COMPSCI 574: Intelligent Visual Computing Spring 23

Prachi Jain

Assignment 1: Basic Implicit Surface Reconstruction

1 Naive Reconstruction

1.1 Sphere

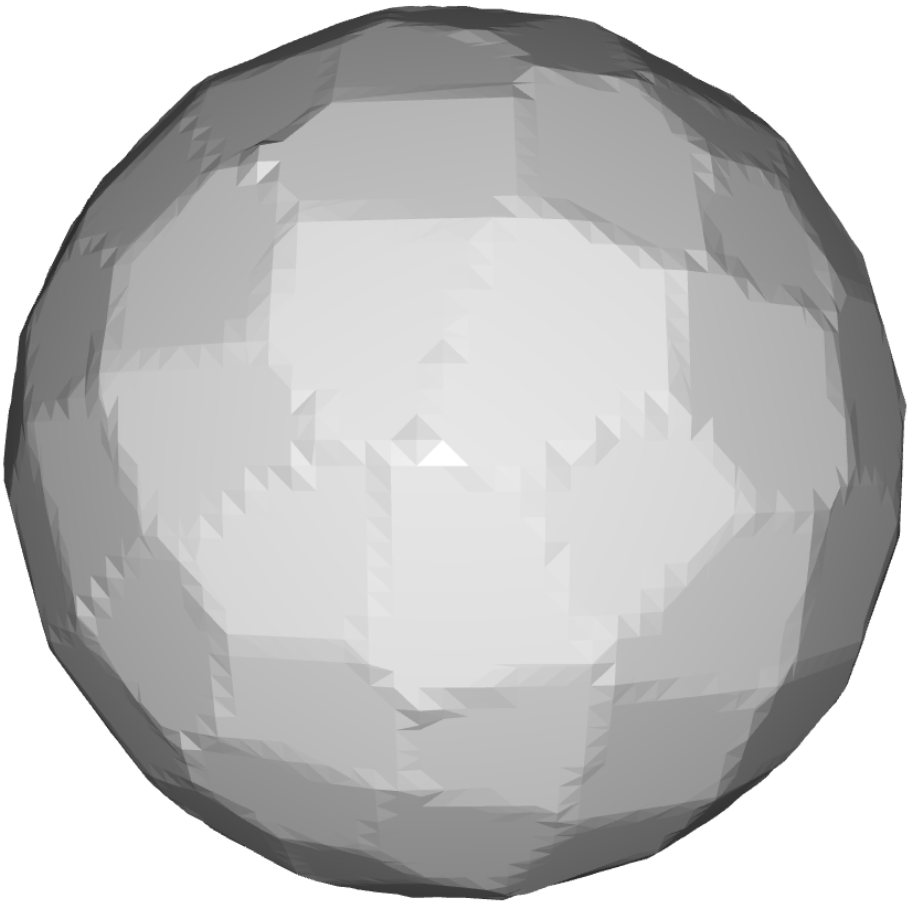

Figure 1: Naive Reconstruction of Sphere

1.2 Bunny with 500 Points

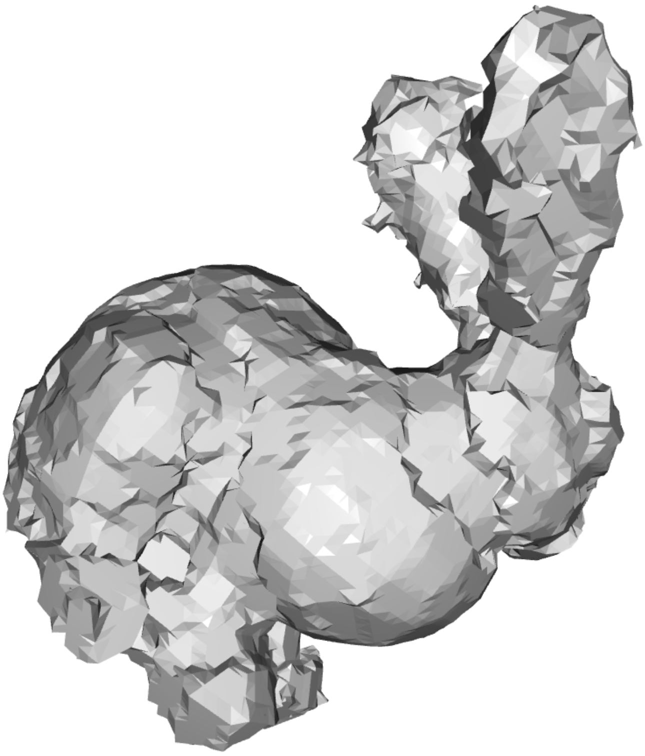

Figure 2: Naive Reconstruction of Bunny from 500 points

1.3 Bunny with 1000 Points

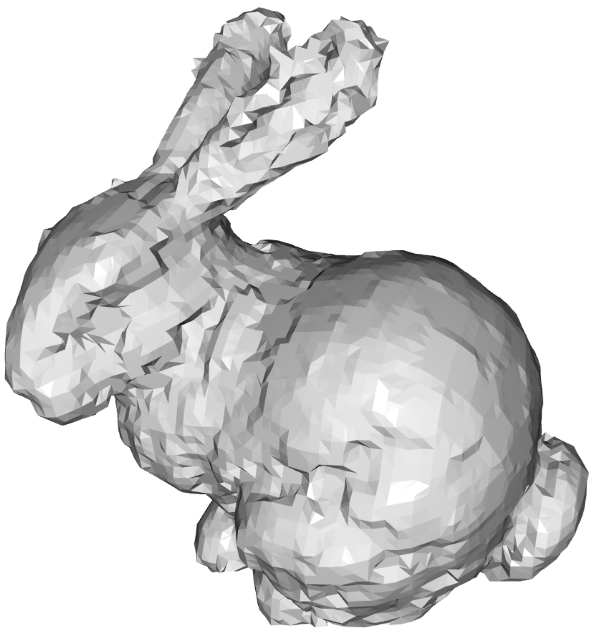

Figure 3: Naive Reconstruction of Bunny from 1000 points

2 MLS Reconstruction

2.1 Sphere

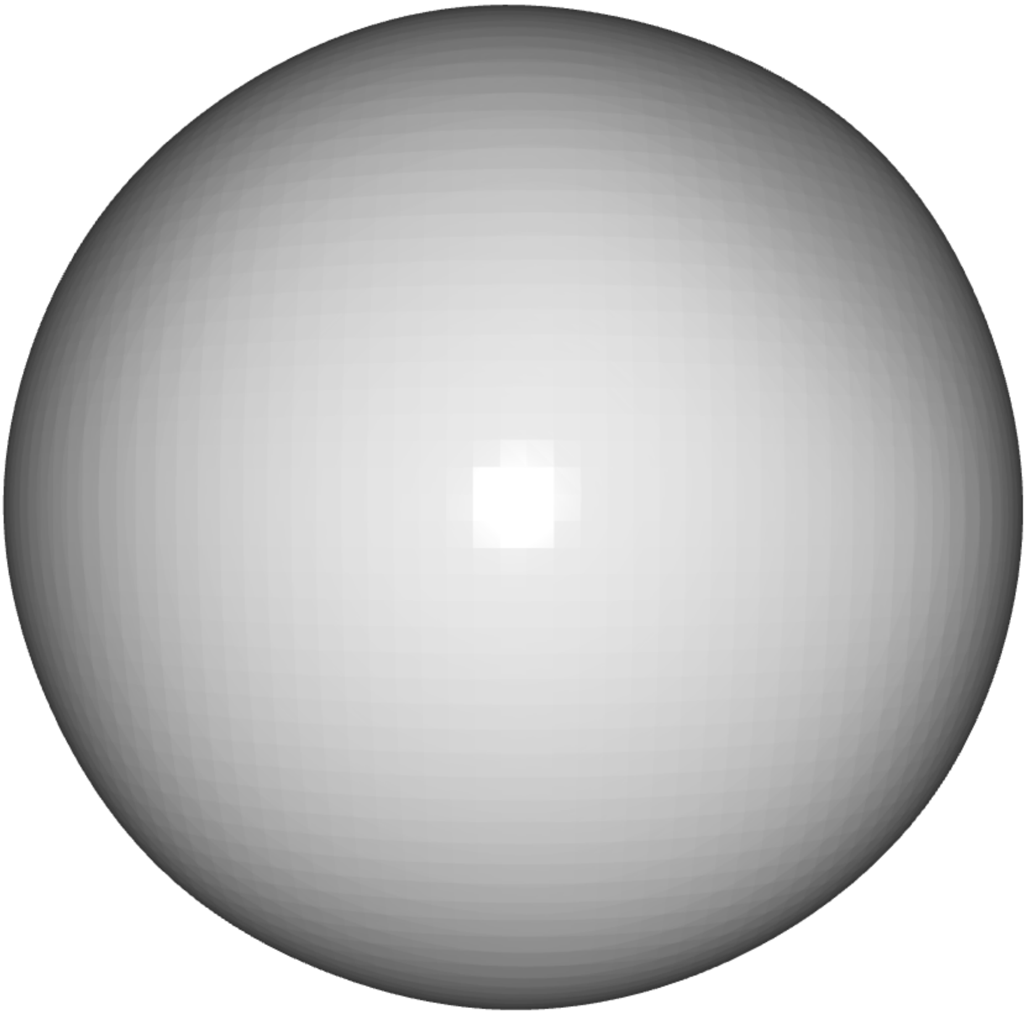

Figure 4: Reconstruction of Sphere using Moving Least Squares

2.2 Bunny with 500 Points

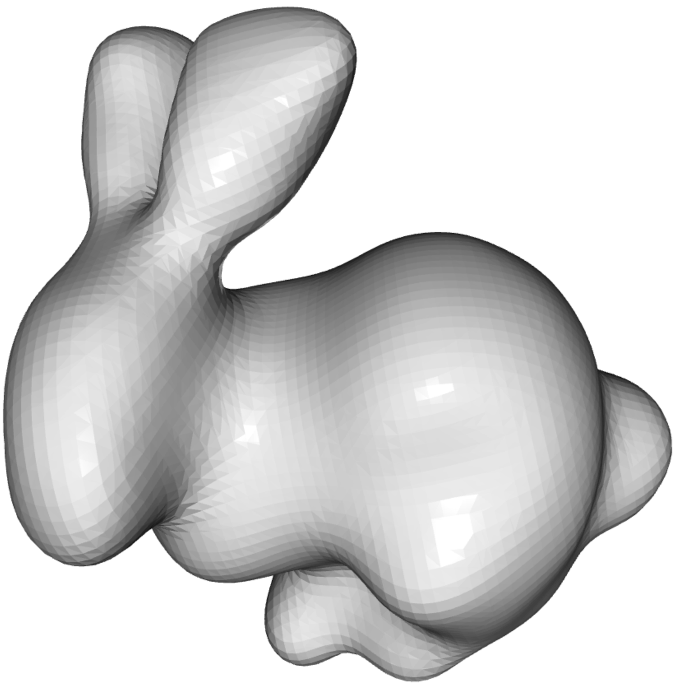

Figure 5: Reconstruction of Bunny from 500 points using Moving Least Squares

2.3 Bunny with 1000 Points

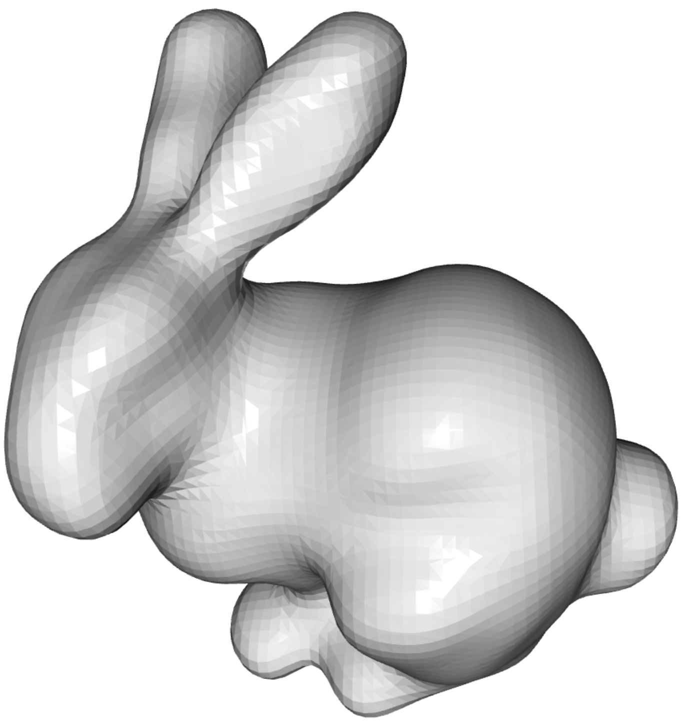

Figure 6: Reconstruction of Bunny from 1000 points using Moving Least Squares

3 Implementation

Please find attached code below to replicate the above results.

```
import argparse
import numpy as np
from skimage import measure
from sklearn.neighbors import KDTree
import open3d as o3d

def createGrid(points, resolution=96):
    """
    constructs a 3D grid containing the point cloud
    each grid point will store the implicit function value
    Args:
        points: 3D points of the point cloud
        resolution: grid resolution i.e., grid will be NxNxN where N=resolution
                    set N=16 for quick debugging, use *N=64* for reporting results
    Returns:
        X,Y,Z coordinates of grid vertices
        max and min dimensions of the bounding box of the point cloud
    """
    max_dimensions = np.max(points,axis=0) # largest x, largest y, largest z coordinates among all surface points
    min_dimensions = np.min(points,axis=0) # smallest x, smallest y, smallest z coordinates among all surface points
    bounding_box_dimensions = max_dimensions - min_dimensions # com6pute the bounding box dimensions of the point cloud
    max_dimensions = max_dimensions + bounding_box_dimensions/10  # extend bounding box to fit surface (if it slightly extends beyond the point cloud)
    min_dimensions = min_dimensions - bounding_box_dimensions/10
    X, Y, Z = np.meshgrid( np.linspace(min_dimensions[0], max_dimensions[0], resolution),
                           np.linspace(min_dimensions[1], max_dimensions[1], resolution),
                           np.linspace(min_dimensions[2], max_dimensions[2], resolution) )

    return X, Y, Z, max_dimensions, min_dimensions

def sphere(center, R, X, Y, Z):
    """
    constructs an implicit function of a sphere sampled at grid coordinates X,Y,Z
    Args:
        center: 3D location of the sphere center
        R     : radius of the sphere
        X,Y,Z coordinates of grid vertices
    Returns:
        IF    : implicit function of the sphere sampled at the grid points
    """
    IF = (X - center[0]) ** 2 + (Y - center[1]) ** 2 + (Z - center[2]) ** 2 - R ** 2
    return IF

def showMeshReconstruction(IF):
```

```
    """
    calls marching cubes on the input implicit function sampled in the 3D grid
    and shows the reconstruction mesh
    Args:
        IF    : implicit function sampled at the grid points
    """
    verts, triangles, normals, values = measure.marching_cubes(IF, 0)

    # Create an empty triangle mesh
    mesh = o3d.geometry.TriangleMesh()
    # Use mesh.vertex to access the vertices' attributes
    mesh.vertices = o3d.utility.Vector3dVector(verts)
    # Use mesh.triangle to access the triangles' attributes
    mesh.triangles = o3d.utility.Vector3iVector(triangles.astype(np.int32))
    mesh.compute_vertex_normals()
    o3d.visualization.draw_geometries([mesh])

def mlsReconstruction(points, normals, X, Y, Z):
    """
    surface reconstruction with an implicit function f(x,y,z) representing
    MLS distance to the tangent plane of the input surface points
    The method shows reconstructed mesh
    Args:
        input: filename of a point cloud
    Returns:
        IF    : implicit function sampled at the grid points
    """

    ################################################
    # <================START CODE<================>
    ################################################

    # replace this random implicit function with your MLS implementation!
    #IF = np.random.rand(X.shape[0], X.shape[1], X.shape[2]) - 0.5
    IF = np.zeros(shape = X.shape)

    # this is an example of a kd-tree nearest neighbor search (adapt it accordingly for your task)
  # use kd-trees to find nearest neighbors efficiently!
  # kd-tree: https://en.wikipedia.org/wiki/K-d_tree
    Q = np.array([X.reshape(-1), Y.reshape(-1), Z.reshape(-1)]).transpose()
    tree = KDTree(points)

    # Finding 50 nearest neighbors
    _, idx = tree.query(Q, k = 50)
    totalIndices = len(idx)

    # Linear KNN Search for finding 1 - closest neighboring surface point for Beta - controlling weight decay
    #M, N = points.shape
    #idxBeta = np.zeros((M, 1), dtype = int)
    #confidence = np.array_equal(points, points)
    #for i in range(0, M):
    #    distBeta = np.sum(np.power(points[:, :] - points[i, :], 2), axis=1)
    #    if confidence:
    #        distBeta[i] = np.inf
    #    idxBeta[i] = np.argmin(distBeta)

```

```
    # Finding nearest neighboring surface point for Beta - controlling weight decay
    # For k = 1 point is pointing to the point itself. Therefore considering k = 2 and second column
    _, idx2 = tree.query(points, k = 2)
    idxBeta = list(zip(*idx2))
    idxBeta = np.array(idxBeta[1])

    # Beta Computation
    beta = 2 * np.mean(np.sqrt(np.sum(np.power(points-points[idxBeta].squeeze(), 2),axis=1)))

    # Implicit function Computation
    IF_ = []
    for i in range(totalIndices):
        Grid = Q[i]
        Point  =  points[idx[i]]
        Norm =  normals[idx[i]]

        # Compute Phi matrix
        phiMatrix = np.exp(np.sum(np.power(Grid - Point, 2), axis = 1) * (1/np.power(beta, 2) * (-1)))
        distance = []
        for i in range(len(Norm)):
            distance.append((np.dot(Norm[i], (Grid - Point)[i]) * phiMatrix[i])/np.sum(phiMatrix))
        IF_.append((np.sum(distance)))

    IF = np.array(IF_).reshape(X.shape)

    ################################################
    # <================END CODE<================>
    ################################################

    return IF


def naiveReconstruction(points, normals, X, Y, Z):
    """
    surface reconstruction with an implicit function f(x,y,z) representing
    signed distance to the tangent plane of the surface point nearest to each
    point (x,y,z)
    Args:
        input: filename of a point cloud
    Returns:
        IF    : implicit function sampled at the grid points
    """


    ################################################
    # <================START CODE<================>
    ################################################

    # replace this random implicit function with your naive surface reconstruction implementation!
    #IF = np.random.rand(X.shape[0], X.shape[1], X.shape[2]) - 0.5
    IF = np.zeros(shape = X.shape)
```

```python

    # this is an example of a kd-tree nearest neighbor search (adapt it accordingly for your task)
  # use kd-trees to find nearest neighbors efficiently!
  # kd-tree: https://en.wikipedia.org/wiki/K-d_tree
    Q = np.array([X.reshape(-1), Y.reshape(-1), Z.reshape(-1)]).transpose()
    tree = KDTree(points)

    # Finding 1 nearest neighbor
    _, idx = tree.query(Q, k = 1)
    totalIndices = len(idx)
    IF_ = []

    # Implicit function Computation
    for i in range(totalIndices):
        Grid = Q[i]
        Point = points[idx[i]].squeeze()
        Norm  = normals[idx[i]].squeeze()
        IF_.append(np.dot(Norm, Grid - Point))

    IF = np.array(IF_).reshape(X.shape)

    ################################################
    # <================END CODE<================>
    ################################################

    return IF

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Basic surface reconstruction')
    parser.add_argument('--file', type=str, default = "sphere.pts", help='input point cloud filename')
    parser.add_argument('--method', type=str, default = "sphere",\
                        help='method to use: mls (Moving Least Squares), naive (naive reconstruction), sphere (just shows a sphere)')
    args = parser.parse_args()

    #load the point cloud
    data = np.loadtxt(args.file)
    points = data[:, :3]
    normals = data[:, 3:6]

    # create grid whose vertices will be used to sample the implicit function
    X,Y,Z,max_dimensions,min_dimensions = createGrid(points, 64)

    if args.method == 'mls':
        print(f'Running Moving Least Squares reconstruction on {args.file}')
        IF = mlsReconstruction(points, normals, X, Y, Z)
    elif args.method == 'naive':
        print(f'Running naive reconstruction on {args.file}')
        IF = naiveReconstruction(points, normals, X, Y, Z)
    else:
        # toy implicit function of a sphere - replace this code with the correct
        # implicit function based on your input point cloud!!!
        print(f'Replacing point cloud {args.file} with a sphere!')
        center =  (max_dimensions + min_dimensions) / 2
        R = max( max_dimensions - min_dimensions ) / 4
        IF =  sphere(center, R, X, Y, Z)
```

```python

    #Fix issue: Changing axes as the orginal one is caused due to artifact of skimage marching cubes
    showMeshReconstruction(IF.transpose(1, 0, 2))
```