

Computer Vision Assignment 3 & 4

Adam Temmel (adte1700)

December 6, 2021

Assignment 3

Mark the following points in a graph. Draw the Hough space for the points (using ρ and θ). In the Hough space, identify the crossing of the graphs (or the "best possible solution"). Draw the line corresponding to the best solution in the graph.

I constructed the following plots (using the $\rho = x\cos(\theta) + y\sin(\theta)$ formula):

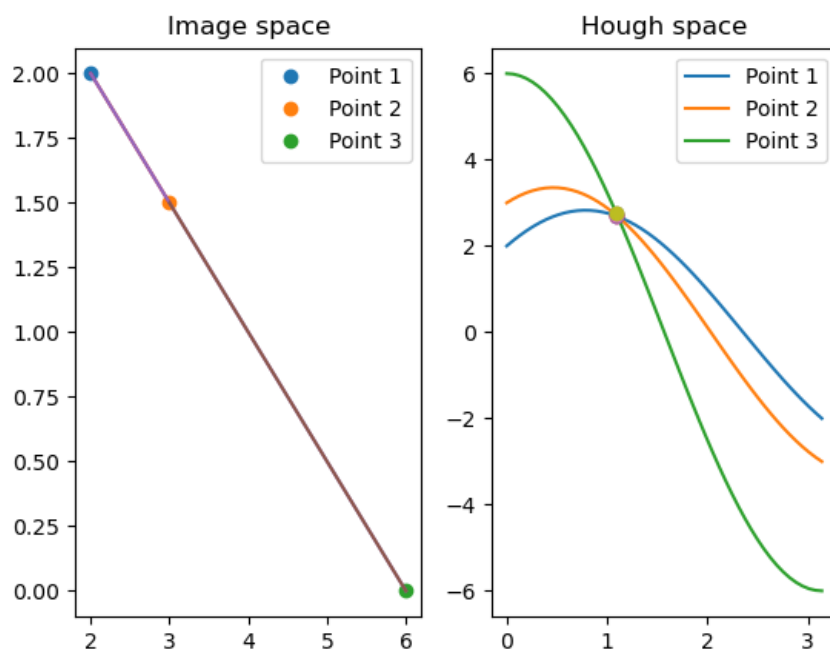


Figure 1: Hough space generated from the points (2, 2), (3, 1.5), (6, 0)

A line has been drawn between the points which crossing each other.

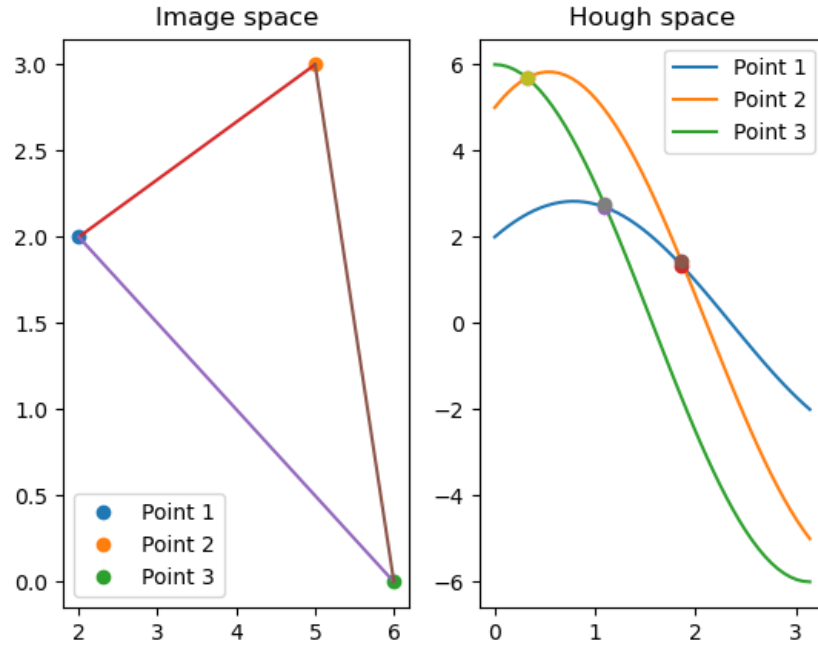


Figure 2: Hough space generated from the points $(2, 2)$, $(5, 3)$, $(6, 0)$

Here, several crossings appeared, but only once per point combination. As such, I plotted all of the crossings, as none of the available crossings seemed better or worse than the others. This might not be the way to go about things when you use this to solve a "real" problem, but I chose to plot all available data for the sake of completeness.

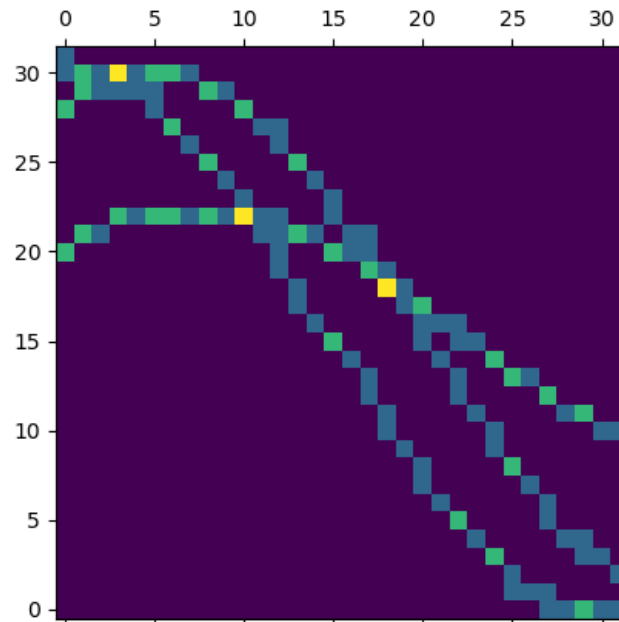


Figure 3: Visualisation of the voting results within the hough space created from the points $(2, 2)$, $(5, 3)$, $(6, 0)$

Find out more about the feature descriptors mentioned in the lecture by looking into the book, online lectures and the Internet. Which descriptor would you use when in a implementation for Augmented Reality? (Hint: There is no correct answer, just good arguments!)

I found a paper which compared some of the feature descriptors mentioned purely in terms of accuracy[1]. It describes SIFT as being very robust, with the drawback of requiring more computational time when compared to other algorithms. SURF is also presented as a robust algorithm, while being slightly faster than SIFT. BRIEF, while not being scale/orientation invariant requires a lot less processing power than SIFT/SURF. Lastly, ORB is described as being around twice as fast as SIFT, with the drawback of not being scale invariant. As augmented reality in many cases involve the need for real-time computing, computation speed may very well be of interest for this particular field. As such, my answer would be to prefer ORB unless the potential accuracy drop between ORB and its competitors is considered to

great, in which SIFT or SURF might be preferable.

Take two pictures with your mobile phone, where the images partly cover the same scene. Make sure the scene is fairly simple, but contains "flat region", "edges" and "corners"

Implement a feature detector and apply it on the images. Choose among Harris operator, $\max(\lambda-)$, harmonic mean, Triggs, etc. Remember to smooth the image. E.g. use a "derivative of Gaussian" filter.

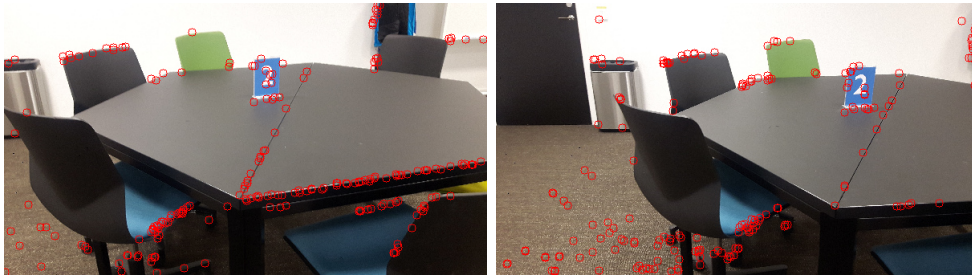


Figure 4: Feature points extracted from running my own feature detector (Harris) on two similar images

```

1  #!/usr/bin/env python
2
3  import numpy as np
4  import cv2
5
6  path = './to_detect_2.jpg' # file to work with
7  img = cv2.imread(path, 0)  # read as greyscale
8
9  # sobel kernels
10 gx_kernel = np.array([
11     [1, 0, -1],
12     [2, 0, -2],
13     [1, 0, -1],
14 ])
15
16 gy_kernel = np.array([
17     [ 1,  2,  1],
18     [ 0,  0,  0],
19     [-1, -2, -1],
20 ])
21
22 # blur
23 blur_img = cv2.GaussianBlur(img, (5,5), 3)
24 # perform sobel and build tensor components
25 gx = cv2.filter2D(blur_img, -1, gx_kernel)
26 gy = cv2.filter2D(blur_img, -1, gy_kernel)
27 gxy = gx * gy
28 gx2 = gx ** 2
29 gy2 = gy ** 2
30
31 # See also:
32 # https://en.wikipedia.org/wiki/Harris_corner_detector
33 k = 0.06
34 h, w = img.shape
35 result = np.array([[0.0] * w for _ in range(h)], float)
36 for y in range(h):
37     for x in range(w):
38         # structure tensor, see end of chapter 'Development'
39         mat = np.array([
40             [gx2[y][x], gxy[y][x]],
41             [gxy[y][x], gy2[y][x]],
42         ], dtype=float)
43         # response calculation, see chapter
44         # 'Response Calculation'
45         tr = mat[0,0] + mat[1,1]
46         det = mat[0,0] * mat[1,1] - mat[1,0]*mat[0,1]
47         result[y][x] = det - (k * (tr * tr))

```

Figure 5: First half of implementation of Harris corner detector

```

1 # mark points of interest
2 color_img = cv2.imread(path)
3 threshold = 24500
4 for y in range(h):
5     for x in range(w):
6         if result[y][x] > threshold:
7             cv2.circle(color_img, (x, y), 5, (0, 0, 255), 1)
8 cv2.imwrite("harris_out.png", color_img)

```

Figure 6: Second half of implementation of Harris corner detector

The Harris corner detector I implemented can be summarized into the following steps:

1. Read a greyscale version of the image you wish to process.
2. Blur the image.
3. Apply a Sobel kernel to the blurred image (for both the X and Y directions).
4. Calculate the structure tensor components.
5. Apply the response calculation for each pixel within the Sobeled image, resulting in a new matrix with the same dimensions as the image.
6. For each pixel in the new image, compare its value to a given threshold, if the value is greater than the threshold, this pixel is regarded as a feature point.

Find an implementation of your favourite feature detector/descriptor (SIFT, ORB, etc.), and apply it on your images. Do you get the same feature points as for your implementation?

Although the differences between my implementation of the Harris feature detector and OpenCV's SIFT feature detector were not as many as originally anticipated, my feature detector still introduced an amount of discrepancies large enough for the SIFT feature detector results to be interpreted as better. In particular, the edges of the table were interpreted as corners in my implementation, whereas SIFT is doing a much better job at *only* identifying corners.

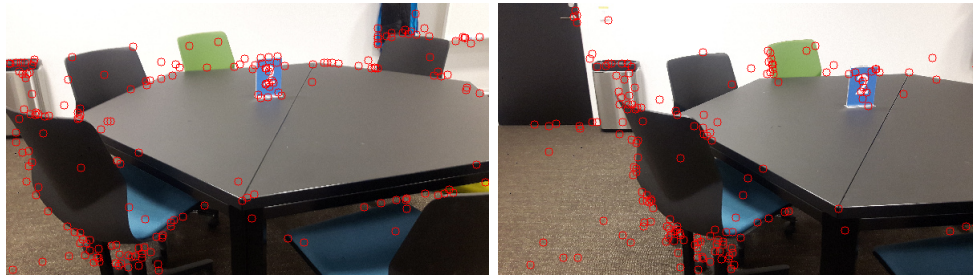


Figure 7: Feature points extracted from running OpenCV's implementation of the SIFT feature detector on two similar images

Assignment 4

A rectangle is defined by its corners $(0,0)$, $(0,3)$, $(5,3)$, and $(5,0)$.

Write the similarity transform matrix for the rectangle to move 3 point horizontally, -2 points vertically, turning 15 degrees clockwise.

Figure 8: First half of calculating the total point transform

```

1      #!/usr/bin/env python
2  import math
3  import matplotlib.pyplot as plt
4
5  corners = [[0, 0, 1], [0, 3, 1], [5, 3, 1], [5, 0, 1]]
6
7  # translation matrix
8  translate = [
9      [1, 0, 3],
10     [0, 1, -2],
11     [0, 0, 1],
12 ]
13 # rotation matrix
14 angle = -15
15 s = math.sin(angle * (math.pi / 180))
16 c = math.cos(angle * (math.pi / 180))
17 rotate = [
18     [ c, -s, 0],
19     [ s,  c, 0],
20     [ 0,  0, 1],
21 ]

```

Figure 9: First half of calculating the total point transform


```

1  def mat_mul_mat(lhs, rhs):
2      m, n, o = len(lhs), len(rhs[0]), len(rhs)
3      res = [[0] * n for _ in range(m)]
4      for i in range(m):
5          for j in range(n):
6              for k in range(o):
7                  res[i][j] += rhs[i][k] * lhs[k][j]
8      return res
9
10 def mat_mul_vec(lhs, rhs):
11     m, n = len(lhs), len(rhs)
12     res = [0] * n
13     for i in range(m):
14         for j in range(n):
15             res[i] += lhs[i][j] * rhs[j]
16     return res
17
18 # total transform
19 transform = mat_mul_mat(translate, rotate)
20
21 # calculate new corners
22 new_corners = [mat_mul_vec(transform, vec) for vec in corners]
23
24 # plot
25 x1 = [corner[0] for corner in corners]
26 y1 = [corner[1] for corner in corners]
27 x2 = [corner[0] for corner in new_corners]
28 y2 = [corner[1] for corner in new_corners]
29 plt.xlim([-10, 10])
30 plt.ylim([-10, 10])
31 plt.plot(x1, y1, 'o')
32 plt.plot(x2, y2, 'o')
33 plt.show()

```

Figure 10: Second half of calculating the total point transform

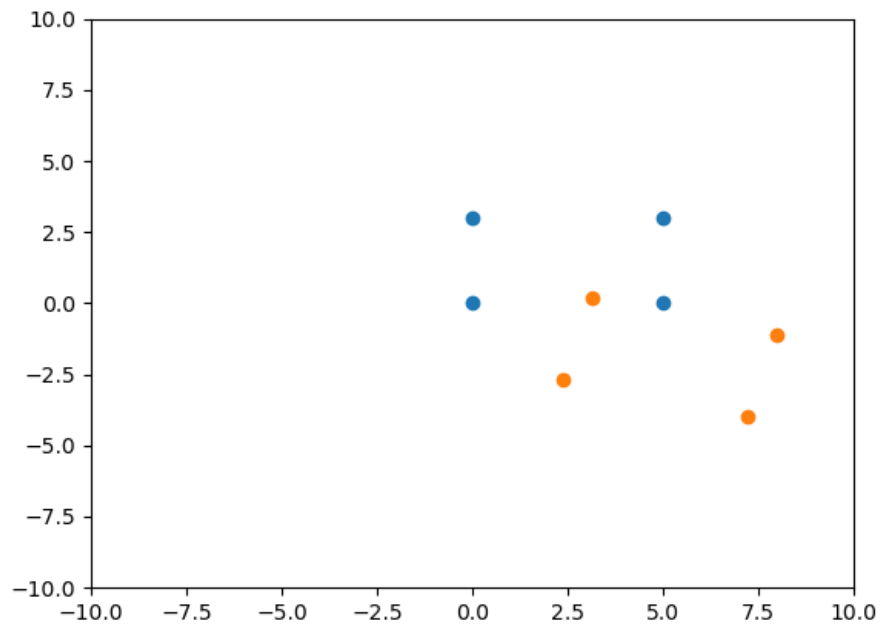


Figure 11: The resulting plot of the code in figure 9 and figure 10.

The rectangle is transformed to have its corners at (1,1), (3,3), (6,3), and (5,2). Compute the transformation matrix. You may use Matlab (or similar) to compute the solution

```

1  #!/usr/bin/env python
2  import numpy as np
3
4  corners = [[0, 0, 1], [0, 3, 1], [5, 3, 1], [5, 0, 1]]
5  corners_new = [[1, 1, 1], [3, 3, 1], [6, 3, 1], [5, 2, 1]]
6
7  def build_homo(p, q):
8      assert(len(p) == len(q))
9      assert(len(p[0]) == len(q[0]))
10     n = len(p)
11     m = len(p[0])
12     res = [[0] * (m + 1) * 2 for _ in range(n * 2)]
13     for i in range(n):
14         v1, v2 = p[i], q[i]
15         for j in range(m):
16             res[i * 2][j] = v1[j]
17             res[i * 2][j + m] = 0
18             res[(i * 2) + 1][j + m] = v1[j]
19             res[(i * 2) + 1][j] = 0
20
21             res[i * 2][m * 2 + 0] = -v2[0] * v1[0]
22             res[i * 2][m * 2 + 1] = -v2[0] * v1[1]
23
24             res[(i * 2) + 1][m * 2 + 0] = -v2[1] * v1[0]
25             res[(i * 2) + 1][m * 2 + 1] = -v2[1] * v1[1]
26
27     return res
28
29 homo = build_homo(corners, corners_new)
30 _, _, v = np.linalg.svd(np.array(homo))
31 h = list(v[-1])
32 h.append(1) # append a single 1 to the bottom right corner
33 h = np.array(h)
34 print(h.reshape(3, 3))

```

Figure 12: Code for calculating the transformation matrix between the corners

```

1 $ ./2dtransformations2.py
2 [[ 0.78820675 -0.32015553  0.02336885]
3  [ 0.40298246 -0.16184404 -0.22825153]
4  [ 0.15866714 -0.09910638  1.          ]]

```

Check Middlebury web site and find out the principles of some of the best performing stereo algorithms. Write a short description of the one you want to use, and motivate your choice

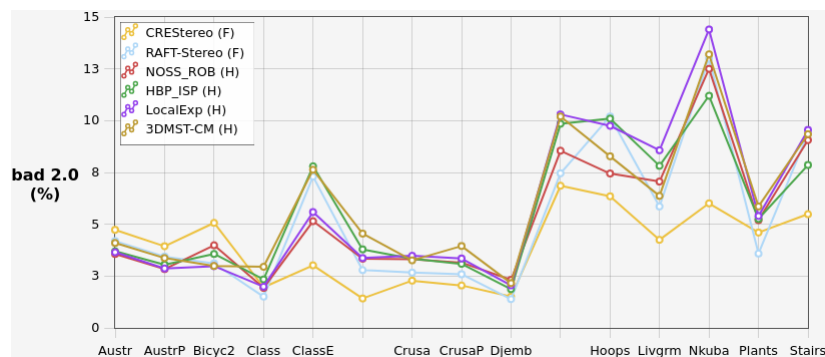


Figure 13: Graph describing the error rate occurred when different stereo algorithms were pitted against eachother.

From the list of stereo algorithms presented over at middlebury, one proved to be performing a little bit better than the others in terms of least error rate. This algorithm was named CREStereo, and was submitted by an anonymous user on the 10th of November, 2021. Although no details regarding the algorithm was presented on the site, it is briefly presented as a "cascaded recurrent network with adaptive correlation" [2]. A cascade-correlation based neural network begins with a network comprised of a very few amount of nodes, instead opting to introduce new nodes to the network iteratively, which in turn entirely avoids the problem of backpropagation.[3]

Take two rectified images from Middlebury's web page (or use your own rectified images.) Write a short program that performs the plane sweep algorithm. Try out two of the metrics given in Ex 12.4

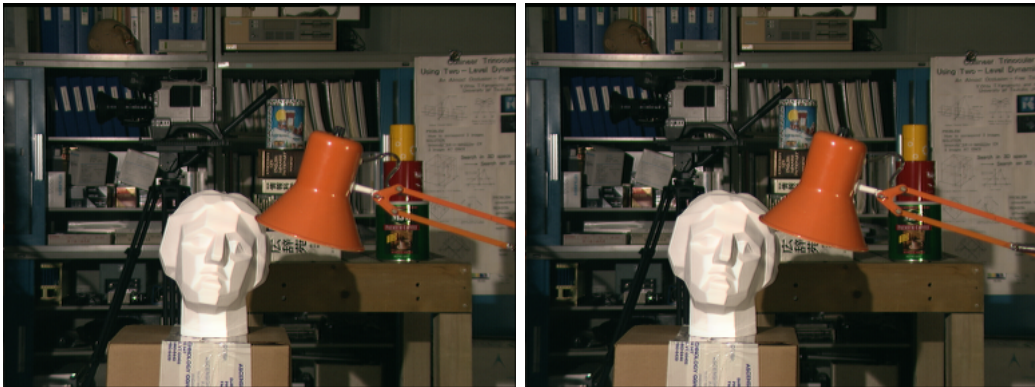


Figure 14: Two rectified images, courtesy of Middlebury's web page

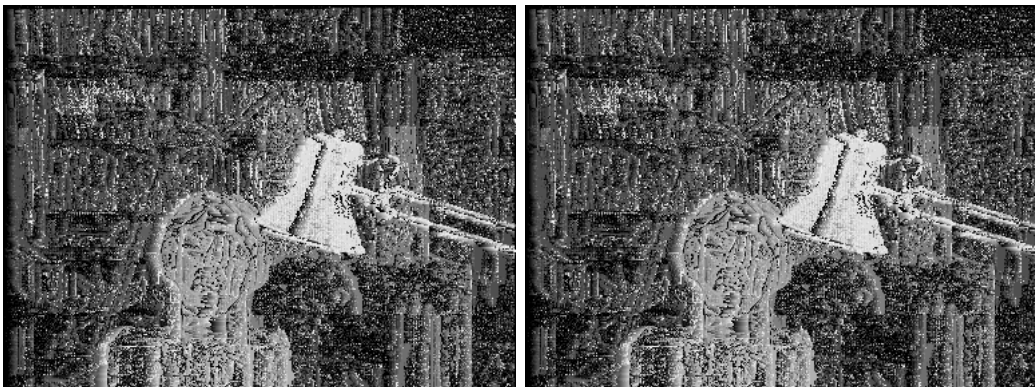


Figure 15: The output of the plane sweep algorithm, using the squared error for the left image and absolute error for the right image.

```

1  #!/usr/bin/env python
2  import numpy as np
3  import cv2
4
5  left_img = cv2.imread('./scene1.row3.col3.ppm', 0)
6  right_img = cv2.imread('./scene1.row3.col4.ppm', 0)
7
8  def plane_sweep(left_img, right_img, eval_method):
9      out_img = np.array([[np.uint8(0)] * len(i) for i in left_img])
10     h_img, w_img = left_img.shape
11     # for each pixel
12     for y_in in range(h_img):
13         for x_in in range(w_img):
14             min_value = np.Infinity
15             suitable_disp = 0
16             max_disparity = 16
17             # walk to the right
18             for disparity in range(min(max_disparity, x_in + 1)):
19                 # evaluate the difference between the
20                 # images at this location
21                 result = eval_method(float(left_img[y_in, x_in])
22                                     - float(right_img[y_in, x_in - disparity]))
23                 # if the difference is the lowest
24                 # difference found so far
25                 if result < min_value:
26                     # save it
27                     min_value = result
28                     suitable_disp = disparity
29                 # write the lowest difference to the result image
30                 out_img[y_in, x_in] = ((suitable_disp)
31                                         / max_disparity) * 255
32     return out_img
33
34 img_out_sq_err = plane_sweep(left_img, right_img, lambda x: x ** 2)
35 img_out_abs_err = plane_sweep(left_img, right_img, lambda x: abs(x))
36
37 cv2.imwrite("out_sq_err.png", img_out_sq_err)
38 cv2.imwrite("out_abs_err.png", img_out_abs_err)

```

Figure 16: Code describing an implementation of the plane-sweep algorithm.

Re-use the images from last week, or take the two new images with your mobile phone; try to keep the camera central point at one position. Identify a number of feature points with your previous code, SIFT or any other feature point detector. Select a subset of these. (You can use RANSAC if you want). Now the actual work: Compute the homography from one image to the other (or to a new image plane) using the SVD-algorithm. Warp (project) the pixels into that plane. (Which warping method shall you use?) Done!



Figure 17: The images meant to be stitched together

```

1  #!/usr/bin/env python
2  import numpy as np
3  import random
4  import cv2
5
6  def ransac(points_left, points_right, desc_left, desc_right):
7      # number of tries
8      n = 10000
9      best_points = 0
10     best_homo = None
11     max_len = min(len(points_left), len(points_right))
12
13     # find similar points in the left and right image
14     bf = cv2.BFMatcher()
15     matches = bf.match(desc_left, desc_right)
16     mp1 = [points_left[m.queryIdx].pt for m in matches]
17     mp2 = [points_right[m.trainIdx].pt for m in matches]
18
19     for _ in range(n):
20         # take 4 random points
21         l = [random.randint(0, max_len - 1) for _ in range(4)]
22         p = np.array([mp1[i] for i in l])
23         q = np.array([mp2[i] for i in l])
24
25         # try building a homography
26         potential_homo, _ = cv2.findHomography(p, q)
27         if np.array_equal(potential_homo, None):
28             continue # building a homography can fail
29         # warp all points
30         points_transformed = [potential_homo
31                               @ (point[0], point[1], 1) for point in mp1]
32         good_points = 0
33         for i in range(len(mp1)):
34             p_i = points_transformed[i][:2]
35             q_i = mp2[i]
36             diff = np.linalg.norm(p_i - q_i)
37             # evaluate the difference
38             if diff < 10: # if below a threshold
39                 good_points += 1 # good!
40             if best_points < good_points:
41                 best_points = good_points
42                 best_homo = potential_homo
43     return best_homo # return the best homography found

```

Figure 18: Part 1 of the image stitching algorithm


```

1
2 # read in images
3 path_left = './to_detect_2_1.jpg'
4 path_right = './to_detect_2_2.jpg'
5
6 img_left = cv2.imread(path_left)
7 img_right = cv2.imread(path_right)
8
9 # find feature points
10 sift = cv2.SIFT_create()
11 points_left, desc_left = sift.detectAndCompute(img_left, None)
12 points_right, desc_right = sift.detectAndCompute(img_right, None)
13
14 # create a suitable homography using RANSAC
15 homo = ransac(points_left, points_right, desc_left, desc_right)
16
17 # H = homography, l = left image, r = right image
18 #  $H * l = r \Rightarrow H^{-1} * r = l$ 
19
20 # take the inverse of the homography and warp the right
21 # image into the perspective of the left image
22 warped = cv2.warpPerspective(img_right, np.linalg.inv(homo),
23                               (img_right.shape[1] * 2, img_right.shape[0]))
24 # splat the left image on top
25 warped[:img_left.shape[0], :img_left.shape[1]] = img_left
26
27 # all done!
28 cv2.imwrite("stitched.jpg", warped)

```

Figure 19: Part 2 of the image stitching algorithm



Figure 20: Result of the image stitching algorithm

References

- [1] B.R. Kavitha, Ramya G, and Priya Govindaraj. “Performance comparison of various feature descriptors in object category detection application using SVM classifier”. In: *International Journal of Innovative Technology and Exploring Engineering* 8 (Jan. 2019), pp. 461–464.
- [2] *Reference list of currently published stereo methods*. URL: <https://vision.middlebury.edu/stereo/eval3/referenceList.php>.
- [3] Scott E. Fahlman and Christian Lebiere. *The Cascade-Correlation Learning Architecture*. URL: <https://proceedings.neurips.cc/paper/1989/file/69adc1e107f7f7d035d7baf04342e1ca-Paper.pdf>.