Universidade de Aveiro
Departamento de Electrónica, Telecomunicações e Informática
Mestrado em Engenharia de Computadores e Telemática

Information and Codification

IC - Report Project 1

Fabio Matias - 108011

Gonçalo Cunha - 108352

Matilde Teixeira - 108193

Aveiro

Janeiro 2025

Contents

1 Introduction

The objective of this lab work is to explore and implement efficient encoding and compression techniques across various types of data, including audio, images, and video. By building foundational tools such as a BitStream class for bit-level file operations, the work progresses into more advanced methods like Golomb coding and predictive coding. These techniques are applied to achieve both lossless and lossy compression, enabling optimized storage and transmission of data.

The tasks are divided into clearly defined parts, beginning with bit manipulation and moving toward more complex implementations like predictive coding for audio, image, and video files. Each step emphasizes efficiency and adaptability, ensuring robust solutions for practical applications. This lab work not only serves as a hands-on exploration of coding techniques but also provides a platform for testing and analyzing the effectiveness of these approaches in real-world scenarios.

2 Bit Stream Class

2.1 Task Summary

The BitStream Class task focuses on creating a reusable and efficient C++ class for precise bit-level file operations. The class will support reading and writing individual bits, sequences of bits, and strings as bit series, maintaining bit order and ensuring efficiency. It will also include error handling for scenarios like reading beyond the file end.

File management is handled through methods that open streams in specified modes, ensure proper closure in the destructor, and check for the end of the file during reading.

Writing operations use a buffered approach, where writeBit packs bits into an 8-bit buffer before writing to the file, reducing I/O overhead. The writeBits method supports multi-bit writes, with input validation to ensure correctness. The flushBuffer function ensures any partial data in the buffer is written, maintaining data integrity.

Reading operations mirror this structure, with readBit extracting bits from a buffer that is refilled from the file as needed. Multi-bit reads via readBits aggregate bits into integers, with input validation to prevent errors.

For handling textual data, writeString and readString operate at the byte level, encoding and decoding strings efficiently using the bit-based methods.

This implementation ensures performance through buffered I/O, robustness via input validation and exception handling, and versatility through low- and high-level operations, making it suitable for precise and efficient bitwise file manipulation.

2.2 Performance Metrics

2.2.1 Processing Time

The bitStream class demonstrates efficient processing capabilities for large-scale string operations. Writing 1,000,000 strings was completed in 0.436 seconds, while reading the same volume took 0.584 seconds. The reading operation shows a 34% longer processing time compared to writing, indicating that while both operations perform well at scale, there may be room for further optimization in the read functionality.

2.3 Limitations and improvements

The BitStream class, as designed, addresses the core requirements outlined in the task, providing functionality for efficient bit-level manipulation in file operations. For the required functionalities, the class handles writing and reading single bits, integers of variable bit lengths, and strings as a series of bits, ensuring correct bit-ordering and efficient packing into bytes. While the class focuses on packing bits efficiently, the implementation might not fully exploit modern CPU architectures or memory hierarchies, leaving room for optimization in speed and memory usage, especially when dealing with large files or frequent operations.

Beyond the task's requirements, our implementation could benefit from enhancements to improve usability and extend its versatility. For instance, the addition of buffering mechanisms could further optimize performance by reducing the number of file I/O operations. Implementing support for handling larger data types beyond the standard range of 64 bits might make the class more applicable to specialized use cases. Advanced compression or error-correction features could provide added functionality for tasks requiring reliability in data storage or transmission.

Moreover, while the class operates effectively with individual files, its applicability could be expanded with support for streams, allowing integration with memory buffers or network sockets. Improved debugging and logging tools could also make development and troubleshooting easier, particularly in complex systems relying on bit-level operations. Introducing compatibility with multi-threaded environments would enable concurrent processing, enhancing performance in modern applications. Lastly, user-friendly documentation and an intuitive API design could make the class more accessible to developers, ensuring broader adoption and ease of integration into larger projects.

While our BitStream class is a foundational tool for efficient bit-level file manipulation, addressing these limitations and incorporating improvements would significantly enhance its functionality, robustness, and adaptability for a wider range of applications.

3 Golomb Coding

3.1 Task Summary

The Golomb Coding task focuses on creating a C++ class to implement Golomb coding, a compression technique particularly effective for encoding integers with geometric distributions. This class will leverage the BitStream class developed earlier to perform efficient bit-level operations and will support both positive and negative integers. For negative values, the class will offer two user-configurable encoding approaches: sign and magnitude (where the sign is encoded separately) and positive/negative interleaving (a zigzag mapping to convert all numbers into non-negative integers). The class will also be parameterized, allowing users to adjust the encoding length via the parameter m for adaptability to different probability distributions.

As it was one of the requirements, our golomb class supports two encoding modes: interleaving, which uses zigzag mapping for signed integers, and sign/magnitude, where a sign bit is explicitly stored. In the encode method, the input integer is mapped based on the selected mode. The value is divided into a quotient and remainder using m. The quotient is unary-coded, while the remainder is encoded with a dynamic bit length determined by m to minimize redundancy. Interleaving mode uses zigzag mapping to ensure positive values for all inputs.

The decode method reverses this process. The unary-coded quotient and binary remainder are read and reconstructed. If interleaving is active, zigzag decoding restores the signed value. For sign/magnitude, the sign bit is applied after decoding. With efficient handling of signed values, dynamic parameter updates via setM, and compact encoding logic, the golomb class ensures a concise and adaptable implementation of Golomb coding.

This task is critical for exploring real-world applications of data compression, as Golomb coding is widely used in areas like image and audio compression. By completing this task, the groundwork will be established for integrating Golomb coding into more complex codecs in later stages of the project.

3.2 Performance Metrics

For performance testing, we applied our Golomb approach to a txt file, containing 100000 numbers (13.0MB), we tested for multiple values of M, in these case those values are rather high given the big interval of numbers in the file, but, as to push the class to its limit, the use of such high values seemed appropriate.

3.2.1 Processing Time

m	Interleaving	Encoding Time (s)	Decoding Time (s)
4096	False	0.0149285	0.11384
4096	True	0.0190267	0.197182
2048	False	0.0191951	0.216387
2048	True	0.0281451	0.394327
1024	False	0.0282559	0.397145
1024	True	0.0452415	0.750243
512	False	0.0444753	0.760957
512	True	0.0783772	1.48174
256	False	0.0795321	1.49172
256	True	0.145956	2.94442
128	False	0.144818	2.96777
128	True	0.283587	5.87402

The data shows a clear inverse relationship between the M value and processing time. As M decreases from 4096 to 128, both encoding and decoding times increase significantly. Notably, decoding consistently requires more processing time than encoding, with ratios ranging from 7-20x slower. The introduction of interleaving further increases processing overhead, approximately doubling both encoding and decoding times across all M values. This suggests that while interleaving may offer certain benefits, it comes with a substantial performance cost that should be carefully considered based on specific use case requirements.

3.2.2 Compressing Ratios

m	Interleaving	Output File Size (bytes)
4096	False	321403
4096	True	461450
2048	False	461450
2048	True	754129
1024	False	754129
1024	True	1351985
512	False	1351985
512	True	2560209
256	False	2560209
256	True	4989162
128	False	4989162
128	True	9859572

The output file sizes demonstrate that larger M values achieve better compression ratios. With M=4096 and no interleaving, the output file size is 321,403 bytes, representing the best compression performance from the test cases. When M decreases, the output file size increases exponentially, reaching nearly 10MB at M=128 with interleaving. Interleaving consistently results in larger output files, typically increasing the file size by 40-100% compared to non-interleaved compression with the same M value. This indicates that while smaller M values might be necessary for certain applications, they come with a significant storage overhead that must be balanced against other requirements.

3.3 Comparative Analysis

When comparing Golomb encoding to other lossless compression methods, its strengths become particularly evident. While Huffman coding is widely used, it requires two separate data passes and carries the overhead of storing frequency tables. These limitations make it less suitable for our numeric data compression needs. Additionally, its tree-based structure demands higher memory usage and more complex implementation than Golomb's straightforward approach.

Run-Length Encoding (RLE) presents even more significant limitations for our use case. Despite its simplicity, RLE's effectiveness is severely limited to scenarios with consecutive repeated values. When dealing with varied numeric sequences, as in our dataset, RLE's performance deteriorates dramatically, potentially resulting in larger file sizes than the original data.

The dictionary-based LZ77/LZ78 algorithms, while powerful for general-purpose compression, exhibit several drawbacks in our context. Their requirement for maintaining and searching through dictionaries results in higher memory usage and slower processing speeds. The overhead of storing dictionary references further reduces their efficiency when dealing with purely numeric data.

Arithmetic coding, despite its theoretical excellence in approaching entropy limits, presents practical challenges that make it less suitable than Golomb encoding. Its reliance on complex floating-point arithmetic introduces risks of numerical precision errors and significantly increases computational requirements. The ability to tune Golomb encoding's performance through parameter M offers additional flexibility, allowing for optimization based on specific data characteristics and application requirements. This adaptability, combined with its inherent strengths in handling numeric data, reinforces its position as the most appropriate compression solution for our needs, despite the availability of more general-purpose alternatives.

3.4 Limitations and improvements

We believe that our Golomb Coding class is well implemented within the scope of the given task, however, there is also the recognizable idea that our implementation is not perfect, and could be improved in some ways.

Possible improvements beyond the immediate scope include dynamically adjusting the parameter m based on input data distribution to enhance compression efficiency and optimizing performance by using more efficient bit manipulation or caching techniques. Expanding the coding options, such as supporting Rice coding for power-of-two m, would increase versatility. Enhancements in file handling, like supporting complex file formats and adding command-line options for parameters, would improve usability. Additionally, making the class thread-safe would enable efficient parallel processing for large datasets.

There is also a possible error handling component, which we deemed not to be necessary, given the limited use cases in which our class would, and will be, utilized, for that reason, and for the sake of time, we decided not to go further into error handling mechanisms.

4 Audio Encoding

4.1 Task Summary

The Audio Encoding with Predictive Coding task focuses on developing an audio compression system that combines predictive coding with Golomb encoding to achieve efficient compression. The task involves implementing both lossless and lossy encoding methods, leveraging predictive models and the previously developed BitStream and Golomb coding classes for efficient data handling.

The lossless codec encodes prediction residuals, which represent the difference between actual and predicted audio samples. For mono audio, temporal prediction is used, where each sample is predicted based on previous samples in the same channel. For stereo audio, the codec incorporates both temporal and inter-channel prediction, utilizing information from both the current and the other channel to improve efficiency. The codec must support a fixed user-defined mm parameter for Golomb coding, while also offering an adaptive mechanism to dynamically determine the optimal mm value during encoding and decoding. This ensures better performance across different audio datasets.

The lossy codec extends the lossless encoder by introducing quantization, a process that reduces the precision of audio samples to achieve a desired compression level. Users can specify a target average bitrate, and the encoder dynamically adjusts the quantization levels to meet this requirement. This approach trades off some audio quality for a significant reduction in data size, making it suitable for applications where storage or bandwidth constraints are critical.

Both codecs rely heavily on the BitStream class for bit-level data handling and the Golomb coding class for efficient encoding of residuals or quantized samples. The implementation is designed to support common PCM audio formats, such as .wav files, ensuring broad applicability.

This task is essential for understanding the practical application of predictive coding techniques in audio compression. The lossless and lossy codecs provide a foundation for future exploration of multimedia compression and highlight the trade-offs between data size and quality. The work showcases the power of combining predictive models with efficient encoding schemes to address real-world challenges in data compression.

4.2 Performance Metrics

The AudioCodec system demonstrates effective compression capabilities and high-quality audio reconstruction across different configurations. The results highlight the trade-offs between lossless and lossy compression modes in terms of compression efficiency, computational requirements, and signal fidelity.

4.2.1 Processing Time

Mode	**Encoding Time**	**Decoding Time**
Lossless Adaptive	Moderate	Low
Lossless Fixed	Moderate	Low
Lossy Adaptive	High	Moderate
Lossy Fixed	High	Moderate

Table 3: Processing Time Across Compression Modes

Lossless fixed configurations exhibit the shortest processing times due to simpler operations, such as delta encoding and fixed Golomb parameter settings.

Lossy adaptive configurations have the highest encoding times, primarily due to the Discrete Cosine Transform (DCT) and the additional complexity of dynamic parameter optimization.

Decoding times for all configurations are consistently lower than encoding times as they avoid computationally expensive tasks such as DCT computations.

4.2.2 Compression Ratios

Mode	**Compression Ratio**	**Space Saving**
Lossless Adaptive	1.59:1	37.12%
Lossless Fixed	1.75:1	42.96%
Lossy Adaptive	3.94:1	74.60%
Lossy Fixed	3.94:1	74.60%

Table 4: Compression Ratios and Space Savings

Lossless configurations achieve moderate compression ratios, with **lossless adaptive** achieving a ratio of **1.59:1** and a space saving of **37.12%**, and **lossless fixed** achieving a slightly higher ratio of **1.75:1** and a space saving of **42.96%**. These results highlight the trade-off between precise reconstruction and the overhead introduced by metadata and delta encoding.

In contrast, lossy configurations leverage the Discrete Cosine Transform (DCT) and quantization to achieve significantly higher compression ratios of **3.94:1** and a space saving of **74.60%**. This demonstrates the codec's ability to compactly represent audio signals while effectively prioritizing relevant information.

4.2.3 Error Metrics

Mode	**Mismatched Samples**	**SNR (dB)**	**Subjective Quality**
Lossless Adaptive	0 (0.00%)	∞	5.0 (Imperceptible)
Lossless Fixed	0 (0.00%)	∞	5.0 (Imperceptible)
Lossy Adaptive	88144 (99.94%)	30.38	5.0 (Imperceptible)
Lossy Fixed	88144 (99.94%)	30.38	5.0 (Imperceptible)

Table 5: Error Metrics for Signal Fidelity

In lossless configurations, the reconstructed audio is identical to the original, with **0 mismatched samples** and an infinite **Signal-to-Noise Ratio (SNR)**. These modes guarantee perfect reconstruction, making them ideal for scenarios requiring high fidelity, such as archival storage or professional audio editing.

Lossy configurations, on the other hand, exhibit a high percentage of mismatched samples (**99.94%**). However, the introduced errors are imperceptible, as indicated by the subjective quality score of **5.0 (Imperceptible)** on the ITU-R scale and an SNR of **30.38 dB**.

The codec effectively prioritizes relevant components, discarding less critical information to achieve significant compression gains while maintaining high perceptual quality.

4.2.4 Comparative Analysis

Compression Efficiency

Lossy compression modes significantly outperform lossless modes in terms of compression ratios, achieving a ratio of **3.94:1** with a space saving of **74.60%**, compared to the moderate efficiency of lossless configurations, which achieve up to **1.75:1** with a space saving of **42.96%**.

The compression efficiency of lossless modes is constrained by the overhead introduced by metadata and delta encoding, limiting their ability to achieve higher compression ratios.

Quality Retention

Lossless modes ensure perfect signal reconstruction with **0 mismatched samples** and an **infinite Signal-to-Noise Ratio (SNR)**, making them ideal for applications requiring exact reproduction, such as archival storage and professional audio editing.

Despite introducing mismatched samples (**99.94%**), lossy modes maintain imperceptible quality loss, with an **SNR of 30.38 dB** and a subjective quality score of **5.0 (Imperceptible)**, demonstrating their effectiveness in preserving perceptually relevant audio components.

Processing Time

Lossless fixed configurations exhibit the lowest computational overhead, as they rely on simpler operations such as delta encoding and fixed Golomb parameter settings.

In contrast, lossy adaptive configurations incur the highest encoding times due to the computational complexity of the **Discrete Cosine Transform (DCT)** and dynamic parameter optimization.

However, decoding times are consistently lower across all configurations, as they avoid computationally intensive operations performed during encoding.

Adaptivity

Adaptive configurations enhance compression efficiency by dynamically adjusting parameters based on the characteristics of the audio signal.

This adaptivity proves particularly beneficial in lossless modes, where the optimized selection of Golomb parameters ensures better handling of variable signal characteristics.

Similarly, in lossy modes, adaptivity allows for more efficient compression while maintaining high perceptual quality.

4.3 Limitations and Improvements

The AudioCodec system demonstrates robust performance and flexibility across various configurations. However, it has certain limitations that could be addressed to improve its efficiency and applicability.

In lossless modes, the compression efficiency is limited by the overhead introduced by metadata and delta encoding. This overhead results in moderate compression ratios and restricts the utility of lossless modes for highly compressible or small datasets.

In lossy modes, the adaptive configurations incur a higher computational overhead due to parameter optimization and the complexity of the Discrete Cosine Transform (DCT). Furthermore, the current implementation does not incorporate psychoacoustic models, which are essential for optimizing lossy compression by discarding imperceptible audio frequencies while retaining perceived audio quality.

Another limitation is the use of a fixed block size during processing. Fixed block sizes may be suboptimal for audio signals with variable characteristics, where adaptive block sizing could improve both compression efficiency and processing time. Additionally, the lack of advanced entropy coding techniques, such as Huffman or arithmetic coding, prevents the codec from fully exploiting redundancies in the encoded data.

To address these limitations, several improvements could be implemented. For lossless modes, incorporating advanced entropy coding techniques, such as Huffman or arithmetic coding, would significantly reduce overhead and improve compression ratios.

By integrating psychoacoustic modeling into the lossy compression pipeline would enable more effective reduction of imperceptible frequencies, increasing compression efficiency while preserving perceived audio quality. Adaptive block sizing could further optimize compression for complex signals by dynamically adjusting the block size based on signal characteristics.

These enhancements would make the AudioCodec system more versatile and efficient for a wide range of use cases.

5 Image and Video Coding

5.1 Task Summary

The Image and Video Coding with Predictive Coding task involves developing codecs that apply predictive coding techniques for both lossless and lossy encoding. By leveraging prediction models and Golomb coding, the goal is to efficiently compress image and video data while maintaining flexibility to balance quality and compression levels.

The first step is implementing a lossless image codec, which uses spatial prediction to estimate pixel values based on neighboring pixels (as in JPEG or JPEG-LS). The residuals—differences between actual and predicted pixel values—are encoded using Golomb coding with an optimized mm parameter to ensure efficient compression. This lays the groundwork for encoding static images with high fidelity.

Building on this, the intra-frame video codec adapts the image codec to handle video sequences. Each frame is treated as a standalone image, encoded independently using spatial prediction. Parameters such as image dimensions and Golomb coding settings are embedded in the encoded video file to ensure correct decoding.

The task then advances to video coding, covering both intra-frame and inter-frame coding which incorporates temporal prediction through motion compensation. This approach identifies changes between consecutive frames, encoding only those differences. Regularly spaced intra-frames (I-frames) are used for reference, while inter-frames (P-frames) focus on relative changes. The user can configure the interval for I-frames, as well as the block size and search range for motion estimation. A mode decision mechanism determines whether to encode blocks in intra or inter mode based on their impact on bitrate, optimizing overall efficiency.

Finally, the project introduces lossy video coding, where quantization is applied to prediction residuals before Golomb coding. Quantization reduces the precision of residuals, enabling significant compression at the expense of some data loss. Users can control the level of quantization to balance compression and quality according to their needs.

This task is a comprehensive exploration of predictive coding techniques, progressing from static image compression to dynamic video encoding. By addressing both lossless and lossy methods, it highlights the trade-offs between fidelity and compression efficiency, providing robust solutions for a wide range of multimedia applications.

Additionally to the requirements above mentioned, we implemented a technique Discrete Cosine Transformation (DCT), a mathematical technique to transform spatial-domain image data into frequency domain.

In this project we applied this technique to each frame or block of video (8×8 or 16×16 block of pixel data), concentrating most of the energy in low-frequency components. This allowed for efficient compression by quantizing and discarding less significant high-frequency components. The DCT is applied independently to I-frames and residuals in P-frames, balancing file size reduction with video quality preservation.

During decoding, the inverse DCT and dequantization restore the data.

5.2 Performance Metrics

5.2.1 Processing Time

Image

In the performance tests 3 images were used to benchmark our enconder, babbon.png, a 512x512px image with 224.5 kB, peppers.png a 186,9 kB image with the same resolution, and at last, image.ppm, which is a 768x512px image of flowers with 1.2 MB. Given this, the results were the following:

Image	Encoding Time (sys)
baboon.png	0m0,195s
peppers.png	0m0,221s
image.ppm	0m0,283s

Table 6: Encoding Times for Images

The image encoding performance tests demonstrate strong efficiency across different image sizes and formats. The system processed three distinct test images: a 512x512 pixel baboon.png (224.5 KB), a similarly sized peppers.png (186.9 KB), and a larger 768x512 pixel flower image in PPM format (1.2 MB). The encoding times show a clear correlation with file size and complexity.

The baboon image, despite being larger than peppers.png, achieved the fastest encoding time at 0.195 seconds. This suggests that file size alone does not determine processing speed, and other factors such as image complexity and color distribution likely play significant roles. The peppers image required 0.221 seconds for encoding, representing a modest 13% increase in processing time despite being smaller in file size.

The PPM format image, being significantly larger at 1.2 MB, required 0.283 seconds for encoding. This represents a 45% increase in processing time compared to baboon.png, while handling approximately 5.3 times more data. This sub-linear scaling of processing time relative to file size indicates excellent efficiency in handling larger images, suggesting effective implementation of the encoding algorithms for varying data volumes.

These results demonstrate that the encoder maintains consistent performance across different image formats and sizes, with processing times remaining under 0.3 seconds even for megabyte-scale images. The system's ability to handle both PNG and PPM formats efficiently while maintaining reasonable processing times indicates robust and well-optimized implementation.

Video

In terms of video the encoding times can be seen in the table below.

The encoding times for the tested videos highlight differences in complexity and size.

Video	**Encoding Time (s)**
`akiyo_cif.y4m`	249.93
`bus_cif.y4m`	132.79
`deadline_cif.y4m`	1212.86

Table 7: Encoding Times for Videos

The `bus_cif.y4m` video (150 frames) is the fastest to encode at 132.79 seconds, while `akiyo_cif.y4m` (300 frames) takes slightly longer at 249.93 seconds.

The `deadline_cif.y4m` video (1374 frames) requires significantly more time, 1212.86 seconds, indicating scalability challenges for larger or more complex videos. These results emphasize the importance of optimizing processing efficiency to handle such cases effectively.

5.2.2 Compressing Ratios

Image

The compression results across the three test images reveal interesting patterns in the encoder's performance.

Image	Lossless	Compression Ratio	Space Saving
baboon.png	Yes	1.03:1	2.71%
peppers.png	No	1.22:1	18.03%
image.ppm	Yes	1.24:1	19.42%

Table 8: Compression Ratios for Images

The analysis of baboon.png shows modest compression gains, achieving a ratio of 1.03:1 and space savings of 2.71%. This relatively conservative compression outcome maintains lossless quality, preserving the image's full fidelity.

The results for peppers.png demonstrate more substantial compression, with a ratio of 1.22:1 yielding space savings of 18.03%. However, it's important to note that this improved compression comes at the cost of being lossy, indicating a trade-off between file size reduction and image quality preservation.

The most notable performance is observed with image.ppm, which achieved both lossless compression and the highest compression ratio of 1.24:1, resulting in space savings of 19.42%. This is particularly significant as it represents the best overall outcome, maintaining perfect image fidelity while delivering the most efficient compression among the test cases.

The varying results across different image types suggest that the encoder's effectiveness is influenced by both image content and format. The superior performance with the PPM format indicates that the encoder is particularly well-suited for handling raw image data, where it can achieve significant space savings without sacrificing image quality.

This pattern of results provides valuable insights for optimizing the encoder's application across different image types and use cases.

Video

Video	Compression Ratio
`akiyo_cif.y4m`	0.24:1
`bus_cif.y4m`	0.23:1
`deadline_cif.y4m`	0.13:1

Table 9: Compression Ratios for Videos

The compression ratios achieved during the encoding process varied significantly among the tested videos. The highest compression ratio was observed for `akiyo_cif.y4m` at 0.24:1, followed closely by `bus_cif.y4m` at 0.23:1.

The `deadline_cif.y4m` video demonstrated the lowest compression ratio of 0.13:1, likely due to its higher frame count (1374 frames) and smaller block size (8x8), which increases the overhead and reduces the efficiency of compression.

These results indicate a trade-off between video size and compression efficiency. While the higher frame count and smaller block sizes contribute to finer granularity, they also inflate the size of the encoded data.

5.2.3 Error Metrics

Image

Image	Lossless	PSNR (if applicable)
baboon.png	Yes	N/A
peppers.png	No	9.75 dB
image.ppm	Yes	N/A

Table 10: Error Measures for Images

The error measures for the tested images highlight the codec's flexibility in supporting both lossless and lossy compression. The baboon.png and image.ppm images were encoded losslessly, ensuring perfect reconstruction with no perceptual or numerical degradation.

In contrast, peppers.png underwent lossy compression, achieving a PSNR of 9.75 dB, indicating visible quality degradation while still retaining discernible content.

Video

Video	PSNR (dB)	MSE
`akiyo_cif.y4m`	11.92	-
`bus_cif.y4m`	14.35	-
`deadline_cif.y4m`	15.40	-

Table 11: Error Metrics for Videos

The error metrics for the videos (`akiyo_cif.y4m`, `bus_cif.y4m`, and `deadline_cif.y4m`) suggest a trade-off between quality and compression. The PSNR values indicate moderate degradation in video quality, with `deadline_cif.y4m` achieving the highest average PSNR of 15.40 dB, likely due to its configuration and frame complexity.

5.3 Comparative Analysis

Image

Our image encoder, based on predictive coding and Golomb encoding, offers a straightforward and educational approach to lossless compression, but it is relatively simplistic compared to industry-standard image encoders like PNG, JPEG-LS, or even modern lossy formats like JPEG or WebP. While it leverages predictive coding to minimize redundancy in image data, its capabilities are limited by fixed predictor selection and its reliance on Golomb coding, which is not as adaptive or sophisticated as entropy coders used in standard formats.

In comparison to PNG, a lossless standard widely used for images, our encoder lacks the flexibility and efficiency of PNG's Deflate algorithm, which combines LZ77 compression and Huffman coding to adaptively compress image data. PNG also supports advanced filtering techniques to improve compression ratios based on image content, whereas this encoder relies on a fixed prediction strategy that may not be optimal for all image types.

Compared to JPEG-LS, a specialized lossless and near-lossless standard, our encoder falls short in adaptability and compression performance. JPEG-LS employs context modeling and adaptive entropy coding to dynamically adjust to local image properties, achieving higher compression ratios while maintaining image fidelity. The encoder here uses a single value of the Golomb parameter mm per channel, which limits its ability to adapt to varying data distributions within the image.

Overall, this encoder serves as an educational tool that illustrates the basics of predictive coding and residual-based compression.

Video

Our video encoder implementation demonstrates a foundational approach to video compression, incorporating both lossy and lossless modes.

It features customizable parameters such as block size, quantization level, and target bitrate, allowing flexibility in compression settings. It supports both inter-frame (P-frame) and intra-frame (I-frame) compression in adherence to common practices in video encoding.

Additionally, the encoder evaluates performance metrics like PSNR and compression ratio, which align with the methods used to measure the quality of industry-standard codecs.

However, when compared to modern industry standards like H.264, H.265 (HEVC), and AV1, the implementation shows limitations in both efficiency and sophistication.

Advanced motion estimation and entropy coding techniques, such as CABAC in H.264/HEVC, are absent, which may result in suboptimal compression efficiency and larger file sizes.

Moreover, the encoder does not support features like B-frames, multi-pass encoding, or scalability for higher resolutions and HDR content, which are critical for contemporary applications.

Hardware optimization and energy efficiency, essential for practical deployment, are also not addressed, placing it at a disadvantage compared to codecs optimized for GPU or ASIC acceleration.

While functional for basic tasks, the encoder requires significant enhancements to compete with industry leaders in quality, speed, and resource efficiency.

5.4 Limitations and improvements

Image

Beyond the immediate scope, the encoder could benefit from multi-threaded processing to improve speed, particularly for high-resolution images.

Adaptive prediction models, potentially leveraging machine learning, could dynamically adjust to image characteristics for better residual minimization. Similarly, context-adaptive encoding, where predictors and parameters adjust to local properties, could enhance compression efficiency.

Memory usage poses a scalability challenge for very large images, as the implementation relies on in-memory processing. Hardware acceleration using GPUs or SIMD instructions and extending the encoder for video applications by addressing temporal redundancy are further opportunities for improvement.

For a broader scope, parallelization, advanced prediction techniques, memory-efficient strategies, and hybrid compression methods could take the encoder to the next level, enabling more efficient and versatile applications.

Video

The current video codec demonstrates a strong foundation but has limitations in scalability, efficiency, and advanced features.

Long encoding times for complex videos highlight the need for improved error handling and optimization. The lack of advanced techniques like adaptive quantization, bidirectional prediction (B-frames), and dynamic bitrate control reduces compression efficiency and quality adaptability.

Additionally, the codec does not support hardware acceleration, multi-threading, or error resilience, limiting its suitability for real-time or network-based applications.

Improvements include implementing advanced motion estimation, adaptive quantization, and entropy coding for better compression. Adding B-frame support, multi-pass encoding, and dynamic bitrate adjustment would enhance quality and efficiency.

6 Conclusion

In conclusion, throughout this report and project we studied in depth the encoding methods and the use of the compression standards taught during the semester.

By using both the taught methods and applying the principles from industry standards, we could experiment with different types of data, learning in the process the challenges and compromises involved in their encoding and decoding.