HW1 Review

Neil Nie, 02/11/2023

EdStem

If stuck on something, please check EdStem! There is a lot of good stuff on there. Also you can ask your own questions!

If you are not stuck, still check EdStem! Things that “instructors” say or endorsed on EdStem are like addendums to the handout. So if we clarified something on EdStem that is inconsistent with your interpretation, this could lead to trouble.

Reminders

- Due Date:
  - Due on Thursday Feb 24th at 5:00 PM EST.
  - Late submission policy is on the courseworks website.
- Office Hours:
  - Please come to my or Zeyi's or Hammad's office hours if you have questions about the homework.
  - Please attend Professor's office hours if you have questions about the lecture.
- Don't forget to review the lecture slides and review the lecture recording.

Submission Checklist

- Implementation:
  - `transforms.py`
  - `ply.ply`
  - `tsdf.py`
- Submissions:
  - Please submit on courseworks.
  - Do not submit your environment folder.
  - Your submission should be a .zip file.
  - Your hw1/supplemental folder should contain:
    - point_cloud.ply
    - mesh.ply

Clarifications

- For problems 1.4 & 1.5, you only need to show numerical equivalence.
- In the tsdf.py file original stencil code line 92, the function call is deprecated. Please replace that line with the following: `voxel_points, triangles, normals, _ = measure.marching_cubes(tsdf_volume, method="lewiner", level=0)`
- Note that Numpy is row-major. The pixel coordinate u is the horizontal axis and v is the vertical axis. Therefore when indexing the image matrix please use [v, u]
- The camera pose is camera --> world. You might need to apply an inverse.

1. Transforms

What is an SE(3)?

$$SE(3) = \left\{ \mathbf{A} \;\middle|\; \mathbf{A} = \left[\begin{array}{c|c} \mathbf{R} & \mathbf{r} \\ \hline \mathbf{0}_{1\times 3} & 1 \end{array}\right], \mathbf{R} \in R^{3\times 3}, \mathbf{r} \in R^3, \mathbf{R}^T\mathbf{R} = \mathbf{R}\mathbf{R}^T = \mathbf{I}, |\mathbf{R}| = 1 \right\}$$

https://www.seas.upenn.edu/~meam620/slides/kinematicsI.pdf

What is an SE(3) in words?

SE(3) is a name for a set of 4x4 matrices that follow some rules:

1. Check to make sure the matrix is a 4x4 matrix.
2. The top left 3x3 matrix, which we will call R, have entries that are all real numbers. Nice.
3. The first three entries of the last column, which we will call t, are all real numbers. Ok.
4. Last row is 0, 0, 0, 1. Fine.
5. R times R transpose equals R transpose times R, which also equals the 3x3 identity matrix. Cool!
6. The determinant of R equals 1, never -1!

A Note on Notation (*sigh* bad overused pun?)

It helps to think of an SE(3) as something that takes you *from* one place *to* another place, like a car or an airplane or a (fill in the blank). The notation for this is actually pretty good. You will often see an SE(3) represented like this

$$^{to}T_{from}$$

We can also invert this transform to undo our trip:

$$(^{to}T_{from})^{-1} = {}^{from}T_{to}$$

How to Solve for the SE(3) Transform

1. Translate your target frame to your origin frame, this translation is relative to your target frame. That translation vector becomes your last column.
2. Solve for the rotation matrix.
    a. List the series of rotations and their angles to align the target frame with the origin frame. For example, first rotate around z by +90 then rotate around y by -90.
    b. Plug the values into the formula and solve.
    c. Note: composition of rotations are multiplied from left to right.

How to Solve for the SE(3) Transform

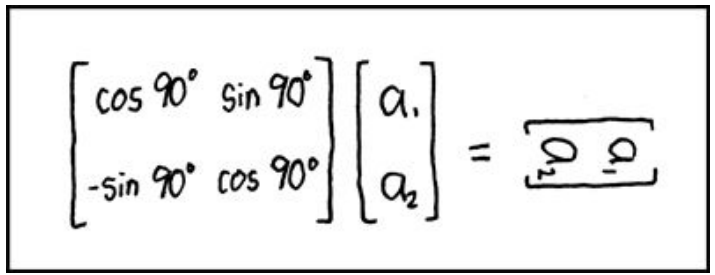
$$\mathbf{R}_x(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix}$$

$$\mathbf{R}_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix}$$

$$\mathbf{R}_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The Importance of SE(3) Transforms

SE(3)s are useful when when we are keeping track of many coordinate frames. They give us an easy way to go from a point in one coordinate frame to another coordinate frame. This is useful in many 3D computer vision and robotics applications where we are concerned with camera, robot parts, and the ways in which they are oriented with respect to each other. SE(3) are also easily composable and invertible as we will soon seen



Composition of SE(3)s

SE(3) compose nicely. Say we have a coordinate for cities A, B, and C. We might have a little trip planned where we start at A, then go to B, and then to C. We might have these transforms to represent the legs of our trip:

$$^{B}T_{A}, ^{C}T_{B}$$

If we wanted to skip going to B because it is bad, we can calculate the following to figure out how to go from space A straight to C using matrix multiplication:

$$^{C}T_{A} = ^{C}T_{B}\,^{B}T_{A}$$

Notice B kinda “cancels-out” in this notation, which makes it easier to think about.

SE(3)s and Cameras?

SE(3)s are commonly used to represent camera poses in the world coordinate system. This is also called an extrinsic. You will frequently see camera poses as world to camera (sometimes called the rig.)

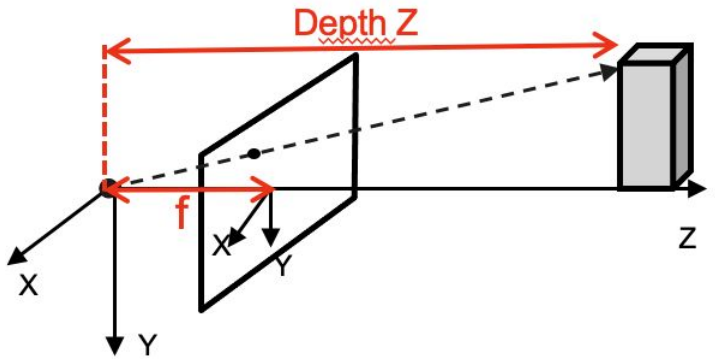
$$ {}^{\text{camera}}T_{\text{world}} $$

The world frame can just be an arbitrary coordinate frame. But there is usually a little more structure to the camera frame. For example, if we take the POV of the camera, y might represent down, x to the right, and z into the world.

How to Implement `depth_to_point_cloud`

- Note: Only output those points whose **depth > 0**.
- For all of the u, 0 <= u < image_width, and for all the v, 0 <= v < image_height
- The image center is at (v_0, u_0)
- The image is row-major. Index v first, then u.

```
X = (u - u_0) / f_u * Z
Y = (v - v_0) / f_v * Z
Z = Z
```



2. NumPy

Numpy

To do the homework, you should be using the popular python library numpy. It is common to `import numpy as np`.

Check out numpy docs or google to figure out how to do specific things. However, here are some basics and some words of caution.

Numpy

`np.array(...)` can be used to wrap python lists.

```
>>> import numpy as np
>>> np.array([1, 3, 4])
array([1, 3, 4])
```

Numpy

It is easy to do element-wise operations with `np.array(...)`

```
>>> import numpy as np
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a + 5 # add 5 to each entry
array([[6, 7],
       [8, 9]])
```

Numpy Multiplication

```
>>> a = np.array([[1, 2], [3, 4]])
>>> b = np.array([[5, 5], [5, 5]])
>>> a * b # element-wise multiplication
array([[ 5, 10],
       [15, 20]])
>>> np.matmul(a, b) # matrix multiplication
array([[15, 15],
       [35, 35]])
>>> np.dot(a, b) # matrix multiplication
array([[15, 15],
       [35, 35]])
>>> a @ b # matrix multiplication
array([[15, 15],
       [35, 35]])
```

Numpy Tips

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a.shape # get the dims of the array.
(2, 2)
>>> a[1, 1] = 5 # index and set element
>>> a
array([[1, 2],
       [3, 5]])
>>> a.reshape(1, -1) # flatten a 2d array
array([[1, 2, 3, 5]])
>>> a[:,1] # get all rows for the 1st col
array([2, 5])
>>> c = np.logical_and(a > 2, a < 5)# filter a based on some conditions (and-ed)
>>> c
array([[False, False],
       [ True, False]])
>>> a[c] # index into a based on truth values from c
array([3])
```

3. PLY

Interpreting a PLY file

Number of points

Data fields for each point

```
ply
format ascii 1.0
element vertex 3 # number of points
property float x # first entry of a point.
property float y
property float z
property float nx # first normal component of the point.
property float ny
property float nz
property uchar red # red component of the point color.
property uchar green
property uchar blue
end_header
0.0 0.0 1.0 1.0 0.0 0.0 0 0 155 # x y z nx ny nz red green blue
0.0 1.0 0.0 1.0 0.0 0.0 0 0 155
1.0 0.0 0.0 1.0 0.0 0.0 0 0 155
```

Points with the xyz, nx ny nz, and rgb values

Interpreting a PLY file

Number of points

Data fields for each point

```
ply
format ascii 1.0
element vertex 3 # number of points.
property float x # first entry of a point.
property float y
property float z
property float nx # first normal component of the point.
property float ny
property float nz
property uchar red # red component of the point color.
property uchar green
property uchar blue
element face 1 # number of faces.
property list uchar int vertex_index
end_header
0.0 0.0 1.0 1.0 0.0 0.0 0 0 155 # x y z nx ny nz red green blue
0.0 1.0 0.0 1.0 0.0 0.0 0 0 155
1.0 0.0 0.0 1.0 0.0 0.0 0 0 155
3 2 1 0 # (number of vertices in the face) (point index 1) ...
```

Number of faces

One face

The PLY Class

- Constructor takes in `ply_path=None, triangles=None, points=None, normals=None, colors=None`
- Case 1: ply_path provided and valid.
  - Read the content from ply_path, and update the instance variables.
- Case 2: ply_path not provided, but other parameters are provided.
  - Initialize the instance of ply using the points, colors… parameters.
- Case 3: ply_path and other parameters are provided.
  - Ignore the other parameters, read from disk.

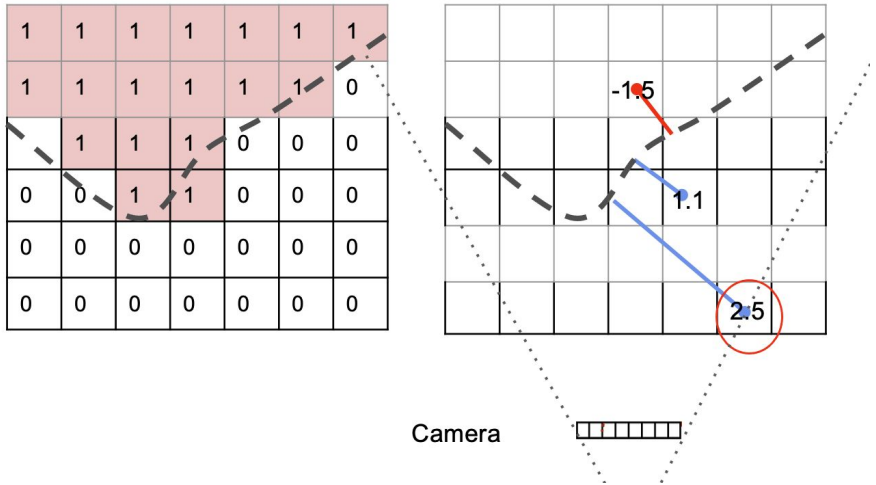
The PLY Class

- The write() method will write mesh, point cloud, or oriented point cloud to ply file, given the ply_path parameter.
- The read() file reads the content of the ply from disk and updates the object's instance variables.
- Tip: use Python built in file I/O: `with open(ply_path, 'w') as f:`

2. TSDF

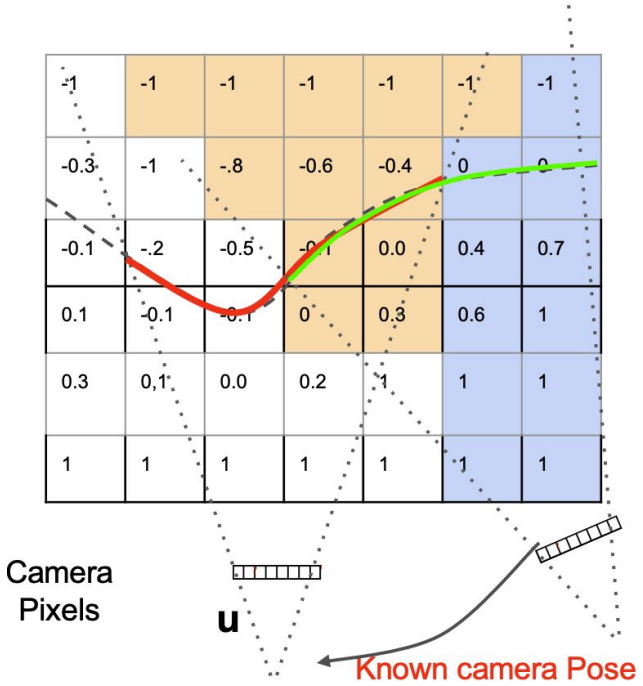
TSDF

- Store the distance from each voxel to the closest surface d
- Sign indicated occluded/free space
- Truncate the distance value with `_truncation_margin = 2.5, d(d>_truncation_margin) = _truncation_margin`
- Normalize: `d = d / d_max`

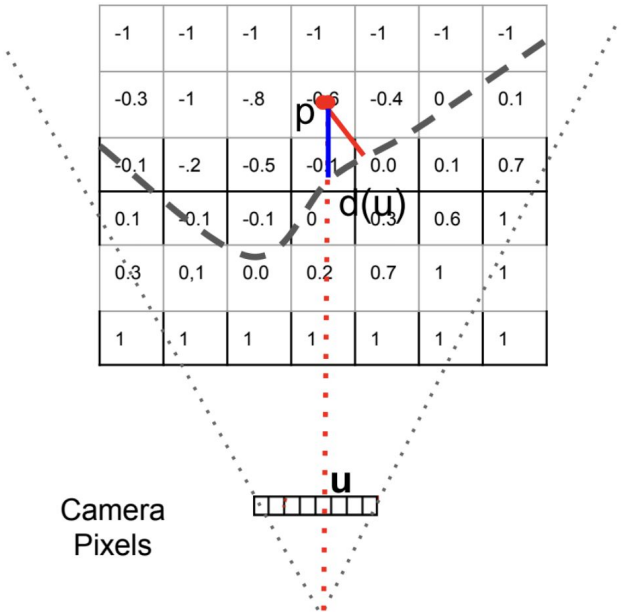


Multi-view TSDF Overview

- Update the voxel values for all the voxels observed by the new camera
- For the voxels already observed in the first frames, the new TSDF value will be a weighted sum of new and old value



Multi-view TSDF Algorithm Overview

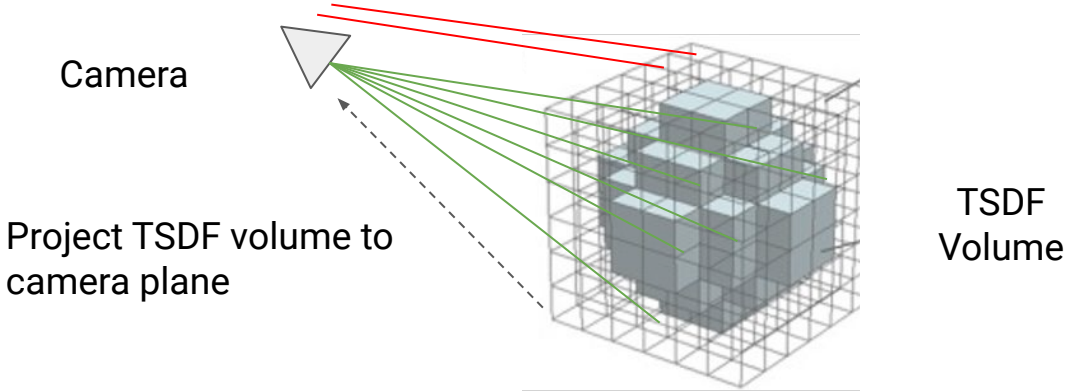


1. Convert voxel coordinate to world coordinates.
2. Convert world coordinates to camera coordinates (u, v) using the SE(3).
3. Determine which voxel to update by filtering out invalid voxels.
4. Set the signed projective distance d = d(u, v) - p_z
5. Normalize and truncation
6. Update tsdf and colors.

(data parallel across voxels)

Getting the Valid Voxels Indices

1. Eliminate pixels not in the image bounds, or behind the image plane.
2. Filter out zero depth values.
3. This will return a list of indices in the TSDF volume that we can update with the most recent observation: depth and color image.



Integrate TSDF

- `voxel_to_world(...) # easiest, knock this out first`
- `get_valid_points(...)`
- `get_new_tsdf_and_weights(...)`
- `get_new_colors_with_weights(...)`
- `integrate(...) # hardest, but you can do it!`

voxel_to_world(...)

```
For each voxel in voxel coordinates:
    For x, y, z dims of voxel:
        Convert dim to corresponding dim in world coordinates and store
Return world coordinates
```

get_new_tsdf_and_weights(...)

```
for index i in the old tsdf volume that we need to update:
    new_weight[i] = old weight[i] + observation weight
    new_tsdf[i] = (old weight * old tsdf value + obs_weight * margin_distance)
    / new_weight
return new_weights, new_tsdf
```

Parameter name	Type / Shape	Description
`tsdf_old`	`(numpy.array [n, ])`	n is equal to the number of voxels to be integrated at this timestamp. Old tsdf values that need to be updated based on the current observation.
`margin_distance`	`(numpy.array [n, ])`	The tsdf values of the current observation. It should be of type numpy.array [n,], n is the number of valid voxels.
`w_old`	`(numpy.array [n, ])`	Old TSDF weight values
`observation_weight`	`float`	Weight to give each new observation.

get_new_colors_with_weights(...)

Compute the new R, G, and B value.

ex: the new R = sum of the old color value weighted by the old weight, and the new color weighted by observation weight, and normalized by the new weight.

return [r_new, g_new, b_new]

Parameter name	Type / Shape	Description
color_old	(numpy.array [n, 3])	Old colors from self._color_volume in RGB.
color_new	(numpy.array [n, 3])	Newly observed colors from the image in RGB
w_old	(numpy.array [n,])	Old weights from the self._tsdf_volume
w_new	(numpy.array [n,])	New weights from calling get_new_tsdf_and_weights observation_weight (float, optional): The weight to assign for the current observation. Defaults to 1.

`get_valid_points(...)`

Compute a boolean array for indexing the voxel volume and other variables. Every time the method **integrate** is called, not every voxel in the volume will be updated. This method returns a boolean matrix called valid_points (n,), where n = # of voxels. Index i of valid_points is true if this voxel will update, false if the voxel will not update.

Parameter name	Type / Shape	Description
`depth_image`	`(numpy.array [h, w])`	A z depth image
`voxel_u`	`(numpy.array [n, ])`	Voxel coordinate projected into camera coordinate, axis is u
`voxel_v`	`(numpy.array [n, ])`	Voxel coordinate projected into camera coordinate, axis is v
`voxel_z`	`(numpy.array [n, ])`	Voxel coordinate projected into world coordinate axis z
`Returns: valid_points`	`(numpy.array [n, ])`	A boolean matrix that will be used to index into the voxel grid. Note the dimension of this variable. A true value represent this voxel is valid, a false value represent this voxel is not valid.

get_valid_points(...)

```
# TODO 1:
#  Eliminate pixels not in the image bounds or that are behind the image plane
# TODO 2:
#  Filter out zero depth values
```

`integrate(...)`

Integrate an RGB-D observation into the TSDF volume, by updating the weight volume, tsdf volume, and color volume. You need to implement and use several helper methods.

Parameter name	Type / Shape	Description
`color_image`	`(numpy.array [h, w, 3])`	An rgb image.
`depth_image`	`(numpy.array [h, w])`	A z depth image.
`camera_intrinsics`	`(numpy.array [3, 3])`	given as [[fu, 0, u0], [0, fv, v0], [0, 0, 1]]
`camera_pose`	`(numpy.array [4, 4])`	SE3 transform representing pose (camera to world)
`observation_weight`	`(float, optional)`	The weight to assign for the current observation. Defaults to 1.

integrate(...)

```
# TODO: 1. Project the voxel grid coordinates to the world space by calling
`voxel_to_world`. Then, transform the points in world coordinate to camera
coordinates, which are in (u, v). You might want to save the voxel z
coordinate for later use.

# TODO: 2. Get all of the valid points in the voxel grid by implementing the
helper get_valid_points. Be sure to pass in the correct parameters.

# TODO: 3. With the valid_points array as your indexing array, index into the
self._voxel_coords variable to get the valid voxel x, y, and z.
```

integrate(...)

```
# TODO: 4. With the valid_points as your indexing array, get the valid pixels.
Use those valid pixels to index into the depth_image, and find the valid
margin distance.

# TODO: 5. Compute the new weight volume and tsdf volume by calling
`get_new_tsdf_and_weights`. Then update the weight volume and tsdf volume.

# TODO: 6. Compute the new colors for only the valid voxels by using
get_new_colors_with_weights, and update the current color volume. The
color_old and color_new parameters can be obtained by indexing the valid
voxels in the color volume and indexing the valid pixels in the rgb image.
```

Thank you, Good Luck, Q&A