Problem 1

$$V(t) = 1000 \sin \sqrt{\pi t}$$

By rearranging KVL, as shown in the problem sheet, we can describe the circuit with the differential equation and initial condition

$$f(t, q(t)) = \frac{dq(t)}{dt} = \frac{V}{R} - \frac{q(t)}{RC} \qquad\qquad q(0) = 4$$

where $R = 1000\Omega$ and $C = 0.002F$, q is measured in amperes and V is measured in volts.

a. The formula for Euler's method is $y_{n+1} = y_n + hf(x_n, y_n)$, where $f(x, y) = \frac{dy(x)}{dx}$. [1] In this problem, we have $y(x) = q(t)$ and $f(x, y) = f(t, q) = \frac{dq(t)}{dt}$. Thus, the iteration formula here becomes $q_{n+1} = q_n + hf(t_n, q_n)$.

   With $h = 0.1$, this expands to

$$\begin{aligned}
q_{n+1} =&\ q_n + h\left[\frac{V(t_n)}{R} - \frac{q_n}{RC}\right] \\
=&\ q_n + (0.1)\left[\frac{1000 \sin \sqrt{\pi t_n}}{1000} - \frac{q_n}{(1000)(0.002)}\right] \\
=&\ q_n + (0.1)\big(\sin \sqrt{\pi t_n} - q_n/2\big) \\
=&\ 0.95 q_n + 0.1 \sin \sqrt{\pi t_n}
\end{aligned}$$

   With $t_0 = 0$ and $q_0 = 4$, we get

$$\begin{aligned}
q(0 + 0.1) =&\ 0.95(4) + 0.1 \sin \sqrt{\pi(0)} \\
q(0.1) =&\ 3.8
\end{aligned}$$

[1] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. Numerical Recipes in C. Cambridge University Press.

b. Runge-Kutta's method is described by the equations [1]

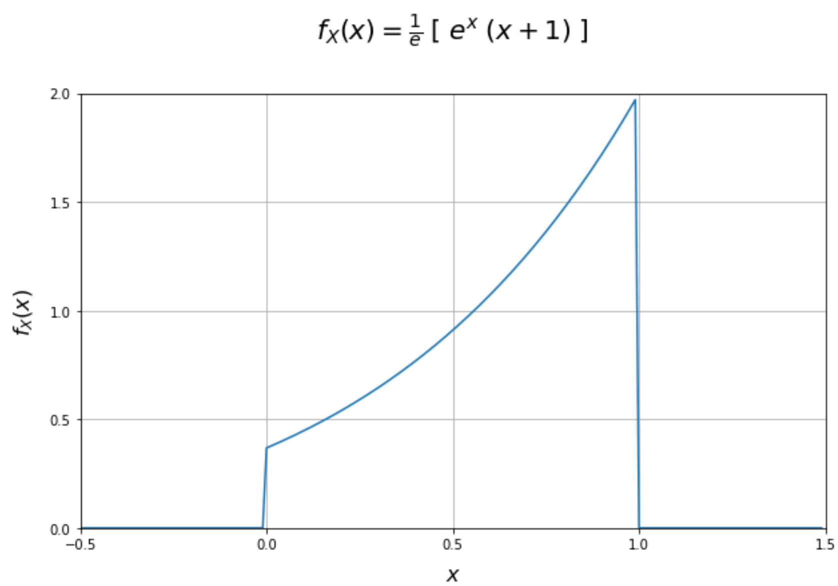
$$k_1 = hf(x_n, y_n)$$
$$k_2 = hf(x_n + \frac{h}{2}, y_n + \frac{k_1}{2})$$
$$k_3 = hf(x_n + \frac{h}{2}, y_n + \frac{k_2}{2})$$
$$k_4 = hf(x_n + h, y_n + k_3)$$

$$y_{n+1} = y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5)$$

I coded this calculation in Jupyter Notebook (see Appendix) to get $\mathbf{q(0.1)} = \mathbf{3.838887}$ coulombs.

Problem 2

Note: Code used to perform calculations for each method can be found in Appendix.



We know that $f_X(x) = 0$ outside of $[0, 1]$, so we are only interested in $E[x]$ within this region (if you considered the distribution from $(-\infty, \infty)$, we would get $E[x] = 0$). Further, $E[x]_{[a,b]} = \frac{1}{b-a}\int_a^b f_X(x)dx$. On the closed unit interval, $a = 0$ and $b = 1$, so $\frac{1}{b-a} = \frac{1}{1} = 1$, and the expected value of $X \in [0,1]$ is equal to the integral of $\int_0^1 f_X(x)dx$. Each of the following methods estimates that interval, and therefore directly estimates the expected value of X on $[0,1]$.

For all the following methods, we break the interval $[0,1]$ up into segments of length $h = 0.1$. Let N denote the number of segments of length h into which $[0,1]$ is divided. Then

$$N = \left\lfloor \frac{(1-0)}{h} \right\rfloor - 1 = \left\lfloor \frac{1}{h} \right\rfloor - 1$$

.

Note, this implies $(N+1)h = 1$.

a. **Rectangular method**

Here I will use the **right** rectangular method. The integral is estimated by breaking the function into N rectangles of base h and height $f_X(x)$. Because this is the right rectangular method, the height is taken as the value of the function at the right x value of the rectangle.

Given step-size h, the area of the rectangle with leftmost endpoints located at $x = x'$ is

$$A_{RR}(x') = h\ f_X(x' + h)$$

.

$$E[x] \approx \sum_{n=1}^{N+1} A_{RR}(a + nh) = h\,[\,f_X(h) + f_X(2h) + ... + f_X(1)\,]$$

$$\approx 1.0835$$

We know that this is an overestimate because the right rectangular sum overestimates the integral of a monotonically increasing function, which is the case for $f_X(x)$ on $[0,1]$.

b. **Midpoint method**

The midpoint method is very similar to the rectangular method. The only difference is that the height of the rectangle is chosen to be the value of the function $f_X(x)$ where x is located at the horizontal midpoint of the rectangle, instead of the right or left side.

Given step-size h, the area of the trapezoid with leftmost endpoints located at $x = x'$ is

$$A_{MP}(x') = h\ f_X(x' + \frac{h}{2})$$

.

$$E[x] \approx \sum_{n=0}^{N} A_{MP}(a+nh) \;=\; h\, f_X\Big(\,(\,n+\frac{1}{2}\,)\,h\Big)$$

$$= h\,\big[\, f_X(\,\frac{h}{2}\,) \;+\; f_X(\,\frac{3h}{2}\,) \;+\; ... \;+\; f_X(\,1-\frac{h}{2}\,)\,\big]$$

$$E[x] \approx 0.9991$$

As expected, the midpoint estimate is close to the right rectangular estimate, but is slightly lower. For a monotonically increasing function, the estimates A will always follow the pattern $A_{left} \leq A_{midpoint} \leq A_{right}$.

c. **Trapezoidal method**

In the trapezoidal method, instead of breaking the area under the curve into rectangles with a height chosen by an endpoint or midpoint of each interval, we instead break it into trapezoids, where the bottom two vertices of the trapezoid lies on the x-axis at the endpoints of the interval x' and $x'+h$, and the remaining two vertices lie at $f_X(x')$ and $f_X(x'+h)$.

The area of the trapezoid with leftmost endpoints located at $x = x'$ is

$$A_{TP}(x') = h\;\frac{f_X(x') + f_X(x'+h)}{2}$$

.

$$E[x] \;\approx \sum_{n=0}^{N} A_{TP}(a+nh) \;\;=\;\; \frac{h}{2}\,\big(\,f_X(nh) + f_X(n+1)\,\big)$$

$$= \frac{h}{2}\,\Big[\;\big(\,f_X(0) + f_X(h)\big) \;+\; \big(\,f_X(h) + f_X(2h)\big) \;+\; ... \;+\; \big(\,f_X(\,Nh\,) + f_X(1)\,\big)\,\Big]$$
$$= \frac{h}{2}\,\big[\;f_X(0) + 2f_X(h) + 2f_X(2h) + ... + 2f_X(Nh) + f(1)\,\big]$$

$$E[x] \approx 1.0019$$

Problem 3

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

Analytical Solution

$$\begin{aligned}
\int (1-y)dy &= \int \frac{1}{x^2} dx \\
y - \frac{1}{2}y^2 &= -\frac{1}{x} + c \\
\frac{1}{2}y^2 - y &= \frac{1}{x} + c
\end{aligned}$$

Given $y(1) = -1$:

$$\frac{1}{2}(-1)^2 - (-1) = \frac{1}{(1)} + c \quad \longrightarrow \quad c = \frac{1}{2}$$

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

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

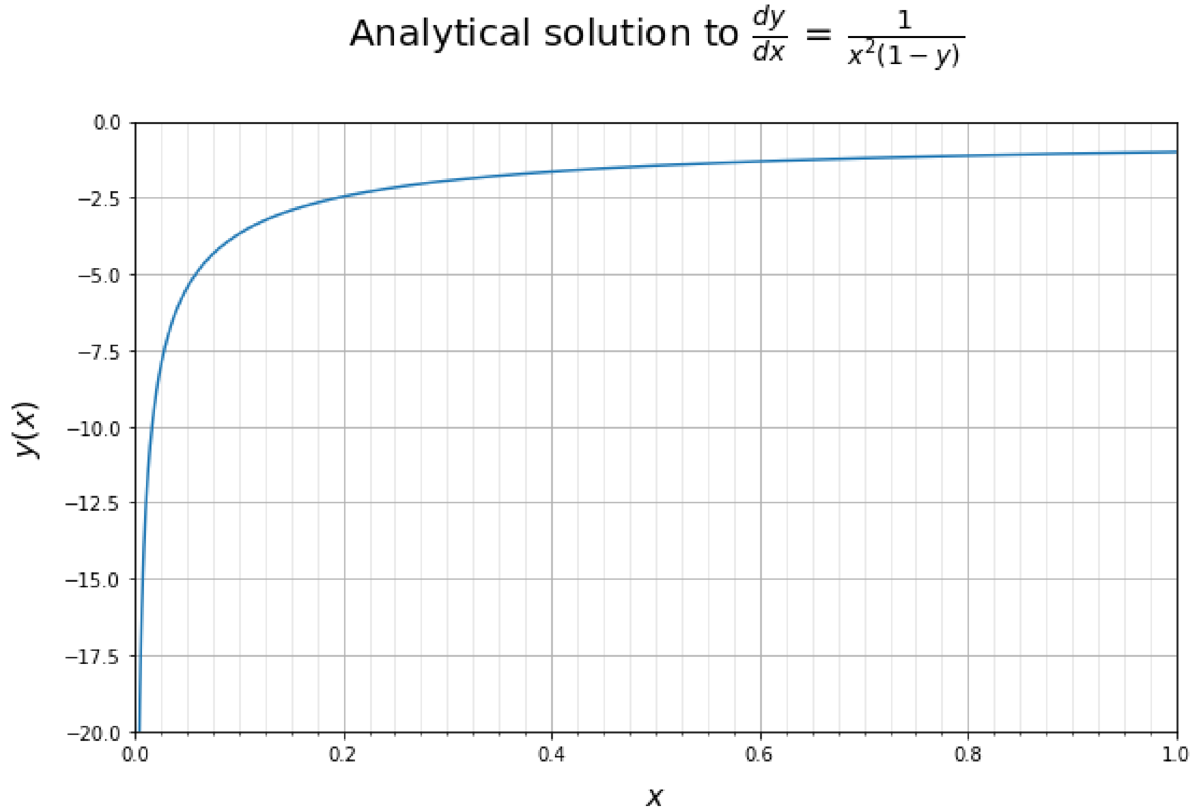
$$y^2 - 2y + 1 = \frac{2}{x} + 2$$

$$(y-1)^2 = 2\left(\frac{1}{x} + 1\right)$$

$$y - 1 = \pm\sqrt{2\left(\frac{1}{x} + 1\right)}$$

$$y(x) = 1 \pm \sqrt{2\left(\frac{1}{x} + 1\right)}\,, \qquad x \notin (-1, 0\,]$$

If we only want the case that satisfies $y(1) = -1$, then our final solution is $y(x) = 1 - \sqrt{2\left(\frac{1}{x} + 1\right)}$.



Euler's Method

$$f(x, y) = \frac{1}{x^2(1-y)} \qquad \textbf{EM:} \;\; y' = y + hf(x, y)$$

Using Euler's Method with a step size of 0.05, I found that $y(0) \approx -8.125$. From the plot of my analytical solution above, we can see that as $x \to 0^+$, $y(x) = -\infty$, so this approximation is quite far off. However, just by eyeballing it, we can see that $y = -8.125$ at just under $x = 0.025$. Overall, it isn't a great description of the behavior at $x = 0$, but it estimates the behavior at a near-zero value fairly well.

4$^{\text{th}}$-order Runge-Kutta Method

Using the implementation from Problem 1 (see Appendix for code) and the definition of $f(x, y)$ from Euler's Method above, I found the estimate $y(0) \approx -7.068\text{e}26$. This is a *much* better estimate of the behavior of $y(x)$ approaching $-\infty$ as x approaches 0.

Richardson Extrapolation

Using $n = 2, 4, 6, ..., [n_j = 2j]$ as suggested in *Numerical Recipes*, I implemented a Richardson's Extrapolation (see Appendix for code) using $H = -0.05$. The program breaks each macro-step of size H into micro-steps of size $h = H/n$. n begins at 2, and the program runs the modified midpoint method to estimate the value of $y(x + H)$. n increases according to the aforementioned series until the estimate of $y(x + H)$ is within an acceptable error (10%) of the analytical solution. At $x = 0$, since the analytical solution has no finite value, I instead calculated an error estimate using the Lagrange interpolation-based error estimate table from the Lecture 8 slides.

With the method, the estimated the value of $y(0) \approx -7.4548e27$. This is even closer to the "true" value of $-\infty$ than the 4^{th}-order Runge-Kutta Method, with an estimate about 10 times the magnitude of RG.

Problem 4

a. I implemented a RANSAC algorithm to determine the dominant plane in a point cloud (The code for my implementation can be found in the Appendix). The program loosely works in the following steps:

   - Clean up the data by removing points with NaN values or those recorded at the origin.
   - Choose a random point p from the point cloud.
   - Find k-nearest neighbors of p and find the plane of best fit (I use Principal Component Analysis).
   - Determine the Sum of Squared Error (SSE) for the estimated plane, where the error of each point is the distance to the plane.
   - If the SSE is lower than the previous best estimate, this plane is the new best estimate.
   - Repeat until one of the following end conditions
     - all points in the cloud are sampled,
     - the chosen number of points are sampled,
     - the SSE is sufficiently low, or
     - the dominant plane remains unchanged for a sufficient number of iterations.

   In this implementation, I used the end condition in which the algorithm stopped after N points were sampled, for a sufficiently large $N = 1000$ determined through trial and error. In a more robust implementation, I would use one of the latter two end conditions.

   By finding the plane that minimizes the error for the largest number of points, RANSAC asymptotically finds the dominant plane in the point cloud. This lends the algorithm to ignore smaller planes from clutter on the table that only approximate a comparatively small number of points in the point cloud.

As proof of concept for my implementation, the normal vectors for the empty table and the cluttered table are very similar, with plane parameters listed below. I used the normal of the plane and a point on the plane to define the estimate, since the normal and centroid were already calculated as a part of my algorithm.

	Normal	**Centroid**
Empty	(0.0164, 0.8231, 0.5677)	(-0.2112, 0.3247, 1.2002)
Cluttered	(-0.0118, 0.8619, 0.5070)	(0.1480, -0.4149, 2.3279)

b. I extended my implementation to finding the major planes of a hallway by the following. When an iteration of RANSAC returned a plane A, I found all points $\mathbf{p_A}$ "supported" by that plane, i.e. with a small enough error w.r.t. the plane. These points were then removed from the point cloud, as they were considered to be "accounted for" by A.

The program repeats this process until it extracts a predetermined N planes or so many points are removed from the point cloud that there is little left to work with. In a more robust implementation, I would use similar end conditions as discussed in 4a, for example so that planes would be accepted until the SSE of an estimated plane was too large.

Using $N = 3$ to estimate the floor and two walls of the hallway, I found the plane parameters:

	Normal	**Centroid**
Plane 1	(-0.0399, 0.8192, 0.57217543)	(0.1387, -0.1489, 1.7127)
Plane 2	(0.0595, 0.8378, 0.5427)	(-0.8549 , -0.3953, 2.2356)
Plane 3	(0.0117, 0.8808, 0.4733)	(0.2849, 0.0190, 1.3528)

Appendix

[CODE] Problem 2

Python code written and executed in Jupyter Notebook.

Imports, function declarations, and parameters for parts a-c.

```
import numpy as np

###### FUNCTION DECLARATIONS
def fX(x):
    ''' Distribution function.
    Only works on single elements because of how the if statement
    is written. '''
    if 0 <= x <= 1:
        return 1/e * exp(x) * (x+1)
    else:
        return 0

def array1D_fX(x):
    ''' Broadcast fX(x) elementwise through 1D array.'''
    return np.array([fX(elem) for elem in x])

###### GLOBAL PARAMETERS
h=0.1
X = np.arange(0,1,h) # properly spaced x values for summations, [0,1)

```

2a. Rectangular Sum

```
summation = 0
for x in X:
    summation += fX(x+h) # add h to do /right/ rectangular method

right_rect_est = summation * h
right_rect_est
## [Out] 1.083492405630938

```

2b. Midpoint Sum

```
summation = 0
for x in X:
    summation += fX(x+h/2)

midpoint_est = summation * h
midpoint_est
## [Out] 0.9990569948560943

```

2c. Trapezoidal Sum

```
summation = 0
for x in X:
    summation += 1/2 * (fX(x) + fX(x+h))

trapezoid_est = summation * h
trapezoid_est
## [Out] 1.0018863776895104
```

[CODE] Problem 3

Python code written and executed in Jupyter Notebook.

Imports & function declarations

```
import numpy as np

###### FUNCTION DECLARATIONS
def y_a(x):
''' Analytical solution '''
    return 1 - sqrt(2/x+2)

def dydx(x,y):
''' Diff EQ '''
    return 1/((x**2)*(1-y))
```

Euler's Method

```
def f(x,y):
    return dydx(x,y)

def EM_step(x,y,step):
    ''' One step of Euler's Method '''
    return y + step*f(x,y)

step = -0.05
x = 1
y = -1
for x in np.arange(1,0,step):
    y = EM_step(x,y,step)
y
## [Out] -8.124934344340135
```

4th Order Runge-Kutta Method

```
# Runge-Kutta method
step = -0.05
x = 1
y = -1
for x in np.arange(1,0,step):
    y = RK4(x,y,step) # RK4 function defined in Problem 2 code

y
## [Out] -7.068182004270559e+26

```

Richardson Extrapolation

```
def RE(x,y,H,n=2):
    ''' Run Modified Midpoint Estimation for n micro-steps of H. '''
    h = H/n # inner step-size

    # First mini-step, Modified midpoint method
    z0 = y
    z1 = z0 + h*f(x,z0)
    # initialize iterating variables
    zm = z1
    zlast = z0

    for m in range(1,n):
        znext = zlast + 2*h*f(x+m*h,zm)
        # Update for incrementing m
        zlast = zm
        zm = znext

    # m=n, and zm and zlast have been updated by last lines of loop
    yn = 1/2 * ( zm + zlast + h*f(x+H,zm) )
    return yn

################################################

# Parameters and Macros
x0 = 1
y0 = -1

x = x0
yn = y0
n = 2
H = -0.05

xgoal = 0

ERROR_BOUND = 1
ITERS_MAX = 1000

################################################
```

```python

    # Iteration

    iters = 0
    while np.abs(x-xgoal) > np.abs(H/2): # while we are greater than a macro-step
                                         # from the goal (halved to avoid
                                         # floating point error)
        x = x+H
        y = yn
        n = 2

        # Run for n=2 to initialize error table
        yn = RE(x=x,y=y,H=H,n=2)
        # Start extrapolation table
        ex_table = []
        ex_table.append(np.empty([n+2],float))
        ex_table[0][0] = yn

        try:
            ERROR_BOUND = np.abs(y_a(x))/20 # est should be w/i 5% of solution
            error = np.abs(y_a(x)-yn)
        except ValueError:
            # Will occur when x=0, since y_a(0) divides by zero -- instead just
            # use previous step's final error to bound
            ERROR_BOUND = error
            # Initialize error as infinitely high, since we need to run another
            # modified midppoint in order to estimate error from
            # extrapolation table
            error = float('inf')

        while error > ERROR_BOUND and iters < ITERS_MAX:
            n += 2
            yn = RE(x=x,y=y,H=H,n=n)
            if x > 0.001: # If x is positive nonzero, y(x) should be finite,
                          # so we can check our estimate against the
                          # analytical solution directly.
                error = np.abs(y_a(x)-yn)
            else:
                # Extrapolation table to determine error
                ex_table.append(np.empty([n,1]))
                for j in range(1,n-1):
                    # calculate dif between values at j-1 and j in last two rows
                    dif = (2*ex_table[-1][j-1]-ex_table[-2][j-1]) / (n/(n-2)**2 - 1)
                    # assign value at j in last row
                    ex_table[-1][j] = ex_table[-1][j-1] + dif
                # Difference between consecutive columns in final row
                # is error estimate
                error = np.abs(dif[0])

            iters += 1

    yn
    ## [Out] -7.454836751011965e+27

```

[CODE] Problem 4

Python code written and executed in Jupyter Notebook.

Imports & helper methods

```
import numpy as np
import pandas as pd
from scipy.spatial import KDTree

def read_cloud(filename):
    ''' Read in pointcloud from .pcd file using pyntcloud module,
    extract points as pandas dataframe and return dataframe only.

    Reference:
    pyntcloud - Python PointCloud module
    Copyright HAKUNA MATATA
    Project page: https://pyntcloud.readthedocs.io/en/latest/index.html'''

    import pyntcloud as pc
    return pc.PyntCloud.from_file(filename).points



def drop_null_pts(df, reset_index = True):
    ''' Filter out rows that contain NaN or are (0,0,0)
    (per Piazza discussion, these "represent points too far away or too close")
 '''
    df = df.dropna(how = 'all')
    df = df.drop(df[(df['x'] == 0) & (df['y'] == 0.0) & (df['z'] == 0.0)].index
)

    if reset_index:
        df = df.reset_index(drop=True) # Re-number points with null points
removed

    return df


```

Algorithm submethods

```python
def extract_plane(pts):
    ''' Fit a plane to a numpy array of points with PCA '''
    # Find and subtract out the centroid
    centroid = np.mean(pts,axis=0)
    pts -= centroid

    # Find the covariance matrix for the points
    cov = np.cov(pts,rowvar=False)

    # Find eigen-decomposition of cov matrix
    eigenvals, eigenvecs = np.linalg.eig(cov)

    # Make sure they are sorted from highest eigenvalue to lowest
    order = eigenvals.argsort()[::-1]
    eigenvals = eigenvals[order]
    eigenvecs = eigenvecs[:,order]

    # 3rd eigenvector is normal to the plane
    normal = eigenvecs[:,2]

    return normal, centroid



def calculate_SSE(cloud, normal, p):
    ''' Determine sum of squared error of the point cloud w.r.t. the plane.

    The Squared Error of a point to the estimated plane is
                                    (N.x - N.p)^2
    where . is the dot product, N = eigenvecs[:,-1] (the normal to the estimated plane),
    and p = point on the plane.
    '''
    # Point-wise squared error
    SE = (cloud.dot(normal) - np.dot(normal,p) ) ** 2
    # Sum squared error
    SSE = SE.sum()
    return SSE
```

myRANSAC implementation

```
def myRANSAC(cloud, k=None, n_iters=None, return_SSE=False, return_stability=
False):
    ''' Use a RANSAC algorithm to find the dominant plane in a poin tcloud.

    Parameters
    ---------------
    cloud            -  Point cloud, Pandas dataframe of Nx3 points
    k                -  k for k-nearest neighbors tree
    n_iters          -  maximum iterations before returning the best plane
    return_SSE       -  flag to return Sum of Squared Error of plane over the
                        whole cloud
    return_stability -  a measure of how many iterations the best estimate
                        plane persisted for
    '''

    # If no k, use rule-of-thumb k=sqrt(num_points)
    if k is None:
        k = int(np.sqrt(len(cloud)))

    # Build a nearest-neighbors tree
    tree = KDTree(cloud,leafsize=k)

    # Initialize sum of squared errors
    best_SSE = float('inf')
    best_normal = None
    best_centroid = None

    # Get samples as numpy array
    if n_iters is None:
        # If no n_iters input, do the whole cloud
        sample = cloud.to_numpy()
    else:
        # Otherwise randomly sample n_samples points from cloud
        sample = cloud.sample(n=n_iters).to_numpy()


    # For each sample point, estimate local plane and estimate how well
    # it approximates the whole cloud
    for pt in sample:
        # Find k nearest neighbors
        dists, indices = tree.query(pt,k=k)
        NN = cloud.iloc[indices].to_numpy()

        # Extract best-fitting plane with PCA as a normal vector and point on
plane
        normal, centroid = extract_plane(NN)

        # Calculate support for this plane within entire cloud as SSE
        SSE = calculate_SSE(cloud,normal,centroid)

        # Compare SSE with previous best estimate and update estimate if better
        if SSE < best_SSE:
            best_normal = normal
```

```
                best_centroid = centroid
                best_SSE = SSE

                # Measure how long a best estimate has stuck around
                stability = 0
            else:
                stability += 1



        # Check if final plane is oriented towards camera -> flip if not
        # Assume camera is at (0,0,0), so we want the normal to be the
        # opposite direction of the offset of the plane (i.e. the centroid).
        if np.dot(normal,centroid) > 0:
            normal *= -1

        # Collect requested return values
        RETURN = [best_normal, best_centroid]

        if return_SSE:
            RETURN.append(best_SSE)
        if return_stability:
            RETURN.append(stability/len(sample))

        return RETURN
```

My extended multi-plane RANSAC implementation

```
    # Finding dominant planes in the hallway
    # Strategy: Find first dominant plane, catalog, remove "supporting points",
    #           repeat until end condition

    def pts_on_plane(cloud,normal,centroid,delta=None,kdelta=0.1):
        # Point-wise squared error
        SE = (cloud.dot(normal) - np.dot(normal,centroid) ) ** 2
        # Total squared error
        SSE = SE.sum()

        # If unspecified, delta (the cutoff for whether a point is
        # "close enough" to the plane) will be set to kdelta times the
        # average error
        if delta is None:
            delta = np.sqrt(SSE)/len(cloud) * kdelta

        # Collect indices of points within delta of estimated plane
        ERR = np.sqrt(SE)
        inds = ERR[ERR < delta].index.tolist()

        return inds


    def my_extended_RANSAC(cloud, k=None, n_iters=None, return_SSE=False, return_stability=False):

```

```
    planes = []
    MAX_PLANES = 3
    min_size = int(len(cloud) * 0.05)

    while(len(planes)<MAX_PLANES and len(cloud) > min_size):
        # Run myRANSAC to get dominant plane
        out = myRANSAC(cloud, k=k, n_iters=n_iters, return_SSE=return_SSE,
return_stability=return_stability)
        planes.append(out)

        # Find supported points and drop them
        pts = pts_on_plane(cloud,out[0],out[1], kdelta=0.25)
        cloud = cloud.drop(pts)

    return planes
```

Main()

```
# Filenames
empty     = 'Empty.pcd'
cluttered = 'TableWithObjects.pcd'
hallway   = 'Hallway1a.pcd'

# Import point clouds
empty_cloud     =  drop_null_pts(read_cloud(empty))
cluttered_cloud =  drop_null_pts(read_cloud(cluttered))
hallway_cloud   = drop_null_pts(read_cloud(hallway))


#Empty
empty_plane = myRANSAC(empty_cloud, k=10, n_iters=1000)
empty_plane
### [Out] [array([0.01643926, 0.82314469, 0.56772584]),
###             array([-0.2111812,  0.32468896, 1.2002    ], dtype=float32)]


#Cluttered
cluttered_plane = myRANSAC(cluttered_cloud, k=10, n_iters=1000)
cluttered_plane
### [Out] [array([-0.01183628,  0.86188197,  0.50697078]),
###             array([ 0.14802603, -0.4148992 ,  2.3279    ], dtype=float32)]

# Hallway
planes = my_extended_RANSAC(hallway_cloud, k=10, n_iters=1000)
planes
### [Out] [[array([-0.03992603,  0.81915883,  0.57217543]),
###             array([ 0.13874812, -0.14890206,  1.7126999 ], dtype=float32)],
###        [array([0.05949353, 0.83783288, 0.54267539]),
###             array([-0.8548542 , -0.39532846,  2.2356    ], dtype=float32)],
###        [array([0.01169389, 0.88082039, 0.47330613]),
###            array([0.28489506, 0.0190324 , 1.3528    ], dtype=float32)]]
```