In [1]:
```
import open3d
import numpy as np
import util
```

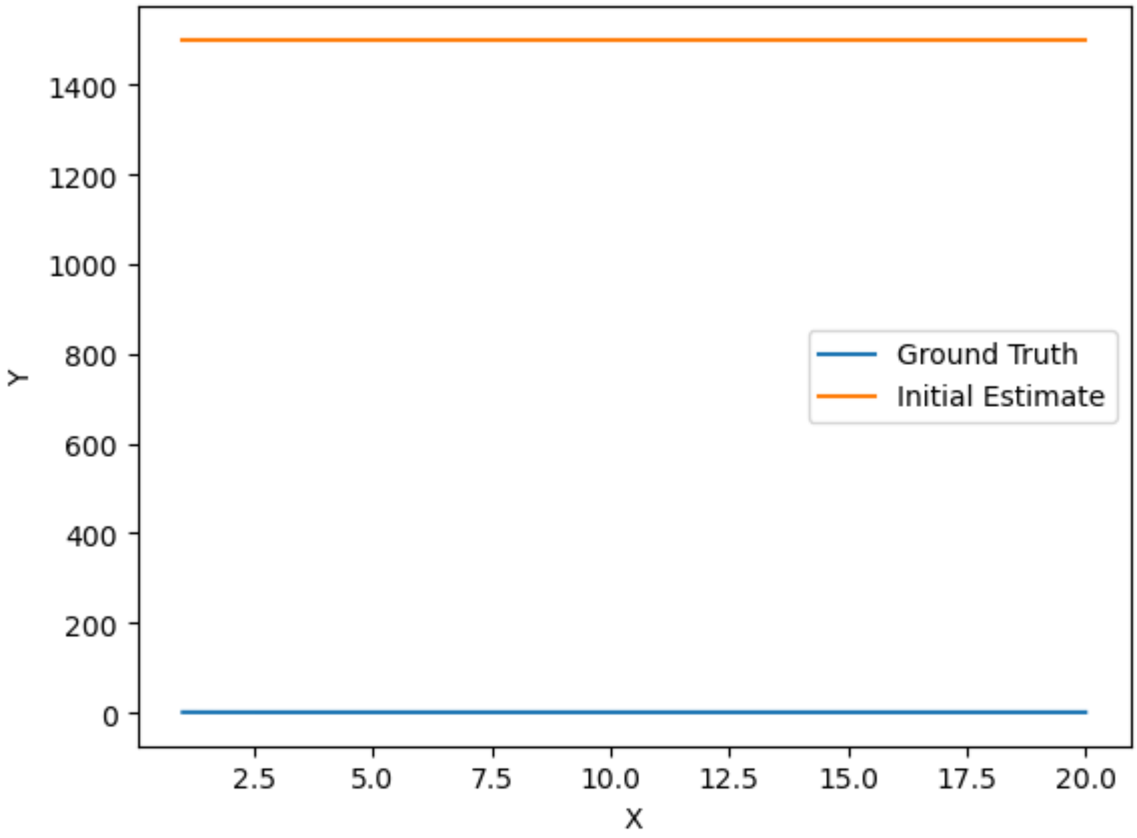
```
Jupyter environment detected. Enabling Open3D WebVisualizer.
[Open3D INFO] WebRTC GUI backend enabled.
[Open3D INFO] WebRTCWindowSystem: HTTP handshake server disabled.
```

1. Optimization: You are given the function: exp(-a *x)* sin(x) + b. Implement Levenberg Marquadt using numpy and solve for the parameters of the above function. Optimize for the following parameters: a=2, b = 1. Do this for 50 observations that lie between 1 and 20. Plot the loss values over time and data fit curves. Ensure that your initial estimates are not very close to the final parameters. Write down the jacobian formula in the notebook. **[3 points]**

In [2]:
```
import matplotlib.pyplot as plt
```

In [3]:
```
def make_curve(x, a, b):
    y = (np.exp(-a*x) * np.sin(x)) + b
    return y
```

In [4]:
```
[a_observed, b_observed] = [2, 1]          # Ground Truth: a=2, b=1
[a_estimate, b_estimate] = [10, 1500]      # Ground Truth: a=10, b=1500

xobs = np.linspace(1, 20, 50)
yobs = []
yest = []

yobs = make_curve(xobs, a_observed, b_observed)
yest = make_curve(xobs, a_estimate, b_estimate)

plt.clf()
plt.plot(xobs, yobs)
plt.plot(xobs, yest)
plt.xlabel("X")
plt.ylabel("Y")
plt.legend(['Ground Truth', 'Initial Estimate'])
plt.show()
```



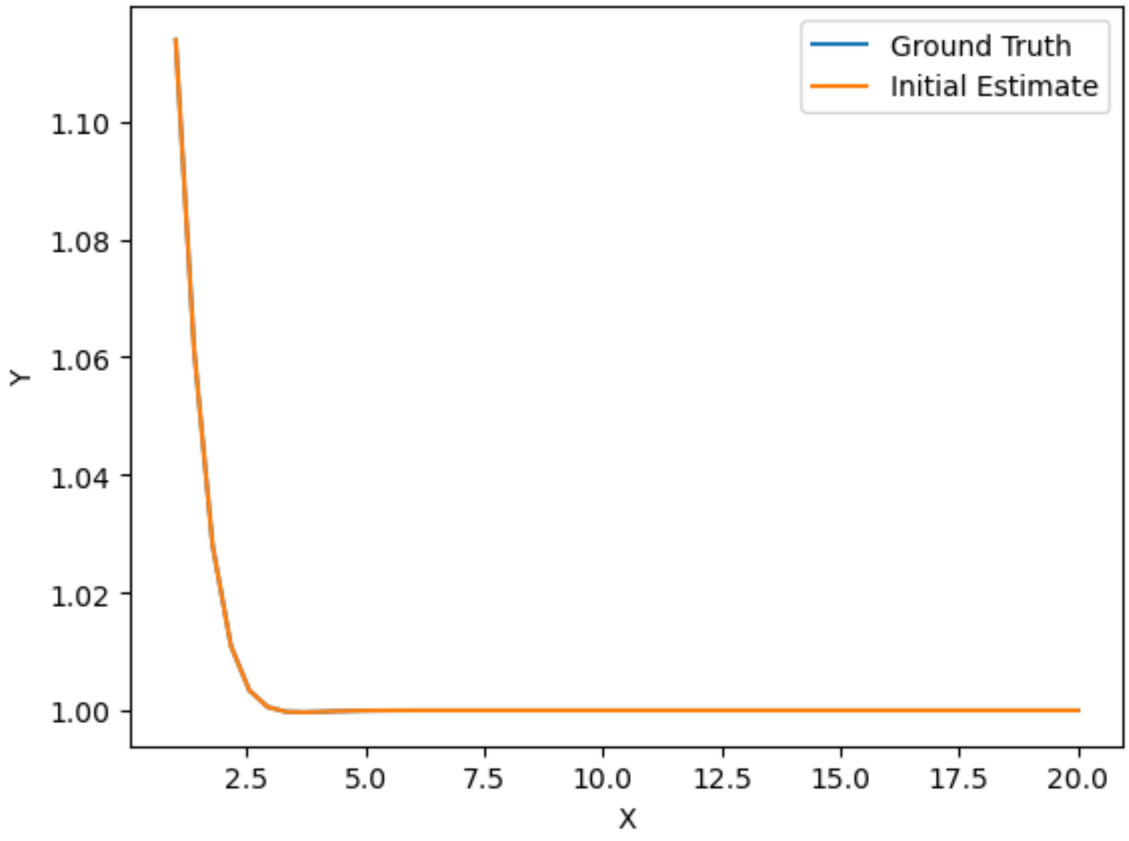
In [5]:

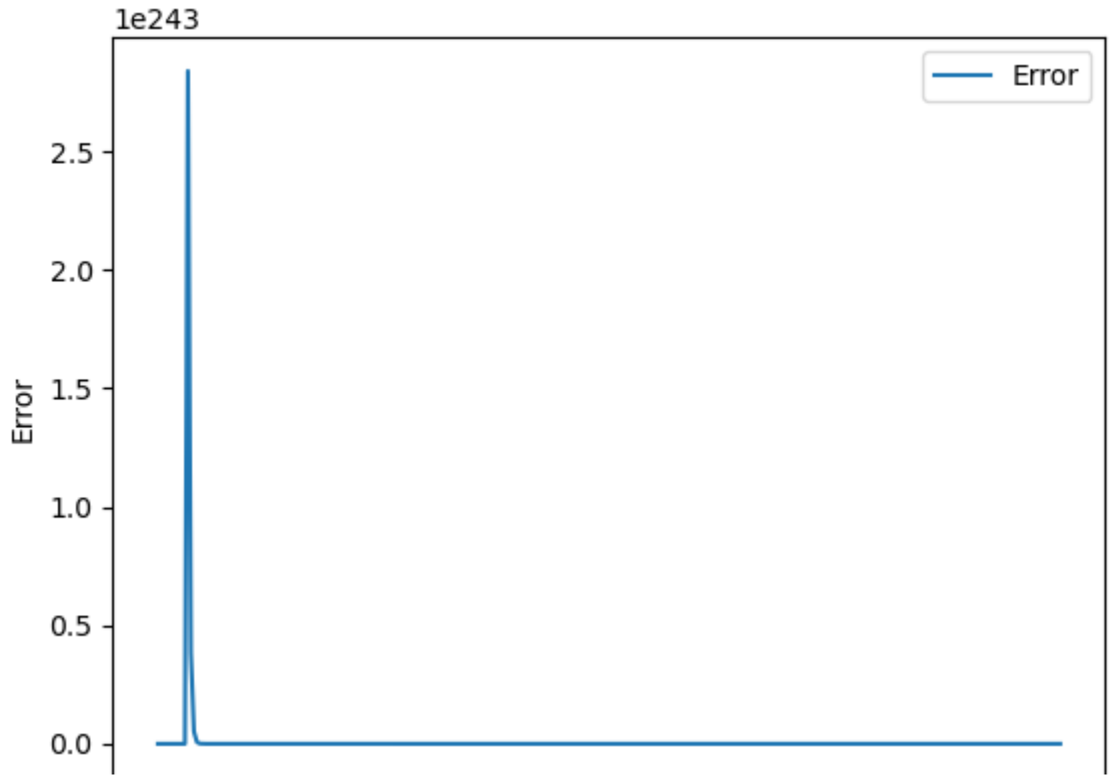
```
def a_der(x, a, b):
    y = -1 * (np.exp(-a*x) * np.sin(x)) * x
    return y

def b_der(x, a, b):
    y = np.ones(x.shape[0])
    return y

def jacobian(x, a, b):
    return np.c_[a_der(x, a, b), b_der(x, a, b)]

def residual(x, y, a, b):
    return make_curve(x, a, b) - y
```

In [6]:
```
def levenberg_marquardt(xobs, yobs, a, b, lamda, tolerance, max_iterations

    total_iterations = max_iterations
    weights = np.array([a, b], dtype=np.double)
    errors = [np.linalg.norm(residual(xobs, yobs, weights[0], weights[1]))

    for _ in range(max_iterations):

        J = jacobian(xobs, weights[0], weights[1])
        R = residual(xobs, yobs, weights[0], weights[1])
        new_error = np.linalg.norm(R) ** 2

        weights = weights - np.linalg.inv((J.T @ J) + (lamda * np.eye(J.sha

        if len(errors) > 0:
            if new_error > errors[-1]:
                lamda = lamda * 2
            else:
                lamda = lamda / 3

        errors.append(new_error)

        if new_error<tolerance:
            total_iterations = _
            break

    return weights[0], weights[1], total_iterations, errors
```

In [7]:
```
# Parameters
lamda = 1e-2
tolerance = 1e-50
max_iterations = 5000

a, b, tot_iter, errors = levenberg_marquardt(xobs, yobs, a_estimate, b_est
print(f"Total iterations: {tot_iter}")
print(f"After optimization a={a}, b={b}")

# Plotting the curves after fitting
plt.clf()
yest = make_curve(xobs, a, b)
plt.plot(xobs, yobs)
plt.plot(xobs, yest)
plt.xlabel("X")
plt.ylabel("Y")
plt.legend(['Ground Truth', 'Initial Estimate'])
plt.show()

# Plotting error vs iteration
plt.clf()
plt.plot(np.array(errors))
plt.xlabel("Iteration")
plt.ylabel("Error")
plt.legend(['Error'])
plt.show()
```

```
Total iterations: 300
After optimization a=2.000000000000004, b=1.0
```





Q) Write down the jacobian formula in the notebook.

Solution

Given function f(x) = e^{-ax} * sin(x) + b.

The parameters we are optimizing for are a and b .

Hence the Jacobian will be:

J = $[\frac{\partial f(x)}{\partial a} \quad \frac{\partial f(x)}{\partial b}]$

J = $[-e^{-ax} \cdot \sin(x) \cdot x \quad 1]$

2. Linear least square: You are given a bin file from the Kitti raw sequence. Estimate the ground plane from the given bin file. After estimating the ground plane, visualize this in open3d by drawing 200-300 points on the ground with a different color on top of the plot obtained from the LiDAR scan. Use RANSAC to estimate the ground plane. Will this work without RANSAC? Why or Why not? Write down the equation of the ground plane obtained and also mention the parameters used for doing RANSAC **[6 points]**

Expected result is displayed here:

In [8]:
```python
def read_bin_file(file_name):
    '''
    Read the bin file
    '''
    points = np.fromfile(file_name, dtype=np.float32).reshape(-1, 4)
    points = points[:,:3]                # exclude reflectance values, bec
    points = points[1::5,:]              # remove every 5th point for disp
    return points
```

In [9]:
```python
filename = "./data/000013.bin"
points = read_bin_file(filename)

# Function used to visualize point clouds, takes a list of 3 x N numpy arr
util.visualize_pointclouds([points.T])
```

In [10]:
```python
def equation_plane(points):

    [point1, point2, point3] = points
    [x1, y1, z1] = point1
    [x2, y2, z2] = point2
    [x3, y3, z3] = point3

    a1 = x2 - x1
    b1 = y2 - y1
    c1 = z2 - z1
    a2 = x3 - x1
    b2 = y3 - y1
    c2 = z3 - z1

    a = b1 * c2 - b2 * c1
    b = a2 * c1 - a1 * c2
    c = a1 * b2 - b1 * a2
    d = (- a * x1 - b * y1 - c * z1)

    return a/d, b/d, c/d
```

In [11]:

```
def plane_points(a, b, c):

    X = []
    Y = []
    Z = []

    x_temp = np.linspace(-80, 80, 25)
    y_temp = np.linspace(-80, 80, 25)

    for i in range(25): X.extend(x_temp)
    for i in range(25): Y.extend(y_temp)

    X = np.array(X)
    Y = np.array(Y)
    Y = np.sort(Y)

    Z = []

    for i in range(X.shape[0]):
        Z.append(a*X[i] + b*Y[i] + c)

    Z = np.array(Z)

    pts = np.array([X, Y, Z])
    pts = pts.T
    return pts
```

In [12]:
```
def _a_der(x):
    return x

def _b_der(y):
    return y

def _c_der(x):
    y = np.ones(x.shape[0])
    return y

def _jacobian(x, y):
    return np.c_[_a_der(x), _b_der(y), _c_der(x)]

def _residual(inliers, a, b, c):
    [x, y, z] = inliers.T
    return (a*x+b*y+c) - z

def leastsq_plane(inliers, lamda, tolerance, max_iterations):

    total_iterations = max_iterations
    weights = np.array([50, 50, 50], dtype=np.double)
    errors = [np.linalg.norm(_residual(inliers, weights[0], weights[1], we

    for _ in range(max_iterations):

        J = _jacobian(inliers[:,0], inliers[:,1])
        R = _residual(inliers, weights[0], weights[1], weights[2])
        new_error = np.linalg.norm(R) ** 2

        weights = weights - np.linalg.inv((J.T @ J) + (lamda * np.eye(J.sha

        if len(errors) > 0:
            if new_error > errors[-1]:
                lamda = lamda * 2
            else:
                lamda = lamda / 3

        errors.append(new_error)

        if new_error<tolerance:
            total_iterations = _
            break

    return weights
```

In [13]:
```
def ransac(points, p, e, s, delta):

    number_of_iterations = np.log2(1-p)/np.log2(1-(1-e)**s)
    number_of_iterations = number_of_iterations.astype(int) + 1
    print(f'Running {number_of_iterations} iterations of RANSAC.')
    max_inliers = -np.inf
    [a_res, b_res, c_res] = [0, 0, 0]

    for _ in range(number_of_iterations):

        index = np.random.choice(points.shape[0], 3, replace=False)
        points_random = points[index]
        [a, b, c] = equation_plane(points_random)

        dden = np.sqrt(a**2+b**2+c**2)
        distances = [abs((a*i[0]+b*i[1]+c*i[2]+1)/dden) for i in points]
        distances = np.array(distances)
        inlier_count = (distances<delta).sum()

        if max_inliers < inlier_count:
            max_inliers = inlier_count
            [a_res, b_res, c_res] = [a, b, c]

    inliers = []
    dden = np.sqrt(a**2+b**2+c**2)

    for i in points:
        if abs((a*i[0]+b*i[1]+c*i[2]+1)/dden) < delta:
            inliers.append(i)

    print(f'Gathered {len(inliers)} inliers.')

    return np.array(inliers)
```

Parameters of RANSAC are set in the cell below.

In [14]:
```
# Parameters for RANDAC
p = 1 - 1e-15     # probability of success
e = 0.4           # Outlier ratio
s = 3             # Number of points needed to fit the plane
delta = 0.85      # Margin to find inliers

inliers = ransac(points, p, e, s, delta)
```

```
Running 142 iterations of RANSAC.
Gathered 19323 inliers.
```

Equation of plane has been printed in the cell below.

In [15]:
```
# Parameters for least squares
lamda = 1e-2
tolerance = 1e-50
max_iterations = 50

[a1, b1, c1] = leastsq_plane(inliers, lamda, tolerance, max_iterations)
print(f'Equation of gound plane is z = {a1} x + {b1} y + {c1}')
```

```
Equation of gound plane is z = -0.0027129673169673514 x + 0.027163392491272
16 y + -1.7065516643087877
```

In [16]:
```
ground_points = plane_points(a1, b1, c1)
util.visualize_pointclouds([points.T, ground_points.T])
```

Will this work without RANSAC? Why or Why not?

No, the LiDAR scan includes points not belonging to the ground plane (such as trees, signboards, etc.) If we try to regress a plane on these points, the non-ground points will act as noise, and the plane obtained won't be the correct ground plane. So we should apply RANSAC (or some other outlier filtering algorithm) to filter out the outliers and then optimize for parameters of the plane based on the inlier points.

In [ ]:
```
```