CS4277/CS5477 Lab 4: Plane Sweep Stereo

Introduction

In this assignment, you will compute a dense 3D model from multi-view stereo, i.e. the plane sweeping algorithm.

This assignment is worth **20%** of the final grade.

References:

- Lecture 3
- Lecture 10

Optional references:

- A space-sweep approach to true multi-image matching [link]

Instructions

This workbook provides the instructions for the assignment, and facilitates the running of your code and visualization of the results. For each part of the assignment, you are required to **complete the implementations of certain functions in the accompanying python file** (`lab4.py`).

To facilitate implementation and grading, all your work is to be done in that file, and **you only have to submit the .py file**.

Please note the following:

1. Fill in your name, email, and NUSNET ID at the top of the python file.
2. The parts you need to implement are clearly marked with the following:

   ```
   """ YOUR CODE STARTS HERE """

   """ YOUR CODE ENDS HERE """
   ```

3. Note that for each part, there may certain functions that are prohibited to be used. It is important **NOT to use those prohibited functions** (or other functions with similar functionality). If you are unsure whether a particular function is allowed, feel free to ask any of the TAs.

Submission Instructions

Upload your completed `lab4.py` onto the relevant work bin in Luminus.

Part 1: Computation of Relative Extrinsics, Plane Sweep Homography

In this part, you will first create helper modules for plane sweep stereo. As we have discussed in the lectures, two cameras observing a plane are related by a homography. In this part, you will compute the homography that relates two given camera views.

Below, we provide the camera intrinsic matrix, as well as the camera poses for two cameras. To simplify the problem, both cameras share the same intrinsic matrix K. The camera poses are given as a 3×4 matrix $\mathbf{M} = [\mathbf{R}|\mathbf{t}]$ (consisting of a rotation matrix $\mathbf{R}$ and a translation vector $\mathbf{t}$) that transforms points in the world frame (X_w, Y_w, Z_w) to the input camera frame (X_c, Y_c, Z_c), i.e.

$$\begin{bmatrix} X_i \\ Y_i \\ Z_i \end{bmatrix} = \mathbf{M} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

[ ]:
```
# Intrinsic matrix
K = np.array([[615.,   0., 320.],
              [  0., 615., 240.],
              [  0.,   0.,   1.]])

M_ref = np.array([[ 3.93242408e-01, 7.74463219e-05,  9.19434828e-01, -1.
 ↪44320811e+00],
                  [ 1.51103741e-01, 9.86397614e-01, -6.47101783e-02,  1.
 ↪79934373e-01],
                  [-9.06933332e-01, 1.64376828e-01,  3.87881670e-01,  1.
 ↪12435457e+00]])
M_i = np.array([[ 5.78006380e-01, -4.51644884e-05,  8.16032243e-01, -1.
 ↪43107499e+00],
                [ 1.09801992e-01,  9.90910301e-01, -7.77193532e-02,  1.
 ↪43357565e-01],
                [-8.08611246e-01,  1.34524248e-01,  5.72757435e-01,  8.
 ↪10162936e-01]])
```

Before we can compute the plane sweep homographies between two cameras, we need to first compute the relative poses (i.e. extrinsic matrices) between two cameras. Given the pose of camera of interest $\mathbf{M}_i$ and the reference camera pose $\mathbf{M}_{ref}$, compute the relative pose between the two cameras which transforms points in the reference frame the frame of the camera of interest.

Implement the following functions(s): compute_relative_pose()

- Prohibited Functions: `np.linalg.inv()`

Once you have the relative pose, you can now compute the homographies that relate the images in the two cameras for given (inverse) depths. To simplify the problem, assume that the planes are fronto-parallel with respect to the reference camera. For D inverse depth values, your function should output a matrix of size $(D, 3, 3)$.

Implement the following functions(s): get_plane_sweep_homographies().

Note that the depths are provided as *inverse* depths $1/d$.

[ ]:
```
from lab4 import get_plane_sweep_homographies, compute_relative_pose

# Inverse depths
inv_depths = np.array([0.1, 0.06944444, 0.03888889, 0.00833333])
relative_pose = compute_relative_pose(M_i, M_ref)
homographies = get_plane_sweep_homographies(K, relative_pose, inv_depths)
print(homographies)
```

If implemented correctly, the above should print:

```
[[[ 1.086e+00  2.067e-02 -1.541e+02]
  [ 5.322e-02  9.887e-01 -5.925e+00]
  [ 3.393e-04 -4.318e-05  8.811e-01]]

 [[ 1.086e+00  2.067e-02 -1.582e+02]
  [ 5.322e-02  9.887e-01 -3.899e+00]
  [ 3.393e-04 -4.318e-05  8.806e-01]]

 [[ 1.086e+00  2.067e-02 -1.623e+02]
  [ 5.322e-02  9.887e-01 -1.873e+00]
  [ 3.393e-04 -4.318e-05  8.801e-01]]

 [[ 1.086e+00  2.067e-02 -1.663e+02]
  [ 5.322e-02  9.887e-01  1.527e-01]
  [ 3.393e-04 -4.318e-05  8.796e-01]]]
```

Part 2: Plane Sweep Stereo

In this part, you will write the code for performing the plane sweep stereo. We will perform plane sweeps on images from the Tsukuba dataset.

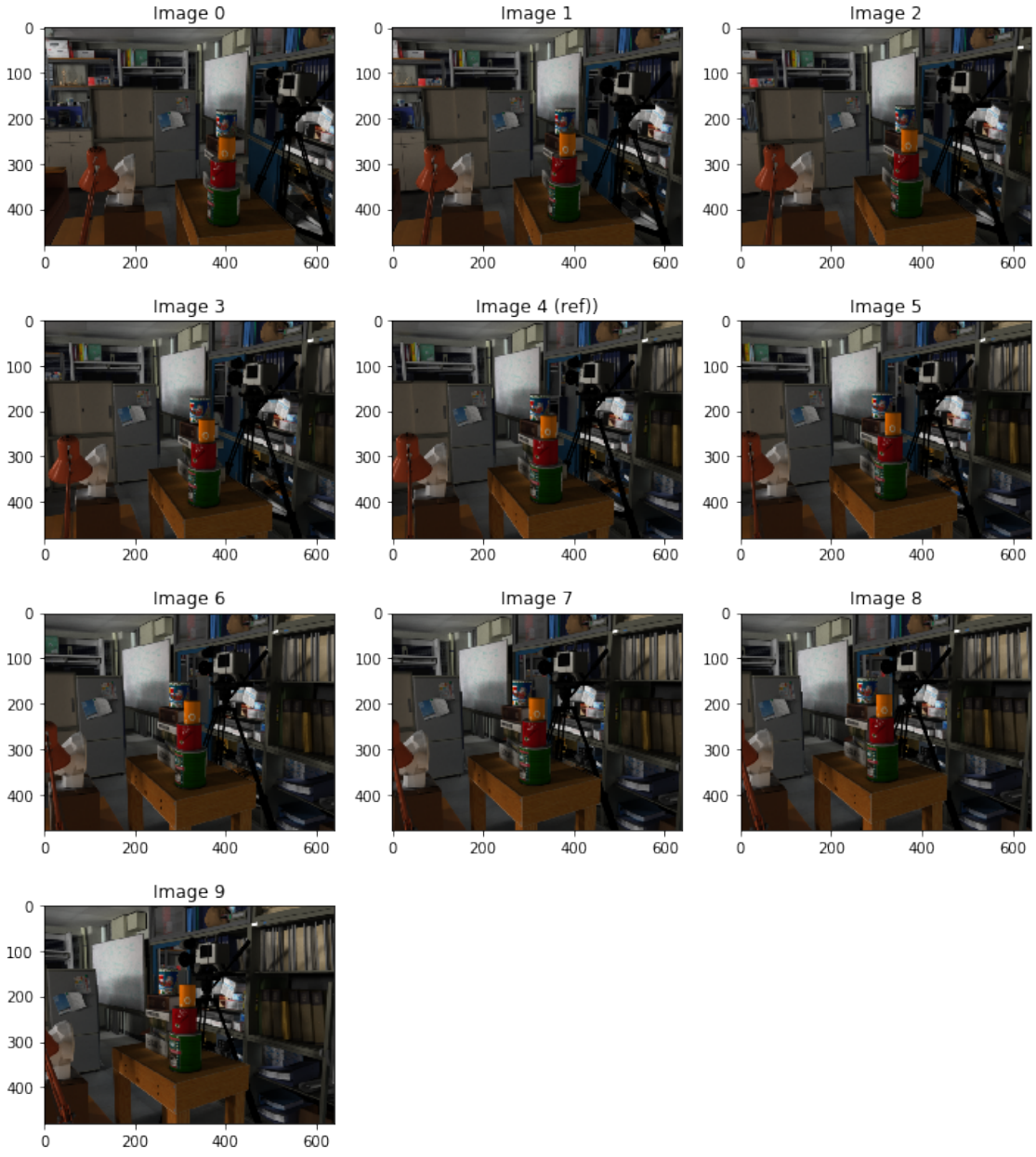
The dataset we will work on contains $n = 10$ images. We will use one of these as the reference camera view, and warp all images to this view. Let us first load the dataset and visualize the images.

[ ]:
```
from lab4 import load_data

ref_id = 4  # use image 4 as the reference view
data_folder = 'data/tsukuba'
images, K, (img_height, img_width) = load_data(data_folder)
ref_pose = images[ref_id].pose_mat
print('Reference camera pose:\n', ref_pose)

# Visualizes the source images
plt.figure(figsize=(12, 14))
num_rows = math.ceil(len(images) / 3)
plt.tight_layout()
```

```
for i in range(len(images)):
    plt.subplot(num_rows, 3, i+1)
    plt.imshow(images[i].image)
    if i == ref_id:
        plt.title('Image {} (ref))'.format(i))
    else:
        plt.title('Image {}'.format(i))
```



The images are stored as instances of our `Image` class. You can see its implementation in `lab4.py`, but otherwise you should only need to access the following member variables: - `pose_mat`: 3x4

transform M to transform points from world to camera frame, i.e. $p_c = M \cdot p_w$ - `image`: The image bitmap itself

Now, the task is to implement the function to compute the plane sweep volume. The steps to do so are as follows:

1. Compute the relative pose between each image and the reference view
2. For every fronto-parallel plane at depth d
    - Compute the homography transforms to warp each of the images to this reference view
    - Warp the images
3. Compute the variance at every pixel and depth. Compute the variance for each of the RGB channels separately, then take the average.

Note that not every image will cover every pixel of the reference view at all depths. You need to take this into account when computing the variance (i.e. the variance may be computed on less than n images).

After computing the plane sweep volume, the depths can be obtained as the depth that results in the minimum variance.

Implement the following functions(s): compute_plane_sweep_volume(), compute_depths()

Your function should output two required and one optional variables: 1. `ps_volume`: Plane sweep volume of size (D, H, W) 2. `accum_count`: Number of valid images considered in computing the variance for each pixel, with size (D, H, W) 3. `extras` (optional): Any extra information you might want to use for part 4 of the assignment.

Hint: You might find the following functions useful: `get_plane_sweep_homographies()` (from Part 1), and `cv2.warpPerspective()`.

Note that plane sweep stereo is very computation heavy. **A portion of the score for this part will be based on how fast your code runs.**

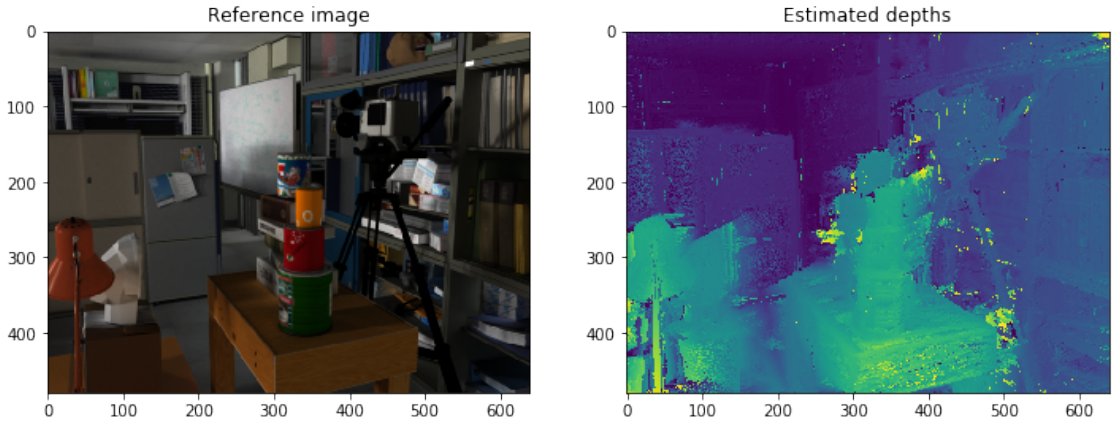
[ ]:
```
# Sweep D=256 planes from 0.8 to 6.0 meters away
num_depths = 256
inv_depths = np.linspace(1/0.8, 1/8.0, num=num_depths)

ps_volume, accum_count, extras = compute_plane_sweep_volume(images, ref_pose, K,␣
 ↪inv_depths,
                                                            (img_height,␣
 ↪img_width))
```

[ ]:
```
# Compute (inverse) depth from plane sweep volume and visualize
inv_depth_img = compute_depths(ps_volume, inv_depths)

plt.figure(figsize=(12,8))
plt.subplot(1,2,1)
plt.imshow(images[ref_id].image)
plt.title('Reference image');
```

```
plt.subplot(1,2,2)
plt.imshow(inv_depth_img)
plt.title('Estimated depths');
```



If implemented correctly, the above should show a reasonable estimate of the depths of the reference image (brighter colors indicates nearer objects).

Part 3: Unprojection of depth map

You will now unproject the depth maps back into colored 3D points. This will allow you to visualize the 3D model as a point cloud.

Implement the following functions(s): unproject_depth_map()

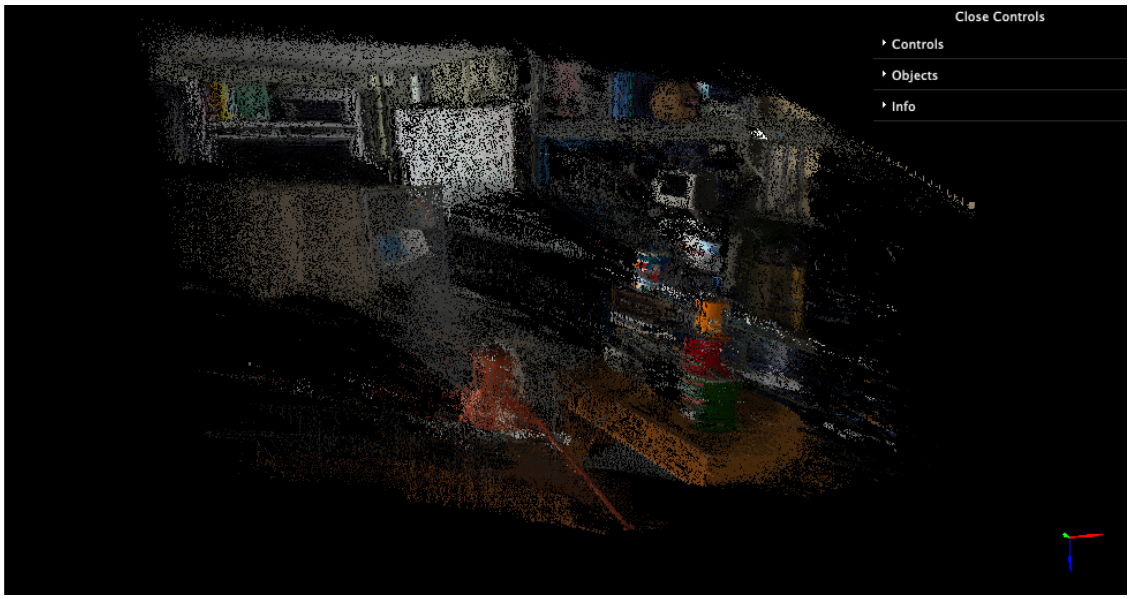
The function should take in the inverse depth maps and output an Nx3 array for the locations and another Nx3 array for RGB values. The function should also take in an optional mask which you will generate in Part 4 to indicate which pixels have confident depth estimates.

You might find the following functions useful: `np.meshgrid()`

[ ]:
```
xyz, rgb = unproject_depth_map(images[ref_id].image, inv_depth_img, K)
```

Let see how our model looks like in 3D. You might need to rotate the view to obtain a nicer view of the point cloud.

[ ]:
```
plot = k3d.plot(grid_visible=False)
plt_points = k3d.points(positions=xyz.astype(np.float32),
                        colors=rgb2hex(rgb),
                        point_size=0.5, shader="dot",)
plot += plt_points
plot.display()
```



Part 4: Improve the results

You will notice that the above depth map (and subsequently 3D point cloud) is very noisy. In this portion, we try to denoise the depth map and mask out less confident pixels.

Implement the following functions(s): post_process()

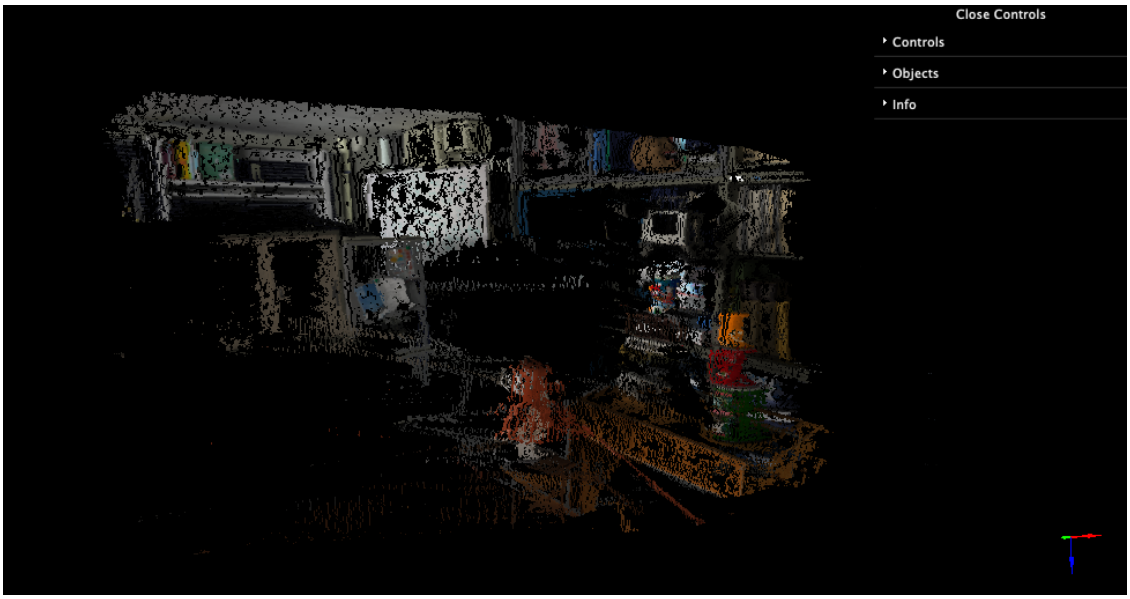
No specific instructions are provided for this part, so use your creativity here! One simple way is to consider how many measurements are used to compute the variance (i.e. `accum_count`). The score for this part will be based on how well the resulting 3D model looks. We will also evaluate on a hold-out image set, so do not cheat!

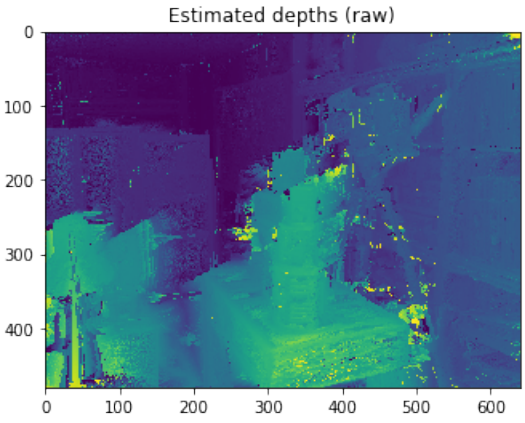
[ ]:
```
inv_depth_img2, valid_mask = post_process(ps_volume, inv_depths, accum_count,␣
↪extras)
```

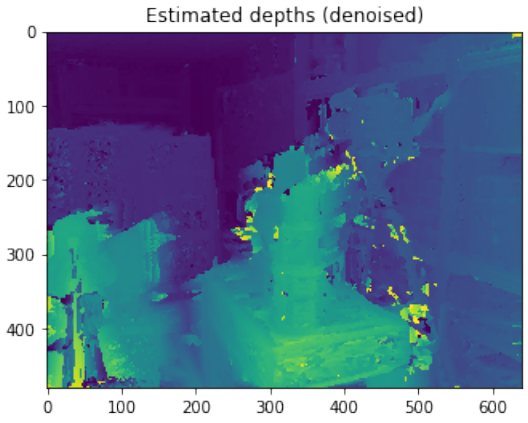
Let us take a look at the post processed depths and unprojected points and see if it looks better now.

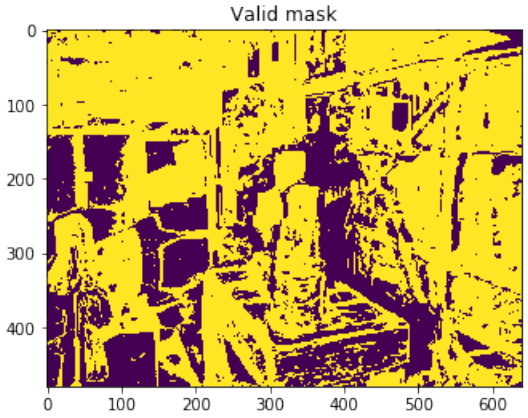
[ ]:
```
plt.figure(figsize=(12,9))
plt.subplot(2,2,1)
plt.imshow(inv_depth_img)
plt.title('Estimated depths (raw)');
plt.subplot(2,2,2)
plt.imshow(inv_depth_img2)
plt.title('Estimated depths (denoised)');
plt.subplot(2,2,4)
plt.imshow(valid_mask)
plt.title('Valid mask');
```

```
xyz, rgb = unproject_depth_map(images[ref_id].image, inv_depth_img2, K,␣
 ↪valid_mask)
plot2 = k3d.plot(grid_visible=False)
plt_points2 = k3d.points(positions=xyz.astype(np.float32),
                         colors=rgb2hex(rgb),
                         point_size=0.5, shader="dot",)
plot2 += plt_points2
plot2.display()
```




Estimated depths (raw)
0
100
200
300
400
0
100
200
300
400
500
600


Estimated depths (denoised)
0
100
200
300
400
0
100
200
300
400
500
600


Valid mask
0
100
200
300
400
0
100
200
300
400
500
600