# Homework 1

*Out: Wednesday, Jan 26*

*Due: Thursday, Feb 17 @ 5:00pm EST*

## Instructions

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

This homework is one of four assignments. It is worth 100 points.

For problem 1, compile your answers in PDF and title it `hw1.pdf`. You are encouraged to use the LaTeX template provided in the zip file. We recommend Overleaf for this purpose. Once you are done, add `hw1.pdf` to the directory `hw1/supplemental`.

For problem 2, you will directly be editing python files provided by the TAs. If there are any known issues please document them in a problem 2 section of `hw1.pdf`. Otherwise you can keep this section blank.

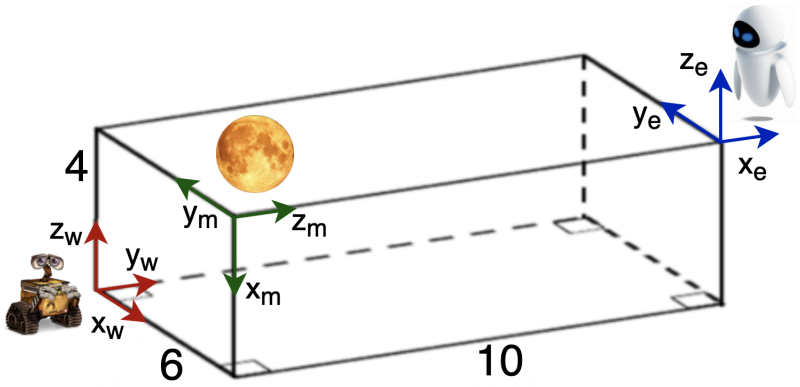
When you are done, zip the `hw1` directory and upload it to coursework. We will review your answers, provide constructive feedback, and run your code against our grading scripts!

up close to check

$R^{-1} = R^T$

# Problem 1 *(20 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. (4 *point*) What is the moon's position in Wall-e's reference frame? What is Eve's position in Wall-e's coordinate frame?

2. (4 *point*) Write the following poses ${}^{m}T_{w}$, ${}^{e}T_{m}$, ${}^{e}T_{w}$ as $SE(3)$s.

3. (2 *points*) Write a few sentences about what this transform ${}^{e}T_{w}$ represents in terms of translation and rotation.

4. (3 *point*) Show that ${}^{e}T_{m}\ {}^{m}T_{w} = {}^{e}\ T_{w}$.

5. (3$z$ *point*) Show that ${}^{e}T_{w} = ({}^{w}T_{e})^{-1}$.

6. (4 *point*) The moon has moved in its positive $z$ direction by 2 units and its positive $x$ direction by 2 units, show that ${}^{e}T_{w}$ hasn't changed from part 2. Briefly describe why ${}^{e}T_{w}$ did not change.

# Problem 2 *(80 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.

## Part 1 (15 *point*)

In the first part of this problem, you will implement parts of `transforms.py`. This file contains some generic transforms definitions. In `transforms.py` you should implement the following:

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

## Part 2 (20 *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.

## Part 3 (55 *point*)

You have now developed the pieces to implement a tsdf fusion loop! As discussed 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.
```

The main tsdf fusion loop is the `integrate(...)` method. The goal is to implement implement the following functions, using the instance variables described above and the functions you implemented in `transforms.py`. Update variables as needed to integrate new rgb-d observations into the tsdf volume. You can find the description for each of the methods below. The detailed documentation of each of these methods can be found in `tsdf.py`.

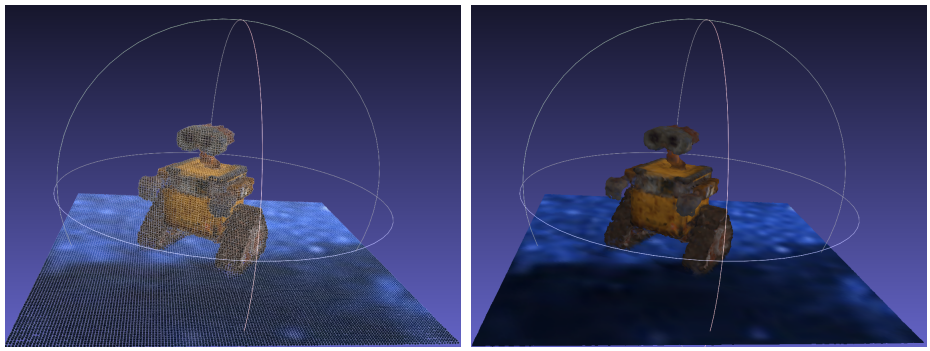
```
voxel_to_world(...): Convert from voxel coordinates to world
    ↪coordinates (in effect scaling voxel_coords by voxel_size).

get_new_tsdf_and_weights(...): Based on the margin distance, the
    ↪current tsdf, the current weights, and the observation weight,
    ↪compute the new tsdf values and the weight values.

get_new_colors_with_weights(...) Compute the new RGB values for the
    ↪color volume given the current values in the color volume, the
    ↪RGB image pixels, and the old and new weights.

get_valid_points(...): Compute a boolean array for indexing the voxel
    ↪ volume and other variables. Note that 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 with dimension (n, ), where n = # of voxels. Index
    ↪i of valid_points will be true if this voxel will be updated,
    ↪false if the voxel needs not to be updated.

integrate(...) The main tsdf fusion method where you will use all of
    ↪the helper methods above.
```

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:



When visualizing the mesh, open the file with MeshLab and select `Face` as the shading option. When visualizing the point cloud, open the file with Meshlab and select `Dot Decorator` and increase the point size. These tips will improve the visualization.