

# ECE66100 Homework #9

Ziyue “Alan” Xiang

November 28, 2022

## Contents

|                                                                 |    |
|-----------------------------------------------------------------|----|
| I. Image Rectification . . . . .                                | 3  |
| II. Task 1 . . . . .                                            | 6  |
| III. Task 2 . . . . .                                           | 6  |
| IV. Task 3 . . . . .                                            | 7  |
| V. Figures . . . . .                                            | 8  |
| A. Source Code . . . . .                                        | 19 |
| A.1. Source Listing of code/common.py . . . . .                 | 19 |
| A.2. Source Listing of code/point_selector.py . . . . .         | 20 |
| A.3. Source Listing of data/LoopAndZhang/src/main.cpp . . . . . | 20 |
| A.4. Source Listing of data/LoopAndZhang/src/util.cpp . . . . . | 22 |
| A.5. Source Listing of code/09.ipynb . . . . .                  | 29 |

## List of Figures

|                                                                     |    |
|---------------------------------------------------------------------|----|
| 1. The keypoints selected in image. . . . .                         | 8  |
| 2. The keypoint correspondence in left and right images. . . . .    | 8  |
| 3. The keypoint correspondences after image rectification. . . . .  | 9  |
| 4. The Canny edge detection result of the rectified images. . . . . | 9  |
| 5. The histogram of SSD in interest point matching. . . . .         | 10 |
| 6. The interest point matching result. . . . .                      | 10 |
| 7. The projective reconstruction result (angle 1). . . . .          | 11 |
| 8. The projective reconstruction result (angle 2). . . . .          | 12 |

9. The projective reconstruction result with less visual elements (angle 1). . . . . 13

10. The projective reconstruction result with less visual elements (angle 2). . . . . 14

11. `img_1` and `img_2` output by the `LoopAndZhang` program. . . . . 15

12. `img_1_dst` and `img_2_dst` output by the `LoopAndZhang` program. . . . . 15

13. The left and right images used in Task 3. . . . . 15

14. The ground truth disparity map for Task 3. . . . . 16

15. The output disparity map for  $M = 7$ . . . . . 16

16. The output disparity map for  $M = 9$ . . . . . 17

17. The error map for  $M = 7$ . . . . . 17

18. The error map for  $M = 9$ . . . . . 18

List of Tables

## I. Image Rectification

In image rectification, we are given two images  $\mathcal{I}_L$  (the left image) and  $\mathcal{I}_R$  (the right image), both of which are of size  $(W, H)$ . Suppose we have established pixel location correspondences  $\{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N\}$  and  $\{\mathbf{x}'_1, \mathbf{x}'_2, \dots, \mathbf{x}'_N\}$  such that  $\mathbf{x}_i$  in the left image matches  $\mathbf{x}'_i$  in the right image. Given the relationship

$$\mathbf{x}_i \mathbf{F} \mathbf{x}'_i = 0, \quad (1)$$

where

$$\mathbf{F} = \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix}, \quad (2)$$

if we write  $\mathbf{x}_i = (x_i, y_i, 1)$  and  $\mathbf{x}'_i = (x'_i, y'_i, 1)$ , then Eq. (1) is equivalent to

$$\begin{bmatrix} x_i x'_i & x'_i y_i & x'_i & x_i y'_i & y_i y'_i & y'_i & x_i & y_i & 1 \end{bmatrix}^T \begin{bmatrix} f_{11} & f_{12} & f_{13} & f_{21} & f_{22} & f_{23} & f_{31} & f_{32} & f_{33} \end{bmatrix} = 0. \quad (3)$$

Eq. (3) allows us to estimate  $\mathbf{F}$  given at least 8 pairs of correspondences using linear least squares. That is,  $N \geq 8$ .

After the estimation of  $\mathbf{F}$ , we need to estimate the camera projection matrices for the left (denoted by  $\mathbf{P}$ ) and right camera (denoted by  $\mathbf{P}'$ ). For this step, we assume the canonical configuration. That is, we set

$$\mathbf{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad (4)$$

and then compute  $\mathbf{P}'$  based on  $\mathbf{P}$ . It is worth noticing that Eq. (4) implies the left epipole is at  $(0, 0, 0)$ . However, the default left correspondences  $\mathbf{x}_i$  are in image coordinates, which are centered at  $(W/2, H/2)$ . Therefore, to make sure the correspondences are compatible with the canonical configuration, an important preprocessing step is to translate  $\mathbf{x}_i$  and  $\mathbf{x}'_i$  by  $(-W/2, -H/2)$ . From  $\mathbf{F}$ , we can extract the right epipole  $\mathbf{e}'$ , which satisfies

$$\mathbf{e}'^T \mathbf{F} = 0. \quad (5)$$

More concretely, the right epipole  $\mathbf{e}'$  is a left null vector of  $\mathbf{F}$ . Denote matrix representation of cross product for vector  $\mathbf{v} = [v_1 \ v_2 \ v_3]^T$  by  $[\mathbf{v}]_{\times}$ , it is known that

$$[\mathbf{v}]_{\times} = \begin{bmatrix} 0 & -v_3 & v_2 \\ v_3 & 0 & -v_1 \\ v_2 & v_1 & 0 \end{bmatrix}. \quad (6)$$

Now the right projection matrix  $\mathbf{P}'$  can be written as

$$\mathbf{P}' = [[\mathbf{e}']_{\times} \mathbf{F} \mid \mathbf{e}'].$$

This estimation of  $\mathbf{P}'$  can be refined with nonlinear least squares methods. Given  $\mathbf{P}$  and  $\mathbf{P}'$ , as well as a pair of correspondences  $(\mathbf{x}_i, \mathbf{x}'_i)$ , we can use triangulation to acquire the original world point  $\mathbf{X}_i$  that leads to the image pair  $(\mathbf{x}_i, \mathbf{x}'_i)$ . More specifically, the following equation is satisfied,

$$\begin{bmatrix} x_i \mathbf{P}_3^T - \mathbf{P}_1^T \\ y_i \mathbf{P}_3^T - \mathbf{P}_2^T \\ x'_i \mathbf{P}_3'^T - \mathbf{P}_1'^T \\ y'_i \mathbf{P}_3'^T - \mathbf{P}_2'^T \end{bmatrix} \mathbf{X}_i = 0, \quad (7)$$

where  $\mathbf{P}_j$  and  $\mathbf{P}'_j$  indicate the  $j$ -th row of  $\mathbf{P}$  and  $\mathbf{P}'$ , respectively. The solution for  $\mathbf{X}_i$  can be computed using linear least squares. For the  $N$  correspondence pairs, we can determine  $N$  world points  $\mathbf{X}_1, \dots, \mathbf{X}_N$ . For a world point  $\mathbf{X}_i$ , we can back-project it back to the left and right image planes to form  $\hat{\mathbf{x}}_i$  and  $\hat{\mathbf{x}}'_i$ , which are given by

$$\begin{aligned} \hat{\mathbf{x}}_i &= \mathbf{P} \mathbf{X}_i \\ \hat{\mathbf{x}}'_i &= \mathbf{P}' \mathbf{X}_i. \end{aligned}$$

The “goodness” of the camera projection matrices can be evaluated using the difference between  $(\mathbf{x}_i, \hat{\mathbf{x}}_i)$  and  $(\mathbf{x}'_i, \hat{\mathbf{x}}'_i)$ . That is, we can define the loss function as

$$\ell = \sum_{i=1}^N (\|\mathbf{x}_i - \hat{\mathbf{x}}_i\|^2 + \|\mathbf{x}'_i - \hat{\mathbf{x}}'_i\|^2) \quad (8)$$

and then use it with nonlinear least squares methods to refine the estimation of  $\mathbf{P}'$ .

Once we acquire the refined version of  $\mathbf{P}'$ , the epipole  $\mathbf{e}'$  of the refined right camera is given by the forth column. Then, a refined version of  $\mathbf{F}$  can be computed by

$$\mathbf{F} = [\mathbf{e}']_{\times} \mathbf{P}' \mathbf{P}^+, \quad (9)$$

where  $\mathbf{P}^+$  is the pseudo-inverse of  $\mathbf{P}$ .

With more precise estimation of  $\mathbf{F}$  and  $\mathbf{P}'$ , we can start computing the left homography  $\mathbf{H}$  and right homography  $\mathbf{H}'$  for image rectification. Write the right epipole as  $\mathbf{e}' = [e'_1 \ e'_2 \ e'_3]$ . We need to compute the angle  $\theta$  such that

$$\theta = \arctan2(e'_2, e'_1). \quad (10)$$

The rotation matrix

$$\mathbf{R}_{\theta} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (11)$$

can rotate  $\mathbf{e}'$  to the  $xy$ -plane. That is, if we write  $\mathbf{Re}' = [\tilde{e}_1 \ \tilde{e}_2 \ \tilde{e}_3]$ , then  $\tilde{e}_2 = 0$ . We know that the focal length of the camera  $f$  is given by

$$f = \frac{\tilde{e}_1}{\tilde{e}_3}. \quad (12)$$

Now, the right homography  $\mathbf{H}'$  is given by

$$\mathbf{H}' = \mathbf{GR}_\theta \mathbf{T}, \quad (13)$$

where

$$\mathbf{G} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -\frac{1}{f} & 0 & 1 \end{bmatrix}, \quad (14)$$

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & \frac{w}{2H} \\ 0 & 1 & \frac{2}{2} \\ 0 & 0 & 1 \end{bmatrix}. \quad (15)$$

To compute the left homography, we first define

$$\mathbf{M} = \mathbf{P}'\mathbf{P}^+ \quad (16)$$

$$\mathbf{H}_0 = \mathbf{H}'\mathbf{M} \quad (17)$$

$$\mathbf{H}_A = \begin{bmatrix} a & b & c \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \quad (18)$$

Then, we compute  $\mathbf{z}_i = (\zeta_i, \eta_i, 1)$  and  $\mathbf{z}'_i = (\zeta'_i, \eta'_i, 1)$ , where

$$\mathbf{z}_i = \mathbf{H}_0 \mathbf{x}_i \quad (19)$$

$$\mathbf{z}'_i = \mathbf{H}' \mathbf{x}'_i. \quad (20)$$

We can build the linear system

$$\begin{bmatrix} \zeta_1 & \eta_1 & 1 \\ \vdots & \vdots & \vdots \\ \zeta_N & \eta_N & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} \zeta'_1 \\ \vdots \\ \zeta'_N \end{bmatrix} \quad (21)$$

to solve for  $a, b, c$  using linear least squares. The left homography  $\mathbf{H}$  is given by

$$\mathbf{H} = \mathbf{H}_A \mathbf{H}_0. \quad (22)$$

## II. Task 1

- We perform the steps described in Section I
- We manually selected 16 keypoints in two views, as shown in Fig. 1
- The keypoint correspondences in the two views are shown in Fig. 2
- After image rectification, the keypoint correspondences are shown in Fig. 3
- We apply Canny edge detection to the images after rectification; the results are shown in Fig. 4
- For a positive pixel (known as an interest point) in the Canny edge detection result in one rectified view, we can find its corresponding interest point in the same row of another rectified view by using the Sum of Squared Difference (SSD) criterion
- In Fig. 5, we show the histogram of the SSD of all matched interest points; In Fig. 6, we show the interest point matching result; Since there are a large number of interest point matches, we only show  $1/200$  of those with SSD less than 0.02 in Fig. 6
- We show the projective reconstruction result in Figs. 7 and 8, where each figure includes the 3D view of the reconstruction from a distinct camera location; note that only a small subset of the reconstructed points are shown
- In Figs. 9 and 10, we show the projective reconstruction from the same angles as Figs. 7 and 8, but with less visual elements

## III. Task 2

- The LoopAndZhang algorithm decomposes the rectifying homographies into three components:

$$\mathbf{H} = \mathbf{H}_{sh}\mathbf{H}_{sim}\mathbf{H}_p, \quad (23)$$

where  $\mathbf{H}_{sh}$  is a shearing homography,  $\mathbf{H}_{sim}$  is a similarity homography, and  $\mathbf{H}_p$  is a projective homography. This decomposition of  $\mathbf{H}$  allows us to estimate each component individually, which results in a better overall estimation of  $\mathbf{H}$ .

- We experimented the automatic correspondence detection method in the LoopAndZhang program, and it led to bad results; therefore, we pass our manually selected keypoint correspondences in Task 1 to the LoopAndZhang program
- The `img_1` and `img_2` images output by the program are shown in Fig. 11; the `img_1_dst` and `img_2_dst` images output by the program are shown in Fig. 12
- It can be seen that given manually selected correspondences, the result of LoopAndZhang is similar to that of Task 1; the method fails to automatically determine correspondences for the two input images in Task 1

## IV. Task 3

- In dense stereo matching with Census Transform, for a pixel location in left image  $(x, y)$  and a pixel location in right image  $(x', y)$ , we first take  $M \times M$  windows centered at the two pixel locations in two respective images. Then, for each pixel value in each window, we change the value to one if the pixel value is greater than the center pixel value, and zero otherwise. The two windows will now be binary-valued. We can apply XOR operation between the two windows and count the number of ones in the result. The result can be used as the “distance” between the two windows. The  $x'$  that results in the smallest distance can be used for disparity computation.
- The two images for this task are shown in Fig. 13; the ground truth disparity map is shown in Fig. 14
- The max disparity used in our experiments is 52
- We experimented with two window sizes:  $7 \times 7$  and  $9 \times 9$ , the accuracy of  $M = 7$  is 0.5869; the accuracy of  $M = 9$  is 0.6522
- The disparity map of  $M = 7$  is shown in Fig. 15; the error map is shown in Fig. 17
- The disparity map of  $M = 9$  is shown in Fig. 16; the error map is shown in Fig. 18
- It can be seen that bigger window size resulted in higher accuracy
- From the error maps, it can be seen that the edge areas are more challenging to match

V. Figures



Figure 1: The keypoints selected in image.

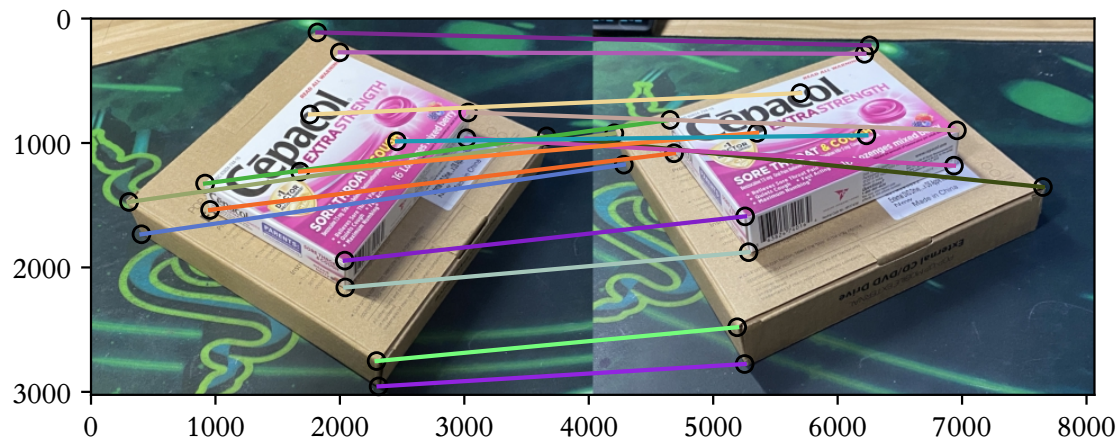


Figure 2: The keypoint correspondence in left and right images.

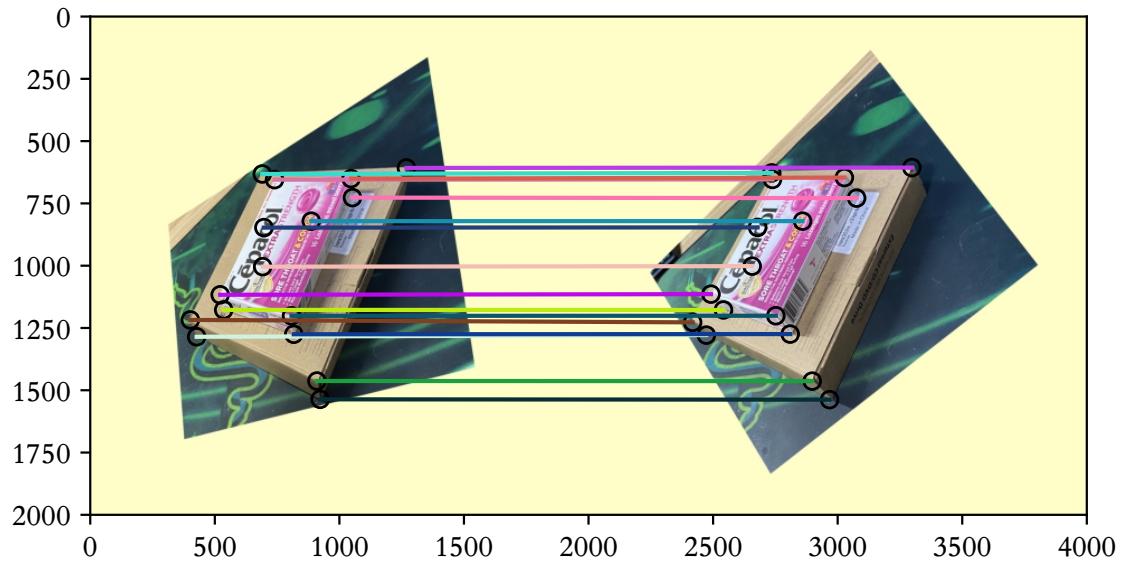


Figure 3: The keypoint correspondences after image rectification.

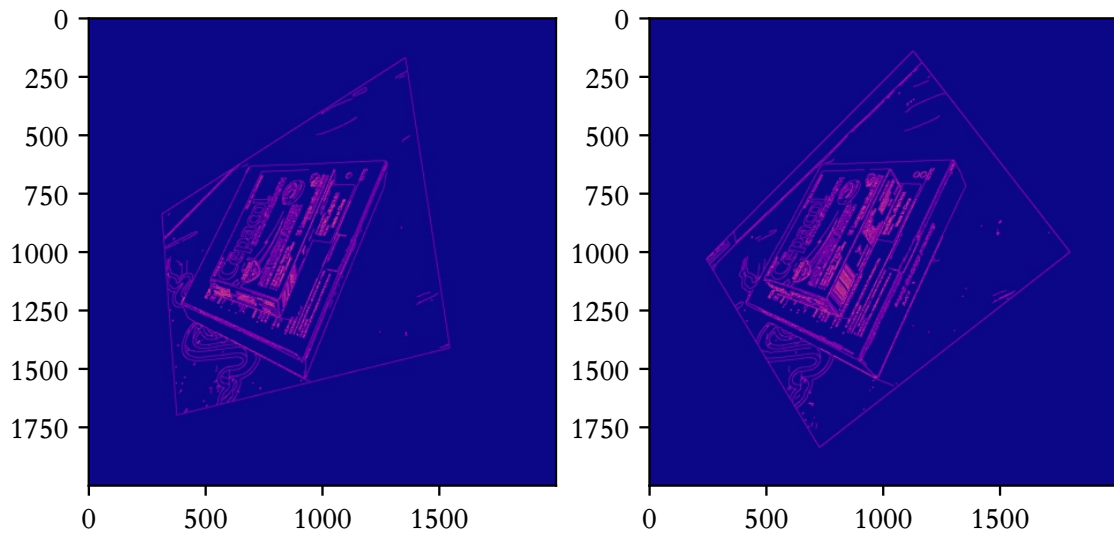


Figure 4: The Canny edge detection result of the rectified images.

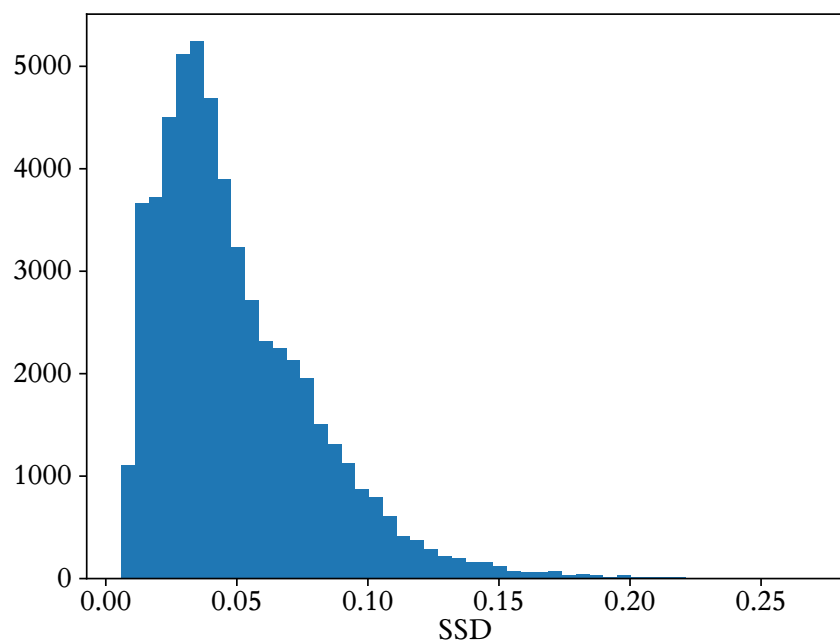


Figure 5: The histogram of SSD in interest point matching.

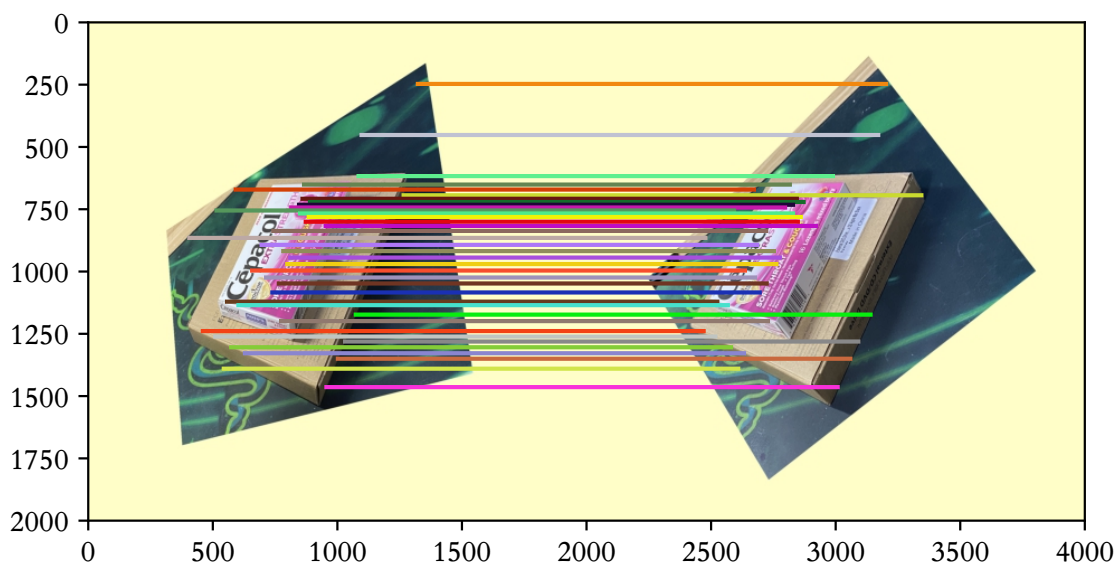


Figure 6: The interest point matching result.

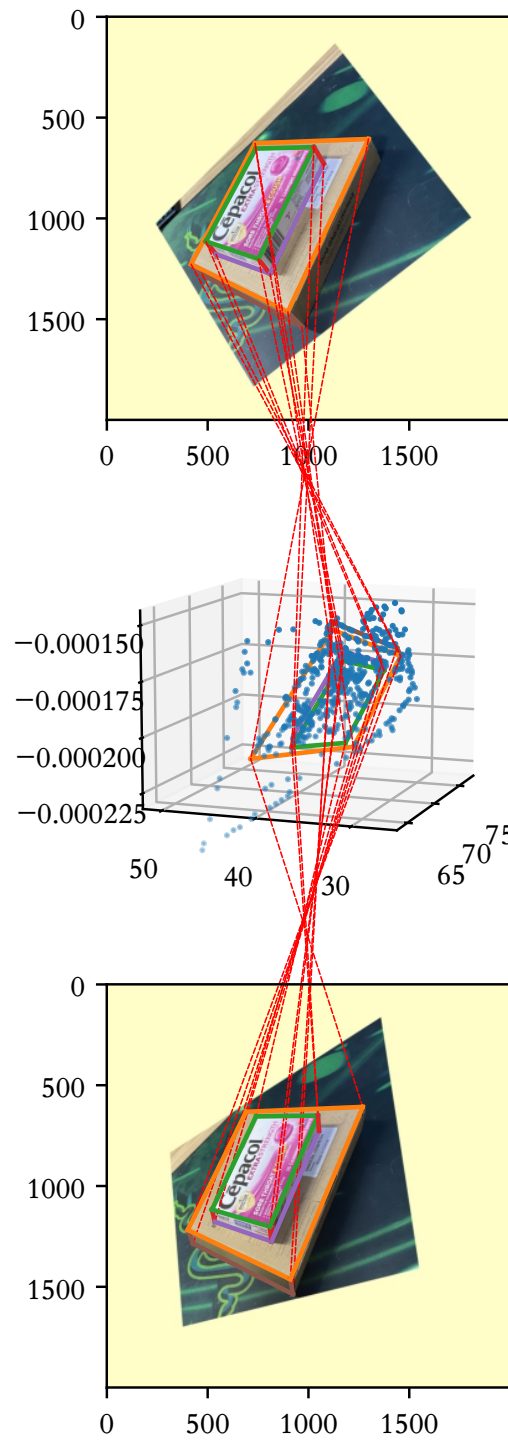


Figure 7: The projective reconstruction result (angle 1).

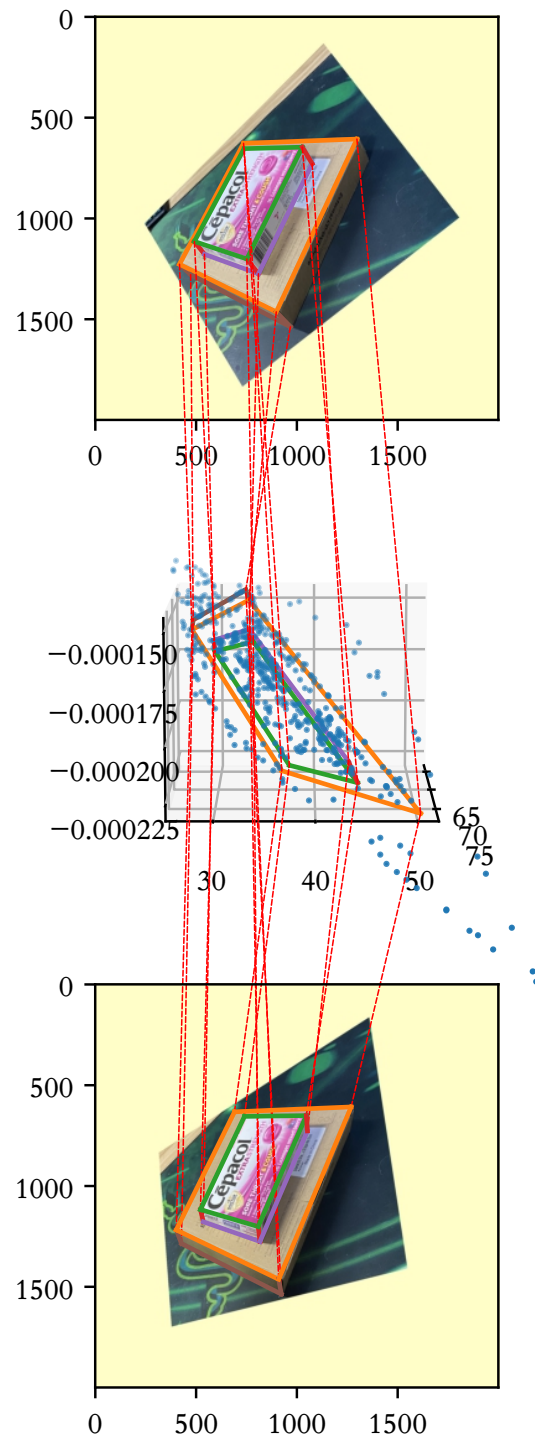


Figure 8: The projective reconstruction result (angle 2).

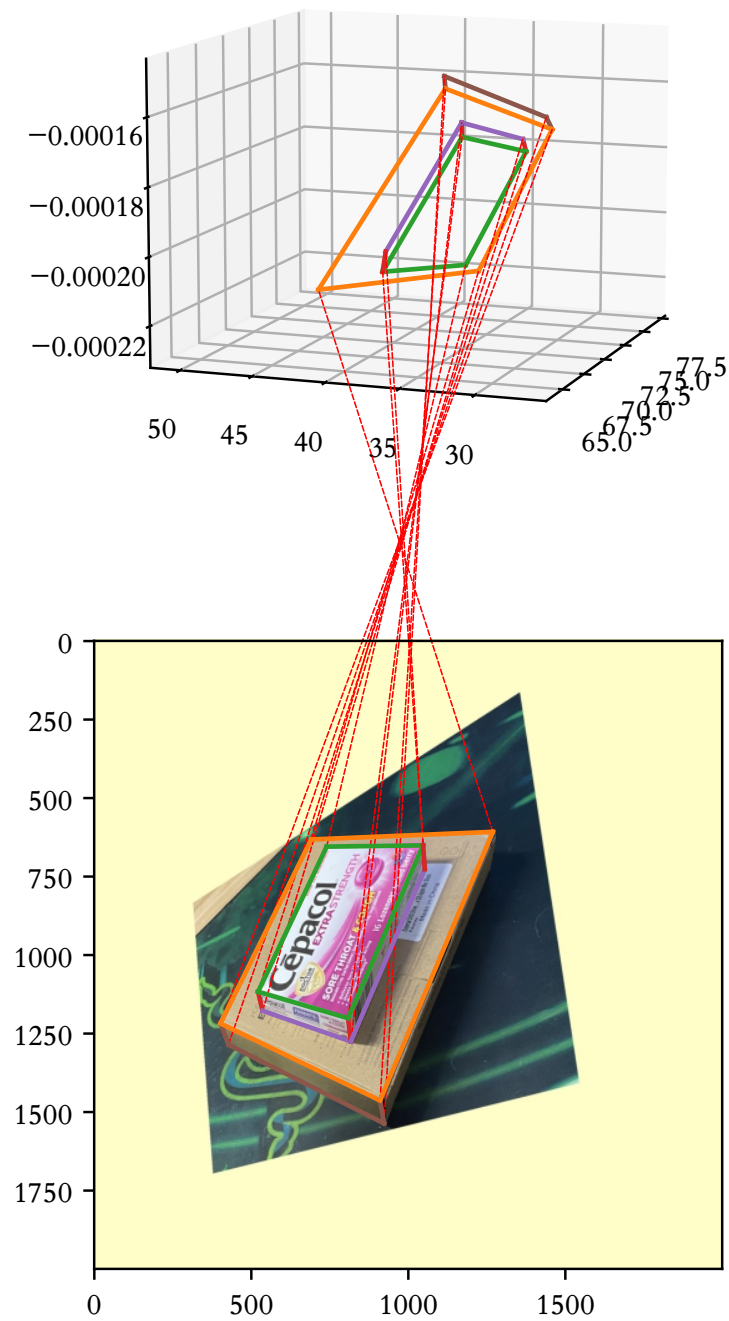


Figure 9: The projective reconstruction result with less visual elements (angle 1).

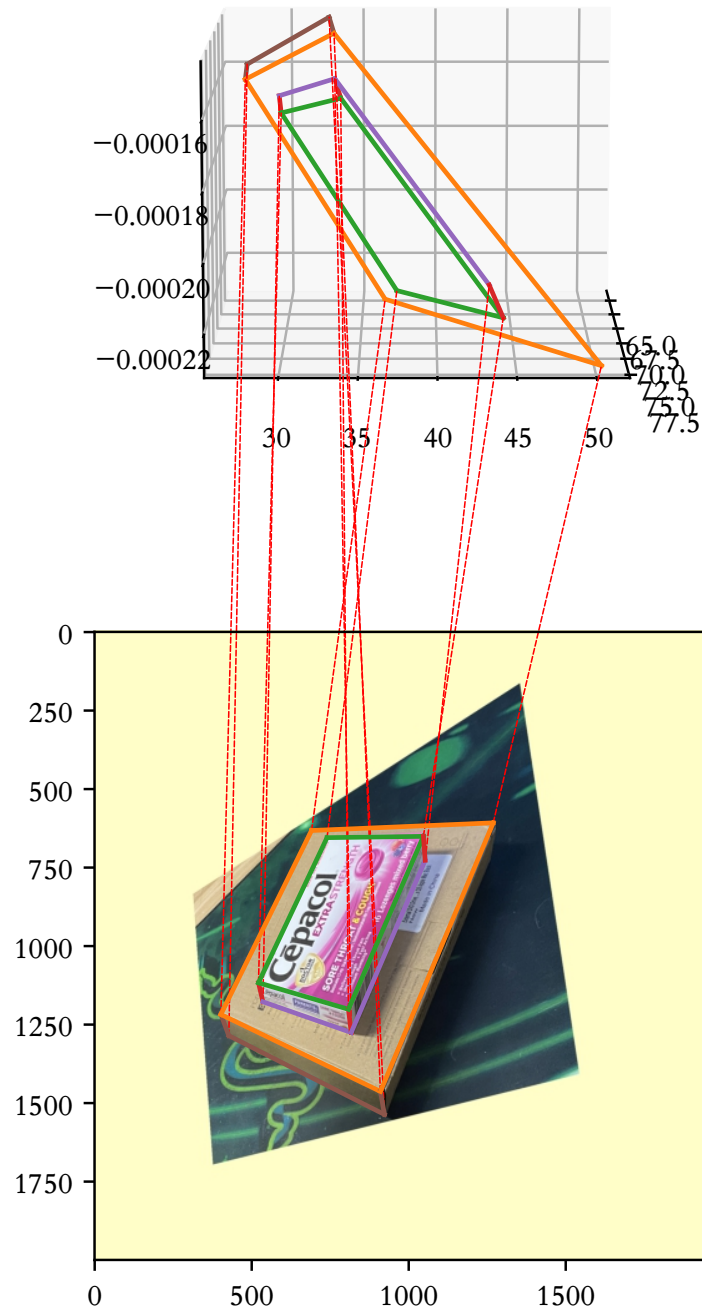


Figure 10: The projective reconstruction result with less visual elements (angle 2).

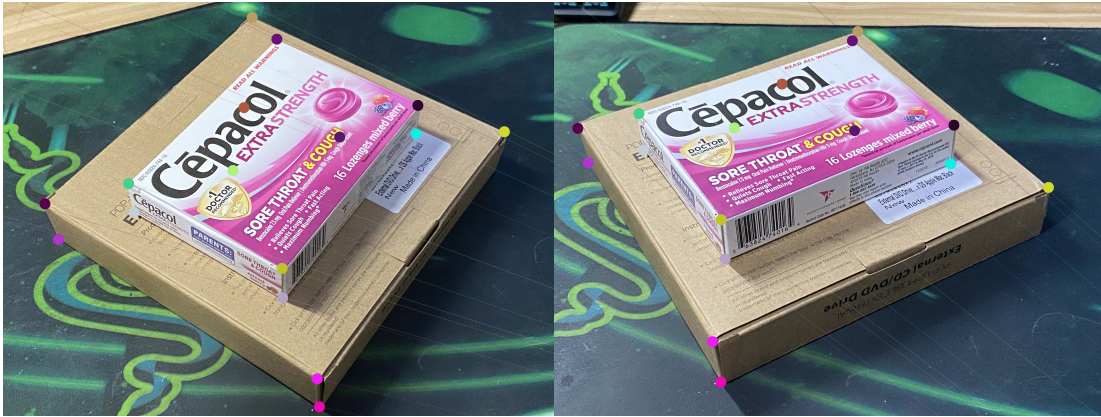


Figure 11: `img_1` and `img_2` output by the LoopAndZhang program.



Figure 12: `img_1_dst` and `img_2_dst` output by the LoopAndZhang program.



Figure 13: The left and right images used in Task 3.

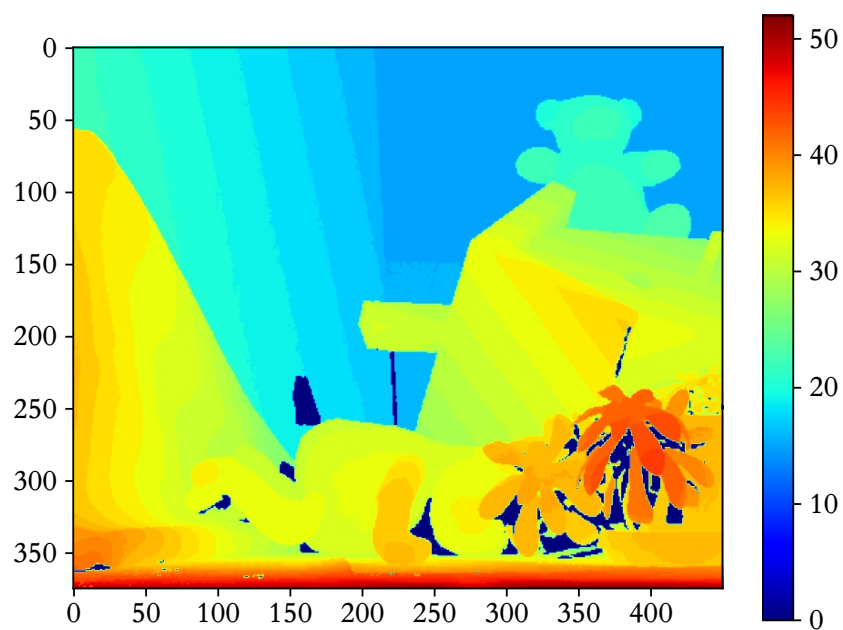


Figure 14: The ground truth disparity map for Task 3.

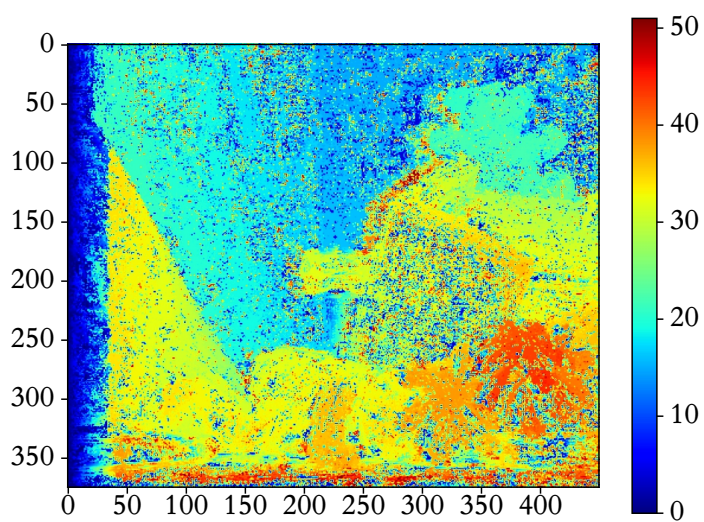


Figure 15: The output disparity map for  $M = 7$ .

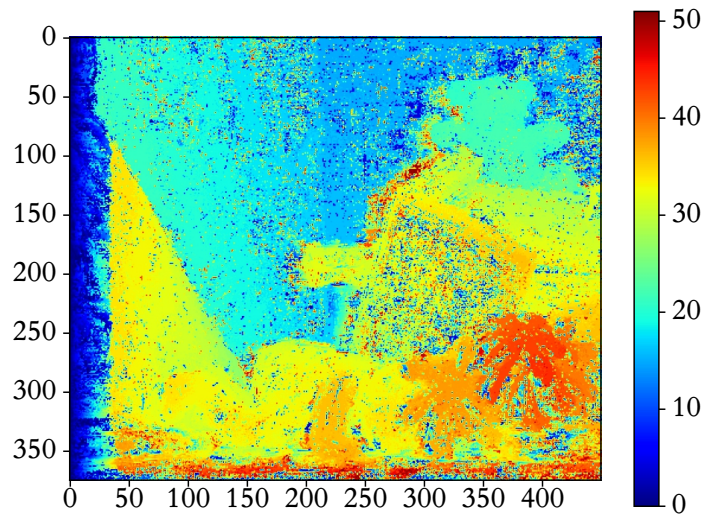


Figure 16: The output disparity map for  $M = 9$ .

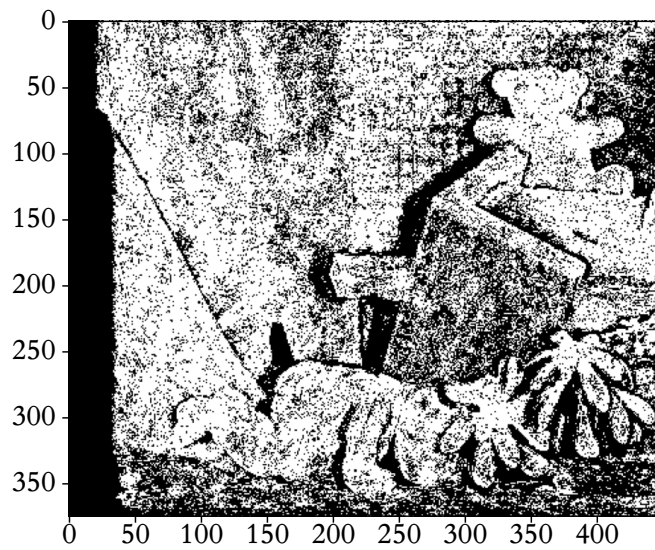


Figure 17: The error map for  $M = 7$ .

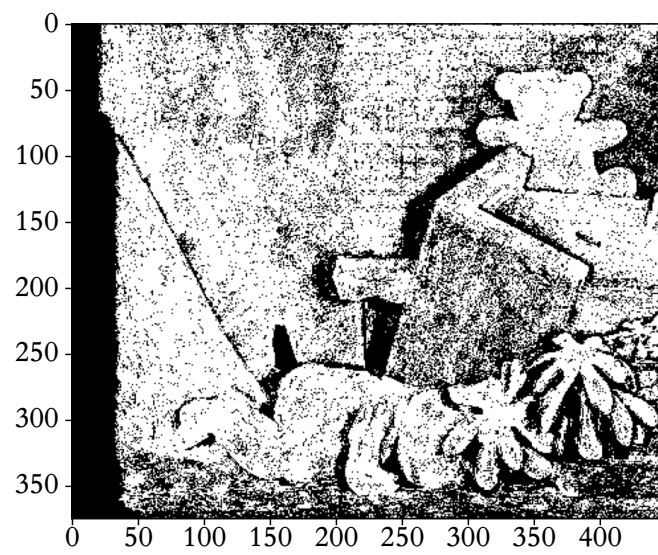


Figure 18: The error map for  $M = 9$ .

# A. Source Code

## A.1. Source Listing of code/common.py

```
1 import matplotlib.pyplot as plt
2 plt.rcParams['svg.fonttype'] = 'none'
3
4 from IPython.display import set_matplotlib_formats
5 set_matplotlib_formats('svg')
6
7 import os
8 import lxml.html
9 import lxml.etree
10 from PIL import Image
11 import io
12 import base64
13
14 notebook_path = os.path.abspath(os.getcwd())
15 tex_src_path = os.path.split(notebook_path)[0]
16
17 print(f'notebook path is {notebook_path}')
18 print(f'TeX source path is {tex_src_path}')
19
20 # use jpeg compression
21 def save_jpeg_svg(out_fn, **savefig_kwargs):
22     bio = io.BytesIO()
23     plt.savefig(bio, format='svg', **savefig_kwargs)
24     bio.seek(0)
25     svg = lxml.html.parse(bio)
26     for ele in svg.findall('//image'):
27         image_str = ele.get('xlink:href')
28         if image_str.startswith('data:image/'):
29             image_format, _, image_b64 = image_str.partition(',')
30             image_bytes = base64.b64decode(image_b64.encode())
31             imgbio = io.BytesIO()
32             imgbio.write(image_bytes)
33             imgbio.seek(0)
34             image_data = Image.open(imgbio)
35             if image_data.mode == 'RGBA':
36                 image_data = image_data.convert('RGB')
37             # save the image as jpeg
38             imgbio = io.BytesIO()
39             image_data.save(imgbio, format='jpeg', quality=100)
40             # encode as b64
41             image_jpeg_b64 = 'data:image/jpeg;base64,' + base64.b64encode(imgbio.getvalue()).decode()
42             ele.set('xlink:href', image_jpeg_b64)
43             print('successfully encoded one image element using jpeg')
44     svg_root = svg.find('//svg')
45     new_svg = lxml.etree.tostring(svg_root, pretty_print=True)
46     with open(out_fn, 'wb') as outfile:
47         outfile.write(new_svg)
48
49 import numpy as np
50
51 def table_to_latex(table, table_style='booktabs'):
52     nrows = len(table)
53     ncols = len(table[0])
54     for row in table:
55         assert len(row) == ncols
56
57     if table_style == 'booktabs':
58         tabular_spec = 'c' * ncols
59     else:
60         tabular_spec = '|c|' * 'c|' * (ncols-1)
61
62     # find length
63     str_array = np.asarray(table, dtype=np.object)
64     str_array_len = np.vectorize(len)(str_array)
65     col_len = np.max(str_array_len, axis=0)
66     def row_formatter(*cols):
67         res = []
68         for i, col_data in enumerate(cols):
69             string_fmt = '{:%ds}' % col_len[i]
70             res.append(string_fmt.format(col_data))
71         return res
72
73     latex_rows = [' & '.join(row_formatter(*row)) for row in table]
74     if table_style == 'booktabs':
75         latex_content = latex_rows[0] + '\\midrule \n'
76         end_of_line = '\\ \n'
77         latex_content += end_of_line.join(latex_rows[1:]) + '\\bottomrule \n'
78     else:
79         end_of_line = '\\hline\n'
80         latex_content = end_of_line.join(latex_rows) + end_of_line
81
82     if table_style == 'booktabs':
83         latex_string = '\\begin{tabular}{%s}\n\\toprule\n%s\\end{tabular}' % (tabular_spec, latex_content)
84     else:
85         latex_string = '\\begin{tabular}{%s}\n\\hline\n%s\\end{tabular}' % (tabular_spec, latex_content)
86
87     return latex_string
88
89 def latex_e_notation(s):
90     if 'e' in s:
91         front, e, exp = s.partition('e')
92         exp = int(exp)
93         if exp != 0:
94             s = r'{}\times 10^{{{}}}'.format(front, exp)
95         else:
96             s = front
97     return s
98
99 def latex_2d_array(arr, fmt=':06f'):
100     arr = np.asarray(arr)
101     assert arr.ndim == 2
102
103     fmt_str = '{%s}' % fmt
104
105     row_strs = []
106     for row in arr.tolist():
107         col_str = [fmt_str.format(col) for col in row]
108         for i in range(len(col_str)):
109             col_str[i] = latex_e_notation(col_str[i])
110
111         row_strs.append(
112             ' & '.join(col_str)
113         )
114
115     return '\\\\n'.join(row_strs)
```

A.2. Source Listing of code/point\_selector.py

```
1  # use matplotlib to manually select correspondences
2
3  import matplotlib.pyplot as plt
4  import skimage.io
5  import json
6  import os
7  from matplotlib.widgets import Button
8
9  this_dir = os.path.dirname(__file__)
10
11 input_img = os.path.join(this_dir, '..', 'figures', 'my_img2.jpg')
12 output_json = os.path.join(this_dir, '..', 'data', 'my_img2.json')
13
14 selections = []
15
16
17 # def button_onclick(event):
18 #     global selections
19 #     if len(selections) > 0:
20 #         selections.pop()
21 #         print('popped the last point')
22 #         print(selections)
23
24
25 img = skimage.io.imread(input_img)
26 plt.imshow(img)
27 fig = plt.gcf()
28
29 def canvas_onclick(event):
30     global selections
31     # print('%s click: button=%d, x=%d, y=%d, xdata=%f, ydata=%f' %
32     #       ('double' if event.dblclick else 'single', event.button,
33     #       event.x, event.y, event.xdata, event.ydata))
34     selections.append((event.xdata, event.ydata))
35     print(selections)
36     plt.plot(event.xdata, event.ydata, 'o', color='red')
37     fig.canvas.draw()
38
39 cid = fig.canvas.mpl_connect('button_press_event', canvas_onclick)
40
41 # ax_remove = fig.add_axes([0.81, 0.05, 0.1, 0.075])
42 # button_remove = Button(ax_remove, 'Remove')
43 # button_remove.on_clicked(button_onclick)
44
45 plt.show()
46
47 with open(output_json, 'w') as outfile:
48     json.dump(selections, outfile)
```

A.3. Source Listing of data/LoopAndZhang/src/main.cpp

```
1  #include <opencv2/core/core.hpp>
2  // #include <opencv2/highgui/highgui.hpp>
3
4  #include "util.hpp"
5
6  #include <iostream>
7  #include <cctype>
8
9  using namespace cv;
10 using namespace std;
11
12
13 template<typename T>
14 typename T::value_type vmax(const T &arr){
15     return *max_element(arr.begin(), arr.end());
16 }
17
18 template<typename T>
19 typename T::value_type vmin(const T &arr){
20     return *min_element(arr.begin(), arr.end());
21 }
22
23 int main(){
24     /***** EPIPOLAR GEOMETRY *****/
25     printf("OpenCV: %s", getBuildInformation().c_str());
26
27     // FIXME:
28     Mat img_1 = imread("../img/my_img3.jpg");
29     Mat img_2 = imread("../img/my_img4.jpg");
30
31     // Mat img_1 = imread("img/perra_7.jpg");
32     // Mat img_2 = imread("img/perra_8.jpg");
33
34     // Mat img_1 = imread("../img/madera_1.jpg");
35     // Mat img_2 = imread("../img/madera_2.jpg");
36
37     Mat fund_mat = Mat::zeros(3,3,CV_64F);
38
39     Vec3d epipole;
40
41     vector<Vec3d> lines_1, lines_2;
42     vector<Point2d> good_matches_1, good_matches_2;
43
44     // Get epipolar geometry
45     computeEpiLines(img_1, img_2,
46                    epipole, fund_mat,
47                    lines_1, lines_2,
48                    good_matches_1, good_matches_2);
49
50
51     Mat A, B, Ap, Bp;
52
53     Mat e_x = crossProductMatrix(epipole);
54
55     /***** PROJECTIVE *****/
56
57     // Get A,B matrix for minimizing z
58     obtainAB(img_1, e_x, A, B);
59     obtainAB(img_2, fund_mat, Ap, Bp);
60
61     // Get initial guess for z
62     Vec3d z = getInitialGuess(A, B, Ap, Bp);
63 }
```

```

64 // Optimizes the z solution
65 optimizeRoot(A, B, Ap, Bp, z);
66
67 // Get w
68 Mat w = e_x * Mat(z);
69 Mat wp = fund_mat * Mat(z);
70
71 w /= w.at<double>(2,0);
72 wp /= wp.at<double>(2,0);
73
74 // Get final H_p and Hp_p matrix for projection
75 Mat H_p = Mat::eye(3, 3, CV_64F);
76 H_p.at<double>(2,0) = w.at<double>(0,0);
77 H_p.at<double>(2,1) = w.at<double>(1,0);
78
79 Mat Hp_p = Mat::eye(3, 3, CV_64F);
80 Hp_p.at<double>(2,0) = wp.at<double>(0,0);
81 Hp_p.at<double>(2,1) = wp.at<double>(1,0);
82
83 /***** SIMILARITY *****/
84
85 // Get the translation term
86 double vp_c = getTranslationTerm(img_1, img_2, H_p, Hp_p);
87
88 // Get the H_r and Hp_r matrix directly
89 Mat H_r = Mat::zeros(3, 3, CV_64F);
90
91 H_r.at<double>(0,0) = fund_mat.at<double>(2,1) - w.at<double>(1,0) * fund_mat.at<double>(2,2);
92 H_r.at<double>(1,0) = fund_mat.at<double>(2,0) - w.at<double>(0,0) * fund_mat.at<double>(2,2);
93
94 H_r.at<double>(0,1) = w.at<double>(0,0) * fund_mat.at<double>(2,2) - fund_mat.at<double>(2,0);
95 H_r.at<double>(1,1) = H_r.at<double>(0,0);
96
97 H_r.at<double>(1,2) = fund_mat.at<double>(2,2) + vp_c;
98 H_r.at<double>(2,2) = 1.0;
99
100 Mat Hp_r = Mat::zeros(3, 3, CV_64F);
101
102 Hp_r.at<double>(0,0) = wp.at<double>(1,0) * fund_mat.at<double>(2,2) - fund_mat.at<double>(1,2);
103 Hp_r.at<double>(1,0) = wp.at<double>(0,0) * fund_mat.at<double>(2,2) - fund_mat.at<double>(0,2);
104
105 Hp_r.at<double>(0,1) = fund_mat.at<double>(0,2) - wp.at<double>(0,0) * fund_mat.at<double>(2,2);
106 Hp_r.at<double>(1,1) = Hp_r.at<double>(0,0);
107
108 Hp_r.at<double>(1,2) = vp_c;
109 Hp_r.at<double>(2,2) = 1.0;
110
111 /***** SHEARING *****/
112
113 Mat H_1 = H_r*H_p;
114 Mat H_2 = Hp_r*Hp_p;
115
116 Mat H_s, Hp_s;
117
118 // Get shearing transforms with the method described on the paper
119 getShearingTransforms(img_1, img_2, H_1, H_2, H_s, Hp_s);
120
121 /***** RECTIFY IMAGES *****/
122
123 Mat H = H_s * H_r * H_p;
124 Mat Hp = Hp_s * Hp_r * Hp_p;
125
126 vector<double> all_max_x, all_min_x;
127 vector<double> all_max_y, all_min_y;
128
129 // Get homography image of the corner coordinates from all the images
130 vector<Point2d> corners_all(4), corners_all_t(4);
131 double min_x, min_y, max_x, max_y;
132 min_x = min_y = +INF;
133 max_x = max_y = -INF;
134
135 corners_all[0] = Point2d(0,0);
136 corners_all[1] = Point2d(img_1.cols,0);
137 corners_all[2] = Point2d(img_1.cols,img_1.rows);
138 corners_all[3] = Point2d(0,img_1.rows);
139
140 perspectiveTransform(corners_all, corners_all_t, H);
141
142 for (int j = 0; j < 4; j++) {
143     min_x = min(corners_all_t[j].x, min_x);
144     max_x = max(corners_all_t[j].x, max_x);
145
146     min_y = min(corners_all_t[j].y, min_y);
147     max_y = max(corners_all_t[j].y, max_y);
148 }
149
150 int img_1_cols = max_x - min_x;
151 int img_1_rows = max_y - min_y;
152 all_max_x.push_back(max_x);
153 all_min_x.push_back(min_x);
154 all_max_y.push_back(max_y);
155 all_min_y.push_back(min_y);
156
157 // Get homography image of the corner coordinates from all the images
158 min_x = min_y = +INF;
159 max_x = max_y = -INF;
160
161 corners_all[0] = Point2d(0,0);
162 corners_all[1] = Point2d(img_2.cols,0);
163 corners_all[2] = Point2d(img_2.cols,img_2.rows);
164 corners_all[3] = Point2d(0,img_2.rows);
165
166 perspectiveTransform(corners_all, corners_all_t, Hp);
167
168 for (int j = 0; j < 4; j++) {
169     min_x = min(corners_all_t[j].x, min_x);
170     max_x = max(corners_all_t[j].x, max_x);
171
172     min_y = min(corners_all_t[j].y, min_y);
173     max_y = max(corners_all_t[j].y, max_y);
174 }
175
176 int img_2_cols = max_x - min_x;
177 int img_2_rows = max_y - min_y;
178 all_max_x.push_back(max_x);
179 all_min_x.push_back(min_x);
180 all_max_y.push_back(max_y);
181 all_min_y.push_back(min_y);
182
183
184 img_1_rows = img_2_rows = vmax(all_max_y) - vmin(all_min_y);
185 img_1_cols = img_2_cols = vmax(all_max_x) - vmin(all_min_x);

```

```

186
187 // Apply homographies
188 Mat img_1_dst(img_1_rows, img_1_cols, CV_64F);
189 Mat img_2_dst(img_2_rows, img_2_cols, CV_64F);
190
191 warpPerspective( img_1, img_1_dst, H, img_1_dst.size() );
192 warpPerspective( img_2, img_2_dst, Hp, img_2_dst.size() );
193
194 Vec3d epipole_dst;
195
196 vector<Vec3d> lines_1_dst, lines_2_dst;
197 vector<Point2d> good_matches_1_dst, good_matches_2_dst;
198
199 perspectiveTransform(good_matches_1, good_matches_1_dst, H);
200 perspectiveTransform(good_matches_2, good_matches_2_dst, Hp);
201
202
203 // Get epipolar geometry and draw epilines
204 computeEpilines(img_1_dst, img_2_dst, epipole_dst, fund_mat, lines_1_dst, lines_2_dst, good_matches_1_dst, good_matches_2_dst);
205
206 drawEpilines(img_1, img_2, lines_1, lines_2, good_matches_1, good_matches_2, 16);
207 drawEpilines(img_1_dst, img_2_dst, lines_1_dst, lines_2_dst, good_matches_1_dst, good_matches_2_dst, 16);
208
209 cout << "\nH = " << H << "\nHp = " << Hp << endl;
210
211 cout << "\nEpipolo antes: " << epipole/epipole[2] << "\nEpipolo después: " << epipole_dst << endl;
212
213 // draw(img_1, "1");
214 // draw(img_1_dst, "1 rectificada");
215
216 // char c = 'a';
217
218 // draw(img_2, "2");
219 // draw(img_2_dst, "2 rectificada");
220
221 // c = 'a';
222
223 // while (tolower(c) != 'q')
224 //   c = waitKey();
225
226 // destroyAllWindows();
227
228 Mat image_8U;
229 img_1.convertTo(image_8U, CV_8U);
230 imwrite("lnz_img_1.jpg", image_8U);
231
232 img_1_dst.convertTo(image_8U, CV_8U);
233 imwrite("lnz_img_1_dst.png", image_8U);
234
235 img_2.convertTo(image_8U, CV_8U);
236 imwrite("lnz_img_2.jpg", image_8U);
237
238 img_2_dst.convertTo(image_8U, CV_8U);
239 imwrite("lnz_img_2_dst.png", image_8U);
240
241 }

```

#### A.4. Source Listing of data/LoopAndZhang/src/util.cpp

```

1  #include "util.hpp"
2
3  string type2str(int type) {
4      string r;
5
6      uchar depth = type & CV_MAT_DEPTH_MASK;
7      uchar chans = 1 + (type >> CV_CN_SHIFT);
8
9      switch ( depth ) {
10         case CV_8U:  r = "8U"; break;
11         case CV_8S:  r = "8S"; break;
12         case CV_16U: r = "16U"; break;
13         case CV_16S: r = "16S"; break;
14         case CV_32S: r = "32S"; break;
15         case CV_32F: r = "32F"; break;
16         case CV_64F: r = "64F"; break;
17         default:    r = "User"; break;
18     }
19
20     r += "C";
21     r += (chans*'0');
22
23     return r;
24 }
25
26
27 double computeEpilines(Mat &one, Mat &other, Vec3d &epipole, Mat &fund_mat, vector<Vec3d> &lines_1, vector<Vec3d> &lines_2, vector<Point2d> &good_matches_1, vector<Point2d>
↩ &good_matches_2){
28
29     //fund_mat = fundamentalMat(one, other, good_matches_1, good_matches_2);
30     fund_mat = manualFundMat(good_matches_1, good_matches_2);
31
32     computeCorrespondEpilines(good_matches_1, 1, fund_mat, lines_2);
33     computeCorrespondEpilines(good_matches_2, 2, fund_mat, lines_1);
34
35     // Computes the distances between
36     // the lines and their corresponding points.
37     double distance_1 = 0.0, distance_2 = 0.0;
38     for (size_t i = 0; i < lines_1.size(); i++) {
39         Vec2d point_1 = good_matches_1[i];
40         Vec2d point_2 = good_matches_2[i];
41
42         Vec3d line_1 = lines_1[i];
43         Vec3d line_2 = lines_2[i];
44
45         // Error computation with distance point-to-line
46         distance_1 += abs(line_1[0]*point_2[0] +
47             line_1[1]*point_2[1] +
48             line_1[2]) /
49             sqrt(line_1[0]*line_1[0] + line_1[1]*line_1[1]);
50
51         distance_2 += abs(line_2[0]*point_1[0] +
52             line_2[1]*point_1[1] +
53             line_2[2]) /
54             sqrt(line_2[0]*line_2[0] + line_2[1]*line_2[1]);
55     }
56
57     // The epipole is the left-null vector of F

```

```

58     Mat epi_mat;
59     SVD::solveZ(fund_mat, epi_mat);
60
61     // It seems that this assignment is outdated
62     // Thanks @Judalucifer from Github
63     //
64     // epipole[0] = epi_mat.at<double>(0,0);
65     // epipole[1] = epi_mat.at<double>(0,1);
66     // epipole[2] = epi_mat.at<double>(0,2);
67
68     epipole[0] = epi_mat.at<double>(0,0);
69     epipole[1] = epi_mat.at<double>(1,0);
70     epipole[2] = epi_mat.at<double>(2,0);
71
72     return (distance_1+distance_2)/(2*lines_1.size());
73 }
74
75
76 void drawEpilines(Mat &one, Mat &other, vector<Vec3d> &lines_1, vector<Vec3d> &lines_2, vector<Point2d> &good_matches_1, vector<Point2d> &good_matches_2, int num_lines){
77     RNG rng;
78     theRNG().state = clock();
79
80     for (size_t i = 0; i < lines_1.size(); i++) {
81         Vec2d point_1 = good_matches_1[i];
82         Vec2d point_2 = good_matches_2[i];
83
84         Vec3d line_1 = lines_1[i];
85         Vec3d line_2 = lines_2[i];
86         // Draws only num_lines lines
87         if(i % (lines_1.size()/num_lines) == 0 ){
88             Scalar color(rng.uniform(0, 255),
89                         rng.uniform(0, 255),
90                         rng.uniform(0, 255));
91
92             line(one,
93                 Point(0, -line_1[2]/line_1[1]),
94                 Point(one.cols, -(line_1[2] + line_1[0]*one.cols)/line_1[1]),
95                 color
96             );
97             circle(one,
98                 Point2d(point_1[0], point_1[1]),
99                 40,
100                color,
101                // CV_FILLED);
102                FILLED);
103
104             line(other,
105                 Point(0,
106                     -line_2[2]/line_2[1]),
107                 Point(other.cols,
108                     -(line_2[2] + line_2[0]*other.cols)/line_2[1]),
109                 color
110             );
111             circle(other,
112                 Point2d(point_2[0], point_2[1]),
113                 40,
114                 color,
115                 // CV_FILLED);
116                 FILLED);
117         }
118     }
119 }
120
121
122 Mat fundamentalMat(Mat &one, Mat &other,
123                   vector<Point2d> &good_matches_1,
124                   vector<Point2d> &good_matches_2){
125
126     pair<vector<Point2d>, vector<Point2d> > matches;
127     Mat F;
128     vector<unsigned char> mask;
129
130     vector<Point2d> first, second;
131     int flag = FM_8POINT;
132
133     if (good_matches_1.empty() && good_matches_2.empty()){
134         matches = match(one, other, descriptor_id::BRUTE_FORCE, detector_id::BRISK);
135         first = matches.first;
136         second = matches.second;
137         flag |= FM_RANSAC;
138     }
139     else{
140         first = good_matches_1;
141         second = good_matches_2;
142     }
143
144
145     F = findFundamentalMat(first, second,
146                           flag,
147                           1., 0.99, mask );
148
149     vector<Point2d> final_1, final_2;
150
151     for (size_t i = 0; i < mask.size(); i++) {
152         if(mask[i] == 1){
153             final_1.push_back(first[i]);
154             final_2.push_back(second[i]);
155         }
156     }
157
158     good_matches_1 = vector<Point2d>(final_1);
159     good_matches_2 = vector<Point2d>(final_2);
160
161     return F;
162 }
163
164
165 pair< vector<Point2d>, vector<Point2d> > match(Mat &one, Mat &other, enum descriptor_id descriptor , enum detector_id detector){
166     // 1 - Get keypoints and its descriptors in both images
167     vector<KeyPoint> keypoints[2];
168     Mat descriptors[2];
169
170     descriptors[0] = detectFeatures(one, detector, keypoints[0]);
171     descriptors[1] = detectFeatures(other, detector, keypoints[1]);
172
173     // 2 - Match both descriptors using required detector
174     // Declare the matcher
175     Ptr<DescriptorMatcher> matcher;
176
177     // Define the matcher
178     if (descriptor == descriptor_id::BRUTE_FORCE) {
179         // For ORB and BRISK descriptors, NORM_HAMMING should be used.

```

```

180         // See http://sl.ugr.es/norm_ORB_BRISK
181         matcher = new BFMatcher(NORM_HAMMING, true);
182     }
183     else{
184         matcher = new FlannBasedMatcher();
185         // FlannBased Matcher needs CV_32F descriptors
186         // See http://sl.ugr.es/FlannBase_32F
187         for (size_t i = 0; i < 2; i++) {
188             if (descriptors[i].type() != CV_32F) {
189                 descriptors[i].convertTo(descriptors[i], CV_32F);
190             }
191         }
192     }
193
194     // Match!
195     vector<DMatch> matches;
196     matcher->match( descriptors[0], descriptors[1], matches );
197
198     // 3 - Create lists of ordered keypoints following obtained matches
199     vector<Point2d> ordered_keypoints[2];
200
201     for( unsigned int i = 0; i < matches.size(); i++ )
202     {
203         // Get the keypoints from the matches
204         ordered_keypoints[0].push_back( keypoints[0][matches[i].queryIdx].pt );
205         ordered_keypoints[1].push_back( keypoints[1][matches[i].trainIdx].pt );
206     }
207
208     return pair< vector<Point2d>, vector<Point2d> >(ordered_keypoints[0], ordered_keypoints[1]);
209 }
210
211
212 Mat detectFeatures(Mat image, enum detector_id det_id, vector<KeyPoint> &keypoints){
213     // Declare detector
214     Ptr<Feature2D> detector;
215
216     // Define detector
217     if (det_id == detector_id::ORB) {
218         // Declare ORB detector
219         detector = ORB::create(
220             500,           //nfeatures = 500
221             1.2f,          //scaleFactor = 1.2f
222             4,             //nlevels = 8
223             21,            //edgeThreshold = 31
224             0,             //firstLevel = 0
225             2,             //WTA_K = 2
226             ORB::HARRIS_SCORE, //scoreType = ORB::HARRIS_SCORE
227             21,            //patchSize = 31
228             20             //fastThreshold = 20
229         );
230     }
231     else{
232         // Declare BRISK and BRISK detectors
233         detector = BRISK::create(
234             30,           // thresh = 30
235             3,            // octaves = 3
236             1.0f          // patternScale = 1.0f
237         );
238     }
239
240     // Declare array for storing the descriptors
241     Mat descriptors;
242
243     // Detect and compute!
244     detector->detect(image, keypoints);
245     detector->compute(image, keypoints, descriptors);
246
247     return descriptors;
248 }
249
250
251 void draw(Mat img, string name){
252     namedWindow( name, WINDOW_AUTOSIZE );
253
254     // Converts to 8-bits unsigned int to avoid problems
255     // in OpenCV implementations in Microsoft Windows.
256     Mat image_8U;
257     img.convertTo(image_8U, CV_8U);
258
259     imshow( name, image_8U );
260 }
261
262
263 void obtainAB(const Mat &img, const Mat &mult_mat, Mat &A, Mat &B){
264     int width = img.cols;
265     int height = img.rows;
266
267     int size = 3;
268
269     Mat PPt = Mat::zeros(size, size, CV_64F);
270
271     PPt.at<double>(0,0) = width*width - 1;
272     PPt.at<double>(1,1) = height*height - 1;
273
274     PPt *= (width*height) / 12.0;
275
276     double w_1 = width - 1;
277     double h_1 = height - 1;
278
279     double values[3][3] = {
280         {w_1*w_1, w_1*h_1, 2*w_1},
281         {w_1*h_1, h_1*h_1, 2*h_1},
282         {2*w_1, 2*h_1, 4}
283     };
284
285     Mat pcpct(size, size, CV_64F, values);
286
287     pcpct /= 4;
288     A = mult_mat.t() * PPt * mult_mat;
289     B = mult_mat.t() * pcpct * mult_mat;
290 }
291
292
293 Mat crossProductMatrix(Vec3d elem){
294     double values[3][3] = {
295         {0, -elem[2], elem[1]},
296         {elem[2], 0, -elem[0]},
297         {-elem[1], elem[0], 0}
298     };
299
300     Mat sol(3, 3, CV_64F, values);
301 }

```

```

302     return sol.clone();
303 }
304
305
306 Vec3d maximize(Mat &A, Mat &B){
307     Mat D; // Output of cholesky decomposition: upper triangular matrix.
308     if( choleskyCustomDecomp(A, D) ){
309
310         Mat D_inv = D.inv();
311
312         Mat DBD = D_inv.t() * B * D_inv;
313
314         // Solve the equations system using eigenvalue decomposition
315
316         Mat eigenvalues, eigenvectors;
317         eigen(DBD, eigenvalues, eigenvectors);
318
319         // Take largest eigenvector
320         Mat y = eigenvectors.row(0);
321
322         Mat sol = D_inv*y.t();
323
324         Vec3d res(sol.at<double>(0,0), sol.at<double>(1,0), sol.at<double>(2,0));
325
326         return res;
327     }
328
329     // At this point, there is an error!
330     Mat eigenvalues;
331     eigen(A, eigenvalues);
332
333     cout << "\n\n\n----- ERROR -----" << endl;
334     cout << "A = " << A << endl;
335     cout << "A eigenvalues: " << eigenvalues << endl << endl;
336
337     return Vec3d(0, 0, 0);
338 }
339
340
341 Vec3d getInitialGuess(Mat &A, Mat &B, Mat &Ap, Mat &Bp){
342
343     Vec3d z_1 = maximize(A, B);
344     Vec3d z_2 = maximize(Ap, Bp);
345
346     return (normalize(z_1) + normalize(z_2))/2;
347 }
348
349
350 Mat manualFundMat( vector<Point2d> &good_matches_1,
351                   vector<Point2d> &good_matches_2){
352     // Taking points by hand
353     vector<Point> origin, destination;
354
355     // origin.push_back(Point(63, 31));
356     // origin.push_back(Point(69, 39));
357     // origin.push_back(Point(220, 13));
358     // origin.push_back(Point(444, 23));
359     // origin.push_back(Point(355, 45));
360     // origin.push_back(Point(347, 55));
361     // origin.push_back(Point(80, 319));
362     // origin.push_back(Point(85, 313));
363     // origin.push_back(Point(334, 371));
364     // origin.push_back(Point(342, 381));
365
366     // origin.push_back(Point(213, 126));
367     // origin.push_back(Point(298, 158));
368     // origin.push_back(Point(219, 266));
369
370     // destination.push_back(Point(159, 51));
371     // destination.push_back(Point(167, 59));
372     // destination.push_back(Point(81, 28));
373     // destination.push_back(Point(293, 20));
374     // destination.push_back(Point(440, 38));
375     // destination.push_back(Point(435, 45));
376     // destination.push_back(Point(171, 372));
377     // destination.push_back(Point(178, 363));
378     // destination.push_back(Point(420, 305));
379     // destination.push_back(Point(424, 311));
380
381     // destination.push_back(Point(188, 140));
382     // destination.push_back(Point(235, 156));
383     // destination.push_back(Point(202, 278));
384
385     origin.push_back(Point(306,1474));
386     origin.push_back(Point(405,1734));
387     origin.push_back(Point(2311,2956));
388     origin.push_back(Point(2295,2752));
389     origin.push_back(Point(3662,947));
390     origin.push_back(Point(1820,112));
391     origin.push_back(Point(2000,271));
392     origin.push_back(Point(917,1326));
393     origin.push_back(Point(957,1538));
394     origin.push_back(Point(2043,2161));
395     origin.push_back(Point(2039,1945));
396     origin.push_back(Point(3018,963));
397     origin.push_back(Point(3030,755));
398     origin.push_back(Point(1680,1230));
399     origin.push_back(Point(1760,771));
400     origin.push_back(Point(2459,986));
401     destination.push_back(Point(174,923));
402     destination.push_back(Point(250,1174));
403     destination.push_back(Point(1220,2776));
404     destination.push_back(Point(1161,2481));
405     destination.push_back(Point(3618,1354));
406     destination.push_back(Point(2223,215));
407     destination.push_back(Point(2183,283));
408     destination.push_back(Point(621,815));
409     destination.push_back(Point(653,1082));
410     destination.push_back(Point(1252,1877));
411     destination.push_back(Point(1224,1590));
412     destination.push_back(Point(2902,1182));
413     destination.push_back(Point(2926,899));
414     destination.push_back(Point(1320,919));
415     destination.push_back(Point(1668,599));
416     destination.push_back(Point(2199,939));
417
418     vector<unsigned char> mask;
419     Mat fund_mat = findFundamentalMat(origin, destination,
420                                     FM_8POINT | FM_RANSAC,
421                                     20, 0.99, mask );
422
423     for (size_t i = 0; i < mask.size(); i++) {

```

```

424         if(!mask[i] == 1){
425             good_matches_1.push_back(origin[i]);
426             good_matches_2.push_back(destination[i]);
427         }
428     }
429
430     return fund_mat;
431 }
432
433
434 double getTranslationTerm(const Mat &img_1, const Mat &img_2, const Mat &H_p, const Mat &Hp_p){
435     double min_1 = getMinYCoord(img_1, H_p);
436     double min_2 = getMinYCoord(img_2, Hp_p);
437
438     double offset = min_1 < min_2 ? min_1 : min_2;
439
440     return -offset;
441 }
442
443
444 double getMinYCoord(const Mat &img, const Mat &homography){
445     vector<Point2d> corners(4), corners_trans(4);
446
447     corners[0] = Point2d(0,0);
448     corners[1] = Point2d(img.cols,0);
449     corners[2] = Point2d(img.cols,img.rows);
450     corners[3] = Point2d(0,img.rows);
451
452     perspectiveTransform(corners, corners_trans, homography);
453
454     double min_y;
455     min_y = +INF;
456
457     for (int j = 0; j < 4; j++) {
458         min_y = min(corners_trans[j].y, min_y);
459     }
460
461     return min_y;
462 }
463
464
465 Mat getS(const Mat &img, const Mat &homography){
466     int w = img.cols;
467     int h = img.rows;
468
469     Point2d a((w-1)/2, 0);
470     Point2d b(w-1, (h-1)/2);
471     Point2d c((w-1)/2, h-1);
472     Point2d d(0, (h-1)/2);
473
474     vector<Point2d> midpoints, midpoints_hat;
475     midpoints.push_back(a);
476     midpoints.push_back(b);
477     midpoints.push_back(c);
478     midpoints.push_back(d);
479
480     perspectiveTransform(midpoints, midpoints_hat, homography);
481
482     Point2d x = midpoints_hat[1] - midpoints_hat[3];
483     Point2d y = midpoints_hat[2] - midpoints_hat[0];
484
485     double coeff_a = (h*h*x.y*x.y + w*w*y.y*y.y) / (h*w * (x.y*y.x - x.x*y.y));
486     double coeff_b = (h*h*x.x*x.y + w*w*y.x*y.y) / (h*w * (x.x*y.y - x.y*y.x));
487
488     Mat S = Mat::eye(3, 3, CV_64F);
489     S.at<double>(0,0) = coeff_a;
490     S.at<double>(0,1) = coeff_b;
491
492     Vec3d x_hom(x.x, x.y, 0.0);
493     Vec3d y_hom(y.x, y.y, 0.0);
494
495     if( coeff_a < 0 ){
496         coeff_a *= -1;
497         coeff_b *= -1;
498
499         S.at<double>(0,0) = coeff_a;
500         S.at<double>(0,1) = coeff_b;
501     }
502
503
504     Mat EQ18 = (S * Mat(x_hom)).t() * (S * Mat(y_hom));
505     cout << "EQ18 " << EQ18 << endl;
506
507     Mat EQ19 = ((S * Mat(x_hom)).t() * (S * Mat(x_hom))) / ((S * Mat(y_hom)).t() * (S * Mat(y_hom))) - (1.*w*w)/(1.*h*h);
508     cout << "EQ19 " << EQ19 << endl;
509
510     return S;
511 }
512
513
514 void getShearingTransforms(const Mat &img_1, const Mat &img_2,
515                             const Mat &H_1, const Mat &H_2,
516                             Mat &H_s, Mat &Hp_s){
517
518     Mat S = getS(img_1, H_1);
519     Mat Sp = getS(img_2, H_2);
520
521     double A = img_1.cols*img_1.rows + img_2.cols*img_2.rows;
522     double Ap = 0;
523
524     vector<Point2f> corners(4), corners_trans(4);
525
526     corners[0] = Point2f(0,0);
527     corners[1] = Point2f(img_1.cols,0);
528     corners[2] = Point2f(img_1.cols,img_1.rows);
529     corners[3] = Point2f(0,img_1.rows);
530
531     perspectiveTransform(corners, corners_trans, S*H_1);
532     Ap += contourArea(corners_trans);
533
534     float min_x_1, min_y_1;
535     min_x_1 = min_y_1 = +INF;
536     for (int j = 0; j < 4; j++) {
537         min_x_1 = min(corners_trans[j].x, min_x_1);
538         min_y_1 = min(corners_trans[j].y, min_y_1);
539     }
540
541     corners[0] = Point2f(0,0);
542     corners[1] = Point2f(img_2.cols,0);
543     corners[2] = Point2f(img_2.cols,img_2.rows);
544     corners[3] = Point2f(0,img_2.rows);
545

```

```

546     perspectiveTransform(corners, corners_trans, Sp*H_2);
547     Ap += contourArea(corners_trans);
548
549     float min_x_2, min_y_2;
550     min_x_2 = min_y_2 = +INF;
551     for (int j = 0; j < 4; j++) {
552         min_x_2 = min(corners_trans[j].x, min_x_2);
553         min_y_2 = min(corners_trans[j].y, min_y_2);
554     }
555
556     double scale = sqrt(A/Ap);
557
558     double min_y = min_y_1 < min_y_2 ? min_y_1 : min_y_2;
559
560     // We define W2 as the scale transformation and W1 as the translation
561     // transformation. Then, W = W1*W2.
562
563     Mat W;
564     Mat Wp;
565
566     Mat W_1 = Mat::eye(3, 3, CV_64F);
567     Mat Wp_1 = Mat::eye(3, 3, CV_64F);
568
569     Mat W_2 = Mat::eye(3, 3, CV_64F);
570     Mat Wp_2 = Mat::eye(3, 3, CV_64F);
571
572     W_2.at<double>(0,0) = W_2.at<double>(1,1) = scale;
573     Wp_2.at<double>(0,0) = Wp_2.at<double>(1,1) = scale;
574
575     if(isImageInverted(img_1, W_2*H_1)){
576         W_2.at<double>(0,0) = W_2.at<double>(1,1) = -scale;
577         Wp_2.at<double>(0,0) = Wp_2.at<double>(1,1) = -scale;
578     }
579
580     corners[0] = Point2d(0,0);
581     corners[1] = Point2d(img_1.cols,0);
582     corners[2] = Point2d(img_1.cols,img_1.rows);
583     corners[3] = Point2d(0,img_1.rows);
584
585     perspectiveTransform(corners, corners_trans, W_2*S*H_1);
586
587     min_x_1 = min_y_1 = +INF;
588     for (int j = 0; j < 4; j++) {
589         min_x_1 = min(corners_trans[j].x, min_x_1);
590         min_y_1 = min(corners_trans[j].y, min_y_1);
591     }
592
593     corners[0] = Point2d(0,0);
594     corners[1] = Point2d(img_2.cols,0);
595     corners[2] = Point2d(img_2.cols,img_2.rows);
596     corners[3] = Point2d(0,img_2.rows);
597
598     perspectiveTransform(corners, corners_trans, Wp_2*Sp*H_2);
599
600     min_x_2 = min_y_2 = +INF;
601     for (int j = 0; j < 4; j++) {
602         min_x_2 = min(corners_trans[j].x, min_x_2);
603         min_y_2 = min(corners_trans[j].y, min_y_2);
604     }
605
606     min_y = min_y_1 < min_y_2 ? min_y_1 : min_y_2;
607
608     W_1.at<double>(0,2) = -min_x_1;
609     Wp_1.at<double>(0,2) = -min_x_2;
610
611     W_1.at<double>(1,2) = Wp_1.at<double>(1,2) = -min_y;
612
613     W = W_1*W_2;
614     Wp = Wp_1*Wp_2;
615
616     H_s = W*S;
617     Hp_s = Wp*Sp;
618 }
619
620
621 bool choleskyCustomDecomp(const Mat &A, Mat &L){
622     L = Mat::zeros(3,3,CV_64F);
623
624     for (int i = 0; i < 3; i++){
625         for (int j = 0; j <= i; j++){
626             double sum = 0;
627             for (int k = 0; k < j; k++){
628                 sum += L.at<double>(i,k) * L.at<double>(j,k);
629             }
630
631             L.at<double>(i,j) = A.at<double>(i,j) - sum;
632             if (i == j){
633                 if (L.at<double>(i,j) < 0.0){
634                     if (L.at<double>(i,j) > -1e-5){
635                         L.at<double>(i,j) *= -1;
636                     }
637                     else{
638                         cout << "ERROR: " << L.at<double>(i,j) << endl;
639                         return false;
640                     }
641                 }
642                 L.at<double>(i,j) = sqrt(L.at<double>(i,j));
643             }
644             else{
645                 L.at<double>(i,j) /= L.at<double>(j,j);
646             }
647         }
648     }
649 }
650
651 L = L.t();
652
653 return true;
654 }
655
656
657
658 bool isImageInverted(const Mat &img, const Mat &homography){
659     vector<Point2d> corners(2), corners_trans(2);
660
661     corners[0] = Point2d(0,0);
662     corners[1] = Point2d(0,img.rows);
663
664     perspectiveTransform(corners, corners_trans, homography);
665
666     return corners_trans[1].y - corners_trans[0].y < 0.0;
667 }

```

```

668
669 vector< vector<double> > MatToVector(const Mat &mat){
670     vector< vector<double> > array;
671
672     for (size_t i = 0; i < mat.rows; i++) {
673         vector<double> row;
674
675         for (size_t j = 0; j < mat.cols; j++) {
676             row.push_back(mat.at<double>(i,j));
677         }
678
679         array.push_back(row);
680     }
681
682     return array;
683 }
684
685
686 double myfunction(const Mat &A, const Mat &B,
687                  const Mat &Ap, const Mat &Bp,
688                  double x){
689     vector< vector<double> > a = MatToVector(A);
690     vector< vector<double> > b = MatToVector(B);
691     vector< vector<double> > ap = MatToVector(Ap);
692     vector< vector<double> > bp = MatToVector(Bp);
693
694     double summ_1 =
695     (2*ap[0][0]*x+ap[1][0]+ap[0][1])/(x*(bp[0][0]*x+bp[0][1])+bp[1][0]*x+bp[1][1]);
696
697     double den_summ_2 = x*(bp[0][0]*x+bp[0][1])+bp[1][0]*x+bp[1][1];
698     den_summ_2 = den_summ_2+den_summ_2;
699
700     double summ_2 =
701     ((2*bp[0][0]*x+bp[1][0]+bp[0][1])*(x*(ap[0][0]*x+ap[0][1])+ap[1][0]*x+ap[1][1]))/den_summ_2;
702
703     double summ_3 =
704     (2*a[0][0]*x+a[1][0]+a[0][1])/(x*(b[0][0]*x+b[0][1])+b[1][0]*x+b[1][1]);
705
706     double den_summ_4 = x*(b[0][0]*x+b[0][1])+b[1][0]*x+b[1][1];
707     den_summ_4 = den_summ_4 * den_summ_4;
708
709     double summ_4 =
710     ((2*b[0][0]*x+b[1][0]+b[0][1])*(x*(a[0][0]*x+a[0][1])+a[1][0]*x+a[1][1]))/den_summ_4;
711
712     return summ_1 - summ_2 + summ_3 - summ_4;
713 }
714
715
716 double derivative(const Mat &A, const Mat &B,
717                  const Mat &Ap, const Mat &Bp,
718                  double x){
719     vector< vector<double> > a = MatToVector(A);
720     vector< vector<double> > b = MatToVector(B);
721     vector< vector<double> > ap = MatToVector(Ap);
722     vector< vector<double> > bp = MatToVector(Bp);
723
724
725     double summ_1 = (2*ap[0][0])/(x*(bp[0][0]*x+bp[0][1])+bp[1][0]*x+bp[1][1]);
726
727     double den_summ_2 = (x*(bp[0][0]*x+bp[0][1])+bp[1][0]*x+bp[1][1]);
728     den_summ_2 = den_summ_2 * den_summ_2;
729
730     double summ_2 =
731     (2*bp[0][0]*(x*(ap[0][0]*x+ap[0][1])+ap[1][0]*x+ap[1][1]))/den_summ_2;
732
733     double den_summ_3 = (x*(bp[0][0]*x+bp[0][1])+bp[1][0]*x+bp[1][1]);
734     den_summ_3 = den_summ_3 * den_summ_3;
735
736     double summ_3 =
737     (2*(2*ap[0][0]*x+ap[1][0]+ap[0][1])*(2*bp[0][0]*x+bp[1][0]+bp[0][1]))/den_summ_3;
738
739     double den_summ_4 = (x*(bp[0][0]*x+bp[0][1])+bp[1][0]*x+bp[1][1]);
740     den_summ_4 = den_summ_4 * den_summ_4 * den_summ_4;
741
742     double aux_num_summ_4 = (2*bp[0][0]*x+bp[1][0]+bp[0][1]);
743     aux_num_summ_4 = aux_num_summ_4 * aux_num_summ_4;
744
745     double summ_4 =
746     (2*aux_num_summ_4*(x*(ap[0][0]*x+ap[0][1])+ap[1][0]*x+ap[1][1]))/den_summ_4;
747
748     double summ_5 = (2*a[0][0])/(x*(b[0][0]*x+b[0][1])+b[1][0]*x+b[1][1]);
749
750
751     double den_summ_6 = (x*(b[0][0]*x+b[0][1])+b[1][0]*x+b[1][1]);
752     den_summ_6 = den_summ_6 * den_summ_6;
753
754     double summ_6 =
755     (2*b[0][0]*(x*(a[0][0]*x+a[0][1])+a[1][0]*x+a[1][1]))/den_summ_6;
756
757     double den_summ_7 = (x*(b[0][0]*x+b[0][1])+b[1][0]*x+b[1][1]);
758     den_summ_7 = den_summ_7 * den_summ_7;
759
760     double summ_7 =
761     (2*(2*a[0][0]*x+a[1][0]+a[0][1])*(2*b[0][0]*x+b[1][0]+b[0][1]))/den_summ_7;
762
763     double den_summ_8 = (x*(b[0][0]*x+b[0][1])+b[1][0]*x+b[1][1]);
764     den_summ_8 = den_summ_8 * den_summ_8 * den_summ_8;
765
766     double aux_num_summ_8 = (2*b[0][0]*x+b[1][0]+b[0][1]);
767     aux_num_summ_8 = aux_num_summ_8 * aux_num_summ_8;
768
769     double summ_8 =
770     (2*aux_num_summ_8*(x*(a[0][0]*x+a[0][1])+a[1][0]*x+a[1][1]))/den_summ_8;
771
772     return summ_1 - summ_2 - summ_3 + summ_4 + summ_5 - summ_6 - summ_7 + summ_8;
773 }
774
775
776 double NewtonRaphson(const Mat &A, const Mat &B,
777                     const Mat &Ap, const Mat &Bp,
778                     double init_guess){
779     double current = init_guess;
780     double previous;
781
782     double fx = myfunction(A,B,Ap,Bp, current);
783     double dfx = derivative(A,B,Ap,Bp, current);
784
785     cout << "\n\nPrimera aproximación de z = " << current << " con derivada = " << fx << endl;
786     int iterations = 0;
787
788     do {
789         previous = current;

```

```
790         current = current - fx / dfx;
791
792         fx = myfunction(A,B,Ap,Bp, current);
793         dfx = derivative(A,B,Ap,Bp, current);
794
795         iterations++;
796     } while (abs(fx) > 1e-15 && iterations < 150);
797     // Double-precision values have 15 stable decimal positions
798     cout << "Aproximación mejorada de z = " << current << " con derivada = " << fx << "\n" << endl;
799     return current;
800 }
801
802
803 void optimizeRoot(const Mat &A, const Mat &B,
804                 const Mat &Ap, const Mat &Bp,
805                 Vec3d &z){
806
807     double lambda = z[0];
808
809     z[0] = NewtonRaphson(A,B,Ap,Bp, lambda);
810     z[1] = 1.0;
811     z[2] = 0.0;
812 }
```

A.5. Source Listing of code/09.ipynb

```
In []: %load_ext autoreload
%autoreload 2
# %aimport common
from common import *
plt.rcParams['figure.dpi'] = 180
# from common import save_jpeg_svg

In []: import os
from tabulate import tabulate
from pathlib import Path
import sympy as sp
from scipy.optimize import minimize
import numpy as np
import scipy.signal
import skimage, skimage.io, skimage.filters, skimage.transform, skimage.color, skimage.feature, skimage.measure, skimage.morphology
import subprocess
import sys
import pickle
import json

figure_output_dir = os.path.abspath(os.path.join(os.getcwd(), '..', 'figures'))
figure_output_dir = Path(figure_output_dir)
data_dir = os.path.abspath(os.path.join(os.getcwd(), '..', 'data'))
data_dir = Path(data_dir)
print(figure_output_dir, data_dir)
```

Show manually selected point correspondences

```
In []: my_img1 = skimage.io.imread(figure_output_dir/'my_img3.jpg')
with open(data_dir/'my_img3.json') as infile:
    my_img1_kp = json.load(infile)
plt.imshow(my_img1)
for ind, (x, y) in enumerate(my_img1_kp):
    plt.plot(x, y, 'o', color='red')
    plt.text(x - 25, y + 15, str(ind + 1), ha='right', color='red')

save_jpeg_svg(figure_output_dir/'my_img_points.svg', bbox_inches='tight')
```

```
In []: my_img2 = skimage.io.imread(figure_output_dir/'my_img4.jpg')
with open(data_dir/'my_img4.json') as infile:
    my_img2_kp = json.load(infile)
plt.imshow(my_img2)
for ind, (x, y) in enumerate(my_img2_kp):
    plt.plot(x, y, 'o', color='red')
    plt.text(x - 25, y + 15, str(ind + 1), ha='right', color='red')
```

```
In []: my_img1 = skimage.io.imread(figure_output_dir/'my_img3.jpg')
with open(data_dir/'my_img3.json') as infile:
    my_img1_kp = np.asarray(json.load(infile))
my_img2 = skimage.io.imread(figure_output_dir/'my_img4.jpg')
with open(data_dir/'my_img4.json') as infile:
    my_img2_kp = np.asarray(json.load(infile))

fig, ax = plt.subplots(1,1)
skimage.feature.plot_matches(
    ax,
    my_img1,
    my_img2,
    np.flip(my_img1_kp, axis=1),
    np.flip(my_img2_kp, axis=1),
    np.stack([np.arange(my_img1_kp.shape[0]) * 2, axis=1]
)

save_jpeg_svg(figure_output_dir/'my_img_correspondence.svg', bbox_inches='tight')
```

```
In []: # export the key points for LoopAndZhang

manual_kp_strs = []

for kp in np.round(my_img1_kp).astype(np.int32):
    manual_kp_strs.append(f'origin.push_back(Point({kp[0]},{kp[1]}));')
for kp in np.round(my_img2_kp).astype(np.int32):
    manual_kp_strs.append(f'destination.push_back(Point({kp[0]},{kp[1]}));')

print('\n'.join(manual_kp_strs))
```

# Estimating the fundamental matrix

In `[]`:

```
from scipy.optimize import least_squares
from scipy.spatial.transform import Rotation

# convert to homogeneous coordinates
def p_to_hc(points):
    points = np.asarray(points)
    assert points.ndim == 2
    assert points.shape[1] == 2
    ones = np.ones((points.shape[0], 1), points.dtype)
    return np.concatenate([points, ones], axis=1)

# convert to physical coordinates
def p_to_phy(points):
    points = np.asarray(points)
    assert points.ndim == 2
    assert points.shape[1] == 3
    points = points.astype(np.float64)
    points = points / points[:, -1, np.newaxis]
    assert np.max(np.abs(points[:, -1] - 1)) < 1.0e-6
    return points[:, :-1]

class EstimateF:

    def __init__(self):
        self.f = []

        for i in range(3):
            for j in range(3):
                sym = sp.symbols('f_{%d%d}' % (i + 1, j + 1))
                self.f.append(sym)

        self.F = sp.Matrix(np.asarray(self.f, np.object_).reshape(3,3))
        self.f = sp.Matrix(self.f)

        self.sx, self.sy, self.sxp, self.syp = sp.symbols("x y x' y'")
        self.x = sp.Matrix([self.sx, self.sy, 1])
        self.xp = sp.Matrix([self.sxp, self.syp, 1])

        self.eqn = (self.xp.T * self.F * self.x)[0]
        self.eqn = sp.expand(self.eqn)

        # extract the coefficients for each f (estimation from correspondences)
        self.f_coef = []
        for i in range(len(self.f)):
            self.f_coef.append(self.eqn.coeff(self.f[i]))

        self.config = dict()
        self.state = None

    @staticmethod
    def cross_matrix(v):
        assert v.ndim == 1
        assert v.size == 3
        return np.asarray([
            [0, -v[2], v[1]],
            [v[2], 0, -v[0]],
            [-v[1], v[0], 0]
        ])

    def estimate_F_linear(self):
        kp1 = self.state['kp1']
        kp2 = self.state['kp2']

        assert kp1.shape == kp2.shape
        assert kp1.ndim == 2
        assert kp1.shape[1] == 2

        eqns = []

        # create a substitution dict
        for i in range(kp1.shape[0]):
            p1 = kp1[i]
            p2 = kp2[i]
            sub = [(self.sx, p1[0]),
                    (self.sy, p1[1]),
                    (self.sxp, p2[0]),
                    (self.syp, p2[1])]
            f_coef = [x.subs(sub) for x in self.f_coef]
            f_coef = np.asarray(f_coef, np.float64)
            eqns.append(f_coef)

        eqns = np.asarray(eqns)

        # solve the linear system
        u, d, vh = np.linalg.svd(eqns)
        f_sol = vh[-1, :] # use the last column of the V matrix as the solution

        f_sol = f_sol.reshape((3, 3))
        # condition the result
        u, d, vh = np.linalg.svd(f_sol)
        d[2] = 0
        self.state['linear_F'] = u @ np.diag(d) @ vh

    # return left and right epipoles
    def estimate_epipoles(self, F_name):
        F = self.state[F_name]
        u, d, vh = np.linalg.svd(F)
        self.state[F_name+'_e1'] = vh[-1, :]
        u, d, vh = np.linalg.svd(F.T)
        self.state[F_name+'_e2'] = vh[-1, :]

    def build_canonical_P(self, F_name):
        # P1: left camera
        P1 = np.eye(3, 4, dtype=np.float64)

        # P2: right camera
        F = self.state[F_name]
        e2 = self.state[F_name + '_e2']
        s = self.cross_matrix(e2)
```

```

P2 = np.concatenate(
    [s @ F, e2[:, np.newaxis]], axis=1
)

# check rank of right matrix
u, d, vh = np.linalg.svd(P2)
if np.count_nonzero(np.abs(d) > 1.0e-6) < 3:
    raise ArithmeticError(f'P2 is expected to have rank 3; the singular values are {repr(d)}')

self.state[F_name + '_P1'] = P1
self.state[F_name + '_P2'] = P2

@staticmethod
def find_world_point(P1, P2, x1, x2):
    assert x1.ndim == 1 and x1.size == 3
    assert x2.ndim == 1 and x2.size == 3
    # build A
    A = np.asarray([
        x1[0] * P1[2, :] - P1[0, :],
        x1[1] * P1[2, :] - P1[1, :],
        x2[0] * P2[2, :] - P2[0, :],
        x2[1] * P2[2, :] - P2[1, :]
    ])

    assert A.shape == (4,4)

    u, d, vh = np.linalg.svd(A)
    return vh[-1, :] # use the last column of the V matrix as the solution

def refine_P2(self, F_name):
    P1 = self.state[F_name + '_P1']
    P2 = self.state[F_name + '_P2']
    kp1 = self.state['kp1']
    kp2 = self.state['kp2']
    kp1_hc = p_to_hc(kp1)
    kp2_hc = p_to_hc(kp2)

    def cost_func(P2_flat):
        P2 = P2_flat.reshape((3, 4))

        # find world point
        wps = []
        for i in range(kp1_hc.shape[0]):
            wps.append(self.find_world_point(P1, P2, kp1_hc[i], kp2_hc[i]))

        wps = np.asarray(wps)

        # back project them using P1, P2
        kp1_bp = (P1 @ wps.T).transpose()
        kp2_bp = (P2 @ wps.T).transpose()

        kp1_phy = p_to_phy(kp1_bp)
        kp2_phy = p_to_phy(kp2_bp)

        # construct the cost
        kp1_d = kp1_phy - kp1
        kp2_d = kp2_phy - kp2

        kp1_c = np.linalg.norm(kp1_d, axis=1)
        kp2_c = np.linalg.norm(kp2_d, axis=1)
        assert kp1_c.shape[0] == kp2_c.shape[0] == kp1_d.shape[0]

        return np.stack([kp1_c, kp2_c], axis=1).flatten()

    P2_opt_result = least_squares(cost_func, P2.flatten(), method='lm')
    self.state[F_name + '_P2_refine_opt'] = P2_opt_result
    P2_refined = P2_opt_result.x.reshape((3,4))
    # normalize P2 (the last column needs to have norm 1)
    P2_refined /= np.linalg.norm(P2_refined[:, -1])
    self.state[F_name + '_P2_refined'] = P2_refined

def refine_F(self, new_F_name, P2_name, P1_name='linear_F_P1'):
    P1 = self.state[P1_name]
    P2 = self.state[P2_name]
    assert abs(1 - np.linalg.norm(P2[:, -1])) < 1.0e-8

    # get the epipole
    e = P2[:, -1]
    new_F = self.cross_matrix(e) @ P2 @ np.linalg.pinv(P1)

    self.state[new_F_name] = new_F

def compute_right_H(self, P2):
    # get epipole
    e = P2[:, -1]
    # compute angle
    ang = np.arctan2(e[1], e[0])
    # apply reverse rotation
    R = Rotation.from_rotvec(-ang * np.asarray([0,0,1])).as_matrix()

    e = R @ e
    e_phy = e / e[2]
    assert np.abs(e_phy[1]) < 1.0e-8
    f = e_phy[0]

    T = np.eye(3)
    T[0, 2] = -self.state['img_shape'][1] / 2
    T[1, 2] = -self.state['img_shape'][0] / 2

    G = np.eye(3)
    G[2,0] = -1/f

    return G @ R @ T

def compute_left_H(self, P1, P2, right_H):
    M = P2 @ np.linalg.pinv(P1)
    H0 = right_H @ M

    kp1 = self.state['kp1']
    kp2 = self.state['kp2']
    kp1_hc = p_to_hc(kp1)
    kp2_hc = p_to_hc(kp2)

    kp1_tf = (H0 @ kp1_hc.T).transpose()

```

```

kp2_tf = (right_H @ kp2_hc.T).transpose()
kp1_tf_phy = p_to_phy(kp1_tf)
kp2_tf_phy = p_to_phy(kp2_tf)

# add 1 to last col
kp1_tf_phy_1 = p_to_hc(kp1_tf_phy)

opt_result = np.linalg.lstsq(kp1_tf_phy_1, kp2_tf_phy[:, 0])
# # print(opt_result)

# def cost_func(args):
#     a,b,c = args
#     return a * kp1_tf_phy[:, 0] + b * kp1_tf_phy[:, 1] + c - kp2_tf_phy[:, 0]

# opt_result = least_squares(cost_func, (1, 0, 0), verbose=2)
# # print(opt_result)

Ha = np.eye(3)
Ha[0, :] = opt_result[0]

return Ha @ H0, opt_result

# kps are stored in (x, y) format
def run(self, kp1, kp2, img_shape):
    self.state = dict(
        kp1=kp1,
        kp2=kp2,
        img_shape=img_shape
    )

    center_tf = np.asarray([img_shape[1], img_shape[0]], np.float64) / 2
    self.state['center_tf'] = center_tf
    # translate kp1, kp2 to the center
    self.state['kp1'] = self.state['kp1'] - center_tf
    self.state['kp2'] = self.state['kp2'] - center_tf

    self.estimate_F_linear()
    self.estimate_epipoles('linear_F')
    self.build_canonical_P('linear_F')
    self.refine_P2('linear_F')

    # build a newer F
    self.refine_F('refined_F', 'linear_F_P2_refined')
    self.state['right_H'] = self.compute_right_H(self.state['linear_F_P2_refined'])
    left_H, left_H_opt = self.compute_left_H(self.state['linear_F_P1'], self.state['linear_F_P2_refined'], self.state['right_H'])
    self.state['left_H'] = left_H
    self.state['left_H_opt'] = left_H_opt

    return self.state

# apply transformation H to the input image
def transform_one(self, img, H, r_range, c_range):

    rr, cc = np.meshgrid(r_range, c_range, indexing='ij')
    coords = np.stack([cc, rr], axis=2) # use x,y coords

    coords_flat = coords.reshape((-1,2))
    coords_hc = p_to_hc(coords_flat)
    coords_tf = (np.linalg.inv(H) @ coords_hc.T).transpose()
    coords_tf_phy = p_to_phy(coords_tf)
    coords_tf_phy += self.state['center_tf']
    coords_tf_phy = np.flip(coords_tf_phy, axis=1) # to r,c coords
    coords_input = coords_tf_phy.reshape(coords.shape)
    coords_input = np.transpose(coords_input, (2,0,1))

    img = skimage.img_as_float32(img)
    fill_colors = np.asarray([255, 254, 200]) / 255.0

    assert img.ndim == 3
    channels = []
    for i in range(img.shape[2]):
        channels.append(skimage.transform.warp(img[:, :, i], coords_input, cval=fill_colors[i]))

    return np.stack(channels, axis=2)

def transform(self, img_left, img_right, r_range=None, c_range=None):

    assert img_left.shape == img_right.shape

    if r_range is None:
        r_range = np.arange(img_left.shape[0])

    if c_range is None:
        c_range = np.arange(img_left.shape[1])

    new_img_l = self.transform_one(img_left, self.state['left_H'], r_range, c_range)
    new_img_r = self.transform_one(img_right, self.state['right_H'], r_range, c_range)

    # also transform the key points
    def transform_kp(kp, H):
        kp_hc = p_to_hc(kp)
        kp_tf = (H @ kp_hc.T).transpose()
        kp_tf_phy = p_to_phy(kp_tf)
        kp_tf_phy = np.flip(kp_tf_phy, axis=1) # r, c
        # convert to image coordinates
        kp_tf_phy = kp_tf_phy - np.asarray([r_range[0], c_range[0]])
        kp_tf_phy = kp_tf_phy / np.asarray([r_range[1] - r_range[0], c_range[1] - c_range[0]])
        return kp_tf_phy

    kp1 = transform_kp(self.state['kp1'], self.state['left_H'])
    kp2 = transform_kp(self.state['kp2'], self.state['right_H'])

    return new_img_l, new_img_r, kp1, kp2

```

```
In □: print(est_f.state['left_H']/est_f.state['left_H'])[2,2])
```

```
In □: print(est_f.state['right_H']/est_f.state['right_H'])[2,2])
```

```
In □: est_f = EstimateF()
state = est_f.run(my_img1_kp, my_img2_kp, my_img1.shape)
print(state.keys())
img_l, img_r, kp1, kp2 = est_f.transform(my_img1, my_img2,
r_range=np.linspace(-2500, 2500, 2000),
c_range=np.linspace(-4500, 500, 2000))
```

```
# m = np.eye(3)
# m[1,1] = 0.5
# plt.imshow(est_f.transform_one(my_img1, m, np.arange(my_img1.shape[0]), np.arange(my_img1.shape[1])))
```

In :

```
# fig, axs = plt.subplots(1,2)
# axs[0].imshow(img_l)
# axs[1].imshow(img_r)
fig, ax = plt.subplots(1,1)
skimage.feature.plot_matches(
    ax,
    img_l,
    img_r,
    kp1, kp2,
    np.stack([np.arange(kp1.shape[0])*2, axis=1)
])
save_jpeg_svg(f'{figure_output_dir}/img_rectification.svg', bbox_inches='tight')
```

## Interest point extraction

In :

```
import joblib
from scipy.spatial.distance import cdist

class InterestPoint:

    def __init__(self):
        self.state = None

    def canny(self):
        low_thres = 0.2
        high_thres = 0.4
        img_l = self.state['img_l']
        img_r = self.state['img_r']
        assert img_l.shape == img_r.shape

        self.state['img_l_gray'] = skimage.color.rgb2gray(img_l)
        self.state['img_r_gray'] = skimage.color.rgb2gray(img_r)
        self.state['img_l_canny'] = skimage.feature.canny(self.state['img_l_gray'], low_threshold=low_thres, high_threshold=high_thres)
        self.state['img_r_canny'] = skimage.feature.canny(self.state['img_r_gray'], low_threshold=low_thres, high_threshold=high_thres)

    def canny_clear_edge(img, r=3):
        img = img.copy()
        img[:,r,:] = False
        img[-r:,:] = False
        img[:, -r] = False
        img[:, -r:] = False
        return img

    self.state['img_l_canny'] = canny_clear_edge(self.state['img_l_canny'])
    self.state['img_r_canny'] = canny_clear_edge(self.state['img_r_canny'])

    def ssd(self):
        img_l = self.state['img_l']
        img_r = self.state['img_r']

        l_canny = self.state['img_l_canny']
        r_canny = self.state['img_r_canny']

        ssd_size = 5
        ssd_size_2 = ssd_size // 2
        ssd_norm_factor = ssd_size ** 2

        # first of all, pad the images
        img_l_padded = np.pad(img_l, [(ssd_size_2, ssd_size_2), (ssd_size_2, ssd_size_2), (0,0)], mode='reflect')
        img_r_padded = np.pad(img_r, [(ssd_size_2, ssd_size_2), (ssd_size_2, ssd_size_2), (0,0)], mode='reflect')

        kp1 = []
        kp2 = []
        ssd = []

        for i in range(img_l.shape[0]):
            # extract the valid regions
            l_regions = []
            r_regions = []

            # use canny edge result to mask out unwanted pixels
            l_pos = np.where(l_canny[i])[0]
            r_pos = np.where(r_canny[i])[0]

            ind1 = slice(i, i + 2 * ssd_size_2 + 1)
            for j in l_pos:
                ind2 = slice(j, j + 2 * ssd_size_2 + 1)
                l_regions.append(img_l_padded[ind1, ind2].flatten())

            for j in r_pos:
                ind2 = slice(j, j + 2 * ssd_size_2 + 1)
                r_regions.append(img_r_padded[ind1, ind2].flatten())

            # compute pairwise SSD
            if len(l_regions) > 0 and len(r_regions) > 0:
                l_regions = np.asarray(l_regions)
                r_regions = np.asarray(r_regions)
                #print(i, l_regions.shape, r_regions.shape)
                assert l_regions.shape[1] == ssd_norm_factor * img_l.shape[2]
                lr_ssd = cdist(l_regions, r_regions) / ssd_norm_factor

                while True:
                    min_loc = np.unravel_index(np.argmin(lr_ssd), lr_ssd.shape)
                    min_val = lr_ssd[min_loc[0], min_loc[1]]
                    if min_val == np.inf:
                        break

                    # put this in as a match
                    kp1.append((i, l_pos[min_loc[0]]))
                    kp2.append((i, r_pos[min_loc[1]]))
                    ssd.append(min_val)

            # mask out values
            lr_ssd[min_loc[0], :] = np.inf
            lr_ssd[:, min_loc[1]] = np.inf
```

```
self.state['kp1'] = np.asarray(kp1) # r,c
self.state['kp2'] = np.asarray(kp2) # r,c
self.state['kp_ssd'] = np.asarray(ssd)

def run(self, img_l, img_r):
    self.state = dict(
        img_l=img_l,
        img_r=img_r
    )

    self.canny()
    self.ssd()
```

```
In □: ip = InterestPoint()
ip.run(img_l, img_r)
```

```
In □: fig, axs = plt.subplots(1,2)
axs[0].imshow(ip.state['img_l_canny'], cmap='plasma')
axs[1].imshow(ip.state['img_r_canny'], cmap='plasma')
save_jpeg_svg(f'{figure_output_dir}/canny.svg', bbox_inches='tight')
```

```
In □: plt.hist(ip.state['kp_ssd'], bins=50)
plt.xlabel('SSD')
save_jpeg_svg(f'{figure_output_dir}/ssd_hist.svg', bbox_inches='tight')
```

```
In □: matching_kp_mask = ip.state['kp_ssd'] < 0.02
matching_kp1 = ip.state['kp1'][matching_kp_mask]
matching_kp2 = ip.state['kp2'][matching_kp_mask]

fig, ax = plt.subplots(1,1)
skimage.feature.plot_matches(
    ax,
    ip.state['img_l'],
    ip.state['img_r'],
    matching_kp1, matching_kp2 ,
    np.stack([np.arange(matching_kp1.shape[0])*2, axis=1][::200],
    only_matches=True
)
save_jpeg_svg(f'{figure_output_dir}/ip_matching.svg', bbox_inches='tight')

print('kp shape:', kp1.shape)

# # also, plot the set of points
# draw_r = 2
# def get_kp_img(kp, img):
#     ret = np.zeros((img.shape[0], img.shape[1]), np.bool_)
#     for i in range(kp.shape[0]):
#         rr, cc = skimage.draw.disk((kp[i,0], kp[i,1]), draw_r, shape=ret.shape)
#         ret[rr,cc] = True
#     return ret

# fig, axs = plt.subplots(1,2)
# axs[0].imshow(get_kp_img(kp1, ip.state['img_l']))
# axs[1].imshow(get_kp_img(kp2, ip.state['img_l']))
```

```
In □: from mpl_toolkits.mplot3d import proj3d
from matplotlib.patches import ConnectionPatch

lines_to_show = [
    (1,2),
    (1,4),
    (3,4),
    (4,5),
    (1,6),
    (2,3),
    (5,6),
    (8,9),
    (8,11),
    (7,8),
    (7,13),
    (9,10),
    (10,11),
    (10,12),
    (11,13),
    (12,13)
]

colors = {
    (1,4): 'c1',
    (4,5): 'c1',
    (5,6): 'c1',
    (1,6): 'c1',
    (8,11): 'c2',
    (11,13): 'c2',
    (7,13): 'c2',
    (7,8): 'c2',
    (8,9): 'c3',
    (10,11): 'c3',
    (12,13): 'c3',
    (9,10): 'c4',
    (10,12): 'c4',
    (1,2): 'c5',
    (3,4): 'c5',
    (2,3): 'c5'
}

lines_to_show = np.asarray(lines_to_show) - (1,1)

subsample_fac = 100
matching_kp1 = ip.state['kp1'][:subsample_fac]
matching_kp2 = ip.state['kp2'][:subsample_fac]

# convert the location of kps back to the image plane
def convert_kp_image_plane(kp, H):
    kp_hc = p_to_hc(kp)
    kp_tf = (np.linalg.inv(H) @ kp_hc.T).transpose()
    return kp_tf
```

```
m_kp1_img_plane = convert_kp_image_plane(matching_kp1, est_f.state['left_H'])
m_kp2_img_plane = convert_kp_image_plane(matching_kp2, est_f.state['right_H'])
wps = []
for i in range(m_kp1_img_plane.shape[0]):
    wp = est_f.find_world_point(
        est_f.state['linear_F_P1'],
        est_f.state['linear_F_P2_refined'],
        m_kp1_img_plane[i],
        m_kp2_img_plane[i]
    )
    wps.append(wp)

wps = np.asarray(wps)
wps = wps / wps[:, -1, np.newaxis]

fig = plt.figure(figsize=(6,8))

ax = fig.add_subplot(311)
aximg1 = ax
ax.imshow(img_r)
for i1, i2 in lines_to_show:
    p1 = kp2[i1]
    p2 = kp2[i2]
    c = colors[(i1+1,i2+1)].upper()
    ax.plot([p1[1], p2[1]], [p1[0], p2[0]], color=c)

ax = fig.add_subplot(312, projection='3d')
ax3d = ax
ax.scatter(wps[:, 0], wps[:, 1], wps[:, 2], marker='o', s=1)

m_kp1_img_plane = convert_kp_image_plane(kp1, est_f.state['left_H'])
m_kp2_img_plane = convert_kp_image_plane(kp2, est_f.state['right_H'])

world_points = dict()

# plot lines
for i1, i2 in lines_to_show:
    p11, p12 = m_kp1_img_plane[i1], m_kp1_img_plane[i2]
    p21, p22 = m_kp2_img_plane[i1], m_kp2_img_plane[i2]
    wp1 = est_f.find_world_point(
        est_f.state['linear_F_P1'],
        est_f.state['linear_F_P2_refined'],
        p11,
        p21
    )
    wp2 = est_f.find_world_point(
        est_f.state['linear_F_P1'],
        est_f.state['linear_F_P2_refined'],
        p12,
        p22
    )

    wp1 = wp1 / wp1[-1]
    wp2 = wp2 / wp2[-1]

    world_points[i1] = wp1
    world_points[i2] = wp2

    c = colors[(i1+1,i2+1)].upper()

    ax.plot(
        [wp1[0],wp2[0]],
        [wp1[1],wp2[1]],
        zs=[wp1[2],wp2[2]],
        color=c
    )
    #ax.plot([p1[1], p2[1]], [p1[0], p2[0]], color=c)

ax.set_xlim(63.450801013861366, 77.69490092256974)
ax.set_ylim(25.885132020740006, 51.55057465037077)
ax.set_zlim(-0.0002303183327774123, -0.00014524106430428713)

ax = fig.add_subplot(313)
ax.imshow(img_l)
aximg2 = ax
for i1, i2 in lines_to_show:
    p1 = kp1[i1]
    p2 = kp1[i2]
    c = colors[(i1+1,i2+1)].upper()
    ax.plot([p1[1], p2[1]], [p1[0], p2[0]], color=c)

added_artist = []
for i, azim in enumerate(np.linspace(0, 360, 10)):

    for artist in added_artist:
        artist.remove()
    added_artist.clear()

    ax3d.view_init(elev=10, azim=azim)
    # now, connect lines
    # transform world points to 2d
    world_points_2d = dict()
    for key, val in world_points.items():
        world_points_2d[key] = proj3d.proj_transform(*val[:3], ax3d.get_proj())[:2]

    for key, val in world_points_2d.items():
        xyA = np.flip(kp1[key])
        con = ConnectionPatch(xyA=xyA, xyB=val, coordsA="data", coordsB="data",
                               axesA=aximg2, axesB=ax3d, color="red", linestyle='--', linewidth=0.5)
        added_artist.append(ax.add_artist(con))
        xyA = np.flip(kp2[key])
        con = ConnectionPatch(xyA=xyA, xyB=val, coordsA="data", coordsB="data",
                               axesA=aximg1, axesB=ax3d, color="red", linestyle='--', linewidth=0.5)
        added_artist.append(ax.add_artist(con))

    save_jpeg_svg(f'{figure_output_dir}/recon_w_points/{i}.svg', bbox_inches='tight')
```

In □:

```
from mpl_toolkits.mplot3d import proj3d
from matplotlib.patches import ConnectionPatch

lines_to_show = [
    (1,2),
    (1,4),
    (3,4),
    (4,5),
```

```

(1,6),
(2,3),
(5,6),
(8,9),
(8,11),
(7,8),
(7,13),
(9,10),
(10,11),
(10,12),
(11,13),
(12,13)
]

colors = {
    (1,4): 'c1',
    (4,5): 'c1',
    (5,6): 'c1',
    (1,6): 'c1',
    (8,11): 'c2',
    (11,13): 'c2',
    (7,13): 'c2',
    (7,8): 'c2',
    (8,9): 'c3',
    (10,11): 'c3',
    (12,13): 'c3',
    (9,10): 'c4',
    (10,12): 'c4',
    (1,2): 'c5',
    (3,4): 'c5',
    (2,3): 'c5'
}

lines_to_show = np.asarray(lines_to_show) - (1,1)

subsample_fac = 50
matching_kp1 = ip.state['kp1'][:subsample_fac]
matching_kp2 = ip.state['kp2'][:subsample_fac]

# convert the location of kps back to the image plane
def convert_kp_image_plane(kp, H):
    kp_hc = p_to_hc(kp)
    kp_tf = (np.linalg.inv(H) @ kp_hc.T).transpose()
    return kp_tf

m_kp1_img_plane = convert_kp_image_plane(matching_kp1, est_f.state['left_H'])
m_kp2_img_plane = convert_kp_image_plane(matching_kp2, est_f.state['right_H'])

fig = plt.figure(figsize=(6,8))
ax = fig.add_subplot(211, projection='3d')
ax3d = ax
# ax.scatter(wps[:, 0], wps[:, 1], wps[:, 2], marker='o', s=1)

m_kp1_img_plane = convert_kp_image_plane(kp1, est_f.state['left_H'])
m_kp2_img_plane = convert_kp_image_plane(kp2, est_f.state['right_H'])

world_points = dict()

# plot lines
for i1, i2 in lines_to_show:
    p11, p12 = m_kp1_img_plane[i1], m_kp1_img_plane[i2]
    p21, p22 = m_kp2_img_plane[i1], m_kp2_img_plane[i2]
    wp1 = est_f.find_world_point(
        est_f.state['linear_F_P1'],
        est_f.state['linear_F_P2_refined'],
        p11,
        p21
    )
    wp2 = est_f.find_world_point(
        est_f.state['linear_F_P1'],
        est_f.state['linear_F_P2_refined'],
        p12,
        p22
    )

    wp1 = wp1 / wp1[-1]
    wp2 = wp2 / wp2[-1]

    world_points[i1] = wp1
    world_points[i2] = wp2

    c = colors[(i1+1,i2+1)].upper()

    ax.plot(
        [wp1[0],wp2[0]],
        [wp1[1],wp2[1]],
        zs=[wp1[2],wp2[2]],
        color=c
    )
    #ax.plot([p1[1], p2[1]], [p1[0], p2[0]], color=c)

print(ax.get_xlim())
print(ax.get_ylim())
print(ax.get_zlim())

ax = fig.add_subplot(212)
ax.imshow(img_l)
for i1, i2 in lines_to_show:
    p1 = kp1[i1]
    p2 = kp1[i2]
    c = colors[(i1+1,i2+1)].upper()
    ax.plot([p1[1], p2[1]], [p1[0], p2[0]], color=c)

added_artist = []
for i, azim in enumerate(np.linspace(0, 360, 10)):

    for artist in added_artist:
        artist.remove()
    added_artist.clear()

    ax3d.view_init(elev=10, azim=azim)
    # now, connect lines
    # transform world points to 2d
    world_points_2d = dict()

```

```
for key, val in world_points.items():
    world_points_2d[key] = proj3d.proj_transform(*val[:3], ax3d.get_proj())[:2]

for key, val in world_points_2d.items():
    xyA = np.flip(kp1[key])
    con = ConnectionPatch(xyA=xyA, xyB=val, coordsA="data", coordsB="data",
                          axesA=ax, axesB=ax3d, color="red", linestyle='--', linewidth=0.5)
    added_artist.append(ax.add_artist(con))

save_jpeg_svg(figure_output_dir/'recon_wo_points'/f'{i}wo.svg', bbox_inches='tight')
```

Process the results from LoopAndZhang

```
In □:
img1 = skimage.io.imread(data_dir/'LoopAndZhang'/'build'/'lnz_img_1_dst.png')
img1 = skimage.transform.rescale(img1, 0.3, multichannel=True)
skimage.io.imsave(figure_output_dir/'lnz_img_1_dst.jpg', img1)
```

```
In □:
img2 = skimage.io.imread(data_dir/'LoopAndZhang'/'build'/'lnz_img_2_dst.png')
# rescale to the same height
scale = img1.shape[0]/img2.shape[0]
img2 = skimage.transform.rescale(img2, scale, multichannel=True)
skimage.io.imsave(figure_output_dir/'lnz_img_2_dst.jpg', img2)
```

Census Transform

```
In □:
class CensusTransform:

    def __init__(self,
                 max_disparity=10,
                 window_size=5
    ):
        self.max_disparity=max_disparity
        self.window_size = window_size
        assert self.window_size % 2 == 1
        self.window_size_2 = self.window_size // 2

    def pad_img(self, img):
        assert img.ndim == 2
        return np.pad(img, [(self.window_size_2, self.window_size_2)] * 2, mode='reflect')

    @staticmethod
    def transform_window(win):
        h = win.shape[0] // 2
        return win > win[h, h]

    def estimate_row_disparity(self, img_l_pad, img_r_pad, row_ind):
        col_size = img_l_pad.shape[1] - 2 * self.window_size_2
        disp_candidate = np.arange(self.max_disparity)

        ret = []
        for j in range(col_size):
            # compute valid disparity values
            right_col_loc = j - disp_candidate
            good_right_col_loc = np.bitwise_and(right_col_loc >= 0, right_col_loc < col_size)
            good_disp_candidate = disp_candidate[good_right_col_loc]

            win_l = img_l_pad[row_ind : row_ind + self.window_size, j : j + self.window_size]
            assert win_l.size == self.window_size ** 2
            assert win_l.shape[0] == win_l.shape[1]
            win_l_bin = self.transform_window(win_l)

            disp_ret = []

            for disp in good_disp_candidate:
                win_r = img_r_pad[row_ind : row_ind + self.window_size, j - disp : j - disp + self.window_size]
                assert win_l.shape == win_r.shape
                win_r_bin = self.transform_window(win_r)
                disp_ret.append((disp, np.count_nonzero(win_l_bin != win_r_bin)))

            best_disp, _ = min(disp_ret, key=lambda x: x[1])
            ret.append(best_disp)

        return np.asarray(ret)

    def run(self, img_l, img_r):
        assert img_l.shape == img_r.shape
        assert img_l.ndim == 2
        img_l_pad = self.pad_img(img_l)
        img_r_pad = self.pad_img(img_r)

        all_disp = [self.estimate_row_disparity(img_l_pad, img_r_pad, i) for i in range(img_l.shape[0])]
        return np.asarray(all_disp)
```

```
In □:
dm_ground_truth = skimage.io.imread(data_dir/'Task3Images'/'disp2.png')
assert dm_ground_truth.dtype == np.uint8
dm_ground_truth = dm_ground_truth.astype(np.float32)
dm_ground_truth /= 4.0
dm_ground_truth = dm_ground_truth.astype(np.uint8)
print(dm_ground_truth.max())
```

```
In □:
dm_img_l = skimage.io.imread(data_dir/'Task3Images'/'im2.png')
dm_img_r = skimage.io.imread(data_dir/'Task3Images'/'im6.png')
ct = CensusTransform(window_size=7, max_disparity=52)
disp = ct.run(
    skimage.color.rgb2gray(dm_img_l),
    skimage.color.rgb2gray(dm_img_r)
)
non_black_mask = dm_ground_truth > 0
diff = np.abs(dm_ground_truth - disp)
acc = np.count_nonzero(diff[non_black_mask] < 2) / np.count_nonzero(non_black_mask)
print(acc)
plt.imshow(disp, cmap='jet')
plt.colorbar()
```

```
save_jpeg_svg(figure_output_dir/'disp_win7.svg')
plt.close()
# error mask
error_mask = np.zeros_like(non_black_mask)
error_mask[diff < 2] = True
error_mask[np.bitwise_not(non_black_mask)] = False
plt.imshow(error_mask, cmap='gray')
save_jpeg_svg(figure_output_dir/'disp_error_win7.svg')
```

In :

```
dm_img_l = skimage.io.imread(data_dir/'Task3Images/'/'im2.png')
dm_img_r = skimage.io.imread(data_dir/'Task3Images/'/'im6.png')
ct = CensusTransform(window_size=9, max_disparity=52)
disp = ct.run(
    skimage.color.rgb2gray(dm_img_l),
    skimage.color.rgb2gray(dm_img_r)
)
non_black_mask = dm_ground_truth > 0
diff = np.abs(dm_ground_truth - disp)
acc = np.count_nonzero(diff[non_black_mask] < 2) / np.count_nonzero(non_black_mask)
print(acc)
plt.imshow(disp, cmap='jet')
plt.colorbar()
save_jpeg_svg(figure_output_dir/'disp_win9.svg')
plt.close()
# error mask
error_mask = np.zeros_like(non_black_mask)
error_mask[diff < 2] = True
error_mask[np.bitwise_not(non_black_mask)] = False
plt.imshow(error_mask, cmap='gray')
save_jpeg_svg(figure_output_dir/'disp_error_win9.svg')
```

In :

```
plt.imshow(dm_ground_truth, cmap='jet')
plt.colorbar()
save_jpeg_svg(figure_output_dir/'disp_gt.svg', bbox_inches='tight')
```

In :