Group Members: Ahmet Can Karaaslan, Can Güray Erkan, Larissa Malgaz, Melis Kekeç, Serap Tuba Kudaloğlu

Advisors: Prof. Dr. Çağatay Başdoğan, Aybars Ağaya, Emir Bahadır Ünsal

Project: Haptic Vest for Indoor Navigation

Date: 19 July 2024

Koç University Summer Research Program

Haptic Vest for Indoor Navigation - Final Report

Our Team

INTRODUCTION

This project explores the development of a haptic vest designed to aid indoor navigation using tactile feedback. The vest translates spatial information from the surroundings into touch-based signals, helping the wearer to navigate without sight. By using sensors and actuators, the vest provides guidance to avoid obstacles, move through doorways, and reach specific destinations within indoor spaces.

IMAGE PROCESSING & ARTIFICIAL INTELLIGENCE

Image processing is a crucial step to detect what is happening around a visually impaired individual. There have been multiple parameters in image processing, such as detecting what object, the count of those objects, the objects excluded from object detection, the side of the camera the objects are, and several more. YOLOv8 library is used to implement real-time object detection through a web camera. The YOLOv8 model is eminent for its efficiency in detecting a wide range of objects. Initially, we integrated the YOLOv8 model with our webcam feed to detect multiple object classes provided by the library. A different frame with a different color was generated for each distinct object. The floating numbers represent confidence values. For instance, a person is detected with 0.96 confidence in a dark blue frame while a TV is detected with 0.49 confidence demonstrated in a white frame. As *Image I* demonstrates, The output is visible on the terminal with the number of objects first, followed by the defined name of those objects: “[...] 4 persons, 1 bottle, 4 chairs, 3 tvs, 3 laptops, 2 keyboards, 462.8 ms” (*Image I*). There

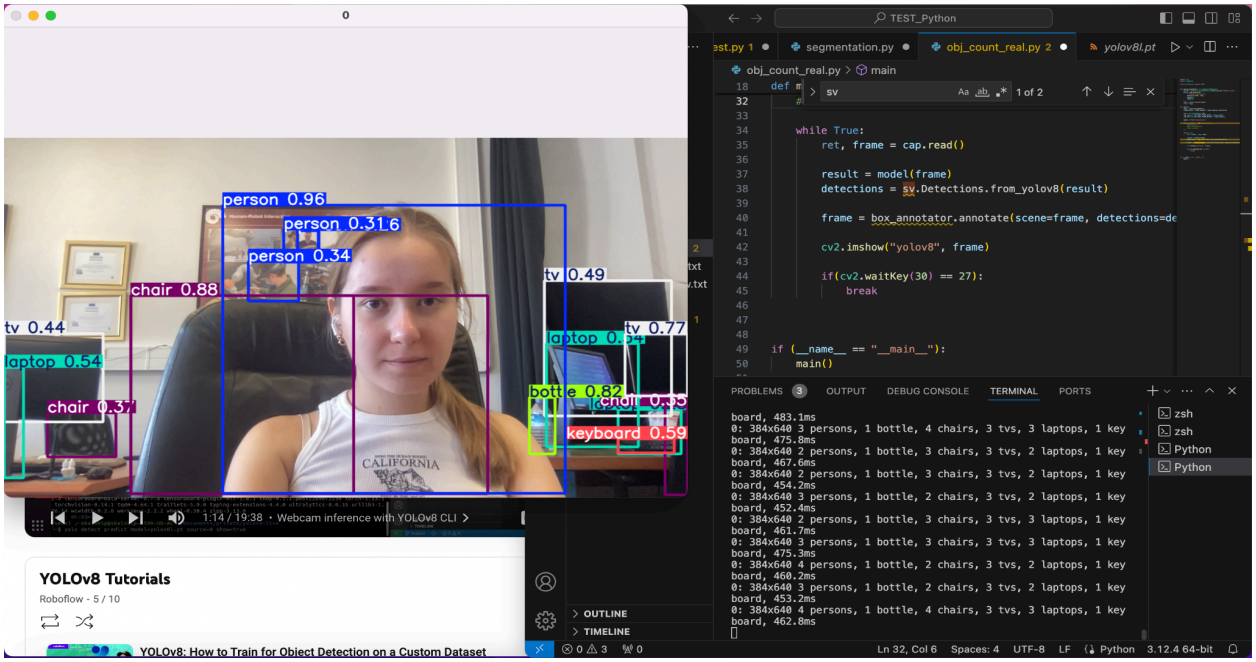


Image I: YOLOv8 on Live Camera

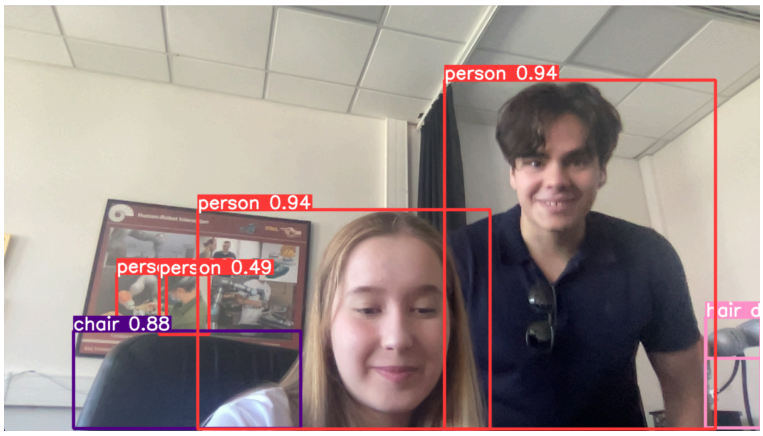


Image II: YOLOv8 on Live Camera Demonstrating Two People

We examined the difference between YOLOv8 and YOLOv5, as well as the functional differences between models of YOLOv8. The overall goal was to decide on the optimum model based on the performance, time, and power of the computers. *Chart I* compares the performance of YOLOv8 with YOLOv5 across different model sizes (Nano, Small, Medium, Large, Xtra Large) in terms of detection, segmentation, and classification. YOLOv8 shows significant improvements over YOLOv5, especially in the smaller model sizes (Nano and Small), with detection performance improvements up to +33.21% for the Nano model. *Graph I* evaluates the performance of various YOLO versions (v5, v6, v7, and v8) on a dataset. The left graph plots model accuracy against the number of parameters. The right graph plots model accuracy against the latency of the model. YOLOv8 consistently outperforms the previous versions, achieving

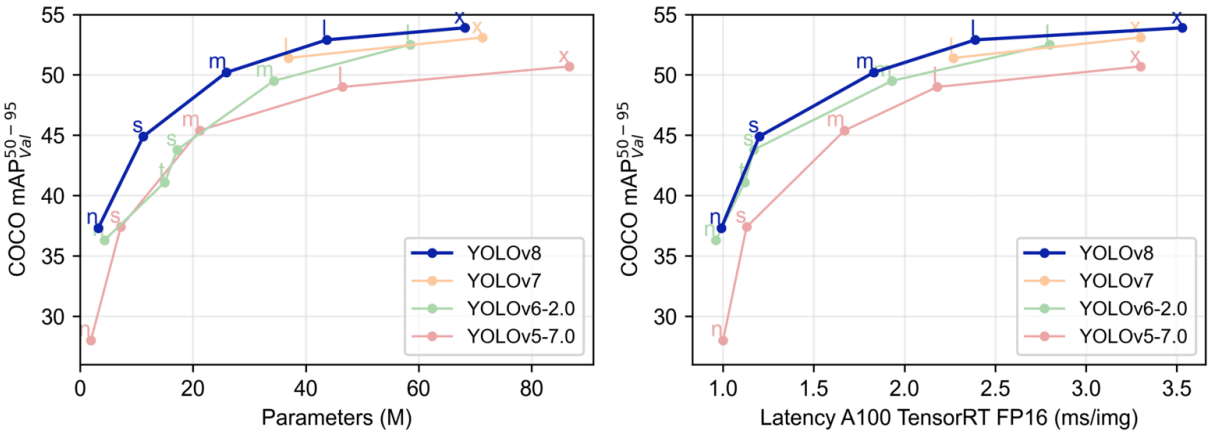
higher accuracy at comparable or lower latencies, indicating both higher precision and efficiency. Considering these differences, the image-processing leg of this project was conducted using YOLOv8, with small and nano models. *Image I* is run with the *small* model, while *Image II* is run with the *nano* model. *Image I* visualizes a more accurate model, in which a wider range of objects were detected, while *Image II* gives a less complex result.

Performance Comparison of YOLOv8 vs YOLOv5

Model Size	Detection#	Segmentation#	Classification*
Nano	+33.21%	+32.97%	+3.10%
Small	+20.05%	+18.62%	+1.12%
Medium	+10.57%	+10.89%	+0.66%
Large	+7.96%	+6.73%	0.00%
Xtra Large	+6.31%	+5.33%	-0.76%

#Image Size = 640 *Image Size = 224

Chart I: Performance Comparison of YOLOv8 vs. YOLOv5



Graph I: Performance of YOLO Versions

We then modified the detection settings to exclude the "person" class, ensuring our focus remained on other objects within the frame. Subsequently, we defined two distinct zones on the screen: one on the left and the other on the right, visually distinguishable by red and green

overlays, respectively. Our implementation included algorithms to count and display the number of detected objects within each zone. This setup allowed us to dynamically track the distribution of objects in these specified areas, providing clear and real-time feedback on their presence and quantity. The overall system effectively demonstrated the practical application of YOLOv8 in monitoring and analyzing spatial object distribution.

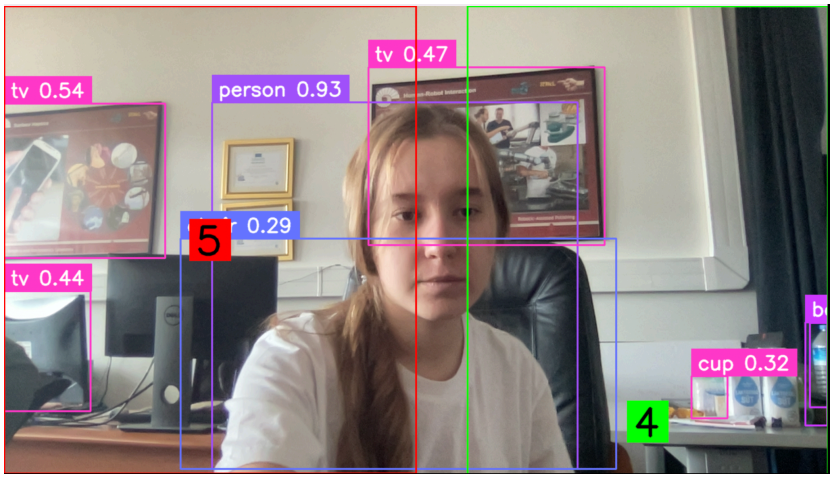


Image III: Left and Right Zone on Camera

```python
ZONE_POLYGON_LEFT = np.array([
    [0, 0],
    [1280 // 2, 0],
    [1280 // 2, 720],
    [0, 720]
])

ZONE_POLYGON_RIGHT = np.array([
    [720, 0],
    [1280 , 0],
    [1280 , 1280],
    [720, 1280]
])


def parse_arguments() -> argparse.Namespace:
    parser = argparse.ArgumentParser(description="YOLOv8 live")
    parser.add_argument(
        "--webcam-resolution",
        default=[1280, 720],
        nargs=2,
        type=int
    )
    args = parser.parse_args()
    return args
```

Image IV: Defining Zones with Python YOLOv8

```python
def main():
    print("Code is started")
    args = parse_arguments()
    frame_width, frame_height = args.webcam_resolution

    cap = cv2.VideoCapture(0)
    cap.set(cv2.CAP_PROP_FRAME_WIDTH, frame_width)
    cap.set(cv2.CAP_PROP_FRAME_HEIGHT, frame_height)

    model = YOLO("yolov8n.pt")
    print("Model is loaded")

    bounding_box_annotator = sv.BoundingBoxAnnotator(thickness=2)
    label_annotator= sv.LabelAnnotator(text_thickness=2, text_scale=1)

    zone_left = sv.PolygonZone(polygon= ZONE_POLYGON_LEFT, frame_resolution_wh=tuple(args.webcam_resolution))
    zone_left_annotator = sv.PolygonZoneAnnotator(zone=zone_left, color=sv.Color.RED, thickness=2, text_thickness=4, text_scale=2)

    zone_right = sv.PolygonZone(polygon= ZONE_POLYGON_RIGHT, frame_resolution_wh=tuple(args.webcam_resolution))
    zone_right_annotator = sv.PolygonZoneAnnotator(zone=zone_right, color=sv.Color.GREEN, thickness=2, text_thickness=4, text_scale=2)
```

Image V: Creating Zones with Python YOLOv8

As demonstrated in *Image VI, Image VII,* and *Image VIII*, the code to find the path defines a "PathNavigator" class to assist navigation by providing vocal directions based on a specified path. Utilizing the Google Text-to-Speech (gTTS) library, the class converts text instructions into audible commands. The "PathNavigator" initializes with a graph and an orientation, and contains methods to parse the path, determine directional changes, and play corresponding voice commands. The _add_direction method issues the correct direction commands based on the current orientation and movement. The run method monitors for path updates and provides directions until the user exits by pressing 'q'. This approach combines image processing, pathfinding, and text-to-speech to aid indoor navigation with real-time auditory instructions.

```python
def determine_directions(self, pairs):
    for i in range(len(pairs) - 1):
        first, second = pairs[i], pairs[i + 1]


        row1, col1 = first
        row2, col2 = second

        print(row1 , row2, col1, col2)

        if col2 < col1:
            self._add_direction("left", "turn backwards and go forward", "go forward", "turn left and go forward", "turn right and go forward")
            self.orientation = "left"

        elif col2 > col1:
            self._add_direction("right", "go forward", "turn backwards and go forward", "turn right and go forward", "turn left and go forward")
            self.orientation = "right"
        if row2 > row1:
            self._add_direction("down", "turn right and go forward", "turn left and go forward", "turn backwards and go forward", "go forward")
            self.orientation = "down"
        elif row2 < row1:
            self._add_direction("up", "turn left and go forward", "turn right and go forward", "go forward", "go backwards and go forward")
            self.orientation = "up"
```

Image VI: The Function to Determine Direction

```
def _add_direction(self, new_orientation, right_cmd, left_cmd, up_cmd, down_cmd):
    if self.orientation == "left":
        # self.directions += left_cmd + "; "
        self.play_voice(left_cmd)
        print("working")
        time.sleep(3)
    elif self.orientation == "right":
        # self.directions += right_cmd + "; "
        self.play_voice(right_cmd)
        print("working")
        time.sleep(3)
    elif self.orientation == "up":
        # self.directions += up_cmd + "; "
        self.play_voice(up_cmd)
        print("working")
        time.sleep(3)
    elif self.orientation == "down":
        # self.directions += down_cmd + "; "
        self.play_voice(down_cmd)
        print("working")
        time.sleep(3)

    print(self.directions)
```

```
def run(self):
    print("run started")

    old = None


    graphpath = self.graph.pathID
    print(graphpath)

    while True:
        if self.graph.pathID is not None and self.graph.pathID != old:
            old = self.graph.pathID

            pairs = self.parse_pairs(self.graph.pathID)
            self.determine_directions(pairs)

            print("yes")

        if keyboard.is_pressed('q'):
            print("Exiting...")
            break
```

Image VII & Image VIII: The Functions that Orient the Directions and Run the Algorithm

AUDIO

Processing

The processing of audio was a vital part of the project. We tried to make it so that the interactions between the user and the computer were mainly done through an auditory medium. Throughout this process, transcribing the audio was the biggest challenge. However, we overcame this issue by utilizing certain libraries and APIs.

The SpeechRecognition library was a very useful library. Considering the "[s]peech recognition engine/API support," SpeechRecognition offered us a variety of choices ("SpeechRecognition 3.10.4.") Out of the many choices presented, we decided to utilize Google Speech Recognition API for a multitude of reasons. Firstly, it required no key, unlike many other APIs for audio transcription. Secondly, it was able to transcribe audio in many different languages, including Turkish, which was crucial. Lastly, the transcription was very accurate. A segment of code utilizing the Google Speech Recognition API is presented below (this segment was not included in the final version of the project, instead it demonstrated that the process

worked.)

```
def recognize_and_respond(self):
    recognizer = sr.Recognizer()
    with sr.AudioFile("messageOfUser.wav") as source:
        audio = recognizer.record(source)
    recognized_text =  recognizer.recognize_google(audio,language="tr-TR")
    with open("recognized_text.txt", "w") as file:
        file.write(recognized_text)
```

As it can be seen above, the transcription process was extremely efficient with the usage of the Google Speech Recognition API.

However, at some point during the project, we did explore other APIs for the purpose of transcribing audio. At this point, we discovered AssemblyAI and their transcription tool. This tool had a few advantages over Google Speech Recognition: firstly, the transcribed text was punctuated correctly, whereas Google's API churned out a large list of words with no commas. Secondly, AssemblyAI's tool could be used to complete other tasks, such as analyzing the sentiment (positive, neutral, negative) behind a certain auditory message. A segment of code utilizing AssemblyAI's transcription tool is presented below (Again, this segment was not featured in the final project).

```
def theBetterTranscriber(nameoftheThing):
  FILE_URL = "/Users/ahmetcankaraaslan/Desktop/NewAudioStuff/" + nameoftheThing
  print("Below, you can find the normal transcription:")
  transcriber = aai.Transcriber()
  transcript = transcriber.transcribe(FILE_URL)
```

In the end, despite the advantages listed above, we decided to utilize Google Speech Recognition in our final version of the project as the advantages did not turn out to be particularly useful for our needs.

In the final version of the project, the transcribed audio was saved as a text file, which would be sent to ChatGPT servers, who would produce a result.

Text-to-speech

Text-to-speech was, when compared to the transcription process, remarkably easy. After utilizing the read() function in order to attach the text found in a file to a string variable, Google's Google Translate's text-to-speech API and the gTTS library were used to turn the string into sound. The code presented below shows the entire process:

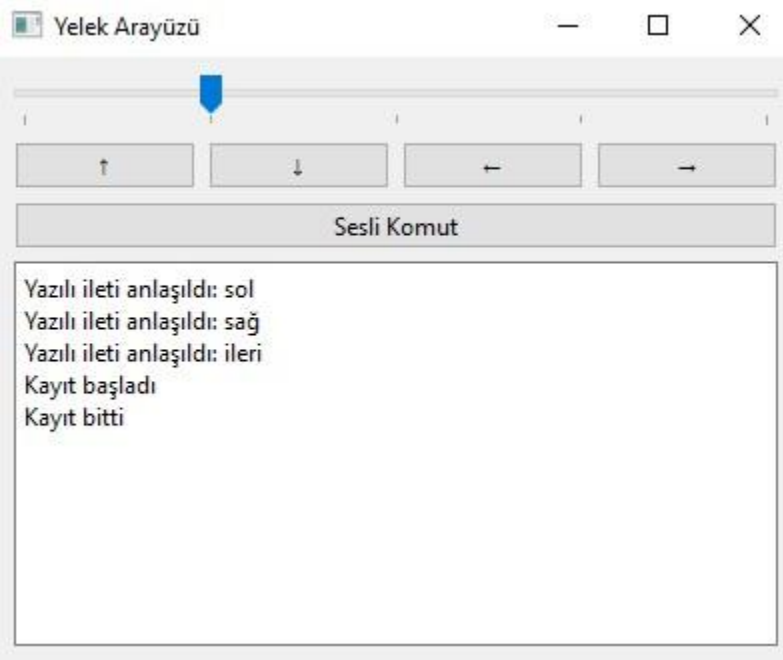
```
with open("gptResponse.txt", "r") as file:
    response = file.read()
    language = 'tr'
    myobj = gTTS(text=response, lang=language, slow=False)
    myobj.save("gptVoiceMessage.mp3")
    print(myobj.type())
    print("HELLO!")
    os.system("afplay gptVoiceMessage.mp3")
```

USER INTERFACE

We designed a user interface to allow users to start the program, ask questions to the AI, initiate voice commands, and control the vest's vibrations using both voice commands and on-screen arrows. When the program starts, a page appears on the user's device featuring four arrows for navigation: right, left, forward, and backward. Below these arrows, there is a button to start and stop voice recording for voice commands. Another button below this allows users to ask questions to the AI. At the bottom, a feedback box displays the user's questions and the arrows they clicked. This UI helps users easily interact with the AI and control the vest, enhancing their overall experience and efficiency. It streamlines the process of issuing commands and receiving responses, making it intuitive and user-friendly. Even though we decided not to use the UI at the end of the project, this thought us valuable skills like using PySide6 or QtCreator.

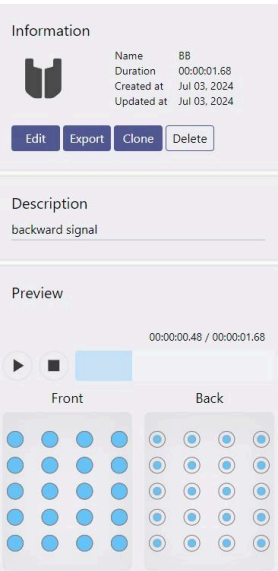
```python
from PySide6.QtWidgets import QWidget, QVBoxLayout, QPushButton, QMainWindow, QApplication, QTextEdit, QSlider, QHBoxLayout
from PySide6.QtCore import Qt
import directions
import keyboard
from directions import Directions
import sys
#import sounddevice as sd
#import numpy as np
import queue
import threading
import wave
import speech_recognition as sr
from gtts import gTTS
import os
import pyaudio

class VoiceRecorder:
    def __init__(self, ui, direc):
        #direc yeni eklendi
        self.direc = direc #yeni eklendi
        self.ui = ui
        self.is_recording = False
        self.frames = []
        self.p = pyaudio.PyAudio()
        self.stream = None

        # No need to create the Start Recording button here

    def toggle_recording(self):
        if self.is_recording:
            self.stop_recording()
            self.ui.log_message("Kayıt bitti")
        else:
            self.start_recording()
            self.ui.log_message("Kayıt başladı")

    def start_recording(self):
        self.is_recording = True
        self.frames = []
        self.stream = self.p.open(
            format=pyaudio.paInt16,
            channels=1,
            rate=16000,
            input=True,
            frames_per_buffer=3200
        )
        threading.Thread(target=self.record).start()

    def stop_recording(self):
        self.is_recording = False
        self.stream.stop_stream()
        self.stream.close()
        try:
            threading.Thread(target=self.save_recording).start()
        except:
            print("Uyarı")

    def record(self):
        while self.is_recording:
            data = self.stream.read(3200)
            self.frames.append(data)

    def save_recording(self):
        wf = wave.open("user.wav", 'wb')
        wf.setnchannels(1)
        wf.setsampwidth(self.p.get_sample_size(pyaudio.paInt16))
        wf.setframerate(16000)
        wf.writeframes(b''.join(self.frames))
        wf.close()
        try:
            self.recognize_and_respond()
        except:
            print("Lütfen tekrar deneyin")

    def recognize_and_respond(self):
        recognizer = sr.Recognizer()
        with sr.AudioFile("user.wav") as source:
            audio = recognizer.record(source)
        recognized_text = recognizer.recognize_google(audio, language="tr-TR")

        print(recognized_text)

        if "ileri" in recognized_text:
            self.ui.send_feedback("Sesli ileti anlaşıldı: ileri")
            self.direc.fwd(self.ui.intensity)
            os.system("start assets/audio/ileri.wav")

        elif "geri" in recognized_text:
            self.ui.send_feedback("Sesli ileti anlaşıldı: geri")
            self.direc.bwd(self.ui.intensity)
            os.system("start assets/audio/geri.wav")

        elif "sol" in recognized_text:
            self.ui.send_feedback("Sesli ileti anlaşıldı: sol")
            self.direc.left(self.ui.intensity)
            os.system("start assets/audio/sol.wav")

        elif "sağ" in recognized_text:
            self.ui.send_feedback("Sesli ileti anlaşıldı: sağ")
            self.direc.right(self.ui.intensity)
            os.system("start assets/audio/sağ.wav")

        else:
            print("Dediğinizi anlamadım")
```

```python
class UI(QMainWindow):
    def __init__(self, direc):
        super().__init__()

        self.setWindowTitle("Yelek Arayüzü")
        self.setMinimumSize(400, 300)

        self.direc = direc
        self.voice_recorder = VoiceRecorder(self, direc) #yeni eklendi

        # Central Widget and Layout
        central_widget = QWidget()
        self.setCentralWidget(central_widget)
        layout = QVBoxLayout()
        central_widget.setLayout(layout)

        # Intensity Slider
        self.intensity_slider = QSlider(Qt.Horizontal)
        self.intensity_slider.setRange(0, 4)
        self.intensity_slider.setValue(1)
        self.intensity = 1
        self.intensity_slider.setTickInterval(0.1)
        self.intensity_slider.setTickPosition(QSlider.TicksBelow)
        self.intensity_slider.valueChanged.connect(self.slider_value_changed)
        layout.addWidget(self.intensity_slider)

        # Arrow Buttons
        arrow_layout = QHBoxLayout()
        layout.addLayout(arrow_layout)

        arrow_up_button = QPushButton("↑")
        arrow_up_button.pressed.connect(lambda: self.send_feedback("Yazılı ileti anlaşıldı: ileri"))
        arrow_up_button.pressed.connect(lambda: self.direc.fwd(self.intensity))
        arrow_up_button.pressed.connect(lambda: os.system("start assets/audio/ileri.wav"))
        arrow_layout.addWidget(arrow_up_button)

        arrow_down_button = QPushButton("↓")
        arrow_down_button.pressed.connect(lambda: self.send_feedback("Yazılı ileti anlaşıldı: geri"))
        arrow_down_button.pressed.connect(lambda: self.direc.bwd(self.intensity))
        arrow_down_button.pressed.connect(lambda: os.system("start assets/audio/geri.wav"))
        arrow_layout.addWidget(arrow_down_button)

        arrow_left_button = QPushButton("←")
        arrow_left_button.pressed.connect(lambda: self.send_feedback("Yazılı ileti anlaşıldı: sol"))
        arrow_left_button.pressed.connect(lambda: self.direc.left(self.intensity))
        arrow_left_button.pressed.connect(lambda: os.system("start assets/audio/sol.wav"))
        arrow_layout.addWidget(arrow_left_button)

        arrow_right_button = QPushButton("→")
        arrow_right_button.pressed.connect(lambda: self.send_feedback("Yazılı ileti anlaşıldı: sağ"))
        arrow_right_button.pressed.connect(lambda: self.direc.right(self.intensity))
        arrow_right_button.pressed.connect(lambda: os.system("start assets/audio/sağ.wav"))
        arrow_layout.addWidget(arrow_right_button)

        # Voice Command Button (Replaces Start Recording button)
        self.voice_command_button = QPushButton("Sesli Komut")
        self.voice_command_button.clicked.connect(self.toggle_voice_command)
        layout.addWidget(self.voice_command_button)

        # Log Panel
        self.log_textedit = QTextEdit()
        layout.addWidget(self.log_textedit)

    def slider_value_changed(self, value):
        self.send_feedback(f"Şiddetin güncellendiği değer: {value}")
        self.intensity = value

    def send_feedback(self, message):
        self.log_message(message)

    def log_message(self, message):
        self.log_textedit.append(message)

    def toggle_voice_command(self):
        self.voice_recorder.toggle_recording()

if __name__ == "__main__":
    direc = Directions()
    app = QApplication(sys.argv)
    main_window = UI(direc)
    main_window.show()
    sys.exit(app.exec())
```



HAPTIC COMMUNICATION & WEBSOCKETS

We initially familiarized ourselves with WebSockets and prepared a demonstration where we first established communication within a single computer using a live server. After this, we extended the setup to enable text-based communication between two separate computers. We then connected the keyboard of a computer in the Robotics-Mechatronics Laboratory to the vest. Using unique tactile feedback we developed for each directional command, we were able to evoke the corresponding signal in the vest with each press of the up, down, right, and left arrow keys on our keyboard. Following this development, we incorporated a custom user interface through which the wearer could evoke the



corresponding commands in the vest. This UI had many customization options such as a tactile feedback intensity slider and various buttons to evoke various messages. We were then partially able to dictate the signal to be sent by the vest through audial commands. To incorporate spatial information into our physical output, we enhanced our server-client communication by implementing an automated "Got it!" response, allowing for instant message sending without waiting for a response. This way, an array of spatial information (coordinate data and detections of surrounding objects) could be communicated instantly to the haptic vest.

```python
import asyncio
import websockets

async def handler(websocket, path):
    while True:
        try:
            message = await websocket.recv()
            print(f"Received message from client: {message}")
            response = "Got it!"
            await websocket.send(response)
        except websockets.ConnectionClosed:
            print("Connection closed")
            break

async def main():
    async with websockets.serve(handler, "192.168.14.74", 8765):
        print("Server started, waiting for connection...")
        await asyncio.Future()

if __name__ == "__main__":
    asyncio.run(main())
```

(server)

```python
import asyncio
import websockets

class WebSocketClient:
    def __init__(self, uri):
        self.uri = uri
        self.websocket = None

    async def connect(self):
        async with websockets.connect(self.uri, timeout=30) as websocket:
            print("Connected to server")
            self.websocket = websocket
            await self.communicate()

    async def communicate(self):
        while True:
            message = await self.websocket.recv()
            print(f"Received message from server: {message}")

    async def send_message(self, message):
        if self.websocket:
            await self.websocket.send(message)

    def run(self):
        asyncio.run(self.connect())
```

(client)

GITHUB

GitHub's public repositories enable developers to modify, customize, and enhance software. We used functions like “git push, git commit -m, git add, git status, git stash pop/list." We created a repository and branches for ourselves, where we pushed everything we did so that we could record our process. By utilizing GitHub, we were able to track changes made by each team member and easily merge our work together. This allowed us to efficiently manage our project and ensure that everyone's contributions were properly documented.

ARUCO MARKERS, A* & DIJKSTRA

ArUco markers are binary-encoded, two-dimensional fiducial patterns that are intended to be easily found by computer vision systems. We utilized 15 ArUco markers and strategically placed them around the lab. These markers corresponded to fifteen specific points on the lab map. By integrating these markers with the A* algorithm, we successfully generated the shortest path from the user's location to their desired object. The camera detects the IDs of the ArUco markers and updates the user’s position accordingly. Subsequently, it provides directional commands, guiding the user on which direction to move and to which ArUco marker they should proceed next. This process continues step-by-step until the user reaches the ArUco marker associated with the target object. We tried using Dijkstra instead of A* first but realized that A* is significantly more practical than Dijkstra. As IOP science also mentions, “A* algorithm is just like Dijkstra's algorithm, and the only difference is that A* tries to look for a better path by using a heuristic function, which gives priority to nodes that are supposed to be better than others while

Dijkstra's just explore all possible ways" (Rachmawati, Gustin)

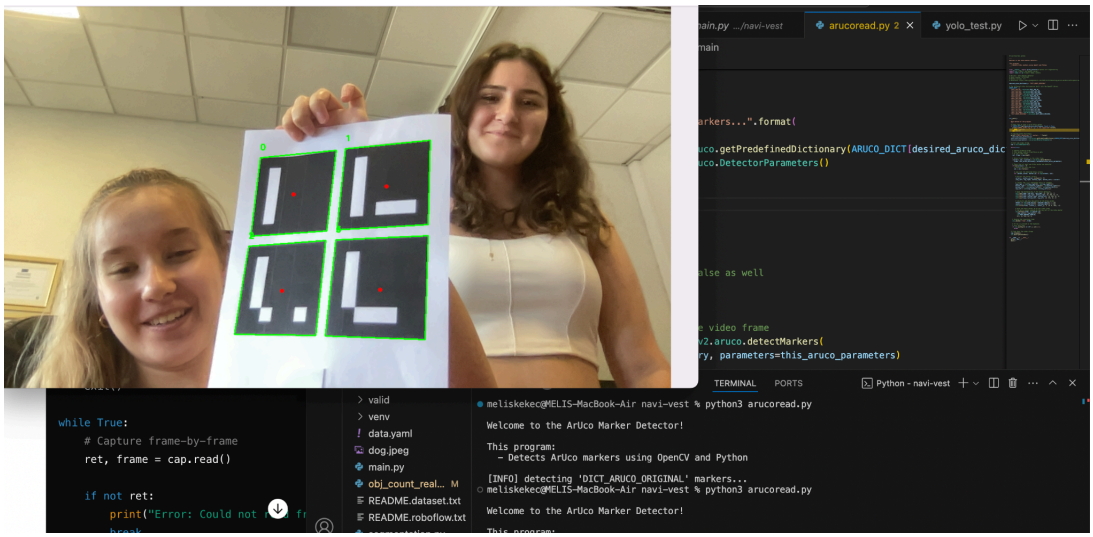


Image VI: Detecting ArUco Markers in Accordance with Assigned Numbers

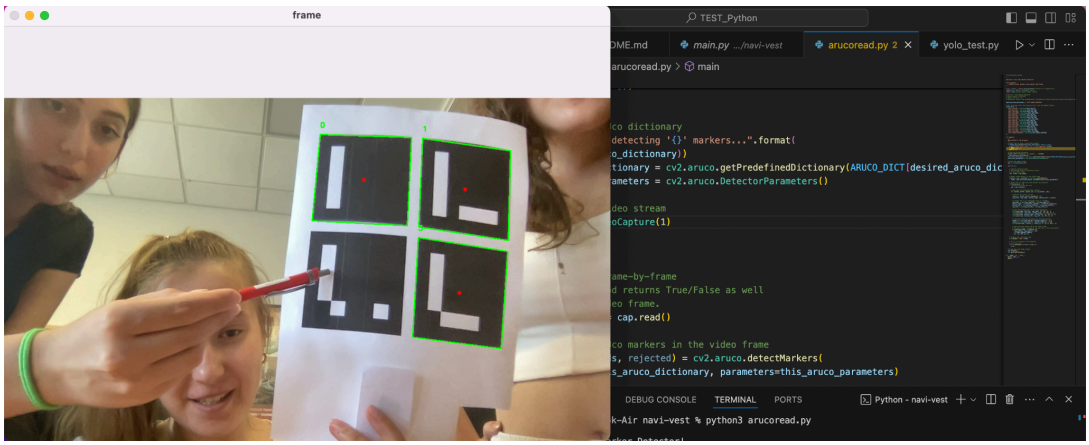


Image VII: Testing ArUco Markers in Accordance with Assigned Numbers

CONCLUSION

In conclusion, our project successfully demonstrates the integration of advanced image processing, artificial intelligence, and haptic feedback to aid individuals in indoor navigation. The use of YOLOv8 for real-time object detection has proven to be both efficient and effective, significantly improving the accuracy and speed of object recognition compared to earlier versions like YOLOv5. Our experiments with different model sizes provided valuable insights, leading us to select the optimal models for various application scenarios.

The audio processing component, primarily leveraging Google's Speech Recognition API, enabled seamless interaction between the user and the system. This auditory interface

ensured that visually impaired users could receive and process navigational information efficiently. Although we explored other transcription tools, Google's solution provided the best balance of accuracy and usability for our needs.

The user interface we designed facilitated easy interaction with the system, allowing users to issue commands and receive feedback intuitively. This interface, combined with the haptic feedback delivered through the vest, provided a comprehensive navigational aid that enhanced the user's spatial awareness and independence.

Our implementation of WebSockets for real-time communication between the user interface and the haptic vest ensured that tactile feedback was delivered promptly, aligning with the spatial data processed by the system. The integration of ArUco markers and the A* algorithm for pathfinding further enhanced the system's capability to guide users accurately within indoor environments.

GitHub's public repositories allow developers to modify and enhance software. We used various Git functions and created a repository with branches to track and merge our team's work. This enabled efficient project management and ensured proper documentation of everyone's contributions.

We placed 15 ArUco markers around the lab, corresponding to points on the map. Using these markers with the A* algorithm, we generated the shortest path from the user to their target object. The camera detects the markers' IDs, updates the user's position, and provides step-by-step directions until the user reaches the object. Although we initially considered Dijkstra's algorithm, A* proved more practical due to its heuristic function that prioritizes better paths (Rachmawati, Gustin).

Overall, this project has laid a solid foundation for developing advanced assistive technologies for indoor navigation. The combination of state-of-the-art image processing, audio interaction, and haptic feedback creates a robust system that can significantly improve the quality of life by enabling safer and more efficient navigation in indoor spaces. We hope that our work inspires further research and development in this important field.

Works Cited

https://www.google.com/imgres?q=YOLOv8%20models%2C%20(nano%2C%20..&imgurl=https%3A%2F%2Flearnopencv.com%2Fwp-content%2Fuploads%2F2023%2F01%2Fyolov8-models-comparison-new.png&imgrefurl=https%3A%2F%2Flearnopencv.com%2Fultralytics-yolov8%2F&docid=I-qR0cwjbHUTbM&tbnid=Y1HGFz3Zb3P27M&vet=12ahUKEwjE1ZCZorCHAxXpSvEDHTpgBgYQM3oECB0QAA..i&w=1600&h=900&hcb=2&ved=2ahUKEwjE1ZCZorCHAxXpSvEDHTpgBgYQM3oECB0QAA

https://learnopencv.com/ultralytics-yolov8/

Rachmawati, Dian, and Lysander Gustin. "IOPscience." Journal of Physics: Conference Series, IOP Publishing, 1 June 2020, iopscience.iop.org/article/10.1088/1742-6596/1566/1/012061/meta.