Homework 1

Out: Thursday, September 10

Due: Friday, October 2 @ 5:00pm EST

This homework covers the basics of rigid $SE(3)$ transforms, point cloud and mesh file I/O, truncated signed distance functions (tsdfs), and convolution. The homework should help you practice the material from the first five lectures!

This homework is one of four assignments. It is worth 100 points with an extra credit problem worth an additional 10 points.

For problems 1-4, compile your answers in `hw1.pdf` using the LaTeX template provided on the course website. We recommend Overleaf for this purpose. Add `hw1.pdf` to the directory `hw1/supplemental`.

For problem 5, you will directly be editing python files provided by the TAs. If there are any know issues please document them in a problem 5 section of `hw1.pdf`. Otherwise you can keep this section blank.

For problem 6 (extra credit), you will directly be editing python files provided by the TAs. Add any requested written answers in `hw1.pdf`.

When you are done, zip the `hw1` directory and upload to coursework. We will review your answers, provide constructive feedback, and run your code against our grading scripts!

Problem 1 *(6 points)*

Given the following matrices:

$$A = \begin{bmatrix} 1 & 0 & 0 & 20 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 30 \\ 0 & 0 & 0 & 0 \end{bmatrix}, B = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & -1 & 3 \\ 0 & 1 & 0 & 5 \\ 0 & 0 & 0 & 1 \end{bmatrix}, C = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & -1 & 3 \\ 0 & -1 & 0 & 5 \\ 0 & 0 & 0 & 1 \end{bmatrix}, D = \begin{bmatrix} -1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 3 \\ 0 & 1 & 0 & 5 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

1. (3 *point*) Determine if each matrix belongs to the $SE(3)$ group of valid homogeneous transforms. Justify your answers and show any relevant calculations.
2. (3 *point*) For any transform $T \in SE(3) \cap \{A, B, C, D\}$ compute the inverse transform T^{-1}. Verify $T^{-1}T = I$, $TT^{-1} = I$. Show any relevant calculations.

Problem 2 *(6 points)*

Given ${}^0T_1, {}^1T_2, {}^0T_2 \in SE(3)$, where ${}^iT_j$ gives the transform from frame j to frame i, write expressions for the following:

1. (3 *point*) ${}^0T_2$ in terms of ${}^1T_2$ and ${}^0T_1$.
2. (3 *point*) ${}^1T_2$ in terms of ${}^0T_1$ and ${}^0T_2$.

Problem 3 *(9 points)*

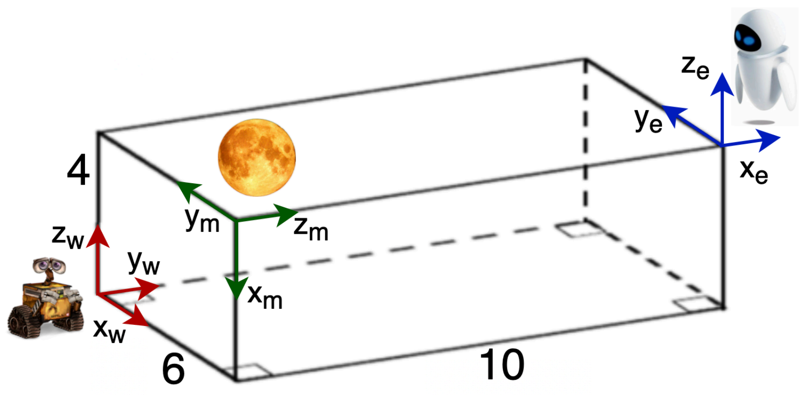
For this problem, no need to show your work, but give a list of the steps or expressions you evaluated to get the answer.
Given the following $SE(3)$ poses:

$$ {}^0T_1 = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 10 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \; {}^1T_2 = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -3 \\ 0 & 1 & 0 & -10 \\ 0 & 0 & 0 & 1 \end{bmatrix}. $$

1. (3 *point*) The coordinate of a 3D point p, is given as ${}^2p = [2, 4, 6]^T$ in the coordinate space of frame 2. Compute ${}^1p$, the coordinate of p in frame 1.
2. (3 *point*) Compute the transform ${}^0T_2$, the transform that takes a point from coordinate space of frame 2 to the coordinate space of frame 0.
3. (3 *point*) The coordinate of a 3D point q, is given as ${}^0q = [2, 4, 6]^T$ in the coordinate space of frame 0. Compute ${}^2q$, the coordinate of q in frame 2.

Problem 4 *(9 points)*

Here we have three coordinate frames. Eve's reference frame e, Wall-e's reference frame w, and the moon's reference frame m.



1. (3 *point*) Write the following poses ${}^mT_w$, ${}^eT_m$, ${}^eT_w$ as $SE(3)$s. Pick one of these poses and write a sentence about what this pose represents.
2. (3 *point*) Verify that ${}^eT_m\ {}^mT_w ={}^e T_w$. Show your work.
3. (3 *point*) Verify that ${}^eT_w = ({}^wT_e)^{-1}$. Show your work.

Problem 5 *(70 points)*

In this problem you will implement light-weight transforms functions, 3D I/O functions, and a tsdf voxelization module in python.

Make sure your default python interpreter is python3. If not use the command `python3` and `pip3` instead of `python` and `pip`. Dependencies for hw1 can be installed using the following commands where we first create a python virtual environment:

```
python -m venv hw1
```

To activate your environment run:

```
source hw1/bin/activate // Linux or OSX
hw1\Scripts\activate.bat // Windows
```

To install dependencies, once you have activated your environment run:

```
pip install -r requirements.txt
```

To exit the environment, when you are not working on this project, run:

```
deactivate // Linux of OSX
hw1\Scripts\deactivate.bat // Windows
```

You can read more about python virtual environments here.

1. (15 *point*) In the first part of this problem, you will implement parts of `transforms.py`. This file contains some generic transforms definitions related to problems 1-4. In `transforms.py` you should implement the following:

   ```
   transform_is_valid(...)
   transform_concat(...)
   transform_point3s(...)
   transform_inverse(...)
   camera_to_image(...)
   depth_to_point_cloud(...)
   ```

   Detailed descriptions of function inputs and outputs are given in `transforms.py`. We have also provided `transforms_test.py` to provide basic unit tests for your transforms functions. Before proceeding, it is a good idea to make sure all of these tests pass. Subsequent parts of the project can depend on some or all of these functions depending on your implementation. To run unit tests execute the command:

   ```
   python transforms_test.py
   ```

   Note: these tests are meant to help you, but are not necessarily exhaustive. You are encouraged to do further testing if you feel certain cases are not properly tested.

2. (15 *point*) Here you will write a class to read and write polygon file formats or `.ply` files. You can read more about the `.ply` format here. A `.ply` file supports both text and binary encodings. For the purposes of this problem, we are concerned with writing points, normals, colors, and triangle faces in text. Without faces, we call our output an *oriented point cloud*. With faces we call our output a *mesh*. We have provided a sample *oriented point cloud* called `point_sample.ply`, here annotated with python style comments for clarity:

```
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
```

We also provide a sample *mesh* file called `triangle_sample.ply`, here annotated for clarity:

```
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) ...
```

In `ply.py` implement the following in the `Ply` class:

```
__init__(...)
```

```
write(...)
read(...)
```

After implementing these functions you should be able to read and write back `point_sample.ply` and `triangle_sample.ply` found in the `data` folder. Consider writing a python function that checks this to be confident in your code. Also consider downloading a viewer like Meshlab to make sure that your *oriented point clouds* and *meshes* look as expected. Your implementation will be called by our stencil code in later parts of this project.

3. (40 *point*) You have now developed the pieces to implement a tsdf fusion loop! As covered in lecture, tsdfs are a way to integrate RGB-D images from different camera poses into a cohesive, implicit 3D reconstruction. We say the tsdf is implicit, because it does not directly contain 3D point locations or faces. Rather, the tsdf encodes the distance from voxels to the nearest surface, up to a truncation threshold. If a voxel is "behind" a surface, the tsdf value for this voxel will be negative. If the voxel is exactly on the surface, the value should be zero. If the voxel is in front of the surface, the value will be positive. Using this implicit representation, we can recover an explicit mesh by looking for zero crossings, where tsdf value switches between negative and positive.

   In `tsdf.py` we define the `TSDFVolume` class. As part of the stencil code, we define many instance variables to be used.

```
self._voxel_size : float side length in meters of each 3D voxel
   ↪cube.

self._truncation_margin : float tsdf truncation margin, the max
   ↪allowable distance away from a surface in meters.

self._volume_bounds : Numpy array [3, 2] of float32s, where rows
   ↪index [x, y, z] and cols index [min_bound, max_bound]. Note:
   ↪these bounds are rounded in such a way that dimension length
   ↪divided by self._voxel_size would be a whole number. Units
   ↪are in meters.

self._volume_origin : Origin of the voxel grid in world coordinate
   ↪(not voxel coordinates). Units are in meters.

self._tsdf_volume : Numpy array of float32s representing tsdf
   ↪volume where each voxel represents a volume self._voxel_size
   ↪^3. Shape of this volume is determined by (max_bound -
   ↪min_bound)/ self._voxel_size. Each entry contains the
```

```
    ↪distance to the nearest surface in meters, truncated by self.
    ↪_truncation_margin.

self._weight_volume : Numpy array of float32s with same shape as
    ↪self._tsdf_volume. Each entry represents the observation
    ↪weight assigned to each voxel. For example, if observation
    ↪weight is globally set to 1.0, this volume keeps track of the
    ↪ number of times a voxel has been seen for purposes of taking
    ↪ weighted averages.

self._color_volume : Numpy array of float32s with shape [self.
    ↪_tsdf_volume.shape, 3] in range [0.0, 255.0]. So each entry
    ↪in the volume contains the average r, g, b color.

self._voxel_coords : Numpy array [number of voxels, 3] of uint8s.
    ↪Each row indexes a different voxel [[0, 0, 0], [0, 0, 1],
    ↪..., [0, 1, 0], [0, 1, 1], ..., [1, 0, 0], [1, 0, 1], ..., [x
    ↪-1, y-1, z-2], [x-1, y-1, z-1]]. When a new observation is
    ↪made, we need to determine which of these voxel coordinates
    ↪is "valid" so we can decide what voxels to update.
```

Implement the following functions, using these variables and your functions in `transforms.py`. Update variables as needed to integrate new rgb-d observations into the tsdf volume.

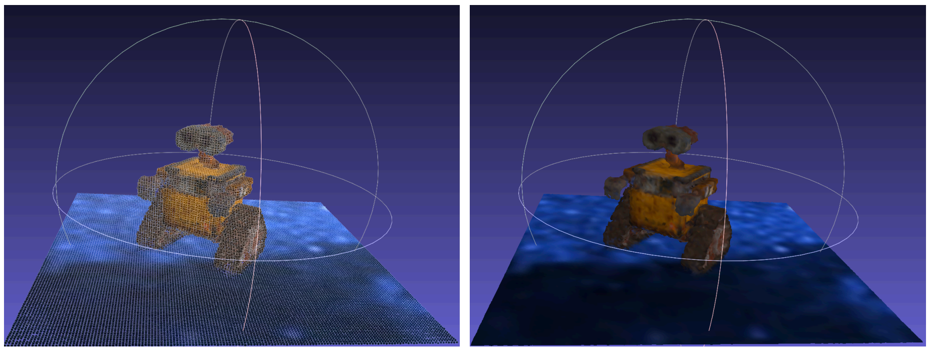
```
voxel_to_world(...)
integrate_volume_helper(...)
integrate(...)
```

You can also take a look at functions `get_volume(...)`, which is a getter to retrieve volumetric data from your fusion, and `get_mesh(...)`, which reconstructs a *mesh* from a tsdf using the marching cubes algorithm.

To test your implementation, you can run the following, which should conduct fusion on 10 rgb-d frames in the `data` directory with some predefined volume settings:

```
python tsdf_run.py
```

Using your `ply.py` and `tsdf.py` implementations, this script should create two files—`point_cloud.ply` and `mesh.ply`—in `hw1/supplemental`, which should show a Wall-E robot. Both the point cloud and mesh should have points spread out at 1 cm resolution. Here are some example outputs:



Problem 6 (Extra Credit, *10 points*)

1. (3 *point*) Filtering has been an invaluable tool for classical computer vision problems like edge detection. With the recent success of Convolutional Neural Networks (CNNs), there has been an emphasis on optimizing or learning weights in convolutional filters to extract useful representations for downstream tasks: image classification, object detection, semantic segmentation, etc. While learning filters is super useful (and very cool), you can gain useful intuition about what convolutional filters are doing by hand-crafting your own!

   In this problem you will be implementing the `apply_filer(...)` function in the `GrayscaleConvManager` class in `grayscale_conv_manager.py`. Note: you are only allowed to use numpy and stencil functions. If you use external packages like `scipy` or `cv2` to implement the algorithmic components of convolution, you will not be able to get extra credit. As part of the stencil code, we provide the following instance variables:

   ```
   self._image : image read in through a grayscale .png file. Numpy
       ↪array [height, width].
   self._kernel : Numpy array [k, k] where k is the kernel height and
       ↪height in pixels and k is odd.
   ```

   The goal is to convolve an input filter with an input image. Note: here when we say convolution, we mean convolution as referred to by the computer vision community, not by signal processing community. Hence do not worry about transposing the convolution filters. Implement the following function using the instance variables:

   ```
   apply_filter(...)
   ```

2. (1 *point*) Open `grayscale_conv_manager_run.py` and look at the dictionary `kernels` on `line 9`. Add 2 [3 x 3] filters, one to get horizontally oriented and one to get vertically oriented sobel edges. Write a few sentence explaining what

the sobel filters are doing and the intuition behind why are doing this. Run the following to test:

```
python grayscale_conv_manager_run.py
```

This will create images `lion_<filter name>.png`, `butler_<filter name>.png`, `walle_<filter name>.png` in `hw1/supplemental` for each `<filter name>`.

3. (1 *point*) In `grayscale_conv_manager_run.py`. Add 2 more filters of your own design—get creative! Write a few sentence explaining what your filters are doing and why they are doing this. Run the command line for the sub-problem above to iterate and test.

4. (5 *point*) So far we have considered convolution only for 1-channel grayscale images. However, the same principles hold when we are considering multi-channel images or feature maps. In general we want the depth of the convolutional filter to match the number of channels of our input volume. For example, if we are dealing with a 224 x 224 rgb-d image, we might want to apply a filter of size 3 x 3 x 4 across the image to extract features from the input volume.

   Implement the blank file `conv_manager.py` to take rgb images, depth images, or aligned rgb and depth images and apply a convolutional kernel to filter the image. Feel free to take inspiration from `grayscale_conv_manager.py` or implement your own design!

   Next implement `conv_manager_run.py` that processes `data/frame-x.color.png` and `data/frame-x.depth.png` with two hard coded kernels. Feel free to take inspiration from `grayscale_conv_manager.py` and to use `image.py` to help with file I/O. Write a few sentences explaining what your filters are doing and why they are doing this.

   Make sure that your script can be run using the following command line:

   ```
   python conv_manager_run.py
   ```

   and that the filtered result visualization is written to `hw1/supplemental`. Note: this is extremely important as it will allow us to run your code with our grading scripts. If we cannot run your code, you will not be able to get credit for this sub-problem.

   Some things to consider. Should each channel of your kernel be the same? There are perhaps differences in the ways that rgb and depth data "look" and hence in the ways they should be filtered.