# למידת נתונים במערכות זמן אמת

## פרויקט ניווט – סיכום פרויקט

## מועד הגשה: 31.08.23

---

נועם סגל 325725802

זהבה רוזנטל 326109501

---

### הישגים עיקריים

1. פיתוח אלגוריתם אופטימלי והוכחתו באופן מתמטי, למציאת החדר.
2. שימוש בפונקציה האמיתית $\frac{a}{\cos(x-\phi)}$ שמתארת כל מקטע ולא בקירוב פולינומיאלי (פרבולה).
3. התאמה מוצלחת של מרובעים לחדרים בסריקות שונות של המעבדה שהתקבלו מהרחפנים, וגם מהסימולטור.
4. מימוש אלגוריתם מקורב למציאת מיקום הדלת.

### סיכום של מה שעשינו הסמסטר

1. בשבוע הראשון (ניסינו הרבה דברים שלא עבדו):
   - הגדרנו וציירנו את הבעיה
   - הסברנו מתמטית את התנהגות הפונקציה הפולרית של הנקודות במלבן
   - מצאנו את הפונקציה האמיתית (Ground Truth) שמסבירה את התנהגות הגרף בכל מקטע
   - ציירנו מלבן בתצוגה פולרית ב־DESMOS לצורך הדגמה
   - ניסינו לאורך כל שבוע זה לעבוד עם 1 חלקי הנורמה במקום הנורמה הרגילה
   - ניסינו להתאים את הפונקציה הנ"ל בעזרת np.piecewise עם פתרון חישובי שמנסה למזער את פונקציית ההפסד בהינתן ניחוש ראשוני
   - ניסינו לבצע התאמת פורייה לגרף הפולרי
   - ניסינו לבצע התאמת wavelet לגרף הפולרי
   - ניסינו להתאים מרובע ישירות לנתונים הקרטזים בעזרת רגרסיה של NumPy
   - ניסינו להתאים PolyFit לגרף הפולרי, לגזור, ולקחת נקודות מקסימום/מינימום
2. בשבוע השני (ניסינו עוד דברים שלא עבדו אבל בתלת ממד):
   - עבדנו עם נקודות בתלת ממד
   - ניסינו להתאים *RBF* לנתונים ולקחת נקודות מקסימום/מינימום
   - ניסינו להשתמש ב*PCA* כדי להוריד לשני ממדים
   - פיצלנו את התלת ממד לשני מישורים מאונכים *XY*, *XZ*
   - על המישורים הנ"ל ניסינו להתאים *PolyFit*
3. בשבוע השלישי (ביקשנו עזרה במייל, קיבלנו שאלות אלגוריתמיות לפתרון):
   - פתרנו את השאלות שקיבלנו במייל, הוכחנו וניתחנו סיבוכיות

- ראינו שאם צפיפות הנקודות אחידה עבור המלבן אז היא אינה אחידה עבור הפולרי, כי ככל שהזווית גדלה ביחס לאנך מכסים יותר נקודות. לכן הפינות הכי צפופות
- ניסינו לחלק את המקטעים הפולריים של הצלעות לפי צפיפות ציר האיקס בנקודות הפולריות (ראינו שזה עובד רק עבור חדר ריבועי ולא מלבן כללי)
- קיבלנו ממך (דני) עצה בנוגע לאלגוריתם, התחלנו לעבוד על לכתוב אותו מתמטית

4. בשבוע הרביעי:
   - פיתחנו אלגוריתם חדש לבעיה, אופטימלי לחלוטין – הגענו אליו באמצעות שילוב של האלגוריתם לחלוקת הנקודות באופן אופטימלי לפי הצפיפות שלהן, לבין האלגוריתם שהתחלנו לעבוד עליו בשבוע הקודם אבל עם פונקציה יותר מדויקת מאשר פרבולה
   - ניתחנו את סיבוכיות האלגוריתם, ומצאנו דרך לממש אותו בזמן לינארי (עם גישת הפרד ומשול בשימוש באלגוריתם *SMAWK*. השימוש ב־*SMAWK* די ברור ברגע שרואים את האלגוריתם שלנו)
   - התחלנו לממש את האלגוריתם בפייתון בסיבוכיות ריבועית
5. בשבוע החמישי:
   - סיימנו לממש את האלגוריתם בפייתון
   - הרצנו אותו על נתונים סינתטיים עם מעט רעש וראינו שהוא עובד עליהם מעולה
6. בשבוע השישי:
   - מצאנו דרך להתמודד עם כמות גדולה של נתונים על ידי חלוקה ל$batches$, והרצת האלגוריתם בנפרד על כל $batch$ ולבסוף לקיחת ממוצע של התוצאות.
   - מימשנו שיטה זו בפייתון
   - כעת כשיכולנו להתמודד עם כמות גדולה של נקודות הרצנו את האלגוריתם על נתונים מהסימולטור (לוקח בערך 8 שניות) וגם מהרחפנים האמיתיים (לוקח בערך 1.5 שניות)
   - מצאנו בעיות ועשינו *Debugging*
   - מצאנו את נקודות החיתוך של הפונקציות בין המקטעים
   - החזרנו את הפתרון חזרה מפולרי לקרטזי ושרטטנו את המלבן המקורב באמצעות נקודות החיתוך שמצאנו בגרף הפולרי
   - פיתחנו אלגוריתם מקורב למציאת הדלת: נפטרנו מהנקודות שבתוך החדר, חילקנו כל צלע ל *Clusters* אופטימליים בסיבוכיות לינארית ע"י אלגוריתם *ckmeans.1d.dp*, וחישבנו את ה־*Sparseness* של כל *Cluster* לפי האורך שלו מנורמל במספר הנקודות שבו.

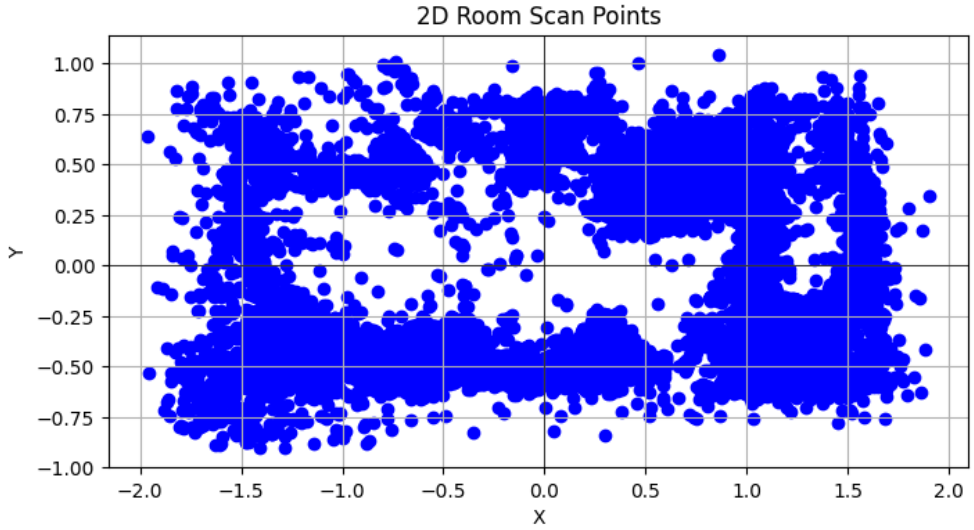
## התוצר הסופי

הקוד השלם מצורף במייל, וגם כאן בנספח בסוף המסמך.

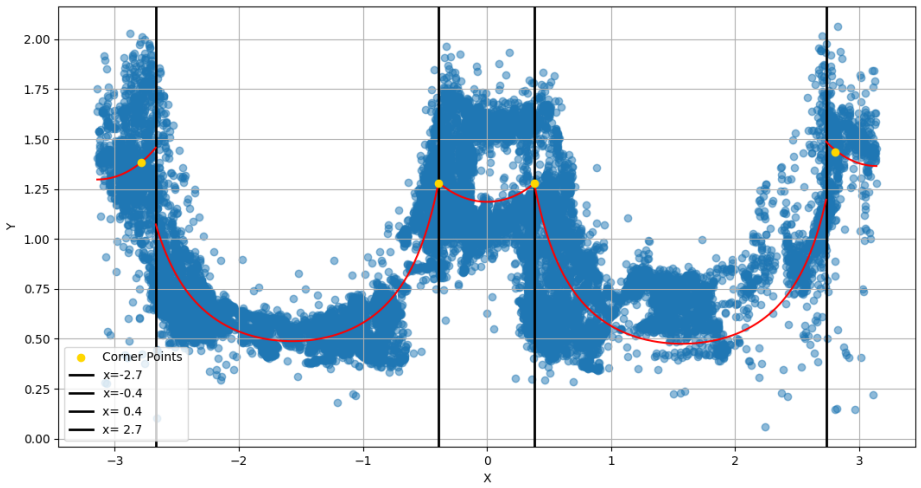
**להלן הגרפים שמראים את מציאת החדר בדו ממד** עבור החדרים האמיתיים והנתונים מהסימולטור:

<u>הנתונים מהסימולטור (*map.xyz*) מישור *XY*:</u>

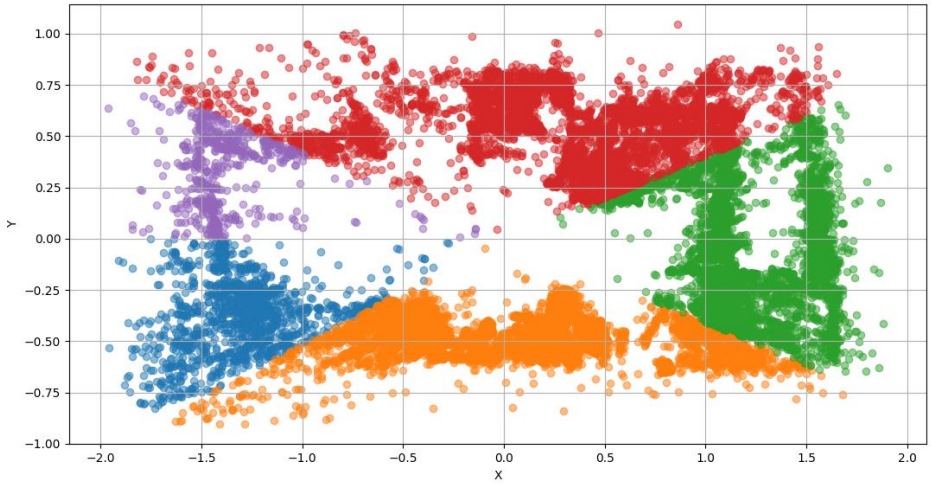
הנתונים המקוריים:



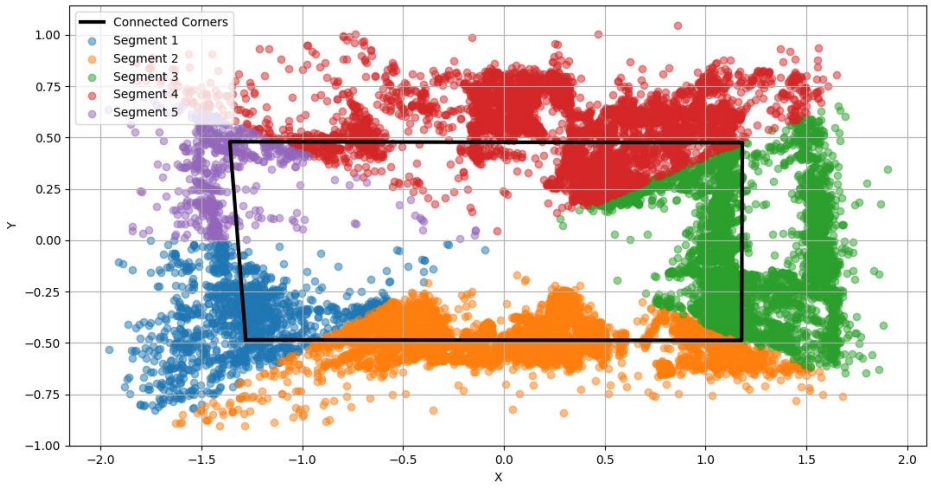

הנתונים הפולריים + חלוקה למקטעים + העברת הפונקציות + נקודות חיתוך:



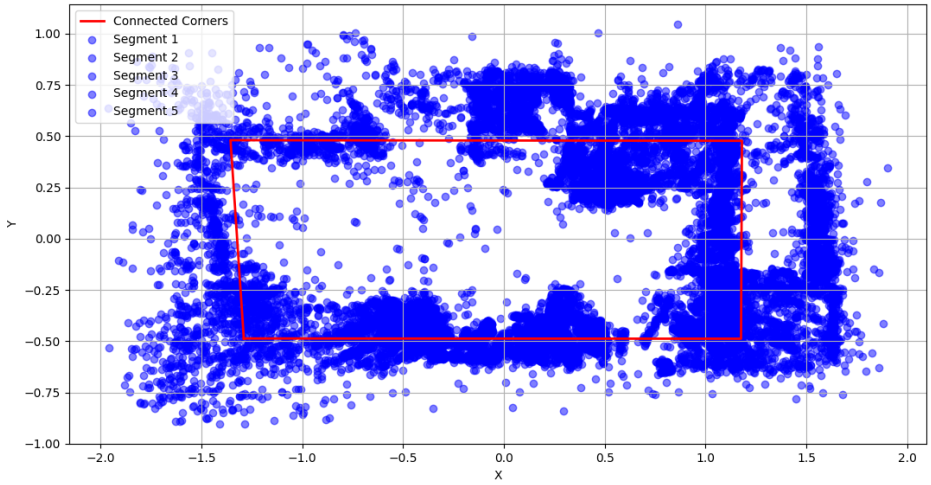

חלוקת הריבוע למקטעים שמצאנו בגרף הפולרי:




חלוקה לקטעים + חיבור פינות המלבן שמצאנו (לפי נקודות החיתוך בגרף הפולרי):

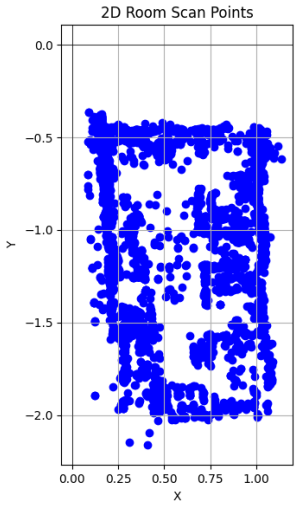



ההתאמה הסופית (בלי צביעת המקטעים):

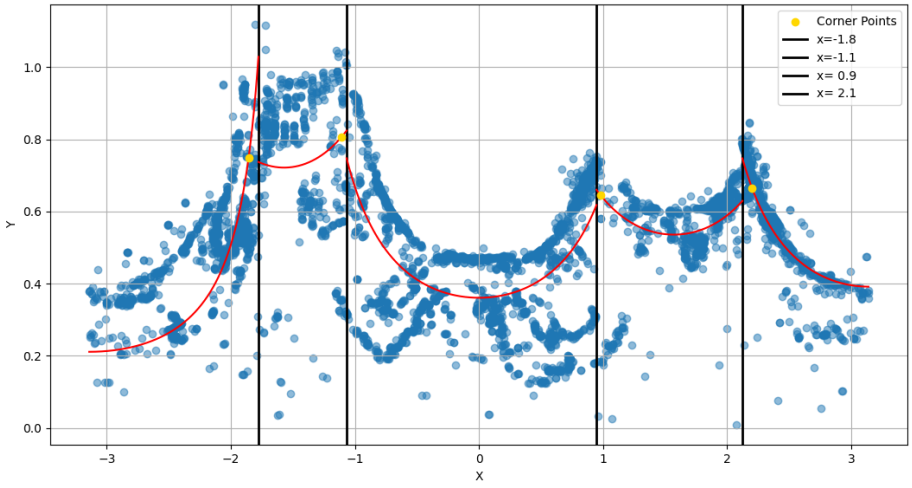

Connected Corners
Segment 1
Segment 2
Segment 3
Segment 4
Segment 5
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
Y
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
X

חדר 1 מישור *XZ*:

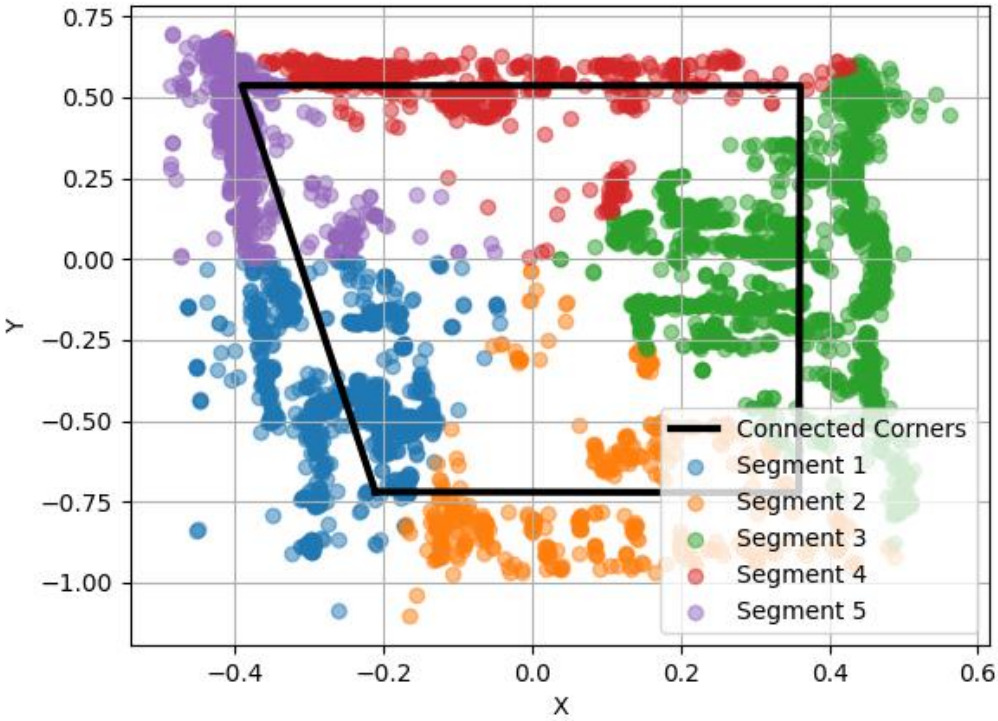
הנתונים המקוריים:




הנתונים הפולריים + חלוקה למקטעים + העברת הפונקציות + נקודות חיתוך:

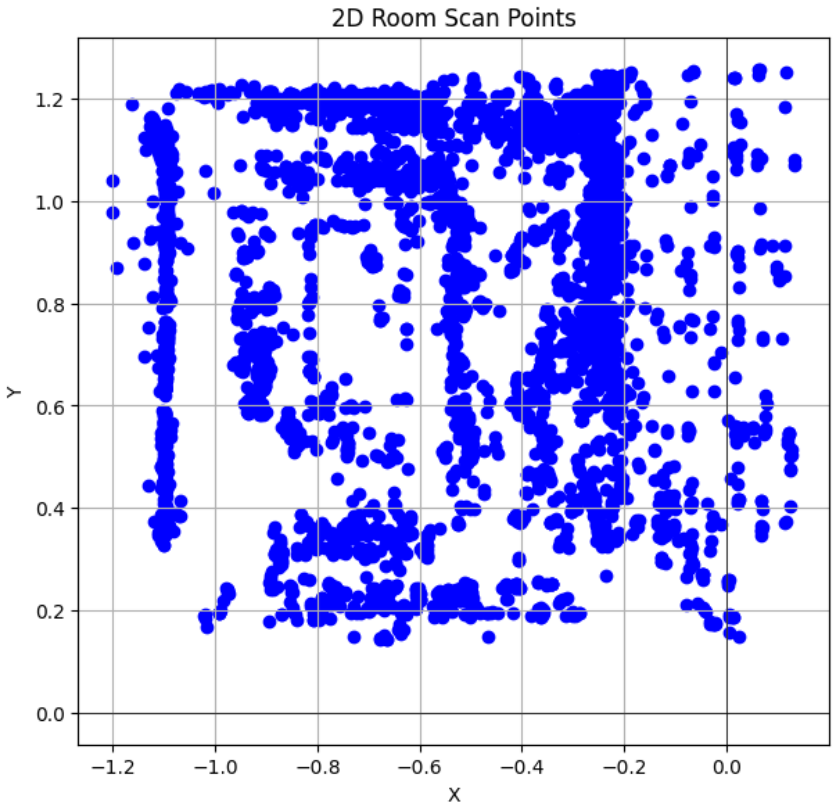



חלוקה לקטעים + חיבור פינות המלבן שמצאנו (לפי נקודות החיתוך בגרף הפולרי):

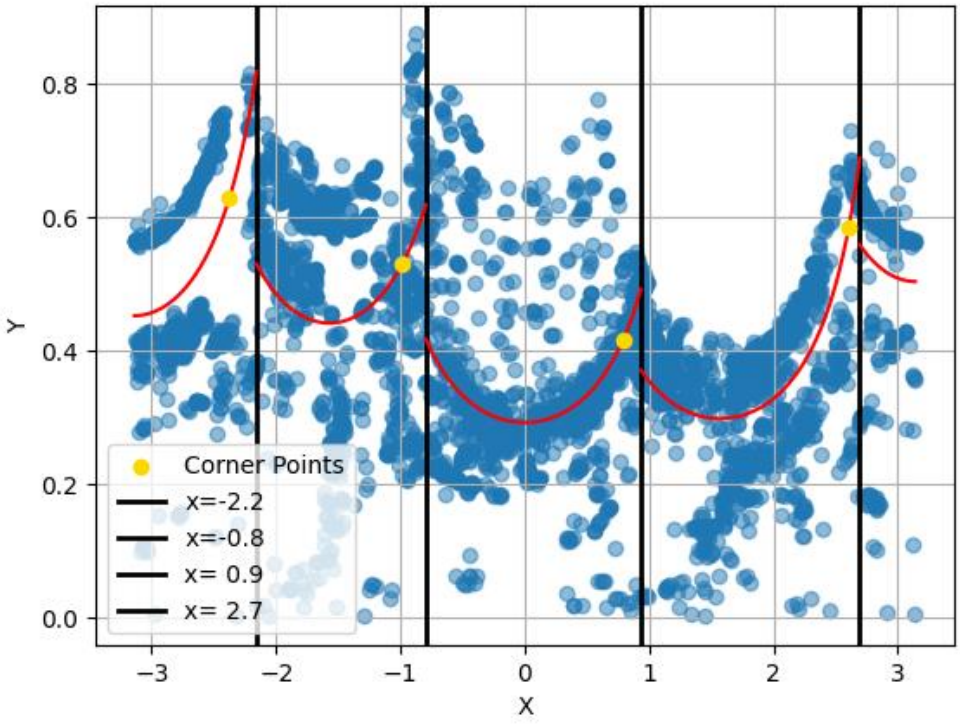

0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
Y
−0.4
−0.2
0.0
0.2
0.4
0.6
X
Connected Corners
Segment 1
Segment 2
Segment 3
Segment 4
Segment 5

חדר 4 מישור *XZ*:

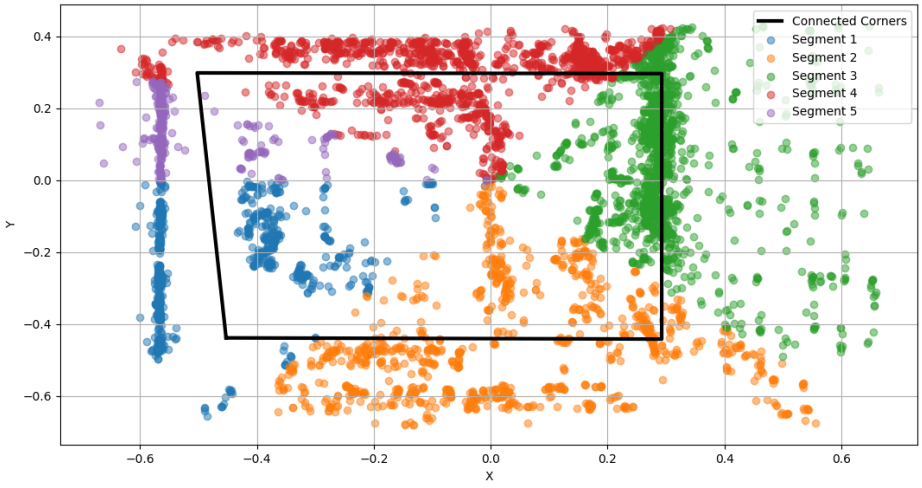
הנתונים המקוריים:




הנתונים הפולריים + חלוקה למקטעים + העברת הפונקציות + נקודות חיתוך:

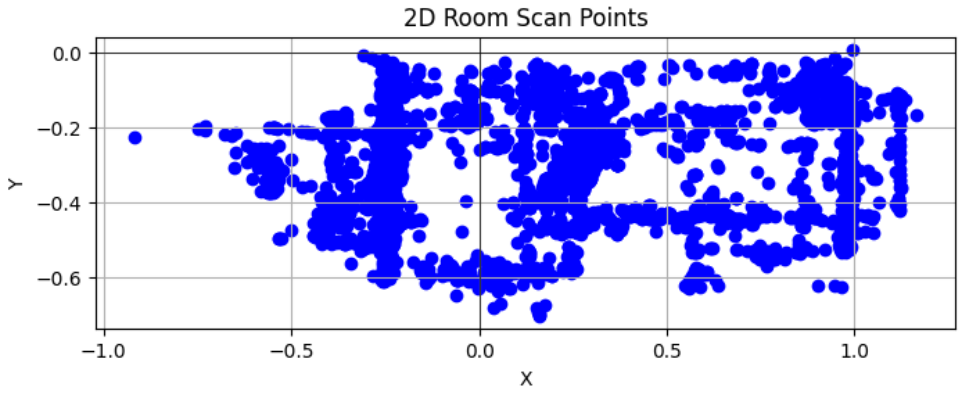



חלוקה לקטעים + חיבור פינות המלבן שמצאנו (לפי נקודות החיתוך בגרף הפולרי):

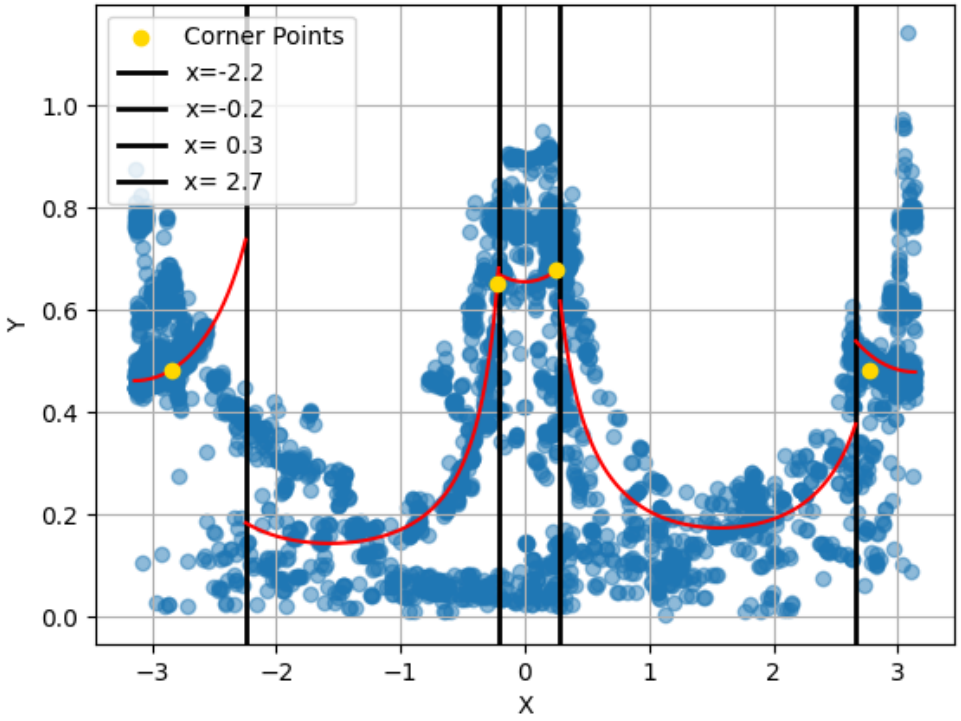


חדר 5 מישור *XY*:

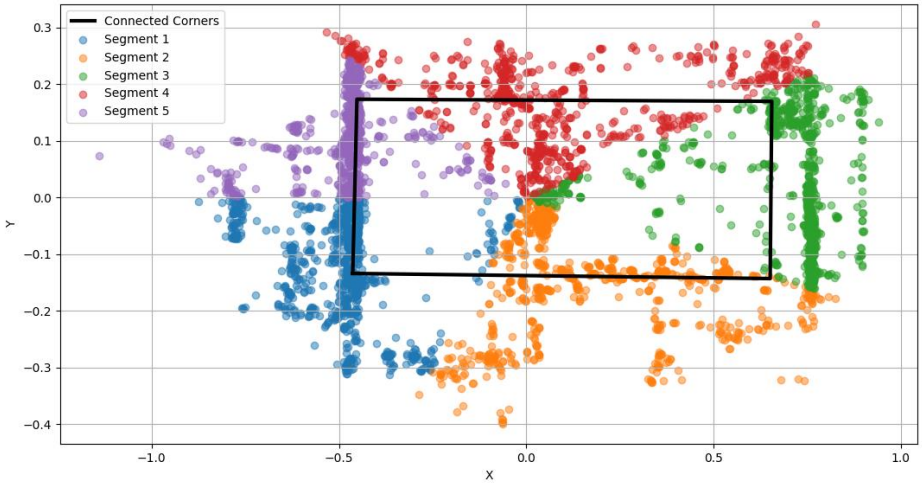
הנתונים המקוריים:




הנתונים הפולריים + חלוקה למקטעים + העברת הפונקציות + נקודות חיתוך:




חלוקה לקטעים + חיבור פינות המלבן שמצאנו (לפי נקודות החיתוך בגרף הפולרי):


Connected Corners
Segment 1
Segment 2
Segment 3
Segment 4
Segment 5
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
−0.4
Y
−1.0
−0.5
0.0
0.5
1.0
X

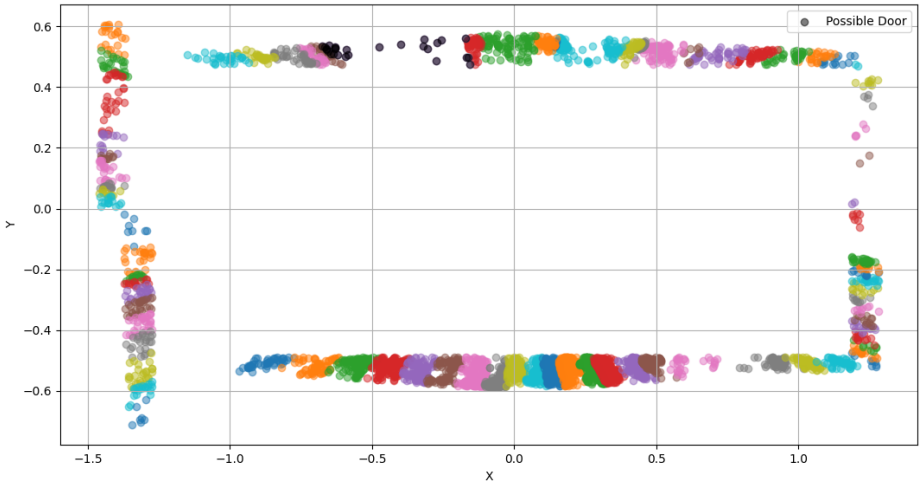
**ולהלן הגרפים שמראים את מציאת הדלת** עבור החדרים עבורם הספקנו להריץ:

אגב האלגוריתם:

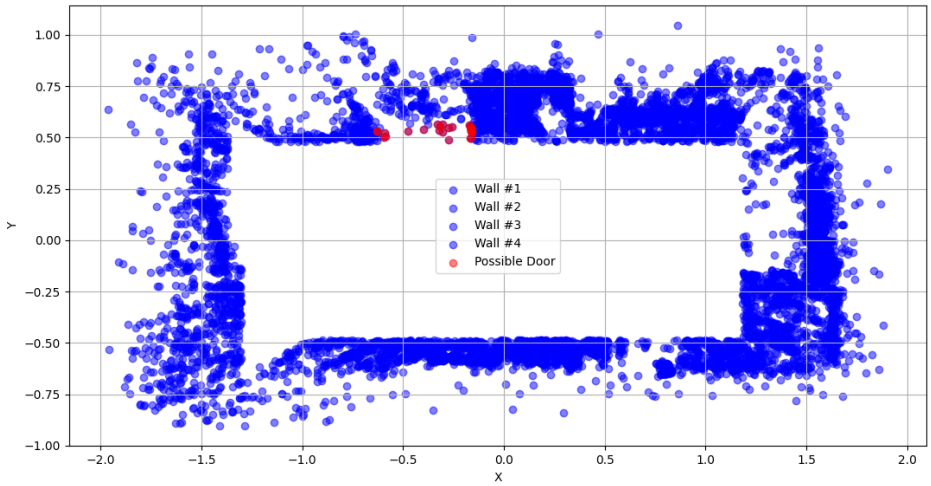
נפטרים מהנקודות שבתוך החדר, ורחוקות מהחדר. לכל צלע עושים קלאסטרינג חד ממדי לאורך הצלע ומחשבים כמה כל קלאסטר ריק. בסוף צובעים את הקלאסטר הכי ריק.

<u>עבור הסימולטור (*map.xyz*) מישור *XY*:</u>

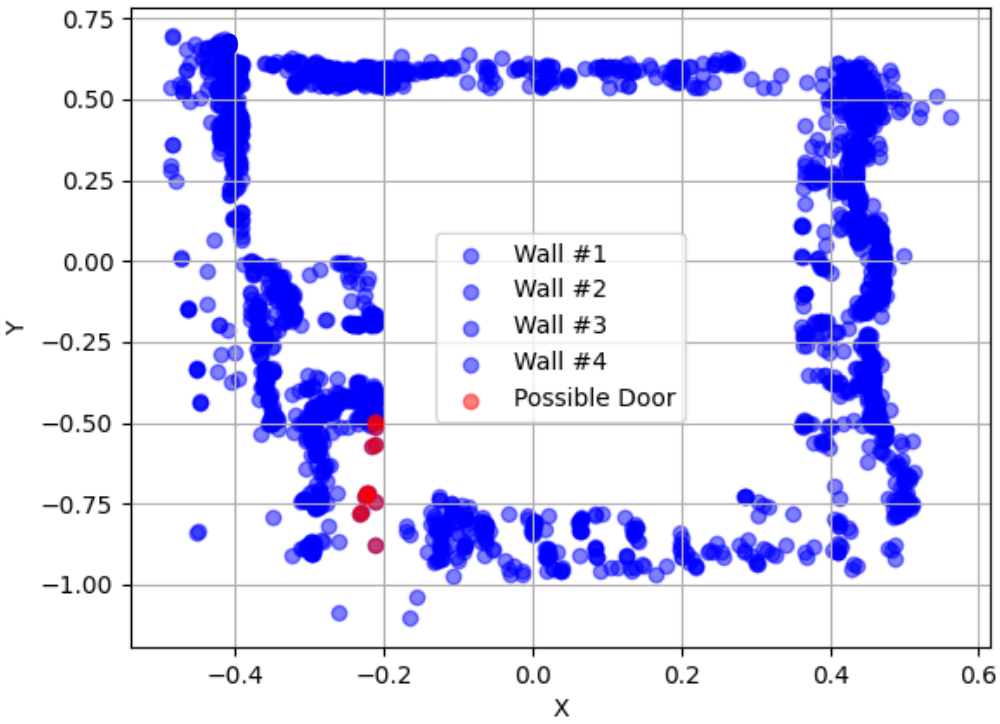
הקלאסטרים הצבועים לצורך הדגמה:



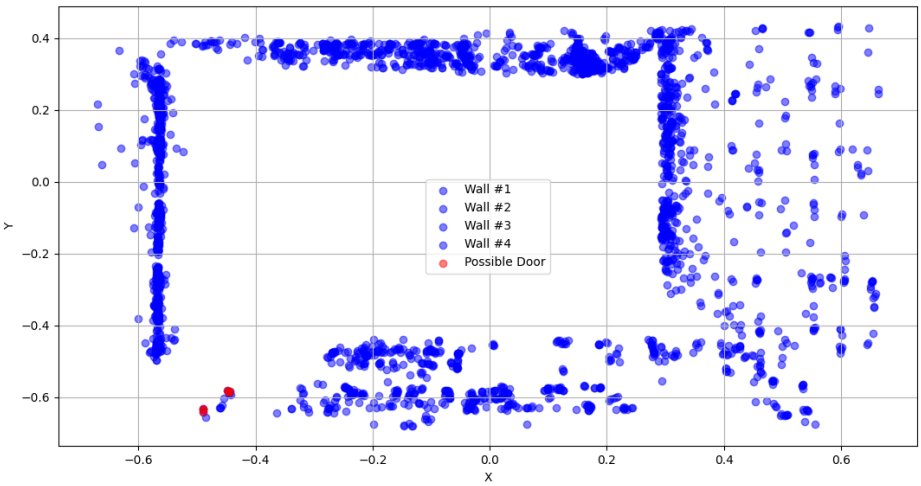

ורק הקלאסטר של הדלת (וללא הנקודות שבתוך החדר):



חדר 1 מישור *XZ*:




חדר 4 מישור *XZ*:



## אתגרים

בהתחלה התקשינו מאוד כי ניסינו לעבוד עם יוריסטיקות. ניסינו הרבה מאוד גישות שונות, וזה בזבז לנו הרבה זמן. אך על הצד החיובי, מאוחר יותר ניצלנו חלק מהקוד שכתבנו במהלך השבועיים הראשונים של התנסות זו. התקשינו גם במציאת פולינום מתאים לתלת ממד באמצעות היוריסטיקות האלה.

עוד אתגר היה עבודה בלחץ הזמן של סמסטר הקיץ עם עוד קורסים.

אתגר נוסף היה פתרון החידות שקיבלנו במייל ושימוש בהן בפרויקט, הן הובילו אותנו בסופו של דבר לעוד אתגר, שהיה פיתוח של אלגוריתם משופר למציאת החדר, ומימוש האלגוריתם התיאורטי בקוד שיכול גם לרוץ מהר.

קושי נוסף היה התמודדות עם כמות גדולה של נתונים. לשם כך היה צורך לעבוד הרבה על שיפור סיבוכיות מקום וזמן (עוד יש מקום לשיפור אבל זמן הריצה הנוכחי הוא 1.5 שניות בד"כ, תלוי בכמות הנקודות).

## מחקר עתידי

לדעתנו בהינתן עוד זמן, היינו מרחיבים את הפרויקט באופן הבא:

1. פתרון בתלת ממד, על ידי מציאת המלבנים של שני מישורים מאונכים XZ,XY (כפי שעשינו וצירפנו גרפים), ואז שקלול התוצאות לכדי פינות של תיבה (תלת ממדית) וזה לא אמור להיות קשה.
2. מימוש הקוד שלנו בסיבוכיות לינארית, על ידי החלפת הלולאה שמוצאת את המינימום בכל שורה במטריצה בשימוש באלגוריתם SMAWK שיכול למצוא את המינימום בכל שורה במטריצה ב־$O(1)$ עבור מטריצה "מונוטונית לחלוטין" (מתקיים במקרה שלנו).
3. הפחתת רעשים (למרות שהם לא מאוד הפריעו). או על ידי דגימת נקודות והיפטרות מהחציון/רבעון של נקודות הכי רחוקות מהמלבן שמצאנו (הצעת דני), או ע"י העברת אפרוקסימציה פשוטה כמו PolyFit (הצעת פארס).
4. שיפור האלגוריתם למציאת הדלת, אולי גם ממש ציור של מלבן, שהוא הדלת, ולא רק סימון של נקודות שנמצאות באזור של הדלת (מה שעשינו).
5. מציאת חלונות. זה שונה מדלת כי הדלת תמיד נמצאת על ההיקף של אחד המלבנים (ממבט עילי) ובחלונות זה לא מתקיים. בכך ניתן גם להבדיל בין דלת וחלון.

## במי (ובמה) נעזרנו

- קיבלנו הכוונה ממך (דני) שעזרה מאוד עם הגישה של העבודה שהיינו צריכים לסגל לעצמנו. הוכחה ופיתוח של אלגוריתם שהוא נכון, ורק אז להתחיל לתכנת.
- נעזרנו בפארס, שעזר לנו למצוא נתונים לעבוד איתם
- חלק משורות הקוד נכתבו תחילה על ידינו, ואז הכנסנו אותם ל 3.5-GPT כדי לראות אם ניתן לשפר את זמן הריצה שלהן.
- נעזרנו גם בויקיפדיה ו־Stack Exchange בשביל מידע תיאורטי בעיקר.
- קיבלנו טיפ לגביי מציאת הדלת מאליאס דאו, עשינו אחרת מהם אבל הטיפ עזר (הוא הציע קלאסטרינג, למרות שאנחנו עשינו קלאסטרים חד ממדיים אופטימליים ולא דו ממדיים יוריסטים כמוהם).
- התיעוד של פייתון והספריות הקשורות:
    - Numpy, MatPlotLib, CKMeans.1D.DP, Time, tqdm

## למי עזרנו

- נג'אח כמאל – עזרה עם הקוד (שלחנו לו קוד של PolyFit) וטיפים להתקדמות
- באסיל דבאח – שלחנו לו את הקוד (הסופי) של האלגוריתם שלנו למציאת המלבנים והקוד של ציור הגרפים.
- נטלי נקארה – עזרה בפיתוח האלגוריתם: לקחת את נקודות החיתוך של הפרבולות בתור הפינות; לא לחלק את ציר איקס לחלקים שווים כי זה לא בהכרח ריבוע. עזרה עם התאמת פונקציה לכל מקטע.
- יוגב קשת – טיפים במציאת אלגוריתם לחלוקת הפרבולות האופטימלית (כמה מקטעים לחפש, עזרה בהתאמת רגרסיה)

**נספח: הקוד**

```
import numpy as np  # linear algebra
import matplotlib.pyplot as plt  # data visualization
import matplotlib.patheffects as pe  # path effects for line
contour

import time
from tqdm import tqdm  # progress bar
from ckmeans_1d_dp import ckmeans  # optimal 1D clustering


# ------------------------------
# Utility functions


# get function pointer and arguments and print elapsed time
def time_it(func, *args):
    start_time = time.time()
    result = func(*args)
    end_time = time.time()
    elapsed_time = end_time - start_time
    print(f"Elapsed time: {elapsed_time} seconds")
    return result


# load room-scan points from CSV
def load_scan(path, delimiter, dims):
    scan_data = np.loadtxt(path, delimiter=delimiter,
usecols=tuple(range(dims)), dtype=float)
    return scan_data.reshape(-1, dims)


# get height and width of rectangle and return randomly generated
points on it
def generate_random_points_on_rectangle(width, height,
num_points, angle=0, noise=False, noise_std=0.1):
    # set random seed
    np.random.seed(42)

    # random values for edge selection (0: left, 1: top, 2:
right, 3: bottom)
    sides = np.random.randint(0, 4, num_points)

    # random positions along each edge
    random_positions = np.random.random(num_points)

    # arrays for x and y coordinates
    x_coordinates = np.zeros(num_points)
    y_coordinates = np.zeros(num_points)

    # points on the perimeter
    # left side
    x_coordinates[sides == 0] = -width / 2
    y_coordinates[sides == 0] = (random_positions[sides == 0] -
0.5) * height

    # top side
```

```
    x_coordinates[sides == 1] = (random_positions[sides == 1] -
0.5) * width
    y_coordinates[sides == 1] = height / 2

    # right side
    x_coordinates[sides == 2] = width / 2
    y_coordinates[sides == 2] = (random_positions[sides == 2] -
0.5) * height

    # bottom side
    x_coordinates[sides == 3] = (random_positions[sides == 3] -
0.5) * width
    y_coordinates[sides == 3] = -height / 2

    # combine x and y into points
    random_points = np.column_stack((x_coordinates,
y_coordinates))

    # rotate points by angle (degrees)
    angle_rad = np.radians(angle)
    rotation_matrix = np.array([[np.cos(angle_rad),
np.sin(angle_rad)],
                                [-np.sin(angle_rad),
np.cos(angle_rad)]])
    random_points = np.dot(random_points, rotation_matrix)

    # add noise if needed
    if noise:
        noise_array = np.random.normal(0, noise_std,
size=(num_points, 2))
        random_points += noise_array

    return random_points


# get points and return X=theta, Y=norm
def get_theta_and_norms(points):
    # theta and r for each point
    theta = np.arctan2(points[:, 1], points[:, 0])  # relative to
x axis
    norm = np.linalg.norm(points, axis=1)
    order = np.argsort(theta)

    return theta[order], norm[order]


# get polar coordinates, return euclidean coordinates
def polar_to_cartesian(theta, r):
    x = r * np.cos(theta)
    y = r * np.sin(theta)
    return x, y


# ------------------------------------------
# Segmentation algorithm


# O(n^2). alpha[i,j] is the sum y_k/cos(x_k - phi) for all k in
```

```
range [j,i]
def get_alpha_matrix(X, Y, phi):
    terms = np.triu(Y / np.cos(X - phi), 0).T
    A = terms.cumsum(axis=0)
    return A


# O(n^2). beta[i,j] is the sum (1/cos(x_k - phi))**2 for all k in
range [j,i]
def get_beta_matrix(X, phi):
    terms = np.triu(1 / (np.cos(X - phi) ** 2), 0).T
    B = terms.cumsum(axis=0)

    # add 1 to zeros of the upper triangle to avoid DivideByZero
    B += np.triu(np.ones((X.shape[0], X.shape[0])), 1)
    return B


# O(n^2). S[i,j] is the sum (y_k -
(alpha[i,j]/beta[i,j])/cos(x_k))**2 for all k in range [j,i]
# i.e., S[i,j] is the cost of partitioning [i,j]
def get_scatter_matrix(X, Y, phi):
    # number of elements in X and Y
    n = X.shape[0]
    assert Y.shape[0] == n

    # calculate alpha and beta matrices
    A = get_alpha_matrix(X, Y, phi)
    B = get_beta_matrix(X, phi)

    # shift A and B 1 row down so that M_n becomes M_(n-1)
    zero_row = np.zeros((1, n), dtype=float)
    one_row = np.ones((1, n), dtype=float)
    shifted_A = np.concatenate((zero_row, A[:-1]), axis=0)
    shifted_B = np.concatenate((one_row, B[:-1]), axis=0)

    # terms to accumulate
    terms = np.triu(Y**2, 0).T + ((shifted_A**2) / shifted_B) -
((A**2) / B)

    S = terms.cumsum(axis=0)
    return S


# O(n^2). Assume S.shape=(n, n, 5). return D with shape (n + 1,
6)
# D[i,m] is the optimal price for ending segment m in point i
(from 1)
def get_D_matrix(S):
    n = S.shape[1]  # Assuming S is a numpy array with shape (n,
n, 5)
    D = np.zeros((n + 1, 6))  # Initialize the table D with
zeros, shape (n+1, 6)

    for i in range(1, n + 1):
        for m in range(1, 6):
            j_range = np.arange(m, i + 1)
            if j_range.size == 0:
```

```
                D[i, m] = np.inf
            else:
                # you can add + j_range to encourage smaller j
values
                temp = D[j_range - 1, m - 1] \
                       + S[i - 1, j_range - 1, m - 1] \
                       + (m == 1) * S[j_range - 2, 0, 0]
                D[i, m] = np.min(temp)
    return D


# O(n^2). Assume S.shape=(n, n, 5), D.shape=(n+1, 6).
# arg_D[i, m] is the optimal start index 'j' s.t. segment index
'm' ends in index 'i'
# note that 'index' = 'point' - 1.
def get_arg_D_matrix(S):
    n = S.shape[1]
    D = get_D_matrix(S)
    arg_D = np.zeros((n, 4), dtype=int)

    for i in range(1, n + 1):
        for m in range(2, 6):
            j_range = np.arange(m, i + 1)
            if j_range.size == 0:
                continue
            else:
                temp = D[j_range - 1, m - 1] + S[i - 1, j_range -
1, m - 1]
                arg_D[i - 1, m - 2] = j_range[np.argmin(temp)]

    return arg_D


# backtrack arg_D matrix to find optimal segmentation points
def get_segmentation_points(X, Y):
    S = np.array([get_scatter_matrix(X, Y, m * np.pi / 2) for m
in range(5)]).transpose([1, 2, 0])
    arg_D = get_arg_D_matrix(S)  # shape (n, 4)

    indexes = []
    i = arg_D.shape[0]
    for num_partition in range(4)[::-1]:
        curr_val = arg_D[i - 1, num_partition]
        indexes.append(curr_val)
        i, num_partition = curr_val - 1, num_partition - 1
    return indexes[::-1]  # reverse order


#  get X[start idx. : end idx.], Y[start idx. : end idx.],
segment idx. 'm'. return a/cosine(X + phase)
def segment_func(X, Y, m):
    phase = m * np.pi / 2
    alpha = np.sum(Y / np.cos(X - phase))
    beta = np.sum(1 / (np.cos(X - phase) ** 2))
    a = alpha / beta
    return lambda x: a / np.cos(x - phase)
```

```
# End of the segmentation algorithm!
# -----------------------------------------------
# data batching and average fit


#  O(k*log(n)). Segment the input array X based on the values in
the partition array.
# if is_x_values=True then partition_values are treated as x
values, otherwise they are indexes.
def partition_x_y(X, Y, partition_values, is_x_values=True):
    n = X.shape[0]
    assert n == Y.shape[0]

    # Find the indices where partition values should be inserted
    if is_x_values:
        partition_indices = np.searchsorted(X, partition_values,
side='right')
    else:
        partition_indices = partition_values

    # array of indices to split X
    split_indices = np.concatenate(([0], partition_indices, [n]))

    # you can also use np.split to segment X based on the
split_indices
    x_segments = [X[split_indices[i]:split_indices[i + 1]] for i
in range(len(split_indices) - 1)]
    y_segments = [Y[split_indices[i]:split_indices[i + 1]] for i
in range(len(split_indices) - 1)]

    return x_segments, y_segments


# O(n^2) get X, Y batch. return partition x values, segment
fitted functions
def get_partitions_and_fits(x_batch, y_batch, partitions=None):
    n = x_batch.shape[0]
    assert n == y_batch.shape[0]

    if not partitions:
        partition_indexes =
np.array(get_segmentation_points(x_batch, y_batch)) - 1
        seg_starts = np.concatenate([[1], partition_indexes]) - 1
        seg_ends = np.concatenate([partition_indexes, [n - 1]])
        starts_ends = np.array(list(zip(seg_starts, seg_ends)))

    fitted_functions = [segment_func(x_data[se[0]:se[1]],
y_data[se[0]:se[1]], i)
                        for i, se in enumerate(starts_ends)]

    return x_batch[partition_indexes], fitted_functions


# get X data, Y data, batch size and return data uniform samples
in iterable batches
def batch_data(X, Y, batch_size, shuffle=True, random_seed=None):
    data_size = X.shape[0]
    assert data_size == Y.shape[0]
```

```
    num_batches = data_size // batch_size

    # Ensure data can be evenly divided into batches
    if data_size % batch_size != 0:
        num_batches += 1

    # Set the random seed for reproducibility
    if random_seed is not None:
        np.random.seed(random_seed)

    # Shuffle the data if specified
    if shuffle:
        indices = np.arange(data_size)
        np.random.shuffle(indices)
        X = X[indices]
        Y = Y[indices]

    for batch_idx in range(num_batches):
        start_idx = batch_idx * batch_size
        end_idx = (batch_idx + 1) * batch_size

        X_batch = X[start_idx:end_idx]
        Y_batch = Y[start_idx:end_idx]

        # sort batch
        order = np.argsort(X_batch)
        X_batch = X_batch[order]
        Y_batch = Y_batch[order]
        yield X_batch, Y_batch


# O(n^2). get entire X, Y dataset. divide into batches.
# return partition x values, segment fitted functions.
def fit_dataset(X, Y, batches=False, batch_size=1000):
    n = X.shape[0]
    assert n == Y.shape[0]

    if not batches:
        batch_size = n

    batch_partitions = []
    batch_fits = []

    x_y_batches = batch_data(X, Y, batch_size)
    for x_batch, y_batch in tqdm(x_y_batches):
        batch_partition, batch_fit =
get_partitions_and_fits(x_batch, y_batch)
        batch_partitions.append(batch_partition)  # partition x
values
        batch_fits.append(batch_fit)
    average_partitions = np.mean(batch_partitions, axis=0)

    segmented_x, segmented_y = partition_x_y(X, Y,
average_partitions)

    segmented_fits = [segment_func(segmented_x[i],
segmented_y[i], i) for i in range(5)]
```

```
    return average_partitions, segmented_fits


# get partitions not including 0 and N, and segment fits. return
max points X,Y
def get_fit_meet_points(x_partitions, data_fits):
    # initialize a list to store the intersections
    intersections_x = []
    intersections_y = []
    for i, x in enumerate(x_partitions):
        x_range = np.linspace(x - 1, x + 1, 100)

        # Calculate the values of both functions over the entire
range
        y1 = data_fits[i](x_range)
        y2 = data_fits[i+1](x_range)

        # find intersection
        meet_indices = np.argwhere(np.diff(np.sign(y1 - y2)))
        if meet_indices.shape[0] > 1:
            intersection_idx =
np.argmin(np.abs(x_range[meet_indices] - x))

intersections_x.append(x_range[meet_indices][intersection_idx])

intersections_y.append(data_fits[i](x_range[meet_indices][interse
ction_idx]))
        else:
            intersections_x.append(x_range[meet_indices])

intersections_y.append(data_fits[i](x_range[meet_indices]))

    return intersections_x, intersections_y
    # meet_points_y = [max((data_fits[i](x), data_fits[i+1](x)))
    #                  for i, x in enumerate(x_partitions)]
    # return x_partitions, meet_points_y


# ------------------------------------------------------------
# graphing the results


# plot 2D points
def plot_polygon(points):
    plt.figure(figsize=(8, 8))
    plt.scatter(points[:, 0], points[:, 1], marker='o',
color='blue')
    plt.gca().set_aspect('equal', adjustable='box')
    plt.axhline(0, color='black', linewidth=0.5)
    plt.axvline(0, color='black', linewidth=0.5)
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.title('2D Room Scan Points')
    plt.grid(True)
    plt.show()
```

```
# get polar X,Y, partition x values, fitted segment functions.
# project Y onto fit and return projected segmented cartesian X_Y
outside of rectangle
def pol_to_cart_outside_fit(X, Y, x_partitions, data_fits):
    segmented_x, segmented_y = partition_x_y(X, Y, x_partitions)

    projected_segmented_x_y = []
    for i, seg_fit in enumerate(data_fits):
        condition = (seg_fit(segmented_x[i]) <= segmented_y[i])

projected_segmented_x_y.append(polar_to_cartesian(segmented_x[i][
condition],

(segmented_y[i][condition])))
    return projected_segmented_x_y


# get polar X,Y, partition x values, fitted segment functions.
# project Y onto fit and return projected segmented cartesian X_Y
in range of distance from rectangle
def pol_to_cart_near_fit(X, Y, x_partitions, data_fits):
    segmented_x, segmented_y = partition_x_y(X, Y, x_partitions)

    projected_segmented_x_y = []
    for i, seg_fit in enumerate(data_fits):
        condition = (seg_fit(segmented_x[i]) <= segmented_y[i]) &
(seg_fit(segmented_x[i]) + 0.1 >= segmented_y[i])

projected_segmented_x_y.append(polar_to_cartesian(segmented_x[i][
condition],

(segmented_y[i][condition])))
    return projected_segmented_x_y


# get X, Y, partition x values, fitted segment functions.
# scatter (X,Y) and plot the fits.
def plot_polar_fit(X, Y, x_partitions, data_fits, max_x=None,
max_y=None):
    plt.scatter(X, Y, alpha=0.5, zorder=1)

    if (max_x is not None) and (max_y is not None):
        plt.scatter(max_x, max_y, color="gold", zorder=3,
label="Corner Points")

    x_partitions = np.hstack([x_data[0], x_partitions, x_data[-
1]])
    p_amount = x_partitions.shape[0]
    for idx, p in enumerate(x_partitions[1:]):
        if idx < p_amount - 2:
            plt.axvline(p, color='black', linewidth=2,
label=f"x={p: .1f}")
        x_dense = np.linspace(x_partitions[idx], p, 1000)
        plt.plot(x_dense, data_fits[idx](x_dense), color="red",
zorder=2)

    plt.xlabel('X')
    plt.ylabel('Y')
```

```
    plt.legend()
    plt.grid(True)
    plt.show()


# get X, Y, partition x values, fitted segment functions.
# convert to cartesian coordinates. Scatter (X,Y) rectangle and
plot the fitted edges.
def plot_cartesian_fit(X, Y, x_partitions, data_fits, max_x=None,
max_y=None):
    X, Y = partition_x_y(X, Y, x_partitions)
    X_Y = [(polar_to_cartesian(x_seg, y_seg)) for x_seg, y_seg in
zip(X, Y)]

    if (max_x is not None) and (max_y is not None):
        corners_x, corners_y = polar_to_cartesian(max_x, max_y)
        corners_x = np.append(corners_x, corners_x[0])
        corners_y = np.append(corners_y, corners_y[0])
        plt.plot(corners_x, corners_y, color="black",
linewidth=3, zorder=2,
                 label="Connected Corners")

    for i, (edge_x, edge_y) in enumerate(X_Y):
        plt.scatter(edge_x, edge_y, alpha=0.5, zorder=1,
label=f"Segment {i + 1}")

    plt.xlabel('X')
    plt.ylabel('Y')
    plt.legend()
    plt.grid(True)
    plt.show()


# O(n). get partitioned projected cartesian X,Y. cluster edges 1
dimensionally. find most sparse cluster.
def plot_door_cluster(X, Y, x_partitions, data_fits):
    x_y_out = pol_to_cart_outside_fit(X, Y, x_partitions,
data_fits)
    x_y_near = pol_to_cart_near_fit(X, Y, x_partitions,
data_fits)

    # connect halved edge
    x_y_out[0] = np.hstack((x_y_out[0], x_y_out[-1]))
    x_y_out = x_y_out[:-1]

    x_y_near[0] = np.hstack((x_y_near[0], x_y_near[-1]))
    x_y_near = x_y_near[:-1]

    max_sparse = 0  # max sum of distances squared within cluster
normalized by amount of points
    sparse_cluster_x = None
    sparse_cluster_y = None

    def cluster_range(x, y, idx):
        if idx % 2 == 0:
            return np.max(y) - min(y)
        else:
            return max(x) - min(x)
```

```
    for i, (edge_x, edge_y) in enumerate(x_y_out):
        plt.scatter(edge_x, edge_y, color="blue", alpha=0.5,
zorder=1, label=f"Wall #{i + 1}")

    for i, (edge_x, edge_y) in enumerate(x_y_near):
        if i % 2 == 0:
            clusters = ckmeans(edge_y, 20)
        else:
            clusters = ckmeans(edge_x, 20)

        # split edge into clusters
        split_indices =
np.cumsum(np.array(clusters.size).astype(int))
        edge_x = np.split(edge_x, split_indices)
        edge_y = np.split(edge_y, split_indices)

        for j, (cluster_x, cluster_y) in enumerate(zip(edge_x[:-
1], edge_y[:-1])):
            # find most sparse cluster
            if clusters.size[j] > 0:
                sparseness = cluster_range(cluster_x, cluster_y,
i) / clusters.size[j]
                if sparseness > max_sparse:
                    max_sparse = sparseness
                    # take the relevant cluster
                    sparse_cluster_x, sparse_cluster_y =
(edge_x[j], edge_y[j])

    plt.scatter(sparse_cluster_x, sparse_cluster_y, color="red",
alpha=0.5, zorder=2, label=f"Possible Door")

    plt.xlabel('X')
    plt.ylabel('Y')
    plt.legend()
    plt.grid(True)
    plt.show()


# ----------------------------------------------------------------
------------------


N = 1000
# rect_points = generate_random_points_on_rectangle(100, 50, N,
angle=0, noise=True, noise_std=1)
rect_points = load_scan('./5.xyz', ' ', 2)
# center the points
mean_x = np.mean(rect_points[:, 0])
mean_y = np.mean(rect_points[:, 1])
centered_points = rect_points - np.array([mean_x, mean_y])

angles, norms = get_theta_and_norms(centered_points)

x_data, y_data = angles, norms
partitions, fits = time_it(fit_dataset, x_data, y_data, True,
500)
print(f"Segmentation Points: {partitions}")
```

```
x_meets, y_meets = get_fit_meet_points(partitions, fits)

# plot_polygon(rect_points)
plot_polar_fit(x_data, y_data, partitions, fits, x_meets,
y_meets)
plot_cartesian_fit(x_data, y_data, partitions, fits, x_meets,
y_meets)
plot_door_cluster(x_data, y_data, partitions, fits)
```