1 Question 1

Consider the following differential equation over the interval $(0, 1]$:

$$\frac{dy}{dx} = \frac{1}{x^2(1-y)}$$

with $y(1) = -1$.

Often we use this function form to describe forces when navigating through a field of obstacles. In its general form this is called potential field navigation and borrows numerous results from physics.

(a) Obtain an exact analytical solution to the equation. Solve for $y(0)$ (even though in theory the equation is not defined for $x = 0$).

Analytical Solution:

$$\frac{dy}{dx} = \frac{1}{x^2(1-y)}$$

Separating variables:

$$(1-y)\,dy = \frac{1}{x^2}\,dx$$

Integrating both sides:

$$\int (1-y)\,dy = \int \frac{1}{x^2}\,dx$$

Solving the integrals:

$$y - \frac{y^2}{2} = -\frac{1}{x} + C$$

Using the initial condition $y(1) = -1$:

$$-1 - \frac{(-1)^2}{2} = -1 + C$$

$$C = -\frac{1}{2}$$

Therefore, the solution is:

$$y - \frac{y^2}{2} = -\frac{1}{x} - \frac{1}{2}$$

Solving for $y(0)$:

$$y(0) = -\infty$$

(b) Implement and use Euler's method to solve the differential equation numerically. Use a step size of 0.05. How accurate is your numerical solution?

Euler's method:

```
import numpy as np

def dy_dx(x, y):
    return 1 / (x**2 * (1 - y))

def euler_method(h):
    # Start from x = 1 and work backwards to x = 0
    x = np.arange(1, 0 - h, -h)  # Going backwards from 1 to 0
    y = np.zeros_like(x)
    y[0] = -1  # Initial condition at x = 1

    # Implement backward Euler steps
```

```
        for i in range(1, len(x)):
            y[i] = y[i - 1] - h * dy_dx(x[i - 1], y[i - 1])

        return y[-1]  # Return y(0)

    y0 = euler_method(0.05)
```

After running the code, the output is:

$$y(0) \approx -8.12493$$

(c) Implement and use a fourth-order Runge-Kutta method to solve the differential equation numerically. Again, use a step size of 0.05. Again, how accurate is your numerical solution?

Runge-Kutta method:

```
def fourth_order_Runge_Kutta(h):
    x = np.arange(1,0-h,-h)  # Going backwards from 1 to 0
    y = np.zeros_like(x)
    y[0] = -1  # Initial condition at x=1

    # Implement fourth_order_Runge_Kutta
    for i in range(1,len(x)):
        k1 = -h * dy_dx(x[i-1], y[i-1])
        k2 = -h * dy_dx(x[i-1]- h/2, y[i-1]+k1/2)
        k3 = -h * dy_dx(x[i-1] - h/2, y[i-1] + k2/2)
        k4 = -h * dy_dx(x[i-1] - h, y[i-1] + k3)
        y[i] = y[i-1] + (k1 + 2*k2 + 2*k3 + k4) / 6

    return y[-1]

y0=fourth_order_Runge_Kutta(0.05)
```

After running the code, the output is:

$$y(0) \approx -8.63194142289099e+26$$

(d) Implement and use a Richardson extrapolation to solve the equation, again with a step size of 0.05. How accurate is your solution compared to the analytical solution?

Richardson extrapolation:

```
def richardson_extrapolation(method, h, p):
    y_h = method(h)           # Solution with step size h
    y_half_h = method(h / 2)  # Solution with step size h/2
    # Apply Richardson Extrapolation
    y_extrapolated = y_half_h + (y_half_h - y_h) / (2**p - 1)
    return y_extrapolated

    y0 = richardson_extrapolation(fourth_order_Runge_Kutta, 0.05,p=4)
```

After running the code, the output is:

$$y(0) \approx -2.5375856347702562e+26$$

2 Question 2

We have multiple robots that can generate point clouds such as those coming from a RealSense camera. In many cases we want to use the robots to detect objects in its environment. We provide three data files:

- `Empty2.asc`, which contains data for an empty table
- `TableWithObjects2.asc`, which contains data for a cluttered table
- `CSE.asc`, which contains data of the CSE building viewed from the bear courtyard

Each file has the point cloud data in a format where each line contains x_i y_i z_i. You can use `np.loadtxt` to load a pointcloud into a numpy array.

2.1 (a) Provide a method to estimate the plane parameter for the table. Test it both with the empty and cluttered table. Describe how you filter out the data from the objects. You have to be able to estimate the table parameters in the presence of clutter.

2.1.1 Solution:

To visualize the 3D point cloud data from both Empty2.asc and TableWithObjects2.asc files, I used the Open3D library. Next, to fit a plane to the 3D point cloud and handle noise and outliers, I implemented the Random Sample Consensus (RANSAC) algorithm. This approach allows for robust plane fitting even with noisy data and outliers. For this, I used 'RANSACRegressor' from the 'sklearn' library, which returns the normal vector and the points that fit the plane within the specified RANSAC residual threshold.

Although we successfully identified the plane, some points lying on the plane are not part of the table itself, such as points from the seat in the background. To remove these unwanted points, I used K-Means clustering to separate the points into two clusters. This allows us to isolate the points belonging to the table from those associated with other objects in the background.

Normal factor for Empty table:

$$n = [0.01704676 - 0.86597298 - 0.49980016]$$

Normal factor for Table with objects:

$$n = [0.01319424 - 0.86837008 - 0.49574118]$$

2.1.2 code to get the plane form Empty.asc

```
import numpy as np
from sklearn.linear_model import RANSACRegressor
from sklearn.cluster import KMeans
import open3d as o3d

def load_file(files):
    return np.loadtxt(files)

def fit_plane_ransac(pointcloud):
    # Separate the point cloud into 3D coordinates (xi, yi, zi)
    X = pointcloud[:, :2]  # xi, yi as the coordinates
    Y = pointcloud[:, 2]   # zi as the values

    # Fit the RANSAC model
    ransac = RANSACRegressor(min_samples=10,residual_threshold=0.1, max_trials=2000,random_state=0)
    ransac.fit(X, Y)
    a, b = ransac.estimator_.coef_
    c = -1.0

    normal = np.array([a, b, c])
    normal = normal / np.linalg.norm(normal)

    return normal, ransac.inlier_mask_

def cluster_plane_points(pointcloud, inlier_mask):
    # Extract the points that lie on the plane
    plane_points = pointcloud[inlier_mask]

    # Apply K-Means clustering to separate table and non-table points
    kmeans = KMeans(n_clusters=2, random_state=0)
    kmeans.fit(plane_points)
    labels = kmeans.labels_

    # Create separate point clouds for table and non-table
    table_points = plane_points[labels == 0]
    non_table_points = plane_points[labels == 1]

    return table_points, non_table_points


pointcloud_empty=load_file("Empty2-1.asc")

point_cloud = o3d.geometry.PointCloud()
point_cloud.points = o3d.utility.Vector3dVector(pointcloud_empty)
o3d.visualization.draw_geometries([point_cloud])

plane_params_empty, inlier_mask = fit_plane_ransac(pointcloud_empty)

point_cloud = o3d.geometry.PointCloud()
point_cloud.points = o3d.utility.Vector3dVector(pointcloud_empty[inlier_mask])
o3d.visualization.draw_geometries([point_cloud])

table_points, non_table_points = cluster_plane_points(pointcloud_empty, inlier_mask)

# # Visualize the results
table_cloud = o3d.geometry.PointCloud()
table_cloud.points = o3d.utility.Vector3dVector(table_points)

o3d.visualization.draw_geometries([table_cloud])
print(plane_params_empty)
```

2.1.3 code to get the plane form TableWithObject.asc

```
import numpy as np
from sklearn.linear_model import RANSACRegressor
import open3d as o3d
from sklearn.cluster import KMeans

def load_file(files):
    return np.loadtxt(files)
```

```
def fit_plane_ransac(pointcloud):
    # Separate the point cloud into 3D coordinates (xi, yi, zi)
    X = pointcloud[:, :2]  # xi, yi as the coordinates
    Y = pointcloud[:, 2]   # zi as the values

    # Fit the RANSAC model
    ransac = RANSACRegressor(min_samples=10,residual_threshold=0.1, max_trials=4000,random_state=0)
    ransac.fit(X, Y)

    a, b = ransac.estimator_.coef_
    c = -1.0

    normal = np.array([a, b, c])
    normal = normal / np.linalg.norm(normal)

    return normal, ransac.inlier_mask_

def cluster_plane_points(pointcloud, inlier_mask):
    # Extract the points that lie on the plane
    plane_points = pointcloud[inlier_mask]

    # Apply K-Means clustering to separate table and non-table points
    kmeans = KMeans(n_clusters=2, random_state=0)
    kmeans.fit(plane_points)
    labels = kmeans.labels_

    # Create separate point clouds for table and non-table
    table_points = plane_points[labels == 0]
    non_table_points = plane_points[labels == 1]

    return table_points, non_table_points


pointcloud_object=load_file("TableWithObjects2-1.asc")
point_cloud = o3d.geometry.PointCloud()
point_cloud.points = o3d.utility.Vector3dVector(pointcloud_object)
o3d.visualization.draw_geometries([point_cloud])

plane_params_object, inlier_mask2 = fit_plane_ransac(pointcloud_object)

point_cloud = o3d.geometry.PointCloud()
point_cloud.points = o3d.utility.Vector3dVector(pointcloud_object[inlier_mask2])
o3d.visualization.draw_geometries([point_cloud])

table_points, non_table_points = cluster_plane_points(pointcloud_object, inlier_mask2)
table_cloud = o3d.geometry.PointCloud()
table_cloud.points = o3d.utility.Vector3dVector(table_points)

o3d.visualization.draw_geometries([table_cloud])

print(plane_params_object)
```

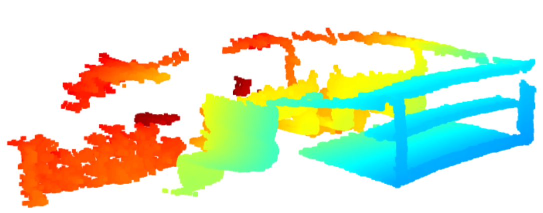
Figure 1: Empty2 point cloud

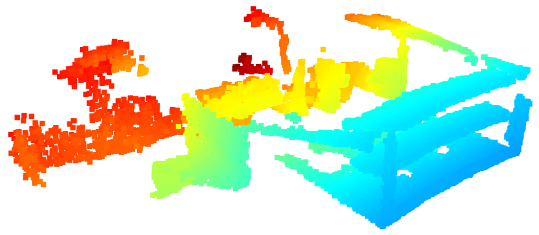
Figure 2: TableWithObject2 point cloud

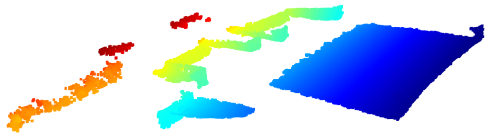
Figure 3: Empty2 after RANSAC

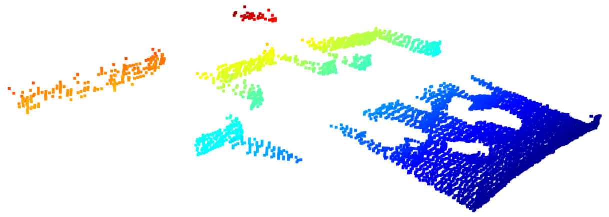
Figure 4: TableWithObject2 after RANSAC

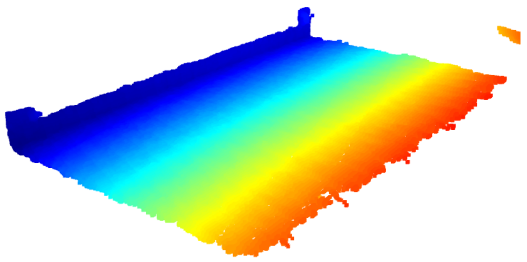
Figure 5: Empty2 after K-Means clustering

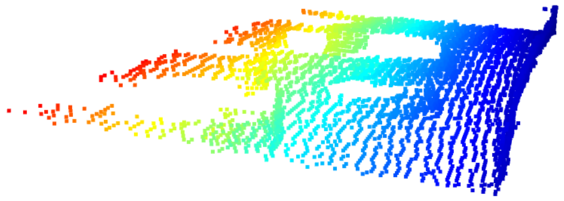
Figure 6: TableWithObject2 after K-Means clustering

2.2 (b) Describe and show how the method can be generalized to extract all the dominant planes of the CSE building viewed in the outdoor environment.

2.2.1 Solution

We can also use the Open3D library to visualize the 3D point cloud of a CSE building(as show in fig, and then apply the same method as in part A to extract the ground plane. To obtain the wall plane, we need to identify the outliers from the first RANSAC, and then use those outlier points to fit a second plane, distinct from the ground plane. As a result, we can found all the dominant planes of the CSE building viewed in the outdoor environment.

2.2.2 code to get the dominant plane of the CSE building

```
import open3d as o3d
import numpy as np
from sklearn.linear_model import RANSACRegressor



def load_file(files):
    return np.loadtxt(files)

def fit_plane_ransac(pointcloud):
    # Separate the point cloud into 3D coordinates (xi, yi, zi)
    X = pointcloud[:, :2]  # xi, yi as the coordinates
    Y = pointcloud[:, 2]   # zi as the values

    # Fit the RANSAC model
    ransac = RANSACRegressor(residual_threshold=0.2, max_trials=100,random_state=0)
    ransac.fit(X, Y)

    a, b = ransac.estimator_.coef_
    c = -1.0

    normal = np.array([a, b, c])
    normal = normal / np.linalg.norm(normal)
    outlier = ~ransac.inlier_mask_
    return normal, ransac.inlier_mask_, outlier

def fit_wall_ransac(pointcloud,outlier):
     X= pointcloud[outlier][:,[0,2]]
     Y= pointcloud[outlier][:,1]
     ransac = RANSACRegressor(residual_threshold=0.2, max_trials=100,random_state=0)
     ransac.fit(X, Y)

     return ransac.inlier_mask_



pointcloud_cse=load_file("CSE-1.asc")

point_cloud = o3d.geometry.PointCloud()
point_cloud.points = o3d.utility.Vector3dVector(pointcloud_cse)
o3d.visualization.draw_geometries([point_cloud])

plane_params_ground, inlier_mask,outlier= fit_plane_ransac(pointcloud_cse)
```

```
filter_data = pointcloud_cse[outlier]
point_cloud = o3d.geometry.PointCloud()
point_cloud.points = o3d.utility.Vector3dVector(pointcloud_cse[inlier_mask])
o3d.visualization.draw_geometries([point_cloud])

inlier_mask1=fit_wall_ransac(pointcloud_cse, outlier)
point_cloud = o3d.geometry.PointCloud()
point_cloud.points = o3d.utility.Vector3dVector(filter_data[inlier_mask1])
o3d.visualization.draw_geometries([point_cloud])
```

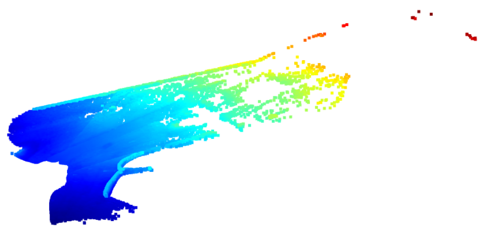
Figure 7: ground plane

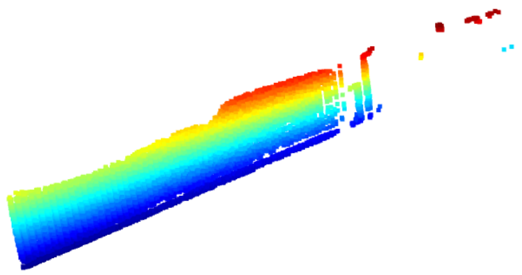
Figure 8: wall plane

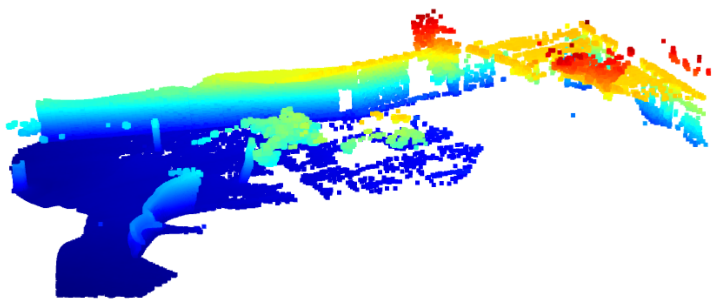
Figure 9: Point cloud of CSE building