Par :: Yann Guidon

Introduction à la compression de données : mise en évidence de l'entropie

L'article précédent a examiné les caractéristiques de notre fichier « de référence » qui nous permettra de tester nos algorithmes avec des données représentatives et adaptées.

Cependant, seules des notions de base en Traitement Numérique du Signal ont été survolées. Bien qu'indispensables pour la suite, ces techniques ne concourent pas directement à la réduction de la taille de nos données. En partant de la théorie de Shannon, nous allons maintenant examiner l'entropie de différents types de fichiers et comprendre pourquoi on arrive à réduire leur taille (lorsque c'est possible).

1 Qu'est-ce que la compression ?

Comprimer une information consiste à changer sa représentation dans le but de réduire la place qu'elle occupe. Ce n'est peut-être pas une définition rigoureusement scientifique mais cela correspond tout à fait à ce que nous faisons en pratique. Si on accepte que des informations soient perdues, cela s'appelle une « réduction ». Le programme utilise alors une transformation irréversible (comme pour les algorithmes de codage des formats JPEG et MPEG) mais cette perte est intolérable pour traiter les textes, les programmes informatiques ou des enregistrements sonores originaux, entre autres.

Comme les domaines connexes, en particulier la cryptologie et la correction d'erreurs, la compression est une science pluridisciplinaire qui a pris son essor au siècle dernier, lorsque les ordinateurs et les systèmes de communication sont apparus. Et comme pour les autres domaines, il faut avoir une excellente connaissance des propriétés des informations à traiter.

1.1 Théorie de l'Information

Les ordinateurs utilisent des *bits* qui peuvent être vus comme des atomes pour coder les nombres. Pourtant les informations ne sont pas nécessairement de nature binaire, l'utilisation de statistiques entraîne l'apparition de nombres décimaux (oui, des nombres non entiers, avec une virgule !). La clé de voûte de tout ce qui concerne la compression et la cryptologie est le concept d'**entropie**, qui est la **quantité d'information** contenue dans un signal ou un message par exemple.

Le mot « entropie » (dérivé du mot grec signifiant « retour », « transformation ») a d'abord été associé au 19e siècle au travail de Ludwig Bolzmann (1844-1906) dans le domaine de la thermodynamique (l'étude de l'énergie et de ses transformations). L'entropie de Bolzmann est une mesure du degré de désorganisation ou des degrés de liberté, d'un milieu. Il faut aussi préciser que ce n'est pas une mesure quantitative d'énergie, mais qualitative de sa répartition. Pour une définition plus juste, référez-vous aux ouvrages de thermodynamique et en particulier sur sa deuxième Loi qui stipule que l'entropie totale d'un système ne peut pas diminuer. En 1948, Claude E. Shannon (1916-2001) a étudié le domaine du traitement et de la transmission de l'information numérique et déduit une formule qui, à un facteur multiplicatif près, est de même forme que celle de Bolzmann. En raison d'autres similitudes, comme par exemple la signification très proche de la grandeur représentée (qui tourne autour du désordre, de l'incertitude et fondamentalement : l'inconnu), il a gardé le nom d'entropie. C'est de cette dernière dont il est question dans cet article.

1.2 Définition de l'entropie d'un flux de données

L'entropie d'un fichier est communément quantifiée en « bits par caractère/symbole/échantillon » selon qu'il s'agit d'un texte, d'un signal échantillonné ou d'autre chose... Et ce nombre de bits n'est pas forcément entier. Pour quantifier l'information, on se sert de l'énoncé de Shannon :

L'entropie H d'un symbole S de probabilité P à l'intérieur d'un flux de données est donnée par

$$H(S) = P * \log2(1/P)$$

Cette définition s'applique seulement à un flux de données absolument aléatoire, totalement dépourvu de corrélations. Idéalement, il s'agit d'un « *générateur sans mémoire* », c'est-à-dire : dont la sortie ne peut pas être déterminée en fonction des données précédentes.

Qu'est-ce qu'un codeur d'entropie ?

Ce terme est incontournable dans l'étude de la compression et il apparaîtra souvent dans cette série, parfois de manière générique. Je vais le décrire ici succinctement et préciser le fonctionnement exact dans d'autres articles. Un codeur d'entropie est un dispositif qui transforme de manière réversible un flux de symboles ou de nombres en un flux binaire dont la taille tend vers l'entropie de Shannon.

La plupart du temps, c'est la dernière étape d'un programme de compression, transformant les symboles « en clair » en une « mixture de bits » prenant moins de place.

Au passage, les données ne sont plus alignées sur des frontières de mots afin de réduire l'encombrement au minimum. Il existe plusieurs catégories de tels codeurs aux propriétés différentes et aux variations (donc appellations) nombreuses. Les trois familles suivantes sont les plus courantes :

Les codes à longueur variable, comme Elias ou Rice :
Ce font les codes les plus simples, transformant tout nombre en un code binaire unique de taille approximativement proportionnelle à la valeur du nombre. Les plus courants sont composés d'un préfixe uniforme suivi d'un bit « terminateur » ou « séparateur » dont la valeur est complémentaire à celle du préfixe (c'est donc une convention arbitraire).

Vient ensuite un éventuel « exposant » et enfin un champ binaire de « mantisse ». Le préfixe étant décodé en premier, il donne des indications sur la taille de l'exposant qui donne lui-même la taille de la mantisse... Ce système de poupées russes (parfois même récursif) peut être modulé à loisir pour donner naissance à des codes spécifiques.

Valeur à coder	codage Rice	codage « Pod »	codage Elias/ Gamma	Start-Step-Stop {1,2,N}	Start-Step-Stop {2,2,N}
0	**1**	**1**	**1**	**0** 0	**0** 00
1	0**1**	0**1**	01 0	0**1**	**0** 01
2	00**1**	00**1** 0	0**1** 1	**10** 000	**0** 10
3	000**1**	00**1** 1	00**1** 00	**10** 001	**0** 11
4	0000**1**	000**1** 00	00**1** 01	**10** 010	**10** 0000
5	0000**1**	000**1** 01	00**1** 10	**10** 011	**10** 0001
6	000000**1**	000**1** 10	00**1** 11	**10** 100	**10** 0010
7	0000000**1**	000**1** 11	000**1** 000	**10** 101	**10** 0011
8	00000000**1**	0000**1** 000	000**1** 001	**10** 110	**10** 0100
9	000000000**1**	0000**1** 001	000**1** 010	**10** 111	**10** 0101
10	0000000000**1**	0000**1** 010	000**1** 011	11**0** 00000	**10** 0110

Quelques codes binaires simples tirés du site de Robin White dont l'URL est donné en fin d'article. Le bit « séparateur » est indiqué en gras. Le MSB doit être inséré en premier dans le flux comprimé

Les résultats sont concaténés dans le flux binaire de sortie, qui prend normalement moins de place en moyenne que les données originales.

Par exemple, le code Rice est généralement préféré pour coder des nombres très petits (entre 0 et 4 par exemple) et il est optimal pour des séries dont la distribution est laplacienne (où la probabilité de rencontrer le nombre N est proportionnelle à 2^N. Autrement, le résultat est médiocre et d'autres codes plus complexes existent comme les codes *taboo, start/stop, start/step/stop*, et de nombreux autres dont les propriétés sont paramétrables pour s'adapter aux données codées.

La relation presque linéaire entre la valeur du nombre et la taille de son code fait que cette famille de codes est inutilisable dans les compresseurs généralistes (capables de comprimer des textes ou des programmes), mais sa simplicité la rend attractive pour les compresseurs de signaux. De plus, on peut l'utiliser pour coder un nombre isolé, contrairement aux autres familles de codes qui nécessitent un grand nombre de données pour être efficaces.

Ces codes sont les plus faciles à coder, les plus variés, les plus anciens, les plus tolérants aux erreurs de transmission (leur sous-efficacité permet une resynchronisation naturelle du flux de sortie après le changement d'un ou plusieurs bits) mais ils demandent un grand soin dans leur sélection (manuelle) et leur utilisation.

Les algorithmes de Fano-Shannon ou Huffman :
Cette famille associe encore à chaque symbole en entrée un code binaire de taille variable en sortie.

Cependant, chaque code binaire est déterminé non pas par une formule mathématique mais au démarrage du flux, par un algorithme qui analyse les statistiques du flux d'entrée. Cette analyse permet de mieux s'approcher de l'entropie de Shannon dans les cas où il n'y a pas de relation de proportionnalité entre la valeur du symbole et sa probabilité d'utilisation (c'est le cas des textes au format ASCII, par exemple).

L'analogie la plus évidente est le code morse qui associe une lettre (pour nous, c'est un symbole) à une suite de traits et points (nos 0 et 1 du flux binaire en sortie). La longueur de chaque code est inversement proportionnelle au nombre moyen d'apparitions dans un texte : le `e` prend moins de place (donc de traits/points ou de bits) que le `x`.

La différence avec le code Morse est que l'espace temporel qui délimite deux caractères est impossible à coder avec des `Ø` et des `1`, il faut donc faire attention à ce que la table de traduction ne soit pas « ambiguë » (il ne faut pas qu'un code soit le début d'un autre code, par exemple `1Ø` et `1Ø11`, ce dont on s'assure en construisant un arbre binaire).

L'algorithme de Huffman, très courant, est une amélioration de celui de Fano-Shannon. Ils génèrent la table de codage des symboles mais cette table doit être incluse dans le flux de sortie. Il existe des techniques de codage sophistiquées pour que cette table prenne le moins de place possible.

De plus, cette table doit être placée en tête du flux comprimé, afin que le décodeur sache comment interpréter la suite. En conséquence, le flux d'entrée doit être analysé complètement avant de pouvoir commencer à émettre les premières données comprimées. Pour éviter de faire deux passes, certains programmes utilisent des versions adaptatives (la table est redéfinie au fur et à mesure du flux) ou bien incluent des tables prédéfinies dans le fichier exécutable.

Cette dernière solution utilise par exemple les statistiques d'un type de fichier courant comme un texte en anglais. Les formats JPEG et Vorbis définissent aussi des tables « de référence » qui conviennent à la majorité des signaux mais des marqueurs sont prévus pour pouvoir, au besoin, insérer une table optimisée pour un fichier particulier.

Les codeurs arithmétiques ou d'intervalle
Ils sont plus efficaces que les précédents car le flux de sortie n'est pas une concaténation de codes binaires plus ou moins longs, mais parce que le flux est vu comme un nombre de précision presque infinie.

Les algorithmes considèrent la sortie comme un nombre entre `Ø` et `1`, et codent non pas un symbole, mais l'intervalle assigné à ce symbole. En sortie, il n'y a plus de frontière bien définie entre les symboles codés, ce qui oblige à décoder complètement le flux si on recherche un délimiteur particulier. C'est encore plus compliqué si on veut couper le flux en morceaux indépendants, ce qui ne pose pas de souci avec les

Qu'est-ce qu'un codeur d'entropie ? (suite)

algorithmes utilisant des codes « discrets » (non confondus avec les voisins, comme les deux familles précédentes).

Les opérations mathématiques utilisées sont les simples additions – soustractions – multiplications – divisions que nous connaissons mais appliquées à un flux qui est un nombre à précision variable. Des optimisations mathématiques permettent cependant de limiter la précision effective des calculs.

Les versions les plus simples utilisent aussi un modèle statistique déterminé par une analyse préalable des données lors d'une première passe. Des versions adaptatives ou avec des probabilités plus complexes sont souvent utilisées pour ne nécessiter qu'une seule passe ou améliorer le taux de compression.

Pour ces deux raisons et d'autres encore, comme les brevets, la nécessité d'utiliser des multiplications ou la vitesse de codage, cette famille est peu utilisée. Par exemple, le format JPEG prévoit la possibilité d'utiliser des codeurs arithmétiques mais en pratique, les tables de Huffman prédéfinies sont les plus employées.

Comme pour les codes de Huffman, coder arithmétiquement un nombre isolé n'a pas d'intérêt et la transmission des statistiques implique les mêmes compromis.

1.3 Un autre type de générateur de données

Une bonne illustration serait un flux créé par un détecteur de particules, qui mémoriserait successivement leur type. Le fichier généré serait donc composé de symboles correspondant, dans le cas de désintégrations d'un matériau radioactif, aux particules `alpha`, `beta` et `gamma`.

γ	β	γ	α	β	γ	α	γ	α
β/	α	β	β	α	γ	α	γ	β
α	β	...						

:: Flux aléatoire « sans mémoire » généré par un détecteur de particules imaginaire ::

Ce dispositif fictif a été fabriqué dans un laboratoire de fiction où un stagiaire imaginaire s'est dit que le meilleur moyen de représenter les particules serait d'utiliser la table suivante :

symbole code
α = 00
β = 01
γ = 10
11 est réservé

C'est un codage qui a l'avantage d'être simple et de permettre à quatre particules de tenir dans un octet. La recherche d'un événement est donc très facile puisque le numéro du bit et de l'octet est donné facilement à partir du numéro de la particule.

Il reste aussi de la place (le code `11`) pour éventuellement coder une quatrième particule dans le futur, si on devait en découvrir une autre ou même signaler (« échapper ») le début d'une zone des données annexes, contenant par exemple des *timecodes* ou des paramètres de mesure.

10	01	10	00	01	10	00	10	00
01	00	01	01	00	10	00	10	01
00	01	...						

:: Flux codé avec deux bits par symbole/particule ::

Tout allait bien jusqu'à ce qu'une réduction du budget du laboratoire imaginaire incita le directeur à remettre en cause les conditions d'utilisation de la coûteuse armoire de stockage de cinquante pétaoctets, chargée d'archiver toutes les mesures.

Le laboratoire fit donc appel à un doctorant qui aurait plus de bagage théorique pour analyser le système de codage en profondeur, libérant ainsi de l'espace pour d'autres projets. Le doctorant découvrit vite comment s'y prendre après quelques observations.

La première approche consiste à coder les particules avec un nombre variable de bits, pour en économiser par-ci par-là. Il s'aperçoit que l'espace de codage est sous-utilisé puisque le code `11` n'est jamais exploité (le premier développeur n'ayant jamais eu le temps de programmer le codage des métadonnées). Cela représente une efficacité de 75% seulement, laissant espérer un gain d'espace de 12 pétaoctets.

Notre doctorant va donc utiliser un « code à longueur variable » très simple, puisque le nombre de symboles est défini à l'avance (il ressemble ainsi à un *phasing-in code* qui sera utilisé dans d'autres articles).

symbole code
α = 0
β = 10
γ = 11

Cette table de codage arbitraire permet bien de gagner de la place mais on perd la possibilité de faire des recherches rapides dans les fichiers. Il faut les balayer entièrement depuis le début pour reconstituer la frontière entre chaque code et savoir quelle particule est à une certaine position.

11	10	11	0	10	11	0	11	0
10	0	10	10	0	11	0	11	10
0	10	...						

:: Flux utilisant un code à longueur variable
Gain : 40-33=7 bits (21% de réduction de taille) ::

1.4 Optimisation du codage

Notre doctorant remarque ensuite que le flux est totalement décorrélé puisqu'il est théoriquement impossible de prédire quand la désintégration naturelle d'un atome a lieu. Il n'y a pas non plus d'effet de bord au début et à la fin du flux car le phénomène observé est indépendant de la mesure, à condition que la période de décroissance radioactive soit très supérieure à la durée de la mesure.

Par contre, on connaît bien le nombre et le type de particules créées par la désintégration en fonction du matériau. La probabilité de chaque type de particule dans le flux est une sorte de « signature » du matériau radioactif.

Ce que Shannon explique, c'est la contribution de chaque symbole (représentant ici une particule d'un type identifié) à l'information contenue dans le flux.

Par exemple, si la probabilité d'apparition d'une particule `alpha` est plus grande que celle d'une particule `beta`, son entropie est plus faible car le symbole apparaît plus souvent et on peut allouer moins de bits à ce symbole pour occuper moins de place.

C'est ce que suppose la deuxième table de traduction : en assignant le code le plus court (`0`) à la particule `alpha`, le gain de place est maximal si c'est la particule la plus courante. Dans le cas contraire, le gain est marginal : il faut donc adapter la table de traduction aux données à comprimer ! C'est la fonction des algorithmes de Fano-Shannon et de Huffman.

Shannon a démontré que pour que la représentation soit idéalement compacte, le nombre de bits pour représenter chaque symbole dépend du logarithme de (l'inverse de) la probabilité de ce symbole.

Une fois que le flux est compacté, il ne devrait plus comprendre que des zéros et des uns équiprobables. La preuve de cette affirmation est que s'ils n'étaient pas équiprobables, il serait alors possible de comprimer le signal encore plus en exploitant cette différence de probabilité.

A ce sujet, on remarque que si la particule `alpha` représentait une très grande majorité du flux à coder, le nombre de bits à `0` dans le flux binaire de sortie serait supérieur au nombre de bits à `1`, signifiant la sous-efficacité du codage, qui a donc ses limites.

10	0	10	0	11	0	0	10	0
0	10	11	0	10	0	11	0	0
0	10	...						

:: Exemple de flux compressé contenant 17 0 pour 12 1 : le codage n'est donc pas optimal ::

Une autre manière de montrer les limites des codes à longueur variable est d'imaginer un flux composé à 90% de symboles `alpha` : Shannon nous dit que chacun devrait occuper 0,1368 bits mais la taille minimale du code est 1 bit. L'inefficacité est flagrante d'autant plus qu'ici la réduction maximale de la taille est de 50% (en passant d'un code à deux bits à un code à un seul bit).

Les codes à longueur variable décrits ici ont donc des défauts importants lorsque les probabilités s'écartent de l'idéal. Les codeurs arithmétiques n'ont pas ces problèmes et sont donc plus efficaces.

L'entropie d'un flux correctement comprimé s'approche donc d'un bit par bit, comme pour un signal aléatoire. D'ailleurs, un fichier comprimé avec un bon algorithme peut parfois être utilisé pour créer des séquences pseudo-aléatoires mais celles-ci sont de qualité médiocre car ce ne sont pas des générateurs « sans mémoire ».

Armé de ces principes de base, nous allons pouvoir examiner et compresser nos signaux, en partant de l'hypothèse qu'ils sont aléatoires puis en affinant leur description.

1.5 Définition de l'entropie d'un flux de données

En pratique, les fichiers informatiques ont souvent une longueur finie et les informations sont traitées par petits blocs. On ne traite alors plus des probabilités mais des occurrences.

En posant $P = 1/N$, N étant le nombre d'occurrences de `S` dans un message de longueur finie, on obtient la formule modifiée :

$$H(S) = \log_2(N) / N$$

L'entropie du message (fichier, flux ou bloc de ceux-ci) est la somme de l'entropie de tous les symboles.

Et pour la calculer, il faut d'abord fabriquer l'histogramme du bloc, c'est-à-dire compter le nombre d'occurrences de chaque symbole.

1.6 Applications numériques

Si nous prenons un texte aléatoire, utilisant équiprobablement les 26 lettres de l'alphabet, il faudra

$$26 \text{ (lettres)} \times 1/26 \text{ (chance d'apparaître)} \times \log_2(1/(1/26)) = \log_2(26) = 4{,}7 \text{ bits par caractère}$$

pour coder tout texte arbitraire (sans espace, ni majuscule ou ponctuation), sans contexte et de longueur infinie.

Lorsque les probabilités sont égales pour chaque symbole, la formule est ainsi réduite à un simple logarithme.

On reconnaît ainsi la méthode utilisée pour coder un nombre pouvant prendre N valeurs : il faut

$$\log_2(N) \text{ bits}$$

En pratique, ce résultat est souvent arrondi à l'entier supérieur.

Dès que la probabilité de chaque symbole est prise en compte, il faut gérer les occurrences et inclure le nombre total de symboles dans le calcul :

CECI EST UN TEXTE SIMPLE

Lettre	Probabilité	Entropie du caractère
C	2/24	0,298746875
E	5/24	0,4714655
I	2/24	0,298746875
L	1/24	0,1910401
M	1/24	0,1910401
N	1/24	0,1910401
P	1/24	0,1910401
S	2/24	0,298746875
T	3/24	0,430827
U	1/24	0,1910401
X	1/24	0,1910401
(espace)	4/24	0,430827
	Somme :	3,076854 bits

Ainsi, sans autre information de contexte supplémentaire, ce message ne peut être codé en moins de `24 x 3,0768 = 73,84` bits.

En suivant cette méthode, on peut trouver la taille idéale d'un symbole pour chaque lettre.

Chaque symbole utilisera une place inversement proportionnelle au logarithme de sa probabilité d'apparition, donc les caractères les plus utilisés prendront moins de place.

Cette transformation d'un caractère vers un symbole est effectuée par un « codeur d'entropie », tel que l'algorithme de Huffman ou un codeur arithmétique. Nous allons programmer des codeurs d'entropie dans la suite de la série.

Il faut souvent une étape de collecte des statistiques puis l'élaboration d'une table de traduction et le codeur transforme ensuite chaque caractère de longueur fixe par un symbole de longueur variable, ce qui permet de réduire la place utilisée par les données.

Mais ce qui est absent du calcul de Shannon, c'est le moyen de coder chaque lettre et surtout le moyen d'indiquer au décompresseur/décodeur quelle lettre correspond à quel code. L'entropie nous indique une limite inférieure théorique, mais y parvenir est bien difficile.

1.7 D'autres moyens de gagner de la place

Nous utilisons généralement un octet (huit bits) pour coder un caractère et les textes ont des caractéristiques particulières, comme la probabilité d'apparition d'un caractère en fonction des caractères précédents (le « contexte » de l'information).

Claude Shannon a montré que pour un humain (anglophone), l'entropie réelle d'un texte en langue anglaise est d'environ un bit par caractère. En français, cela signifie qu'avec suffisamment d'informations sur le contexte, comme le sujet de la conversation, des souvenirs communs ou des références culturelles, il suffit généralement d'un `oui` ou d'un `non` de l'interlocuteur pour déduire la ou les prochaines lettres du message qu'il veut transmettre.

Par exemple, le contexte

Les roses rouge_

Indique avec une grande probabilité, d'après la langue utilisée (ressemblant fort bien au français) et les règles de grammaire, que la prochaine lettre sera un `s` puisque `rouge` est vraisemblablement l'adjectif du nom `roses`.

Pourtant, rien ne permet d'affirmer avec certitude que ce sera bien un `s`. Ce pourrait aussi être le verbe `rougeoient`, mais `s` est plus probable.

Pour connaître la prochaine lettre et probablement les suivantes, il suffit donc bien d'une question fermée :

« Est-ce que la prochaine lettre est 's' ? »

La réponse est `oui` ou `non`, codable sur un seul bit, ce qui correspond bien à une entropie d'un bit.

On pourrait encore aller plus loin en allouant un nombre de bits proportionnel à la probabilité de la réponse la plus évidente. Comme la réponse `oui` est très probable, on pourrait lui allouer une petite fraction de bit au moyen d'un codeur arithmétique.

1.8 Le piège des canaux parallèles

L'entropie n'est qu'une estimation de la limite inférieure de compressibilité et un décodeur seul ne permet pas toujours de restaurer les informations originales.

Aussi, il faut faire attention à ne pas créer inutilement de « canaux parallèles » : ce sont des informations qui peuvent parfois sembler anodines et qu'on oublie dans le bilan préliminaire de la compression. En déplaçant des informations, on peut croire augmenter le taux de compression alors que le contraire se produit.

C'est la question traitée implicitement par la Théorie Algorithmique de l'Information : dans quelle mesure peut-on déplacer des informations vers le programme sans le rendre inefficace sur d'autres données ou le rendre trop gros ? La somme des tailles du programme et des données compressées est-elle supérieure à la taille de la donnée originale ?

L'exemple du calcul de l'entropie d'une chaîne de caractères illustre ce problème : on pourrait réduire la taille de la chaîne précédente à 74 bits au lieu des 192 bits d'origine, mais il faut en plus compter les informations spécifiques au codage employé, comme les tables de traduction des symboles ou même la longueur du message. Le programme de compression pourrait contenir ces tables. Par exemple, certains compresseurs courants utilisent des tables fixes contenant des statistiques spécifiques aux textes en langue anglaise. Cela explique que leur efficacité varie beaucoup et n'est pas optimale pour la plupart des autres types de données.

Dans notre exemple, la table est spécifique et on ne peut pas utiliser de programme existant. Coder cette table dans un programme spécialisé reviendrait à déplacer une partie des données (les tables) vers le logiciel. Ce dernier ne permettrait pas de traiter d'autres données puisqu'il en contient une partie spécifique, même sous une autre forme.

Au final, la taille cumulée du message comprimé et du logiciel est plus grande que le message original : le bilan est négatif.

Il faudrait donc mémoriser les tables au début du fichier. Dans l'exemple numérique, le gain de compression réel serait probablement faible car la table prendrait peut-être beaucoup de place. Cette approche est plus adaptée aux très longues chaînes car la taille de la table (souvent au format fixe, quelques centaines d'octets) devient alors négligeable.

Dans certains cas, les « canaux parallèles » (ou informations annexes) peuvent poser des problèmes de gestion de flux, d'interdépendance ou de stockage. L'art du programmeur consiste donc à utiliser au mieux les différentes techniques existantes pour ne pas se retrouver piégé dans un algorithme compliqué qui comprime peu.

1.9 Un paradoxe de l'entropie ?

Il existe une sorte de « barrière d'entropie » : les méthodes de compression deviennent de plus en plus compliquées à mesure que le taux s'approche de cette barrière.

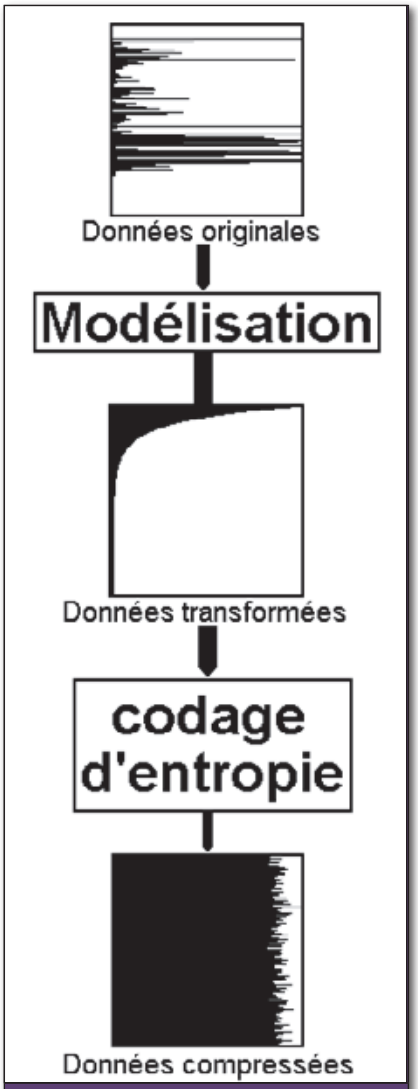
Par exemple, les meilleurs compresseurs d'aujourd'hui s'approchent de 1,9 bits par caractère pour un texte anglais, au prix de calculs très lourds, ce qui est encore loin de ce que Shannon avait évalué.

Les progrès sont incessants dans le domaine des compresseurs de textes par exemple, mais certains programmes doivent être épaulés par des bases de connaissances de dimensions encyclopédiques pour extraire un peu de sens des données. Dans ce cas, les efforts sont plus fournis dans le domaine de l'intelligence artificielle qu'en théorie de l'information.

L'efficacité d'un compresseur réside donc dans la sophistication du modèle développé autour de l'information à compresser. Un modèle général donnera des résultats modestes pour de nombreuses applications, alors qu'un modèle très élaboré sera spécialisé dans la compression efficace d'un type très particulier de données.

Une compression efficace est essentiellement liée à une modélisation sophistiquée, qui nécessite une excellente compréhension des caractéristiques des données. Les notions d'entropie et de compression sont donc intimement liées car la mesure de l'entropie est souvent

réalisée en utilisant un modèle qui permet (justement) de comprimer l'information. Si ce modèle est adapté, la compression s'approche donc de l'entropie. Ce paradoxe est résolu facilement car la modélisation n'est qu'une étape qui précède le codeur d'entropie dans un programme de compression. La modélisation a pour but de créer un flux de données aux propriétés compatibles avec le codeur d'entropie. Le comportement de ce dernier étant souvent très bien connu, c'est la modélisation qui détermine l'efficacité de l'ensemble du programme.



:: Fig. 1 :: Structure fondamentale d'un programme de compression de données ::

On peut remarquer que la quantité brute de données augmente après la modélisation, parfois à cause de l'augmentation de la dynamique des nombres. Le codeur d'entropie va ensuite enlever les bits inutilisés et conduire (la plupart du temps) à une réduction effective du nombre total de bits nécessaires à représenter la donnée d'origine.

Enfin, il faut aussi savoir s'arrêter de complexifier un programme lorsque celui-ci donne un résultat satisfaisant alors que des efforts supplémentaires de programmation ne justifieraient pas le gain marginal obtenu. D'une part, cela aurait tendance à spécialiser l'algorithme qui ne pourrait plus traiter qu'un seul type de données, d'autre part, les contraintes de lourdeur de développement, de place mémoire et de temps d'exécution nous rappellent rapidement les limites pratiques du monde réel.

Mais qu'est-ce que l'information ?

La notion d'information, intuitivement comprise par les informaticiens, devient de plus en plus élusive à mesure qu'on essaie de la cerner. Prévoyez donc de l'aspirine et beaucoup de recherches documentaires si vous voulez y voir un peu plus clair mais je vais essayer de faire le tour du sujet sans vous noyer dedans.

D'abord, la définition de l'entropie de Shannon n'est que d'une utilité partielle. Elle ne s'applique pas universellement à toutes les données à cause des modélisations implicites, réalisées par exemple lors du découpage du flux d'entrée en symboles individuels. Un excellent exemple est fourni dans cet article, où des analyses d'une même donnée, en utilisant des symboles sur 8 et 16 bits, conduisent à des entropies différentes.

Il se trouve cependant que la définition de Shannon est la plus couramment utilisée, en particulier pour nos fichiers sagement découpés en octets, et elle est indispensable pour les questions de codage optimal de symboles dans des flux idéaux, mais elle ne résout pas tous les problèmes.

En considérant un flux de données binaires de longueur donnée, c'est-à-dire l'information la plus abstraite possible, on fait alors appel aux théories d'Andrei N. Kolmogorov (1903-1987) et nous entrons alors dans le domaine de la Théorie Algorithmique de l'Information. La « Complexité de Kolmogorov », en particulier, est définie comme l'entropie minimale absolue d'une chaîne particulière et est égale à la taille du programme le plus court pouvant générer la chaîne.

Calculer cette entropie revient à trouver et programmer le meilleur compresseur possible pour la chaîne d'entrée, ce qui nous fait retomber dans l'analyse des données. La différence notable avec l'entropie shannonienne est que dans ce cas le flux de sortie est un programme, exécuté par exemple au moyen d'une quelconque Machine de Turing universelle, avec éventuellement des données qu'il va interpréter d'une manière ou d'une autre.

La modélisation est toujours indispensable et comprend en plus un aspect purement informatique (les questions du langage et des connaissances pré-requises, assimilées à des canaux parallèles, sont très importantes), mais on n'est plus cantonné à traiter des flux dont le modèle statistique est statique.

Le programme peut simplement indiquer **recopier l'entrée vers la sortie** si aucune compression ne semble possible, ce qui oblige à allonger le flux d'un bit au minimum. Cette idée rejoint le fameux *counting argument* qui prouve qu'un compresseur universel (capable de réduire d'au moins un bit **toute** information) ne peut fonctionner.

A ce niveau, on se heurte à de plus en plus de problèmes théoriques et philosophiques, touchant aux fondations des mathématiques : la Théorie des Nombres. Pour comprimer un flux de nombres, il faut faire le tri entre les nombres importants et ceux qui ne le sont pas, pour réduire la place de ces derniers. Quand on ne peut découper le flux en nombres, on considère que ce flux est le nombre.

Or, mathématiquement parlant, tous les nombres ont la même importance ! Par exemple, Gregory J. Chaitin montre par l'absurde que s'il existait des nombres « remarquables » et d'autres non, le premier nombre « non remarquable » serait justement remarquable, ce qui serait un paradoxe évident. La « remarquabilité » vient de notre intérêt pour la propriété, pas du nombre : on retombe encore sur l'importance de la modélisation.

Dans *Scientific American*, G. Chaitin montre aussi dès 1975 que la plupart des nombres sont « aléatoires » au sens algorithmique, mais aussi qu'il est impossible de le prouver. Comment faire alors pour compresser des données ?

Si on ajoute les travaux des années 30 de Gödel sur l'incomplétude des mathématiques et de Turing sur l'incalculabilité, la situation devient de plus en plus complexe et insaisissable.

Mais qu'est-ce que l'information ? (suite)

Cependant, il existe des certitudes démontrées :

- Il n'existe pas de mesure absolue du contenu d'information, en particulier à cause de la dépendance envers la modélisation. Sans modélisation, on ne peut pas déterminer quel nombre est « important » ou non et on ne peut pas compresser. Si on ne peut pas compresser, on ne peut pas déterminer l'entropie et vice versa.
- La limite de codage de Shannon ne peut être brisée (dans son domaine d'application). On peut aussi prouver que les codeurs arithmétiques sont les meilleurs algorithmes possibles pour un modèle probabilistique donné.
- Il n'existe pas de compresseur universel, car il n'est pas possible de modéliser toutes les données possibles : le flux de sortie devrait alors contenir au moins le numéro du modèle choisi, ce qui reviendrait au moins à recopier l'entrée vers la sortie car le numéro du modèle serait de même taille que la donnée à coder. Il existe d'autres preuves formelles pour confirmer cette affirmation.
- Inversement, il n'est pas non plus possible de prouver qu'il n'existe pas de compresseur efficace pour coder une donnée (ou un type de donnée) particulière puisqu'on ne peut pas prouver qu'elle est aléatoire.

Les démonstrations et réflexions sur ce sujet sont disponibles sur les sites web des auteurs respectifs, indiqués en fin d'article.

Pour s'en sortir, il faut toujours prendre une définition arbitraire à un moment ou un autre et nous retrouvons alors les outils mathématiques classiques.

En fait, dans l'informatique de tous les jours, nous sommes dépendants de la dualité algorithme/structure de données qui nous permet de formaliser nos problèmes informatiques quotidiens.

Toutes nos données impliquent une structuration qui permet leur traitement, ce qui nous permet de créer des compresseurs adaptés et plus efficaces. Le ruban infini de la Machine de Turing est bien pratique pour les théoriciens mais ne correspond pas à une application réelle.

La conclusion de ce passage est que même s'il est impossible de comprimer infiniment des données, il sera toujours possible d'améliorer un algorithme de compression.

Le domaine d'expérimentation est encore plus vaste si on ajoute les innombrables contraintes de temps et de place ainsi que les compromis de plus en plus sophistiqués imposés par nos ordinateurs.

1.10 Taxonomie

Deux grandes familles d'algorithmes existent pour réduire la taille d'un ensemble de données, selon le type de celles-ci : les données constituées de symboles, appartenant à un alphabet par exemple, et celles constituées de grandeurs échantillonnées (ou leurs transformées).

La première famille est très connue et s'applique aux textes ou aux programmes informatiques par exemple. Les algorithmes les plus connus sont `LZ77`, `LZW`, `PPM`, `Huffman`, `Burrows-Wheeler`... Leurs principes s'appliquent aux données utilisant un alphabet fixe, dont chaque symbole a des fréquences et des probabilités d'occurrence mises à profit de manière plus ou moins complexe. **La valeur numérique associée à chaque symbole n'est pas considérée comme importante** et aucune perte n'est admise.

La deuxième famille s'applique aux signaux ou aux grandeurs, mesurées ou échantillonnées, comme les statistiques, les relevés géologiques, les sons ou les images. Ils se distinguent de la première famille par leurs propriétés statistiques : la distribution est généralement plus régulière et la dynamique des valeurs peut être beaucoup plus grande.

De plus, une certaine quantité d'erreur peut être admise (par exemple pour les images, avec JPEG notamment), mais nous nous intéresserons particulièrement aux algorithmes de compression sans perte (*lossless*, restituant l'original bit à bit) à une dimension.

Comme la différence entre ces deux familles d'algorithmes réside dans les pertes, il en résulte en pratique une différence par le type et le format des données. Les algorithmes sans perte utilisent en général uniquement des nombres entiers.

Par contre, les algorithmes acceptant les pertes fonctionnent souvent avec des nombres à virgule flottante ou alors « fractionnaires » (un nombre entier « normalisé » dans un intervalle tel que -1 à +1 par exemple) puisque la contrainte n'est plus l'exactitude, mais le taux d'erreur.

Il est donc facile de confondre des notions telles que la « résolution » (le nombre de bits utilisés pour représenter les nombres), la « précision » (le nombre de bits « utiles » dans l'échantillon) et la « définition » (terme trop flou dans ce contexte pour être utilisable). Les paramètres importants comme le niveau de bruit, le taux de distorsion et le volume d'enregistrement (la dynamique utilisée) interviennent à toutes les étapes pour réduire la qualité de l'échantillonnage.

Toutefois, les applications subissant de fortes contraintes d'efficacité et de vitesse utilisent souvent une version adaptée à base de nombres entiers, plus faciles à gérer. Par exemple, les algorithmes MPEG (son et vidéo) et JPEG sont définis dans une norme utilisant des nombres flottants. Mais les applications pratiques emploient des entiers avec juste le nombre de bits nécessaires pour que les erreurs d'arrondis introduisent un bruit inférieur à une limite spécifiée par la norme.

2 Propriétés statistiques du premier ordre

Nous allons étudier la compression de fichiers en commençant par examiner ceux-ci sous l'angle indiqué par Shannon : l'histogramme des données, permettant d'obtenir une première indication sur leur entropie.

En ce qui concerne les données de type « texte », il existe des fichiers de référence pour comparer les performances des compresseurs. Mais il y a plus simple pour commencer notre étude, par exemple : le fichier `linux-2.4.18.tar`, en l'occurrence l'archive des sources du noyau Linux de mon ordinateur. C'est un fichier très gros (131727360 octets) qui contient peu de données binaires et qui regorge de texte en clair dans la documentation et les commentaires du code source.

Comme les lecteurs le savent certainement, sa version comprimée (`linux-2.4.18.tar.bz2`) est disponible sous licence GPL sur Internet, sur **http://www.kernel.org** et ses miroirs.

On peut déjà calculer que l'entropie se situe environ à

taille décomprimée / (8 x taille comprimée avec BZIP2)
= 131727360 / (8 x 24161675)
= 1,467 bits par caractère

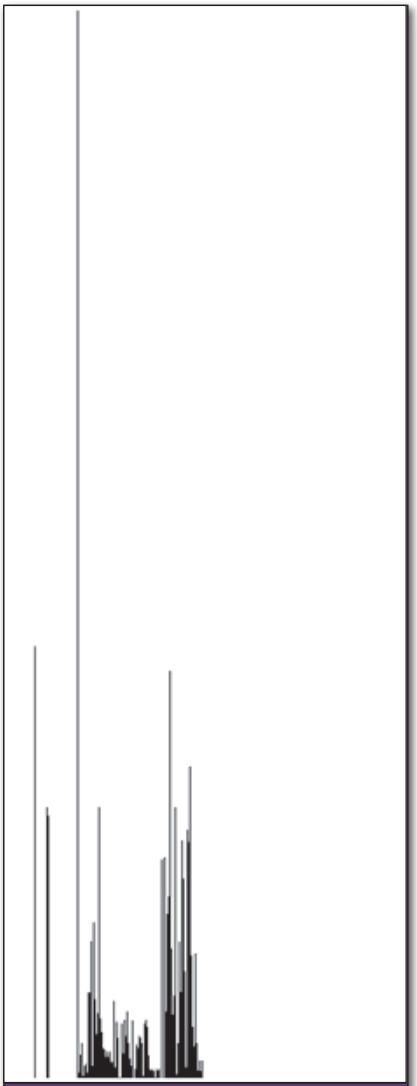
Le résultat est meilleur que 1,9 (typique d'un texte anglais) puisqu'une grande partie des données, en dehors de la documentation et des commentaires en anglais, consiste en du code C formaté de manière particulière.

Pour analyser les données, le petit programme `stats_o0.c` a été écrit.

```
/*
  stats_o0.c
  histogramme d'un fichier d'octets
      et calcul de l'entropie

  création : Mon Sep 15 05:23:51 CEST 2003
    par whygee@f-cpu.org

  compilation :
    gcc -W -Wall -lm -o stats_o0 stats_o0.c
*/

#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#define MAX_X 800 /* largeur de mon écran pour
 afficher avec un dump vers le framebuffer */

unsigned long int histogramme[256];

int main(int argc, char *argv[]) {
  FILE *file_in;
  FILE *file_out;
  unsigned long int taille=0, i, j, max=0, imax=0;
  unsigned long long int lli;
  double total_entropie=0.0, entropie, proba;

  if (argc==3) {
    file_in=fopen(argv[1], "rb");
    if (file_in!=NULL) {
      file_out=fopen(argv[2], "wb");
      if (file_out!=NULL) {

        /* collecte de l'histogramme */
        while ((i=fgetc(file_in))!=EOF) {
          histogramme[i]++;
          taille++;
        }

        /* recherche du maximum
           pour normaliser l'histogramme */
        for (i=0; i<256; i++) {
          if (histogramme[i]>max) {
            max=histogramme[i];
            imax=i;
          }

          /* calcul de l'entropie */
          if (histogramme[i] > 0) {
            proba=histogramme[i];
            proba/=taille;
            entropie=proba*-log(proba)/M_LN2;
            total_entropie+=entropie;
            printf("histogramme[%ld]=%ld,      entropie=%G\
n",
              i,histogramme[i],entropie);
          }
        }

        /* écriture du résultat */
        for (i=0; i<256; i++) {
          lli=histogramme[i];
          lli*=MAX_X;
          lli=lli/max;
          for (j=0; j<lli; j++)
            fputc(15, file_out);
          for (; j<MAX_X; j++)
            fputc(1, file_out);
        }
        printf("max : histogramme[%ld]=%ld\n",imax,max);
        printf("entropie = %G\n",total_entropie);
      }
    }
  }
  exit(EXIT_SUCCESS);
}
```

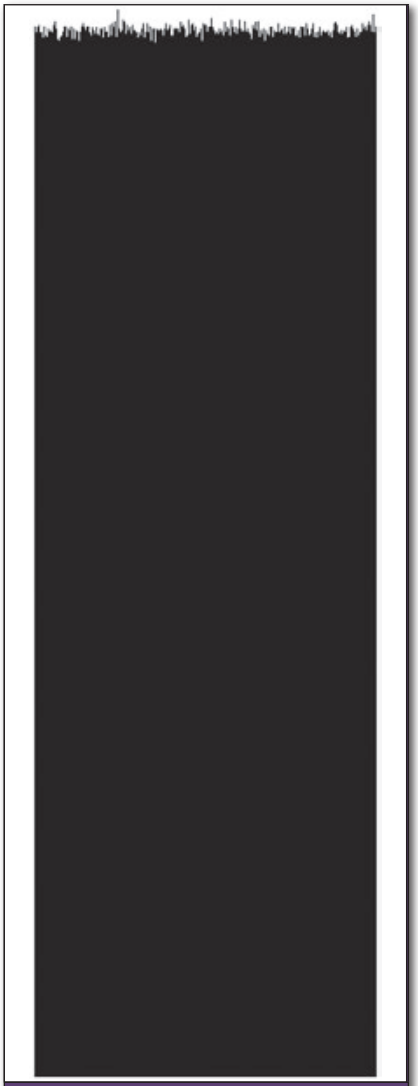
:: Programme 1 : stats_o0.c ::

Il génère un histogramme et collecte des informations comme la valeur maximale et l'entropie de chaque caractère, puis crée un fichier pouvant être affiché directement dans un *framebuffer* ou repris dans un logiciel de dessin :



:: Fig. 2a :: Histogramme du fichier linux-2.4.18.tar ::

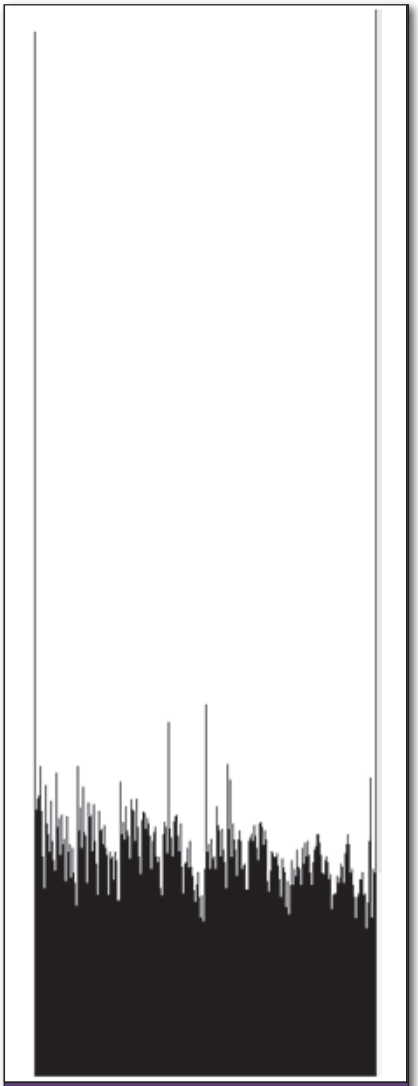
L'histogramme ci-dessus montre la distribution des caractères utilisés dans les fichiers source de Linux. Le caractère 32 (l'espace en code ASCII) est le plus représenté avec 17023221 occurrences, soit une espace toutes les 7,738 lettres en moyenne. Il y a aussi 4198625 caractères 10 (*Line Feed* en code ASCII ou `\n` en C), ce qui signifie qu'il y a plus de quatre millions de lignes. L'histogramme n'est pas homogène, avec des pics correspondant à certains caractères très courants, les caractères alphabétiques et de ponctuation sont en grande quantité et les autres sont très peu utilisés. Le code ASCII fait que l'histogramme ne regroupe pas les occurrences. C'est un très bon exemple d'alphabet sans relation forte d'ordre (l'ordre du symbole n'est pas en relation avec son importance ou son entropie). Ce programme calcule une entropie de 5.53253 bits par caractère, ce qui est beaucoup plus élevé que le gain obtenu avec `bzip2`. Ce dernier programme tient compte d'un contexte très large, alors que nous avons calculé l'entropie pour des caractères indépendants.



:: Fig. 2b :: 10MO en sortie de /dev/urandom entropie : 7.99998 ::

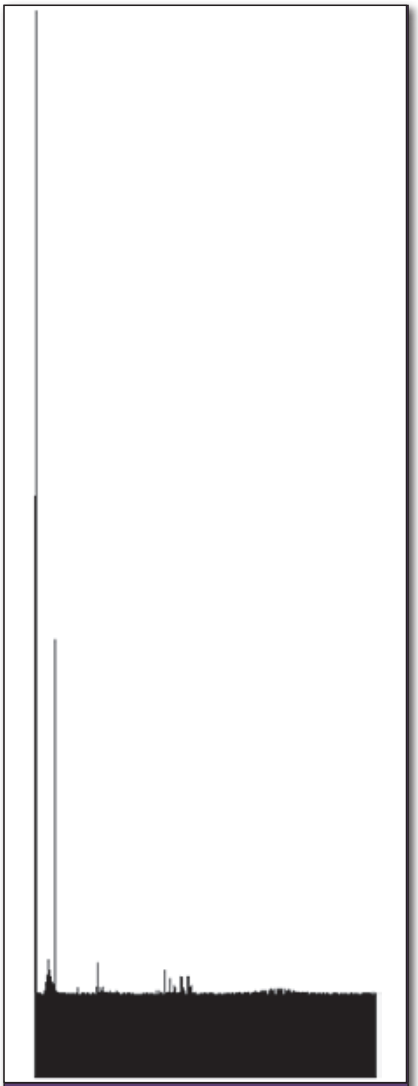
Pour comparer, le programme a été utilisé pour analyser d'autres types de données (fig 2b) :

L'histogramme est quasiment plat. L'entropie quasiment égale à 8 montre d'une part que l'algorithme de calcul d'entropie fonctionne et d'autre part que l'algorithme de génération de nombres pseudo-aléatoires de Linux est proche de l'idéal. En effet, à raison de 8 bits par symbole et de 256 symboles équiprobables, on retrouve bien une entropie de 8. La petite erreur (on ne trouve pas 8 mais une valeur inférieure très très proche) peut provenir (entre autres) de la quantité finie de nombres (un effet de bord est possible) et de la précision limitée des nombres flottants utilisés (rendant visible l'accumulation de 256 erreurs d'arrondis)(fig. 2c).



:: Fig. 2c :: fichiers .MP3 entropie : 7.94739 ::

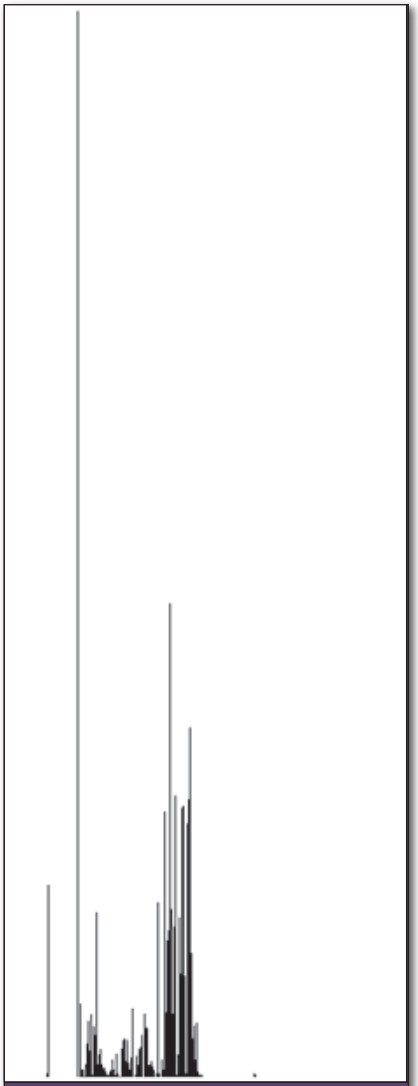
Les fichiers MPEG sont comprimés, ce qui explique l'entropie proche de 8. Les pics pour les valeurs 0 et 255 correspondent à des délimiteurs de blocs, qui permettent la recherche rapide dans les fichiers et qui évitent un échec total de lecture si un bloc est incomplet ou endommagé(fig. 2d).



:: Fig. 2d :: fichiers dans /usr/bin entropie : 7.84412 ::

Les fichiers binaires contiennent un peu de texte parmi les instructions, de même que des zones de remplissage pour aligner les données : ce sont les pics pour les valeurs 0 et 1.

Toutefois, `bzip2` n'arrive à réduire la taille de ces programmes que de 9%. Note importante pour ceux qui veulent reproduire cette expérience : le programme `strip` a préalablement enlevé les informations de débogage des programmes (fig. 2e).

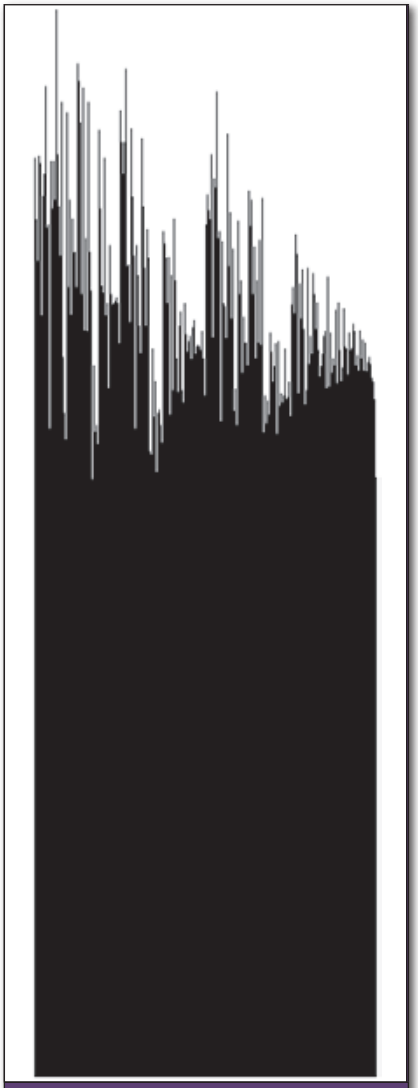


:: Fig. 2e :: 17MO de fichiers dans /usr/share/man/ entropie : 5.22434 ::

L'entropie est similaire à celle des sources du noyau Linux et le caractère le plus utilisé est l'espace : nous sommes bien en présence de texte (fig. 2f).

On remarque une variance plus grande que pour un fichier aléatoire, ce qui laisse penser que malgré sa puissance, le programme `bzip2` peut encore être amélioré. En effet, `bzip2` utilise l'algorithme de Huffman alors que son prédécesseur `bzip` utilisait un codeur arithmétique (plus performant mais couvert par des brevets dont ceux d'IBM)(fig. 2g).

Voici l'histogramme d'un fichier au format OGG Vorbis trouvé dans le CD-ROM du magazine (*Divergence Numérique* n°12). Je ne connais pas les paramètres utilisés pour le codage.



:: Fig. 2f :: linux-2.4.18.tar.bz2 entropie : 7.9903 ::

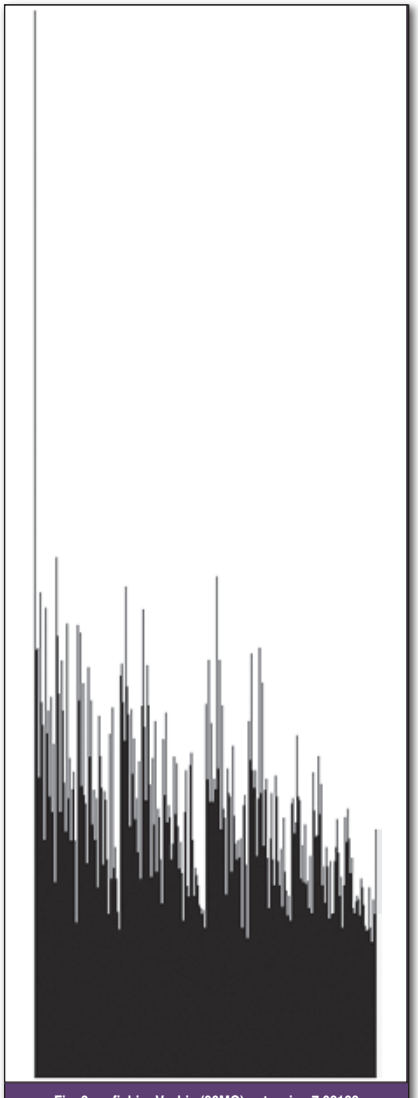
On retrouve le même type de figures géométriques, vaguement auto-similaires, que dans l'histogramme des fichiers .bz2. Ce format utilise des tables de Huffman prédéfinies, ce qui explique qu'il est sous-optimal. En revanche, une option de codage permet de définir une table spécifique mais elle augmente la taille du fichier (fig. 2h).

Le format FLAC utilise un codage d'entropie plus rudimentaire que Huffman, mais il est possible de l'optimiser bloc par bloc en effectuant une recherche plus ou moins exhaustive (selon l'option désirée) des paramètres optimaux.

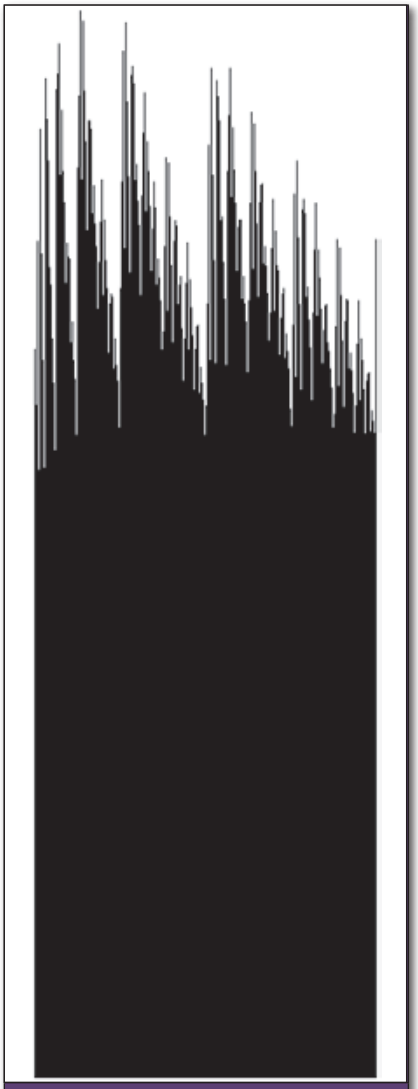
Comparée à un fichier Vorbis, l'entropie est plus grande, ce qui montre que le codage est plus efficace et que sa simplicité est compensée par une approche en « force brute ». On peut donc choisir un compromis entre vitesse de codage et taux de compression, bien qu'une recherche maximale ne donne pas un gain substantiel et utilise beaucoup plus de ressources de calcul.

Cela n'a pas d'influence sur la vitesse de décodage car le compresseur indique au décompresseur le paramètre qu'il a déterminé pour chaque bloc, il n'y a donc pas de recherche à faire.

On peut remarquer que trois formats phares du monde Open Source (bzip2, OGG Vorbis et FLAC) utilisent des codages sous-optimaux, surtout quand on observe l'apparente disparité entre les formats MP3 (H=7.94739) et OGG Vorbis (H=7.29174, pourtant souvent jugés de meilleure qualité à l'écoute).



:: Fig. 2g :: fichier Vorbis (90MO) entropie : 7.92182 ::

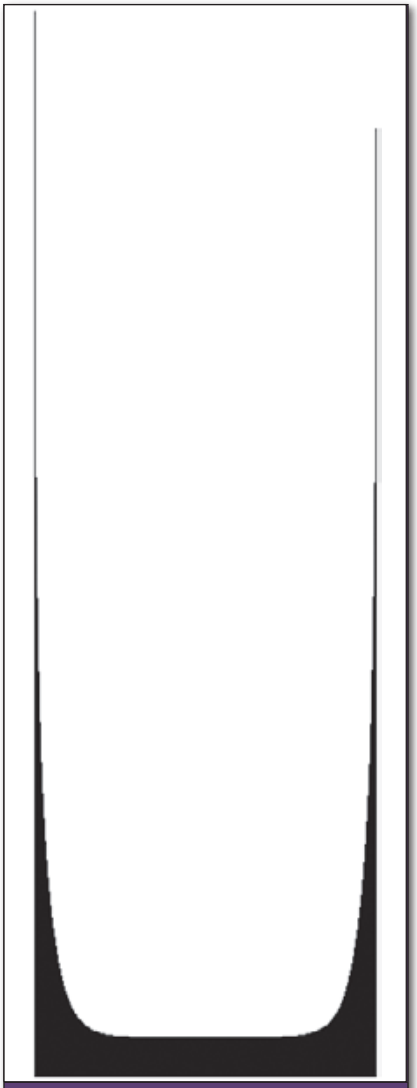


:: Fig. 2h :: tetouatator.flac (330MO) entropie : 7.98924 ::

La pression des brevets n'explique pas tout : il existe un équivalent libre aux codes arithmétiques appelé *range coding* qui est très légèrement moins efficace et deux fois plus rapide, mais d'autres préoccupations entrent en jeu comme la portabilité des algorithmes ou la vitesse totale de codage.

La capacité à supporter les erreurs de stockage ou de transmission, sur un ou plusieurs bits, est aussi importante, et les codes simples comme Huffman ou Rice ont des propriétés de confinement que ne peuvent garantir les codeurs arithmétiques. Au bout d'un temps variable, le décodeur va se resynchroniser naturellement sur la frontière des symboles, alors qu'un flux comprimé de manière optimale sera irrécupérable.

Enfin, il ne faut pas oublier que nous n'avons observé que le flux de sortie et non toutes les étapes de modélisations, qui sont loin d'être triviales (fig. 2i).



:: Fig. 2i :: tetouatator.raw entropie : 7.29174 ::

Voici l'analyse la plus intéressante : on voit clairement que la distribution ressemble à une double courbe exponentielle bien lisse.

La partie constante qui s'ajoute à cette courbe correspond à une mauvaise utilisation du format d'échantillons : un octet sur deux est presque aléatoire dans ce contexte.

Il faudra donc modifier le programme pour gérer les entiers signés sur 16 bits.

L'entropie de 7.29174 bits/octets indique qu'il est possible de comprimer ce fichier. Toutefois cette entropie n'est pas une limite réelle car `bzip2` peut comprimer ce fichier beaucoup plus :

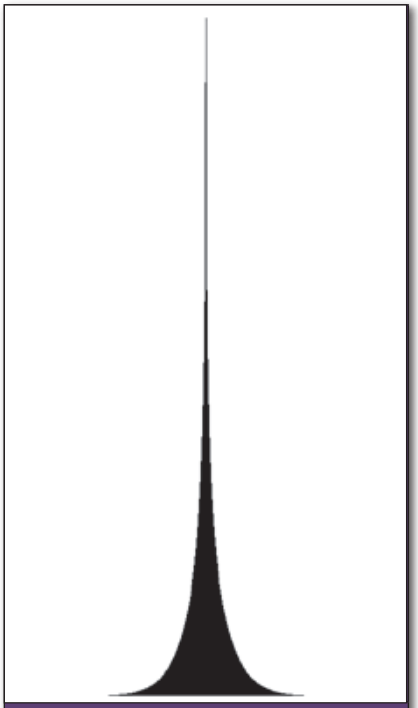
```
$ time bzip2 -k tetouatator.raw
real    19m34.691s
user    19m14.300s
sys     0m9.870s
$ ls -al
618136176 tetouatator.raw
481849907 tetouatator.raw.bz2
```

Avec 6,236 bits par octet, `bzip2` réduit d'environ 28% la taille du fichier original. Il est possible de compresser encore plus puisque `bzip2` travaille sur des octets et ne tire pas de bénéfice du codage sur 16 bits. L'algorithme utilisé par FLAC réduit la taille de `tetouatator.raw` à environ 330MO, soit presque 4,2 bits par octet (cela varie un peu selon les paramètres d'optimisation).

```
/*
  stats_o0-s16.c
  histogramme d'un fichier d'entiers signés
          et calcul de l'entropie

  création : Mon Sep 15 05:23:51 CEST 2003
    par whygee@f-cpu.org

  compilation :
    gcc -W -Wall -lm -o stats_o0-s16 stats_o0-s16.c
*/

#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#define MAX_X 800

unsigned long int histogramme[512];

int main(int argc, char *argv[]) {
  FILE *file_in;
  FILE *file_out;
  signed short int s16;
  unsigned long int taille=0, i, j, max=0, imax=0;
  unsigned long long int lli;
  double total_entropie=0.0, entropie, proba;

  if (argc==3) {
    file_in=fopen(argv[1], "rb");
    if (file_in!=NULL) {
      file_out=fopen(argv[2], "wb");
      if (file_out!=NULL) {

        /* collecte de l'histogramme */
        while (fread(&s16,1,2,file_in) == 2) {
          histogramme[(s16>>7)+256]++;
          taille++;
        }

        if (taille > 0) {
          /* recherche du maximum */
          for (i=0; i<512; i++) {
            if (histogramme[i]>max) {
              max=histogramme[i];
              imax=i;
            }

            /* calcul de l'entropie */
            if (histogramme[i] > 0) {
              proba=histogramme[i];
              proba/=taille;
              entropie=proba*-log(proba)/M_LN2;
              total_entropie+=entropie;
              printf("histogramme[%ld]=%ld,      entropie=%G\
n",
                i,histogramme[i],entropie);
            }
          }
          /* écriture du résultat */
          for (i=0; i<512; i++) {
            lli=histogramme[i];
            lli*=MAX_X;
            lli=lli/max;
            for (j=0; j<lli; j++)
              fputc(15, file_out);
            for (; j<MAX_X; j++)
              fputc(1, file_out);
          }
          printf("max : histogramme[%ld]=%ld\n",imax,max);
          printf("entropie = %G\n",total_entropie);
        }
      }
    }
  }
  exit(EXIT_SUCCESS);
}
```

:: Programme 2 : stats_o0-s16.c ::

Le programme de collecte de statistique en version 16 bits (`stats_o0-s16.c`) tient compte de la représentation en complément à deux et donne le résultat suivant (fig. 3).



:: Fig. 3 :: Histogramme avec des nombres de 16 bits entropie : 6.52667 ::

Le résultat est une magnifique courbe exponentielle symétrique. L'entropie a beaucoup diminué : elle n'est plus de 7.29174 bits par octet mais 6.52667 bits par échantillon.

D'après ce résultat, on pourrait réduire la taille du fichier original d'environ 60% !

2.2 Une petite erreur...

Seulement, ce n'est pas si simple et ce résultat parait bien optimiste. Le même programme a été lancé avec de nouveaux paramètres de taille, les résultats sont présentés dans le tableau suivant :

Résolution	H
1024	7,52135
512	6,52667
256	5,53453

La progression est évidente et l'erreur de programmation apparaît. Pour chaque doublement du nombre de points d'accumulation, l'entropie augmente d'un bit. En clair, la méthode utilisée (pour des raisons d'affichage) tronque les bits de poids faible des échantillons, réduisant artificiellement l'entropie.

On peut évaluer l'entropie réelle si on compte les bits qui ont été supprimés. Dans le cas de 1024 points, 6 bits manquent, il faut donc additionner 6 au résultat, ce qui donne H = 13,52135. En mesurant sur 65536 points (sans enlever de bits), on obtient H = 13,503 (6,7515 bits par octet), ce qui confirme la validité du calcul.

C'est déjà moins encourageant, on peut espérer gagner dans l'absolu au mieux 2,5 bits par échantillon ou 1,25 bit par octet. Il y a cependant un gain de 0,54 bit par octet par rapport à l'algorithme sur 8 bits, mais il reste autant à gagner pour égaler le résultat de `bzip2`.

3 Comptons les bits

Faisons un petit détour pour relier la notion d'entropie à d'autres notions plus intuitives pour les informaticiens. Les puristes vont sursauter en remarquant mes raccourcis « cavaliers » entre des techniques qui n'ont rien à voir entre elles, mais considérons cela comme une expérience particulièrement curieuse.

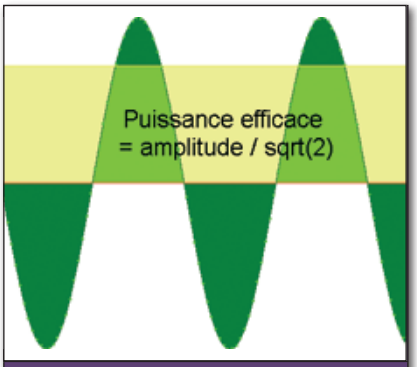
Dans les exemples d'analyses de textes au format ASCII, le code 32 (représentant l'espace) a une faible entropie et une très grande probabilité, mais cette probabilité n'est pas en relation avec la valeur du code.

Il n'y a pas de règle simple pour établir la probabilité du code 31 ou 33 en fonction du code 32 ou plus généralement, la probabilité d'un code `x` en fonction du code `x+1` ou `x-1`. Dans le cas de la table de code ASCII, c'est une convention arbitraire qui n'a pas été créée pour être compacte ou compactable facilement.

Au contraire, les données que nous étudions ont la propriété d'avoir une valeur (ou du moins, une valeur absolue) approximativement inversement proportionnelle à leur probabilité.

Le nombre de bits étant le logarithme de la valeur, on peut donc **estimer que la « quantité d'information » est proportionnelle au nombre de bits utilisés.** La double courbe exponentielle de l'histogramme précédent illustre cela clairement. Nous avons vu dans la première partie, lors de l'analyse du fichier `tetouatator.raw`, que la totalité de la dynamique n'est pas exploitée tout le temps, ce qui laisse donc des MSB (bits de poids fort de l'échantillon) inutilisés.

Encore d'autres MSB sont inutilisés car le signal est alternatif et sa forme est souvent proche du sinus. En partant de cette hypothèse, on peut déterminer que l'énergie du signal est égale à son amplitude crête à crête divisée par la racine carrée de 2. C'est une des formules classiques d'électricité qui permet de déduire la puissance efficace (ou « RMS », *Root Mean Square*) d'un courant alternatif (sinus pur).



:: Fig. 4 :: Rapport entre l'amplitude crête à crête et la puissance efficace (et l'entropie) d'un signal sinusoïdal ::

Dans une certaine mesure, nous pouvons appliquer cette idée à des signaux complexes en faisant appel au théorème de composition de Fourier, qui montre que tout signal cyclique peut être décomposé en une somme infinie de sinus et de cosinus.

Nous n'avons pas ici de signaux cycliques infinis mais l'idée tient la route dans certaines conditions, comme par exemple dans la Transformée Discrète de Fourier.

Imaginons qu'une partie du fichier contient un signal sinusoïdal d'une amplitude de 10 bits, on peut en estimer son énergie moyenne à `10/sqrt(2)=7,071 bits` par échantillon.

Si on applique cette idée à notre fichier dont l'entropie a été mesurée à 13,503 bits par échantillon, on trouve une amplitude maximale théorique de `13,503*sqrt(2)=19,096 bits` par échantillon. C'est 3 bits de plus que la dynamique permise par le format.

Heureusement, la musique n'est pas une sinusoïde pure, elle contient de nombreuses fréquences combinées, comme le montrent les spectrogrammes. L'analyse est donc valable localement, à l'échelle d'une ou plusieurs périodes du signal.

En pratique, compter des bits semble très simple mais comment les compter correctement ? Pour des entiers positifs, l'algorithme est très simple : on décale le nombre à droite tant qu'il reste non nul.

Ceci est la version en C, certains processeurs ont des instructions dédiées pour réaliser cette fonction avec des unités spécialisées

```
while (sample != 0) {
  sample >>= 1;
  compteur++;
}
```

Mais l'échantillon est signé, et la boucle ne se terminerait pas si un échantillon négatif devait être examiné (le bit de signe à 1 remplirait tout le mot).

Évidemment, on peut « corriger » cela :

```
if (sample < 0)
  sample = -sample;
```

A son tour, la négation peut avoir un effet pervers qu'il faut éviter. La représentation en complément à deux définit un intervalle entre -32768 à +32767. En « repliant » -32768, on obtient +32768 qui ne peut pas être représenté dans le format de l'échantillon. Une solution consiste à transtyper (*cast*) la variable car +32768 tient dans un `unsigned short int`. J'ai choisi d'utiliser une variable temporaire pour rendre le code le plus clair possible.

```
if (sample < 0)
  positif = -sample;
else
  positif = sample;
```

Mais en « repliant » les nombres négatifs vers les nombres positifs, on perd un bit, un peu comme ce qui s'est passé précédemment avec le calcul d'entropie erroné. Logiquement, il faut ajouter un bit par échantillon. J'ai « factorisé » cela en ajoutant le nombre d'échantillons à la fin.

Le programme nb_bits.c reste tout de même assez concis.

```
/*
   nb_bits.c
   création :
     Mon Jun 28 05:40:54 CEST 2004
     par whygee@f-cpu.org

   compilation :
     gcc -o nb_bits nb_bits.c

   invocation :
     nb_bits nom_du_fichier.raw
 */

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  FILE *file_in;
  unsigned long long int nb_bits=0;
  unsigned long int taille=0, positif;
  double moyenne;
  signed short int sample;

  file_in=fopen(argv[1], "rb");
  if (file_in!=NULL) {
    while (
      fread(&sample, 1, sizeof(sample), file_in)
        == sizeof(sample))
    {
      taille++;

      if (sample < 0)
        positif = -sample;
      else
        positif = sample;

      while (positif != 0) {
        positif >>= 1;
        nb_bits++;
      }
    }
    if (taille > 0) {
      moyenne=nb_bits+taille;
      moyenne/=taille;
      printf("%Ld bits (%G bits/échantillon)\n",
        nb_bits,moyenne);
    }
  }
  exit(EXIT_SUCCESS);
}
```

:: Programme 3 : nb_bits.c ::

Le résultat est très intéressant.

```
~/lm-cpr$ ./nb_bits tetouatator
3227511965 bits (11.4427 bits/échantillon)
```

ou 5,72 bits par octet. Une valeur de 6,72 (proche de 6,7515 déterminé par le calcul d'entropie) serait plus logique.

Le nouveau piège est que le programme considère maintenant tous les échantillons comme puissances de deux. Toute valeur est arrondie par défaut : par exemple le nombre 3127 ou 110000110111, n'est plus traité que comme 100000000000 ou 2048.

Les valeurs réelles sont donc **discrétisées** car seule la position du MSB compte. Comment doit-on compter les bits qui ont été mis à zéro ? Doit-on arrondir à l'excès ou à défaut ? Cette question va revenir souvent et la réponse dépend de l'algorithme mis en œuvre.

Conclusion : il existe une relation entre le nombre de bits consommés par un signal, son entropie et son énergie. Ici, la nature du signal, composé de sinusoïdes, permet ce rapprochement risqué mais bien pratique. Mais n'allez pas le répéter à votre professeur de physique ou de mathématiques.

4 Propriétés contextuelles : analyse du deuxième ordre

Les premières analyses nous ont permis de connaître la probabilité d'occurrence d'une valeur dans le flux de données. Cependant, ce sont des informations indépendantes de la position, ne tenant pas compte des informations alentours, sans contexte.

Cette fois-ci, l'analyse tient compte du caractère précédent. Il est possible de tenir compte de plus de caractères au prix d'un espace mémoire qui peut devenir gigantesque en utilisant des espaces à N dimensions pour un contexte de N caractères. En traitant des octets, un contexte de 3 caractères occupe déjà seize millions de mots. Un contexte plus grand dépasse la capacité d'adressage d'un processeur 32 bits.

Cette profondeur d'analyse statistique est exprimée par un « ordre », celle du chapitre précédent étant d'ordre zéro puisque aucun caractère voisin n'était pris en compte. Ce chapitre effectue une analyse d'ordre 1 avec le programme stats_o1.c

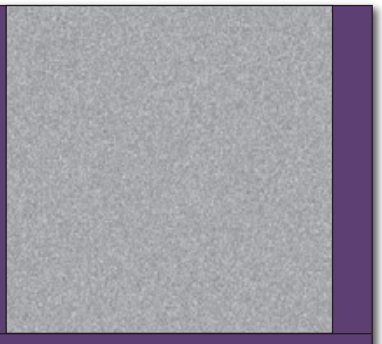
```
/*
         stats_o1.c
  histogramme du premier ordre d'un fichier
  d'octets et calcul de l'entropie

  création : Mon Sep 15 05:23:51 CEST 2003
    par whygee@f-cpu.org
  version : Sat Jun 26 07:02:04 CEST 2004

  compilation :
    gcc -W -Wall -lm -o stats_o1 stats_o1.c
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

unsigned long int histo[256][256];
```

```
int main(int argc, char *argv[]) {
  FILE *file_in;
  FILE *file_out;
  unsigned long int i, j, k, max, min,
     last=0, taille=0;
  unsigned long long int lli;
  double h=0.0, entropie, proba;

  file_in=fopen(argv[1], "rb");
  if (file_in!=NULL) {
    file_out=fopen("histo1", "wb");
    if (file_out!=NULL) {
      /* creation de l'histogramme en 2D */
      while ((i=fgetc(file_in))!=EOF) {
        histo[last][i]++;
        last=i;
        taille++;
      }

      min=max=histo[0][0];
      for (i=0; i<256; i++)
        for (j=0; j<256; j++) {
          k=histo[i][j];
          /* recherche du maximum et du minimum */
          if (k>max)
            max=k;
          if (k<min)
            min=k;
          /* calcul de l'entropie */
          if (k>0) {
            proba=k;
            proba/=taille;
            entropie=proba*-log(proba)/M_LN2;
            h+=entropie;
          }
      }

      printf("max = %ld\n",max);
      printf("min = %ld\n",min);
      printf("entropie = %G\n",h);

      /* mise à l'echelle et écriture */
      for (i=0; i<256; i++)
        for (j=0; j<256; j++) {
          lli=histo[i][j];
          lli*=255;
          lli=lli/max;
          fputc(lli, file_out);
        }
    }
  }
  exit(EXIT_SUCCESS);
}
```

:: Programme 4 : stats_o1.c ::

Les images d'histogrammes suivantes ont toutes été manipulées pour faire ressortir un contraste suffisant et montrer les structures (ou leur absence).

Pour « calibrer » le programme avec un flux aléatoire, voici l'analyse de dix millions d'octets générés par /dev/urandom :



:: Fig. 5a :: /dev/urandom
max = 35, min = 2, H = 15.9527 (7,97635) ::

Entropie à 15,95 ? Pourquoi pas 8 ? Parce que chaque point de l'histogramme correspond à deux échantillons et chaque échantillon sert à calculer deux points. Il faut donc diviser l'entropie par deux et on obtient presque le 8 attendu.

On note aussi que la qualité des nombres générés par `/dev/urandom` est inférieure à celle de `/dev/random` car ce dernier fait très attention à sa propre entropie et il se sert des événements du système pour se rafraîchir. A partir d'un certain point, si des données indépendantes du générateur n'arrivent pas, tous les nombres générés par `/dev/urandom` deviennent dépendants des précédents, ce qui réduit le caractère « aléatoire » du système.

Par contre, `/dev/urandom` n'attend pas d'événement externe (actions sur les touches du clavier, paquets sur le réseau...) quand il détecte que l'entropie baisse, ce qui permet de générer des nombres pseudo-aléatoires à très grande vitesse.

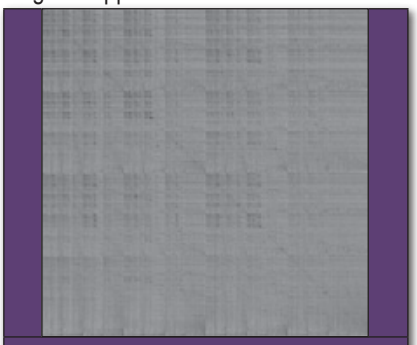
Ce n'est pas adapté à un usage scientifique ou cryptographique mais une entropie de 7,976 suffit souvent pour les autres usages et c'est meilleur qu'une archive compressée avec `bzip2`.

Voici l'analyse d'ordre 1 du fichier `linux-2.4.18.tar` :

:: Fig. 5b :: linux-2.4.18.tar max = 6769812, min = 0, H = 9.43783 (4,718915) ::

Le programme indique que la paire de caractères « espace » se reproduit 6769812 fois. Il faut ensuite une échelle logarithmique pour distinguer graphiquement les autres paires représentées par des points.

Il apparaît que ce fichier a des caractéristiques spécifiques permettant de le comprimer : non seulement certains caractères apparaissent beaucoup plus que d'autres, mais ils apparaissent souvent successivement. La modélisation du premier ordre permet déjà de réduire l'entropie d'environ 0,8 bit par octet (10%). On voit aussi apparaître une symétrie diagonale, expliquée par le fait que chaque donnée est utilisée deux fois : d'abord pour la colonne puis la ligne. La donnée gardant sa valeur lors du changement d'axe, un effet de miroir diagonal apparaît.



:: Fig. 5c :: linux-2.4.18.tar.bz2 max = 7719, min = 89, H = 15,9512 (7,9756) ::

Voici l'analyse du fichier comprimé. Il devient plus évident qu'il y a peut-être encore une occasion de comprimer un tout petit peu car des motifs géométriques et des points apparaissent assez clairement. Cependant, l'entropie très proche de 8 montre que l'ordre 1 n'est pas une bonne approche.

:: Fig. 5d :: divnum12.ogg max = 285096 min = 344 entropie = 15.7287 (7.86435) ::

:: Fig. 5e :: tetouatator.flac max = 287199 min = 1969 entropie = 15.9655 (7.98275) ::

Encore une fois, le codage avec FLAC semble plus efficace que celui d'OGG Vorbis.

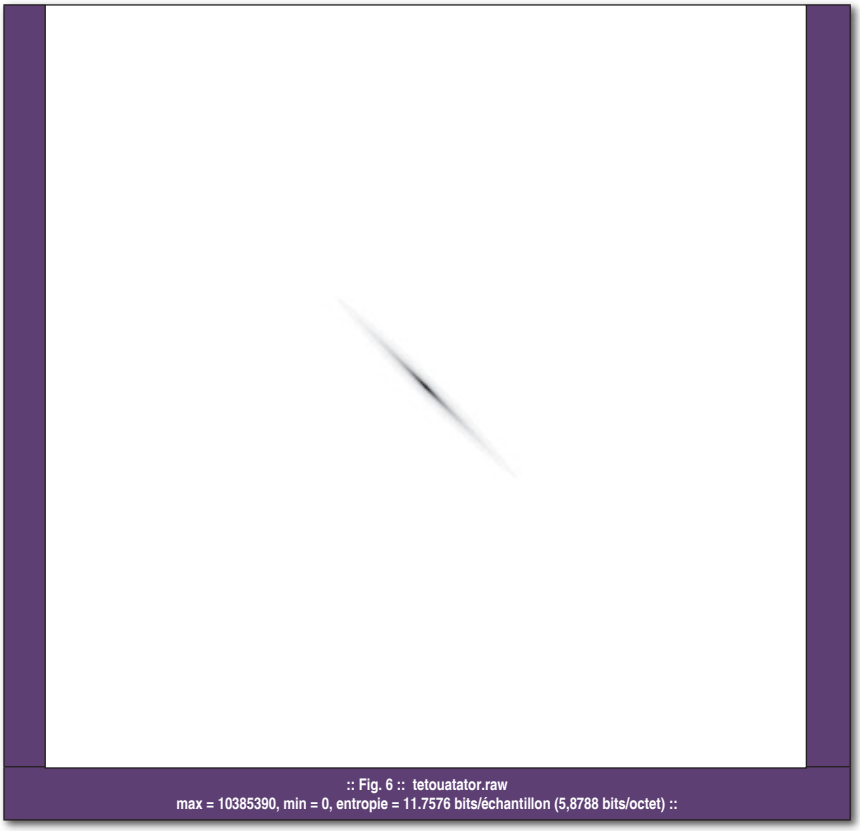
Enfin, pour analyser notre fichier son, il faut aussi une version spéciale à plus d'un titre : d'une part pour traiter les entiers signés sur 16 bits et d'autre part pour désentrelacer les deux canaux.

Cette fois-ci, il faut en plus calculer correctement l'entropie en tenant compte des facteurs identifiés :

- Il y a d'abord la mise à l'échelle d'un facteur 2 car deux échantillons servent pour chaque point
- Un autre facteur 2 car les canaux gauche et droite sont accumulés sur le même histogramme
- Il faut enfin rajouter 7 bits en raison de la suppression de 7 LSB sur chaque échantillon.

```
/*
         stats_o1-s16.c
  histogramme du premier ordre d'un fichier
  d'échantillons stéréo et calcul de l'entropie

  création : Mon Jun 28 08:13:29 CEST 2004
    par whygee@f-cpu.org

  compilation :
    gcc -lm -o stats_o1-s16 stats_o1-s16.c

  invocation :
    ./stats_o1-s16 nom_fichier.raw sortie_image.raw
 */

#include
#include
#include

unsigned long int histo[512][512];

int main(int argc, char *argv[]) {
  FILE *file_in, *file_out;
  signed short int droite, gauche,
    last_droite=256, last_gauche=256;
  unsigned long long int lli, taille=0;
  unsigned long int i, j, k, max, min;
  double h=0.0, entropie, proba;

  file_in=fopen(argv[1], "rb");
  if (file_in!=NULL) {
    file_out=fopen("histo1-s16", "wb");
    if (file_out!=NULL) {
      /* creation de l'histogramme en 2D */
      while (
        ( fread(&gauche, 1, sizeof(gauche), file_in)
        + fread(&droite, 1, sizeof(droite), file_in))
             == (sizeof(gauche) * 2 )) {
        taille++;
        droite=(droite >> 7)+256;
        gauche=(gauche >> 7)+256;
        histo[last_droite][droite]++;
        histo[last_gauche][gauche]++;
        last_droite=droite;
        last_gauche=gauche;
    }

    min=max=histo[0][0];
    for (i=0; imax)
          max=k;
        if (k
```

:: Programme 5 : stats_o1-s16.c ::



:: Fig. 6 :: tetouator.raw
max = 10385390, min = 0, entropie = 11.7576 bits/échantillon (5,8788 bits/octet) ::

Comme pour l'image du diagramme XY du premier article, j'ai dû forcer sur la correction de contraste pour faire apparaître quelque chose.

L'échelle logarithmique est encore plus raide et il n'y a pas grand-chose à montrer, à part une petite tache diagonale au centre.

En exploitant une simple corrélation temporelle et en utilisant le format approprié, on descend en dessous de 6 bits par octet, ce qui dépasse le résultat de `bzip2`.

Est-il possible de faire encore mieux ? Oui car FLAC descend à 4,2 bits par octet. Comment ? C'est encore toute une histoire :-)

Conclusion

Cet article a montré que certains types de fichiers pouvaient être comprimés et dans quelle mesure, grâce à un outil simple et puissant : le calcul d'entropie à partir d'un histogramme. A cette occasion, j'ai dû utiliser une formule mathématique, j'espère que le logarithme ne vous a pas effrayé :-)

Pourtant, on ne sait pas encore **comment** effectuer la compression, bien que des détails commencent à se profiler : en quelques petits chapitres, nous avons survolé quatre techniques de codage et leurs caractéristiques respectives.

Il va falloir être patient car les articles suivants vont lever un coin du voile après l'autre, mais il faut bien commencer par le début.

Merci encore à tous les relecteurs enthousiastes dont les remarques constructives ont sensiblement amélioré la qualité de cette série d'articles : les woof/moules, Steven Pigeon... Les autres se reconnaîtront !

Enfin, je dédie cet article à Tristan et sa maman qui font de moi un oncle comblé. Si au moins j'avais le temps de m'en occuper !

Références

- Le miroir des fichiers source sont aussi disponibles ici : http://ygdes.com/lm-cpr/index_src.htm
- Le fameux article de Claude E. Shannon « *A Mathematical Theory of Communication* », *The Bell System Technical* J. **27**, 379-423 & 623-656, July and Oct. 1948, http://cm.bell-labs.com/cm/ms/what/shannonday/shannon1948.pdf
- La page personnelle de Gregory J. Chaitin, avec de nombreux textes (en anglais et en français) sur la Théorie Algorithmique de l'Information : http://www.cs.auckland.ac.nz/CDMTCS/chaitin/
- Texte d'une conférence récente (en anglais) de Gregory J. Chaitin, traitant de la plupart des points théoriques soulevés ici : http://www.cs.auckland.ac.nz/CDMTCS/chaitin/summer.html
- Article de Gregory J. Chaitin sur les relations entre les preuves mathématiques et les nombres dits « aléatoires » : « *Randomness and Mathematical Proof* », *Scientific American* **232**, No. 5 (*May* 1975), pp. 47-52, http://www.cs.auckland.ac.nz/CDMTCS/chaitin/sciamer.html
- Une petite FAQ (anglaise) qui remet à plat les idées sur la Théorie de l'Information : http://www.geocities.com/CollegePark/9315/infofaq.htm
- L'incontournable site de Mark Nelson : http://www.datacompression.info
- Le site de Robin White (en anglais) fourmille d'informations utiles : « *First Principles* » : *Lossless Compression of Audio* http://www.firstpr.com.au/audiocomp/lossless/index.html