ISP Final Report

Table of Content

Exercise 1

Task 2 table results

	Total number of cars	**Cars per minute**
Traffic_Laramie_1.mp4	6	2.02338
Traffic_Laramie_2.mp4	4	2.27101

Frame differencing and background subtraction

Frame differencing and background subtraction are both techniques used in video analysis, often for the purpose of detecting moving objects.

Frame differencing involves taking two consecutive frames and finding the absolute difference between the pixel values of these frames. This technique will highlight the regions of the frame where there is movement between frames. In Python using OpenCV, this can be implemented using the `cv2.absdiff()` function.

Background subtraction, on the other hand, is a bit more complex technique where you maintain a model of the background over time and subtract this model from the current frame to identify the foreground objects. The idea here is to isolate the moving objects from the stationary background. In OpenCV, there are several algorithms available for background subtraction such as MOG, MOG2, and KNN, which can be used through functions like `cv2.createBackgroundSubtractorMOG2()` or `cv2.createBackgroundSubtractorKNN()` to create a background subtractor, and then using the `apply()` method of the created object to perform background subtraction on each frame.

A brief analysis of the application

The Exercise 1.1 application works by reading a video file using `cv2.VideoCapture()` frame by frame. Each frame is then de-noised by applying `cv2.GaussianBlur()`. Which is a low-pass filter that removes high-frequency components (e.g. noise, edges) from the image. This is a common practice in image processing to prepare the image for further processing.

Then, each frame is processed by an instance of a background subtractor initialized by `cv2.createBackgroundSubtractorKNN()` method. I've explored existing background subtraction techniques[1] and found KNN more suitable for this case because it produces less noise and preserves the shape of the objects for a longer period of time. This is important because moving objects may stay in the same place for a few frames and we don't want to lose them.

The results of the `apply()` method are the foreground mask. The foreground mask is then thresholded using `cv2.threshold()` method to remove the noise and produce a binary image. The binary image is then processed by `cv2.morphologyEx()` method to fill the holes in the foreground objects. The resulting image is then processed by `cv2.findContours()` method to find the contours of the foreground objects. The contours are then filtered by a hard-coded minimum area defined by a `MIN_CONTOUR_AREA` constant. Then we remove the contours that are in the top half of the image because we are only interested in the cars on the Main Street. The remaining contours are then processed by `cv2.convexHull()` method because cars morphologically are convex objects. The resulting contours are then enclosed by a rectangle using `cv2.boundingRect()` method. The resulting rectangles are then drawn on the original frame using `cv2.rectangle()` method.

This approach is fast, simple, and highly configurable. The downside is that it requires a manual tuning of the parameters to get the best results. The most important parameter is `MIN_CONTOUR_AREA` which defines the minimum area of

the contour to be considered as a car. With a smaller value, walking people and bicycles will be misclassified as cars. With a larger value, small cars will be misclassified as noise. So this parameter should be tuned for each specific case.

To detect moving objects, the exercise 1.2 application utilizes the same background extraction mechanism as the Exercise 1.1. To keep track of moving objects, this application has a `Tracker` class that internally has a list of objects with its coordinates and updates the coordinates of the objects based on the contours found in each frame. The movements of objects are predicted by using a Kalman filter implemented by `cv2.KalmanFilter()` method.

The Kalman filter is a mathematical model that uses noisy measurements observed over time to produce estimates of unknown variables. In this case, the unknown variables are the coordinates of the objects. The Kalman filter is used to predict the next position of the objects based on the previous positions. We assume that a car cannot disappear in the middle of the scene. So the predicted position is used when the objects are temporarily not detected during a few consecutive frames. It makes the tracking more robust.

One of the disadvantages of this application is that it is not robust enough to detect cars that stopped for a long time. This is because the background subtractor will consider the stopped car as a part of the background. To overcome this problem, we can use a more sophisticated tracking algorithm such as a KLM tracker, DeepSORT, or YOLO.

Exercise 2

A link to the application running in a Jupyter notebook link in Coursera: https://hub.labs.coursera.org:443/connect/sharedxklwduue?forceRefresh=false&isLabVersioning=true

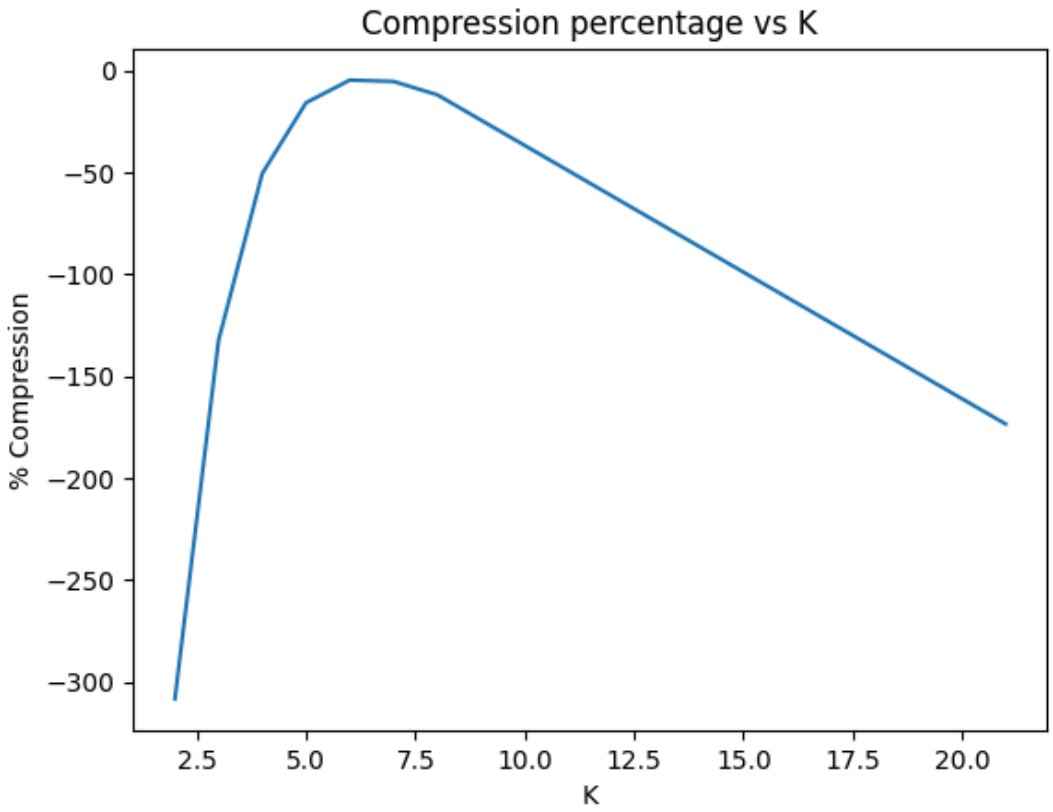
Results and analysis

The results with the proposed in the task description k parameters are the following:

	Original size	Rice (K = 4 bits)	Rice (K = 2 bits)	% Compression (K = 4 bits)	% Compression (K = 2 bits)
Sound1.wav	1002088	1516265	4115718	-51.3106	-310.714
Sound2.wav	1008044	1575347	4348595	-56.2776	-331.389

The size of the encoded file is about 1.5 bigger than the original file when K=4 bits and 4 times bigger when K=2 bits. In Rice coding, the parameter K determines how the input data is split between the unary part and the binary part of the code words. A larger value of K means that more bits will be used in the binary part and fewer in the unary part. The optimal value for K depends on the statistical characteristics of the input values, in particular, on the number of zeros in the input data and the distribution of the input values. The larger the number of zeros in the input data, the smaller the value of K will be optimal. The higher the variance of the input values, the larger the value of K should be. The optimal value of K can be found experimentally.

To find the optimal value of K, the following experiment was conducted. The input file was encoded with different values of K. The size of the encoded file was measured. The optimal value of K is the one that gives the smallest size of the encoded file. The experiment was conducted for K from 1 to 20 bits. The results are shown in the table below.



So, the best result is achieved when K = 6. However, the compression is -4.6% for Sound1.wav and -6.24% for Sound2.wav, i.e. the compressed file is slightly larger than the original file. Unfortunately, the compression ratio does not seem to be higher than 1 for any K value. The overhead of the encoding is too large compared to the file size.

A brief analysis of the application

The application implements Rice encoding and decoding scheme with variable k param. There were implemented different functions to encode and decode the messages: `rice_encode()` and `rice_decode()`. The correctness of the implementation was tested. The message was encoded and decoded with the same k param. The decoded message was the same as the original one. It also tests that the decoder works correctly in a case when a number of zeros or ones padded to the left of the message.

The application also implements helper functions to convert a list of numbers into a string of bits and vice: versa `bits_to_bytes()` and `bytes_to_bits()`. The correctness of the implementation was tested. The list of numbers was converted into a string of bits and then back to the list of numbers. The list of numbers was the same as the original one.

Also, there is a function `is_files_equal()` to compare two files. The function compares two files bit by bit. If the files are the same, the function returns True, otherwise it returns False. The function was tested on two equal and two different files. The function returned True for two equal files and False for two different files.

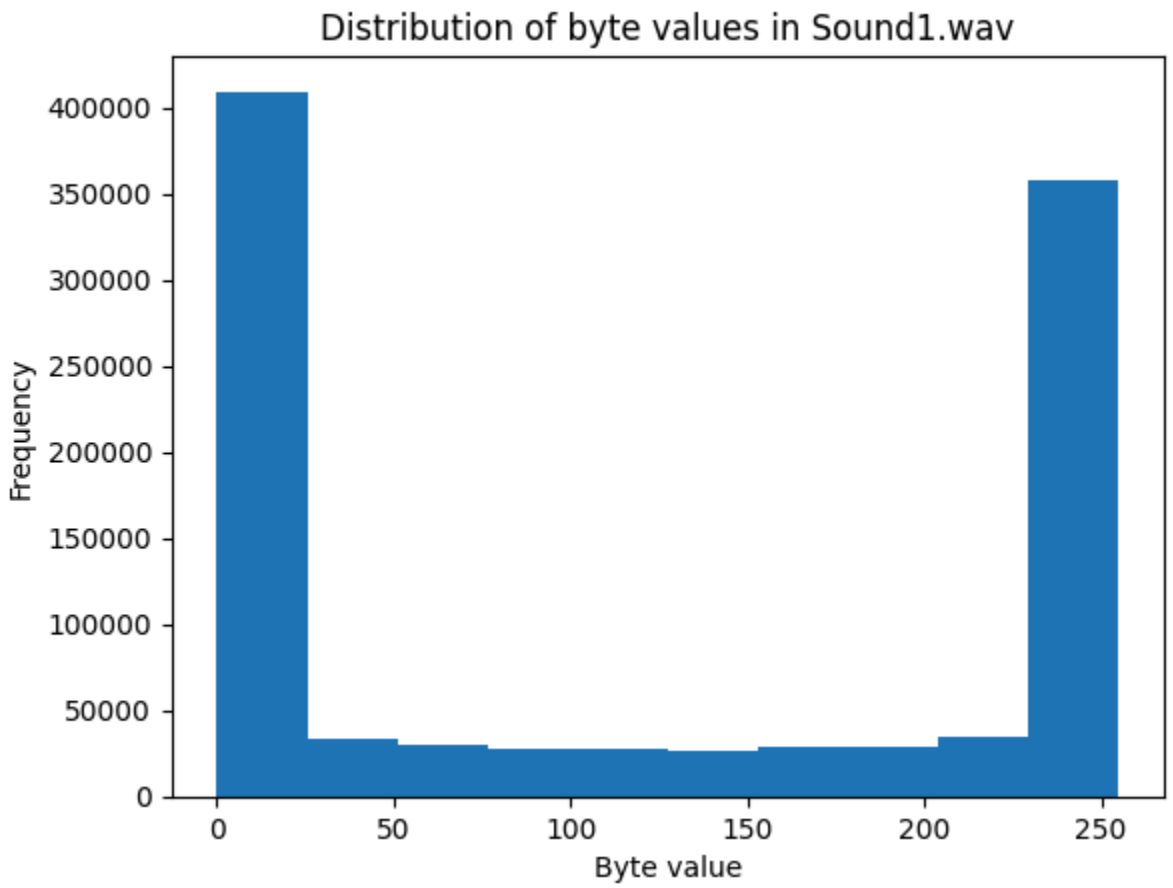
The application works by reading the input file as a sequence of bytes in a function `encode_file_rice()` and rice encoding the sequence of bytes by applying the `rice_encode()` function to each byte. The result is a list of byte strings with the encoded messages, joined into one big string. Then encoded to bytes by applying the `bits_to_bytes()` function. The result is written to the output ex2 file.

To decode the file, the procedure is the opposite. The input file is read as a sequence of bytes. Then the bytes are converted to a string of bits by applying the `bytes_to_bits()` function. The string of bits is decoded by applying the `rice_decode()` function to each byte. The result is a list of bytes with the decoded messages is written to the output Env_Dec file.

Further development implemented

For further development, I explored how different K-parameter values affect the compression ratio. I ran the program with different K values and plotted the compression ratio against K. The result is presented above in the Results and analysis section. The best result is achieved when K = 6. However, the

compression is -4.6% for Sound1.wav and -6.24% for Sound2.wav, i.e. the compressed file is slightly larger than the original file.
Unfortunately, the compression ratio does not seem to be higher than 1 for any K value. The overhead of the encoding is too large compared to the file size. By its nature, Rice coding is more effective for datasets in which there are more small values than large values. So I've explored the distribution of the values in the Sound1.wav file:



As can be seen from the plot, there are more large values and small values with similar frequency. And a small number of values are in the middle. This is why Rice coding is not so effective for this dataset. To make it more effective, we need to find a way to make the distribution more skewed. I have tried to join any two consecutive values together, but it doesn't help much. The distribution looked similar.

I suspected that there are long sequences of 0s and 1s in the original dataset. I supposed that a Run-Length Encoding (RLE) should be adequately effective for this situation. I have implemented Run-Length Encoding (RLE) and tested it against provided files. Results are provided below:

	Original size	**RLE Encoded**	**% Compression**
Sound1.wav	1002088	2004092	-99.9916
Sound2.wav	1008044	2016080	-99.9992

The file size was doubled by an RLE encoding, but the distribution of the values was skewed to the left, as expected. I've tried to combine RLE and Rice encoding with K=6:

	Original size	**RLE+Rice Encoded**	**% Compression**
Sound1.wav	1002088	1931642	-92.7617
Sound2.wav	1008044	1953034	-93.7449

The Rice encoding slightly reduced the size of RLE-encoded files, but it was not better than the unencoded message, or Rice-encoded with K=6.

Conclusion: The STD of the Sound1.wav file is 112 supposedly a big variance of the values. In this situation, for any compression algorithm to be more effective an original file should be splitted into chunks. Each chunk will have their own statistical characteristics of data and it will be possible to determine the optimal compression algorithm parameters for each chunk. In the case of the Rice, different K values for different chunks.

Exercise 3

A link to the application running in a Jupyter notebook link in Coursera: https://hub.labs.coursera.org:443/connect/sharedxklwduue?forceRefresh=false&isLabVersioning=true

Report

The report that the application examining the films submitted by the organization:

```
file: Exercise3_Films/Last_man_on_earth_1964.mov
Wrong video codec: prores
Wrong audio codec: pcm_s16le
Wrong frame rate: 24000/1001
Wrong video bitrate: 9.285191 Mb/s
Wrong audio bitrate: 1536.0 Kb/s
File is INVALID.

file:
Exercise3_Films/Voyage_to_the_Planet_of_Prehistoric_Women.mp4
Wrong video codec: hevc
Wrong audio codec: mp3
Wrong frame rate: 30000/1001
Wrong video bitrate: 8.038857 Mb/s
Wrong audio bitrate: 320.0 Kb/s
File is INVALID.

file: Exercise3_Films/The_Gun_and_the_Pulpit.avi
Wrong container format: avi
Wrong video codec: rawvideo
Wrong audio codec: pcm_s16le
Wrong aspect ratio: 180:101
Wrong resolution: 720x404
Wrong video bitrate: 87.438878 Mb/s
Wrong audio bitrate: 1536.0 Kb/s
```

```
File is INVALID.

file: Exercise3_Films/Cosmos_War_of_the_Planets.mp4
Wrong frame rate: 30000/1001
Wrong aspect ratio: 314:177
Wrong resolution: 628x354
Wrong audio bitrate: 317.103 Kb/s
File is INVALID.

file: Exercise3_Films/The_Hill_Gang_Rides_Again.mp4
Wrong video bitrate: 7.53773 Mb/s
File is INVALID.
```

FFmpeg installation and configuration

I am using macOS on an Apple Silicon (M1) machine. I have installed FFmpeg using Homebrew:

```
brew install ffmpeg
```

The version of FFmpeg installed is:

```
ffmpeg -version
```

```
ffmpeg version 6.0 Copyright (c) 2000-2023 the FFmpeg developers
built with Apple clang version 14.0.3 (clang-1403.0.22.14.1)
```

On other platforms, FFmpeg can be installed using the instructions from the official website: https://ffmpeg.org/download.html. For example, on Ubuntu Linux, FFmpeg can be installed using the following command:

```
sudo apt install ffmpeg
```

A brief analysis of the application

The application calls ffprobe command on each provided file and analyzes the JSON output of the command to determine the properties of the file. Despite the fact that there is a special library ffmpeg-python for Python to work with FFmpeg, I decided to use the ffprobe command directly, because I would like to demonstrate the proficiency in working with the ffprobe directly.

The determined by ffprobe properties matched with the provided requirements. The application prints the properties that do not match the requirements. The application marked the file as INVALID if at least one property does not match the requirements.

The application calls ffmpeg command on each file marked as INVALID to convert the file to the required format. All converted files are stored in the "results/" directory with the "_formatOK" suffix in the file name.

The validity of the converted file is checked again by ffprobe by the same criteria as for the original file. All the files that are marked as VALID. Also, I have checked the converted files manually by playing them in the VLC player. All the converted files are valid and playable.

Definitions

Video format (container) is a file format to wrap various media elements such as video, audio, subtitles, and metadata into one file. The examples of video formats are MP4 (MPEG-4 Part 14), AVI (Audio Video Interleave), MKV (Matroska), WEBM, and many others.

Video codec is an algorithm to compress and decompress video files. "Codec" is a portmanteau of "compressor-decompressor" or, more commonly, "coder-decoder". The main goal of a codec is to reduce the file size without

significantly, if at all, reducing the quality of the video. The most commonly used video codecs are: H.264, H.265, VP9, AV1, and many others.

Audio codec is an algorithm to compress and decompress audio files. As for video codecs, the main goal of an audio codec is to reduce the file size. There are lossless and lossy audio codecs. The lossless audio codecs do not reduce the quality of the audio file, while the lossy audio codecs reduce the quality of the audio file, but the file size is smaller. The most commonly used audio codecs are: AAC, MP3, FLAC, and many others.

A **frame rate** is the number of frames displayed per second (fps) in a video. Usually, the frame rate is more than 24 fps. The higher the frame rate, the smoother the video.

The **aspect ratio** is the ratio of the width of a video to its height. The most common aspect ratios are 16:9 and 4:3. The aspect ratio of a video is usually determined by the aspect ratio of the display device. For example, the aspect ratio of a modern TV is 16:9, while the aspect ratio of an old TV is 4:3.

The **resolution** of a video is the number of pixels in a video. The resolution is usually determined by the aspect ratio of the video, the size of the screen and the density of pixels. For example, the resolution of a video with an aspect ratio of 16:9 is 1920x1080 (Full HD) or 3840x2160 (4K).

The **video bit rate** is the amount of data processed per unit of time in a video file, typically measured in kilobits per second (kbps) or megabits per second (Mbps). Bit rate can have a significant impact on the visual quality of a video and its file size. The higher the bit rate, the higher the quality of the video and the larger the file size.

The **audio bit rate** is the amount of data processed per unit of time in an audio file. As for video bit rate, the audio bit rate is usually measured in kilobits per second (kbps) or megabits per second (Mbps) and the higher the bit rate, the higher the quality of the audio file and the larger the file size.

The **audio channels** are separate audio tracks in a single audio or video file. The most common audio channels are mono (1 channel), stereo (2 channels), and surround sound (5.1 channels).

References

[1] Kritika Singh, Dr. Vishal Gupta. "*A COMPARATIVE STUDY OF MOG AND KNN FOR FOREGROUND DETECTION*". JETIR May 2018, Volume 5, Issue 5 [Online] Available from: https://www.jetir.org/papers/JETIR1805563.pdf [30 Aug 2023]