La réception radiofréquence définie par logiciel (*Software Defined Radio* – SDR)

J.-M Friedt, G. Goavec-Mérou,

13 septembre 2012

La diffusion à large échelle de la communication radiofréquence dans les appareils grand public induit la disponibilité de composants aux performances compatibles avec ces modes de communications (fonctionnement de 50 à 2500 MHz ou plus, bandes passantes de l'ordre de la dizaine de MHz) et pour des prix ridiculement faibles grâce à la production de masse. Nous allons profiter de cette mode en présentant l'utilisation d'un récepteur de télévision numérique – DVB basé sur le composant Elonics E4000 – dont l'étage de réception s'avère tellement simple qu'il est compatible avec de nombreux modes de transmission analogique et numérique de données par liaison radiofréquence. Le flux de données brut issu du récepteur radio est traité par des blocs implémentant les fonctionnalités les plus classiques dans le logiciel `gnuradio` et son interface graphique `gnuradio-companion`. Nous verrons comment, au delà de la simple utilisation des blocs de traitement disponibles, implémenter nos propres algorithmes de décodage sur le flux d'informations reçu du récepteur radiofréquence. Nous nous intéresserons en particulier au *packet*, au protocole ACARS, et aux radiomodems encodant l'information en FSK et AFSK.

1 Introduction à la radio définie par logiciel (SDR)

La radio définie par logiciel – *Software Defined Radio* (SDR) – est une approche au traitement de données radiofréquences déportant un maximum de traitement sur du logiciel au lieu de dépendre du matériel [1, 2, 3, 4].

Compte tenu des fréquences et du débit d'informations mis en jeu lors des transmissions sans fil radiofréquences, il a été classiquement nécessaire de traiter un maximum d'informations par du matériel dédié, pour ne finalement qu'éventuellement traiter par logiciel le flux d'informations résultant, généralement dans les fréquences audibles (<100 kHz). Mentionnons parmi ces exemples les habituels récepteurs radiofréquences FM et AM qui ne font appel à aucun logiciel, aux récepteurs amateurs de mode numérique basés sur un modem audiofréquences (*packet*[1]), transmission d'images (SSTV par le vénérable JVFAX sous MS-DOS, des outils plus récents étant probablement disponibles aujourd'hui sous GNU/Linux) ou la réception d'images satellites que nous avions déjà présenté dans ces pages [5].

Classiquement, le traitement logiciel d'informations transmises sans fil se cantonne aux fréquences audio du fait de la disponibilité d'une carte d'acquisition largement disponible – la carte son équipant pratiquement tous les ordinateurs personnels depuis une dizaine d'années – et la puissance de calcul requise qui n'est pas négligeable, même pour traiter un flux de données aussi "lent" que 100 kHz. En effet, le débit du traitement de l'information est imposé par le flux entrant, et toute interruption (par exemple pour gérer une autre tâche dévolue au système d'exploitation) se traduit par une perte irrémédiable de données. Aucun des systèmes d'exploitation grand public n'ayant de prétention de latence bornée (temps réel), et surtout pas GNU/Linux dans sa version de base, la disponibilité d'une puissance de calcul sur-dimensionnée est la seule garantie d'un traitement sans pertes d'informations.

La limitation sur les interfaces d'acquisitions a en partie changé avec l'avènement des cartes d'acquisition vidéo : capables d'échantillonner plusieurs millions de mesures par seconde, le flux de données devient compatible avec l'analyse par logiciel d'une bande de fréquences de plusieurs MHz de large. Cependant, l'étage le plus amont de traitement de l'information radiofréquence – amplificateur et transposition vers une fréquence acceptable par un convertisseur analogique-numérique – reste matériel. Récemment, un récepteur de télévision numérique terrestre sous forme de clé USB a induit une explosion d'activité dans le domaine de la SDR libre lorsqu'il s'est avéré que le flux de données fourni au logiciel de décodage

1. `http://www.baycom.org/~tom/ham/linux/multimon.html` décode un certain nombre de modes numériques à partir des signaux sonores issus d'un récepteur radio et numérisés par carte son

vidéo était généraliste (nous développerons plus loin le flux de données I/Q fourni par cette interface matérielle) : pourquoi donc se limiter aux bandes de télévision et ne pas sonder n'importe quelle bande de fréquence accessible par l'étage radiofréquence, et ce en vue d'en interpréter les informations par traitement logiciel de l'information ?

Dans ce but, un environnement logiciel sympathique est `gnuradio` [2], qui fournit un certain nombre de blocs de traitements de base qui permettent de rapidement appréhender le problème.

La séquence d'analyse que nous proposons commence par une brève présentation du matériel utilisé, avant d'exploiter directement quelques blocs de traitement du signal disponibles sous `gnuradio` en s'appuyant pour la construction des applications sur son interface graphique, `gnuradio-companion` [3]. L'utilisation de blocs existants n'étant d'à peu près aucun intérêt technique pour le développeur, autre que pour un aspect ludique, nous nous attellerons ensuite à la tâche de traiter un flux de données inconnu, à savoir le flux de données issu d'un radiomodem d'une part, et d'autre part le flux de données émis par les avions suivant le protocole ACARS. Finalement, nous verrons comment remplacer le post-traitement des informations acquises pour en extraire les données numériques, par un traitement en temps réel sous forme de bloc compatible avec `gnuradio-companion`.

2 Présentation du matériel

Historiquement, le développement de SDR sur plateforme libre s'est organisé autour des systèmes USRP (*Universal Software Radio Peripheral*) développés par Ettus Research [4]. Bien que de coût raisonnable en comparaison des systèmes professionnels proposés, l'investissement de l'ordre du millier d'euros reste néanmoins cantonné à l'amateur motivé qui a besoin des performances de tels dispositifs dédiés (bande passante importante, sensibilité). Les USRP sont supportées dans `gnuradio` au travers des pilotes `gr-uhd` (UHD signifiant *USRP Hardware Driver*). Ce point est utile à mentionner car tous les outils faisant référence à uhd nous seront *inutiles* dans la suite de cette discussion.

En effet, alors que les USRP ont pu voir le jour grâce à la chute vertigineuse des coûts de composants radiofréquences développés notamment pour les applications de communication numérique sans fil (synthétiseurs de fréquence, modulateur et démodulateur I/Q sur des bandes allant de quelques MHz aux GHz, notamment par Analog Devices, Maxim ou Semtech), cette tendance s'accélère aujourd'hui avec l'avènement de circuits compacts regroupant tous les éléments radiofréquences analogiques (*front end*) habituellement séparés en éléments discrets.

En particulier, un nouveau composant, le Elonics E4000 (récepteur radiofréquence direct − sans exploitation d'une fréquence intermédiaire − fonctionnant de 64 à 1700 MHz) a été la source d'une explosion de circuits à très faible coût pour la SDR. Ce composant se limite en effet à amplifier le signal radiofréquence reçu par une antenne, le mélanger avec un oscillateur local pour transposer la fréquence [5] suivi de filtres passe-bas pour ne conserver que la composante basse-fréquence. Ce composant *analogique* fournit donc les composantes I et Q (identité et en quadrature) sur une bande passante allant jusqu'à 8 MHz.

Ce signal analogique doit ensuite être numérisé pour traitement par un ordinateur personnel. Un premier projet exploite un convertisseur analogique-numérique de carte audio : Funcube [6]. Les faibles volumes de production induisent un coût de l'ordre de la centaine d'euros, pour un dispositif à la sensibilité acceptable pour de la réception de sources en orbite terrestre (satellites), mais avec une bande passante réduite aux performances d'une carte son, donc une centaine de kHz.

Afin de palier à cette carence sur la bande passante, un projet introduisant un FPGA pour accélérer la fréquence d'échantillonnage a été engagé sous le nom de OsmoSDR [7], avec la participation de développeurs aussi célèbres que Harald Welte [8], garantie de sérieux du projet. Les développements logiciels autour de ce projet ont finalement profité de la commercialisation de récepteurs de télévision numérique terrestre de qualité médiocre mais dont les volumes de production induisent des coûts d'achat imbattables. En

2. `gnuradio.org`
3. `http://www.joshknows.com/grc`
4. `http://www.ettus.com/`
5. nous avions déjà exposé dans ces pages le principe du mélangeur qui transpose une fréquence incidente f et une fréquence de référence f_0 pour générer $f \pm f_0$, la composante somme des fréquences étant éliminée par un filtre passe-bas)
6. `http://tetra.osmocom.org/trac/wiki/Funcube_Dongle` et `http://www.funcubedongle.com/`
7. `sdr.osmocom.org`
8. `https://har2009.org/program/speakers/253.en.html`

effet, le portage à V4L de ces clés USB de réception de télévision a fait apparaître que tout le décodage du flux de données se fait de façon logiciel (un triste rappel des modems d'il y a une quinzaine d'année, aussi nommés winmodems, qui avaient engagé cette tendance de réduction du matériel en transposant le décodage au logiciel – modems inutilisables dans un environnement libre en l'absence des pilotes appropriés [9]) et que ce flux de données I/Q serait approprié pour des applications bien plus intéressantes que la visualisation d'émissions de télévision.

Il est donc désormais possible de se lancer dans le domaine fantastique de la SDR pour un investissement matériel inférieur à 20 euros : nous avons pour notre part expérimenté avec des clés EZCAP (DVB-T/DAB/FM) telles que (par exemple) disponible à `http://www.dealextreme.com/p/mini-dvb-t-digital-tv-usb-2-0-dongle-with-fm-dab-remote-controller-92096` ou `https://www.cosycave.co.uk/product.php?id_product=104` pour les sources que nous avons testé (pourquoi un vendeur de plantes prétendues thérapeutiques vend des récepteurs de télévision ? mystère, l'appât du gain n'a pas de limites).

Le projet exploitant le convertisseur analogique-numérique embarqué sur ces clés, le Realtek RTL2832U, est issu de OsmoSDR et décrit à `http://sdr.osmocom.org/trac/wiki/rtl-sdr`.

2.1 Installation des outils gnuradio

L'environnement gnuradio est dynamique et en particulier les outils autour des clés USB pour réception de télévision numérique change rapidement. Il est donc de bon ton de recompiler ses outils depuis les dernières versions des sources. Ce processus a été automatisé dans un script fort utile, disponible à `http://www.sbrac.org/files/build-gnuradio`. Il est cependant significatif de réaliser un aspect quelque peu surprenant de la compilation de gnuradio : au lieu d'imposer les dépendances nécessaires à un ensemble de fonctionnalités, l'inclusion ou l'exclusion de morceaux de gnuradio est induite par une recherche des bibliothèques disponibles. Ainsi, une première compilation se traduit par l'exclusion d'à peu près toutes les fonctionnalités de gnuradio, et ce n'est qu'en installant petit à petit les bibliothèques nécessaires (`libfftw3-dev`, modules Python) que nous finirons par atteindre l'objectif de compiler tous les modules, sauf uhd qui ne sera pas nécessaire. En particulier, il est fondamental de compiler la partie `gnuradio-companion`, une interface graphique pour accéder aux modules de traitement du signal, dont nous ferons intensivement usage dans la suite de ce document.

2.2 Utilisation en oscilloscope du RTL2832U

Ayant installé les outils gnuradio et en particulier `gnuradio-companion`, nous pouvons prendre en main une clé USB de réception de télévision numérique terrestre EZCAP. Le premier constat est que le connecteur de type antenne de télévision 75 Ω est à peu près inexploitable avec des antennes de radioamateur ou même pour nos propres expérimentations d'antennes dédiées à diverses bandes de fréquences, donc nous commençons par le remplacer par une embase BNC.

Afin de bien maîtriser la distinction entre les fonctions du composant analogique E4000 et du convertisseur analogique-numérique suivi de communication numérique RTL2832U, nous allons injecter un signal sinusoïdal sur ce qui apparaît à l'évidence sur la carte électronique comme les signaux balancés des composantes I et Q (*i.e.* I+ et I- ainsi que Q+ et Q-). Ce faisant, nous générons le graphique de traitement du signal gnuradio le plus simple possible : une source de données sous la forme du bloc `OsmoSDR Source` (accessible dans le menu des fonctions `OsmoSDR` à droite de gnuradio-companion) – l'absence de ce menu indique que la compilation des outils ne s'est pas faite convenablement – et un puits de données sous la forme d'un oscilloscope accessible dans le menu `WX GUI Widget` (bloc `WX GUI Scope Sink`).

Nous constatons sur la Fig. 1 que les voies I et Q présentent bien un signal sinusoïdal, en accord avec la forme du signal injecté, à une fréquence de 200 kHz et échantillonné à 2 MS/s (10 points/période). L'horloge interne au convertisseur EZCAP et le quartz cadençant le synthétiseur de fréquence sont synchronisés à mieux que 2 parties par million (2 ppm) puisque nous avons dû programmer la fréquence de 199999,720 Hz pour que l'échantillonnage fasse exactement 10 points/période à 2 MS/s (en l'absence de déclenchement sur un front, cette condition s'observe visuellement sur la sortie oscilloscope de `gnuradio` lorsque la sinusoïde arrête de se déplacer horizontalement à une vitesse égale à l'inverse de la différence des fréquences), un résultat impressionnant compte tenu de la simplicité du montage n'exploitant qu'un quartz de qualité commerciale pour cadencer l'électronique numérique.

9. `http://en.wikipedia.org/wiki/Softmodem`

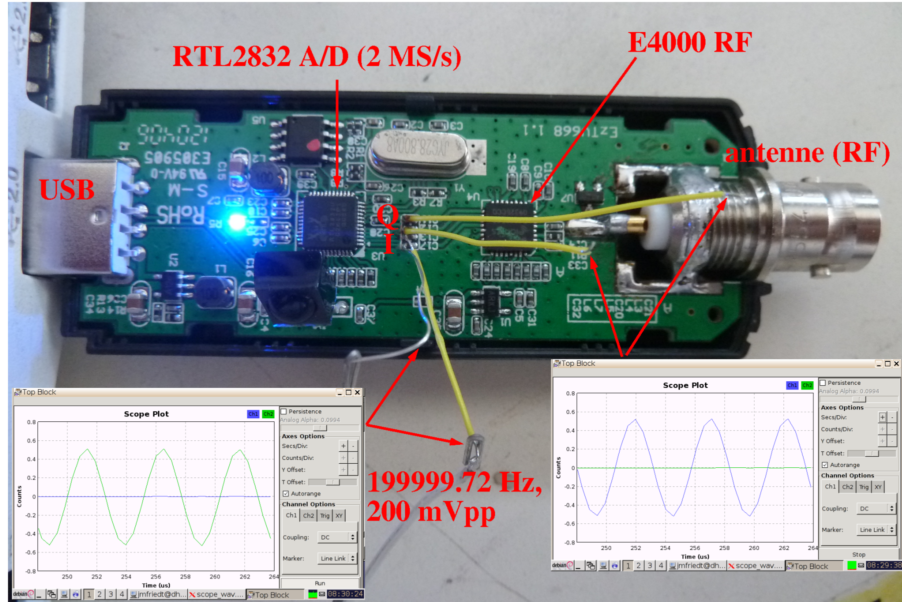


Figure 1 – Utilisation d'un récepteur de télévision numérique terrestre comme oscilloscope avec une bande passante de plus de 2 MHz.

Nous avons mentionné un gain d'un facteur de l'ordre de $2,5 \times 10^6/48000 \simeq 50$ sur la fréquence d'échantillonnage en exploitant le RTL2832U comme carte d'acquisition à la place d'une carte son. Cependant, ce chiffre ne doit pas cacher la dynamique médiocre du convertisseur analogique-numérique de ce composant qui ne code l'information que sur 8 bits. Une dynamique de $20 \times \log_{10}(2^8) = 48$ dB est médiocre compte tenu de la dynamique des signaux radiofréquences acquis qui atteint souvent 60 dB. Un étage de préamplification capable de s'adapter à la puissance radiofréquence reçue est donc souhaitable (ajustement automatique par le E4000 ou de façon logicielle en ajustant manuellement le gain de ce composant). Le sur-échantillonnage d'un facteur 50 ne compense pas la médiocrité de la résolution des convertisseurs par rapport à ceux d'une carte son. En effet, il est bien connu [6] qu'en appliquant une moyenne glissante sur N points acquis, l'écart type sur ces mesures (supposées poluées par un bruit uniforme) décroît comme $\sqrt{N}$. Ainsi, il faut quadrupler la fréquence d'échantillonnage pour diviser par 2 le bruit sur le signal acquis et donc ajouter un bit sur la résolution du convertisseur analogique-numérique. La ref. [6] formalise ce point en notant que pour une fréquence d'échantillonnage f_{se}, le gain en résolution (p bits) par application d'une moyenne glissante par rapport aux acquisitions à fréquence f_e est

$$f_{se} = 4^p \times f_e$$

ou, en passant en échelle logarithmique, $10\log_{10}(f_{se}/f_e) = 10\log_{10}(4)\times p$ soit, en notant que $10\log_{10}(4) \simeq 6$, nous retrouvons [3] que le gain en résolution en bits p pour un sur-échantillonnage de f_{se}/f_e est $1/6 \times 10\log_{10}(f_{se}/f_e)$. Dans tous les cas, nous sommes loin d'atteindre les 16 bits de résolution d'une carte son, puisque $10\log_{10}(50)/6 \simeq 3$ soit un total de 11 bits sur le convertisseur équivalent fonctionnant à la même fréquence que la carte son.

Ce rapide petit calcul nous incite à nous rappeler que la fréquence d'échantillonnage n'est pas nécessairement la caractéristique principale d'un convertisseur analogique-numérique, et dans ce cas les plus de 2 Méchantillons/s ne se justifient que pour analyser un signal de bande passante importante, mais de dynamique réduite. Si la bande passante ne se justifie pas, une carte son sera plus efficace pour échantillonner des signaux avec une dynamique accrue.

Tous ces traitements se font sur un eeePC701 et ne nécessitent donc pas des ressources énormes en terme de puissance de calcul ou de mémoire.

3 Premiers pas : utilisation avec gnuradio-companion

Nos essais de réception radiofréquence ont pour ambition de recevoir des signaux plus faibles que les émissions commerciales sur bande FM. En particulier, nous nous intéresserons à la réception d'images émises par des satellites sur des fréquences autour de 137 MHz. Il s'avère que la majorité des fréquences qui vont nous intéresser dans ce document se situent autour de cette valeur – 88 à 108 MHz pour la FM commerciale, 108 à 137 MHz pour la bande aérienne, 154 MHz pour le *packet radio* – et nous exploiterons donc dans tous ces cas une antenne dipôle ajustée pour 137 MHz. Afin de maintenir la portabilité d'un récepteur radiofréquence se branchant sur port USB, l'antenne se doit d'être démontable et tenir dans un sac à dos. Pour ce faire, un tube de cuivre creux de 8 mm de diamètre est coupé en deux morceaux de 55 cm de longueur ($300/137/4 \simeq 0,55$ avec 300 m/μs la célérité d'une onde électromagnétique et 137 la fréquence d'intérêt en MHz) et munis, à chaque extrémité, de fiches banane de 4 mm de diamètre. Ce montage robuste s'assemble facilement sur un support en plastique muni de deux embases 4 mm femelles auxquelles est soudé un bout – aussi court que possible – de cable coaxial RG58 muni à une extrémité d'une fiche BNC (Fig. 2). Exceptionnellement, nous ajouterons un amplificateur faible bruit Hittite HMC478 (alimentation en 5 V, facteur de bruit 2 dB) entre l'antenne et le récepteur EZCAP, bien que le gain d'un tel montage reste à démontrer. Le pré-amplificateur faible bruit (*LNA*) est alimenté en 5 V par un second port USB.

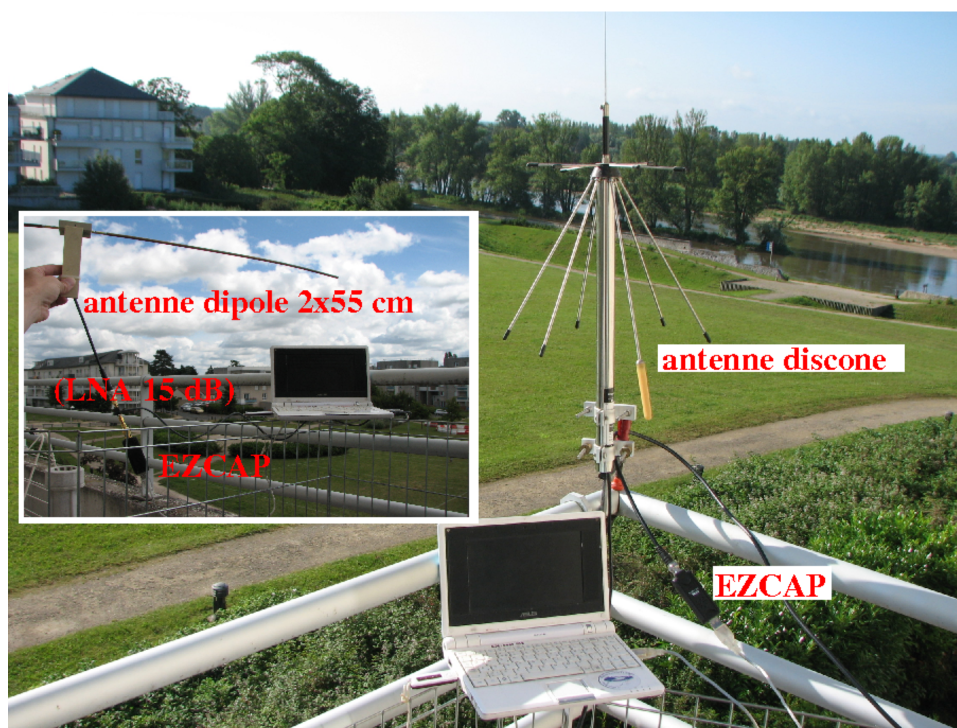


Figure 2 – Deux montages d'antennes exploités avec le récepteur EZCAP : une antenne dipôle facilement démontable pour les bandes de fréquences autour de 137 MHz, et sinon une antenne discone large bande. Aussi importante que la nature de l'antenne, l'emplacement sélectionné doit être aussi libre que possible d'obstacles. Ici, un balcon au 3ème étage d'un batiment orienté vers l'est est acceptable pour la réception FM et d'avions, mais insuffisamment dégagé pour la réception d'images issues de satellites.

Dans le cas particulier de l'ADS-B qui occupe une bande de fréquences autour de 1090 MHz, une antenne discone large bande a été exploitée (Diamond Antenna D190, annoncée comme fonctionnelle entre 100 et 1500 MHz) (Fig. 2).

3.1 La bande FM commerciale

Le test le plus simple car permettant de recevoir un signal émis en continu avec une forte puissance est la bande FM commerciale, entre 88 et 108 MHz. Cette bande de fréquences est qualifiée de large bande (*Wide FM*) du fait de l'encombrement spectral important occupé par chaque station (180 kHz). Les canaux alloués aux radio sont donc séparés d'environ 200 kHz [9, p.138].

Ce test fera office de validation du bon fonctionnement de l'installation : interface graphique `gnuradio-companion`, source OsmoSDR (comme utilisée auparavant pour l'exemple de l'oscilloscope), démodulation par le bloc WFM et puits de données sous forme de la carte son d'une part (après démodulation) et analyse du spectre acquis (avant démodulation) pour identifier la fréquence d'émission des stations d'autre part.

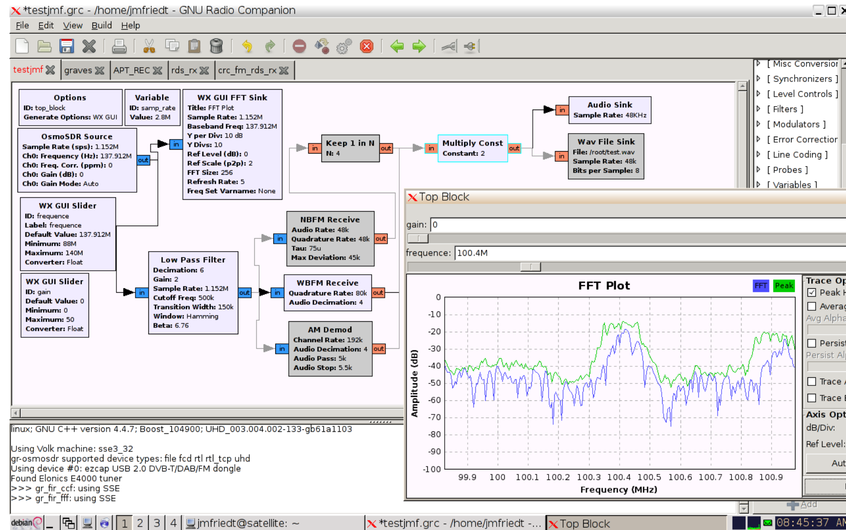


FIGURE 3 – Exemple de chaînes de traitement pour le décodage de la bande FM commerciale (88-108 MHz) ou la bande d'aviation où les communications se font en AM (au dessus de la bande FM commerciale, 108-137 MHz). Le mode de démodulation FM étroite n'a pas fourni de résultat significativement différent du démodulateur WFM, mais est inclus pour illustrer la décimation en sortie de ce démodulateur par une fonction qui ne garde que 1 échantillon sur 4 (et donc fournit le flux de données au débit approprié à la carte son). Le spectre, en bas à droite, du signal acquis, présente une extension de $-f_e/2$ à $f_e/2$ avec $f_e = 1152$ kHz la fréquence d'échantillonnage : nous y trouvons évidemment un pic autour de la fréquence de la chaîne radio écoutée (100,4 MHz) ainsi qu'un pic autour de la fréquence adjacente (100,9 MHz), qui n'interfère néanmoins pas avec la démodulation grâce au filtre passe-bas situé après l'analyseur de spectres.

Cet exemple fournit l'opportunité d'introduire une première contrainte sur l'assemblage des blocs de traitement : il nous faut garantir la continuité du flux de données le long de la chaîne de traitement, et donc ne pas fournir trop de données à un bloc qui ne peut tout traiter, ni faire attendre un bloc qui n'aura plus rien à faire. Ainsi, dans l'exemple de la Fig. 3, si nous partons de la fin de la chaîne de traitement (le puits qu'est la carte son du PC), nous *devons* fournir un flux continu de données à la fréquence d'échantillonnage choisie à 48 kHz. Nous constatons que le bloc de démodulation FM large bande (WFM) effectue une décimation d'un facteur 4 (*i.e.* sort 4 fois moins de données qu'il en reçoit), donc le flux de données en entrée de ce bloc doit être $4 \times 48 = 192$ kHz. Le filtre passe bas effectue quant à lui une décimation d'un facteur 6, donc nécessite un flux en entrée de $192 \times 6 =$1152 kHz, qui est bien la valeur que nous observons dans le *sampling rate* de ce bloc. La source OsmoSDR étant directement connectée au filtre passe bas, sa fréquence d'échantillonnage est elle aussi sélectionnée à 1152 kS/s. On notera qu'une limitation de la source OsmoSDR est de ne *pas* pouvoir descendre sous quelques centaines de kHz de fréquence d'échantillonnage : on conservera donc habituellement une source échantillonnant entre 1 et 2,5 MS/s, quitte à décimer par un filtre passe bas si la suite des étapes ne nécessitent que des traitements sur des bandes passantes réduites. Le lecteur est encouragé à changer les fréquences d'échantillonnage ou facteurs de décimations sur un tel graphique pour appréhender l'effet d'erreurs sur

ces valeurs.

Une extension proposée dans le cadre de `gnuradio-companion` est le décodage de l'information RDS (*Radio Data System*) [10] qui fournit une information sous forme numérique sur la station écoutée (nom de la station, fréquences correspondantes aux alentours, nature du programme transmis, éventuellement information de trafic ...). Cet exemple est intéressant car il illustre la souplesse du décodage d'informations transmises sur porteuse radiofréquence par logiciel. En effet, le même flux de données est d'une part démodulé en un flux audible, et d'autre part en un flux de données numériques, sans que le matériel n'ait été modifié (Fig. 4). Par ailleurs, notons que l'auteur a recréé un récepteur FM comme nous le ferions à base de composants électroniques analogiques, en asservissant une boucle vérouillée en phase (PLL) sur la sortie filtrée en passe-bas du récepteur radio, au lieu d'utiliser le bloc de démodulation prêt à l'emploi de `gnuradio-companion`. Les divers filtres passe-bande permettent ensuite de sélectionner la nature de l'information traitée dans les diverses bandes de fréquence audio (son ou RDS).

Ci-dessous, quelques exemples de captures des signaux numériques transmis dans la bande FM commerciale à côté des signaux analogiques radiofréquences audibles, illustrant quelques uns des messages transmis sur ce mode de communication.

```
00A (BASIC) - PI:F211 - PTY:None (country:EG/FR/NO/BY/BA, area:National, program:17)
==>  RTL   <== -TP-  -Music-STEREO - AF:
00A (BASIC) - PI:F211 - PTY:None (country:EG/FR/NO/BY/BA, area:National, program:17)
==>  RTL   <== -TP-  -Music-STEREO - AF:
00A (BASIC) - PI:F211 - PTY:None (country:EG/FR/NO/BY/BA, area:National, program:17)
==>  RTL   <== -TP-  -Music-STEREO - AF:104.00MHz
00A (BASIC) - PI:F211 - PTY:None (country:EG/FR/NO/BY/BA, area:National, program:17)
==>  RTL   <== -TP-  -Music-STEREO - AF:92.40MHz, 93.90MHz
00A (BASIC) - PI:F211 - PTY:None (country:EG/FR/NO/BY/BA, area:National, program:17)
==>  RTL   <== -TP-  -Music-STEREO - AF:95.00MHz, 97.50MHz
00A (BASIC) - PI:F211 - PTY:None (country:EG/FR/NO/BY/BA, area:National, program:17)
==>  RTL   <== -TP-  -Music-STEREO - AF:101.20MHz, 101.50MHz
00A (BASIC) - PI:F211 - PTY:None (country:EG/FR/NO/BY/BA, area:National, program:17)
==>  RTL   <== -TP-  -Music-STEREO - AF:102.20MHz, 103.20MHz
00A (BASIC) - PI:F211 - PTY:None (country:EG/FR/NO/BY/BA, area:National, program:17)
@@@@@ Lost Sync (Got 46 bad blocks on 50 total)
@@@@@ Sync State Detected
@@@@@ Lost Sync (Got 46 bad blocks on 50 total)
@@@@@ Sync State Detected
00A (BASIC) - PI:F219 - PTY:None (country:EG/FR/NO/BY/BA, area:National, program:25)
==>  IRL   <== -TP-  -Music-STEREO - AF:99.60MHz, 99.80MHz
00A (BASIC) - PI:F219 - PTY:None (country:EG/FR/NO/BY/BA, area:National, program:25)
==>  IRL   <== -TP-  -Music-STEREO - AF:99.60MHz, 99.80MHz
00A (BASIC) - PI:F219 - PTY:None (country:EG/FR/NO/BY/BA, area:National, program:25)
==>  IRL N <== -TP-  -Music-STEREO - AF:97.80MHz, 98.40MHz
00A (BASIC) - PI:F219 - PTY:None (country:EG/FR/NO/BY/BA, area:National, program:25)
==> VIRL N <== -TP-  -Music-STEREO - AF:100.40MHz
@@@@@ Lost Sync (Got 47 bad blocks on 50 total)
@@@@@ Sync State Detected
@@@@@ Lost Sync (Got 49 bad blocks on 50 total)
@@@@@ Sync State Detected
00A (BASIC) - PI:FC67 - PTY:None (country:EG/FR/NO/BY/BA, area:Regional 9, program:103)
==>  GRAY  <== -TP-  -Music-STEREO - AF:100.10MHz
00A (BASIC) - PI:FC67 - PTY:None (country:EG/FR/NO/BY/BA, area:Regional 9, program:103)
==>  GRAY  <== -TP-  -Music-STEREO - AF:97.10MHz
02A (RT) - PI:FC67 - PTY:None (country:EG/FR/NO/BY/BA, area:Regional 9, program:103)
Radio Text A:  RADIO    STAR  TOUS LES  HITS  BESANCON  106.6   GRAY    100.2
00A (BASIC) - PI:FC67 - PTY:None (country:EG/FR/NO/BY/BA, area:Regional 9, program:103)
==>  10AY  <== -TP-  -Music-STEREO - AF:100.70MHz, 87.80MHz
```

Les trames que nous avons sélectionnées illustrent d'une part les champs de type AF (*alternative frequencies*) grâce auxquels un récepteur radio sait comment rechercher un nouveau canal pour le même programme radiophonique une fois la liaison courante trop faible, et d'autre part le champ RT (*radio text*) dans lequel un texte libre d'au plus 64 caractères est transmis. Nous n'avons pas eu l'occasion de

10. `https://cgran.org/wiki/RDS` : noter que, apparemment pour un dysfonctionnement de l'option `-fvar-tracking-assignement` des versions récentes de GCC, nous avons été obligés de désactiver les options `-g -O2` de la compilation (*i.e.* rester en `-O0`), faute de quoi la compilation échoue après avoir occupé toute la mémoire disponible.

tester TA (*traffic announcement*) qui est probablement une des applications les plus utiles de ce mode de communication numérique sur la bande FM commerciale.

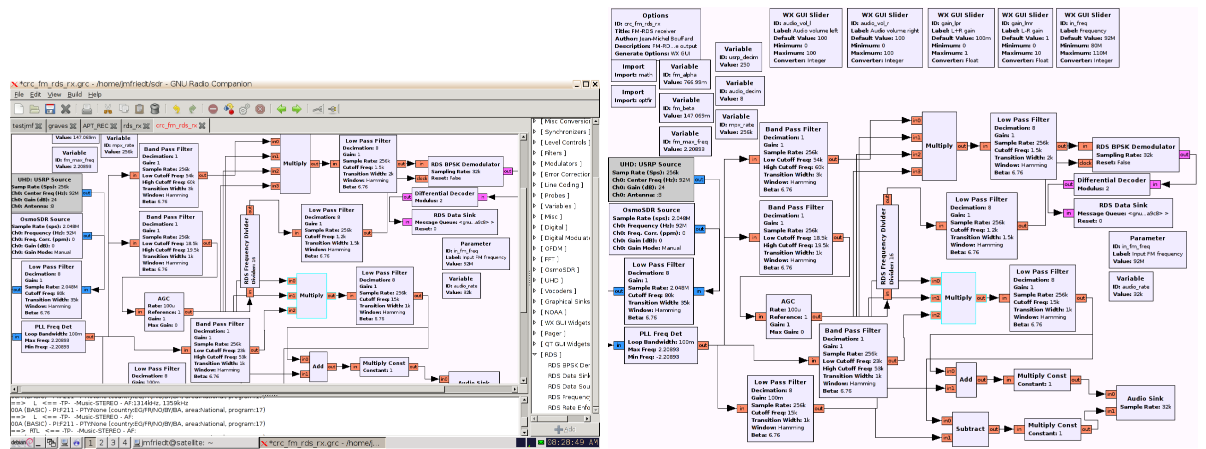


FIGURE 4 – Gauche : capture d'écran du décodeur RDS en fonctionnement. Noter dans le terminal, en bas de la fenêtre, les messages qui s'affichent (les même messages s'affichent dans le terminal d'où a été lancé `gnuradio-companion`), et dans le menu de droite les fonctions RDS qui ont été ajoutées après compilation des blocs disponibles à `https://cgran.org/wiki/RDS`. Droite : schéma complet issu de `http://mmbtools.crc.ca/content/view/45/73/#fm_rds_rx` mais épuré de quelques sorties graphiques qui consommaient trop de ressources de calcul sur eeePC 701.

Un dernier exemple propose le transfert d'heure (champ CT) par RDS ainsi que l'utilisation du champ de texte libre (RT) pour annoncer le titre en cours de diffusion :

```
04A (CT) - PI:F221 - PTY:None (country:EG/FR/NO/BY/BA, area:National, program:33)
Clocktime: 10.08.2012, 17:27 (+2.0h)
02A (RT) - PI:F221 - PTY:None (country:EG/FR/NO/BY/BA, area:National, program:33)
Radio Text A: Mozart : Concerto pour piano n 14:1er mvt
```

3.2 ADS-B

ADS-B (*Automatic Dependent Surveillance-Broadcast*) est un protocole numérique de communication entre un avion et le sol initié sur une requête provenant d'un RADAR qui sollicite une information de position d'un aéronef supposé être équipé d'un récepteur GPS. Depuis le sol, un récepteur (nous) ne peut entendre que la réponse de l'avion à la sollicitation : cette réponse se fait à la fréquence de 1090 MHz. Ainsi, ADS-B fournit une information complémentaire à l'analyse purement passive de réflexion de l'onde électromagnétique par l'avion : au lieu de se contenter du temps de vol et orientation de l'antenne au moment de l'émission de l'impulsion électromagnétique, la position de l'avion est complétée par sa localisation GPS.

Ce protocole a été implémenté sous forme de script généré par `gnuradio`, disponible à `https://www.cgran.org/wiki/gr-air-modes` et décrit en détail dans les transparents [10].

La mise en commun des informations acquises par de tels récepteurs d'ADS-B permet de cartographier la position des avions dans le monde de façon indépendante des contrôleurs aériens, tel que par exemple disponible à `http://www.radarvirtuel.com/`. Le logiciel proposé par N. Foster pour `gnuradio` inclut une sortie au format KML compatible avec Google Earth et Google Maps (Fig. 5).

3.3 Satellites

Deux essais sur les satellites en orbite basse polaire de la NOAA [11] et sur la station spatiale internationale (ISS) se sont révélés sans succès, probablement par manque de sensibilité du récepteur. L'ajout d'un amplificateur à faible bruit de 25 dB n'ayant pas amélioré la situation, il n'est pas exclu que les réglages n'aient été mauvais. En particulier, une mise à jour récente du pilote du convertisseur

11. `http://rof.li/pic/groundplane/`

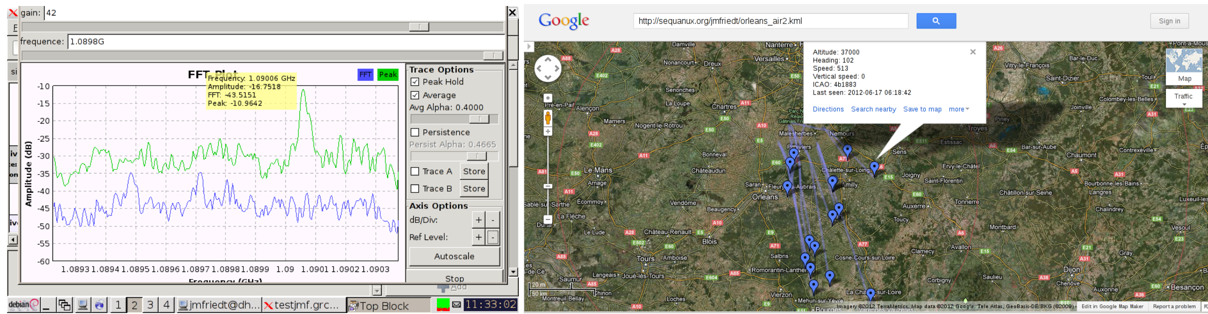


FIGURE 5 – Gauche : spectre acquis lors de l'écoute de transmissions en ADS-B, avec une fréquence centrale légèrement décalée par rapport à la fréquence d'intérêt. Droite : carte des avions suivis depuis un récepteur situé près d'Orléans, avec une antenne située sur un balcon dégagé vers l'est uniquement.

analogique-numérique permet de désactiver le contrôle de gain automatique, qui semble être un handicap significatif pour la réception de signaux faibles [12].

4 Décodage d'un mode numérique

L'utilisation de fonctionnalités existantes n'est que d'un intérêt limité et le développeur se lasse vite d'utiliser les modules disponibles sur le dépôt des projets associés à gnuradio CGRAN [13]. Notre véritable intérêt consiste en la maîtrise des méthodes de démodulation en vue d'adapter ces techniques à des méthodes qui ne sont pas encore implémentées, et ainsi utiliser l'outil opensource qu'est `gnuradio` comme un outil de prototypage souple. L'aspect opensource du projet est fondamental pour apprendre par la lecture des codes d'autrui [14]. Nous allons proposer d'aborder 3 modes de communication numériques : la FSK des radiomodems, l'AFSK tel que utilisé en *packet radio* amateur ou commercial, et le protocole ACARS de communication des avions civils (et militaires par compatibilité avec le réseau civil), protocole plus ancien mais plus "intéressant" que l'ADS-B puisque transportant des messages sur l'état de l'aéronef ou des messages de l'équipage vers le sol.

Bien que tous ces modes aient été conçus dans des débits de données compatibles avec les cartes son, la solution de récepteur de télévision numérique terrestre (DVB) offre néanmoins une solution intégrée qui évite l'achat d'un scanner radiofréquence. Bien qu'un scanner soit un outil souple d'emploi et bien plus sensible qu'une clé USB, son coût peut paraître rédhibitoire pour l'amateur désireux de s'engager dans la voie du décodage de modes numériques sans vouloir effectuer un investissement financier conséquent. Par ailleurs, le décodage numérique des signaux par logiciel permet de dynamiquement adapter les caractéristiques des filtres en n'étant limité que par la puissance de calcul disponible, alors qu'un scanner n'offre qu'un nombre fini de modes de décodage et généralement 2 ou 3 largeurs de filtres, implémentés sous forme matériel et donc inaccessibles pour l'acheteur de l'instrument.

4.1 Les modes de modulation et traitement numérique du signal

Nous ne pouvons prétendre introduire ici tous les éléments de traitement du signal associés aux modulations sur porteuse radiofréquence en vue de transmettre un signal. Afin de faciliter l'introduction aux concepts pour le lecteur qui n'est pas familier avec ces notions, rappelons qu'un signal périodique $s(t)$ est caractérisé pour son comportement temporel t par trois grandeurs : son amplitude A, sa fréquence f et sa phase φ selon $s(t) = A(t) \times \sin(2\pi f(t) + \varphi(t))$. Ces trois grandeurs peuvent évoluer dans le temps, individuellement ou simultanément, pour coder un signal transmis : $A(t)$ implique une modulation d'amplitude (AM en analogique, ou ASK – *Amplitude Shift Keying* – pour les modes numériques), $f(t)$ implique une modulation de fréquence (FM en analogique, ou FSK – *Frequency Shift Keying* – pour les modes numériques) et $\varphi(t)$ implique une modulation de phase (PSK – *Phase Shift Keying*) [11]. Chacune de ces modulations a des propriétés d'encombrement spectral, de robustesse à diverses

12. `http://www.oz9aec.net/index.php/gnu-radio/gnu-radio-blog/477-noaa-apt-reception-with-gqrx-and-rtlsdr`
13. `https://cgran.org`
14. `http://gnuradio.org/redmine/projects/gnuradio/wiki/OutOfTreeModules`

sources de bruit et de simplicité de mise en œuvre qui impliquent un choix contraint par les paramètres de communication recherchés. Un cas particulier que nous aborderons plus bas est l'AFSK – *Audio Frequency Shift Keying* – dans lequel les signaux ne sont pas codés par une fréquence ou une phase, mais par une modulation en fréquence audible de la porteuse radiofréquence. Il s'agit d'un mode développé spécifiquement pour l'utilisation avec des modes de communication de la voix (téléphonie dans le cas des modems, ou transceivers radiofréquences) qui ne nécessitent pas que la sortie soit adaptée pour des gammes de fréquences autres que les bandes de fréquences audibles (typiquement 1000-5000 Hz).

Ayant introduit ces concepts, nous constatons que le *dé*codage de signaux numériques consiste en l'extraction d'un de ces paramètres pour en tracer l'évolution dans le temps et par conséquent remonter aux signaux transmis. Le cas de l'ASK est le plus simple : nous calons le récepteur sur la fréquence de la porteuse (éventuellement avec un asservissement pour garantir que la dérive entre oscillateur de l'émetteur et du récepteur soit compensée), et un filtre passe bas détecte l'enveloppe du signal. L'amplitude du signal traduit alors la valeur du bit transmis. La FSK est à peine plus compliquée dans le principe puisque nous exploitons le signal de contrôle de l'oscillateur local asservi par verrouillage de phase sur le signal reçu, mais son exploitation est rendue triviale par la disponibilité sous `gnuradio-companion` du bloc `Modulators` → `WBFM receive`.

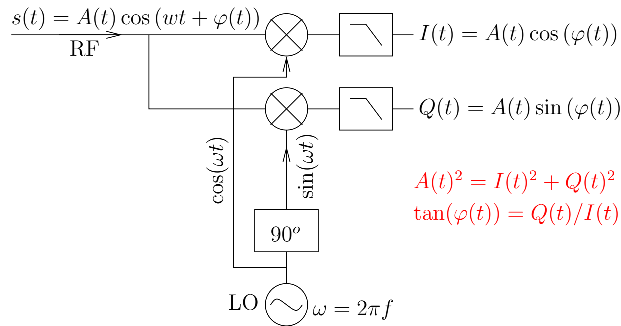


FIGURE 6 – Principe de la démodulation IQ pour extraire les deux grandeurs représentatives d'un signal d'entrée radiofréquence $s(t)$ connaissant sa fréquence f (ou pulsation angulaire $\omega = 2\pi f$), à savoir son amplitude $A(t)$ et sa phase $\varphi(t)$, toutes deux pouvant être utilisées pour coder de l'information si elles varient avec le temps t. La fréquence de coupure des deux filtres passe-bas après les mélangeurs définissent la bande passante du détecteur (qui définit en partie le débit de communication puisqu'il s'agit de la vitesse à laquelle les symboles déterminés par $A(t)$ et/ou $\varphi(t)$ peuvent varier).

La démodulation d'un signal passe dans un premier temps par la génération des signaux I (*Inphase*) et Q (*quadrature*) que nous avons déjà mentionné sans les définir (Fig. 6) : il s'agit du mélange du signal incident (dit RF – *RadioFrequency*) avec un oscillateur local de référence (dit LO – *Local Oscillator*) en vue de générer $I = A(t)\sin((RF - LO) \times t)$ et $Q = A(t)\cos((RF - LO) \times t)$. Alors $I + j \times Q = A(t)\exp(j \times 2\pi(RF - LO) \times t)$ avec $j^2 = -1$ [12] (noter que sin et cos sont séparés par un déphasage de 90°, donc en pratique les schémas de démodulateur I/Q indiquent ces opérations par un mélangeur donc la composante LO a été sur une des voies déphasé de 90°).

Ces deux signaux sont fournis par le composant Elonics E4000 après que l'utilisateur aie configuré LO. Dans ces conditions, une représentation simple des concepts de modulation que nous venons de citer est résumée dans le *diagramme de constellation* [12, p.11]. Ce diagramme trace en abscisse la composante I et en ordonnée la composante Q tel que nous venons de les définir et nous permet de revenir au plan complexe dont les propriétés sont bien connues : la distance d'un point à l'origine représente le module du complexe (dans notre cas $A(t)$) et l'angle entre l'axe des abscisses (I) et le point représenté dans le plan complexe représente sa phase $\varphi(t)$. Ainsi, un diagramme de constellation représente dans le plan complexe (I, Q) les divers codes possibles. Une modulation en phase se traduit par des points distribués le long d'un cercle de rayon constant centré sur l'origine, tandis qu'une modulation en amplitude se traduit par des points à distance variable de l'origine. La modulation en fréquence est un peu particulière puisque la fréquence est la dérivée de la phase et par conséquent les points dans le plan complexe tournent sur un cercle centré sur l'origine.

Nous allons, dans les sections qui vont suivre, illustrer le décodage de ces modes de modulation sur

des exemples concrets mis en œuvre sur des émetteurs commercialement disponibles dont nous désirons décoder les informations transmises. Le radiomodem XE1203F de Semtech émet sur un codage modulé en fréquence, le *packet radio* émet des signaux modulés par des fréquences audibles, et le protocole de communication avec les avions ACARS module un signal en amplitude.

4.2 Cas de la FSK du Semtech XE1203F

Le radiomodem XE1203F [15] de Semtech est un transceiver half duplex (soit il émet, soit il reçoit) susceptible de moduler un oscillateur en fréquence (FSK) pour coder les deux états possibles de la donnée numérique transmise. La fréquence de sa porteuse est programmable par pas de 500 Hz : nous sélectionnons 434 MHz, en plein milieu de la bande Industrielle, Scientifique et Médicale (ISM). L'excursion de la modulation FM entre les deux états est programmable – nous choisissons 55 kHz, et le débit des informations est défini par le flux de données issu de l'interface asynchrone (UART) du microcontrôleur (dans notre cas un ST STM32F103) connecté aux broches DATA et DATAIN du radiomodem (flux de données numériques transmis). Ces connexions de broches semblent être le mode de communication le plus simple entre le radiomodem XE1203F et le bus de communication asynchrone (RS232) d'un microcontrôleur, bien que les bits de début (start) et de fin (stop) de communication introduisent quelques subtilités dans l'interprétation qu'en fait le radiomodem.

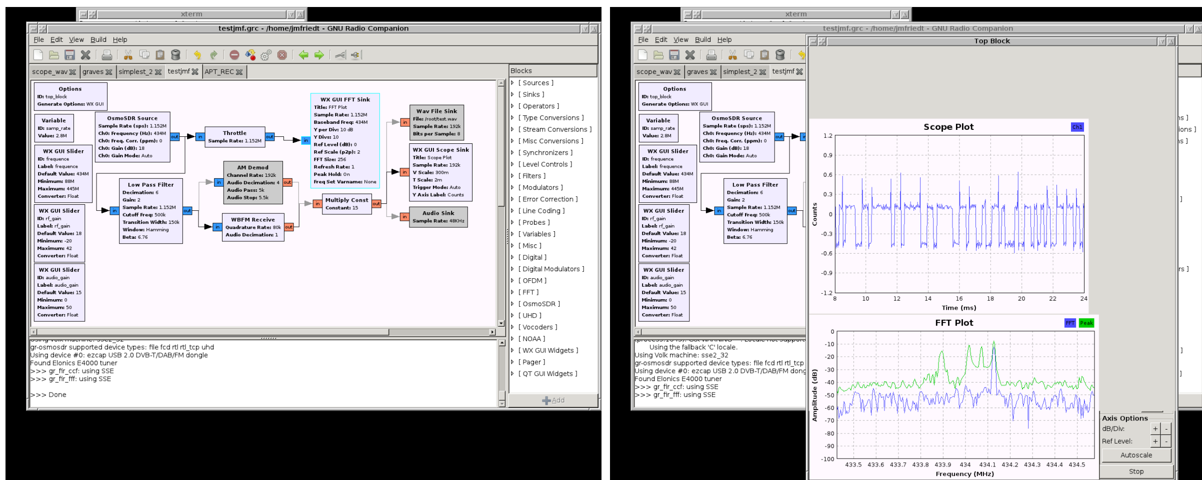


FIGURE 7 – Gauche : schéma blocs pour décoder le flux d'informations transmis par un radiomodem en FSK (il s'agit du même bloc de WFM que nous avions déjà utilisé pour les bandes FM commerciales). Droite : sortie oscilloscope au cours de l'enregistrement. Les deux états de transmission des données sont déjà clairement visibles et laissent présager d'un décodage aisé.

L'application du démodulateur WFM (qui est approprié pour une excursion aussi importante que 55 kHz) donne le signal de la Fig. 7 : nous observons de façon évidente des transitions représentatives du flux de données sur l'oscilloscope connecté en sortie de démodulateur. La FSK ne nécessite aucun autre traitement particulier pour retrouver les valeurs des bits puisque l'oscillateur radiofréquence est directement attaqué par la valeur du signal numérique transmis : le démodulateur fournit un signal basse fréquence proportionnel au signal d'erreur de la boucle asservie en phase, donc directement une tension proportionnelle à la fréquence codant chaque bit.

Par ailleurs, l'intervalle de temps entre deux transitions est en accord avec un débit de 4800 bits/s. Enfin, l'analyse de ces données pour en extraire une séquence compréhensible nécessite de s'approprier la séquence d'émission que nous avons implémenté dans le microcontrôleur selon les consignes de la fiche de données de Semtech :

1. la transmission commence par une séquence d'oscillations entre 0 et 1 nommée *bit synchronizer* pour que le récepteur cale son oscillateur sur le signal émis. Compte tenu du 0 en start bit et 1 en stop bit, nous constatons qu'une alternance de 0 et de 1 s'obtient en envoyant la valeur 0x55 (le bit de poids le plus faible est envoyé en premier), *i.e.* le caractère “U”,

15. `http://www.semtech.com/apps/filedown/down.php?file=xe1203f.pdf`

2. nous utilisons la fonctionnalité de coder une adresse du radiomodem émetteur, notamment après avoir constaté que le bruit radiofréquence ambiant a tendance à être détecté comme une transmission erronée en l'absence de cette fonctionnalité (*Pattern recognition block*),
3. finalement, le motif d'identification ayant été détecté, le message lui-même est transmis.

Nous vérifions pour chaque octet décodé que le dernier bit (stop bit) est à 1, sinon nous indiquons une erreur. Cette vérification permet en partie de s'affranchir du bruit sur le canal radiofréquence de transmission qui risquerait de nous laisser croire qu'un start bit s'est déclenché (transition de 1 à 0), bruit qui a peu de chances de se retrouver au niveau du stop bit en l'absence d'une vraie transmission de données.

Afin de faciliter le développement du décodage, la stratégie de développement que nous proposons consiste à systématiquement commencer par enregistrer dans un fichier binaire (ou WAV pour y ajouter un entête de format incluant taille des données et fréquence d'échantillonnage) un flux de données, y appliquer dans un premier temps un algorithme de traitement développé sous GNU/Octave, avant de le traduire en C pour inclure cet algorithme dans le formalisme de `gnuradio` (section 5) avant de finalement appliquer à un "vrai" flux de données issu du récepteur radio.

La figure 8 propose un exemple de schéma bloc pour valider l'enregistrement et afficher en mode oscilloscope les valeurs stockées dans un fichier binaire (nous avons toujours exploité le mode par défaut d'un enregistrement mono-voie – opposé à stéréo – en 8 bits/données).

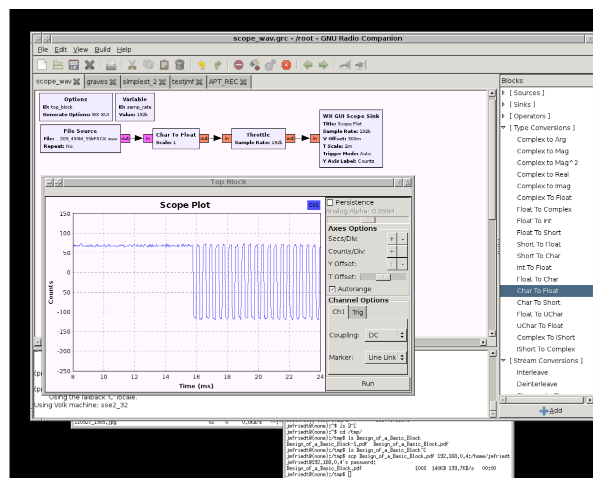


FIGURE 8 – Bloc pour rejouer un signal enregistré dans un fichier binaire muni d'une entête WAV : noter l'utilisation du bloc `throttle` pour imposer le débit d'émission des données lues dans le fichier.

4.3 Cas de l'AFSK

L'Audio Frequency Shift Keying encode les deux états possibles sur une fréquence audible, typiquement entre 1000 et 1550 Hz pour un état, et 2000 à 2500 Hz pour l'autre. En particulier, le mode de transmission *packet* hérite des protocoles mis en place à l'époque de la communication numérique sur lignes téléphoniques câblées analogiques au moyen de modems.

L'intérêt pour ce mode tient en deux aspects : d'une part il s'agit d'un mode de communication numérique couramment utilisé par les radioamateurs, et d'autre part le réseau de bus Ginko de Besançon exploite des modems basés sur ce protocole de communication pour localiser ses bus (émissions sur 154,150 et 154,154 MHz).

Le mode le plus commun de *packet radio* est un codage des deux états possible du bit par un signal à 2200 et 1200 Hz (protocole Bell 202), et transmission de données au rythme de 1200 bits/s. Sur l'exemple de la Fig. 9, les ronds bleus sont échantillonnés tous les 40 points, soit pour un échantillonnage à 46 kS/s, un débit d'information de 48000/40=1200 bits/s. Comme divers protocoles de modulation sont disponibles (par exemple codage sur 1300 et 2100 Hz tel que défini dans ITU-V.23 [16]), nous avons

16. `http://wiki.ham.fi/AFSK.en`

choisi d'appliquer des filtres large bande coupant autour de 1700 Hz, garantissant ainsi une compatibilité avec à peu près tous les cas décrits sur le web.

Le script GNU/Octave ci-dessous introduit les premiers pas pour le traitement des signaux enregistrés par `gnuradio` après démodulation FM en bande étoite (NFM) en vue d'en extraire les valeurs des bits successifs. L'approche sélectionnée consiste en une paire de filtres passe-bande autour des deux fréquences supposées coder les deux états transmis, et est donc souple d'emploi pour s'adapter à diverses valeurs de fréquences audio.

```
fe=48000;
deb=  338100;
fin=  488000;
SEUIL=200;

f=fopen(filename='120723_ginko.wav');d=fread(f,inf,'uint8');
d=d(deb:fin);N=fin(m)-deb(m);
f=linspace(0,fe,N);
d=d-mean(d);
```

Initialisation des variables – en partie de la fréquence d'échantillonnage `fe`, et lecture d'un segment de données dans un fichier enregistré par la fonction `wav` de `gnuradio-companion` (8 bits/échantillon, une voie).

```
c=ones(30,1)/30;
h=firls(42,[0 500 1000 1500 1900 fe/2]/fe*2,[0 0 1 1 0 0]);
[H,freq]=freqz(h,1,512,fe);
y12=filter(h,1,d(1:N));
yc12=conv(abs(y12),c);yc12=yc12(15:end-15);plot(yc12,'c');hold on
```

Nous définissons un filtre de réponse impulsionnelle finie (FIR) dont le gabarit est défini par les amplitudes du second vecteur pour les fréquences du premier vecteur. Dans ce cas, nous laissons passer l'énergie entre 1000 et 1500 Hz (amplitude de 1) et coupons en dehors de ces fréquences. Un filtre numérique est toujours défini pour des fréquences normalisées par rapport à la fréquence d'échantillonnage. Finalement, après filtrage, un filtre passe-bas est appliqué sous forme de convolution avec une fenêtre rectangulaire de 30 éléments de long.

```
h=firls(42,[0 1700 2000 2600 4600 fe/2]/fe*2,[0 0 1 1 0 0]);
y24=filter(h,1,d(1:N));
yc24=conv(abs(y24),c);yc24=yc24(15:end-15);plot(yc24,'m')
```

La procédure est réitérée pour un filtre passe-bande entre 2000 et 2600 Hz.

```
ind=[9160:40:48000];                          % http://wiki.ham.fi/AFSK.en
v1=find(yc12(ind)>yc24(ind));tout(v1)=1;      % 1300 Hz = 1 = mark
v2=find(yc12(ind)<=yc24(ind));tout(v2)=0;     % 2400 Hz = 0 = space
res(1)=0;
for k=1:length(tout)
   if (tout(k)==0) res(k+1)=1-res(k); else res(k+1)=res(k);end
end
plot(ind,tout,'o');
```

`v1` et `v2` définissent l'état de bit le plus probable en comparant la puissance en sortie des deux filtres passe-bande. Tel que décrit à `http://n1vg.net/packet/index.php`, un 0 (2400 Hz) encode un changement d'état du bit par rapport à la valeur précédente, et un 1 (1300 Hz) la reconduction de l'état précédent inchangé. Le choix de ne prendre qu'un point sur 40 est imposé par le débit supposé de 1200 bits/seconde sur des données échantillonnées à 48 kHz.

```
binaire=reshape(tout(1:floor(length(tout)/8)*8),8,floor(length(tout)/8))';
code_asc=binaire(:,1)+binaire(:,2)*2+binaire(:,3)*4+binaire(:,4)*8+binaire(:,5)*16+binaire(:,6)*32+binaire→
    ↪(:,7)*64; % +binaire(:,8)*128;
printf('%02x ',code_asc);
printf('\n');
```

Finalement, les données sont réorganisées en paquets en faisant l'hypothèse d'un octet pour 8 bits, et le résultat affiché sous forme d'une séquence de valeurs hexadécimales (`%02x`) ou de caractères ASCII.

L'obtention de la séquence de bits n'est que le début de l'aventure : il faut maintenant être capable d'en extraire une information pertinente, et ce en identifiant le baud rate (les protocoles associés au *packet* nous laissent penser qu'un flux de 1200 bits/s est le plus probable, en accord avec nos observations sur la Fig. 9), le nombre de bits/donnée et l'éventuel bit de parité, et finalement la signification de chaque octet décodé. Ces points n'ont pas encore été élucidés.

4.4 Cas de l'AM et le protocole ACARS

ACARS est un protocole de communication utilisé en aéronautique à l'échelle internationale pour transférer diverses informations au cours du vol d'un aéronef, que ce soit de façon automatique au cours de phases du vol bien définies (taxi avec des informations de type quantité d'essence, en cours de vol avec par exemple l'horaire d'arrivée prévu et des informations météorologiques, en approche avec les portes de débarquement et l'état des réacteurs, après atterrissage avec le volume d'essence restant, des informations sur l'équipage ou des dysfonctionnements informatiques).

La communication numérique en Europe suivant le protocole ACARS se fait sur la fréquence de 131,725 MHz, ou accessoirement sur 131,525 MHz et 131,550 MHz. Compte tenu de la bande passante

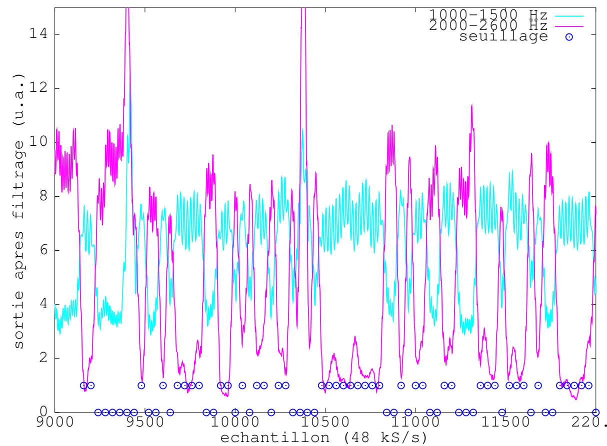


FIGURE 9 – Résultat du décodage, avec application de deux filtres passe-bande sur les signaux bruts issus du récepteur radiofréquence après démodulation en NFM (Narrow FM). Les bits successifs sont clairement visibles, compatibles avec un flux de données en 1200 bits/s.

de la démodulation par `gnuradio`, toutes ces fréquences peuvent être analysées simultanément. Pour des raisons que nous ne saurions expliquer, toutes les communications liées à l'aéronautique sont modulées en amplitude (AM).

ACARS est un protocole ancien et simple à décoder, bien documenté dans `http://files.radioscanner.ru/files/download/file4094/acars.pdf` et `http://www.tapr.org/aprsdoc/ACARS.TXT`. Nous y apprenons que deux fréquences encodent les deux états binaires possibles de l'information transmise : les fréquences de 1200 et 2400 Hz sont utilisées pour transmettre des informations à 2400 bits/seconde.

Il existe peu d'implémentations libres : `acarsd` à `http://www.acarsd.org/` est gratuit mais ne diffuse pas ses sources, tandis que `acarsdec` à `http://sourceforge.net/projects/acarsdec/` puis `http://www.r-36.net/src/acarsdec/` semble s'être arrêté à l'état de prototype qui n'a plus évolué depuis 2007 (ce qui ne retire rien à son intérêt pédagogique, mais l'exemple fourni n'est plus fonctionnel et l'hébergeur actuel du code n'en connaît pas le fonctionnement).

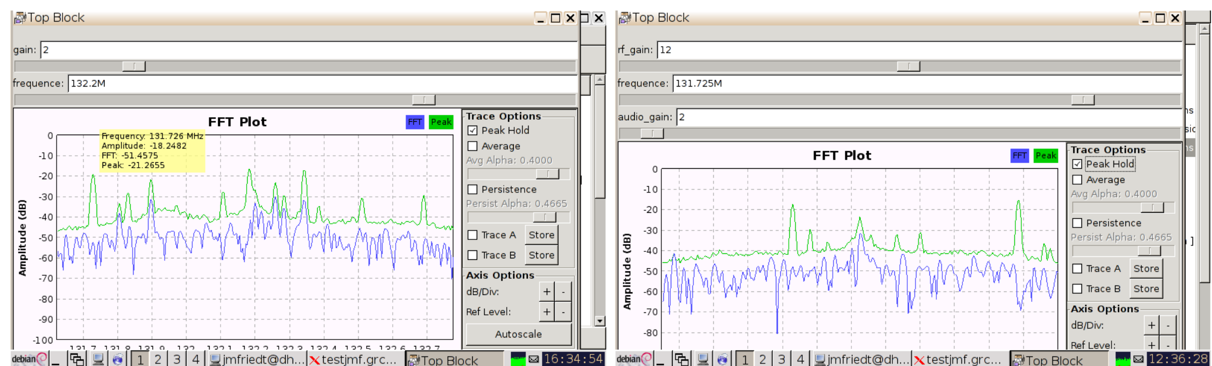


FIGURE 10 – Spectres dans la bande aérienne de 108 à 137 MHz, où toute communication se fait en modulation d'amplitude (AM).

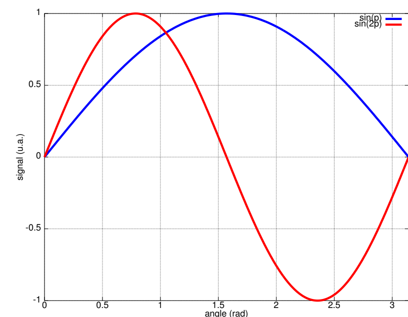
Le choix des fréquences de modulation (1200 et 2400 Hz) relativement au débit (2400 bits/seconde) peuvent paraître surprenant tant que nous n'avons pas mis en œuvre un décodeur de ce protocole : la première approche naïve (que nous avons utilisés suite aux expérimentations décrites auparavant sur le *packet*) consiste à répartir des filtres passe-bande à réponse impulsionnelle finie (FIR) et filtrer les deux bandes de 1200 et 2400 Hz. Cependant, cette approche nécessite un échantillonnage rapide du signal, car pour une fréquence de 48 kHz d'échantillonnage (compatible avec une sortie sur carte son), une période de signal à 2400 Hz ne comporte que 20 points, un nombre réduit pour définir un filtre passe-bande

efficace (et notamment présentant un coefficient nul à la fréquence nulle pour éliminer la composante DC qui rend les seuils difficiles à établir sinon). Cette approche fonctionne, mais ne tire pas partie du choix judicieux des paramètres que nous venons de citer selon les considérations suivantes :

1. la proportionnalité du débit (bitrate) et des deux fréquences utilisées pour coder l'information garantit la continuité de la phase entre bits successifs. En effet, une fois que la porte de découpe des bits est synchronisée sur l'oscillateur générant les signaux audio, nous sommes certains que la sinusoïde du signal audio passe toujours par 0 à un changement de bit,
2. le fait de coder un bit par une demi période audio et l'autre bit par une période complète exploite pleinement la capacité de l'intercorrélation à identifier la nature du bit encodé.

Nous allons développer ce dernier point qui est au cœur d'un décodeur plus efficace que la simple convolution d'un filtre passe-bande sur l'ensemble des données acquises. Notons dans un premier temps les relations suivantes, qui permettent de valider que la détection du mauvais état d'un bit se traduit par un signal de valeur moyenne nulle (intégrale sur la longueur d'un bit, soit 1/2400 s) tandis que la détection du bon bit se traduit par une valeur moyenne non nulle et égale pour les deux valeurs de bits détectés :

- $\int_0^1 \sin(2\pi t)\sin(\pi t)dt \propto \int_0^1 (\cos(3\pi t) - \cos(\pi t))\,dt = sin(3\pi) - sin(0) - (sin(\pi) - sin(0)) = 0$
- $\int_0^1 \sin(2\pi t)\sin(2\pi t)dt = 1/2 \times \int_0^1 (\cos(4\pi t) - \cos(0))\,dt = 1/2 \times (sin(4\pi) - sin(0) + 1) = 1/2$
- $\int_0^1 \sin(\pi t)\sin(\pi t)dt = 1/2 \times \int_0^1 (\cos(2\pi t) - \cos(0))\,dt = 1/2 \times (sin(4\pi) - sin(0) + 1) = 1/2$



Pour ceux qui ont oublié leur table de trigonométrie, il s'agit d'une excellente occasion de tester le site de Wolfram à `http://www.wolframalpha.com/` en lui demandant de calculer `int_0^1 sin^2(pi*x)dx`. Noter par ailleurs qu'il est relativement trivial d'intuiter que l'intégrale sur une période d'une fonction paire par une fonction impaire de valeur moyenne nulle est nulle, et que l'intégrale du carré d'une fonction (donc dont toutes les valeurs sont positives) est non-nulle (figure de droite).

Nous constatons donc que le choix de moduler à $f_m = 1200$ Hz un état des données et à $2f_m = 2400$ Hz l'autre état répond à un soucis d'efficacité du décodage : la convolution du signal acquis avec $\sin(2\pi f_m t)$ donne une valeur moyenne nulle si le segment de signal ne code pas le bon état et donne 1/2 si l'état recherché est présent. Ainsi, il "suffit" d'initialement se synchroniser sur le flux de données – rôle des 16 premiers octets qui oscillent à 2400 Hz, puis effectuer deux produits des séquences suivantes avec $\sin(2\pi f_m t)$ et $\sin(\pi f_m t)$ pour retrouver l'état des bits successifs (Fig. 11).

Finalement, ayant obtenu un flux de bits, il reste à en interpréter le contenu : ACARS exploite un codage dans lequel les *transitions* d'un bit au suivant sont notifiées, au lieu de coder directement l'état des bits eux-mêmes. Ainsi, un signal à 1200 Hz indique un changement d'état par rapport au bit précédent, alors que 2400 Hz code le maintien de l'état précédent [17].

Un exemple de mise en œuvre par prototypage sous GNU/Octave propose l'implémentation suivante

```
function binaire=fft_decod(filename,deb,fin,seuil)
jmfdebug=0;
fe=48000;
f=fopen(filename);
d=fread(f,inf,'uint8');
d=d(deb:fin);
N=fin-deb;
```

Nous lisons le segment de données compris entre les indices `deb` et `fin` dans le fichier `filename`,

```
c=ones(60,1)/60;
dm=conv(d,c);dm=dm(60/2:end-60/2);
d=d-dm;   % retranche moyenne glissante sur 3 periodes
```

application d'une moyenne glissante (fenêtre rectangulaire de 60 points de longueur, soit 1,25 ms à `fe` de 48 kHz),

```
t=[0:519]; % 2400 Hz dans 48 kHz = 20 points/periode *26
c2400x13=exp(i*t*2400/fe*2*pi); % recherche du max des 13 periodes a 2400 Hz
s=conv(c2400x13,d);
s=s(length(t)/2:end-length(t)/2);
[a,b]=max(real(s));   % max de ressemblance pour decallage de b
% plot(d(b-260:b+260)/120,'g');hold on; plot(real(c2400x13),'r');
```

17. `leonardodaga.insyde.it/Corsi/AD/Documenti/ARINCTutorial.pdf`

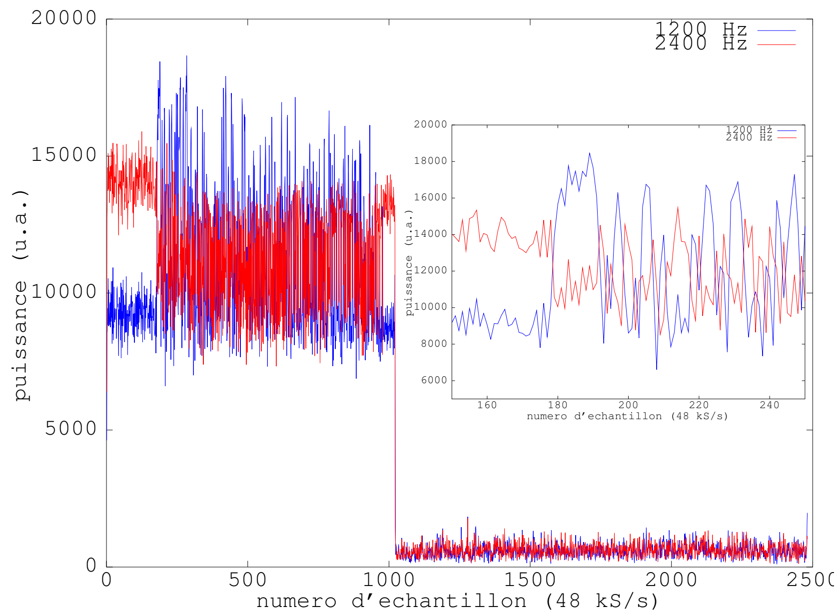


FIGURE 11 – Signaux issu d'un signal audio enregistré lors d'une transmission ACARS sur 131,725 MHz avec application des algorithmes de filtrage à 1200 et 2400 Hz tel que décrit dans le texte. Inséré à droite, un zoom sur les premiers signaux exploitables en fin de synchronisation du récepteur sur l'émetteur (séquence de plusieurs périodes à 2400 Hz) : l'identification de ce point de départ est un élément clé du bon décodage de la suite de la trame car définit le premier bit du premier octet sur lequel tout le reste du décodage se base.

le point clé de l'algorithme est de synchroniser le récepteur sur les oscillations de l'émetteur, et pour ce faire nous devons trouver le maximum de l'intercorrélation entre la séquence des signaux à 2400 Hz émis par ACARS dans ce but de synchronisation. Nous générons donc un vecteur `c2400x13` qui contient les valeurs d'une sinusoïde échantillonnée à `fe` et de fréquence 2400 Hz, et recherchons le maximum d'intercorrélation qui, pour un signal réel, revient à une convolution.

```
  b=mod(b,20)+5;     % revient au debut par pas de 2pi
  d=d(b+400:end); % bien se caler est fondamental pour la suite
                  % est-il judicieux d'essayer a +/-1 ?
```

Ayant identifié le point de mesure permettant de synchroniser les sinusoïdes synthétisées de façon logiciel et les données acquises, nous avons choisi de replacer le curseur au début de la séquence de mesure en nous déplaçant par pas de 20 échantillons soit une période (48000/2400) de sinusoïde à 2400 Hz. Cette étape est peut-être inutile, mais évite de risquer de s'accrocher sur une séquence longue de signaux à 2400 Hz se trouvant au milieu du message ACARS transmis,

```
t=[0:19];  % 2400 Hz dans 48 kHz = 20 points/periode
c2400=exp(i*t*2400/fe*2*pi);
c1200=exp(i*t*1200/fe*2*pi);
s12=conv(c1200,d);
s24=conv(c2400,d);
% plot(d); hold on;plot(real(s12),'r'); plot(real(s24),'r');
```

Nous recherchons désormais les bits individuels, codés soit par une période de 2400 Hz, soit une demi période de 1200 Hz, soit dans tous les cas 20 échantillons,

```
t=[0:19];  % 2400 Hz dans 48 kHz = 20 points/periode
fin20=floor(length(s12)/20)*20;
s12=s12(1:fin20);s24=s24(1:fin20);
rs12=reshape(abs(s12),20,length(s12)/20);
rs24=reshape(abs(s24),20,length(s24)/20);
rs12=sum(rs12);
rs24=sum(rs24);
```

contrairement au cas précédent du XE1203F où nous nous étions contentés de prendre un point sur 40 pour tenir compte de la fréquence de bits transmis, cette fois nous allons moyenner les valeurs des échantillons au sein de chaque bit pour améliorer le rapport signal à bruit (et donc la capacité de décodage du message en milieu bruité). Pour ce faire, nous réorganisons les deux vecteurs de mesures filtrées par les séquences de sinusoïdes à 1200 et 2400 Hz sous forme d'une matrice de 20 points de large (la longueur en échantillons d'un bit), et allons sommer les éléments pour effectuer la moyenne. `rs12` et `rs24` contiennent les informations permettant d'identifier si un bit est plus problement à 1 ou 0 et mérite d'être affiché

dans une phase préliminaire de déverminage (Fig. 11).

```
if (jmfdebug==1) plot(rs12,'bo-');end
if (jmfdebug==1) hold on;plot(rs24,'ro-');legend('1200','2400');end

seuil=max(rs24)*0.55;
l0=find((rs24+rs12)>seuil);   % on ne garde que les points utiles
rs12=rs12(l0);rs24=rs24(l0);
l1=find(rs24>seuil);l1=l1(1)
rs12=rs12(l1:end);
rs24=rs24(l1:end);
```

Finalement, la séquence d'échantillons est traitée pour la convertir en valeurs binaires : nous recherchons, par critère de seuil, les valeurs pertinentes (signal au-dessus du bruit, ci-dessus) et les valeurs des bits par comparaison des sorties des deux filtres (convolutions, ci-dessous).

```
l=find(rs12>rs24);l=l(1)
rs12=rs12(l:end);
rs24=rs24(l:end);
pos12=find(rs12>rs24);
pos24=find(rs24>rs12);
toutd(pos12)=0;
toutd(pos24)=1;
```

Les valeurs binaires sont finalement réorganisées en octets, selon la méthode vue auparavant sur le radiomodem, avec la valeur du bit codant la transition d'état ou le maintien de la valeur par rapport à la valeur courante du bit.

```
n=1;
tout(n)=1;n=n+1;  % les deux premiers 1 sont oublie's car on se sync sur 1200
tout(n)=1;n=n+1;
for k=1:length(toutd)
  if (toutd(k)==0) tout(n)=1-tout(n-1); else tout(n)=tout(n-1);endif
n=n+1;
end
binaire=reshape(tout(1:floor(length(tout)/8)*8),8,floor(length(tout)/8))';
code_asc=binaire(:,1)+binaire(:,2)*2+binaire(:,3)*4+binaire(:,4)*8+binaire(:,5)*16+binaire(:,6)*32+binaire→
    ↪(:,7)*64;
checksomme=1-mod(sum(binaire(:,1:7)')',2); % verification
printf('%02x ',code_asc); printf('\n');
printf('%c',   code_asc); printf('\nCRC:'); printf('%d',checksomme-binaire(:,8)); printf('\n');
```

Les valeurs sont affichées sous forme hexadécimale, code ASCII correspondant, et validation du bit de parité.

En résumé, ce code exploite la trame ACARS de la façon suivante :

1. identification de l'occurence du début de trame sous forme de plusieurs oscillations à 2400 Hz : nous nous sommes imposés de trouver au moins 13 périodes du signal à 2400 Hz pour définir le début de trame et synchroniser notre traitement sur la phase du signal acquis,
2. identification des segments de 1 (2400 Hz) et de 0 (1200 Hz) par convolution d'une période de sinusoïde de chacune des fréquences avec le signal à traiter. Seule une unique convolution est utile car l'incertitude sur la phase (qui nécessiterait par exemple 20 convolutions pour le signal à 2400 Hz pour identifier la phase parmi les 20 valeurs possibles qui maximisent l'intercorrélation) a déjà été résolue lors de l'étape précédente,
3. le codage des informations représentant une transition d'un état à un autre et non l'état lui-même, il reste à convertir la séquence de bits issue de la convolution en des valeurs interprétables en ASCII. Noter en particulier que la documentation à `http://www.pervisell.com/ham/raftmode.htm#I76` est erronée concernant la parité du flux de données. Il est par ailleurs remarquable d'enfin comprendre la signification d'entrées étranges de la table ASCII (`man ascii`) telles que le caractère 01 (début d'entête), 02 (début de texte) ou 03 (fin du texte) qui sont exploitées par ACARS. Par ailleurs, le fait de coder la transitions induit que dès la première erreur de décodage, tout le reste du message sera illisible. Ainsi, nous aurons souvent l'identifiant de l'avion émettant le message, mais obtenir tout le contenu du message lui-même est difficile car dépendant de l'absence d'erreur de traitement des bits tout au long du message.

L'application d'une convolution entre deux séries de données de longueur respectives M et N pose toujours un problème de définition de l'origine. En effet, la version numérique (discrète) de la convolution c entre u_i, $i \in [1..M]$ et v_j, $j \in [1..N]$, est

$$c(n) = \sum_{m=-\infty}^{+\infty} u_m \times v_{n-m}$$

avec u et v égaux à 0 en dehors des intervalles cités ci-dessus. Dans le cas du passage dans le domaine de Fourier, les séquences u ou v sont complétées de 0 (*zero padding*) pour avoir la même taille (et s'approcher de la puissance de 2 supérieure la plus proche dans le cas de la transformée de Fourier rapide).

Dans l'équation ci-dessus, l'intégrale dont les bornes sont $\pm\infty$ peut en pratique être longue à calculer, l'infini étant difficile à atteindre. On intégrera donc, pour deux séries de points de longueurs M et N, soit

entre $\max(M, N)$ si nous faisons glisser la série des u sur les v pour générer ainsi un total de $\max(M, N)$ points, ou alternativement une série de $M + N$ points si nous complétons une des deux séries par des 0 pour égaler la longueur des deux séries. Cette approche est par exemple celle proposée dans le code traduit en C que nous développerons ci-dessous (section 5.2), puisque le passage par Fourier impose de multiplier point par point deux séquences de longueurs égales. Le problème de la position de l'origine se pose donc selon le formalisme sélectionné, et la principale difficulté rencontrée dans la conversion du code GNU/Octave vers C se trouve à ce niveau (sans compter les conventions de normalisation de la transformée de Fourier qui imposent de recalculer les seuils dans la nouvelle implémentation).

Le programme GNU/Octave ci-dessus identifie le début de trame (série d'oscillations à 2400 Hz), affiche la séquence des valeurs hexadécimales décodées, leur interprétation dans le code ASCII, et l'application du bit de parité à la séquence ainsi décodée. Dans le résultat ci-dessous, nous constatons que toutes les valeurs ont été convenablement décodées sauf les 9 dernières, probablement sans importance car situées après le caractère ASCII 0x03 (fin de la chaîne de texte).

Nous affichons ci-dessous dans un premier temps les valeurs hexadécimales des octets décodés (commençant toujours par la séquence de synchronisation 0x2b 0x2a 0x16 0x16 si le décodage est effectué correctement), l'interprétation selon le code ASCII si le caractère est affichable, et finalement l'adéquation avec le bit de parité (0 pour la cohérence avec ce bit, ±1 s'il y a erreur) :

```
binaire=fft_decod('acars_orleans.wav',101001+2.15e6,101001+2.15e6+40000,7000);

2b 2a 16 16 01 58 2e 47 2d 45 55 55 47 15 48 31 39 02 43 30 33 41 42 41 39 32 31 36 23 43 46 42
57 52 4e 2f 57 4e 31 32 30 36 33 30 30 38 33 33 30 30 33 34 30 30 30 30 30 36 4e 41 56 20 49 4c
53 20 32 20 46 41 55 4c 54 20 20 20 20 20 20 20 20 20 0d 03 6d 1d 7f 7f 7f 7f 7f 7f 7f
m*X.G-EUUGH19C03ABA9216#CFBWRN/WN12063008330034000006NAV ILS 2 FAULT
CRC:0000000000000000000000000000000000000000000000000000000000000000000000000000000000-100-1-1-1-1-1-1
```

Il est toujours intéressant de savoir qu'un avion – ici G-EUUG qui est un Airbus A320 de British Airways [18] allant de Londres à l'Italie au cours du vol BA9216 – est en train de voler avec un défaut sur son instrumentation secondaire d'atterissage [19].

Les abréviations des messages sont décrits à `http://www.angelfire.com/sc/scannerpost/acars.html`

5 Du prototypage à l'exploitation sous gnuradio

Gnuradio-companion convertit un schéma de traitement défini de façon graphique en script python, faisant appel aux fonctions correspondantes. Cependant, les données qui transitent entre les blocs, *i.e.* le flux issu des démodulateurs I/Q, ne passent que d'un bloc à l'autre sans être accessibles depuis le script Python (code 1, correspondant au schéma graphique de la Fig. 8). Chacun de ces blocs est lui-même codé en Python ou C(++), suivant un protocole d'échange de données [20]. Nous allons donc traduire le code C d'exemple de décodage des trames FSK pour respecter les conventions de blocs gnuradio-companion et ainsi effectuer le traitement en temps réel (au lieu d'enregistrer les informations dans un fichier binaire en vue de son post-traitement).

Le développement de blocs pour la gnuradio nécessite l'utilisation d'un environnement particulier constitué d'un répertoire contenant lui même des sous-répertoires dans lesquels seront stockés les différents fichiers nécessaires à l'exploitation du bloc.

La première étape pour la création d'un (ou plusieurs blocs) est, donc, de disposer de cet environnement de développement. Toutefois, afin d'éviter l'étape fastidieuse de la création manuelle de l'arborescence, nous allons utiliser le script `gr-modtool.py` [21]. Cet outil sert à la fois à générer le squelette du répertoire et de ses sous répertoires mais également de le peupler.

La création du projet se fait avec la commande suivante :

```
gr_modtool.py create plop
```

À l'issue de cette commande, un répertoire `gr-nomduprojet` apparaît dans le répertoire courant et comporte (entre autres) les répertoires :

– `grc` pour les fichiers de descriptions utilisés par gr-companion,

18. `www.planespotters.net`
19. `http://www.scancat.com/Code-30_html_Source/acars.html` pour le contenu des trames et en particulier de l'entête.
20. `http://gnuradio.org/redmine/projects/gnuradio/wiki/BlocksCodingGuide`
21. `git://github.com/mbant/gr-modtool.git`

```
self.wxgui_scopesink2_0 = scopesink2.scope_sink_f(
      self.GetWin(),
      title="Scope Plot",
[...]
  )
self.Add(self.wxgui_scopesink2_0.win)
self.gr_throttle_0 = gr.throttle(gr.sizeof_float*1, samp_rate)
self.gr_file_source_0 = gr.file_source(gr.sizeof_char*1, "nom_du_fichier.wav", True)
self.gr_char_to_float_0 = gr.char_to_float(1, 1)

self.connect((self.gr_throttle_0, 0), (self.wxgui_scopesink2_0, 0))
self.connect((self.gr_char_to_float_0, 0), (self.gr_throttle_0, 0))
self.connect((self.gr_file_source_0, 0), (self.gr_char_to_float_0, 0))

tb = top_block()
tb.Run(True)
```

Table 1 – Le code Python généré par `gnuradio-companion` ne donne pas accès au flux de données radiofréquences mais ne fait que définir des blocs et les connecter entre eux pour router les informations d'une unité de traitement à la suivante.

- `include` pour les entêtes,
- `lib` pour les fichiers sources C++ d'extension `.cc`.

L'ajout d'un bloc de traitement du signal numérique dans le projet se fait avec la commande (il faut se trouver dans le répertoire du projet) :

```
gr_modtool.py add -t general plup
```

L'argument -t spécifie le type de bloc, dans le cas présent nous demandons un type générique mais il existe d'autres types tels que des sinks, des sources, des décimateurs, ... Comme nous le verrons un peu plus bas, le type va impacter sur la signature de certaines méthodes.

```
Enter valid argument list, including default arguments:
Add Python QA code? [Y/n]
Add C++ QA code? [Y/n]
```

La première ligne permet de directement spécifier les paramètres que le bloc recevra au moment de son instanciation (type de donnée, gain, seuil, ...). Les deux lignes suivantes correspondent à l'ajout (ou pas) de test-unitaire pour le bloc.

À l'issue de cette étape, les répertoires ont étés peuplés avec les fichiers nécessaires et ceux-ci partiellement remplis.

Trois fichiers retiendront principalement notre attention :

- `grc/nomDuProjet_nomDuBloc.xml` : ce fichier fournit la description du bloc à `gnuradio-companion`. Il stipule, entre autres, le nom du bloc (balise `name`), dans quelle catégorie il doit être rangé (balise `category`), les paramètres (balise `param`) ainsi que leurs noms, type et variables correspondantes, et les entrées (`sink`) et les sorties (`source`). Un exemple d'un tel fichier qui passe un paramètre (valeur réelle `seuil`) à la méthode de traitement des données `decodeur` de la classe `acars` est

```
<?xml version="1.0"?>
<block>
  <name>decodeur</name>
  <key>acars_decodeur</key>
  <category>acars</category>
  <import>import acars</import>
  <make>acars.decodeur($seuil)</make>
  <param>
    <name>Threshold</name>
    <key>seuil</key>
    <type>real</type>
  </param>
  <sink>
    <name>in</name>
    <type>float</type>
  </sink>
</block>
```

- `include/nomDuProjet_nomDuBloc.h` et `lib/nomDuProjet_nomDuBloc.cc` : la classe et son implémentation. Par défaut ils contiennent :
  - un constructeur privé,
  - un destructeur public,
  - une fonction `nomDuProjet_make_nomDuBloc` définie comme `friend` vis-à-vis de la classe (donc pouvant accéder au constructeur) et dont le rôle est de retourner une instance de la classe ;
  - une méthode publique `general_work`.

Ces concepts abstraits de l'arborescence seront illustrés par des bouts de code plus loin dans ce document (section 5.3), et le lecteur est encouragé à consulter le code d'exemple disponible à `http://jmfriedt.free.fr/gr-acars.tar.gz` ou sur le site de CGRAN à `www.cgran.org/wiki/ACARS`.

La dernière méthode peut également s'appeler `work` selon le type du bloc. Cette méthode est sans doute la plus importante car c'est elle qui reçoit le flux venant du bloc précédent dans la chaîne, réalise le traitement et fournit les nouvelles données. Sa signature est la suivante :

```
int plop_plup::general_work (int noutput_items,
```

```
gr_vector_int &ninput_items,
gr_vector_const_void_star &input_items,
gr_vector_void_star &output_items)
```

avec `noutput_items` qui comme son nom ne l'indique pas, donne le nombre d'informations reçues, `input_items` un tableau contenant les données entrantes et qui doit être recasté dans le bon type, `output_items` le tableau que le bloc doit remplir avec les données traitées. `ninput_items` peut ne pas exister, selon le type du bloc, et semble ne pas servir à grand chose.

Afin de compiler notre application, nous opérons comme avec toute archive rencontrée jusqu'ici pour `gnuradio` : création d'un sous-répertoire `build` dans l'arborescence des sources du module à compiler, exécuter `cmake ../` après être rentré dans ce répertoire, puis `make VERBOSE=1 && make install`. Noter qu'une fois le bloc nouvellement créé dans `gnuradio-companion` (dans la liste des menus de la colonne de droite), il est inutile de relancer l'interface graphique suite à chaque recompilation. En effet, le code Python qui est exécuté en pratique est régénéré à chaque fois que nous cliquons sur l'icone avec des engrenages, et le module nouvellement compilé rechargé. Nous aurons donc toujours la dernière version du module compilé *et installé* par `make install` disponible lors de l'exécution de la chaîne de traitement (Fig.12).

5.1 Cas du radiomodem XE1203F

Ayant prototypé les méthodes de filtrage sous GNU/Octave, nous désirons pouvoir afficher les données décodées en temps réel. Les étapes de ce portage se font en 3 étapes :

1. conversion d'un script GNU/Octave en C et vérification de toutes ses fonctionnalités en lisant un enregistrement d'un fichier audio (ou binaire si sa fréquence d'échantillonnage n'est pas compatible avec celle de la carte son) et application des filtres implémentés en C. Cette étape a encore l'avantage de pouvoir traiter l'intégralité des données contenues dans le fichier (code 2),
2. conversion du programme cité ci-dessus pour travailler sur des segments de données de taille aléatoire (Fig. 14), représentatives des bouts de données qui seront fournis par les blocs de démodulation et de filtrage de `gnuradio`,
3. intégration du code C(++) ainsi généré dans le formalisme d'un bloc `gnuradio`.

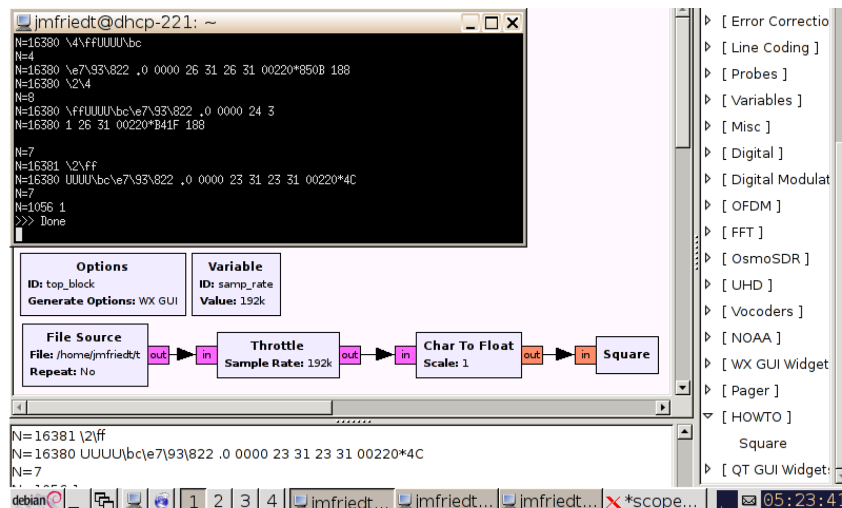


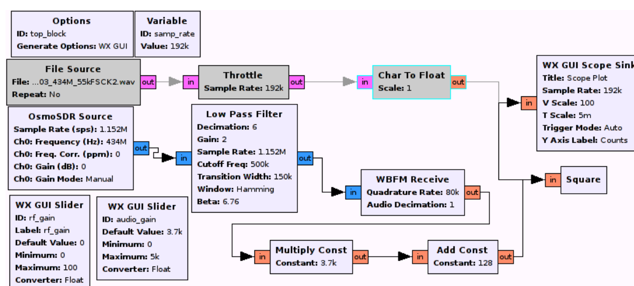
FIGURE 12 – Décodage du flux d'informations issu d'un radiomodem XE1203F par traitement logiciel des données I et Q pour en extraire l'information numérique émise, dans un premier temps enregistré dans un fichier binaire (.wav) et restitué pour valider le bon fonctionnement du bloc (nommé `square`). Noter, en bas du menu à droite, l'ajout d'une entré `HOWTO` avec le module `square`, qui correspond au résultat de la compilation de notre module de traitement. Il est inutile de relancer `gnuradio-companion` lors de modifications des fonctions de `square` : le bloc est rechargé lors de la génération du code Python à chaque nouvelle exécution de la séquence de traitements.

```
[... entetes habituels ...]
#define MAXSIZE 100000
#define NSEARCH 260  // 13 periods * 20 pts/period

void conv1(unsigned char* in,int* out,int N,int len)
{int somme=0,k;
 for (k=0;k<len;k++) somme+=(int)in[k];    // moyenne glissante sur len points
 for (k=len;k<N;k++) {somme=somme-in[k-len]+in[k];out[k-len]=somme/len;}
}

#define segment 8192

int main(int argc,char **argv)
{char filename[255];
 int deb,seuil,courant;
 int k,i,fe=192000,f,N;
 int dm[MAXSIZE];
 unsigned char d[MAXSIZE],code_asc;
 int b=0,l0=0,pos=0;

 if (argc<4) printf("%s filename deb seuil\n",argv[0]); else
   {sprintf(filename,"%s",argv[1]);deb=atoi(argv[2]);seuil=atoi(argv[3]);}

 deb=0;
 do {
   f=open(filename,O_RDONLY);  // lecture de segment donnees dans le fichier de points
   k=segment;
   i=lseek(f,deb,SEEK_SET);
   if (i!=deb) fprintf(stderr,"lseek error %d\n",k);
   N=read(f,&d[pos],k); printf("\nN=%d\n",N);
   close (f);

   conv1(d,dm,N,20);        // c=ones(20,1)/20;dm=conv(d,c);dm=dm(20/2:end-20/2);
   for (k=0;k<N;k++) if (dm[k]<seuil) dm[k]=1; else dm[k]=0;
       // k=find(dm<100);dm(k)=0; k=find(dm>=100);dm(k)=1;dm=1-dm;
       // % etat au repos est 1, start bit est passage a 0

   courant=0;
   while (courant<(N+pos-380)) {    //  k=find(dm(courant:end)<1);dm=dm(k(1)+courant-1:end);
     do {courant++;} while ((dm[courant]==1)&&(courant<(N+pos-380)));
     if (courant<(N+pos-380)) // db=dm(20:40:20+40*9);        % 10 bits = START+9, 40 points/bit
       {code_asc=dm[courant+60]+dm[courant+100]*2+dm[courant+140]*4+dm[courant+180]*8+
                 dm[courant+220]*16+dm[courant+260]*32+dm[courant+300]*64+dm[courant+340]*128;
        if (code_asc>0) {
            if (dm[courant+380]==0) {} // printf("err "); // stop bit error
               else
               {if (((code_asc>=32)&&(code_asc<128))||(code_asc=='\n')||(code_asc=='\r'))
                   printf("%c",code_asc);
                   else printf("\\%x",code_asc);
               }
        }
        courant+=380;  // milieu du stop bit de l'octet qui vient d'etre traite'
       }
   }
   memcpy(d,&d[courant],N+pos-courant); // copie ce qui reste de donnees
   pos=N+pos-courant;
   deb+=segment;
 } while (N==segment); // boucle sur deb pour simuler blocs grc
 printf("\n");
}
```

TABLE 2 – Traduction en C du code prototypé sous GNU/Octave pour le décodage des transmissions d'un radiomodem XE1203F modulé en fréquence (FSK) pour une transmission numérique de données à 4800 bits/s. Nous avons laissé en commentaires les principales séquences du code pour GNU/Octave que nous avons traduites. Noter la lecture du fichier par segments de `segment` éléments, et concaténation des valeurs qui n'ont pu être traitées car ne formant pas un bit d'information complet, représentatif du flux de données que nous fournira `gnuradio` lors d'un décodage en temps réel.

Le problème de la gestion de la taille des données à traiter est résolu de la façon suivante : nous savons qu'un octet complet tient sur 10 bits (start, 8 bits de données, stop) donc 400 échantillons pour un signal échantillonné à 192000 Hz et transmis à 4800 bits/s (192000/4800×10=400). Nous allons donc tout d'abord rechercher l'occurence d'un start bit (passage de 1 – état au repos – à 0 – état du start bit [22]), et si la masse de données à traiter dépasse 400 bits, nous continuons le traitement consistant à tester la valeur du bit tous les 40 échantillons (puisque 192000/4800=40). Comme dans toute communication asynchrone, le start bit permet de se resynchroniser pour les 9 bits qui vont suivre : nous vérifions que le stop bit est bien à 1, et recherchons échantillon par échantillon l'occurence du prochain passage à 0 pour relancer un décodage. Si la quantité de données restant à traiter après détection d'un start bit est inférieur à 400, alors nous copions en début de tableau les données restantes, et concaténons les nouvelles données pour relancer le traitement sur un jeu de données complet. Si aucune transmission n'est détectée sur les données acquises, la recherche de la transition du start bit atteint la fin du tableau et nous pouvons sereinement traiter le nouveau paquet de données sans craindre d'avoir coupé une transmission en cours. Le résultat de cette implémentation (Fig. 13) est visible sur la table 3.

22. `http://en.wikipedia.org/wiki/Asynchronous_serial_communication`



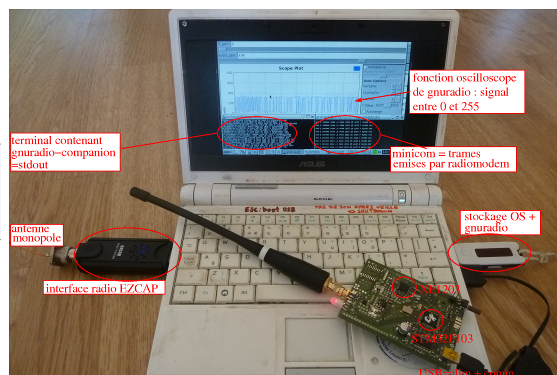


FIGURE 13 – Gauche : schéma bloc de `gnuradio-companion` pour le décodage des trames acquises par EZCAP et fournies après démodulation au bloc de traitement que nous avons développé et testé sur carte son. Noter le gain important (environ 3700) et le biais (128) qu'il faut ajouter pour ajuster la gamme des données issues du bloc de démodulation aux valeurs lues dans le fichier .wav exploité pour les traitements. Droite : montage expérimental, avec le radiomodem XE1203F commandé par un microcontrôleur STM32, et le module EZCAP pour l'écoute du flux de données émis.

```
N=1361
N=5444
N=6805 \fcUUUU\bc\e7\93\2
N=1361 2 .
N=682 0
N=1361 000
N=1361 0 4\fe
N=1361 6 3
N=1361 1 4\fd
N=4083 0 31 01138
N=5444 *7C12 188

N=5444
N=1361 \f0
N=1361 \f0
N=1361
N=2722
N=1361
N=1361
N=5444 \10\fc
N=6805 UUUU\bc\e7\93\822
```

FIGURE 14 – Le bloc compatible `gnuradio` équipé, pour le déverminage, de l'affichage d'une variable N contenant le nombre de données fournies par le bloc de démodulation : nous constatons que cette valeur évolue de façon plus ou moins aléatoire, et qu'il est donc impossible de faire une hypothèse sur la tailles des données à traiter. La solution proposée est décrite dans le texte.

5.2 Cas de l'ACARS – *attention, paragraphe addictif*

La philosophie que nous avons adopté ici pour convertir le code de traitement prototypé sous GNU/Octave vers un flux de données de taille aléatoire consiste à concaténer les données reçues pour former un bloc contenant l'intégralité des données, et de n'appliquer la phase de décodage que sur l'intégralité de ce bloc, pour revenir aux conditions déjà connues sous GNU/Octave. L'intérêt d'un bloc est déterminé par la convolution entre le signal acquis et un filtre centré sur 2400 Hz. Nous avons déjà vu que toute communication ACARS s'initie avec plusieurs périodes d'un signal à 2400 Hz pour laisser le temps au contrôle de gain automatique du récepteur radiofréquence de se stabiliser et synchroniser l'horloge du récepteur avec celle de l'avion. Ce signal sert donc à identifier le début d'un message. Ensuite, nous concaténons les données pour atteindre la taille maximale que pourrait atteindre un message ACARS (nous avions tenté de concaténer tant que la puissance du signal reçu dépasse un seuil prédéterminé, mais cette approche s'avère *très* sensible à la valeur du seuil de détection), soit 258 caractères compte

```
UUUU\bc\e7\fb\822 .0 \fc000 465 31 534 6002109
\c\f8\e0UUU\bc\e7\93\82\fe .0 00\fc0 558 31 500 7 02110
\fc\c0UUU\bc\e7\93\822 0 0000595 31 16\f9 8 02110
U\e0UUUU\bc\93\822 .0\fc0000 6\ff7 31 426 9 02110
\ff\b0E\e0UUUU\bc\e7\93\fe2 .0 0T00 225 31374 10 0210
\fc\f0UUUU\bc\e7\93\82\fe .0 000 333 31 24 12 02110
\ff\ff\e0\c1UUU\bc\e7\93\822 .0 0\bc00 203\bc31 419 13 02111
\ff\5\ff\f8\c0U5UU\bc\e7\93\822 .0 00\e40 367 \e71 246 14 02111
\f8\f0\e0UUU\bc\e7\93\822 .0 000\94 195 3\95 296 15 02111
\f8\f0UUUU\bc\e7\93\822 \fe0 0000\fc284 31 30\fd 17 02111
\ff\e0\5\f0\f0\e0UUUU\bc\e7\93\822 .0 0000\d02\f09 31 311 \859 02112
\ff\e0\f0\f0\e8\e0UUUU\bc\e7\93\822 .0 0000 336 31 \e741 20 021\e52
```

```
2 .0 0000 465 31 534 6 02109
2 .0 0000 558 31 500 7 02110
2 .0 0000 595 31 169 8 02110
2 .0 0000 637 31 426 9 02110
2 .0 0000 225 31 374 10 02110
2 .0 0000 333 31 324 12 02110
2 .0 0000 203 31 419 13 02111
2 .0 0000 367 31 246 14 02111
2 .0 0000 195 31 296 15 02111
2 .0 0000 284 31 305 17 02111
-> 2 .0 0000 201 31 325 18 02112
2 .0 0000 209 31 311 19 02112
2 .0 0000 336 31 341 20 02112
```

TABLE 3 – Comparaison des trames issues du décodage par `gnuradio` des signaux reçus par récepteur EZCAP (gauche), et les trames émises par le radiomodem. Les trames de gauche commencent par le signal de synchronisation UUUU, suivi d'un identifiant du modem émetteur, suivi de la trame. Bien que la conversion ne soit pas parfaite, la majorité des trames émises (droite) se retrouve dans les signaux décodés (gauche). Nous avons indiqué dans la table de droite une ligne préfixée par une flèche qui a complètement été omise lors du décodage par échec du décodage de la trame UUUU d'initialisation de la transmission.

tenu de la taille de texte la plus longue admissible.

La traduction du code GNU/Octave en C se fait alors sans trop de problème si ce n'est pour la convolution qui n'est efficacement implémentée que par le passage dans le domaine de Fourier. Heureusement, une bibliothèque vient à notre secours pour éviter de réimplementer ces fonctions : `libfftw3`.

> `gnuradio` exploite un environnement de compilation visant à être portable entre un*x, MacOS X, et MS-Windows, `cmake`, ou le *Cross Platform Make* (`www.cmake.org`). Alors qu'ajouter une bibliothèque dans un classique `Makefile` se résumait à ajouter le nom de la bibliotèque précédé de `-l`, `cmake` a pour ambition de trouver tout seul les bonnes bibliothèques aux bons emplacements, et ce quelquesoit l'environnement de développement. Pour ajouter `libfftw3`, nous avons du piocher dans le script de recherche de l'arborescence disponible à `http://code.google.com/p/qmcpack/source/browse/trunk/CMake/FindFFTW.cmake`, et compléter le `CMakeLists.txt` (pendant du classique `Makefile` de notre enfance) par `find_package(FFTW)` et `INCLUDE_DIRECTORIES($FFTW_INCLUDE_DIR)` pour la recherche d'entête et `LINK_LIBRARIES($FFTW_LIBRARIES)` pour la portabilité des bibliothèques.

```
[... entetes habituels ...]
#include <fftw3.h>
#define MAXSIZE 1000000
#define NSEARCH 260  // 13 periods * 20 pts/period

void remove_avg(unsigned char *d,int *out,int tot_len,int fil_len)
{int tmp,k,avg=0;
 for (k=0;k<fil_len;k++) avg+=d[k];    // initialise moyenne glissante
 for (k=0;k<tot_len-fil_len;k++) {out[k]=d[k]-avg/fil_len; avg-=d[k]; avg+=d[k+fil_len]; }
 for (k=tot_len-fil_len;k<tot_len;k++) out[k]=d[k]-avg/fil_len;
}
```

Cette première fonction de retrait de la valeur moyenne – rappelons que l'intercorrélation qui sera utilisée pour trouver la séquence de sinusoïdes à 2400 Hz suppose un signal de valeur moyenne nulle – retire le résultat d'une fenêtre glissante de `fil_len` éléments du tableau `d` pour générer `out`. Ce pré-traitement est fondamental pour toute la suite des opérations.

```
int main(int argc,char **argv)
{char filename[255];
 int deb,fin,seuil,k,i,fe=48000,f,N,t,*out,n,b=0,l0=0;
 unsigned char d[MAXSIZE],*toutd,*tout;
 double a=0.,*rs12,*rs24,*rc12,*rc24,c2400[20], c1200[20],s2400[20], s1200[20];
 fftw_complex *c2400x13,*fc2400x13,*fd,*s,mul,*ss;
 fftw_plan plan_a, plan_b, plan_R;

 if (argc<5) printf("%s filename deb fin seuil\n",argv[0]); else
   {sprintf(filename,"%s",argv[1]); deb=atoi(argv[2]); fin=atoi(argv[3]); seuil=atoi(argv[4]); }
 if ((fin-deb)>MAXSIZE) {fin=deb+MAXSIZE; fprintf(stderr,"deb=%d fin=%d\n",deb,fin); }
 f=open(filename,O_RDONLY); if (f<0) fprintf(stderr,"open error %d\n",f);
 k=lseek(f,deb,SEEK_SET);   if (k!=deb) fprintf(stderr,"lseek error %d\n",k);
 N=read(f,d,fin-deb); close (f);             // d=d(deb:fin); N=fin-deb;
 out=(int*)malloc(sizeof(int)*N);
 remove_avg(d,out,N,60);  // c=ones(60,1)/60; dm=conv(d,c);dm=dm(60/2:end-60/2); d=d-dm;
```

La lecture du fichier de points acquis par `gnuradio-companion` est triviale – un octet par donnée qui ne pose donc pas de problème d'*endianness* – et l'indice de début de la séquence analysée dans le fichier va nous servir par la suite à simuler le fait que le flux de données issue des démodulateurs de `gnuradio` peuvent avoir des tailles variables.

```
 c2400x13 = (fftw_complex *) fftw_malloc (sizeof (fftw_complex) * N);
[... idem pour initialiser fc2400x13, fd, s, ss ...]
 for (t=0;t<520;t++)  // t=[0:520]; c2400x13=exp(i*t*2400/fe*2*pi);
```

```
    {c2400x13[t][0]=cos((double)t*2400./fe*2*M_PI);
     c2400x13[t][1]=sin((double)t*2400./fe*2*M_PI);
    }
for (t=520;t<N;t++) {c2400x13[t][0]=0;c2400x13[t][1]=0;} // 13 periodes 2400 Hz
for (k=0;k<N;k++) {s[k][0]=(double)out[k];s[k][1]=0.;}
plan_a=fftw_plan_dft_1d(N, c2400x13, fc2400x13, FFTW_FORWARD, FFTW_ESTIMATE);
plan_b=fftw_plan_dft_1d(N, s, fd , FFTW_FORWARD, FFTW_ESTIMATE);
plan_R=fftw_plan_dft_1d(N, fd,ss, FFTW_BACKWARD, FFTW_ESTIMATE);
fftw_execute (plan_a); fftw_execute (plan_b);
for (k=0;k<N;k++)       // produit des transformees de Fourier pour intercorrelation
    {mul[0]=fc2400x13[k][0]*fd[k][0]-fc2400x13[k][1]*fd[k][1];
     mul[1]=fc2400x13[k][1]*fd[k][0]+fc2400x13[k][0]*fd[k][1];
     fd[k][0]=mul[0]/(float)N;
     fd[k][1]=mul[1]/(float)N;
    }
fftw_execute (plan_R);  // puis retour dans le domaine reel par FFT inverse
fftw_destroy_plan (plan_a); fftw_destroy_plan (plan_b); fftw_destroy_plan (plan_R); // s=conv(c2400x13,d);
```

L'appel aux fonctions de transformée de Fourier proposées par `fftw3`, opération fastidieuse à implémenter par ailleurs, passe par l'allocation de mémoire des divers tableaux exploités par les étapes intermédiaires de calcul de l'intercorrélation en passant dans le domaine de Fourier (pour rappel, nous avions déjà explicité dans ces pages [13] le gain en temps de calcul – de N^2 à $N\ln(N)$ – lors de la recherche d'un motif dans une séquence de points de taille N par intercorrélation). Les motifs recherchés sont définis sous forme de segments de sinusoïdes normalisés par la fréquence d'échantillonnage `fe=48000` Hz.

```
for (k=0;k<N-NSEARCH;k++) if (ss[k+NSEARCH-2][0]>a) {a=ss[k+NSEARCH-2][0];b=k;} // [a,b]=max(real(s));
printf("a=%f b=%d\n",a,b);
b=b%20;
```

Le maximum d'intercorrélation identifie l'indice, dans le temps, pour lequel le motif des sinusoïdes à 2400 Hz a été identifié dans la séquence de points expérimentaux. Cette identification est un point clé pour optimiser la suite des opérations car résoud l'incertitude sur la phase. En effet, nous avons vu dans la présentation des séquences I et Q issues des mélangeurs que deux paramètres inconnus sont l'amplitude et la phase du signal. L'intercorrélation permet d'identifier les deux paramètres, mais au prix d'un calcul lourd. Si la phase est connue, alors il ne reste que l'amplitude à identifier, et ceci ne nécessite que N multiplications en faisant glisser le motif d'une période de sinusoïde sur les points expérimentaux *par pas d'une période* (et non plus par pas d'un point d'échantillonnage). C'est ce que nous allons faire ci-dessous avec une période de sinusoïde à 2400 Hz et une demi-période à 1200 Hz, chacune encodant un état du flux de données numériques. Nous prendrons ensuite le module du produit de convolution pour éviter le cas de l'opposition de phase qui donnerait un résultat de magnitude élevée mais négatif.

```
for (t=0;t<20;t++)  // t=[0:520]; c2400x13=exp(i*t*2400/fe*2*pi);
    {c2400[t]=cos((double)t*2400./fe*2*M_PI); //  t=[0:20];  % 2400 Hz dans 48 kHz = 20 points/periode
     s2400[t]=sin((double)t*2400./fe*2*M_PI); //  c2400=exp(i*t*2400/fe*2*pi);
     c1200[t]=cos((double)t*1200./fe*2*M_PI); //  c1200=exp(i*t*1200/fe*2*pi);
     s1200[t]=sin((double)t*1200./fe*2*M_PI);
    }

rs12=(double*)malloc(sizeof(double)*(N-b)/20); // fin20=floor(length(s12)/20)*20;
[... idem pour rs24, rc12, rc24 ...]
l0=0;
for (k=b;k<N-20;k+=20)
    {rs12[l0]=0.; rs24[l0]=0.; rc12[l0]=0.; rc24[l0]=0.;
     for (t=0;t<20;t++)
       {rs24[l0]+=((double)out[k+t]*s2400[t]); rc24[l0]+=((double)out[k+t]*c2400[t]);
        rs12[l0]+=((double)out[k+t]*s1200[t]); rc12[l0]+=((double)out[k+t]*c1200[t]);
       }
    rs12[l0]=sqrt(rs12[l0]*rs12[l0]+rc12[l0]*rc12[l0]); rs24[l0]=sqrt(rs24[l0]*rs24[l0]+rc24[l0]*rc24[l0]);
    l0++;
    }
```

Les données brutes sont passées dans le filtre qu'est la convolution, il ne reste plus maintenant qu'à comparer les amplitudes des signaux issus de chaque filtre pour estimer si le bit est plus probablement un 1 ou un 0 :

```
l0=0;        // il faut liberer toutes les allocations de memoire (malloc) par free ... omis ici
do l0++; while ((rs24[l0]+rs12[l0])<2*seuil);  // cherche debut
do l0++; while ((rs24[l0]+rs12[l0])>2*seuil);  // cherche fin
fin=l0;
l0=0;
do l0++; while (rs24[l0]<seuil);     // l1=find(rs24>seuil);l1=l1(1);rs12=rs12(l1:end);rs24=rs24(l1:end);
do l0++; while (rs12[l0]<rs24[l0]); // l =find(rs12>rs24); l=l(1);  rs12=rs12(l:end); rs24=rs24(l:end);
toutd=(char*)malloc(fin-l0);
tout=(char*)malloc(fin-l0+2);
for (k=l0;k<fin;k++)  // pos12=find(rs12>rs24);pos24=find(rs24>rs12);toutd(pos12)=0;toutd(pos24)=1;
     if (rs24[k]>rs12[k]) toutd[k-l0]=1; else toutd[k-l0]=0;
```

et ayant un flux de données binaires contenu dans `toutd`, nous achevons le décodage en inversant l'état courant des bits selon les valeurs de `toutd` et affichons dans un premier temps les valeurs hexadécimales des octets décodés – une trame ACARS devant commencer par 2B 2A 16 16 01 – puis la séquence des caractères affichables en les interprétant par leur code ASCII.

```
n=0; tout[n]=1;n++;tout[n]=1;n++; // les deux premiers 1 sont oublie's car on se sync sur 1200
for (k=0;k<fin-l0;k++) {if (toutd[k]==0) tout[n]=1-tout[n-1]; else tout[n]=tout[n-1];
                        n=n+1;
                       }
  for (k=0;k<fin-l0;k+=8)
      printf("%02x ",tout[k]+tout[k+1]*2+tout[k+2]*4+tout[k+3]*8+tout[k+4]*16+tout[k+5]*32+tout[k+6]*64);
  printf("\n");
  for (k=0;k<fin-l0;k+=8)
      printf("%c",    tout[k]+tout[k+1]*2+tout[k+2]*4+tout[k+3]*8+tout[k+4]*16+tout[k+5]*32+tout[k+6]*64);
} // checksomme=1-mod(sum(binaire(:,1:7)')',2); % verification
```

Nous avons omis le calcul du bit de parité qui ne nous aide de toute façon pas à retrouver la valeur initiale

de l'octet corrompu lors de la transmission, les codes correcteurs permettant ce genre de manipulation étant bien plus complexes qu'un simple bit en fin de chaque octet.

Ayant validé que le code en C fournit des résultats cohérents avec le script GNU/Octave observé initialement, nous incluons ce programme dans l'environnement C++ imposé par `gnuradio-companion` tel que vu auparavant. La FFT nécessitant un nombre minimum de données à traiter, nous avons fait le choix de, plutôt que concaténer juste le morceau final de données qui n'a pu être décodé tel que nous l'avions vu pour la radiomodem XE1203, accumuler autant de points que le nombre maximum de caractère que peut contenir un message ACARS, à savoir 258 caractères. Le décodage des 13 séquences adjacentes d'oscillations à 2400 Hz se fait en continu sur le flux de données acquis (avec ici la contrainte d'un nombre minimum de 20 points/période×13 périodes=260 points), et lorsqu'une condition de seuil est atteinte (signifiant un début de trame), nous accumulons 40 points/bit×8 bit/caractère×258 caractères soit environ 83000 points. L'algorithme de décodage que nous avons implémenté en C s'applique alors à ces 83000 points comme s'ils avaient été lus dans un fichier (Fig. 15). Nous avons cependant observé une très grande sensibilité du décodage aux valeurs de seuils sélectionnés, un point peu satisfaisant qui mérite à être amélioré par l'utilisation de contrôles de gains automatiques. Le message décodé diffère légèrement de ce qui est obtenu avec le programme en C ci-dessus et contient

```
+*2.HB-JZTH2-M10DDS39AZ0G    N47307E0060672854M380240049G    N47317E0060012986M410240050G
N47333E005JLGLOHLrTLMMLFOJN8____1KHLJ2EPOJKHIL.50145524@0%0G____1K0381E00JKNGLML[2KHZ=C+0508
N4H+AJ90053&4LC12MT&2""80:*8_  N$4DK7E@_Z31J@
```

émis par un Airbus A319 (HB-JZT) de Easyjet suisse (cohérent pour une écoute effectuée à Besançon) et semble contenir des transmission de coordonnées (47,307 N, 6.607 E correspond bien au Nord-Est de la Franche-Comté) dont nous ne connaissons pas la signification. La fin du message est probablement corrompue.

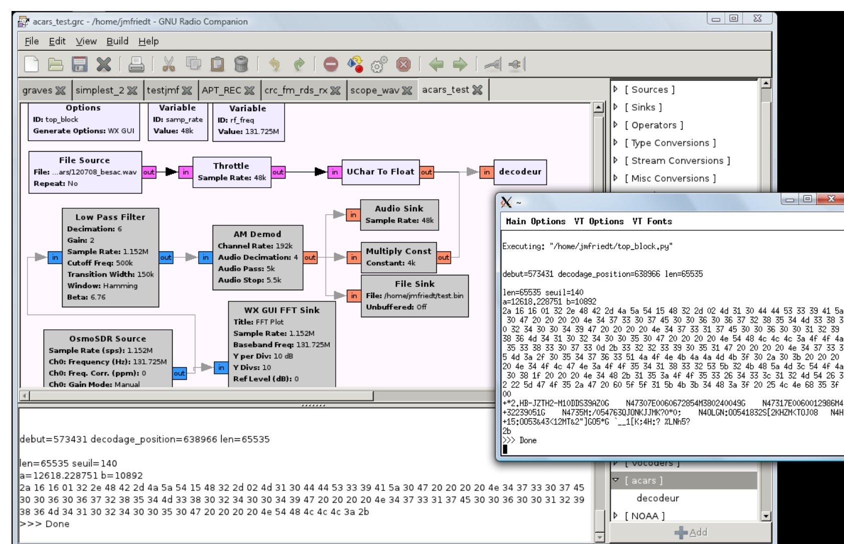


FIGURE 15 – Décodage par un bloc `gnuradio-companion` implémentant les algorithmes de décodage décrits dans le texte d'une trame ACARS enregistrée dans un fichier audio échantillonné à 48 kéchantillons/s. Remplacer le fichier audio par la sortir du bloc `osmosdr` pour un traitement en temps réel (blocs en gris foncé) des données permet de bien identifier le début de transmission mais le décodage du message échoue souvent, probablement du fait d'un mauvais ajustement des seuils.

Un exemple de décodage en temps réel du flux de données transmis par le bloc décrit dans ce paragraphe est (tout d'abord la valeur des octets reçus, puis leur interprétation dans le code ASCII pour les caractères affichables)

```
63 70 5c 72 27 7d 04 09 5c 42 43 26 7e 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f
2b 2a 16 16 01 47 2e 45 49 2d 44 54 4e 15 48 31 35 02 44 35 33 43 41 5a 30 32 34 33 23 44 46 42 33 2c 33
39 2c 30 35 39 2c 30 34 35 39 39 2c 30 31 33 36 2f 56 31 30 32 2c 31 32 2c 32 32 32 2c 32 39 32 2c 30 33
2c 30 31 2c 30 30 30 30 30 2f 56 32 30 32 2c 30 38 2c 30 38 31 2c 30 36 39 2c 30 30 2c 30 30 2c 30 30 30
```

```
30 30 2f 56 33 30 32 2c 30 36 2c 31 34 36 2c 33 31 32 2c 30 36 32 32 2f 56 34 31 30 2c 30 36 2c 32 32 33
2c 30 32 35 2c 30 36 32 31 2f 56 35 30 30 2c 30 30 2c 30 30 30 2c 30 30 30 2c 30 30 30 30 2f 56 36 30 30
2c 30 30 2c 30 30 30 2c 30 30 30 2c 30 30 30 30 2f 56 37 30 34 31 2c 30 38 37 2c 30 30 30 36 31 2c 32 32
32 32 32 32 32 32 32 32 32 47 31 31 2f 56 38 30 34 31 2c 30 38 36 2c 30 30 30 38 31 17 24 3e 7f
cp\r'}\BC&~+*G.EI-DTNH15D53CAZ0243#DFB3,39,059,04599,0136/V102,12,222,292,03,01,00000/V202,08,081,069,00,
00,00000/V302,06,146,312,0622/V410,06,223,025,0621/V500,00,000,000,0000/V600,00,000,000,0000/V7041,087,00
061,22222222222G11/V8041,086,00081$>
```

Nous constatons que la séquence acquise est trop longue avant le préambule de 0x2b 0x2a 0x16 0x16, avec la séquence des 0x7f qui correspond aux octets à 2400 Hz pour la synchronisation du récepteur avec l'émetteur. Néanmoins, le décodage s'effectue bien, avec l'identification d'une trame émise par EI-DTN, un A320 de Alitalia qui cette fois volait au cours du trajet AZ0243 entre Londres et Milan (cohérent avec une réception depuis Paris vers 21h15 alors que cet avion est supposé partir, en l'absence de retard, de Londres à 19h50 pour arriver à Milan vers 21h45). Ici encore, le contenu du message semble cohérent mais incompréhensible.

Ainsi, sans être parfaite, notre implémentation du décodage en temps réel des trames ACARS sous `gnuradio-companion` permet d'obtenir l'identification de la majorité des avions passant à proximité du récepteur, et dans certains cas des messages aussi longs que

```
+*R.OO-SNBH19COOASNO7LR#CFBFLR/FR12081915430036110006PRESS REG-V 4001HA1 OR  SOL 10HA1
OR SENSE LINE/IDBMC 1!
```

Nous ne savons pas ce que "appuyer sur le bouton REG-V" effectue (apparemment une histoire de valve de régulation de la pression de la cabine), mais cela semble important pour que le pilote en informe le sol, et est un message récurent que nous retrouvons dans diverses archives de messages ACARS sur le web ! Ces résultats ne semblent pas significativement plus mauvais que les décodeurs exploitant l'acquisition par carte son de la sortie d'un scanner radiofréquence que nous avons testé (le maintenant défunt KRACARS [23] notamment, pour DOS).

Les perspectives d'améliorations du récepteur de mode numérique ACARS tiennent probablement en une détection automatique d'un paramètre de seuil qui nous sert à identifier le premier bit du premier octet, dont toute la suite du message découle. Ce seuil est actuellement défini de façon statique, une façon peu apte à s'adapter aux vastes gammes de conditions de réception radiofréquence rencontrées en pratique. Par ailleurs, la découpe des trames (identifiant de l'avion, identifiant du vol, texte transmis) est actuellement embryonaire et n'a que pour vocation de faciliter la prise en main du logiciel de décodage par l'utilisateur néophyte qui pourrait être rebuté par la lecture des séquences brutes d'octets. Toujours suivant les préconisations de `http://www.scancat.com/Code-30_html_Source/acars.html`, après avoir validé les 5 octets de synchronisation et d'entête (SOH, ASCII 0x01) en début de trame, nous extrayons l'identifiant de l'avion (octets 6 à 12), l'identifiant de début de texte (STX, code ASCII 0x03) avant de rechercher l'identifiant du vol (octets 22 à 27) qui sont les informations les plus utiles. Ensuite, dans le meilleur des cas, si le message est décodable, les caractères ASCII affichables au-delà du 28ème octet sont fournis.

Ainsi, le message brut

```
2b 2a 16 16 01 32 2e 2e 48 42 4a 4b 4c 15 31 37 30 02 4d 33 33 41 47 53 30 38 37 31 50 4f 41 30 31 47 53 30 38
37 31 2f 32 37 32 37 31 39 33 34 4c 53 47 47 45 4e 5a 56 2f 4e 34 37 20 31 37 2e 31 2f 45 20 20 36 20 31 2e 35
20 2f 32 36 32 2f 20 34 33 2f 33 30 32 2f 2d 20 33 39 2f 41 55 54 4f 52 50 2f 31 35 03 43 6a 7f
+*2..HBJKL170M33AGS0871POA01GS0871/27271934LSGGENZV/N47 17.1/E  6 1.5 /262/ 43/302/- 39/AUTORP/15Cj
```

se traduit en

```
Aircraft=..HBJKL
STX
Seq. No=4d 33 33 41 M33A
Flight=GS0871
POA01GS0871/27271934LSGGENZV/N47 17.1/E  6 1.5 /262/ 43/302/- 39/AUTORP/15ETX
```

soit un Falcon 2000 répertorié par les autorités suisses [24] comme basé à Genève, entendu vers 21h45 depuis Besançon, avec l'identifiant de vol GS0871, de nouveau fournissant des coordonnées GPS en accord avec la localisation du récepteur. Le lecteur possédant entre 15 et 20 millions de dollars pourra considérer de s'acheter ce jouet d'occasion. D'un autre côté, un tel investissement permet de réaliser une

23. `http://www.qsl.net/g4hbt/zips/krcrs12.zip`
24. `http://www.bazl.admin.ch/fachleute/luftfahrzeugregister/index.html?lang=en`

matrice de 1000×1000 récepteurs EZCAP et ainsi de former un récepteur RADAR à balayage de faisceau (`http://www.haarp.alaska.edu/`), autrement plus intéressant qu'une boite de conserve volante.

Par ailleurs, on notera une certaine cohérence dans la séquence des messages

```
Mon Aug 27 22:03:31 2012
Aircraft=.PH-XRX
STX
Seq. No=53 35 32 41 S52A
Flight=HV5078
ETX
Mon Aug 27 22:04:30 2012
Aircraft=.PH-XRX
STX
Seq. No=53 35 33 41 S53A
Flight=HV5078
ETX
Mon Aug 27 22:09:00 2012
Aircraft=.PH-XRX
STX
Seq. No=53 35 35 41 S55A
Flight=HV50HG
...
```

indiquant que cette information (S52A, S53A, S55A) est convenablement décodée.

Si le lecteur, en lisant ce code et en le mettant en œuvre, laisse tourner son PC la nuit dans l'espoir de recevoir le message d'un avion excessivement en retard, ou se réveille le matin en sursaut pour savoir combien de passages d'aéronefs il a enregistré – le mal est fait, le virus de la réception de signaux radiofréquences a infecté une nouvelle cible, le seul remède est de ... lire le code pour l'améliorer et augmenter ses performances de décodage.

5.3 Passage de paramètres à l'instanciation

Le dernier point qui peut faciliter la vie du développeur est de pouvoir changer, en remplissant le champ correspondant dans le bloc `gnuradio-companion`, des paramètres des blocs utilisés dans l'application, sans être obligé de recompiler la bibliothèque correspondante. Nous constatons que de nombreux blocs acceptent des arguments (valeurs numériques, chaînes de caractères) et nous désirons exploiter cette possibilité. Tout commence par la définition des arguments et l'ajout des prototypes associés dans l'interface graphique (répertoire `grc`), *i.e.* le fichier XML descriptif du bloc. Les balises définissant un passage de variable sont `<param> ... </param>` qui prennent comme champs le nom de la variable associée `<key>` et la nature de cette variable, par exemple un flottant `<type> real </type>`. Cette déclaration de variable doit *impérativement* se situer avant le paramètre `<sink>` sous peine d'obtenir un message d'erreur (cryptique) au chargement du bloc. Il reste dans le fichier XML à passer la variable au programme C++ en complétant la méthode `<make>` par le paramètre, dans notre cas `acars.decodeur($seuil)` (avec `seuil` le nom de la variable tel que définit dans `key`).

Du point de vue de la déclaration de la classe (répertoire `include`), la signature de la fonction d'entrée ainsi que celle du constructeur de la classe doivent être adaptées pour contenir l'argument `acars_make_decodeur (float seuil1);` dans toutes ses occurences. Finalement, le constructeur, au niveau de l'implémentation, de la classe C++ elle-même (répertoire `lib`) reçoit l'argument sous forme de `acars_decodeur::acars_decodeur (float seuil1)`. Dans notre cas, une consigne de seuillage est ainsi transférée depuis l'interface graphique de `gnuradio-companion` vers la classe C++ sans nécessiter de recompilation de la bibliothèque (bloc).

Un exemple simple de passage d'un unique paramètre est facilement accessible dans l'archive de `gnuradio` sous `gnuradio/gr-digital/*/digital_diff_decoder_bb.*` : la recherche de l'occurence du terme `modulus` permet de se retrouver dans le cheminement de la variable entre les divers fichiers définissant le bloc, et de s'en inspirer pour notre propre application (Fig. 16).

5.4 Passage dynamique de paramètres

Le passage de paramètre selon la méthode proposée ci-dessus définit la valeur de l'argument (`seuil`) au moment du lancement de l'application `gnuradio-companion` par passage de paramètre au constructeur de la classe associée au bloc. Une fois ce paramètre fourni, il est impossible de le modifier depuis

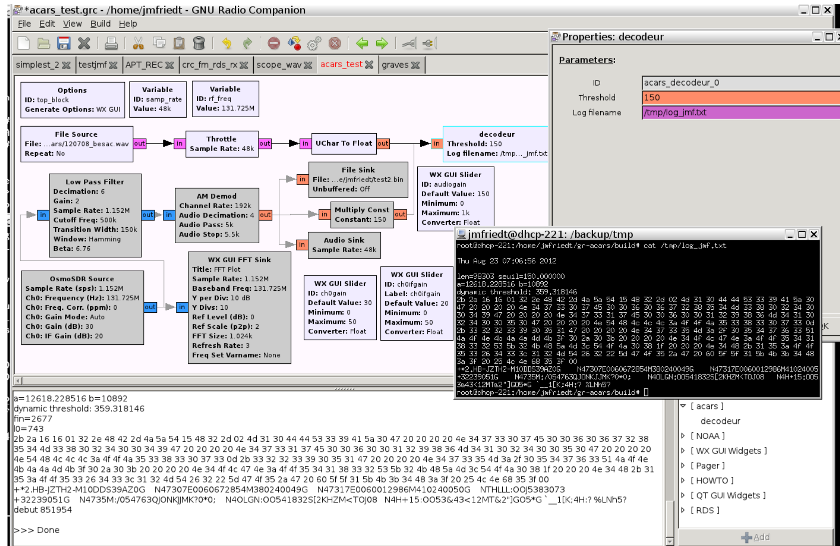


FIGURE 16 – Le bloc complété par le passage de deux arguments que sont une valeur de seuil (flottant) et le nom de fichier d'enregistrement des données acquises (chaîne de caractères). Nous constatons dans l'`xterm` que les données sont bien stockées dans l'emplacement annoncé, et les valeurs sont cohérentes avec celles affichées par `gnuradio-companion`.

l'application en cours d'exécution. Cependant, de nombreux bloc permettent de faire varier des variables depuis des composants graphiques représentant des ascenceurs (*sliders*) : comment font-ils ?

Un argument additionnel à la déclaration du bloc dans le fichier XML permet de définir une méthode pour mettre à jour une variable membre lorsque le paramètre qui lui est associé dans l'interface graphique change de valeur : dans le fichier contenu dans le répertoire `grc`, nous complétons la définition du bloc par le nom de la méthode appelée pour la mise à jour du paramètre (argument `callback`)

```
<make>acars.decodeur($seuil,$filename)</make>
<callback>set_seuil($seuil)</callback>
<param>
  <name>Threshold</name>
  <key>seuil</key>
  <type>real</type>
</param>
```

dont la signature associée est déclarée dans le fichier de définition de l'interface de la classe (répertoire `include`) :

```
public:
        ~acars_decodeur ();
        void set_seuil(float seuil1);
```

et finalement dans la bibliothèque, nous implémentons la méthode (*setter*) qui modifie le contenu de la variable membre (privée) :

```
void acars_decodeur::set_seuil(float seuil1)
{printf("new threshold: %f\n",seuil1);fflush(stdout);_seuil=seuil1;}
```

Ainsi, nous vérifierons que chaque fois que la valeur du paramètre `seuil` est modifiée en faisant glisser un ascenceur dans l'interface graphique, la valeur est bien modifiée dans le bloc de traitement de l'information correspondant (Fig. 17).

L'archive finale de ce développement (Figs. 18 et 19), incluant tous les concepts développés dans cet article, est disponible à `http://jmfriedt.free.fr/gr-acars.tar.gz` ou sur le site de CGRAN à `www.cgran.org/wiki/ACARS`.

6 Conclusion

Nous nous sommes efforcés de proposer une démarche de travail pour exploiter un récepteur radiofréquence fournissant un flux de données I et Q en vue de décoder des trames numériques. Nous nous sommes en particulier concentrés sur les périphériques faible coût centrés sur le composant E4000 couplé au convertisseur analogique-numérique RTL2832U.

Au-delà de la simple utilisation des blocs de traitement de signaux fournis avec l'outil `gnuradio`, nous

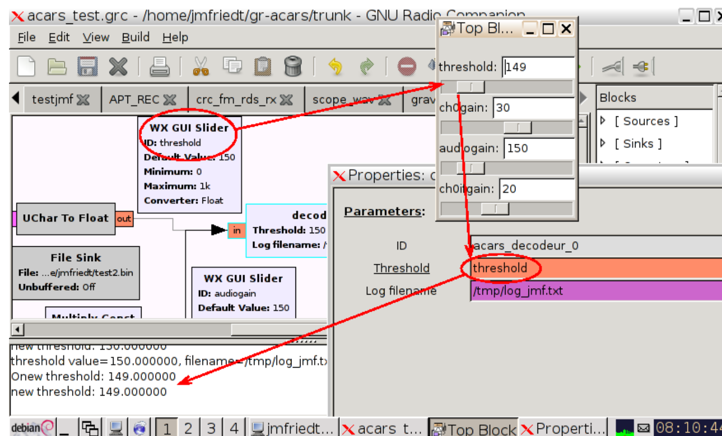


FIGURE 17 – Le bloc de décodage des trames ACARS accepte désormais la définition du paramètre seuil au moyen d'un ascenceur (bloc *slider*) avec modification de la valeur en cours d'exécution. Dans cet exemple, nous constatons qu'un ascenceur associé à la variable `threshold` a été défini avec une valeur par défaut de 150 (tel qu'annoncé dans le terminal en bas de `gnuradio-companion`), et changer la valeur dans l'onglet associé de l'interface graphique change bien la valeur du paramètre exploité lors du décodage des trames.

avons identifié le cheminement du flux de données afin de développer nos propres blocs de traitement. Nous avons démontré la capacité à implémenter des traitements en temps réel sur le flux d'informations issu du récepteur en prototypant dans un premier temps sous GNU/Octave sur des enregistrements, avant de convertir l'algorithme en C pour inclusion dans `gnuradio`.

Parmi les perspectives, l'exploitation sur des signaux plus "intéressant" que sont le GPS [25] ou le GSM serait ambitieuse mais certainement source de nombreuses découvertes.

Sans relation directe avec cette présentation, un collègue d'un des auteurs (JMF) nous faisait remarquer récemment comment l'époque de Capitaine Crunch et de la naissance d'Apple devait être une époque bénie du *hack*. Nous profitons de ces pages pour exprimer notre désaccord avec cette nostalgie : la disponibilité de puissances de calcul quasi-infinies, pour un coût abordable pour une majorité de la population d'Europe occidentale et d'Amérique du Nord, un mode de communication rapide et sans censure pour les domaines et les régions qui nous concernent, et des documentations ou échantillons de composants électroniques facilement disponibles, font de l'époque actuelle une période au moins aussi excitante pour le *hack* que l'époque des blue box et autres activités de *phreaking* [14], comme nous avons tenté de le démontrer dans cet exposé. La légende du détournement d'un sifflet pour communiquer avec les routeurs téléphoniques est certes attractive, mais il n'y a aucun doute que cette découverte fortuite était précédée d'heures d'analyses des protocoles de communication et d'études fastidieuses des conventions de communication des réseaux téléphoniques d'antans (John Draper n'était-il pas ingénieur en dehors de ses activités de compréhension des systèmes téléphoniques) : un bon *hack* est impossible sans une compréhension détaillée de la technologie sous-jacente. Nous osons copier ici les paroles reproduites du site de Wikipedia [26], sans en vérifier l'origine, pour leur sagesse [27] :

> I don't do that. I don't do that anymore at all. And if I do it, I do it for one reason and one reason only. I'm learning about a system. The phone company is a System. A computer is a System, do you understand ? If I do what I do, it is only to explore a system. Computers, systems, that's my bag. The phone company is nothing but a computer.

25. `http://michelebavaro.blogspot.fr/2012/04/spring-news-in-gnss-and-sdr-domain.html` mais surtout `http://www.gnss-sdr.org/documentation/gnss-sdr-operation-realtek-rtl2832u-usb-dongle-dvb-t-receiver` démontrent le résultat impressionant d'utiliser un récepteur de télévision numérique terrestre pour recevoir des signaux issus de satellites situés à une distance de 20000 km.
26. `http://en.wikipedia.org/wiki/John_Draper`
27. `http://www.slate.com/articles/technology/the_spectator/2011/10/the_article_that_inspired_steve_jobs_secrets_of_the_little_blue_.single.html` (lu en Septembre 2012).

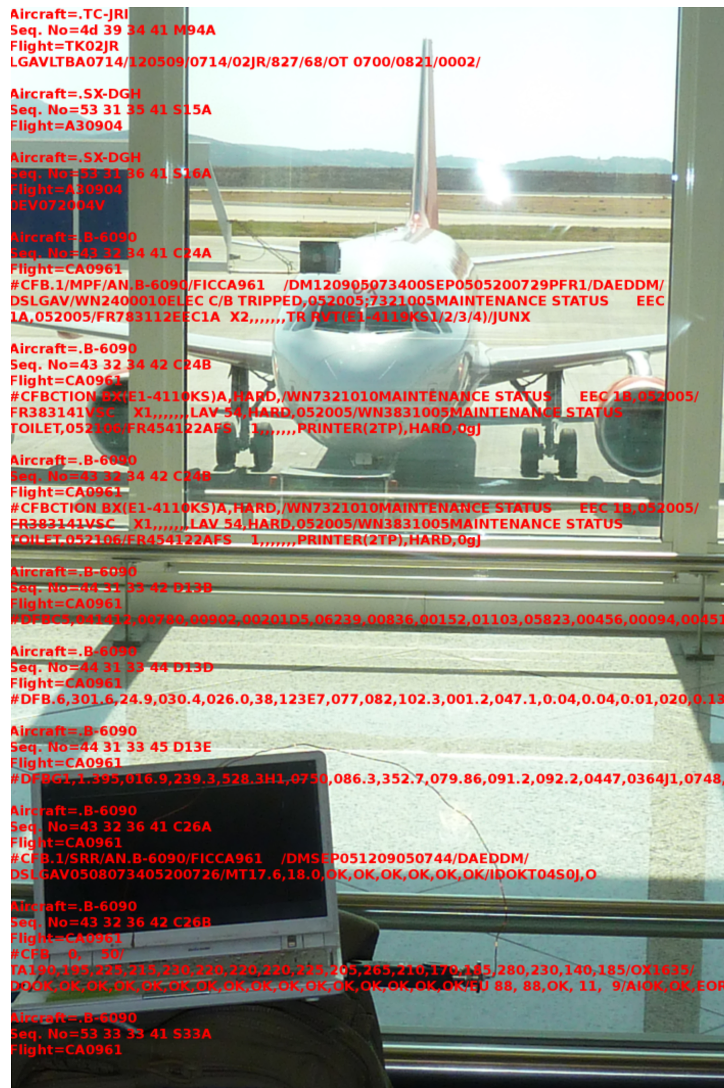


FIGURE 18 – Exemple de trames ACARS acquises au moyen d'un récepteur EZCAP muni d'une antenne formée d'un bout de fil de cuivre émaillé de ≃50 cm de longueur. Cette activité rend les heures d'attente dans les aéroports beaucoup plus ludiques, ici à Athènes (Grèce).

Remerciements

D. Bodor nous a informé de l'existence des récepteurs DVB à base de E4000+RTL2832U et de l'engouement qu'ils ont suscité dans la communauté des développeurs de SDR. Les interlocuteurs de la mailing list `osmocom-sdr` ont levé un certain nombre d'incertitudes sur les données techniques des récepteurs radiofréquences à base de RTL2832U et E4000. Le gouvernement équatorien mérite toute notre gratitude, ne serait-ce que pour son hébergement de `http://gen.lib.rus.ec/` où nous obtenons la majorité des documents de la bibliographie, et autres actions politiques qui dépassent le cadre de cet article. J. Boibessot (Armadeus Systems) et É. Carry (FEMTO-ST) ont, par leurs corrections lors de relectures, amélioré le manuscrit.

Références

[1] K. Borre, D.M. Akos, N. Bertelsen, P. Rinder & S.H. Jensen, *A Software-Defined GPS and Galileo Receiver : A Single-Frequency Approach*, Birkhäuser Boston (2007) ainsi que les transparents à `http:`

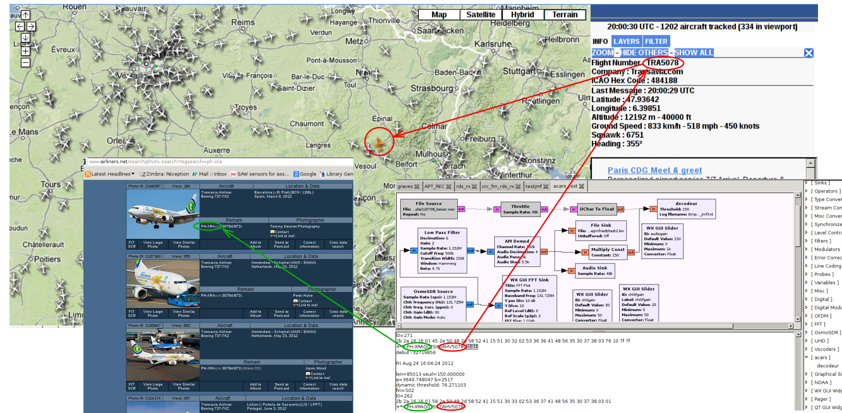


FIGURE 19 – Analyse des trames ACARS obtenues en temps réel par notre module `gr-acars` : le site web `radarvirtuel.com/index-fr.html` valide la présence, à proximité de Besançon (position de l'antenne réceptrice), d'un avion de Transavia, tandis que le site `airliners.net` associe une immatriculation d'avion (PH-RXA) à un exploitant. Les informations acquises par ACARS sont donc cohérentes avec les autres données disponibles sur les sites web associés au suivi du traffic aérien, et fournit un complément idéal au décodeur ADS-B présenté au début de ce document.

`//kom.aau.dk/project/softgps/` et en particulier `http://kom.aau.dk/project/softgps/GNSS_SummerSchool_DGC.pdf`

[2] J. Hamkins & M.K. Simon, *Autonomous Software-Defined Radio Receivers for Deep Space Applications*, Deep Space Communications and Navigation Series Vol. 9, disponible à `http://descanso.jpl.nasa.gov/Monograph/series9/Descanso9_Full_rev2.pdf`

[3] P.B. Kenington *RF and baseband techniques for software defined radio* Artech House (2005)

[4] la littérature sur SDR pullule d'ouvrages plus ou moins intéressants. Parmi ceux que nous avons trouvé de peu d'intérêt car loin de la pratique, W. Tuttlebee, *software defined radio – enabling technologies* Wiley (2002), ou encore M. Dillinger, K. Madani, N. Alonistioti, *Software Defined Radio – Architectures, Systems and Functions* Wiley (2003), que le lecteur pourra éviter sans craindre de rater trop d'informations.

[5] J.-M Friedt & S. Guinot, *La réception d'images météorologiques issues de satellites : principes de base*, GNU/Linux Magazine France, Hors Série 24 (Février 2006), disponible à `http://jmfriedt.free.fr/LM_sat1.pdf`

[6] Note d'application AN2668, *Improving STM32F101xx and STM32F103xx ADC resolution by oversampling*, ST Microelectronics, 2008

[7] D.J. Mudgway, *Uplink-Downlink – A History of the Deep Space Network, 19571997*, NASA SP-2001-4227, The NASA History Series (2001), disponible à partir de `history.nasa.gov/SP-4227/Uplink-Downlink.pdf`

[8] M.K. Simon, *Bandwidth-Efficient Digital Modulation with Application to Deep-Space Communications*, Deep Space Communications and Navigation Series Vol. 3, disponible à `http://descanso.jpl.nasa.gov/Monograph/series3/complete1.pdf`

[9] T.G. Thomas & S. Chandra Sekhar, *Communication Theory*, Tata Mcgraw Hill Publishing (2006)

[10] N. Foster, *Tracking Aircraft With GNU Radio*, GNU Radio Conference (2011), disponible à `http://gnuradio.org/redmine/attachments/download/246/06-foster-adsb.pdf`

[11] T. McDermott, *Wireless Digital Communications : Design and Theory 2nd Ed.*, Tucson Amateur Packet Radio Corporation – TAPR (1998)

[12] Agilent, *Digital Modulation in Communications Systems – An Introduction*, Application Note 1298 disponible à `cp.literature.agilent.com/litweb/pdf/5965-7160E.pdf`, ou M. Steer, *Microwave and RF design – a systems approach*, SciTech Publishing, Inc (2010) dont le premier chapitre

est disponible à `http://www.ece.ucsb.edu/yuegroup/Teaching/ECE594BB/Lectures/steer_rf_chapter1.pdf`

[13] J.-M Friedt, *Auto et intercorrélation, recherche de ressemblance dans les signaux : application à l'identification d'images floutées*, GNU/Linux Magazine France 139 (Juin 2011), disponible à `http://jmfriedt.free.fr/`

[14] S. Wozniak & G. Smith, *iWoz : Computer Geek to Cult Icon : How I Invented the Personal Computer, Co-Founded Apple, and Had Fun Doing It*, W. W. Norton & Company (2007)