Exercise Report

Exercise 1

```
import sys

# Install OpenCV (only required if it's not already installed)
!{sys.executable} -m pip install opencv-contrib-python
```

Ensures that the OpenCV library is installed before execution. If running in an environment where OpenCV is already installed, this step can be skipped.

```
import cv2 # opencv library
import math
```

Imports OpenCV for handling video frames and math for mathematical operations. These are essential libraries for video processing tasks.

```
# List of video files
video_paths = ["Traffic_Laramie_1.mp4", "Traffic_Laramie_2.mp4"]

# Loop through each video and display properties
for video_path in video_paths:
    cap = cv2.VideoCapture(video_path)

    if not cap.isOpened():
        print(f"Error: Could not open video file {video_path}")
        continue  # Skip to the next video if it can't be opened

    # Get video properties
    frame_width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))   # Width of the frames
    frame_height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT)) # Height of the frames
    frame_rate = cap.get(cv2.CAP_PROP_FPS)                 # Frames per second
    total_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))  # Total number of frames
    duration = total_frames / frame_rate if frame_rate > 0 else 0  # Duration in seconds

    # Print video properties
    print(f"\nVideo Properties for {video_path}:")
    print(f"  - Frame Width: {frame_width} pixels")
    print(f"  - Frame Height: {frame_height} pixels")
    print(f"  - Frame Rate: {frame_rate:.2f} FPS")
    print(f"  - Total Frames: {total_frames}")
    print(f"  - Duration: {duration:.2f} seconds")

    cap.release()  # Release video capture before moving to the next
```

Loads multiple video files, checks if they open successfully, and extracts key properties such as resolution, FPS, and duration. Ensures robust error handling if a video file cannot be opened. Extracting video properties is useful for preprocessing before motion detection.

```
# Define Region of Interest (ROI) to limit detection area
detection_area = frame[260:600, 0:1040]
```

A specific region of the video frame is selected to limit detection to relevant areas, reducing noise from background activity

```
# Apply background subtraction to detect moving objects
fgmask = bg_sub.apply(detection_area)
_, fgmask = cv2.threshold(fgmask, 254, 255, cv2.THRESH_BINARY)  # Binarize the mask
contours, _ = cv2.findContours(fgmask, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)  # Find object contours
```

The function then applies background subtraction to detect moving objects, followed by thresholding to obtain a binary mask. Contours are extracted from the binary mask to identify potential vehicles.

```
# Large objects, likely cars, split into two parts for better tracking
if x < 520 and area > 13500 and h > 130:
    cx1, cy1 = int(x + (x + w) / 2), int(y + (y + h/2) / 2)
    cx2, cy2 = int(x + (x + w) / 2), int(y + h/2 + (y + h/2) / 2)
    current_frame_centroids.extend([(cx1, cy1), (cx2, cy2)])

    # Draw bounding boxes around detected cars
    cv2.rectangle(detection_area, (x, y), (x + w, y + int(h/2)), (0, 255, 0), 1)
    cv2.rectangle(detection_area, (x, y + int(h/2)), (x + w, y + h), (0, 255, 0), 1)

# Smaller objects, also likely vehicles
elif area > 4700:
    cx, cy = int(x + (x + w) / 2), int(y + (y + h) / 2)
    current_frame_centroids.append((cx, cy))
```

The detected objects are then filtered based on their area, distinguishing between large vehicles (which may be split into two parts) and smaller vehicles. Bounding boxes are drawn around detected vehicles to visualize tracking

```
# Calculate Euclidean distance between tracked and current centroid
distance = math.sqrt((tracked_obj[0] - ccentroid[0])**2 + (tracked_obj[1] - ccentroid[1])**2)

# If the object is within a reasonable distance, track it
if distance < 100:
    if nearest_track_obj and nearest_track_obj[2] > distance:
        nearest_track_obj = [ccentroid[0], ccentroid[1], distance, tracked_obj[2]]
```

A dictionary, tracked_objects, is used to maintain a record of tracked vehicles based on their centroids. The function calculates the Euclidean distance between detected centroids and

existing tracked objects to determine if they should be assigned a new tracking ID or linked to an existing object. If no match is found, a new object is added to tracked_objects.

```
if centroid[0] < 215:  # If the object moves past a certain point
    car_counter += 1  # Increase car count
```

Vehicles are counted when they cross a predefined boundary.

```
# Display the car count on the video frame
cv2.putText(frame, 'Cars Going to City Center: ', (10, 50),
            cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 1, cv2.LINE_AA)
cv2.putText(frame, str(car_counter), (500, 50),
            cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 2, cv2.LINE_AA)
```

The car count is displayed on the video frame using OpenCV

```
# Initialize background subtractor (you can reuse it)
bg_sub = cv2.createBackgroundSubtractorMOG2(history=15000, varThreshold=20, detectShadows=False)
```

A background subtractor is initialized once per video using the MOG2 algorithm. The history parameter defines how many frames the model should remember, and varThreshold controls sensitivity.

History ensures that long-term static objects are recognized as background.

varThreshold helps differentiate moving objects from the background.

detectShadows disables shadow detection to reduce false positives.

Final_Traffic_Analysis_Table

Video File	Total number of cars	Cars per minute
Traffic_Laramie_1.mp4	6	2.02
Traffic_Laramie_2.mp4	6	3.41

The application uses computer vision algorithms to process various traffic surveillance footage in order to detect and count vehicles. It counts the number of cars that pass through a predetermined area by iterating over a set of video files, subtracting the backdrop, and using centroid-based tracking. By keeping a distinct function for car counting that can be applied to other video files without requiring changes, the method guarantees modularity.

The application's automation and scalability are two of its main advantages. It becomes an effective tool for examining traffic patterns across several places or timeframes by enabling the sequential processing of many recordings without necessitating modifications to the fundamental logic. Vehicle detection is made more effective by removing immobile backdrop objects through the usage of OpenCV's cv2.createBackgroundSubtractorMOG2 for motion detection. The program also has error-handling features that keep the process from stopping altogether in the case that a video file doesn't open.

However, some restrictions might affect the tracking accuracy. The current solution may have trouble with overlapping cars, which could result in miscounts, because it depends on centroid proximity to track moving vehicles. A more advanced tracking algorithm, like Kalman Filters or Deep SORT, which are better suited for managing occlusions and object re-identification, could be used to solve this problem.

The use of predefined parameters for background removal is another possible disadvantage. Because the history and varThreshold values are preset, they might not adjust well to changes in illumination or video quality. The accuracy of detection would be increased by a more dynamic method that modifies these parameters according on the features of each video. Additionally, rather than analysing real-time traffic inputs, the application can only process recorded films at this moment. It might be more useful for traffic monitoring in the real world if its capabilities were expanded to support live video broadcasts.

To sum up, the application offers a strong basis for counting and detecting vehicles in traffic footage. It is a useful tool for batch processing because of its automated workflow and modular design. For real-world traffic monitoring applications, however, its utility and dependability would be greatly increased by advancements in tracking precision, adaptive parameter tuning and real-time processing.

Exercise 2

In [1]:
```
import os
import wave
from scipy.io import wavfile
import numpy as np
import struct
from functools import partial
```

Several necessary Python libraries are imported at the start of the notebook. WAV file operations are handled by the wave module, whilst binary data processing is done by the struct module. Numerical operations and the management of reading and writing WAV files are supported by the numpy and scipy.io.wavfile libraries. Furthermore, functools.partial is present, though it's unclear what exactly it does given the code samples that were retrieved.

In [2]:
```
def read_wav_header(input_file):
    """ Reads and extracts WAV file header information. """
    header_info = input_file.read(44)
    header_fields = {
        "Chunk Size": struct.unpack('<L', header_info[4:8])[0],
        "Format": header_info[8:12],
        "Subchunk Size": struct.unpack('<L', header_info[16:20])[0],
        "Audio Format": struct.unpack('<H', header_info[20:22])[0],
        "Number of Channels": struct.unpack('<H', header_info[22:24])[0],
        "Sample Rate": struct.unpack('<L', header_info[24:28])[0],
        "Byte Rate": struct.unpack('<L', header_info[28:32])[0],
        "Block Align": struct.unpack('<H', header_info[32:34])[0],
        "Bits per sample": struct.unpack('<H', header_info[34:36])[0],
        "File Size": struct.unpack('<L', header_info[40:44])[0]
    }
    for key, value in header_fields.items():
        print(f"{key}: {value}")
    return header_info
```

Reading the WAV file header, which includes important metadata including chunk size, format type, number of channels, sample rate, byte rate, block alignment, bits per sample, and total file size, is one of the notebook's initial tasks. Read_wav_header(input_file), the function in charge of this, extracts and shows these values, giving important details about the structure of the audio file being processing. A single-channel (mono) WAV file with a sample rate of 44,100 Hz and a 16-bit sample depth is displayed in the metadata that was recovered from an example execution.

In [3]:
```python
def encode_rice(sample, k):
    """ Applies Rice encoding to a single sample. """
    M = 2**k
    sign_bit = '1' if sample < 0 else '0'
    s = abs(sample)
    R = bin(s & (M - 1))[2:].zfill(k)  # Extract remainder bits
    Q = '1' * (s >> k) + '0'  # Unary quotient encoding
    return sign_bit + Q + R
```

The next major component of the notebook is the implementation of Rice encoding, a lossless data compression technique. The function encode_rice(sample, k) takes an individual audio sample and applies Rice encoding by separating it into a sign bit, a quotient encoded in unary representation, and a remainder encoded in binary form. This encoding technique relies on parameter k, which determines the bit division between the quotient and remainder.

In [12]:
```python
original_size = os.path.getsize("Sound1.wav")
encoded_file_size = os.path.getsize("Sound1_Enc.ex2")
decoded_file_size = os.path.getsize("Sound1_EncDec.wav")

print("Where k = 4:")
print(f"original file size: {original_size}")
print(f"encoded file size: {encoded_file_size}")
print(f"decoded file size: {decoded_file_size}")
print(f"compression: {round((original_size - encoded_file_size)/original_size * 100, 2)}%")
```

```
Where k = 4:
original file size: 1002088
encoded file size: 1875237
decoded file size: 1002086
compression: -87.13%
```

Following encoding, the notebook assesses how well the audio data was compressed using Rice encoding. The encoded file size is substantially bigger than the original, according to the compression study results for k=4. The encoded file grows to about 1,875,237 bytes, although the original file size is recorded at about 1,002,088 bytes. The lossless nature of the encoding technique is confirmed by the decoded file size being almost the same as the original. With a compression ratio of roughly -87.13%, the compression efficiency is negative. This implies that rice encoding, in this specific case, increases file size rather than effectively compressing the provided audio dataset.

	Original size	Rice (K = 4 bits)	Rice (K = 2 bits)	% Compression (K = 4 bits)	% Compression (K = 2 bits)
Sound1.wav	1002088	1875237	1994540	-87.13%	-99.04%
Sound2.wav	1008044	2499947	2535876	-148.0%	-151.56%

The observed inefficiency in compression can likely be attributed to the characteristics of the audio data. Rice encoding is generally more effective when dealing with data that has a predictable structure with frequent repetitions or low entropy. In the case of audio signals,

especially those with high variability or a wide range of frequency components, Rice encoding may not be optimal. Alternative compression methods such as Huffman coding, FLAC, or predictive coding techniques could potentially achieve better results.

Why the compression rates of the files are so different

The significant difference in compression rates between the two audio files, Sound1.wav and Sound2.wav, can be attributed to variations in their signal characteristics, which directly impact the effectiveness of Rice encoding. Compression efficiency depends on the predictability and redundancy of the data. When an audio file contains repetitive patterns, such as silence, sustained tones, or gradual variations, lossless compression methods like Rice encoding can efficiently reduce file size. However, if the file consists of highly complex and unpredictable variations, such as dynamic music, speech, or environmental noise, compression becomes inefficient, often leading to an increase in file size rather than a reduction.

Rice encoding with K=4 produced a compression rate of -87.13% for Sound1.wav, according to the results, which indicates that the encoded file grew noticeably larger than the original. With a rate of -99.04%, using K=2 made the compression even worse. This suggests that the audio stream was not successfully compressed by the encoding strategy, maybe as a result of an unfavourable distribution of sample values. However, the compression performance of Sound2.wav was much worse, with K=4 producing -148.0% and K=2 producing -151.56%. This suggests that the file contains even more random fluctuations, making Rice encoding useless.

Further Development

The first major improvement involves the introduction of an adaptive k-selection mechanism. The function adaptive_k_selection(samples) determines the optimal k value based on the standard deviation of the audio samples. This function dynamically selects k = 2 when the standard deviation of the signal is below 32, otherwise setting k = 4. By allowing k to adapt based on the characteristics of the input data, the encoding process becomes more efficient for varying signal types. This approach is particularly useful for handling different types of audio content, as it prevents suboptimal fixed values from degrading compression performance.

In addition to adaptive k selection, the notebook incorporates Golomb encoding as an alternative to Rice encoding. The function encode_golomb(sample, k) modifies the encoding process by replacing the unary quotient representation in Rice encoding with a binary representation. This adjustment improves efficiency by ensuring that the quotient is stored compactly, reducing redundancy in cases where high values of k would otherwise lead to

excessive unary encoding lengths. The function also utilizes the bitarray library to efficiently pack the encoded bits, reducing memory overhead and improving storage efficiency.

The inclusion of Golomb encoding is significant because it generalizes Rice encoding, allowing for better compression in scenarios where data distributions vary. In cases where Rice encoding results in negative compression efficiency, as seen in the earlier analysis, Golomb encoding can offer improved performance by leveraging a more compact representation of the quotient. By adapting the encoding technique to the nature of the data, this approach increases the likelihood of achieving actual compression rather than expansion.

In [28]:
```
# Example
original_size = os.path.getsize("Sound1.wav")
rice_encoding_compression('Sound1.wav')
delta_golomb_packed_size = os.path.getsize("Sound1_DeltaGolombPacked.bin")
analyze_compression(original_size, delta_golomb_packed_size)
```

```
Chunk Size: 1002080
Format: b'WAVE'
Subchunk Size: 16
Audio Format: 1
Number of Channels: 1
Sample Rate: 44100
Byte Rate: 88200
Block Align: 2
Bits per sample: 16
File Size: 1002044
Using adaptive k = 4 (Delta + Golomb encoding + Bit-Packing)
Compressed bytes: 83445
Uncompressed bytes: 172374

Compression Analysis:
+-----------------------------------------------+---------------------+-------------------+
| File Type                                     |   File Size (bytes) | Compression (%)   |
+===============================================+=====================+===================+
| Original                                      |             1002088 | 0%                |
+-----------------------------------------------+---------------------+-------------------+
| Encoded (Delta + Golomb + Bit-Packing, max 4) |              514443 | 48.66%            |
+-----------------------------------------------+---------------------+-------------------+
```

https://hub.labs.coursera.org:443/connect/sharedqfkwhzwg?forceRefresh=false&isLabVersioning=true

Exercise 3

Installing ffprobe and ffmpeg

```
(base) Penda@M2-14-inch-Macbook-Pro ~ % brew install ffmpeg
==> Auto-updating Homebrew...
Adjust how often this is run with HOMEBREW_AUTO_UPDATE_SECS or disable with
HOMEBREW_NO_AUTO_UPDATE. Hide these hints with HOMEBREW_NO_ENV_HINTS (see `man b
rew`).
==> Auto-updated Homebrew!
Updated 2 taps (homebrew/core and homebrew/cask).
==> New Formulae
evans        iguana       jenkins-cli  kbld         otel-cli     restish
==> New Casks
cloudflare-warp@beta                    font-special-gothic-expanded-one
desktime                                sokim
font-source-han-code-jp                 soundanchor
font-special-gothic-condensed-one

You have 3 outdated formulae installed.

Warning: ffmpeg 7.1.1_1 is already installed and up-to-date.
To reinstall 7.1.1_1, run:
  brew reinstall ffmpeg
```

Installing brew install ffmpeg

```
Last login: Mon Mar  3 20:10:20 on ttys002
(base) Penda@M2-14-inch-Macbook-Pro ~ % ffprobe -version

ffprobe version 7.1.1 Copyright (c) 2007-2025 the FFmpeg developers
built with Apple clang version 16.0.0 (clang-1600.0.26.6)
configuration: --prefix=/opt/homebrew/Cellar/ffmpeg/7.1.1 --enable-shar
le-pthreads --enable-version3 --cc=clang --host-cflags= --host-ldflags=
classic' --enable-ffplay --enable-gnutls --enable-gpl --enable-libaom -
ibaribb24 --enable-libbluray --enable-libdav1d --enable-libharfbuzz --e
jxl --enable-libmp3lame --enable-libopus --enable-librav1e --enable-lib
able-librubberband --enable-libsnappy --enable-libsrt --enable-libssh -
ibsvtav1 --enable-libtesseract --enable-libtheora --enable-libvidstab -
```

```
(base) Penda@M2-14-inch-Macbook-Pro ~ % ffmpeg -version
ffmpeg version 7.1.1 Copyright (c) 2000-2025 the FFmpeg developers
built with Apple clang version 16.0.0 (clang-1600.0.26.6)
configuration: --prefix=/opt/homebrew/Cellar/ffmpeg/7.1.1_1 --enable-
able-pthreads --enable-version3 --cc=clang --host-cflags= --host-ldfl
d_classic' --enable-ffplay --enable-gnutls --enable-gpl --enable-liba
-libaribb24 --enable-libbluray --enable-libdav1d --enable-libharfbuzz
ibjxl --enable-libmp3lame --enable-libopus --enable-librav1e --enable-
```

Verify with ffmpeg -version and ffprobe -version

```
In [1]: !pip install ffprobe-python ffmpeg-python
        Requirement already satisfied: ffprobe-python in /Users/Penda/anaconda3/lib/python3.11/site-packages (1.0.3)
        Requirement already satisfied: ffmpeg-python in /Users/Penda/anaconda3/lib/python3.11/site-packages (0.2.0)
        Requirement already satisfied: future in /Users/Penda/anaconda3/lib/python3.11/site-packages (from ffmpeg-python)
        (0.18.3)
```

These libraries allow interaction with ffmpeg and ffprobe via Python.

```
In [2]: import subprocess

        try:
            subprocess.run(["ffprobe", "-version"], check=True)
            print("ffprobe is working correctly!")
        except FileNotFoundError:
            print("Error: ffprobe not found. Make sure it is installed and in your system's PATH.")

        Error: ffprobe not found. Make sure it is installed and in your system's PATH.
```

The script attempts to check if ffprobe is correctly installed. If ffprobe is not found, it prints an error message indicating that it may not be in the system's PATH.

```
In [3]: import os
        os.environ["PATH"] += os.pathsep + "/opt/homebrew/bin"
```

The PATH environment variable is modified to ensure ffmpeg and ffprobe are accessible.

Brief Analysis

The application is designed to analyze and standardize video files by ensuring they conform to a predefined set of technical specifications. It identifies problematic fields within the video and audio streams and applies corrective measures using FFmpeg filters.

This application's main objective is to check and adjust video files to meet certain technical specifications. Using the MP4 container and preserving the H.264 video and AAC audio codecs, the application makes sure that video files adhere to a standard format. It also makes sure that the audio channel configuration is set to stereo, manages video and audio bit rates within predetermined bounds, adjusts the resolution to 640x360, enforces a 16:9 aspect ratio, and checks for a steady frame rate of 25 FPS.

In order to process video files, the application first retrieves them from a specified directory. After that, it checks to see if the audio and video streams meet the necessary requirements. The program uses FFmpeg filters to process the files and modify characteristics like frame rate, resolution, and bit rate if any differences are detected. Following the application of the corrections, the output is produced in a formatted state that complies with the necessary requirements.

Python is used in conjunction with the FFmpeg and FFprobe libraries to implement the application. To guarantee correct operation, it starts with installing and configuring these tools. The video and audio streams are inspected using a core function called find_problematic_fields, which finds any violations from the necessary standards. FFmpeg is used to alter the video stream by changing its frame rate, resolution, and bit rate if any problems are found. In a similar manner, the audio parameters are adjusted as needed to guarantee adherence to codec and

channel layout specifications. The application performs a final check after the changes are performed to ensure that the processed video now satisfies the required standards.

Brief Description

A video format, sometimes referred to as a container, is a type of file format that contains information, audio, video, and subtitles all in one. It controls the storage and playback of these many media components. MP4, AVI, MKV, and MOV are a few popular video containers; each offers varying degrees of compression, compatibility, and support for multiple codecs.

Video files can be compressed and decompressed using a video codec, which lowers their size without sacrificing visual quality. Video files would be too big for effective streaming and storage if they weren't compressed. H.264, which is frequently used for streaming and playback, and H.265 (HEVC), which offers superior compression while maintaining quality, are examples of popular video codecs.

Similar to a video codec, an audio codec is used for sound compression and decompression. It has an impact on an audio track's overall sound quality and file size. MP3 is still a commonly accepted standard for compressed audio, but AAC (Advanced Audio Codec) is one of the most popular audio codecs, providing effective compression with high-quality output.

The number of individual frames shown in a video every second is referred to as the frame rate. It affects the video's motion smoothness and is measured in frames per second (FPS). While 25 frames per second is the norm for PAL television broadcasts and 30 frames per second is the norm for NTSC television and web videos, a lower frame rate, such 24 frames per second, is sometimes utilised in movies to create a theatrical atmosphere.

The proportionate relationship between a video frame's width and height is referred to as its aspect ratio. It affects how a video looks on various screens. While older televisions and some archival content use a 4:3 aspect ratio, the majority of current screens, including HDTVs and internet video platforms, use a 16:9 aspect ratio.

The number of pixels in a video frame, or resolution, dictates how clear and detailed the image is. Typically, it is represented in pixels as width × height. High clarity, for instance, is provided by a resolution of 1920 x 1080, or Full HD, but a smaller resolution, like 640 x 360, produces a smaller file size but worse image quality.

Megabits per second (Mbps) is the common unit of measurement for video bit rate, which is the amount of data processed each second in a video file. Better video quality is possible with a greater bit rate, but the file size also grows. For typical streaming, a video with a bit rate of 2 to 5 Mbps is seen to strike a decent compromise between efficiency and quality.

Conversely, audio bit rate, which is commonly expressed in kilobits per second (kbps), quantifies the data rate of an audio stream. While lower bit rates might result in apparent

compression artefacts, particularly in content with a lot of music and dialogue, higher bit rates—like 256 kbps—offer better audio quality.

The quantity of distinct sound sources in an audio track is referred to as audio channels. There is only one channel in a mono audio file, therefore all of the speakers play the same sound. The two channels of stereo audio enable directional sound, making for a more engaging listening experience. More sophisticated configurations, like 5.1 or 7.1 surround sound, which are frequently used in home theatres and movie theatres, divide audio among several speakers to produce a three-dimensional soundscape.

https://hub.labs.coursera.org:443/connect/sharedqfkwhzwg?forceRefresh=false&isLabVersioning=true

Exercise 4

1. Review and critically discuss at least three emerging computer vision applications that you find most interesting with justifications.

One new use of computer vision that is revolutionising the shopping experience is retail automation. Cashier-less businesses, like Amazon Go, where computer vision systems track customers and the things they pick up, were pioneered by companies like Amazon. Traditional checkout procedures are no longer necessary thanks to cameras and sensors placed throughout the store that track movements and immediately charge clients as they leave. Because it lowers operating expenses for businesses and improves customer convenience, this application is incredibly inventive. However, because consumers are continuously watched, privacy issues are brought up. Furthermore, the technology needs to be extremely precise in order to prevent billing problems. Retail automation is a promising use of computer vision that has the potential to completely change how people shop in the future, despite these obstacles.

In natural environments, computer vision is being used to track biodiversity and observe wildlife. Computer vision algorithms are used to analyse the photos and videos of animals taken by cameras positioned in forests, oceans, and other

habitats. These systems are able to count populations, identify species, and even recognise behaviours like poaching or migration trends. Because it offers useful data for safeguarding endangered species and maintaining ecosystems, this application is essential for conservation efforts. However, obstacles include the difficulty of setting up cameras in harsh or distant locations and the requirement for sizable datasets to train models for rare species. Computer vision is proving to be an effective tool for animal conservation in spite of these challenges.

Sports are using computer vision more and more to evaluate player performance and enhance training regimens. During games or practice sessions, movements are recorded by cameras and sensors. Computer vision algorithms then examine this data to provide insights regarding tactics, methods, and physical conditioning. For instance, to evaluate a football player's speed, agility, and ball handling, algorithms can follow their actions. Coaches and athletes can benefit from this application since it offers objective statistics to improve performance and lower injury risk. The intricacy of assessing team dynamics and the requirement for high-speed cameras to record quick motions present difficulties, though. Sports analytics is a developing subject that exemplifies the versatility of computer vision in spite of these obstacles.

2. Provide technical discussions on two popular or state-of-the-art computer vision techniques adapted with suitable diagrams, where relevant.

The foundation of many computer vision applications is made up of convolutional neural networks, or CNNs. Their architecture, which was created especially for image processing, consists of several layers that gradually extract and examine visual characteristics. In order to identify patterns such as edges and textures, convolutional layers apply filters to the input image. The feature maps' spatial dimensions are then decreased by pooling layers, which streamlines the data while maintaining crucial details. Lastly, using the features that were retrieved, fully linked layers carry out regression or classification. The range of architectures, filter sizes, and activation functions employed by CNNs demonstrates their versatility. Their adaptability is further increased via transfer learning, which involves fine-tuning previously trained CNNs on fresh datasets.

YOLO is a cutting-edge real-time image processing technique for object detection. By dividing the image into a grid and predicting bounding boxes and class probabilities for each grid cell, YOLO differs from conventional techniques that rely on area recommendations. Because of this method, YOLO can be incredibly quick and precise, which makes it appropriate for uses like retail automation, surveillance, and autonomous driving.
With the ability to process photos in milliseconds, YOLO's quickness is its main asset. In busy settings, it could have trouble identifying little things or objects that overlap. Because of its precision and efficiency, YOLO is utilised extensively despite these drawbacks.

3. Discuss two potential creative and novel uses of these computer vision and audio feature processing suitable for addressing real-life problems.

The creation of assistive technology for those with visual impairments is an innovative use of computer vision and audio processing. Computer vision systems can employ cameras to scan their surroundings and give users aural feedback in real time. For instance, a wearable gadget or smart cane might be able to read material aloud, identify faces, and detect impediments, enabling those with visual impairments to move around more freely.

By enhancing the lives of those who are blind or visually challenged, this program solves a major real-world issue. Among the difficulties include making sure the system is accurate in a variety of settings, lightweight, and reasonably priced. Nonetheless, this technology's potential influence makes it a very intriguing field for study.

Monitoring the environment and detecting pollution are two more innovative applications of computer vision and audio processing. To identify pollution sources like smoke, chemical leaks, or noise pollution, cameras and microphones can be placed in industrial or urban locations. While computer vision algorithms can identify pollution by analysing visual data, audio processing can identify specific sounds linked to environmental dangers or dangerous noise levels.

Addressing environmental issues and maintaining public health depend heavily on this application. For instance, it could assist cities in detecting unlawful dumping activities or monitoring the quality of the air. The integration of data from many sensors and the requirement for reliable models that can function in a range of weather situations present difficulties. This technology has the potential to significantly affect environmental sustainability in spite of these obstacles.

Code Report

Exercise 1

```
import sys
!{sys.executable} -m pip install opencv-contrib-python

import cv2
import math

# List of video files
video_paths = ["Traffic_Laramie_1.mp4", "Traffic_Laramie_2.mp4"]

# Loop through each video and display properties
for video_path in video_paths:
    cap = cv2.VideoCapture(video_path)

    if not cap.isOpened():
        print(f"Error: Could not open video file {video_path}")
        continue  # Skip to the next video if it can't be opened

    # Get video properties
    frame_width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))   # Width of the frames
    frame_height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT)) # Height of the frames
    frame_rate = cap.get(cv2.CAP_PROP_FPS)                 # Frames per second
    total_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))  # Total number of frames
    duration = total_frames / frame_rate if frame_rate > 0 else 0  # Duration in seconds

    # Print video properties
    print(f"\nVideo Properties for {video_path}:")
```

```
    print(f"  - Frame Width: {frame_width} pixels")
    print(f"  - Frame Height: {frame_height} pixels")
    print(f"  - Frame Rate: {frame_rate:.2f} FPS")
    print(f"  - Total Frames: {total_frames}")
    print(f"  - Duration: {duration:.2f} seconds")

    cap.release()  # Release video capture before moving to the next

def car_counter(video_cap, bg_sub):
    '''
    This function tracks and counts moving cars in a specified area using background
subtraction.
    '''

    # Check if the video file was successfully opened
    if not video_cap.isOpened():
        print("Error: Could not open video file.")
        return

    # Get video properties
    amount_of_frames = int(video_cap.get(cv2.CAP_PROP_FRAME_COUNT))  # Total number
of frames
    video_fps = video_cap.get(cv2.CAP_PROP_FPS)  # Frames per second
    video_duration = amount_of_frames / video_fps if video_fps > 0 else 1  # Avoid division by
zero

    # Initialize variables for tracking
    previous_frame_centroids = []  # Stores centroids from previous frame
    tracked_objects = {}  # Dictionary to track objects
    tracking_id = 0  # Unique ID for each tracked object
    car_counter = 0  # Counter for detected cars

    # Process each frame in the video
    for _ in range(amount_of_frames):
        current_frame_centroids = []  # Stores centroids in the current frame

        # Read a frame from the video
        ret, frame = video_cap.read()
        if not ret or frame is None:
            print("End of video or error reading frame.")
            break

        # Define Region of Interest (ROI) to limit detection area
        detection_area = frame[260:600, 0:1040]
```

```
        # Apply background subtraction to detect moving objects
        fgmask = bg_sub.apply(detection_area)
        _, fgmask = cv2.threshold(fgmask, 254, 255, cv2.THRESH_BINARY)  # Binarize the mask
        contours, _ = cv2.findContours(fgmask, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
# Find object contours

        # Loop through detected contours
        for cnt in contours:
            area = cv2.contourArea(cnt)  # Get contour area
            x, y, w, h = cv2.boundingRect(cnt)  # Get bounding box for detected object

            # Large objects, likely cars, split into two parts for better tracking
            if x < 520 and area > 13500 and h > 130:
                cx1, cy1 = int(x + (x + w) / 2), int(y + (y + h/2) / 2)
                cx2, cy2 = int(x + (x + w) / 2), int(y + h/2 + (y + h/2) / 2)
                current_frame_centroids.extend([(cx1, cy1), (cx2, cy2)])

                # Draw bounding boxes around detected cars
                cv2.rectangle(detection_area, (x, y), (x + w, y + int(h/2)), (0, 255, 0), 1)
                cv2.rectangle(detection_area, (x, y + int(h/2)), (x + w, y + h), (0, 255, 0), 1)

            # Smaller objects, also likely vehicles
            elif area > 4700:
                cx, cy = int(x + (x + w) / 2), int(y + (y + h) / 2)
                current_frame_centroids.append((cx, cy))

                # Draw a bounding box around detected objects
                cv2.rectangle(detection_area, (x, y), (x + w, y + h), (0, 255, 0), 1)

        # Track detected objects using centroids
        for ccentroid in current_frame_centroids:
            nearest_track_obj = []  # Store the closest tracked object

            for object_key in tracked_objects:
                tracked_obj = tracked_objects[object_key]
                # Calculate Euclidean distance between tracked and current centroid
                distance = math.sqrt((tracked_obj[0] - ccentroid[0])**2 + (tracked_obj[1] -
ccentroid[1])**2)

                # If the object is within a reasonable distance, track it
                if distance < 100:
                    if nearest_track_obj and nearest_track_obj[2] > distance:
                        nearest_track_obj = [ccentroid[0], ccentroid[1], distance, tracked_obj[2]]
```

```
                elif not nearest_track_obj:
                    nearest_track_obj = [ccentroid[0], ccentroid[1], distance, tracked_obj[2]]

            # If no nearby tracked object is found, assign a new tracking ID
            if not nearest_track_obj:
                tracked_objects[tracking_id] = [ccentroid[0], ccentroid[1], tracking_id, True]
                tracking_id += 1
            else:
                update_track_id = nearest_track_obj[3]
                tracked_objects[update_track_id] = [ccentroid[0], ccentroid[1], update_track_id, True]

            # Clear temporary tracking object
            nearest_track_obj.clear()

        # Remove old tracked objects and count cars if they exit the frame
        for object_key in list(tracked_objects.keys()):
            centroid = tracked_objects[object_key]
            if centroid[3]:
                tracked_objects[centroid[2]][3] = False
            else:
                if centroid[0] < 215:  # If the object moves past a certain point
                    car_counter += 1  # Increase car count
                del tracked_objects[centroid[2]]  # Remove old tracked objects

        # Display the car count on the video frame
        cv2.putText(frame, 'Cars Going to City Center: ', (10, 50),
                    cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 1, cv2.LINE_AA)
        cv2.putText(frame, str(car_counter), (500, 50),
                    cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 2, cv2.LINE_AA)

        # Show the processed video frame
        cv2.imshow("Frame", frame)

        # Press 'q' to stop the video early
        if cv2.waitKey(1) == ord('q'):
            break

    # Print final car count and statistics
    print(f"Total cars going to city center: {car_counter}")
    print(f"Cars per minute: {round(car_counter / (video_duration / 60), 2)}")

    # Release video capture and close all OpenCV windows
    video_cap.release()
    cv2.destroyAllWindows()
```

```
# List of video files
video_files = ['Traffic_Laramie_1.mp4', 'Traffic_Laramie_2.mp4']

# Loop through both videos without modifying the main processing code
for video_path in video_files:
    video_cap = cv2.VideoCapture(video_path)  # Open the video

    # Initialize background subtractor (you can reuse it)
    bg_sub = cv2.createBackgroundSubtractorMOG2(history=15000, varThreshold=20,
detectShadows=False)

    if not video_cap.isOpened():
        print(f"Failed to open {video_path}")
        continue  # Skip to the next video

    print(f"Processing video: {video_path}")

    # Run your existing car counting code here, unchanged
    # (Make sure your car_counter function is defined and used here)
    car_counter(video_cap, bg_sub)

    # Release video capture after processing
    video_cap.release()

cv2.destroyAllWindows()  # Close all OpenCV windows after processing both videos
```

Exercise 2

```
import os
import wave
from scipy.io import wavfile
import numpy as np
import struct
from functools import partial

def read_wav_header(input_file):
    """ Reads and extracts WAV file header information. """
    header_info = input_file.read(44)
    header_fields = {
        "Chunk Size": struct.unpack('<L', header_info[4:8])[0],
        "Format": header_info[8:12],
        "Subchunk Size": struct.unpack('<L', header_info[16:20])[0],
        "Audio Format": struct.unpack('<H', header_info[20:22])[0],
```

```
        "Number of Channels": struct.unpack('<H', header_info[22:24])[0],
        "Sample Rate": struct.unpack('<L', header_info[24:28])[0],
        "Byte Rate": struct.unpack('<L', header_info[28:32])[0],
        "Block Align": struct.unpack('<H', header_info[32:34])[0],
        "Bits per sample": struct.unpack('<H', header_info[34:36])[0],
        "File Size": struct.unpack('<L', header_info[40:44])[0]
    }
    for key, value in header_fields.items():
        print(f"{key}: {value}")
    return header_info
def read_wav_header(input_file):
    """ Reads and extracts WAV file header information. """
    header_info = input_file.read(44)
    header_fields = {
        "Chunk Size": struct.unpack('<L', header_info[4:8])[0],
        "Format": header_info[8:12],
        "Subchunk Size": struct.unpack('<L', header_info[16:20])[0],
        "Audio Format": struct.unpack('<H', header_info[20:22])[0],
        "Number of Channels": struct.unpack('<H', header_info[22:24])[0],
        "Sample Rate": struct.unpack('<L', header_info[24:28])[0],
        "Byte Rate": struct.unpack('<L', header_info[28:32])[0],
        "Block Align": struct.unpack('<H', header_info[32:34])[0],
        "Bits per sample": struct.unpack('<H', header_info[34:36])[0],
        "File Size": struct.unpack('<L', header_info[40:44])[0]
    }
    for key, value in header_fields.items():
        print(f"{key}: {value}")
    return header_info
def encode_rice(sample, k):
    """ Applies Rice encoding to a single sample. """
    M = 2**k
    sign_bit = '1' if sample < 0 else '0'
    s = abs(sample)
    R = bin(s & (M - 1))[2:].zfill(k)  # Extract remainder bits
    Q = '1' * (s >> k) + '0'  # Unary quotient encoding
    return sign_bit + Q + R
def rice_encoding_compression(filename, k):
    """ Compresses WAV file samples using Rice encoding. """
    try:
        input_file = open(filename, 'rb')
        encoded_file = open(f"{filename.split('.')[0]}_Enc.ex2", "w")
        header_info = read_wav_header(input_file)

        compressed_bytes = 0
```

```
        uncompressed_bytes = 0
        M = 2**k

        for samplebyte in iter(partial(input_file.read, 2), b''):
            sample = struct.unpack('<h', samplebyte)[0]

            if -128 < sample < 127:
                encoded_code = encode_rice(sample, k)
                hex_obj = hex(int(encoded_code, 2))[2:]
                encoded_file.write(f"/{hex_obj}")
                compressed_bytes += 1
            else:
                encoded_file.write(f"\\{samplebyte.hex()}")
                uncompressed_bytes += 1

        print(f"Compressed bytes: {compressed_bytes}")
        print(f"Uncompressed bytes: {uncompressed_bytes}")

    except Exception as e:
        print("An error occurred:", e)
    finally:
        input_file.close()
        encoded_file.close()
def write_wav_header_info(decoded_file):
    """ Sets header parameters for decoded WAV file. """
    decoded_file.setnchannels(1)
    decoded_file.setsampwidth(2)
    decoded_file.setframerate(44100)
def convert_hex_to_decoded_data(hex_string, is_compressed, M):
    """ Converts hex-encoded data back into WAV format. """
    if is_compressed:
        integer_value = int(hex_string, 16)
        encoded_code = bin(integer_value)[2:]

        sign_bit = encoded_code[0] if len(encoded_code) > 1 else '0'
        encoded_code = encoded_code[1:] if len(encoded_code) > 1 else encoded_code

        Q = encoded_code.count('1')
        encoded_code = encoded_code[Q:]
        R = int(encoded_code[-4:], 2) if encoded_code else 0
        s = Q * M + R

        if sign_bit == '1':
            s = -s
```

```
        return struct.pack('<h', s)
    else:
        return bytearray.fromhex(hex_string)
def rice_compression_decoding(filename, k):
    """ Decompresses a Rice encoded file back into a WAV file. """
    try:
        M = 2**k
        decoded_file = wave.open(f"{filename.split('.')[0]}Dec.wav", "wb")
        write_wav_header_info(decoded_file)

        with open(filename, 'r') as input_file:
            hex_string = ""
            following_data_is_compressed = False
            for c in iter(partial(input_file.read, 1), ''):
                if c == "/":
                    if hex_string:
                        data = convert_hex_to_decoded_data(hex_string,
following_data_is_compressed, M)
                        decoded_file.writeframes(data)
                    hex_string = ""
                    following_data_is_compressed = True
                elif c == "\\":
                    if hex_string:
                        data = convert_hex_to_decoded_data(hex_string,
following_data_is_compressed, M)
                        decoded_file.writeframes(data)
                    hex_string = ""
                    following_data_is_compressed = False
                else:
                    hex_string += c
        print("Decompression complete.")
    except Exception as e:
        print("An error occurred during decompression:", e)
    finally:
        decoded_file.close()
def analyze_compression(original, encoded, decoded):
    """ Analyzes compression efficiency. """
    print("Compression Analysis:")
    print(f"Original file size: {original} bytes")
    print(f"Encoded file size: {encoded} bytes")
    print(f"Decoded file size: {decoded} bytes")
    print(f"Compression: {round((original - encoded) / original * 100, 2)}%")
rice_encoding_compression('Sound1.wav', 4)
rice_compression_decoding('Sound1_Enc.ex2', 4)
```

```
original_size = os.path.getsize("Sound1.wav")
encoded_file_size = os.path.getsize("Sound1_Enc.ex2")
decoded_file_size = os.path.getsize("Sound1_EncDec.wav")

print("Where k = 4:")
print(f"original file size: {original_size}")
print(f"encoded file size: {encoded_file_size}")
print(f"decoded file size: {decoded_file_size}")
print(f"compression: {round((original_size - encoded_file_size)/original_size * 100, 2)}%")
rice_encoding_compression('Sound1.wav', 2)
rice_compression_decoding('Sound1_Enc.ex2', 2)
original_size = os.path.getsize("Sound1.wav")
encoded_file_size = os.path.getsize("Sound1_Enc.ex2")
decoded_file_size = os.path.getsize("Sound1_EncDec.wav")

print("Where k = 2:")
print(f"original file size: {original_size}")
print(f"encoded file size: {encoded_file_size}")
print(f"decoded file size: {decoded_file_size}")
print(f"compression: {round((original_size - encoded_file_size)/original_size * 100, 2)}%")
rice_encoding_compression('Sound2.wav', 4)
rice_compression_decoding('Sound2_Enc.ex2', 4)
original_size = os.path.getsize("Sound2.wav")
encoded_file_size = os.path.getsize("Sound2_Enc.ex2")
decoded_file_size = os.path.getsize("Sound2_EncDec.wav")

print("Where k = 4:")
print(f"original file size: {original_size}")
print(f"encoded file size: {encoded_file_size}")
print(f"decoded file size: {decoded_file_size}")
print(f"compression: {round((original_size - encoded_file_size)/original_size * 100, 2)}%")
rice_encoding_compression('Sound2.wav', 2)
rice_compression_decoding('Sound2_Enc.ex2', 2)
original_size = os.path.getsize("Sound2.wav")
encoded_file_size = os.path.getsize("Sound2_Enc.ex2")
decoded_file_size = os.path.getsize("Sound2_EncDec.wav")

print("Where k = 2:")
print(f"original file size: {original_size}")
print(f"encoded file size: {encoded_file_size}")
print(f"decoded file size: {decoded_file_size}")
print(f"compression: {round((original_size - encoded_file_size)/original_size * 100, 2)}%")

from bitarray import bitarray
```

```
from tabulate import tabulate
def adaptive_k_selection(samples):
    """ Determines optimal k based on signal variance, with a cap at k = 4. """
    std_dev = np.std(samples)
    return 2 if std_dev < 32 else 4  # Maximum k capped at 4
def encode_golomb(sample, k):
    """ Applies Golomb encoding to a single sample. """
    M = 2**k
    sign_bit = '1' if sample < 0 else '0'
    s = abs(sample)
    R = bin(s & (M - 1))[2:].zfill(k)  # Extract remainder bits
    Q = s >> k  # Quotient calculation

    # Golomb encoding: quotient is now stored in binary instead of unary
    Q_bin = bin(Q + 1)[2:]  # +1 ensures nonzero encoding
    encoded_code = sign_bit + Q_bin + R
    return bitarray(encoded_code)  # Store as bitarray for efficient packing
def apply_delta_encoding(samples):
    """ Converts raw sample values into delta values. """
    delta_samples = [samples[0]]  # First sample remains the same
    for i in range(1, len(samples)):
        delta_samples.append(samples[i] - samples[i-1])
    return delta_samples
def read_wav_header(input_file):
    """ Reads and extracts WAV file header information. """
    header_info = input_file.read(44)
    header_fields = {
        "Chunk Size": struct.unpack('<L', header_info[4:8])[0],
        "Format": header_info[8:12],
        "Subchunk Size": struct.unpack('<L', header_info[16:20])[0],
        "Audio Format": struct.unpack('<H', header_info[20:22])[0],
        "Number of Channels": struct.unpack('<H', header_info[22:24])[0],
        "Sample Rate": struct.unpack('<L', header_info[24:28])[0],
        "Byte Rate": struct.unpack('<L', header_info[28:32])[0],
        "Block Align": struct.unpack('<H', header_info[32:34])[0],
        "Bits per sample": struct.unpack('<H', header_info[34:36])[0],
        "File Size": struct.unpack('<L', header_info[40:44])[0]
    }
    for key, value in header_fields.items():
        print(f"{key}: {value}")
    return header_info
def rice_encoding_compression(filename):
    """ Compresses WAV file samples using Delta + Golomb encoding with bit-packing. """
    try:
```

```
        input_file = open(filename, 'rb')
        encoded_file = open(f"{filename.split('.')[0]}_DeltaGolombPacked.bin", "wb")
        header_info = read_wav_header(input_file)

        samples = []
        for samplebyte in iter(partial(input_file.read, 2), b''):
            sample = struct.unpack('<h', samplebyte)[0]
            samples.append(sample)

        delta_samples = apply_delta_encoding(samples)  # Apply delta encoding
        k = adaptive_k_selection(delta_samples)  # Dynamically select k, capped at 4
        print(f"Using adaptive k = {k} (Delta + Golomb encoding + Bit-Packing)")

        compressed_bits = bitarray()
        compressed_bytes = 0
        uncompressed_bytes = 0
        M = 2**k

        for sample in delta_samples:
            if -128 < sample < 127:
                encoded_bits = encode_golomb(sample, k)
                compressed_bits.extend(encoded_bits)
                compressed_bytes += len(encoded_bits) // 8
            else:
                uncompressed_bytes += 2  # Each uncompressed sample is 2 bytes
                compressed_bits.frombytes(struct.pack('<h', sample))

        compressed_bits.tofile(encoded_file)
        print(f"Compressed bytes: {compressed_bytes}")
        print(f"Uncompressed bytes: {uncompressed_bytes}")

    except Exception as e:
        print("An error occurred:", e)
    finally:
        input_file.close()
        encoded_file.close()
def analyze_compression(original, encoded):
    """ Analyzes compression efficiency and prints a table. """
    data = [
        ["Original", original, "0%"],
        ["Encoded (Delta + Golomb + Bit-Packing, max 4)", encoded, f"{round((original - encoded) /
original * 100, 2)}%"]
    ]
```

```
    print("\nCompression Analysis:")
    print(tabulate(data, headers=["File Type", "File Size (bytes)", "Compression (%)"],
tablefmt="grid"))
original_size = os.path.getsize("Sound1.wav")
rice_encoding_compression('Sound1.wav')
delta_golomb_packed_size = os.path.getsize("Sound1_DeltaGolombPacked.bin")
analyze_compression(original_size, delta_golomb_packed_size)
```

Exercise 3

```
!pip install ffprobe-python ffmpeg-python
import subprocess

try:
    subprocess.run(["ffprobe", "-version"], check=True)
    print("ffprobe is working correctly!")
except FileNotFoundError:
    print("Error: ffprobe not found. Make sure it is installed and in your system's PATH.")

import os
os.environ["PATH"] += os.pathsep + "/opt/homebrew/bin"
subprocess.run(["ffprobe", "-version"], check=True)
from ffprobe import FFProbe
from os import listdir
from os.path import isfile, join
import ffmpeg
import pathlib

# current folder path
curr_path = pathlib.Path().resolve()

def find_problematic_fields(video_stream, audio_stream):
    '''
    Takes in the ffmpeg stream as input parameter which will be used to filter the video input.
    Only filter for video framerate and resolution will be performed, other field settings will be
done when
    the ffmpeg output video file is made.

    Returns problematic_fields (String), video stream and audio stream
    '''
    problematic_fields = ""
    # file format setting will be set along with output file
    if file_format != "mp4":
        problematic_fields += "file_format "
```

```
    # video codec setting will be set along with output file
    if v_codec_name != "h264":
        problematic_fields += "video_codec "

    # audio codec setting will be set along with output file
    if a_codec_name != "aac":
        problematic_fields += "audio_codec "

    if v_frame_rate != "25":
        problematic_fields += "video_frame_rate "
        # section 11.90 fps of the link provided
        video_stream = ffmpeg.filter(video_stream, 'fps', fps=25, round='near')

    # aspect ratio setting will be set along with output file
    if v_aspect_ratio != "16:9":
        problematic_fields += "aspect_ratio "

    if v_resolution != "640 x 360":
        problematic_fields += "resolution "
        video_stream = ffmpeg.filter(video_stream, 'scale', w='640', h='360')

    # video bitrate setting will be set along with output file
    if v_bitrate < 2 or v_bitrate > 5:
        problematic_fields += "video_bitrate "

    # audio bitrate setting will be set along with output file
    if a_bitrate < 0 and a_bitrate > 256:
        problematic_fields += "audio_bitrate "

    # channels will be set to stereo by default as later on, we will be using the video and audio
stream
    # to generate the output file
    if a_channel_layout != "stereo" or a_channels != 2:
        problematic_fields += "channels_layout "

    return problematic_fields, video_stream, audio_stream
# Ensure curr_path is defined before using it
if 'curr_path' not in locals():
    raise NameError("curr_path is not defined. Please set it before running this cell.")

# Ensure the directory exists
import os
if not os.path.exists(f"{curr_path}/Exercise3_Films"):
```

```
    raise FileNotFoundError(f"Directory {curr_path}/Exercise3_Films does not exist.")

# Get all files in the Exercise3_Films folder, skipping system files
files = [f for f in os.listdir(f"{curr_path}/Exercise3_Films") if
os.path.isfile(os.path.join(f"{curr_path}/Exercise3_Films", f)) and not f.startswith(".")]

# Open a text file to write analysis results
result_file = open("file_analysis_report.txt", "w")

for file in files:
    print(f"\nProcessing file: {file}")
    try:
        metadata = FFProbe(f"{curr_path}/Exercise3_Films/{file}")

        # Ensure metadata contains valid streams
        video_stream = metadata.streams[0] if len(metadata.streams) > 0 else None
        audio_stream = metadata.streams[1] if len(metadata.streams) > 1 else None

        # Assign values safely
        file_format = file.split('.')[-1]
        v_codec_name = video_stream.codec_name if video_stream else "Unknown"
        a_codec_name = audio_stream.codec_name if audio_stream else "Unknown"
        v_frame_rate = (float(video_stream.nb_frames) / float(video_stream.duration)) if
(video_stream and video_stream.nb_frames and video_stream.duration) else "Unknown"
        v_aspect_ratio = video_stream.display_aspect_ratio if video_stream else "Unknown"
        v_resolution = f"{video_stream.width} x {video_stream.height}" if video_stream else
"Unknown"
        v_bitrate = (float(video_stream.bit_rate) / 1000000) if (video_stream and
video_stream.bit_rate) else None
        a_bitrate = (float(audio_stream.bit_rate) / 1000) if (audio_stream and
audio_stream.bit_rate) else None
        a_channel_layout = audio_stream.channel_layout if audio_stream else "Unknown"
        a_channels = audio_stream.channels if audio_stream else "Unknown"

        # Display extracted information
        print(f"Video format: {file_format}")
        print(f"Video codec: {v_codec_name}")
        print(f"Audio codec: {a_codec_name}")
        print(f"Frame rate: {v_frame_rate} FPS")
        print(f"Aspect ratio: {v_aspect_ratio}")
        print(f"Resolution: {v_resolution}")
        print(f"Video bitrate: {v_bitrate if v_bitrate is not None else 'Unknown'} Mb/s")
        print(f"Audio bitrate: {a_bitrate if a_bitrate is not None else 'Unknown'} kb/s")
        print(f"Audio channel layout: {a_channel_layout}")
```

```
        print(f"Number of channels: {a_channels}")

        # Ensure find_problematic_fields is defined
        if "find_problematic_fields" not in globals():
            raise NameError("find_problematic_fields function is not defined. Please define it before
running this cell.")

        # Check and fix problematic fields
        stream = ffmpeg.input(f"{curr_path}/Exercise3_Films/{file}")
        video_stream = stream.video if hasattr(stream, 'video') else None
        audio_stream = stream.audio if hasattr(stream, 'audio') else None

        # Handle v_bitrate safely in find_problematic_fields
        problematic_fields, video_stream, audio_stream = find_problematic_fields(video_stream,
audio_stream)

        # Write findings to file
        if not problematic_fields:
            result_file.write(f"{file} - no issues found\n")
        else:
            result_file.write(f"{file} - {problematic_fields}\n")
            output_filename = f"{curr_path}/OutputFiles/{file.split('.')[0]}_formatOK.mp4"
            stream = ffmpeg.output(video_stream, audio_stream, output_filename, format='mp4',
vcodec='h264', acodec='aac', video_bitrate='2.5M', audio_bitrate='256k', aspect='16:9')
            ffmpeg.run(stream, capture_stdout=True, capture_stderr=True)

    except ffmpeg.Error as e:
        print("FFmpeg Error:")
        print("stdout:", e.stdout.decode('utf8'))
        print("stderr:", e.stderr.decode('utf8'))

# Close the result file after analysis
result_file.close()

files = [f for f in listdir(f"{curr_path}/OutputFiles/") if isfile(join(f"{curr_path}/OutputFiles/", f))]

for file in files:
    print(f"file: {file}")
    metadata=FFProbe(f"{curr_path}/OutputFiles/{file}")
    # retrieve stream information
    video_stream = metadata.streams[0]
    audio_stream = metadata.streams[1]

    # assign stream fields to variables
```

```
file_format = file.split('.')[1]
v_codec_name = video_stream.codec_name
a_codec_name = audio_stream.codec_name
v_frame_rate = float(video_stream.nb_frames) / float(video_stream.duration)
v_aspect_ratio = video_stream.display_aspect_ratio
v_resolution = f"{video_stream.width} x {video_stream.height}"
v_bitrate = int(video_stream.bit_rate) / 1000000
a_bitrate = int(audio_stream.bit_rate) / 1000
a_channel_layout = audio_stream.channel_layout
a_channels = audio_stream.channels
print(f"Video format (container): {file_format}")
print(f"Video codec: {v_codec_name}")
print(f"Audio codec: {a_codec_name}")
print(f"Frame rate: {format(v_frame_rate, '.2f')} FPS")
print(f"Aspect ratio: {v_aspect_ratio}")
print(f"Resolution: {v_resolution}")
print(f"v_bitrate: {format(v_bitrate, '.2f')}Mb/s")
print(f"a_bitrate: {format(a_bitrate, '.2f')}kb/s")
print(f"channel layout: {a_channel_layout}")
print(f"channels: {a_channels}\n")
```