

CS4277/CS5477 Assignment 6: Dense 3D model from multi-view stereo

1 Introduction

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

References: * Lecture 11

Optional references: * A space-sweep approach to true multi-image matching [[link](#)]

Honour Code. This coding assignment constitutes 10% of your final grade in CS4277/CS5477. Note that plagiarism will not be condoned! You may discuss with your classmates and check the internet for references, but you **MUST NOT** submit code/report that is copied directly from other sources!

1.1 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** (lab6.py).

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 """  
...
```

, and you should write your code in between the above two lines.

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).

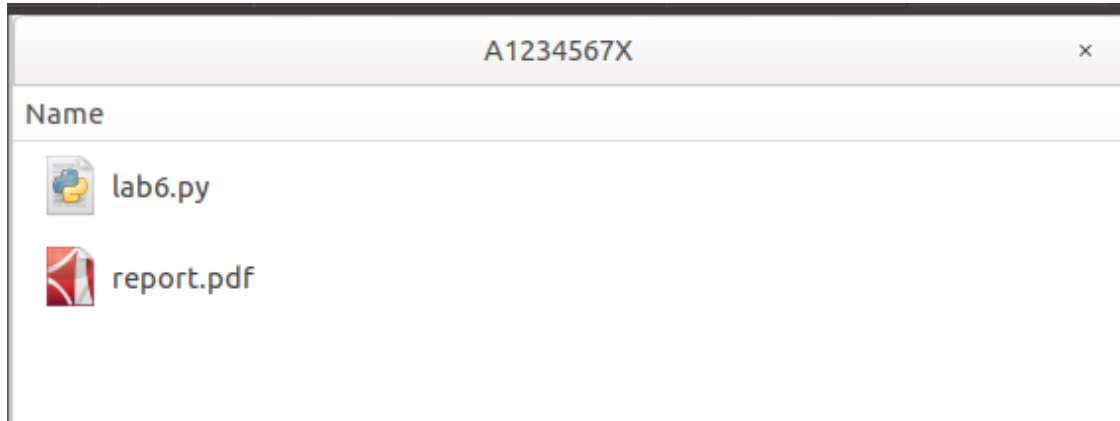
1.2 Submission Instructions

Items to be submitted:

- **Source code (lab6.py).** This is where you fill in all your code.

- **Report (report.pdf).** This should describe your implementation and be no more than one page.

Please clearly indicate your name and student number (the one that looks like A1234567X) in the report as well as the top of your source code. Zip the two files together and name it in the following format: A1234567X.zip (replace with your student number). Opening the zip file should show:



Submit your assignment by **17 April 2021, 2359HRS** to LumiNUS. 25% of the total score will be deducted for each day of late submission.

2 Computation of Plane Sweep Homography

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 relative pose between the two cameras. To simplify the problem, both cameras share the same intrinsic matrix K . The relative pose are given as a 3×4 matrix M_{rel} that transforms points in the reference camera frame (X_r, Y_r, Z_r) to the input camera frame (X_i, Y_i, Z_i) , i.e.

$$\begin{bmatrix} X_i \\ Y_i \\ Z_i \end{bmatrix} = M_{rel} \begin{bmatrix} X_r \\ Y_r \\ Z_r \\ 1 \end{bmatrix}$$

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

```
[1]: %matplotlib inline
      %load_ext autoreload
      %autoreload 1
      %import lab6

      import math
      import json
```

```

import cv2
import k3d
import matplotlib.pyplot as plt
import numpy as np

from lab6 import *

np.set_printoptions(precision=6) # Print less digits

```

```

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

# relative pose M_rel
relative_pose = np.array([[0.85666636, -0.07970693, 0.50967593, -5.07139457],
                          [0.107982, 0.99381078, -0.02607722, 2.38452488],
                          [-0.5044429, 0.07737531, 0.85997118, 1.34159884]])

# Inverse depths
inv_depths = np.array([0.1, 0.06944444, 0.03888889, 0.00833333])

```

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

Given the above inputs, compute the homographies that relate the images in the two camera for the 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 $(D, 3, 3)$.

```

[4]: from lab6 import get_plane_sweep_homographies

homographies = get_plane_sweep_homographies(K, relative_pose, inv_depths)
print(homographies)

```

If implemented correctly, the above should print:

```

[[[ 5.941920e-01 -3.944661e-02  1.390076e+02]
  [-8.887377e-02  1.024006e+00  1.518804e+02]
  [-8.202324e-04  1.258135e-04  1.226410e+00]]

 [[ 5.941920e-01 -3.944661e-02  2.211897e+02]
  [-8.887377e-02  1.024006e+00  9.723281e+01]
  [-8.202324e-04  1.258135e-04  1.185417e+00]]

 [[ 5.941920e-01 -3.944661e-02  3.033718e+02]
  [-8.887377e-02  1.024006e+00  4.258523e+01]
  [-8.202324e-04  1.258135e-04  1.144424e+00]]

 [[ 5.941920e-01 -3.944661e-02  3.855539e+02]

```

```
[-8.887377e-02  1.024006e+00 -1.206237e+01]
[-8.202324e-04  1.258135e-04  1.103430e+00]]]
```

3 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.

```
[5]: from lab6 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)

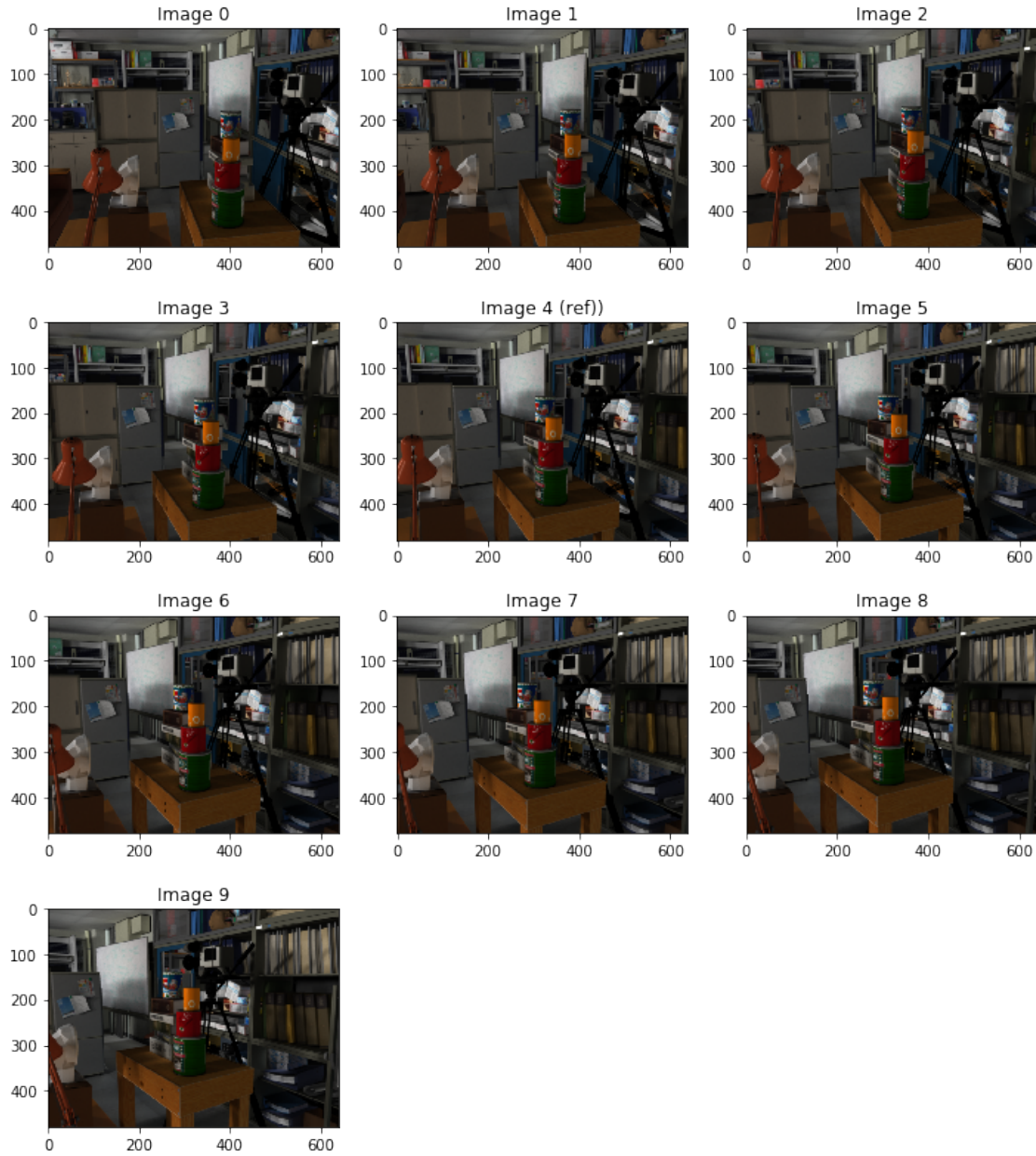
# Visualizes the source images
plt.figure(figsize=(12, 14))
num_rows = math.ceil(len(images) / 3)
plt.tight_layout()
for i in range(len(images)):
    plt.subplot(num_rows, 3, i+1)
    plt.imshow(images[i].image)
    if i == ref_id:
        plt.title('Image {} (ref)'.format(i))
    else:
        plt.title('Image {}'.format(i))
```

Loading data from data/tsukuba...

Loaded data containing 10 images.

Reference camera pose:

```
[[ 3.932424e-01  7.744632e-05  9.194348e-01 -1.443208e+00]
 [ 1.511037e-01  9.863976e-01 -6.471018e-02  1.799344e-01]
 [-9.069333e-01  1.643768e-01  3.878817e-01  1.124355e+00]]
```



The images are stored as instances of our `Image` class. You can see its implementation in `lab6.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 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 variables:

- `ps_volume`: Plane sweep volume of size (D, H, W)
- `accum_volume`: Number of valid images considered in computing the variance, with size (D, H, W)

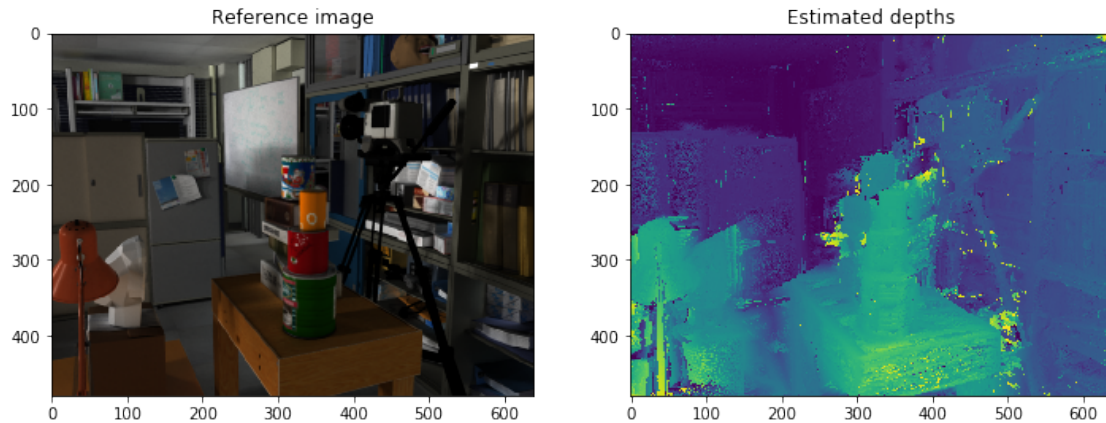
Hint: You might find the following functions useful: `get_plane_sweep_homographies()` (from Part 1), and `cv2.warpPerspective()`.

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

ps_volume, accum_count = compute_plane_sweep_volume(images, ref_pose, K,
→inv_depths,
                                                    (img_height, img_width))
```

```
[7]: # 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).

4 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 function(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.

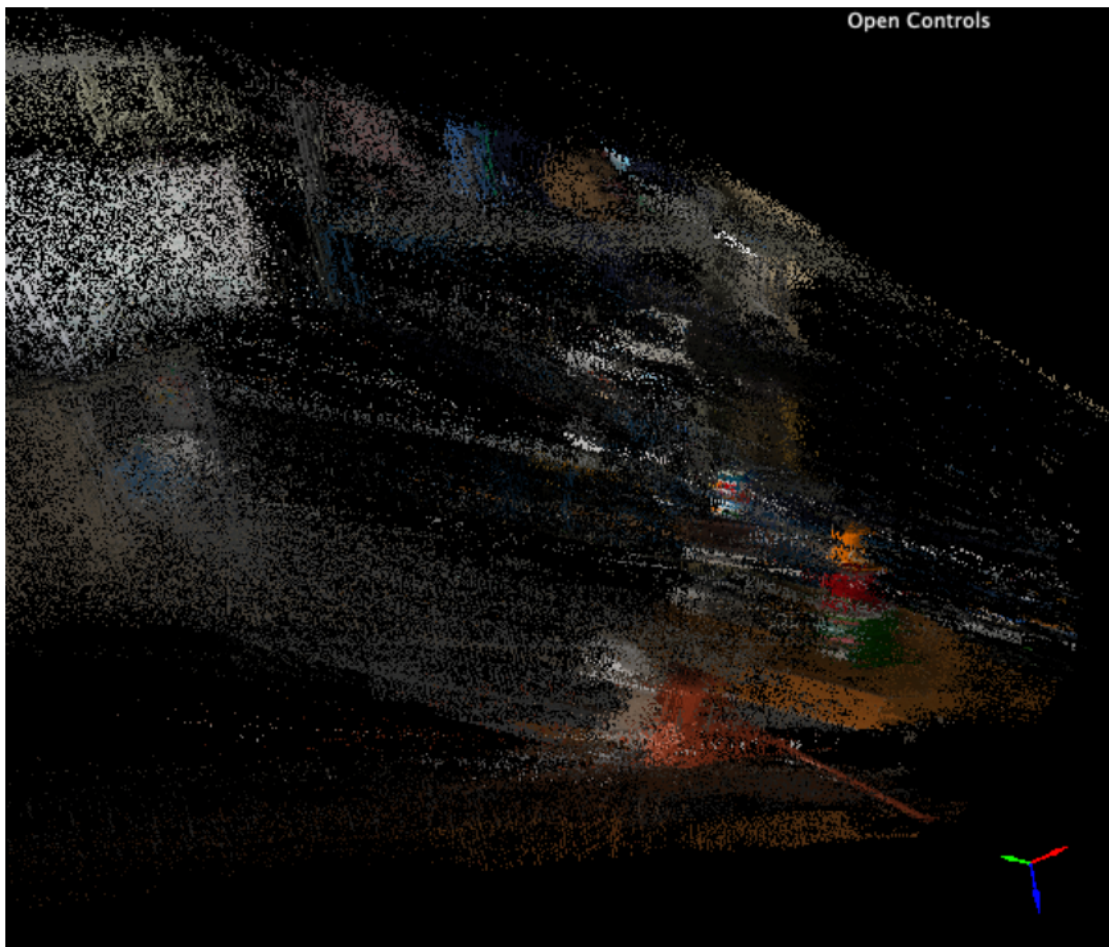
You might find the following functions useful: `np.meshgrid()`

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

Let see how our model looks like in 3D

```
[10]: 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()
```

5 Post Processing

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 function(s): `post_process()`

No specific instructions are provided for this part, so use your creativity here! Some ideas include:

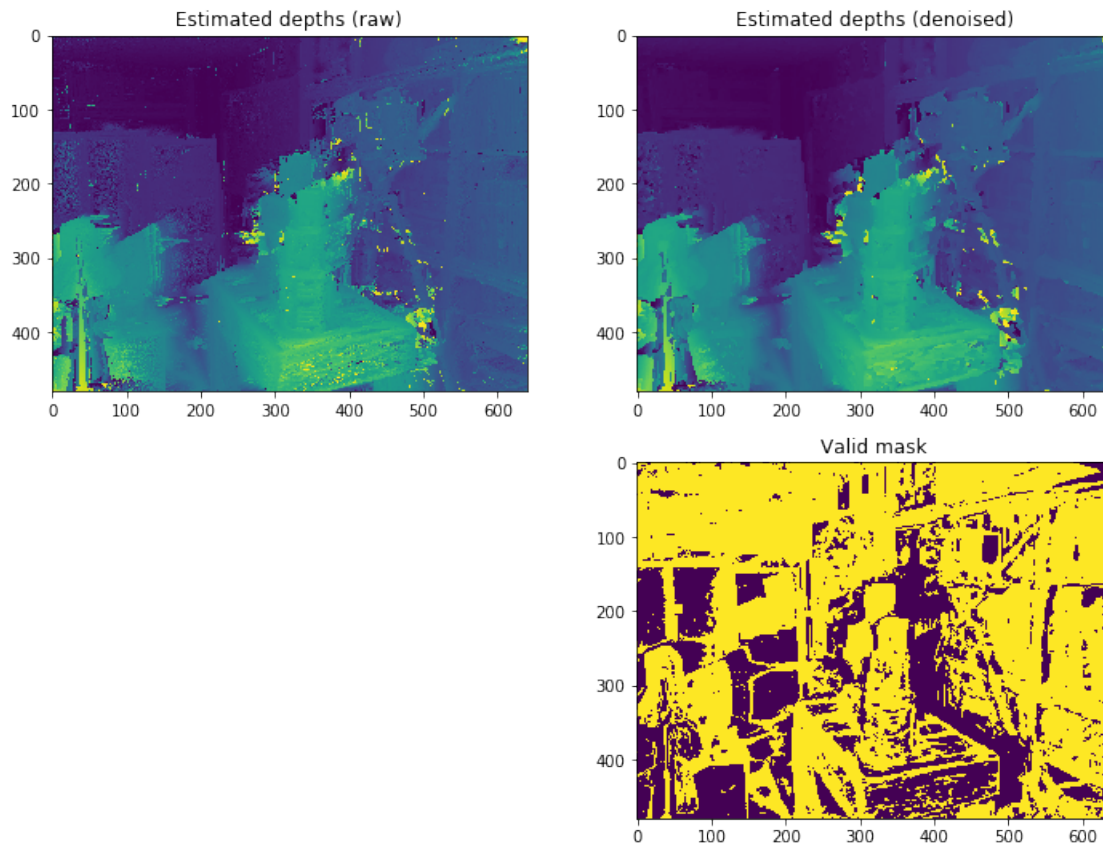
- use smoothing filter to denoise the variance estimates, e.g. see `scipy.ndimage.*filter()`.
- mask out pixels/depts with large variances.
- consider how many images are used to compute the variance, i.e. the `accum_count` variable.

A portion of 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!

```
[11]: inv_depth_img2, valid_mask = post_process(ps_volume, inv_depths, accum_count)
```



```
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');
```



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

```
[26]: 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()
```

