Décodage d'images numériques issues de satellites météorologiques en orbite basse : le protocole LRPT de Meteor-M2

J.-M Friedt, 9 mars 2019

Nous avons attaqué le problème du décodage des images transmises par protocole LRPT de Meteor-M2. Tout comme l'utilisateur d'une browser web qui désire afficher une image transmise par HTML avec pour seul outil une sonde d'oscilloscope branchée sur un câble Ethernet, nous devons remonter petit à petit les couches OSI pour passer de la couche matérielle à la couche applicative. L'étude précédente nous avait permis de retrouver, à partir des mesures de phases, la constellation QPSK et des bits déconvolués par algorithme de Viterbi. Nous avions échoué à trouver le mot de synchronisation des trames dans cette séquence de bits. Poursuivons donc l'exploration pour aller jusqu'au décodage des images JPEG dont on affichera le contenu pour avoir une vue de l'espace.

1 Rotation de la constellation

L'absence de corrélation observée à l'issue de l'étude précédente indique que nous ne comprenons pas comment les bits sont encodés dans le message acquis, sous réseve que le mot de synchronisation encodé par convolution soit correct, hypothèse que nous ferons compte tenu des informations fournies sur [2]. Lorsque nous avions étudié la modulation en phase binaire (BPSK – *Binary Phase Shift Keying*), nous avions vu que deux cas pouvaient se produire [3, 4] : soit le signal acquis était en phase avec l'oscillateur local, et les bits issus de la comparaison de la phase entre le signal reçu et la reproduction locale de la porteuse (boucle de Costas) étaient ceux attendus, soit le signal était en opposition de phase et le bits étaient inversés par rapport à la valeur attendue. Dans les deux cas, la corrélation avec le mot de synchronisation accumule de l'énergie le long de la séquence de bits et se traduit, en prenant la valeur absolue, par un pic de corrélation, que nous ayons la bonne séquence dans la trame acquise ou son opposé. Ce cas simple se complique dans le cas de QPSK (*Quadrature Phase Shift Keying*) dans lequel la phase prend quatre états possibles qui encodent chacun une paire de bits (Fig. 1). L'assignation entre la valeur de la phase et la paire de bits correspondante n'est pas évidente, mais surtout toute erreur dans l'assignation symbole-pair de bits se traduit par une séquence erronée qui ne corrèle pas avec le mot recherché. À titre d'exemple, si nous attribuons 0° à la paire de bits 10 et 90° à 11 (code de Gray dans lequel seul 1 bit change entre deux valeurs adjacentes de la phase), alors une séquence 0-90° se traduit par 1011 alors que l'assignation 0° à 11 et 90° à 01 interpète la même séquence de phases par 1101 : les deux messages issus de la même séquence de phases mesurées sont complètement différentes et n'ont aucune chance d'accumuler l'énergie nécessaire au pic de corrélation lors de la comparaison avec le motif de synchronisation de référence. On notera, en comparant la Fig. 1 à la dernière figure de la première partie de cet article, que nous avions placé la paire de bits 00 sur la mauvaise valeur de phase (voir Fig. 7 de la section 5 de l'article précédent pour l'assignation entre chaque symbole et chaque paire de bits dans le *Constellation Soft Decoder* – `digital.constellation_calcdist(([-1-1j, -1+1j, 1+1j, 1-1j]), ([0, 1, 3, 2]), 4, 1).base()`, et l'analyse que nous proposons ici en Fig. 2). Par ailleurs, le code de Gray peut évoluer dans le sens horaire ou trigonométrique, induisant encore un risque d'erreur. Il ne semble pas y avoir de façon d'identifier l'assignation phase-paire de bits autre que la force brute dans laquelle toutes les combinaisons de codes possibles sont testées, tel que nous l'enseigne le contenu de `m.c.patts[][]` de `medet.dpr` dans `meteor_decoder` :

The mapping onto the QPSK constellation shall be according the Gray encoding

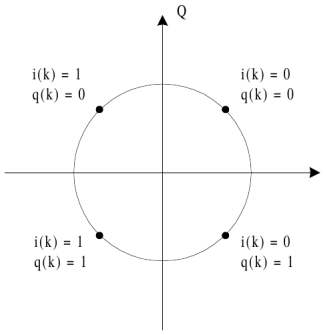

Figure 6.4 - QPSK constellation diagram

Figure 1: Constellation QPSK : chaque état possible de phase encode 2 bits. L'assignation de chaque symbole vers les paires de bits seront l'objet d'une partie du travail de décodage. Cette figures est issue de [1, Fig. 6.4,p.16].

```
11111100101000101011011000111101101100000000110110010111100101000
01010110111110111101001110010100110110101010010011000001110000010
00000011010111010100100111000010010011111111001001101000011010111
10101001000001000010110001101011001001010101101100111110001111101
11111100010100010111100100111110011100000000111001101011011010000
01010110000010000001110010010111000110101010011100111101001111100
00000011101011101000011011000001100011111111000110010100100101111
10101001111101111110001101101000111001010101100011000010110000001
```

donne toutes les combinaisons possibles de paires de bits dans le mot encodé par convolution, et la corrélation du message acquis se fait avec toutes les déclinaisons de ce code.

Cependant, nous avons encore un problème : ces inversions de bits se font facilement sur des valeurs binaires du mot de référence en inversant 1 et 0, mais que faire avec les *soft bits* qui encodent chaque phase reçue dans le message acquis avec une valeur continue codée sur 8 bits ? Sommes nous obligés de décider maintenant de la valeur attribuée à chaque phase (*soft* → *hard bits*), ou pouvons nous manipuler les valeurs brutes ? Il nous faut interpréter les échanges de bits comme des opérations de rotation ou de symétrie de la constellation (Fig. 3). Nous constatons qu'échanger des bits correspond à des opérations entre partie réelle et

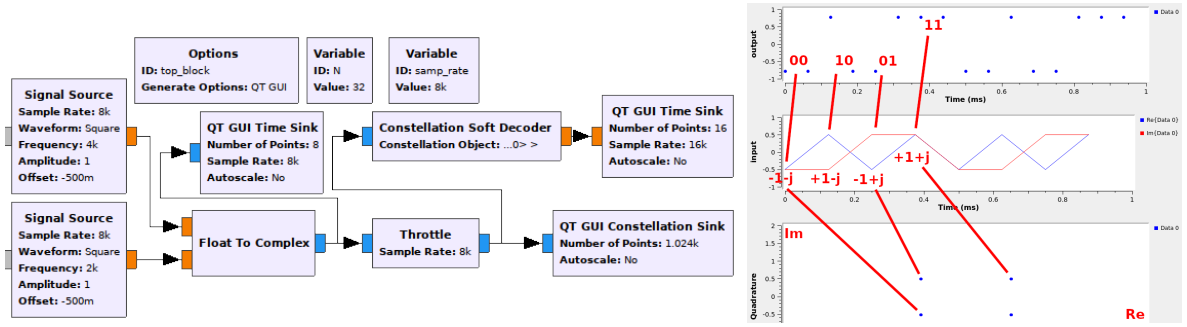

FIGURE 2 – Exemple de signaux synthétiques permettant d'analyser la configuration du *Constellation Soft Decoder* par `digital.constellation_calcdist(([-1-1j, -1+1j, 1+1j, 1-1j]), ([0, 1, 3, 2]), 4, 1).base()`. Nous constatons que l'assignation des symboles aux paires de bits ne correspond pas à la norme (Fig. 1).

partie imaginaire, soit de rotation, soit de symétrie le long d'un des axes du plan complexe. Ainsi, en manipulant la partie réelle et imaginaire des données acquises, nous pouvons atteindre le même résultat que l'échange des bits, mais en conservant les valeurs continues des *soft bits* et en repoussant l'attribution de chaque bit (0 ou 1) à chaque valeur de phase au décodage par l'algorithme de Viterbi.

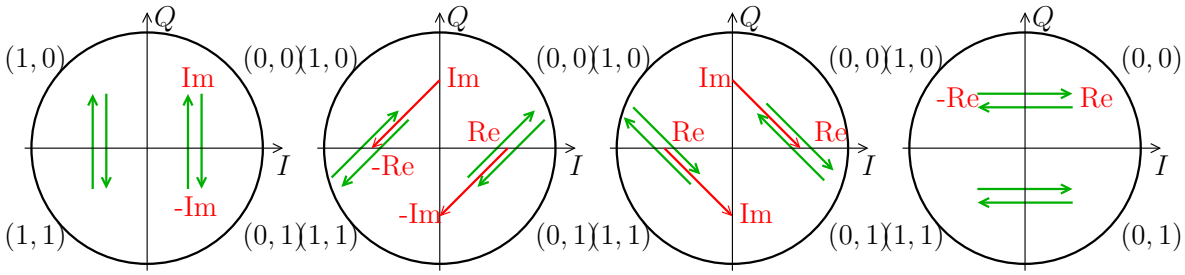

FIGURE 3 – Rotations et symétries de la constellation, et le résultat correspondant sur les axes réel et imaginaire (I et Q) des données brutes acquises.

L'identification de la correspondance entre les 4 états QPSK dans le diagramme de constellation (I en abscisse, Q en ordonnée) et les paires de bits correspondant nécessite donc une attaque brute, dans laquelle tous les cas possibles sont testés. La corrélation des phases des signaux acquis avec les diverses permutations possibles du code d'entête de trame encodé par convolution est illustrée en Fig. 4. Une seule condition d'attribution des symboles aux paires de bits fournit une séquence périodique de pics de corrélation (Fig. 4, bas) : il s'agit de la bonne correspondance que nous exploiterons pour la suite du décodage.

Cette permutation sera désormais appliquée à tous les couples I/Q du message acquis car nous savons que nous retrouverons alors la séquence de bits émise par le satellite lors du décodage par algorithme de Viterbi appliqué aux *soft bits* résultant.

1.1 Des bits aux phrases : mise en œuvre du décodage par algorithme de Viterbi

Nous avons maintenant une séquence de phases comprises dans l'ensemble $[0; \pi/2; \pi; 3\pi/2]$ convenablement organisée pour devenir une séquence de bits dans laquelle nous retrouvons le mot de synchronisation, et nous n'avons qu'une unique attribution des divers symboles $\{00; 01; 11; 10\}$ à chaque phase qui fournit une solution présentant cette corrélation. Il nous reste à décoder pour retirer l'encodage par convolution, puis appliquer aux bits résultants (qui étaient donc les bits encodés par le satellite avant convolution) une séquence de XOR (OU exclusif) avec un polynome garantissant de rendre la séquence aussi aléatoire que possible pour éviter tout risque de répétition trop long du même état de bit.

Nous avons mentionné la disponibilité de `libfec` implémentant efficacement le décodage des signaux encodés par code de convolution. Nous étendons ici l'exemple simple au cas pratique de la trame complète.

Notre première idée a consisté à décoder d'un coup l'ensemble du fichier acquis. De cette façon, nous cachons le problème de l'initialisation et de la fin du décodage du code de convolution par l'algorithme de Viterbi. Cela fonctionne, puisque nous vérifions après décodage que, tous les 1024 octets, nous retrouvons le code de synchronisation 0x1ACFFC1D.

Attention : nous avons rencontré un problème d'erreur d'accès à un segment mémoire (*segmentation fault* en allouant le tableau permettant de charger l'ensemble du fichier. En effet, le fichier contenant les octets en *soft bits* fait 11,17 MB, donc nous avions alloué un tableau statique comme le ferait tout bon développeur sur système embarqué qui n'a pas accès

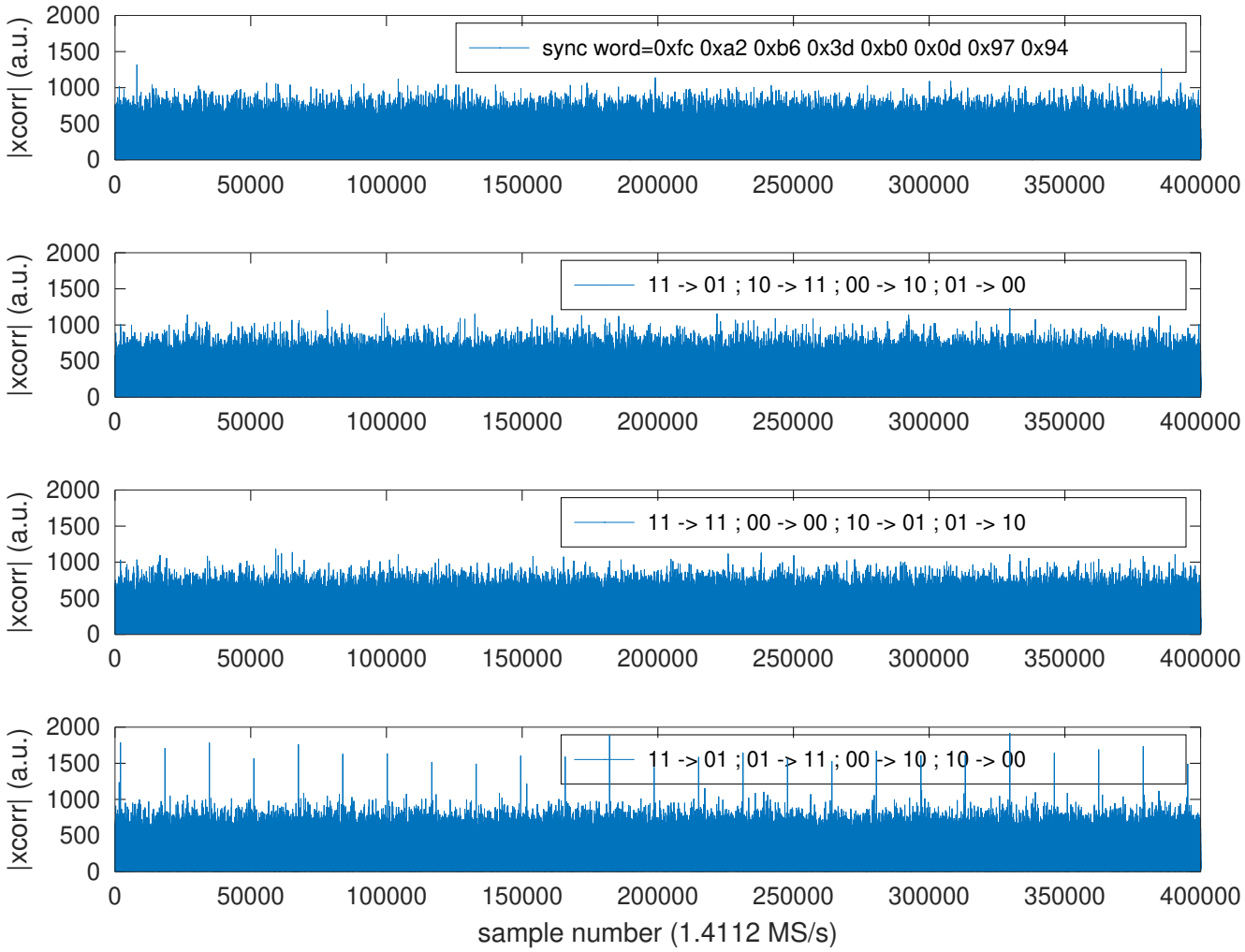

FIGURE 4 – Corrélation pour les 4 cas possibles de rotation de la constellation QPSK avec le code connu de début de trame. Nous constatons que seul le quatrième cas – en bas – fournit des pics périodiques de corrélation représentatifs du début de trame. C'est donc cette attribution des 4 symboles de QPSK aux paires de bits correspondant qui est correcte.

à l'allocation dynamique de mémoire `malloc`, gourmande en ressources. Cependant, ce faisant nous plaçons le tableau sur la pile, et GNU/Linux impose par défaut une pile de 8192 kB, tel que nous le dit `ulimit -s` : 8192. Plutôt qu'imposer une limite de taille de pile plus importante, nous nous autorisons sur notre système d'exploitation une allocation dynamique de mémoire pour placer le tableau sur le tas et non sur la pile, libérant ainsi la contrainte de place mémoire occupée.

```
1  #include <stdio.h>  // from libfec/vtest27.c
2  #include <stdlib.h> // gcc -o demo_libfec demo_libfec.c -I./libfec ./libfec/libfec.a
3  #include <fcntl.h>
4  #include <unistd.h> // read
5  #include <fec.h>
6  #define MAXBYTES (11170164/16)  // file size /8 (bytes-> bits) /2 (Viterbi)
7
8  #define VITPOLYA 0x4F
9  #define VITPOLYB 0x6D
10 int viterbiPolynomial[2] = {VITPOLYA, VITPOLYB};
11
12 int main(int argc,char *argv[]){
13   int i,framebits,fd;
14   unsigned char data[MAXBYTES],*symbols;
15   void *vp;
```

```
16
17   symbols=(unsigned char*)malloc(8*2*(MAXBYTES+6)); // *8 for bytes->bits & *2 Viterbi
18 // root@rugged:~# ulimit -a
19 // stack size              (kbytes, -s) 8192
20 // -> static allocation (stack) of max 8 MB, after requires malloc on the heap
21   fd=open("./extrait.s",O_RDONLY); read(fd,symbols,MAXBYTES*16); close(fd);
22
23   for (i=1;i<MAXBYTES*16;i+=2) symbols[i]=-symbols[i]; // I/Q constellation rotation
24   framebits = MAXBYTES*8;
25   set_viterbi27_polynomial(viterbiPolynomial);
26   vp=create_viterbi27(framebits);
27   init_viterbi27(vp,0);
28   update_viterbi27_blk(vp,&symbols[4756+8],framebits+6);
29   chainback_viterbi27(vp,data,framebits,0);
30   for (i=0;i<20;i++) printf("%02hhX",data[i]);
31   printf("\n");
32   fd=open("./sortie.bin",O_WRONLY|O_CREAT,S_IRWXU|S_IRWXG|S_IRWXO);
33   write(fd,data,framebits);
34   close(fd);
35   exit(0);
36 }
```

L. Teske [2] nous informe cependant que cette approche n'est pas optimale, car elle impose de charger tout le fichier en mémoire d'un coup. Nous savons que les blocs de 1024 octets (2048 octets après encodage) sont encodés individuellement, donc au lieu de décoder tout le fichier, nous pouvons nous contenter de chercher le code de synchronisation encodé, et décoder les 2048 octets qui suivent à partir de ce point. Pour plus de sécurité et laisser le décodeur s'initialiser, on prendra soin de prendre quelques valeurs avant et après le bloc à décoder. La partie `main` du code résultant est de la forme

```
1  fdi=open("./extrait.s",O_RDONLY);
2  fdo=open("./sortie.bin",O_WRONLY|O_CREAT,S_IRWXU|S_IRWXG|S_IRWXO);
3  read(fdi,symbols,4756+8);  // offset
4  framebits = MAXBYTES*8;
5
6  do {
7    res=read(fdi,symbols,2*framebits+50);  // prend un peu plus
8    lseek(fdi,-50,SEEK_CUR);               // revient en marche arriere
9    for (i=1;i<2*framebits;i+=2) symbols[i]=-symbols[i]; // I/Q constellation rotation
10   set_viterbi27_polynomial(viterbiPolynomial);
11   vp=create_viterbi27(framebits);
12   init_viterbi27(vp,0);
13   update_viterbi27_blk(vp,symbols,framebits+6);
14   chainback_viterbi27(vp,data,framebits,0);
15   write(fdo,data,MAXBYTES);              // resultat du decodage
16 } while (res==(2*framebits+50));        // ... tant qu'il y a des data a lire
17 close(fdi); close(fdo);
```

Cette approche de traitements par blocs de données permettra en plus d'appréhender le code correcteur d'erreur par bloc de Reed Solomon (section A). L'utilisation de ce code correcteur par blocs est optionnel : les blocs de données issus du décodage par l'algorithme de Viterbi sont utilisables en l'état, et dans un premier temps nous passerons le code correcteur d'erreur de Reed Solomon sous silence, en n'exploitant que les premiers 1024-4-128=892 octets dans chaque bloc (après avoir rejeté donc les 4 octets du mot de synchronisation en début de bloc et les 128 octets de code correcteur en fin de bloc). Nous reviendrons sur ce point en annexe.

Alternativement, le lecteur qui préfère se cantonner à GNU/Octave au lieu de passer au C pourra exploiter le code fourni à `https://github.com/Filios92/Viterbi-Decoder/blob/master/viterbi.m` qui fonctionne lui aussi parfaitement avec

```
1 f=fopen("extrait.s");   % soft bits generated from GNURadio
2 d=fread(f,inf,'int8');  % read file
3 d(2:2:end)=-d(2:2:end); % rotation constellation
4 phrase=(d<0)';          % soft -> hard bits
5 [dv,e]=viterbi([1 1 1 1 0 0 1 ; 1 0 1 1 0 1 1 ],phrase,0);
6 data=(dv(1:4:end)*8+dv(2:4:end)*4+dv(3:4:end)*2+dv(4:4:end));
```

Finalement, le lecteur désireux de poursuive son exploration de Meteor-M2 exclusivement par GNU Radio n'est pas en reste : les codes correcteurs d'erreur et `libfec` sont implémentés dans `gr-satellite` de `github.com/daniestevez/gr-satellites` et décrits en détails dans le blog de D. Estévez à `destevez.net`. Sous sa direction lors de divers échanges de courriers électroniques, nous avons finir par faire aboutir le décodage du mot de synchronisation par son bloc de déconvolution par algorithm de Viterbi selon le schéma de traitement proposé en Fig. 5. Partant d'un fichier binaire contenant le mot encodé `fca2b63db00d9794`, par exemple généré par

```
echo -e -n "\xfc\xa2\xb6\x3d\xb0\x0d\x97\x94" > input.bin
```

nous visons à retrouver le mot de synchronisation `1acffc1d` qui démontrerait notre compréhension du décodeur. **Attention :** le point qui nous a bloqué n'est pas la configuration du décodeur, qui reprend exactement la notation de `libfec`, mais le format des données en entrée. En effet, GNU Radio s'attend, tel que décrit au détour de la page d'aide de `fec_decode_ccsds_27_fb` (bloc GNU Radio Companion `Decode CCSDS 27`) de `gr-fec`, à recevoir une entrée comprise entre -1 et +1 (et non 0 à +1 comme nous nous y serions attendu pour une représentation de bits : il s'agit ici de *soft bits* compris entre $\exp(j\pi) = -1$ et $\exp(j0) = +1$ et non de *hard bits*). Nous prenons donc les bits issus de la lecture du fichier contenant le mot encodé, multiplions par 2 (donc facteur d'homothétie inverse de 0,5, valeur à renseigner dans `Char to Float`), retranchons 1 et éventuellement échangeons les valeurs des bits (multiplication par -1) pour atteindre le bon résultat et non son opposé. Nous vérifions sur la sortie graphique de Fig. 5 que la lecture du fichier est fonctionnelle, et en observant la sortie de ce flux de traitement (par `xxd` par exemple) nous trouvons

```
$xxd result.bin | cut -d: -f2
 1acf fc1d 1acf fc1d 0334 53c0 1acf fc1d  .........4S.....
```

qui n'est pas parfait mais ressemble bien à nos attentes. Nous trouvons bien la séquence `1acffc1d` mais parfois une séquence erronée `033453c0` se glisse du fait d'une mauvaise inialisation du décodeur de Viterbi. En effet Viterbi fait l'hypothèse d'une initialisation avec tous les éléments du registre à décalage à 0, qui n'est pas nécessairement le cas ici lors d'une lecture répétitive du mot encodé. D. Estévez présente à `destevez.net/2017/01/coding-for-hit-satellites-and-other-ccsds-satellites/` les diverses déclinaisons sur la configuration du décodeur pour respecter les diverses déclinaisons de la norme du CCSDS.

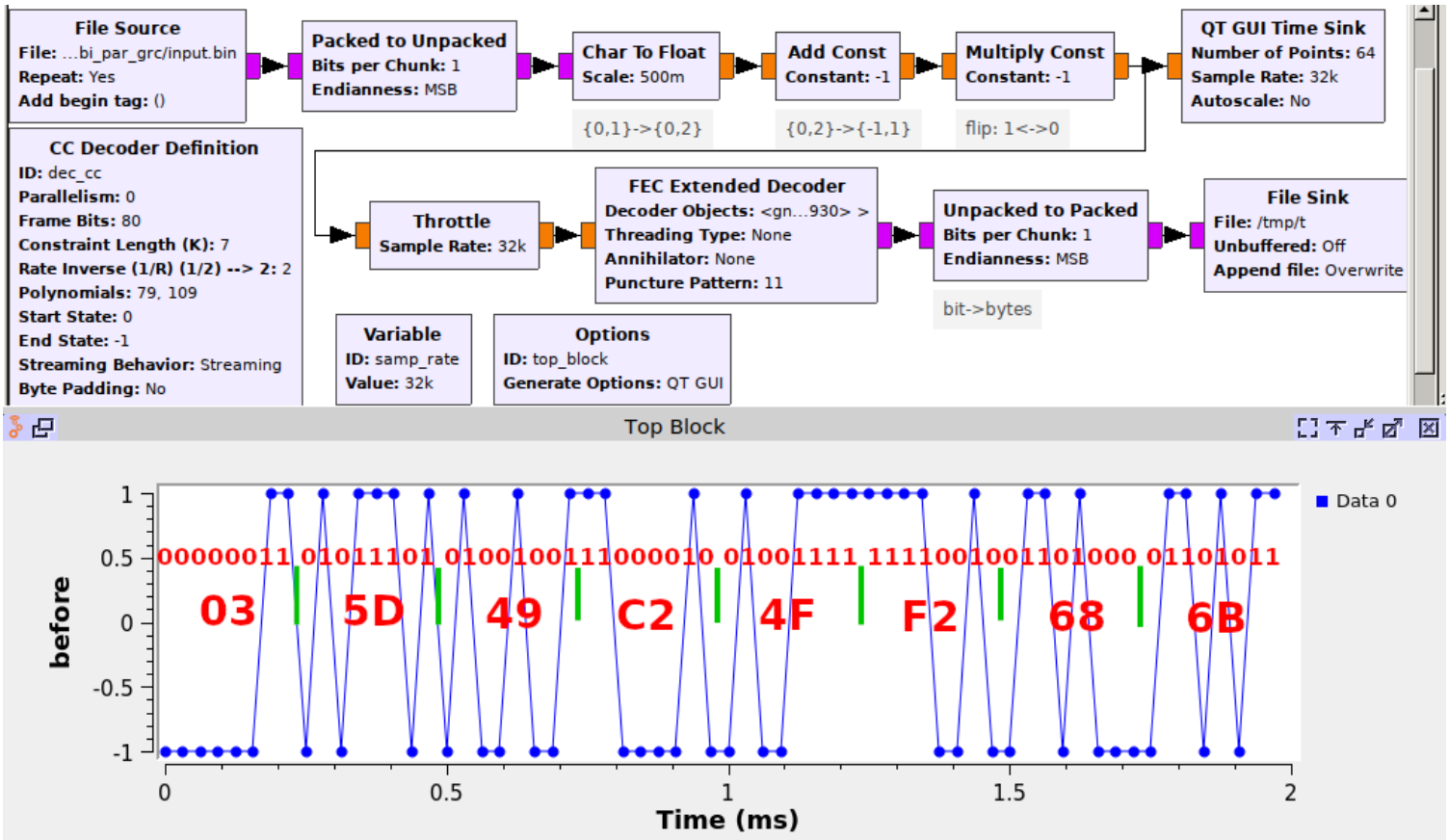

FIGURE 5 – Flux de traitement dans GNU Radio Companion exploitant `gr-satellite` et le bloc généraliste de décodage du code de convolution par Viterbi `FEC Extended Decoder` configuré pour respecter la norme du CCSDS pour démontrer le décodage du mot de synchronisation. L'entrée se généralisera donc au fichier enregistré et contenant les soft-bits de Meteor-M2. Le mot annoté sur le graphique du bas est l'opposé du mot fourni dans le fichier d'entrée 0xFCA2B63DB00D9794 issu de l'encodage du code de synchronisation pour que la sortie réponde à nos attentes : l'inversion de bits est prise en charge par la multiplication par -1 juste avant l'affichage en haut à droite du flux de traitement.

Nous prétendons avoir décodé les messages venant du satellite, mais sommes nous certains de la validité de ces bits ? Afin de chercher rapidement une séquence connue de bits, nous reprenons le cas proposé au début de cet article, à savoir rechercher par corrélation un motif connu. Or il se trouve que, sans rien connaître à l'encodage des paragraphes qui formera l'objet de la suite de ce texte, la description du protocole à [5] inclut dans son Appendice A une description de la trame de télémétrie sensée contenir la date à bord du satellite, et cette trame est identifiée par le code PRS-64 formé de la séquence "2 24 167 163 146 221 254 191". Sommes nous capables de trouver cette séquence d'octets dans les trames décodées ? Ayant obtenu des bits que nous supposons

avoir stocké dans le tableau `dv`, nous les concaténons sous GNU/Octave en quartets puis en octets (tableau `data`) et en trames (matrice `fin` ci-dessous) :

```
% data est tableau d'octets issus de Viterbi tel que vu auparavant
for k=1:24                              % on analyse les 24 premiers blocs
 d(:,k)=data(1+(k-1)*2048:k*2048);      % trames de 2048 quartets
 dd(:,k)=d(1:2:end,k)*16+d(2:2:end,k); % quartets -> octets
 fin(:,k)=dd(5:end,k);                  % retire l'entete de synchro de chaque paquet
 fin(:,k)=[bitxor(fin(1:255,k)',pn) bitxor(fin(1+255:255+255,k)',pn) ...
             bitxor(fin(1+255*2:255+255*2,k)',pn) bitxor(fin(1+255*3:255+255*3,k)',pn)];
end
```

Nous constatons que nous avons du appliquer un code `pn` qui a été conçu pour rendre la séquence de bits aussi aléatoire que possible (et donc distribuer l'information) par masquage bijectif (XOR). Cette structure aléatoire des trames évite de longues séquence de la même valeur de bits rendant la récupération de l'horloge compliquée. La séquence de `pn` longue de 255 éléments est fournie à `https://www.teske.net.br/lucas/2016/11/goes-satellite-hunt-part-4-packet-demuxer/` et nous nous contentons de l'appliquer à nos octets regroupés par paquets de 255 (les 4 octets d'entête, qui ne subissent pas cette transformation, ont déjà été retirés).

Finalement ces trames sont analysées pour y rechercher l'occurence de la séquence magique PRS-64 représentative de la trame de télémétrie. L'obtention de cette séquence prouve que notre procédure est cohérente, car non seulement nous identifions la séquence d'octets indicatrice de la télémesure – séquence qui n'a à peu près aucune chance d'apparaître par hasard – mais en plus l'analyse de la télémesure fournit un résultat cohérent avec les conditions d'acquisition et le décodage par `meteor_decoder` (`Onboard time: 11:48:33.788`) sous la forme

```
date_header=final(589:589+7,9)' % on a trouve' la date dans CADU 9 (sur les 24 traitees)
date=final(589+8:589+11,9)'
ans =
    11
    48
    33
   197
```

Nous avons réussi a trouver l'heure qu'il était à bord du satellite au moment de la capture de l'image, validant la procédure d'obtention des octets à partir du flux de données I/Q. Maintenant que nous sommes confiants sur la séquence de bits, le reste de l'analyse n'est plus que de l'assemblage de trame et du décodage d'images, une activité plus classique d'informatique que de traitement du signal.

2 Des phrases aux paragraphes

La séquence de bits respecte la norme décrite dans les documents techniques, nous avons donc fini notre travail de décodage. Pas tout à fait ... le satellite transmet une image, et nous avons obtenu une date. C'est un peu maigre comme résultat. Pouvons nous aller plus loin ?

C'est là que les couches de la norme OSI apparaissent. Une image est une somme conséquente d'informations, trop grosse pour être contenue dans un unique paquet transmis depuis le satellite. Pire, le satellite envoie au moins 3 bandes spectrales (selon que nous soyons le jour ou la nuit, ces bandes changent ... c'est quoi le jour ou la nuit au Spitsberg, avec ses 3 mois de jour et 3 mois de nuit complets ?!) qui sont interlacées entre les divers paquets. Nous comprenons mieux pourquoi la norme OSI sépare les divers niveaux d'abstraction : une image est une entité découpée en couches de couleurs qui sont elles mêmes découpées en blocs (codage JPEG) qui sont eux mêmes découpés en paquets qui sont transmis vers le sol avec toute l'artillerie de correction d'erreurs et de redondance pour maximiser les chances que le récepteur reçoive un flux de données intègre. J'ai bien écrit JPEG dans la phrase précédente : pour quelqu'un qui a été élevé avec tous les méfaits des compressions à pertes d'images et des artéfacts induits par la compression JPEG, est-il possible que les images transmises par satellites soient codées ainsi ? Le compromis tient probablement entre le débit de données accessibles sur une liaison relativement basse fréquence et la masse de données à transmettre pour récupérer une image haute résolution : on prendra évidemment soin, lors du traitement de telles images, de ne pas s'appitoyer sur les artéfacts de codage des blocs de 8×8 pixels qui vont être l'objet de la discussion qui va suivre.

Le point fort peu clair dans la documentation tient sur la définition du *Minimum Code Unit* (MCU) : nous apprenons que chaque MCU comporte 196 zones adjacentes d'une image, chacune formée de 14 imagettes de 8×8 pixels. Le point qui ne nous était pas apparu clair dans la documentation est que les *MCU successifs sont indépendants les uns des autres*. Ainsi, une ligne d'image est formée de 14 MCUs, chacune contenant 14 imagettes de 8×8 pixels : 14×14=196 et $14 \times 14 \times 8 = 1568$ pixels est bien la largeur d'une image issue de Meteor-M2. Donc notre objectif est de décoder des imagettes de 8×8 pixels encodées en JPEG, de les concaténer, et ce jusqu'à former une ligne d'une image dans une composante spectrale. La procédure se répète pour les deux autres bandes spectrales, avant de recommencer suite à une trame de maintenance (identifiant 70 fournie dans le champ APID, *APplication IDentifier* – les images ont quand à elles des identifiants APID compris entre 64 et 69 [5]).

Par ailleurs, un paquet MCU peut ne pas entrer complètement dans la charge utile de la couche protocolaire M-PDU : il se peut que la charge utile d'un M-PDU contienne plusieurs MCU successifs (par exemple lorsque les imagettes compressées en JPEG sont petites, tel que lors de la transmission d'une zone de couleur uniforme) ou bien qu'un MCU soit distribué entre deux

M-PDU. Ainsi, l'entête du paquet VCDU contient un pointeur qui nous informe de l'indice auquel le prochain M-PDU démarre. Avant ce pointeur, nous obtiendrons la fin du paquet M-PDU précédent.

Nous avions placé dans la matrice `fin` les diverses trames successives décodées après application de l'algorithme de Viterbi et *derandomization* : nous affichons le contenu des premières lignes de cette matrice pour constater la cohérence d'un certain nombre de motifs – entête des paquets – avant d'atteindre la charge utile qui sera elle aléatoire. Le document qui nous est apparu le plus limpide pour décoder les trames est [6].

```
octave:28> fin(1:16,:) % voir 20020081350.pdf NASA p.9 du PDF
    64    64    64    64    64    64    64    64    64    64    64    64    64    64  Version
     5     5     5     5     5     5     5     5     5     5     5     5     5     5  Type
   140   140   140   140   140   140   140   140   140   140   140   140   140   140  \
   163   163   163   163   163   163   163   163   163   163   163   163   163   163  - counter
    43    44    45    46    47    48    49    50    51    52    53    54    55    56  /
     0     0     0     0     0     0     0     0     0     0     0     0     0     0 sig. field
     0     0     0     0     0     0     0     0     0     0     0     0     0     0 VCDU insert
     0     0     0     0     0     0     0     0     0     0     0     0     0     0 ... zone
     0     0     2     1     0     0     0     0     0     0     0     0     2     0 5 bits @ 0
    18   142    28    54    18   130    78   226     0    20    70    28    82    32 M_PDU header

    77   166   239   222    73    82    83   199     8    28   232   247   165   183 M_PDU...
   133   188   229    42    24    23   220    94    68   117    92   151    87   203 ... 882 bytes
    75   177   221   215     0    48    49   128    13   166     8   218   126   212
    42   138   254    87    12    32   249    87    34   172   247   107     9   142
   146   238   236    80   215    96   143   121     0   124    12    89    86   191
   179   227    64   144    89    59   240   105   105   251    46    43     0   199
                                                     ^
```

que nous analysons comme suit, de la première à la dernière ligne, en commençant par le *VCDU Primary Header* de 6 octets :

1. 64 correspond à la version constante de 01 suivie de 6 zeros pour les bits de poids fort du VCDU Id (S/C id). Ainsi, les 8 premiers bits sont 0100 0000,
2. suit l'identifiant du satellite qui transmet (champ Type de VCDU Id) : ce champ est renseigné à [5] comme valant 5 si l'instrument est présent et 63 si l'instrument est absent. Ici une valeur de 5 est un bon présage pour la suite du décodage des images. Par ailleurs, [7, p.149] indique qu'un VCDU Id de 5 (AVHRR LR) sera associé aux canaux APID 64..69 que nous identifierons plus tard.
3. le VCDU Counter de 3 octets s'incrémente à chaque paquet, tel que nous l'observons avec le décompte sur le dernier octet (140 163 43..56) du triplet d'octets correspondant au compteur de trame,
4. tous les champs suivants (*signaling field*) sont renseignés comme nuls pour indiquer la transmission de données en temps réel, tout comme les champs VCDU Insert zone et l'absence de cryptographie [7, p.150],
5. enfin, les deux derniers octets de l'entête fournissent un pointeur qui nous indique à quel emplacement se trouvera le début du premier paquet contenu dans cette trame. Cette information est probablement la plus importante car un paquet M_PDU a toutes les chances de se partager entre plusieurs trames, et donc savoir où commence le premier paquet M_PDU contenu dans la trame permet de synchroniser le début du décodage d'une nouvelle image. Les 5 premiers bits sont toujours nuls [7, p.147] tandis que les 11 derniers bits donnent l'adresse, dans la trame, du début du premier paquet utile. Dans la séquence proposée ici, le pointeur se calcule comme

   `x=fin(9,:)*256+fin(10,:)+12 = 30 154 552 322 30 142 90 238 12 32 82 40 606 44`
6. les 882 octets qui suivent sont la charge utile M-PDU contenant les champs du canal virtuel (*Virtual channel field*). Nous pourrons nous convaincre que la position de l'entête identifiée ci-dessus est correcte par

   `for k=1:length(x);fin(x(k),k),end`

   qui renvoit `64 64 64 64 65 65 65 65 68 64 64 64 64 65` qui est la liste des identifiants des canaux virtuels que nous analyserons ci-dessous, ie les diverses longueurs d'ondes observées pour former les images (APID compris entre 64 et 69 [5])
7. la 9ème colonne est un peu spéciale car elle contient le premier paquet de la séquence de transmissions du canal d'APID 68, donc un entête d'offset 0 par rapport à la fin de l'entête du VCDU, qui permet de commencer à appréhender le format de la charge utile M_PDU sans avoir à en rechercher le début. Nous verrons ainsi que 8=0000 1000 est la version (ID=000/Type Indicator=0/Secondary Header 1=present/000 APID), puis APID=68 est un des canaux de mesure [5] et finalement la longueur du paquet (en octets) est fournie par {0 105}.

3 Que de texte ... des images maintenant

Nous avons identifié comment décoder la trame VCDU, maintenant il reste à analyser la charge utile qu'est le M_PDU. Plusieurs M_PDU peuvent se regrouper dans une même trame VCDU (par exemple lorsque la charge utile qu'est une imagette JPEG est fortement compressée) et un M_PDU peut se distribuer entre deux VCDU successifs – il n'y a aucune raison pour que la taille de la charge utile M_PDU soit multiple de la taille d'une trame VCDU.

Nous avons identifié auparavant les pointeurs, dans l'entête du VCDU, de l'adresse de début de chaque paquet M_PDU contenu dans le VCDU. Si nous affichons les premiers octets de chaque M_PDU, nous observons un motif cohérent

```
  8    8    8    8    8    8    8    8    8    8    8
 68   68   68   68   68   68   68   70   64   64   64
 13   13   13   13   13   13   13  205   77   13   13
 34   35   36   37   38   39   40   41   42   43   44
  0    0    0    0    0    0    0    0    0    0    0
105   47   49   69   81  107   57   57   97   77   79
  0    0    0    0    0    0    0    0    0    0    0
  0    0    0    0    0    0    0    0    0    0    0
  2    2    2    2    2    2    2    2    2    2    2
136  136  136  136  136  136  136  136  136  136  136
181  181  181  181  181  181  181  181  186  186  186
124  124  124  124  124  124  124  124   76   76   76
  0    0    0    0    0    0    0    0    0    0    0
  0    0    0    0    0    0    0    0    0    0    0
 98  112  126  140  154  168  182    2    0   14   28
  0    0    0    0    0    0    0   24    0    0    0
  0    0    0    0    0    0    0  167    0    0    0
255  255  255  255  255  255  255  163  255  255  255
240  240  240  240  240  240  240  146  240  240  240
 77   77   77   77   77   77   77  221   77   79   81
243  186  210  178  136  175  242  154  173  238  235
197   41  160  177  253  120  216  191  166  148   77
 60  194  210  146  236    9  151   11   88  100  166
240  156   80  106   84   81  201   48   42  228  208
105   41    5   65  152  245  135   33  131  208    9
254  104   41   23  193  172   56  197   38  210   28
 91   52   20  205    1  233  249    0   62  115  118
```

que nous allons analyser [5] dans le cas particulier de la première colonne :

1. "68" est l'identifiant (*packet identifier*) de l'instrument à bord du satellite qui transmet l'information, APID. Comme une image d'un instrument s'étalera sur plusieurs paquets successifs, il est normal que nous retrouvions plusieurs fois de suite le même APID. Un cas intéressant est l'APID 70 de la colonne 8 qui indique une trame de télémesure, qui nous avait permis auparavant de trouver l'heure à bord du satellite au moment de la prise de vue.
2. "13 34" est formé des deux premiers bits indiquant s'il s'agit du premier paquet d'une séquence (01) ou la suite d'une transmission (00) suivi du compteur de paquets, sur 14 bits, qui nous servira à savoir si nous avons perdu une imagette dans une ligne. Nous constatons bien que l'octet de poids faible s'incrémente à chaque nouveau M_PDU.
3. les deux octets qui suivent indiquent la longueur du paquet M_PDU, ici 105 octets,
4. suit la date codée sur 64 bits, soient un jour sur deux octets "0 0", un nombre de millisecondes dans la journée codé sur 32 bits "2 136 181 124" valable pour tous les paquets d'une même image, et finalement un complément de dates en microsecondes codé sur 16 bits et fixé, pour Meteor-M2, à "0 0" [5].
5. La description du champ de données indique l'indice du premier MCU (*Minimum Code Unit*), imagette dont l'assemblage formera une ligne de l'image finale. Cet indice de MCU s'incrémente de 14 entre deux paquets successifs, ici `98 112 126`, puisque les imagettes sont regroupées par paquets de 14 pour améliorer la compression [5]
6. finalement, l'entête de l'image contient 16 bits fixés à 0 [5] (*Scan Header* suivi du *Segment Header* contenant un indicateur de présence de facteur de qualité sur 16 bits fixé à 0xFF 0xF0 ou `255 240` [5] suivi de la valeur de ce facteur de qualité qui interviendra dans la quantification de l'imagette JPEG – dans notre cas 77 mais variable le long d'une ligne de l'image finale.
7. Suivent les données des 14 MCUs successifs formant 14 imagettes de 8×8 pixels, ou 64 octets successifs dans la nomenclature de [5] (dans le cas de la première trame, cette charge utile commence par `243 197`).

Cette description un peu fastidieuse est nécessaire pour bien comprendre le lien entre les VCDUs et les M_PDU qui finalement représentent deux abstractions du flux de données à deux niveaux différents des couches OSI. Une fois cette distinction assimilée, l'assemblage des imagettes JPEG pour former une ligne n'est plus que question de rigueur dans le suivi de la norme. Sous GNU/Octave, les octets représentant chaque MCU composé de 14 imagettes sont regroupés dans des fichiers individuels suivant le bout de programme

```
1 for col=1:23                              % numero de colonne = numero de frame VCDU
2   first_head=fin(9,col)*256+fin(10,col) % 70 pour colonne 11
3   fin([1:first_head+1]+9,col)';       % debut de la ligne 11 : 1er header en 70
4   fin([1:22]+first_head+11,col)';     % debut du MCU de la ligne 11
5
6   clear l secondary apid m
7   l=fin(first_head+16-1,col)*256+fin(first_head+16,col); % vector of packet lengths
8   secondary=fin(first_head+16-5,col);     % initialise la liste des entetes
9   apid=fin(first_head+16-4,col);          % initialise la liste des APIDs
```

```
  m=fin([first_head+12:first_head+12+P],col);
  k=1;
  while ((sum(l)+(k)*7+first_head+12+P)<(1020-128))
    m=[m fin([first_head+12:first_head+12+P]+sum(l)+(k)*7,col)];
    secondary(k+1)=fin(first_head+16+sum(l)+(k)*7-5,col);
    apid(k+1)=fin(first_head+16+sum(l)+(k)*7-4,col);
    l(k+1)=fin(first_head+16-1+sum(l)+(k)*7,col)*256+fin(first_head+16+sum(l)+(k)*7,col);
           % 16=offset from VDU beginning
    k=k+1;
  end
  for k=1:length(l)-1  % sauvegarde dans un fichier des octets de chaque MCU
    jpeg=fin([1:l(k)]+first_head+12+19+sum(l(1:k-1))-1+7*(k-1),col);
    f=fopen(['jpeg',num2str(apid(k),'%03d'),'_',num2str(col,'%03d'),'_',num2str(k,'%03d'),'.bin'],'w');
    fwrite(f,jpeg,'uint8');
    fclose(f);
  end

  k=length(l);                                        % dernier paquet incomplet
  jpeg=fin([1+first_head+12+19+sum(l(1:k-1))-1+7*(k-1):end],col);
  first_head=final(9,colonne+1)*256+final(10,col+1); % cherche dans VCDU suivant
  jpeg=[jpeg ; final([1:first_head]+10,col+1)];
  % on annoncait 79 octets dans le dernier paquet : il en manque 925-892=33
  f=fopen(['jpeg',num2str(apid(k),'%03d'),'_',num2str(col,'%03d'),'_',num2str(k,'%03d'),'.bin'],'w');
  fwrite(f,jpeg,'uint8');
  fclose(f);
end
```

Ayant sauvé dans des fichiers individuels `jpeg*.bin` les MCUs, nous devons aborder le problème d'interpréter leur contenu comme des images. Ceci nécessite de ré-implémenter le décodage de Huffman, RLE puis transformée en cosinus discrète pour convertir chaque imagette au format JPEG en une matrice de pixels affichable. En particulier le décodage de Huffman est pénible à implémenter par sa manipulation des données par paquets de bits qui ne sont pas multiples de 8 mais dépend de la taille de chaque information dans l'arbre binaire d'encodage. Nous nous contentons dans un premier temps de lire et comprendre le code source du décodeur de `meteor_decoder`, traduit en C++ à `github.com/infostellarinc/starcoder/blob/master/gr-starcoder/lib/meteor`, et à l'intégrer dans un petit bout de code pour valider le contenu des paquets de bits issus du traitement précédent et que nous affirmons encoder des imagettes au format JPEG, pour aborder plus en détail cette séquence de traitement dans la section qui va suivre. Cette fois, la séquence d'encodage (et donc de décodage) est très bien décrite dans [8]. Le fait que le décodeur de `meteor_decoder` accepte nos MCUs comme valides et en déduise des imagettes cohérentes qui s'assemblent convenablement prouve que notre décodage des VCDUs puis MCUs a été correcte.

```
#include "meteor_image.h"
using namespace gr::starcoder::meteor;

int main(int argc,char **argv)
{int fd,len,k,quality=77;    // quality fixe
 unsigned char packet[1100]; // sera arg plus tard
 imager img=imager();
 if (argc>1) quality=atoi(argv[1]);
 fd=open("jpeg.bin",O_RDONLY);
 len=read(fd, packet, 1100);
 close(fd);
 img.dec_mcus(packet, len, 65,0,0,quality);
}
```

se linke sur `meteor_image.cc` et `meteor_bit_io.cc` de l'archive github citée ci-dessus. En exploitant ce programme à qui nous fournissons en format binaire la charge utile qu'est chaque MCU, nous obtenons en sortie une matrice de 14×64 éléments que nous nommerons `imag`, chaque ligne de 64 octets étant elle même une imagette de 8×8 pixels. En réorganisant sous GNU/Octave ces 64 élements par

```
m=[];for k=1:size(imag)(1) a=reshape(imag(k,:),8,8); m=[m  a'];end
```

nous obtenons une matrice de 112×8 pixels que nous affichons par `imagesc(m)` pour visualiser l'image telle que présentée en Fig. 6 (gauche). Cette procédure est répétée pour les 14 MCUs qui forment une ligne de l'image finale : la Fig. 6 illustre la concaténation de la première série d'imagettes (gauche) avec une seconde série, démontrant la continuité des motifs. Cette séquence se poursuit pour toute une ligne d'image acquise à une longueur d'onde donnée par un instrument donné (donc une valeur d'APID donnée) avant de reprendre pour un nouvel APID et ainsi former en parallèle plusieurs images acquises à plusieurs longueurs d'ondes. On notera l'excellent facteur de compression de JPEG sur ces zones relativement homogènes : il ne faut qu'une soixantaine d'octets pour encoder ces images de $14 \times 64 = 896$ pixels. Les zones les plus structurées nécessitent tout de mêmes des MCUs de plusieurs centaines d'octets, voir jusqu'à 700 octets.

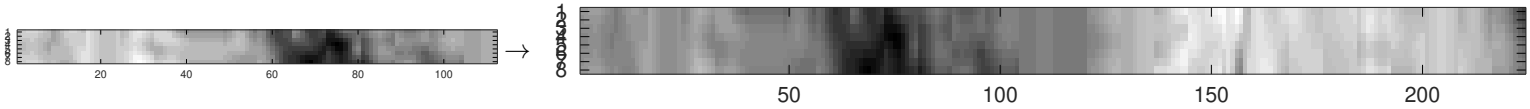

FIGURE 6 – Décodage d'un MCU (gauche) formé de 14 imagettes de 8×8 pixels successives, et concaténation avec le MCU suivant (droite) pour former une image de $28 \times 8 = 224$ pixels de large et 8 pixels de haut. L'image complète finale sera ainsi assemblée petit à petit par morceaux de MCU. Cet exemple porte sur l'APID 68.

Le résultat de l'assemblage des images décompressées de JPEG vers des matrices de 8×8 pixels est illustré en Fig. 7 pour l'instrument d'APID 68. Nous commencçons à entrevoir une séquence cohérente d'images, mais clairement il manque quelques vignettes sur chaque ligne car quelques paquets étaient corrompus et n'ont pas pu donner lieu à un décodage (Fig. 7).

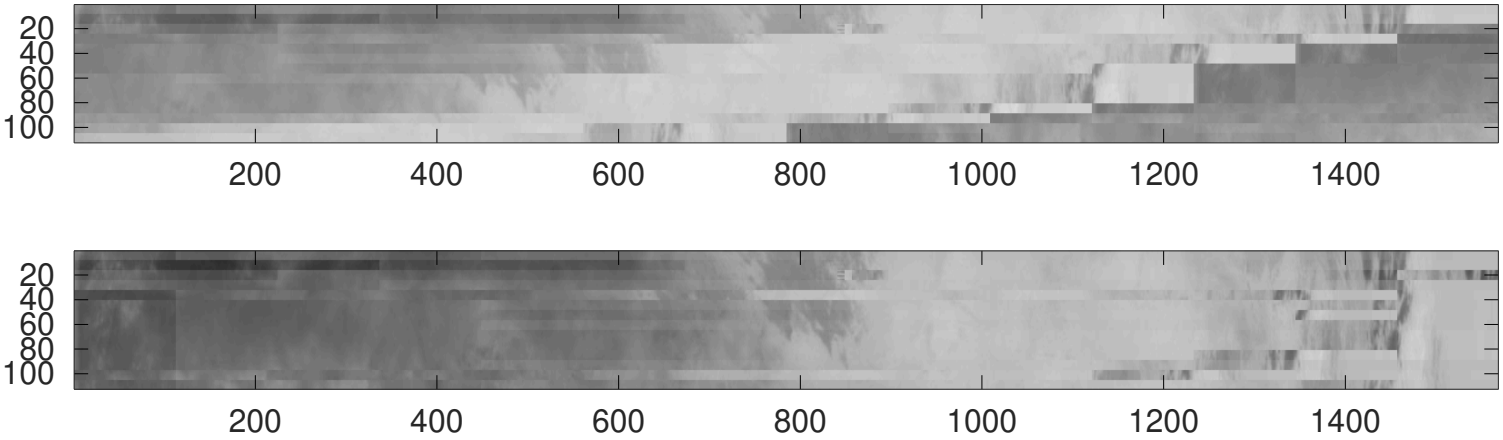

FIGURE 7 – Haut : résultat du décodage des imagettes JPEG et assemblage sans tenir compte du compteur. Nous observons clairement un décalage du motif lorsque des paquets manquent, résultant dans une image à peine exploitable. Bas : si le compteur de paquet n'atteint pas la valeur attendue de 14 imagettes/MCU, alors des faux paquets manquant sont insérés : cette fois l'image est convenablement alignée. Ici l'APID est 68.

Nous palions à ce dysfonctionnement, au moins provisoirement, en duppliquant chaque image manquante tel que indiqué par le compteur de MCU : à défaut de retrouver l'information perdue, nous arrivons au moins à aligner selon la verticale de l'image des vignettes adjacentes et ainsi obtenir une image reconnaissable (Fig. 8). Dans cet exemple nous n'avons pas utilisé l'indice de qualité qui modifie les facteurs que quantification de la compression JPEG en fonction de la nature de l'image, et des discontinuités sur les tons de gris restent visibles.

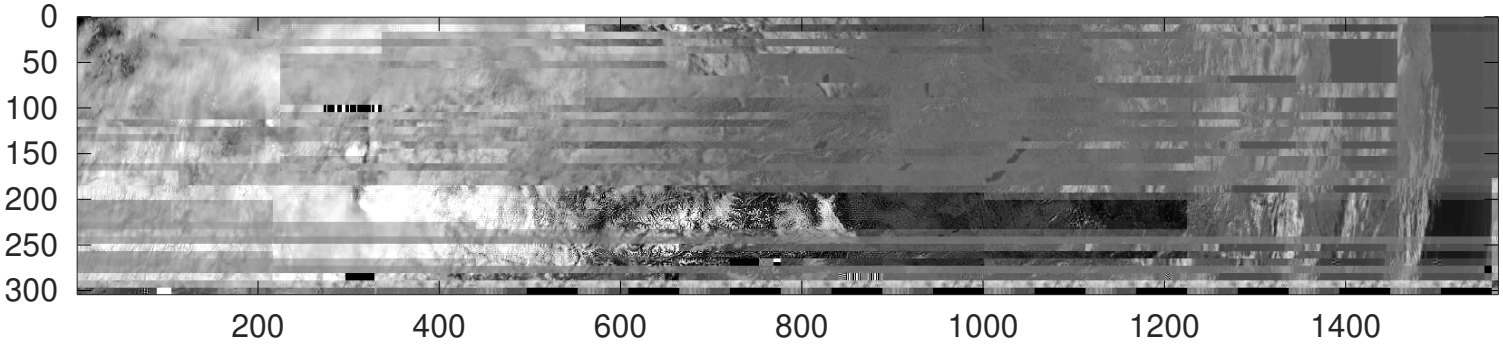

FIGURE 8 – Résultat du décodage de l'APID 65 avec exploitation du compteur pour identifier les imagettes manquantes, mais sans exploitation de l'information de qualité du code JPEG. Les caractéristiques géographiques sont visibles, mais de forts contrastes existent au sein de l'image, la rendant peu esthétique.

En intégrant le facteur de qualité comme argument passé au décodeur d'imagettes de `meteor_decoder` sur lequel nous lions notre programme, les tons de gris s'homogénéisent pour donner un résultat convaincant (Fig. 9) proche de la référence qu'est l'image entièrement décodée par `meteor_decoder` (Fig 10). On notera qu'il s'agissait d'un passage du satellite loin de l'optimal lors d'une écoute depuis la France puisque sont clairement visibles l'Istrie, les Alpes italiennes et autrichiennes ainsi que le lac Balaton en Hongrie (structure allongée à l'abscisse 1050 environ en milieu d'image), laissant penser à un passage s'approchant de l'horizon Est.

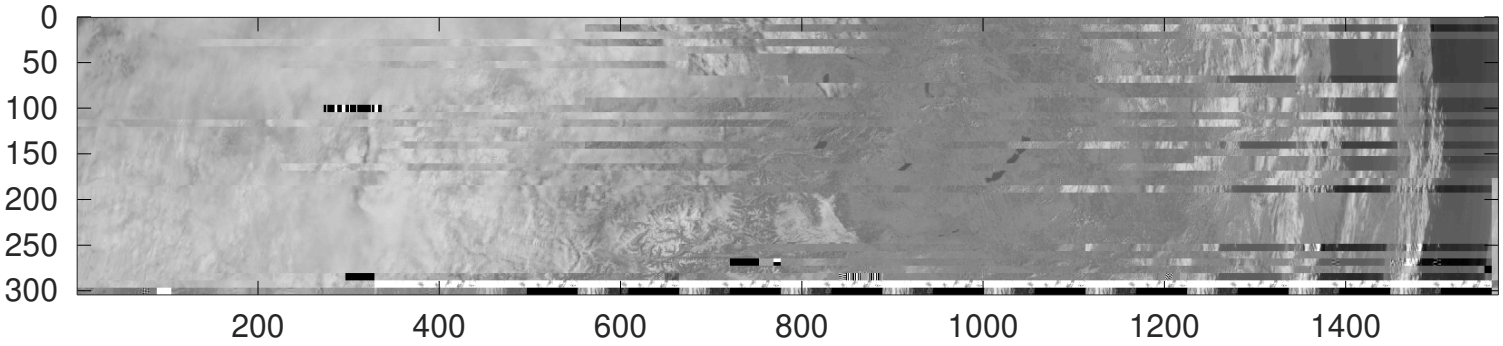

FIGURE 9 – Résultat du décodage de l'APID 65 avec exploitation du compteur pour identifier les imagettes manquantes, et de l'information de qualité du code JPEG. Les imagettes individuelles sont maintenant à peine visibles et les tons de gris évoluent continument sur l'image.

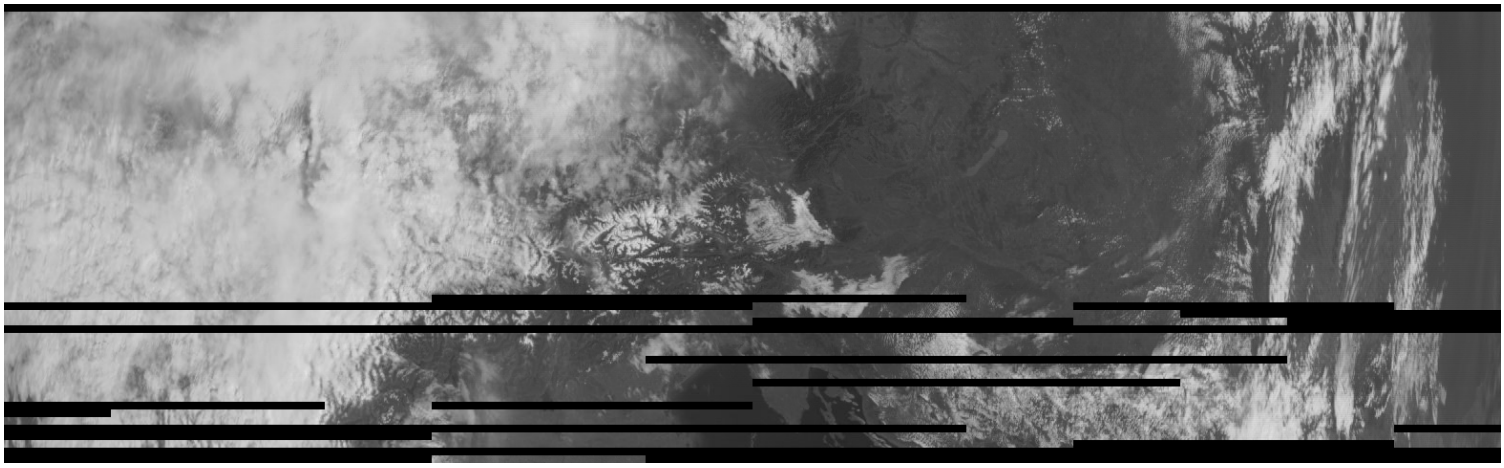

FIGURE 10 – Résultat du décodage de l'APID 65 par `medet` servant de référence pour comparaison avec la figure 9.

4 Décodage des images JPEG

Nous avons exploité `gr-starcoder` sans en comprendre le contenu. Cela ne saurait nous satisfaire : nous devons comprendre par quelle magie des coefficients de Fourier ont été convertis en une imagette de d'intensité lumineuse de pixels dans le domaine spatial (nous sommes en tons de gris donc nous oublions le concept de couleur : seule l'intensité lumineuse, ou luminance, importe dans ce traitement).

Nous désirons comprendre comment, à partir du MCU fourni par le décodage des trames, retrouver une imagette au format JPEG de 8 par 8 pixels. Nous savons qu'il y a un codage sans pertes de Huffman, avec éventuellement élimination des redondances en copiant une valeur qui se répète (RLE – *Run Length Encoding*), pour finalement convertir les coefficients de Fourier dans le domaine spatial par une transformée en cosinus discrète.

Le principal point de friction tient en la définition de la table de Huffman. Tout le monde pourra visiter les multiples pages décrivant [9, section 2.2, p.111] le calcul de l'arbre binaire permettant de trouver une représentation des données minimisant le nombre de bits nécessaires à représenter les informations les plus fréquentes, quitte à augmenter le nombre de bits nécessaires à représenter les informations les moins fréquentes : statistiquement, le fichier s'en trouvera souvent réduit en taille (c'est pourquoi la compression d'un fichier texte, avec beaucoup de redondance, est très efficace alors que la compression d'un exécutable binaire, où tous les octets sont à peu près équiprobables, ne donne que des résultats médiocres). Un point qui peut rendre le fichier compressé *plus volumineux* que le fichier original est le fait de transmettre la table de correspondance entre séquences de bits et données décodées. Dans le cas de LRPT, le choix est fait d'exploiter une table *fixe*, standardisée après analyse de suffisamment d'images pour avoir une distribution statistiquement représentative du nombre d'apparition de chaque symboles. La norme LRPT nous explique simplement de consulter [8]... un raccourci quelque peu grossier puisque nous n'avons identifié qu'une unique page utile sur les 186 que compte la norme ! Les tables qui permettront de retrouver les arbres de décodage (pour les composantes DC – fréquence nulle – et AC – fréquences non nulles – de la transformée de Fourier de l'image) se trouve en page 158. Nous allons donc nous efforcer, grâce aux explications de `https://www.impulseadventure.com/photo/jpeg-huffman-coding.html` et `imrannazar.com/Let's-Build-a-JPEG-Decoder` (section 3), de comprendre comment lire ces tables, nommées DHT pour Define Huffman Table.

On nous explique que le nombre de code composés de N bits est

```
X'00 02 01 03 03 02 04 03 05 05 04 04 00 00 01 7D'
```

et que l'assignation de la valeur à chacun de ces codes suit la séquence

```
X'01 02 03 00 04 11 05 12 21 31 41 06 13 51 61 07' ...
```

Noter que ces séquences sont celles visibles dans `huffman.pas` de `meteor_decoder` (premières lignes du tableau `t_ac_0`). La partie implicite est comment former lesdits codes ! La première table nous informe qu'il y a 0 code de 1 bit, 2 codes de 2 bits, 1 code de 3 bits, 3 codes de 4 bits, 3 codes de 5 bits, deux codes de 6 bits... Il nous faut donc former cette séquence de bits. L'opération consiste à compter en binaire, et lorsque nous incrémentons le nombre de bits nécessaires à continuer la séquence, nous partons de la séquence précédente que nous complétons d'un 0 (en rouge ci-dessous). Cela donne en pratique
00 : premier code de deux bits
01 : second code de deux bits
100 : unique code de 3 bits
1010 : premier code de 4 bits
1011 : deuxième code de 4 bits
1100 : troisième code de 4 bits
11010 : premier code de 5 bits
11011 : deuxième code de 5 bits
11100 : troisième code de 5 bits
111010 : premier code de 6 bits
111011 : second code de 6 bits
1111000 : premier code de 7 bits
...

et la table qui suit nous informe de la valeur associée à chacun de ces codes
00 = 1
01 =2
100 =3
1010 =0
1011 =4
1100 =0x11=17
11010 =5
11011 =0x12=18
11100 =0x21=33
111010 =0x31=49
111011 = 0x41=65
1111000 = 6
1111001 = 0x13=19
...

On se rassurera en constatant que la somme des éléments de la première table vaut 162, qui est bien le nombre d'éléments dans la seconde table : il y aura donc bien une valeur pour chaque code. Bien que les valeurs soient comprises entre 0 et 255, nous constatons que 125 de ces éléments nécessiteront 16 bits pour être codés au lieu de 8. Néanmoins, comme la majorité des valeurs se regroupe autour des quelques éléments, ces valeurs codées sur 16 bits apparaissent suffisamment peu souvent pour qu'en moyenne, la compression soit rentable.

Une façon élégante, même si peu utile, pour se raccrocher aux arbres binaires bien connus du codage de Huffman, est de représenter le code tel que proposé sur la Fig. 11.

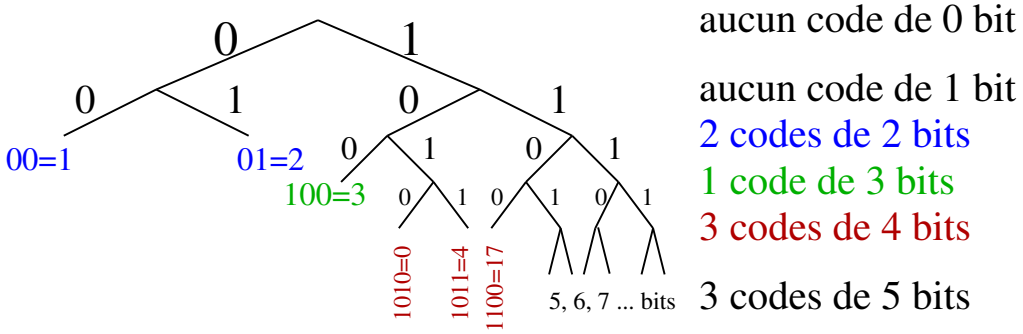

FIGURE 11 – Représentation en arbre binaire de la table du code de Huffman proposé dans [8].

Le problème est le même mais plus simple pour la composante DC qui contient beaucoup moins de valeurs :

```
X'00 01 05 01 01 01 01 01 01'
```

dont le contenu est séquentiel de 0 à 0xb=11. Ici encore nous constatons aucun code de 1 bit, 1 code de 2 bits, 5 codes de 3 bits, puis 1 code de 4 à 9 bits. Cette séquence s'écrit donc
00 = 0
010 = 1
011 = 2
100 = 3
101 = 4
110 = 5
1110 = 6
11110 = 7
111110 = 8
...

Nous voici maintenant équipés pour décoder une image JPEG. Si nous observons la séquence binaire du début d'un MCU issu des décodages précédents, nous obtenons

```
$ xxd -b jpeg.bin | head -2
00000000: 11111011 01010011 11000111 11110110 11110000 11101000  .S....
00000006: 10011111 10110011 00011111 10001101 00111110 00100000  ....>
```

qui commence par la composante DC de la transformée en cosinus discrète de l'image, suivi des 63 composantes AC (la transformée de Fourier est bijective, donc les 64 pixels de l'imagette de 8 par 8 pixels donnent 64 coefficients de Fourier, avec éventuellement un certain nombre de valeurs nulles). À chaque fois nous aurons le couple "nombre de bits" suivi de "valeur". En suivant à partir du début jusqu'à atteindre le premier code DC valable, nous constatons que nous obtenons 111110 (premier 0 atteint) soit la valeur 8 dans la seconde table qui nous venons d'exposer. Cela signifie que la valeur de la composante DC est codée sur les 8 bits qui suivent, soit 11 010100=212 en décimal. Nous verrons plus bas si cette valeur est correcte. Suit le premier des 63 coefficients AC : ici encore nous lisons la séquence de bits jusqu'à atteindre le premier code valable dans la première table exposée ci-dessus : 11 11000 est le premier code valable que nous rencontrons, qui est associé à la valeur 6. Le premier coefficient AC est donc codé par les 6 bits qui suivent, soit 111 111=63 en décimal. Continuons pour découvrir une autre subtilité de l'assignation des codes aux séquences de bits : nous poursuivons où nous nous étions arrêté la lecture des bits pour trouver 1011 qui correspond à la valeur 4 et indique que nous lisons les 4 bits qui suivent, soit 0 111 pour connaître la valeur du coefficient suivant. Et là une simple conversion binaire ne convient plus, puisque la représentation n'est pas en complément à 2 mais séquentielle, avec le codage du nombre signé composé uniquement de 0s représentant la valeur minimum de l'intervalle et le codage formé de 1s la valeur maximale. Donc 0 111 ne vaut pas +7 comme le ferait un bête convertisseur binaire vers décimal mais -8 puisque sur 4 bits nous pouvons représenter de -15 à +15 en excluant l'intervalle de -7 à +7 qui aurait été codé avec moins de bits. Ceci est résumé dans le tableau F1 de la page 89 de [8] (ou Table 5 de `www.impulseadventure.com/photo/jpeg-huffman-coding.html` que nous reproduisons ici en Tab. 1 Suit ensuite le code 100 qui vaut 3 et dont les 3 bits qui suivent valent 00 1 soit -6, et nous continuons ainsi pour décoder tous les coefficients de Fourier ou rencontrer la valeur 0 pour le nombre de bits, signifiant la fin du MCU avec tous les autres coefficients nuls.

Une fois les coefficients obtenus, ils sont réorganisés selon le motif de zigzag décrit en p.16 (Fig 5) de [8] ou explicité avec l'indice de chaque case de la matrice en p.30 (Fig. A.6), et la transformée en cosinus discrète est effectuée pour passer du domaine spectral au domaine spatial.

Longueur	Bits		Valeur	
0			0	
1	0	1	-1	1
2	00,01	10,11	-3, -2	2,3
3	000,001,010,011	100,101,110,111	-7,-6,-5.-4	4,5,6,7
4	0000,...,01111	1000,...,1111	-15,...,-8	8,...,15
5	00000,...,011111	10000,...,11111	-31,...,-16	16,...,31
6	000000,...	...,111111	-63,...,-32	32,...,64
7	0000000,...	...,1111111	-127,...,-32	32,...,64
8	00000000,...	...,11111111	-255,...,-128	128,...,255
...	...	...	...	...

TABLE 1 – Correspondance entre le nombre de bits affectés à une valeur, la séquence de bits et la valeur elle même. On notera que l'intervalle pris en charge par un nombre inférieur de bits est exclu de l'intervalle représenté par un nombre de bits donné.

toolbox de GNU/Octave – page 27 de [8]) :

Nous pouvons nous convaincre de la validité de l'analyse en modifiant légèrement `meteor_image.cc` de `github.com/infostellarinc/starcoder/blob/master/gr-starcoder/lib/meteor` pour afficher les coefficients avant réorganisation et transformée en cosinus : nous affichons le contenu (nombres à virgule flottante !) de `zdct[]` juste avant la boucle `dct[i] = zdct[ZIGZAG[i * dqt[i];` qui réorganise les coefficients selon le motif de zigzag. Ce faisant, nous obtenons

212 63 -8 -6 -27 -156 52 65 -2 10 3 -1 21 -1 -4 6 -14 0 6 2 4 1 0 2 4 5 0 16 1 -12 3 -1 0 -3 0 17 3 -2 1 0 2 7 -8 -4 3 -1 -1 1 -5 -1 0 -1 1 5 1 -1 -2 1 -1 -1 -1 1 1 0

qui commence bien par la séquence que nous avons identifiée pour se poursuivre avec les 64 coefficients. Après réorganisation selon le motif de zigzags et application du facteur d'homothétie qui a permis d'annuler un certain nombre de coefficients lors de la compression, nous obtenons un vecteur que nous nommerons `coefficients`

1484 315 -780 364 -44 108 368 28 -48 -162 390 -9 -168 0 -336 -200 -36 -12 147 0 90 78 224 -104 60 -8 60 52 -23 80 111 145 24 20 34 0 0 -50 47 35 44 0 -75 29 -37 -48 -52 -42 23 0 -72 40 0 -112 -55 46 561 126 -220 -45 52 -46 47 0

qui sont les 64 coefficients dans le domaine spectral sur lesquels s'applique la transformée en cosinus discrète (fonction `idct2` de la *signal processing*

$$\text{iDCT2}(x,y) = \frac{1}{4}\sum_{u=0}^{7}\sum_{v=0}^{7} C_u C_v S_{vu} \cos\left(\frac{(2x+1)\pi u}{16}\right)\cos\left(\frac{(2y+1)\pi v}{16}\right)$$

avec $C_0 = 1/\sqrt{2}$ et $C_{k\neq 0} = 1$ et S la matrice des coefficients de Fourier.

0	1	5	6	14	15	27	28
2	4	7	13	16	26	29	42
3	8	12	17	25	30	41	43
9	11	18	24	31	40	44	53
10	19	23	32	39	45	52	54
20	22	33	38	46	51	55	60
21	34	37	47	50	56	59	61
35	36	48	49	57	58	62	63

Figure A.6 – Zig-zag sequence of quantized DCT coefficients

Figure 12: Assignation de la position de chaque coefficient dans le codage en zigzag des coefficients maximisant d'avoir les coefficients importants en début de séquence et d'éliminer bon nombre de coefficients de fréquence élevée (vers la droite et le bas) lors de la quantification.

Après l'iDCT, le décodeur fournit la séquence de 64 valeurs que nous nommerons `image`

410 299 332 320 292 508 307 222 304 189 185 283 95 353 290 160 252 474 370 552 558 497 273 109 149 261 215 264 328 340 226 -4 417 289 641 512 644 510 245 95 282 56 407 255 466 390 160 -33 418 131 613 569 551 572 96 59 371 47 498 416 460 478 -45 78

que nous réorganisons en imagette de 8×8 pixels et vérifions qu'à la constante de 128 près ajoutée par `gr-starcoder` (tel que préconisé par la norme), nous obtenons le même résultat entre `idct2(reshape(coefficients,8,8))` et `reshape(image,8,8)` (Fig. 13).

Le facteur d'homothétie de chaque coefficient, qui est la source, par quantification, de la perte dans l'algorithme de compression JPEG, s'est déduit comme suit. Partant de la table de quantification HTK des coefficients de Fourier fournie en Table K.1 page 143 de [8] (Fig. 14), un facteur de qualité Q est fourni avec chaque MCU pour indiquer le facteur de compression de l'image. De ce facteur de qualité Q nous déduisons $F = 5000/Q$ si $20 < Q < 50$ et $F = 200 - 2 \times Q$ si $50 < Q < 100$. Ce facteur de quantification pondère la table de quantification originale par $PTK = F/100 \times HTK$ et finalement chaque coefficient transmis Sq est le résultat de l'arrondi à l'entier le plus proche des coefficients de Fourier S normalisés par la table de quantification ajustée $Sq = S/PTK$.

Table K.1 – Luminance quantization table

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

Figure 14: Table de quantification fournie dans [8, p.143]

5 Conclusion

Cette exploration du protocole LRPT des liaisons numériques entre les satellites et le sol a été l'opportunité d'appréhender l'ensemble des couches OSI, de la couche physique (fréquence de transmission) au codage (QPSK et code de convolution) aux paquets (mots) et images (phrases) contenus dans le message. Cette présentation avait pour vocation d'illustrer la puissance pédagogique de la radio logicielle : chacune des étapes de traitement fait appel à un principe physique ou mathématique incroyablement ennuyeux si appréhendé de façon purement théorique, mais qui prend tout son sens dans les étapes du jeu qui doit se conclure par le décodage d'une image.

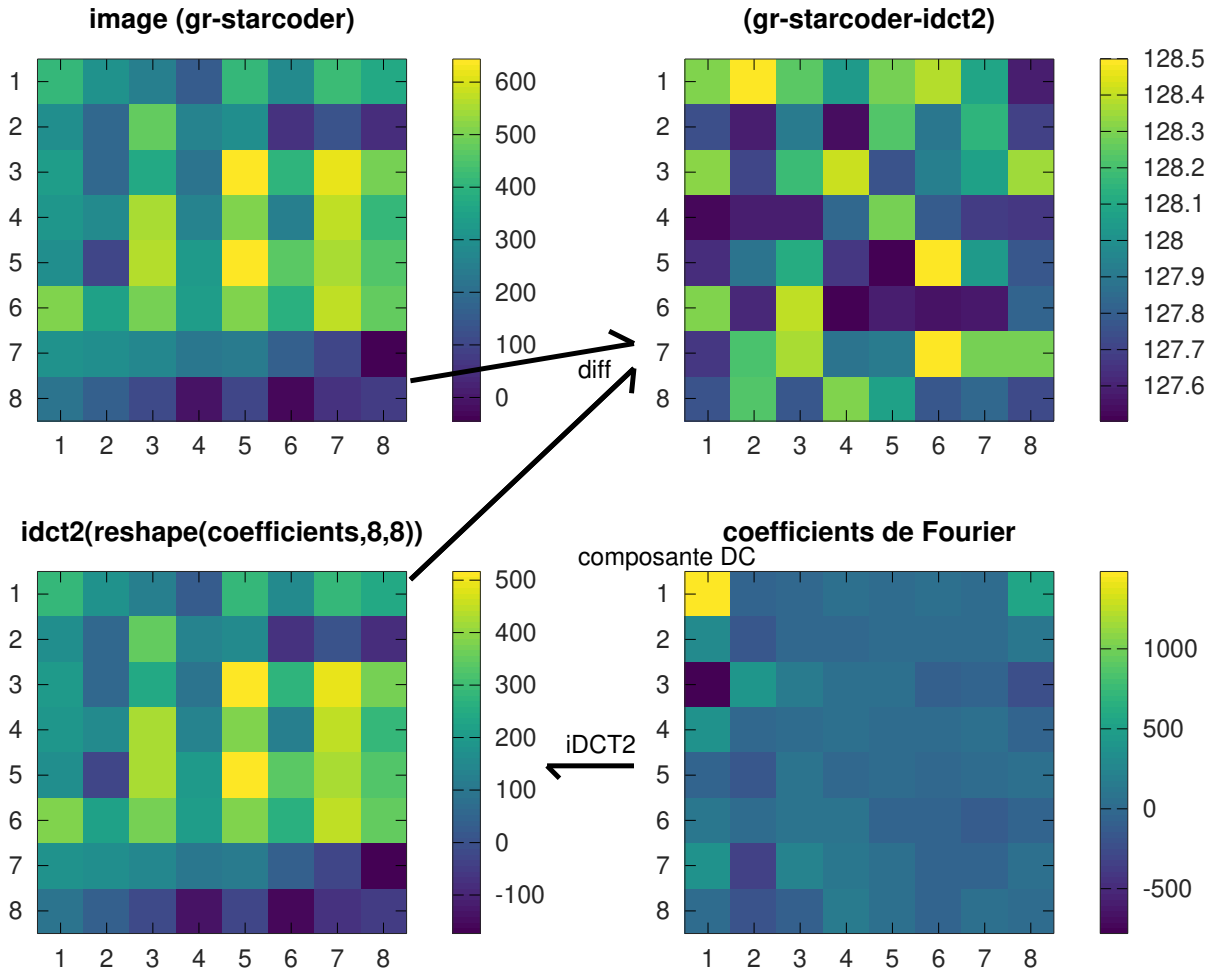

FIGURE 13 – Une fois la séquence de coefficients de Fourier obtenue, la fin du traitement est triviale : réorganisation le long du motif en zigzag, application du facteur d'homothétie définissant la qualité de l'image, puis finalement transformée en cosinus discrète inverse. Partant des coefficients de Fourier (en bas à droite, en vérifiant que la composante continue du signal est bien en coordonnées (1,1) en haut à gauche de la matrice), nous comparons (en haut à droite) le calcul effectué par `gr-starcoder` (traduction en C++ de `meteor_decoder`) et celui obtenu par transformée en cosinus discret 2D inverse (`idct2` de GNU/Octave). Nous constatons que l'erreur est à peu près constante et égale à 128 – constante ajoutée par `gr-starcoder` après calcul de l'iDCT (en haut à droite).

Ainsi, nous avons découvert par la pratique la subtilité de la modulation QPSK et les diverses solutions à l'assignation de chacun des 4 états de phase possible à une paire de bits – problème qui nous avait échappé dans l'étude d'un signal modulé en BPSK – puis mis en œuvre le décodage du code de convolution par algorithme de Viterbi. Ayant identifié la cohérence des séquences de bits issues de la déconvolution, nous avons provisoirement négligé le codage par blocs de Reed Solomon pour dépiler les diverses couches du protocole encapsulant les imagettes formant l'image finale après assemblage. Le lecteur le plus perspicace pourra conclure sur la réimplémentation du décodage JPEG que nous avons simplement implémenté ici sans en reprendre les fondements théoriques. Finalement, le décodage du code correcteur par blocs de Reed Solomon est mis en œuvre et son bon fonctionnement validé, malgré un bénéfice relativement mineur dans le cas qui nous a intéressé ici. M. Braun [10] citait au cours de son exposé au FOSDEM sur les codes correcteurs d'erreur le théorème de Shannon tel que décrit dans son article (Fig. 15) que "tout système de codage suffisamment complexe est susceptible d'atteindre le débit de transfert optimal dans un canal de communication bruité" : le concept de "suffisamment complexe" devient plus clair à l'issue de cette étude d'un mode de liaison avec un émetteur spatial, et fournit les bases pour aborder des modulations plus complexes encore telles qu'utilisées par exemple dans la télévision numérique terrestre (DVB-T).

THEOREM 2: *Let P be the average transmitter power, and suppose the noise is white thermal noise of power N in the band W. By sufficiently complicated encoding systems it is possible to transmit binary digits at a rate*

$$C = W \log_2 \frac{P+N}{N} \qquad (19)$$

with as small a frequency of errors as desired. It is not possible by any encoding method to send at a higher rate and have an arbitrarily low frequency of errors.

Figure 15: Extrait de l'article de Shannon sur le débit de communication dans un canal bruité.

Ces fondements doivent fournir les bases pour décoder multitudes d'autres télémesures spatiales, tel que démontré par la richesse du blog de D. Estévez qui met en œuvre ses compétences sur multitudes de satellites à `destevez.net/`, par exemple `destevez.`

net/2017/01/ks-1q-decoded/. Un dépôt sur github.com/jmfriedt/meteor-m2 regroupe les divers fichiers de données et scripts utilisés au cours des deux épisodes de cet article.

Remerciements

L. Teske et D. Estévez ont répondu à mes demandes de compléments d'informations par courrier électronique. M. Addouche (Univ. de Franche Comté à Besançon) m'a présenté le décodage du code de convolution par algorithme de Viterbi. Tous les ouvrages et références qui ne sont pas librement disponibles sur le web ont été obtenues auprès de Library Genesis à gen.lib.rus.ec, une ressource d'une valeur inestimable pour nos recherches.

Références

[1] A. Boissin *& al.*, *Single space segment – HRPT/LRPT direct broadcast services specification*, ESA EUMETSAT EPS/METOP MO-DS-ESA-SY-0048 rev. 5 (1998)

[2] L. Teske, *GOES Satellite Hunt* à www.teske.net.br/lucas/satcom-projects/satellite-projects/

[3] J.-M Friedt, G. Cabodevila, *Exploitation de signaux des satellites GPS reçus par récepteur de télévision numérique terrestre DVB-T*, OpenSilicium **15**, Juillet-Sept. 2015

[4] J.-M Friedt, *Radio Data System (RDS) – analyse du canal numérique transmis par les stations radio FM commerciales, introduction aux codes correcteurs d'erreur*, GNU/Linux Magazine France **204** (Mai 2017)

[5] *Structure of "Meteor-M 2" satellite data transmitted through VHF-band in direct broadcast mode*, à planet.iitp.ru/english/spacecraft/meteor_m_n2_structure_2_eng.htm

[6] W. Fong *& al.*, *Low Resolution Picture Transmission (LRPT) Demonstration System – Phase II Report, Version 1.0* (2002), à https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20020081350.pdf

[7] P. Ghivasky *& al.*, *MetOp Space to Ground Interface Specification*, doc. MO-IF-MMT-SY0001 rev. 07C, EADS/ASTRIUM/METOP (29 Mars 2004) à http://web.archive.org/web/20160616220044/http://www.meteor.robonuka.ru/wp-content/uploads/2014/08/pdf_ten_eps-metop-sp2gr.pdf. Le site www.meteor.robonuka.ru a servi d'inspiration tout au long de cette étude mais a malheureusement disparu : seule archive.org conserve trace des documents introuvables par ailleurs.

[8] *Information technology – Digital compression and coding of continuous-tone still images – requirements and guidelines*, ITU CCITT recommandation T.81 (1993) à www.w3.org/Graphics/JPEG/itu-t81.pdf

[9] J.-G. Dumas, J.-L. Roch, É. Tannier, S. Varrette, *Théorie des codes – Compression, cryptage, correction*, Dunod (2007) et en.wikipedia.org/wiki/Huffman_coding sont limpides sur la construction de l'arbre.

[10] M. Braun, *Protect your bits : Introduction to gr-fec*, FOSDEM Free Software Radio devroom (2019)

[11] C.E. Shannon, *Communication in the Presence of Noise*, Proc. IRE **37**(1), 10–21 (1949)

[12] S.T.J. Fenn, D. Taylor & M. Benaissa, *The design of Reed-Solomon codecs over the dual basis*, Microelectronics Journal **26** 383–391 (1995) et la datasheet du Mitel Semiconductor MS13544 qui implémente les codes de correction convolutifs et par bloc dans un circuit intégré compatible d'applications spatiales (microelectronics.esa.int/vhdl/doc/MS13544.pdf)

A Code correcteur d'erreurs par bloc de Reed Solomon

Le codage par convolution a pour vocation de compenser du bruit distribué sur des bits du fait de la liaison radiofréquence bruitée entre l'émetteur et le récepteur, et fait l'hypothèse d'un bruit uniforme aléatoire qui peut impacter chaque bit indépendamment de ses voisins. Il ne saurait cependant corriger des blocs de données corrompues par un interférent ponctuel : ce type d'erreur est pris en charge par la correction par bloc, par exemple de Reed-Solomon. Ce code s'apparente à la correction par blocs que nous avions déjà vu dans RDS avec le codage BCH [4]. Ici, chaque paquet de 255 octets est formé de 223 octets utiles et 32 octets de code correcteur, qui permettent d'identifier une erreur de transmission et d'en corriger une partie. Ce type de code correcteur s'appelle donc RS(255,223) puisque sur 255 octets transmis, 223 sont des données et donc les 32 derniers sont le code correcteur d'erreur. libfec fournit la bibliothèque nécessaire à la correction d'erreurs par RS(255,223), tel que décrit dans [2]. Comme le but de la correction par bloc est de corriger une série de bits erronés, il est judicieux de distribuer l'information le long de la trame transmise afin de minimiser l'impact de l'interférence qui affecte plusieurs bits adjacents. Ainsi, au lieu de diviser la trame de 1020 octets de long en 4 trames adjacentes de 255 octets, le choix est fait d'interlacer les 4 séquences de données et leur code correcteur d'erreur tel que illustré en Fig. 16, avec le code correcteur d'erreur des 4 séquences placé en fin de trame.

Nous pouvons nous entraîner dans un premier temps pour comprendre l'implémentation de Reed Solomon dans libfec :

```
1 #include <fec.h>    // gcc -o jmf_rs jmf_rs.c -I./libfec ./libfec/libfec.a
2
3 int main()
4 {int j;
```

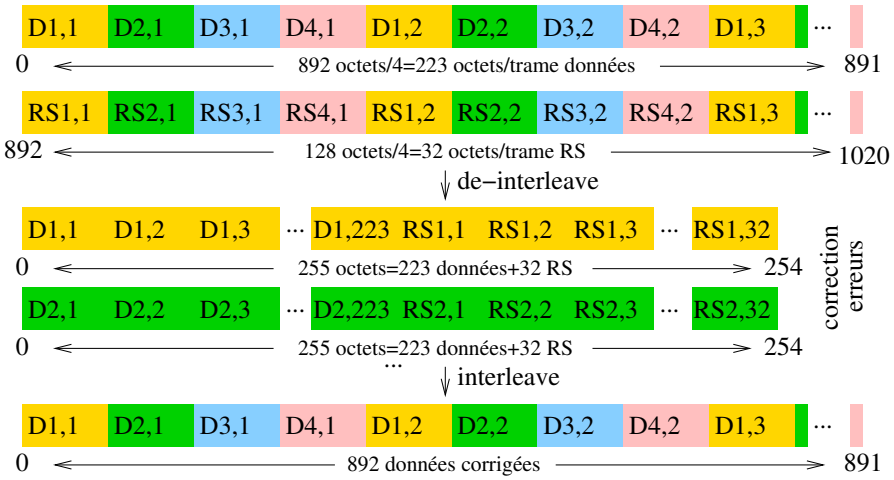

FIGURE 16 – Organisation des données le long d'une trâme CVCDU de 1020 octets de long (nous avons déjà éliminé le mot de synchronisation de 4 octets en entête). Les 892 premiers octets contiennent les données `D` qui seront considérées comme interlacées pour les 4 séquences de 223 octets corrigeables par Reed Solomon RS(255,232), et les derniers 128 octets contiennent, toujours de façon interlacée, les 4 séquences de 32 octets de codes de correction `RS`. Appliquer la correction nécessite donc de dé-interlacer les données (haut → milieu), appliquer Reed Solomon pour identifier et corriger les erreurs (milieu), et re-interlacer les données (milieu→bas) pour les replacer dans leur ordre d'origine, mais après correction des octets potentiellement corrompus lors de la liaison radiofréquence.

```
uint8_t rsBuffer[255];

uint8_t tmppar[32];
uint8_t tmpdat[223];

for (j=0;j<255; j++) rsBuffer[j]=(rand()&0xff);  // received data
for (j=0;j<223;j++)  tmpdat[j]=rsBuffer[j];      // backup data
encode_rs_ccsds(tmpdat,tmppar,0);                // create RS code
for (j=223;j<255;j++) rsBuffer[j]=tmppar[j-223]; // append RS after data
rsBuffer[42]=42; tmpdat[42]=42;                  // introduce errors
rsBuffer[43]=42; tmpdat[43]=42;                  // ... on purpose
rsBuffer[44]=42; tmpdat[44]=42;    // ... to check correction capability
rsBuffer[240]=42;tmppar[240-223]=42;
printf("RS:%d\n",decode_rs_ccsds(rsBuffer, NULL, 0, 0)); // check that RS can correct
for (j=0;j<223;j++)
  if (rsBuffer[j]!=tmpdat[j]) {printf("%d:␣%hhd␣->␣%hhd␣;␣",j,tmpdat[j],rsBuffer[j]);}
for (j=223;j<255;j++)
  if (rsBuffer[j]!=tmppar[j-223]) {printf("%d:␣%hhd␣->␣%hhd␣;␣",j,tmppar[j-223],rsBuffer[j]);}
}
```

dans cet exemple nous créons des données (aléatoires) que nous encodons, puis nous modifions 4 données de la charge utile et 1 donnée du code correcteur, et testons la capacité de correction du décodeur. Le résultat,

```
RS:4
42: 42 -> 5 ; 43: 42 -> 23 ; 44: 42 -> 88 ; 240: 42 -> 95
```

est conforme à nos attentes : 4 erreurs sont identifiées et corrigées.

L'application du programme d'exemple sur les 128 octets de fin de trame chargés de corriger les 892 octets du début de trame ... ne fonctionne pas du tout ! Encore une astuce indiquée par Lucas Teske que nous n'avons pas trouvé dans les documentations : il s'agit d'un code correcteur *dual basis Reed Solomon* dans lequel les octets sont encore une fois passés dans une table de transposition tel que décrit dans `github.com/opensatelliteproject/libsathelper/blob/master/src/reedsolomon.cpp`. L'implémentation du code correcteur d'erreur pas blocs dans cette nouvelle base est justifiée dans [12, Tab.4] comme un gain signficiatif (66%) de ressources par rapport à une implémentation sur les données brutes. Une fois cette transposition effectuée, le code correcteur fonctionne, selon l'exemple ci-dessous qui inclut toute la séquence de décodage, à savoir algorithme de Viterbi, application (XOR) du polynome pour retirer la distribution aléatoire des données, regroupement des paquets de données avec leur code correcteur de Reed Solomon, application du polynome de transposition, correction d'erreur, retrait du polynome de transposition pour finalement obtenir les données corrigées :

```
#include <fec.h> // gcc -o demo_rs demo_rs.c -I./libfec ./libfec/libfec.a

```

```
// github.com/opensatelliteproject/libsathelper/blob/master/src/reedsolomon.cpp
// dual basis Reed Solomon !
#include "dual_basis.h"

unsigned char pn[255] ={           // randomization polynomial
    0xff, 0x48, 0x0e, 0xc0, 0x9a, 0x0d, 0x70, 0xbc, \
    0x8e, 0x2c, 0x93, 0xad, 0xa7, 0xb7, 0x46, 0xce, \
[...]
    0x08, 0x78, 0xc4, 0x4a, 0x66, 0xf5, 0x58  };

#define MAXBYTES (1024)

#define VITPOLYA 0x4F
#define VITPOLYB 0x6D

#define RSBLOCKS 4

#define PARITY_OFFSET 892

void interleaveRS(uint8_t *idata, uint8_t *outbuff, uint8_t pos, uint8_t I) {
  for (int i=0; i<223; i++) outbuff[i*I+pos]=idata[i];
}

int viterbiPolynomial[2] = {VITPOLYA, VITPOLYB};

int main(int argc,char *argv[]){
 int res,i,j,framebits,fdi,fdo;
 unsigned char data[MAXBYTES],symbols[8*2*(MAXBYTES+6)]; // *8 for bytes->bits & *2 Viterbi
 void *vp;
 int derrors[4] = { 0, 0, 0, 0 };
 uint8_t rsBuffer[255],*tmp;
 uint8_t rsCorData[1020];

 fdi=open("./extrait.s",O_RDONLY);
 fdo=open("./sortie.bin",O_WRONLY|O_CREAT,S_IRWXU|S_IRWXG|S_IRWXO);
 read(fdi,symbols,4756+8);  // offset
 framebits = MAXBYTES*8;

 do {
  res=read(fdi,symbols,framebits*2+50);  // 50 additional bytes to finish viterbi decoding
  lseek(fdi,-50,SEEK_CUR);               // go back 50 bytes
  for (i=1;i<2*framebits;i+=2) symbols[i]=-symbols[i]; // I/Q constellation rotation
  set_viterbi27_polynomial(viterbiPolynomial);
  vp=create_viterbi27(framebits);        // convolution -> Viterbi
  init_viterbi27(vp,0);
  update_viterbi27_blk(vp,symbols,framebits+6);
  chainback_viterbi27(vp,data,framebits,0);
  tmp=&data[4];                          // rm synchronization header
  for (i=0;i<1020; i++) tmp[i]^=pn[i%255];  // XOR decode (dual basis)

  for (i=0; i<RSBLOCKS; i++)
      { for (j=0;j<255; j++) rsBuffer[j]=tmp[j*4+i];        // deinterleave
        for (j=0;j<255; j++) rsBuffer[j]=ToDualBasis[rsBuffer[j]];
        derrors[i] = decode_rs_ccsds(rsBuffer, NULL, 0, 0); // decode RS
        for (j=0;j<255; j++) rsBuffer[j]=FromDualBasis[rsBuffer[j]];
        interleaveRS(rsBuffer, rsCorData, i, RSBLOCKS); // interleave
        printf(":%d",derrors[i]);
      }
  write(fdo,data,4);                      // header
  write(fdo,rsCorData,MAXBYTES-4);  // corrected frame
 } while (res==(2*framebits+50));
 close(fdi);
 close(fdo);
 exit(0);
}
```

Ce code est donc l'apothéose de toute la séquence de conversion des phases issues des coefficients I/Q vers les bits prêts à être

assemblés pour retrouver les trames puis les images JPEG, en tenant compte des deux codes correcteurs d'erreur, par convolution et par bloc. Le résultat de cette correction additionnelle est illustrée en Fig. 17 et démontre comment l'ajout du code de correction par bloc permet d'étendre la plage d'analyse des images JPEG alors que le satellite s'approche de l'horizon.

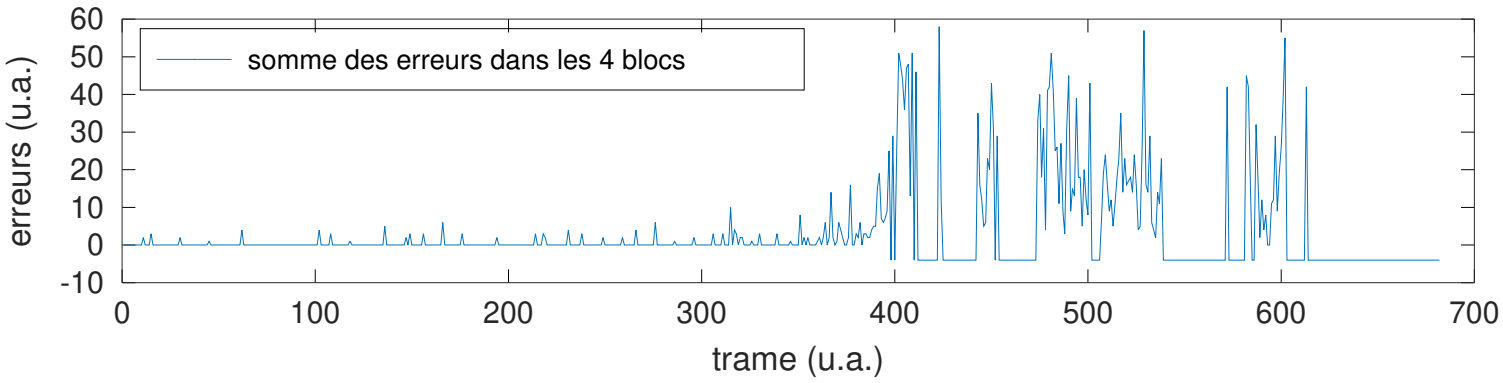

FIGURE 17 – Résultat de la correction d'erreurs au cours du passage du satellite. Nous constatons clairement que la correction par blocs devient d'autant plus efficace que le satellite s'abaisse sur l'horizon et que le bilan de liaison se dégrade. -1 indique qu'il y avait trop d'erreurs dans la réception pour permettre une correction des octets erronés.

On notera qu'un certain nombre de codes se sont vus tronquer, pas soucis de compacité, des fichiers d'entête annonçant des bibliothèques standard et d'entrée sortie du C : le lecteur n'aura aucun mal à retrouver ces fichiers d'entête (`stdio.h`, `stdlib.h`, `unistd.h` ...) permettant d'ouvrir, lire, écrire ou fermer des fichiers ainsi que communiquer avec la console.