
50         /// Apply filter
51         filter2D(src, dst, ddepth , kernel, anchor, delta,
52 BORDER_DEFAULT );
53         imshow( window_name, dst );
54         ind++;
55     }
56
57     return 0;
58 }
```

۲،۳،۲ توضیح

خطوط ۴۰ تا ۵۴ در یک حلقه بینهایت قرار دارند. در این حلقه اندازه کرنل آپدیت و فیلتر خطی به عکس اعمال می‌شود. جزئیات اعمالی که در این حلقه صورت می‌گیرد به صورت زیر است:

ابتدا در خطوط ۴۶ و ۴۷ کرنل تعریف می‌شود. خط ۴۶ برای آپدیت کردن مقدار kernel_size است؛ این مقدار عددی بین ۳ و ۱۱ است. در خط ۴۷ برای ساخت کرنل ابتدا یک ماتریس مربعی با ابعاد kernel_size * kernel_size ساخته می‌شود و سپس آن ماتریس را بر تعداد درایه‌هایش (یعنی kernel_size * kernel_size) تقسیم می‌کند.

بعد از درست کردن کرنل، در خط ۵۱ برای اعمال کرنل ساخته شده به تصویر از تابع filter2D استفاده می‌شود. این تابع ۷ آرگومان به شرح زیر دارد:

src: تصویر ورودی	-
dst: تصویر خروجی (تصویر فیلتر شده)	-
ddepth: عمق تصویر خروجی است. یک مقدار منفی (مثل -۱) بیان می‌کند که عمق dst باید مشابه عمق src باشد.	-
kernel: کرنل مورد استفاده است.	-
anchor: موقعیت لنگر کرنل است. با قرار دادن Point(-1,-1) مقدار پیش فرض (یعنی مختصات مرکز کرنل) استفاده می‌شود.	-
delta: مقداری که نتیجه کانولوشن هر پیکسل اضافه می‌شود. به صورت پیش فرض صفر است.	-

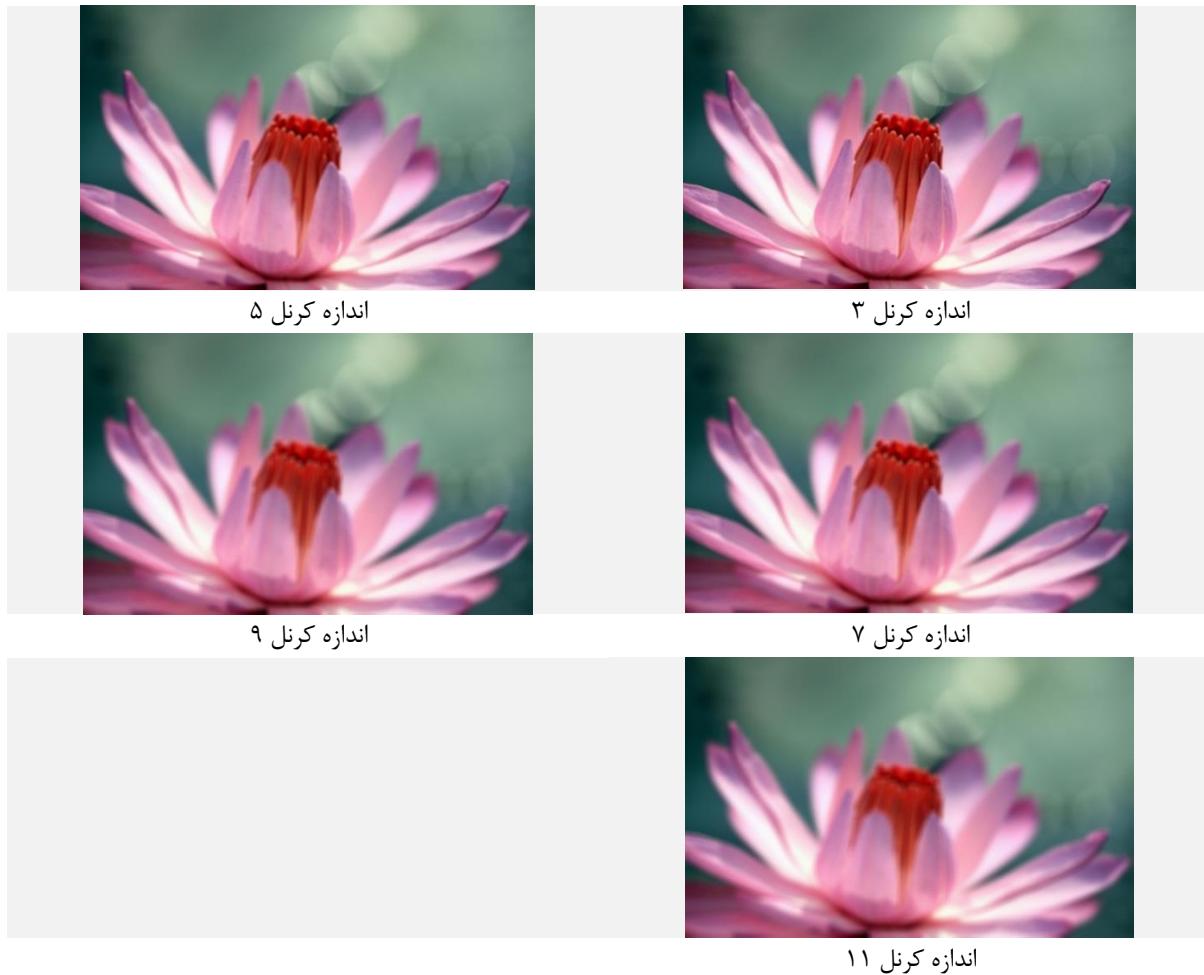
- **BORDER_DEFAULT**: نحوه کانوال کردن پیکسل‌های کناری تصویر را مشخص می‌کند.

۲,۳,۳ خروجی

برنامه را به صورت زیر اجرا می‌کنیم:

2DFilter.exe "PICTURES /2.jpg"

هر ۵۰۰ میلی ثانیه اندازه کرنل تغییر می‌کند. نمونه‌ای از اجرای برنامه را در زیر می‌بینید:



۲,۴ تبدیل آفاین

به هر تبدیلی که بتوان آن را به فرم یک ضرب ماتریسی و به دنبال آن یک جمع برداری بیان کرد، تبدیل آفاین می‌گویند. بنابراین می‌توانیم اعمال زیر را به عنوان تبدیل آفاین در نظر بگیریم:

۱. چرخش (تبدیل خطی)
۲. انتقال (جمع برداری)
۳. عملیات مقیاسی (تبدیل خطی)

یک تبدیل آفاین نشان دهنده یک رابطه بین دو عکس است و معمول‌ترین راه برای نشان دادن یک تبدیل آفاین، استفاده از یک ماتریس ۲ در ۳ است:

$$A = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix}_{2 \times 2} \quad B = \begin{bmatrix} b_{00} \\ b_{10} \end{bmatrix}_{2 \times 1}$$

$$M = [A \quad B] = \begin{bmatrix} a_{00} & a_{01} & b_{00} \\ a_{10} & a_{11} & b_{10} \end{bmatrix}_{2 \times 3}$$

با توجه به اینکه می‌خواهیم یک بردار دو بعدی به صورت $\begin{bmatrix} x \\ y \end{bmatrix} = X$ را با استفاده از A و B تبدیل کنیم، می‌توانیم این کار را به صورت معادل زیر هم انجام دهیم:

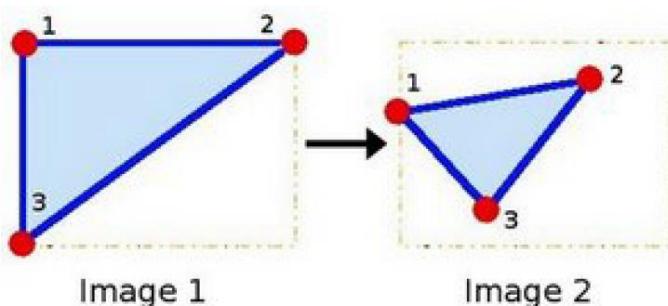
$$T = A \cdot \begin{bmatrix} x \\ y \end{bmatrix} + B \text{ or } T = M \cdot [x, y, 1]^t$$

$$T = \begin{bmatrix} a_{00}x + a_{01}y + b_{00} \\ a_{10}x + a_{11}y + b_{10} \end{bmatrix}$$

گفتیم که هر تبدیل آفاین یک رابطه بین دو عکس است. اطلاعات مربوط به این رابطه می‌تواند به دو طریق وجود داشته باشد:

۱. X و T را در اختیار داریم و می‌دانیم که این دو با هم رابطه دارند. بنابراین هدف ما پیدا کردن M است.
۲. M و X را در اختیار داریم و برای به دست آوردن T = M \cdot X استفاده می‌کنیم. ممکن است M را به طور صریح در اختیارمان قرار دهند (یعنی همان ماتریس ۲ در ۳ را داده باشند)، یا اینکه آن را به صورت یک رابطه هندسی بین نقاط داشته باشیم.

از آنجایی که M دو تصویر را به هم ربط می‌دهد، می‌توان آن را به صورت ساده‌ترین حالت یعنی حالتی که سه نقطه از هر کدام از دو تصویر را به هم ربط می‌دهد، در نظر گرفت. به شکل زیر نگاه کنید:



نقاط ۱، ۲ و ۳ در تصویر ۱ (که یک مثلث تشکیل داده‌اند)، به تصویر ۲ نگاشت شده‌اند و هنوز هم به شکل یک مثلث هستند ولی شکل آن مثلث تغییر کرده است. اگر ما تبدیل آفاین این سه نقطه را پیدا کنیم، می‌توانیم آن را به همه پیکسل‌های یک تصویر اعمال کنیم.

۲۴۱ کد

```

1 #include "opencv2/highgui/highgui.hpp"
2 #include "opencv2/imgproc/imgproc.hpp"
3 #include <iostream>
4 #include <stdio.h>
5
6 using namespace cv;
7 using namespace std;
8
9 /// Global variables
10 char* source_window = "Source image";
11 char* warp_window = "Warp";
12 char* warp_rotate_window = "Warp + Rotate";

```

```

13
14  /** @function main */
15  int main( int argc, char** argv )
16  {
17      Point2f srcTri[3];
18      Point2f dstTri[3];
19
20      Mat rot_mat( 2, 3, CV_32FC1 );
21      Mat warp_mat( 2, 3, CV_32FC1 );
22      Mat src, warp_dst, warp_rotate_dst;
23
24      /// Load the image
25      src = imread( argv[1], 1 );
26
27      /// Set the dst image the same type and size as src
28      warp_dst = Mat::zeros( src.rows, src.cols, src.type() );
29
30      /// Set your 3 points to calculate the Affine Transform
31      srcTri[0] = Point2f( 0,0 );
32      srcTri[1] = Point2f( src.cols - 1, 0 );
33      srcTri[2] = Point2f( 0, src.rows - 1 );
34
35      dstTri[0] = Point2f( src.cols*0.0, src.rows*0.33 );
36      dstTri[1] = Point2f( src.cols*0.85, src.rows*0.25 );
37      dstTri[2] = Point2f( src.cols*0.15, src.rows*0.7 );
38
39      /// Get the Affine Transform
40      warp_mat = getAffineTransform( srcTri, dstTri );
41
42      /// Apply the Affine Transform just found to the src image
43      warpAffine( src, warp_dst, warp_mat, warp_dst.size() );
44
45      /** Rotating the image after Warp */
46
47      /// Compute a rotation matrix with respect to the center of the
48      image
49      Point center = Point( warp_dst.cols/2, warp_dst.rows/2 );
50      double angle = -50.0;
51      double scale = 0.6;
52
53      /// Get the rotation matrix with the specifications above
54      rot_mat = getRotationMatrix2D( center, angle, scale );
55
56      /// Rotate the warped image
57      warpAffine( warp_dst, warp_rotate_dst, rot_mat, warp_dst.size()
58 );
59
60      /// Show what you got
61      namedWindow( source_window, CV_WINDOW_AUTOSIZE );
62      imshow( source_window, src );
63
64      namedWindow( warp_window, CV_WINDOW_AUTOSIZE );
65      imshow( warp_window, warp_dst );
66
67      namedWindow( warp_rotate_window, CV_WINDOW_AUTOSIZE );
68      imshow( warp_rotate_window, warp_rotate_dst );
69
70      /// Wait until user exits the program
71      waitKey(0);
72
73      return 0;

```

۲,۴,۲ توضیح

همانطور که قبلاً توضیح داده شد، برای به دست آوردن یک تبدیل آفاین به رابطه بین دو مجموعه سه نقطه‌ای نیاز داریم. در خطوط ۳۰ تا ۳۷ این دو مجموعه تعریف شده‌اند. موقعیت این نقاط تقریباً شبیه همانی است که در قسمت قبل دیدید.

اکنون با داشتن این نقاط در خط ۴۰ برای به دست آوردن تبدیل آفاین از تابع `getAffineTransform` استفاده می‌کنیم. خروجی این تابع به صورت یک ماتریس ۲ در ۳ است.

در خط ۴۳ با استفاده از تابع `warpAffine` تبدیل آفاین را به تصویر `src` اعمال می‌کنیم. این تابع چهار آرگومان دارد:

- .۱ `src`: تصویر ورودی
- .۲ `warp_dst`: تصویر خروجی
- .۳ `warp_mat`: تبدیل آفاین
- .۴ `warp_dst.size()`: اندازه تصویر خروجی است.

در خطوط ۴۹ تا ۵۷ می‌خواهیم تصویر تبدیل شده `warp_dst` را بچرخانیم. برای چرخاندن یک تصویر به مختصات مرکزی که تصویر باید نسبت به آن بچرخد و میزان زاویه‌ای که تصویر باید بچرخد نیاز داریم، این پارامترها در خطوط ۴۹ تا ۵۱ تعریف شده‌اند.

در خط ۵۴ برای تولید ماتریس چرخشی از تابع `getRotationMatrix2D` استفاده می‌کنیم. این تابع یک ماتریس ۲ در ۳ بر می‌گردد.

سپس در خط ۵۷، ماتریس چرخشی را به تصویر `warp_mat` اعمال می‌کنیم.

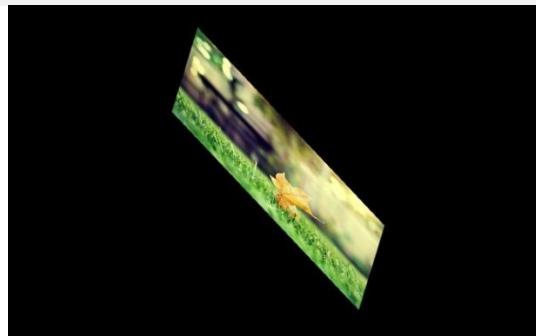
۲,۴,۳ خروجی



تصویر تبدیل شده



تصویر ورودی



تصویر چرخیده شده از روی تصویر تبدیل شده

۲,۵ نگاشت

در این بخش به چگونگی استفاده از تابع `remap` برای پیاده سازی نگاشتهای ساده می پردازیم.

به فرایند تغییر مکان پیکسل های یک تصویر و بردن آنها به مکان های دیگر در یک تصویر جدید، نگاشت می گویند. می توان عملیات نگاشت هر پیکسل (x, y) را به صورت زیر نشان داد:

$$g(x, y) = f(h(x, y))$$

که g تصویر نگاشت یافته، f تصویر منبع و h هم تابع نگاشت است.

به عنوان مثال فرض کنید عکسی به نام `I` داریم و می خواهیم یک نگاشت به صورت زیر روی آن اعمال کنیم:

$$h(x, y) = (I.cols - x, y)$$

به راحتی می توان دید که نتیجه این نگاشت چرخیدن عکس در جهت X است. مثلاً عکس زیر را در نظر بگیرید:



تصویر اصلی



تصویر نگاشت شده

به تغییر موقعیت دایره قرمز نسبت به X توجه کنید (با توجه به اینکه X جهت افقی است).

۲,۶,۱ کد

```

1 #include "opencv2/highgui/highgui.hpp"
2 #include "opencv2/imgproc/imgproc.hpp"
3 #include <iostream>
4 #include <stdio.h>
5
6 using namespace cv;
7
8 /// Global variables
9 Mat src, dst;
10 Mat map_x, map_y;
11 char* remap_window = "Remap demo";
12 int ind = 0;
13
14 /// Function Headers
15 void update_map( void );
16
17 /**
18 * @function main
19 */

```

```

20 int main( int argc, char** argv )
21 {
22     /// Load the image
23     src = imread( argv[1], 1 );
24
25     /// Create dst, map_x and map_y with the same size as src:
26     dst.create( src.size(), src.type() );
27     map_x.create( src.size(), CV_32FC1 );
28     map_y.create( src.size(), CV_32FC1 );
29
30     /// Create window
31     namedWindow( remap_window, CV_WINDOW_AUTOSIZE );
32
33     /// Loop
34     while( true )
35     {
36         /// Each 1 sec. Press ESC to exit the program
37         int c = waitKey( 1000 );
38
39         if( (char)c == 27 )
40             { break; }
41
42         /// Update map_x & map_y. Then apply remap
43         update_map();
44         remap( src, dst, map_x, map_y, CV_INTER_LINEAR,
45 BORDER_CONSTANT, Scalar(0,0, 0) );
46
47         /// Display results
48         imshow( remap_window, dst );
49     }
50     return 0;
51 }
52
53 /**
54 * @function update_map
55 * @brief Fill the map_x and map_y matrices with 4 types of mappings
56 */
57 void update_map( void )
58 {
59     ind = ind%4;
60
61     for( int j = 0; j < src.rows; j++ )
62     { for( int i = 0; i < src.cols; i++ )
63     {
64         switch( ind )
65         {
66             case 0:
67                 if( i > src.cols*0.25 && i < src.cols*0.75 && j >
68 src.rows*0.25 && j < src.rows*0.75 )
69                 {
70                     map_x.at<float>(j,i) = 2*( i - src.cols*0.25 ) +
71 0.5 ;
72                     map_y.at<float>(j,i) = 2*( j - src.rows*0.25 ) +
73 0.5 ;
74                 }
75             else
76                 { map_x.at<float>(j,i) = 0 ;
77                     map_y.at<float>(j,i) = 0 ;
78                 }
79             break;
80         case 1:

```

```

81         map_x.at<float>(j,i) = i ;
82         map_y.at<float>(j,i) = src.rows - j ;
83         break;
84     case 2:
85         map_x.at<float>(j,i) = src.cols - i ;
86         map_y.at<float>(j,i) = j ;
87         break;
88     case 3:
89         map_x.at<float>(j,i) = src.cols - i ;
90         map_y.at<float>(j,i) = src.rows - j ;
91         break;
92 } // end of switch
93 }
94 ind++;
95 }
96 }
```

۲،۵،۲ توضیح

در خط ۴۴ برای انجام نگاشت، تابع remap را صدا می‌زنیم. این تابع پنج آرگومان به شرح زیر دارد:

- .۱ src: تصویر ورودی
- .۲ dst: تصویر خروجی که همان اندازه src است.
- .۳ map_x: تابع نگاشت در جهت x است. معادل قسمت اول تابع $h(i,j)$ است.
- .۴ map_y: مشابه بالایی، فقط در جهت y است. توجه کنید که map_x و map_y هر دو هم اندازه src هستند.
- .۵ CV_INTER_LINEAR: نوع دورن یابی ای که برای پیکسل‌های غیر صحیح به کار می‌رود. پیش فرض آن همین مقدار است.
- .۶ BORDER_CONSTANT: نوع قابی که به اطراف تصویر اضافه می‌شود.

در این برنامه چهار نوع نگاشت به تصویر ورودی اعمال می‌شود. تابع $h(i,j)$ هر کدام از این نگاشتها به صورت زیر است:

i. کاهش اندازه تصویر به نصف و نشان دادن آن در مرکز:

$$h(i,j) = \left(2 * i - \frac{src.cols}{2} + 0.5, 2 * j - \frac{src.rows}{2} + 0.5 \right)$$

برای همه زوج‌های (i,j) به صورتی که:

$$\frac{src.cols}{4} < i < \frac{3 * src.cols}{4} \text{ and } \frac{src.rows}{4} < j < \frac{3 * src.rows}{4}$$

ii. بالا به پایین کردن تصویر: $h(i,j) = (i, src.rows - j)$

iii. چپ به راست کردن تصویر: $h(i,j) = (src.cols - i, j)$

iv. ترکیب ii و iii: $h(i,j) = (src.cols - i, src.rows - j)$

تمام موارد بالا در خطوط ۶۶ تا ۹۲ پیاده سازی شده‌اند. در اینجا map_x نشان دهنده مختص اول $h(i,j)$ و map_y مختص دوم آن است.

۲,۵,۳ خروجی



بالا به پایین کردن تصویر ورودی



تصویر ورودی



بالا به پایین و چپ به راست کردن تصویر ورودی



چپ به راست کردن تصویر ورودی



کاهش اندازه تصویر به نصف و نشان دادن آن در مرکز

۲,۶ تغییر کنتراست و روشنایی تصویر

به طور کلی کنتراست یکی از مشخصات تصویر است که به صورت نسبت روشنایی درخشنان‌ترین رنگ (سفید) به روشنایی تاریک‌ترین رنگ (سیاه) موجود در آن تصویر تعریف می‌شود. در تصاویر هر چه نسبت کنتراست بالاتر باشد مطلوب‌تر است. این مقدار نسبت بیشترین میزان روشنایی رنگ سفید به کمترین میزان روشنایی رنگ سیاه کامل را نشان می‌دهد.

از آنجا که کنتراست و شدت روشنایی هر دو از مشخصه‌های پیکسلی هستند، می‌خواهیم در این بخش به بهانه تغییر کنتراست و روشنایی تصاویر، شما را با تبدیل‌های پیکسلی آشنا کنیم.

به طور کلی اگر $f(i,j)$ مقدار تصویر ورودی در پیکسل (j,i) باشد، آنگاه می‌توان با استفاده از معادله زیر و تعیین مقادیر مناسب برای α و β ، کنتراست و روشنایی تصویر f را تغییر داد:

$$g(i,j) = \alpha \cdot f(i,j) + \beta$$

در اینجا (i, j) مقدار جدید تصویر در پیکسل (i, j) است. با استفاده از α می‌توان میزان کنتراست را تغییر داد و با استفاده از β هم می‌توان میزان روشنایی را تغییر داد. توجه کنید که اگر می‌خواهید فقط میزان روشنایی را تغییر دهید، باید α را یک بگذارید و عدد β را فقط تغییر دهید. اگر هم می‌خواهید فقط میزان کنتراست را تغییر دهید کافی است عدد β را صفر بگذارید و عدد α را هر آنچه می‌خواهید قرار دهید.

۲,۶,۱ کد

```

1 #include <opencv2/core/core.hpp>
2 #include <opencv2/highgui/highgui.hpp>
3 #include <iostream>
4
5 int main( int argc, char** argv )
6 {
7     double alpha; /*< Simple contrast control */
8     int beta; /*< Simple brightness control */
9
10    /// Read image given by user
11    cv::Mat image = cv::imread( argv[1] );
12    cv::Mat new_image = cv::Mat::zeros( image.size(), image.type() );
13
14    /// Initialize values
15    std::cout << " Basic Linear Transforms " << std::endl;
16    std::cout << "-----" << std::endl;
17    std::cout << "* Enter the alpha value [1.0-3.0]: ";
18    std::cin >> alpha;
19    std::cout << "** Enter the beta value [0-100]: ";
20    std::cin >> beta;
21
22    /// Do the operation new_image(i,j) = alpha*image(i,j) + beta
23    for( int y = 0; y < image.rows; y++ )
24    { for( int x = 0; x < image.cols; x++ )
25        { for( int c = 0; c < 3; c++ )
26            {
27                new_image.at<cv::Vec3b>(y,x)[c] =
28                    cv::saturate_cast<uchar>( alpha *
29 image.at<cv::Vec3b>(y,x)[c] ) + beta );
30            }
31        }
32    }
33
34    /// Show stuff
35    cv::imshow("Original Image", image);
36    cv::imshow("New Image", new_image);
37    cv::imwrite("D:/TRANSLATION/OpenCV Book for PGU/PICTURES/USED IN
38 CODES/out2.jpg", new_image);
39
40    /// Wait until user press some key
41    cv::waitKey();
42
43    return 0;
44 }
```

۲,۶,۲ توضیح

در خط ۱۲، به دلیل اینکه می‌خواهیم تغییراتی در تصویر ورودی ایجاد کنیم، یک شیء جدید **Mat** درست می‌کنیم. همچنین می‌خواهیم این **Mat** جدید ویژگی‌های زیر را هم داشته باشد:

- مقدار اولیه صفر در همه پیکسل‌ها
- اندازه و نوع مطابق با عکس اصلی

تابع `Mat` یک شیء `Mat::zeros` با اندازه و نوع داده شده بر می‌گرداند که همه پیکسل‌های آن صفر است.

حالا برای تغییر کنتراست و روشنایی، باید به همه پیکسل‌های تصویر ورودی دسترسی پیدا کنیم. به دلیل اینکه با `RGB` کار می‌کنیم، به ازای هر پیکسل ۳ مقدار داریم (`R`, `G` و `B`)؛ پس باید به هر کدام از آنها جداگانه دسترسی پیدا کنیم. خطوط ۲۳ تا ۳۲ همین کار را انجام می‌دهند.

به نکات زیر توجه کنید:

- برای دسترسی به هر کدام از پیکسل‌ها از قاعدة روبرو استفاده می‌کنیم: `image.at<Vec3b>(y,x)[c]` که `y` شماره سطر، `X` شماره ستون و `c` هم `G` یا `B` یعنی (۰، ۱ یا ۲) است.
- به خاطر اینکه عمل $\alpha \cdot f(i,j) + \beta$ ممکن است مقادیری بزرگتر از حد مجاز یا غیر صحیح (اگر آلفا اعشاری باشد) تولید کند، از تابع `saturate_cast` استفاده می‌کنیم تا مطمئن شویم که مقدار معتبری در پیکسل خروجی قرار می‌گیرد.

در آخر خروجی را در یک پنجره نشان می‌دهیم.

نکته: به جای استفاده از حلقه `for` برای اعمال فرمول بالا، می‌توانستیم از تابع زیر استفاده کنیم:

```
image.convertTo(new_image, -1, alpha, beta);
```

این تابع مقدار `newImage` را از معادله $newImage = \alpha * image + \beta$ به دست می‌آورد. به هر حال هدف ما در این بخش این بود که شما را با نحوه دسترسی مستقیم به پیکسل‌ها و انجام تبدیلات پیکسلی آشنا کنیم. هر دو روش خروجی یکسانی دارند.

۲.۶.۳ خروجی

برنامه را به صورت زیر اجرا می‌کنیم:

```
BasicLinearTransform.exe "PICTURES /2.jpg"
> Basic Linear Transforms
> -----
> * Enter the alpha value [1.0-3.0]: 1.5
> * Enter the beta value [0-100]: 12
```

خروجی به صورت زیر است:



۲،۷ هموار سازی تصویر

هموار سازی یا مات کردن، یک عمل ساده پردازش تصویری است که بسیار مورد استفاده قرار می‌گیرد. دلایل زیادی برای هموار کردن یک تصویر وجود دارد. در این بخش از هموار کردن برای کاهش نویز استفاده می‌کنیم (با کاربردهای دیگر آن در بخش‌های دیگر آشنا خواهید شد).

به منظور هموار کردن تصویر باید از فیلترها استفاده کرد که عموماً از فیلترهای خطی استفاده می‌شود؛ در این فیلترها مقدار یک پیکسل خروجی (یعنی $(g(i,j))$) به جمع وزن دار پیکسل‌های ورودی بستگی دارد (یعنی $f(i+k, j+l)$).

$$g(i,j) = \sum_{k,l} f(i+k, j+l)h(k, l)$$

به $h(k, l)$ کرنل می‌گویند. کرنل همان ضرایب فیلتر است.

معادله بالا به ما نشان می‌دهد که فیلتر، یک پنجره لغزنده از ضرایب است که روی تصویر حرکت می‌کند. از آنجا که فیلترهای متنوع زیادی وجود دارند، اینجا فقط آنهايی که بیشتر مورد استفاده قرار می‌گيرند را توضیح می‌دهیم:

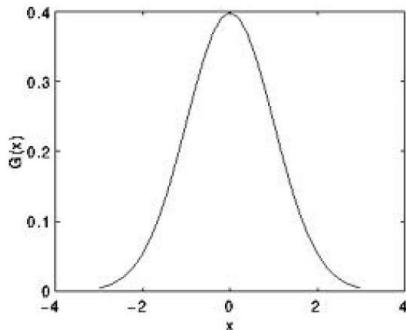
فیلتر جعبه‌ای نرمال شده:^{۳۵}

- ساده‌ترین فیلتر موجود است که هر پیکسل خروجی، میانگین همسایه‌های آن است.
- کرنل این فیلتر به صورت زیر است:

$$K = \frac{1}{K_{width} \cdot K_{height}} \begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{bmatrix}$$

فیلتر گوسی:^{۳۶}

- احتمالاً کاربردی‌ترین فیلتر (اما نه سریع‌ترین) است. فیلتر کردن به روش گوسی از کانون ال ہر نقطه موجود در آرایه ورودی با یک کرنل گوسی و سپس جمع همه آنها برای به دست آوردن خروجی، انجام می‌شود.
- نمودار کرنل گوسی یک بعدی به صورت زیر است:



تابع توزیع گوسی

اگر فرض کیم عکس یک بعدی باشد، با توجه به شکل بالا پیکسلی که در مرکز قرار گرفته بزرگترین وزن را خواهد داشت. با افزایش فاصله از پیکسل مرکزی وزن‌ها نیز کمتر می‌شوند.

نکته: به یاد داشته باشید که گوسی دو بعدی به شکل زیر است:

$$G_0(x, y) = Ae^{\frac{-(x-\mu_x)^2}{2\sigma_x^2} + \frac{-(y-\mu_y)^2}{2\sigma_y^2}}$$

که μ نشان دهنده میانگین و σ نشان دهنده انحراف معیار (به ازای متغیرهای x و y) است.

فیلتر میانه:^{۳۷}

فیلتر میانه گیر روی پیکسل‌های عکس عبور می‌کند و هر پیکسل را با میانه پیکسل‌های همسایه‌اش جایگزین می‌کند.

فیلتر دوگانه:^{۳۸}

تا اینجا فیلترهایی معرفی کردیم که هدف اصلاحیان هموار کردن تصویر ورودی بود. برخی اوقات فیلترها نه فقط نویزها را از بین می‌برند بلکه لبه‌ها را نیز از بین می‌برند. برای جلوگیری از مورد دوم (حداقل در برخی مواقع) می‌توان از فیلتر دوگانه استفاده کرد.

فیلتر دوگانه به روشی مشابه با فیلتر گوسی به هر کدام از پیکسل‌های همسایه، وزنی اختصاص می‌دهد. هر وزن از دو قسمت تشکیل شده است؛ قسمت اول همان وزنی است که در فیلتر گوسی وجود دارد. قسمت دوم هم اختلاف بین روش‌نایی پیکسل فعلی با همسایه‌هایش است. چون هدف این کتاب آموزش آسی وی است از ذکر جزئیات این فیلتر خودداری می‌کنیم.

۲,۷,۱ کد

```

1 #include <iostream>
2 #include <vector>
3
4 #include "opencv2/imgproc/imgproc.hpp"
5 #include "opencv2/highgui/highgui.hpp"
6 #include "opencv2/features2d/features2d.hpp"
7
8 using namespace std;
9 using namespace cv;
10
11 /// Global Variables
12 int DELAY_CAPTION = 1500;
13 int DELAY_BLUR = 100;

```

Median Filter ^{۳۹}

Bilateral Filter ^{۴۰}

```

14     int MAX_KERNEL_LENGTH = 31;
15
16     Mat src; Mat dst;
17     char window_name[] = "Smoothing Demo";
18
19     /// Function headers
20     int display_caption( const char* caption );
21     int display_dst( int delay );
22
23
24     /**
25      * function main
26      */
27     int main( void )
28     {
29         namedWindow( window_name, WINDOW_AUTOSIZE );
30
31         /// Load the source image
32         src = imread( "D:/TRANSLATION/OpenCV Book for PGU/PICTURES/USED IN
33 CODES/1.jpg", 1 );
34
35         if( display_caption( "Original Image" ) != 0 ) { return 0; }
36
37         dst = src.clone();
38         if( display_dst( DELAY_CAPTION ) != 0 ) { return 0; }
39
40
41         /// Applying Homogeneous blur
42         if( display_caption( "Homogeneous Blur" ) != 0 ) { return 0; }
43
44         for ( int i = 1; i < MAX_KERNEL_LENGTH; i = i + 2 )
45             { blur( src, dst, Size( i, i ), Point(-1,-1) );
46                 if( display_dst( DELAY_BLUR ) != 0 ) { return 0; } }
47
48
49         /// Applying Gaussian blur
50         if( display_caption( "Gaussian Blur" ) != 0 ) { return 0; }
51
52         for ( int i = 1; i < MAX_KERNEL_LENGTH; i = i + 2 )
53             { GaussianBlur( src, dst, Size( i, i ), 0, 0 );
54                 if( display_dst( DELAY_BLUR ) != 0 ) { return 0; } }
55
56
57         /// Applying Median blur
58         if( display_caption( "Median Blur" ) != 0 ) { return 0; }
59
60         for ( int i = 1; i < MAX_KERNEL_LENGTH; i = i + 2 )
61             { medianBlur ( src, dst, i );
62                 if( display_dst( DELAY_BLUR ) != 0 ) { return 0; } }
63
64
65         /// Applying Bilateral Filter
66         if( display_caption( "Bilateral Blur" ) != 0 ) { return 0; }
67
68         for ( int i = 1; i < MAX_KERNEL_LENGTH; i = i + 2 )
69             { bilateralFilter ( src, dst, i, i*2, i/2 );
70                 if( display_dst( DELAY_BLUR ) != 0 ) { return 0; } }
71
72         /// Wait until user press a key
73         display_caption( "End: Press a key!" );
74

```

```

75     waitKey(0);
76
77     return 0;
78 }
79
80 /**
81 * @function display_caption
82 */
83 int display_caption( const char* caption )
84 {
85     dst = Mat::zeros( src.size(), src.type() );
86     putText( dst, caption,
87             Point( src.cols/4, src.rows/2 ),
88             FONT_HERSHEY_COMPLEX, 1, Scalar(255, 255, 255) );
89
90     imshow( window_name, dst );
91     int c = waitKey( DELAY_CAPTION );
92     if( c >= 0 ) { return -1; }
93     return 0;
94 }
95
96 /**
97 * @function display_dst
98 */
99 int display_dst( int delay )
100 {
101     imshow( window_name, dst );
102     int c = waitKey( delay );
103     if( c >= 0 ) { return -1; }
104     return 0;
105 }

```

۲،۷،۲ توضیح

در فصل قبل با خیلی از دستورهایی که در کد بالا استفاده شده‌اند، آشنا شدید؛ به همین دلیل فقط به توابع هموار سازی می‌پردازیم.

فیلتر جعبه‌ای نرمال شده (خط ۴۵):

تابع blur چهار آرگومان دارد:

- عکس منبع: src
- عکس مقصد: dst
- اندازه کرnel مورد استفاده را مشخص می‌کند (عرض w و ارتفاع h).
- موقعیت نقطه لنگر^{۳۹} نسبت به همسایه‌ها را مشخص می‌کند. اگر یک مقدار منفی باشد، مرکز کرنل به عنوان نقطه مرجع در نظر گرفته می‌شود.

فیلتر گوسی (خط ۵۳):

تابع GassuainBlur پنج آرگومان دارد:

- عکس منبع: src

- عکس مقصد: dst
- اندازه کرnel مورد استفاده را مشخص می کند (یعنی همسایه های در گیر را مشخص می کند). w و h باید مثبت و فرد باشند، در غیر این صورت از σ_x و σ_y برای محاسبه این اندازه ها استفاده می شود.
- انحراف معیار σ_x : نسبت به x است. با قرار دادن مقدار صفر مشخص می کنیم که σ_x باید از روی اندازه کرnel محاسبه شود.
- انحراف معیار σ_y : نسبت به y است. با قرار دادن مقدار صفر مشخص می کنیم که σ_y باید از روی اندازه کرnel محاسبه شود.

فیلتر میانه (خط ۶۱):

تابع medianFilter سه آرگومان دارد:

- عکس منبع: src
- عکس مقصد؛ باید همنوع src باشد.
- اندازه کرnel است (چون از کرnel مربعی استفاده می شود فقط یک پارامتر کافی است). باید فرد باشد.

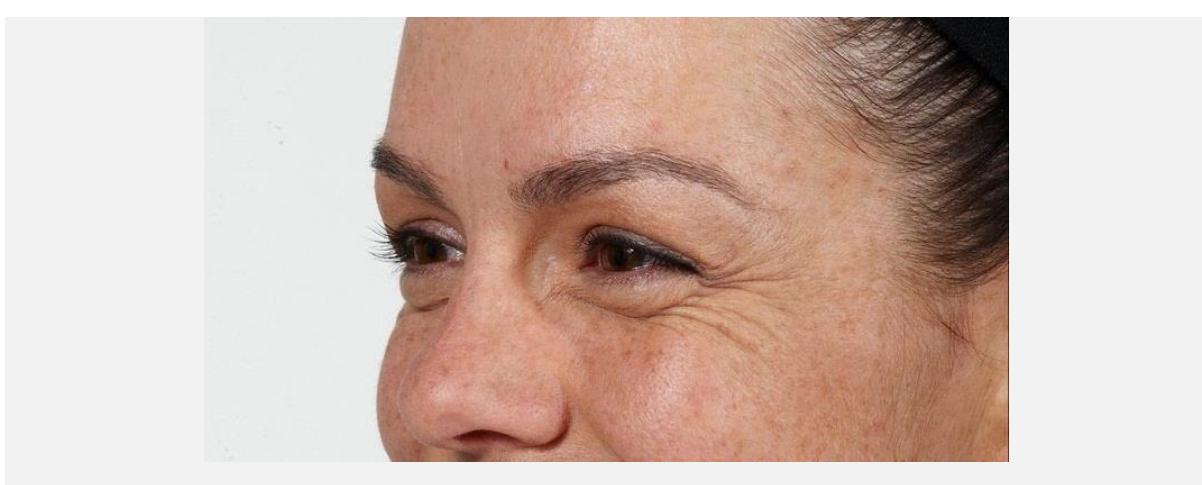
فیلتر دوگانه (خط ۶۹):

تابع bilateralFilter پنج آرگومان دارد:

- تصویر منبع: src
- تصویر مقصد: dst
- قطر همسایگی هر پیکسل d: قدر همسایگی هر پیکسل
- انحراف معیار در فضای رنگی: σ_{color}
- انحراف معیار در فضای مختصات (در مقیاس پیکسل): σ_{space}

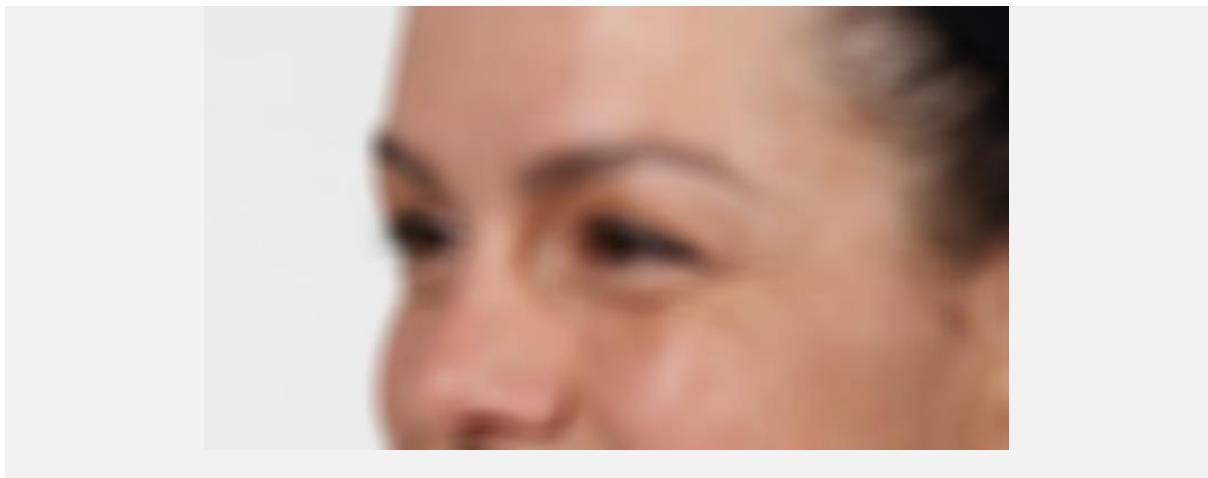
۲,۷,۳ خروجی

برنامه پس از اجرا تصویری را باز می کند و چهار فیلتر توضیح داده شده را به ترتیب روی آن اعمال می کند.

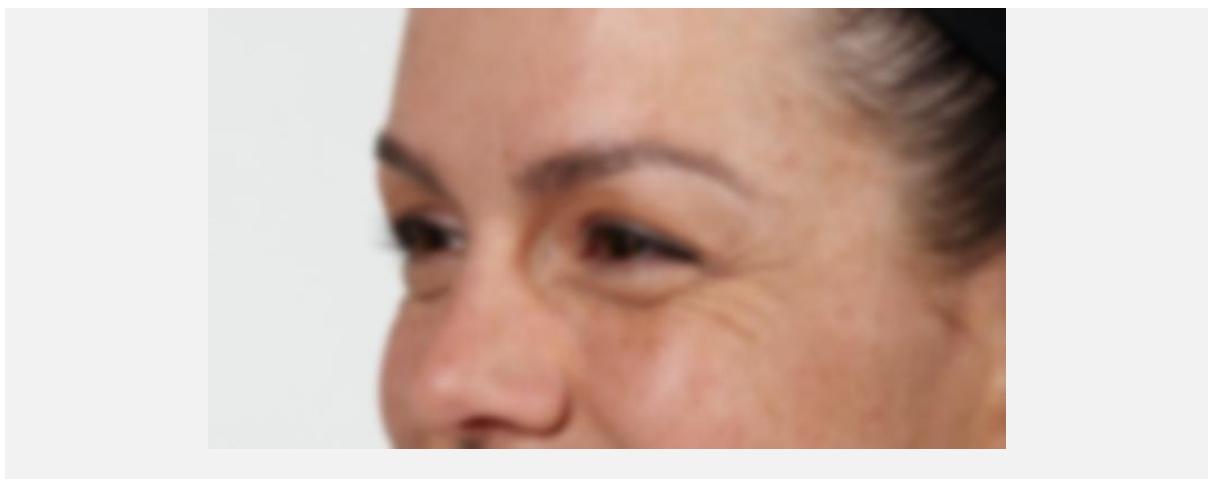


تصویر ورودی

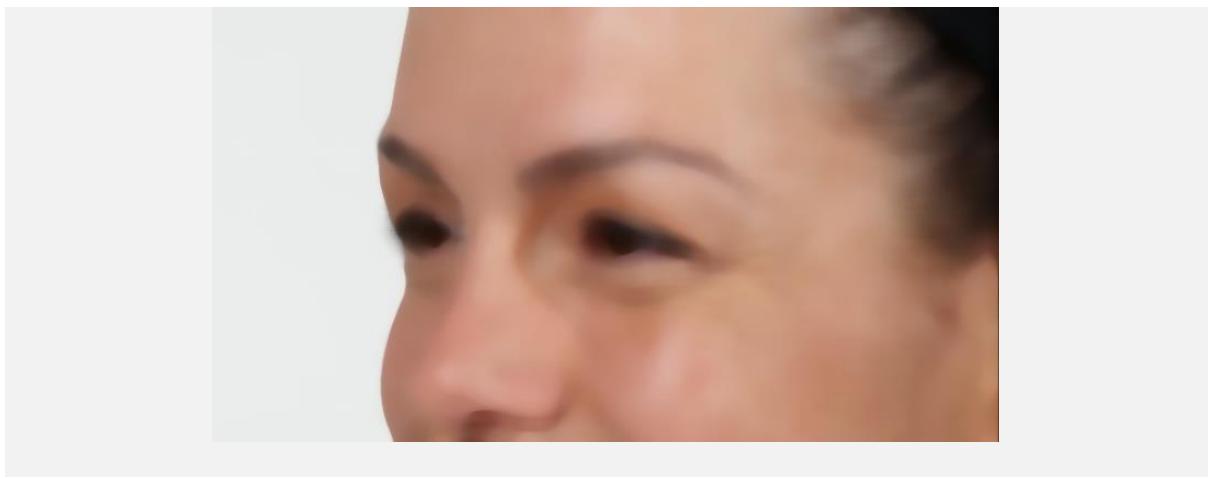
Standard derivation σ .



فیلتر جعبه‌ای نرمال شده با اندازه کرنل ۲۵



فیلتر گوسی با اندازه کرنل ۲۵



فیلتر میانه با اندازه کرنل ۲۵



فیلتر دوگانه با اندازه کرنل ۲۵

به راحتی می‌توان دید که کیفیت تصویر ایجاد شده با فیلتر دوگانه از سایر تصاویر بیشتر است.

۲,۸ اضافه کردن قاب به تصویر

در بخش قبل با کانولوشن و نحوه اعمال آن روی یک تصویر آشنا شدیم. مسئله بر سر پیکسل‌های کناری تصویر بود. چگونه می‌توان این پیکسل‌ها را کانوال کرد در حالی که لنگر کرنل روی آنها قرار دارد و قسمتی از کرنل خارج از تصویر افتاده است؟!

بیشتر توابع آسی وی برای حل این مشکل تصویر را در یک ماتریس بزرگ‌تر کپی می‌کنند و سپس به طور خودکار اطراف تصویر را پر می‌کنند. به این طریق می‌توان بدون هیچ مشکلی کانولوشن را روی تمام پیکسل‌های مورد نیاز اعمال کرد و سپس در انتهای پیکسل‌های اضافه را پاک کرد.

در این بخش دو روش اضافه کردن قاب به تصویر بیان خواهد شد:

.۱: قاب اطراف تصویر را با یک مقدار ثابت پُر می‌کند (مثلاً با عدد ۰).

.۲: سطوح‌های آخر تصویر اصلی را در قسمت قاب کپی می‌کند.

۲,۸,۱ کد

```

1 #include "opencv2/imgproc/imgproc.hpp"
2 #include "opencv2/highgui/highgui.hpp"
3 #include <stdlib.h>
4 #include <stdio.h>
5
6 using namespace cv;
7
8 /// Global Variables
9 Mat src, dst;
10 int top, bottom, left, right;
11 int borderType;
12 Scalar value;
13 char* window_name = "copyMakeBorder Demo";
14 RNG rng(12345);
15
16 /** @function main */
17 int main( int argc, char** argv )
18 {
19     int c;
```

```

21
22     /// Load an image
23     src = imread( argv[1] );
24
25     if( !src.data )
26     { return -1;
27         printf(" No data entered, please enter the path to an image file
28 \n");
29     }
30
31     /// Brief how-to for this program
32     printf( "\n \t copyMakeBorder Demo: \n" );
33     printf( "\t ----- \n" );
34     printf( " ** Press 'c' to set the border to a random constant value
35 \n";
36     printf( " ** Press 'r' to set the border to be replicated \n");
37     printf( " ** Press 'ESC' to exit the program \n");
38
39     /// Create window
40     namedWindow( window_name, CV_WINDOW_AUTOSIZE );
41
42     /// Initialize arguments for the filter
43     top = (int) (0.05*src.rows); bottom = (int) (0.05*src.rows);
44     left = (int) (0.05*src.cols); right = (int) (0.05*src.cols);
45     dst = src;
46
47     imshow( window_name, dst );
48
49     while( true )
50     {
51         c = waitKey(500);
52
53         if( (char)c == 27 )
54             { break; }
55         else if( (char)c == 'c' )
56             { borderType = BORDER_CONSTANT; }
57         else if( (char)c == 'r' )
58             { borderType = BORDER_REPLICATE; }
59
60         value = Scalar( rng.uniform(0, 255), rng.uniform(0, 255),
61         rng.uniform(0, 255) );
62         copyMakeBorder( src, dst, top, bottom, left, right, borderType,
63         value );
64
65         imshow( window_name, dst );
66     }
67
68     return 0;
69 }
```

۲،۸،۲ توضیح

در خطوط ۴۳ و ۴۴ متغیرهایی که اندازه قاب را مشخص می‌کنند (بالا، پایین، چپ و راست) تعریف شده است. در اینجا مقدارشان پنج درصد اندازه تصویر src قرار داده شده‌اند.

در خط ۴۹ برنامه وارد یک حلقه while می‌شود. با فشردن کلید "c" یا "r"، متغیر borderType به ترتیب یکی از مقادیر BORDER_REPLICATE یا BORDER_CONSTANT را می‌گیرد.

در هر تکرار (هر ۵۰۰ میلی ثانیه)، متغیر `value` در خط ۶۰ به روز می‌شود. این مقدار یک عدد تصادفی است که توسط `rng` و در بازهٔ ۰ تا ۲۵۵ تولید می‌شود.

در انتهای در خط ۶۲ برای اضافه کردن قاب، تابع `copyMakeBorder` صدای زده می‌شود. آرگومان‌های این تابع به شرح زیر هستند:

- `src`: تصویر منبع
- `dst`: تصویر مقصد
- `:top, bottom, left, right`: طول قاب‌های هر طرف تصویر در مقیاس پیکسل هستند.
- `:borderType`: نوع قاب را تعیین می‌کند. می‌تواند ثابت یا تکرار شونده باشد.
- `:value`: اگر مقدار `BORDER_CONSTANT`، همان مقدار ثابتی است که در قاب قرار می‌گیرد.

۲,۸,۳ خروجی

برنامه را به صورت زیر اجرا می‌کنیم:

`CopyMakeBorder.exe "PICTURES /2.jpg"`



۲,۹ مولد عدد تصادفی و ترسیم متن

در بخش **شکل‌های هندسی ساده**، با دادن پارامترهایی مثل مختصات، رنگ، نازکی و... به توابع هندسی، شکل‌های متفاوتی کشیدیم. در آنجا مقادیر ثابتی را به عنوان پارامتر به آن توابع ارسال می‌کردیم. در این بخش می‌خواهیم از مقادیر تصادفی برای آن پارامترها استفاده کنیم و همچنین عکس مورد نظرمان را با تعداد زیادی شکل هندسی پر کنیم.

برای تولید اعداد تصادفی از کلاس RNG استفاده خواهیم کرد.

۲،۹،۱

```
1 #include <opencv2/core/core.hpp>
2 #include <opencv2/highgui/highgui.hpp>
3 #include <iostream>
4 #include <stdio.h>
5
6 using namespace cv;
7
8 /// Global Variables
9 const int NUMBER = 100;
10 const int DELAY = 5;
11
12 const int window_width = 900;
13 const int window_height = 600;
14 int x_1 = -window_width/2;
15 int x_2 = window_width*3/2;
16 int y_1 = -window_width/2;
17 int y_2 = window_width*3/2;
18
19 /// Function headers
20 static Scalar randomColor( RNG& rng );
21 int Drawing_Random_Lines( Mat image, char* window_name, RNG rng );
22 int Drawing_Random_Rectangles( Mat image, char* window_name, RNG rng );
23
24 int Drawing_Random_Ellipses( Mat image, char* window_name, RNG rng );
25
26 int Drawing_Random_Polylines( Mat image, char* window_name, RNG rng );
27
28 int Drawing_Random_Filled_Polygons( Mat image, char* window_name,
29 RNG rng );
30 int Drawing_Random_Circles( Mat image, char* window_name, RNG rng );
31 int Displaying_Random_Text( Mat image, char* window_name, RNG rng );
32 int Displaying_Big_End( Mat image, char* window_name, RNG rng );
33
34 /**
35 * @function main
36 */
37
38 int main( void )
39 {
40     int c;
41
42     /// Start creating a window
43     char window_name[] = "Drawing_2 Tutorial";
44
45     /// Also create a random object (RNG)
46     RNG rng( 0xFFFFFFFF );
47
48     /// Initialize a matrix filled with zeros
49     Mat image = Mat::zeros( window_height, window_width, CV_8UC3 );
50     /// Show it in a window during DELAY ms
51     imshow( window_name, image );
52     waitKey( DELAY );
53
54     /// Now, let's draw some lines
55     c = Drawing_Random_Lines(image, window_name, rng);
56     if( c != 0 ) return 0;
```

```

57     /// Go on drawing, this time nice rectangles
58     c = Drawing_Random_Rectangles(image, window_name, rng);
59     if( c != 0 ) return 0;
60
61     /// Draw some ellipses
62     c = Drawing_Random_Ellipses( image, window_name, rng );
63     if( c != 0 ) return 0;
64
65     /// Now some polylines
66     c = Drawing_Random_Polylines( image, window_name, rng );
67     if( c != 0 ) return 0;
68
69     /// Draw filled polygons
70     c = Drawing_Random_Filled_Polygons( image, window_name, rng );
71     if( c != 0 ) return 0;
72
73     /// Draw circles
74     c = Drawing_Random_Circles( image, window_name, rng );
75     if( c != 0 ) return 0;
76
77     /// Display text in random positions
78     c = Displaying_Random_Text( image, window_name, rng );
79     if( c != 0 ) return 0;
80
81     /// Displaying the big end!
82     c = Displaying_Big_End( image, window_name, rng );
83     if( c != 0 ) return 0;
84
85     waitKey(0);
86     return 0;
87 }
88
89 // Function definitions
90
91 /**
92  * @function randomColor
93  * @brief Produces a random color given a random object
94  */
95 static Scalar randomColor( RNG& rng )
96 {
97     int icolor = (unsigned) rng;
98     return Scalar( icolor&255, (icolor>>8)&255, (icolor>>16)&255 );
99 }
100
101
102 /**
103  * @function Drawing_Random_Lines
104  */
105 int Drawing_Random_Lines( Mat image, char* window_name, RNG rng )
106 {
107     Point pt1, pt2;
108
109     for( int i = 0; i < NUMBER; i++ )
110     {
111         pt1.x = rng.uniform( x_1, x_2 );
112         pt1.y = rng.uniform( y_1, y_2 );
113         pt2.x = rng.uniform( x_1, x_2 );
114         pt2.y = rng.uniform( y_1, y_2 );
115
116
117

```

```

118     line( image, pt1, pt2, randomColor(rng), rng.uniform(1, 10), 8
119 );
120     imshow( window_name, image );
121     if( waitKey( DELAY ) >= 0 )
122     { return -1; }
123 }
124
125     return 0;
126 }
127
128 /**
129 * @function Drawing_Rectangles
130 */
131 int Drawing_Random_Rectangles( Mat image, char* window_name, RNG rng
132 )
133 {
134     Point pt1, pt2;
135     int lineType = 8;
136     int thickness = rng.uniform( -3, 10 );
137
138     for( int i = 0; i < NUMBER; i++ )
139     {
140         pt1.x = rng.uniform( x_1, x_2 );
141         pt1.y = rng.uniform( y_1, y_2 );
142         pt2.x = rng.uniform( x_1, x_2 );
143         pt2.y = rng.uniform( y_1, y_2 );
144
145         rectangle( image, pt1, pt2, randomColor(rng), MAX( thickness, -1
146 ), lineType );
147
148         imshow( window_name, image );
149         if( waitKey( DELAY ) >= 0 )
150             { return -1; }
151     }
152
153     return 0;
154 }
155
156 /**
157 * @function Drawing_Random_Ellipses
158 */
159 int Drawing_Random_Ellipses( Mat image, char* window_name, RNG rng )
160 {
161     int lineType = 8;
162
163     for ( int i = 0; i < NUMBER; i++ )
164     {
165         Point center;
166         center.x = rng.uniform(x_1, x_2);
167         center.y = rng.uniform(y_1, y_2);
168
169         Size axes;
170         axes.width = rng.uniform(0, 200);
171         axes.height = rng.uniform(0, 200);
172
173         double angle = rng.uniform(0, 180);
174
175         ellipse( image, center, axes, angle, angle - 100, angle + 200,
176                 randomColor(rng), rng.uniform(-1,9), lineType );
177
178         imshow( window_name, image );

```

```

179     if( waitKey(DELAY) >= 0 )
180     { return -1; }
181   }
182 }
183
184 return 0;
185 }
186
187 /**
188 * @function Drawing_Random_Polylines
189 */
190 int Drawing_Random_Polylines( Mat image, char* window_name, RNG rng
191 )
192 {
193   int lineType = 8;
194
195   for( int i = 0; i< NUMBER; i++ )
196   {
197     Point pt[2][3];
198     pt[0][0].x = rng.uniform(x_1, x_2);
199     pt[0][0].y = rng.uniform(y_1, y_2);
200     pt[0][1].x = rng.uniform(x_1, x_2);
201     pt[0][1].y = rng.uniform(y_1, y_2);
202     pt[0][2].x = rng.uniform(x_1, x_2);
203     pt[0][2].y = rng.uniform(y_1, y_2);
204     pt[1][0].x = rng.uniform(x_1, x_2);
205     pt[1][0].y = rng.uniform(y_1, y_2);
206     pt[1][1].x = rng.uniform(x_1, x_2);
207     pt[1][1].y = rng.uniform(y_1, y_2);
208     pt[1][2].x = rng.uniform(x_1, x_2);
209     pt[1][2].y = rng.uniform(y_1, y_2);
210
211     const Point* ppt[2] = {pt[0], pt[1]};
212     int npt[] = {3, 3};
213
214     polyline(image, ppt, npt, 2, true, randomColor(rng),
215 rng.uniform(1,10), lineType);
216
217     imshow( window_name, image );
218     if( waitKey(DELAY) >= 0 )
219     { return -1; }
220   }
221   return 0;
222 }
223
224 /**
225 * @function Drawing_Random_Filled_Polygons
226 */
227 int Drawing_Random_Filled_Polygons( Mat image, char* window_name,
228 RNG rng )
229 {
230   int lineType = 8;
231
232   for ( int i = 0; i < NUMBER; i++ )
233   {
234     Point pt[2][3];
235     pt[0][0].x = rng.uniform(x_1, x_2);
236     pt[0][0].y = rng.uniform(y_1, y_2);
237     pt[0][1].x = rng.uniform(x_1, x_2);
238     pt[0][1].y = rng.uniform(y_1, y_2);
239     pt[0][2].x = rng.uniform(x_1, x_2);

```

```

240     pt[0][2].y = rng.uniform(y_1, y_2);
241     pt[1][0].x = rng.uniform(x_1, x_2);
242     pt[1][0].y = rng.uniform(y_1, y_2);
243     pt[1][1].x = rng.uniform(x_1, x_2);
244     pt[1][1].y = rng.uniform(y_1, y_2);
245     pt[1][2].x = rng.uniform(x_1, x_2);
246     pt[1][2].y = rng.uniform(y_1, y_2);
247
248     const Point* ppt[2] = {pt[0], pt[1]};
249     int npt[] = {3, 3};
250
251     fillPoly( image, ppt, npt, 2, randomColor(rng), lineType );
252
253     imshow( window_name, image );
254     if( waitKey(DELAY) >= 0 )
255         { return -1; }
256     }
257     return 0;
258 }
259
260 /**
261 * @function Drawing_Random_Circles
262 */
263 int Drawing_Random_Circles( Mat image, char* window_name, RNG rng )
264 {
265     int lineType = 8;
266
267     for (int i = 0; i < NUMBER; i++)
268     {
269         Point center;
270         center.x = rng.uniform(x_1, x_2);
271         center.y = rng.uniform(y_1, y_2);
272
273         circle( image, center, rng.uniform(0, 300), randomColor(rng),
274                 rng.uniform(-1, 9), lineType );
275
276         imshow( window_name, image );
277         if( waitKey(DELAY) >= 0 )
278             { return -1; }
279     }
280
281     return 0;
282 }
283
284 /**
285 * @function Displaying_Random_Text
286 */
287 int Displaying_Random_Text( Mat image, char* window_name, RNG rng )
288 {
289     int lineType = 8;
290
291     for ( int i = 1; i < NUMBER; i++ )
292     {
293         Point org;
294         org.x = rng.uniform(x_1, x_2);
295         org.y = rng.uniform(y_1, y_2);
296
297         putText( image, "Testing text rendering", org, rng.uniform(0,8),
298                 rng.uniform(0,100)*0.05+0.1, randomColor(rng),
299                 rng.uniform(1, 10), lineType );
300

```

```

301     imshow( window_name, image );
302     if( waitKey(Delay) >= 0 )
303     { return -1; }
304   }
305
306   return 0;
307 }
308
309 /**
310  * @function Displaying_Big_End
311 */
312 int Displaying_Big_End( Mat image, char* window_name, RNG )
313 {
314   Size textSize = getTextSize("OpenCV forever!",
315 FONT_HERSHEY_COMPLEX, 3, 5, 0);
316   Point org((window_width - textSize.width)/2, (window_height -
317 textSize.height)/2);
318   int lineType = 8;
319
320   Mat image2;
321
322   for( int i = 0; i < 255; i += 2 )
323   {
324     image2 = image - Scalar::all(i);
325     putText( image2, "OpenCV forever!", org, FONT_HERSHEY_COMPLEX,
326 3,
327             Scalar(i, i, 255), 5, lineType );
328
329     imshow( window_name, image2 );
330     if( waitKey(Delay) >= 0 )
331     { return -1; }
332   }
333
334   return 0;
}

```

۲,۹,۲ توضیح

از تابع main شروع می‌کنیم. اولین کاری که در این تابع انجام داده‌ایم، ساخت یک شیء RNG است (خط ۴۶). کلاس RNG یک سازنده عدد تصادفی است. در این مثال rng یک شیء از نوع RNG است که با ۰xFFFFFFFF مقداردهی اولیه شده است.

سپس یک ماتریس با مقدار اولیه صفر (که یعنی به رنگ سیاه است) و با طول، عرض و نوع CV_8UC3 درست می‌کنیم (خط ۴۹).

در ادامه شکل‌های تصادفی را رسم می‌کنیم. اگر به خطوط ۵۴ تا ۸۴ نگاه کنید خواهید دید که این خطوط به هشت قسمت، که در واقع هشت تابع هستند، تقسیم شده‌اند. پیاده سازی این توابع تقریباً یکسان است، پس فقط سه تا از این توابع را بررسی می‌کنیم:

تابع Drawing_Random_Lines •

در این تابع:

- حلقة for (خطوط ۱۱۰ تا ۱۲۰) به تعداد NUMBER تکرار می‌شود. از آنجایی که تابع line در این حلقه قرار

دارد، پس یعنی به تعداد NUMBER، خط رسم خواهد شد.

- دو سر خط با نقاط pt1 و pt2 مشخص می‌شوند. مثلاً pt1 به شکل زیر مقدار دهی می‌شود:

```

pt1.x = rng.uniform( x_1, x_2 );
pt1.y = rng.uniform( y_1, y_2 );

```

- می دانیم که rng یک شیء RNG است. در کد بالا تابع `rng.uniform(a,b)` را صدا می زنیم. این تابع یک عدد تصادفی با توزیع یکنواخت بین مقادیر a و b تولید می کند (شامل a و فاقد b).
- از توضیحات بالا متوجه می شویم که رأس های pt1 و pt2 کاملاً تصادفی خواهند بود؛ پس موقعیت خطها غیر قابل پیش بینی می شود و این یک جلوه بصری زیبا تولید می کند.
- توجه کنید که در تابع `line` به جای رنگ، تابع `randomColor(rng)` را قرار داده ایم. این تابع به شکل زیر پیاده سازی شده است:

```
static Scalar randomColor( RNG& rng )
{
    int icolor = (unsigned) rng;
    return Scalar( icolor&255, (icolor>>8)&255, (icolor>>16)&255 );
}
```

همان طور که می بینید، خروجی این تابع یک `Scalar` است که با سه مقدار تصادفی پر شده و این سه مقدار نشان دهنده R، G و B هستند. پس رنگ خطها هم تصادفی خواهد بود!

توضیحات بالا برای تابع دیگر که دایره، مستطیل، چند ضلعی و... تولید می کنند یکسان است. فقط پارامترهای دیگر مثل مرکز دایره یا بیضی و رأس های مستطیل و چند ضلعی نیز تصادفی خواهند بود.

• **تابع Display_Random_Text**

در این تابع همه چیز آشنا به نظر می رسد، اما عبارت زیر جدید است:

```
putText( image, "Testing text rendering", org, rng.uniform(0,8),
          rng.uniform(0,100)*0.05+0.1, randomColor(rng),
          rng.uniform(1, 10), lineType );
```

- این تابع چه کاری انجام می دهد؟! در مورد این مثال:
- متن "Testing text rendering" را روی عکس `image` رسم می کند.
- گوشة پایین و چپ متن در نقطه `org` قرار دارد.
- نوع فونت، یک عدد تصادفی در بازه ۰ تا ۸ است.
- مقیاس فونت با عبارت `rng.uniform(0, 100) * 0.05 + 0.1` که یک عدد تصادفی بین ۰.۱ تا ۵.۱ است، مشخص شده.
- رنگ متن تصادفی است (با تابع `randomColor(rng)` به دست می آید).
- ضخامت متن بین ۰ تا ۱۰ است که با تابع `rng.uniform(1, 10)` مشخص می شود.

در نتیجه به تعداد `NUMBER`، متن روی عکسman خواهیم داشت که در مکان های تصادفی قرار گرفته اند.

• **تابع Display_Big_End**

به جز تابع `getTextSize` (که اندازه متن داده شده را بر می گرداند)، تنها قطعه کد زیر جدید است:

```
image2 = image - Scalar::all(i);
```

یعنی `image2` برابر است با تفریق(`i`).`Scalar::all` از `image`. در حقیقت مقدار هر پیکسل `image2` برابر است با تفریق مقدار آن از `image` (به یاد داشته باشید که هر پیکسل مشکل از سه مقدار R، G و B است، پس این عمل روی همه آنها تأثیر می گذارد).

همچنین به یاد داشته باشید که عمل تفريقي خودش به صورت خودکار عمل `saturate_cast` را انجام می‌دهد تا همیشه مقادير معتبر (که در اين مثال بین ۰ تا ۲۵۵ است) را توليد کند.

۴.۹.۳ خروجي

روندي اجرائي برنامه به صورت زير است:

۱. ابتدا به تعداد NUMBER خط تصادفي روی صفحه نشان داده می‌شود:



۲. سپس تعدادي مستطيل تصادفي نشان داده می‌شود.

۳. حالا تعدادي بيضي با مختصات، اندازه، نازکي و اندازه قوس تصادفي کشide می‌شود:



۴. بعد تعدادي چند ضلعي تو خالي کشide می‌شود:



۵. سپس چند ضلعي های توپر.

۶. و آخرين شكل هندسي، دايره:



۷. بعد از آن، متن "Testing text rendering" چندين بار با فونت، اندازه، رنگ و مختصات مختلف نشان داده می‌شود.

۸. در نهايit عبارت OpenCV For Ever روی تصویر به تنهايی نمايش داده می‌شود:



۲،۱۰ بزرگنمایی و کوچکنمایی تصویر

معمولًا نیاز است که اندازه یک عکس را تعییر داد. در آ سی وی به منظور تغییر اندازه عکس‌هاتابع تبدیل هندسی `resize` وجود دارد ولی در این بخش به بیان مفهوم هرم تصویر می‌پردازیم که کاربردهای زیادی در برنامه‌های بینایی دارد.

هرم تصویر مجموعه‌ای از تصاویر است که همگی از یک تصویر اصلی سرچشمه گرفته‌اند و به ترتیب کوچک می‌شوند تا به یک اندازه مشخص برسند.

دو نوع هرم تصویر داریم:

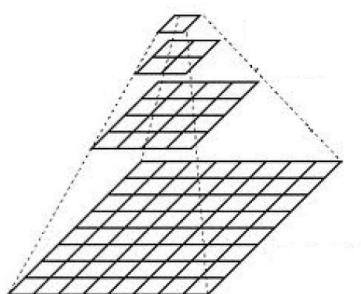
- هرم گوسی^{۴۱}: در این روش برای ساخت هرم از میانگین‌گیری گوسی استفاده می‌شود.

- هرم لaplاسی^{۴۲}: مشابه هرم گوسی است با این تفاوت که از یک تبدیل لaplاس برای ساخت هرم استفاده می‌کند.

هرم موجود در آ سی وی از نوع گوسی است.

هرم گوسی:

هرم را به صورت یک مجموعه از لایه‌ها در نظر بگیرید که هر چه لایه در سطح بالاتری قرار گرفته باشد، کوچک‌تر خواهد بود.



هرم تصویر (سطح زیرین تصویر اصلی است)

لایه‌ها از پایین به بالا شماره گذاری شده‌اند. پس لایه $i+1$ که با G_{i+1} نشان داده می‌شود، از لایه i که با G_i نشان داده می‌شود کوچک‌تر است.

در هرم گوسی برای تولید لایه $i+1$ از روی لایه i ، به شیوه زیر عمل می‌کنیم:

- G_i را با یک کرنل گوسی زیر کانوالو می‌کنیم:

$$\frac{1}{16} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

- همه سطرها و ستون‌های زوج را پاک می‌کنیم.

Gaussian pyramid ^{۴۱}

Laplacian pyramid ^{۴۲}

عکس به دست آمده دقیقاً یک چهارم عکس بیشین خواهد بود. با تکرار فرایند بالا روی عکس اصلی (یعنی G_0)، می‌توان تمام هرم را درست کرد.

رویه بالا به منظور کوچکنمایی تصویر بود. برای بزرگنمایی تصویر باید مراحل زیر را انجام دهیم:

- ابتدا ابعاد لایه آ را دو برابر کرده و سطرهای ستون‌های زوج آن را با صفر پر می‌کنیم.
- تصویر به دست آمده در مرحله قبل را در کرنل بالا (البته با درایه‌های آن که در ۴ ضرب شده‌اند) کانوالو می‌کنیم. با این کار مقدار پیکسل‌های جدید معرفی شده در مرحله قبل که با صفر پر شده بودند، تخمین زده می‌شوند.

دو رویه بالا (یعنی کوچکنمایی و بزرگنمایی)، در توابع `pyrUp` و `pyrDown` پیاده سازی شده‌اند.

۲.۱۰. کد

```

1 #include "opencv2/imgproc/imgproc.hpp"
2 #include "opencv2/highgui/highgui.hpp"
3 #include <math.h>
4 #include <stdlib.h>
5 #include <stdio.h>
6
7 using namespace cv;
8
9 /// Global variables
10 Mat src, dst, tmp;
11 char* window_name = "Pyramids Demo";
12
13 /**
14 * @function main
15 */
16 int main( int argc, char** argv )
17 {
18     /// General instructions
19     printf( "\n Zoom In-Out demo \n" );
20     printf( "----- \n" );
21     printf( " * [u] -> Zoom in \n" );
22     printf( " * [d] -> Zoom out \n" );
23     printf( " * [ESC] -> Close program \n \n" );
24
25     /// Test image - Make sure it's divisible by 2^{n}
26     src = imread( argv[1] );
27     if( !src.data )
28     { printf(" No data! -- Exiting the program \n");
29         return -1; }
30
31     tmp = src;
32     dst = tmp;
33
34     /// Create window
35     namedWindow( window_name, CV_WINDOW_AUTOSIZE );
36     imshow( window_name, dst );
37
38     /// Loop
39     while( true )
40     {
41         int c;
42         c = waitKey(10);
43
44         if( (char)c == 27 )
45             { break; }

```

```

46     if( (char)c == 'u' )
47     { pyrUp( tmp, dst, Size( tmp.cols*2, tmp.rows*2 ) );
48     printf( "** Zoom In: Image x 2 \n" );
49     }
50 else if( (char)c == 'd' )
51 { pyrDown( tmp, dst, Size( tmp.cols/2, tmp.rows/2 ) );
52 printf( "** Zoom Out: Image / 2 \n" );
53 }
54
55 imshow( window_name, dst );
56 tmp = dst;
57 }
58 return 0;
59 }

```

۲،۱۰،۲ توضیح

در خط ۴۴ مشخص شده که اگر کاربر کلید ESC را فشار دهد از برنامه خارج شود.

در خط ۴۶ مشخص شده که اگر کاربر کلید u را فشار دهد عمل بزرگنمایی توسط تابع pyrUp انجام شود. این تابع سه آرگومان دارد:

- tmp: تصویر ورودی است. در ابتدا برابر با تصویر src است.
- dst: تصویر خروجی است (همان که ببروی صفحه نمایش داده خواهد شد و در حقیقت دو برابر تصویر ورودی است).
- Size(tmp.cols*2, tmp.rows*2) : اندازه تصویر خروجی است. از آنجایی که قصد بزرگنمایی داریم، تابع pyrUp انتظار دارد که این اندازه دو برابر اندازه تصویر ورودی باشد.

در خط ۵۰ مشخص شده که اگر کاربر کلید d را فشار دهد عمل کوچکنمایی توسط تابع pyrDown انجام شود. این تابع سه آرگومان دارد:

- tmp: تصویر ورودی است. در ابتدا برابر با تصویر src است.
- dst: تصویر خروجی است (همان که ببروی صفحه نمایش داده خواهد شد و در حقیقت نصف تصویر ورودی است).
- Size(tmp.cols/2 , tmp.rows/2) : اندازه تصویر خروجی است. از آنجایی که قصد کوچکنمایی داریم، تابع pyrDown انتظار دارد که این اندازه نصف اندازه تصویر ورودی باشد.

توجه داشته باشید که ابعاد تصویر ورودی به توابع pyrUp و pyrDown باید حتماً ضریبی از دو باشد، در غیر این صورت خطایی مبتنی بر رعایت نکردن این محدودیت دریافت خواهد کرد.

۲،۱۰،۳ خروجی

برنامه را به صورت زیر اجرا می کنیم:

PyrUpDown.exe "PICTURES /1.jpg"



تصویر ورودی



نتیجه دو بار کوچکنمایی تصویر اصلی



نتیجه دو بار بزرگنمایی تصویر کوچک شده

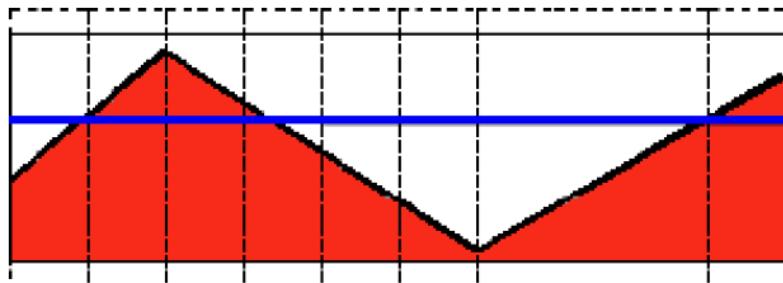
همانطور که می‌بینید تصویر آخر دارای وضوح کمتری نسبت به تصویر ورودی است.

۲،۱۱ آستانه‌گذاری

در این بخش به بررسی چگونگی انجام عملیات آستانه‌گذاری تصویر به کمک تابع `threshold` می‌پردازیم.

آستانه‌گذاری ساده‌ترین روش گروه بندی است. از آستانه‌گذاری می‌توان جهت جدا سازی نواحی مختلف تصویر استفاده کرد. در این سی و پنج نوع عمل آستانه‌گذاری وجود دارد.

برای ادامه این بخش فرض کنید یک تصویر مرجع با مقادیر روشنایی $src(x,y)$ برای پیکسل‌های آن داریم. خط افقی آبی رنگ مقدار آستانه را نمایش می‌دهد.



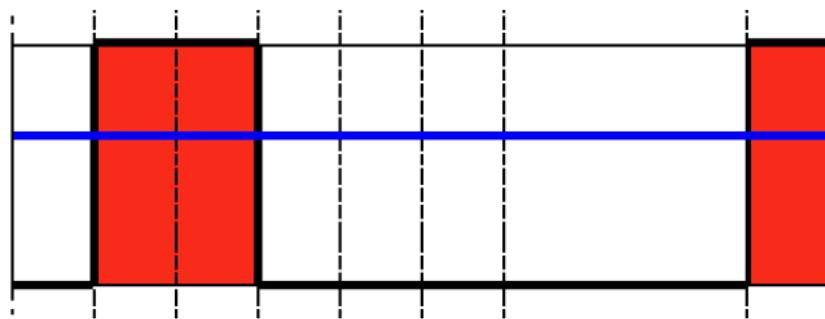
تصویر مرجع و خط آستانه

آستانه‌گذاری باینری^{۴۳}:

در این نوع به صورت زیر عمل می‌شود:

$$dst(x, y) = \begin{cases} maxVal & \text{if } src(x, y) > thresh \\ 0 & \text{otherwise} \end{cases}$$

يعنى اگر روشنایی پیکسل $src(x,y)$ بيشتر از مقدار $thresh$ بود، آنگاه مقدار آن پیکسل برابر با $MaxVal$ قرار داده می‌شود. در غير اين صورت مقدار آن برابر صفر قرار داده می‌شود.



نتیجه آستانه‌گذاری باینری

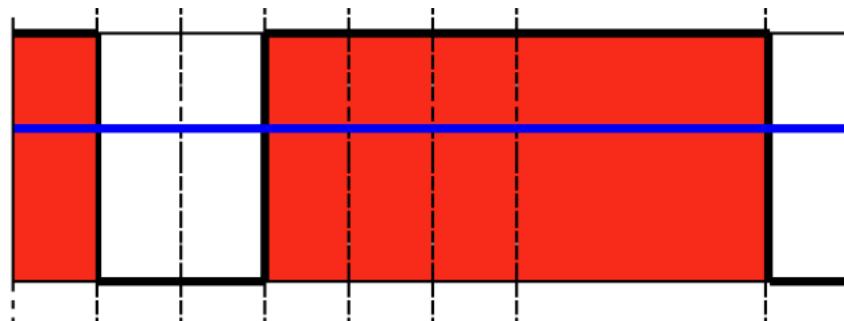
آستانه‌گذاری باینری معکوس^{۴۴}:

در این نوع به صورت زیر عمل می‌شود:

$$dst(x, y) = \begin{cases} 0 & \text{if } src(x, y) > thresh \\ maxVal & \text{otherwise} \end{cases}$$

يعنى اگر روشنایی پیکسل $src(x,y)$ بيشتر از $thresh$ بود، آنگاه مقدار آن پیکسل برابر با صفر قرار داده می‌شود. در غير اين صورت مقدار آن برابر $MaxVal$ خواهد بود.

Threshold binary^{۴۵}
Threshold binary, inverted^{۴۶}

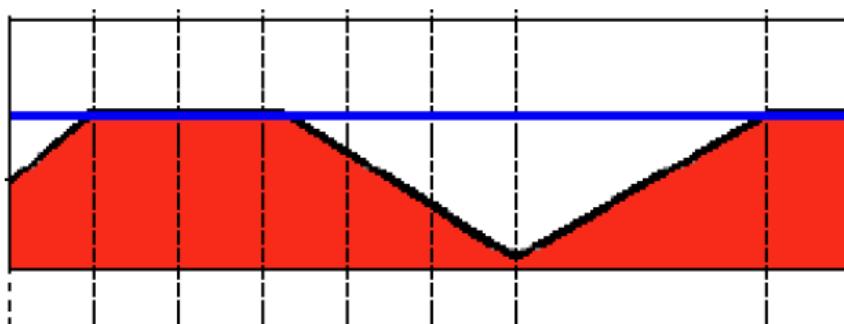


نتیجه آستانه‌گذاری باینری معکوس

بریدن^{۴۵}:

در این نوع به صورت زیر عمل می‌شود:

$$dst(x, y) = \begin{cases} threshold & \text{if } src(x, y) > thresh \\ src(x, y) & \text{otherwise} \end{cases}$$

بیشترین مقدار مجاز برای پیکسل‌ها، مقدار $thresh$ است. اگر مقدار پیکسلی بیشتر از $thresh$ بود، مقداری از آن بریده می‌شود.

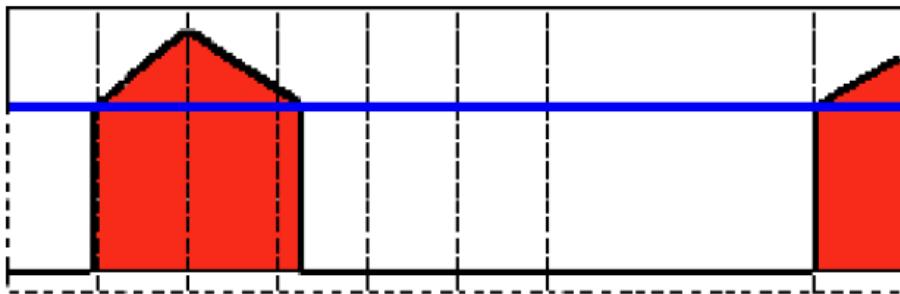
نتیجه بریدن

آستانه‌گذاری به صفر^{۴۶}:

در این نوع به صورت زیر عمل می‌شود:

$$dst(x, y) = \begin{cases} src(x, y) & \text{if } src(x, y) > thresh \\ 0 & \text{otherwise} \end{cases}$$

اگر ($src(x, y)$) کوچک‌تر از $thresh$ بود، مقدار آن صفر قرار داده خواهد شد.Truncate^{۴۷}Threshold to zero^{۴۸}



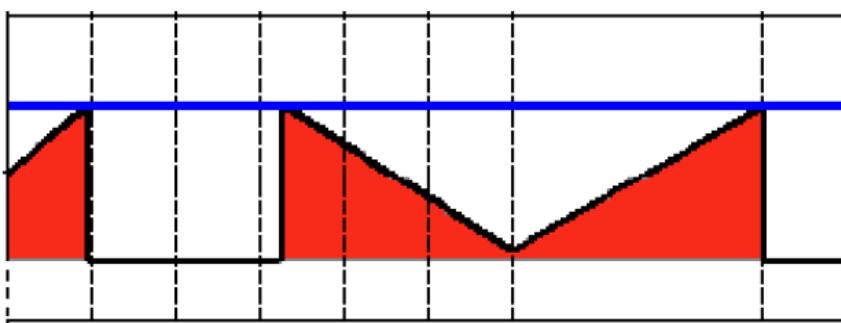
نتیجه آستانه‌گذاری به صفر

آستانه‌گذاری به صفر معکوس^{۴۷}:

در این نوع به صورت زیر عمل می‌شود:

$$dst(x,y) = \begin{cases} 0 & \text{if } src(x,y) > thresh \\ src(x,y) & \text{otherwise} \end{cases}$$

اگر $src(x,y)$ بزرگ‌تر از $thresh$ بود، مقدار آن را صفر می‌گذاریم.



نتیجه آستانه‌گذاری به صفر معکوس

۲,۱۱,۱

```

1 #include "opencv2/imgproc/imgproc.hpp"
2 #include "opencv2/highgui/highgui.hpp"
3 #include <stdlib.h>
4 #include <stdio.h>
5
6 using namespace cv;
7
8 /// Global variables
9
10 int threshold_value = 0;
11 int threshold_type = 3;
12 int const max_value = 255;
13 int const max_type = 4;
14 int const max_BINARY_value = 255;
15

```

Threshold to zero, inverted^{۴۸}

```

16 Mat src, src_gray, dst;
17 char* window_name = "Threshold Demo";
18
19 char* trackbar_type = "Type: \n 0: Binary \n 1: Binary Inverted \n
20 2: Truncate \n 3: To Zero \n 4: To Zero Inverted";
21 char* trackbar_value = "Value";
22
23 /// Function headers
24 void Threshold_Demo( int, void* );
25
26 /**
27 * @function main
28 */
29 int main( int argc, char** argv )
30 {
31     /// Load an image
32     src = imread( argv[1], 1 );
33
34     /// Convert the image to Gray
35     cvtColor( src, src_gray, CV_RGB2GRAY );
36
37     /// Create a window to display results
38     namedWindow( window_name, CV_WINDOW_AUTOSIZE );
39
40     /// Create Trackbar to choose type of Threshold
41     createTrackbar( trackbar_type,
42                     window_name, &threshold_type,
43                     max_type, Threshold_Demo );
44
45     createTrackbar( trackbar_value,
46                     window_name, &threshold_value,
47                     max_value, Threshold_Demo );
48
49     /// Call the function to initialize
50     Threshold_Demo( 0, 0 );
51
52     /// Wait until user finishes program
53     while(true)
54     {
55         int c;
56         c = waitKey( 20 );
57         if( (char)c == 27 )
58             { break; }
59     }
60
61 }
62
63
64 /**
65 * @function Threshold_Demo
66 */
67 void Threshold_Demo( int, void* )
68 {
69     /* 0: Binary
70      1: Binary Inverted
71      2: Threshold Truncated
72      3: Threshold to Zero
73      4: Threshold to Zero Inverted
74 */
75
76

```

```

77     threshold( src_gray, dst, threshold_value,
78     max_BINARY_value,threshold_type );
79
80     imshow( window_name, dst );
}

```

۲,۱۱,۲ توضیح

در خط ۳۵ تصویر ورودی را با استفاده از تابع cvtColor به فضای رنگی سیاه و سفید می‌بریم.

در خطوط ۴۱ و ۴۵ دو ترکبار برای دریافت نوع آستانه‌گذاری و مقدار آستانه از کاربر درست می‌کنیم. هر وقت کاربر مقدار یکی از ترکبارها را تغییر دهد، تابع Threshold_Demo صدا زده می‌شود.

در خط ۷۶ از تابع threshold استفاده شده است. این تابع دارای ۵ آرگومان است:

- src_gray: تصویر ورودی
- dst: تصویر خروجی
- threshold_value: مقدار آستانه است.
- max_BINARY_value: مقداری که در آستانه‌گذاری باینری استفاده می‌شود.
- threshold_type: نوع عمل آستانه‌گذاری است. در خطوط ۶۹ تا ۷۳ این نوع‌ها آورده شده‌اند.

۲,۱۱,۳ خروجی



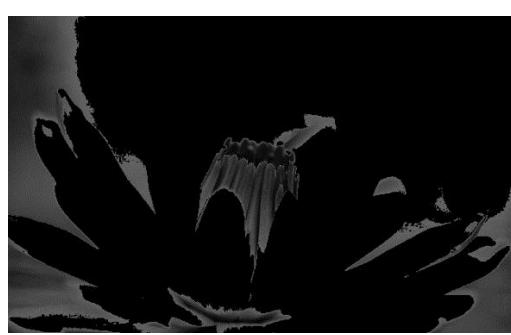
نتیجه آستانه‌گذاری صفر با آستانه ۱۰۰



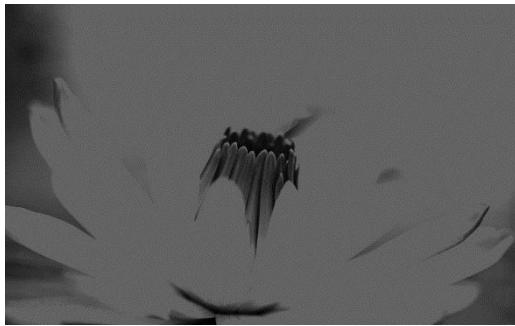
تصویر ورودی



نتیجه آستانه‌گذاری صفر با آستانه ۱۰۰



نتیجه آستانه‌گذاری صفر معکوس با آستانه ۱۰۰



نتیجه آستانه‌گذاری بریدن با آستانه ۱۰۰



نتیجه آستانه‌گذاری باینری معکوس با آستانه ۱۰۰

۲،۱۲ بافت‌نگار تصویر

در علم آمار هیستوگرام یا بافت‌نگار یک نمودار ستونی و پله‌ای برای نشان دادن داده‌ها است. نمودار بافت‌نگار همانند نمودار ستونی است و تنها اختلافی که بین این دو وجود دارد، نمایش ستون‌هاست.

در پردازش تصویر به نموداری که توسط آن تعداد پیکسل‌های هر سطح روشنایی در تصویر ورودی مشخص می‌شود بافت‌نگار تصویر می‌گویند. به عنوان مثال به شکل زیر توجه کنید:

234	143	203	176	109	229	177	220	192	9	229	142	130	64	0	63	26	8	89	82
27	68	231	75	141	107	149	210	13	239	141	35	68	242	110	208	244	0	33	88
54	42	17	215	239	254	47	41	86	180	55	253	235	47	122	208	78	110	152	100
9	186	102	71	104	193	89	171	37	233	18	147	174	1	143	211	176	188	192	68
179	20	238	192	190	132	41	248	22	134	83	133	110	254	176	238	186	234	51	204
232	25	0	183	174	129	61	30	110	189	0	173	197	183	153	43	22	87	68	118
235	35	151	185	129	81	239	170	195	94	38	21	67	101	58	37	196	149	52	154
155	242	54	0	104	109	189	47	130	254	225	156	31	181	121	15	126	35	252	205
223	114	79	128	147	6	201	65	89	107	58	44	253	84	38	1	62	5	231	218
55	188	237	188	80	101	131	241	68	193	124	151	111	28	190	4	240	78	117	145
152	155	229	78	90	217	219	105	118	77	38	49	2	9	214	181	205	118	135	33
182	94	176	199	20	149	57	223	232	113	32	45	177	15	31	179	100	119	208	81
224	118	124	172	75	29	69	180	187	195	41	44	8	170	158	101	131	31	28	112
238	83	38	7	83	69	173	183	98	237	87	227	18	218	248	237	75	192	201	146
88	195	224	207	140	23	31	118	234	34	182	116	23	47	68	242	189	152	116	248
140	37	101	230	246	145	122	64	27	58	229	1	225	143	91	100	98	90	40	195
231	4	178	139	121	95	97	174	249	182	77	115	223	188	182	65	232	83	196	
179	180	223	230	87	182	149	78	176	19	17	+	184	176	183	102	83	81	192	206
173	137	185	242	181	181	214	49	74	238	197	37	98	102	15	217	148	8	182	188
85	9	17	222	18	210	70	21	78	241	184	216	93	93	209	102	155	212	119	47

تصویری از مقادیر موجود در درایه‌های یک ماتریس

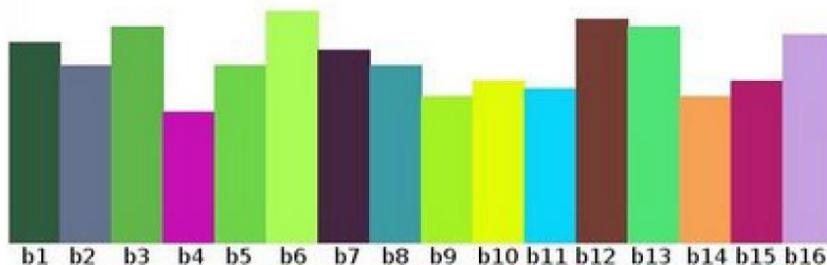
از آنجایی که دامنه مقادیر این داده‌ها معلوم است (بین ۰ تا ۲۵۵ است)، می‌توان آن دامنه را به زیر قسمت‌هایی که سطل^{۴۸} نام دارند، تقسیم کرد:

$$[0, 255] = [0, 15] \cup [16, 31] \cup \dots \cup [240, 255]$$

$$\text{domain} = \text{bin}_1 \cup \text{bin}_2 \cup \dots \cup \text{bin}_n = 15$$

بدین ترتیب می‌توان تعداد پیکسل‌هایی که در سطل bin_i قرار می‌گیرند را به دست آورد. با اجرای این روش روی ماتریس بالا، نمودار زیر را به دست می‌آید:

bin^{۴۸}



محور افقی نشان دهنده سطلهای و محور عمودی نشان دهنده تعداد پیکسلهای موجود در آن سطل است

این فقط یک مثال از طرز کار بافت‌نگار و همچنین دلیل مفید بودن آن بود. کاربرد بافت‌نگار فقط مختص شمارش روش‌نایی پیکسل‌های تصویر نیست، بلکه می‌توان از آن برای اندازه گیری ویژگی‌های دیگر تصویر هم استفاده کرد (مثل گرادیان، جهت‌ها و...).

یک بافت‌نگار از قسمت‌های زیر تشکیل شده است:

۱. ابعاد^{۴۹}: تعداد پارامترهایی که قرار است برای آنها داده جمع کنیم. در مثال ما این مقدار یک است؛ چون فقط مقادیر روش‌نایی هر پیکسل را شمارش می‌کنیم (در یک تصویر سیاه و سفید).
۲. تعداد سطلهای: تعداد زیر قسمت‌های موجود در هر بُعد است. در مثال ما این مقدار برابر با ۱۶ عدد است.
۳. بازه دامنه^{۵۰}: حدود مقادیر مورد اندازه گیری است. در مثال ما به صورت [0, 255] است.

در صورتی که دو ویژگی اندازه گیری شود، بافت‌نگار به دست آمده به صورت سه بعدی در می‌آید (محور X و محور Z نشان دهنده bin_x و محور Y نشان دهنده عدد شمارش شده برای هر ترکیب (bin_x, bin_y) است). برای ابعاد بیشتر به همین صورت محورهای بیشتری داریم.

۲,۱۲,۱ کد

```

1 #include "opencv2/highgui/highgui.hpp"
2 #include "opencv2/imgproc/imgproc.hpp"
3 #include <iostream>
4 #include <stdio.h>
5
6 using namespace std;
7 using namespace cv;
8
9 /**
10  * @function main
11 */
12 int main( int argc, char** argv )
13 {
14     Mat src, dst;
15
16     // Load image
17     src = imread( argv[1], 1 );
18
19     if( !src.data )
20     { return -1; }
21
22     // Separate the image in 3 places ( B, G and R )
23     vector<Mat> bgr_planes;
```

dims^{۴۹}
Domain^{۵۰}

vt

```
24     split( src, bgr_planes );
25
26     /// Establish the number of bins
27     int histSize = 256;
28
29     /// Set the ranges ( for B,G,R )
30     float range[] = { 0, 256 } ;
31     const float* histRange = { range };
32
33     bool uniform = true; bool accumulate = false;
34
35     Mat b_hist, g_hist, r_hist;
36
37     /// Compute the histograms:
38     calcHist( &bgr_planes[0], 1, 0, Mat(), b_hist, 1, &histSize,
39     &histRange, uniform, accumulate );
40     calcHist( &bgr_planes[1], 1, 0, Mat(), g_hist, 1, &histSize,
41     &histRange, uniform, accumulate );
42     calcHist( &bgr_planes[2], 1, 0, Mat(), r_hist, 1, &histSize,
43     &histRange, uniform, accumulate );
44
45     // Draw the histograms for B, G and R
46     int hist_w = 512; int hist_h = 400;
47     int bin_w = cvRound( (double) hist_w/histSize );
48
49     Mat histImage( hist_h, hist_w, CV_8UC3, Scalar( 0,0,0 ) );
50
51     /// Normalize the result to [ 0, histImage.rows ]
52     normalize(b_hist, b_hist, 0, histImage.rows, NORM_MINMAX, -1,
53     Mat() );
54     normalize(g_hist, g_hist, 0, histImage.rows, NORM_MINMAX, -1,
55     Mat() );
56     normalize(r_hist, r_hist, 0, histImage.rows, NORM_MINMAX, -1,
57     Mat() );
58
59     /// Draw for each channel
60     for( int i = 1; i < histSize; i++ )
61     {
62         line( histImage, Point( bin_w*(i-1), hist_h -
63         cvRound(b_hist.at<float>(i-1)) ),
64             Point( bin_w*(i), hist_h - cvRound(b_hist.at<float>(i)) ),
65         Scalar( 255, 0, 0 ), 2, 8, 0 );
66         line( histImage, Point( bin_w*(i-1), hist_h -
67         cvRound(g_hist.at<float>(i-1)) ),
68             Point( bin_w*(i), hist_h - cvRound(g_hist.at<float>(i)) ),
69         Scalar( 0, 255, 0 ), 2, 8, 0 );
70         line( histImage, Point( bin_w*(i-1), hist_h -
71         cvRound(r_hist.at<float>(i-1)) ),
72             Point( bin_w*(i), hist_h - cvRound(r_hist.at<float>(i)) ),
73         Scalar( 0, 0, 255 ), 2, 8, 0 );
74     }
75
76     /// Display
77     namedWindow("calcHist Demo", CV_WINDOW_AUTOSIZE );
78     imshow("calcHist Demo", histImage );
79
80     waitKey(0);
81
82
83
84
```

```

85     return 0;
86 }

```

۲،۱۲،۲ توضیح

در خط ۲۴ با استفاده از تابع `split` تصویر ورودی را به صفحات `R`, `G` و `B` تقسیم می‌کنیم. ورودی اول این تابع تصویری است که می‌خواهیم آن را تقسیم کنیم و ورودی دوم آن برداری است که صفحات (ماتریس‌های) خروجی در آن قرار می‌گیرند.

حالا همه چیز برای حساب کردن بافت‌نگارهای صفحات آماده است. از آنجایی که با صفحات `B`, `G` و `R` کار می‌کنیم، پس می‌دانیم که برد مقادیر در بازه $[0, 255]$ است.

در خط ۲۷ تعداد سطلهای مشخص می‌کنیم.

در خطوط ۳۰ و ۳۱ برد مقادیر را مشخص می‌کنیم (همانطور که گفته‌یم در بازه $[0, 255]$ است).

چون می‌خواهیم که سطلهای هم اندازه و بافت‌نگارها هم در ابتدای خالی باشند، در خط ۳۳ متغیر `uniform` را برابر `true` و متغیر `accumulate` را برابر `false` قرار می‌دهیم.

در ادامه برای حساب کردن بافت‌نگارها از تابع `calcHist` استفاده می‌کنیم:

در خطوط ۳۸ تا ۴۰ برای حساب کردن بافت‌نگار هر صفحه از تابع `calcHist` استفاده می‌کنیم. این تابع ۱۰ آرگومان دارد:

- .۱ **&bgr_planes[۰]:** صفحه (یا آرایه) ورودی است (که در این مورد صفحه \cdot است).
- .۲ **:** تعداد آرایه‌های ورودی است (در اینجا مقدارش ۱ است. می‌توان لیستی از آرایه‌ها به ورودی تابع داد و مقدار بیشتری را در اینجا مشخص کرد).
- .۳ **:** کانالی که باید اندازه گیری شود. در این مورد فقط روشنایی است (آرایه‌ها به صورت تک کاناله هستند)، پس باید \cdot گذاشت.
- .۴ **(Mat):** ماتریسی که به عنوان ماسک برای ماتریس ورودی استفاده می‌شود (مقدار صفر در ماسک بدین معنی است که پیکسل متناظر آن در ماتریس ورودی نباید در شمارش شرکت داشته باشد). اگر این ماتریس تعریف نشود، از ماسکی استفاده نخواهد شد.
- .۵ **b_hist:** بافت‌نگار خروجی در این ماتریس ذخیره می‌شود.
- .۶ **: ابعاد بافت‌نگار**
- .۷ **histSize:** تعداد سطلهای در هر کدام از بُعدها
- .۸ **: histRange**: برد مقادیر مورد اندازه گیری در هر بُعد
- .۹ **: uniform**: مشخص می‌کند که اندازه سطلهای برابر باشند یا نه.
- .۱۰ **: accumulate**: مشخص می‌کند که اندازه سطلهای در ابتدای خالی باشند یا نه.

در خطوط ۵۲ تا ۵۷، قبل از ترسیم بافت‌نگارها، مقادیر آنها را با استفاده از تابع `normalize` نرمال می‌کنیم تا بین دو مقدار تعیین شده قرار بگیرند. این تابع ۶ آرگومان دارد:

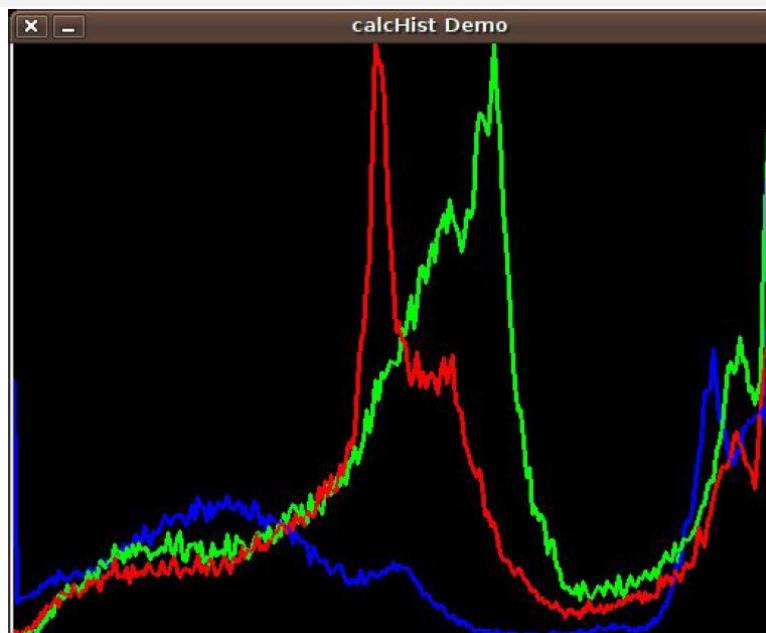
- .۱ **b_hist:** آرایه ورودی
- .۲ **b_hist:** آرایه خروجی
- .۳ **:** حد پایینی برای نرمال سازی مقادیر آرایه‌های ورودی

- .۴: حد بالایی برای نرمال سازی مقادیر آرایه‌های ورودی **histImage.rows**
- .۵: این آرگومان بیان کننده نوع فرایند نرمال سازی است (در این نوع، مقادیر بین دو مقدار **MIN** و **MAX** قرار می‌گیرند).
- .۶: ۱- نوع داده آرایه خروجی را مشخص می‌کند. در اینجا با قرار دادن ۱- نوع آرایه خروجی با نوع آرایه ورودی یکی است.
- .۷: ماسک آرایه ورودی **Mat()**

۲،۱۲،۳ خروجی



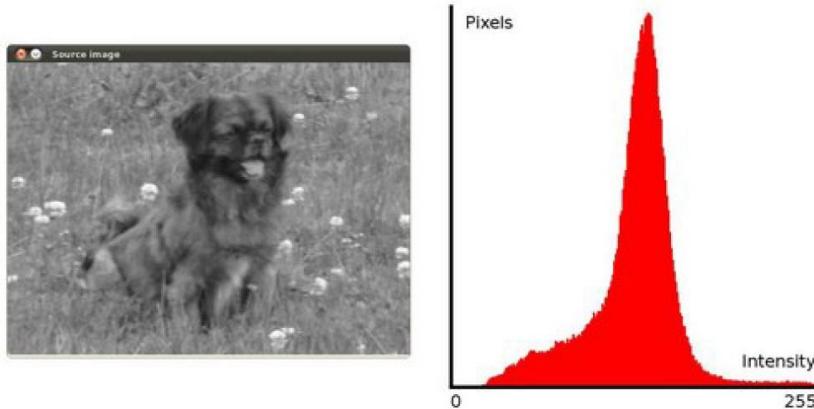
تصویر ورودی



بافت‌نگار هر کدام از کanal‌های تصویر ورودی

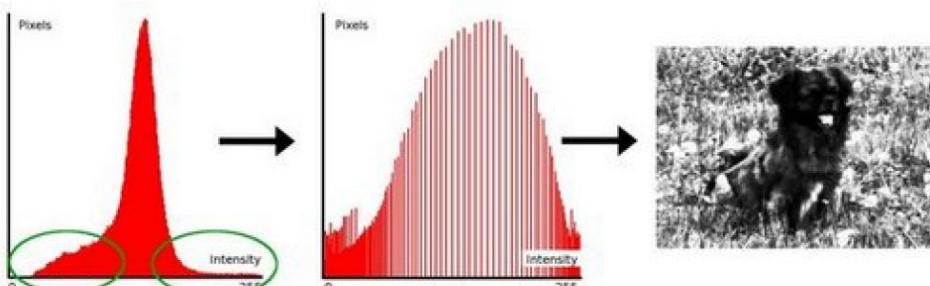
۲.۱۳ برابر سازی بافت نگار

همانطور که در بخش های قبل توضیح دادیم، بافت نگار تصویر یک نمایش گرافیکی از توزیع روشنایی آن تصویر است. این نمودار تعداد پیکسل هایی که یک مقدار روشنایی خاص را دارند، تعیین می کند.



یک تصویر سیاه و سفید و بافت نگار آن

برابر سازی بافت نگار روشی برای بهبود کنترast تصویر است. این کار از طریق وسیع کردن برد روشنایی انجام می شود. در شکل بالا می بینید که بیشتر پیکسل ها در دسته های میانی برد روشنایی قرار گرفته اند. کاری که برابر سازی بافت نگار انجام می دهد، وسیع کردن این برد است. به شکل زیر توجه کنید. در این شکل دایره های سبز رنگ نشان دهنده نواحی کم پوشش هستند (یعنی پیکسل های کمتری دارای این مقادیر از روشنایی هستند). بعد از اعمال برابر سازی، بافت نگاری شبیه شکل میانی به دست می آید. تصویر سمت راست نتیجه این برابر سازی است.



فرایند برابر سازی بافت نگار یک تصویر - تصویر سمت راست برابر سازی است

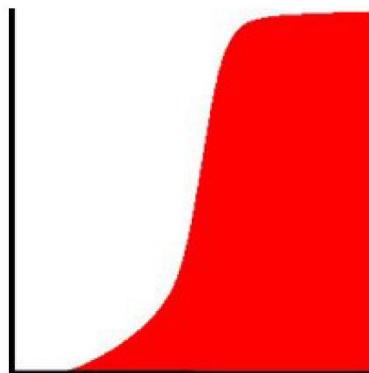
عملیات برابر سازی، یک توزیع را به یک توزیع وسیع تر و یکنواخت تر نگاشت می کند. پس مقادیر روشنایی، روی همه برد موجود توزیع می شوند.

برای رسیدن به اهداف برابر سازی، تابع مورد استفاده برای برابر سازی باید از نوع تابع توزیع تجمعی^{۵۱} (CDF) باشد. برای بافت نگار ($H(i)$) توزیع تجمعی آن، یعنی ($H'(i)$ ، به صورت زیر است:

$$H'(i) = \sum_{0 \leq j < i} H(j)$$

^{۵۱} Cumulative Distribution Function (CDF)

برای استفاده از این تابع به عنوان تابع نگاشت باید $(i) H'(i)$ را به گونه‌ای نرمال سازی کنیم که مقادیر آن بین ۰ تا ۲۵۵ قرار بگیرند. برای مثال بالا، تابع تجمعی به صورت زیر می‌شود:



تابع توزیع تجمعی بافت‌نگار تصویر سگ

در انتها برای به دست آوردن مقادیر روشنایی تصویر برابر سازی شده، از یک نگاشت ساده استفاده می‌کنیم:

$$\text{equalized}(x, y) = H'(src(x, y))$$

۲.۱۳.۱ کد

```

1 #include "opencv2/highgui/highgui.hpp"
2 #include "opencv2/imgproc/imgproc.hpp"
3 #include <iostream>
4 #include <stdio.h>
5
6 using namespace cv;
7 using namespace std;
8
9 /** @function main */
10 int main( int argc, char** argv )
11 {
12     Mat src, dst;
13
14     char* source_window = "Source image";
15     char* equalized_window = "Equalized Image";
16
17     /// Load image
18     src = imread( argv[1], 1 );
19
20     if( !src.data )
21     { cout<<"Usage: ./Histogram_Demo <path_to_image>"<<endl;
22       return -1; }
23
24     /// Convert to grayscale
25     cvtColor( src, src, CV_BGR2GRAY );
26
27     /// Apply Histogram Equalization
28     equalizeHist( src, dst );
29
30     /// Display results
31     namedWindow( source_window, CV_WINDOW_AUTOSIZE );
32     namedWindow( equalized_window, CV_WINDOW_AUTOSIZE );
33
34     imshow( source_window, src );

```

```

35     imshow( equalized_window, dst );
36
37     // Wait until user exits the program
38     waitKey(0);
39
40     return 0;
41 }

```

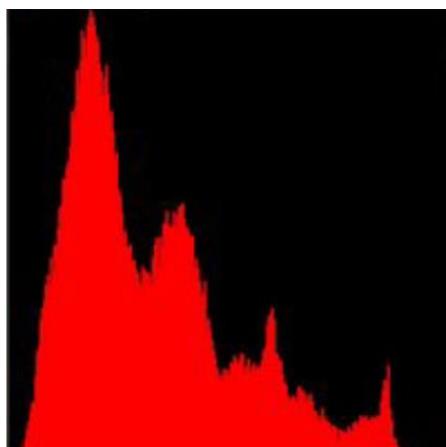
۲،۱۳،۲ توضیح

در خط ۲۵ تصویر ورودی را به یک تصویر سیاه و سفید تبدیل می‌کنیم.

در خط ۲۸ با استفاده از تابع `equalizeHist` عملیات برابرسازی بافت‌نگار را انجام می‌دهیم. این تابع دو آرگومان دارد که اولی تصویر ورودی و دومی تصویر خروجی است.

۲،۱۳،۳ خروجی

از تصویر زیر که کنتراست کمی دارد به عنوان تصویر ورودی استفاده می‌کنیم:

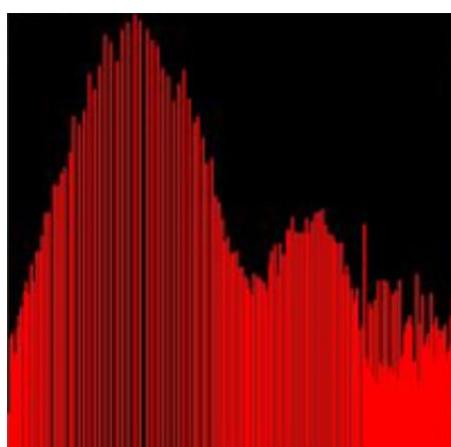


بافت‌نگار تصویر ورودی



تصویر ورودی

بعد از اعمال برابرسازی نتیجه به صورت زیر می‌شود:



بافت‌نگار تصویر خروجی



تصویر خروجی

۲،۱۴ عملگر سابل

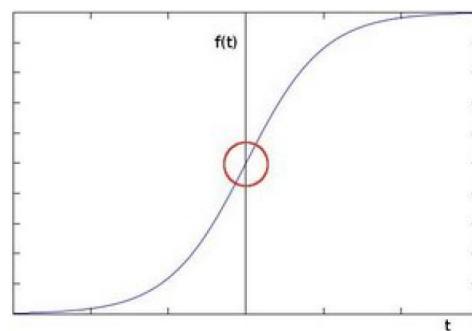
پیدا کردن مشتق‌های یک تصویر در کاربردهایی مثل یافتن لبه‌های موجود در تصویر مفید است.



قسمت مشخص شده با دایره قرمز یک لبه است

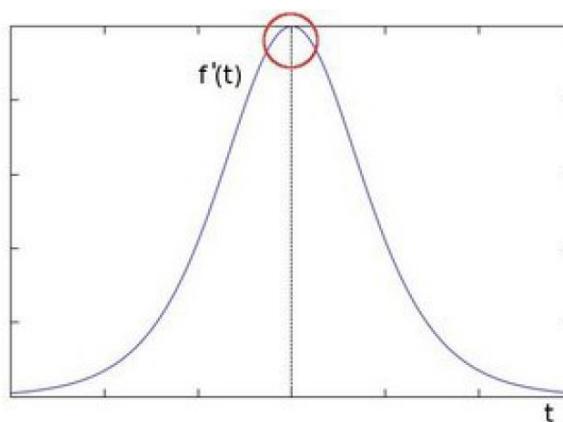
به راحتی می‌توان دید که در یک لبه روشنایی پیکسل‌ها به شدت تغییر می‌کند. یکی از روش‌های توصیف تغییرات، مشتق‌ها هستند. یک تغییر زیاد در شبیب، نماینگر یک تغییر قابل توجه در روشنایی تصویر است.

برای اینکه قضیه واضح‌تر شود، فرض کنید یک تصویر تک بعدی به صورت زیر داریم. محلی که با دایره قرمز مشخص شده است یک لبه است. این نقطه جایی است که تغییر قابل توجهی در شدت روشنایی دارد.



یک تصویر یک بعدی- ناحیه مشخص شده با دایره قرمز یک لبه است

می‌توان محل لبه‌ها را به راحتی با گرفتن مشتق اول پیدا کرد (در اینجا به شکل یک نقطه بیشینه در آمده است).



مشتق تصویر تک بعدی قبل

از توضیحات بالا نتیجه می‌گیریم که یک راه برای پیدا کردن لبه‌های تصویر این است که به دنبال پیکسل‌هایی بگردیم که گرادیانشان بیشتر از گرادیان همسایه‌هایشان (یا به طور کلی بیشتر از یک آستانه) است. در ادامه نحوه به دست آوردن گرادیان تصویر را به کمک عملگر سابل بررسی می‌کنیم.

عملگر سابل:

عملگر سابل^{۵۲} یک عملگر دیفرانسیلی گستته است. این عملگر به طور تقریبی گرادیان تابع روشنایی عکس را حساب می‌کند.

عملگر سابل، دو عمل هموار سازی گوسی و دیفرانسیل گیری را با هم ترکیب می‌کند.

با فرض اینکه عکس مورد نظر ا باشد، روند کار این عملگر به صورت زیر است:

۱. دو مشتق زیر را حساب می‌کند:

۱. مشتق افقی: از طریق کانوالو G_x با کرنلی با اندازه ۳، G_x (مشتق در راستای افقی) به شکل زیر به دست می‌آید:

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * I$$

۲. مشتق عمودی: از طریق کانوالو G_y با کرنلی با اندازه ۳، G_y (مشتق در راستای عمودی) به شکل زیر به دست می‌آید:

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * I$$

۲. برای به دست آوردن گرادیان پیکسل‌های تصویر باید G_x و G_y را به شکل زیر با هم ترکیب کرد:

$$G = \sqrt{G_x^2 + G_y^2}$$

البته بعضی اوقات از معادله ساده‌تر زیر هم استفاده می‌شود:

$$G = |G_x| + |G_y|$$

نکته: وقتی اندازه کرنل ۳ است، کرنل سابلی که در قسمت قبل نشان داده شد می‌تواند به صورت قابل توجه ای نتایج نادرست تولید کند (به هر حال سابل فقط یک تقریب برای مشتقهای است). در این میان حل این مشکل تابع **Scharr** ارائه شده که به اندازه سابل سریع است و البته در عین حال دقیق بسیار بیشتر از آن دارد. تابع **Scharr** از کرنل‌های زیر استفاده می‌کند:

$$G_x = \begin{bmatrix} -3 & 0 & +3 \\ -10 & 0 & +10 \\ -3 & 0 & +3 \end{bmatrix}$$

$$G_y = \begin{bmatrix} -3 & -10 & -3 \\ 0 & 0 & 0 \\ +3 & +10 & +3 \end{bmatrix}$$

۲,۱۴,۱

```
1 #include "opencv2/imgproc/imgproc.hpp"
```

八

```
2 #include "opencv2/highgui/highgui.hpp"
3 #include <stdlib.h>
4 #include <stdio.h>
5
6 using namespace cv;
7
8 /** @function main */
9 int main( int argc, char** argv )
10 {
11
12     Mat src, src_gray;
13     Mat grad;
14     char* window_name = "Sobel Demo - Simple Edge Detector";
15     int scale = 1;
16     int delta = 0;
17     int ddepth = CV_16S;
18
19     int c;
20
21     /// Load an image
22     src = imread( argv[1] );
23
24     if( !src.data )
25     { return -1; }
26
27     GaussianBlur( src, src, Size(3,3), 0, 0, BORDER_DEFAULT );
28
29     /// Convert it to gray
30     cvtColor( src, src_gray, CV_RGB2GRAY );
31
32     /// Create window
33     namedWindow( window_name, CV_WINDOW_AUTOSIZE );
34
35     /// Generate grad_x and grad_y
36     Mat grad_x, grad_y;
37     Mat abs_grad_x, abs_grad_y;
38
39     /// Gradient X
40     //Schar( src_gray, grad_x, ddepth, 1, 0, scale, delta,
41     BORDER_DEFAULT );
42     Sobel( src_gray, grad_x, ddepth, 1, 0, 3, scale, delta,
43     BORDER_DEFAULT );
44     convertScaleAbs( grad_x, abs_grad_x );
45
46     /// Gradient Y
47     //Schar( src_gray, grad_y, ddepth, 0, 1, scale, delta,
48     BORDER_DEFAULT );
49     Sobel( src_gray, grad_y, ddepth, 0, 1, 3, scale, delta,
50     BORDER_DEFAULT );
51     convertScaleAbs( grad_y, abs_grad_y );
52
53     /// Total Gradient (approximate)
54     addWeighted( abs_grad_x, 0.5, abs_grad_y, 0.5, 0, grad );
55
56     imshow( window_name, grad );
57
58     waitKey(0);
59
60     return 0;
61 }
```

۲،۱۴،۲ توضیح

در خط ۲۷ برای کاهش نویز یک هموار سازی گوسی به تصویر ورودی اعمال می‌کنیم (با اندازه کرنل ۳). در خطوط ۴۲ و ۴۹ به ترتیب مشتقات در جهت‌های x و y را حساب می‌کنیم. برای این کار از تابع Sobel استفاده می‌کنیم. این تابع ۹ آرگومان به شرح زیر دارد:

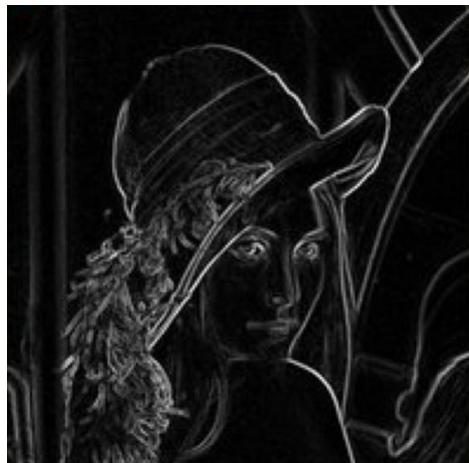
- ۱: **src_gray**: تصویر ورودی است. در اینجا از نوع CV_8U است.
- ۲: **grad_x/grad_y**: تصویر خروجی
- ۳: **ddepth**: عمق تصویر خروجی است. برای جلوگیری از سرریز از CV_16S استفاده می‌کنیم.
- ۴: **x_order**: مرتبه مشتق در جهت x است.
- ۵: **y_order**: مرتبه مشتق در جهت y است.
- ۶: **BORDER_DEFAULT** و **delta**, **scale**: از مقادیر پیش‌فرض استفاده می‌کنیم.

توجه کنید که برای به دست آوردن گرادیان در جهت x باید $x_{order} = 1$ و $y_{order} = 0$ قرار دهیم. بر عکس این کار را برای جهت y انجام می‌دهیم.

در خطوط ۴۴ و ۵۱، نتایج جزئی را به نوع CV_8U تبدیل می‌کنیم.

سرانجام در خط ۵۴ برای تخمین گرادیان کلی، گرادیان در جهت‌های x و y را با هم جمع می‌کنیم (توجه کنید که این کار اصلاً یک محاسبه دقیق نیست!! ولی کار ما را راه می‌اندازد).

۲،۱۴،۳ خروجی



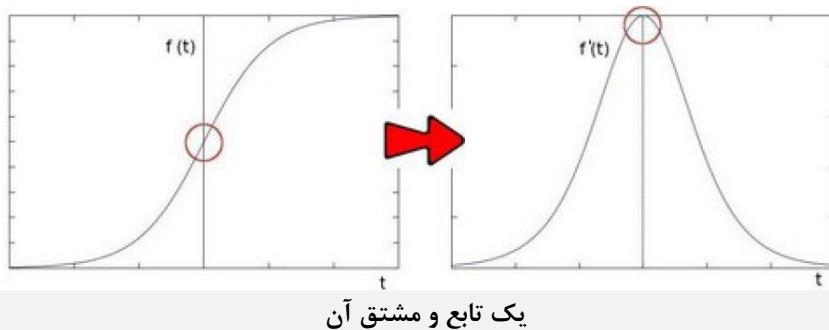
تصویر خروجی



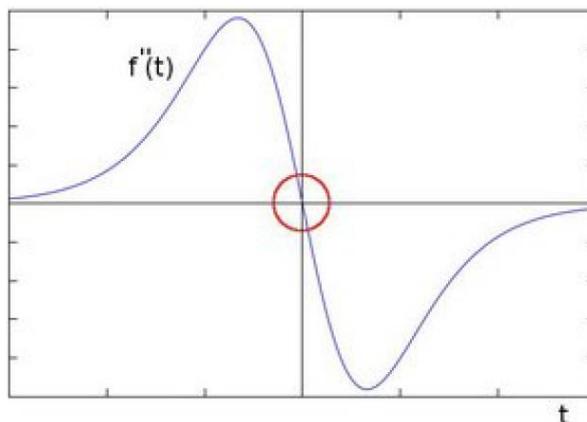
تصویر ورودی

۲،۱۵ عملگر لاپلاس

در بخش قبل با نحوه استفاده از عملگر سابل آشنا شدیم. این عملگر بر مبنای تغییرات روشنایی پیکسل‌های عکس بود. با گرفتن مشتق اول از تابع روشنایی دیدیم که می‌توان لبه را به عنوان یک بیشینه در نظر گرفت. مثلاً مشتق اول یک تابع می‌تواند به شکل زیر باشد:



و وقتی مشتق دوم گرفته شود به شکل زیر خواهد بود:



همانطور که می‌بینید، محل مورد نظر در مشتق دوم برابر صفر است! بنابراین می‌توان از این قضیه برای پیدا کردن لبه‌های یک تصویر استفاده کرد. توجه کنید که نقطه‌های صفر فقط در لبه‌ها رخ نمی‌دهند (می‌توانند در نقطه‌هایی بی معنی نیز رخ دهند)؛ البته می‌توان این مشکل را تا حدودی با فیلتر کردن نقاط برطرف کرد. در ادامه با عملگر لاپلاس آشنا می‌شویم.

عملگر لاپلاس:

از توضیحات بالا بر می‌آید که می‌توان از مشتق دوم برای پیدا کردن لبه‌ها استفاده کرد و از آنجایی که تصاویر دو بعدی هستند، باید از هر دو بعد مشتق بگیریم. اینجاست که عملگر لاپلاس به کار می‌آید و به صورت زیر تعریف می‌شود:

$$\text{Laplace}(f) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

در اسی وی تابعی به نام Laplacian وجود دارد که معادله بالا را پیاده سازی کرده است. در واقع از آنجایی که لاپلاس از گرادیان‌های تصویر استفاده می‌کند، این تابع به صورت داخلی از عملگر سابل برای محاسباتش استفاده می‌کند.

۲,۱۵,۱

```

1 #include "opencv2/imgproc/imgproc.hpp"
2 #include "opencv2/highgui/highgui.hpp"
3 #include <stdlib.h>
4 #include <stdio.h>
5
6 using namespace cv;
7

```

```

8  /** @function main */
9  int main( int argc, char** argv )
10 {
11     Mat src, src_gray, dst;
12     int kernel_size = 3;
13     int scale = 1;
14     int delta = 0;
15     int ddepth = CV_16S;
16     char* window_name = "Laplace Demo";
17
18     int c;
19
20     /// Load an image
21     src = imread( argv[1] );
22
23     if( !src.data )
24     { return -1; }
25
26     /// Remove noise by blurring with a Gaussian filter
27     GaussianBlur( src, src, Size(3,3), 0, 0, BORDER_DEFAULT );
28
29     /// Convert the image to grayscale
30     cvtColor( src, src_gray, CV_RGB2GRAY );
31
32     /// Create window
33     namedWindow( window_name, CV_WINDOW_AUTOSIZE );
34
35     /// Apply Laplace function
36     Mat abs_dst;
37
38     Laplacian( src_gray, dst, ddepth, kernel_size, scale, delta,
39     BORDER_DEFAULT );
40     convertScaleAbs( dst, abs_dst );
41
42     /// Show what you got
43     imshow( window_name, abs_dst );
44
45     waitKey(0);
46
47     return 0;
48 }

```

۲،۱۵،۲ توضیح

در خط ۲۷ به منظور کاهش نویز از فیلتر هموار ساز گوسی استفاده می کنیم.

در خط ۳۸ عملگر لاپلاس را با استفاده از تابع **Laplacian** به تصویر ورودی اعمال می کنیم. این تابع ۷ آرگومان به شرح زیر دارد:

- .۱ **src_gray**: تصویر ورودی (که در این مورد سیاه و سفید است. می تواند رنگی نیز باشد).
- .۲ **dst**: تصویر خروجی
- .۳ **ddepth**: عمق تصویر خروجی است. از آنجا که در اینجا تصویر ورودی از نوع **CV_8U** است، برای جلوگیری از سرریز، عمق تصویر خروجی را **CV_16S** می گذاریم.
- .۴ **kernel_size**: اندازه کرنل عملگر سابل که داخل تابع **Laplacian** استفاده می شود. در اینجا از اندازه ۳ استفاده می کنیم.
- .۵ **BORDER_DEFAULT** و **delta**، **scale**: این ها را به صورت پیشفرض رها می کنیم.

در خط ۴۰ خروجی تابع Laplacian را به نوع CV_8U تبدیل می‌کنیم.

۲,۱۵,۳ خروجی



درختها و نیمرخ گاو خوب در آمدیده‌اند؛ البته ناحیه‌هایی که دارای روشنایی یکسان هستند، مثل اطراف سر گاو، خیلی خوب نشده‌اند. همانطور که می‌بینید به دلیل کتراست زیاد ناحیه سقف خانه پشت گاو، لبه‌ها خیلی واضح مشخص شده‌اند.

۲,۱۶ عملگر کَنِی

لبه‌یاب کَنِی در سال ۱۹۸۶ توسط جان اف. کَنِی^{۵۳} توسعه یافت. همچنین خیلی‌ها آن را با نام یابنده بهینه^{۵۴} می‌شناسند. الگوریتم کَنِی سه هدف اساسی زیر را دنبال می‌کند:

۱. نرخ خطای پایین: یعنی اینکه فقط لبه‌های واقعی را کشف کند (نه نقاطی که لبه نیستند).
۲. مکان‌یابی خوب: فاصله میان پیکسل‌های کشف شده لبه و پیکسل‌های واقعی لبه باید کمینه باشد.
۳. پاسخ کمینه: به ازای هر لبه فقط یک پاسخ وجود داشته باشد.

رویه حساب لبه‌ها در لبه‌یاب کَنِی به صورت زیر است:

۱. نویزها را فیلتر می‌کند. بدین منظور از فیلتر گوسی استفاده می‌شود. به عنوان مثال می‌توان از کرنل زیر استفاده کرد:

$$K = \frac{1}{159} \begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}$$

۲. گرادیان تابع روشنایی تصویر را به دست می‌آورد. بدین منظور از روشی مشابه با عملگر سابل استفاده می‌شود:

- ا. دو کرنل زیر را در تصویر کانوال می‌کند (یکی برای جهت x و یکی برای جهت y):

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$$

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

ii. از روش زیر، دامنه و جهت آن را پیدا می‌کند:

$$G = \sqrt{G_x^2 + G_y^2}$$

$$\theta = \arctan\left(\frac{G_y}{G_x}\right)$$

جهت‌ها در یکی از چهار گروه ۰، ۴۵، ۹۰ یا ۱۳۵ قرار می‌گیرند.

۳. فیلتری جهت حذف نقاط غیر بیشینه اعمال می‌کند. با این کار پیکسل‌هایی که جزوی از لبه نیستند، حذف می‌شوند. بنابراین فقط لبه‌های نازک باقی می‌مانند.

۴. در قدم آخر کنی از یک روش آستانه‌گذاری با دو آستانه بالا و پایین استفاده می‌کند:

i. اگر مقدار پیکسلی از آستانه بالایی بیشتر باشد، به عنوان لبه پذیرفته می‌شود.

ii. اگر مقدار پیکسلی از آستانه پایینی کمتر باشد، رد می‌شود.

iii. اگر مقدار پیکسلی بین آستانه بالایی و آستانه پایینی بود، فقط در صورتی که عنوان لبه پذیرفته می‌شود که همسایه پیکسلی باشد که مقدارش بالاتر از آستانه بالایی است.

پیشنهاد می‌شود که نسبت آستانه بالایی: پایینی یکی از دو مقدار ۱:۲ یا ۱:۳ باشد.

۲.۱۶.۱ کد

```

1 #include "opencv2/imgproc/imgproc.hpp"
2 #include "opencv2/highgui/highgui.hpp"
3 #include <stdlib.h>
4 #include <stdio.h>
5
6 using namespace cv;
7
8 /// Global variables
9
10 Mat src, src_gray;
11 Mat dst, detected_edges;
12
13 int edgeThresh = 1;
14 int lowThreshold;
15 int const max_lowThreshold = 100;
16 int ratio = 3;
17 int kernel_size = 3;
18 char* window_name = "Edge Map";
19
20 /**
21 * @function CannyThreshold
22 * @brief Trackbar callback - Canny thresholds input with a ratio 1:3
23 */
24 void CannyThreshold(int, void*)
25 {
26     /// Reduce noise with a kernel 3x3
27     blur( src_gray, detected_edges, Size(3,3) );
28
29     /// Canny detector
30     Canny( detected_edges, detected_edges, lowThreshold,
31 lowThreshold*ratio, kernel_size );
32
33     /// Using Canny's output as a mask, we display our result

```

```

34     dst = Scalar::all(0);
35
36     src.copyTo( dst, detected_edges );
37     imshow( window_name, dst );
38 }
39
40
41 /** @function main */
42 int main( int argc, char** argv )
43 {
44     /// Load an image
45     src = imread( argv[1] );
46
47     if( !src.data )
48     { return -1; }
49
50     /// Create a matrix of the same type and size as src (for dst)
51     dst.create( src.size(), src.type() );
52
53     /// Convert the image to grayscale
54     cvtColor( src, src_gray, CV_BGR2GRAY );
55
56     /// Create a window
57     namedWindow( window_name, CV_WINDOW_AUTOSIZE );
58
59     /// Create a Trackbar for user to enter threshold
60     createTrackbar( "Min Threshold:", window_name, &lowThreshold,
61     max_lowThreshold, CannyThreshold );
62
63     /// Show the image
64     CannyThreshold(0, 0);
65
66     /// Wait until user exit program by pressing a key
67     waitKey(0);
68
69     return 0;
70 }
71
72
73

```

۲،۱۶،۲ توضیح

به نکات زیر توجه کنید:

- .۱ در خط ۱۶ نسبت آستانه پایین: بالا را ۳:۱ انتخاب کرده‌ایم.
- .۲ در خط ۱۷ اندازه کرنل را ۳ می‌گذاریم (این کرنل برای عملیات سابلی است که در تابع Canny استفاده می‌شود).
- .۳ در خط ۱۵ بیشنه مقدار مجاز برای آستانه پایینی را ۱۰۰ می‌گذاریم.

در تابع CannyThreshold در خط ۲۷، برای کاهش نویز تصویر ورودی را هموار می‌کنیم.

سپس در خط ۳۰ تابع Canny را صدا می‌زنیم. این تابع ۵ آرگومان به شرح زیر دارد:

- .۱ **detected_edges**: تصویر ورودی است. این تصویر باید سیاه و سفید باشد.
- .۲ **detected_edges**: تصویر خروجی است. می‌توان همان تصویر ورودی را به عنوان خروجی در نظر گرفت.
- .۳ **lowThreshold**: مقداری که کاربر به وسیله ترکبار مشخص کرده است.

.۴: مقدار این متغیر سه برابر آستانه پایینی است (یعنی سه برابر `lowThreshold`).

.۵: مقدارش را ۳ می‌گذاریم (این اندازه کرنل Sobel ای است که در تابع Canny استفاده می‌شود).

در نهایت در خط ۳۶ از تابع `copyTo` برای نگاشت نواحی ای که به عنوان لبه کشف شده‌اند به یک پس زمینه سیاه استفاده می‌کنیم. لازم به ذکر است که این تابع فقط مقادیری از عکس منبع را کپی می‌کند که صفر نباشد. از آنجایی که خروجی Canny به صورت کانتورهای لبه روی یک زمینه سیاه است، تمام نواحی `dst` به جز نواحی ای که لبه‌های کشف شده قرار داشته باشند، سیاه خواهند بود.

۲,۱۶,۳ خروجی



تصویر خروجی با آستانه پایینی ۱۰۰



تصویر ورودی

۲,۱۷ کاپچرهای تصویر

۲,۱۷,۱

```
1 #include "opencv2/highgui/highgui.hpp"
2 #include "opencv2/imgproc/imgproc.hpp"
3 #include <iostream>
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 using namespace cv;
8 using namespace std;
9
10 Mat src; Mat src_gray;
11 int thresh = 100;
12 int max_thresh = 255;
13 RNG rng(12345);
14
15 /// Function header
16 void thresh_callback(int, void* );
17
18 /** @function main */
19 int main( int argc, char** argv )
20 {
21     /// Load source image and convert it to gray
22     src = imread( argv[1], 1 );
23
24     /// Convert image to gray and blur it
25     cvtColor( src, src_gray, CV_BGR2GRAY );
26     blur( src_gray, src_gray, Size(3,3) );
27 }
```

```

28     /// Create Window
29     char* source_window = "Source";
30     namedWindow( source_window, CV_WINDOW_AUTOSIZE );
31     imshow( source_window, src );
32
33     createTrackbar( " Canny thresh:", "Source", &thresh, max_thresh,
34 thresh_callback );
35     thresh_callback( 0, 0 );
36
37     waitKey(0);
38     return(0);
39 }
40
41 /** @function thresh_callback */
42 void thresh_callback(int, void* )
43 {
44     Mat canny_output;
45     vector<vector<Point>> contours;
46     vector<Vec4i> hierarchy;
47
48     /// Detect edges using canny
49     Canny( src_gray, canny_output, thresh, thresh*2, 3 );
50     /// Find contours
51     findContours( canny_output, contours, hierarchy, CV_RETR_TREE,
52 CV_CHAIN_APPROX_SIMPLE, Point(0, 0) );
53
54     /// Draw contours
55     Mat drawing = Mat::zeros( canny_output.size(), CV_8UC3 );
56     for( int i = 0; i< contours.size(); i++ )
57     {
58         Scalar color = Scalar( rng.uniform(0, 255),
59 rng.uniform(0,255), rng.uniform(0,255) );
60         drawContours( drawing, contours, i, color, 2, 8, hierarchy,
61 0, Point() );
62     }
63
64     /// Show in a window
65     namedWindow( "Contours", CV_WINDOW_AUTOSIZE );
66     imshow( "Contours", drawing );
67
68 }

```

۲،۱۷،۲ توضیح

به منظور کاهش نویز، در خط ۲۶ تصویر سیاه و سفید را با فیلتر ۳ در ۳ هموار می‌کنیم.

در خط ۴۲ تابع `thresh_callback` تعریف شده است. در این تابع ابتدا یک ماتریس برای نگه داری خروجی لبیاب گنجینه درست می‌کنیم. همچنین یک بردار دو بعدی از نوع `Point` برای نگه داری کانتورهای به دست آمده و یک بردار تک بعدی از نوع `Vec4i` برای نگه داری خروجی اختیاری تابع `findContours`، یعنی `hierarchy`، تعریف می‌کنیم.

نکته: یک بردار ۱ در ۴ است که همه عناصر آن از نوع `int` هستند.

در خط ۴۹ با استفاده از تابع `Canny` لبه‌های تصویر سیاه و سفید را پیدا می‌کنیم. از آستانه‌ای استفاده می‌کنیم که کاربر وارد کرده است (یعنی متغیر `.thresh`)

سپس در خط ۵۱ با فراخوانی تابع `findContours`، کانتورهای تصویر خروجی از تابع `Canny` (یعنی `canny_output`) را پیدا می‌کنیم. این تابع ۶ آرگومان به شرح زیر دارد:

۱. **canny_output**: تصویر ورودی است. این تصویر باید از نوع `CV_8UC1` باشد. در اینجا از خروجی تابع `Canny` به عنوان `canny_output` ورودی این تابع استفاده می‌کنیم. از آنجا که خروجی تابع `Canny` یک عکس تک بعدی ۸ بیتی است پس مشکلی وجود ندارد.

۲. **contours**: کانتورهای کشف شده هستند. هر کانتور به شکل برداری از نقاط در این متغیر ذخیره می‌شود.

۳. **hierarchy**: بردار خروجی اختیاری که اطلاعاتی در مورد توپولوژی تصویر به ما می‌دهد.

۴. **CV_RET_TREE**: شیوه بازیابی کانتورها است. می‌تواند به جای این مقدار، هر کدام از مقادیر زیر باشد:

<code>CV_RETR_EXTERNAL</code>	▪
<code>CV_RETR_LIST</code>	▪
<code>CV_RETR_CCOMP</code>	▪
<code>CV_RET_TREE</code>	▪

۵. **CV_CHAIN_APPROX_SIMPLE**: شیوه تقریب کانتورها است. می‌تواند به جای این مقدار، هر کدام از مقادیر زیر باشد:

<code>CV_CHAIN_APPROX_NONE</code>	۱
<code>CV_CHAIN_APPROX_SIMPLE</code>	۲
<code>CV_CHAIN_APPROX_TC89_KCOS</code> و <code>CV_CHAIN_APPROX_TC89_L1</code>	۳

۶. **Point(0,0)**: یک آفست اختیاری که هر کدام از کانتورها با آن جمع می‌شوند. در اینجا نقطه `(0,0)` را به عنوان آفست در نظر گرفته‌ایم. آفست زمانی مفید است که کانتورها از روی یک `ROI` به دست آمده باشند و قرار باشد بعداً کانتورها را روی عکس اصلی نشان دهیم.

بعد از پیدا کردن کانتورها، هم می‌توانیم آنها را به صورت یکجا (و با یک رنگ) بکشیم و هم اینکه هر کدام از کانتورها را با یک رنگ خاص نمایش دهیم، به هر حال در هر دو حالت از تابع `drawContours` استفاده می‌کنیم. در حالت اول با یک بار فراخوانی تابع `drawContours` و در حالت دوم، تابع `drawContours` را در یک حلقه می‌گذاریم و به تعداد کانتورهای کشف شده حلقه را تکرار می‌کنیم و هر دفعه یکی از کانتورها را می‌کشیم. در اینجا به روش دوم عمل می‌کنیم. تابع `drawContours` در خط ۶۰ دارای ۹ آرگومان به شرح زیر است:

۱. **drawing**: تصویر ورودی است.

۲. **contours**: بردار کانتورها است. هر کانتور به صورت برداری از نقاط در `contours` قرار می‌گیرند.

۳. **i**: شماره کانتوری که باید کشیده شود. اگر یک عدد منفی در اینجا قرار دهیم، همه کانتورها کشیده می‌شوند.

۴. **color**: رنگ کانتور است.

۵. **۲**: نازکی کانتور است.

۶. **۸**: نوع خطی که بین نقاط کانتور کشیده می‌شود.

۷. **hierarchy**: اطلاعات اختیاری در مورد سلسله مراتب عکس است. فقط زمانی نیاز است که بخواهیم بعضی از کانتورها را بکشیم (مثلاً در اینجا که هر دفعه فقط یکی از کانتورها را می‌کشیم، این متغیر نیاز است).

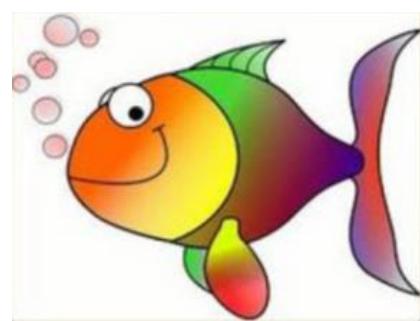
۸. **۰**: حداکثر سطح برای کانتورهای کشیده شده است. اگر صفر باشد فقط کانتور مشخص شده کشیده می‌شود. اگر یک باشد کانتور مشخص شده و تمام کانتورهای تو در توى آن را می‌کشد.

۹. **Point()**: آفست اختیاری‌ای است که مختصات کانتورها ابتدا با آن جمع و سپس روی عکس کشیده می‌شوند.

۲,۱۷,۳ خروجی



تصویر خروجی



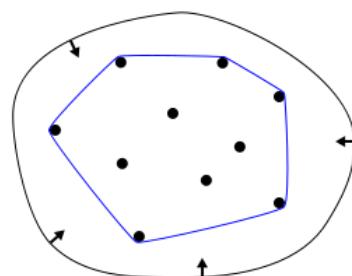
تصویر ورودی

دقت کنید که رنگ کانتورها به صورت تصادفی انتخاب شده‌اند و هیچ ربطی به رنگ‌های تصویر ورودی ندارند.

۲,۱۸ پوش محدب

پوش محدبِ مجموعه‌ای از نقاط صفحه، کوچک‌ترین چند ضلعی محدبی است که تمامی نقاط درون آن یا روی محیط آن قرار داشته باشند.

برای این که تصویر بهتری از پوش محدب به دست آورید، نقاط صفحه را مانند میخ‌هایی در نظر بگیرید که به دیوار کوبیده شده‌اند. حال کش تنگی را در نظر بگیرید که همه میخ‌ها را احاطه کرده است. در این صورت پوش محدب نقاط شکلی خواهد بود که کش به خود می‌گیرد.



نحوه پیدا کردن پوش محدب مجموعه‌ای از نقاط

۲,۱۸,۱ کد

```

1 #include "opencv2/highgui/highgui.hpp"
2 #include "opencv2/imgproc/imgproc.hpp"
3 #include <iostream>
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 using namespace cv;
8 using namespace std;
9
10 Mat src; Mat src_gray;
11 int thresh = 100;
12 int max_thresh = 255;
13 RNG rng(12345);
14

```

```

15  /// Function header
16  void thresh_callback(int, void* );
17
18  /** @function main */
19  int main( int argc, char** argv )
20  {
21      // Load source image and convert it to gray
22      src = imread( argv[1], 1 );
23
24      // Convert image to gray and blur it
25      cvtColor( src, src_gray, CV_BGR2GRAY );
26      blur( src_gray, src_gray, Size(3,3) );
27
28      // Create Window
29      char* source_window = "Source";
30      namedWindow( source_window, CV_WINDOW_AUTOSIZE );
31      imshow( source_window, src );
32
33      createTrackbar( " Threshold:", "Source", &thresh, max_thresh,
34      thresh_callback );
35      thresh_callback( 0, 0 );
36
37      waitKey(0);
38      return(0);
39  }
40
41  /** @function thresh_callback */
42  void thresh_callback(int, void* )
43  {
44      Mat src_copy = src.clone();
45      Mat threshold_output;
46      vector<vector<Point>> contours;
47      vector<Vec4i> hierarchy;
48
49      // Detect edges using Threshold
50      threshold( src_gray, threshold_output, thresh, 255, THRESH_BINARY
51 );
52
53      // Find contours
54      findContours( threshold_output, contours, hierarchy,
55      CV_RETR_TREE, CV_CHAIN_APPROX_SIMPLE, Point(0, 0) );
56
57      // Find the convex hull object for each contour
58      vector<vector<Point>> hull( contours.size() );
59      for( int i = 0; i < contours.size(); i++ )
60      { convexHull( Mat(contours[i]), hull[i], false ); }
61
62      // Draw contours + hull results
63      Mat drawing = Mat::zeros( threshold_output.size(), CV_8UC3 );
64      for( int i = 0; i < contours.size(); i++ )
65      {
66          Scalar color = Scalar( rng.uniform(0, 255),
67          rng.uniform(0,255), rng.uniform(0,255) );
68          drawContours( drawing, contours, i, color, 1, 8,
69          vector<Vec4i>(), 0, Point() );
70          drawContours( drawing, hull, i, color, 1, 8, vector<Vec4i>(),
71          0, Point() );
72      }
73
74      // Show in a window
75      namedWindow( "Hull demo", CV_WINDOW_AUTOSIZE );

```

```
77     imshow( "Hull demo", drawing );
78 }
```

۲,۱۸,۲ توضیح

در خط ۴۲ تابع `thresh_callback` تعریف شده است. در این تابع ابتدا یک کپی از ماتریس `src` به نام `src_copy` درست می‌کنیم. سپس یک ماتریس برای نگه داری خروجی تابع `threshold` درست می‌کنیم. همچنین یک بردار دو بعدی از نوع `Point` برای نگه داری کانتورهای به دست آمده و یک بردار تک بعدی از نوع `Vec4i` برای نگه داری خروجی اختیاری تابع `findContours`، یعنی `hierarchy`، تعریف می‌کنیم.

نکته: `Vec4i` یک بردار ۱ در ۴ است که همه عناصر آن از نوع `int` هستند.

در خط ۵۰ با استفاده از تابع `threshold` تصویر سیاه و سفید را آستانه‌گذاری می‌کنیم و آن را به یک تصویر باینری تبدیل می‌کنیم. برای این کار از آستانه وارد شده توسط کاربر (`thresh`) استفاده می‌کنیم

با فراخوانی تابع `findContours` در خط ۵۴، کانتورهای تصویر خروجی تابع `threshold` (یعنی `threshold_output`) را پیدا می‌کنیم.

در خط ۵۸ یک بردار دو بعدی به نام `hull` از نوع `Point` برای نگه داری پوش محدب هر کدام از کانتورها درست می‌کنیم. سپس در یک حلقه، تابع `convexHull` را روی هر کدام از کانتورها صدای زنیم و خروجی‌ها را در بردار `hull` ذخیره می‌کنیم. تابع `hull` دارای ۳ آرگومان به شرح زیر است:

۱. **Mat(contours[i])**: آرایه ورودی از نقاط است. می‌توانیم آنها را در یک بردار یا یک ماتریس قرار دهیم.
۲. **hull[i]**: پوش محدب مربوط به نقاط ورودی در این بردار قرار می‌گیرند.
۳. **false**: جهت پوش محدب را مشخص می‌کند. اگر `true` باشد به معنی این است که جهت به صورت موافق با عقربه‌های ساعت است و اگر `false` باشد به معنی خلاف عقربه‌های ساعت است. در هر حالت، نقطه `(0,0)` در گوش سمت چپ و بالا قرار دارد.

۲,۱۸,۳ خروجی



خروجی



وروودی

چند ضلعی بنفس رنگ که همه تصویر را در برگرفته، همان پوش محدب تصویر دست است.

۲،۱۹ قاب‌های مستطیلی و دایره‌ای برای کانتورها

۲،۱۹،۱

```
1 #include "opencv2/highgui/highgui.hpp"
2 #include "opencv2/imgproc/imgproc.hpp"
3 #include <iostream>
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 using namespace cv;
8 using namespace std;
9
10 Mat src; Mat src_gray;
11 int thresh = 100;
12 int max_thresh = 255;
13 RNG rng(12345);
14
15 // Function header
16 void thresh_callback(int, void* );
17
18 /** @function main */
19 int main( int argc, char** argv )
20 {
21     // Load source image and convert it to gray
22     src = imread( argv[1], 1 );
23
24     // Convert image to gray and blur it
25     cvtColor( src, src_gray, CV_BGR2GRAY );
26     blur( src_gray, src_gray, Size(3,3) );
27
28     // Create Window
29     char* source_window = "Source";
30     namedWindow( source_window, CV_WINDOW_AUTOSIZE );
31     imshow( source_window, src );
32
33     createTrackbar( " Threshold:", "Source", &thresh, max_thresh,
34     thresh_callback );
35     thresh_callback( 0, 0 );
36
37     waitKey(0);
38     return(0);
39 }
40
41 /** @function thresh_callback */
42 void thresh_callback(int, void* )
43 {
44     Mat threshold_output;
45     vector<vector<Point> > contours;
46     vector<Vec4i> hierarchy;
47
48     // Detect edges using Threshold
49     threshold( src_gray, threshold_output, thresh, 255, THRESH_BINARY
50 );
51     // Find contours
52     findContours( threshold_output, contours, hierarchy,
53     CV_RETR_TREE, CV_CHAIN_APPROX_SIMPLE, Point(0, 0) );
54
55     // Approximate contours to polygons + get bounding rects and
56     circles
```

```

57     vector<vector<Point>> contours_poly( contours.size() );
58     vector<Rect> boundRect( contours.size() );
59     vector<Point2f>center( contours.size() );
60     vector<float>radius( contours.size() );
61
62     for( int i = 0; i < contours.size(); i++ )
63     { approxPolyDP( Mat(contours[i]), contours_poly[i], 3, true );
64         boundRect[i] = boundingRect( Mat(contours_poly[i]) );
65         minEnclosingCircle( (Mat)contours_poly[i], center[i],
66                             radius[i] );
67     }
68
69
70
71     /// Draw polygonal contour + bonding rects + circles
72     Mat drawing = Mat::zeros( threshold_output.size(), CV_8UC3 );
73     for( int i = 0; i < contours.size(); i++ )
74     {
75         Scalar color = Scalar( rng.uniform(0, 255),
76                               rng.uniform(0,255), rng.uniform(0,255) );
77         drawContours( drawing, contours_poly, i, color, 1, 8,
78                     vector<Vec4i>(), 0, Point() );
79         rectangle( drawing, boundRect[i].tl(), boundRect[i].br(),
80                     color, 2, 8, 0 );
81         circle( drawing, center[i], (int)radius[i], color, 2, 8, 0 );
82     }
83
84     /// Show in a window
85     namedWindow( "Contours", CV_WINDOW_AUTOSIZE );
86     imshow( "Contours", drawing );
87 }
88

```

۲،۱۹،۲ توضیح

در خط ۴۲ تابع `thresh_callback` تعریف شده است. در این تابع ابتدا یک ماتریس برای نگه داری خروجی تابع `threshold` درست می‌کنیم. همچنین یک بردار دو بعدی از نوع `Point` برای نگه داری کانتورهای به دست آمده و یک بردار تک بعدی از نوع `Vec4i` برای نگه داری خروجی اختیاری تابع `findContours`، یعنی `hierarchy`، تعریف می‌کنیم.

در خط ۴۹ با استفاده از تابع `threshold` تصویر سیاه و سفید را آستانه‌گذاری می‌کنیم و آن را به یک تصویر باینری تبدیل می‌کنیم. برای این کار از آستانه وارد شده توسط کاربر (`thresh`) استفاده می‌کنیم.

با فراخوانی تابع `findContours`، کانتورهای تصویر خروجی تابع `threshold` (یعنی `threshold_output`) را پیدا می‌کنیم.

در خطوط ۵۷ تا ۶۰ بردارهایی برای نگه داری پارامترهای قاب‌های مستطیلی و دایره‌ای درست می‌کنیم.

در حلقه موجود در خط ۶۱ به ترتیب کارهای زیر را انجام می‌دهیم:

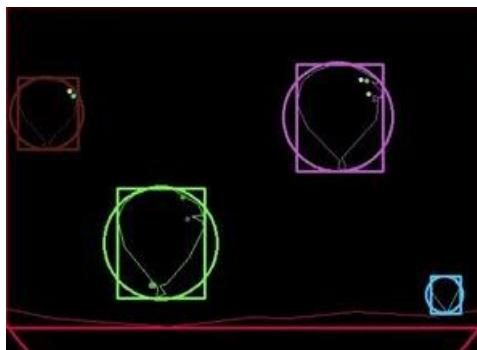
۱. ابتدا در خط ۶۲ با استفاده از تابع `approxPolyDP` برای `contours[i]` یک چند ضلعی تقریبی پیدا می‌کنیم و آن را در `contours_poly[i]` ذخیره می‌کنیم. عدد ۳ دقت تقریب را مشخص می‌کند. به عبارتی می‌توان گفت که این عدد حداقلر فاصله بین منحنی تقریبی و منحنی اصلی است. هم به معنی این است که می‌خواهیم یک چند ضلعی بسته داشته باشیم (یعنی در منحنی تقریبی، اولین نقطه به آخرین نقطه متصل باشد).

۲. سپس در خط ۶۳ با استفاده از تابع `boundingRect` قاب مستطیلی مربوط به چند ضلعی تقریبی به دست آمده برای `approxPolyDP` را حساب و آن را در `[i]` ذخیره می‌کنیم. ورودی این تابع همان خروجی تابع `contours[i]` است که در یک ماتریس قرار داده شده (می‌توان آن را در یک بردار هم قرار داد).

۳. بعد در خط ۶۴ با استفاده از تابع `minEnclosingCircles`. قاب دایره‌ای مربوط به چند ضلعی تقریبی به دست آمده برای `contours[i]` را حساب می‌کنیم و مرکز دایره را در `[i]` و شعاع آن را در `radius[i]` ذخیره می‌کنیم.

سپس در خطوط ۷۳ تا ۸۲ در یک حلقه حرکت می‌کنیم و تمام کانتورها و قاب‌های دایره‌ای و مستطیلی مربوط به آنها را روی تصویر `drawing` می‌کشیم (برای کشیدن کانتورها از تابع `drawContours` و برای کشیدن دایره‌ها و مستطیل‌ها از توابع `circle` و `rectangle` استفاده می‌کنیم).

۲,۱۹,۳ خروجی



تصویر خروجی



تصویر ورودی

۲,۲۰,۱ کد ۲,۲۰ قاب‌های چرخیده مستطیلی و بیضوی برای کانتورها

```
1 #include "opencv2/highgui/highgui.hpp"
2 #include "opencv2/imgproc/imgproc.hpp"
3 #include <iostream>
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 using namespace cv;
8 using namespace std;
9
10 Mat src; Mat src_gray;
11 int thresh = 100;
12 int max_thresh = 255;
13 RNG rng(12345);
14
15 /// Function header
16 void thresh_callback(int, void* );
17
18 /** @function main */
19 int main( int argc, char** argv )
20 {
21     /// Load source image and convert it to gray
22     src = imread( argv[1], 1 );
23
24     /// Convert image to gray and blur it
25     cvtColor( src, src_gray, CV_BGR2GRAY );
```

```

26     blur( src_gray, src_gray, Size(3,3) );
27
28     /// Create Window
29     char* source_window = "Source";
30     namedWindow( source_window, CV_WINDOW_AUTOSIZE );
31     imshow( source_window, src );
32
33     createTrackbar( " Threshold:", "Source", &thresh, max_thresh,
34 thresh_callback );
35     thresh_callback( 0, 0 );
36
37     waitKey(0);
38     return(0);
39 }
40
41 /** @function thresh_callback */
42 void thresh_callback(int, void* )
43 {
44     Mat threshold_output;
45     vector<vector<Point> > contours;
46     vector<Vec4i> hierarchy;
47
48     /// Detect edges using Threshold
49     threshold( src_gray, threshold_output, thresh, 255, THRESH_BINARY
50 );
51     /// Find contours
52     findContours( threshold_output, contours, hierarchy,
53 CV_RETR_TREE, CV_CHAIN_APPROX_SIMPLE, Point(0, 0) );
54
55     /// Find the rotated rectangles and ellipses for each contour
56     vector<RotatedRect> minRect( contours.size() );
57     vector<RotatedRect> minEllipse( contours.size() );
58
59     for( int i = 0; i < contours.size(); i++ )
60     { minRect[i] = minAreaRect( Mat(contours[i]) );
61         if( contours[i].size() > 5 )
62             { minEllipse[i] = fitEllipse( Mat(contours[i]) );
63             }
64
65     /// Draw contours + rotated rects + ellipses
66     Mat drawing = Mat::zeros( threshold_output.size(), CV_8UC3 );
67     for( int i = 0; i < contours.size(); i++ )
68     {
69         Scalar color = Scalar( rng.uniform(0, 255),
70 rng.uniform(0,255), rng.uniform(0,255) );
71         // contour
72         drawContours( drawing, contours, i, color, 1, 8,
73 vector<Vec4i>(), 0, Point() );
74         // ellipse
75         ellipse( drawing, minEllipse[i], color, 2, 8 );
76         // rotated rectangle
77         Point2f rect_points[4]; minRect[i].points( rect_points );
78         for( int j = 0; j < 4; j++ )
79             line( drawing, rect_points[j], rect_points[(j+1)%4],
80 color, 1, 8 );
81     }
82
83     /// Show in a window
84     namedWindow( "Contours", CV_WINDOW_AUTOSIZE );
85     imshow( "Contours", drawing );
86
87 }
```

۲،۲۰،۲ توضیح

در خط ۴۲ تابع `thresh_callback` تعریف شده است. در این تابع ابتدا یک ماتریس برای نگه داری خروجی تابع `threshold` درست می‌کنیم. همچنین یک بردار دو بعدی از نوع `Point` برای نگه داری کانتورهای به دست آمده و یک بردار تک بعدی از نوع `Vec4i` برای نگه داری خروجی اختیاری تابع `findContours`، یعنی `hierarchy`، تعریف می‌کنیم.

در خط ۴۹ با استفاده از تابع `threshold` تصویر سیاه و سفید را آستانه‌گذاری می‌کنیم و آن را به یک تصویر باینری تبدیل می‌کنیم. برای این کار از آستانه وارد شده توسط کاربر (`thresh`) استفاده می‌کنیم.

در خط ۵۲ با فراخوانی تابع `findContours`، کانتورهای تصویر خروجی تابع `threshold` (یعنی `threshold_output`) را پیدا می‌کنیم.

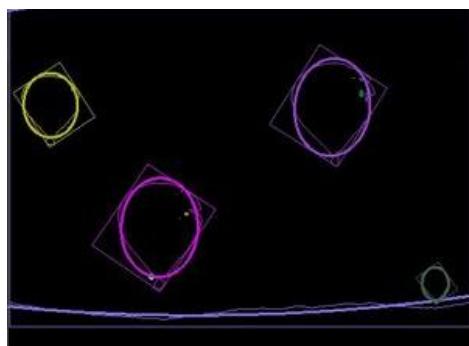
در خطوط ۵۶ و ۵۷ دو بردار از نوع `RotatedRect` برای نگه داری مستطیل‌ها و بیضی‌های کشف شده درست می‌کنیم.

در حلقه موجود در خط ۵۹، روی هر کدام از کانتورها عملیات زیر را انجام می‌دهیم:

۱. ابتدا در خط ۶۰ با استفاده از تابع `minAreaRect` کوچکترین مستطیل در بر گیرنده همه نقاط کانتور `[i]` را `contours` پیدا می‌کنیم و آن را در `[i]` `minRect` قرار می‌دهیم.
۲. سپس در خط ۶۲ با استفاده از تابع `fitEllipse` و به شرطی که تعداد نقاط موجود در `[i]` `contours` بیشتر از ۵ عدد باشد، کوچکترین بیضی که در بر گیرنده همه نقاط `[i]` `contours` است را پیدا می‌کنیم و آن را در `[i]` `minEllipse` قرار می‌دهیم.

سپس در خطوط ۶۷ تا ۸۲ در یک حلقه حرکت می‌کنیم و کانتورها و قاب‌های مستطیلی و بیضوی آنها را روی تصویر `drawing` می‌کشیم. برای کشیدن قاب‌های مستطیلی چرخیده، چون تابعی برای این کار نداریم، ابتدا نقاط مربوط به چهار گوشه مستطیل را به دست می‌آوریم و سپس مستطیل را به صورت چهار خط می‌کشیم.

۲،۲۰،۳ خروجی



تصویر خروجی



تصویر ورودی

۳ فصل سوم- حوزه زمان

۱،۱ تبدیل فوریه گستته

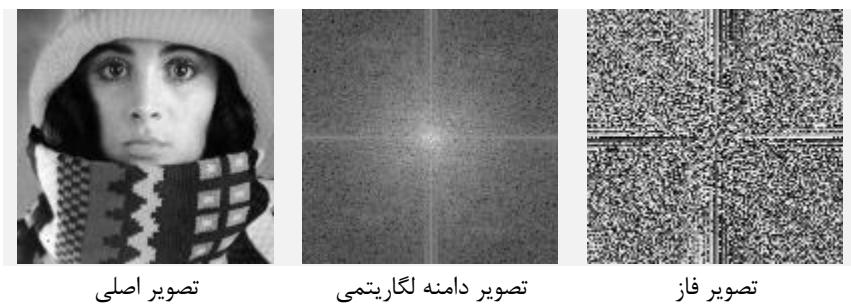
تبدیل فوریه تصویر را به مؤلفه های سینوسی و کسینوسی آن تبدیل می کند. به عبارت دیگر، تصویر را از دامنه مکانی^{۵۵} به دامنه زمانی یا فرکانسی^{۵۶} می برد. اصل قضیه این است که هرتابع را می توان به صورت جمع تعداد بینهایت سینوس و کسینوس درآورد و تبدیل فوریه راهی برای انجام این کار است. به صورت ریاضی وار، تبدیل فوریه دو بعدی برای یک تصویر به صورت زیر است:

$$F(k, l) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} f(i, j) e^{-i2\pi(\frac{ki}{N} + \frac{lj}{N})}$$

$$e^{ix} = \cos x + i \sin x$$

در این معادله f مقدار تصویر در دامنه مکانی و F هم مقدار تصویر در دامنه زمانی است. نتیجه این تبدیل اعداد مختلط است. می توانیم نتیجه را به صورت یک ماتریس از قسمت حقیقی^{۵۷} و یک ماتریس از قسمت مجازی^{۵۸} این اعداد و یا به صورت دامنه^{۵۹} و فاز^{۶۰} اعداد مختلط نشان دهیم. در اکثر الگوریتم های پردازش تصویر بیشتر با دامنه سروکار داریم؛ چون فاز یک تصویر اطلاعات اصلی آن تصویر را نگه داری می کند و هر گونه تغییر در مقادیر آن ممکن است باعث نابودی آن تصویر شود. ولی تغییر دامنه موجب تغییر در کیفیت تصویر می شود.

در زیر دامنه و فاز تصویر سمت چپ نشان داده شده است:



برای بازسازی کامل یک تصویر از تبدیل فوریه آن، هم به فاز و هم به دامنه آن تصویر نیاز است. در زیر می بینید که نتیجه بازسازی تصویر بالا فقط با استفاده از دامنه یا فاز آن به چه صورتی در می آید:

Spatial Domain^{۵۵}

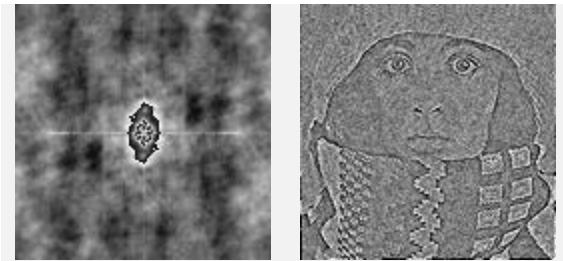
Frequency Domain^{۵۶}

Real^{۵۷}

Imaginary^{۵۸}

Magnitude^{۵۹}

Phase^{۶۰}



فقط دامنه

فقط فاز

برای بازسازی یک تصویر فقط با استفاده از فاز آن باید مقدار دامنه را یک قرار دهیم و سپس عمل بازسازی را انجام دهیم. همچنین برای بازسازی یک تصویر فقط با استفاده از دامنه آن باید مقدار فاز را صفر قرار دهیم و سپس عمل بازسازی را انجام دهیم.

كـ ٣,١,١

```

1 #include "opencv2/core/core.hpp"
2 #include "opencv2/imgproc/imgproc.hpp"
3 #include "opencv2/highgui/highgui.hpp"
4
5 #include <iostream>
6
7 using namespace cv;
8
9 // Rearrange the quadrants of a Fourier image so that the origin is
10 // at the image center
11 void shiftDFT(Mat &fImage )
12 {
13     Mat tmp, q0, q1, q2, q3;
14
15     // first crop the image, if it has an odd number of rows or
16     columns
17
18     fImage = fImage(Rect(0, 0, fImage.cols & -2, fImage.rows & -2));
19
20     int cx = fImage.cols / 2;
21     int cy = fImage.rows / 2;
22
23     // rearrange the quadrants of Fourier image
24     // so that the origin is at the image center
25
26     q0 = fImage(Rect(0, 0, cx, cy));
27     q1 = fImage(Rect(cx, 0, cx, cy));
28     q2 = fImage(Rect(0, cy, cx, cy));
29     q3 = fImage(Rect(cx, cy, cx, cy));
30
31     q0.copyTo(tmp);
32     q3.copyTo(q0);
33     tmp.copyTo(q3);
34
35     q1.copyTo(tmp);
36     q2.copyTo(q1);
37     tmp.copyTo(q2);
38 }
39
40 int main()
41 {
42     // Load an image
43     Mat I1 = imread("PICTURES/1.jpg", CV_LOAD_IMAGE_GRAYSCALE);
44     Mat I2 = imread("PICTURES/2.jpg", CV_LOAD_IMAGE_GRAYSCALE);

```

```

45     Mat fI1;
46     Mat fI2;
47     I1.convertTo(fI1, CV_32F);
48     I2.convertTo(fI2, CV_32F);
49
50     //expand input image to optimal size
51     int m = getOptimalDFTSize( I1.rows );
52     int n = getOptimalDFTSize( I1.cols );
53
54     Mat padded1, padded2;
55
56     // on the border add zero values
57     copyMakeBorder(fI1, padded1, 0, m - I1.rows, 0, n - I1.cols,
58 BORDER_CONSTANT, Scalar::all(0));
59     copyMakeBorder(fI2, padded2, 0, m - I2.rows, 0, n - I2.cols,
60 BORDER_CONSTANT, Scalar::all(0));
61
62     //Perform DFT
63     Mat fourierTransform1;
64     Mat fourierTransform2;
65     Mat planes1[2], planes2[2];
66
67     dft(fI1, fourierTransform1, DFT_SCALE|DFT_COMPLEX_OUTPUT);
68     dft(fI2, fourierTransform2, DFT_SCALE|DFT_COMPLEX_OUTPUT);
69
70     // rearrange the quadrants of Fourier image
71     shiftDFT(fourierTransform1);
72     shiftDFT(fourierTransform2);
73
74     split(fourierTransform1, planes1); // planes1[0] = Re(DFT(I1)),
75     planes1[1] = Im(DFT(I1))
76     split(fourierTransform2, planes2); // planes2[0] = Re(DFT(I2)),
77     planes2[1] = Im(DFT(I2))
78
79     Mat ph1, mag1;
80     cartToPolar(planes1[0], planes1[1], mag1, ph1);
81
82     Mat ph2, mag2;
83     cartToPolar(planes2[0], planes2[1], mag2, ph2);
84
85     /// if we want to show only the magnitude of the images:
86     // ph1 = Mat::zeros(planes1[0].rows, planes1[0].cols, CV_32FC1);
87     // ph2 = Mat::zeros(planes2[0].rows, planes2[0].cols, CV_32FC1);
88
89     /// if we want to show only the phase of the images:
90     // mag1 = Mat::ones(planes1[0].rows, planes1[0].cols, CV_32FC1);
91     // mag2 = Mat::ones(planes2[0].rows, planes2[0].cols, CV_32FC1);
92
93     polarToCart(mag2, ph1, planes1[0], planes1[1]);
94     polarToCart(mag1, ph2, planes2[0], planes2[1]);
95
96     merge(planes1, 2, fourierTransform1);
97     merge(planes2, 2, fourierTransform2);
98
99     // rearrange the quadrants of Fourier image
100    shiftDFT(fourierTransform1);
101    shiftDFT(fourierTransform2);
102
103    //Perform IDFT
104    Mat inverseTransform1, inverseTransform2;
105

```

```

106     dft(fourierTransform1, inverseTransform1,
107     DFT_INVERSE|DFT_REAL_OUTPUT);
108     dft(fourierTransform2, inverseTransform2,
109     DFT_INVERSE|DFT_REAL_OUTPUT);
110
111     imshow("original image 1", I1);
112     imshow("original image 2", I2);
113
114     cv::Mat out1, out2;
115     inverseTransform1.convertTo(out1, CV_8U);
116     inverseTransform2.convertTo(out2, CV_8U);
117
118     imshow("result image 1", out1);
119     imshow("result image 2", out2);
120
121     waitKey(0);
122 }

```

۳،۱،۲ توضیح

ابتدا در خطوط ۴۳ و ۴۴ دو تصویر را از روی دیسک به صورت سیاه و سفید می‌خوانیم. بهتر است که ابعاد هر دو تصویر یکسان باشد؛ البته اگر هم اندازه نباشند هم در ادامه اندازه تصویر دوم هم اندازه تصویر اول می‌کنیم. می‌خواهیم در ادامه فاز و دامنه این دو تصویر را حساب کنیم و نمایش دهیم.

در خطوط ۴۸ و ۴۹، نوع عکس را از ۸ بیتی صحیح به ۳۲ بیتی اعشاری تبدیل می‌کنیم.

در خطوط ۵۲ و ۵۳ اندازه بهینه تصاویر برای انجام تبدیل فوریه را حساب می‌کنیم. برای اینکه عملیات تبدیل فوریه گسترش به صورت بهینه انجام شود، ابعاد تصویر مورد نظر بهتر است عددی باشد که بتوان آن را به صورت ضرب توان‌های سه عدد ۳، ۲ و ۵ نوشت. می‌توان کوچکترین عدد بزرگتر از بُعد فعلی که همچین شرایطی داشته باشد را با استفاده از تابع `getOptimalDFTSize` به دست آورد. در اینجا ابتدا در خط ۵۲ اندازه بهینه برای تعداد سطرها را به دست می‌آوریم و سپس اندازه بهینه برای تعداد ستون‌ها را حساب می‌کنیم. البته لازم به ذکر است که تابع `dft` می‌تواند با تصویر با هر ابعادی کار کند و این کارها فقط برای بهبود سرعت این تابع است.

در خطوط ۵۵ تا ۶۱ ابعاد تصاویر ورودی را افزایش می‌دهیم تا مطابق با ابعاد بهینه شوند. مثلاً برای افزایش ابعاد تصویر ورودی اول به صورت زیر از تابع `copyMakeBorder` استفاده می‌کنیم:

```
copyMakeBorder(fI1, padded1, 0, m - I1.rows, 0, n - I1.cols,
BORDER_CONSTANT, Scalar::all(0));
```

این تابع برای افزایش ابعاد تصویر ورودی یک قاب با اندازه مشخص شده به اطراف تصویر اضافه می‌کند. در اینجا `fI1` تصویر ورودی و `padded1` نام ماتریسی است که خروجی در آن قرار می‌گیرد. بعد از آن چهار عدد آمده که عدد اول مشخص کننده اندازه قاب بالایی، عدد دوم اندازه قاب پایینی، عدد سوم اندازه قاب سمت راست و عدد چهارم اندازه قاب سمت چپ است. دو آرگومان آخر به معنی این است که می‌خواهیم رنگ قاب یکنواخت و سیاه باشد.

در خطوط ۶۸ و ۶۹ با استفاده از تابع `dft`، تبدیل فوریه گسترش دو تصویر ورودی (که به ابعاد بهینه در آمده‌اند) را حساب می‌کنیم. در این تابع از دو پرچم استفاده کردیم. `DFT_SCALES` مشخص می‌کند که خروجی بر تعداد عناصر ماتریس تقسیم شود. `DFT_COMPLEX_OUTPUT` هم مشخص می‌کند که خروجی به صورت عدد مختلط در فضای کارتزین باشد.

در خطوط ۷۲ و ۷۳ مرکز تبدیل فوریه را از چهار گوش به مرکز ماتریس منتقل می‌کنیم. این کار را با صدا زدن تابع `shiftDFT` انجام می‌دهیم.

ماتریس‌های fourierTransform1 و fourierTransform2 دو کانال اول قسمت حقیقی تبدیل فوریه و در کanal دوم قسمت مجازی تبدیل فوریه قرار دارد. برای اینکه راحت‌تر به این دو مؤلفه دسترسی داشته باشیم، در خطوط ۷۵ و ۷۷، با استفاده ازتابع split کانال‌های ماتریس‌های fourierTransform1 و fourierTransform2 را جدا می‌کنیم و در آرایه1 و planes2 قرار می‌دهیم. توجه کنید که در اینجا داده‌های ماتریس‌ها کپی نمی‌شوند و فقط اشاره گری به آنها بر گردانده می‌شود.

در خطوط ۸۱ و ۸۴ با استفاده از تابع cartToPolar مختصه‌های planes1 و planes2 را از فضای کارتزین به فضای قطبی می‌بریم و نتیجه را در mag1، ph1 و ph2 ذخیره می‌کنیم. حالا به فاز و دامنه تصاویر دسترسی داریم. می‌توانیم آن‌ها را نمایش دهیم و یا تغییر دهیم و یا هر کاری می‌خواهیم با آنها انجام دهیم. در ادامه می‌خواهیم برای بازسازی تصویر اول از دامنه تصویر دوم و برای تصویر دوم از دامنه تصویر اول استفاده کنیم.

اگر می‌خواهید تصاویر نهایی را فقط با استفاده از دامنه بازسازی کنید، لازم است که فاز هر دو تصویر را صفر کنید. برای اینکار خطوط ۸۷ و ۸۸ را باید از حالت توضیح خارج کنید. همچنین اگر می‌خواهید تصاویر نهایی را فقط با استفاده از فاز بازسازی کنید، لازم است که دامنه هر دو تصویر را یک کنید. برای اینکار خطوط ۹۱ و ۹۲ را باید از حالت توضیح خارج کنید.

در خطوط ۹۴ و ۹۵ با استفاده از تابع polarToCart مؤلفه‌های فوریه تصاویر را از مختصات قطبی به مختصات کارتزین می‌بریم. در اینجا ماتریس‌های mag1، ph1 و ph2 مؤلفه‌های مختصات قطبی هستند. نتیجه در planes1 و planes2 ذخیره می‌شود. توجه کنید که در خط ۹۴ برای بازسازی تصویر اول از دامنه تصویر دوم استفاده کردایم، همچنین در خط ۹۵ برای بازسازی تصویر دوم از دامنه تصویر اول استفاده کردایم.

در خط ۹۷ با استفاده از تابع merge دو ماتریس موجود در آرایه planes1 را به یک ماتریس دو کاناله تبدیل می‌کنیم و در ماتریس fourierTransform1 قرار می‌دهیم. همین کار را در خط ۹۸ برای planes2 انجام می‌دهیم.

در خطوط ۱۰۱ و ۱۰۲ مجدداً مرکز ماتریس تبدیل فوریه را به چهار گوشه منتقل می‌کنیم.

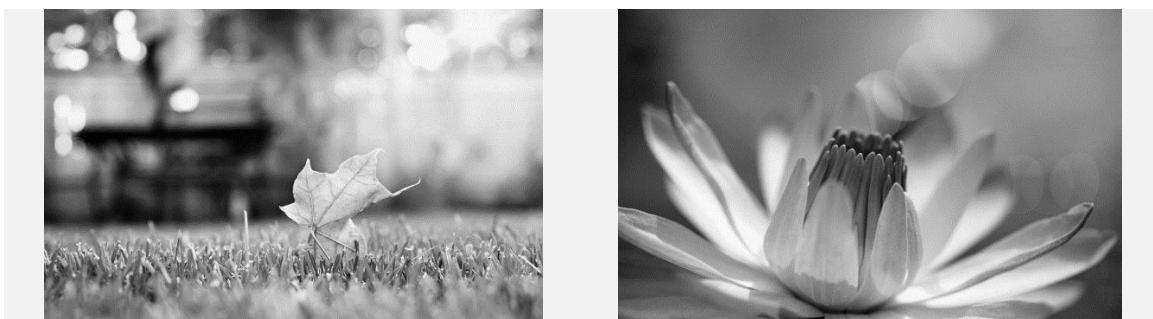
در خطوط ۱۰۷ و ۱۰۹ تبدیل معکوس فوریه را انجام می‌دهیم. مثلاً در خط ۱۰۷ به صورت زیر عمل می‌کنیم:

```
dft(fourierTransform1, inverseTransform1, DFT_INVERSE|DFT_REAL_OUTPUT);
```

در اینجا fourierTransform1 ماتریس دو کاناله ای است که در آن تبدیل فوریه به فرم اعداد مختلط در فضای کارتزین قرار دارد. خروجی که به صورت یک ماتریس تک کاناله از اعداد حقیقی است، در ماتریس inverseTransform1 قرار می‌گیرد. پرچم DFT_INVERSE بدین معناست که می‌خواهیم تبدیل معکوس فوریه انجام دهیم و DFT_REAL_OUTPUT به این معنی است که می‌خواهیم خروجی به صورت اعداد حقیقی باشد.

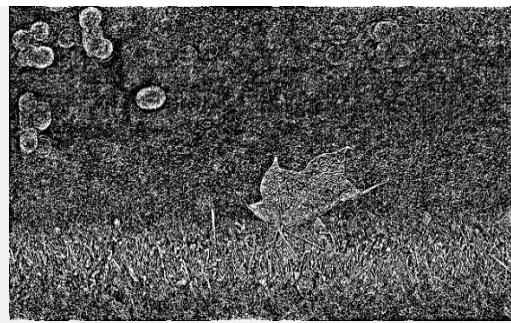
در خطوط ۱۱۶ و ۱۱۷ ماتریس‌هایی که از تبدیل فوریه معکوس به دست آمده بودند را به نوع ۸ بیتی تبدیل می‌کنیم و سرانجام در خطوط ۱۱۹ و ۱۲۰ خروجی نهایی را نشان می‌دهیم.

۳,۱,۳ خروجی

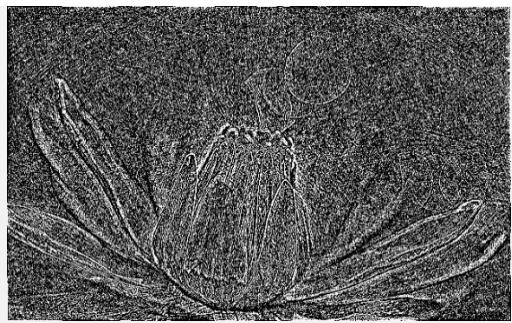


تصویر ورودی دوم

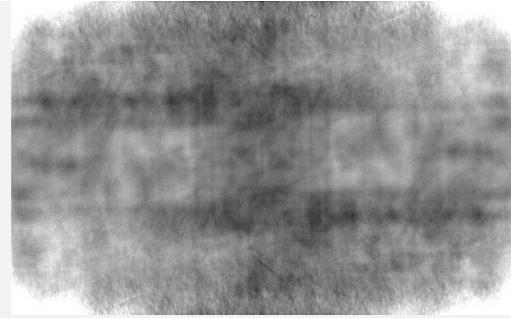
تصویر ورودی اول



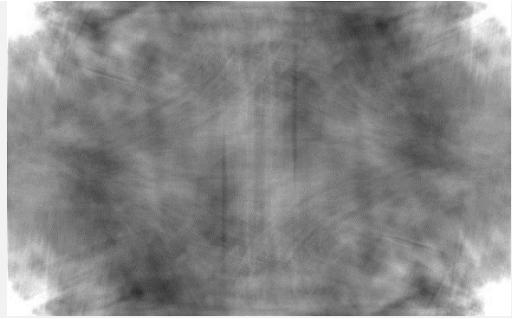
بازسازی تصویر دوم فقط با استفاده از فاز آن



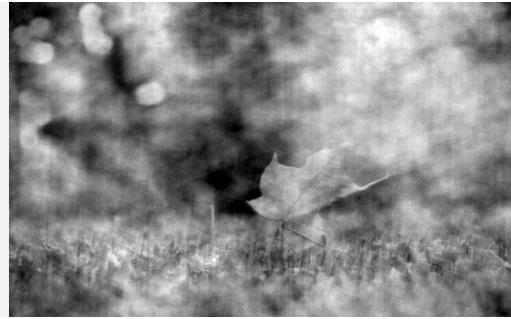
بازسازی تصویر اول فقط با استفاده از فاز آن



بازسازی تصویر دوم فقط با استفاده از دامنه آن



بازسازی تصویر اول فقط با استفاده از دامنه آن



بازسازی تصویر اول با استفاده از فاز خودش و دامنه تصویر دوم بازسازی تصویر دوم با استفاده از فاز خودش و دامنه تصویر اول

۴ فصل چهارم - مورفولوژی

۴،۱ فرسایش و گشايش

به طور کلی به دسته‌ای از عملیات که تصاویر را بر پایه اشکال هندسی پردازش می‌کنند، عملیات مورفولوژی می‌گویند. عملیات مورفولوژی به تصویر ورودی یک عنصر سازنده^{۶۱} اعمال می‌کنند و یک تصویر خروجی تولید می‌کنند. در فرایند مورفولوژی میزان شباهت اشکال موجود در تصویر داده شده را با عنصر سازنده که خود یک شکل هندسی است، بررسی می‌شود.

ابتدایی ترین عملیات مورفولوژی، عملیات فرسایش^{۶۲} و گشايش^{۶۳} هستند که کاربردهای بسیاری دارند، از جمله:

- از بین بردن نویز
- جدا کردن عناصر خاص و متصل کردن قسمت‌های جدا از هم موجود در یک تصویر
- پیدا کردن قسمت‌های بسیار تیره (حفره‌ها^{۶۴}) و بسیار روشن تصویر

در ادامه با در نظر گرفتن تصویر زیر، عملیات فرسایش و گشايش را بررسی می‌کنیم.



تصویر ورودی

گشايش:

این عمل از کانولوشن تصویر A با کرنل B که می‌تواند هر اندازه و شکلی داشته باشد (که معمولاً دایره یا مربع است)، تشکیل شده است. کرنل B یک لنگر دارد که معمولاً در مرکز کرنل است.

همین‌طور که کرنل B روی تصویر حرکت می‌کند، پیکسل بیشینه‌ای که در پنجره B است را پیدا کرده و مقدار آن را به جای مقدار پیکسلی که لنگر روی آن قرار دارد می‌گذارد. پس این عمل باعث رشد نواحی روشن در تصویر می‌شود (نام گشايش از اینجا می‌آيد). به عنوان مثال اگر عملگر گشايش را به تصویر ورودی بالا اعمال کنیم، نتیجه به صورت زیر خواهد بود:

Structuring element^{۶۱}

Erode^{۶۲}

Dilate^{۶۳}

holes^{۶۴}

نتیجه اعمال گشایش روی تصویر ورودی

پس زمینه تصویر (قسمت روشن) نسبت به ناحیه تیره مربوط به حرف ج، گشایش می‌یابد.

فرسایش:

همین طور که کرنل B روی تصویر حرکت می‌کند، پیکسل کمینه‌ای که در پنجره B است را پیدا کرده و مقدار آن را به جای مقدار پیکسلی که لنگر روی آن قرار دارد می‌گذارد.

عملگر فرسایش را روی تصویر ورودی اعمال می‌کنیم. شکل زیر نتیجه این کار را نشان می‌دهد. در این تصویر می‌بینید که ناحیه‌های روشن تصویر (یعنی پس زمینه) نازک‌تر شده‌اند (یعنی فرسایش یافته‌اند)، در حالی که ناحیه‌های تیره بزرگ‌تر شده‌اند.

نتیجه اعمال فرسایش روی تصویر ورودی

۴.۱.۱ کد

```

1 #include "opencv2/imgproc/imgproc.hpp"
2 #include "opencv2/highgui/highgui.hpp"
3 #include <stdlib.h>
4 #include <stdio.h>
5
6 using namespace cv;
7
8 /// Global variables
9 Mat src, erosion_dst, dilation_dst;
10
11 int erosion_elem = 0;
12 int erosion_size = 0;
13 int dilation_elem = 0;
14 int dilation_size = 0;
15 int const max_elem = 2;
16 int const max_kernel_size = 21;
17

```

```

18  /** Function Headers */
19 void Erosion( int, void* );
20 void Dilation( int, void* );
21
22 /** @function main */
23 int main( int argc, char** argv )
24 {
25     /// Load an image
26     src = imread( argv[1] );
27
28     if( !src.data )
29     { return -1; }
30
31     /// Create windows
32     namedWindow( "Erosion Demo", CV_WINDOW_AUTOSIZE );
33     namedWindow( "Dilation Demo", CV_WINDOW_AUTOSIZE );
34     cvMoveWindow( "Dilation Demo", src.cols, 0 );
35
36     /// Create Erosion Trackbar
37     createTrackbar( "Element:\n 0: Rect \n 1: Cross \n 2: Ellipse",
38     "Erosion Demo",
39                     &erosion_elem, max_elem,
40                     Erosion );
41
42     createTrackbar( "Kernel size:\n 2n +1", "Erosion Demo",
43                     &erosion_size, max_kernel_size,
44                     Erosion );
45
46     /// Create Dilation Trackbar
47     createTrackbar( "Element:\n 0: Rect \n 1: Cross \n 2: Ellipse",
48     "Dilation Demo",
49                     &dilation_elem, max_elem,
50                     Dilation );
51
52     createTrackbar( "Kernel size:\n 2n +1", "Dilation Demo",
53                     &dilation_size, max_kernel_size,
54                     Dilation );
55
56     /// Default start
57     Erosion( 0, 0 );
58     Dilation( 0, 0 );
59
60     waitKey(0);
61     return 0;
62 }
63
64 /** @function Erosion */
65 void Erosion( int, void* )
66 {
67     int erosion_type;
68     if( erosion_elem == 0 ){ erosion_type = MORPH_RECT; }
69     else if( erosion_elem == 1 ){ erosion_type = MORPH_CROSS; }
70     else if( erosion_elem == 2 ) { erosion_type = MORPH_ELLIPSE; }
71
72     Mat element = getStructuringElement( erosion_type,
73                                         Size( 2*erosion_size + 1,
74                                         2*erosion_size+1 ),
75                                         Point( erosion_size,
76                                         erosion_size ) );
77
78     /// Apply the erosion operation

```

```

79     erode( src, erosion_dst, element );
80     imshow( "Erosion Demo", erosion_dst );
81 }
82
83 /** @function Dilation */
84 void Dilation( int, void* )
85 {
86     int dilation_type;
87     if( dilation_elem == 0 ){ dilation_type = MORPH_RECT; }
88     else if( dilation_elem == 1 ){ dilation_type = MORPH_CROSS; }
89     else if( dilation_elem == 2 ) { dilation_type = MORPH_ELLIPSE; }
90
91     Mat element = getStructuringElement( dilation_type,
92                                         Size( 2*dilation_size + 1,
93                                               2*dilation_size+1 ),
94                                         Point( dilation_size,
95                                               dilation_size ) );
96     /// Apply the dilation operation
97     dilate( src, dilation_dst, element );
98     imshow( "Dilation Demo", dilation_dst );
99 }

```

۴,۱,۲ توضیح

تابع Erosion (خط ۶۵):

تابعی که عمل گشایش را انجام می‌دهد، تابع erode است (خط ۷۹). این تابع سه آرگومان دریافت می‌کند:

- `src`: تصویر ورودی
- `erosion_dst`: تصویر خروجی
- `element`: کرنلی که برای عمل گشایش از آن استفاده خواهد شد. اگر آن را تعیین نکنیم به صورت پیشفرض یک ماتریس ۳×۳ استفاده می‌شود. در غیر این صورت برای مشخص کردن شکل آن از تابع getStructuringElement استفاده می‌شود.
- می‌توان یکی از سه شکل زیر را به عنوان کرنل انتخاب کرد:
 - `MORPH_RECT`: مستطیل
 - `MORPH_CROSS`: صلیب
 - `MORPH_ELLIPSE`: بیضی

سپس فقط باید اندازه و مختصات لنگر را انتخاب کرد. اگر مختصات لنگر مشخص نشود، به صورت پیش‌فرض نقطه مرکزی

کرنل به عنوان محل لنگر در نظر گرفته می‌شود.

تابع Dilation (خط ۸۴):

کد این تابع بسیار مشابه با کد تابع Erosion است. اینجا هم باید نوع کرنل، مختصات لنگر و اندازه کرنل را مشخص کرد.

۴,۱,۳ خروجی



۴,۲ تبدیل های مورفولوژی پیچیده

در این بخش به بررسی نحوه اعمال عملیات های مورفولوژی پیچیده مثل باز کردن^{۶۵}، بستن^{۶۶}، گرادیان به روش مورفولوژی، تاپ هت^{۶۷} و بلک هت^{۶۸} می پردازیم. تمام این اعمال بر پایه عملیات فرسایش و گشايش به وجود آمده اند. در ادامه توضیح مختصری در مورد هر کدام از این اعمال می دهیم.

باز کردن:

از طریق گشايش گرفتن از فرسایش یافته یک تصویر به دست می آید:

$$dst = open(src, element) = dilate(erode(src, element))$$

Opening^{۶۹}

Closing^{۷۰}

Top Hat^{۷۱}

Black Hat^{۷۲}

این عمل برای از بین بردن اشیای کوچک مناسب است (بافت اینکه اشیا به رنگ روشن و روی زمینه تیره قرار داشته باشند). به شکل زیر نگاه کنید. تصویر سمت چپ تصویر ورودی و تصویر سمت راست نتیجه عملیات باز کردن است. همانطور که می‌بینید ناحیه‌های کوچک بین حرف ناپدید شده‌اند.



نتیجه اعمال باز کردن روی تصویر سمت چپ

بستن:

از طریق فرسایش گرفتن از گشايش یافته یک تصویر به دست می‌آید:

$$dst = \text{close}(src, element) = \text{erode}(\text{dilate}(src, element))$$

این عمل برای از بین بردن حفره‌های کوچک مناسب است (نواحی تیره).



نتیجه اعمال بستن روی تصویر سمت چپ

گرادیان به روش مورفولوژی:

از اختلاف بین گشايش و فرسایش یک تصویر به دست می‌آید:

$$dst = \text{morphgrad}(src, element) = \text{dilate}(src, element) - \text{erode}(src, element)$$

این عمل برای پیدا کردن محیط یک شیء مناسب است.



نتیجه اعمال گرادیان به روش مورفولوژی روی تصویر سمت چپ

تاب هَت:

از اختلاف بین یک تصویر با باز شده آن به دست می‌آید:

$$dst = \text{tophat}(src, element) = src - \text{open}(src, element)$$



نتیجه اعمال تاب هَت روی تصویر سمت چپ

بلک هَت:

از اختلاف بین بسته شده یک تصویر با خودش به دست می‌آید:

$$dst = \text{blackhat}(src, element) = \text{close}(src, element) - src$$



نتیجه اعمال بلک هَت روی تصویر سمت چپ

```

1 #include "opencv2/imgproc/imgproc.hpp"
2 #include "opencv2/highgui/highgui.hpp"
3 #include <stdlib.h>
4 #include <stdio.h>
5
6 using namespace cv;
7
8 /// Global variables
9 Mat src, dst;
10
11 int morph_elem = 0;
12 int morph_size = 0;
13 int morph_operator = 0;
14 int const max_operator = 4;
15 int const max_elem = 2;
16 int const max_kernel_size = 21;
17
18 char* window_name = "Morphology Transformations Demo";
19
20 /** Function Headers */
21 void Morphology_Operations( int, void* );
22
23 /** @function main */
24 int main( int argc, char** argv )
25 {
26     /// Load an image
27     src = imread( argv[1] );
28
29     if( !src.data )
30     { return -1; }
31
32     /// Create window
33     namedWindow( window_name, CV_WINDOW_AUTOSIZE );
34
35     /// Create Trackbar to select Morphology operation
36     createTrackbar("Operator:\n 0: Opening - 1: Closing \n 2: Gradient -\n 3: Top Hat \n 4: Black Hat", window_name, &morph_operator,
37     max_operator, Morphology_Operations );
38
39     /// Create Trackbar to select kernel type
40     createTrackbar( "Element:\n 0: Rect - 1: Cross - 2: Ellipse",
41     window_name,
42             &morph_elem, max_elem,
43             Morphology_Operations );
44
45     /// Create Trackbar to choose kernel size
46     createTrackbar( "Kernel size:\n 2n +1", window_name,
47             &morph_size, max_kernel_size,
48             Morphology_Operations );
49
50     /// Default start
51     Morphology_Operations( 0, 0 );
52
53
54     waitKey(0);
55     return 0;
56 }
57

```

```

58  /**
59   * @function Morphology_Operations
60   */
61 void Morphology_Operations( int, void* )
62 {
63   // Since MORPH_X : 2,3,4,5 and 6
64   int operation = morph_operator + 2;
65
66   Mat element = getStructuringElement( morph_elem, Size( 2*morph_size
67 + 1, 2*morph_size+1 ), Point( morph_size, morph_size ) );
68
69   /// Apply the specified morphology operation
70   morphologyEx( src, dst, operation, element );
71   imshow( window_name, dst );
72 }
73

```

۴,۲,۲ توضیح

در خط ۷۱ از تابع MorphologyEx برای انجام عملیات مورفولوژی روی تصویر ورودی استفاده می‌شود. این تابع چهار آرگومان دارد:

- :تصویر ورودی src
- :تصویر خروجی dst
- :نوع تبدیل مورفولوژی‌ای که باید اعمال شود. از پنج مقدار زیر می‌توان استفاده کرد:
 - بازکردن: 2
 - بستن: 3
 - گرادیان: 4
 - تاپ هت: 5
 - بلک هت: 6

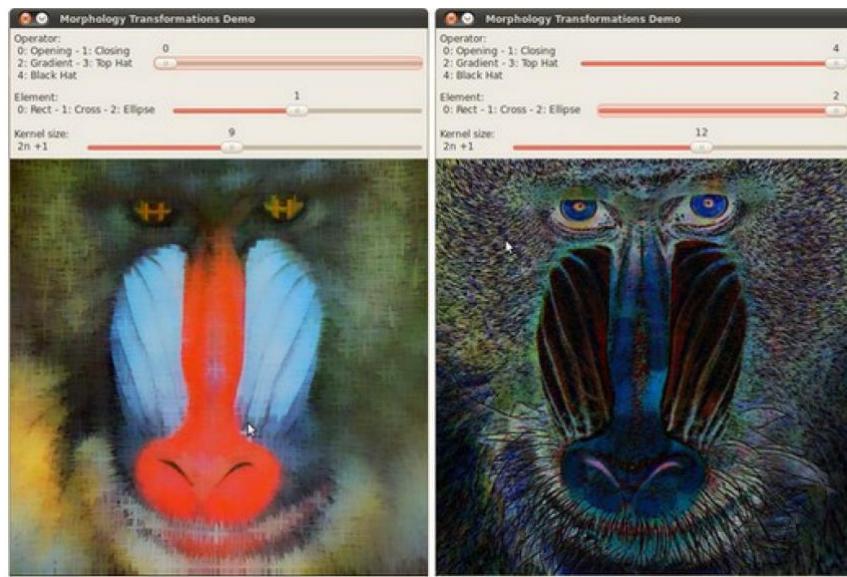
به خاطر اینکه مقدارها بین دو و شش است، در خط ۶۴ دو واحد به مقدار گرفته شده از ترکبار اضافه می‌کنیم.

▪:کرنل مورد استفاده است. از تابع getStructuringElement برای ساخت آن استفاده شده است.

۴,۲,۳ خروجی



تصویر ورودی



نمایی از برنامه – سمت راست عمل بلک هت و سمت چپ عمل باز کردن

۵ فصل پنجم-شناصایی الگو

۱,۵ مقایسه بافت‌نگارها

برای مقایسه دو بافت‌نگار H_1 و H_2 باید ابتدا معیاری برای بیان میزان شباهت دو بافت‌نگار انتخاب کرد. در این سی وی برای مقایسه بافت‌نگارها ازتابع comapreHist استفاده می‌شود. این تابع سه معیار شباهت در اختیار ما قرار می‌دهد:

۱. همبستگی^{۶۹}: (CV_COMP_CORREL)

$$d(H_1, H_2) = \frac{\sum_I (H_1(I) - H'_1)(H_2(I) - H'_2)}{\sqrt{\sum_I (H_1(I) - H'_1)^2} \sqrt{\sum_I (H_2(I) - H'_2)^2}}$$

که

$$H'_k = \frac{1}{N} \sum_J H_k(J)$$

و N هم تعداد سطلهای بافت‌نگار است.

۲. کای-اسکوئر^{۷۰}: (CV_COMP_CHISQR)

$$d(H_1, H_2) = \sum_I \frac{(H_1(I) - H_2(I))^2}{H_1(I)}$$

۳. اشتراک^{۷۱}: (CV_COMP_INTERSECT)

$$d(H_1, H_2) = \sum_I \min(H_1(I), H_2(I))$$

۴. فاصله باتاچاریا^{۷۲}: (CV_COMP_BHATTACHARYYA)

$$d(H_1, H_2) = \sqrt{1 - \frac{1}{\sqrt{H'_1 H'_2 N^2}} \sum_I \sqrt{H_1(I) \cdot H_2(I)}}$$

۵,۱,۱

```

1 #include "opencv2/highgui/highgui.hpp"
2 #include "opencv2/imgproc/imgproc.hpp"
3 #include <iostream>
4 #include <stdio.h>
5
6 using namespace std;
7 using namespace cv;
8
9 /**
10  * @function main
11 */
12 int main( int argc, char** argv )
13 {
14     Mat src_base, hsv_base;
15     Mat src_test1, hsv_test1;
16     Mat src_test2, hsv_test2;
```

Correlation^{۶۹}

Chi-Square^{۷۰}

Intersection^{۷۱}

Bhattacharyya^{۷۲}

```

17     Mat hsv_half_down;
18
19     /// Load three images with different environment settings
20     if( argc < 4 )
21     {
22         printf("** Error. Usage: ./compareHist_Demo
23 <image_settings0> <image_setting1> <image_settings2>\n");
24         return -1;
25     }
26
27     src_base = imread( argv[1], 1 );
28     src_test1 = imread( argv[2], 1 );
29     src_test2 = imread( argv[3], 1 );
30
31     /// Convert to HSV
32     cvtColor( src_base, hsv_base, COLOR_BGR2HSV );
33     cvtColor( src_test1, hsv_test1, COLOR_BGR2HSV );
34     cvtColor( src_test2, hsv_test2, COLOR_BGR2HSV );
35
36     hsv_half_down = hsv_base( Range( hsv_base.rows/2, hsv_base.rows
37 - 1 ), Range( 0, hsv_base.cols - 1 ) );
38
39     /// Using 50 bins for hue and 60 for saturation
40     int h_bins = 50; int s_bins = 60;
41     int histSize[] = { h_bins, s_bins };
42
43     // hue varies from 0 to 179, saturation from 0 to 255
44     float h_ranges[] = { 0, 180 };
45     float s_ranges[] = { 0, 256 };
46
47     const float* ranges[] = { h_ranges, s_ranges };
48
49     // Use the 0-th and 1-st channels
50     int channels[] = { 0, 1 };
51
52
53     /// Histograms
54     MatND hist_base;
55     MatND hist_half_down;
56     MatND hist_test1;
57     MatND hist_test2;
58
59     /// Calculate the histograms for the HSV images
60     calcHist( &hsv_base, 1, channels, Mat(), hist_base, 2, histSize,
61     ranges, true, false );
62     normalize( hist_base, hist_base, 0, 1, NORM_MINMAX, -1, Mat() );
63
64     calcHist( &hsv_half_down, 1, channels, Mat(), hist_half_down, 2,
65     histSize, ranges, true, false );
66     normalize( hist_half_down, hist_half_down, 0, 1, NORM_MINMAX, -1,
67     Mat() );
68
69     calcHist( &hsv_test1, 1, channels, Mat(), hist_test1, 2,
70     histSize, ranges, true, false );
71     normalize( hist_test1, hist_test1, 0, 1, NORM_MINMAX, -1, Mat()
72 );
73
74     calcHist( &hsv_test2, 1, channels, Mat(), hist_test2, 2,
75     histSize, ranges, true, false );
76     normalize( hist_test2, hist_test2, 0, 1, NORM_MINMAX, -1, Mat()
77 );

```

```

78
79     /// Apply the histogram comparison methods
80     for( int i = 0; i < 4; i++ )
81     {
82         int compare_method = i;
83         double base_base = compareHist( hist_base, hist_base,
84 compare_method );
85         double base_half = compareHist( hist_base, hist_half_down,
86 compare_method );
87         double base_test1 = compareHist( hist_base, hist_test1,
88 compare_method );
89         double base_test2 = compareHist( hist_base, hist_test2,
90 compare_method );
91
92         printf( " Method [ %d ] Perfect, Base-Half, Base-Test(1) ,
93 Base-Test(2) : %f, %f, %f, %f \n", i, base_base, base_half ,
94 base_test1, base_test2 );
95     }
96
97     printf( "Done \n" );
98
99     return 0;
100 }
101
102

```

۵,۱,۲ توضیح

در خطوط ۳۲ تا ۳۴، تصاویر پایه را به فرمت HSV تبدیل می‌کنیم.

در خط ۳۶ یک ماتریس که به نصف پایینی تصویر پایه اشاره می‌کند تعریف می‌کنیم (در فرمت HSV).

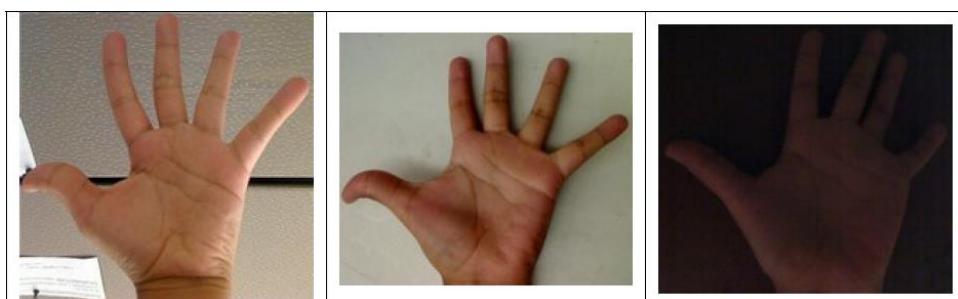
در خطوط ۳۹ تا ۵۰، آرگومان‌های مورد نیاز برای حساب کردن بافت‌نگارها را مقدار دهی می‌کنیم.

در خطوط ۵۹ تا ۷۷، بافت‌نگار تصاویر HSV را حساب و نرمال می‌کنیم.

در خطوط ۸۰ تا ۹۵، به ترتیب چهار بارتابع مقایسه را بین بافت‌نگار تصویر پایه و بافت‌نگار تصاویر دیگر اعمال می‌کنیم.

۵,۱,۳ خروجی

سه تصویر زیر را به عنوان ورودی به کار می‌بریم:



تصویر ورودی به برنامه-تصویر سمت چپ، تصویر پایه است

تصویر سمت چپ را به عنوان تصویر پایه و دو تصویر دیگر را به عنوان تصاویر آزمایشی در نظر می‌گیریم. همچنین توجه کنید که تصویر پایه هم نسبت به خودش و هم نسبت به نصفهٔ پایینی اش، مقایسه می‌شود.

وقتی بافتنگار تصویر پایه را با خودش مقایسه می‌کنیم انتظار جواب کامل داریم. همچنین هنگامی که این بافتنگار را با بافتنگار نصفهٔ پایین اش مقایسه می‌کنیم، از آنجا که هر دو تصویر از یک منبع هستند، جواب باید بیانگر میزان زیادی از شباهت باشد. به خاطر اینکه شرایط نور در دو تصویر آزمایشی با تصویر پایه متفاوت است، نتیجه مقایسه بافتنگار آن‌ها با بافتنگار تصویر پایه نباید خیلی خوب باشد.

خروجی برنامه به صورت زیر است:

METHOD	BASE-BASE	BASE-HALF	BASE-TEST1	BASE-TEST2
CORRELATION	1.000000	0.93076	0.18207	0.12044
CHI-SQUARE	0.000000	4.94046	21.1845	49.2734
INTERSECTION	24.39154	14.9598	3.88902	5.77508
BHATTACHARYYA	0.000000	0.22260	0.64657	0.80186

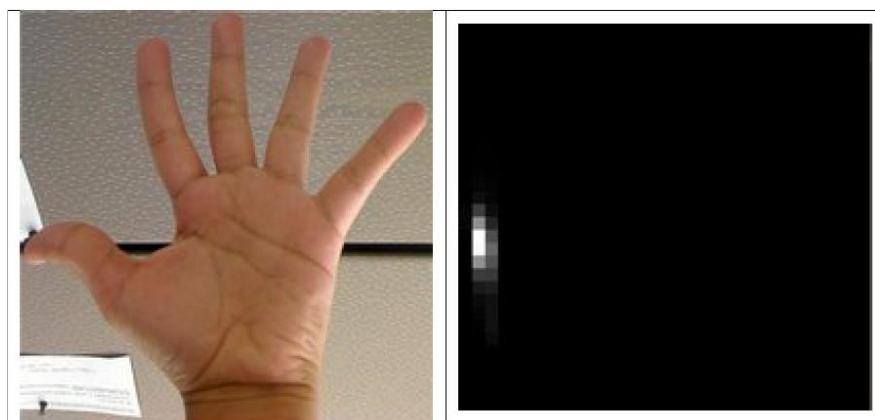
در روش‌های همبستگی و اشتراکی هر چه نتیجه بزرگتر باشد، شباهت بیشتر است. همانطور که انتظار داشتیم نتیجه مقایسهٔ پایه با خودش، بزرگ‌ترین مقدار را دارد. همچنین نتیجه مقایسه بافتنگار پایه با بافتنگار تصویر نصفه، دومین بزرگ‌ترین مقدار است. در دو روش دیگر، هر چه نتیجه کوچکتر باشد شباهت بیشتر است.

۵.۲ بَك پروجکشن

بَك پروجکشن روشهای ارزیابی میزان تطبیق توزیع بافتنگار یک تصویر با توزیع بافتنگار مبنا است.

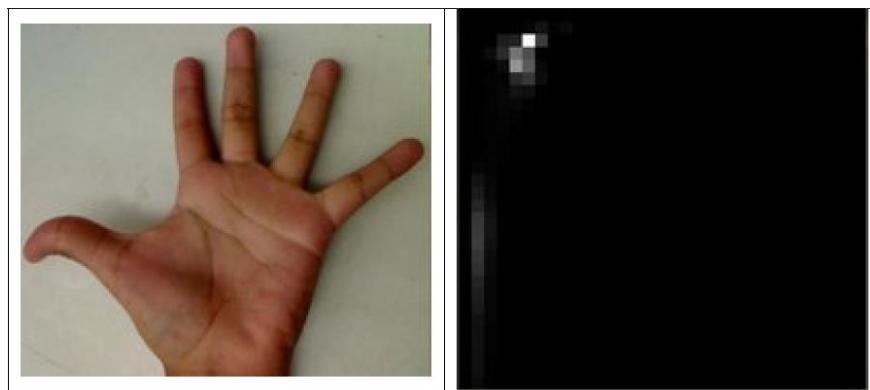
به بیان ساده‌تر، به منظور اجرای بَك پروجکشن، بافتنگار ویژگی مورد نظر را حساب می‌کنیم و سپس از آن برای پیدا کردن این ویژگی در تصاویر جدید استفاده می‌کنیم. مثلاً اگر اگر بافتنگار رنگ پوست را داشته باشیم، آن وقت می‌توانیم از آن برای پیدا کردن نواحی‌ای که احتمال دارد پوست باشد، در تصاویر جدید استفاده کنیم.

فرض کنید از روی تصویر زیر یک بافتنگار S-H مربوط به پوست به دست آورده‌ایم. این بافتنگار را بافتنگار مبنا قرار می‌دهیم. برای اینکه فقط بافتنگار قسمت پوست را به دست آوریم، چند ماسک به تصویر اعمال می‌کنیم و سپس بافتنگار را حساب می‌کنیم.



سمت چپ تصویر دست و سمت راست تصویر بافتنگار آن

حالا فرض کنید تصویر زیر را به عنوان تصویر آزمایشی به کار ببریم:



سمت چپ تصویر آزمایشی دست و سمت راست تصویر بافت‌نگار آن

می‌خواهیم با استفاده از بافت‌نگار مینا، نواحی پوست در تصویر آزمایشی را پیدا کنیم. برای این کار به صورت زیر عمل می‌کنیم:

۱. هر کدام از پیکسل‌های تصویر آزمایشی (یعنی (j, i)) را در سطله متناظر با آن پیکسل ($s_{i,j}, h_{i,j}$) قرار می‌دهیم.
۲. سطله ($s_{i,j}, h_{i,j}$) در بافت‌نگار مینا را جستجو می‌کنیم و مقدار آن را می‌خوانیم.
۳. این مقدار را در مکان $((j, i))$ تصویر جدید ذخیره می‌کنیم (این تصویر همان نتیجه بک پروجکشن است). همچنین برای اینکه تصویر نهایی قابل دیدن باشد، باید آن را نرمال سازی کنیم.

با انجام مراحل بالا روی تصویر آزمایشی نتیجه زیر را به دست می‌آوریم:



نتیجه بک پروجکشن روی تصویر آزمایشی دست

مقادیری که در تصویر نتیجه ذخیره شده‌اند نشان دهنده احتمال تعلق آن پیکسل به ناحیه مطلوب هستند (طبق بافت‌نگار مینا). مثلاً در مورد تصویر بالا، هر چه ناحیه‌ای روشن‌تر باشد احتمالش بیشتر است که به یک ناحیه پوست تعلق داشته باشد، در حالی که ناحیه‌های تیره احتمال کمتری دارند.

۵.۲.۱ گد

```

1 #include "opencv2/imgproc/imgproc.hpp"
2 #include "opencv2/highgui/highgui.hpp"
3
4 #include <iostream>

```

```

5
6     using namespace cv;
7     using namespace std;
8
9     /// Global Variables
10    Mat src; Mat hsv; Mat hue;
11    int bins = 25;
12
13    /// Function Headers
14    void Hist_and_Backproj(int, void* );
15
16    /** @function main */
17    int main( int argc, char** argv )
18    {
19        /// Read the image
20        src = imread( argv[1], 1 );
21        /// Transform it to HSV
22        cvtColor( src, hsv, CV_BGR2HSV );
23
24        /// Use only the Hue value
25        hue.create( hsv.size(), hsv.depth() );
26        int ch[] = { 0, 0 };
27        mixChannels( &hsv, 1, &hue, 1, ch, 1 );
28
29        /// Create Trackbar to enter the number of bins
30        char* window_image = "Source image";
31        namedWindow( window_image, CV_WINDOW_AUTOSIZE );
32        createTrackbar("Hue bins: ", window_image, &bins, 180,
33        Hist_and_Backproj );
34        Hist_and_Backproj(0, 0);
35
36        /// Show the image
37        imshow( window_image, src );
38
39        /// Wait until user exits the program
40        waitKey(0);
41        return 0;
42    }
43
44
45    /**
46     * @function Hist_and_Backproj
47     * @brief Callback to Trackbar
48     */
49    void Hist_and_Backproj(int, void* )
50    {
51        MatND hist;
52        int histSize = MAX( bins, 2 );
53        float hue_range[] = { 0, 180 };
54        const float* ranges = { hue_range };
55
56        /// Get the Histogram and normalize it
57        calcHist( &hue, 1, 0, Mat(), hist, 1, &histSize, &ranges, true,
58        false );
59        normalize( hist, hist, 0, 255, NORM_MINMAX, -1, Mat() );
60
61        /// Get Backprojection
62        MatND backproj;
63        calcBackProject( &hue, 1, 0, hist, backproj, &ranges, 1, true );
64
65        /// Draw the backproj

```

```

66     imshow( "BackProj", backproj );
67
68     /// Draw the histogram
69     int w = 400; int h = 400;
70     int bin_w = cvRound( (double) w / histSize );
71     Mat histImg = Mat::zeros( w, h, CV_8UC3 );
72
73     for( int i = 0; i < bins; i ++ )
74     { rectangle( histImg, Point( i*bin_w, h ), Point( (i+1)*bin_w, h
75 - cvRound( hist.at<float>(i)*h/255.0 ) ), Scalar( 0, 0, 255 ), -1 );
76 }
77
78     imshow( "Histogram", histImg );
79 }
80

```

۵,۲,۲ توضیح

در خط ۲۲ تصویر ورودی را به HSV تبدیل می‌کنیم.

در اینجا فقط از مقدار Hue برای محاسبه بافت‌نگار استفاده می‌کنیم (خطوط ۲۵ تا ۲۷). همانطور که می‌بینید، از تابع mixChannels برای گرفتن کanal صفر (یعنی کanal Hue) استفاده کرده‌ایم. پارامترهای این تابع به شرح زیر هستند:

۱. آرایه مبدأ که کanal‌ها از آن کپی می‌شوند.
۲. تعداد کanal‌های آرایه مبدأ.
۳. آرایه مقصد که کanal‌ها در آن کپی می‌شوند.
۴. تعداد کanal‌های آرایه مقصد.
۵. آرایه از جفت اندیس‌ها که نشان دهنده طریقه کپی کanal‌ها از آرایه مبدأ به آرایه مقصد است. در اینجا کanal صفر &hue در کanal صفر &hue کپی می‌شود.
۶. تعداد جفت اندیس‌ها است.

در سطر ۴۹، تابع Hist_and_Backproj قرار دارد. این تابع آرگومان‌های مورد نیاز تابع calcHist را مقدار دهی می‌کند. تعداد سطلهای هم از طریق ترکبار ارسال می‌شوند.

در خطوط ۵۷ و ۵۹، ابتدا با استفاده از تابع calcHist بافت‌نگار را حساب و سپس آن را با استفاده از تابع normalize در بازه ۰ تا ۲۵۵ نرمال می‌کنیم.

در خط ۶۳ بک پروجکشن تصویر hue را با استفاده از تابع calcBackProject به دست می‌آوریم. این تابع ۸ آرگومان دارد که شما با همه آنها آشنا هستید (مشابه همان‌هایی هستند که برای محاسبه بافت‌نگار به کار می‌بردیم)، فقط ماتریس backproj اضافه شده است که نتیجه بک پروجکشن تصویر ورودی (یعنی &hue) در آن ذخیره می‌شود.

۵,۲,۳ خروجی

به عنوان ورودی از تصویر یک دست استفاده می‌کنیم. خروجی به صورت زیر خواهد بود. می‌توانید مقدار سطلهای را تغییر دهید و تأثیر آن را روی خروجی ببینید.



سمت چپ: تصویر ورودی - وسط: بافت‌نگار ناحیه دست - سمت راست: یک پروجکشن تصویر ورودی

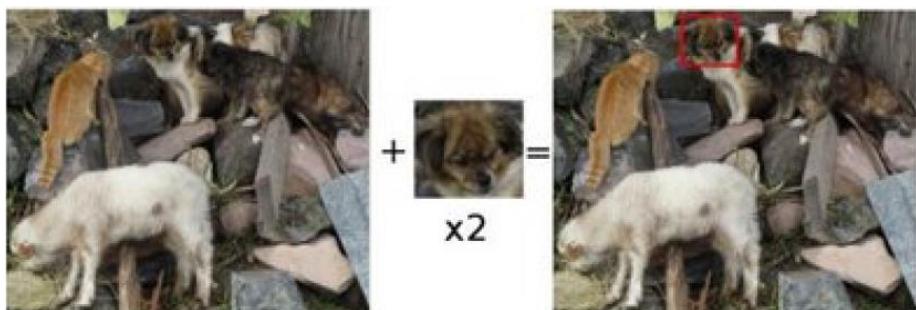
۵,۳ تطبیق الگو

تطبیق الگو تکنیکی برای پیدا کردن ناحیه‌هایی از یک تصویر است که مطابق (مشابه) با یک تصویر الگو (تکه تصویر) هستند.

برای انجام این عمل به دو عنصر اولیه نیاز است:

۱. تصویر منبع (I): تصویری که می‌خواهیم یک تطبیق در آن پیدا کنیم.
۲. تصویر الگو (T): تکه تصویری که با تصویر منبع مقایسه می‌شود.

هدف پیدا کردن بزرگترین ناحیه تطبیق است.



دو عنصر مورد نیاز برای انجام یک تطبیق الگو

برای پیدا کردن ناحیه تطبیق باید تصویر الگو را روی تصویر منبع حرکت دهیم و هر قسمت از تصویر منبع را با تصویر الگو مقایسه کنیم.



روند پیدا کردن یک تطبیق در تصویر منبع

منظور از حرکت دادن این است که تکه تصویر را هر بار یک پیکسل حرکت دهیم (از چپ به راست و از بالا به پایین). به هر حال، در هر توقف یک مقدار حساب می‌شود که مشخص می‌کند تطبیق در آن محل چقدر خوب یا بد بوده است (به زبان دیگر بیان می‌کند تکه تصویر چقدر شبیه به آن ناحیه است).

به ازای هر محل T روی A ، مقدار حساب شده را در ماتریس R (ماتریس نتیجه) ذخیره می‌کنیم. هر محل (x,y) در R حاوی مقدار حساب شده برای آن محل است.



مثالی از ماتریس R

تصویر بالا ماتریس R است که با حرکت دادن الگو روی تصویر ورودی با معیار `TM_CCORR_NORMED` به دست آمده است. هر چه یک محل روش‌تر باشد به این معنی است که شباهتش بیشتر است. همانطور که می‌بینید محلی که با دایره قرمز مشخص شده، احتمالاً همان محلی است که بیشترین مقدار را دارد؛ پس آن محل (یعنی مستطیلی که گوشه بالا و سمت چپ آن روی این نقطه قرار دارد و طول و عرضش هم اندازه با طول و عرض الگو است) را به عنوان ناحیه تطبیق در نظر می‌گیریم.

در عمل برای پیدا کردن محل عنصر بیشنه (یا کمینه، بسته به نوع معیار استفاده شده)، در ماتریس R ، از تابع `minMaxLoc` استفاده می‌کنیم.

اُسی وی عملیات تطبیق الگو را در تابع `matchTemplate` پیاده سازی کرده است. روش‌های موجود به صورت زیر هستند:

`CV_TM_SQDIFF` .۱

$$R(x, y) = \sum_{x', y'} (T(x', y') - I(x + x', y + y'))^2$$

CV_TM_SQDIFF_NORMED ↗

$$R(x, y) = \frac{\sum_{x', y'} (T(x', y') - I(x + x', y + y'))^2}{\sqrt{\sum_{x', y'} T(x', y')^2 \cdot \sum_{x', y'} I(x + x', y + y')^2}}$$

CV_TM_CCORR ↗

$$R(x, y) = \sum_{x', y'} (T(x', y') \cdot I(x + x', y + y'))$$

CV_TM_CCORR_NORMED ↗

$$R(x, y) = \frac{\sum_{x', y'} T(x', y') \cdot I'(x + x', y + y')}{\sqrt{\sum_{x', y'} T(x', y')^2 \cdot \sum_{x', y'} I(x + x', y + y')^2}}$$

CV_TM_CCOEFF ↘

$$R(x, y) = \sum_{x', y'} (T'(x', y') \cdot I(x + x', y + y'))$$

↳

$$T'(x', y') = T(x', y') - \frac{1}{w \cdot h} \cdot \sum_{x'', y''} T(x'', y'')$$

$$I'(x + x', y + y') = I(x + x', y + y') - \frac{1}{w \cdot h} \cdot \sum_{x'', y''} I(x + x'', y + y'')$$

CV_TM_CCOEFF_NORMED ↘

$$R(x, y) = \frac{\sum_{x', y'} (T'(x', y') \cdot I'(x + x', y + y'))}{\sqrt{\sum_{x', y'} T'(x', y')^2 \cdot \sum_{x', y'} I'(x + x', y + y')^2}}$$

☞ 5,3,1

```

1 #include "opencv2/highgui/highgui.hpp"
2 #include "opencv2/imgproc/imgproc.hpp"
3 #include <iostream>
4 #include <stdio.h>
5
6 using namespace std;
7 using namespace cv;
8
9 /// Global Variables
10 Mat img; Mat templ; Mat result;
11 char* image_window = "Source Image";
12 char* result_window = "Result window";
13
14 int match_method;
15 int max_Trackbar = 5;
16
17 /// Function Headers
18 void MatchingMethod( int, void* );
19
20 /** @function main */
21 int main( int argc, char** argv )
22 {
23     /// Load image and template

```

```

24     img = imread( argv[1], 1 );
25     templ = imread( argv[2], 1 );
26
27     /// Create windows
28     namedWindow( image_window, CV_WINDOW_AUTOSIZE );
29     namedWindow( result_window, CV_WINDOW_AUTOSIZE );
30
31     /// Create Trackbar
32     char* trackbar_label = "Method: \n 0: SQDIFF \n 1: SQDIFF NORMED
33 \n 2: TM CCORR \n 3: TM CCORR NORMED \n 4: TM COEFF \n 5: TM COEFF
34 NORMED";
35     createTrackbar( trackbar_label, image_window, &match_method,
36 max_Trackbar, MatchingMethod );
37
38     MatchingMethod( 0, 0 );
39
40     waitKey(0);
41     return 0;
42 }
43
44 /**
45 * @function MatchingMethod
46 * @brief Trackbar callback
47 */
48 void MatchingMethod( int, void* )
49 {
50     /// Source image to display
51     Mat img_display;
52     img.copyTo( img_display );
53
54     /// Create the result matrix
55     int result_cols = img.cols - templ.cols + 1;
56     int result_rows = img.rows - templ.rows + 1;
57
58     result.create( result_cols, result_rows, CV_32FC1 );
59
60     /// Do the Matching and Normalize
61     matchTemplate( img, templ, result, match_method );
62     normalize( result, result, 0, 1, NORM_MINMAX, -1, Mat() );
63
64     /// Localizing the best match with minMaxLoc
65     double minVal; double maxVal; Point minLoc; Point maxLoc;
66     Point matchLoc;
67
68     minMaxLoc( result, &minVal, &maxVal, &minLoc, &maxLoc, Mat() );
69
70     /// For SQDIFF and SQDIFF_NORMED, the best matches are lower
71 values. For all the other methods, the higher the better
72     if( match_method == CV_TM_SQDIFF || match_method ==
73 CV_TM_SQDIFF_NORMED )
74     { matchLoc = minLoc; }
75     else
76     { matchLoc = maxLoc; }
77
78     /// Show me what you got
79     rectangle( img_display, matchLoc, Point( matchLoc.x + templ.cols
80 , matchLoc.y + templ.rows ), Scalar::all(0), 2, 8, 0 );
81     rectangle( result, matchLoc, Point( matchLoc.x + templ.cols ,
82 matchLoc.y + templ.rows ), Scalar::all(0), 2, 8, 0 );
83
84     imshow( image_window, img_display );

```

```

85     imshow( result_window, result );
86
87     return;
88 }
89

```

۵,۳,۲ توضیح

در خط ۶۱ با استفاده از تابع `matchTemplate` عملیات تطبیق الگو را اجرا می‌کنیم. آرگومان‌های این تابع تصویر ورودی (*I*، الگو *T*)، ماتریس نتیجه (*R*) و نوع تطبیق (که از ترکبار به دست می‌آید) هستند.

در خط ۶۲ نتیجه را نرمال می‌کیم.

در خط ۶۴، با استفاده از تابع `minMaxLoc` محل مقادیر بیشینه و کمینه ماتریس *R* را پیدا می‌کنیم. این تابع ۶ آرگومان دارد که به شرح زیر هستند:

- .۱ آرایه ورودی (آرایه‌ای که مورد جستجو قرار می‌گیرد).
- .۲ متغیرهایی برای ذخیره مقادیر کمینه و بیشنه موجود در *result*.
- .۳ محل مقادیر بیشینه و کمینه در این نقاط قرار می‌گیرند.
- .۴ ماسک آرایه ورودی.

در دو روش تطبیق `CV_SQDIFF` و `CV_SQDIFF_NORMED`، بهترین تطبیق‌ها دارای کمترین مقادیر هستند در حالی که در دیگر روش‌ها بهترین تطبیق‌ها دارای بیشترین مقادیر هستند. پس برای سادگی، در خطوط ۷۳ تا ۷۶ بسته به روش مورد استفاده، متغیر `matchLoc` را مقدار دهی می‌کنیم.

۵,۳,۳ خروجی



تصویر الگو



تصویر ورودی

ماتریس‌های نتیجه زیر تولید می‌شوند (ردیف اول روش‌های `SQDIFF` و `CCOEFF`، ردیف دوم هم همین روش‌ها، ولی نسخه نرمال شده آنها هستند). در ستون اول، تاریکترین محل، بهترین تطبیق است و در دو ستون دیگر روش‌ترین محل، بهترین تطبیق است.



ماتریس‌های نتیجه تولید شده توسط فرایند تطبیق الگو

نتیجه نهایی در شکل زیر نشان داده است. دقت کنید که CCDEFF و CCORR جواب نادرستی را به عنوان پاسخ داده‌اند؛ البته نسخه نرمال شده آنها درست عمل کرده است. این ممکن است به خاطر این واقعیت باشد که ما فقط اولین بزرگترین تطبیق را در نظر می‌گیریم و به تطبیق‌های بزرگ دیگر کاری نداریم.



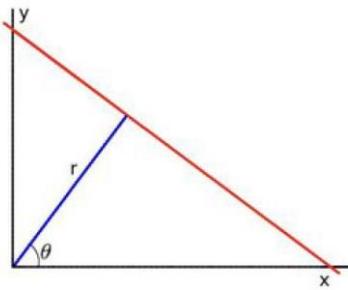
نتیجه نهایی

۵.۴ تبدیل خط هُو

از تبدیل خط هُو^{۷۳} برای کشف خطوط استفاده می‌شود. برای اعمال این تبدیل نیاز است که ابتدا تصویر مورد نظر لبیابی شود. همانطور که می‌دانید، می‌توان خط را به وسیلهٔ دو پارامتر مشخص کرد. برای مثال:

- a. در مختصات کارتزین پارامترها به صورت (m, b) هستند.

b. در مختصات قطبی پارامترها به صورت (r, θ) هستند.



یک خط و نحوه نمایش آن در مختصات کارتزین و قطبی

در تبدیل هُو از فرم قطبی استفاده می‌شود، بنابراین معادله خط را می‌توان به شکل زیر نوشت:

$$y = \left(-\frac{\cos \theta}{\sin \theta} \right) x + \left(\frac{r}{\sin \theta} \right)$$

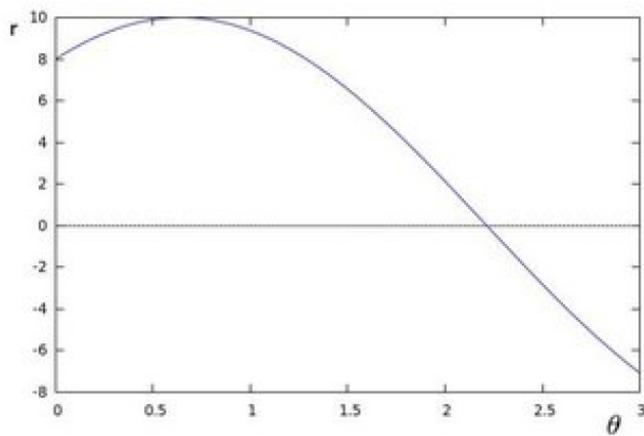
که در آن $r = x \cos \theta + y \sin \theta$ است.

به طورکلی، به ازای هر نقطه به صورت (x_0, y_0) می‌توانیم خانواده خطوطی که از آن نقطه می‌گذرند را به صورت زیر تعریف کنیم:

$$r = x_0 \cos \theta + y_0 \sin \theta$$

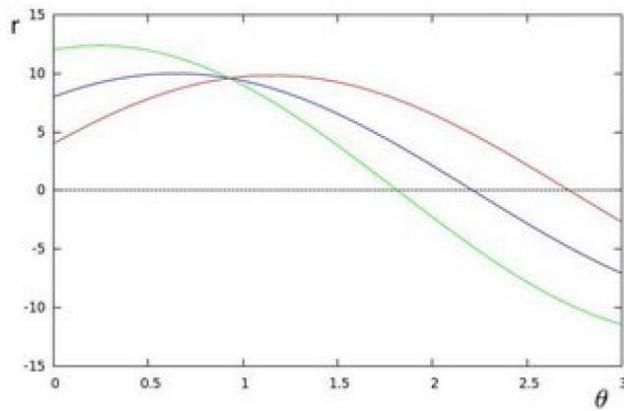
هر جفت (r_θ, θ) ، نشان دهنده یک خط است که از نقطه (x_0, y_0) می‌گذرد.

اگر خانواده تمام خطوطی که از نقطه (x_0, y_0) می‌گذرند را در صفحه $r - \theta$ رسم کنیم، یک شکل سینوسی به وجود می‌آید. مثلاً به ازای $y_0 = 6$ و $x_0 = 8$ شکل زیر به وجود می‌آید:



جفت (r_θ, θ) تمام خطوطی که از نقطه $(8, 6)$ می‌گذرند

می‌توان مشابه کار بالا را برای تمام نقاط موجود در یک تصویر انجام داد. اگر منحنی دو نقطه مختلف در صفحه $r - \theta$ هم دیگر را قطع کنند، به این معنی است که آن دو نقطه روی یک خط قرار دارند. مثلاً اگر به دنبال نقطه مثال قبل، منحنی‌های نقاط $(4, 9)$ و $(12, 3)$ را بکشیم، نمودار حاصل به شکل زیر در می‌آید:



رسم منحنی خطوطی که از سه نقطه (8, 6)، (4, 9) و (12, 3) می‌گذرند

در این نمودار، سه منحنی یکدیگر را در نقطه $(0.925, 9.6)$ قطع کرده‌اند. این همان پارامترهای (θ, r) خطی است که نقاط $(6, 4)$ ، $(9, 4)$ و $(3, 12)$ روی آن قرار گرفته‌اند.

به صورت کلی می‌توان یک خط را با به دست آوردن نقاط برخورد بین منحنی‌ها کشف کرد. هر چه تعداد بیشتری منحنی در یک برخورد وجود داشته باشد، به این معنی است که خطی که نشان دهنده آن برخورد است، تعداد بیشتری نقطه دارد. به صورت کلی می‌توان با تعیین یک آستانه، حداقل تعداد مجاز برخوردهای مورد نیاز برای خط حساب کردن تعدادی نقطه را مشخص کرد. این همان کاری است که تبدیل هُو انجام می‌دهد. این تبدیل تعداد برخوردهای منحنی همه نقاط تصویر را به دست می‌آورد و اگر تعداد برخوردها بیشتر از آستانه بود، آن را به عنوان خط با پارامترهای (θ, r_θ) ، در نظر می‌گیرد.

در اسی وی دو نوع تبدیل خط هُو پیاده سازی شده است:

تبدیل خط استاندارد هُو:

بسیار شبیه همان چیزی است که توضیح دادیم. به عنوان خروجی، برداری از جفت‌های (θ, r_θ) را می‌دهد. این تبدیل در تابع **HoughLines** پیاده سازی شده است.

تبدیل خط احتمالی هُو:

این تبدیل بسیار بهینه‌تر از تبدیل خط استاندارد پیاده سازی شده است. به عنوان خروجی، نقاط انتهایی خطوط کشف شده را بر می‌گرداند (x_0, y_0, x_1, y_1) . این تبدیل در تابع **HoughLinesP** پیاده سازی شده است.

۵.۴.۱ کد

```

1 #include "opencv2/highgui/highgui.hpp"
2 #include "opencv2/imgproc/imgproc.hpp"
3
4 #include <iostream>
5
6 using namespace cv;
7 using namespace std;
8
9 void help()
10 {
11     cout << "\nThis program demonstrates line finding with the Hough
12 transform.\n"
13     "Usage:\n"
14 }
```

```

15             "./houghlines <image_name>, Default is pic1.jpg\n" <<
16     endl;
17 }
18
19 int main(int argc, char** argv)
20 {
21
22     Mat src = imread(argv[1], 0);
23     if(src.empty())
24     {
25         help();
26         cout << "can not open " << argv[1] << endl;
27         return -1;
28     }
29
30     Mat dst, cdst;
31     Canny(src, dst, 50, 200, 3);
32     cvtColor(dst, cdst, CV_GRAY2BGR);
33
34 #if 0
35     vector<Vec2f> lines;
36     HoughLines(dst, lines, 1, CV_PI/180, 100, 0, 0 );
37
38     for( size_t i = 0; i < lines.size(); i++ )
39     {
40         float rho = lines[i][0], theta = lines[i][1];
41         Point pt1, pt2;
42         double a = cos(theta), b = sin(theta);
43         double x0 = a*rho, y0 = b*rho;
44         pt1.x = cvRound(x0 + 1000*(-b));
45         pt1.y = cvRound(y0 + 1000*(a));
46         pt2.x = cvRound(x0 - 1000*(-b));
47         pt2.y = cvRound(y0 - 1000*(a));
48         line( cdst, pt1, pt2, Scalar(0,0,255), 3, CV_AA);
49     }
50 #else
51     vector<Vec4i> lines;
52     HoughLinesP(dst, lines, 1, CV_PI/180, 50, 50, 10 );
53     for( size_t i = 0; i < lines.size(); i++ )
54     {
55         Vec4i l = lines[i];
56         line( cdst, Point(l[0], l[1]), Point(l[2], l[3]),
57               Scalar(0,0,255), 3, CV_AA);
58     }
59 #endif
60     imshow("source", src);
61     imshow("detected lines", cdst);
62
63     waitKey();
64
65     return 0;
66 }
67 }
```

۵,۴,۲ توضیح

در خط ۳۱ با استفاده از لبه‌یاب گنجی، لبه‌های عکس را پیدا می‌کنیم.

حالا می‌توانیم تبدیل خط هُو را اعمال کنیم. هر دوتابع آسی وی که برای این کار هستند را توضیح می‌دهیم:

تبديل خط استاندارد هو:

ابتدا در خط ۳۶ تبدیل خط استاندارد را با استفاده از تابع HoughLines حساب می‌کنیم. این تابع ۷ آرگومان دارد که به شرح زیر هستند:

- .۱ **dst**: تصویر ورودی (که در حقیقت همان خروجی لبه‌یاب) است. این تصویر باید نوع سیاه و سفید باشد (اگر چه در حقیقت یک عکس باینری است).
- .۲ **lines**: برداری که پارامترهای (r, θ) خطوط کشف شده در آن قرار می‌گیرند.
- .۳ **rho**: وضوح پارامتر r بر حسب تعداد پیکسل‌ها است. در اینجا از ۱ پیکسل استفاده می‌کنیم.
- .۴ **theta**: وضوح پارامتر θ بر حسب رادیان است. در اینجا از 1 درجه استفاده می‌کنیم (که یعنی $CV_PI/180$).
- .۵ **threshold**: حداقل تعداد برخوردها برای در نظر گرفتن به عنوان خط است.
- .۶ **stn** و **srn**: به صورت پیش‌فرض صفر هستند.

سپس در خطوط ۳۸ تا ۴۹ نتیجه را با کشیدن تمام خطهای کشف شده، نشان می‌دهیم.

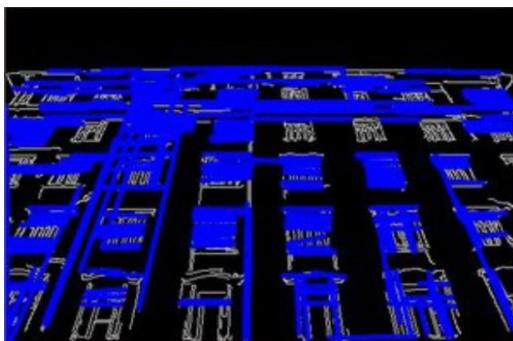
تبديل خط احتمالی هو:

ابتدا در خط ۵۲ تبدیل خط احتمالی را با استفاده از تابع HoughLinesP حساب می‌کنیم. این تابع ۷ آرگومان دارد که به شرح زیر هستند:

- .۱ **dst**: تصویر ورودی (که در حقیقت همان خروجی لبه‌یاب) است. این تصویر باید از نوع سیاه و سفید باشد (اگر چه در حقیقت یک عکس باینری است).
- .۲ **lines**: برداری که پارامترهای $(x_{start}, y_{start}, x_{end}, y_{end})$ خطوط کشف شده در آن قرار می‌گیرند.
- .۳ **rho**: وضوح پارامتر r بر حسب تعداد پیکسل‌ها است. در اینجا از ۱ پیکسل استفاده می‌کنیم.
- .۴ **theta**: وضوح پارامتر θ بر حسب رادیان است. در اینجا از 1 درجه استفاده می‌کنیم (که یعنی $CV_PI/180$).
- .۵ **threshold**: حداقل تعداد برخوردها برای در نظر گرفتن به عنوان خط است.
- .۶ **minLineLength**: حداقل تعداد نقطه‌هایی که می‌توانند یک خط را تشکیل دهند. خطوطی که کمتر از این تعداد نقطه دارند رد می‌شوند.
- .۷ **maxLineGap**: حداکثر فضای خالی بین دو نقطه که روی یک خط قرار دارند.

سپس در خطوط ۵۳ تا ۵۸ نتیجه را با کشیدن تمام خطهای کشف شده، نشان می‌دهیم.

۵,۴,۳ خروجی



نتیجه اعمال تبدیل خط احتمالی هو با آستانه ۲۰



تصویر ورودی

۵,۵ تبدیل دایره هو

تبدیل دایره هو بسیار شبیه به تبدیل خط هو که در بخش قبل توضیح داده شد عمل می‌کند. در مورد کشف خط، یک خط با دو پارامتر (r, θ) تعریف می‌شود ولی برای تعریف دایره به سه پارامتر نیاز داریم:

$$C: (x_{center}, y_{center}, r)$$

که مرکز دایره (x_{center}, y_{center}) نقطه سبز رنگ در شکل زیر) و r همشعاع آن را مشخص می‌کند. با این سه پارامتر می‌توانیم هر دایره‌ای با هر مختصات و شعاعی تعریف کنیم. دایره قرمز در شکل زیر مثالی از کشف دایره است:



مثالی از کشف دایره

به منظور رعایت بهینگی، اسی وی این تبدیل با استفاده از روش گرادیان هو پیاده سازی کرده است که متفاوت با روش مورد استفاده در تبدیل استاندارد هو است.

۱ ۵,۵ کد

```
1 #include "opencv2/highgui/highgui.hpp"
2 #include "opencv2/imgproc/imgproc.hpp"
3 #include <iostream>
4 #include <stdio.h>
5
6 using namespace cv;
7
8 /** @function main */
9 int main(int argc, char** argv)
10 {
11     Mat src, src_gray;
```

```

12
13     /// Read the image
14     src = imread( argv[1], 1 );
15
16     if( !src.data )
17     { return -1; }
18
19     /// Convert it to gray
20     cvtColor( src, src_gray, CV_BGR2GRAY );
21
22     /// Reduce the noise so we avoid false circle detection
23     GaussianBlur( src_gray, src_gray, Size(9, 9), 2, 2 );
24
25     vector<Vec3f> circles;
26
27     /// Apply the Hough Transform to find the circles
28     HoughCircles( src_gray, circles, CV_HOUGH_GRADIENT, 1,
29     src_gray.rows/8, 200, 100, 0, 0 );
30
31     /// Draw the circles detected
32     for( size_t i = 0; i < circles.size(); i++ )
33     {
34         Point center(cvRound(circles[i][0]), cvRound(circles[i][1]));
35         int radius = cvRound(circles[i][2]);
36         // circle center
37         circle( src, center, 3, Scalar(0,255,0), -1, 8, 0 );
38         // circle outline
39         circle( src, center, radius, Scalar(0,0,255), 3, 8, 0 );
40     }
41
42     /// Show your results
43     namedWindow( "Hough Circle Transform Demo", CV_WINDOW_AUTOSIZE );
44     imshow( "Hough Circle Transform Demo", src );
45
46     waitKey(0);
47     return 0;
48 }
49
50

```

۵,۵,۲ توضیح

در خط ۲۳ به منظور کاهش نویز و جلوگیری از کشف دایره‌های اشتباه، یک فیلتر گوسی اعمال می‌کنیم.

در خط ۲۸ با استفاده از تابع **HoughCircles** تبدیل دایره هُو را اعمال می‌کنیم. این تابع دارای ۹ آرگومان به شرح زیر است:

- .۱: **src_gray**: تصویر ورودی که باید سیاه و سفید باشد.
- .۲: یک بردار که مجموعه‌ای از سه مقدار x_c, y_c, r را برای هر دایره کشف شده نگه می‌دارد.
- .۳: **CV_HOUGH_GRADIENT**: روش کشف را مشخص می‌کند. در حال حاضر فقط همین روش در آسی وجود دارد.
- .۴: نسبت معکوس وضوح است.
- .۵: حداقل فاصله بین دایره‌های کشف شده است.
- .۶: آستانه بالایی برای لبه‌یاب گُنی که در این تابع استفاده می‌شود.
- .۷: آستانه‌ای برای کشف مرکز دایره است.

.۸. ۰: حداقل شعاع مجاز برای دایره‌های کشف شده است. اگر معلوم نیست، مقدارش را صفر بگذارید.

.۹. ۰: حداکثر شعاع مجاز برای دایره‌های کشف شده است. اگر معلوم نیست، مقدارش را صفر بگذارید.

۵,۵,۳ خروجی



خروجی برنامه