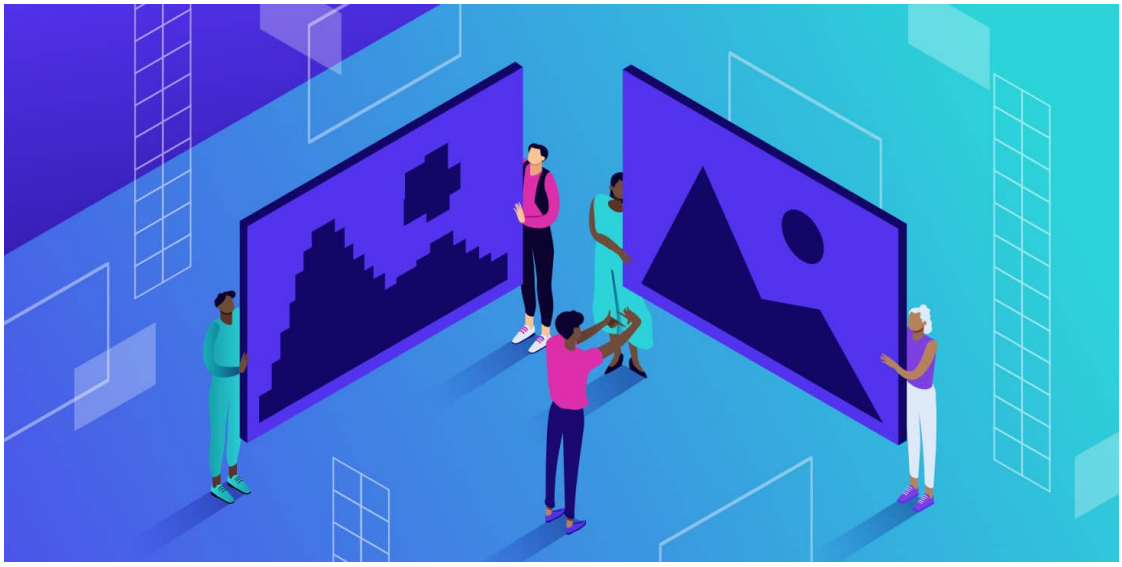
INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN

Projet scientifique encadré PSE

STPI/PSE/2024 – 006

Algorithmes de compression : pourquoi, Comment ?



Etudiants :

Michel VESPIER **Marius LANDREAU**

Célia ANDRADE PASCAL **Tim DHIEN**

Réda BOUZID **Clément AUCLAIR**

Enseignant-responsable du projet :

Alexis LECOMTE

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN

Date de remise du rapport : ***15/06/2024***

Référence du projet : ***STPI/PSE/2024 – 006***

Intitulé du projet : ***Algorithmes de compression : pourquoi, Comment ?***

Type de projet : ***Modélisation***

Objectifs du projet :

Ce projet vise à découvrir ce qui se cache derrière le mot "compression", étant souvent quelque chose d'obscur pour les non initiés à l'informatique. Pour éclairer cette notion, ce projet s'articule autour de trois axes principaux : le texte, les images et le son. Le but est de produire du code pour, concernant la première partie, faire notre propre algorithme d'Huffman, applicable cependant aux autres types de fichiers comme les fichiers liés à l'image. Concernant le son, compresser des fichiers WAV via la théorie de Fourier. Enfin, nous essayerons de reproduire les algorithmes de compressions PNG et JPEG pour les différents types de compressions sans et avec pertes. Mais l'objectif principal de ce projet est d'appréhender sous le prisme d'étudiants de deuxième année, la compression et d'y mettre les mains.

Mots-clefs du projet :

Encodage

Huffman

Redondance

Traitement

TABLE DES MATIERES

NOTATIONS, ACRONYMES

ASCII : American Standard Code for Information Interchange

HTTP : HyperText Transfer Protocol

DCT : Descriptive Cosine Transform

Code RGB : Red Green Blue

DFT : The Discrete Fourier Transform - Transformation de Fourier Discrète

FFT : Fast Fourier Transform

1. INTRODUCTION

Depuis l'émergence de l'informatique et la montée en puissance de l'hyperconnexion mondiale, l'information joue un rôle clé dans nos activités humaines. En effet, l'humanité, dans sa folie croissante, veut toujours plus, toujours plus vite et en particulier dans le domaine de l'information. Cependant, une problématique se pose lorsque l'on cherche à globaliser l'information et à échanger de nombreuses informations d'un côté du globe à un autre en un temps record. A vrai dire, la science qui s'occupe de répondre à ces questions n'est nulle autre que celle de la "compression".

Le but ici ne sera pas tant de dresser les enjeux qui entourent cette science, bien qu'un simple portrait fut peint en quelques lignes, mais bien de pénétrer cette science et d'aborder des domaines de cette science d'un point de vue très terre à terre. Ainsi, comme évoqué dans les objectifs de ce projet, il s'agira dans ce rapport de projet scientifique, d'établir clairement ce qui borde ces grandes thématiques de compression que sont les fichiers : textes, images et audios.

2. MÉTHODOLOGIE / ORGANISATION DU TRAVAIL

Afin de parvenir à mener ce projet à bien, nous avons, sous les précieux conseils de Mr Lecomte, séparé le travail en 3 parties distinctes.

De ce fait, nous avons d'abord travaillé séparément sur ces thématiques : le texte avec Huffman, l'image et le son. Le code a été la partie la plus importante de celui-ci puisque nous avons passé la majorité de cet enseignement à coder, chercher et implémenter des algorithmes. En outre, pour que tout le monde puisse programmer, nous avons choisi d'utiliser le langage Python pour toutes ses bibliothèques disponibles et sa facilité de prise en main.

Simplement, ce projet a été mené d'une manière similaire à ce que nous avons pu connaître concernant le projet informatique. C'est-à-dire que nous commencions par faire des recherches pour implémenter et tester nos codes. Et à chaque semaine, notre référent de projet, Alexis Lecomte, suivait l'avancée des différentes étapes du code pour nous aiguiller sur la marche à suivre. Dans le sens où si nous voulions nous aventurer vers des algorithmes trop difficiles ou des types de compression sophistiqués, il donnait une marche à suivre abordable et applicable selon les capacités de chacun.

Nous avons ainsi choisi de se répartir en 3 groupes dans les différents formats de compression que nous avons choisi de traiter. Par conséquent, au début du projet a été décidé d'affecter Réda et Michel au groupe de la compression de texte avec déjà en vu l'algorithme d'Huffman, Tim et Clément à l'image et Marius et Célia au son.

Mais pour ce type de projet, les compétences en informatique étant différentes de tout à chacun, l'avancée dans les différents codes a été disparate. Par exemple, le groupe qui a traité le son était celui qui avait le plus de difficulté avec la partie code. Donc les groupes ont été amenés à changer pour le bien du projet. Ainsi, Michel étant la personne la plus qualifiée en informatique du groupe et étant le plus avancé sur sa partie vers le milieu du projet, il est venu aider le groupe du son qui a pu par la suite proposer un code fonctionnel de compression décompression de son.

En fait, travailler sur sa partie a eu une double implication pour nous. Cela a été un avantage dans le sens où nous avons pu creuser notre domaine de compression et comprendre mieux les tenants et aboutissants de celui-ci. Cependant, cela a aussi été un désavantage puisque nous n'avons pas pu nous plonger complètement dans les domaines traités par les autres groupes.

3. TRAVAIL RÉALISÉ ET RÉSULTATS

3.1. Le texte et l'algorithme d'Huffman

Le codage de Huffman, inventé en 1952 par David Albert Huffman, est un algorithme de compression de données sans perte, ce qui signifie que lors de la décompression des données, on retrouve intégralement les mêmes données.

3.1.1. Principe

L'objectif de cette compression est d'affecter un code de longueur variable à chaque caractère d'entrée en fonction de sa fréquence d'apparition. L'idée étant d'affecter le code le plus petit possible pour les caractères les plus fréquents afin d'optimiser leur représentation. Ainsi, pour les caractères les plus rares, nous perdons quelques bits lors de leur représentation, que l'on regagne d'une manière bien plus significative pour les caractères les plus fréquents. Le codage de Huffman est alors très important dans la compression de données dans lesquelles il y a des caractères qui apparaissent fréquemment.

Illustration

Considérons un texte où le caractère 'z' est présent 15 fois. Ce caractère aura alors une représentation plus longue que s'il était codé avec la table ASCII, disons sur 12 bits au lieu de 8 bits. Ici la perte lors de la représentation de ce caractère sera alors de 15*(12-8) = 60 bits. Néanmoins, dans ce même texte le caractère 'a' est présent 3500 fois, et possède alors une représentation plus courte sur 2 bits. Le gain serait alors pour ce caractère de 3500*(8-2) = 21000 bits. Ce qui démontre bien tout l'intérêt de cette méthode.

3.1.2. Méthode

Dans un premier temps, on vient lire le fichier en entier afin de créer la table d'occurrence des caractères. Cette table est ordonnée dans l'ordre décroissant des fréquences.

Après avoir constitué la table des fréquences, on vient créer l'arbre binaire. Dans cet arbre, les caractères sont représentés individuellement sous formes de nœuds feuilles. Chaque nœud est associé à un poids, correspondant à la fréquence ou à la probabilité d'apparition des symboles.

Afin de créer cet arbre, on suit la procédure suivante:

1 - On identifie les deux nœuds libres ayant les poids les plus faibles.

2 - On crée un nœud parent pour ces deux nœuds. Le poids du nœud parent est égal à la somme des poids de ses deux nœuds enfants.

3 - Le nœud parent nouvellement créé est ajouté à la liste des nœuds libres, tandis que les deux nœuds enfants en sont retirés.

4 - On répète ces étapes tant qu'il reste plus d'un seul nœud libre. Le dernier nœud restant devient alors la racine de l'arbre.

À travers cette procédure, on se base sur l'idée selon laquelle la meilleure manière de coder une information est de trouver la longueur optimale pour représenter chaque caractère de la séquence initiale. La longueur optimale du code correspondant à un caractère est donnée par le nombre de branches qu'il faudra parcourir partant du sommet de l'arbre de Huffman pour arriver à ce caractère qui sera toujours une feuille de l'arbre de Huffman.

Pour récupérer les codes de chaque caractère de l'arbre de Huffman, on met, par convention, un 0 sur chaque embranchement vers la gauche et un 1 sur chaque embranchement de droite, et ce en descendant l'arbre.

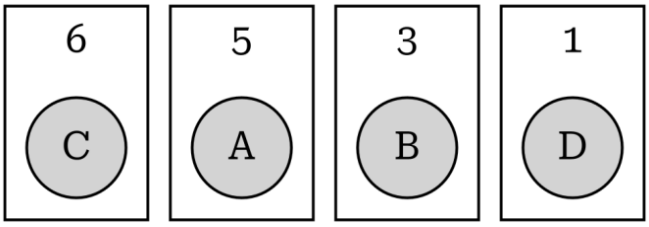
Cette méthode permet d'éviter toute ambiguïté dans le processus de décodage en utilisant le concept de code préfixe, qui empêche qu'un code associé à un caractère ne soit présent dans le préfixe d'un code d'un autre caractère.

Illustration:

On souhaite coder le texte suivant: "BCAADDDCCACACAC"

Chaque caractère occupe 8 bits pour un total de 15 caractères, soit 8*15= 120 bits

On commence par créer la table d'occurrence, que l'on trie par ordre décroissant. On obtient alors:

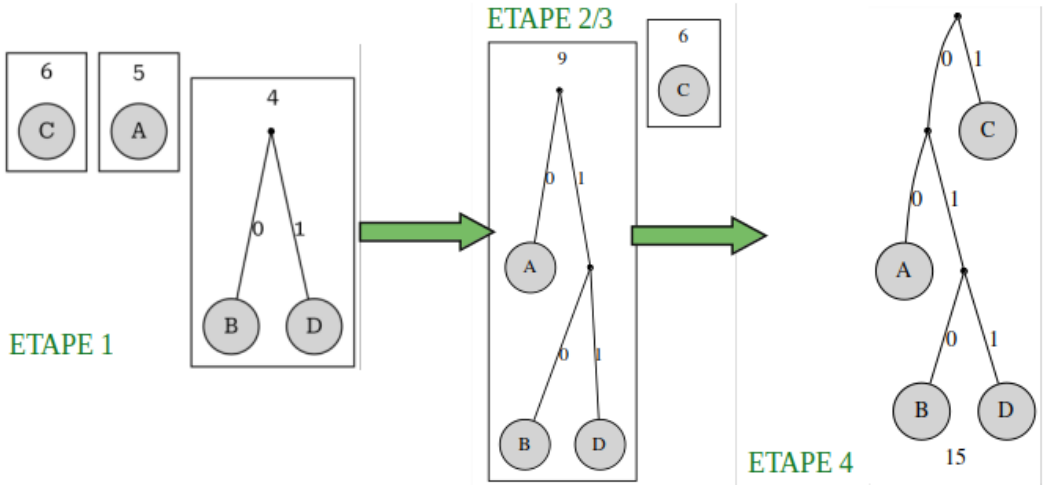


On suit maintenant les étapes de la procédure pour créer l'arbre binaire:

1-Identification des nœuds les plus légers

2 et 3-Création d'un nœud parent et mise à jour de la liste des nœuds libres

4- Répétition des Étapes



Finalement, lorsque l'on voudra décompresser le texte, il faudra également envoyer l'arbre binaire associé pour décoder le texte compressé.

On aura alors:

Caractère	Fréquence	Code	Taille
A	5	11	5*2=10 bits
B	1	100	1*3=3 bits
C	6	0	6*1=6 bits
D	3	101	3*3= 9 bits
4*8 = 32 bits	15 bits		28 bits

Après compression, la taille totale du texte est donc de 32+15+28 = 75 bits, ce qui est inférieur au 120 bits initial.

Le code de l'implémentation du codage de Huffman est à retrouver ici.

3.1.3. Analyse statistique

Afin de décompresser le texte transmis, il faut fournir au décompresseur les informations sur le texte tel que les clés de décodage de chaque caractère. Ces informations se trouvent d'en l'en-tête du fichier. Ainsi, pour les fichiers les plus courts, cet en-tête peut prendre proportionnellement plus d'espace que pour les plus longs, ce qui peut rendre la compression de Huffman moins intéressante.

Cet algorithme possède également d'autres légers inconvénients, comme le fait de devoir lire entièrement le fichier en amont afin de créer la table d'occurrence, ce qui peut prendre plus ou moins de temps bien que ce temps reste négligeable avec la puissance des machines de nos jours. De plus, avec cette méthode, il est impérativement nécessaire de transmettre la table de décodage associée au texte.

Une solution à ce problème est l'utilisation d'une table préétablie, qui en fonction de la langue du texte notamment, associe à chaque caractère son code. Ces tables s'appuient sur d'énormes quantités de texte dans une langue particulière pour déterminer quels sont les caractères les plus fréquents. Par exemple, en français, on trouve la table de fréquence des caractères suivante dans le corpus de Wikipedia.

Néanmoins cette méthode présente des limites, en effet, certains biais peuvent exister, par exemple si le texte traite en particulier un domaine précis, ou si un prénom y apparaît souvent, alors certains mots sont voués à apparaître plus souvent pouvant alors gonfler artificiellement la fréquence de certaines lettres.

Nous allons maintenant essayer de réaliser l'analyse statistique du codage de Huffman sur des textes en français. Pour ce faire, nous avons besoin d'une quantité énorme de textes écrits en français, nous allons nous appuyer sur un site générant aléatoirement un nombre de paragraphes choisis dans cette langue, que l'on retrouve à l'adresse suivante : enneagon.org. Il est spécifié que "*le texte généré est constitué de phrases dont les mots*

sont sélectionnés aléatoirement. Le tirage suit la répartition statistique d'un référentiel textuel en langue française", c'est exactement ce dont nous avons besoin.

Par défaut, en entrant sur l'adresse du lien, le site nous génère aléatoirement un paragraphe constitué de 8 phrases. Ces phrases changent à chaque actualisation de la page. Afin de récupérer le texte contenu dans cette page, nous allons utiliser le protocole HTTP.

Le protocole HTTP ou HyperText Transfer Protocol est un protocole de communication spécifiquement pensé pour le web créé en 1980 par Tim Berners-Lee. Plus de 90% des échanges web se font via le protocole HTTP. Son rôle est de définir comment les messages doivent être structurés et comment sont organisés les échanges de données entre navigateurs et serveurs web. Il permet d'échanger notamment le contenu des sites webs comme les textes, images, vidéos, code sources, etc… avec des en-têtes décrivant le contenu des messages.

Dans notre cas, nous allons utiliser la bibliothèque *requests*, qui est une bibliothèque largement utilisée pour envoyer des requêtes HTTP.

Dans notre programme, lorsque nous appelons requests.get(url), la bibliothèque requests envoie une requête HTTP GET au serveur hébergé à l'URL enneagon.org. Cette requête demande au serveur de renvoyer le contenu de la page web. Le serveur reçoit la requête, traite celle-ci et renvoie une réponse au client (dans ce cas, notre script Python). La réponse contient le contenu demandé ainsi que des informations supplémentaires comme le code d'état HTTP, les en-têtes HTTP, etc. Le fichier contenant le code pour l'analyse est à retrouver ici.

Nous avons alors utilisé ce principe pour demander au site de nous envoyer des milliers de phrases générées aléatoirement. À partir de toutes ces requêtes envoyées, nous avons pu constituer un fichier contenant environ 4 000 phrases pour 83 290 mots et 523 705 caractères. Ce fichier est à retrouver ici.

Dès lors nous avons lancé l'analyse statistique pour voir le nombre d'occurrences de chaque caractères, leurs fréquences ainsi que leur code associé par l'algorithme de Huffman. Les résultats de ce test sont à retrouver ici.

On observe un taux de compression de 56% (ou de -44%), mais aussi et sans surprise, des résultats concordants avec la fréquence de caractère dans le corpus de Wikipédia en 2008 en français. À ceci près qu'il faut recalculer les fréquences sans considérer les caractères de ponctuation, comme les espaces notamment et qui sont de loin les caractères les plus fréquents (ils ne sont pas considéré dans la table de Wikipedia).

Par exemple, pour les caractères 'e', le plus fréquents en français, on trouve une fréquence de 11.89% selon notre analyse. Sans considérer tous les caractères de

ponctuation du texte, la fréquence du caractère ‘e’ monte à 14,92%. C’est un résultat assez proche des 14,715% du corpus de Wikipedia. Ce résultat est par ailleurs vrai pour tous les caractères du texte. Ici les résultat sans ponctuation. Le taux de compression à également augmenté étant donné que le nombre de caractères à considérer à diminué, ce qui est cohérent.

3.2. Les images

3.2.1. Introduction

Avant de parler de la compression d’image, il est nécessaire d’expliquer comment une image est représentée du point de vue d’une machine.

Une image est composée de pixels, ce sont des petits points de couleur qui servent d’unité de longueur pour les affichages de manière générale. Un pixel ne peut afficher qu’une couleur à la fois, c’est la combinaison de ces pixels qui nous permet de voir notre image. Une image possède donc un nombre de pixels égale au produit de sa largeur et de sa hauteur. La machine lit une image en parcourant un à un les pixels qui doivent être affichés, les images sont souvent représentées ligne par ligne, chacune représentant les pixels de gauche à droite, selon le format de fichier les lignes peuvent être lues de haut en bas ou de bas en haut.

Maintenant, il est nécessaire de comprendre comment un pixel est représenté. La description d’un pixel se fait généralement avec son code RGB (Red Green Blue) qui renseigne la couleur du pixel, un code RGB est composé de trois entiers compris entre 0 et 255, le premier entier indique le niveau de rouge dans la couleur désirée, le deuxième le niveau de vert et le dernier le niveau de bleu. Chaque pixel est donc codé sous forme de triplet d’entier, par exemple le triplet (0,0,0) renseigne la couleur noire et à l’inverse (255,255,255) le blanc. Il est possible de représenter avec un code RGB 2553 soit plus de 16 millions de couleurs.

3.2.2. Compression sans perte

Dans le cas de l’étude de la compression sans perte des images, nous avons fait le choix d’étudier le format le plus utilisé à savoir le format PNG. Ce format utilise l’algorithme de compression deflate, qui veut dire dégonfler en anglais.

1 Compression [1][4]

Le principe du deflate, et de la compression sans perte en générale, est assez simple, il consiste à prendre un fichier contenant une image, et à créer un nouveau fichier qui contient toutes les données relatives à cette image à l’identique, mais qui occupe moins d’espace en mémoire. Pour ce faire, le deflate s’appuie sur deux principes : la redondance des pixels et la présence de motifs au sein d’une image.

Comme vu précédemment, la compression utilise souvent la redondance des données, c’est-à-dire le fait qu’une donnée a tendance à se répéter plusieurs fois dans un

même fichier. C'est pareil pour les images, les pixels sont très souvent redondants, c'est pour cela que l'algorithme deflate utilise le codage d'Huffman. Mais ceci constitue la dernière étape de la compression.

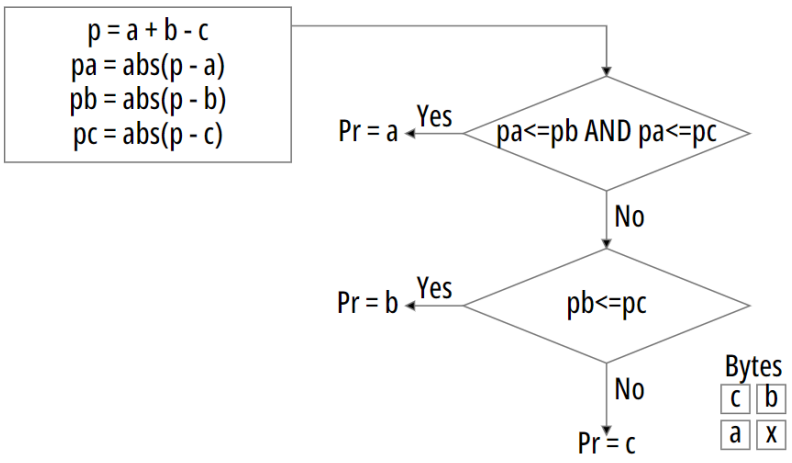
En effet, d'autres étapes interviennent avant le codage d'Huffman afin de maximiser son efficacité.

Le filtrage [4][9]

La première étape est un filtrage des pixels, le principe est de définir une règle qui permet modifier les valeurs des pixels et de retrouver les valeurs d'origine.

Il existe 5 types de filtres, un type de filtre doit forcément être spécifié pour chaque ligne de l'image et sera appliqué à chaque pixel de la ligne. Les filtres s'appliquent à chaque nuance de couleur des pixels, autrement dit à chaque élément du triplet formé par le codage RGB. Chacune de ces nuances est traitée séparément par rapport aux autres, un filtre ne s'applique jamais entre deux valeurs qui proviennent de nuances différentes.

- Le premier filtre n'est pas un filtre car il n'effectue rien, il sert à ne pas filtrer les données avant la compression, car il est obligatoire de spécifier un filtre pour chaque ligne.
- Le second consiste à soustraire les valeurs du triplet RGB d'un pixel avec celles du pixel à sa gauche. Pour cela, le pixel le plus à gauche de chaque ligne n'est pas affecté par cette transformation.
- Pour le troisième, il est nécessaire de préciser que les lignes de pixels sont codées de haut en bas en PNG, le filtre est assez similaire au deuxième mais au lieu de soustraire les valeurs du pixel à gauche, on soustrait les valeurs du pixel d'au-dessus. Par conséquent, appliquer ce filtre sur la ligne du haut revient à ne pas filtrer.
- Le quatrième filtre soustrait à chaque pixel la moyenne entière entre les valeurs du pixel à gauche et les valeurs du pixel d'au-dessus.
- Le dernier filtre est le plus complexe, il permet d'étudier le sens de variations des nuances: vertical, horizontal ou diagonal. En conséquence on soustrait au pixel cible la valeur de nuance du pixel voisin qui convient le mieux, de sorte à ce que la différence soit la plus proche de 0. Pour cela, il réalise l'algorithme ci-dessous.



Ici a représente la valeur de nuance du pixel à gauche de x, b celle du pixel au dessus de x et c celle du pixel en haut à gauche par rapport à x. La valeur de pa correspond à la différence entre les deux pixels voisins horizontaux, pb la différence entre les deux pixels voisins verticaux, et pc la somme des deux différences précédentes avant l'application de la valeur absolue sur ces dernières. Si pa est la plus petite des trois différences, alors c'est que la variation de nuance se fait à la verticale donc on soustrait à notre valeur celle de son voisin de gauche, si pb est la plus petite alors la variation est horizontale donc on soustrait à notre valeur celle de son voisin du dessus, enfin s'il s'agit de pc, la variation de nuance est alors diagonale donc on retire à notre valeur celle de son voisin en haut à gauche.

Pour observer une illustration du principe du cinquième filtre voir annexe n°1.

Pour chaque filtre, la différence est modulo 256, donc si on obtient une valeur négative on la ramène entre 0 et 255. Par exemple, si on la différence est -5, la valeur sera 256-5=251.

Maintenant discutons de l'intérêt de filtrer nos données. Le filtrage a pour unique but de maximiser la redondance, car les différences réalisées en filtrant permettent de concentrer les valeurs autour de 0 et de 255.

Parlons maintenant efficacité. Le premier filtre n'augmente pas la redondance et donc est le moins efficace. Pour les autres filtres, cela dépend de la ligne en question, généralement les filtres 4 et 5 sont les plus efficaces. Mais comment choisit-on le meilleur filtre à appliquer sur une ligne ? Tout d'abord, il est trop difficile de prédire quel filtre sera le meilleur, le plus simple consiste à tester tous les filtres sur une ligne et à calculer le score correspondant à chaque filtre. Le score total est calculé est sommant le score pour chaque valeur de nuance, le score d'une valeur de nuance correspondant à la distance la plus petite par rapport à 0 et 255. Autrement dit, si la valeur est comprise entre 0 et 127 alors son score est égale à la valeur, sinon il vaut 256 moins la valeur. L'explication est simple, le principe du filtrage est de rapprocher les valeurs des extrémités (0 et 255) donc plus la distance par rapport à l'extrémité la plus proche est petite, plus le filtrage est efficace.

Un comparatif des différents filtres est disponible en annexe n°2.

L'algorithme LZ77 [2][3]

La deuxième étape de l'algorithme deflate est l'utilisation de l'algorithme LZ77. LZ pour Lempel-Ziv, car cet algorithme a été créé par Abraham Lempel et Jacob Ziv en 1977, d'où le numéro 77.

Cet algorithme parcourt une à une les valeurs et essaie de regarder s'il n'a pas déjà rencontré un motif de données, débutant par la valeur étudiée, identique. Pour ce faire, il utilise une fenêtre glissante de taille n, composée d'un dictionnaire de taille d et d'un tampon de taille t, tel que n=d+t. Le dictionnaire est rempli des d dernières valeurs rencontrées, c'est ce qui nous permet de mémoriser les données déjà parcourues. Le tampon, lui, mémorise la valeur étudiée et les t-1 valeurs suivantes, c'est dans cet espace que l'on cherche le plus grand motifs de données, débutant par la valeur étudiée et présent à l'identique dans le dictionnaire. Si aucun motif de taille supérieure ou égale à 3 n'est rencontré, alors on réécrit la valeur sans changement. Dans le cas où un motif a été trouvé, on écrit à la place un couple (longueur, distance). La longueur correspond à la longueur du motif et la distance , au

nombre de valeurs séparant les premières valeurs des deux motifs. La valeur suivante étudiée sera alors la prochaine en dehors du motif reconnu, on dit que la fenêtre « glisse ».

Codage de Huffman [8]

Enfin la dernière étape est d'appliquer le codage d'Huffman aux données. Pour cela, deux arbres sont créés, un pour les vraies valeurs de pixels filtrés et les longueurs, et un autre pour les distances. Ensuite, il suffit de remplacer chaque valeur par son code d'Huffman équivalent dans le bon arbre et la compression est terminée.

2 Décompression [8]

Pour la décompression, on utilise l'algorithme inflate, qui veut dire gonfler en anglais. Il consiste à réaliser les étapes précédentes dans l'ordre inverse. Pour cela, on récupère les deux arbres d'Huffman, on substitue les codes d'Huffman par leur valeur, on remplace ensuite chaque couple (longueur, distance) par le motif correspondant, et enfin on retire le filtre toujours en parcourant l'image de haut en bas et les lignes de gauche à droite mais au lieu de faire des soustractions, on additionne la valeur voisine.

3 Implémentation [5][6]

L'objectif de ce projet est de réaliser nous même ces algorithmes. Nous aurions pu faire l'algorithme deflate à notre façon et stocker les résultats dans un fichier compréhensible uniquement par nos algorithmes, mais nous avons plutôt choisi de nous baser sur les spécifications du format PNG et écrire nous même un fichier PNG en partant de zéro, afin que notre système d'exploitation puisse lui aussi ouvrir l'image compressée. Nous n'allons pas entrer dans les détails de comment un fichier PNG est structuré, mais nous allons nous focaliser uniquement sur la partie du fichier contenant les données compressées.

Tout d'abord, nous récupérons préalablement dans le fichier, la taille de l'image, la taille du dictionnaire de la partie LZ77 et le type de compression qu'elle a subie. En effet, il y a trois types de compression. Le premier indique que les données ne sont pas compressées, pour le deuxième, les données sont codées avec des arbres de Huffman statiques, et pour le dernier, les données sont codées avec des arbres de Huffman dynamiques.

Pour le premier type, les deux octets suivants le type de filtre correspondent aux nombres d'éléments à lire. Les deux d'après servent à vérifier que le nombre précédent est bien celui que l'on doit avoir, et enfin on lit autant d'octet que d'éléments à lire pour obtenir tous les pixels.

Pour le second type, les arbres de Huffman sont dits statiques car ils ne sont pas précisés mais déjà connus du programme, c'est donc toujours les deux mêmes arbres utilisés.

Pour le troisième type, les arbres sont donc précisés juste après le type de filtre, nous n'avons pas choisi de traiter ce type de compression car l'écriture des arbres dans le fichier nous a semblé trop complexe.

Pour les deux types de compression, seuls les arbres diffèrent. On décode d'abord une valeur avec l'arbre de Huffman des longueurs et vraies valeurs, si la valeur est comprise

entre 0 et 255 c'est une vraie valeur, si elle vaut 256, cela signifie que la compression est terminée, si elle est comprise entre 257 et 285 alors c'est une longueur encodée selon la table de longueur (voir annexe n°4). En réalité la valeur peut correspondre à un nombre ou un intervalle selon la valeur, si c'est un intervalle alors il faut lire un certain nombre de bits, indiqués aussi dans la table, pour pouvoir connaître la longueur précise. Ensuite on décode une valeur avec l'arbre de Huffman des distances, la valeur est comprise entre 0 et 29 mais est encodée selon la table de distance (voir annexe n°5) qui fonctionne sur le même principe que la table de longueur.

Pour la partie LZ77, la taille du tampon est de 258 et celle du dictionnaire varie entre les puissances de 2 comprises entre 28 et 215.

En terme d'efficacité, le type de compression à arbres de Huffman dynamiques est le plus efficace car dans les arbres statiques toutes les valeurs possibles sont référencées alors que certaines n'apparaissent peut-être pas, contrairement aux arbres dynamiques où seules les valeurs qui apparaissent y sont référencées. Ensuite, plus la taille du dictionnaire est importante, plus la compression est efficace mais plus elle prend de temps.

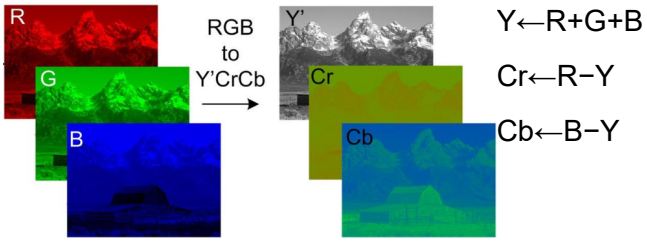
3.2.3. Compression avec perte

Par la suite, nous avons voulu observer un type de compression avec perte. Ici celui que nous allons vous présenter est sûrement le type de fichier le plus utilisé sur terre : le format JPG. Ce format utilise d'une façon courte mais complexe une transformée de Fourier dans son algorithme de compression afin de pouvoir gagner de l'espace mémoire.

1 Compression

Le principe de l'algorithme JPG commence d'abord des pixels de bases sous format RGB pour passer à un autre, le format YCbCr (luminance représentant la lumière et chrominance bleu et rouge représentant les importances de bleu et rouge). Un simple rapport entre les couleurs peut nous permettre de passer d'un format à l'autre sans pertes:

Image RGB to YCBCR



Ce format précis se base sur un principe de nos yeux, nous sommes plus sensibles aux variations de lumières qu'aux variances de couleurs. En se basant sur cette idée le format YCbCr va nous permettre alors de pouvoir efficacement valorisé le fait de garder les informations de lumière plutôt que celle de couleur.

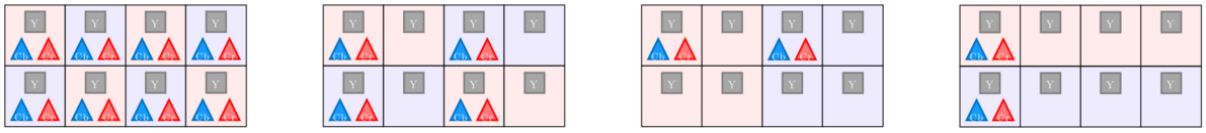
Sous échantillonnage

Le programme va alors diminuer le nombre de valeurs de Chrominance bleu et rouge d'une image via une suite de moyenne. La compression JPEG en comporte plusieurs types, on peut par exemple faire une moyenne seulement entre 2 pixels qui sont à côté ou encore en prendre 4 qui forment un carré. On peut alors appeler le type de modifications des données de bases comment étant:

4:2:0 pour indiquer que pour 4 pixels on garde les 4 valeurs de Lumières(luminances,Y) et seulement 2 valeurs pour les chrominances bleu et rouge(visible sur le schéma ci dessous)

4:4:4 De la même façon, il est possible d'appliquer aucune moyenne sur un fichier et d'en garder toutes les valeurs.

Sous échantillonnage:



Mais bien sûr c'est ici précisément que nous perdons beaucoup d'informations sur les pixels. Car même s'il est très simple de faire une moyenne en connaissant les 4 ou 2 valeurs de chrominance, il est beaucoup plus dur et impossible de retrouver les valeurs qui ont permis d'obtenir cette moyenne via simplement la moyenne. De cette manière on économise et perds les 3 quarts des valeurs de chrominances. Ce qui représente après calcule une économie de la moitié des informations (On a ¾ des valeurs de chrominances qui sont enlevées qui elles même représenté les ⅔ des données).

La transformé de Fourrier

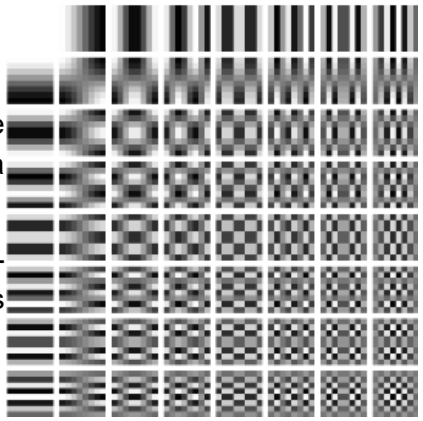
Par la suite nous répartissons ces valeurs en matrice de 8 par 8 afin d'alléger le calcul lourd qui s'apprête à venir. Car après nous allons utiliser une Transformée de Fourier et plus précisément une "descriptive cosine transform" (DCT) qui demande un grand nombre de calculs. *Matrice de DCT avant quantification et après quantification avec matrice de quantification*

$$\begin{bmatrix} -415 & -33 & -58 & 35 & 58 & -51 & -15 & -12 \\ 5 & -34 & 49 & 18 & 27 & 1 & -5 & 3 \\ -46 & 14 & 80 & -35 & -50 & 19 & 7 & -18 \\ -53 & 21 & 34 & -20 & 2 & 34 & 36 & 12 \\ 9 & -2 & 9 & -5 & -32 & -15 & 45 & 37 \\ -8 & 15 & -16 & 7 & -8 & 11 & 4 & 7 \\ 19 & -28 & -2 & -26 & -2 & 7 & -44 & -21 \\ 18 & 25 & -12 & -44 & 35 & 48 & -37 & -3 \end{bmatrix} \quad \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

$$\begin{bmatrix} -26 & -3 & -6 & 2 & 2 & -1 & 0 & 0 \\ 0 & -3 & 4 & 1 & 1 & 0 & 0 & 0 \\ -3 & 1 & 5 & -1 & -1 & 0 & 0 & 0 \\ -4 & 1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

La DCT représente une décomposition d'une image complexe en images simples. On va donc représenter pour chaque valeur de notre matrice l'importance de cette image simple.

Cela nous permettra alors d'obtenir une matrice avec des valeurs importantes en haut à gauche, les motifs importants de l'image et des valeurs faibles en bas à droite les images sans importance.



Comme vous pouvez le voir l'image en haut à gauche représente une image pleine tandis que l'image en bas à droite représente l'image vide.

Pour appliquer une DCT à une matrice "normal" c'est-à-dire nos valeurs de luminances et chrominances nous devons utiliser la formule de la DCT que voici.

Formule DCT

$$F(m,n) = \frac{2}{\sqrt{MN}} C(m)C(n) \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) \cos\frac{(2x+1)m\pi}{2M} \cos\frac{(2y+1)n\pi}{2N}$$

La quantification

Le JPEG cherche ensuite à pouvoir appliquer une compression de Huffman le plus efficacement possible. Pour ce faire il faut maximiser la redondance de données ici grâce au fait que les valeurs les plus importantes sont en à gauche, on peut remarquer que les autres sont beaucoup plus proches de 0 en bas à droite. Pour favoriser cette effet de valeur proche de Zéro l'algorithme va utiliser ce qu'on appelle une matrice de quantification. De façon simple on va diviser nos valeurs proches de 0 par des valeurs beaucoup plus grandes afin de les rapprocher de 0 encore plus. Le programme se base donc sur une matrice de quantification ou il rentre les diviseur de chaque valeur de notre matrice de DCT dans une autre matrice. Pourtant attention, de la même manière que les valeurs d'origine des chrominances ne pouvaient pas être retrouvées via la moyenne ici ci, si nos valeurs sont trop proches de zéro, elles seront perdues. Ici ce n'est pas un problème car nous faisons de la compression avec pertes mais nous pouvons quand même définir la matrice de quantification comme étant représenté que par des 1 afin d'avoir une image sans l'utilisation de la quantification.

Dans cette partie nous ne vous avons pas expliqué comment obtenir cette matrice de quantification car cela est beaucoup plus complexe et hors de notre portée. Nous avons simplement trouvé un PDF en parlant qui nous explique que cette matrice de quantification était déterminée par une étude des valeurs de la DCT excluant celle tout en haut à gauche (car souvent beaucoup plus élevé que les autres). Pour ensuite à l'aide d'un calcul de probabilité en obtenir une matrice de quantification nous faisant perdre le nombre d'information voulu. (*Cf annexe site probabilité*)

Lecture en ZigZag

Toujours dans l'idée d'appliquer Huffman dans les meilleures conditions les matrices de chrominances et luminances après application de la quantification sont transformés en simples listes ou les informations sont écrites en lignes via un procédé de zigzag. Ce zigzag

permet aux valeurs hautes de la DCT d'être placé au début de la liste pour ensuite avoir toutes les valeurs proches de zéro à la fin de celle-ci.

Application d'Huffman

Finalement, afin d'utiliser Huffman le plus efficacement les premières valeurs de la DCT c'est-à-dire les plus importantes sont alors séparées des autres. Nous aurons donc 4 tables de Huffman différentes 2 pour la luminances (les valeurs importantes DC: Direct Current terme, les 63 autres valeurs se rapprochant de 0 , AC: Alternative Current terme) et la même chose mais pour la chrominance rouge et bleu combinés.

2 Décompression

La décompression repasse donc dans le sens inverse à chaque étape de la compression. Passant de la décompression d'Huffman que vous avez vu avant à ensuite un passage de ligne à matrice. Pour ensuite simplement multiplier les valeurs de cette même matrice par les valeurs présentes dans notre matrice de quantification. Enfin utiliser une transformée de Fourier inverse que l'on appelle Inverse Cosine Transform. Cet algorithme est basé sur la DCT vu avant et est présenté dans les sources. La dernière chose à faire est de simplement représenter nos valeurs de chrominances rouge et bleu comme étant celle pour les carrées de 4*4 à la suite.

3 Implémentation

Pour cette partie aussi nous avons essayé de reproduire l'algorithme de compression JPEG via nos propres connaissances en code. Il n'y a eu ici aucun problème vu que les notions sont assez simples à comprendre à part la compression et décompression via l'algorithme d'Huffman mais étant déjà créé avant par le groupe qui s'occupe de la compression textuel il n'y a pas eu de problème. Le défi a été de pouvoir reproduire entièrement le fichier JPEG afin que notre système d'exploitation puisse l'ouvrir de lui-même.

Malheureusement, dans cette partie dû au manque de connaissance sur l'encodage JPEG et à un manque de temps, il a été difficile de pouvoir le reproduire. Pour vous dire il ne nous manque qu'à réussir à décoder les informations après la compression de Huffman sinon toutes les autres informations du fichier sont comprises et reproduisibles .

4 Efficacité de la compression JPEG

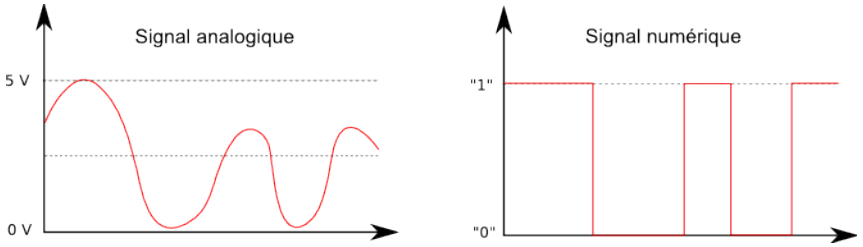
Après avoir fait cela comme expliqué, avant même d'essayer d'appliquer Huffman et donc de faire une Transformé de Fourrier nos images de bases ont été divisées par 2. En considérant, grâce à la partie d'après, que la compression de Huffman représente une compression de 30% des données. On peut en conclure que le JPEG à une compression de 85% cette information est vérifiable sur plusieurs images de taille différente (visible tableau annexe n°3). Nous pouvons quand même interpréter ce pourcentage comme étant normal du au fait que nous utilisons une moyenne sur les premières valeurs et qu'en plus nous faisons une quantification qui sont des actions irréversibles dû aux types de données que nous avons.

3.3. Le son

Comme évoqué dans les mots-clés du projet, nous aborderons tout d'abord l'encodage du son, c'est-à-dire le passage de l'analogique au numérique, étape clé permettant d'aborder par suite la compression par application de la théorie de Fourier.

3.3.1. De l'analogique au numérique

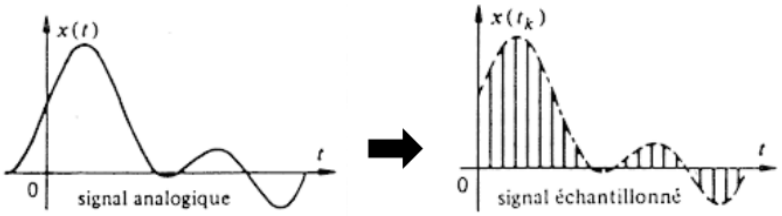
Le codage du son, ou encodage audio, consiste à convertir un signal sonore continu en un ensemble de points discrets représentant ce signal.



Pour ce faire, il nous faut introduire deux étapes clés : l'échantillonnage et la quantification.

1 Échantillonnage

L'échantillonnage est le processus de conversion d'un signal audio continu en un signal discret en prenant des échantillons à intervalles réguliers. C'est un signal discret en termes de temps, mais chaque valeur de l'échantillon peut encore prendre une infinité de valeurs possibles (continu en amplitude).



Un échantillon représente donc la mesure unique de l'amplitude du signal audio à un instant précis. Par ailleurs, la fréquence d'échantillonnage correspond au nombre d'échantillons pris par seconde, mesuré en Hertz (Hz).

Plus le nombre d'échantillons est important, plus l'échantillonnage est précis, mieux le signal est représenté.

2 Quantification

La quantification intervient après l'échantillonnage et a pour but de convertir chaque valeur échantillonnée en une valeur discrète parmi les niveaux disponibles (par exp 00, 01, 10, 11 si 4 niveaux).

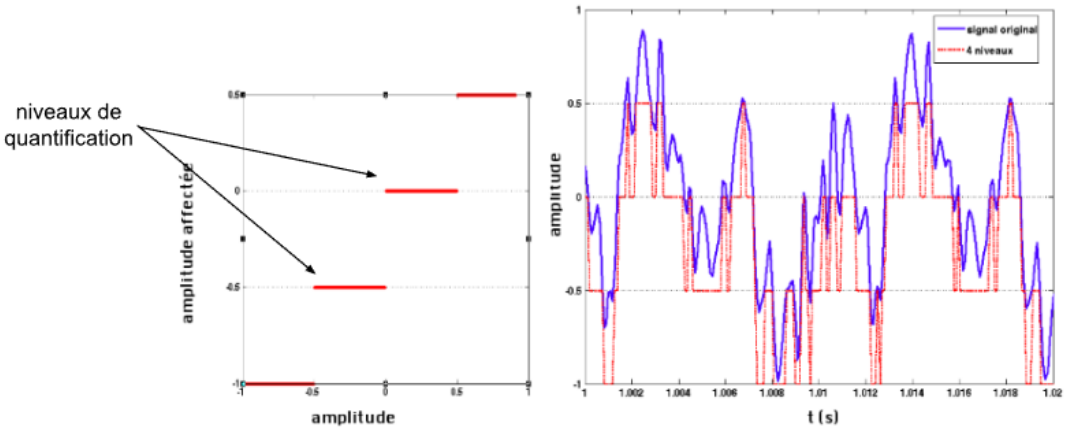
Le nombre de valeurs d'amplitudes que chaque échantillon peut prendre correspond aux niveaux de quantification disponibles. Et, ces niveaux de quantification dépendent de la

largeur choisie pour un échantillon, c'est-à-dire, du nombre de bits utilisés pour représenter chaque échantillon audio.

Les largeurs d'échantillons courantes sont :

- 1 octet : 8 bits (pour les applications de base, telles que les anciens jeux vidéo ou les télécommunications)
- 2 octets : 16 bits (Standard pour les CD audio, offrant une bonne qualité sonore adaptée à la plupart des usages)
- 3 octets : 24 bits (studios d'enregistrement professionnels)

→ Nombre de *niveaux de quantification* = 2^n avec n : nombre de bits pour la largeur d'un échantillon

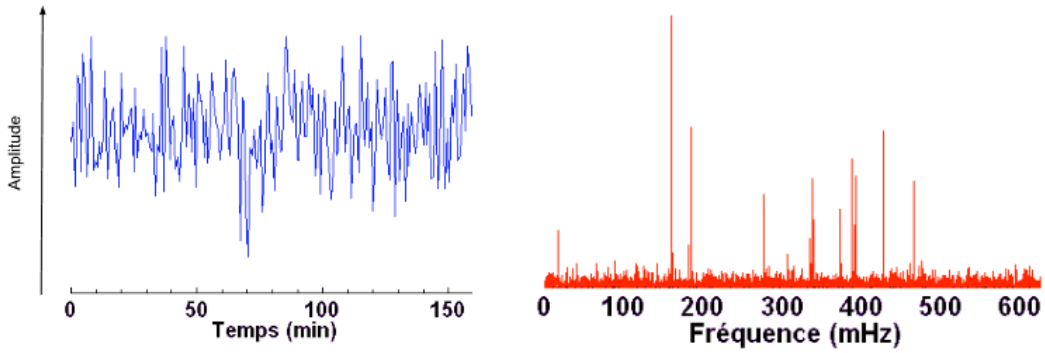


Signal quantifié sur 2 bits

Plus il y a de niveaux de quantification, plus les valeurs d'amplitudes sont bien représentées et donc plus la représentation du signal numérique sera précise.

3.3.2. Transformation de Fourier et applications

1 Le rôle de la transformation de Fourier discrète



Représentation temporelle du signal Représentation fréquentielle du signal

La Transformée de Fourier discrète (DFT) est une méthode permettant d'analyser les composantes fréquentielles d'un signal discret. Cette transformation facilite l'analyse et la manipulation des fréquences présentes dans le signal en identifiant notamment la contribution de chaque fréquence au signal global.

Plus concrètement, elle convertit un signal temporel en une série de coefficients complexes, chacun représentant une composante fréquentielle avec une amplitude et une phase spécifiques.

Les coefficients sont indexés par leur fréquence, c'est-à-dire que chaque coefficient correspond à une certaine fréquence dans le signal.

La formule générale de la transformation de Fourier discrète d'un signal s de N échantillons se note : $S(k) = \sum_{n=0}^{N-1} s(n) \mathrm{e}^{-2\mathrm{i}\pi k \frac{n}{N}} \quad \text{pour} \quad 0 \leqslant k < N.$

Ces coefficients peuvent être résumés de la manière suivante :

avec

-k : la fréquence

-A : l'amplitude

- phi : la phase

$$C_k = A_k e^{i\Theta_k}$$

Compression

Pour pouvoir compresser notre signal à l'aide de la Transformation de Fourier, nous pouvons manipuler ses coefficients.

Tout d'abord, les coefficients dont les amplitudes sont faibles peuvent être réduits à zéro. Les fréquences correspondantes sont considérées comme du bruit ou des détails qui n'affectent pas significativement la qualité du signal reconstruit.

Pour cela, un seuil est d'abord défini pour décider quels coefficients sont significatifs. Les coefficients dont l'amplitude dépasse ce seuil sont jugés importants pour le signal. Les coefficients dont l'amplitude est inférieure à ce seuil sont mis à zéro.

De plus, afin de réduire la quantité d'information à stocker, il est possible d'arrondir les amplitudes en appliquant le procédé de Quantification pour les Amplitudes.

Pour terminer, le signal compressé est reconstruit à partir des coefficients significatifs en appliquant la Transformée de Fourier inverse (IDFT).

2 Application et résultats

Dans le but d'implémenter la compression, nous avons développé un programme en Python.

Chargement des données du fichier source

Pour manipuler nos données audio, nous avons utilisé le module Python `soundfile`. Ce module nous permet d'obtenir les données audio (`signal`), la fréquence d'échantillonnage (`sample_rate`), le nombre d'échantillons (`frame_count`) et le nombre

de canaux (`channel_counts`), ce dernier correspondant au nombre de pistes audios : 1 pour un fichier mono et 2 pour un fichier stéréo…

Décomposition du signal en coefficients de Fourier

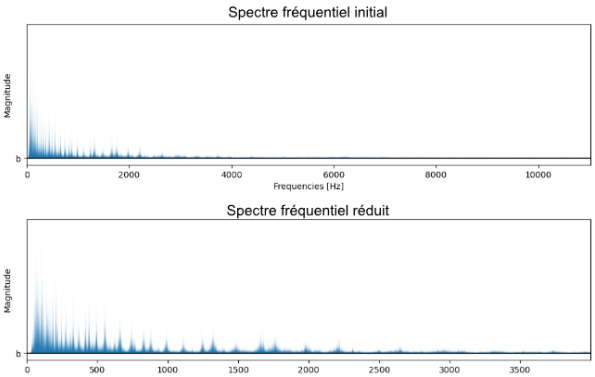
Nous avons appliqué la DFT à l'aide de la fonction du module Numpy `np.fft.rfft` qui renvoie les coefficients complexes expliqués précédemment, associés à leur fréquence (frequencies)

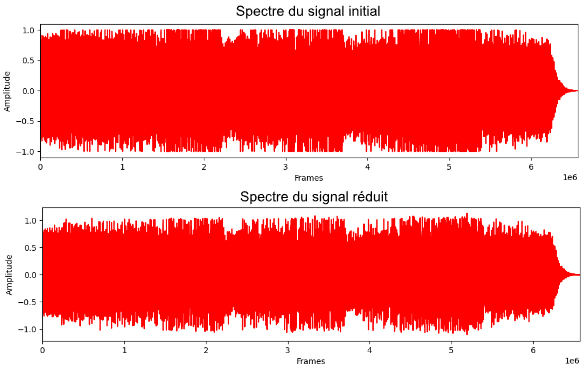
Réduction du nombre de Fréquences

Par la suite, nous avons cherché à filtrer les fréquences et les amplitudes en implémentant la fonction `filt(f)` qui ne conserve que les fréquences inférieures à une certaine limite (`MAX_FREQENCY`). Un pourcentage de fréquences supprimées est disponible.

Affichage

Afin de visualiser la réduction, nous avons affiché (sous forme de plot avec le module `matplotlib`) le spectre fréquentiel (Amplitude en fonction des fréquences) initial ainsi que celui avec les données réduites. Puis, nous avons fait de même avec les signaux (Amplitude en fonction des échantillons).





Décompression et reconstruction du signal

Les données compressées sont ensuite sauvegardées dans un fichier. On utilise le format `.npz` de NumPy pour sauvegarder plusieurs tableaux (nombre d'échantillons, fréquence d'échantillonnage, coefficients complexes) dans un fichier compressé.

On applique ensuite la Transformée de Fourier Inverse à l'aide de la fonction `np.fft.irfft` pour reconstruire le signal audio à partir du spectre de magnitude réduit. Enfin, le signal reconstruit est ensuite sauvegardé dans un fichier audio type WAV.

4. CONCLUSIONS ET PERSPECTIVES

Ce projet a été pour nous l'occasion de découvrir pour certains ou d'approfondir pour d'autres ce qu'est la compression. Mais bien que ce projet nous ait permis de faire des recherches concernant celle-ci, de découvrir certains algorithmes et d'en implémenter, le sujet est si vaste, qu'il ne nous a pas été possible de le traiter sous toutes ses coutures. Nous avons en outre pu introduire l'algorithme phare de ce domaine qu'est l'algorithme d'Huffman, mais aussi aborder des points assez techniques concernant la compression d'image tels que : LZ77, le format JPEG ou PNG et ce qui les entoure; ainsi que présenter brièvement la compression du son à travers des outils forts utiles comme la théorie de Fourier par exemple. Seulement, à la fin de ce projet, le sentiment qui nous imprègne est le même pour tous les membres du groupe. Nous n'avons fait qu'effleurer la surface d'un océan immensément grand, où les vagues apparentes sur lesquelles nous avons surfé, cachent une variété sous-marine extraordinaire.

Il n'est pas anodin de noter que nous n'avons pas, à un seul moment de ce projet, aborder la compression du format d'information le plus couramment utilisé dans notre société hyperconnectée par les réseaux sociaux ou autres types de médias. Ce format est bien évidemment la vidéo. Techniquement, la compression de vidéos est un autre monde, cette variété évoquée ci-avant. Mais, il est clair que la traversée de l'océan de la compression a attisé notre soif inextinguible de connaissances, en particulier concernant une science au cœur des enjeux actuels et futurs tels que : l'énergie ou encore l'écologie.

Ce projet fut en somme, un excellent moyen pour nous d'aborder, sous le prisme d'étudiants de deuxième année à l'INSA, une thématique clé de l'ingénierie, la compression.

5. BIBLIOGRAPHIE

- [1] https://fr.wikipedia.org/wiki/Portable_Network_Graphics (valide à la date du 11/06/2024).

- [2] https://fr.wikipedia.org/wiki/LZ77_et_LZ78 (valide à la date du 11/06/2024).

- [3] https://www.youtube.com/watch?v=-V48ZygMUtg (valide à la date du 11/06/2024).

- [4] https://www.w3.org/TR/png/ (valide à la date du 11/06/2024).

- [5] https://www.ietf.org/rfc/rfc1951.txt (valide à la date du 11/06/2024).

- [6] https://www.ietf.org/rfc/rfc1950.txt (valide à la date du 11/06/2024).

- [7] https://pyokagan.name/blog/2019-10-14-png/ (valide à la date du 11/06/2024).

- [8] https://pyokagan.name/blog/2019-10-18-zlibinflate/ (valide à la date du 11/06/2024).

- [9] https://www.youtube.com/watch?v=X42MZqXHXDM (valide à la date du 11/06/2024).

- https://fairyonice.github.io/2D-DCT.html (valide à la date du 11/06/2024)

- https://yasoob.me/posts/understanding-and-writing-jpeg-decoder-in-python/ (valide à la date du 11/06/2024)

- https://www.ipol.im/pub/art/2022/399/article_lr.pdf (valide à la date du 13/06/2024)

- https://www.irif.fr/~francoisl/DIVERS/m1algohuffman2324.pdf (valide à la date du 13/06/2024)

- https://github.com/Filoji/Fennec-File-Format(valide à la date du 13/06/2024)

- https://www.programiz.com/dsa/huffman-coding (valide à la date du 13/06/2024)

- https://fr.wikipedia.org/wiki/Hypertext_Transfer_Protocol (valide à la date du 13/06/2024)

- https://www.nussbaumcpge.be/public_html/Sup/MP2I/huffman.pdf (valide à la date du 13/06/2024)

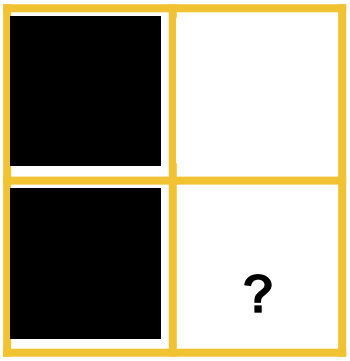
- http://tpedescartes.free.fr/lecodehuffman.html (valide à la date du 13/06/2024)

- http://perso.iut-nimes.fr/flouchet/DSP/TP07.PDF (valide à la date du 13/06/2024)

- https://culturesciencesphysique.ens-lyon.fr/ressource/numerisation-acoustique-Chareyron1.xml (valide à la date du 13/06/2024)

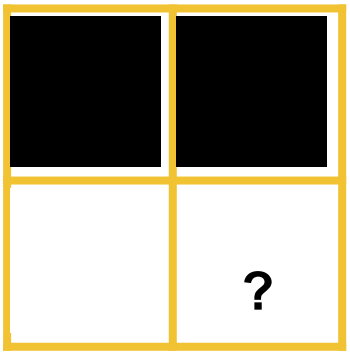
6. ANNEXES

Annexe n°1: Illustration filtre 5 (noir = 0, gris = 127, blanc = 255)



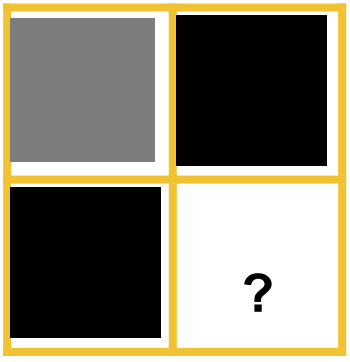
Ici pa = |255-0| = 255, pb = |0-0| = 0, pc = |0+255-2*0| = 255

pb est la plus petite donc le sens de variation des couleurs est horizontal, donc il y a plus de chance pour que x soit blanc donc on lui soustrait b.



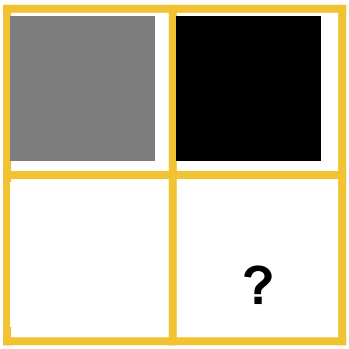
Ici pa = |0-0| = 0, pb = |255-0| = 255, pc = |255+0-2*0| = 255

pa est la plus petite donc le sens de variation des couleurs est vertical, donc il y a plus de chance pour que x soit blanc donc on lui soustrait a.



Ici pa = |0-127| = 127, pb = |0-127| = 127, pc = |0+0-2*127| = 254

pa est la plus petite donc le sens de variation des couleurs est vertical, donc il y a plus de chance pour que x soit noir donc on lui soustrait a.



Ici pa = |0-127| = 127, pb = |255-127| = 128, pc = |255+0-2*127| = 1

pc est la plus petite donc le sens de variation des couleurs est diagonal, donc il y a plus de chance pour que x soit gis donc on lui soustrait c.

Annexe n°2: Comparatif des filtres

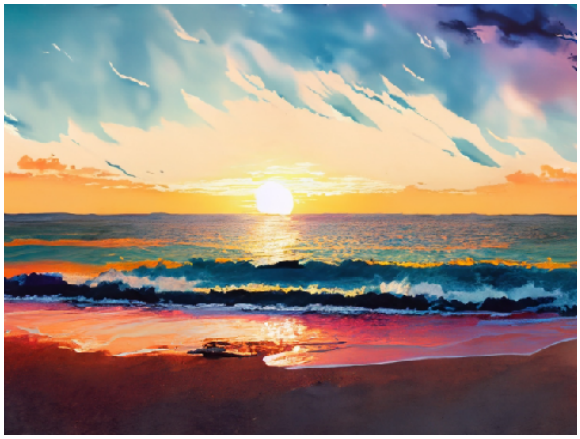


Image de base au format BMP (non-compressée)
taille: 506 Ko

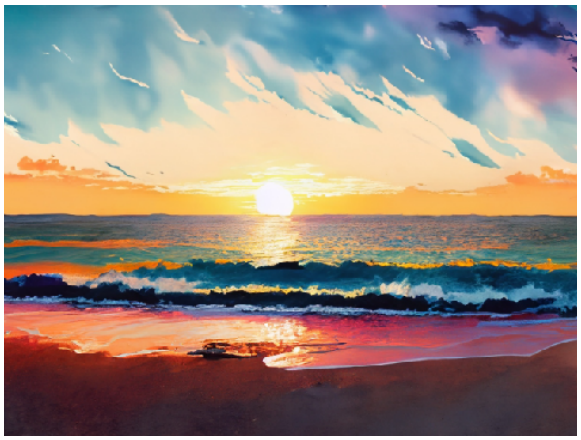


Image au format PNG (correspond aussi à la vue du filtre 1)
taille: 476 Ko
score: 34 297 284

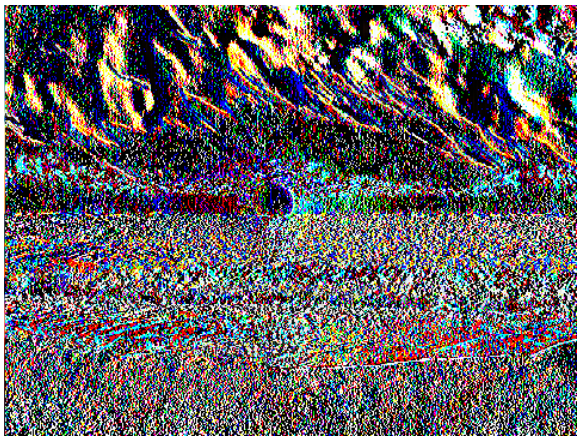


Image vue avec le filtre 2
taille: 392 Ko
score: 1 849 760

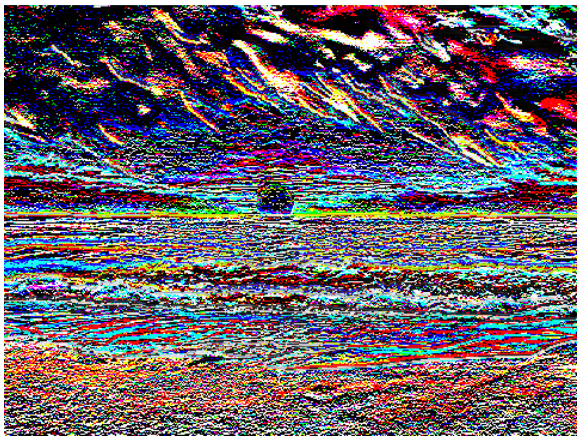


Image vue avec le filtre 3
taille: 412 Ko
score: 3 754 614

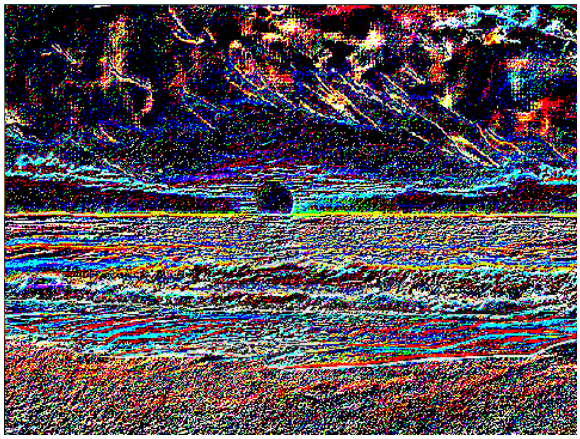


Image vue avec le filtre 4
taille: 385 Ko
score: 2 358 585

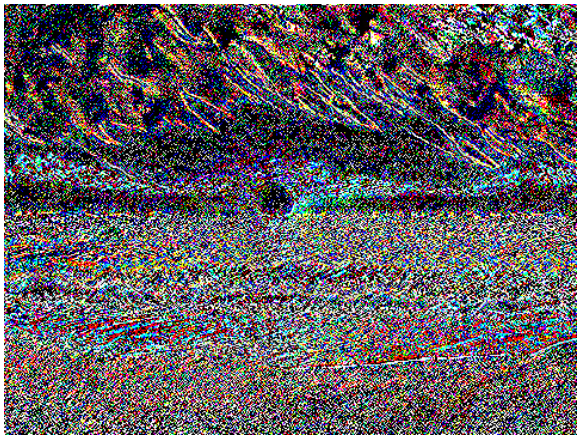


Image vue avec le filtre 5
taille: 386 Ko
score: 1 809 930

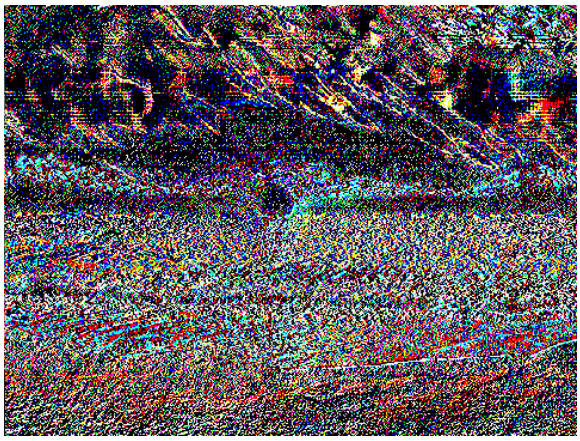


Image vue avec la meilleure combinaison de filtre
taille: 378 Ko
score: 1 697 794

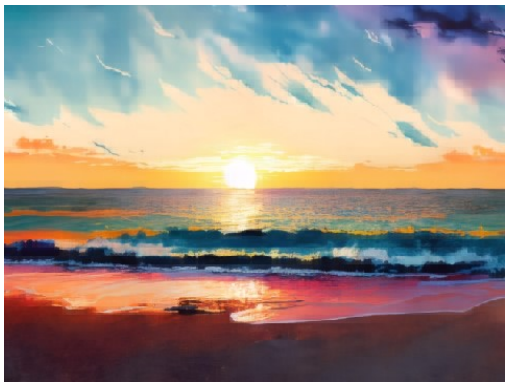


Image compressé via algorithme de JPEG

Poids JPEG(en ko)	Poids BMP (en ko)	Pourcentage de compression
47	469	89,97867804
69	507	86,39053254
63	853	92,61430246
190	2452	92,25122349
157	4097	96,16792775
720	6076	88,15009875
194	9681	97,99607479
13	197	93,40101523
2653	60313	95,60127999
375	2606	85,61013047
903	37969	97,62174405
247	5860	95,78498294
147	2629	94,40852035
52	8386	99,37991891
		93,23974498

Annexe n°3 : Tableau compression image JPEG

Annexe n°4: Table de longueur

Symbol	Extra bits	Length(s)
257	0	3
258	0	4
259	0	5
260	0	6
261	0	7
262	0	8
263	0	9
264	0	10
265	1	11,12
266	1	13,14
267	1	15,16
268	1	17,18
269	2	19-22
270	2	23-26
271	2	27-30
272	2	31-34
273	3	35-42
274	3	43-50
275	3	51-58
276	3	59-66
277	4	67-82
278	4	83-98
279	4	99-114
280	4	115-130
281	5	131-162
282	5	163-194
283	5	195-226
284	5	227-257
285	0	258

Annexe n°5: Table de distance

Symbol	Extra Bits	Distance(s)
0	0	1
1	0	2
2	0	3
3	0	4
4	1	5,6
5	1	7,8
6	2	9-12
7	2	13-16
8	3	17-24
9	3	25-32
10	4	33-48
11	4	49-64
12	5	65-96
13	5	97-128
14	6	129-192
15	6	193-256
16	7	257-384
17	7	385-512
18	8	513-768
19	8	769-1024
20	9	1025-1536
21	9	1537-2048
22	10	2049-3072
23	10	3073-4096
24	11	4097-6144
25	11	6145-8192
26	12	8193-12288
27	12	12289-16384
28	13	16385-24576
29	13	24577-32768