T.C.
SANAYİ VE TEKNOLOJİ BAKANLIĞI
MİLLİ TEKNOLOJİ GENEL MÜDÜRLÜĞÜ

YAPAY ZEKA UZMANLIK PROGRAMI BİTİRME PROJESİ FİNAL RAPORU

PROJE ADI: Classic Computer Vision ile Nesne Tespiti

PROJE YÜRÜTÜCÜSÜ: Şevval Kolaca

MENTÖR : Buğrahan Bayraktar

RAPOR TARİHİ: 05.06.2024

İÇİNDEKİLER

1. ÖZET

Görsel odometri, otonom araçlarda, robotik uygulamalarda, artırılmış gerçeklik uygulamalarında ve hava araçlarında kullanılan bir teknolojidir. Görsel odometri, görüntü işleme tekniklerinin kullanılarak bir nesnenin hareketini izlemek ve belirlemek için kullanılmaktadır. Projenin odak noktası ise görüntü verilerini kullanarak aracın pozisyonunun tahmin edilmesidir.

Yapılan bu çalışmada "Görsel Odometri" problemi ele alınmaktadır. Bu projenin amacı ise görsel odometri yardımı ile, aracın yer belirleme sistemlerinin herhangi bir hatadan kaynaklı olarak kullanılmaz veya güvenilmez hale geldiği durumlarda görüntü verileri ile pozisyon kestirimi yapmaktır.

Yapılan çalışmalar kapsamında çeşitli algoritmalar araştırılmış ve değerlendirilmiştir. İncelenen algoritmalar arasından öznitelik tespit algorittması olarak SIFT algortimasının kullanılması uygun görülmüştür. Öznitelik eşleştirme algoritmaları arasından ise BFMatcher algoritması kullanılmıştır.

Proje kapsamında öncelikle görsellerdeki anahtar noktalar tespit edilmektedir. Tespit edilen özellikler, ardışık görüntüler arasında eşleştirilmektedir. Eşleştirilen görsellerden elde edilen bilgiler ile hareket vekörleri hesaplanmaktadır. Hesaplanan hareket vektörleri yardımıyla yörünge (trajectory) çizimi gerçekleştirilmektedir. Elde edilen yörünge ile araca ait hassas konum bilgileri ile oluşturulmuş yörünge karşılaştırılarak hata değerleri hesaplanmaktadır.

2. PROJE AMACI

Projenin amacı, tek kameralı görsel odometri teknolojisi yardımıyla araca ait kameradan alınan ardışık görüntüler kullanılarak aracın hareketinin tahmin edilmesidir. Özellikle aracın konum belirleme sistemlerinin güvenilmez olduğu veya çalışmadığı durumlarda, sadece görsel verilerin kullanılması ile pozisyon kestirimi yapılması amaçlanmıştır.

4. MATERYAL ve METOT

Proje kapsamında, veri olarak öncelikle kitti veri seti içerisinden ardışık görseller kullanılarak özellik çıkartma algoritmaları test edilmiş, projenin geri kalan aşamalarında ise mentörümüz tarafından sağlanan ve hava aracının uçuşu esnasında kaydedilmiş görüntülerin bulunduğu ground truth verileri kullanılmıştır. Kullanılan görseller, 30 fps hızında çekilmiş bir videodan elde edilmiştir. Toplamda elde edilmiş olan görsel sayısı 3222'dir. Bu görüntüler ardışık olarak işlenmiştir.

Kullanılan veri setinde, hava aracına ait hassas konum bilgilerinin bulunduğu bir CSV dosyası da bulunmaktadır. Bu dosya, hava aracının hareketinin görselleştirilmesi için kullanılmaktadır.

Özellik çıkartma teknikleri önceki aşamalarda test edilmiş ve en etkili algoritma olarak SIFT algoritması seçilmiştir. Özellik eşleştirme işlemleri ise BFMatcher kullanılarak gerçekleştirilmiştir.

Görsellerin işlenmesinde karşılaşılan sorunlar PIL ve OpenCV kütüphaneleri kullanılarak çözülmüştür.

Öznitelik eşleştirme sonucu elde edilen veriler bir CSV dosyasına kaydedilmiştir. Dosya içerisindeki veriler kullanılarak ve Matplotlib kütüphanesi yardımıyla yörünge çizimi gerçekleştirilmiştir.

Proje süresince Python programlama dili, Colab Notebook ve Visual Studio Code kullanılmaktadır.

5. YAPILAN ÇALIŞMALAR

Proje kapsamında görsellerin işlenmesi ve özniteliklerin tespiti için için OpenCV (Open Source Computer Vision Library) ve PIL (Python Imaging Library) kütüphaneleri kullanılmıştır. OpenCV, bilgisayarlı görü ve yapay zeka uygulamaları için geliştirilmiş olan açık kaynaklı bir kütüphanedir. Görüntü işleme, nesne tanıma, öznitelik tespiti ve diğer görüntü işleme problemlerinde sıklıkla kullanılmaktadır [1]. PIL ise python programlama dili için özel olarak geliştirilmiş ve görüntü işleme fonksiyonları sağlayan bir kütüphanedir. PIL, farklı görüntü formatlarını okuma, yazma ve dönüştürme işlmelerini desteklemektedir [2].

Özellik tespiti için SIFT (Scale-Invariant Feature Transform), SURF (Speeded Up Robust Features), ORB (Oriented FAST and rotated BRIEF) ve Harris Köşe Dedektörü yöntemleri incelenmiştir [3]. Analiz Tablo 1'de görünmektedir.

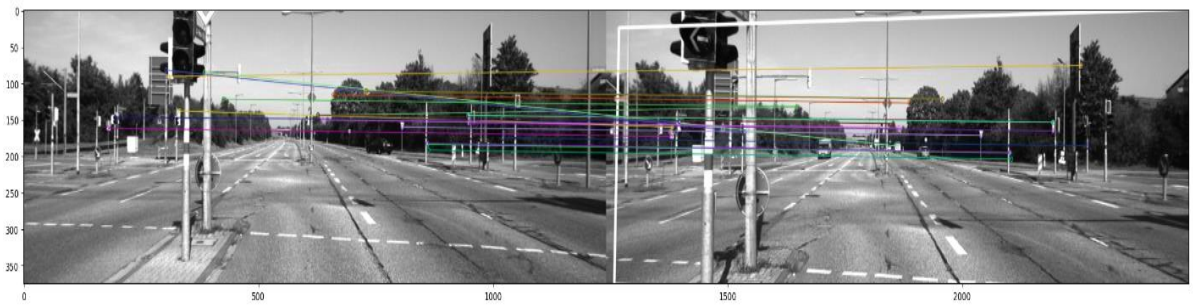
	Scale Invariant Feature Transform (SIFT)	Speeded Up Robust Features (SURF)	Oriented FAST and Rotated BRIEF (ORB)	Harris Köşe Dedektörü
Özellik Tespiti	Ölçek ve dönüşüme dayanıklı	Ölçek ve dönüşüme dayanıklı, hızlı tespit	Dönüşüme dayanıklı, ölçek dayanıklılığı yok	Köşe noktalarını tespit eder
Hız	Yavaş	SIFT'ten daha hızlı	SURF ve SIFT'ten çok daha hızlı	Çok hızlı
Hassasiyet	Yüksek hassasiyet, ölçek değişikliklerine çok dayanıklı	SIFT'ten biraz daha düşük hassasiyet	Orta hassasiyet, özellikle düşük doku alanlarında düşer	Düşük doku bölgelerinde sınırlı performans
Kullanım Alanları	Geniş kapsamlı görüntü ve video analizi	Geniş ölçekli ve gerçek zamanlı uygulamalar	Gerçek zamanlı uygulamalar için ideal	Görüntü hizalama, görüntü takibi
Dayanıklılık	Dönüşüm, ölçek ve aydınlatma değişikliklerine dayanıklı	Dönüşüm ve aydınlatma değişikliklerine dayanıklı	Yalnızca dönüşüm değişikliklerine dayanıklı	Dönüşüm ve aydınlatma değişikliklerine dayanıklı değil
Lisans Kaynaklı Kullanım Problemi	Yok	Var	Yok	Yok

Tablo 1. Özellik Tespit Yöntemleri

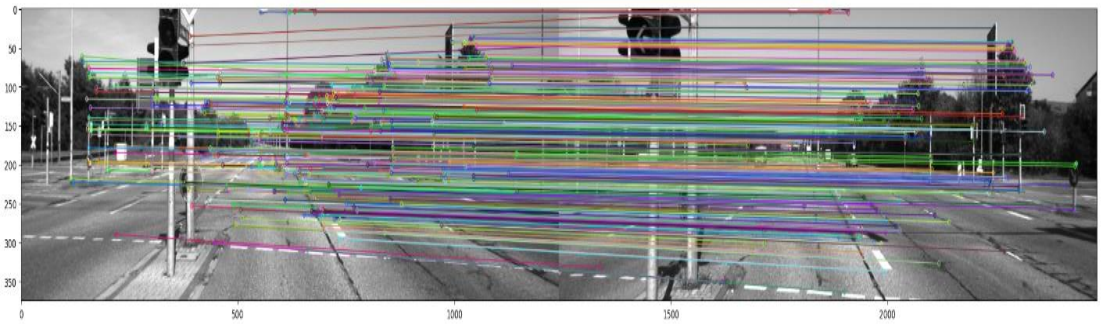
Tablo sonucunda elde ettiğimiz bilgiler şu şekildedir.

- SIFT: En güçlü ve hassas özellik tanımlayıcısıdır ancak yavaş çalışmaktadır.
- SURF: SIFT'a benzer bir şekilde çalışmaktadır ancak daha hızlıdır. Ölçek ve dönüşüm farklılıklarına SIFT dayanıklı değildir. Gerçek zamanlı uygulamalar için uygundur. Lisans Problemi kullanım kısıtlarına sebep olmaktadır.
- ORB: Çok hızlı çalışmaktadır ve gerçek zamanlı uygulamalar için uygundur ancak ölçek değişikliklerine fazla dayanıklı değildir.
- Harris Köşe Dedektörü: Çok hızlıdır ancak ölçek değişikliklerine dayanıklı değildir.

Bu değerlendirmeler sonucunda öncelikle SURF algoritması kullanılması planlanmıştır ancak lisans problemi sebebiyle bu algoritmadan vazgeçilmiştir. Tablo incelendiğinde SURF algoritması ardından en avantajlı iki algoritma olan SIFT ve ORB algoritmaları değerlendirilmiştir. Bu değerlendirmeler Kitti veri seti içerisinden alınan ardışık görseller ile gerçekleştirilmiştir. Öncelikle SIFT ve ORB algoritmaları ile anahtar noktalar bulunmuştur. Sonrasında BFMatcher (Brute-Force Matcher) algoritması ile bulunan öznitelikler eşleştirilmiştir. BFMatcher, OpenCV kütüphanesi içerisinde bulunan ve kNN (k-Nearest Neighbors) algoritmasını kullanan bir yöntemdir. Bir görüntüdeki öznitelikleri diğer görüntüdeki özniteliklerle karşılaştırarak en yakın öznitelik çiftlerini belirlemektedir [4] Performans değerlendirilmesi için bulunan öznitelik çiftleri Lowe oran testi ile test edilmiştir. Lowe oran testi, öznitelik eşleştirme işlemlerinde aykırı değerleri filtrelemek için kullanılan bir tekniktir. Bu test, her öznitelik noktası için en iyi iki eşleşmeyi bulur ve bu eşleşmeler arasındaki mesafe oranını hesaplar. Bu sayede, doğru eşleşmeleri tespit ederken yanlış eşleşmeleri (aykırı değerleri) filtreler [5]. Yapılan işlemler sonucunda elde edilen sonuçlar Şekil 1 ve 2'de gösterilmektedir.



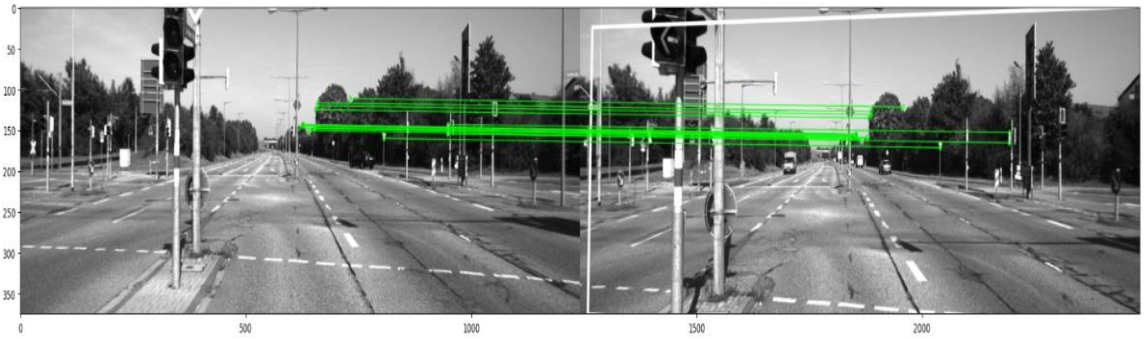
Şekil 1. ORB Algoritması Anahtar Eşleştirilmesi



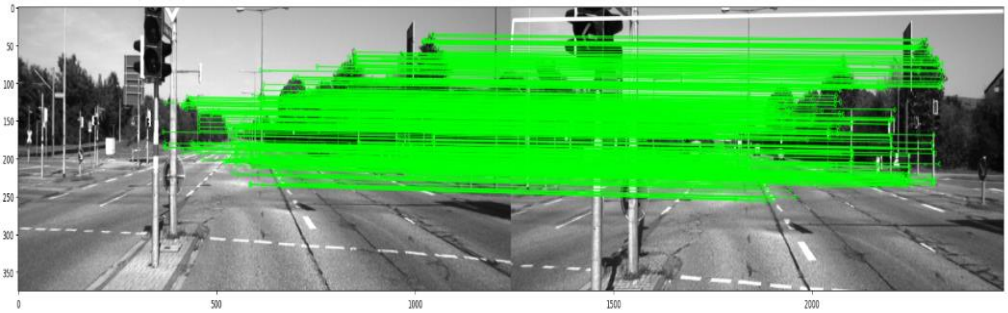
Şekil 2. SIFT Algoritması Anahtar Eşleştirilmesi

ORB algoritması ile elde edilen toplam anahtar çifti sayısı 500 olarak bulunmuştur. Lowe oran testi ile bulunan iyi eşleşme sayısı ise 44 olarak bulunmuştur. SIFT algoritması ile bulunan toplam anahtar çifti sayısı ise 3086'dır ve yapılan Lowe oran testi ile elde edilen en iyi eşleşme sayısı 873'tür. Bu sonuçlarla doğruluk oranları hesaplanmış ve SIFT algoritmasının daha iyi sonuç verdiği sonucuna varılmıştır.

Hesaplanan doğruluk oranları tek başına tam güvenilirlik sağlamadığı için araştırma sürecinde karşılaşılmış olan yöntemlerden bir tanesi olan RANSAC ile homografi hesaplama işlemi yapılmıştır. RANSAC yöntemi, özellik eşleştirmelerinin aykırı değerlerden temizlenmesi için kullanılmaktadır. Bu süreç, hareketli nesnelerden kaynaklanan özellik eşleştirmelerindeki aykırı değerlerin giderilmesinde kritik bir rol oynamaktadır. Aykırı değerlerin belirlenmesi ve aykırı değerlerin arındırılması süreci, özellik eşleştirmelerinde doğruluk ve sağlamlığı artırmak için önemlidir [6]. Ayrıca RANSAC yöntemi, en iyi eşleşen eğriler bulunduktan sonra referans ve hedef resim arasındaki geometrik dönüşümün tahmin edilmesi ve görüntülerin çakıştırılması için kullanılmaktadır. Bu kullanım, iki görüntü arasında sağlam bir geometrik ilişki kurulmasını sağlayarak, eşleştirme ve çakıştırma işlemlerinin başarısını artırmaktadır [7]. Bahsedilen her iki durumda da RANSAC, uygun iç değerleri (inliers) belirlemek için kullanılmaktadır. Bu şekilde genel doğruluk da artmaktadır. Aşağıda gösterilmekte olan Resim 3 ve 4'te ORB ve SIFT algoritmaları sonucunda uygulanmış RANSAC sonuçları gösterilmektedir.



Şekil 3. ORB-RANSAC sonucu



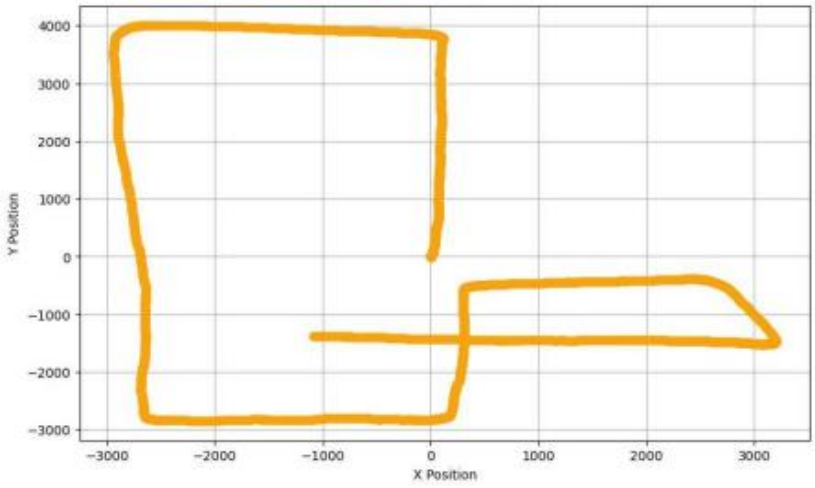
Şekil 4. SIFT-RANSAC sonucu

ORB ile bulunan öznitelikler arasından RANSAC sonucunda eşleşen öznitelik sayısı 22 olarak bulunmuştur. SIFT ile bulunan öznitelikler arasından RANSAC ile bulunan eşleşen öznitelik sayısı ise 729'dur. Bu veriler göz önünde bulundurulduğunda problem kapsamında SIFT kullanılması kararı alınmıştır.

Projenin devamında Ground Truth veriler kullanılmıştır. Ground Truth verilerinin okunması esnasında OpenCV kütüphanesi, görsel türü sebebiyle hata vermektedir. Bu sebeple görseller PIL kütüphanesi kullanılarak okunmuş ardından OpenCV formatına dönüştürülmüştür.

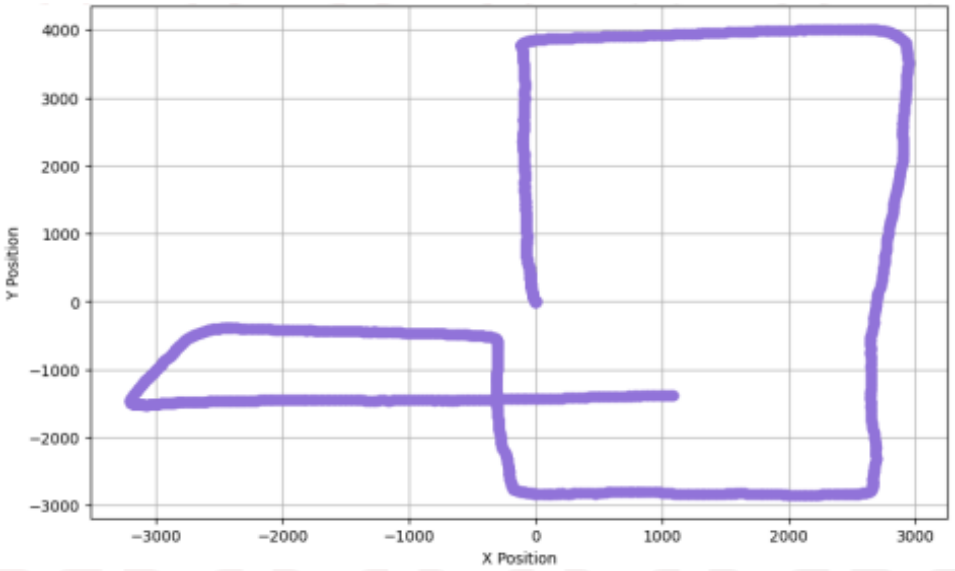
Kullanılmakta olan işletim sistemi Windows'tur. Windows işletim sistemi, dosyaları okurken alfabetik olarak sıralamaktadır. Eğer dosya adları, bizim verilerimizde kullandığımız gibi, sayı değerleri içeriyorsa ve bu sayılar farklı basamak değerlerine sahiplerse dosyaların mantıksal sırası bozulabilmektedir. Proje kapsamında ardışık görüntülerin kullanılması zorunlu bir durum olduğundan bahsedilen poblem istenilmeyen bir durumdur. Bu problemin önüne geçebilmek adına görsel adlarını dikkate alacak bir fonksiyon eklenmiştir.

Hareket vektörleri, ardışık görüntülerdeki öznitelik noktaları arasındaki farklar kullanılarak hesaplanmıştır. Hesaplanan hareket vektörleri ile Şekil 5'te görünmekte olan yörünge oluşturulmuştur.



Şekil 5. Öznitelik eşleştirme sonucu elde edilen yörünge

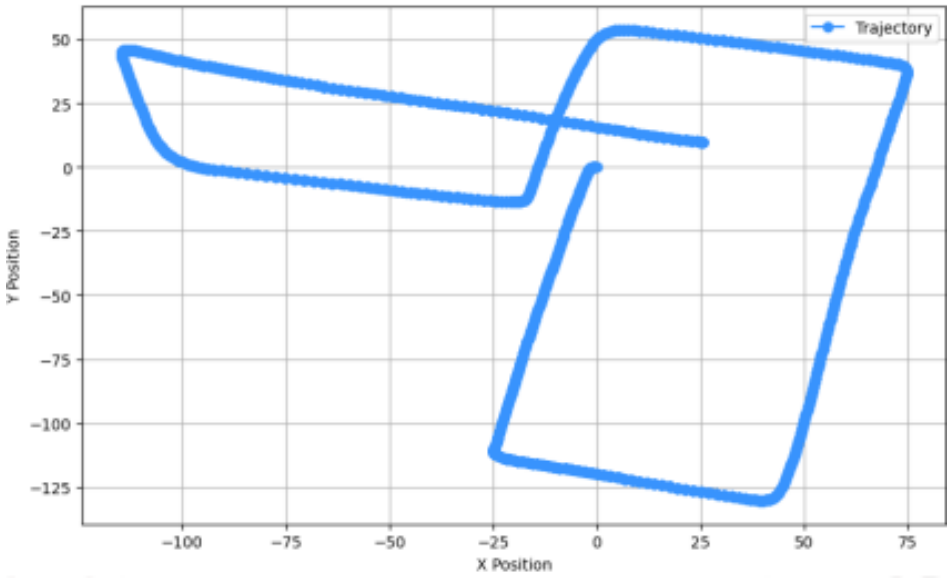
Yörünge çizimi için Matplotlib kütüphanesi kullanılmıştır. Matplotlib kütüphanesi, Python programlama dilinde veri görselleştirme için kullanılan bir kütüphanedir [8]. Şekil 5'teki çizilen yörünge ve veri seti içerisindeki video kaydı karşılaştırıldığında yörünge çiziminde eksen hatası olduğu anlaşılmaktadır. Hava aracının izlediği yol, y ekseni boyunca doğrudur ancak x ekseni göz önüne alındığında tam tersi bir şekilde yörünge çizildiği görülmüştür. Bu problem, görsel odometri algoritmalarının ve matplotlib kütüphanesinin koordinat sistemlerini farklı şekilde yorumlamasından kaynaklanmaktadır. Bu problemin önüne geçmek için uygun dönüşümler yapılmış ve Şekil 6'daki yörünge elde edilmiştir.



Şekil 6. Dönüşüm uygulanmış yörünge

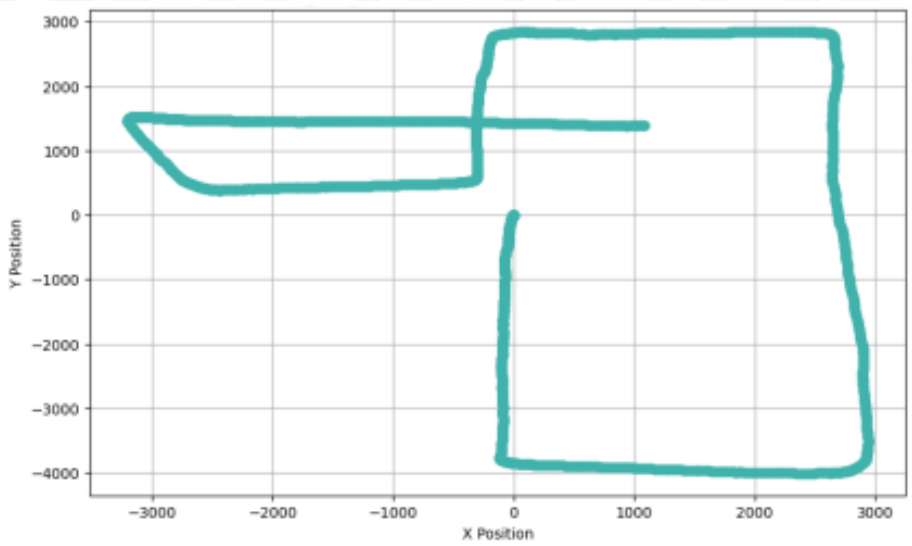
Şekil 6'da gösterilmekte olan yörünge, video kaydı ile karşılaştırıldığında hava aracının hareketiyle örtüşmektedir.

Elde edilen yörüngenin doğruluğunun tespit edilmesi ve hata oranlarının hesaplanması için veri seti içerisinde bulunan ve hava aracının hassas konum verilerini içeren CSV dosyası kullanılarak hava aracına ait gerçek yörünge çizilmiştir. Çizilen gerçek yörünge Şekil 7'de gösterilmektedir.



Şekil 7. Ground Truth verilerine ait yörünge

Görsel odometri ile elde edilen yörünge ve hava aracına ait gerçek yörünge karşılaştırıldığında eksen hatası olduğu gözlemlenmektedir. Bu problemin sebebi daha önce de bahsedildiği gibi görsel odometrinin koordinat sistemini farklı yorumlamasından kaynaklanmaktadır. Dönüşüm uygulanarak bu problem çözülmüştür. Dönüşüm sonucu elde edilen yörünge Şekil 8'de gösterilmektedir.



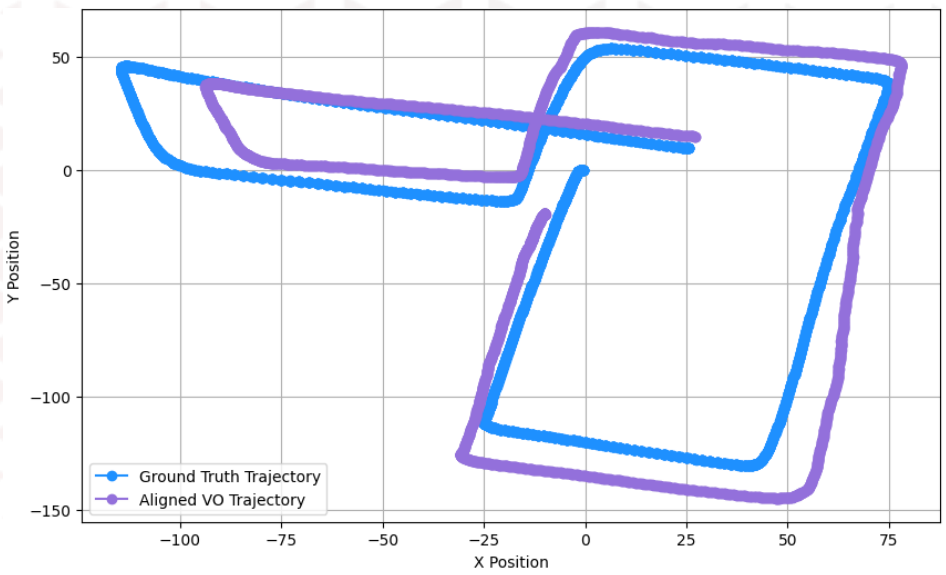
Şekil 8. İkinci dönüşüm sonucu oluşturulan yörünge

Şekil 7 ve 8 incelendiğinde ve karşılaştırıldığında derece ve ölçek farklılıkları olduğu gözlemlenmektedir. Gerçek yörünge çizimi için kullanılan CSV dosyasının içeriği hassas sensör verileri ile oluşturulduğundan bu şekilde farklılıklarla karşılaşmak normaldir. Elde edilen yörünge sonuçlarını optimum hale getirebilmek için yörüngeler üzerinden hizalama

işlemi yapılmıştır. Hizalama işlemi, iki farklı koordinat sisteminin aynı referans sistemine getirmek için kullanılan bir yöntemdir. İçerdiği adımlar şu şekildedir.

- *Ölçekleme:* İki veri seti arasındaki ölçek farklılıklarını gidermek için kullanılmaktadır.
- *Dönme:* İki veri seti arasındaki açısal farklılıkları düzeltmek için kullanılmaktadır.
- *Kaydırma:* İki veri seti arasındaki pozisyon farklılıklarını gidemek için kullanılmaktadır.

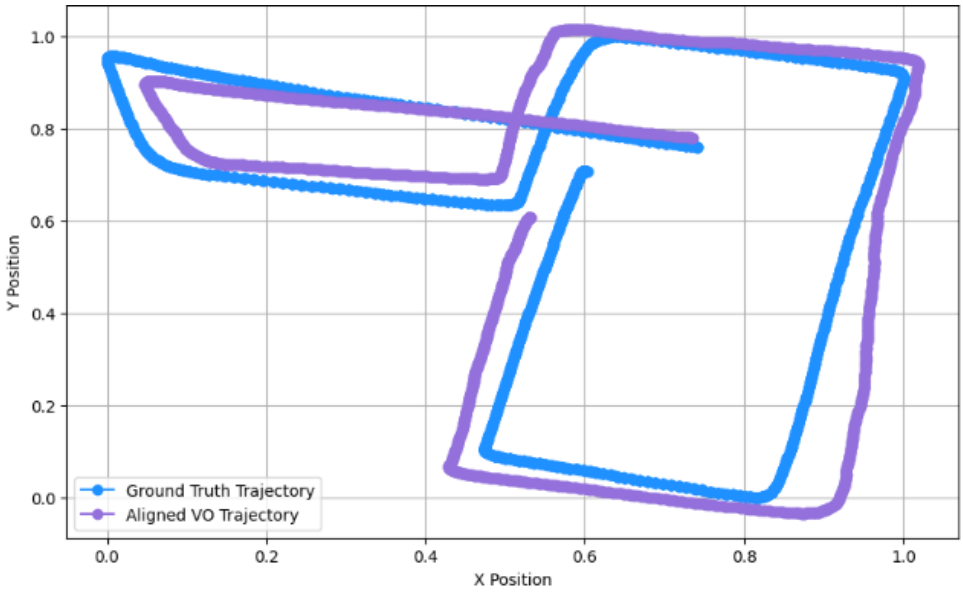
Proje kapsamında hizalama işlemi için ICP (Iterative Closest Point) ve Procrustes analizi kullanılmış ve test edilmiştir. ICP, iki nokta bulutu arasındaki en iyi rotasyon ve translasyon dönüşümünü bulmak için kullanılan bir algoritmadır. Özellikle robot navigasyonu ve haritalama gibi uygulamalarda yaygın olarak kullanılmaktadır. [9]. Bu aşamadan itibaren hizalanmış yörüngeler daha rahat analiz edilebilmesi adına beraber görselleştirilecektir. Şekil 9'da ICP sonrası hizalanmış yörüngeler gösterilmektedir.



Şekil 9. ICP sonrası hizalanmış yörüngeler

Yapılan ICP işlemi sonucunda görsel odometri yörüngesinin, ground truth verileri ile oluşturulan yörüngeye daha iyi bir şekilde uyum sağladığı gözlemlenmektedir.

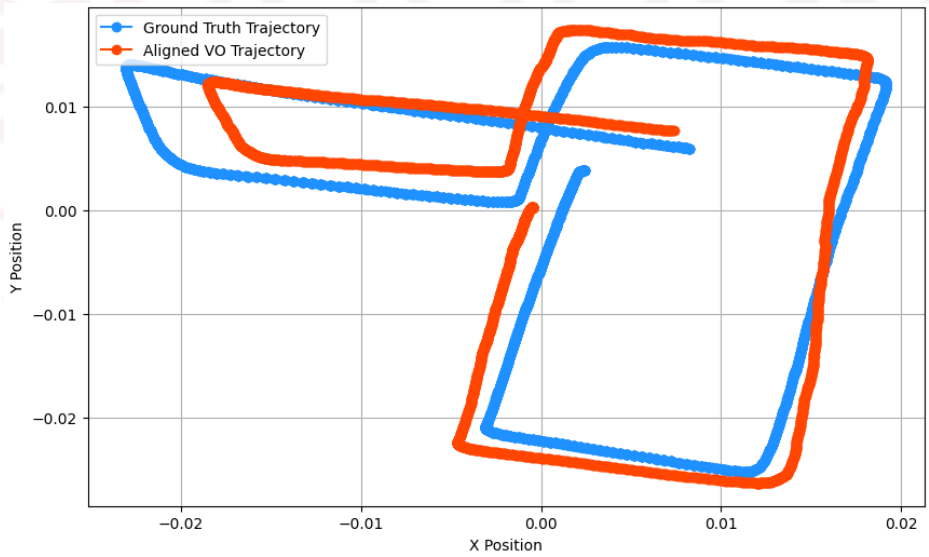
Görsel odometri yörüngesi ve ground truth verileri ile oluşturulan yörünge karşılaştırıldığında büyük bir ölçek farkı olduğu görülmektedir. Hizalama işleminin daha sağlıklı gerçekleşmesi için bahsedilen ölçek farkının önüne geçmek adına görsel odometri verilerine normalizasyon işlemi yapılarak ICP hizalama işlemi tekrarlanmıştır. Normalizasyon sonucu oluşturulan yörünge Şekil 10'da gösterilmektedir.



Şekil 10. Normalizasyon uygulanmış ICP ile hizalanmış yörüngeler

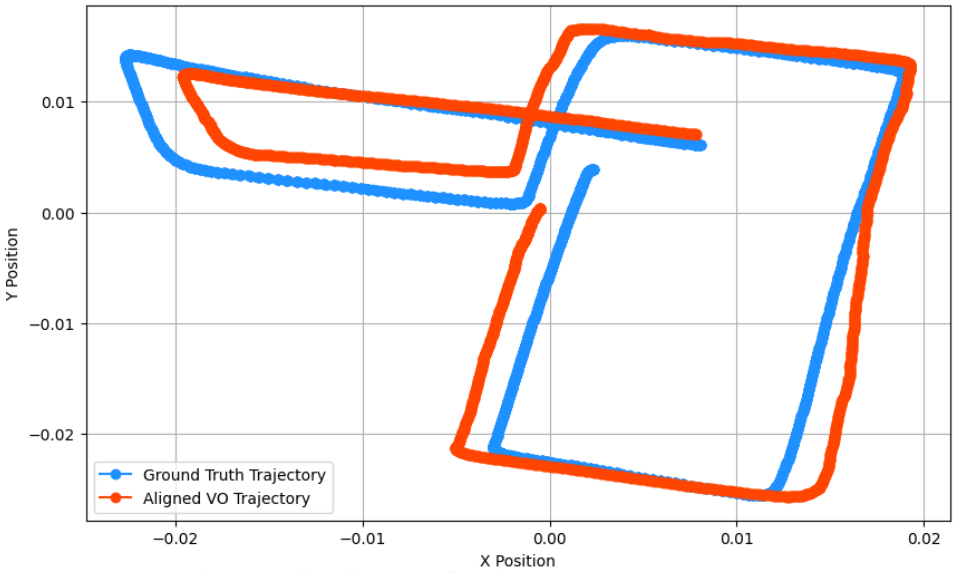
Şekil 9 ve 10 karşılaştırıldığında, normalizasyon uygulanmış verilerle uygulanan ICP hizalama işleminin daha başarılı gerçekleştiği gözlemlenmektedir.

Yörünge hizalama işlemi için tercih edilen bir diğer yöntem ise Procrustes analizidir. Procrustes analizi, şekil karşılaştırma ve hizalama için kullanılan bir yöntemdir. Bu analiz, şekilleri ölçek, çeviri ve dönme dönüşümlerine tabi tutarak en iyi hizalamayı bulmaya çalışmaktadır [10]. Procrustes analizi sonucu hizalanmış yörüngeler Şekil 11'de gösterilmektedir.



Şekil 11. Procrustes Analizi sonucu hizalanan yörüngeler

Procrustes analizi sonucunda görsel odometri ile oluşturulan yörüngenin, ground truth verileri ile oluşturulan yörüngeyeuyum sağladığı gözlemlenmiştir. ICP için gerçekleşteştirilen normalizasyon işlemi, Procrustes analizi öncesinde de uygulanmıştır. Normalizasyon işlemi sonrasında uygulanan Procrustes analizi sonuçları Şekil 12'de gösterilmektedir.



Şekil 12. Normalizasyon sonrası Procrustes Analizi ile hizalanan yörüngeler

Şekil 11 ve 12 karşılaştırıldığında, normalizasyon uygulanmış verilerle uygulanan Procrustes analizi işleminin daha başarılı gerçekleştiği gözlemlenmektedir.

6. UYGULAMA

Proje kapsamında kullanılan kütüphaneler Şekil 13'te gösterilmektedir.

```
import cv2
import matplotlib.pyplot as plt
import numpy as np
from glob import glob
import re   # regular expression -> düzenli ifadeler ile metin işleme
import os
from PIL import Image
import numpy as np
import csv
import pandas as pd
from scipy.spatial import procrustes
from sklearn.neighbors import NearestNeighbors
```

Şekil 13. Proje kapsamında kullanılan kütüphaneler

Öznitelik çıkartma algoritmalarının test edilmesi için Kitti veri setinden alınan ardışık görüntülerin okunması için kullanılan kod parçası Şekil 14'te ve kod çıktısı Şekil 15'te gösterilmektedir.

```
# Görüntüleri yükle
img1 = cv2.imread('1.png', cv2.IMREAD_GRAYSCALE)  # Sorgu görüntüsü
img2 = cv2.imread('2.png', cv2.IMREAD_GRAYSCALE)  # Eğitim görüntüsü

# Görüntüleri göster
plt.figure(figsize=(30, 20))

plt.subplot(1, 2, 1)
plt.imshow(img1, cmap='gray')
plt.title('image1')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(img2, cmap='gray')
plt.title('image2')
plt.axis('off')

plt.show()
```

Şekil 14. Kitti veri setine ait görsellerin okunması

Şekil 15. Okunan görsellerin çıktısı

ORB algoritması ile anahtar noktaların belirlendiği, BFMatcher yöntemi ile bulunan anahtar noktalarının eşleştirildiği ve Lowe Oran Testi ile iyi eşleşmelerin ayıklandığı kod parçası Şekil 16'da gösterilmektedir.

```python
# ORB tanımlayıcıyı başlat
orb = cv2.ORB_create()

# Özellik tespiti (keypoint detection) ve tanımlayıcı (descriptor) hesaplama
keypoints1, descriptors1 = orb.detectAndCompute(img1, None)
keypoints2, descriptors2 = orb.detectAndCompute(img2, None)

# BFMatcher nesnesi oluştur ve eşleştirmeleri hesapla
bf = cv2.BFMatcher()

descriptors1 = descriptors1.astype(np.float32)
descriptors2 = descriptors2.astype(np.float32)

matches = bf.knnMatch(descriptors1, descriptors2, k=2)

# İyi eşleşmeleri ayıkla (David Lowe's ratio test)
good_matches = []
for m, n in matches:
    if m.distance < .75 * n.distance:
        good_matches.append(m)
```

Şekil 16. ORB, BFMatcher ve Lowe Oran Testi

ORB ve BFMatcher algoritmaları sonucunda bulunan eşleşmelerin çizilmesini sağlayan kod parçası Şekil 17'de ve çıktısı Şekil 1'de gösterilmiştir.

```python
# Eşleşmeleri çizdir
img_matches = cv2.drawMatches(img1, keypoints1, img2, keypoints2, good_matches,
                              None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

print("All matches: ", len(matches))
print("Number of good matchers:", len(good_matches))
# Sonucu görselleştir
plt.figure(figsize=(30, 20))
plt.imshow(img_matches)
plt.show()
```

Şekil 17. ORB eşleşmelerinin çizilmesi

SIFT algoritması ile anahtar noktaların belirlendiği, BFMatcher yöntemi ile bulunan anahtar noktalarının eşleştirildiği ve Lowe Oran Testi ile iyi eşleşmelerin ayıklandığı kod parçası aşağıdaki Şekil 19'da gösterilmektedir. Bu kodda aynı zamanda eşleştirme sonuçları da görseleştirilmiştir ve sonuçlar Şekil 2'de görülmektedir.

```python
# SIFT algosini oluştur
sift = cv2.SIFT_create()

# Keypointleri ve desenleri bul
keypoints1, descriptors1 = sift.detectAndCompute(img1, None)
keypoints2, descriptors2 = sift.detectAndCompute(img2, None)

# Brute-Force Matcher nesnesini oluştur
bf = cv2.BFMatcher()

# Eşleştirmeleri bul
matches = bf.knnMatch(descriptors1, descriptors2, k=2)

img_matches = cv2.drawMatches(img1, keypoints1, img2, keypoints2, good_matches,
                              None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

# İyi eşleşmeleri ayıkla (David Lowe's ratio test)
good_matches = []
for m, n in matches:
    if m.distance < .50 * n.distance:
        good_matches.append(m)

print("All matches: ", len(matches))
print("Number of good matches:", len(good_matches))

# Eşleşmeleri çizdir
img_matches = cv2.drawMatches(img1, keypoints1, img2, keypoints2, good_matches,
                              None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

# Sonucu görselleştir
plt.figure(figsize=(30, 20))
plt.imshow(img_matches)
plt.show()
```

Şekil 18. SIFT, BFMatcher ve Lowe Oran Testi

Algoritmaların toplam ve en iyi eşleştirmelerinin sayısının gözlemlenmesi için yazılan kod parçası ve çıktısı Şekil 21 ve 22'de gösterilmektedir.

```python
# Eşleştirme sayılarını yazdırprint
print("Tüm ORB Eşleştirmeleri Sayısı:", len(matches_orb))
print("İyi ORB Eşleştirmeleri Sayısı:", len(good_orb))
print("Tüm Sift Eşleştirmeleri Sayısı:", len(matches_sift))
print("İyi Sift Eşleştirmeleri Sayısı:", len(good_sift))
```

Şekil 20. Eşleştirme Sayılarının gözlemlenmesi için yazılan kod parçası

```
Tüm ORB Eşleştirmeleri Sayısı: 500
İyi ORB Eşleştirmeleri Sayısı: 44
Tüm Sift Eşleştirmeleri Sayısı: 3086
İyi Sift Eşleştirmeleri Sayısı: 873
```

Şekil 21. Eşleştirme sonuçları

ORB ve SIFT algoritmalarının RANSAC yöntemi ile eşleşen özniteliklerin görselleştirilmesi için kullanılan kod parçaları Şekil 22 ve 23'te, çıktıları Şekil 3 ve 4'te gösterilmektedir.

```python
# Homografiyi hesaplamak için en az 4 eşleşme gereklidir
if len(good) > 4:
    src_pts = np.float32([kp1[m.queryIdx].pt for m in good]).reshape(-1, 1, 2)
    dst_pts = np.float32([kp2[m.trainIdx].pt for m in good]).reshape(-1, 1, 2)

    # RANSAC ile Homografi matrisini hesapla
    H, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)
    matchesMask = mask.ravel().tolist()

    # İlk görüntünün sınırlarını al ve homografi kullanarak dönüştür
    h, w = img1.shape
    pts = np.float32([[0, 0], [0, h - 1], [w - 1, h - 1], [w - 1, 0]]).reshape(-1, 1, 2)
    dst = cv2.perspectiveTransform(pts, H)

    # İkinci görüntü üzerine dönüştürülmüş sınırları çiz
    img2 = cv2.polylines(img2, [np.int32(dst)], True, 255, 3, cv2.LINE_AA)

else:
    print("Yeterli eşleşme bulunamadı - %d/%d" % (len(good), 4))
    matchesMask = None

# Eşleştirmeleri çiz
draw_params = dict(matchColor=(0, 255, 0),  # Yeşil renkte çiz
                   singlePointColor=None,
                   matchesMask=matchesMask,  # Maskeyi kullanarak çiz
                   flags=2)

img3 = cv2.drawMatches(img1, kp1, img2, kp2, good, None, **draw_params)

# Görselleştirme
plt.figure(figsize=(30, 20))
plt.imshow(img3, 'gray')
plt.show()
```

Şekil 22. SIFT için RANSAC yöntemi

```python
# Homografiyi hesaplamak için en az 4 eşleşme gereklidir
if len(good) > 4:
    src_pts = np.float32([kp1[m.queryIdx].pt for m in good]).reshape(-1, 1, 2)
    dst_pts = np.float32([kp2[m.trainIdx].pt for m in good]).reshape(-1, 1, 2)

    # RANSAC ile Homografi matrisini hesapla
    H, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)
    matchesMask = mask.ravel().tolist()

    # İlk görüntünün sınırlarını al ve homografi kullanarak dönüştür
    h, w = img1.shape
    pts = np.float32([[0, 0], [0, h - 1], [w - 1, h - 1], [w - 1, 0]]).reshape(-1, 1, 2)
    dst = cv2.perspectiveTransform(pts, H)

    # İkinci görüntü üzerine dönüştürülmüş sınırları çiz
    img2 = cv2.polylines(img2, [np.int32(dst)], True, 255, 3, cv2.LINE_AA)

else:
    print("Yeterli eşleşme bulunamadı - %d/%d" % (len(good), 4))
    matchesMask = None

# Eşleştirmeleri çiz
draw_params = dict(matchColor=(0, 255, 0),  # Yeşil renkte çiz
                   singlePointColor=None,
                   matchesMask=matchesMask,  # Maskeyi kullanarak çiz
                   flags=2)

img3 = cv2.drawMatches(img1, kp1, img2, kp2, good, None, **draw_params)

# Görselleştirme
plt.figure(figsize=(30, 20))
plt.imshow(img3, 'gray')
plt.show()
```

Şekil 23. ORB için RANSAC yöntemi

Windows dosyalama hatası olmaması adına görsel adlarındaki sayılara göre görselleri sıralayan fonksiyon, Şekil 24'te gösterilmektedir.

```python
# dosya adındaki sayılara göre sıralama yapmak için
def sort_key_func(file_path):
    # dosya adındaki sayıları çıkart ve integer'a çevir
    numbers = re.findall(r'\d+', file_path)
    return int(numbers[0]) if numbers else 0
```

Şekil 24. Görselleri ardışık sıralayan fonksiyon

Ground Truth verilerinin dosya yolunun okunmasını gerçekleştiren ve SIFT ile BFMatcher algoritmalarının tanımlandığı kod parçası Şekil 25'te gösterilmektedir.

```python
# görüntülerin bulunduğu yol"
images_path = r"C:\Users\sevva\OneDrive\Masaüstü\yzup-bitirme\frames\*.jpg"

sorted_images_files = sorted(glob(images_path), key=sort_key_func)

# SIFT kullanımmı
sift = cv2.SIFT_create() # en iyi ilk 500 özellik için nfeatures = 500
bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)

# önceki noktaları tutmak için
prev_keypoints = None
prev_descriptors = None
prev_image = None

trajectory = []  # Konum verilerini saklamak için
```

Şekil 25. Görsellerin dosya yolunun okunması ve gerekli değişkenlerin tanımlanması

Eşleştirilen özellikler ile hareket vektörlerin hesaplanmasını sağlayan kod parçası aşağıdaki Şekil 26 ve 27'de gösterilmektedir.

```python
for idx, image_path in enumerate(sorted_images_files):

    # görüntü yükleme ve işleme
    pil_image = Image.open(image_path)
    open_cv_image = np.array(pil_image)[:, :, ::-1]  # PIL'den OpenCV formatına çevir
    gray_image = cv2.cvtColor(open_cv_image, cv2.COLOR_BGR2GRAY) # openCV BGR formatını kullandığı için bgr to gray

    # özellik tespiti ve eşleştirme
    keypoints, descriptors = sift.detectAndCompute(gray_image, None)

    if prev_descriptors is not None:
        matches = bf.match(prev_descriptors, descriptors)
        matches = sorted(matches, key=lambda x: x.distance)

        matched_image = cv2.drawMatches(prev_image, prev_keypoints,
                                        gray_image, keypoints, matches[:50],
                                        None,
                                        flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

        # Hareket vektörünü hesapla ve yörüngeyi güncelle

        # İki ardışık görüntü arasında bulunan eşleşmelerin listesini döngü içine alıyoruz
        motion_vectors = []  # Her bir eşleşme için hareket vektörlerini tutacak liste

        for match in matches:
            # Eşleşme bilgisinden eğitim görüntüsündeki özellik noktasının konumunu alıyoruz
            current_keypoint = keypoints[match.trainIdx].pt  # Mevcut görüntüdeki özellik noktası
            previous_keypoint = prev_keypoints[match.queryIdx].pt  # Önceki görüntüdeki özellik noktası

            # Numpy dizilerine dönüştürüyoruz ki aritmetik işlem yapabilmek için
            current_position = np.array(current_keypoint)
            previous_position = np.array(previous_keypoint)

            # İki özellik noktası arasındaki farkı hesaplıyoruz, bu bir vektör oluşturur
            displacement_vector = current_position - previous_position

            # Hesaplanan vektörü listeye ekliyoruz
            motion_vectors.append(displacement_vector)
```

Şekil 26. Hareket vektörlerinin hesaplanması - 1

```
    # Tüm hareket vektörlerinin ortalamasını alarak genel bir hareket vektörü hesaplıyoruz
    # Bu, ortalama bir taşınımı (translasyonu) ifade eder
    motion_vector = np.mean(motion_vectors, axis=0)

    if len(trajectory) > 0:
        current_position = trajectory[-1] + motion_vector
    else:
        current_position = motion_vector
    trajectory.append(current_position)

    if idx % 100 == 0:  # Her 100 görüntüde bir görselleştir
        plt.figure(figsize=(12, 8))
        plt.imshow(cv2.cvtColor(matched_image, cv2.COLOR_BGR2RGB))
        plt.title(f"Feature Matching: Image {idx}")
        plt.axis('off')
        plt.show()

# Bu görüntüyü bir sonraki iterasyon için sakla
prev_keypoints = keypoints
prev_descriptors = descriptors
prev_image = gray_image
```

Şekil 27. Hareket vektörlerinin hesaplanması - 2

Belirlenen hareket vektörlerinin "trajectory_01.csv" dosyasına yazılması işlemini gerçekleştiren kod Şekil 28'de gösterilmiştir.

```
# Yörünge verilerini CSV dosyasına kaydet

# w -> yazma modu, newline='' -> satır sonu gereksiz bosluk yok
with open('trajectory_01.csv', 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(['x_position', 'y_position'])
    writer.writerows(trajectory)
```

Şekil 28. Yörünge verilerinin CSV dosyasına kaydedilmesi

CSV dosyası yoluyla görsel odometri sonucu oluşturulan yörüngenin görselleştirilmesi için kullanılan kod, Şekil 29'da ve kodun çıktısı Şekil 8'de gösterilmiştir.

```
# CSV dosyasını yükle
data_vo = pd.read_csv(r'C:\Users\sevva\OneDrive\Masaüstü\yzup-bitirme\trajectory_01.csv')

plt.figure(figsize=(10, 6))
plt.plot(-data_vo['x_position'], -data_vo['y_position'],
         marker='o', linestyle='-', color='lightseagreen', label='Trajectory')
plt.title('Computed Trajectory')
plt.xlabel('X Position')
plt.ylabel('Y Position')
plt.grid(True)
plt.show()
```

Şekil 29. Görsel odometri yörüngesinin oluşturulduğu kod

Hassas sensör verilerini içeren CSV dosyası ile oluşturulan yörüngenin görselleştirilmesi için kullanılan kod parçası Şekil 30'da ve kod sonucu oluşan yörünge Şekil 7'de gösterilmiştir.

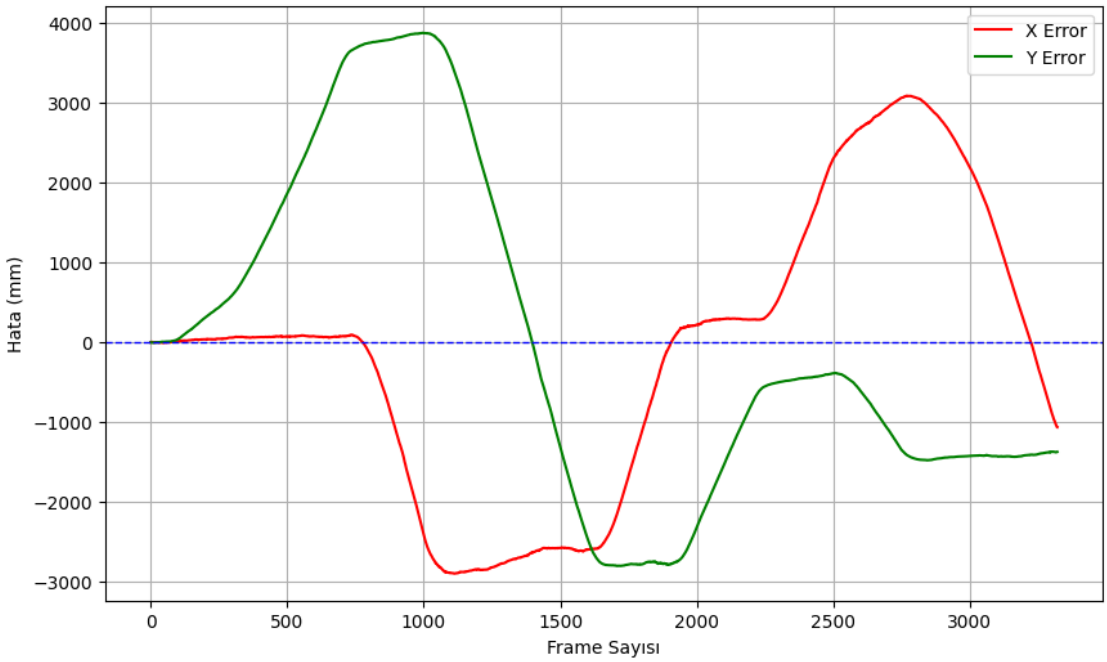
```python
# CSV dosyasını yükle
data_gt = pd.read_csv(r'C:\Users\sevva\OneDrive\Masaüstü\yzup-bitirme\GT_Translations.csv')

# x ve y pozisyonlarını çiz
plt.figure(figsize=(10, 6))
plt.plot(data_gt['translation_x'], data_gt['translation_y'],
         marker='o', linestyle='-', color='dodgerblue', label='Trajectory')
plt.title('Ground Truth Trajectory')
plt.xlabel('X Position')
plt.ylabel('Y Position')
plt.legend()
plt.grid(True)
plt.show()
```

Şekil 30. Ground Truth verilerinin görselleştirilmesini sağlayan kod parçası

Herhangi bir hizalama işlemi yapılmadan önce görsel odometri verileri ve ground truth verilerinden elde edilen yörüngenin karşılaştırılmasıyla elde edilen hata değerleri grafiği hesaplanmış ve görselleştirilmiştir. Kod parçası ve grafik Şekil 31 ve 32'de gösterilmektedir.

```python
# VO ve GT verilerini yükleyin
data_vo = pd.read_csv('trajectory.csv')
data_gt = pd.read_csv('GT_Translations.csv')

# Frame number ve pozisyon sütunlarını alın
x_vo = -data_vo['x_position']
y_vo = -data_vo['y_position']

# Ground truth verilerini yükleyin
x_gt = data_gt['translation_x'][1:]
y_gt = data_gt['translation_y'][1:]

# Uzunlukları kontrol edin
print(f"x_vo: {len(x_vo)}")
print(f"y_vo: {len(y_vo)}")
print(f"x_gt: {len(x_gt)}")
print(f"y_gt: {len(y_gt)}")

# Hataları hesaplayın
error_x = x_gt.values - x_vo.values
error_y = y_gt.values - y_vo.values

# Hata boyutlarını kontrol edin
print(f"error_x: {len(error_x)}")
print(f"error_y: {len(error_y)}")

# Hata grafiği çizimi
plt.figure(figsize=(10, 6))
# X ekseni hatalarını çizin
plt.plot(frame, error_x, label='X Error', color='red')
# Y ekseni hatalarını çizin
plt.plot(frame, error_y, label='Y Error', color='green')
# Y=0 referans çizgisi çiz
plt.axhline(0, color='blue', linewidth=1, linestyle='--')

# Grafiği düzenleyin
plt.xlabel('Frame Say□s□')
plt.ylabel('Hata (mm)')
plt.title('Referans ve Kestirim Pozisyon Bilgisi Kullan□larak Hesaplanan Hata Değerleri')
plt.legend()
plt.grid(True)

# Grafiği gösterin
plt.show()
```

Şekil 31. İlk hata grafiğinin oluşturulduğu kod parçası



Şekil 32. Hesaplanan ilk hata grafiği

ICP fonksiyonu Şekil 33'te görülmektedir. ICP fonksiyonu içerisinde kullanılan ve en iyi uyumu yakalamaya çalışan fonksiyon Şekil 3'te, ICP fonksiyonun uygulanması ise Şekil 35'te gösterilmektedir. Uygulama sonucu hizalanmış yörüngeler Şekil 11'de gösterilmiştir.

```python
def icp(A, B, max_iterations=20, tolerance=1e-5):
    src = np.ones((A.shape[0], A.shape[1] + 1))
    dst = np.ones((B.shape[0], B.shape[1] + 1))
    src[:, :-1] = A
    dst[:, :-1] = B

    prev_error = 0

    for i in range(max_iterations):
        # En yakın komşuları bul
        nbrs = NearestNeighbors(n_neighbors=1, algorithm='auto').fit(dst[:, :-1])
        distances, indices = nbrs.kneighbors(src[:, :-1])

        # Dönüşüm matrisini hesapla
        T = best_fit_transform(src[:, :-1], dst[indices[:, 0], :-1])

        # Kaynak noktalarını dönüştür
        src = np.dot(T, src.T).T

        # Hata hesapla
        mean_error = np.mean(distances)
        if np.abs(prev_error - mean_error) < tolerance:
            break
        prev_error = mean_error

    T = best_fit_transform(A, src[:, :-1])
    return T, distances
```

Şekil 33. ICP fonksiyonu

```python
# ICP hizalama fonksiyonu
def best_fit_transform(A, B):
    assert A.shape == B.shape

    # Orta noktayı hesapla
    centroid_A = np.mean(A, axis=0)
    centroid_B = np.mean(B, axis=0)

    # Merkezden uzaklaşma
    AA = A - centroid_A
    BB = B - centroid_B

    # Kovaryans matrisi hesapla
    H = np.dot(AA.T, BB)

    # SVD kullanarak döndürme matrisi hesapla
    U, S, Vt = np.linalg.svd(H)
    R = np.dot(Vt.T, U.T)

    # Yansıma olmasını önlemek için son SVD'den sonra işareti düzelt
    if np.linalg.det(R) < 0:
        Vt[-1, :] *= -1
        R = np.dot(Vt.T, U.T)

    # Kaydırma vektörü hesapla
    t = centroid_B.T - np.dot(R, centroid_A.T)

    # Transform matrisi oluştur
    T = np.identity(A.shape[1] + 1)
    T[:A.shape[1], :A.shape[1]] = R
    T[:A.shape[1], A.shape[1]] = t

    return T
```

Şekil 3419. En iyi uyum fonksiyonu uygulanması

```python
# VO ve GT verilerini yükleyin
data_vo = pd.read_csv('trajectory_01.csv')
data_gt = pd.read_csv('GT_Translations.csv')

# Verileri numpy array olarak dönüştür
vo_coordinates = np.column_stack((-data_vo['x_position'],
                                  -data_vo['y_position']))
gt_coordinates = np.column_stack((data_gt['translation_x'][1:],
                                  data_gt['translation_y'][1:]))

# Ölçek faktörünü hesaplayın
vo_length = np.sum(np.sqrt(np.sum(np.diff(vo_coordinates, axis=0)**2, axis=1)))
gt_length = np.sum(np.sqrt(np.sum(np.diff(gt_coordinates, axis=0)**2, axis=1)))
scale_factor = gt_length / vo_length

# VO verilerini ölçeklendirin
vo_coordinates *= scale_factor

# ICP hizalamasını gerçekleştirin
T, distances = icp(vo_coordinates, gt_coordinates)

# Dönüştürülmüş VO koordinatlarını hesaplayın
aligned_vo_coordinates = np.dot(T[:2, :2], vo_coordinates.T).T + T[:2, 2]

# Hizalanmış VO verilerini çiz
plt.figure(figsize=(10, 6))
plt.plot(gt_coordinates[:, 0], gt_coordinates[:, 1],
         marker='o', linestyle='-', color='dodgerblue',
         label='Ground Truth Trajectory')
plt.plot(aligned_vo_coordinates[:, 0], aligned_vo_coordinates[:, 1],
         marker='o', linestyle='-', color='mediumpurple',
         label='Aligned VO Trajectory')
plt.title('Aligned Trajectories using ICP')
plt.xlabel('X Position')
plt.ylabel('Y Position')
plt.legend()
plt.grid(True)
plt.show()
```

Şekil 35.ICP fonksiyonunun uygulanması

X ve Y eksenleri özelinde hesaplanan hata değerlerinin hesaplanmasına ait kod parçası Şekil 36'da gösterilemektedir. Kod çıktısı Tablo 2'de, oluşturulan grafik ise Şekil 39'da gösterilmektedir.

```python
# Hataları X ve Y için ayrı ayrı hesapla
error_x = gt_coordinates[:, 0] - aligned_vo_coordinates[:, 0]
error_y = gt_coordinates[:, 1] - aligned_vo_coordinates[:, 1]

# X ve Y hataları için istatistiksel değerleri hesapla ve yazdır
for coord, errors in zip(['X', 'Y'], [error_x, error_y]):
    mse = np.mean(errors**2)
    rmse = np.sqrt(mse)
    mean_error = np.mean(errors)
    std_error = np.std(errors)
    max_error = np.max(errors)
    min_error = np.min(errors)

    print(f"{coord} Hatası - Ortalama: {mean_error:.4f}, Standart Sapma: {std_error:.4f},
          Maksimum: {max_error:.4f}, Minimum: {min_error:.4f}")
    print(f"{coord} Hatası - MSE: {mse:.4f}, RMSE: {rmse:.4f}\n")

# Hata değerlerini çiz
plt.figure(figsize=(12, 6))
frame = np.arange(len(error_x))  # Frame sayısını oluştur
plt.plot(frame, error_x, label='X Error', color='red', linestyle='-', markersize=5)
plt.plot(frame, error_y, label='Y Error', color='green', linestyle='-', markersize=5)
plt.axhline(0, color='blue', linestyle='--', label='Zero Error Line', linewidth=1)  # Sıfır hatası çizgisi

# Y-ekseni sınırlarını ayarla
plt.xlim(left=0)  # X-ekseni sınırlarını ayarla

plt.title('Referans ve Kestirim Pozisyon Bilgisi Kullanılarak Hesaplanan Hata Değerleri')
plt.xlabel('Frame Sayısı')
plt.ylabel('Hata (mm)')
plt.legend()
plt.grid(True)
plt.show()
```

Şekil 36. Hata değerlerinin hesaplanması

Ölçekleme işlemi için gerçekleştirilen normalizasyon işlemi Şekil 37'de bulunmaktadır.

```python
# Normalizasyon işlemi
def normalize(data):
    return (data - np.min(data)) / (np.max(data) - np.min(data))
```

Şekil 37. Normalizasyon işlemi

Normalizasyon işleminden sonra gerçekleştirilen hata değerlerinin hesaplanması işlemi Şekil 36'da olduğu gibi gerçekleştirilmiş yalnızca gerekli değişkenlerin adları değiştirilmiştir.

Procrustes Analizi ile gerçekleştirilen hizalama işlemi için kullanılmış kodlar Şekil 38'de gösterilmektedir.

```python
# procrustes
# VO ve GT verilerini yükle
data_vo = pd.read_csv(r'C:\Users\sevva\OneDrive\Masaüstü\yzup-bitirme\trajectory_01.csv')
data_gt = pd.read_csv(r'C:\Users\sevva\OneDrive\Masaüstü\yzup-bitirme\GT_Translations.csv')

# Verileri numpy array olarak dönüştür
vo_coordinates = np.column_stack((data_vo['x_position'], data_vo['y_position']))  # Negatif işaret kullanımı
gt_coordinates = np.column_stack((data_gt['translation_x'][1:], data_gt['translation_y'][1:]))

# Procrustes analizi ile hizalama (scipy'daki Procrustes metodu ölçek değişikliği yapmaz)
mtx1, mtx2, disparity = procrustes(gt_coordinates, vo_coordinates)

# Hizalanmış VO verilerini çiz
plt.figure(figsize=(10, 6))
plt.plot(mtx1[:, 0], mtx1[:, 1], marker='o', linestyle='-', color='dodgerblue', label='Ground Truth Trajectory')
plt.plot(mtx2[:, 0], mtx2[:, 1], marker='o', linestyle='-', color='orangered', label='Aligned VO Trajectory')
plt.title('Aligned Trajectories')
plt.xlabel('X Position')
plt.ylabel('Y Position')
plt.legend()
plt.grid(True)
plt.show()
```

Şekil 38. Procrustes Analizi

Normalizasyon uygulanmış verilerle gerçekleştirilen Procrustes Analizi ve hata değerlerinin hesaplanarak görselleştirme işlemi, daha önce bahsedilmiş olan kodların gerekli değişken adları değiştirilerek tekrar kullanılması sonucu elde edilmektedir.

7. SONUÇLAR

Görsel odometri algoritmasının doğruluğunu değerlendirmek için X ve Y eksenlerindeki hata değerleri hesaplanmış ve hatalar grafiklerle görselleştirilmiştir. Hata dedğerleri, görsel odometri verileri ve ground truth verileri ile çizilmiş olan yörüngeler karşılaştırılarak hesaplanmıştır. Hata analizleri, görsel odometri algoritmasının performansını değerlendirmek amacıyla yapılmıştır.

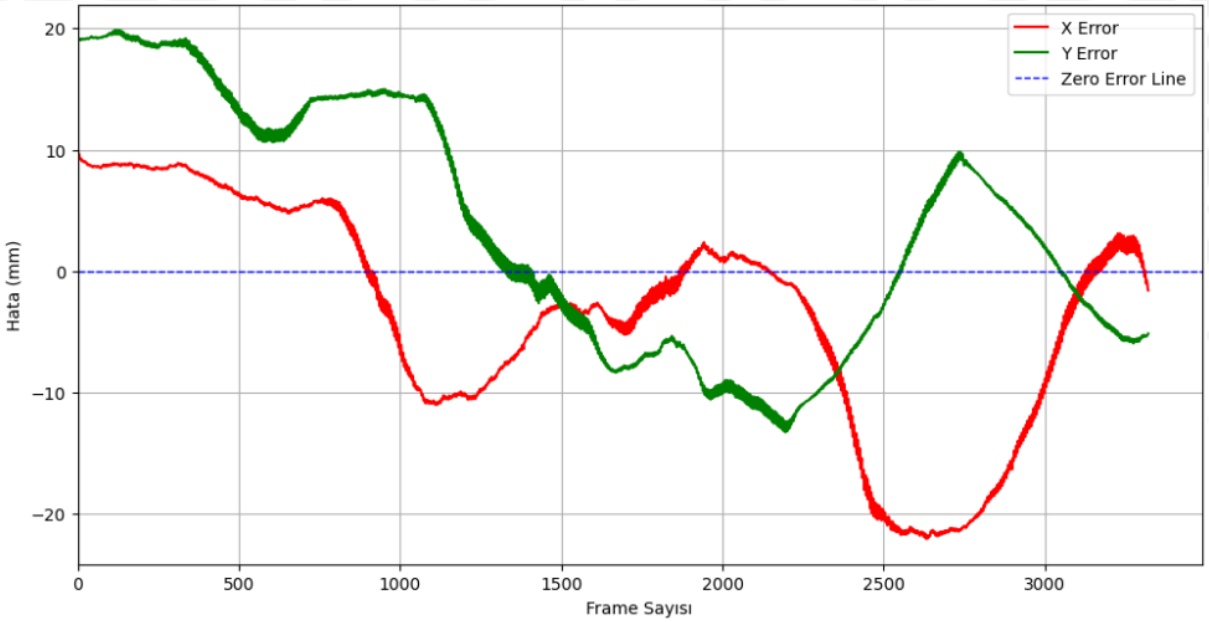
Hizalanmış yörüngeler birbirlerine çok yakın göründüğünden dolayı doğru karar verebilmek adına hata değerleri sayısal olarak hesaplanmıştır.

ICP hizalama işlemi sonucu görsel odometri yörüngesi ve ground truth verileri ile elde edilen yörüngenin karşılaştırılması sonucunda elde edilen hata değerleri Tablo 2'de gösterilmektedir.

	Ortalama	Standart Sapma	Maksimum	Minimum	MSE	RMSE
X Ekseni	-3.4871	9.0477	9.6895	-22.0582	94.0210	9.6964
Y Ekseni	3.5281	10.0951	19.8606	-13.2804	114.3588	10.6939

Tablo 2. ICP hizalama yöntemi hata değerleri

Hata değerlerinin görselleştirilmesi sonucu oluşan grafik Şekil 39'da gösterilmektedir.

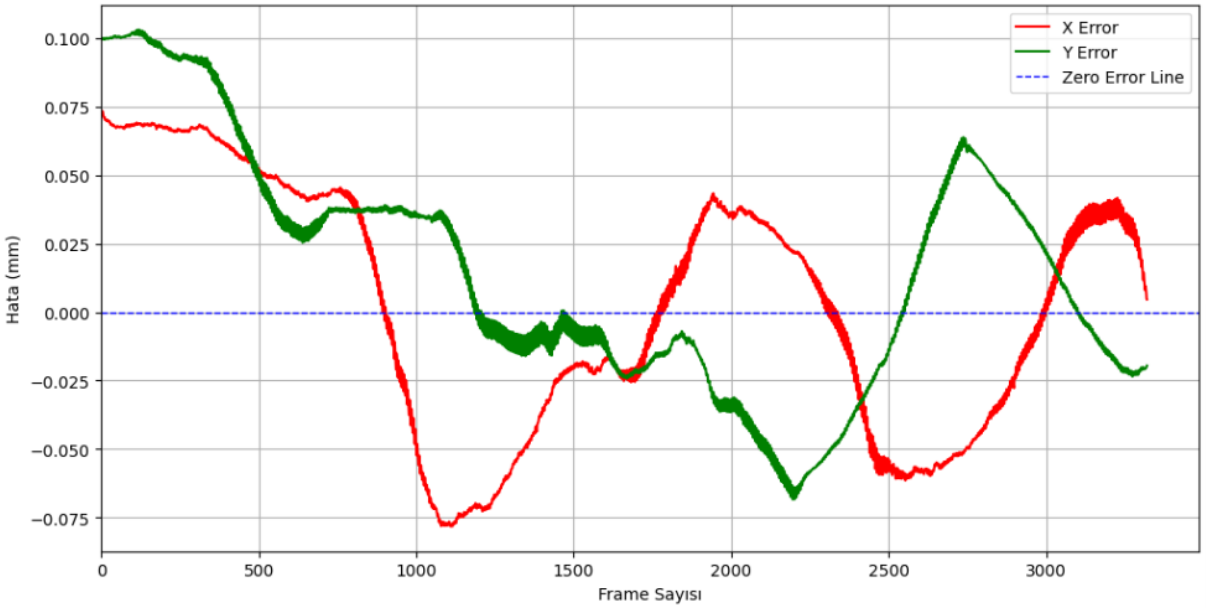


Şekil 39. ICP hizalama sonucu hata grafiği

Normalizasyon uygulanmış görsel odometri verileri ile gerçekleştirilen ICP hizalamasının hata değerleri, Tablo 3'te bulunmaktadır. Hata değerlerinin görselleştirilmiş hali ise Şekil 40'ta gösterilmektedir.

	Ortalama	Standart Sapma	Maksimum	Minimum	MSE	RMSE
X Ekseni	0.0026	0.0448	0.0733	-0.0782	0.0020	0.0449
Y Ekseni	0.0148	0.0436	0.1030	-0.0683	0.0021	0.0460

Tablo 3. Normalizasyon ve ICP hizalama yöntemi hata değerleri



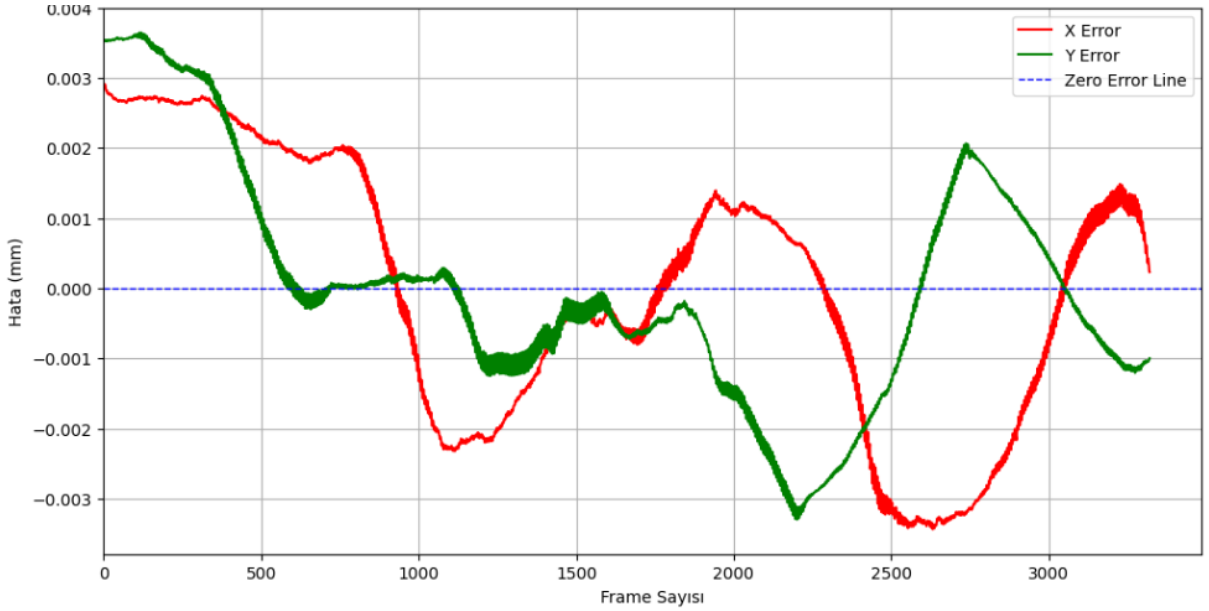
Şekil 40. Normalizasyon ve ICP hizalama yöntemi hata grafiği

Normalizasyon öncesi ve sonrası değerler karşılaştırıldığında, normalizasyon işleminden sonra hata değerlerinin azaldığı gözlemlenmektedir.

Procrustes analizi ile hizalama işlemi sonucu görsel odometri yörüngesi ve ground truth verileri ile elde edilen yörüngenin karşılaştırılması sonucu oluşan hata değerleri Tablo 4'te bulunmaktadadır. Hata değerlerinin görselleştirilmesi ile oluşan grafik Şekil 41'de gösterilmektedir.

	Ortalama	Standart Sapma	Maksimum	Minimum	MSE	RMSE
X Ekseni	0.0000	0.0019	0.0029	-0.0034	0.0000	0.0019
Y Ekseni	-0.0000	0.0016	0.0036	-0.0033	0.0000	0.0016

Tablo 4. Procrustes analizi hizalama yöntemi hata değerleri



Şekil 41. Procrustes analizi sonucu hata değerleri grafiği

Normalizasyon uygulanmış görsel odometri verileri ile gerçekleştirilen Procrustes Analizi ile hizalanmasının hata değerleri Tablo 5'te bulunmaktadır.

	Ortalama	Standart Sapma	Maksimum	Minimum	MSE	RMSE
X Ekseni	0.0000	0.0019	0.0029	-0.0034	0.0000	0.0019
Y Ekseni	-0.0000	0.0016	0.0036	-0.0033	0.0000	0.0016

Tablo 6. Normalizasyon ve Procrustes Analizi sonucu hata değerleri

Tablo 5 ve 6 karşılaştırıldığında normalizasyon ardından hata değerlerinde herhangi bir değişiklik gözlemlenmemektedir.

ICP ve Procrustes Analizi sonucunda elde edilen hata değerleri göz önüne alındığında, Procrustes Analizinin sonuçları milimetrik (mm) olarak çok daha az hatalı gelmektedir. Bu da Procrustes Analizi kullanılması sonucunda, ICP'ye göre, görsel odometrinin daha başarılı bir sonuç verdiği gözlemlenmektedir.

8. KAYNAKÇA

[1] OpenCV Team. (n.d.). About. OpenCV. Retrieved May 28, 2024, from https://opencv.org/about/

[2] Pillow Developers. (n.d.). Pillow: Python Imaging Library (PIL Fork). PyPI. Retrieved May 28, 2024, from https://pypi.org/project/pillow/

[3] Bansal, M., Kumar, M., & Kumar, M. (2021). 2D object recognition: a comparative analysis of SIFT, SURF and ORB feature descriptors. Multimedia Tools and Applications, 80(12), 18839-18857.

[4] OpenCV Team. (n.d.). Feature Matching. OpenCV Documentation. Retrieved May 28, 2024, from https://docs.opencv.org/3.4/dc/dc3/tutorial_py_matcher.html

[5] Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. International journal of computer vision, 60, 91-110.

[6] Wu, M., Lam, S. K., & Srikanthan, T. (2017). A framework for fast and robust visual odometry. IEEE Transactions on Intelligent Transportation Systems, 18(12), 3433-3448.

[7] Işik, Ş. (2014) Öznitelik tabanlı imge çakıştırma (Master's thesis, Fen Bilimleri Enstitüsü)

[8] Hunter, J. D., Droettboom, M., Caswell, T. A., Firing, E., Lee, J. J., et al. (n.d.). Matplotlib: Visualization with Python. Retrieved May 28, 2024, from https://matplotlib.org/

[9] Besl, P. J., & McKay, N. D. (1992, April). Method for registration of 3-D shapes. In Sensor fusion IV: control paradigms and data structures (Vol. 1611, pp. 586-606). Spie.

[10] Ross, A. (2004). Procrustes analysis. Course report, Department of Computer Science and Engineering, University of South Carolina, 26, 1-8.