دانشگاه صنعتی شریف

دانشکده ریاضی

درس بینایی ماشین

تمرین اول

دانشجو :

محمدشهاب سپهری ۹۶۱۰۱۷۷۶

استاد درس : دکتر کمالی تبریزی

دقت کنید در کد تحویل داده شده فولدری به نام Config وجود دارد که پارامترهای مربوط به سوالات در آن قرار دارند و این فولدر حتما در هنگام اجرا باید در فولدر اصلی پروژه باشد تا کدها بتوانند پارامترهای مورد نیاز را از آن بخوانند. همچنین نتایج در فولدری به نام Results (که در صورتی که این فولدر موجود نباشد به صورت خودکار ساخته می‌شود) ذخیره می‌شوند. همچنین دو فولدر دیگر به نام‌های Questions و Utils در فولدر اصلی پروژه وجود دارند که در آن فایل‌های حاوی توابع استفاده شده قرار دارند.

همچنین لازم به ذکر است من این کدها را روی سیستم عامل لینوکس تست کرده بودم و وقتی آن‌ها را در سیستم عامل ویندوزم تست کردم نتایج بخش ۱ تفاوت زیادی داشتند. به همین علت یک فایل requirements از ورژن پکیج‌های استفاده شده قرار دادم تا این تفاوت‌ها کمتر شوند.

۱. برای سوال مجموعه‌ای از توابع را پیاده کردم که همگی در فایل Harris.py در فولدر Questions قرار دارند.

- برای بدست آوردن I_x و I_y از فیلترهای sobel استفاده کردم. این فیلترها به صورت زیر هستند که با کانوالو کردن G_x و G_y در عکس به ترتیب I_x و I_y را حساب کردم.

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}, \quad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & 2 & -1 \end{bmatrix}$$

- برای محاسبه این مقادیر از روابط زیر استفاده کردم:

$$I_x{}^2 = I_x \times I_x, \quad I_y{}^2 = I_y \times I_y, \quad I_{xy} = I_x \times I_y$$

همچنین این پیاده سازی را در تابع get_grad قرار دادم/

- بزرگی گرادیان به سادگی محاسبه شد و نتیجه آن نیز در فولدر Results موجود است.
- در اینجا تابعی به نام apply_guassian_filter تعریف کردم که برای یک عکس دلخواه مقادیر $I_x{}^2$، $I_y{}^2$ و I_{xy} را از تابع get_grad می‌گیرد و سپس به کمک تابع guassian_filter از کتاب‌خانه scipy.ndimage فیلترهای گوسی را به آن‌ها اعمال می‌کند و $S_x{}^2$، $S_y{}^2$ و S_{xy} را بدست آورده و در خروجی می‌دهد.

در اینجا من عکس‌های مربوط به $S_x{}^2$، $S_y{}^2$ و S_{xy} را نیز ذخیره کرده‌ام و در فولدر Results موجودند.

همچنین برای تعیین مقدار σ در نوت بوک Q1.ipynb به ازای σهای مختلف نتایج را رسم کرده‌ام و درنهایت $\sigma = 2$ را انتخاب کرده‌ام.

- محاسبه دترمینان و trace پیچیدگی خاصی نداشت در تابع compute_score ابتدا این دو را حساب کرده‌ام و درنهایت امتیاز (یا همان R) را به کمک آن حساب کرده‌ام.
- همانطور که در بخش قبل گفته شد تابع compute_score امتیازات را محاسبه می‌کرد و عکس مربوطه را خروجی می‌داد که در اینجا نیز به کمک این تابع نتایج را بدست آوردم و با نام‌های گفته شده ذخیره کردم.
- برای تعیین مقدار k مناسب ابتدا باید دقت کرد که باید $k \leq 0.25$ باشد تا امتیاز قابلیت مثبت شدن داشته باشد زیرا طبق نامساوی حسابی-هندسی:

$$(S_x^2 + S_y^2)^2 \geq 4S_x^2 S_y^2$$

حال در نوت بوک Q1.ipynb به ازای مقادیر مختلف k خروجی امتیازها را کشیده‌ام و در نهایت براساس آن‌ها $k = 0.1$ را انتخاب کرده‌ام.

- برای انتخاب ترشهولد نیز دوباره در Q1.ipynb به ازای مقادیر مختلف خروجی‌ها و همچنین تعداد interest pointها را نمایش دادم و براساس آن‌ها ترشهولد را ۱۰ میلیون قرار دادم. در این حالت حدود ۳۰ هزار interest point در هر یک از تصاویر باقی می‌ماند.
- پیاده‌سازی non-maximum suppression را در تابعی با همین نام انجام دادم و روش کار آن به صورت زیر است:
  - در ابتدا یک کپی از ماتریس امتیازات درست می‌کنیم و سپس روی تمام نقاط روی این ماتریس جدید لوپ for می‌زنیم.
  - برای نقطه فعلی که روی آن هستیم اگر امتیاز آن از ترشهولد داده شده (همان ۱۰ میلیون) کمتر بود مقدار آن خانه را صفر می‌کنیم و در غیر این صورت تابع one_point_nms را اجرا می‌کنیم.
  - با توجه به توضیحات قسمت قبل می‌توان دید که تابع one_point_nms یک interest point می‌گیرد. حال وظیفه این تابع این است که تعیین کند که آیا این interest point در میان اطرافیانش مقدار بیشینه را دارد یا خیر. اگر مقدار بیشینه

را داشت، تمام نقاط اطرافش را صفر می‌کنیم و در غیر این صورت مقدار امتیاز خود آن را صفر می‌کنیم. برای اینکار یک کپی از ماتریس امتیازات می‌گیریم و سپس روی همسایگان نقطه مذکور لوپ for می‌زنیم. اگر مقدار امتیاز نقطه ما از امتیاز نقطه در حال بررسی کمتر بود، تابع همان ماتریس امتیاز ورودی را را که تنها امتیاز interest point ورودی را در آن صفر کرده‌است برمی‌گرداند. اما در غیر این صورت امتیاز نقطه فعلی در لوپ for را صفر می‌کند (به همین دلیل است که از ماتریس امتیازات کپی گرفتم زیرا ممکن است ابتدا تعدادی نقطه را چک کنیم که امتیازشان از نقطه مربوطه کمتر است و مقدار امتیاز آن‌ها را صفر کنیم اما بعد از آن به یک نقطه برسیم که امتیاز آن از نقطه فعلی بیشتر باشد و مجبور شویم تمام امتیازاتی را که تا الان صفر کرده‌ایم برگردانیم که برای حل این مشکل از کپی گرفتن از ماتریس اصلی استفاده کردم.).

باید دقت کنید که لوپ for روی دایره به شعاع rپیکسلی حرکت می‌کند. و این شعاع را ۱۰ پیکسل قرار دادم.

- در اینجا ترجیح دادم که برای بردار ویژگی یک مربع متقارن را در اطراف پیکسل بگیرم. لذا ضلع مربع مربوطه را به صورت $2n + 1$ قرار دادم که n را برابر ۲۰ گذاشتم و لذا یک مربع ۴۱ در ۴۱ حول نقطه را به عنوان بردار ویژگی درنظر گرفتم. کل این فرآیند در تابع get_int_points_features انجام می‌شود. همچنین در حالتی که یک نقطه چنین همسایگی‌ای نداشته باشد آن نقطه را دور می‌اندازم.
- از نرم ۲ به عنوان معیار فاصله استفاده کرده‌ام.
- در تابع get_nearests که یک بردار ویژگی و یک لیست از بردارهای ویژگی را می‌گیرد این فرآیند را پیاده کردم که اولین و دومین بردار نزدیک به تک بردار ورودی و فواصل آن‌ها را در خروجی می‌دهد.
- به کمک تابع check_d1_d2_tr این فرآیند را پیاده سازی‌کردم. این تابع با یک لوپ for و با استفاده از تابع get_nearests نقاط مطلوب را می‌یابد و در یک دیکشنری میریزد و این دیکشنری را در خروجی می‌دهد.

در اینجا من ترشهولد را برای نسبت $\frac{d_2}{d_1}$ اعمال کردم (در حالت باید این نسبت بیشتر از مقدار ترشهول شود) و ترشهولد مربوطه را ۱.۱۵ قرار دادم.

- روند بالا را برای نقاط تصویر دوم تکرار کردم.
- در تابع get_corresponding_points این فرآیند را پیاده کرده‌ام و این تابع دو دیکشنری مربوطه را می‌گیرد و در خروجی لیست دوتایی از نقطه‌هایی که با هم مچ شده‌اند را می‌دهد.
- با توجه به نحوه اعمال الگوریتم تا به اینجا احتمال اینکه یک نقطه به چند نقطه از تصویر متناظر شود وجود نداشت و لذا برای این بخش کاری نکردم.
- نتیجه مربوطه را رسم کرده و در فولدر Results قرار دادم.

دقت کنید که مقادیر پارامترهای مربوطه را در توضیحات بالا گفته‌ام. همچنین این مقادیر را می‌توان در فایل Config مربوطه دید که البته n در آنجا خود طول ضلع مربع مربوط به بردار ویژگی نیست و طول ضلع آن مربع برابر $2n + 1$ است.

۲. برای ساخت این تصویر از توضیحات صفحه ۱۵ اسلاید ۸ استفاده کرده‌ام و در ادامه توضیحاتم براساس آن خواهد بود.

در اینجا من دوربینی که دقیقا بالای زمین است را دوربین مرجع (دوربین مربوط به صفحه π) و دوربین کناری را دوربین ثانویه گرفته‌ام. حال ابتدا برای ماتریس K از رابطه زیر استفاده کرده‌ام:

$$K = \begin{bmatrix} f & 0 & P_x \\ 0 & f & P_y \\ 0 & 0 & 1 \end{bmatrix}$$

برای هر دو دوربین فاصله کانونی ثابت و برابر ۵۰۰ است. اما در مورد P این نقطه را عموما مرکز تصویر می‌گیریم که برای دوربین ثانویه با توجه به اینکه اندازه تصویر را داریم می‌توان آن را به سادگی حساب کرد. اما در مورد دوربین مرجع این امر صادق نیست. لذا من با فرض اینکه مختصات‌های تصویر بتوانند منفی شوند، P برای دوربین مرجع همان P دوربین ثانیه قرار دادم و ایرادی که ایجاد می‌شود در بازیابی تصویر است (چون ممکن است تعدادی از خانه‌های آن

مختصات منفی داشته باشند) که آن را در بخش آخر رفع کرده‌ام. پس با این فروض ماتریس K برای هر دو دوربین یکی خواهد شد.

حال نوبت به محاسبه ماتریس R و t مربوط به دوربین ثانویه می‌رسد. این دوربین به اندازه ۴۰ واحد در راستای x (دقت کنید که استای x همان راستای عمودی است) به بالا رفته است (که به دلیل نحوه قرار گیری محورهای مختصات همان پایین رفتن در راستای عمودی است). پس

$$C = \begin{bmatrix} 40 \\ 0 \\ 0 \end{bmatrix}$$

حال برای ماتریس R با کمی دقت می‌توان دید این دوران، دورانی حول محور y است و زاویه دوران برابر $-\arctan\left(\frac{40}{25}\right)$ خواهد بود و لذا اگر قرار دهیم $\theta = -\arctan\left(\frac{40}{25}\right)$:

$$R = \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) \\ 0 & 1 & 0 \\ \sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

حال به سادگی t را به کمک رابطه $t = -RC$ حساب می‌کنیم.

درنهایت بردار n باقی می‌ماند که چون صفحه زمین فوتبال موازی صفحه دوربین مرجع است معادله آن به صورت $-z + 25 = 0$ می‌شود و لذا:

$$n = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}$$

می‌شود. حال به کمک این مقادیر و رابطه زیر ماتریس هموگرافی را حساب می‌کنیم:

$$H = K'(R - \frac{tn^t}{d})K^{-1}$$

دقت کنید این ماتریس مختصات‌های عکس دیده شده توسط دوربین مرجع را به مختصات‌های مربوط به عکس مربوط به دوربین ثانویه (که همان عکس در دسترس ماست) تبدیل می‌کند. حال به حل مشکل P که در قبل گفته بودم می‌پردازم. برای حل این مشکل ابتدا به کمک وارون این هموگرافی تمام مختصات‌های عکس فعلی را به مختصات‌های عکس دیده شده از دوربین بالا تبدیل کرده‌ام (دقت کنید که مختصات‌های خروجی صحیح نیستند و لذا آن‌ها را به عدد صحیح تبدیل کرده‌ام). سپس ماکسیمم و مینیمم مربوط به دو محور x و y را درمیان این

مختصات‌ها بدست آورده‌ام و به کمک آن‌ها سایز عکس دیده شده از دوربین اول را بدست آوردم (اختلاف ماکسیمم و مینیمم به علاوه ۱). سپس هر مختصات عکس خروجی را به کمک ماتریس هموگرافی بدست آمده به مختصات‌های عکس اصلی تبدیل کردم و به این صورت عکس دیده شده از دوربین مرجع را ساختم و با نام مربوطه ذخیره کردم.

۳. در این تمرین از کتاب‌خانه opencv استفاده کرده‌ام که در هر بخش توضیحات مربوط به توابع استفاده شده را خواهم داد. توابع این بخش در فایل matching.py در فولدر Questions قرار دارند.

- برای این بخش از کلاس SIFT کتاب‌خانه opencv استفاده کرده‌ام. به کمک دستور SIFT_create یک آبجکت از این کلاس ساخته‌ام و این آبجکت می‌تواند با دستور detectAndCompute نقاط مهم و توضیح دهنده‌های آن‌ها را بدهد. این کار را در تابع get_sift_keypoints پیاده کرده‌ام.
- برای رسم نقاط از دستور drawKeypoints خود کتاب‌خانه opencv استفاده کرده‌ام که نقاط مهم را روی عکس می‌کشد. این تابع قابلیت کشیدن نقاط را با دایره به اندازه خود نقطه و جهت بکشد که من از این flag استفاده نکردم. برای کشیدن دو تصویر در کنار هم نیز از کتاب‌خانه matplotlin.pyplot استفاده کردم.
- برای یافتن نقاط متناظر از کلاس BFMatcher در opencv استفاده کردم. ابتدا یک آبجکت از این کلاس ساختم که متغیر crosscheck آن را false قرار دادم. این متغیر اگر true باشد نقاط مچ شده در خروجی به این گونه خواهند بود که مثلا برای دو نقطه a و b که به هم مچ شده‌اند، a نزدیک‌ترین نقطه به b است و بالعکس. این کار را من خودم پیاده کردم. به این صورت که به کمک دستور knnMatch کلاس BFMatcher ۲ نقطه نزدیک‌ترین به هر نقطه را بدست آوردم و سپس همان روش سوال ۱ را پیاده کردم. یعنی سپس با یک ترشهولد (در اینجا ۰.۷ و در اینجا دیگر همان نسبت $\frac{d_1}{d_2}$ را بررسی کردم!) نسبت $\frac{d_1}{d_2}$ را بررسی کردم و همین کار را برای نقاط صفحه دوم به اول

انجام دادم و در نهایت اگر دو نقطه به هم مچ شده بودند آن‌ها را نگه داشتم. این فرآیند در تابع get_match_points پیاده شده است.

- برای کشیدن
- برای وصل کردن نقاط متناظر از دستور drawMatches خود opencv استفاده کردم که اینکار را به سادگی انجام می‌دهد و صرفا باید از flagهای مناسب آن استفاده کرد.
- ۲۰ نقطه متناظر را نیز به کمک همان دستور drawMatches کشیدم.
- در اینجا برای RANSAC از دستور findHomography استفاده کردم و برای پارامترها از همان مقادیر دیفالت استفاده کردم. همچنین دقت کنید در ادامه از دستور warpPerspective برای کشیدن عکس خروجی استفاده می‌کنم و چون اگر مختصاتی منفی شود توسط این دستور کشیده نمی‌شود نیاز است مقداری offset به مختصات‌های عکس خروجی بدهیم. یک offset ماتریسی به صورت زیر خواهد بود:

$$offset = \begin{bmatrix} 1 & 0 & x_0 \\ 0 & 1 & y_0 \\ 0 & 0 & 1 \end{bmatrix}$$

که x_0 و y_0 مقدار آفستی هستند که می‌خواهیم. علت این امر نیز خاصیت فضای projective است. با ضرب کردن ماتریس offset از سمت چپ در مختصات می‌توان مختصات حرکت داده شده را بدست آورد. لذا می‌توان کل این تبدیل را به صورت یک هموگرافی با ماتریس $offset \times H$ درنظر گرفت. البته در کد من درنهایت به دلیل جابه‌جایی نقاط ماتریس هموگرافی از عکس اول به عکس دوم بدست آمد که به همین دلیل از آن معکوس گرفتم. این کار را در تابع ماتریس‌های گفته شده به صورت زیر شدند:

```
F:\Shahab\Term8\Vision\1\code>python Q3.py
Homography matrix without offset:
[[ 2.65537311e-01  2.62694754e-02  6.12975971e+02]
 [-1.36514523e-02  5.05080053e-01  4.68216739e+02]
 [-2.08149849e-05  1.02193311e-04  1.00000000e+00]]
Homography matrix with offset:
[[ 3.84838945e+00 -4.76128200e-01  1.36396094e+03]
 [ 1.02464866e-01  1.97172271e+00  1.39979233e+01]
 [ 7.17377930e-05 -2.17797851e-04  1.05800306e+00]]
number of inliers: 26
number of outliers: 43
```

که در اینجا تعداد inlierها و outlierها را هم آورده‌ام. تعداد تکرار استفاده شده در اینجا ۲۰۰ است. البته اگر با استفاده از فرمولی که در اسلایدها برای تکرارها بود و با استفاده از inlierها و outlierها تکرارهای مورد نیاز را محاسبه کنیم برای اطمینان ۹۹ درصد به ۲۲۶ تکرار نیاز است.

- به کمک دستور drawMatches شکل خواسته شده را رسم کردم.
- با بررسی نقاط بدست آمده بنظر missmatchایی وجود نداشت. در این بخش عکس مربوط به وصل کردن نقاط نهایی را با نام inliers_Q3.jpg ذخیره کرده‌ام.
- نتیجه نهایی را ذخیره کردم که به نظر تا حد خوبی پرسپکتیو آن رفع شده است.

۴. توابع این بخش نیز در فایل matching.py در فولدر Questions قرار دارند.

در اینجا تمام مراحل سوال قبل را انجام دادم (بجز مراحل مربوط به ذخیره عکس‌ها) به جز مرحله مربوط به اعمال RANSAC که آن را با تابعی به نام my_RANSAC پیاده کردم که این تابع به این صورت عمل می‌کند:

- این تابع از شما نقاط مچ شده را می‌گیرد. فرض کنید تعداد این نقاط n باشد. این تابع در ابتدا دو ماتریس ۳ در n می‌سازد که این ماتریس‌ها همان نقاط مربوط به نقاط مچ شده در فضای projective (اضافه کردن ۱ به عنوان مختصات سوم) هستند که در کنار هم قرار گرفته‌اند. اینکار را برای افزایش سرعت الگوریتم در هنگام محاسبه inlierها انجام داده‌ام.
- سپس این تابع در یک حلقه while قرار می‌گیرد که این حلقه دارای شرط‌های زیر است:
  - این حلقه به اندازه حداقل یک تعداد معین (که در اینجا ۱۰۰ گذاشته بودم) اجرا می‌شود.

- این حلقه حداکثر به اندازه یک تعداد معین دیگر (که در اینجا ۱۰۰۰۰ گذاشته بودم) اجرا می‌شود.
- اگر N که تعداد تکرارهای مورد نیاز برای سطح اطمینان مورد نظر است (که نحوه محاسبه آن را در اسلایدها دیده بودیم) از تعداد تکرارهای فعلی بیشتر شد و حداقل تکرار خواسته شده انجام شده بود، حلقه قطع می‌شود.

- در این حلقه هر مرحله ۴ زوج تصادفی از نقاط مچ شده انتخاب می‌شوند و به تابع compute_homography داده می‌شوند تا هموگرافی آن‌ها را محاسبه کند. نحوه محاسبه هموگرافی نیز به مانند روش اسلایدهاست که ابتدا ماتریس مربوط به ۸ معادله مربوطه را تشکیل می‌دهم و سپس به کمک تجزیه svd جواب مطلوب را بدست می‌آوردم. البته نکته‌ای که در اینجا متوجه آن شدم این بود که سطرهای اول و دوم ماتریس هموگرافی بدست آمده برعکس بودند که به کمک یک تابع دیگر آن‌ها را جابجا کردم.
- بعد از محاسبه ماتریس هموگرافی نوبت به محاسبه خطا می‌رسد. برای محاسبه خطا اولا باید توجه کرد که ضرب H در نقاط تصویر اول دقیقا خود مختصات تصویر دوم نمی‌شوند و این تساوی یک ضریب دارد که برای هر نقطه نیز متفاوت است. حال در اینجا ابتدا H را در ماتریس ۳ در n مربوط به مختصات نقاط تصویر اول (که در قبل حلقه while) ساخته بودم ضرب می‌کنم و خروجی باید همان ماتریس ۳ در n مربوط به مختصات نقاط تصویر دوم باشد که هر سطر آن در ضریبی ضرب شده است. برای رفع مشکل این ضرایب، هر ستون این ماتریس را بر عضو سومش تقسیم کرده‌ام و البته در این حین چک کرده‌ام که تقسیم بر صفر انجام ندهم (اگر تقسیم بر صفر رخ می‌داد تکرار مربوطه را رد می‌کنیم و یک تکرار دیگر به جای آن انجام می‌دهم). در نهایت ماتریس بدست آمده را می‌توان با ماتریس ۳ در n مربوط به مختصات نقاط تصویر دوم مقایسه کرد و مقدار خطا را برای هر ستون حساب کرد.
- بعد از محاسبه خطا می‌توان براساس یک ترشهولد (که آن را برابر ۳ که همان مقدار دیفالت در تابع مربوطه در findHomography بود، قرار دادم) نقاط inlier و درنهایت supportها را بدست آورد. حال اگر supportها زیاد شدند، هموگرافی جدید را به عنوان

نتیجه قرار می‌دهیم و w و N جدید را برحسب همان روابط موجود در اسلایدها حساب می‌کنیم.

درنهایت نتیجه این تابع شبیه به نتیجه روش قبلی شد. اما دو تفاوت وجود داشت:

۱. تعداد تکرارهای روش من بیشتر می‌شد که البته این امر به دلیل روش استفاده شده برای تخمین تعداد تکرارها بود اما با تست‌هایی که خودم انجام دادم تعداد تکرار بیشتری نسبت به تابع خود opencv نیاز بود تا با احتمال بالایی به جواب مورد قبول برسیم. در زیر یکی از نتایج را می‌بینید:

```
number of iterations: 371
number of inliers: 23
number of outliers: 46
```

البته تعداد تکرار بین ۲۰۰ تا ۸۰۰ متغیر بود (به دلیل تصادفی بودن الگوریتم).

۲. سرعت الگوریتم من کمتر از الگوریتم خود opencv بود. اوج این مسئله را می‌توان در حالتی دید که تست نسبت مربوط به دو نقطه نزدیک‌ترین را حذف کنیم. در این حالت تعداد نقاط مچ شده به شدت زیاد می‌شد و inlierها درصد بسیار کمی را تشکیل می‌دادند. در اینجا الگوریتم من زمان بسیار زیادی برای رسیدن به جواب قابل قبول نیاز داشت (حدود نیم ساعت) در صورتی که الگوریتم خود opencv بسیار سریع‌تر بود. البته یکی از دلایل این تفاوت این است که opencv با c++ پیاده شده است و همین امر آن را سریع‌تر می‌کند اما دلایل دیگری نیز می‌توانند روی این تفاوت موثر باشند.