Homework 2

1 Volume Rendering [6pt]

Problem 1. In this exercise, you will recap what you have learned in Lecture 6, and implement a simple volume rendering method. You will then utilize this method to render multi-view images from a pre-trained NeRF.

We provide a codebase in `Problem1` folder and what you need to do is to fill in some core codes of volume rendering.

Setup. First install Miniconda or Anaconda. Then you can set up the codebase with the following commands:

```
conda env create -f environment.yaml
conda activate hw2_nerf
```

Code structure. There are five main components of our codebase:

- The camera: `pytorch3d.CameraBase`
- The ray data structure: `RayBundle` in `ray_utils.py`
- The scene: `SDFVolume` and `NeuralRadianceField` in `inplicit.py`
- The sampling routine: `StratifiedSampler` in `sampler.py`
- The renderer: `VolumeRenderer` in `renderer.py`

`StratifiedSampler` provides a simple ray marching method that samples multiple points along a ray traveling through the scene. The sampler and the renderer jointly make up a rendering pipeline. Like traditional graphics pipelines, this rendering procedure is independent of the scene and camera.

The scene, sampler and renderer are all packaged together in the `Model` class in `main.py`, where the forward method executes a rendering process with a scene and a sampling strategy as input.

To perform volume rendering, you need to implement the following procedures:

- **Ray sampling from cameras.** You will complement some functions in `ray_utils.py` to generate rays in the world coordinate system from a particular camera.

- **Point sampling along rays.** You will complement the StratifiedSampler class to generate sample points along each ray.
- **Rendering.** You will complement the VolumeRendering class to evaluate a volume function at each sample point along a ray, and then aggregate the evaluations to form a rendering result.

1. [Programming Assignment] Ray sampling. Please look at the render_images function in main.py. The function enumerates each predefined camera view and renders an RGB image from the camera. The first step for the rendering is to acquire all the pixels from an image and generate corresponding camera rays in the world coordinate system:

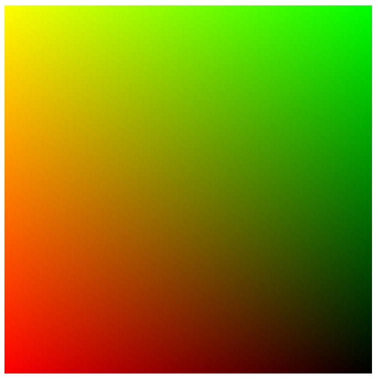
```
xy_grid = get_pixels_from_image(image_size, camera) # TODO (1):
    implement in ray_utils.py
ray_bundle = get_rays_from_pixels(xy_grid, image_size, camera) #
    TODO (1): implement in ray_utils.py
```

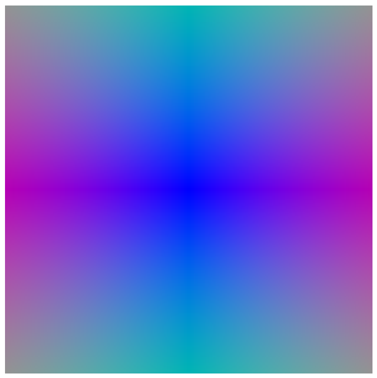
   You need to implement get_pixels_from_image and get_rays_from_pixels in ray_utils.py. The get_pixels_from_image function generates pixel coordinates in the range $[-1, 1]$. The get_rays_from_pixels function generates rays from pixels and transforms the rays from the camera space to the world space.

   You can run the code for this problem by the following command:

```
python main.py --config-name=box
```

   After you have implemented these functions, please verify that your output matches the TA's output by visualizing xy_grid and ray_bundle in render_images. The visualizations of grid and ray should look like ta_images/grid_vis.png and ta_images/ray_vis.png:


(a) Grid


(b) Ray

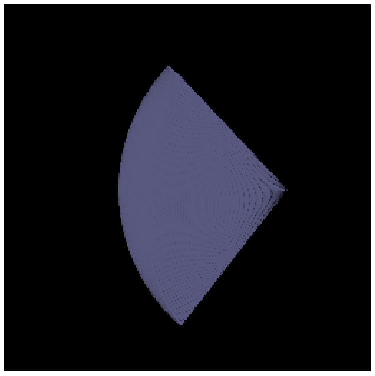
2. [Programming Assignment] Point sampling. The second step for the rendering is to sample multiple 3D points along a ray with a uniform sampling strategy. You need to implement the forward method of `StratifiedSampler` in `sampler.py` with the following routine:

   (a) Uniformly generate a set of distances between [near, far].

   (b) Use these distances to compute point offsets from ray origins `RayBundle.origins` along ray directions `RayBundle.directions`.

   (c) Store the sampled distances and points in `RayBundle.sample_points` and `RayBundle.sample_lengths`.

   After you have finished this method, you can visualize the result by first filling out the relevant codes in `render_images` and then running the command in Problem 1-1. The visualization of sample points should look like `ta_images/point_vis.png`:



3. Theory of transmittance calculation. Here we come to the transmittance calculation, which is the core part of volume rendering. Considering a ray going through a non-homogeneous medium shown in Figure 1, please compute the transmittance from x_4 to x_1 and write down your result in your report.

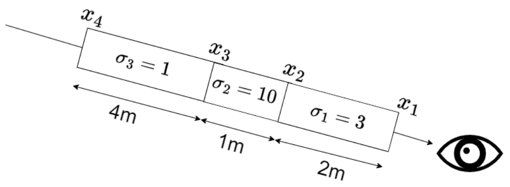


Figure 1: An example of a non-homogeneous medium.

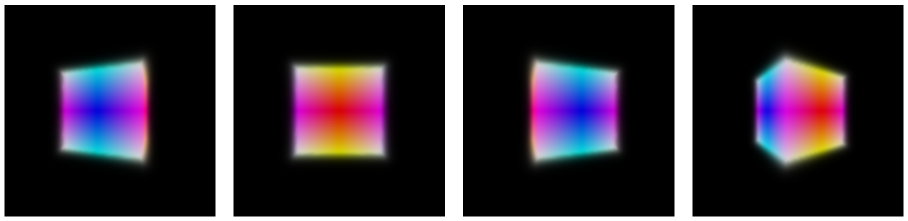
4. [Programming Assignment] Rendering. Let us come back to the implementation of volume rendering. The final step for the rendering is to integrate emissions along a ray to form a color observation at a pixel. You need to complement the forward method of `VolumeRenderer` in `renderer.py`. Two functions `_compute_weights` and `_aggregate` are used in the forward method. The `_compute_weights` function computes the weight $w_i = T(x_0, x_i)(1 - e^{-\sigma_i \Delta t_i})$ for each sample point, where x_0 is the ray origin, $x_{i \geq 1}$ is the sample point, σ is density, Δt is the length of current ray segment, and $T(x_0, x_i) = T(x_0, x_{i-1})e^{-\sigma_{i-1}\Delta t_{i-1}}$ is the transmittance. Note that $T(x_0, x_1) = 1$. The `_aggregate` function aggregates emissions by a weighted sum:

$$L(x, \omega) = \sum_{i=1}^{n} w_i L_e(x_i, \omega), \tag{1}$$

where ω is the ray direction, L_e is the emission, and L is the final rendered color.

You will also render a depth map in addition to color from a volume.

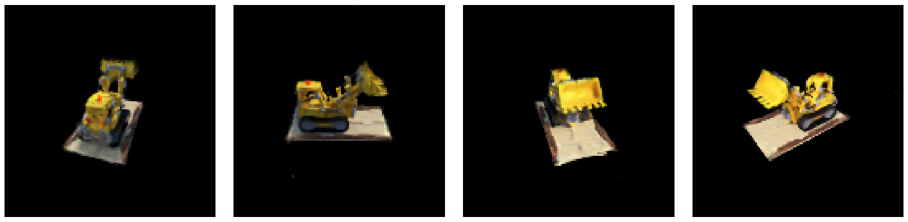
After you have implemented the method, please finish the `render_images` function and run the command in Problem 1-1. Your rendering result will be written to `images/render_cube.gif`:



Up to now, you have finished this problem. In addition to the previously rendered cube that is represented by a very simple signed distance field, TA also provides a pre-trained NeRF of a Lego model. Run the following command:

```
python main.py --config-name=nerf_lego_render
```

Your rendering result will be written to `images/render_nerf.gif`, it should look like:



Please attach all of your rendering results to your submission file.

Note: We provide the NeRF training code under `train_nerf` in `main.py`. Based on your implementation of volume rendering, you can train a simple NeRF network by the following command:

```
python main.py --config-name=nerf_lego
```

Feel free to try it!

2 Single Image to 3D [7pt]

Problem 2. In this problem, you will design a neural network for lifting a single image to 3D with two different 3D losses and analyze the effect of these losses. Different from Problem 1, you need to implement the network design and the training and evaluation routines on your own (FYI: We provide a codebase under `Problem2_framework` folder, you can either utilize this codebase or implement from scratch).

The problem here is to estimate the object's point cloud from its single image. You can use what you learn in the class to design your neural network backbone that first encodes the image to a high-level feature vector and then decodes the feature vector to a 3D point set as the final estimation.

The loss design for the network could be various. In addition to the Chamfer Distance(CD) loss we mentioned in class, we will introduce the Hausdorff Distance(HD) loss in Problem 2-1. You will implement these two losses in your network training and compare their performances.

We focus on point cloud estimation for specific shape primitives with simple geometries to reduce the shape uncertainty and reduce difficulty. We provide a synthetic dataset in `Problem2` folder and you need to select data from it for both training and inference procedures in Problem 2-2. The dataset contains 100 cubes with diverse scales and colors. For each cube, 16 rendered images from various views and corresponding ground-truth point clouds are included.

1. (3D losses) Let P and Q be two finite sets of points in 3D space, namely $P = \{p_i\}_{i=0}^{n-1}$ and $Q = \{q_j\}_{j=0}^{m-1}$ where $p_i, q_j \in \mathbb{R}^3$. We use $d(p_i, q_j) = ||p_i - q_j||_2$ to represent the Euclidean distance between p_i and q_j. The Hausdorff Distance between P and Q is defined as

$$h(P,Q) = \max\{d(P,Q), d(Q,P)\}, \tag{2}$$

where $d(P,Q) = \max\limits_{p_i \in P} [\min\limits_{q_j \in Q} d(p_i, q_j)]$ and $d(Q,P) = \max\limits_{q_j \in Q} [\min\limits_{p_i \in P} d(q_j, p_i)]$.

[Programming Assignment] Implement Hausdorff Distance(HD) loss and Chamfer Distance(CD) loss in python. The implementation of these two losses should be differentiable for network training.

2. [Programming Assignment] (Network design) Given an RGB image, you will design a neural network to predict the object point cloud. You need to select a loss function implemented in Problem 2-1 to measure the difference between the predicted and the ground-truth point clouds and thus train the network. For the network design, a simple encoder-decoder network mentioned in the class could be competent. The encoder leverages CNNs for encoding the image input to a feature vector representing the whole image. The decoder predicts the point cloud from the feature vector. Various effective decoder designs can be found in `https://arxiv.org/pdf/1612.00603.pdf`.

   Although our dataset is generated by rendering from the known object shape in the synthetic scenario, sometimes we may not have the CAD model and will need to capture the object geometry in the real world to train the network. The multi-view RGB or RGB-D reconstruction algorithms could be used to capture the object geometry, but the reconstruction result is noisy due to the sensor noise and the error of the algorithms. Therefore, we provide a dataset with clean point clouds that corresponds to the most ideal synthetic scenario, and a dataset with noisy point clouds that simulate the real-world noise to some extent.

   - (a) Implement your neural network in pytorch. Besides the network architectures mentioned in the class, you are encouraged to design different networks that you consider effective.
   - (b) Select at least 10 cubes for training and at least 10 cubes for evaluation from the dataset in `Problem2` folder. Use HD loss and CD loss respectively to train the network in the `clean` dataset. Evaluate the networks trained by HD loss and CD loss with the same loss in the training. Report the loss values during network training and evaluation and visualize the predicted point clouds of each network.
   - (c) Use HD loss and CD loss respectively to train the network in the `noisy` dataset and compare the results with those from the `clean` dataset.
   - (d) From what you have discovered, analyze why HD loss is less used in practice.

   Note: The file structure of the dataset in `Problem2` folder is shown below:

- `cube_dataset` – The dataset folder.
  - `clean` – The dataset with clean point clouds.
    - `0` – The data of cube 0. The 16 data pairs in the same folder are rendered from the same cube.
      - `0.png` – The image from view 0 of cube 0.
      - `0.ply` – The point cloud in the camera coordinate system from view 0 of cube 0.
      - ...
      - `15.png` – The image from view 15 of cube 0.
      - `15.ply` – The point cloud in the camera coordinate system from view 15 of cube 0.
    - ...
    - `99` – The data of cube 99.
  - `noisy` – The dataset with noisy point clouds. The structure is the same as the `clean` dataset.

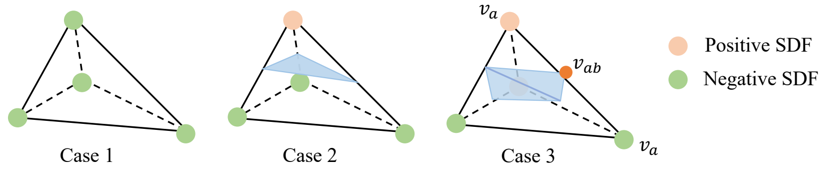
3 Surface Reconstruction [7pt]

Problem 3. Deep Marching Tetrahedra (DMTet) is a hybrid 3D representation that combines both implicit and explicit 3D surface representations. It represents a shape with a discrete SDF defined on vertices of a deformable tetrahedral grid. The SDF is converted to triangular mesh via a differentiable marching tetrahedra algorithm (MT), allowing explicit supervision on the extracted surface to be back-propagated to SDF and change mesh topology. In this problem, we aim at reconstructing meshes from point clouds via DMTet.

Given input point clouds or coarse voxels, DMTet optimizes both the SDF values and the 3D deformation of the tetrahedra grid's vertices so that the resulting mesh, obtained via marching tetrahedra, can approximate the input shape. At its core, DMTet employs a deformable tetrahedral grid representation that encodes a discretized SDF, along with a differentiable MT algorithm. Similar to the Marching Cubes (MC) algorithm covered in class, MT is an isosurfacing method designed to extract an explicit mesh representation from an implicit SDF. However, instead of using a regular grid of cubes, MT partitions space into tetrahedra. This tetrahedral representation helps MT avoid many issues present in MC such as the topological ambiguities.

MT algorithm converts the SDF, encoded in a tetrahedral grid, into a triangular mesh by determining the surface topology within each tetrahedron based on the signs of its vertices. Since each tetrahedron has four vertices, there are a total of $2^4 = 16$ possible sign configurations. Considering rotational symmetry, these configurations reduce to three unique cases, each with an unambiguous surface topology:

- Case 1: All four vertices have the same sign $\rightarrow$ No surface is formed.
- Case 2: Three vertices are negative, while the fourth is positive $\rightarrow$ A triangle separates the positive vertex from the other three.
- Case 3: Two vertices are positive and two are negative $\rightarrow$ A quadrilateral separates them, which can be further subdivided into two triangles.



$\mathbf{v}_{ab}$ is determined by $\mathbf{v}_{ab} = \frac{\mathbf{v}_a \cdot s(\mathbf{v}_b) - \mathbf{v}_b \cdot s(\mathbf{v}_a)}{s(\mathbf{v}_b) - s(\mathbf{v}_a)}$. Note that flipping the sign would not change the surface typology.

DMTet solves the surface reconstruction problem by optimizing the deformation and SDF value of the tetrahedra grid vertices. "Deep" comes from its utilizing neural networks to parameterize the deformation and SDFs. Specifically, for a tetrahedra grid $\mathcal{T} = \{\mathbf{V} \in \mathbb{R}^{N_v \times 3}, \mathbf{T} \in \mathbb{N}^{N_t \times 4}\}$, where N_v is the number of vertices and N_t is the number of tetrahedra, it parameterize the deformation $\mathbf{d}(\mathbf{v})$ and the sign $s(\mathbf{v})$ of a vertex $\mathbf{v}$ via neural networks ϕ with parameters θ: $\mathbf{d}(\mathbf{v}), s(\mathbf{v}) = \phi_\theta(\mathbf{v}, \mathcal{T})$. By defining losses between the extracted surface and the input point cloud, the network can be trained in an end-to-end manner, leveraging the differentiability of the MT algorithm.

In this problem, we aim to implement DMTet and use it to reconstruct surfaces from input point clouds. The original DMTet paper also introduces a tetrahedron subdivision technique, which adaptively refines the tetrahedral grid in specific areas to better capture fine details, such as the thin structures of a mouse's tail. However, you do not need to implement this technique for this homework.

In this problem, you are required to implement the neural networks, the marching tetrahedra algorithm, and the training pipeline for DMTet. While the original implementation of DMTet is not open-sourced, you can find useful references in other GitHub repositories, such as GET3D. Feel free to explore and learn from these resources, but direct copy-and-paste is not allowed.

Requirements are detailed in the following:

1. Network implementation. Implement the following two types of networks to parameterize the SDF and deformation of the tetrahedral grid:
   - MLP. A fully connected neural network that takes vertex positions as input and outputs the SDF and deformation: $\mathbf{d}(\mathbf{v}), s(\mathbf{v}) = \text{MLP}_\theta(\mathbf{v})$. Instead of directly using the raw (x, y, z) coordinates, apply a Fourier-based positional encoding to transform the input into a higher-dimensional

space. This helps prevent over-smoothed reconstructions, similar to NeRF.

- 3D Convolution Network. A 3DConv network that accepts vertex position and outputs SDF and deformation, *i.e.,* $\mathbf{d}(\mathbf{v}), s(\mathbf{v}) = \text{3DConv}_\theta(\mathbf{v})$. Similarly, instead of using the original vertex coordinates, you may add a positional encoding layer.

2. Marching tetrahedra. MT algorithm can be implemented following the process described above. In GET3D, a precomputed look-up table is used to efficiently determine edges by encoding and retrieving them from the table. There are no strict requirements on how you implement this process—as long as your function is correct, you will receive full credit for this part. Additionally, while the `kaolin` package provides a built-in marching tetrahedra function, you may only use it to verify the correctness of your implementation. Directly using it in your homework is not allowed.

3. Reconstruction loss. MT is differentiable. After extracting the mesh, we can optimize the network by defining losses between the mesh and the original point cloud and backward the gradient. Implement the Chamfer Distance loss, which measures the distance between points sampled from the surface of the extracted triangular mesh and the input point cloud. The process of sampling points from the mesh surface should be differentiable. You may either implement the sampling yourself or use a function from an existing Python package, *e.g.,* `kaolin.ops.mesh.sample_points`. In addition to the CD loss, add a Laplacian regularization term to improve the smoothness of the reconstructed surface. This results in the following final loss function:
$$\mathcal{L} = \mathcal{L}_{CD} + \lambda_{reg}\mathcal{L}_{reg}. \tag{3}$$
Try to vary the parameter λ_{reg} and analyze the influence of $\mathcal{L}_{reg}$ on the final reconstruction mesh. One reference implementation of $\mathcal{L}_{reg}$ is as follows:

```python
def laplace_regularizer_const(mesh_verts, mesh_faces):
    term = torch.zeros_like(mesh_verts)
    norm = torch.zeros_like(mesh_verts[..., 0:1])

    v0 = mesh_verts[mesh_faces[:, 0], :]
    v1 = mesh_verts[mesh_faces[:, 1], :]
    v2 = mesh_verts[mesh_faces[:, 2], :]

    term.scatter_add_(0, mesh_faces[:, 0:1].repeat(1,3), (v1 -
        v0) + (v2 - v0))
    term.scatter_add_(0, mesh_faces[:, 1:2].repeat(1,3), (v0 -
        v1) + (v2 - v1))
    term.scatter_add_(0, mesh_faces[:, 2:3].repeat(1,3), (v0 -
        v2) + (v1 - v2))

    two = torch.ones_like(v0) * 2.0
    norm.scatter_add_(0, mesh_faces[:, 0:1], two)
```

```
        norm.scatter_add_(0, mesh_faces[:, 1:2], two)
        norm.scatter_add_(0, mesh_faces[:, 2:3], two)

        term = term / torch.clamp(norm, min=1.0)

        return torch.mean(term**2)
```

You can find the point clouds for this problem in the folder `Problem3_pcs`. You are expected to experiment with tetrahedra grids of various resolutions and analyze their corresponding results. The folder `Problem3_tets` contains six tetrahedra grids in the format `{grid_res}_compress.npz`. For your experiments, please consider at least four of these grids. Each tetrahedra grid file is a dictionary that describes the vertex locations and the tetrahedra structure. Specifically:

- The value of `vertices` is a `NumPy` array with shape $N_v \times 3$, where each row corresponds to the coordinates of a vertex.
- The value of `tets` is a `NumPy` array with shape $N_t \times 4$, where each row contains the indices of the vertices that define a tetrahedron.

For this problem, please submit the following materials:

- Report:
  - Visualizations of the reconstructed meshes and the corresponding sampled point clouds.
  - Comparisons and analysis of the results.
- Sampled Points: Save the sampled points in formats such as `.npy`, `.ply`, etc.
- Reconstructed Meshes: Save the reconstructed meshes in formats such as `.obj`, `.ply`, etc.
- Source code: Include all the source code used for your experiments and implementations.