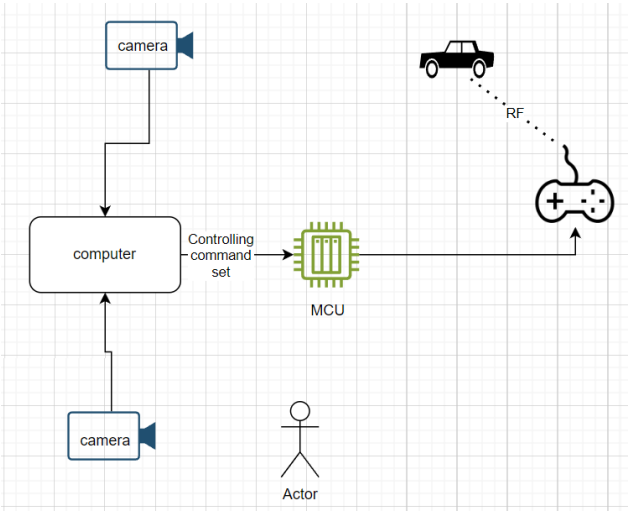
# Hands on

## דו"ח מסכם פרויקט

יואב נחום – 318674249
דור דוד - 208958470
ניקולאי קרוחמל - 320717184
איתן ספיבק - 311391866

# 1 תקציר:

במסגרת הפרויקט בחרנו לבנות משחק מירוצים, בו מטרתו של כל מתמודד היא להשלים הקפה אחת ברכב המירוץ סביב מסלול המירוץ, אותו מרכיבים כרצונינו, בפרק הזמן הקצר ביותר. לשם כך עשינו שימוש בשתי מצלמות. הראשונה אחראית לצלם את השחקן ולעקוב אחרי תנועות ידיו בעזרת כלים של עיבוד תמונה. כך המחשב יוכל לשלוח לרכב על המסלול פקודות ולאפשר שליטה ידידותית של הנהג ברכב, כאילו הוא אוחז בהגה. המצלמה השנייה מצלמת את המסלול, תחילה כדי שמהחשב יוכל לזהות אותו ואת פניותיו, ולאחר מכן כדי לאפשר מעקב אחרי מיקום הרכב אותו המחשב יודע להציג על המסך בצורה נוחה. בנוסף, האלגוריתם העוקב אחרי הרכב מאפשר להבין מתי הרכב עובר כל פניה במסלול ולמדוד את זמני ההקפה הטובים ביותר, שנרשמים אחר כך בטבלת הניקוד.




תרשים 1: תרשים בלוקים של המערכת

## 2 הקדמה:

מטרת הפרויקט המסכם בקורס להציג שימוש והבנה בשלל אלגוריתמי עיבוד התמונה שהוצגו במהלך הקורס. לשם כך החלטנו לבנות משחק מירוצים המאפשר לשחקן שליטה ברכב צעצוע בעזרת תנועת הידיים בלבד. מטרת המשחק היא להרכיב מסלול מירוץ, ואז לנסות להשלים במכונית את ההקפה המהירה ביותר סביבו. לפרויקט צפויים מספר אתגרים, המשמעותיים בהם:

- זיהוי עקבי של הידיים תוך כדי נהיגה על מנת שהשליטה ברכב תיהיה קלה ואינטואיטיבית.
- זיהוי עקבי ומדויק של מיקום הרכב על מנת לאפשר הצגה שלו על פני המפה של המסלול לשחקן.
- זיהוי רובסטי של הידיים שיתאים לכלל המשתמשים.
- זיהוי הידיים ומיקום הרכב בזמן אמת – כלל החישוב צריך להיעשות בקצב גבוה מהfps של המצלמה. המצלמה מצלמת 30 פריימים בשנייה ולכן כלל החישובים צריכים לקחת פחות מ-0.033 sec.

מערכת המשחק מורכבת ממחשב עליו רצה התוכנה, רכב צעצוע הנשלט מרחוק בעזרת שלט ומעגל חשמלי שמטרתו לעקוף את הכפתורים בשלט ולאפשר שליחת פקודות דרך השלט למחשב. את המעגל הרכבנו על השלט של הרכב צעצוע והשתמשנו במיקרו בקר של חברת stm32 בשביל ההתממשקות עם המחשב. בנוסף השתמשנו בסרטים צהובים באורכים שונים כדי לבנות את המסלול ובשתי מצלמות – מצלמה מהמחשב האישי ממנו התוכנה רצה ומצלמת את השחקן ואת תנועות ידיו.
מצלמה שניה ממוקמת מעל המסלול (על מדף גבוה) ומטרתה לצלם את המסלול ולאפשר זיהוי שלו ומעקב אחר מיקום הרכב.

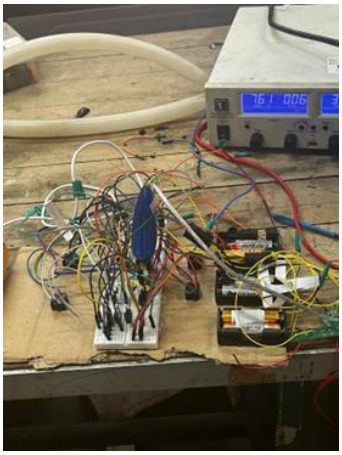
עבור המשחק עצמו השתמשנו בספרית pygame של פייתון כבסיס לבניית השלד הכללי של המשחק. השלד מורכב מ-5 פונקציות שונות:

- Main - המסך הראשי של המשחק ממנו מנווטים לשאר הפונקציות של המשחק.
- Set_player - כאן מבקשים את שם השחקן ומצלמים את ידיו כאילו אוחז בהגה. בתמונת ידיים זו נשתמש בהמשך לחפש את מיקום הידיים של השחקן תוך כדי הנהיגה.
- Detect_map - חלק זה אחראי על זיהוי המסלול, זיהוי הפינות ומתיחת המסלול למבט על בעזרת טרנספורמציה אפינית.
- Play - הפונקציה בה קורא המשחק עצמו, כלומר השליטה ברכב, זיהוי המיקום של הרכב וכו'.
- Show_leaderboard – כאן מראים את טבלת הניקוד של כל נהג לפי המסלול האחרון. הטבלה מתאפסת עם כל שינוי מסלול.


במהלך בניית המשחק השתמשנו במספר הנחות מקלות, הראשונה בהן שהמערכת כולה תיהיה ממוקמת בקומה 1- של סיפרית ארן באוניברסיטת בן גוריון במיקום בו היה ממוקם הדוכן שלנו. מכך נגזר צבע השטיח האופייני, וצבע הקיר מאחורי השחקן – שני עובדות שהקלו עלינו בפיתוח האלגוריתמים לזיהוי הרכב, המסלול והידיים. שנית, השתמשנו בסרטים צהובים אשר בולטים היטב על רקע השטיח על מנת לבנות את המסלול. בנוסף, ניצלנו את העובדה שרכב הצעצוע עצמו צבוע בלבן לשם עקיבה מבוססת צבע.

לבסוף, הנחנו מיקום מצלמות סטטי (לא הזזנו אותן מהבוקר) וסדר קבוע למהלך המשחק – ראשית מגדירים שחקן ומבצעים סגמנטציה של תנועות ידיים( בתנועת 2 אגרופים ), לפקודות המיועדות לרכב. לאחר מכן מבצעים זיהוי מסלול ללא הרכב על המסלול, כאשר המסלול מורכב בתוך ריבוע של 1.8X1.8 מטר, מוגדר מראש ומסומן באיזולירבאנד כחול ובו פרט לסרטים הצהובים אין דבר. בסוף מניחים את הרכב על המסלול ומתחילים לשחק.


**איור 1:** השלט לפני פירוקו



**איור 2:** השלט לאחר פירוקו והמעגל החשמלי היוצר מעקף

## 3 האלגוריתמים מאחורי המשחק:

כעת נפרט על כל האלגוריתמים שמשחקים תפקיד מרכזי בפרויקט, לפי מטרותיהם השונות – זיהוי תנועות הידיים, זיהוי המסלול, זיהוי מיקום הרכב והצגה המסלול לשחקן.

### 3.1 זיהוי ועקיבה של הידיים:

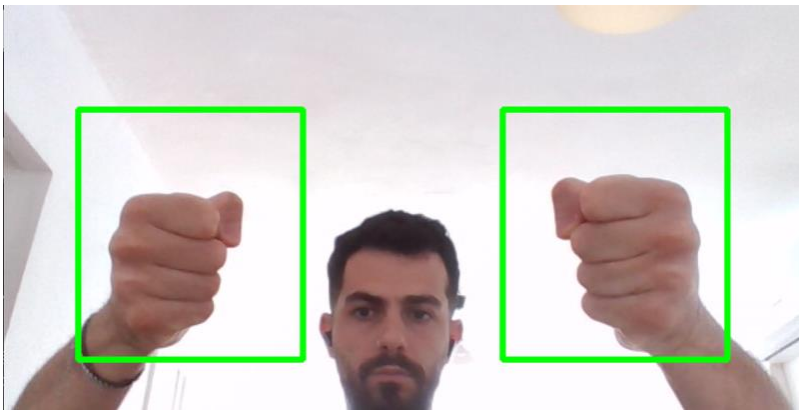
בחלק זה העלנו את השאלה איך נוכל לבצע זיהוי ידיים בצורה הטובה ביותר, על מנת לסמן את הפיקסל המייצג את מרכז היד. לאחר התלבטויות רבות הצענו את האלגוריתם הבא המבוסס על SIFT detector יחד עם כמה הנחות מקלות שצוינו קודם.

האלגוריתם התחלק לשלבים הבאים:

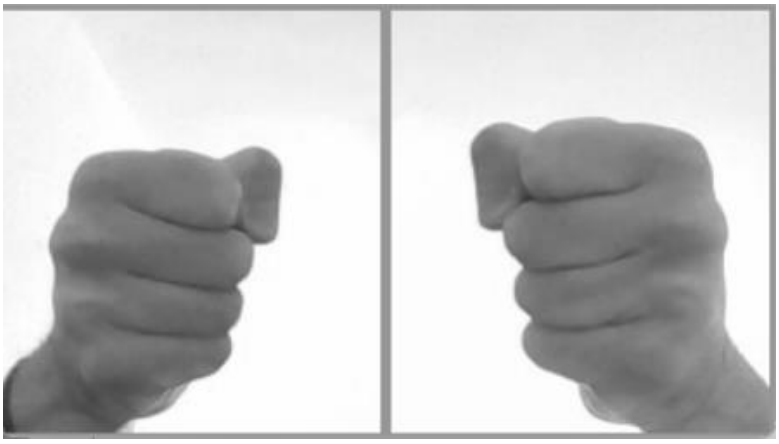
א. <u>שלב 1 - Reference:</u>

לפני תחילת המשחק התבקש מהנהג/ שחקן לשים את הידיים(אגרופים) בתוך 2 ריבועים ירוקים( איור 1) כאילו הוא אוחז בהגה.

לאחר כ-3 שניות גזרנו מתוך הריבועים את התמונות של הידיים שהתקבלו מהמשתמש ועברנו ל-gray scale (איור 2).



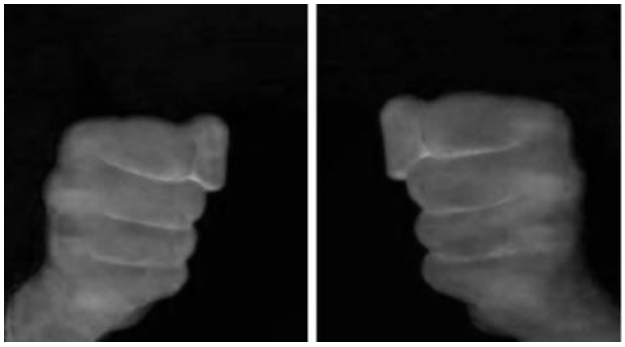
**איור 1 –**



**איור 2 –**

ב. <u>שלב 2 – סינון:</u>

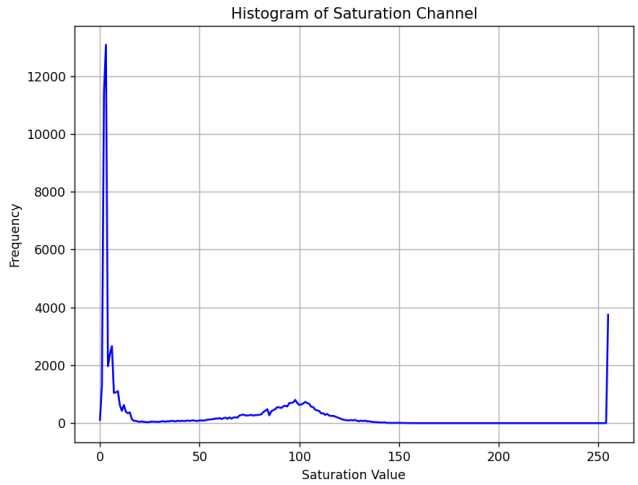
על מנת לקבוע תמונת רפרנס מספיק טובה שתמצא לנו מספיק פיצ׳רים תואמים, ביצענו סינון רקע בסיסי על ידי מעבר לערוצי hsv. כאשר חקרנו את ערוצי הצבע השונים ראינו שעבור הרקע ערוץ s חלש משמעותית לעומת החלק של היד(איור 3).



**איור 3 –**

הסתכלנו על ההיסטוגרמה(איור 4) של ערוץ s של הרפרנס וכך קבענו threshold מתאים על מנת למסך את התמונה לתמונה בינארית בעלי 2 ביטים בלבד: פיקסלים לבנים אשר יהיו שייכים ליד וכל שאר הפיקסלים יהיו שחורים( איור 5). על ידי כך הצלחנו להפריד את הרקע ולהשחיר אותו לגמרי, ובכך להשאיר רק את היד.

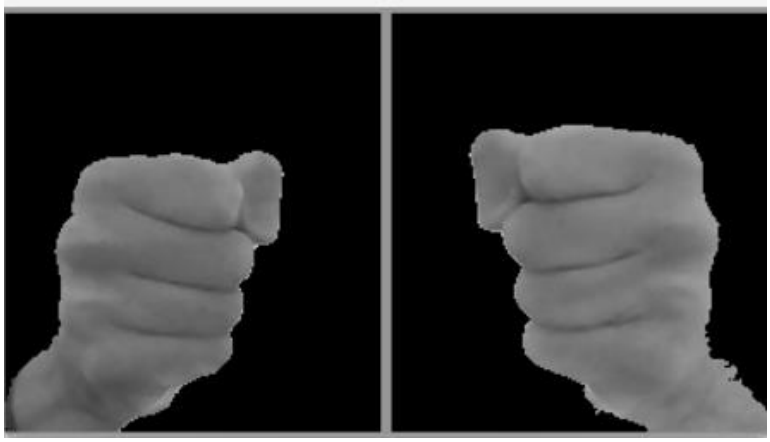
דבר זה הכרחי כדי להימנע מזיהוי פיצ׳רים לא רצויים מהתמונה בזמן אמת.




**איור 4 –**

**איור 5 –**

לבסוף על מנת לקבל את התוצאה הסופית - תמונת רפרנס ממנה יזוהו פיצ'רים, ביצענו מכפלה סקלרית בין תמונת הידיים הממוסכות( איור 5) לבין התמונה ב-gray scale ( איור 2) כיוון שאת התמונה בזמן אמת נציג ב- gray scale (איור 6).



**איור 6 –**

ג. <u>שלב 3 – שימוש ב-SIFT והמניפולציות המתאימות:</u>

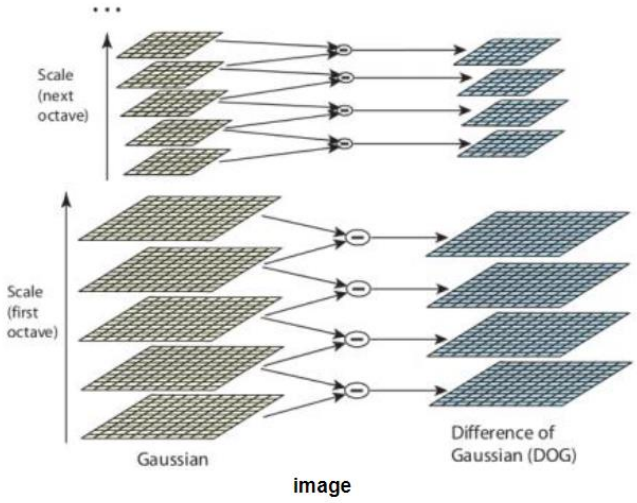
השתמשנו באלגוריתם SIFT למציאת פיצ'רים עבור תמונת רפרנס וכן עבור כל frame הנלקח מהסרטון בזמן אמת.

עבור כל צמד תמונות כנ"ל האלגוריתם יוצר התאמה בין פיצ'רים .

הפרמטרים אותם אלגוריתם ה-SIFT מקבל:

```
cv::SIFT::create ( int      nfeatures,
                   int      nOctaveLayers,
                   double   contrastThreshold,
                   double   edgeThreshold,
                   double   sigma,
                   int      descriptorType
                 )
```

- **nOctaveLayers -** מספר השכבות בכל אוקטבה בה נבצע חיסור בין התמונות שמועברות בפילטר גאוסייני.
הרבה שכבות יביאו למציאת פיצ'רים איכותיים יותר אך עם זאת נקבל כוח חישוב גדול יותר מה שיביא לפגיעה באלגוריתם בזמן אמת.
- **sigma -** הווריאנס של הפילטר הגאוסייני הפועל בכל אוקטבה. עבור מצלמה איכותית נרצה ערך גבוה יותר על מנת שההבדלים בין התמונות שיחוסרו בכל אוקטבה ( difference of Gaussian- DOG ) יהיה גדול יותר.




- **edge threshold –** קריטריון זה מאפשר סינון פיצ'רים בקצוות העצמים בתמונה, על ידי כך שפילטר גאוסייני מועבר בתמונה ויוצר תמונה חדשה ( Low pass ) . לאחר מכן התמונות מחוסרות ( difference of Gaussian- DOG ) ומתקבלת תמונה עם דגש על הקצוות( High pass ). עצם פעולה זו מביאה לקבלת פיצ'רים רלוונטים בעיקר בקצוות העצמים הרלוונטיים( תמונת היד).

לשם כך נקבע את הפרמטר להיות 10 – ערך אשר מבטיח הפחתה של מציאת פיצ'רים לא רלוונטיים בקצוות התמונה.

- **contrast threshold –** קריטריון זה נועד לסינון פיצ'רים באזורים בעלי ניגודיות נמוכה/חלשה. על ידי כך אנו נשארים רק עם פיצ'רים רלוונטיים בהם נרצה להשתמש לסיווג תנועות הידיים.
ככל שנבחר את ערך זה להיות גדול יותר, נקבל פחות פיצ'רים בסך הכל.

ד. <u>שלב 4 – קביעת features טובים:</u>

לאחר שקיבלנו רשימה של features גם מהרפרנס וגם מהreal time frames היינו רוצים לבדוק התאמה ובעצם לבדוק עבור כל פיצ׳ר ברפרנס מי הפיצ׳ר הכי ״קרוב״ לו בתמונת ה-real time.

יתר על כן דרשנו מההתאמה בין הפיצ׳רים דרישה גבוהה יותר אשר תחזיר לנו התאמות טובות בלבד ותזרוק התאמות גבוליות.

לשם כך ביצענו התאמה באמצעות knn, קבענו את k להיות **<u>2</u>**, דבר אשר יחזיר לנו את הפיצ׳ר התואם הכי טוב והשני הכי טוב.

ההתאמה השנייה הכי טובה תשמש לבקרת איכות על ההתאמה הכי טובה.

הדבר אשר קובע את איכות ההתאמה הוא המרחק בין כל פיצ׳ר וכדי לקבוע ההתאמה מוצלחת, הגדרנו את ההתניה הבאה:

m.distance < C*n.distance , כאשר m.distance הוא המרחק של ההתאמה הטובה ביותר,

```
matches = matcher.knnMatch(descriptors_reference, descriptors_RT, k=2)
good_matches = []
good_keypoints_x = []
good_keypoints_y = []
for m, n in matches:
    if m.distance < 0.7 * n.distance:
        good_matches.append(m)
```

n.distance הוא המרחק של ההתאמה השנייה הטובה ביותר ו-C הוא קבוע בין 0 ל-1.

ככל ש-C קטן יותר נקבל פחות התאמות אבל התאמות בעלות סיכוי גבוה יותר להיות מאופיינות כהתאמה נכונה.

מצד שני ככל ש-C גדול יותר נקבל יותר התאמות אבל עם סיכון להתאמה עם סיווג לא נכון.

לאחר ניסוי ותהייה מצאנו כי כאשר C=0.7 נקבל מספיק התאמות תוך כדי שנקבל התאמות טובות.

ה. <u>שלב 5 – הגדרת נקודת אמצע היד:</u>

ראשית, הנחנו כי כל יד נמצאת בצד מסוים של ה-frame.

לקחנו את כל ההתאמות הטובות בכל צד של הפריים:

נקודות אשר שיעור ה-x שלהם קטן מ-280 מסווגות לחלק השמאלי, ונקודות אשר שיעור ה-x שלהן גדול מ-360 מסווגות לחלק הימני.

לאחר מכן ביצענו מיצוע על האינדקסים כך שנקבל נקודה (x,y) אשר מתארת את מרכז היד של כל יד – centroid_left_x, cemtroid_left_y וכן עבור יד ימין.

```
left_good_keypoints = [left for left in good_keypoints if left[1] < 280]
right_good_keypoints = [right for right in good_keypoints if right[1] >= 360]

first_elements_left_x = [item[0] for item in left_good_keypoints]
first_elements_left_y = [item[1] for item in left_good_keypoints]

first_elements_right_x = [item[0] for item in right_good_keypoints]
first_elements_right_y = [item[1] for item in right_good_keypoints]

if len(left_good_keypoints) == 0 or len(right_good_keypoints) == 0:
    cv2.imshow('Matched Keypoints', hands_frame_RT)
    cv2.waitKey(20)
    return None

else:
    centroid_left_x = sum(first_elements_left_x) // len(left_good_keypoints)
    centroid_left_y = sum(first_elements_left_y) // len(left_good_keypoints)

    centroid_right_x = sum(first_elements_right_x) // len(right_good_keypoints)
    centroid_right_y = sum(first_elements_right_y) // len(right_good_keypoints)
```

ו. <u>שלב 6 – קביעת תנועת הרכב:</u>

לצורך כיווני הנסיעה מתחנו קו וירטואלי בין הידיים וקבענו כי שיפוע הישר יקבע את כיוון הרכב.

לרכב 6 תצורות שונות של נהיגה:

- <u>עצירה:</u> כאשר הנהג מניח את הידיים, אין זיהוי של פיצ׳רים טובים.

  קבענו שמתחת ל-10 התאמות טובות לא תהיה תגובה של הרכב( לא יצויר קו).

כאשר יש מעל ל-10 התאמות יוגדרו המצבים הבאים:

- <u>נסיעה קדימה:</u> כאשר הנהג מציב את הידיים בצורה מקבילה לאופק.

  שיפוע הישר בין 0.2 לבין 0.2- מה שיקבע נסיעה קדימה בלבד.

- <u>קדימה וימינה:</u> כאשר הנהג מציב את הידיים בשיפוע קל לכיוון ימין.

  שיפוע הישר בין 0.5- לבין 0.2- מה שיקבע נסיעה קדימה וימינה( צפון מזרח).

- <u>קדימה ושמאלה:</u> כאשר הנהג מציב את הידיים בשיפוע קל לכיוון שמאל.

שיפוע הישר בין 0.5 לבין 0.2 מה שיקבע נסיעה קדימה ושמאלה( צפון מערב).

```
if grad is None:
    comm_platform.Set_package_and_transmit('r', port) # stop car
elif (grad <= 0.2) and (grad >= -0.2):
    comm_platform.Set_package_and_transmit('w', port) # go forward
    print("forward")
elif (grad < -0.2) and (grad >= -0.5):
    print("forward right")
    comm_platform.Set_package_and_transmit('d', port) # go forward and right
elif (grad > 0.2) and (grad <= 0.5):
    print("forward left")
    comm_platform.Set_package_and_transmit('y', port) # go forward and left
elif grad > 0.5:
    print("left")
    comm_platform.Set_package_and_transmit('a', port)  # go right
elif grad < -0.5:
    print("right")
    comm_platform.Set_package_and_transmit('d', port)  # go right
```

- <u>ימינה:</u> כאשר הנהג מציב את הידיים בשיפוע חזק לכיוון ימין.

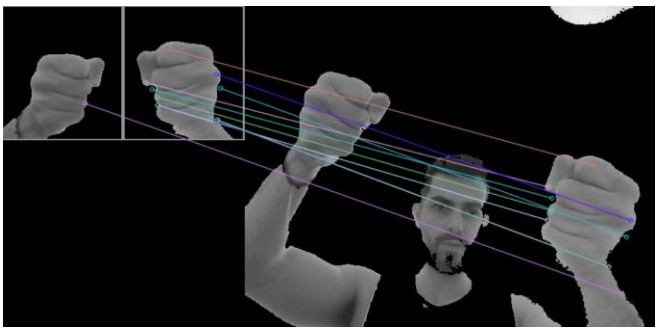
שיפוע הישר קטן מ- 0.5- , מה שיקבע סיבוב ימינה במקום.

- <u>שמאלה:</u> כאשר הנהג מציב את הידיים בשיפוע חזק לכיוון שמאלה.

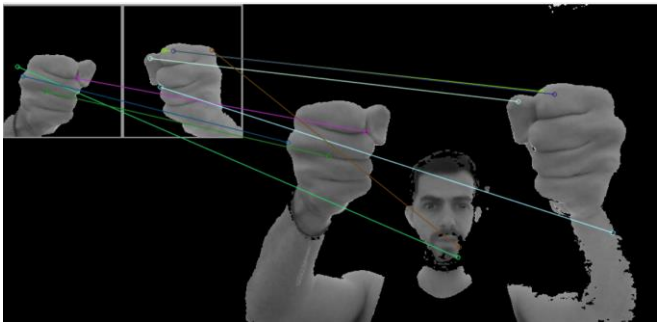
שיפוע הישר גדול מ- 0.5, מה שיקבע סיבוב שמאלה במקום.

## תוצאות ביניים של שלבי אלגוריתם סגמנטייצת תנועות הידיים:

מציאת פיצ'רים בין תמונת הרפרנס של אגרופי הידיים לבין הפריים הרלוונטי בזמן אמת ( איור 1, איור 2).

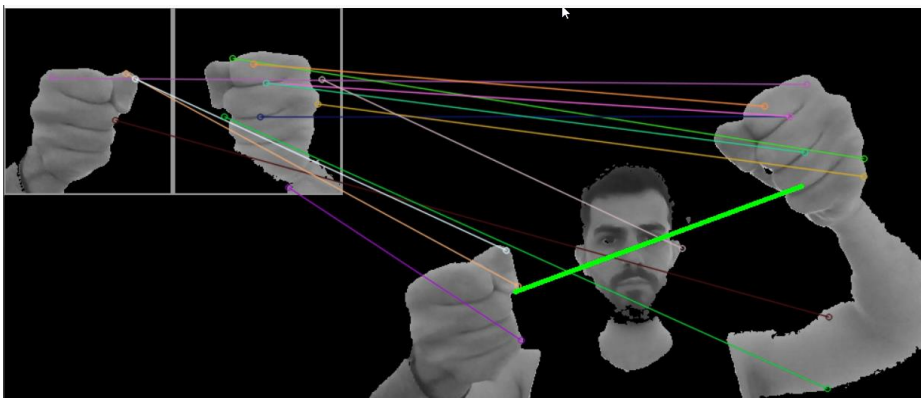


**<u>איור 1 –</u>**

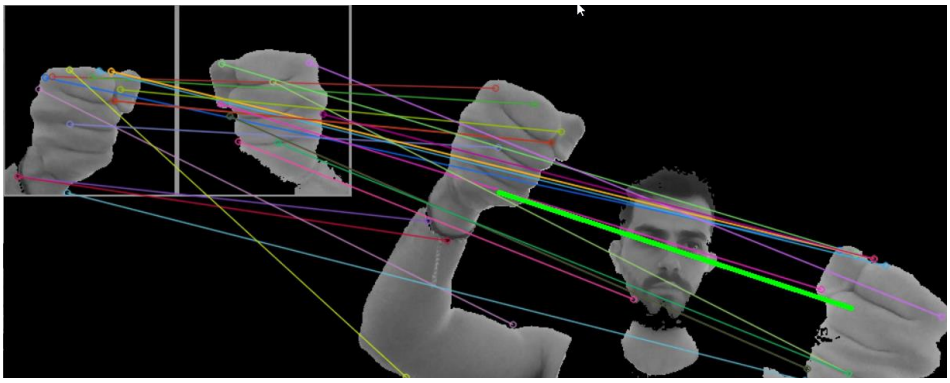


**<u>איור 2 –</u>**

שרטוט הישר הווירטואלי לסיווג כיוון הנסיעה של הרכב – ניתן לראות כי גם כאשר ישנם פיצ'רים אשר סווגו לא נכון עדיין האלגוריתם למציאת מרכז היד יעבוד בצורה טובה והישר הירוק ישורטט כפי שהמשתמש התכוון.(איור 3, איור 4).



**איור 3 -**



**איור 4 -**

## 3.2 זיהוי המסלול:

זיהוי המסלול בוצע על ידי לקיחת פריים בודד ממצלמת המסלול ועיבודו. עיבוד הפריים כלל בתוכו מספר שלבים:

א. שימוש ב Perspective transform כדי לפצות על העיוות הנוצר מכך שהמצלמה מצלמת את המסלול מהמדף העליון ולא ישירות מעל המסלול. מאחר ולא היה לנו כל מידע על הפרמטרים של המצלמה, ולא יכולנו להניח דבר על צורת המסלול, סימנו מסגרת של 1.8x1.8 מטר על הרצפה באיזולירבנד והנחנו שהמסלול נמצא כולו בתחום המסגרת ,וכי המסגרת כולה נמצאת בפריים. לשם ביצוע הטרנספורמציה יש צורך תחילה בשתי רשימות- אחד של נק' מקור בשבילהן השתמשנו בפינות המסגרת, ואחת של נק' יעד שהן הפינות של התמונה. ביצענו שימוש בפקודה cv2.getPerspectiveTransform(src, dst) המקבלת את שתי הרשימות ומוצאת מטריצת טרנספורמציה H כך שמתקיים:

$$x_d = H * x_i$$

מאחורי הקלעים תחילה האלגוריתם ממיר את הקורדינטות בדו-מימד לקורדימטות הומוגניות בתלת מימד ע"י הופסת 1 בקורדינטה השלישית של כל נק' מקור וscaling factor בכל נק' יעד:

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} \rightarrow \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}, \begin{bmatrix} x_d \\ y_d \end{bmatrix} \rightarrow \begin{bmatrix} x_d \\ y_d \\ w_d \end{bmatrix}$$

נשים לב שנמאחר והחזרה לקורדינטות דו-מימד דורשות חלוקה של $x_i$ ו- $y_i$ ב- $w_d$ ניתן לרשום את הקורדינטות ההומוגניות של המקור בצורה הבאה:

$$\begin{bmatrix} x_d \\ y_d \\ w_d \end{bmatrix} = w_d \begin{bmatrix} x_d \\ y_d \\ 1 \end{bmatrix}$$

כעט האלגוריתם מחפש מטריצה H שתקיים לכל אחת מארבע זוגות הנק' :

$$w_d \begin{bmatrix} x_d \\ y_d \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & 1 \end{bmatrix} * \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

נשים לב שיש לנו 12 נעלמים – 8 דרגות חופש בתוך המטריצה H ועוד 4 scaling factor שונים, אחד לכל נקודה. למזלנו יש לנו 4 נקודות וכל אחד תורמת 3 משוואות אותן נוכל לקבל ע"י רישום מפורש של החישוב המטריציוני:

$$\begin{aligned} w_d x_d &= a_{11}x_i + a_{12}y_i + a_{13} \\ w_d y_d &= a_{21}x_i + a_{22}y_i + a_{23} \\ w_d &= a_{31}x_i + a_{32}y_i + 1 \end{aligned}$$

על ידי העברת אגפים ניתן לקבל:

$$\begin{aligned} 0 &= a_{11}x_i + a_{12}y_i + a_{13} - w_d x_d \\ 0 &= a_{21}x_i + a_{22}y_i + a_{23} - w_d y_d \\ -1 &= a_{31}x_i + a_{32}y_i - w_d \end{aligned}$$

נזכור שמשוואות אלו נכונות של $a_{ij}$ ולארבעת ה- $w_d$ . לכן נוכל לרשות ברישום מטריציוני מערכת של 12 משוואות עם 12 נעלמים:

$$\begin{bmatrix} x_{1i} & y_{1i} & 1 & 0 & 0 & 0 & 0 & 0 & -x_{1d} & 0 & 0 & 0 \\ 0 & 0 & 0 & x_{1i} & y_{1i} & 1 & 0 & 0 & -y_{1d} & 0 & 0 & 0 \\ & & & & & \vdots & & & & & & \\ 0 & 0 & 0 & 0 & 0 & 0 & x_{4i} & y_{4i} & -1 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} a_{11} \\ a_{12} \\ a_{13} \\ a_{21} \\ a_{22} \\ a_{23} \\ a_{31} \\ a_{32} \\ w_{1d} \\ w_{2d} \\ w_{3d} \\ w_{4d} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -1 \\ 0 \\ 0 \\ -1 \\ 0 \\ 0 \\ -1 \\ 0 \\ 0 \\ -1 \end{bmatrix}$$

נשים לב שלמערכת הנ"ל לוא דווקא קיים פתרון, שכן האלגוריתם מניח שארבע נק הידע להן אנחנו מחפשים לעשות טרנספורמציה אכן נמצאות על אותו המישור – מה שלא בהכרח נכון. תמיד יש רעש שנוסף לצילום ויכול להזיז מעט את הנקודות. על כן האלגוריתם מחפש את הפתרון הטוב ביותר לפי שיטת Least Squares. בשיטה זו משתמשים במטריצת pseudo-inverse לפתרון הבעיה , והפתרון המשוערך של x מתון ע"י:

$$x = (A^T A)^{-1} A^T b$$

לבסוף כדי האלגוריתם בונה את מטריצת הטרנספורמציה משמונה הקורדינוטות הראשונות של הפתרון המשוערך.

לאחר שמצאנו את מטריצת הטרנספורמציה השתמשנו בפקודה cv2.warpPerspective() על מנת לבצע את שינוי הפרספקטיבה. מאוחרי הקלעים הפקודה רצה על כל פיקסל התמונה, ממירה אותו לקורדינטות הומוגניות באופן זהה להסבר קודם, ומחשבת את הקורדינטות ההומוגנייות של הפיקסל אחרי הטרנספורמציה:

$$x_d = H * x_i^T$$

ולאחר מכן ממירה אותן לזרה לקורדינטות דו מימדיות. מאחר ואין הכרח כי יצאו ערכים שלמים, האלגוריתם משתמש באינטרפולציה כדי לשערך מה אמורים להיות ערכיהם של הפיקסלים בסביבת נק' היעד. קיימות שיטות אינטרפופלציה שונות, אנחנו השתמשנו בשיטת ברירת המחדל של bilinear interpolation, למרות שקיימות אפשרויות נוספות כגון nearest neighbor interpolation, cubic ועוד. לבסוף התמונה מותאמת לגודל התמונה המקורית.

איור 5א – התמונה המקורית

איור 5ב – התמנה המתוחה

ב. לאחר שינוי הפרספקטיבה החלטנו לנצל את הצבע הבוהק של הסרטים מהם מורכב המסלול כדי לזהות אותו בעזרת סגמנטציית צבע פשוטה. תחילה המרנו את התמונה למרחב הצבע Lab. זהו מרחב צבי שמייצג את הצבעים באופן התואם יותר את התפיסה האנושית. הוא מורכב משלושה ערוצים :

a. Lightness – המייצג את הבהירות של הפיקסל
b. a – המייצג את ערך הפיקסל על ציר אדום-ירוק, כאשר הערכים השליליים מצביעים על ירוק חזק וערכים חיוביים מצביעים על אדום חזק.
c. b - המייצג את ערך הפיקסל על ציר כחול-צהוב, כאשר הערכים השליליים מצביעים על כחול חזק וערכים חיוביים מצביעים על צהוב חזק.

כדי להמיר את ערוצי הערוצי ה-RGB לערוצי Lab תחילה ממירים את ערוצי הצבע למרחב צבע XYZ לפי

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \frac{1}{0.17697} \begin{bmatrix} 0.49 & 0.31 & 0.20 \\ 0.17697 & 0.81240 & 0.01063 \\ 0.00 & 0.01 & 0.99 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}.$$

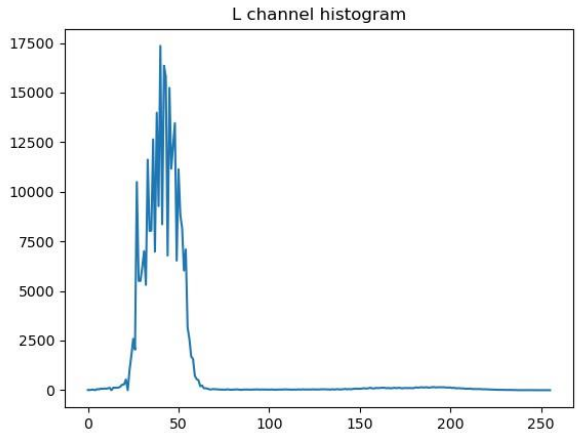
ולאחר מכן מרחב צבע זה ממירים ל-Lab לפי :

$$b^* = 200\left[f\left(\frac{Y}{Y_n}\right) - f\left(\frac{Z}{Z_n}\right)\right] \; a^* = 500\left[f\left(\frac{X}{X_n}\right) - f\left(\frac{Y}{Y_n}\right)\right] \; L^* = 116 f\left(\frac{Y}{Y_n}\right)$$

כאשר הפונקציה f(t) מוגדרת להיות:

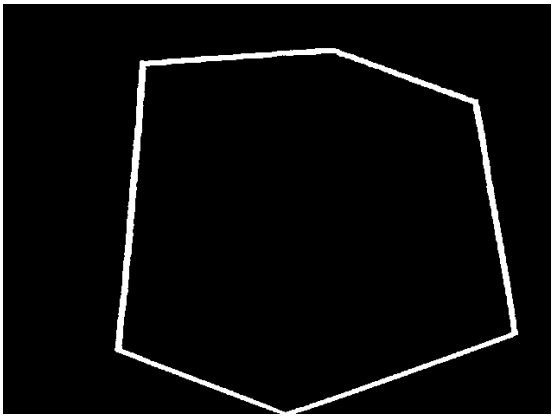
$$f(t) = \begin{cases} t^{1/3} & t > \delta^3 \\ t/(3\delta^2) + 2\delta/3 & \text{else,} \end{cases}$$

מאחר והשטיח בסיפריה בצבע כחול כהה ואילו הסרט צהוב, באופן טבעי ציפינו לעשות את הסגמנטציה לפי ערוץ b. לאחר בחינה של ערוצי הצבע הגענו למסקנה שהתוצאה הכי טובה תתקבל תוך שימוש בערוץ הבהירות. לאחר בחינה של ההיסטוגרמה של ערוץ L הגענו למסקנה שרוב הפיקסלים הכהים נמצאים מתחת לערך 70 בעוד שהפיקסלים הבהירים מתפרסים באופן יחסית אחיד על פני שאר הערכים לכן בחרנו ב70 כthreshold לסינון:




איור 6 – היסטוגרמה של ערוץ בהירות

לאחר הסינון השתמשנו בפילטרים של dilation ו-erosion כדי לסנן רעשים ולוודא שאנחנו מקבלים קו אחיד ורציף של המסלול.



איור 7 – המסלול לאחר סגמנטציה

ג. לאחר זיהוי ובידוד המסלול הסתמכנו על כך שהמסלול מורכב ממקטעים ישרים של סרטים וכל הפניות הן לא פניות חלקות (כלומר שהמשיק להן לא רציף בנק הפניה) כדי לזהות אותן בעזרת Harris Corner Detector. אלגוריתם זה מזהה פינות בתמונה ע"י חישוב מטריצת corner response – מטריצה שמציגה בכל פיקסל את מידת השייכות שלו לפינה בתמונה המקורית. האלגוריתם מחפש נק' שמתרחש בהן שינוי גדול בעת הזזת התמונה לכל אחד משמונת הכיוונים. בעצם, מחפשים נק' בתמונה עבורן הסכום של sum of squared differences יהיה מקסימלי:

$$E(u,v) = \sum_{x,y} w(x,y)[I(x+u,y+v) - I(x,y)]^2$$

כאשר I(u,v) – ערך הבהירות של הפיקסל.

W(x,y) – פוקנציית חלון שמתאפסת מעבר לגודל החלון המוגדר של ההזזה.

x,y – הזזות בכיוונים המתאימים.

את הביטוי בסוגריים ניתן לקרב קירוב מסדר ראשון של טיילור ולרשום בצורה הבאה:

$$E(u,v) \approx [u \quad v]\, M \begin{bmatrix} u \\ v \end{bmatrix}$$

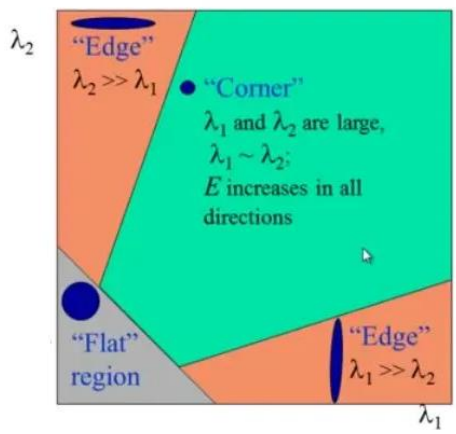
כאשר המטריצה M מקיימת:

$$M = \sum w(x,y) \begin{bmatrix} {I_x}^2 & I_x I_y \\ I_x I_y & {I_y}^2 \end{bmatrix}$$

מתוך חישוב הערכים העצמיים של המטריצה M ניתן לקבל הערכה על גודל השינוי בכיוון של הוקטור המתאים לערך העצמי – ככל שהערך העצמי גדול כך גם גודל השינוי. מתוך הערכים העצמיים הנ"ל האלגורים מחשב את ה-R score של הפיקסל לפני הנוסחא:

$$R = \lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2)^2$$

כאשר $\lambda_1, \lambda_2$ הם הערכים העצמיים של M ו-k הוא פרמטר המשפיע על מידת הרגישות של האלגוריתם לפינות. ערך גבוהה יותר של k יקטין את התחום עבורו באלגוריתם יחשיב את הפיקסל בתור פינה. ניתן לראות המחשה של התחום באיור הבא:




לאחר חישוב ה-R score של כל פיקסל ניתן להפעיל סינון ע"פ threshold כדי לסנן את כל הפיקסלים להם ה-R נמוך מידיי ולהישאר אך ולהישאר על הפיקסלים בהם כנראה יש פינות. בתוכנה עצמה השתמשנו בפקודה cv2.cornerHarris() המקבל כארומנטים את התמונה המקורית, גודל החלון, ערך הפרמטר k (אצלינו לאחר מספר ניסיונות מצאנו ש0.5 עובד טוב) ופרמטר נוסף בשם ksize המשמש את אלגוריתם בחישוב הגרדיאנטים בתמונה בעזרת sobel edge detector- אלגוריתם חיפוש שפות המשתמש בשני 3x3 convolution kernels הנראים כך:

| | | |
|---|---|---|
| -1 | 0 | +1 |
| -2 | 0 | +2 |
| -1 | 0 | +1 |

Gx

| | | |
|---|---|---|
| +1 | +2 | +1 |
| 0 | 0 | 0 |
| -1 | -2 | -1 |

Gy

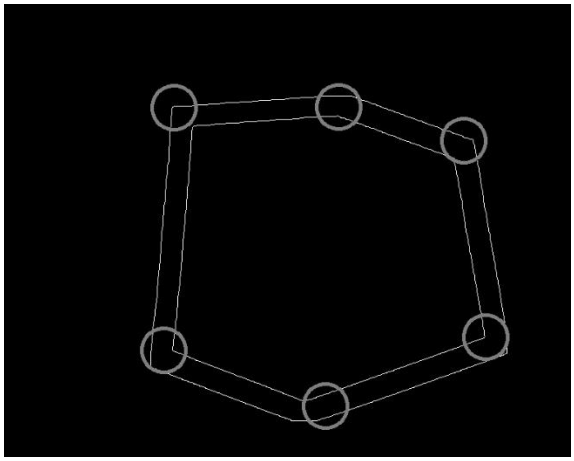
להלן דוגמא לharris response שקיבלנו:

איור 8 – המועמדים הכי טובים לפניות

לאחר מציאת כל הנק' השייכות לפניות במסלול הרצנו לולאה שמקצה לכל פניה נק' מרשימת הנק'. עבור כל נק' הלולאה מחשבת את המרחק מכל אחת מהנק' המזוהות כפניות ואם המרחק קטן מערך שנבחר בצורה אמפירית, אזי הפניה לה הנק' שייכת כבר סומנה ואפשר להעבור לנק' הבאה. אם לא, אז הנק' מתווספת לרשימת הפניות בתור "נציגה" של הפניה. סה"כ פניות במסלול יהיו כאורך רשימת הפניות. בזמן משחק ניתן לקחת המיקום של הרכב, ולבדוק את מרחקו מכל פניה כדי לעקוב אחר אילו פניות הרכב עבר והאם סיים הקפה מלאה.

את התמונה המלאה של המסלול נקבל ע"י ריפוד המסלול ב70 אפסים מכל צד, הפעלת פילטר dilation – חלון הרץ על הפיקסלים בתמונה ומחליף את הפיקסל עליו הוא יושב בערך המקסימלי שקיים בחלון. עשינו זאת על מנת לעבות את המסלול המיוצג ע"י קו דק למסלול עבה. את הפילטר הרצנו בגודל 3x3 20 איטרציות. את הבחירות עשינו בצורה אמפירית. ולבסוף הפעלנו canny edge detector כדי לצייר את המסלול עצמו. canny edge detector מחפש את השפות תחילה ע"י הפעלת פילטר גאוסיאן על מנת להוריד רעשים. לאחר מכן הוא משתמש בsobel operator על מנת לחפש את שפות. לאחר מכן מתבצע non-maximum suppression כדי להשאיר אך ורק את השפות הדומיננטיות ולבסוף מתבצע thresholding כדי לעשות סינון נוסף. על פניו מאחר ומדובר בתמונה בינארית בה השפות ברורות מאוד אין צורך בכל הסינונים והיה מספיק רק להשתמש בsobel operator אבל בחרנו להשתמש בcanny

מאחר ומדובר בשורת קוד אחת – מה שהופך את הקוד לנוח וברור יותר, ואנחנו לא מוגבלים בזמן חישוב בחלק זה של הקוד. להלן התוצאה הסופית:

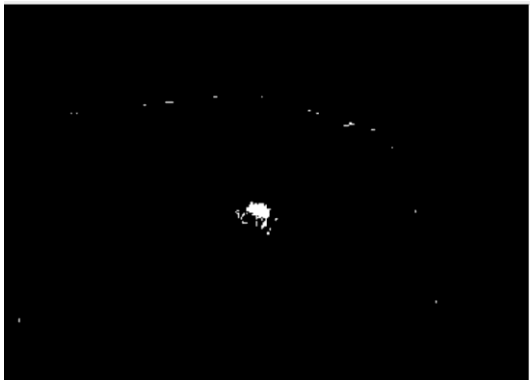


איור 9 – המסלול הסופי

## 3.3 עקיבה אחר הרכב:

עקיבת הרכבת מתבצעת ע"י לקיחת פריים מהמצלמת המסלול בשלב הנסיעה וניתוח שלה בזמן אמת. על מנת לזהות את מיקום הרכב בצורה יעילה כל פריים עובר עיבוד מסתגל, בוא מתבצעים השלבים הבאים:

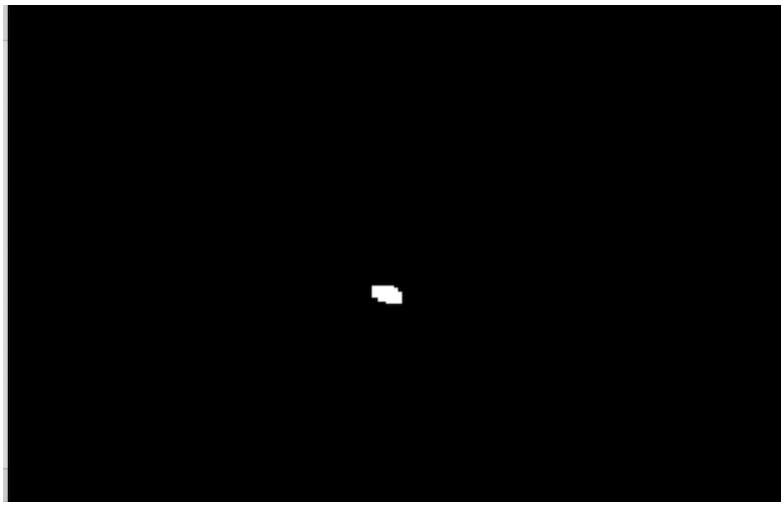
א. ראשית התמונה מומרת ממרחב הצבע RGB למרחב הצבע HSV- מרחב תבע המייצג את הצבעים לפי גוון (Hue), רוויה (Saturation) ובהירות (value). כאן בחרנו לנצל את העובדה כי הרכב הוא בצבע לבן בוהק לעומת הרצפה הכחולה כהה. לאחר מכן ביצענו binary thresholding בעזרת הפונקציה cv2.inRange לפי הערכים upper_white ו-lower_white – הגבול העליון והתחתון של הthresholding בהתאמה.מדובר השתי רשימות של 3 ערכים, אחד



איור 9 – סינון לפי צבע לזיהוי הרכב

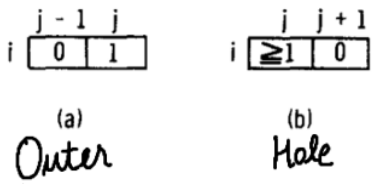
לכל ערוץ צבע. הגבולות הללו מחושבים בכל פריים מחדש לפי הפריים הקודם ע"י חישוב הממוצע בחלון סביב הרכב. זאת על מנת להסתגל לשינויים באלולים ליווצר בבהירות של הרכב בין פריים לפריים ולשפר את הדיוק. אם מדובר בפריים הראשון בתחילת הנסיעה אזי הערכים הם [0,0,168] ו-[172,111,255] עבור הגבול התחתון והעליון בהתאמה. הערכים נבחרו בצורה אמפירית כדי להוות נק' התחלה טובה.

ב. לאחר ביצוע הסגמנטציה התמונה בינארית עוברת פילטרציה של erosion לאחר מכן dilation.
Erosion filter מפעיל חלון שרץ על כל פיקסל בתמונה ומחליף את ערכו בערך המינימלי הנמצא בחלון. האפקט המתקבל הוא הפיכת הצורות בתמונה בינארית ל"רזות" יותר וגם להעלים רעשים בגודל של מספר פיקסלים בודדים.
Dilation filter מפעיל בדיוק את הפעולה ההפוכה – הוא רץ על כל פיקסל בתמונה ומחליף את ערכו בערך המקסימלי הנמצא בחלון. פילטר זה מעבה את העצמים בתמונה וגורם להן לראות יותר עבים.



איור 10 – אובייקט הרכב לאחר סינון רעשים

ג. לאחר קבלת התמונה הבינארית השתמשנו ב()cv2.findContours על מנת למצוא את כל קווי המתאר בתמונה. מאחורי הקלעים האלגוריתם סורק את התמונה הבינאריץ פיקסל אחר פיקסל עד אשר הוא מוצא פיקסל שערכו גדול מאפס. לאחר מכן נעשית בדיקה האם הפיקסל הזה הוא border pixel. קיימים שני סוגים של מתארים (borders)- מתאר חיצוני ומתאר פנימי.
הקריטריונים עבור כל מתאר מתוארים באיור מטה:




אם אכן מדובר בborder pixel אז מוסיפים אותו לרשימת הגבולות ומתחילים לבדוק את שכניו באופן סדרתי, לפי כיוון השעון. לאלגוריתם עוצר כאשר עבר על כל המתאר של האובייקט בתמונה וחזר לפיקסל המקורי ממנו התחיל את המעקב אחר המתאר או כאשר הגיע למבוי סתום – הגבול של הפריים או שאין עוד שכנים השייכים למתאר מלבד לפיקסל הקודם.

לאחר מכן האלגוריתם ממשיך בסריקה ומתחיל לאת המעקב שוב כאשר נתקל בפיקסל השייך למתאר אותו הוא לא הכיר לפני. לבסוף מוחזרת רשימה של קווי המתאר של התמונה ואופציונלית אובייקט המכיל מידע על ההיררכיה בין קווי המתאר, אילו קווי מתאר פנימיים ואילו הם קווי מתאר חיצוניים ומי מכיל את מי.

כעת, אנחנו מניחים כי לאחר הbinary threshold האוביקט הדומיננטי בתמונה שייך לרכב, לכן נחפש את המתאר הכי גדול (זה לו שייכיים הכי הרבה פיקסלים) ונחשב את הנעגל החוסם הקטן ביותר שמסוגל לחסום את המתאר הנ"ל. החישוב מתבצע ע"י cv2.minEnclosingCircle() אשר עוברת על כל הפיקסלים במתאר באופן רקורסיבי ובכל פעם מחשבת את המעגל הקטן ביותר החוסם תת קבוצה שלהם. בכל איטרציה נבחרת נק' אקראית מתוך המתאר ונבדק אם היא במעגל שחושב באיטרציה הקודמת. אם כן אז האלגוריתם ממשיך לפיקסל הרמדומלי הבא עד אשר הו יעבור על כל הפיקסלים. אם לא אז האלגוריתם מחשב את המעגל הקטן ביותר החוסם את המעגל הקודם והנק' הנוכחית ע"י ניצול העובדה שהנק' הנוכחית חייבת להיות על שפת המעגל החדש. לבסוף הפונקציה מחזירה את מרכז המעגל החוסם ואותו אנחנו מכפילים המטריצת הטרנספורמציה שחושבה בערת זיהוי המסלול על מנת לקבל את הקורדינטות של הרכב בתמונה המתוחה.

לאחר זיהוי מיקום הרכב אנחנו מחשבים את מהירות הרכב ע"י חישוב ההפרש במיקום בין הפריים הנוכחי לפריים הקודם וחלוקה בזמן העובר בין פריים לפריים – [msec] 0.033. בנוס, אנחנו מחשבים את כיוון הנסיעה חישוב הזווית ביחס לציר X ע"י tan2(y/x).

לבסוף כדי לשפר את זמן החיפוש של האובייקט בפריים אנחנו משתמשים במיקום והמהירות מהפריים הקודם כדי לשערך היכן האובייקט צפוי להיות בפריים הנוכחי ומחפשים בחלון סביבו. אם הרכב לא נמצא בחלון אז נחפש בכל הפריים.


איור 11- זיהוי מיקום רכב

## 3.4 הצגת המסלול על המסך:

הצגת המסלול בזמן אמת מסתמכת על תמונת המסלול שחושבה בפונקציית map_detection(), על מיקום הרכב ועל האריינטציה שלו. בעצם בכל פריים של המשחק, לאחר קבלת כל האינפוטים אנחנו גוזרים חלון בגודל של 70x70 סביב המיקום של הרכב ומסובבים אותו כך שהאוריינטציה שלו תהיה לכיוון מעלה. לאחר מכן אנחנו מגדילים את החלון לגודל של 700x700 ומציגים לשחקן. את המימוש ניתן לראות בקטע קוד הבא:

```
    def get_surrounding(self, img):
        window = get_window(img, self.location,  size: 70)
        window = cv2.resize(window,  dsize: (700, 700), interpolation=cv2.INTER_AREA)
        rot_mat = cv2.getRotationMatrix2D( center: (350, 350), self.orientation,  scale: 1.0)
        return cv2.warpAffine(window, rot_mat,  dsize: (700, 700), cv2.INTER_NEAREST, cv2.BORDER_CONSTANT)


1 usage   Nikolai Krokhmal
def get_window(track_img, car_pos, size):
    x = car_pos[0]
    y = car_pos[1]
    window = track_img[y-size:y+size, x-size: x+size]
    return window
```

## 4. אנליזה של אלגוריתם ה- SIFT :

בהמשך לסעיף 5 בו הסברנו על אלגוריתם ה-**SIFT** ועל הפרמטרים אותו הוא מקבל, ננתח כיצד שינוי פרמטרים משפיע על ביצועי האלגוריתם ולבסוף על אופן נסיעת הרכב בזמן אמת.

נמדוד את ביצועי האלגוריתם על סמך כמות הפיצ'רים הטובים בכל פריים כאשר מרכזי הידיים נמצאים על קו ישר ששיפועו אפס ( מצב נסיעה קדימה).

ההערכה הכמותית תתבצע על ידי ביצוע כל ניסוי עבור ערכי פרמטרים מסוימים 3 פעמים.

ניקח ממוצע של 10 מדידות בכל פעם ולבסוף נמצע בין שלוש הניסויים לקבלת כמות הפיצ'רים הטובים עבור כל סט פרמטרים שנקבע מראש.

הפרמטרים איתם הגענו לתוצאות הטובות ביותר הם :

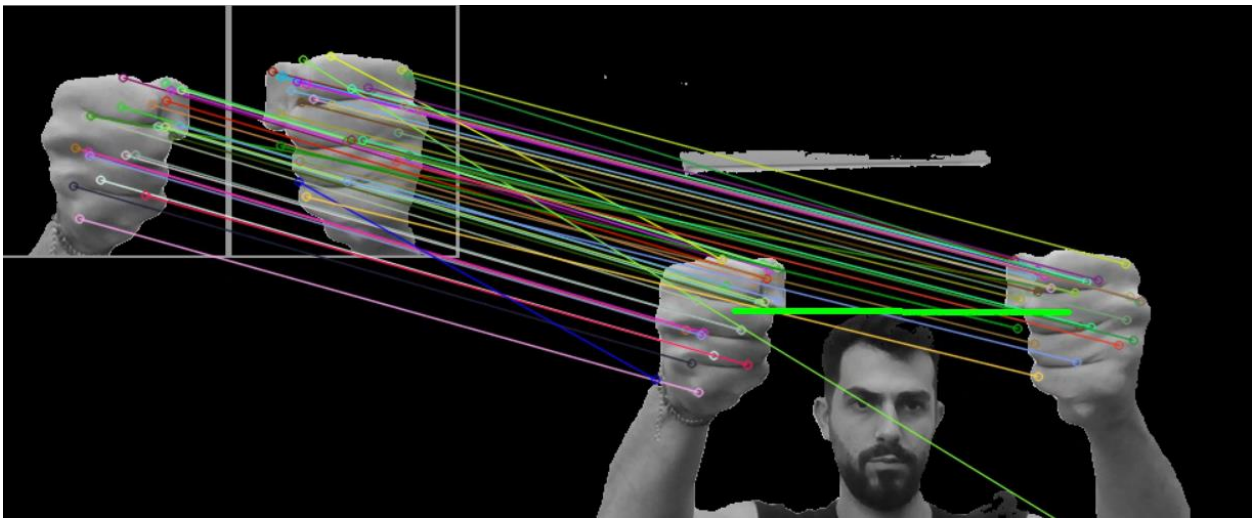
- **nOctaveLayers = 5**.
- **contrastThreshold=0.01**.
- **edgeThreshold=10**.
- **sigma=1.6**.

וכן בשימוש ב-**KNN** היחס בין 2 ההתאמות הכי טובות - **m.distance < C*n.distance** הינו **C**=0.7 .

| פרמטר / ערך משתנה | nOctaveLayers | contrastThreshold | edgeThreshold | sigma | C | כמות הפיצ'רים הטובים בכל ניסוי | ממוצע כמות הפיצ'רים |
|---|---|---|---|---|---|---|---|
| דיפולטיבי | 5 | 0.01 | 10 | 1.6 | 0.7 | 36.4<br>33.8<br>47.9 | 39.3 |
| **nOctaveLayers**<br>נמוך יותר | 3 | --- | --- | --- | --- | 16.7<br>19.9<br>19.7 | 18.7 |
| **contrastThreshold**<br>נמוך יותר | --- | 0.04 | --- | --- | --- | 21.2<br>18.6<br>22.3 | 20.7 |
| **edgeThreshold**<br>גבוה יותר | --- | --- | 50 | --- | --- | 32.3<br>20.6<br>23.2 | 25.3 |
| **Sigma** גבוהה יותר | --- | --- | --- | 2.3 | --- | 16.9<br>16.1<br>14.3 | 15.7 |
| **Sigma** קטן יותר | --- | --- | --- | 1 | --- | 57.2<br>60.5<br>61.8 | 59.3 |
| **C** קטן יותר | --- | --- | --- | --- | 0.65 | 14.6<br>14.4<br>14.4 | 14.4 |

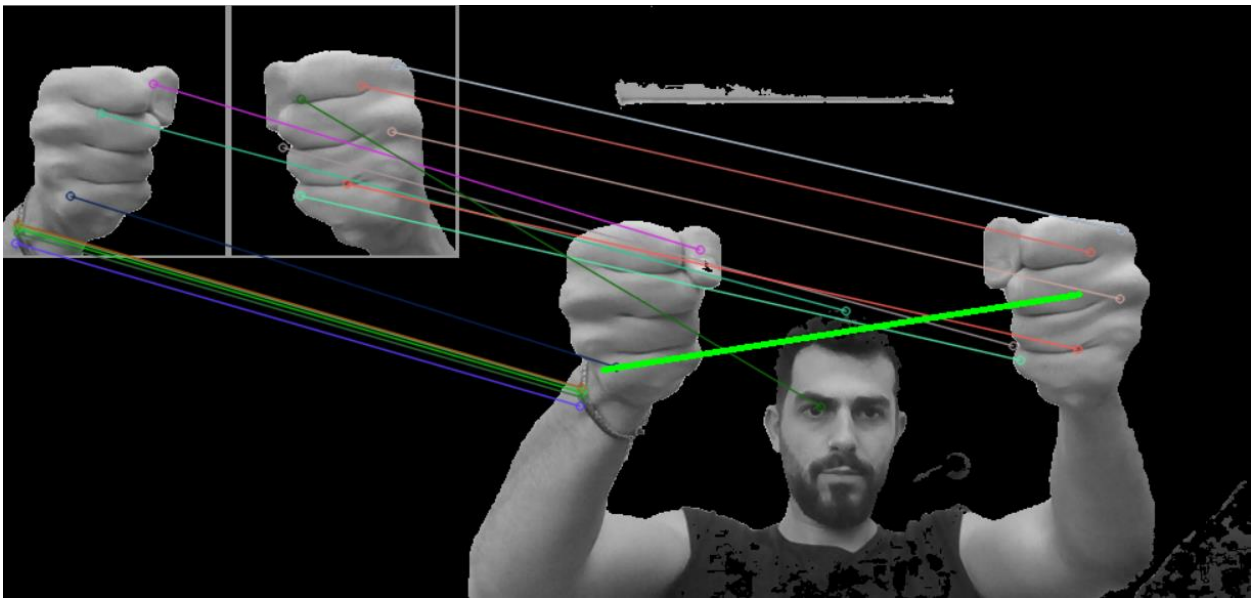
הערות :

**ערכים דיפולטיביים** – עבור בחירה זו של ערכים קיבלנו את כמות הפיצ'רים הטובים הגבוהה ביותר יחד עם כמות פיצ'רים לא נכונים מינימלית.



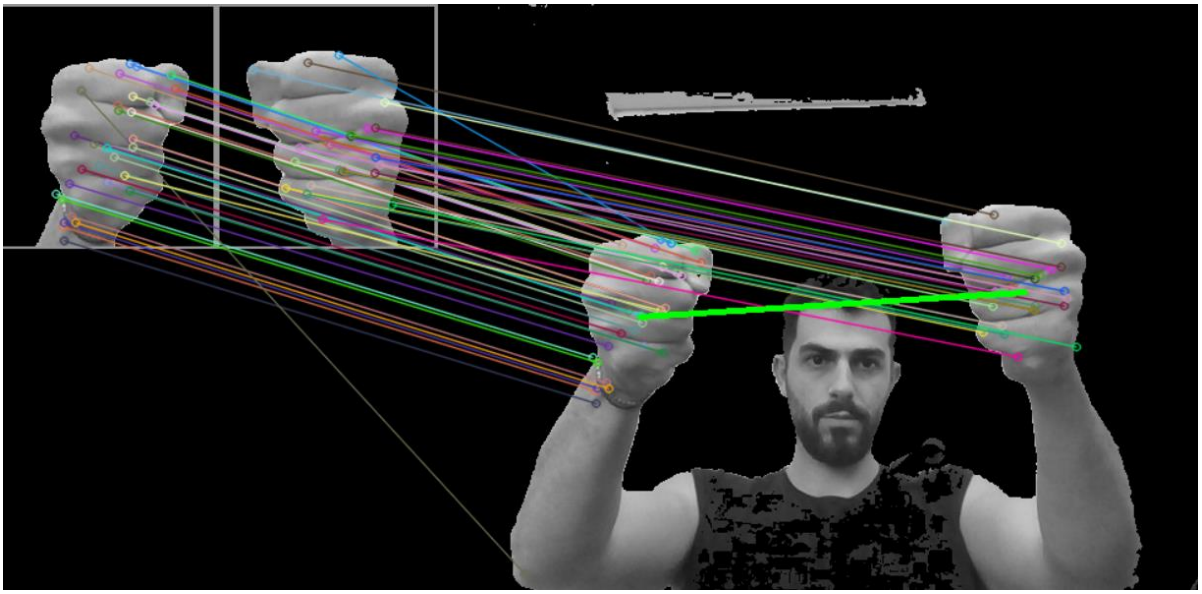
איור 12 – מציאת פיצרים ערכים דיפולטיביים

**contrastThreshold** נמוך יותר – כמות הפיצ'רים שסוננו גדלה אך יחד עם זאת, איבדנו המערכת הייתה פחות רובסטית לשינוי מנח הידיים בזמן אמת.



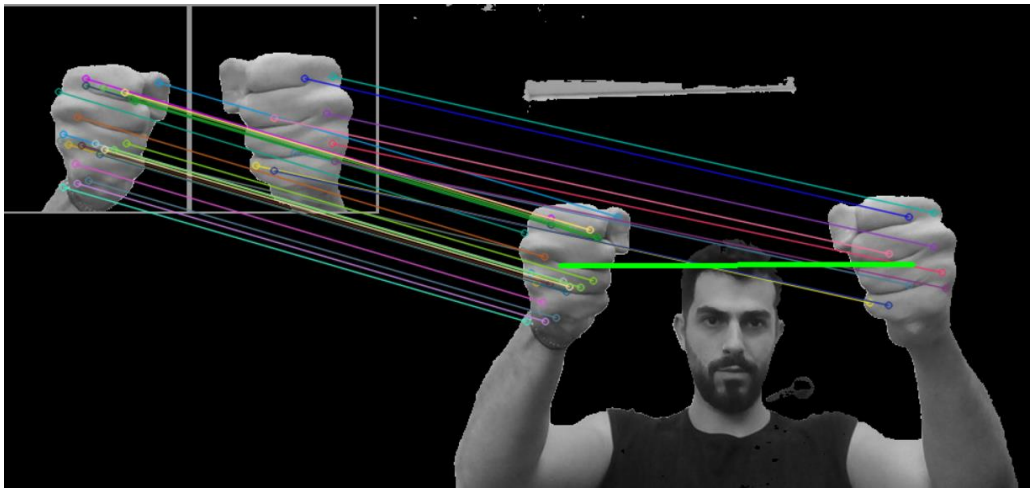
איור 13 – פיצרים אחרי הורדת contrast

**edgeThreshold** גבוה יותר – עבור שינוי ערך זה מ-10 ל-50 אמנם הביא ליותר מציאת פיצ'רים בממוצע (כפי שהסברנו בניתוח אלגוריתם ה-**SIFT**), אך עם זאת התקבלו הרבה פיצ'רים שאינם רצויים ומשבשים את רובסטיות האלגוריתם.



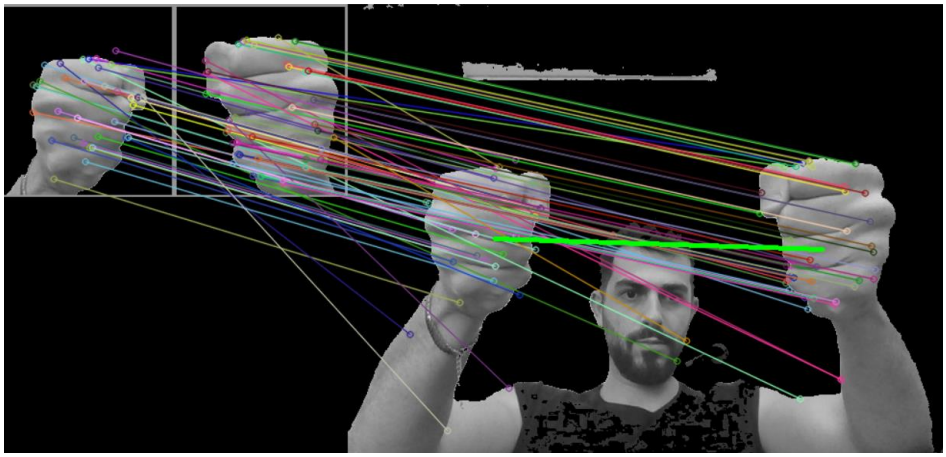
איור 14 – דוגמא לזיהוי על edge threshold גבוהה

**sigma** קטנה יותר – כאשר הקטנו את ערך זה, קיבלנו בממוצע כ-40 פיצ'רים – מספר גדול משמעותית יותר מאשר כל שינוי של פרמטר אחר. תוצאה זו נובעת כיוון שהפילטר הגאוסייני מסנן פחות פיצ'רים עבור ערך קטן יותר. לכן ערך קטן יותר של sigma יהיה שימושי עבור מצלמה באיכות נמוכה למשל.



איור 15 – ערך סיגמא קטן

**C** קטן יותר – עבור C קטן יותר שמגדיר את היחס בין 2 ההתאמות הטובות ביותר הערך הדיפולטיבי היה הערך בו קיבלנו את כמות הפיצ'רים הטובים הגדולה ביותר. כאשר הקטנו את C קיבלנו כמות פיצ'רים קטנה יותר משמעותית, מה שמשפיע על ביצועי האלגוריתם לרעה.



איור 16 – ערך C קטן

## 4 סיכום:

לסיכום, מטרת הפרויקט הושגה – הרכב נסע על פי תנועות הידיים של המשתמש.
הצלחנו לממש עקרונות של עיבוד תמונה שונים :

א. סגמנטציית תנועות ידיים בעזרת:

- מעבר לערוצי צבע רלוונטיים : ערוץ S של מודל HSV לבידוד הידיים מהרקע.
- אלגוריתם ה-SIFT לשם זיהוי פיצ'רים בתמונת רפרנס אל מול תמונות בזמן אמת.

ב. זיהוי מסלול:

- ביצענו מתיחה של התמונה על מנת לשנות פרספקטיבה למבט על ולראות את המסלול בצורה ברורה.
- הצלחנו לזהות היטב את המסלול בעזרת מעבר לערוץ Lab והפעלת thresholding לפי ערוץ b.
- השתמשנו בHarris corner detector על מנת לזהות את כל הפניות במסלול.
- השתמשנו בפילטר עיבוי ואחריו בזיהוי שפות כדי לצייר את המסלול עצמו.

ג. מעקב אחר הרכב:

- מעבר למרחב צבע HSV , על מנת לנצל את גוון הרכב לעומת הרקע
- שימוש בערכי סף המתאימים לצבע לבן (הרכב)
- אופרטורים מורפולגיים על מנת לסנן רעשים ולהוסיף עמידות.
- אלגוריתם find contours על מנת לזהות קווי מתאר.
- דינמיות בגבולות החיפוש מבחינת מיקום וצבע כדי לייעל את התהליך.

## 5 ביבליוגרפיה:


a. Szeliski, Richard. Computer vision: algorithms and applications. Springer Science & Business Media, 2010
b. . Gonzalez, Rafael C., and Richard E. Woods. "Image processing." Digital image processing 2 (2007).
c. Multiple View Geometry in Computer Vision
d. R. Hartley, and A. Zisserman. *Cambridge University Press, New York, NY, USA, 2 edition,* (*2003*)
e. Harris, Chris, and Mike Stephens. "A combined corner and edge detector." *Alvey vision conference*. Vol. 15. No. 50. 1988.
f. Suzuki, Satoshi. "Topological structural analysis of digitized binary images by border following." *Computer vision, graphics, and image processing* 30.1 (1985): 32-46.
g. Welzl, Emo. "Smallest enclosing disks (balls and ellipsoids)." *New Results and New Trends in Computer Science: Graz, Austria, June 20–21, 1991 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005.
h. OpenCV documentation - https://docs.opencv.org/4.x/index.html