Exercise 1

In Task 1, the code detects moving cars on the main street using frame differencing and background subtraction. It initializes the video, preprocesses each frame by converting it to grayscale and applying Gaussian blur, and applies MOG2 background subtraction with morphological operations to clean noise. The function *`detect_cars`* is created to detect cars, which is based on contour analysis and ensures that only large objects in the lower half of the frame are considered. Green bounding boxes highlight detected cars. The final processed frames are written to an output video, effectively visualizing car movement while filtering out irrelevant objects.

Frame differencing is implemented in the function *`apply_frame_differencing`*, which compares each current frame with the initial frame to detect motion. The initial frame is preprocessed with grayscale conversion and Gaussian blur to reduce noise. Each new frame is similarly preprocessed and compared with the initial frame. This highlights pixel-level changes, where non-zero values indicate movement. It is effective for short-term motion detection but struggles with long-term stability due to gradual lighting changes.

To address this limitation, the background subtraction technique in the MOG method is applied using the *`apply_background_subtraction`* function. It identifies foreground objects by comparing the current frame with a dynamic background model, making it more resilient to environmental changes. Morphological operations, like opening and dilation, are added to refine the mask by removing small noise and filling gaps.

Both techniques are combined to improve detection accuracy. However, some limitations remain such as misdetection of pedestrian groups as cars and overlapping detection when multiple vehicles cluster together, as observed in the output videos.

Task 2 builds on Task1 by incorporating a detection zone and centroid tracking to count cars moving from the city's downtown to the city center. The detection zone, drawn as a rotated rectangle, is positioned near the exit lane of the video frame. As each car is detected, its centroid is calculated within the *`detect_and_count_cars`* function. If the centroid falls within the detection zone, the car is flagged for counting.

To avoid duplicate counts caused by flickering, a frame timeout mechanism tracks cars across frames. If a detected centroid is near a previously tracked one within a given time, it is not counted again. New cars are counted and added to the tracking list. The `process_video` function manages the overall workflow. The processed frames, showing detected cars and the current count, are saved to an output video. Lastly, it calculates the total number of cars per minute by dividing the total count by the video duration.

The table summarizes the results:

Video	Total number of cars	Cars per minute
Traffic_Laramie_1.mp4	6	2.02
Traffic_Laramie_2.mp4	4	2.27

While the solution to count the cars heading to the city is straightforward, it might not be ideal for complex traffic scenarios. The system counted 6 cars in Traffic_Laramie_1.mp4, but manual verification revealed one miscount due to centroid placement and one missed vehicle due to overlapping detection with a larger car moving in the opposite direction. This highlights the need for more robust tracking methods to improve accuracy under challenging conditions.

Exercise 2

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

The exercise implements a lossless compression schema for audio files using Rice coding, with the rice coding steps derived from Coursera Exercise 1.7.

The code begins by reading the audio file and selecting a single channel if the file is stereo. A dynamic offset is computed within the *`encode_audio_file`* function to shift any negative sample values into a non-negative range. For each sample, the offset is added and *`rice_encode`* is applied. The sample is divided into a quotient and a remainder, where the quotient is represented in unary form and the remainder in binary with K bits. These components form a variable-length codeword, converted into bytes using the *`bits_to_bytes`* function for efficient storage.

To facilitate decoding, *`encode_audio_file`* appends a header containing metadata as the K value, sample rate, number of samples, and offset. During decoding, the *`decode_audio_file`* function reads the header, converts the bytes stream back to a bit string using *`bytes_to_bits`*, and applies *`rice_decode_stream`* to retrieve the samples. The offset is subtracted, and the reconstructed audio is saved in a WAV file.

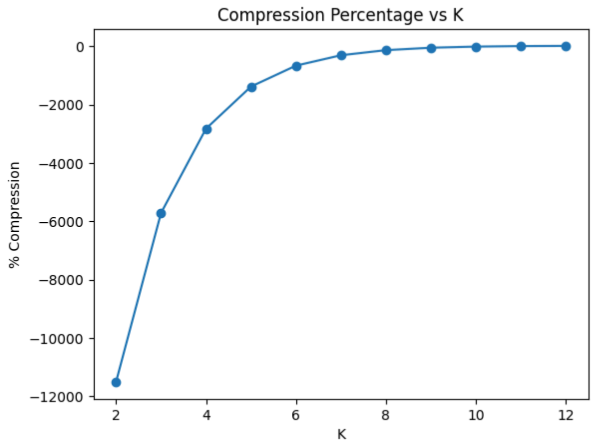
The table summarizes the results:

Audio Files	**Original Size**	**Rice (K = 4 bits)**	**Rice (K = 2 bits)**	**% Compression (K = 4 bits)**	**% Compression (K = 2 bits)**
Sound1.wav	1,002,088	29,311,736	116,277,659	-2,825.07%	-11,503.54%
Sound2.wav	1,008,044	129,644,844	517,602,560	-12761.03%	-51,247.22%

Compared to Sound1.wav, Sound2.wav experiences greater expansion, particularly at K = 2. It is likely due to its broader dynamic range resulting in larger quotients and longer unary codes. Additionally, a decoding issue at K=2 caused mismatched audio dimensions. In summary, lower K values worsen expansion by increasing quotient size, whereas higher K values allocate more bits to the remainder, shortening unary coding and reducing file growth.

```
Check audio files compressed with 2 bits:----------------------
Check Sound1.wav:
The decoded audio is identical to the original.
Check Sound2.wav:
Audio dimensions differ!
Check audio files compressed with 4 bits:----------------------
Check Sound1.wav:
The decoded audio is identical to the original.
Check Sound2.wav:
The decoded audio is identical to the original.
```

To improve compression efficiency, different K values, from 2 to 12 are tested using the *`compute_encoded_size`* function. Result shows that increasing K improves compression, plateauing at K=8, making it the optimal value for the given data.



To further reduce file size, Run-Length Encoding (RLE) is applied using *`rle_encode`* to compress the rice-coded bit stream. Rice coding often produces long sequences of repeated bits, particularly in unary quotient encoding. RLE reduces redundancy by replacing such sequences with the bit value along with its run length in the bit stream.

The *`encode_audio_file_with_rle`* function applies rice coding with K=8, followed by RLE, while *`decode_audio_file_with_rle`* reverses the process using *`rle_decode`* and *`rice_decode_stream`* to recover and retrieve the audio samples.

This combined approach significantly reduced expansion. For Sound1.wav, the compressed size decreases to 10.77MB, while Sound2.wav is reduced to 12.03 MB. This demonstrates that integrating RLE with rice coding effectively enhances the overall compression efficiency.

Audio Files	**Original Size**	**RLE + Rice (K = 8 Bits)**	**% Compression (K = 8 bits)**
Sound1.wav	1,002,088	10,774,528	-975.21%
Sound2.wav	1,008,044	12,030,458	-1093.45%

Exercise 3

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

The installation of ffmpeg and ffprobe follows Coursera Exercise 19, using a static build from 'johnvansickle.com'. It checks for pre-installation with `*shutil.which("ffmpeg")*`. If missing, the script downloads, extracts, and updates `*%env PATH*`. This adds ffmpeg and ffprobe to the system PATH, allowing direct use in Python via `*subprocess.run()*`. A final ffmpeg -version call verifies installation.

The goal of this exercise is to automate the verification and correction of film formats to meet the Narbonnes Online Film Festival's technical requirements. The core process is handled by the `film_format_checker` function, which combines the entire workflow including metadata extraction, format comparison, automatic conversion if needed, and finally, generating a TXT report that documents the verification results for each film.

Within the function, the `extract_metadata` function uses ffprobe to collect metadata from each video file, covering attributes such as container format, codec, and resolution. This metadata is then compared against the festival's required format using the `compare_format` function. Each property is evaluated and individually marked as either `Correct` or `Mismatch`, with the films overall compliance status recorded as either `Format OK` or `Format Incorrect`. For film classified as `Format Incorrect`, the `convert_video` function is called to convert it to the correct format, and generate a new copy in correct format with `_formatOK` appended to its filename.

There are three videos:

- The_Gun_and_the_Pulpit_formatOK.mp4
- Cosmos_War_of_the_Planets_formatOK.mp4
- Voyage_to_the_Planet_of_Prehistoric_Women_formatOK.mp4

are found to have audio bitrates slightly exceeding the maximum 256 kbps limit. This is a known issue caused by AAC encoding variability, during initial verification. To handle this, the `compare_format` function includes a tolerance of ±10 kbps to the audio bitrate check, allowing videos up to 266 kbps to pass. While this approach does not strictly satisfy the stated requirement, it avoids unnecessary reencoding for minor deviations.

To ensure the checking process is traceable and transparent, the `film_format_checker` function finally generates a TXT report, recording the comparison results for each film, highlighting any problematic fields in which submitted video files in incorrect format need to be modified to meet the festival's standard.

There is a brief description of the terms:

1. Video format (container): A file type that holds video and audio streams and metadata such as file name, codec information and bitrate.
2. Codec: A tool for compressing and decompressing video or audio data.
3. Frame rate: The number of frames displayed per second, measured in FPS.
4. Aspect ratio: The width-to-height ratio of the video.

5. Resolution: The number of pixels in each video frame, affecting visual quality.
6. Video Bitrate: The amount of video data processed per second, measured in Mbps, affecting video quality.
7. Audio Bitrate: The amount of audio data processed per second, measured in kbps, affecting sound quality.
8. Audio channels: The number of audio signals that are transmitted. Common configuration includes mono (1 channel) and stereo (2 channels).

Exercise 4

Three emerging computer vision applications:

1) **Deep Learning in Veterinary Diagnostics and Animal Health** [4]

Veterinary healthcare has been a bit slower to adopt AI compared to human healthcare, but things are starting to change. This review explores how deep learning (DL) technology is transforming veterinary diagnostics. Out of 422 reviewed papers, 39 of them applied DL in real veterinary practices, particularly in radiography, cytology, and health record analysis. Interestingly, radiography and cytology dominated the studies which combined accounting for over 60% of applications. One of the key findings is that DL models even outperform experienced veterinarians in diagnosing conditions in certain conditions. In addition, DL models also showed strong potential in automating cell counting and classifying in cytology slides, which could significantly enhance efficiency in pathology processes. Despite these advances, the review highlights some challenges, including lack of sufficient diversity within datasets and model transparency to allow the vets to trust AI-driven results.

2) **Smart Deep Learning-Based Self-Driving Product Delivery Car** [3]

This paper documented their development of self-driving delivery cars. The vehicle's navigation relies on OpenCV and TensorFlow, in which a vision system recognizes roads, obstacles, and delivery zones in real-time. It is basically a small-size autonomous vehicle adapted for package delivery, combining object detection using You Only Look Once (YOLO) with adaptive speed and route planning. While the model was trained with custom data, making it far more adaptable to real-world environments. Also, it offers a cost-effective prototype for future delivery services., especially for last-mile logistics.

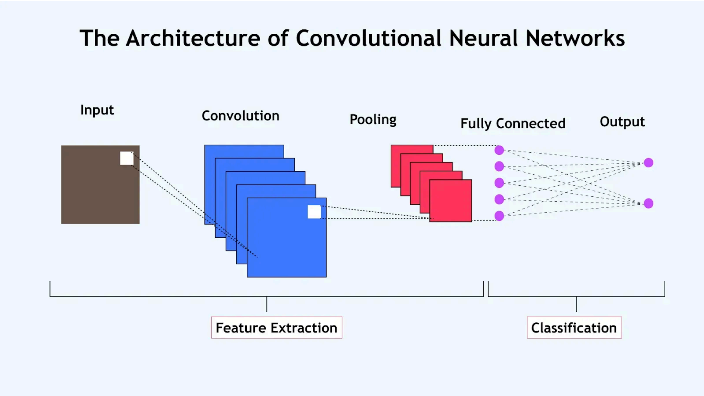
3) **Fish Feeding Behavior Recognition in Aquaculture** [5]

Monitoring fish feeding is an important but labor-intensive task in aquaculture since it requires human workers to feed fish, assess their health, and check water quality. While this paper reviews how combining computer vision, acoustics, and multi-sensor fusion can track feeding behavior and intensity in real time. By analyzing how fish move and respond during feeding, the system can spot early signs of stress or disease long before physical symptoms show up. This multi-model fusion technology, which combines camera footage with environmental sensors, acts as an early warning system, provides a complete feeding context to fish farmers so that they can adjust feeding or investigate the living environment of fish before they escalate.

Two popular computer vision techniques:

1) **Convolutional Neural Networks (CNNs)**

CNNs are the backbone of many image analysis systems. These networks are designed to automatically extract image features, removing the need to manually define features like edges, corners, or texture. CNNs achieve this by processing images through layers of convolutional filters that detect simple patterns like shapes in early layers, and more complex structures like tumors in deeper layers. The pooling layers reduce the size of feature maps, making the network more efficient while retaining important information. Finally, the extracted features are passed to fully connected layers that classify the image into categories.

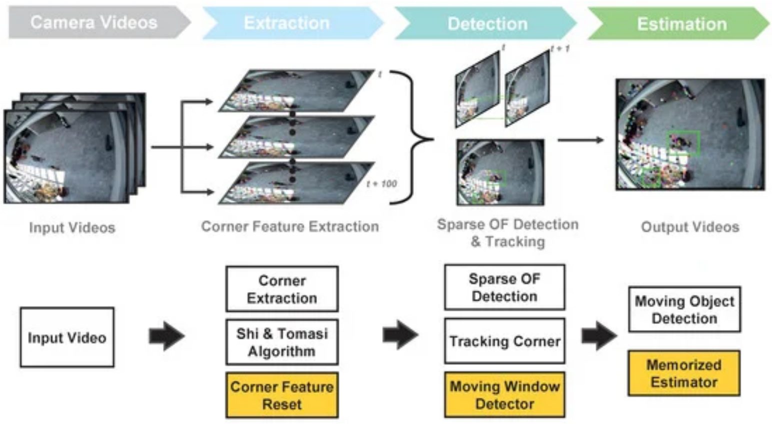


CNN diagram referenced from [2]

In the reviewed veterinary applications, CNNs have been extensively applied to radiographs and cytology slides, where they consistently outperform traditional statistical classifiers. This is because CNNs know how to leverage spatial hierarchies, meaning that a specific pattern at one location might differ in meaning when it appears elsewhere. This adaptive learning capability allows CNNs to optimize feature extraction for specific diseases, making them particularly effective in automating veterinary diagnostic processes. As CNN models continue to evolve, their role in improving diagnostic accuracy and efficiency in veterinary care is likely to expand even further.

2) **Optical Flow**

Optical flow is a widely used computer vision technique for estimating pixel-level motion between consecutive video frames. The process typically starts with corner feature extraction, where key corners are identified using the Shi and Tomasi algorithm. In the detection stage, optical flow calculates a motion vector for each corner point, representing how it moves between frames. These motion vectors capture the apparent motion caused either by camera movement or by object motion within the scene. During the tracking stage, a moving window detector is often used to follow these features over time, helping the system distinguish stationary backgrounds from moving objects.



Optical Flow diagram referenced from [1]

In the reviewed study focused on aquaculture, optical flow is used to track fish movement patterns during feeding sessions. By monitoring how individual fish move and interact, the system can estimate feeding intensity, and detect abnormal movement behaviors that might indicate disease or poor water quality. It works particularly well in clear water environments with good contrast, but performance tends to degrade in murky or visually noisy conditions. This is why modern systems, like the reviewed one, combine optical flow with deep learning models and acoustic sensors to create multi-modal behavior monitoring platforms to provide a more reliable view of fish activity.

Two examples of how combining computer vision and audio processing can address real-life problems:

1) **Driver fatigue and stress are leading contributors to road accidents.** To address this, a multi-modal monitoring system can be developed for vehicles that combines facial expression analysis using CNNs, which tracks blinking frequency, yawns, and facial tension, with Optical Flow to detect head tilts or slumps indicating drowsiness. Additionally, speech patterns and irregular breathing can be monitored through the vehicle's built-in microphones, identifying irregular respiration linked to stress of fatigue. This combination would trigger alerts such as voice warning or seat vibration in order to suggest the driver take a break before fatigue escalated into a dangerous situation.

2) **Most major music streaming platforms lack emotion-aware recommendation systems, limiting their ability to offer truly personalized experiences.** To enhance this, a system can be developed that combines facial expression recognition using CNNs to identify the user's current emotional state, with audio sensors monitoring breathing patterns through the device's microphone. For example, faster and shallower breathing may indicate stress or excitement. This combination of visual and physiological signals could dynamically adjust music recommendations, offering calm tracks when stress is detected, or upbeat songs when positive emotions are identified, creating an emotionally-resonant listening experience to the users.

(2487 words)

Reference

[1] Hosik Choi, Byungmun Kang, and DaeEun Kim. 2022. Moving Object Tracking Based on Sparse Optical Flow with Moving Window and Target Estimator. *Sensors* 22, 8 (April 2022), 2878. https://doi.org/10.3390/s22082878

[2] Mk Gurucharan. 2025. Basic CNN Architecture: A detailed explanation of the 5 layers in convolutional neural networks. *upGrad Blog*. Retrieved from https://www.upgrad.com/blog/basic-cnn-architecture/

[3] Mohammed A. Saeedi, Ahmed H. Alhindi, and Mohammed A. Kamel. 2024. *A Smart Deep Learning Based Self Driving Product Delivery Car*. https://doi.org/10.1109/airc61399.2024.10672057

[4] Sam Xiao, Zhiyong Wang, Kun Hu, Navneet K Dhand, Peter Campbell Thomson, John House, and Mehar S Khatkar. 2025. Review of applications of deep learning in veterinary diagnostic and animal health. *Frontiers in Veterinary Science* 12, (2025). Retrieved from https://www.frontiersin.org/journals/veterinary-science/articles/10.3389/fvets.2025.1511522/abstract

[5] Shulong Zhang, Daoliang Li, Jiayin Zhao, Mingyuan Yao, Yingyi Chen, Yukang Huo, Xiao Liu, and Haihua Wang. 2025. Research advances on fish feeding behavior recognition and intensity quantification methods in aquaculture. *arXiv.org*. Retrieved from https://arxiv.org/abs/2502.15311

Appendix - Exercise Code

Exercise 1

```
"""Exercise1.ipynb
## Task1: Car Detection on Main Street
Import Libraries
"""
!pip install opencv-python-headless
!pip install pyttsx3
!pip install numpy
!pip install opencv-python
!pip install pypiwin32
```

```
import cv2
import numpy as np

"""Create functions to import the provided videos and output path settings for saving the processed
video"""

def initialize_video(video_path):
    """Initialize video capture and retrieve video properties."""
    video = cv2.VideoCapture(video_path)
    if not video.isOpened():
        raise ValueError(f"Error: Cannot open video at {video_path}")

    fps = int(video.get(cv2.CAP_PROP_FPS))
    frame_width = int(video.get(cv2.CAP_PROP_FRAME_WIDTH))
    frame_height = int(video.get(cv2.CAP_PROP_FRAME_HEIGHT))
    total_frames = int(video.get(cv2.CAP_PROP_FRAME_COUNT))

    print(f"Video loaded: {video_path}")
    print(f"FPS: {fps}, Width: {frame_width}, Height: {frame_height}, Total Frames: {total_frames}")

    return video, fps, frame_width, frame_height, total_frames

def initialize_video_writer(output_path, fps, frame_width, frame_height):
    """Initialize video writer for output video."""
    fourcc = cv2.VideoWriter_fourcc(*"mp4v")
    output_video = cv2.VideoWriter(output_path, fourcc, fps, (frame_width, frame_height))

    if not output_video.isOpened():
        raise ValueError(f"Error: Cannot create output video at `{output_path}`")

    print(f"Output video writer initialized at `{output_path}`")
    return output_video

# Initialize video and output writer
video1_path = "/content/Traffic_Laramie_1.mp4"
output1_path = "/content/Traffic_Detection_Output.mp4"
video, fps, frame_width, frame_height, total_frames = initialize_video(video1_path)
output_video = initialize_video_writer(output1_path, fps, frame_width, frame_height)

"""Create functions for **Frame Preprocessing** and **Background Subtraction** (MOG2) for car
detection and tracking"""

# Function to preprocess each frame
def preprocess_frame(frame):
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    blurred = cv2.GaussianBlur(gray, (5, 5), 0)
    return blurred
```

```
# Function to apply frame differencing
def apply_frame_differencing(current_frame, initial_frame):
   frame_diff = cv2.absdiff(initial_frame, current_frame)
   _, thresh = cv2.threshold(frame_diff, 25, 255, cv2.THRESH_BINARY)
   return thresh


# Initialize MOG2 background subtractor
bg_subtractor = cv2.createBackgroundSubtractorMOG2(history=500, varThreshold=16,
detectShadows=True)
# Function to apply MOG background subtractor
def apply_background_subtraction(frame):
   fg_mask = bg_subtractor.apply(frame)
   # Morphological operations
   kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3, 3))
   cleaned_mask = cv2.morphologyEx(fg_mask, cv2.MORPH_OPEN, kernel)
   dilated_mask = cv2.dilate(cleaned_mask, kernel, iterations=2)
   return dilated_mask


# Function to detect and draw green bounding boxes around detected cars on the main street
def detect_cars(frame, mask):
   contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

   # Gets the height of the video frame
   height = int(video.get(cv2.CAP_PROP_FRAME_HEIGHT))

   # Dynamic threshold based on frame size
   MIN_CONTOUR_AREA = (frame_width * frame_height) * 0.001

   # Loop through each detected contour
   for contour in contours:
     # Focus on big objects like cars
      if cv2.contourArea(contour) < MIN_CONTOUR_AREA:
         continue

      # Get the bounding box coordinates
      x, y, w, h = cv2.boundingRect(contour)

      # Focus only on the bottom half (main street area)
      if y > height / 2:
        # Green box for main street cars
        cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2)

   return frame
```

```
"""Preprocessing the video to detect cars on main street"""

# Read the initial frame for frame differencing
ret, initial_frame = video.read()
if not ret:
    print("Error: Could not read the initial frame.")
    video.release()
    exit(1)

# Preprocess the initial frame (grayscale + blur)
initial_gray = cv2.cvtColor(initial_frame, cv2.COLOR_BGR2GRAY)
initial_blur = cv2.GaussianBlur(initial_gray, (5, 5), 0)
print("Initial frame read successfully.")


print("Starting video processing...")
while True:
    ret, frame = video.read()
    if not ret:
        break

    # Preprocess the current frame
    preprocessed_frame = preprocess_frame(frame)

    # Frame differencing
    frame_diff = apply_frame_differencing(preprocessed_frame, initial_blur)

    # Background subtraction (MOG2)
    foreground_mask = apply_background_subtraction(preprocessed_frame)

    # Combine frame differencing and MOG2 masks
    combined_mask = cv2.bitwise_or(frame_diff, foreground_mask)

    # Detect the cars on mainstreet and draw bounding boxes around detected cars
    processed_frame = detect_cars(frame, combined_mask)

    # Write processed frame to output video
    output_video.write(processed_frame)

    # Press 'q' to exit early
    if cv2.waitKey(30) & 0xFF == ord('q'):
        break

print("The car detection is done!")

# Release resources
video.release()
output_video.release()
```

```
cv2.destroyAllWindows()

"""## Task2: Car Counting Towards City Center"""

# Function to draw the detection zone to define the counting area
def draw_detection_zone(frame, rect_center, rect_size, angle):
    """Draw the rotated detection zone on the video frame."""
    # Create the rotated rectangle
    rotated_rect = (rect_center, rect_size, angle)
    box = cv2.boxPoints(rotated_rect).astype(int)

    # Draw the rotated detection zone
    cv2.polylines(frame, [box], isClosed=True, color=(255, 255, 0), thickness=2)
    cv2.putText(frame, "Detection Zone", (rect_center[0] - 40, rect_center[1] - 60),
                cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 0), 1)

    return box

# Function to detect and count cars based on motion within the detection zone
def detect_and_count_cars(frame, mask, detection_zone_box):
    """Detect and count cars using contours and centroid tracking."""
    global tracked_cars, car_count
    contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    detected_centroids = []

    # Gets the height of the video frame
    height = int(video.get(cv2.CAP_PROP_FRAME_HEIGHT))

    # Loop through each detected contour
    for contour in contours:
        # Focus on big objects like cars
        if cv2.contourArea(contour) < 2500:
            continue

        # Get the bounding box adn centroid coordinates
        x, y, w, h = cv2.boundingRect(contour)
        cX, cY = x + w // 2, y + h // 2

        # Focus only on the bottom half (main street area)
        # Draw bounding box and centroid
        if y > height / 2:
          cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2)
          cv2.circle(frame, (cX, cY), 5, (0, 0, 255), -1)

        # Check if the centroid is inside the detection zone
        if cv2.pointPolygonTest(detection_zone_box, (cX, cY), False) >= 0:
            detected_centroids.append((cX, cY))
```

```
    # Update tracked cars and avoid duplicate counting
    new_tracked_cars = []
    for cX, cY in detected_centroids:
        found = False
        for car_x, car_y, frames_left in tracked_cars:
            if abs(cX - car_x) < 20 and abs(cY - car_y) < 20:
                new_tracked_cars.append((cX, cY, FRAME_TIMEOUT))
                found = True
                break

        if not found:
            car_count += 1
            print(f"Car counted! Total count: {car_count}")
            new_tracked_cars.append((cX, cY, FRAME_TIMEOUT))

    # Remove expired tracked cars
    tracked_cars = [(x, y, frames_left - 1) for x, y, frames_left in tracked_cars if frames_left > 0]
    tracked_cars.extend(new_tracked_cars)

    # Display car count
    cv2.putText(frame, f"Cars to City: {car_count}", (10, 40),
                cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 0), 2)

    return frame

def process_video(video, output_video, fps, total_frames):
    """Process video frames, detect cars, and calculate cars per minute."""
    # Read the initial frame for frame differencing
    ret, initial_frame = video.read()
    if not ret:
        print("Error: Could not read the initial frame.")
        video.release()
        exit(1)

    # Preprocess the initial frame (grayscale + blur)
    initial_gray = cv2.cvtColor(initial_frame, cv2.COLOR_BGR2GRAY)
    initial_blur = cv2.GaussianBlur(initial_gray, (5, 5), 0)
    print("Initial frame read successfully.")

    global car_count, tracked_cars, FRAME_TIMEOUT
    # Car tracking variables
    car_count = 0
    tracked_cars = []
    FRAME_TIMEOUT = 30

    # Detection zone parameters
    rect_center = (60, int(frame_height / 2 + 50))
```

```
rect_size = (70, 85)
angle = 10


print("Starting video processing...")
while True:
    ret, frame = video.read()
    if not ret:
        break

    # Preprocess current frame
    preprocessed_frame = preprocess_frame(frame)

    # Apply frame differencing and background subtraction
    frame_diff = apply_frame_differencing(preprocessed_frame, initial_blur)
    foreground_mask = apply_background_subtraction(preprocessed_frame)
    combined_mask = cv2.bitwise_or(frame_diff, foreground_mask)

    # Draw detection zone and get its coordinates
    detection_zone_box = draw_detection_zone(frame, rect_center, rect_size, angle)

    # Detect and count cars
    processed_frame = detect_and_count_cars(frame, combined_mask, detection_zone_box)

    # Write processed frame to output video
    output_video.write(processed_frame)

    # Exit early if 'q' is pressed
    if cv2.waitKey(30) & 0xFF == ord('q'):
        break

print("Video processing complete....")

# Calculate cars per minute
video_duration_minutes = (total_frames / fps) / 60
cars_per_minute = car_count / video_duration_minutes if video_duration_minutes > 0 else 0.0

# Final output
print(f"\n----- Final Results -----")
print(f"Total cars counted passing to the city center: {car_count}")
print(f"Cars per minute: {cars_per_minute:.2f}")
print("Video processing complete.")

# Release resources
video.release()
output_video.release()
cv2.destroyAllWindows()
```

```
# Initialize video and output writer for Video 1
video_1_path = "/content/Traffic_Laramie_1.mp4"
output_1_path = "/content/Traffic_1_Counting_Output.mp4"
video_1, fps_1, frame_width_1, frame_height_1, total_frames_1 = initialize_video(video_1_path)
output_video_1 = initialize_video_writer(output_1_path, fps_1, frame_width_1, frame_height_1)

# Process video 1
process_video(video_1, output_video_1, fps_1, total_frames_1)

# Initialize video and output writer for Video 2
video_2_path = "/content/Traffic_Laramie_2.mp4"
output_2_path = "/content/Traffic_2_Counting_Output.mp4"
video_2, fps_2, frame_width_2, frame_height_2, total_frames_2 = initialize_video(video_2_path)
output_video_2 = initialize_video_writer(output_2_path, fps_2, frame_width_2, frame_height_2)

# Process video 2
process_video(video_2, output_video_2, fps_2, total_frames_2)
```

Exercise 2

```
"""Exercise2.ipynb
Import libraries
"""
import scipy.io.wavfile as wav
import numpy as np
import os

"""Inspect the WAV file sample data"""

sample_rate, data = wav.read("Sound1.wav")
print("Sound1.wav file metadata:")
print("Data type:", data.dtype)
print("Min value:", data.min())
print("Max value:", data.max())

sample_rate, data = wav.read("Sound2.wav")
print("Sound2.wav file metadata:")
print("Data type:", data.dtype)
print("Min value:", data.min())
print("Max value:", data.max())

"""**Rice encoding and decoding functions**"""

# Function for Rice encode an integer S using parameter K.
def rice_encode(S, K):
    """
```

```
    Steps from Coursera Exercise 1.7:
      1. Compute M = 2^K.
      2. Calculate quotient q = S // M and remainder r = S % M.
      3. Encode q in unary: q ones followed by a '0'.
      4. Encode r in binary with fixed width K.
      5. Concatenate the unary and binary codes.

    Args:
      S (int): The integer value to be encoded.
      K (int): The fixed bit-length for the remainder.

    Returns:
      str: The Rice-coded bit string.
    """
    M = 2 ** K
    q = S // M
    r = S % M

    # Unary encoding: q ones followed by a terminating 0
    quotient_code = '1' * q + '0'

    # Remainder code: r as a binary string, padded to K bits.
    remainder_code = format(r, f'0{K}b')

    # The codeword is the concatenation of the quotient code and the remainder code.
    codeword = quotient_code + remainder_code
    return codeword


# Function for Rice decode a given bit string using parameter K.
def rice_decode(codeword, K):
    """
    Steps from Coursera Exercise 1.7:
      1. Count the number of 1s until the first 0. That count is q.
      2. Read the next K bits as a binary number, which is the remainder r.
      3. Calculate the original number as S = q * 2^K + r.

    Args:
      codeword (str): The Rice-coded bit string.
      K (int): The fixed bit-length used during encoding.

    Returns:
      int: The decoded integer S.
    """
    # Count the number of 1s before the first 0 (the unary part).
    q = 0
    index = 0
    while index < len(codeword) and codeword[index] == '1':
```

```
        q += 1
        index += 1

    # The first 0 marks the end of the unary code.
    # Skip the 0
    index += 1

    # Read the next K bits for the remainder.
    remainder_bits = codeword[index:index + K]
    r = int(remainder_bits, 2) if remainder_bits else 0

    # Calculate S using the formula
    M = 2 ** K
    S = q * M + r
    return S

"""test the encode and decode function

"""

K = 4
assert (
    rice_decode(rice_encode(23, K), K) == 23
    and rice_decode("1100011", K) == 35
), "Rice coding test failed!"

print("All tests passed!")

"""Helper functions"""

# Function to convert a bit string to a bytes object
def bits_to_bytes(bit_string):
    # Pad the bit string if necessary to ensure its length is a multiple of 8.
    extra_bits = len(bit_string) % 8
    if extra_bits:
        padding = 8 - extra_bits
        bit_string += '0' * padding

    # Convert the padded bit string to an integer, then to a bytes object.
    num_bytes = len(bit_string) // 8
    return int(bit_string, 2).to_bytes(num_bytes, byteorder='big')


# Function to convert a bytes object to its corresponding bit string
def bytes_to_bits(byte_data):
    return ''.join(format(b, '08b') for b in byte_data)

# Calculate the compression percentage given the size of the original and compressed data
```

```
def compression_percentage(original_size, compressed_size):
    if original_size == 0:
        return 0.0  # Avoid division by zero.
    comp_percent = ((original_size - compressed_size) / original_size) * 100
    return comp_percent

# Function to decode a continuous bit string into a list of samples.
def rice_decode_stream(bit_string, K):
    """
    Returns: A list of decoded integer samples.
    """
    decoded_samples = []
    index = 0
    total_length = len(bit_string)

    while index < total_length:
        # Count ones until the first 0 to determine the quotient q
        q = 0
        while index < total_length and bit_string[index] == '1':
            q += 1
            index += 1

        # If reaching the end without finding a 0, break
        if index >= total_length:
            break

        # Skip the 0 that terminates the unary code
        index += 1

        # Extract the next K bits for the remainder
        if index + K > total_length:
            break
        remainder_bits = bit_string[index:index + K]
        r = int(remainder_bits, 2)
        index += K

        sample = q * (2 ** K) + r
        decoded_samples.append(sample)

    return decoded_samples

"""Encode and Decode audio files"""

# Encode a WAV audio file using rice coding
def encode_audio_file(input_wav_path, output_ex2_path, K):
    """
    Steps:
      1. Read the WAV file.
```

```
  2. Compute offset = -min_value if min_value < 0 (else 0).
  3. Add offset to each sample to make them non-negative.
  4. Rice encode each adjusted sample.
  5. Convert the concatenated bit string to bytes.
  6. Create a header with K, sample rate, num_samples, and offset.
  7. Write header and binary data to the output file.
"""
# Read the audio file.
sample_rate, data = wav.read(input_wav_path)

# If stereo or multi-channel, use one channel (the first channel)
if len(data.shape) > 1:
    data = data[:, 0]

# Ensure data is a 1D integer array.
data = data.flatten()

# Compute dynamic offset: shift data so that the minimum becomes 0.
min_value = int(data.min())
offset = -min_value if min_value < 0 else 0

# Encode each sample after adding the offset.
encoded_bits = ""
for sample in data:
    adjusted_sample = int(sample) + offset
    encoded_bits += rice_encode(adjusted_sample, K)

# Convert the bit string to bytes.
encoded_bytes = bits_to_bytes(encoded_bits)

# Create a header with metadata including the offset.
header_lines = [
    f"K={K}",
    f"sample_rate={sample_rate}",
    f"num_samples={len(data)}",
    f"offset={offset}",
    "----"  # Delimiter between header and binary data.
]
header_str = "\n".join(header_lines) + "\n"
header_bytes = header_str.encode('utf-8')

# Write the header and encoded bytes to the output file.
with open(output_ex2_path, "wb") as f:
    f.write(header_bytes)
    f.write(encoded_bytes)

print("Encoding complete.")
original_size = os.path.getsize(input_wav_path)
```

```
    compressed_size = os.path.getsize(output_ex2_path)
    print("Original file size (bytes):", original_size)
    print("Compressed file size (bytes):", compressed_size)
    print("Compression Percentage: {:.2f}%".format(compression_percentage(original_size,
compressed_size)))

# Function to decode a Rice-coded file back to a WAV audio file
def decode_audio_file(input_ex2_path, output_wav_path):
    """
    Steps:
      1. Read the encoded file and split the header from the binary data.
      2. Parse the header to extract K, sample rate, num_samples, and offset.
      3. Convert the binary data (bytes) back to a bit string.
      4. Rice decode the bit string to retrieve adjusted samples.
      5. Subtract the stored offset from each sample to recover the original signed value.
      6. Write the reconstructed audio data to a new WAV file.
    """
    with open(input_ex2_path, "rb") as f:
        file_content = f.read()

    # Split header and binary data.
    header_str, encoded_bytes = file_content.split(b"----\n", 1)
    header_lines = header_str.decode('utf-8').strip().split("\n")
    header_info = {}
    for line in header_lines:
        key, value = line.split("=")
        header_info[key.strip()] = value.strip()

    # Extract metadata from header.
    K = int(header_info.get("K"))
    sample_rate = int(header_info.get("sample_rate"))
    num_samples = int(header_info.get("num_samples"))
    offset = int(header_info.get("offset"))

    # Convert the encoded bytes back to a bit string.
    encoded_bits = bytes_to_bits(encoded_bytes)

    # Decode the bit stream to retrieve the adjusted (non-negative) samples.
    decoded_samples = rice_decode_stream(encoded_bits, K)

    # Ensure the decoded sample list has the correct length.
    if len(decoded_samples) > num_samples:
        decoded_samples = decoded_samples[:num_samples]
    elif len(decoded_samples) < num_samples:
        decoded_samples.extend([0] * (num_samples - len(decoded_samples)))

    # Subtract the offset to revert to the original signed values.
    original_samples = [s - offset for s in decoded_samples]
```

```
    # Convert to a NumPy array with appropriate data type.
    decoded_array = np.array(original_samples, dtype=np.int16)

    # Write the reconstructed audio data to a WAV file.
    wav.write(output_wav_path, sample_rate, decoded_array)

    print("Decoding complete. Output written to", output_wav_path)

"""**Preprocessing the audio files with rice encoding**

**When k = 2 bits**
"""

K = 2

# Encode a first WAV file.
input_wav = "Sound1.wav"
output_ex2 = "Sound1_k2_Enc.ex2"
encode_audio_file(input_wav, output_ex2, K)

# Decode the encoded file back to WAV.
output_wav = "Sound1_k2_Enc_Dec.wav"
decode_audio_file(output_ex2, output_wav)

# Encode a second WAV file.
input_wav = "Sound2.wav"
output_ex2 = "Sound2_k2_Enc.ex2"
encode_audio_file(input_wav, output_ex2, K)

# Decode the encoded file back to WAV.
output_wav = "Sound2_k2_Enc_Dec.wav"
decode_audio_file(output_ex2, output_wav)

"""**When k = 4 bits**"""

K = 4

# Encode a first WAV file.
input_wav = "Sound1.wav"
output_ex2 = "Sound1_k4_Enc.ex2"
encode_audio_file(input_wav, output_ex2, K)

# Decode the encoded file back to WAV.
output_wav = "Sound1_k4_Enc_Dec.wav"
decode_audio_file(output_ex2, output_wav)

# Encode a second WAV file.
```

```python
input_wav = "Sound2.wav"
output_ex2 = "Sound2_k4_Enc.ex2"
encode_audio_file(input_wav, output_ex2, K)

# Decode the encoded file back to WAV.
output_wav = "Sound2_k4_Enc_Dec.wav"
decode_audio_file(output_ex2, output_wav)

"""Check whether the original and decoded audio files are identical"""

def compare_audio_files(original_wav, decoded_wav):
    # Read both audio files.
    sample_rate_orig, data_orig = wav.read(original_wav)
    sample_rate_dec, data_dec = wav.read(decoded_wav)

    # Compare sample rates.
    if sample_rate_orig != sample_rate_dec:
        print("Sample rates differ!")
        return False

    # Compare array shapes (number of samples/channels).
    if data_orig.shape != data_dec.shape:
        print("Audio dimensions differ!")
        return False

    # Compare the audio data.
    if np.array_equal(data_orig, data_dec):
        print("The decoded audio is identical to the original.")
        return True
    else:
        print("The decoded audio differs from the original.")
        # Optionally, compute the difference:
        diff = np.abs(data_orig - data_dec)
        print("Max difference:", np.max(diff))
        return False


if __name__ == "__main__":
    sound1_original_file = "Sound1.wav"
    sound1_k4_decoded_file = "Sound1_k4_Enc_Dec.wav"
    sound1_k2_decoded_file = "Sound1_k2_Enc_Dec.wav"
    sound2_original_file = "Sound2.wav"
    sound2_k4_decoded_file = "Sound2_k4_Enc_Dec.wav"
    sound2_k2_decoded_file = "Sound1_k2_Enc_Dec.wav"

    print("Check audio files compressed with 2 bits:-----------------------")
    print("Check Sound1.wav:")
    compare_audio_files(sound1_original_file, sound1_k2_decoded_file)
```

```
    print("Check Sound2.wav:")
    compare_audio_files(sound2_original_file, sound2_k2_decoded_file)

    print("Check audio files compressed with 4 bits:-----------------------")
    print("Check Sound1.wav:")
    compare_audio_files(sound1_original_file, sound1_k4_decoded_file)
    print("Check Sound2.wav:")
    compare_audio_files(sound2_original_file, sound2_k4_decoded_file)

"""**Further Implementation: Rice coding + Run-length encoding (RLE)**

Explore how K param affects the compression ratio
"""

import matplotlib.pyplot as plt
from tqdm import tqdm

# Function to compute the size of the Rice-coded output for a WAV file
# and parameter K, without writing the output to disk.
def compute_encoded_size(input_wav_path, K):
    """
    Steps:
      1. Read the WAV file and extract one channel of data.
      2. Compute a dynamic offset to convert signed samples to non-negative.
      3. Rice encode each sample (after applying the offset).
      4. Convert the resulting bit string to bytes.
      5. Return the byte length.
    """
    # Read audio data
    sample_rate, data = wav.read(input_wav_path)
    if len(data.shape) > 1:
        data = data[:, 0]
    data = data.flatten()

    # Compute dynamic offset
    min_value = int(data.min())
    offset = -min_value if min_value < 0 else 0

    # Rice encode each sample (after applying the offset)
    encoded_bits = ""
    for sample in data:
        adjusted_sample = int(sample) + offset
        encoded_bits += rice_encode(adjusted_sample, K)

    # Convert the bit string to bytes
    encoded_bytes = bits_to_bytes(encoded_bits)
    return len(encoded_bytes)
```

```
# Define the range of K values (from 2 to 12)
Ks = range(2, 13)
results = []

# Get the original file size
original_size = os.path.getsize("Sound1.wav")

# Loop over K values and compute the compression percentage
for k in tqdm(Ks):
    encoded_size = compute_encoded_size("Sound1.wav", k)
    comp_percentage = (1 - encoded_size / original_size) * 100
    results.append([k, comp_percentage])

results = np.array(results)

# Plotting the results
plt.title('Compression Percentage vs K')
plt.xlabel('K')
plt.ylabel('% Compression')
plt.plot(results[:, 0], results[:, 1], marker='o')
plt.show()

"""Based on your plot, it appears that as K increases from 2 up to 8, the compression percentage
improves, but beyond K = 8 the gains plateau. Therefore, **K=8** is an optimal choice for rice
coding in terms of compression ratio."""

# Function to encode a bit string using RLE
# The encoding format for each run is "bit:count" with runs separated by commas.
# Example: "0001110000" becomes "0:3,1:3,0:4".
def rle_encode(bit_string):
    """
    Args: bit_string (str): A string containing only 0s and 1s.

    Returns: str: The RLE-encoded string.
    """
    if not bit_string:
        return ""

    encoded_runs = []
    current_bit = bit_string[0]
    count = 1

    for bit in bit_string[1:]:
        if bit == current_bit:
            count += 1
        else:
            encoded_runs.append(f"{current_bit}:{count}")
            current_bit = bit
```

```
        count = 1
    encoded_runs.append(f"{current_bit}:{count}")

    return ",".join(encoded_runs)

# Function to decode an RLE-encoded bit string back to the original bit string.
def rle_decode(rle_string):
    """
    Args: rle_string (str): The RLE-encoded string.

    Returns: str: The decoded bit string.
    """
    if not rle_string:
        return ""

    decoded_bits = []
    runs = rle_string.split(",")
    for run in runs:
        bit, count_str = run.split(":")
        decoded_bits.append(bit * int(count_str))
    return "".join(decoded_bits)

"""test the encode and decode function"""

assert rle_decode(rle_encode("1110001100")) == "1110001100"
print("Test passed!")

# Function to encode a WAV file using rice + rle
def encode_audio_file_with_rle(input_wav_path, output_ex2_path, K):
    """
    Steps:
      1. Read the WAV file, select one channel if necessary, and flatten the data.
      2. Compute a dynamic offset so that the minimum sample becomes zero.
      3. For each sample, add the offset and apply Rice encoding (with parameter K).
      4. Concatenate all codewords into one long Rice-coded bit stream.
      5. Apply RLE encoding to the Rice-coded bit stream.
      6. Create a header containing metadata: K, sample_rate, num_samples, offset, and an RLE flag.
      7. Write the header and the RLE-encoded data (as text) to the output file.
    """
    # Read the WAV file.
    sample_rate, data = wav.read(input_wav_path)
    if len(data.shape) > 1:
        data = data[:, 0]  # Use the first channel if stereo.
    data = data.flatten()

    # Compute dynamic offset so that all samples are non-negative.
    min_value = int(data.min())
    offset = -min_value if min_value < 0 else 0
```

```
    # Rice encode each sample (after applying the offset).
    rice_encoded_bits = ""
    for sample in data:
        adjusted_sample = int(sample) + offset
        rice_encoded_bits += rice_encode(adjusted_sample, K)

    # Now apply RLE encoding to the Rice-coded bit stream.
    rle_encoded_string = rle_encode(rice_encoded_bits)

    # Create a header with metadata.
    header_lines = [
        f"K={K}",
        f"sample_rate={sample_rate}",
        f"num_samples={len(data)}",
        f"offset={offset}",
        f"RLE=1",  # Flag indicating RLE was applied.
        "----"     # Delimiter between header and data.
    ]
    header_str = "\n".join(header_lines) + "\n"

    # Write header and RLE-encoded string to output file.
    with open(output_ex2_path, "w") as f:
        f.write(header_str)
        f.write(rle_encoded_string)

    print("Encoding with RLE complete.")
    original_size = os.path.getsize(input_wav_path)
    compressed_size = os.path.getsize(output_ex2_path)
    print("Original file size (bytes):", original_size)
    print("Compressed file size (bytes):", compressed_size)
    print("Compression Percentage: {:.2f}%".format(compression_percentage(original_size,
compressed_size)))


# Function to decode rice + rle compressed file back to a WAV file
def decode_audio_file_with_rle(input_ex2_path, output_wav_path):
    """
    Steps:
      1. Read the encoded file and split the header from the RLE-encoded data.
      2. Parse the header to extract metadata: K, sample_rate, num_samples, and offset.
      3. RLE decode the encoded string to recover the original Rice-coded bit stream.
      4. Rice decode the bit stream (using the optimal K) to retrieve the adjusted samples.
      5. Subtract the stored offset from each sample to recover the original signed values.
      6. Write the reconstructed samples to a new WAV file using the original sample rate.
    """
    with open(input_ex2_path, "r") as f:
        content = f.read()
```

```
    # Split the header from the RLE-encoded data using the delimiter.
    header_str, rle_encoded_string = content.split("----\n", 1)
    header_lines = header_str.strip().split("\n")
    header_info = {}
    for line in header_lines:
        key, value = line.split("=")
        header_info[key.strip()] = value.strip()

    # Retrieve metadata from the header.
    K = int(header_info.get("K"))
    sample_rate = int(header_info.get("sample_rate"))
    num_samples = int(header_info.get("num_samples"))
    offset = int(header_info.get("offset"))

    # RLE decode to recover the Rice-coded bit stream.
    rice_encoded_bits = rle_decode(rle_encoded_string)

    # Rice decode the bit stream to retrieve the adjusted samples.
    # Pass num_samples as the third argument.
    decoded_samples = rice_decode_stream(rice_encoded_bits, K, num_samples)

    # Adjust the number of samples if needed.
    if len(decoded_samples) > num_samples:
        decoded_samples = decoded_samples[:num_samples]
    elif len(decoded_samples) < num_samples:
        decoded_samples.extend([0] * (num_samples - len(decoded_samples)))

    # Subtract the offset to recover original signed sample values.
    original_samples = [s - offset for s in decoded_samples]

    # Convert list of samples to a NumPy array with appropriate type.
    decoded_array = np.array(original_samples, dtype=np.int16)

    # Write the reconstructed audio data to a new WAV file.
    wav.write(output_wav_path, sample_rate, decoded_array)

    print("Decoding with RLE complete. Output written to", output_wav_path)

"""Rice encoding + RLE"""

K = 8 # Optimal Rice coding parameter

# Encode the first audio file
input_wav = "Sound1.wav"
output_ex2 = "Sound1_Enc_RLE.ex2"
encode_audio_file_with_rle(input_wav, output_ex2, K)
```

```
# Decode the encoded file back to a WAV.
decoded_wav = "Sound1_Enc_RLE_Dec.wav"
decode_audio_file_with_rle(output_ex2, decoded_wav)

# Encode the second audio file
input_wav = "Sound2.wav"
output_ex2 = "Sound2_Enc_RLE.ex2"
encode_audio_file_with_rle(input_wav, output_ex2, K)

# Decode the encoded file back to a WAV.
decoded_wav = "Sound2_Enc_RLE_Dec.wav"
decode_audio_file_with_rle(output_ex2, decoded_wav)
```

Exercise 3

```
"""Exercise 3
Installing ffmpeg and ffprobe
"""
# Commented out IPython magic to ensure Python compatibility.
import subprocess
import json
import shutil
import os

# The code is referred from  Coursera exercise19
# Check if FFmpeg is installed and install it if missing
if not shutil.which("ffmpeg"):
    !curl https://johnvansickle.com/ffmpeg/releases/ffmpeg-release-amd64-static.tar.xz -o ffmpeg.tar.xz
\
        && tar -xf ffmpeg.tar.xz && rm ffmpeg.tar.xz
    ffmdir = !find . -iname ffmpeg-*-static
    path = %env PATH
    path = path + ':' + ffmdir[0]
#     %env PATH $path

# Verify FFmpeg installation
!ffmpeg -version

"""The format of the films specified by the festival organisation is:
* Video format (container): mp4
* Video codec: h.264
* Audio codec: aac
* Frame rate: 25 FPS
* Aspect ratio: 16:9
* Resolution: 640 x 360
* Video bit rate: 2 – 5 Mb/s
* Audio bit rate: up to 256 kb/s
```

```
* Audio channels: stereo
"""

# Define the expected format required by the film festival
EXPECTED_FORMAT = {
    "container": "mp4",
    "video_codec": "h264",
    "audio_codec": "aac",
    "frame_rate": 25,
    "aspect_ratio": "16:9",
    "resolution_width": 640,
    "resolution_height": 360,
    "video_br_min_mbs": 2,
    "video_br_max_mbs": 5,
    "audio_br_max_kbs": 256,
    "audio_channels": 2
}

"""To check film properties, two functions are created:
- Extracts film metadata using FFprobe
- Compare extracted metadata with expected format
"""

# Extract metadata from a video file using ffprobe
def extract_metadata(file_path):
    try:
        # ffprobe command to get metadata in JSON format
        cmd = [
            "ffprobe", "-v", "error", "-show_streams", "-show_format",
            "-of", "json", file_path
        ]
        result = subprocess.run(cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE, text=True)
        metadata = json.loads(result.stdout)

        # Extract required metadata
        video_stream = next((stream for stream in metadata["streams"] if stream["codec_type"] ==
"video"), None)
        audio_stream = next((stream for stream in metadata["streams"] if stream["codec_type"] ==
"audio"), None)

        if not video_stream or not audio_stream:
            raise ValueError("Invalid video file. Missing video or audio stream.")

        # Extract video properties
        extracted_data = {
            "container": metadata["format"]["format_name"],
            "video_codec": video_stream["codec_name"].lower(),
            "audio_codec": audio_stream["codec_name"].lower(),
```

```
            "frame_rate": eval(video_stream["r_frame_rate"]),
            "aspect_ratio": f"{video_stream['width']}:{video_stream['height']}",
            "resolution_width": video_stream["width"],
            "resolution_height": video_stream["height"],
            "video_br_mbs": int(metadata["format"]["bit_rate"]) / 1e6,  # Convert bits to Mbps
            "audio_br_kbs": int(audio_stream.get("bit_rate", 0)) / 1e3,  # Convert bits to kbps
            "audio_channels": audio_stream["channels"]
        }

        return extracted_data

    except Exception as e:
        print(f"Error extracting metadata from {file_path}: {e}")
        return None


# Compare extracted metadata with expected format
def compare_format(metadata, expected_format):
    if not metadata:
        return "Error: Metadata extraction failed", []

    problems = []
    comparison_results = []

    def check_property(name, detected, required, condition):
        status = "Correct" if condition else "Mismatch"
        comparison_results.append(f"{name}: detected ({detected}) → {status}")
        if not condition:
            problems.append(name)

    # Compare properties
    detected_container = metadata["container"]
    check_property("Container", detected_container, expected_format["container"], "mp4" in
detected_container)
    check_property("Video Codec", metadata["video_codec"], expected_format["video_codec"],
metadata["video_codec"] == expected_format["video_codec"])
    check_property("Audio Codec", metadata["audio_codec"], expected_format["audio_codec"],
metadata["audio_codec"] == expected_format["audio_codec"])
    check_property("Frame Rate", f"{metadata['frame_rate']} FPS", f"{expected_format['frame_rate']}
FPS", abs(metadata["frame_rate"] - expected_format["frame_rate"]) <= 0.1)
    check_property("Resolution", f"{metadata['resolution_width']}x{metadata['resolution_height']}",
f"{expected_format['resolution_width']}x{expected_format['resolution_height']}",
metadata["resolution_width"] == expected_format["resolution_width"] and
metadata["resolution_height"] == expected_format["resolution_height"])
    # Convert bitrates before checking
    video_br_mbs = metadata["video_br_mbs"]
    audio_br_kbs = metadata["audio_br_kbs"]
```

```
    check_property("Video Bitrate", f"{video_br_mbs:.2f} Mbps",
f"{expected_format['video_br_min_mbs']} - {expected_format['video_br_max_mbs']} Mbps",
expected_format["video_br_min_mbs"] <= video_br_mbs <=
expected_format["video_br_max_mbs"])
    # Allow slight variation in audio bitrate
    tolerance = 10.0
    check_property("Audio Bitrate", f"{audio_br_kbs:.2f} kbps", f"≤
{expected_format['audio_br_max_kbs']} kbps", audio_br_kbs <=
expected_format["audio_br_max_kbs"] + tolerance)
    check_property("Audio Channels", metadata["audio_channels"],
expected_format["audio_channels"], metadata["audio_channels"] ==
expected_format["audio_channels"])

    # Determine overall status
    overall_status = "Format OK" if not problems else "Format Incorrect"

    return overall_status, comparison_results

"""Automates film format conversion"""

# Converts a video file to the required format
def convert_video(input_file, issues):
    try:
        # Generate output filename with '_formatOK' and in MP4 format
        file_name, file_ext = os.path.splitext(input_file)
        output_file = f"{file_name}_formatOK.mp4"

        # Check if only the container is incorrect,
        # If only the container is incorrect, remux without reencoding
        # If codec, frame rate, resolution, or bitrate are incorrect, perform re-encoding.
        container_issue = any("Container:" in issue for issue in issues)
        other_issues = any("Mismatch" in issue and "Container:" not in issue for issue in issues)

        if container_issue and not other_issues:
            print(f"Remuxing {input_file} → {output_file} (container change only)...")
            cmd = ["ffmpeg", "-i", input_file, "-c", "copy", output_file]
        else:
            print(f"Re-encoding {input_file} → {output_file} (format correction)...")
            cmd = [
                "ffmpeg", "-i", input_file,
                "-c:v", "libx264",      # Convert video to H.264
                "-b:v", "2M",           # Set video bitrate to 2 Mbps
                "-r", "25",             # Set frame rate to 25 FPS
                "-s", "640x360",        # Set resolution to 640x360
                "-c:a", "aac",          # Convert audio to AAC
                "-b:a", "256k",         # Set audio bitrate to 256 kbps
                "-ac", "2",             # Set stereo audio channels
                output_file
```

```
        ]

        # Execute FFmpeg
        result = subprocess.run(cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE, text=True)

        if result.returncode == 0:
            print(f"Successfully converted: {input_file} → {output_file}")
            return output_file
        else:
            print(f"Error converting {input_file}: {result.stderr}")
            return None

    except Exception as e:
        print(f"Unexpected error during conversion: {e}")
        return None

"""Combine whole process to check the file foramt and Generate a report in TXT indicating which
films do not respect the digital format specified by the festival and what are the 'problematic'
fields."""

# Generates a TXT report to summarize the video format verification results
def film_format_checker(video_files, report_filename):
    report_content = []

    report_content.append("Video Format Verification Report\n")
    report_content.append("="*40 + "\n")

    for video in video_files:
        report_content.append(f"File: {os.path.basename(video)}\n")

        # Extract metadata
        metadata = extract_metadata(video)
        status, results = compare_format(metadata, EXPECTED_FORMAT)

        # Append format check results
        report_content.append(f"Status: {status}\n")
        for result in results:
            report_content.append(f"   - {result}")

        # Convert the video if incorrect
        converted_file = None
        if status == "Format Incorrect":
            issues = [res for res in results if "Mismatch" in res]
            converted_file = convert_video(video, issues)
            if converted_file:
                report_content.append(f"Convert to expected format:
{os.path.basename(converted_file)}\n")
            else:
```

```
            report_content.append(f"Conversion Failed!\n")
        else:
            report_content.append("No conversion needed.\n")

        report_content.append("="*40 + "\n")

    # Write to a TXT file
    with open(report_filename, "w") as report_file:
        report_file.writelines("\n".join(report_content))

    print(f" Report saved as: {report_filename}")
    print("\n".join(report_content)) # Display report content

    return report_filename

video_files = [
    "/content/The_Gun_and_the_Pulpit.avi",
    "/content/The_Hill_Gang_Rides_Again.mp4",
    "/content/Cosmos_War_of_the_Planets.mp4",
    "/content/Last_man_on_earth_1964.mov",
    "/content/Voyage_to_the_Planet_of_Prehistoric_Women.mp4"
]

film_format_checker(video_files, "video_format_check.txt")

"""Verify converted files to confirm they align with the expected format"""

# Function to recheck all converted files to confirm they align with the expected format.
def verify_converted_files(converted_files):
    print("Verifying Converted Video:")
    print("=" * 40)

    for converted_video in converted_files:
        print(f"Checking: {os.path.basename(converted_video)}")

        # Extract metadata
        metadata = extract_metadata(converted_video)
        status, results = compare_format(metadata, EXPECTED_FORMAT)

        # Print verification results
        print(f"Status: {status}")
        if status == "Format OK":
            print("File meets all format requirements.")
        else:
            print("Issues still detected after conversion!")
            for result in results:
                print(f"   - {result}")
```

```
        print("=" * 40)

converted_video_files = [
    "/content/The_Gun_and_the_Pulpit_formatOK.mp4",
    "/content/The_Hill_Gang_Rides_Again_formatOK.mp4",
    "/content/Cosmos_War_of_the_Planets_formatOK.mp4",
    "/content/Last_man_on_earth_1964_formatOK.mp4",
    "/content/Voyage_to_the_Planet_of_Prehistoric_Women_formatOK.mp4"
]

verify_converted_files(converted_video_files)
```