Objects – For each main object in the code we present it, the implementation details in short, the functionality it enables and its uses.

**Vertex**

QuarTet.py

Update vertex:

Use -

**Tetrahedron**

QuarTet.py

* 4 vertices
* Occupancy value ()
* Tetrahedrons features – initialize with the average of vertices positions.
* Every tetrahedron has 4 neighbors. If its on the initial cube boundary, consider itself as his neighbor

Functionality:

* Calculate volume using determinant   
  (see put a link here)
* Update vertices location by “direction vector” for each vertex.

Use – The building block of the Quartet, it holds the occupancy value.

**Quartet**

QuarTet.py

Unit cube that is divided into tetrahedrons.

Quartet creation:

* We build a unit cube from smaller 1/N x 1/N x 1/ N cubes. Each of them we separate to 24 parts such that when we put the cubes the diagonals, we created on the small cubes’ boundary coalesce.
* One appearance to any vertex even if it is in several adjacent tetrahedrons.
* We calculate for each tetrahedrons its neighborhoods using a trivial algorithm, T is the number of tetrahedrons. If time permits, we will implement an algorithm that is based on range query. It’s description in Appendix A.

Functionality:

Mesh from quartet:

* for any face of the inner tetrahedrons denote by the 2 tetrahedrons which is their shared face.
* If then add it to the mesh faces.

Use – for visualization

Sample Point Cloud:

* For each tetrahedron T with calculate its volume.
* Sample from the occupied tetrahedrons w.r.t their volume (from larger volume tetrahedrons we sample more)

Use – to calculate the loss (see the loss section)

Use – Quartet object is the input to the networks we define; they get a quartet and calculate for each vertex a direction and for each tetrahedrons its occupancy. We define the quartet in main.py at the start of the execution. and use it from there.

**Quartet Networks**

networks.py

Convolutional Layer (MotherCubeConv): The convolution layer applied to each tetrahedron, T, in the following way: take its 4 neighbors (T\_1, T\_2, T\_3, T\_4) average their features (1. to ensure permutation invariance 2. with T also) and apply a linear layer on top of the new features. The output is the new T features, for the rest of the tetrahedrons we use the old value of T features (1. The new one can be in different shape 2. Permutation invariance).

Pooling Layer (MotherCubePool):

Our Network (OurNet):

* Conv + Pooling Layers defined in parameters. Default . Pooling something = […]
* Linear network which its input is the tetrahedron features (64 in default) and outputs a direction vector (3 coordinates). After calculated on all tetrahedrons, the network applies direction vector on each of the tetrahedron vertices, divided by number of tetrahedrons that share this vertex.
* Linear network which its input is the tetrahedron features (64 in default) and outputs the new occupancy.

**Loss**

networks.py

Input: Network output – the quartet, input point cloud (of size ).

1. Sample a point cloud of size from the quartet (w.r.t volumes, as described in the Quartet section).
2. Return the chamfer distance between the input point cloud and the sampled (from step 1).

Loss intuition: note that both occupancy and vertices location (defines volume) are necessary to compute the sample from step 1. So they both have to be optimized.

The inner tetrahedrons will be encouraged to be filled because we assume the input point cloud is filled (see point cloud section). Therefore, if the inner parts were not occupied, we would suffer from large distance from the inside of the shape to the boundary which can be minimized if we set the occupancy of the inner tetrahedrons to 1 as the network actually learns.

**Main**

main.py

Initialize Quartet. Iteratively apply the network on the quartet, calculate the loss and update (only) network weights. **After each iteration reset the quartet.**

Explaining why the quartet need to be reset (totally to initial values):

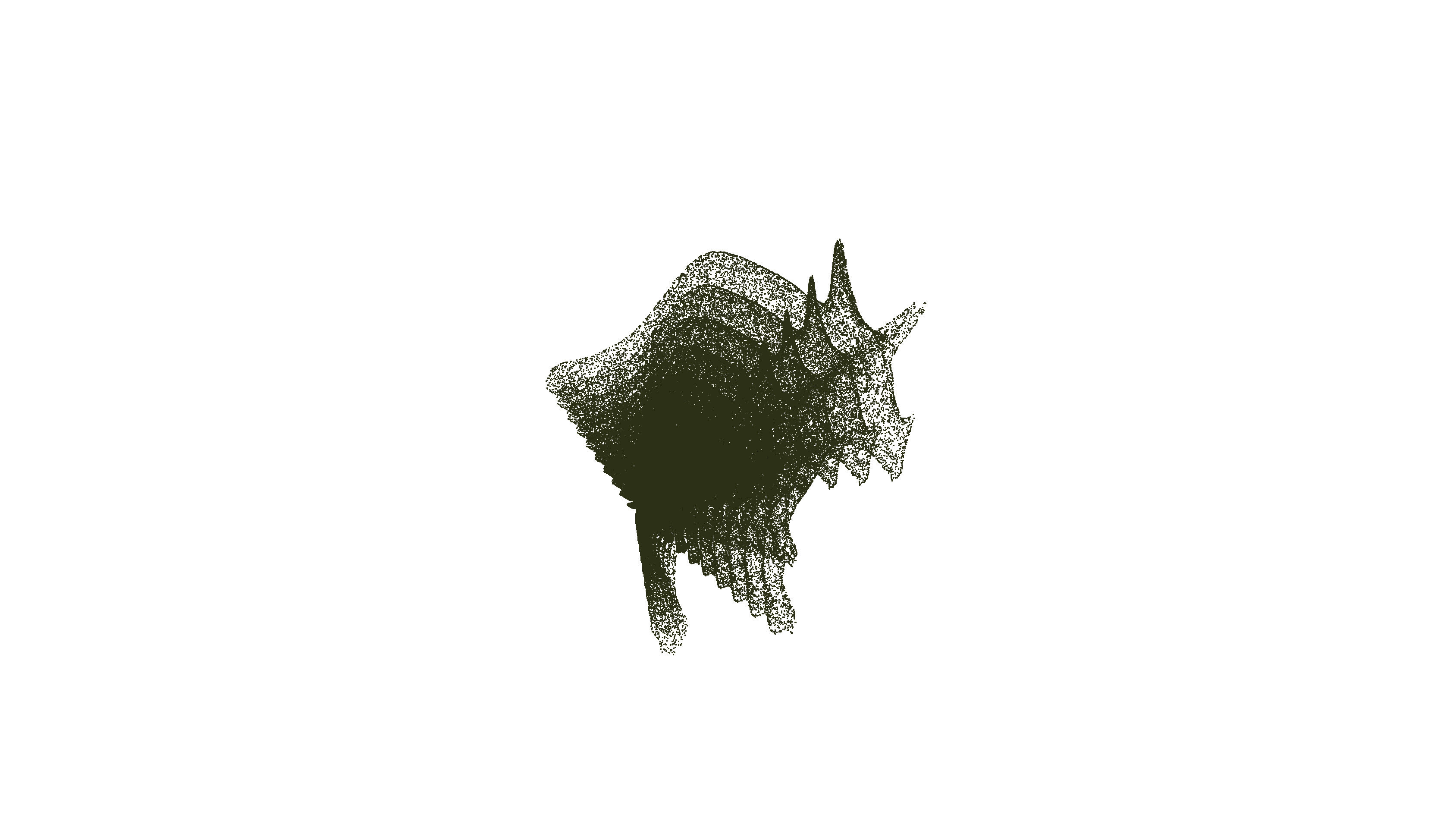
* The network tries to overfit something and therefore it should get the same input during the optimization steps. As for overfitting an image (SIREN) they give it the same pair of (pixel index, pixel value). If we change the vertices locations, we changed the input so if we pass it to the network, it sees it at first time and therefore do not have anything meaningful about it.
* This was a bug that we encountered for long time, after rethinking on overfitting as discussed, we were able to fix it.

The initialization is slow (can take more than 20 seconds) to do at every step, so we have a reset method in quartet that does it fast.

**Point Cloud Filling**

PointCloud.py

Naïve way: center the point cloud and for many different add . This minimizes the shape size by parameter which is inside the shape because . For each alpha we take only a few samples as otherwise the object will be huge (it gets to 17MB).



Using SDF: signed distance function (SDF) in 3D is a function where on input returns the distance to the boundary multiplied by .

Remark – for points inside the shape f is negative and for points outside is positive.

Now we deal with the problem of calculating such SDF given a point cloud of the boundary and the normals. Recently a model named SIREN was published, by researchers at Stanford, and the SIREN ability is to learn also on the model gradient. In short, they found ReLU activation makes a DNN derivative w.r.t input to be not expressive at all (that because ReLU derivative is almost constant – 1 in positive values and 0 in negative). They switched the activation to be sinusoid and argue that the expressiveness holds and that with this activation the derivative of the network is expressive. They modeled problem that wasn’t possible to solve with deep learning properly before this model and one of them is to find SDF.

We will show the loss function for SDF that matches the gradients with the normals, then we briefly explain it (for more details see section \_\_\_ in [\_\_\_]). The SDF is denoted by  and the normals by .

Remarks - “” denotes scalar product which is highest for vectors in the same direction. The paper says that after a massive hyperparameter search the weighting parameters for the 4 losses parts are from left to right: .

The loss has 4 parts:

* The absolute value of the SDF function is the distance function.
* The most important - ensures that the boundary derivatives correspond to the normal (in the same direction).
* That the gradient all over the space is 1 (what means linear growth of distance).

Filling with SDF: Draw uniformly many points in the unit cube. Remove those with positive SDF value. We remain with the negative values points only and because we sampled uniformly at random, we get a uninform filling of the point cloud shape.