University of Aveiro

DETI

Information and Coding
Project nº2

Gonçalo Silva (103244) goncalolsilva@ua.pt
Samuel Teixeira (103325) steixeira@ua.pt
Tiago Alves (104110) tiagojba9@ua.pt

December 3, 2023

Index

List of Figures

Chapter 1

Introduction

The report here conducted follows the structure of the work guide. Part I approaches exercises 1-2. Part II handles exercise 3. Part III describes exercises 4-5. Part IV is about exercise 6. This project was developed using the GitHub platform for version control, it can be accessed at `https://github.com/detiuaveiro/ic_gts`

Chapter 2

Part I

2.1 Exercise 1

2.1.1 Modifications or Insertions in the code

The *Options* class condensates all the various configurable parameters needed thought the code into a shared data structure. Here we can see them and their default values.

```
namespace Options {
String inputFile = "../images/airplane.ppm";
String outputFile = "extractedColor.ppm";
uint8_t channel = 1;
bool showNewImage = false;
}
```

Listing 2.1: Options namespace

The *main* function handles all the processing logic and program flow. We start by checking the provided arguments, including the correct file extensions (done by *parse_args* function) and the supplied color channel. Then we create an empty image, extract the desired color channel of the supplied one and fill the values in the correct channel of the empty image. This functioning can be observed in the code below:

```
// Extract a color channel from an image, creating a single channel
     image with the result
int main(int argc, char** argv) {
    clock_t startTime = clock();

    int ret = parse_args(argc, argv);
    if (ret < 0)
        return 1;
    else if (ret > 0)
        return 0;

    if (Options::channel < 1 || Options::channel > 3) {
```

```cpp
        cerr << "Error: channel number must be between [1-3]: 1 (Blue), 2 "
                "(Green), or 3 (Red)\n";
        return 1;
    }
    Options::channel--;  // adjust for channel starting at 0

    Mat img = imread(Options::inputFile);
    if (img.empty()) {
        cerr << "Error: could not open/find image\n";
        return 1;
    }

    // Create a single-channel image with the specified channel
    cv::Mat outputImage = cv::Mat::zeros(img.rows, img.cols, CV_8UC1);

    for (int i = 0; i < img.rows; ++i) {
        for (int j = 0; j < img.cols; ++j) {
            /* Since outputImage is single-channel, each pixel is represented
                by an unsigned char. However, considering that the input image
                is a 3-channel image, we use the Vec3b vector class that already
                holds the 3 channels, with each being an 8-bit unsigned char that's
                why we use [] to index the correct channel */
            outputImage.at<uchar>(i, j) =
                img.at<cv::Vec3b>(i, j)[Options::channel];
        }
    }

    // Since it's expected a image with one channel, will have to convert it to
    //  grayscale
    cv::Mat outputGray;
    cv::cvtColor(outputImage, outputGray, cv::COLOR_GRAY2BGR);

    // Save the result to the output file
    cv::imwrite(Options::outputFile, outputGray);

    clock_t endTime = clock();

    std::cout << "Program took "
              << (double(endTime - startTime) / CLOCKS_PER_SEC) * 1000
              << " ms to extract Color channel " << Options::channel
              << " save image as " << Options::outputFile << std::endl;

    imshow("Extracted Color Channel", extractedChannel);
    waitKey(0); // wait for any key to be pressed

    return 0;
```

```
}
```

Listing 2.2: main function

2.1.2 Usage

To compile the code, navigate to the $class_2$ folder and execute the command:

```
make
```

Then navigate to the *bin* folder and launch the program using the following command:

```
./extract_image [OPTIONS]
```

To check the available options, use the -h option, like:

```
./extract_image -h
```

This command will generate the following text in the console:

```
Usage: %s [OPTIONS]
  OPTIONS:
  -h, --help         --- print this help
  -i, --input        --- set image file name (default: ../images/
    airplane.ppm)
  -o, --output       --- set extracted image file name (default:
    extractedColor.ppm)
  -c, --channel      --- set channel number, [default] 1 (Blue), 2 (
    Green), or 3 (Red)
  -s, --show         --- flag to show extracted image
```

Listing 2.3: wav_effects arguments

As present in the help menu, to set the input and output file names, use flags $-i$ or $-o$, with valid *.ppm* files. The $-c$ option allows to select the desired channel in the extracted image and the $-s$ flag is used to show the extracted image to the user, after generating (press any key to exit).

2.1.3 Results

Without using built-in functions of the *OpenCV* library, our program executes extremely fast, around 14.982 ms. In Figure 2.1b, we used the *peppers.ppm* image available in class and extracted the Green color channel, compared to the original Figure 2.1a, we can clearly see that the greens are more vibrant than other colors.

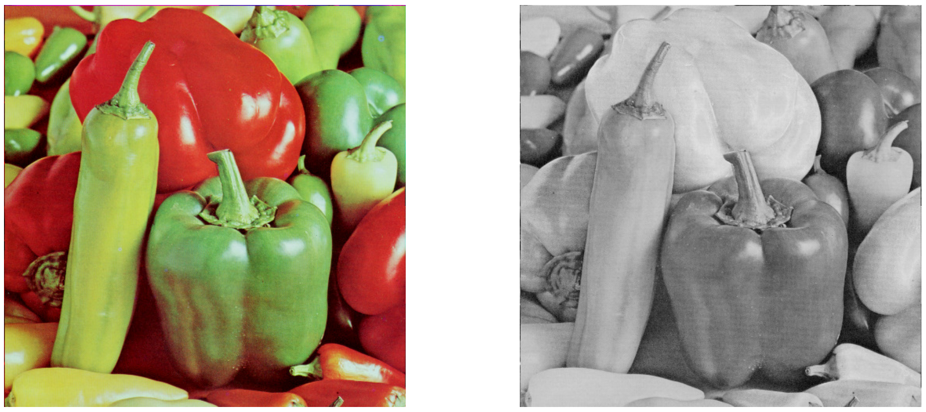

(a) Original Image (b) Green color channel extracted

Figure 2.1: Comparison between original image and the extracted green color channel

2.2 Exercise 2

2.2.1 Negative image

Modifications or Insertions in the code

The code for this exercise is pretty straight forward. We iterate through each byte of the image, considering both the rows and columns (multiplied by the number of channels, which is 3 in the case of an RGB model), and invert the value on that byte.

```
for (int i = 0; i < img.rows; i++)
{
    for (int j = 0; j < img.cols * img.channels(); j++)
    {
        img.at<uchar>(i, j) = 255 - img.at<uchar>(i, j);
    }
}
```

Listing 2.4: main function

Usage

To create the negative version of an image the following command can be used:

```
./negative_image <input file> <output file>
```

The input file is the image where the negative effect will be applied.

Results

We used the baboon.ppm image available in class and obtained the image in Figure 2.2. We can see that the colors in the image are inverted.

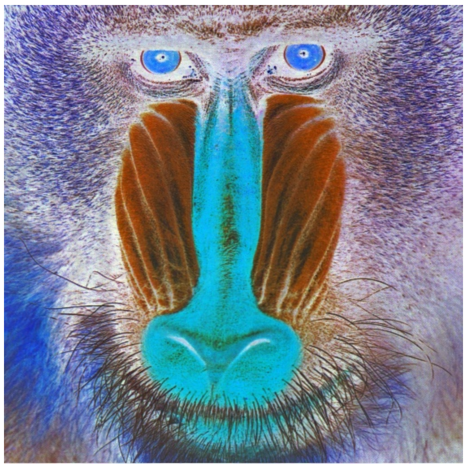


Figure 2.2: Negative version of image

2.2.2 Mirrored image

Modifications or Insertions in the code

To do the mirroring effect the image is divided in two halves. This division can be visualized as a line running through the middle of the columns for horizontal mirroring or through the middle of the rows for vertical mirroring. For each pixel in the first half of the image, represented by a three-byte vector, its value is swapped with the corresponding symmetrical position on the other side of the dividing line.

```cpp
if (option == "h")
    {
        for (int i = 0; i < img.rows; i++)
        {
            for (int j = 0; j < img.cols / 2; j++)
            {
                Vec3b aux = img.at<Vec3b>(i, j);
                img.at<Vec3b>(i, j) = img.at<Vec3b>(i, img.cols - j
     - 1);
                img.at<Vec3b>(i, img.cols - j - 1) = aux;
            }
        }
    }
    else
    {
        for (int i = 0; i < img.rows / 2; i++)
        {
            for (int j = 0; j < img.cols; j++)
            {
                Vec3b aux = img.at<Vec3b>(i, j);
                img.at<Vec3b>(i, j) = img.at<Vec3b>(img.rows - i -
    1, j);
                img.at<Vec3b>(img.rows - i - 1, j) = aux;
            }
        }
    }
```

Listing 2.5: main function

Usage

To create the mirrored version of an image the following command can be used:

```
./negative_image <input file> <option> <output file>
```

The $< input\_file >$ is the image where the mirroring effect will be applied. The mirroring can be done both horizontally and vertically so $< option >$ must be 'h' or 'v'.

Results

By applying horizontal mirroring on airplane.ppm we obtained Figure 2.3. Figure 2.4 shows the vertically mirrored image.

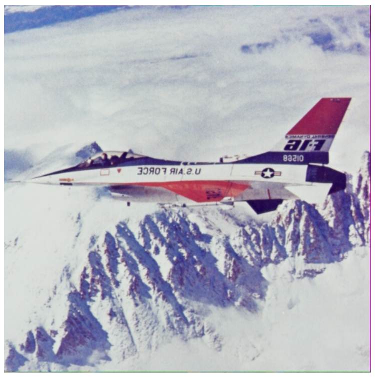


Figure 2.3: Horizontally mirrored image

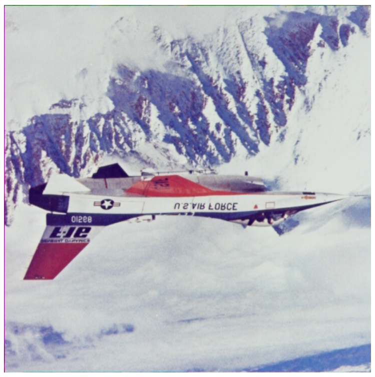


Figure 2.4: Vertically mirrored image

2.2.3 Rotated image

Modifications or Insertions in the code

This code is quite dificult to understand at first glance. The formating of the pixels varies according to the angle that is chosen. So in order to understand, let's break it down.

- **90º degrees -** For this rotation, we want the first row to be the last column of the original image, but with the order of the pixels inverted. The next rows will be the consequent next columns of the original image, until we reach the first one, that will be the last row of the rotated image.

- **180º degrees -** For this rotation, we want the first row to be last row of the original image, but with the order of the pixels inverted. The next rows will follow a similar order, until we reach the first one, that will be the last row of the rotated image.
- **270º degrees -** For this rotation, we want the first row to be the last column of the original image. The next rows will be the consequent next columns of the original image, until we reach the first one, that will be the last row of the rotated image.
- **360º degrees -** All images roatated by 360º or equivalent will just be equal to the original one.

```
if(angle % 360 == 0){
        //input image == output image so it does nothing mk
    }
    else if(angle % 270 == 0){
        // Espelhar verticalmente -> Troca valores das linhas
        for (int i = 0; i < img.rows-1; i++)
        {
            for (int j = 0; j < img.cols-1; j++)
            {
                rotated.at<Vec3b>(img.rows - j - 1,i) = img.at<
    Vec3b>(i, j);
            }

        }
    }
    else if (angle % 180 == 0){
        // Espelhar horizontalmente -> Troca valores das colunas
        for (int i = 0; i < img.rows; i++)
        {
            for (int j = 0; j < img.cols; j++)
            {
                //The first rows are equal to the inverse of the
    original last rows (i.e. row[0] = -(row[n]) )
                rotated.at<Vec3b>(img.rows - i - 1, img.cols - j -
    1) = img.at<Vec3b>(i,j);
            }
        }
    }
    else{
        for (int i = 0; i < img.rows; i++)
        {
            for (int j = 0; j < img.cols; j++)
            {
                rotated.at<Vec3b>(j, img.rows - i -1) = img.at<
    Vec3b>(i, j);
            }

        }

    }
```

Listing 2.6: main function

Usage

To create the rotated version of an image the following command can be used:

```
./rotated_image <input file> <multiple> <output file>
```

The $< input\_file >$ is the image where the rotation will be applied. The rotation is done in a multiple of 90,so the $< multiple >$ factor represents the angle of the rotation (i.e. 90º, 180º, 270º, 360º, etc...)

Results

By applying a 90º degree, 180º, and 270º rotations on airplane.ppm we obtained Figure 2.5, Figure 2.6 and Figure 2.7.

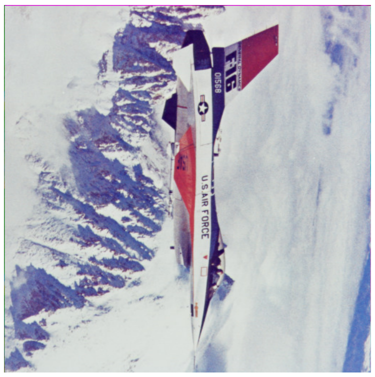


Figure 2.5: Rotated image by 1*90º (90º)

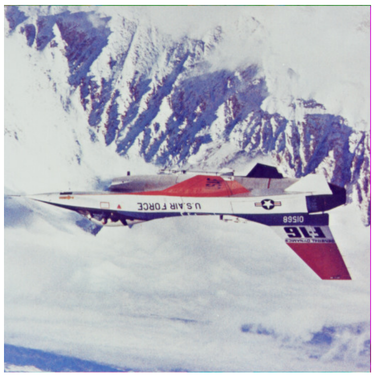


Figure 2.6: Rotated image by 2*90º (180º)

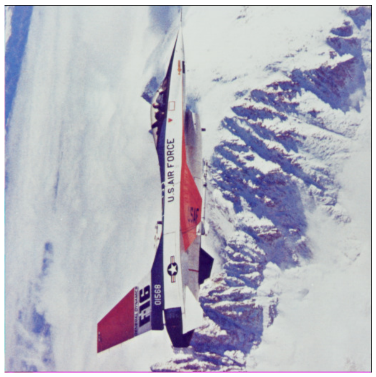


Figure 2.7: Rotated image by 3*90º (270º)

2.2.4 Bright image

Modifications or Insertions in the code

The bright effect is obtained by multiplying a constant to each pixel.

```
    for (int i = 0; i < img.rows; i++)
    {
        for (int j = 0; j < img.cols; j++)
        {
            img.at<Vec3b>(i,j) *= brightFactor;
        }

    }
```

Listing 2.7: main function

Usage

To create the bright version of an image the following command can be used:

```
./negative_image <input file> <bright-factor> <output file>
```

The $< input\_file >$ is the image where the bright effect will be applied. The $< bright - factor$ will make the image more or less bright (the smaller the less bright and vice-versa).

Results

By applying a $< bright - factor >$ of 1.5 to the original image we get the brighter image Figure 2.8 and by applying a $< bright - factor >$ of 0.5 we get the darker image Figure 2.9.

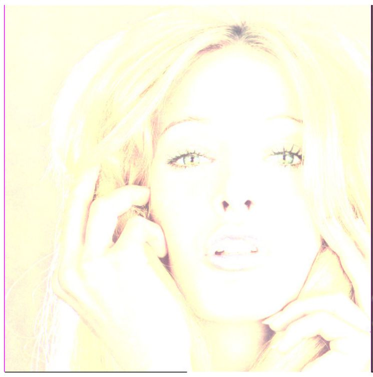


Figure 2.8: Brighter Image

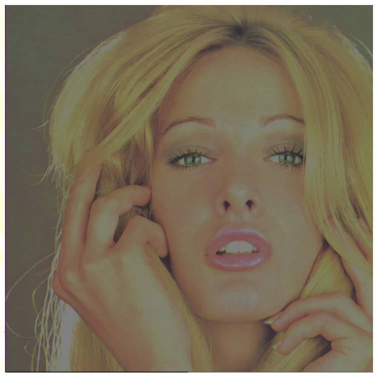


Figure 2.9: Darker Image

Chapter 3

Part II

3.1 Exercise 3

3.1.1 Code organization

We decided to divide the golomb class code into two files, "golomb.h" and "golomb.cpp". The constructor and encode, decode and check_approach methods are implemented in "golomb.cpp". The class has three parameters: distribution parameter m, an instance of the Bitstream class and an approach for handling negative numbers. By passing the Bitstream object to the class, we are able to integrate bitstream utilization during both encoding and decoding operations. The "golomb.h" file looks like this:

```
// All the includes...
enum APPROACH { SIGN_MAGNITUDE, VALUE_INTERLEAVING };

bool check_approach(APPROACH approach);

std::string approach_to_string(APPROACH approach);

class Golomb {
   private:
    int m;  // distribution
    APPROACH approach = SIGN_MAGNITUDE;
    BitStream& bitStream;

    void write_remainder(int r);
    void encode_sign_magnitude(int value);
    void encode_value_interleaving(int value);

   public:
    Golomb(int m, BitStream& bitStream, APPROACH approach =
    SIGN_MAGNITUDE);

    ~Golomb() { flush(); }

    void flush() { bitStream.~BitStream(); }

```

```cpp
    void setM(int newM) { this->m = newM; }
    int getM() { return m; }

    void setApproach(APPROACH newApproach) { this->approach =
    newApproach; }
    int getApproach() { return approach; }

    void encode(int i);

    int decode();
};
```

Listing 3.1: golomb.h

Notably, a destructor has been introduced to the Golomb class, responsible for flushing and finalizing the associated BitStream. Setters and getters were also added for both m and approach parameters.

3.1.2 Encode function

The encode function will be different depending on the approach chosen, so the encode function calls one of other two functions, one for each approach.

```cpp
void Golomb::encode(int value) {
    if (m <= 0)
        throw std::invalid_argument("m must be positive");

    if (approach == SIGN_MAGNITUDE)
        encode_sign_magnitude(value);
    else if (approach == VALUE_INTERLEAVING)
        encode_value_interleaving(value);
    else
        throw std::invalid_argument("Invalid approach");
}
```

Listing 3.2: Encode function

By using Golomb coding we will have two parts to represent the encoded integer, one unary part for the quotient and a binary one for the remainder. The result obtained will be directly written using a bitstream to be more efficient.

On both approaches, as the quotient is represented using unary code, the count of ones corresponds to the integer value of the quotient. To separate the unary code from the binary part a '0' is added after the '1's.

```cpp
int quotient = value / m;
int remainder = value % m;

// Creating the unary with as many 1s as the quotient
for(int i = 0; i < quotient; i++)
    bitStream.writeBit(1);

// Insert 0 to separate the quotient from the remainder
bitStream.writeBit(0);
```

Listing 3.3: Quotient encoding

The calculation of the binary remainder differs based on whether the parameter m is power of 2 or not. Lets start by looking how to get the remainder for a m that is power of 2. In this case, all the possible remainder values are represented by the same number of bits, denoted as b, and given by $b = \lceil \log m \rceil$.

When m is not power of 2 the number of bits is still determined by $b = \lceil \log m \rceil$, but a truncated binary code is implemented in order to save some bits. If the remainder value is lower than $2^b - m$ it is represented using only $b - 1$ bits. Otherwise, we sum $2^b - m$ to the remainder value and the resulting value is represented using b bits.

```
// Remainder calculated in both approaches
int remainder = value % m;

write_remainder(remainder);

void Golomb::write_remainder(int remainder) {
    int b = ceil(log2(m));

    if ((m & (m - 1)) == 0){ // if m is power of 2
        bitStream.writeNBits(remainder, b);
    }else{ // m is not power of 2
        if (remainder < (pow(2, b) - m)){
            bitStream.writeNBits(remainder, b-1);
        }else{
            remainder += pow(2, b) - m;
            bitStream.writeNBits(remainder, b);
        }
    }
}
```

Listing 3.4: Remainder encoding

It is also important to take the handling of negative values into consideration.

If "Sign and Magnitude" approach is chosen and the inputted value is negative, the calculation of the unary and binary parts is done as if the value was positive. After the encoding process of the quotient and the remainder we check if the value was negative. If so, a '1' is appended to the Golomb code, otherwise, a '0' is added. It is also worth to mention that if the integer to be encoded is zero, an additional bit is saved, since it doesn't require sign representation.

```
void Golomb::encode_sign_magnitude(int value) {
    bool isNegative = (value < 0) ? true : false;
    value = abs(value);

    int quotient = value / m;
    int remainder = value % m;

    // Quotient and remainder encoding...

    if (value == 0){
        return;
    }
    if (isNegative)
        bitStream.writeBit(1);
    else
```

```
        bitStream.writeBit(0);
}
```

Listing 3.5: Encode Sign and Magnitude function

Finally, in the "Value Interleaving" approach, if the received number is negative, it is converted to a positive value, multiplied by 2, and 1 is subtracted. This way negative numbers always correspond to odd numbers during the encoding process, while positive values are only multiplied by 2 so they will always be even. After this conversion the encoding is done.

```
void Golomb::encode_value_interleaving(int value) {
    // if number is negative, multiply by 2 and subtract 1
    if (value < 0)
        value = 2 * abs(value) - 1;
    else
        value = 2 * value;

    int quotient = value / m;
    int remainder = value % m;

    // Quotient and remainder encoding...
}
```

Listing 3.6: Encode Value Interleaving function

3.1.3 Decode function

To decode a sequence of bits into an integer, the process begins by reading these bits using a bitstream. The integer quotient and remainder values will be crucial to determine the final decoded value.

In the decoding process, the unary part of the encoded number is obtained by reading bit by bit until a '0', that represents the separation of the unary part from the binary part, is found. During this reading process, the quotient value is incremented, ensuring that when a '0' is found, the integer value of the quotient is available.

```
int Golomb::decode() {

    int value = 0;
    int quotient = 0;
    int remainder = 0;

    // read the unary part until a 0 is found
    while (true) {
        int bit = bitStream.readBit();
        if (bit == 0)
            break;
        quotient++;
    }

    //...
```

Listing 3.7: Decode function

When m is power of 2, the binary part of the Golomb-encoded number always uses the same length of bits for representation. However, when m is not power of 2, the number of bits needed to represent the binary part can either be b or $b-1$. The approach we chose to determine the number of bits that need to be read is to first read $b-1$ bits, and obtain an integer value. If this value is equal or higher than $2^b - m$ an extra bit needs to be read because the binary part is represented using b bits, otherwise, the value obtained already corresponds to the wanted binary remainder.

```
int b = ceil(log2(m));
int values_divider = pow(2, b) - m;

// read the binary part
for (int i = 0; i < b - 1; i++) {
    int bit = bitStream.readBit();
    remainder = (remainder << 1) + bit;
}

int extraBit = 10;
if (remainder >= values_divider) {
    remainder = (remainder << 1) + bitStream.readBit();
    extraBit = 1;
}
```

Listing 3.8: Decode function

Additionally, if an extra bit read was read during the decoding process, $2^b - m$ is subtracted to the remainder ($2^b - m$ was summed to this remainder during the encoding process).

```
if (extraBit != 10) {
        remainder = remainder - values_divider;
    }
```

Listing 3.9: Decode function

With the obtained quotient and remainder, the decoded value is computed using $value = quotient * m + remainder$. The final decoded value is then determined based on the chosen approach for handling negative numbers. For "Sign and Magnitude" approach, an additional bit is read to determine the signal of the number, unless the value is 0. In "Value Interleaving", the decoding process is adapted based on the parity of the number: if it's odd, the opposite of the encoding process is done, by adding 1, dividing it by 2 and negate it; if it's even, it is simply divided by 2.

```
value = quotient * m + remainder;

if (value == 0) {   // no sign
    return value;
} else if (approach == SIGN_MAGNITUDE) {

    int sign = bitStream.readBit();  // Last bit is the sign
    if (sign == 1)
        value = -value;
    return value;

} else if (approach == VALUE_INTERLEAVING) {

    if (value % 2 == 1) {  // if number is odd, add 1 and divide by
     2
        value++;
        value /= 2;
        value = -value;
    } else {
        value /= 2;
    }
    return value;
} else
    std::invalid_argument("Invalid approach");
```

Listing 3.10: Decode function

3.1.4 Tests

We did 12 tests in total, 6 for each approach, all consisting on encoding a sequence of numbers and then decoding them to compare this ones with the initial ones. The values and parameters chosen for each one of the 6 tests were:

- Small positive values (0 to 200) and m power of 2
- Small negative values (-200 to 0) and m power of 2
- Bigger values (-32764 to 32764) and m power of 2
- Small positive values (0 to 200) and m not power of 2
- Small negative values (-200 to 0) and m not power of 2
- Bigger values (-32764 to 32764) and m not power of 2

The code of the first test for "Sign and Magnitude" approach is presented below.

```
TEST(Golomb, smallValues_SignMagnitude_mPowerOf2) {

    const int MAX_VALUE = 200;
    int m = 4;

    BitStream writer('w', TEST_FILE_NAME);
    Golomb golomb(m, writer);
    std::list<int> values;
    for (int i = 0; i <= MAX_VALUE; i++) {
        golomb.encode(i);
        values.push_back(i);
    }
    golomb.~Golomb();

    BitStream reader('r', TEST_FILE_NAME);
    Golomb golomb2(m, reader);
    for (int i = 0; i <= MAX_VALUE; i++) {
        int decoded = golomb2.decode();
        int original = values.front();

        values.pop_front();
        EXPECT_EQ(decoded, original);
    }
    EXPECT_EQ(values.size(), 0);

    golomb2.~Golomb();

    remove(TEST_FILE_NAME);
};
```

Listing 3.11: First test

All the other tests followed the same structure. The results were decoded values equal to the initial ones, as expected.

Chapter 4

Part III

4.1 Exercise 4

4.1.1 Data Structures

The *Options* class condensates all the various configurable parameters required throughout the code into a shared data structure. Here we can see them and their default values.

```cpp
namespace Options {
string musicName = "../songs/sample_mono.wav";
string encodedName = "encodedSample";
size_t blockSize = 1024;
size_t bitRate = -1;
int m = -1;  // automatic
bool lossy = false;
size_t nChannels;
size_t nFrames;
size_t sampleRate;
PREDICTOR_TYPE predictor = AUTOMATIC;
APPROACH approach = SIGN_MAGNITUDE;
PHASE phase = P_AUTOMATIC;
}
```

Listing 4.1: Options namespace

To better organize the code and data that our program relies on we decided to create two data structures. The **File** represents the encoded file contents, with it's various required values. This structure is used inside our *Codec* classes to make decisions and various calculations. A file is composed of a Header and a set of *Blocks*. The number of *Blocks* is calculated using the formula:

$$\lceil \frac{nFrames}{\text{blockSize}} \rceil \times \text{nChannels}$$

```cpp
struct File {
    /* Header */
    uint16_t blockSize;
    uint8_t nChannels;
    uint16_t sampleRate;
    uint32_t nFrames;
    uint16_t bitRate;
    APPROACH approach;
    bool lossy;  // true if lossy, false if lossless
    /* Data */
    vector<Block> blocks;
};
```

Listing 4.2: File Structure

A **Block** contains a Header for customizable information and a *vector* of samples. The number of samples present is directly correlated to the *blockSize* present in the **File** structure. Note that we are talking about samples and not *frames*, since each frames in stereo audio contains two samples. This means that for a Block size of 1024, with mono audio, the *Block* will hold 1024 samples and frames. But with stereo audio, the *Block* will hold 1024 samples and 512 frames. Meaning that stereo audio will have double the samples, thus double the blocks, but same numbers of frames as a mono channel audio.

```
struct Block {
    /* Header */
    uint16_t m;
    uint8_t phase;
    PREDICTOR_TYPE predictor;
    /* Data */
    vector<short> data;
};
```

Listing 4.3: Block Structure

We recognize that the inclusion of a *Data* field in the **Block** may seem unnecessary, because when the data is encoded, it could be written directly to the file. However in our view, the processing power and memory saved would be negligible. Our approach offers consistency (we only write to the file in one function, that writes all header and data at once), simplicity and safety for the data to be manipulated by other functions (calculation of m, prediction calculation and others).

4.1.2 Golomb Codec

The *golomb_codec.h* and *golomb_codec.cpp* files contains the main parts of our Encoding algorithm using Golomb. In here, the structure is divided in three classes: in section 4.1.2, section 4.1.2 and section 4.1.2 we introduce Predictor, GEncoder and GDecoder classes, respectively.

Predictor class

The `Predictor` class manages prediction algorithms for audio data based on different predictor types and phases/correlation. The predictors used were the ones lectured in class for audio files, whilst phase/correlation idea was derived from the fact that in stereo audio files, if the left and right channel are correlated, the occupied space can be reduced if they are encoded together.

```cpp
enum PREDICTOR_TYPE { AUTOMATIC, PREDICT1, PREDICT2, PREDICT3 };

enum PHASE { P_AUTOMATIC, NO_CORRELATION, CORRELATION_MID, CORRELATION_SIDE };

std::string get_type_string(PREDICTOR_TYPE type);

std::string get_phase_string(PHASE phase);

class Predictor {
   private:
    /*! a1, a2 and a3 represent, a(n-1) a(n-2) and a(n-3) respectively
            where n is the index of the predicted sample */
    int predict1(int a1);
    int predict2(int a1, int a2);
    int predict3(int a1, int a2, int a3);

    double calculate_entropy(PREDICTOR_TYPE type, PHASE phase,
                             std::vector<short>& samples);

    int predict_no_correlation(PREDICTOR_TYPE type, std::vector<short>& samples,
                               int index);

    int predict_correlated_mid(PREDICTOR_TYPE type, std::vector<short>& samples,
                               int index);
    int predict_correlated_side(PREDICTOR_TYPE type,
                                std::vector<short>& samples, int index);

    int nChannels = 1;

   public:
    Predictor(int nChannels);
    Predictor();
    ~Predictor();

    /*!
        Pass a set of samples/block and return the best predictor to be used
            (the one that resulted in less occupied space)
    */
    std::tuple<PREDICTOR_TYPE, PHASE> benchmark(
        std::vector<short>& samples, PREDICTOR_TYPE bestPredictor = AUTOMATIC,
        PHASE bestPhase = P_AUTOMATIC);

    /*!
        Predict the next sample based on the type of the predictor and the
            previous samples
    */
    int predict(PREDICTOR_TYPE type, PHASE phase, std::vector<short
    >& samples,
                int index);

    bool check_type(PREDICTOR_TYPE type);
    bool check_phase(PHASE phase);
    void set_nChannels(int nChannels);
    int get_nChannels();
};
```

Listing 4.4: Predictor class declarations

Prediction algorithms (`predict1`, `predict2`, and `predict3`) calculate predicted values based on different input samples. These algorithms are utilized according to the specified predictor type and phase. The data encoded in the file isn't the predicted value itself, but the difference/error. Knowing that the current sample is a_n and $a_0 = 0$, then the following equations are valid for the respective predictor:

$$\texttt{Predictors} = \begin{cases} \text{Predict}_1 : & a_n = mid_1 \\ \text{Predict}_2 : & a_n = 2a_{n-1} - a_{n-2} \\ \text{Predict}_3 : & a_n = 3a_{n-1} - 3a_{n-2} + a_{n-3} \end{cases}$$

```
int Predictor::predict(PREDICTOR_TYPE type, PHASE phase,
                       std::vector<short>& samples, int index) {
    if (!check_type(type) || type == AUTOMATIC) {
        cerr << "Error: Unknown/Invalid Predictor " << unsigned(
    type)
             << " encountered while decoding Block" << endl;
        exit(2);
    }

    if (!check_phase(phase) || phase == P_AUTOMATIC) {
        cerr << "Error: Unknown/Invalid Phase " << unsigned(phase)
             << " encountered while decoding Block" << endl;
        exit(2);
    }

    if (phase == NO_CORRELATION || nChannels == 1)
        return predict_no_correlation(type, samples, index);
    else if (phase == CORRELATION_MID)
        return predict_correlated_mid(type, samples, index);
    else  // correlation SIDE
        return predict_correlated_side(type, samples, index);
}
```

Listing 4.5: predict function

Functions like `predict_no_correlation`, `predict_correlated_mid`, and `predict_correlated_side` handle prediction in scenarios where correlation is present in the audio data, adjusting the prediction based on the phase.

```
int Predictor::predict_no_correlation(PREDICTOR_TYPE type,
```

```cpp
                                                std::vector<short>& samples,
    int index) {
    // this code supports both mono and stereo sound
    int a1, a2, a3;

    if (nChannels == 1) {
        a1 = (index - 1) < 0 ? 0 : samples.at(index - 1);
        a2 = (index - 2) < 0 ? 0 : samples.at(index - 2);
        a3 = (index - 3) < 0 ? 0 : samples.at(index - 3);
    } else {
        // Go two Idx back, because of getting the previous value
    of the same channel
        a1 = (index - 2) < 0 ? 0 : samples.at(index - 2);
        a2 = (index - 4) < 0 ? 0 : samples.at(index - 4);
        a3 = (index - 6) < 0 ? 0 : samples.at(index - 6);
    }

    if (type == PREDICT1)
        return predict1(a1);
    else if (type == PREDICT2)
        return predict2(a1, a2);
    else
        return predict3(a1, a2, a3);
}
```

Listing 4.6: predict_no_correlation function

It is important remembering that the sample vector is structured with interleaved values for the Left and Right samples in stereo audio, i.e L R L R, etc. Encoding with no correlation, which is the only option for mono channel audio files, is achieved by just analyzing the previous samples samples of the respective channel. For example, encoding the mono channel with predictor 3 uses the above predictor equation, but for stereo audio, the following equation is used, being a_n the predicted value of mono, left or right channel:

$$a_n = 3a_{n-2} - 3a_{n-4} + a_{n-6}$$

Exclusive to stereo audio is the encoding method involving channel correlation, which we obtain by calculating the *Mid* and *Side* channels and then applying the configured predictor on those values. With a_n representing the predicted value for a sample (being it from the Left or Right channel), we obtain the following equations:

$$\texttt{Mid Channel} = \frac{\text{Left} + \text{Right}}{2}$$

```cpp
int Predictor::predict_correlated_mid(PREDICTOR_TYPE type,
                                                std::vector<short>& samples,
    int index) {
    std::vector<int> an;
    for (int i = 1; i <= 5; i += 2) {
        // odd index
        int channel1 = 0;
        if ((index - i) > (int)samples.size())
```

```
            channel1 = samples.at(index - i);

        // even index
        int channel2 = 0;
        if ((index - (i - 1)) > (int)samples.size())
            channel2 = samples.at(index - (i - 1));

        an.push_back((channel1 + channel2) / 2);
    }

    if (type == PREDICT1)
        return predict1(an.at(0));
    else if (type == PREDICT2)
        return predict2(an.at(0), an.at(1));
    else
        return predict3(an.at(0), an.at(1), an.at(2));
}
```

Listing 4.7: predict_correlated_mid function

$$\texttt{Side Channel} = \frac{\text{Left} - \text{Right}}{2}$$

```
int Predictor::predict_correlated_side(PREDICTOR_TYPE type,
                                        std::vector<short>& samples,
     int index) {
    std::vector<int> an;
    for (int i = 1; i <= 5; i += 2) {
        // odd index
        int channel1 = 0;
        if ((index - i) > (int)samples.size())
            channel1 = samples.at(index - i);

        // even index
        int channel2 = 0;
        if ((index - (i - 1)) > (int)samples.size())
            channel2 = samples.at(index - (i - 1));

        an.push_back((channel1 - channel2) / 2);
    }

    if (type == PREDICT1)
        return predict1(an.at(0));
    else if (type == PREDICT2)
        return predict2(an.at(0), an.at(1));
    else
        return predict3(an.at(0), an.at(1), an.at(2));
}
```

Listing 4.8: predict_correlated_side function

Determining the best (the one with the highest *hit/miss* ratio) Predictor and Phase/Correlation for each block of samples can be obtained by simulating the **Blocks** raw data. That is accomplished by the `benchmark` function which tests various combinations to identify the best predictor and phase combination that minimizes entropy for a given set of samples. It iterates through predictor types and phases to find the combination that yields the lowest entropy, implementing optimization strategies for efficiency. Using the `calculate_entropy` function, we compute the entropy based on a given predictor type, phase, and a vector of samples. The probability of a predictor determining the correct value is given by P, with i representing the cycle of the various Predictors:

$$\mathrm{P}_i = \frac{\text{number of correct predictions}}{\text{total predictions}}$$

With the Entropy can be calculated using the Shannon's entropy formula:

$$\text{Entropy} = -\sum_i \mathrm{P}_i \cdot \log_2(\mathrm{P}_i)$$

```

std::tuple<PREDICTOR_TYPE, PHASE> Predictor::benchmark(
    std::vector<short>& samples, PREDICTOR_TYPE bestPredictor,
    PHASE bestPhase) {
    double min_entropy =
        std::numeric_limits<double>::max();  // Initialize to max

    PREDICTOR_TYPE initialPredictor = bestPredictor;
    PHASE initialPhase = bestPhase;

    // I know, kinda gross... This code should be optimized
    for (int p = NO_CORRELATION; p <= CORRELATION_SIDE; ++p) {
        PHASE phase;
        if (initialPhase != P_AUTOMATIC)
            phase = initialPhase;
        else
            phase = static_cast<PHASE>(p);

        if (initialPredictor == AUTOMATIC || initialPredictor ==
    PREDICT1) {
            double entropy1 = calculate_entropy(PREDICT1, phase,
    samples);
            if (entropy1 <= min_entropy) {
                min_entropy = entropy1;
                bestPredictor = PREDICT1;
                bestPhase = phase;
            }
        }

        if (initialPredictor == AUTOMATIC || initialPredictor ==
    PREDICT2) {
            double entropy2 = calculate_entropy(PREDICT2, phase,
    samples);
            if (entropy2 <= min_entropy) {
                min_entropy = entropy2;
                bestPredictor = PREDICT2;
                bestPhase = phase;
            }
        }

        if (initialPredictor == AUTOMATIC || initialPredictor ==
    PREDICT3) {
            double entropy3 = calculate_entropy(PREDICT3, phase, samples);
            if (entropy3 <= min_entropy) {
                min_entropy = entropy3;
                bestPredictor = PREDICT3;
                bestPhase = phase;
            }
        }

        // No need to go further, since only no_correlation is supported for mono audio
        //  or the phase wasn't automatic
        if (nChannels == 1 || initialPhase != P_AUTOMATIC)
            break;
    }

    // Use the initial phase if bestPredictor is set to AUTOMATIC
    if (initialPhase != P_AUTOMATIC && initialPredictor == AUTOMATIC &&
        nChannels > 1) {
        bestPhase = initialPhase;
    }

    // Use the initial predictor if bestPhase is set to P_AUTOMATIC
    if (initialPhase == P_AUTOMATIC && initialPredictor != AUTOMATIC &&
        nChannels > 1) {
        bestPredictor = initialPredictor;
    }

    return std::make_tuple(bestPredictor, bestPhase);
}
```

Listing 4.9: benchmark function

```
double Predictor::calculate_entropy(PREDICTOR_TYPE type, PHASE phase,
                                    std::vector<short>& samples) {
    int total_predictions = 0;
    std::vector<int> predictions;

    // Predict based on the given type and count occurrences of predictions
    for (size_t i = 0; i < samples.size(); ++i) {
        int prediction;
        if (type == PREDICT1) {
            prediction = predict(type, phase, samples, i);
        } else if (type == PREDICT2) {
            prediction = predict(type, phase, samples, i);
        } else if (type == PREDICT3) {
            prediction = predict(type, phase, samples, i);
        } else {
            cerr << "Error: Unknown Predictor type encountered" << endl;
            exit(2);
        }
        predictions.push_back(prediction);
        total_predictions++;
    }

    // Calculate probability distribution and entropy
    double entropy = 0.0;
    for (int value : predictions) {
        double probability =
            std::count(predictions.begin(), predictions.end(),
    value) /
            static_cast<double>(total_predictions);
        entropy -= probability * std::log2(probability);
    }

    return entropy;
}
```

Listing 4.10: calculate_entropy function

GEncoder class

The `GEncoder` class implements our data encoding, designed to compress and store information in a specific file format. It uses the *Predictor*, *Golomb* and *bitStream* classes to be able to store the data with the smallest size required, without losing any data (lossless), although already has available functions and parameters for lossy encoding.

```
class GEncoder {
   private:
    BitStream writer;
    Golomb golomb;
    PREDICTOR_TYPE predictor = AUTOMATIC;
    Predictor predictorClass;
    PHASE phase = P_AUTOMATIC;
    int m;

    std::string outputFileName;
    File fileStruct;

    std::vector<unsigned short> abs_value_vector(std::vector<short
    >& values);
    int calculate_m(std::vector<short>& values);
    Block process_block(std::vector<short>& block, int blockId, int
     nBlocks);
    void write_file();

   public:
    GEncoder(std::string outFileName, int m, PREDICTOR_TYPE pred,
    PHASE phase);
    ~GEncoder();

    void encode_file(File file, std::vector<short>& inSamples,
    size_t nBlocks);

    // Stuff used for testing private members
    int test_calculate_m(std::vector<short>& values);
    std::vector<unsigned short> test_abs_value_vector(
        std::vector<short>& values);
};
```

Listing 4.11: GEncoder class declarations

This class uses both values defined directly in the constructor, such as the output file name (given to the *bitStream*), *m* value (given to the *Golomb*), predictor type, and phase/correlation as well as configuration values provided in the `encode_file` function, by a **File** structure. Subsequently, it configures the *Golomb* and upon termination, the destructor ensures proper cleanup, clearing the *Golomb* object, ensuring resource management and memory deallocation.

The life-cycle of this class starts in the `encode_file` function, where the samples of a music and a **File** structure with the header values filled are supplied. This function will divide the samples into a smaller set of values (according to the *blockSize* and *nBlocks*) to be processed in each *Block* by the `process_block` method. Upon the processing of all *Blocks*, it calls the `write_file` method, which will write the data and headers into the output file.

```
void GEncoder::encode_file(File file, std::vector<short>& inSamples
    ,
                           size_t nBlocks) {
    this->fileStruct = file;
    this->predictorClass.set_nChannels(file.nChannels);

    std::cout << "Entering encoding phase" << std::endl;
    // Divide in blocks and process each one
    for (int i = 0; i < (int)nBlocks; i++) {
        std::vector<short> block;
        for (int j = 0; j < file.blockSize; j++)
            block.push_back(inSamples[i * file.blockSize + j]);

        Block encodedBlock = process_block(block, i, nBlocks);

        // Add encoded block to file
        fileStruct.blocks.push_back(encodedBlock);
    }
    if (predictor == AUTOMATIC)
        std::cout << "\n";

    std::cout << "All blocks encoded. Writing to file" << endl;

    write_file();
}
```

Listing 4.12: encode_file function

The `process_block` method accepts a subset of samples and checks whether the user has provided default values for the `Predictor` or `Phase/Correlation`. If default values are provided, they are used to predict the samples. Otherwise, the method calls the `benchmark` function from the `Predictor` class to calculate the most suitable combination for this particular `Block` of data. Subsequently, the function proceeds to compute the **error** to be encoded into the file, as determined by the following equation:

$$\text{error}_i = \sum_{i=1}^{\text{block\_size}} (\text{block}[i] - \text{prediction})$$

With the error obtained, we'll check if the user provided a default value for the parameter m and place it in the *Block* structure. Otherwise, the best m will be calculated using the `calculate_m` function. This function will return a *Block* containing the errors to be written in the file and the specified parameters that they were calculated with, in the header.

```cpp
Block GEncoder::process_block(std::vector<short>& block, int
    blockId,
                              int nBlocks) {

    PREDICTOR_TYPE pred = predictor;
    PHASE ph = phase;
    if (pred == AUTOMATIC || ph == P_AUTOMATIC) {
        std::cout << " - "
                  << "Benchmarking predictor for Block " << std::
    setw(3)
                  << blockId + 1 << "/" << nBlocks << "\r" << std::
    flush;
        std::tuple<int, int> ret = predictorClass.benchmark(block,
    pred, ph);
        pred = static_cast<PREDICTOR_TYPE>(std::get<0>(ret));
        ph = static_cast<PHASE>(std::get<1>(ret));
    }

    Block encodedBlock;
    encodedBlock.predictor = pred;
    encodedBlock.phase = ph;

    for (int i = 0; i < (int)block.size(); i++) {
        int prediction = predictorClass.predict(pred, ph, block, i)
    ;
        int error = block.at(i) - prediction;

        encodedBlock.data.push_back(error);
    }

    // Use attributed m or calculate one
    int bM = m;
    if (bM < 1)
        bM = calculate_m((encodedBlock.data));
    encodedBlock.m = bM;

    return encodedBlock;
}
```

Listing 4.13: process_block function

As the name suggests, the `calculate_m` determines the optimal m parameter, based on the data to be encoded (error), used in the *Golomb* class, which is vital for decreasing the number of information stored. We start by determining the α, which uses the mean of the data and since the values will always be encoded as positives, the `abs_value_vector` function makes the absolute value of the vector, so that the α parameter is optimal. The equations for α and m are as follows:

$$err\bar{r}or = \frac{1}{n}\sum_{i=1}^{n} |\text{error}[i]|$$

$$alpha = \exp\left(-\frac{1.0}{\text{er}\bar{\text{r}}\text{or}}\right)$$

$$m = \left\lceil -\frac{1}{\log(\alpha)} \right\rceil$$

The resulting value is capped, ensuring the minimum value is 1 and the maximum is 16383, which is $2^{14} - 1$.

```
int GEncoder::calculate_m(std::vector<short>& values) {
    /* Calculate alpha */
    double alpha = 1.0;
    if (!values.empty()) {
        std::vector<unsigned short> abs_values = abs_value_vector(
    values);
        double mean = static_cast<double>(std::accumulate(
                          abs_values.begin(), abs_values.end(), 0))
     /
                      abs_values.size();
        alpha = exp(-1.0 / mean);  // Calculate alpha using mean
    }
    // Calculate 'm' based on the formula
    double aux = (-1 / log(alpha));
    int m = std::ceil(aux);

    // guarantee that minimum value is 1
    m = std::max(1, m);

    return std::min(m, 16383);  // cap golomb max size (14 bits)
}
```

Listing 4.14: calcute_m function

The last function in our class life-cycle is the `write_file` method. This the only function responsible for interacting directly with the *bitStream* and *Golomb* for writing data into the file. It reads the **File** structure and starts by writing the Header, then reading every associated **Block**, it'll write it's associated Header and data into the binary file, using the *bitStream* for the Headers and *Golomb* for the encoded data. Macros were created to easily define the respective bit amount for each field in the header.

```
#define BITS_BLOCK_SIZE 16
#define BITS_SAMPLE_RATE 16
#define BITS_N_FRAMES 32
#define BITS_N_CHANNELS 4
#define BITS_BIT_RATE 8
#define BITS_LOSSY 2
#define BITS_APPROACH 4
```

```
#define BITS_M 14
#define BITS_PHASE 2
#define BITS_PREDICTOR 2
```

Listing 4.15: Sizes of Header fields in the encoded file

```
void GEncoder::write_file() {

    // Write file header_values
    writer.writeNBits(fileStruct.blockSize, BITS_BLOCK_SIZE);
    writer.writeNBits(fileStruct.sampleRate, BITS_SAMPLE_RATE);
    writer.writeNBits(fileStruct.nFrames, BITS_N_FRAMES);
    writer.writeNBits(fileStruct.nChannels, BITS_N_CHANNELS);
    writer.writeNBits(fileStruct.bitRate, BITS_BIT_RATE);
    writer.writeNBits(fileStruct.lossy, BITS_LOSSY);
    writer.writeNBits(fileStruct.approach, BITS_APPROACH);
    golomb.setApproach(fileStruct.approach);

    // Write Blocks (data)
    std::cout << "\nWriting data to file..." << endl;

    int count = 1;
    for (auto& block : fileStruct.blocks) {
        // check if block size is correct
        if (block.data.size() != fileStruct.blockSize) {
            cerr << "Error: Block size mismatch" << endl;
            exit(2);
        }

        // Write Block header
        writer.writeNBits(block.m, BITS_M);
        writer.writeNBits(block.phase, BITS_PHASE);
        writer.writeNBits(block.predictor, BITS_PREDICTOR);

        golomb.setM(block.m);  // DONT FORGET THIS

        // Write Block data
        std::cout << " - Writing Block " << std::setw(3) << count++
 << "/"
                  << std::setw(3) << fileStruct.blocks.size()
                  << " to file with m = " << std::setw(3) <<
    unsigned(block.m)
                  << ", p = " << std::setw(3) << unsigned(block.
    predictor)
                  << " and phase = " << std::setw(1) << unsigned(
    block.phase)
                  << "  "
                  << "\r" << std::flush;

        for (short& sample : block.data)
            golomb.encode(sample);
    }
    std::cout << "\nAll data written to file\n\n";
}
```

Listing 4.16: write_file function

Lastly, since some methods of the class are private, but need to be tested by our testing framework, we created the wrapper methods `test_calculate_m` and `test_abs_value_vector`.

GDecoder class

The `GDecoder` class is responsible for reading and decoding a file encoded by the *GEncoder* class.

```cpp
class GDecoder {
   private:
    BitStream reader;
    Golomb golomb;

    std::string inputFileName;
    File fileStruct;
    Predictor predictorClass;

    std::vector<short> decode_block(Block& block);

   public:
    GDecoder(std::string inFileName);
    ~GDecoder();

    File& read_file();
    std::vector<short> decode_file();
};
```

Listing 4.17: GDecoder declarations

Our class life-cycle starts with the `read_file` method. This method reads the data from the encoded file, by order. Which means it starts by the reading the header of the file and using that information, it reads each *Block* header and encoded data (error). Upon reading the entire file, the function will save internally in the class and return a reference to the created **File** structure.

```cpp
File& GDecoder::read_file() {
    std::cout << "Reading file " << inputFileName;

    // Read file header
    fileStruct.blockSize = reader.readNBits(BITS_BLOCK_SIZE);
    fileStruct.sampleRate = reader.readNBits(BITS_SAMPLE_RATE);
    fileStruct.nFrames = reader.readNBits(BITS_N_FRAMES);
    fileStruct.nChannels = reader.readNBits(BITS_N_CHANNELS);
    fileStruct.bitRate = reader.readNBits(BITS_BIT_RATE);
    fileStruct.lossy = reader.readNBits(BITS_LOSSY);
    fileStruct.approach =
        static_cast<APPROACH>(reader.readNBits(BITS_APPROACH));

    // Write Blocks (data)
    int nBlocks{static_cast<int>(
        ceil(static_cast<double>(fileStruct.nFrames) / fileStruct.
    blockSize))};

    nBlocks *= (int)fileStruct.nChannels;

    std::cout << " with " << unsigned(nBlocks) << " blocks" << endl;

    if (!check_approach(fileStruct.approach)) {
        cerr << "Error: Invalid approach type " << fileStruct.approach << endl;
        exit(1);
    }
    golomb.setApproach(fileStruct.approach);
    predictorClass.set_nChannels(fileStruct.nChannels);

    for (int bId = 0; bId < nBlocks; bId++) {
        Block block;
        // Read Block header
        block.m = reader.readNBits(BITS_M);
        block.phase = static_cast<PHASE>(reader.readNBits(BITS_PHASE));
        block.predictor =
            static_cast<PREDICTOR_TYPE>(reader.readNBits(BITS_PREDICTOR));

        // Read Block data
        this->golomb.setM(block.m);
        std::cout << " - Reading Block " << std::setw(3) << bId + 1
                  << " with m = " << std::setw(3) << unsigned(block.m)
                  << ", predictor = " << std::setw(3)
                  << unsigned(block.predictor)
                  << " and phase = " << std::setw(3) << unsigned(block.phase)
                  << endl;

        // check m
        if (block.m < 1) {
            cerr << "Error: Encountered invalid m = " << unsigned(block.m)
                 << endl;
            exit(2);
        }

        for (uint16_t i = 0; i < fileStruct.blockSize; i++)
            block.data.push_back((short)golomb.decode());

        fileStruct.blocks.push_back(block);
    }
    std::cout << "All data read from file" << std::endl;

    return fileStruct;
}
```

Listing 4.18: read_file function

Using the `decode_file`, we'll cycle through the Blocks with the encoded data and call the `decode_block` method, returning the set of samples of the *Block* and joining all the decoded samples into a single $std::vector$. When all blocks are decoded, the values of the music are saved in this vector, which is returned.

```
std::vector<short> GDecoder::decode_file() {
    std::vector<short> outSamples;
    std::cout << "Decoding file with " << unsigned(fileStruct.
    blocks.size())
              << " Blocks" << endl;
    int count = 1;
    for (Block& block : fileStruct.blocks) {
        std::cout << " - Decoding Block: " << count++ << "\r" <<
    std::flush;
        std::vector<short> blockSamples = decode_block(block);
        outSamples.insert(outSamples.end(), blockSamples.begin(),
                          blockSamples.end());
    }
    std::cout << "\nAll Blocks decoded\n" << endl;

    return outSamples;
}
```

Listing 4.19: decode$_{file function}$

The `decode_block` creates iteratively a set of samples, by predicting the value of them, using the *Predictor* class and using the error read in the file. It uses the following formula to determine the current sample.

$$samples_n = \text{error}[i] + \text{prediction}(samples)$$

The *Predictor* and *Phase/Correlation* used for prediction, are the ones present in the header of each *Block*.

```
std::vector<short> GDecoder::decode_block(Block& block) {
    std::vector<short> samples;
    PREDICTOR_TYPE pred = static_cast<PREDICTOR_TYPE>(block.
    predictor);
    PHASE ph = static_cast<PHASE>(block.phase);

    for (int i = 0; i < (int)block.data.size(); i++) {
        int prediction = predictorClass.predict(pred, ph, samples,
    i);
        // error + prediction
        int sample = block.data.at(i) + prediction;
        samples.push_back(sample);
    }

    return samples;
}
```

Listing 4.20: decode_block function

4.1.3 Encoder and Decoder

The `main Decoder` function initializes essential variables and settings for processing audio data. It first checks the command-line arguments, ensuring they're valid for the program to proceed. Next, the input audio file is validated to be in the correct format. After validating the file, relevant information is extracted, like sample rate, number of channels, and audio frames.

The number of blocks are calculated, and to ensure data consistency, zeros are padded to the audio to fit into an integer number of blocks. The `GEncoder` class is initialized with the **File**, to organize and setup the data for encoding. Using the `encode_file` method, the program encodes the audio data, compressing it according to the specified options. It's code can be seen below.

```cpp
int main(int argc, char* argv[]) {
    int ret = process_arguments(argc, argv);
    if (ret < 0)
        return 1;
    else if (ret > 0)
        return 0;

    clock_t startTime = clock();

    /* Check .wav file */
    SndfileHandle sfhIn{Options::musicName};
    if (check_wav_file(sfhIn) < 0)
        return 1;

    if (!check_approach(Options::approach)) {
        cerr << "Error: Invalid approach type " << Options::approach << endl;
        return 1;
    }

    // Set Options variables
    Options::sampleRate = static_cast<size_t>(sfhIn.samplerate());
    Options::nChannels = static_cast<size_t>(sfhIn.channels());
    Options::nFrames = static_cast<size_t>(sfhIn.frames());

    std::vector<short> inputSamples(Options::nChannels * Options::nFrames);
    sfhIn.readf(inputSamples.data(), Options::nFrames);

    size_t nBlocks{static_cast<size_t>(
        ceil(static_cast<double>(Options::nFrames) / Options::blockSize))};

    // Do zero padding, if necessary
    inputSamples.resize(nBlocks * Options::blockSize * Options::nChannels);

    print_processing_information(nBlocks);

    // Create Golomb Encoder class
    GEncoder gEncoder(Options::encodedName, Options::m, Options::predictor,
                      Options::phase);

    // Create file struct
    File f;
    f.sampleRate = Options::sampleRate;
    f.blockSize = Options::blockSize;
    f.nChannels = Options::nChannels;
    f.nFrames = Options::nFrames;
    f.blocks = std::vector<Block>();
    f.bitRate = Options::bitRate;
    f.lossy = Options::lossy;
    f.approach = Options::approach;

    // dont forget to multiply the number of blocks by the number
    of channels
    //  since each the nBlocks is accounting for raw frames and in
    the case of
    //  stereo audio, each frame has two samples, so double the
    calculated blocks
    //  (because blocks have samples, not frames)
    gEncoder.encode_file(f, inputSamples, nBlocks * Options::
    nChannels);

    clock_t endTime = clock();
    std::cout << "Program took " << std::fixed << std::setprecision
    (2)
              << (double(endTime - startTime) / CLOCKS_PER_SEC)
              << " seconds to run. Music compressed to " << Options
    ::encodedName
              << std::endl;

    return 0;
}
```

Listing 4.21: Main function of encoder

The `Decoder` program starts with the `main` function which sets and evaluates the provided program arguments using the `process_arguments` function. After setting the arguments, the next step is to instantiate the `GDecoder` class, associating it with the encoded file (`Options::encodedName`) to facilitate data extraction.

The decoding operation starts by calling the `decode_file` method. This step retrieves the music samples, from the encoded data (error) present in the file. Using the `save_decoded_music` function, the program then proceeds to store the decoded audio data (`decodedSamples`) into a new `.wav` file.

The `save_decoded_music` function constructs an output file in the `.wav` format, ensuring adherence to PCM 16-bit audio standards. The function configures the file's channel count and sample rate based on the attributes of the decoded audio file. The code of this function is defined below.

```
void save_decoded_music(File& f, std::vector<short>& samples) {
    SndfileHandle sfhOut{
        Options::musicName, SFM_WRITE, SF_FORMAT_WAV |
    SF_FORMAT_PCM_16,
        static_cast<int>(f.nChannels), static_cast<int>(f.
    sampleRate)};
    if (sfhOut.error()) {
        cerr << "Error: problem generating output .wav file\n";
        exit(1);
    }

    sfhOut.writef(samples.data(), samples.size() / f.nChannels);
}

```

```
int main(int argc, char* argv[]) {
    int ret = process_arguments(argc, argv);
    if (ret < 0)
        return 1;
    else if (ret > 0)
        return 0;

    clock_t startTime = clock();

    // Create Golomb Encoder class
    GDecoder gDecoder(Options::encodedName);

    File f = gDecoder.read_file();

    print_processing_information(f);

    std::vector<short> decodedSamples = gDecoder.decode_file();

    save_decoded_music(f, decodedSamples);

    clock_t endTime = clock();
    std::cout << "Program took "
              << (double(endTime - startTime) / CLOCKS_PER_SEC) *
    1000
              << " ms to run. Music decompressed to " << Options::
    musicName
              << std::endl;

    return 0;
}
```

Listing 4.22: Main function of Decoder

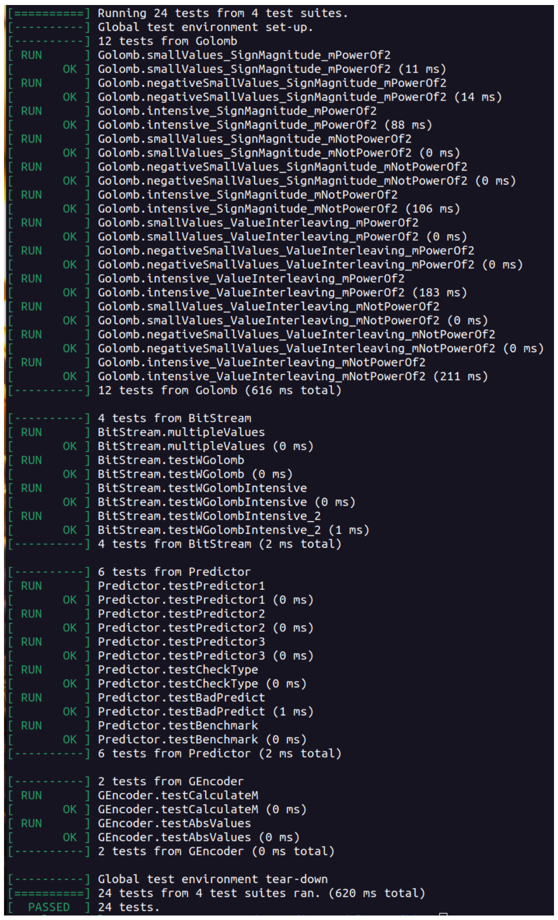
4.1.4 Tests

Following a Test-Driven Development, we used the `Google Test` framework, to ensure that critical individual functions were working properly after new developments. An approach that came in handy when problems aroused during development. Our *Cmakelists* should automatically download and install the necessary frameworks upon compilation of the program and this tests can be ran by just executing the following command in the *bin* folder:

```
./tests
```

Listing 4.23: Example of test to Golomb class

With the output looking like this:



```
[==========] Running 24 tests from 4 test suites.
[----------] Global test environment set-up.
[----------] 12 tests from Golomb
[ RUN      ] Golomb.smallValues_SignMagnitude_mPowerOf2
[       OK ] Golomb.smallValues_SignMagnitude_mPowerOf2 (11 ms)
[ RUN      ] Golomb.negativeSmallValues_SignMagnitude_mPowerOf2
[       OK ] Golomb.negativeSmallValues_SignMagnitude_mPowerOf2 (14 ms)
[ RUN      ] Golomb.intensive_SignMagnitude_mPowerOf2
[       OK ] Golomb.intensive_SignMagnitude_mPowerOf2 (88 ms)
[ RUN      ] Golomb.smallValues_SignMagnitude_mNotPowerOf2
[       OK ] Golomb.smallValues_SignMagnitude_mNotPowerOf2 (0 ms)
[ RUN      ] Golomb.negativeSmallValues_SignMagnitude_mNotPowerOf2
[       OK ] Golomb.negativeSmallValues_SignMagnitude_mNotPowerOf2 (0 ms)
[ RUN      ] Golomb.intensive_SignMagnitude_mNotPowerOf2
[       OK ] Golomb.intensive_SignMagnitude_mNotPowerOf2 (106 ms)
[ RUN      ] Golomb.smallValues_ValueInterleaving_mPowerOf2
[       OK ] Golomb.smallValues_ValueInterleaving_mPowerOf2 (0 ms)
[ RUN      ] Golomb.negativeSmallValues_ValueInterleaving_mPowerOf2
[       OK ] Golomb.negativeSmallValues_ValueInterleaving_mPowerOf2 (0 ms)
[ RUN      ] Golomb.intensive_ValueInterleaving_mPowerOf2
[       OK ] Golomb.intensive_ValueInterleaving_mPowerOf2 (183 ms)
[ RUN      ] Golomb.smallValues_ValueInterleaving_mNotPowerOf2
[       OK ] Golomb.smallValues_ValueInterleaving_mNotPowerOf2 (0 ms)
[ RUN      ] Golomb.negativeSmallValues_ValueInterleaving_mNotPowerOf2
[       OK ] Golomb.negativeSmallValues_ValueInterleaving_mNotPowerOf2 (0 ms)
[ RUN      ] Golomb.intensive_ValueInterleaving_mNotPowerOf2
[       OK ] Golomb.intensive_ValueInterleaving_mNotPowerOf2 (211 ms)
[----------] 12 tests from Golomb (616 ms total)

[----------] 4 tests from BitStream
[ RUN      ] BitStream.multipleValues
[       OK ] BitStream.multipleValues (0 ms)
[ RUN      ] BitStream.testWGolomb
[       OK ] BitStream.testWGolomb (0 ms)
[ RUN      ] BitStream.testWGolombIntensive
[       OK ] BitStream.testWGolombIntensive (0 ms)
[ RUN      ] BitStream.testWGolombIntensive_2
[       OK ] BitStream.testWGolombIntensive_2 (1 ms)
[----------] 4 tests from BitStream (2 ms total)

[----------] 6 tests from Predictor
[ RUN      ] Predictor.testPredictor1
[       OK ] Predictor.testPredictor1 (0 ms)
[ RUN      ] Predictor.testPredictor2
[       OK ] Predictor.testPredictor2 (0 ms)
[ RUN      ] Predictor.testPredictor3
[       OK ] Predictor.testPredictor3 (0 ms)
[ RUN      ] Predictor.testCheckType
[       OK ] Predictor.testCheckType (0 ms)
[ RUN      ] Predictor.testBadPredict
[       OK ] Predictor.testBadPredict (1 ms)
[ RUN      ] Predictor.testBenchmark
[       OK ] Predictor.testBenchmark (0 ms)
[----------] 6 tests from Predictor (2 ms total)

[----------] 2 tests from GEncoder
[ RUN      ] GEncoder.testCalculateM
[       OK ] GEncoder.testCalculateM (0 ms)
[ RUN      ] GEncoder.testAbsValues
[       OK ] GEncoder.testAbsValues (0 ms)
[----------] 2 tests from GEncoder (0 ms total)

[----------] Global test environment tear-down
[==========] 24 tests from 4 test suites ran. (620 ms total)
[  PASSED  ] 24 tests.
```

Figure 4.1: Unit tests conducted using `Google Tests`

An example of the creation of a test can be seen below. Note the $TEST()$ and $EXPECT\_EQ$ macros.

```cpp
// Import of standard or provided libraries
#include <gtest/gtest.h>
#include <cmath>
#include <string>
#include <tuple>

// Import of local libraries
#include <golomb_codec.h>

using namespace std;

TEST(Predictor, testBenchmark) {
    Predictor predictor1(1);

    std::vector<short> predict1_samples = {1, 1, 1, 1, 1};
    std::tuple<int, int> ret1 = predictor1.benchmark(
    predict1_samples);
    EXPECT_EQ(static_cast<PREDICTOR_TYPE>(std::get<0>(ret1)),
    PREDICT1);

    std::vector<short> predict2_samples = {2, 4, 8, 16, 32};
    std::tuple<int, int> ret2 = predictor1.benchmark(
    predict2_samples);
    EXPECT_EQ(static_cast<PREDICTOR_TYPE>(std::get<0>(ret2)),
    PREDICT2);

    std::vector<short> predict3_samples = {1, 5, 14, 30, 55};
    std::tuple<int, int> ret3 = predictor1.benchmark(
    predict3_samples);
    EXPECT_EQ(static_cast<PREDICTOR_TYPE>(std::get<0>(ret3)),
    PREDICT3);
}
```

Listing 4.24: Example of test to Golomb class

4.1.5 Usage

Golomb Encoder

The program can be launched using the following command:

```
./golomb_encoder [OPTIONS]
```

To check the available options, use the -h option, like:

```
./golomb_encoder -h
```

This command will generate the following text in the console:

```
Usage: %s [OPTIONS]
  OPTIONS:
  -h, --help         --- print this help
  -i, --input        --- set music file name (default: ../songs/
    sample01.wav)
  -o, --output       --- set encoded file name (default:
    encodedSample)
  -b, --blockSize    --- set block size (default: 1024)
  -l, --lossy        --- set lossy compression (default: off)
  -m, --modulus      --- set m number (default: automatic
    calculation)
  -p, --predict      --- set predictor [0,3] (default: PREDICT1)
  -a, --approach     --- set approach [0,1] (default: SIGN_MAGNITUDE
    )
  -s, --phase        --- set phase [0,3], only applicable to stereo
    audio (default: P_AUTOMATIC)
```

Listing 4.25: golomb$_encoderarguments$

Golomb Decoder

The program can be launched using the following command:

```
./golomb_decoder [OPTIONS]
```

To check the available options, use the -h option, like:

```
./golomb_decoder -h
```

This command will generate the following text in the console:

```
Usage: %s [OPTIONS]
  OPTIONS:
  -h, --help          --- print this help
  -i, --input         --- set encoded file name (default:
    encodedSample)
  -o, --output        --- set decoded music name (default:
    decodedMusic.wav)
```

Listing 4.26: golomb_decoder arguments

4.1.6 Results

Executing our code for two audio files, one mono and one stereo, revealed the statistics present in Figure 4.2. With this we can calculate the encoded space saved:

$$\text{Mono Reduction} = \frac{\text{Initial Size} - \text{Encoded Size}}{\text{Initial Size}} \times 100\%$$

$$\equiv \frac{0.299}{1.1} \times 100\% \approx 27.18\%$$

$$\text{Stereo Reduction} = \frac{\text{Initial Size} - \text{Encoded Size}}{\text{Initial Size}} \times 100\%$$

$$\equiv \frac{0.68}{5.18} \times 100\% \approx 13.13\%$$

Which shows the our encoding algorithm for stereo files could be improved by exploring other techniques, as encoding one channel directly and expressing the other channel as the difference.

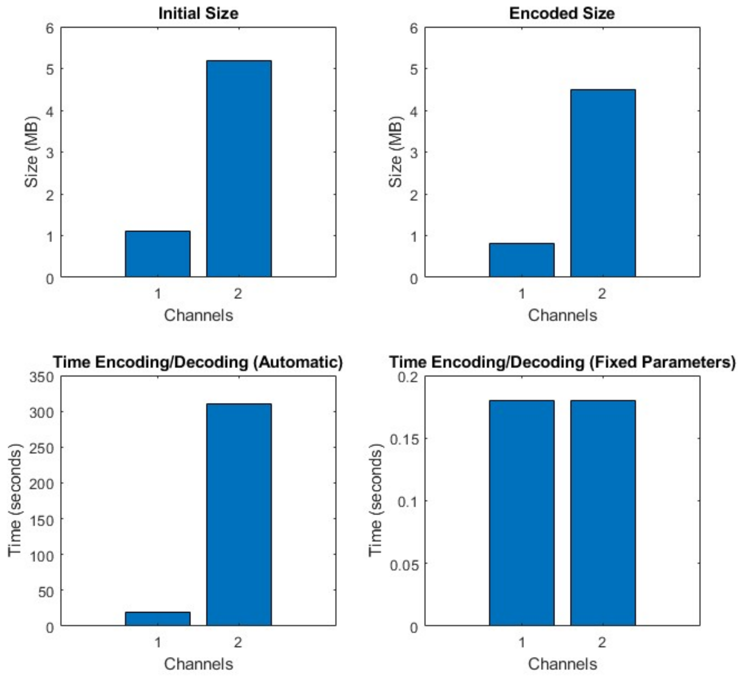
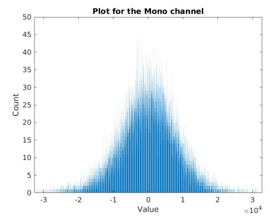
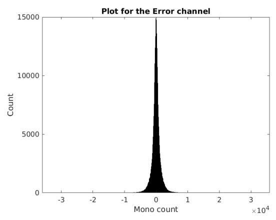


Figure 4.2: Optimization obtained from encoding

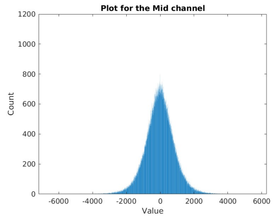
In terms of the resulting encoding distribution, the Figure 4.3 compares the regular $waveform$ file distribution versus our encoded data (error) in the file. It's clearly visible that our program is working properly, since the values are much more concentrated around the sample with value 0. However, as discussed above, this further indicates that our stereo encoding could still be improved, since the value count of the samples closer to 0 isn't as high as for the mono audio file.



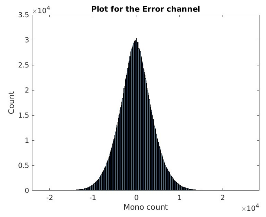
(a) Raw mono audio



(b) Encoded mono audio



(c) Raw stereo audio



(d) Encoded stereo audio

Figure 4.3: Comparison of values saved vs. wav file

4.2 Exercise 5

Contributions

The work carried out in this project was equally distributed between all members, so the evaluation is the same for all.