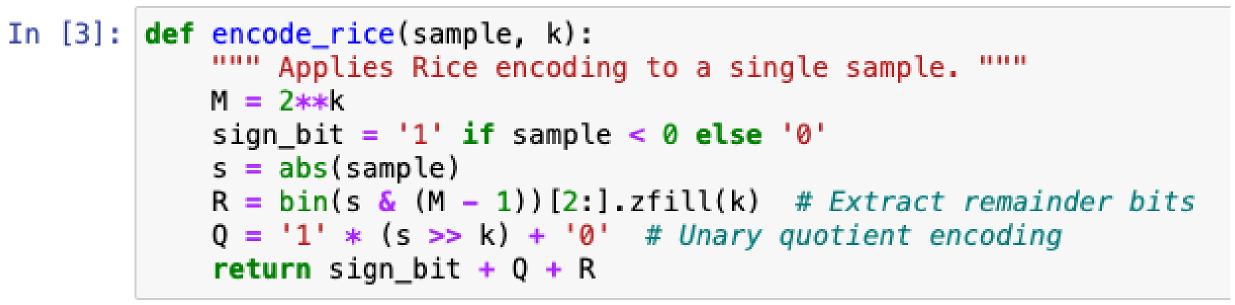
```
In [1]: import os
        import wave
        from scipy.io import wavfile
        import numpy as np
        import struct
        from functools import partial
```

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.

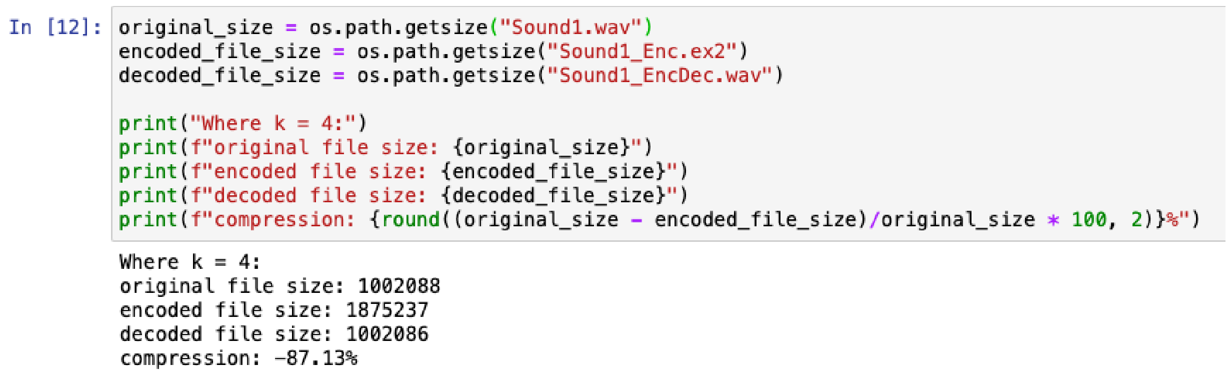
```
In [2]: def read_wav_header(input_file):
            """ Reads and extracts WAV file header information. """
            header_info = input_file.read(44)
            header_fields = {
                "Chunk Size": struct.unpack('<L', header_info[4:8])[0],
                "Format": header_info[8:12],
                "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]: 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.

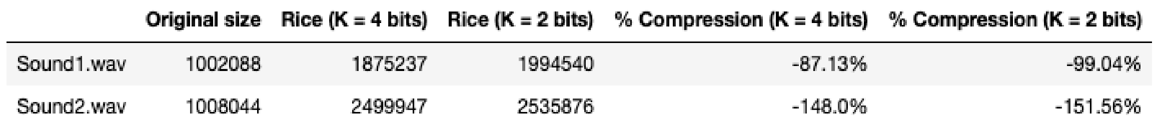


```
In [12]: original_size = os.path.getsize("Sound1.wav")
         encoded_file_size = os.path.getsize("Sound1_Enc.ex2")
         decoded_file_size = os.path.getsize("Sound1_EncDec.wav")

         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.

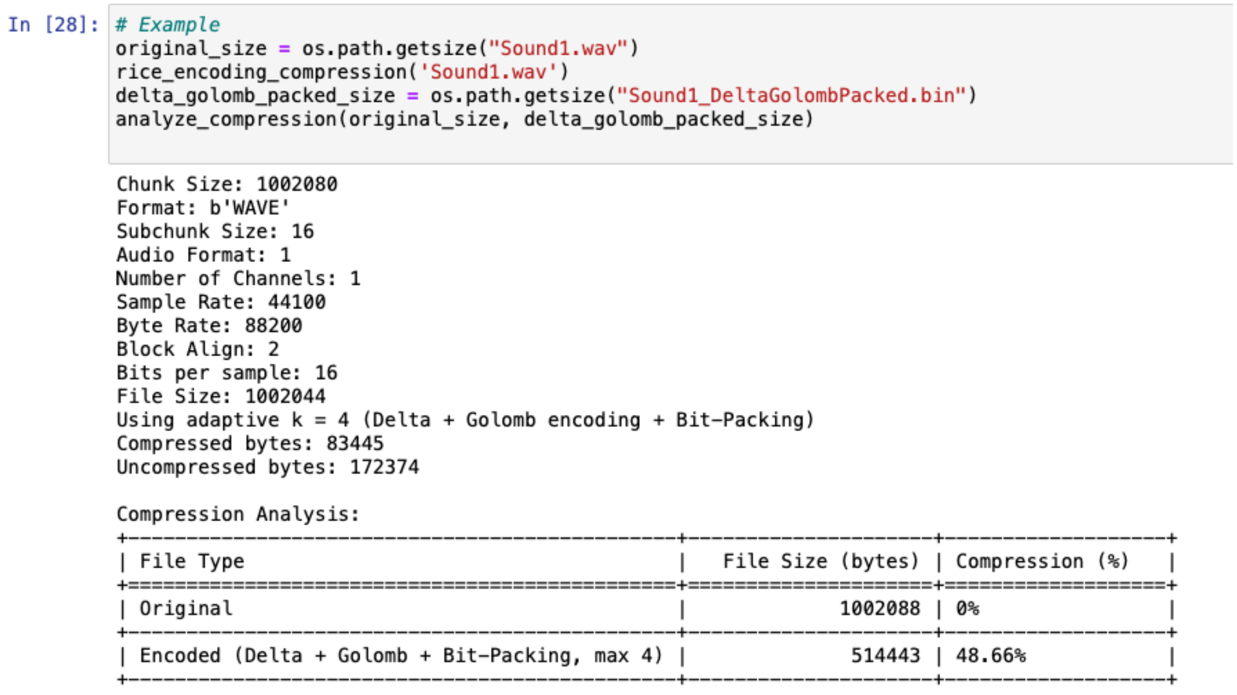
Rice encoding with K=4 produced a compression rate of -87.13% for Sound1.wav, according to the results, which indicates that the encoded file grew noticeably larger than the original. With a rate of -99.04%, using K=2 made the compression even worse. This suggests that the audio stream was not successfully 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