CS231A: Computer Vision, From 3D Reconstruction to Recognition

Homework #3 Solution

(Winter 2021)

Shubham Shrivastava email: shubhams@stanford.edu

1 (Optional) Matching Homographies for Image Rectification (20 points)

Building off the Fundamental Matrix Estimation From Point Correspondences in PSET 2, this problem seeks to rectify a pair of images given a few matching points. The main task in image rectification is generating two homographies H_1, H_2 that transform the images in a way that the epipolar lines are parallel to the horizontal axis of the image. A detailed description of the standard homography generation for image rectification can be found in Course Notes 3. You will implement the methods in `image_rectification.py` (note that you need to copy the implementation of `lls_eight_point_alg` and `normalized_eight_point_alg` from PSET 2 first) and complete the following:

(a) The first step in rectifying an image is to determine the epipoles. Complete the function `compute_epipole()`. Submit the epipoles you calculated and a copy of your code.
[5 points]

Tip: If you want to check your answer, recall that the epipole is where all the epipolar lines intersect.

The two epipoles computed are given follows:

```
e1 [-1.30071143e+03 -1.42448272e+02  1.00000000e+00]
e2 [1.65412463e+03 4.53021078e+01 1.00000000e+00]
```

```python
'''
COMPUTE_EPIPOLE computes the epipole in homogenous coordinates
given matching points in two images and the fundamental matrix
Arguments:
    points1 - N points in the first image that match with points2
    points2 - N points in the second image that match with points1
    F - the Fundamental matrix such that (points1)^T * F * points2 = 0

    Both points1 and points2 are from the get_data_from_txt_file() method
Returns:
    epipole - the homogenous coordinates [x y 1] of the epipole in the image
'''
def compute_epipole(points1, points2, F):
    # compute epipolar line
    epipolar_lines = np.dot(F.T, points2.T).T

    ## compute epipole
    # using the fact that the dot product of a line and a point on that line
    # equates to zero, we can build up a set of linear equations [l.e = 0],
    # whre l= epipolar line and e = epipole; and then solve for e using SVD.
    u, s, v_t = np.linalg.svd(epipolar_lines, full_matrices=True)
    # epipole can be obtained as the last column of v or last row of
    v_transpose
```

```
    e = v_t.T[:,-1]
    # set the last entry to 1.0
    e = e / e[-1]

    return e
```

Listing 1: Code Snippet

(b) Finally, we can determine the homographies H_1 and H_2. We first compute H_2 as the homography that takes the second epipole e_2 to a point at infinity $(f, 0, 0)$. The matching homography H_1 is computed by solving a least-squares problem. Complete the function `compute_matching_homographies()`. Submit the homographies computed for the sample image pair a copy of your code.
[10 points]

The two homography matrices computed are given below:

```
H1:
 [[-1.20007755e+01 -4.15506361e+00 -1.23444649e+02]
 [ 1.41004009e+00 -1.48704395e+01 -2.84177537e+02]
 [-9.21902786e-03 -2.19198036e-03 -1.23033437e+01]]

H2:
 [[ 8.09798121e-01 -1.22036874e-01  7.99331055e+01]
 [-3.00186872e-02  1.01581538e+00  3.63604736e+00]
 [-6.99360971e-04  1.05393949e-04  1.15205550e+00]]
```

```
'''
COMPUTE_MATCHING_HOMOGRAPHIES determines homographies H1 and H2 such that they
rectify a pair of images
Arguments:
    e2 - the second epipole
    F - the Fundamental matrix
    im2 - the second image
    points1 - N points in the first image that match with points2
    points2 - N points in the second image that match with points1
Returns:
    H1 - the homography associated with the first image
    H2 - the homography associated with the second image
'''
def compute_matching_homographies(e2, F, im2, points1, points2):
    '''
    This function converts a vector 'a' of size 3
    to a 3x3 matrix '[a]_x' for cross product to
    dot product conversion.
    '''
    def cross_product_mat(a):
        m = np.zeros((3,3), dtype=np.float32)
        m[1,0] = a[2]
        m[0,1] = -a[2]
        m[2,0] = -a[1]
        m[0,2] = a[1]
        m[2,1] = a[0]
        m[1,2] = -a[0]
        return m

    ## Let's first compute the homography H2 as H2 = T^(-1).GRT
    T = np.eye(3, dtype=np.float32)
    T[0,2] = -im2.shape[1] / 2.0
    T[1,2] = -im2.shape[0] / 2.0
    # translate the epipole
    Te2 = np.dot(T, e2.T)
```

```
    Te2 /= Te2[-1]
    # let's compute the rotation matrix
    alpha = 1 if Te2[0] >= 0 else 0
    R = np.eye(3, dtype=np.float32)
    R[0,0] = alpha * Te2[0]  / np.sqrt(Te2[0] ** 2 + Te2[1] ** 2)
    R[0,1] = alpha * Te2[1]  / np.sqrt(Te2[0] ** 2 + Te2[1] ** 2)
    R[1,0] = -1 * alpha * Te2[1]  / np.sqrt(Te2[0] ** 2 + Te2[1] ** 2)
    R[1,1] = alpha * Te2[0]  / np.sqrt(Te2[0] ** 2 + Te2[1] ** 2)
    RTe2 = np.dot(R, Te2)
    RTe2 /= RTe2[-1]
    # compute G
    G = np.eye(3, dtype=np.float32)
    G[-1,0] = -1. / RTe2[0]
    # compute H2
    H2 = np.dot(np.linalg.inv(T), np.dot(G, np.dot(R, T)))
    # H2 = H2 / H2[-1,-1] # up to scale

    ## Now, let's proceed to compute H1
    e2x = cross_product_mat(e2)
    v = np.array([1., 1., 1.], dtype=np.float32)
    e2v = np.dot(np.resize(e2, (3,1)), np.resize(v, (1,3)))
    M = np.dot(e2x, F) + e2v
    # Ha can now be computed by setting up a linear minimization problem
    p_hat = np.dot(H2, np.dot(M, points1.T)).T
    p_hat = np.array([(p_/p_[-1]) for p_ in p_hat], dtype=np.float32) # homogeneous format
    p_hat_prime = np.dot(H2, points2.T).T
    p_hat_prime = np.array([p_/p_[-1] for p_ in p_hat_prime], dtype=np.float32) # homogeneous format

    # For least-square solution of the problem [W.a = b], error is minimized by setting
    # [W_transpose.W.a = W_transpose.b] ref: https://eeweb.engineering.nyu.edu/iselesni/lecture_notes/least_squares/least_squares_SP.pdf
    # This gives us the solution: [a = {W_transpose.W}_inverse.W_transpose.b]
    # Build W and b matrices
    W = p_hat
    b = p_hat_prime[:,0]
    a = np.dot(np.dot(np.linalg.inv(np.dot(W.T, W)), W.T), b)
    # Alternatively, numpy can be used to directly compute this solution: a = np.linalg.lstsq(W, b)[0]
    Ha = np.eye(3, dtype=np.float32)
    Ha[0] = a
    # formulate H1
    H1 = np.dot(Ha, np.dot(H2, M))
    # H1 = H1 / H1[-1,-1] # up to scale
    # return homography matrices
    return H1, H2
```

Listing 2: Code Snippet

(c) Submit a copy of your rectified images plotted using `plot_epipolar_lines_on_images()`. **[5 points]**

Original unrectified images are shown in Figure 1 and the rectified images are shown in Figure 2 along with their epipolar lines.

2 Space Carving (45 points)

Dense 3D reconstruction is a difficult problem, as tackling it from the Structure from Motion framework (as seen in the previous problem set) requires dense correspondences. Another solution

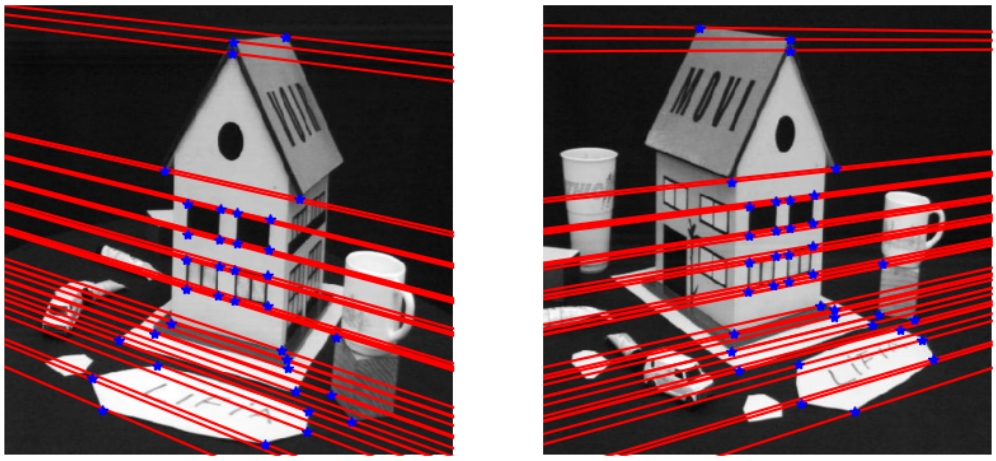

Figure 1: Original Unrectified Images along with their epipolar lines

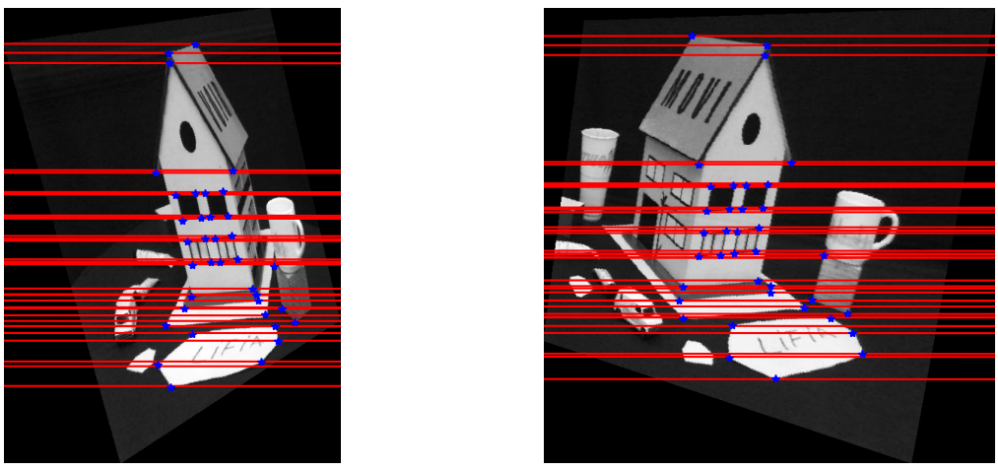

Figure 2: Rectified Images along with their epipolar lines

to dense 3D reconstruction is space carving[1], which takes the idea of volume intersection and iteratively refines the estimated 3D structure. In this problem, you implement significant portions of the space carving framework. In the starter code, you will be modifying the `p2.py` file inside the `p2` directory.

(a) The first step in space carving is to generate the initial voxel grid that we will carve into. Complete the function `form_initial_voxels()`. Submit an image of the generated voxel grid and a copy of your code. [**10 points**]

Initial voxel grid generated is shown in Figure 3 and the corresponding code is shown in listing 3.

```
'''
FORM_INITIAL_VOXELS  create a basic grid of voxels ready for carving

Arguments:
    xlim - The limits of the x dimension given as [xmin xmax]
    ylim - The limits of the y dimension given as [ymin ymax]
    zlim - The limits of the z dimension given as [zmin zmax]
    num_voxels - The approximate number of voxels we desire in our grid
Returns:
```

[1] http://www.cs.toronto.edu/~kyros/pubs/00.ijcv.carve.pdf

```python
    voxels - An ndarray of size (N, 3) where N is approximately equal the
        num_voxels of voxel locations.
    voxel_size - The distance between the locations of adjacent voxels
        (a voxel is a cube)

Our initial voxels will create a rectangular prism defined by the x,y,z
limits. Each voxel will be a cube, so you'll have to compute the
approximate side-length (voxel_size) of these cubes, as well as how many
cubes you need to place in each dimension to get around the desired
number of voxel. This can be accomplished by first finding the total volume of
the voxel grid and dividing by the number of desired voxels. This will give an
approximate volume for each cubic voxel, which can then be used to find the
side-length.
'''
def form_initial_voxels(xlim, ylim, zlim, num_voxels):
    # compute total volume of the voxel grid
    voxel_grid_vol = (xlim[1] - xlim[0]) * (ylim[1] - ylim[0]) * (zlim[1] -
    zlim[0])
    # find volume of each voxel
    voxel_vol = voxel_grid_vol / num_voxels
    # find side length of each voxel
    voxel_side_len = np.cbrt(voxel_vol)
    # build initial voxel grid
    initial_voxels = []
    for x in np.arange(xlim[0], xlim[1], voxel_side_len):
        for y in np.arange(ylim[0], ylim[1], voxel_side_len):
            for z in np.arange(zlim[0], zlim[1], voxel_side_len):
                initial_voxels.append([x, y, z])
    initial_voxels = np.array(initial_voxels, dtype=np.float64)
    return initial_voxels, voxel_side_len
```

Listing 3: Code Snippet

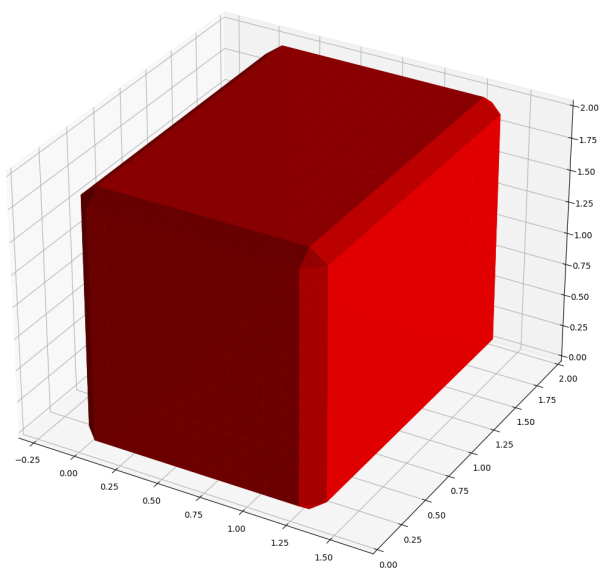

Figure 3: Initial Voxel Grid

(b) Now, the key step is to implement the carving for one camera. To carve, we need the camera frame and the silhouette associated with that camera. Then, we carve the silhouette from our voxel grid. Implement this carving process in `carve()`. Submit your code and a picture of what it looks like after one iteration of the carving. **[15 points]**

Space carving result after one iteration is shown in Figure 4 and the corresponding code is shown in listing 4.

```
'''
CARVE: carves away voxels that are not inside the silhouette contained in
    the view of the camera. The resulting voxel array is returned.

Arguments:
    voxels - an Nx3 matrix where each row is the location of a cubic voxel

    camera - The camera we are using to carve the voxels with. Useful data
        stored in here are the "silhouette" matrix, "image", and the
        projection matrix "P".

Returns:
    voxels - a subset of the argument passed that are inside the silhouette
'''
def carve(voxels, camera):
    ## start carving for z distance lower to higher
    # get 2d position of each voxel and find out whether
    # or not it is contained within the silhouette
    carved_voxels = []
    # compute the projection matrix
    Tmat = np.eye(4, dtype=np.float64)
    Tmat[:3,3] = camera.T
    Rmat = np.eye(4, dtype=np.float64)
    Rmat[:3,:3] = camera.R
    P = np.dot(camera.K, np.dot(Rmat, Tmat)[:3])
    # go through each voxel
    for voxel in voxels:
        # compute 2d projection of this voxel
        voxel_projection_2d = np.dot(camera.P, np.append(voxel, 1.0))
        voxel_projection_2d = np.array((voxel_projection_2d / voxel_projection_2d[-1])[:2], dtype=np.int32)
        # check if this point is within the silhouette
        if (voxel_projection_2d[0] >= 0) and (voxel_projection_2d[0] < camera.silhouette.shape[1]) and \
            (voxel_projection_2d[1] >= 0) and (voxel_projection_2d[1] < camera.silhouette.shape[0]) and \
            (camera.silhouette[voxel_projection_2d[1],voxel_projection_2d[0]] > 0):
             carved_voxels.append(voxel)
    carved_voxels = np.array(carved_voxels, dtype=np.float64)
    return carved_voxels
```

Listing 4: Code Snippet

(c) The last step in the pipeline is to carve out multiple views. Submit the final output after all carvings have been completed, using `num_voxels` $\approx 6,000,000$. **[5 points]**

The final output after space carving from all camera views is shown in Figure 5.

(d) Notice that the reconstruction is not really that exceptional. This is because a lot of space is wasted when we set the initial bounds of where we carve. Currently, we initialize the bounds of the voxel grid to be the locations of the cameras. However, we can do better than this by completing a quick carve on a much lower resolution voxel grid (we use `num_voxels` = 4000) to estimate how big the object is and retrieve tighter bounds. Complete the method `get_voxel_bounds()` and change the variable `estimate_better_bounds` in the `main()` function to `True`. Submit your new carving, which should be more detailed, and your code. **[10 points]**

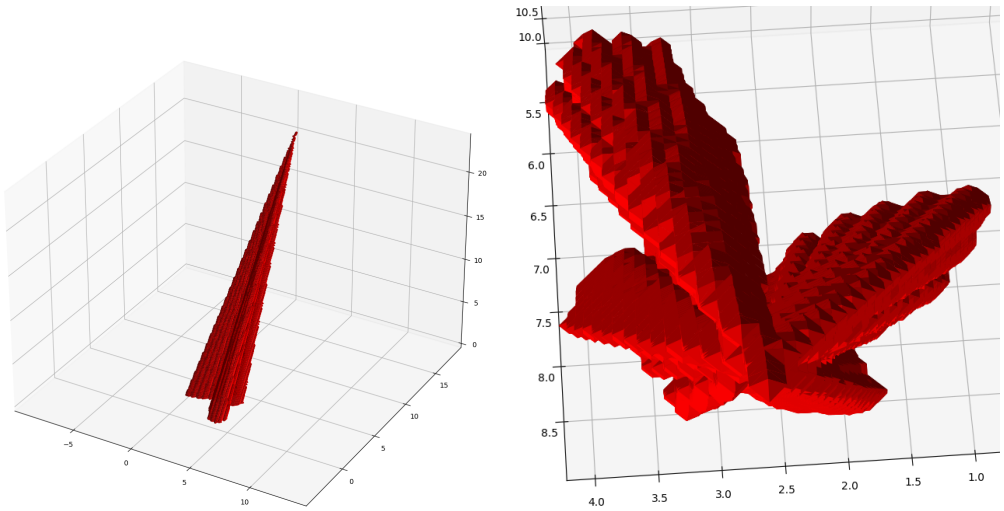

Figure 4: Result after the first iteration of space carving from two different views.

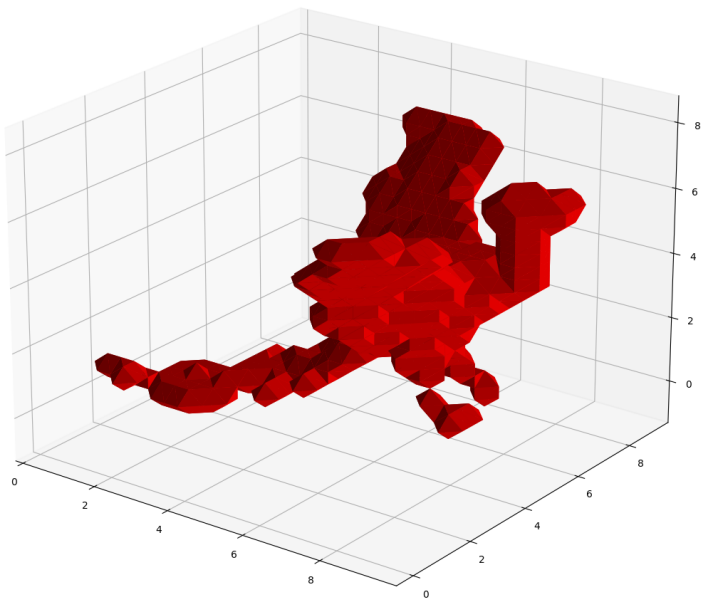

Figure 5: Final output after carving from all camera views.

The new carving as a result of more tightly bounded voxel intialization is shown in Figure 6 and corresponding code snippet is given in listing 5.

```
'''
GET_VOXEL_BOUNDS: Gives a nice bounding box in which the object will be carved
from. We feed these x/y/z limits into the construction of the inital voxel
cuboid.

Arguments:
    cameras - The given data, which stores all the information
        associated with each camera (P, image, silhouettes, etc.)

    estimate_better_bounds - a flag that simply tells us whether to set
    tighter
        bounds. We can carve based on the silhouette we use.

    num_voxels - If estimating a better bound, the number of voxels needed for
```

```
        a quick carving.

Returns:
    xlim - The limits of the x dimension given as [xmin xmax]

    ylim - The limits of the y dimension given as [ymin ymax]

    zlim - The limits of the z dimension given as [zmin zmax]

The current method is to simply use the camera locations as the bounds. In the
section underneath the TODO, please implement a method to find tigther bounds
by doing a quick carving of the object on a grid with very few voxels. From
    this coarse carving,
we can determine tighter bounds. Of course, these bounds may be too strict, so
     we should have
a buffer of one voxel_size around the carved object.
'''
def get_voxel_bounds(cameras, estimate_better_bounds = False, num_voxels =
    4000):
    camera_positions = np.vstack([c.T for c in cameras])
    xlim = [camera_positions[:,0].min(), camera_positions[:,0].max()]
    ylim = [camera_positions[:,1].min(), camera_positions[:,1].max()]
    zlim = [camera_positions[:,2].min(), camera_positions[:,2].max()]

    # For the zlim we need to see where each camera is looking.
    camera_range = 0.6 * np.sqrt(diff( xlim )**2 + diff( ylim )**2)
    for c in cameras:
        viewpoint = c.T - camera_range * c.get_camera_direction()
        zlim[0] = min( zlim[0], viewpoint[2] )
        zlim[1] = max( zlim[1], viewpoint[2] )

    # Move the limits in a bit since the object must be inside the circle
    xlim = xlim + diff(xlim) / 4 * np.array([1, -1])
    ylim = ylim + diff(ylim) / 4 * np.array([1, -1])

    if estimate_better_bounds:
        voxels, voxel_size = form_initial_voxels(xlim, ylim, zlim, num_voxels)
        for c in cameras:
            voxels = carve(voxels, c)

        xlim = [voxels[:,0].min()-voxel_size, voxels[:,0].max()+voxel_size]
        ylim = [voxels[:,1].min()-voxel_size, voxels[:,1].max()+voxel_size]
        zlim = [voxels[:,2].min()-voxel_size, voxels[:,2].max()+voxel_size]

    xlim = np.array(xlim, dtype=np.float64)
    ylim = np.array(ylim, dtype=np.float64)
    zlim = np.array(zlim, dtype=np.float64)
    return xlim, ylim, zlim
```

Listing 5: Code Snippet

(e) Finally, let's have a fun experiment. Notice that in the first three steps, we used perfect silhouettes to carve our object. Look at the `estimate_silhouette()` function implemented and its output. Notice that this simple method does not produce a really accurate silhouette. However, when we run space carving on it, the result still looks decent!

(i) Why is this the case? **[2 points]**

As shown in Figure 7, estimated silhouette is noisy. However, when looked from multiple camera views, most of these noise are carved out because they are not consistent, and hence we still get a decent result as shown in Figure 8.

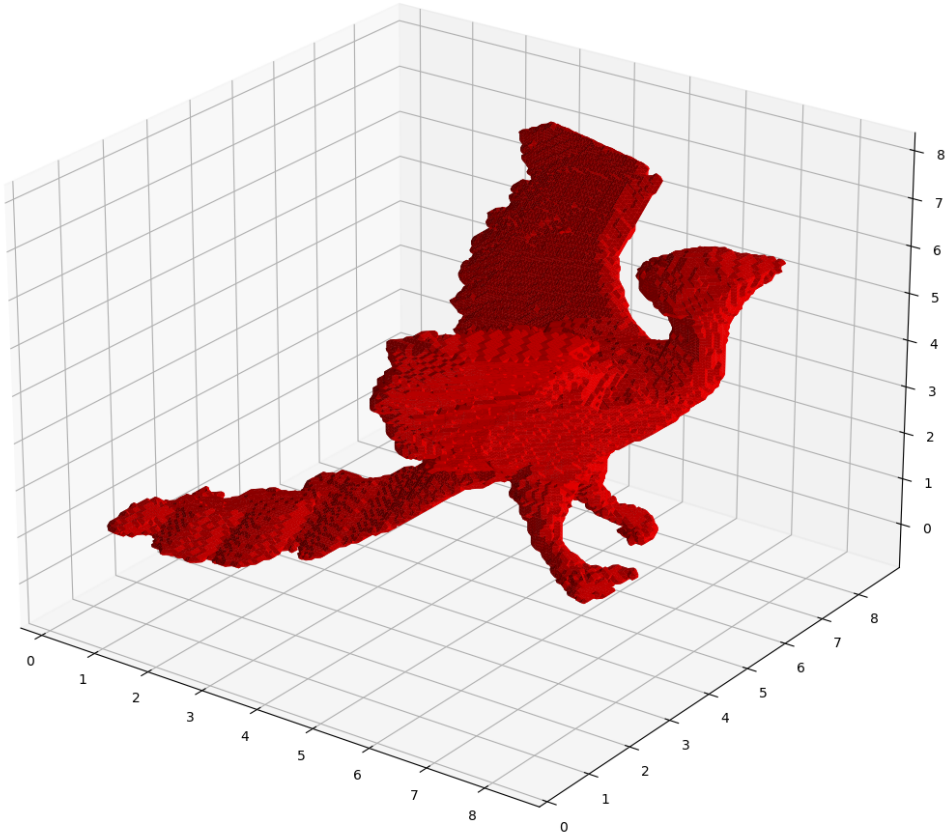

Figure 6: Final output after carving from all camera views based on tightly bounded initial vexels.

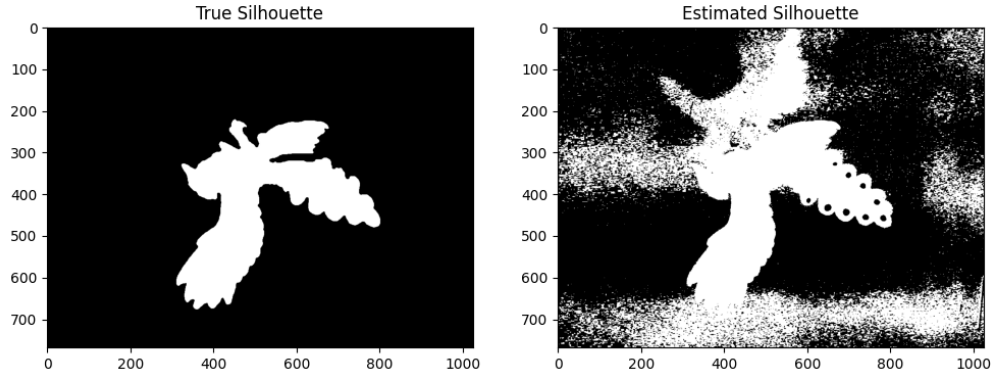

Figure 7: True vs Estimated silhouette.

(ii) What happens if you reduce the number of views? **[1 points]**

When number of views reduces, we end up with a more conservative set of voxels and our resulting model has voxels that do not belong to the object in reality.

(iii) What if the estimated silhouettes weren't conservative, meaning that one or a few views had parts of the object missing? **[2 points]**

If the estimated silhouettes weren't conservative and a few views had parts of the object missing, then those parts would be carved out from our final model. Space carving needs all the object voxels to be projected onto the image plane and be a part of estimated silhouette.

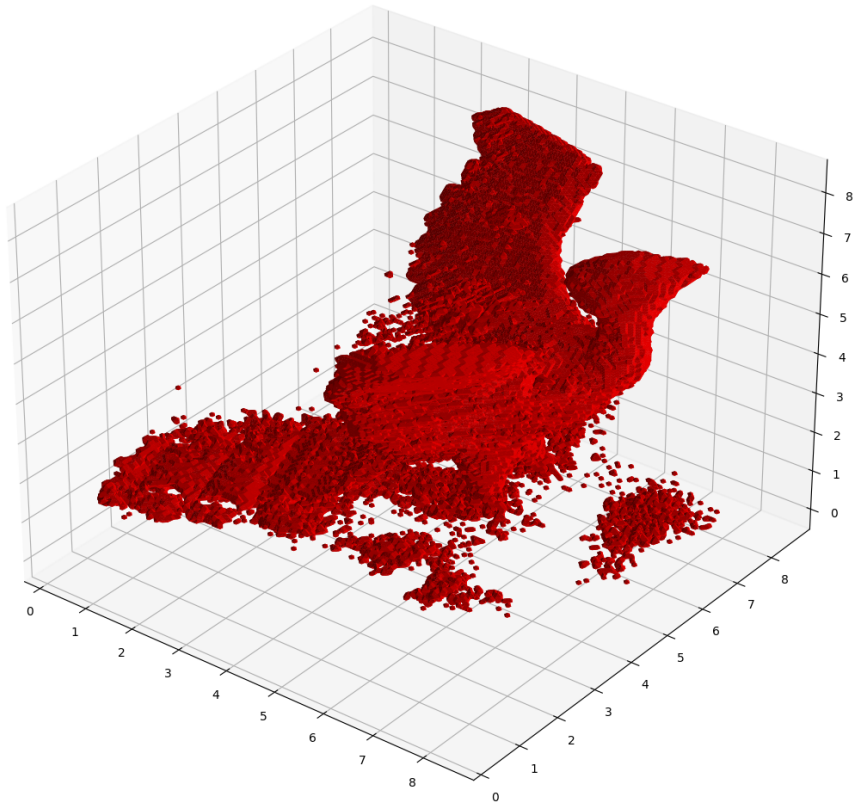

Figure 8: Final output when using noisy silhouette.

3 Representation Learning (15 points)

In this problem, you'll implement a method for image representation learning.

Unlike the previous problems, for this one you will have to work with Google Colaboratory. In Google Drive, follow these steps to make sure you have the ability to work on this problem:

a. Click the wheel in the top right corner and select Settings.

b. Click on the Manage Apps tab.

c. At the top, select Connect more apps which should bring up a GSuite Marketplace window.

d. Search for Colab then click Add.

Now, upload the contents of the 'code/represenation_learning' to a location of your choosing on Drive. Then, navigate to this folder and open the file "RepresentationLearning.ipynb" with Colaboratory. The rest of the instructions are provided in that document. Note that there is no autograder for this problem, as you should be able to confirm whether your implementation works from the images and plots. Include the following in your writeup:

a. Once you finish the section "PyTorch MNIST Data Preparation", include the 3 by 3 grid visualization of the MNIST data (5 points).

b. Once you finish the section "Training for MNIST Digit Prediction", include the graphs of training progress over 10 epochs (5 points).

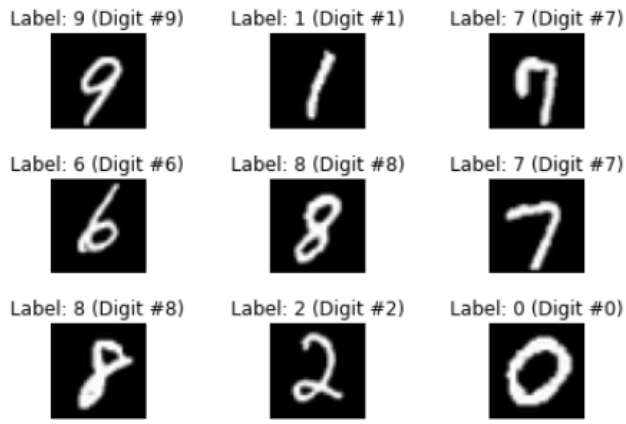

Figure 9: 3x3 grid showing MNIST classification data.

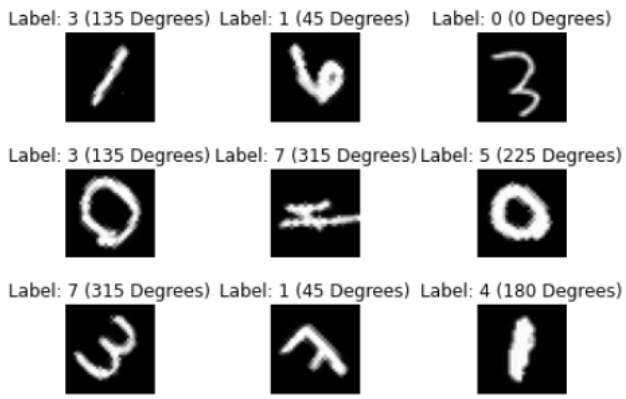

Figure 10: 3x3 grid showing MNIST rotation data.

c. Once you finish the section "Fine-Tuning for MNIST digit classification", include all graphs from this section (5 points).

   These graphs are shown in Figure 14, 15, 16, and 17

Fun fact: the method you just implemented was discovered fairly recently, in the ICLR2018 paper "Unsupervised Representation Learning by Predicting Image Rotations"!

4 (Optional) Monocular Depth Estimation (15 points)

Now that you've had some experience with representation learning on a small dataset, in this problem you will see how learned representations will help with estimating 3D structure in images by implementing training of monocular depth estimation on a real dataset (the NYU Depth Dataset V2). We will be implementing the approach taken in "High Quality Monocular Depth Estimation via Transfer Learning", which showed that taking a bigger model pre-trained on object classification proved to help a lot with training that model to perform monocular depth estimation.

As with the previous problem, you'll be working with Colaboratory, so once again begin by uploading the contents of 'code/monocular_depth_estimation' to a location of your choosing on Google Drive. Then, open the file "MonocularDepthEstimation.ipynb" with Colaboratory. The rest of the instructions are provided in that document. Note that there is no autograder for this problem, as you should be able to confirm whether your implementation works from the images and plots. Include the following in your writeup:

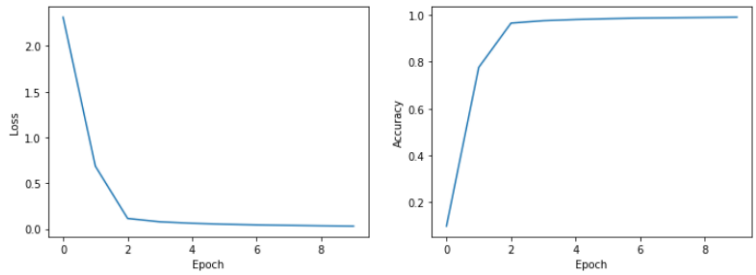

Figure 11: Training progress for 10 epochs on entire MNIST classification dataset.

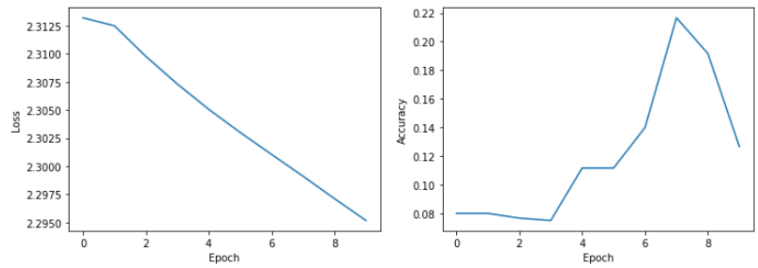

Figure 12: Training progress for 10 epochs on a subset of MNIST classification dataset.

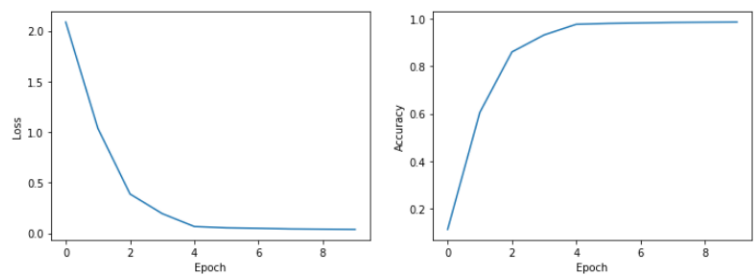

Figure 13: Training progress for 10 epochs on entire MNIST rotation dataset.

a. Once you finish the section "Checking out the data", include the grid visualization of the NYU data (5 points).

   Visualization of data is given in Figure 18

b. Once you finish the section "Testing the trained model", include the visualization of model predictions (5 points).

   Visualization of data is given in Figure 19

c. Once you finish the section "Training without feature transfer", include a screenshot of the Tensorboard graph showing the comparison between training from scratch and fine-tuning (5 points).

   Visualization of model predictions are shown in Figure 20 and the model training loss is shown in Figure 21.

NOTE: this problem requires fairly long training time on Colaboratory (on the order of hours) so be sure to plan accordingly! Checkpoints should be saved in your Drive after training, so you can pick up where you left off after different parts of the problem.

5 Tracking (40 points)

Lastly, a problem having to do with optical and scene flow. Consider the scenario depicted in Fig. 22, where an object O is being interacting by a person, while the robot observes the interaction with

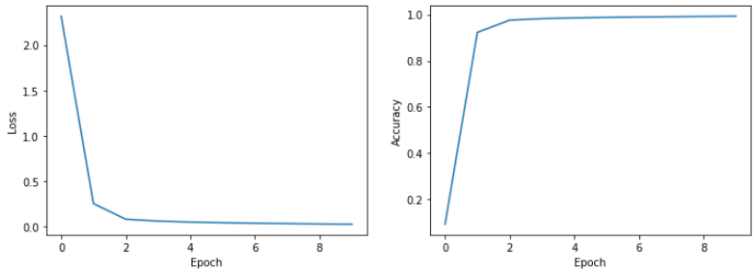

Figure 14: Training progress for 10 epochs on entire MNIST classification dataset with pre-trained image embedder.

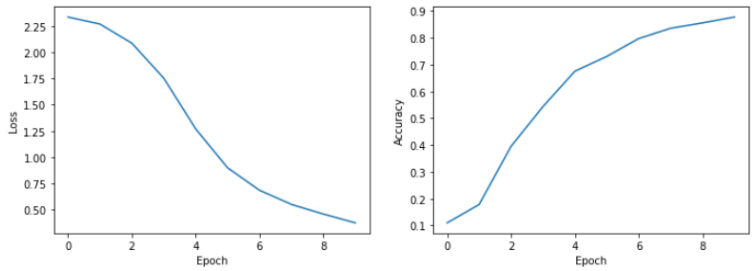

Figure 15: Training progress for 10 epochs on a subset of MNIST classification dataset with pre-trained image embedder.

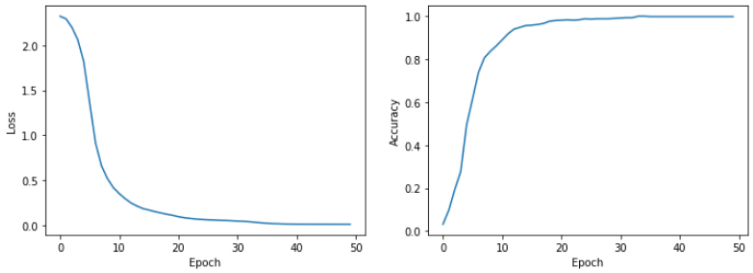

Figure 16: Training progress for 50 epochs on a subset of MNIST classification dataset with pre-trained image embedder.

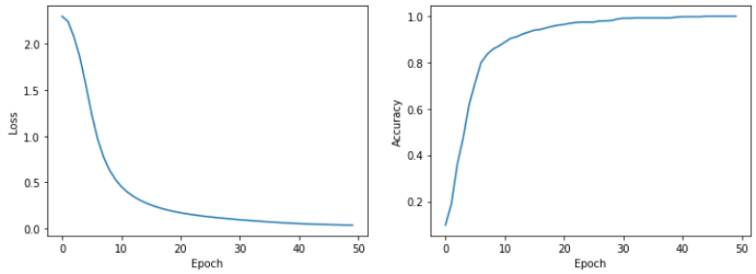

Figure 17: Training progress for 50 epochs on a subset of MNIST classification dataset with pre-trained image embedder (image embedder not optimized during training).

its RGB-D camera C. Files `rgb1.png`, `rgb2.png` and `rgb3.png` contain three RGB frames of the interaction as observed by the robot. Files `depth1.txt`, `depth2.txt` and `depth3.txt` contain the registered depth values for the same frames.

a. Detect and track the $N = 200$ best point features using Luca-Kanade point feature optical flow algorithm on the consecutive images, initialized with Tomasi-Shi corner features. Use the opencv implementation (look at this tutorial for more information: https://docs.opencv.org/3.4/d4/dee/tutorial_optical_flow.html). For the termination criteria use the num-

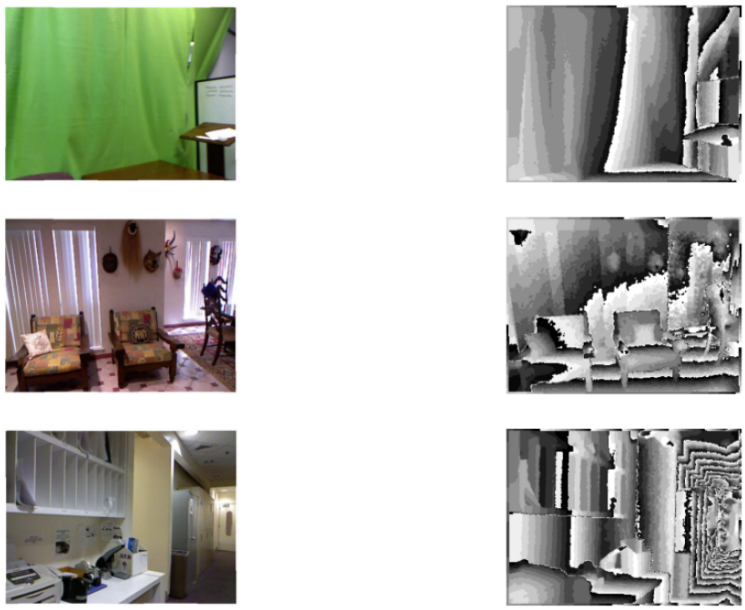

Figure 18: Visualization of NYU dataset.

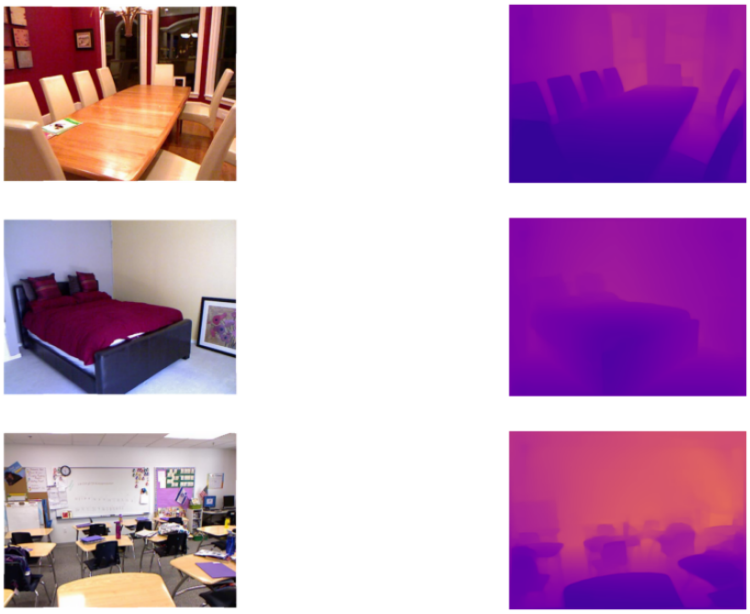

Figure 19: Visualization of model prediction on test dataset

ber of iterations and the computed pixel residual from *Bouget*, 2001. Use only 2 pyramid levels and use minimum eigen values as an error measure. What is the lowest quality level of the features detected in the first frame? Provide a visualization of the flow for block sizes 15×15 and 75×75. Explain the differences observed. Is it always better a larger or a smaller value? What is the trade-off? [**15 points**]

Note: You will need the opencv-python package. Please make sure that you have this installed.

The lowest quality of the features detected in the first frame is 0.01. Visualization of the flow for block sizes 15×15 and 75×75 is given in Figure 23. When a smaller block size

Figure 20: Visualization of model prediction with and without pre-training

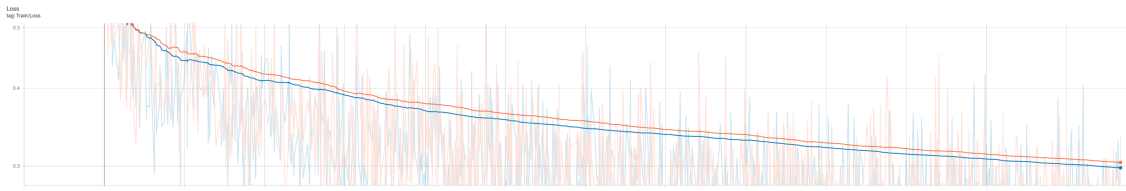

Figure 21: Visualization of training loss with and without pre-training. Orange line is without pre-training and blue line is with pre-training.

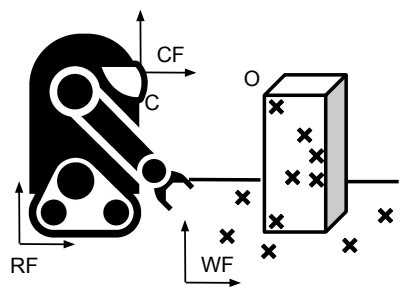

Figure 22

is used, keypoints cannot be tracked robustly through frames, and hence we observe some spurious tracks in the top-left image of Figure 23. While choosing larger number of frames is good for accuracy and robustness, it also increases the computation time and thus might be indesirable depending on use-cases. Code for this part is given in Listing 6

```
def Q5_A():
    """Code for question 5a.

    Output:
      p0, p1, p2: (N,2) list of numpy arrays representing the pixel coordinates of the
      tracked features.  Include the visualization and your answer to the
      questions in the separate PDF.
    """
    # params for ShiTomasi corner detection
    feature_params = dict(
        maxCorners=200,
        qualityLevel=0.01,
        minDistance=7,
        blockSize=7)

    # Parameters for lucas kanade optical flow
```

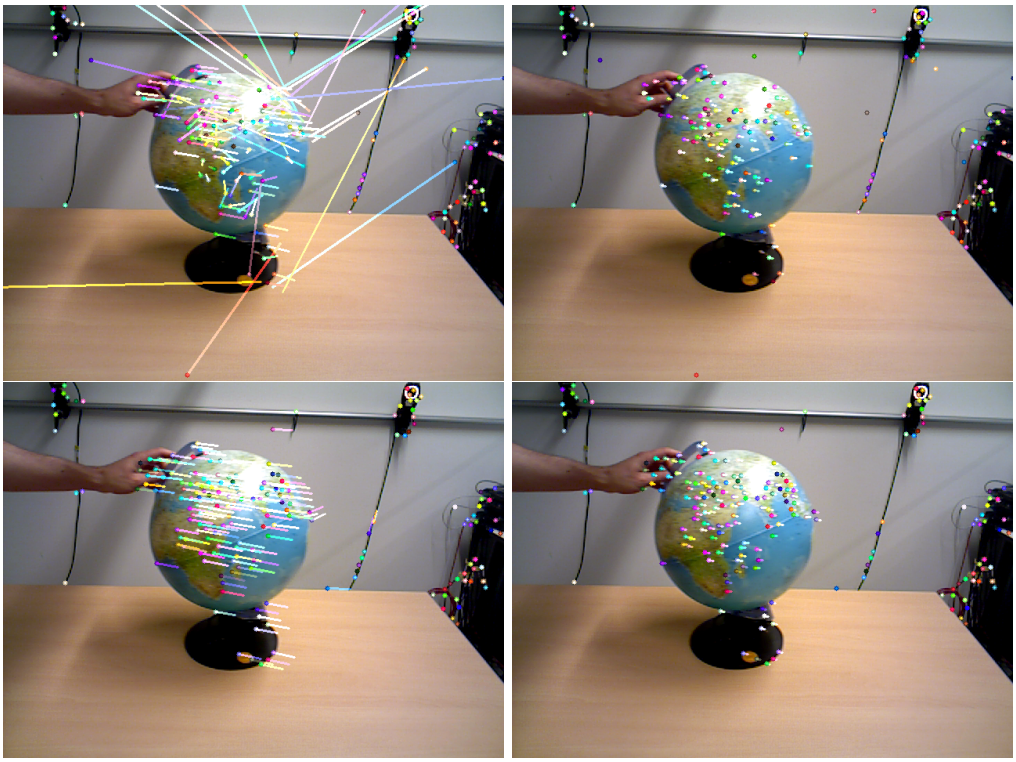

Figure 23: Top: Frame 0 → 1 and Frame 1 → 2 optical flow for detected keypoints with a window size of 15x15, Bottom: Frame 0 → 1 and Frame 1 → 2 optical flow for detected keypoints with a window size of 75x75

```python
    lk_params = dict(
        winSize=(75, 75),
        maxLevel=1,
        criteria=(cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 100, 0.01),
        flags=(cv2.OPTFLOW_LK_GET_MIN_EIGENVALS))

    # Read the frames.
    frame1 = cv2.imread('p5_data/rgb1.png')
    frame2 = cv2.imread('p5_data/rgb2.png')
    frame3 = cv2.imread('p5_data/rgb3.png')
    frames = [frame1, frame2, frame3]

    # Convert to gray images.
    old_frame = frames[0]
    old_gray = cv2.cvtColor(old_frame, cv2.COLOR_BGR2GRAY)
    p0 = cv2.goodFeaturesToTrack(old_gray, mask = None, **feature_params)

    # Create some random colors for drawing
    color = np.random.randint(0, 255, (200, 3))

    # Create a mask image for drawing purposes
    mask = np.zeros_like(frame1)

    pn = []
    for i,frame in enumerate(frames[1:]):
        frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

        # calculate optical flow
        p1, st, err = cv2.calcOpticalFlowPyrLK(old_gray, frame_gray, p0, None, **lk_params)

        # Select good points
        if p1 is not None:
            good_new = p1[st==1]
            good_old = p0[st==1]

            if i == 0:
                pn.append(good_old.reshape(-1, 2))
            pn.append(good_new.reshape(-1, 2))

        # # plot optic flow
        # draw_tracks(i, frame.copy(), mask.copy(), p0, p1, color)

        # update the old points
        old_gray = frame_gray
        p0 = p1

    return pn[0], pn[1], pn[2]
```

Listing 6: Code Snippet

b. Use the registered depth maps to estimate the sparse scene flow between frames for the features obtained in the previous steps. Provide the point clouds at the three frames if all values are not NaN. [**25 points**]

Visualization of the point-clouds are given in Figure 24 and corresponding code is given in Listing 7.

```python
def Q5_B(p0, p1, p2, intrinsic):
    """Code for question 5b.

    Note that depth maps contain NaN values.
```

```python
    Features that have NaN depth value in any of the frames should be excluded
    in the result.

    Input:
      p0, p1, p2: (N,2) numpy arrays, the results from Q2_A.
      intrinsic: (3,3) numpy array representing the camera intrinsic.

    Output:
      p0, p1, p2: (N,3) numpy arrays, the 3D positions of the tracked features
      in each frame.
    """
    depth0 = np.loadtxt('p5_data/depth1.txt')
    depth1 = np.loadtxt('p5_data/depth2.txt')
    depth2 = np.loadtxt('p5_data/depth3.txt')

    p0_3d, p1_3d, p2_3d = [], [], []
    # go through each points and get their 3D coordinates
    for pt0_2d, pt1_2d, pt2_2d in zip(p0, p1, p2):
        z0 = float(depth0[int(pt0_2d[1]), int(pt0_2d[0])])
        z1 = float(depth1[int(pt1_2d[1]), int(pt1_2d[0])])
        z2 = float(depth2[int(pt2_2d[1]), int(pt2_2d[0])])
        if (not np.isnan(z0)) and (not np.isnan(z1)) and (not np.isnan(z2)):
            x = (pt0_2d[0] - intrinsic[0,2]) * z0 / intrinsic[0,0]
            y = (pt0_2d[1] - intrinsic[1,2]) * z0 / intrinsic[1,1]
            p0_3d.append([x, y, z0])

            x = (pt1_2d[0] - intrinsic[0,2]) * z1 / intrinsic[0,0]
            y = (pt1_2d[1] - intrinsic[1,2]) * z1 / intrinsic[1,1]
            p1_3d.append([x, y, z1])

            x = (pt2_2d[0] - intrinsic[0,2]) * z2 / intrinsic[0,0]
            y = (pt2_2d[1] - intrinsic[1,2]) * z2 / intrinsic[1,1]
            p2_3d.append([x, y, z2])
    p0_3d = np.array(p0_3d, dtype=np.float64)
    p1_3d = np.array(p1_3d, dtype=np.float64)
    p2_3d = np.array(p2_3d, dtype=np.float64)

    return p0_3d, p1_3d, p2_3d
```

Listing 7: Code Snippet

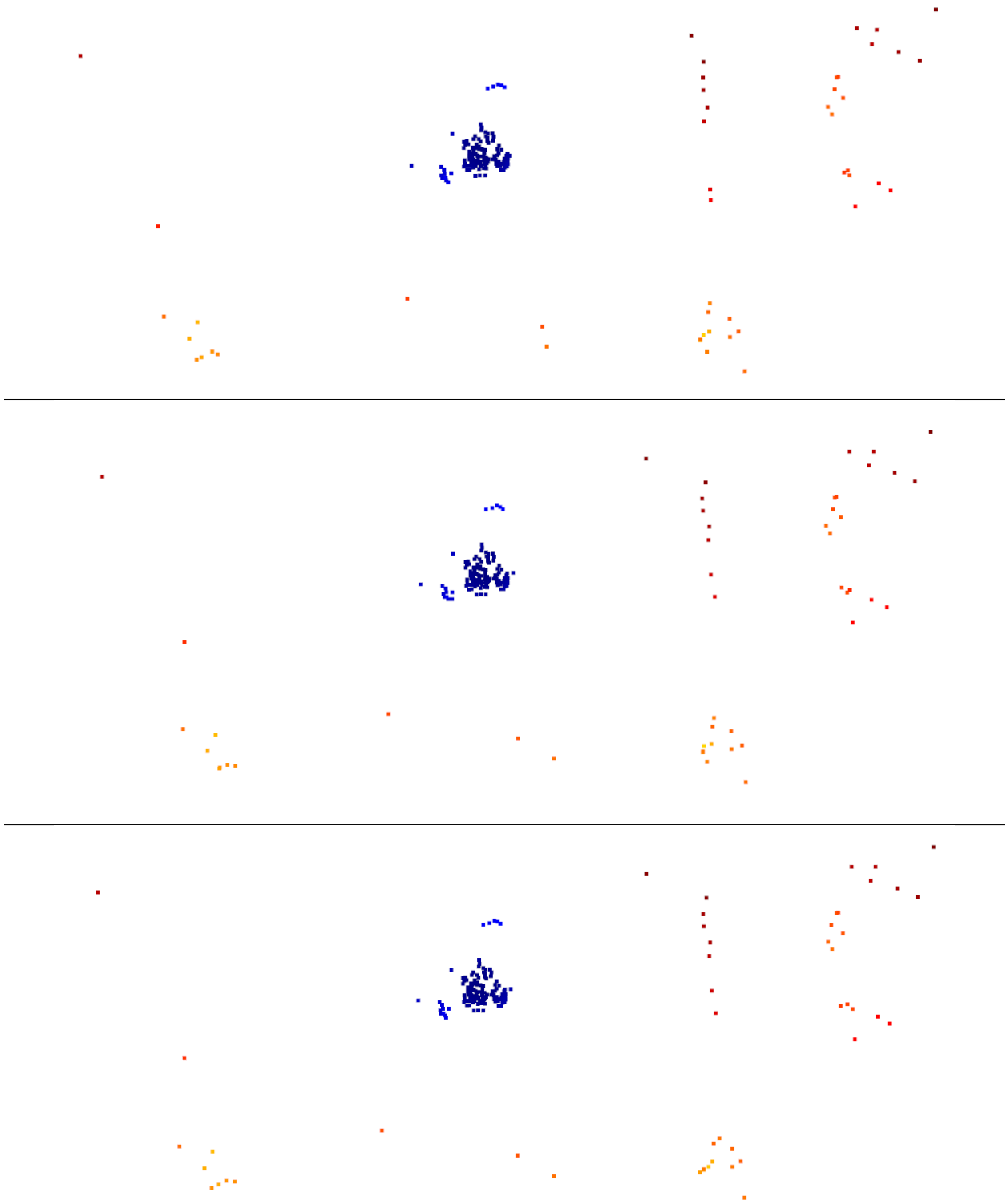

Figure 24: Sparse scene flow point-cloud showing the tracked points across frames. Top: Frame 0, Center: Frame 1, Bottom: Frame 2