Un oscilloscope pour le traitement de signaux radiofréquences : `gr-oscilloscope` pour GNU Radio 3.7 et 3.8

J.-M Friedt, G. Goavec-Merou, 21 décembre 2019

GNU Radio, environnement libre de traitement de signaux radiofréquences, encourage le développement de blocs de traitements dédiés (*Out Of Tree modules – OOT*). Alors qu'une multitude d'interfaces matérielles dédiées sont supportées par GNU Radio [1], elles ont toutes en commun de fournir un flux de données continu de bande passante limitée par les débits de communication vers l'unité de traitement – Ethernet ou USB le plus souvent. Lors de traitements RADAR de distance de cibles, deux conditions sont respectées par les oscilloscopes radiofréquences : le flux de données n'a pas besoin d'être continu, nous n'avons besoin que de traiter une profondeur mémoire déterminée par la plus grande distance (temps de vol) de la cible à détecter; et plus la bande passante est grande, meilleure est la résolution. Une bande passante B d'un GHz permet d'espérer une résolution en distance $\Delta R = c/(2B)$ de l'ordre de 15 cm, objectif typique pour des mesures de glissement de terrain par exemple. Or aucune interface de radio logicielle accessible au grand public ne fournit une telle bande passante – alors que "n'importe quel" oscilloscope radiofréquence fournit actuellement allègrement quelques Géchantillons/s. De plus, nombre d'oscilloscopes fournissent 4 voies, permettant donc 4 flux d'acquisition, propices aux mesures de direction d'arrivée des signaux.

Fort de ce constat nous convainquant de la pertinence d'un oscilloscope pour le traitement de signaux radiofréquences, nous désirions interfacer les oscilloscopes communiquant par interface Ethernet selon le protocole VXI11 [2] à GNU Radio pour faciliter l'acquisition et le prétraitement des données avant d'alimenter GNU/Octave pour les traitements plus poussés.

Alors que ce projet végétait depuis un peu plus d'1,5 an, GNU Radio 3.8 vient d'être publié : le bloc d'acquisition de signaux d'oscilloscopes `gr-oscilloscope` [3] sera donc non-seulement l'opportunité d'apprendre à écrire un bloc avec un nombre variable de sources (1 à 4 voies) paramétrable par l'utilisateur et communiquant par VXI11, mais aussi de découvrir les quelques subtilités de GNU Radio 3.8 et nous rendre compte que finalement, le passage du vénérable 3.7 à 3.8 ne sera pas aussi douloureux que le passage à Qt5 et Python3 laisse penser. Les dépôts des programmes présentés dans cet article sont disponibles à `http://github.com/jmfriedt/gr-oscilloscope` pour GNU Radio 3.7 et `http://github.com/jmfriedt/gr-oscilloscope38` pour GNU Radio 3.8 auxquels le lecteur se référera pour mettre en contexte les extraits de code décrits ici.

1 Principes généraux

La plupart des oscilloscopes radiofréquences ont aujourd'hui plusieurs Méchantillons de profondeur mémoire. L'expérience montre que l'ordonnanceur GNU Radio préfère manipuler des paquets de quelques milliers d'échantillons tout au plus. Notre philosophie sera donc de mémoriser des gros paquets de données dans le bloc source que nous allons développer, et ensuite distiller ces données *acquises simultanément sur toutes les voies de l'oscilloscope* selon les besoins de GNU Radio. Lorsque la réserve de données sera épuisée car traitée par GNU Radio, nous requérons un nouveau paquet de données. L'hypothèse de synchronisme de toutes les voies de l'oscilloscope est un point clé pour le traitement ultérieur de mesures RADAR, et nous avons eu quelques déboires avec les modèles les plus anciens d'oscilloscopes qui désynchronisaient parfois d'un échantillon deux voies de mesure, mais les instruments les plus récents ne semble plus affectés par de tels dysfonctionnement.

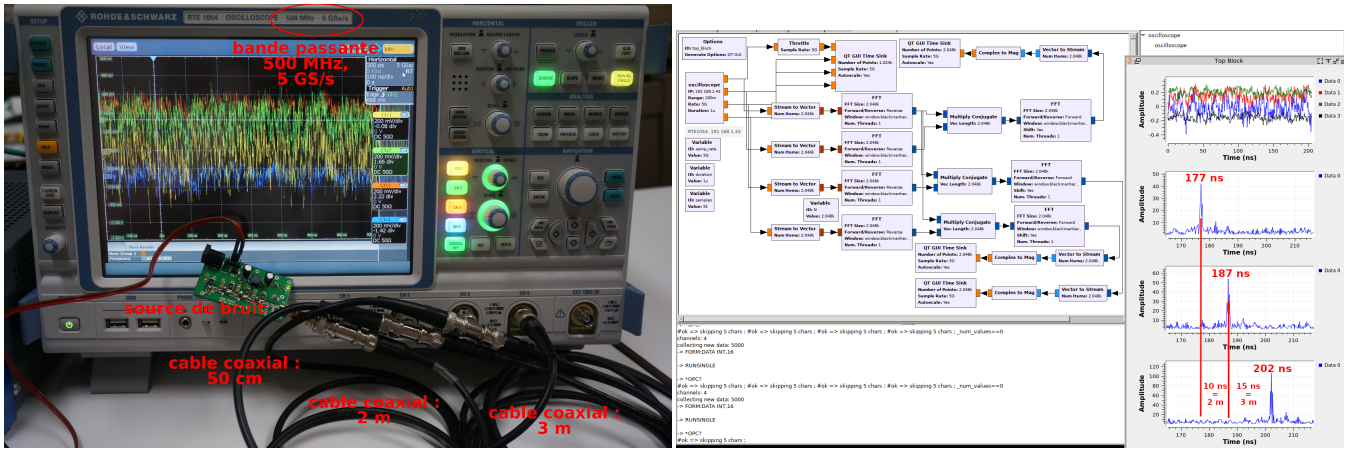


FIGURE 1 – Objectif de l'expérience : mesure du temps de vol d'une onde électromagnétique dans un cable coaxial, opportunité d'illustrer le principe du RADAR à bruit et sa mise en œuvre par un oscilloscope multi-voies radiofréquence.

VXI11 est un protocole qui s'appuie sur RPC (*Remote Procedure Calling* [4]), lui même un protocole permettant d'exécuter des procédures sur les ordinateurs distants connectés par réseau, facilitant ainsi les échanges entre processus (*Inter*

Process Control – IPC), que ce soit par échanges UDP ou TCP selon que la vitesse des échanges ou leur intégrité soit favorisée. Nous ne nous intéresserons pas nécessairement à l'implémentation puisqu'une bibliothèque parfaitement fonctionnelle est disponible [5]. Notre objectif sera donc d'envelopper les appels aux commandes GPIB (*General Purpose Interface Bus*) par protocole VXI11 (implémentation de LXI pour *LAN-based eXtensions for Instrumentation*) vers l'oscilloscope dans un bloc GNU Radio afin de fournir un flux de données compréhensible par l'ordonnanceur qui alimentera ensuite les blocs de traitement ultérieurs. Plutôt que nous focaliser sur une application de RADAR qui nécessite d'ajouter aux entrées de l'oscilloscope des amplificateurs, voir une chaîne de transposition de fréquence que le lecteur n'a pas forcément envie d'assembler pour expérimenter, nous nous intéresserons à une **application de réflectométrie** [6] grâce à laquelle nous mesurerons avec précision la longueur de câbles coaxiaux, et ce en y injectant une source de bruit large bande. Une telle application pourra rappeller le principe du RADAR à bruit, dans lequel une source de bruit est émise **tout en étant enregistrée par une voie de référence**, et la mesure sur une seconde antenne (ou la sortie d'un circulateur dans une configuration monostatique) permet, par corrélation, de trouver le temps de vol avec une résolution temporelle inversement proportionnelle à la bande passante du signal émis (Fig. 1). Le bénéfice de cette approche, par rapport au RADAR impulsionnel classique, est que générer une impulsion brève (donc de fort encombrement spectral, puisque la bande passante de l'impulsion est de l'ordre de l'inverse de sa durée) et énergétique est techniquement complexe : l'intégration sur une longue durée lors de la corrélation du bruit avec le signal mesuré nous permet d'intégrer la puissance du bruit sur une longue durée, et l'énergie étant l'intégrale de la puissance par la durée, nous obtenons l'effet équivalent d'une impulsion énergétique mais technologiquement simple à réaliser. Les RADARs à bruit ont par ailleurs le bon goût de se glisser discrètement dans le fouilli des émissions électromagnétiques et d'être difficile à repérer, un point positif lorsque nous expérimentons en espace libre.

Nous avions largement développé dans ces pages [7] le principe du RADAR passif qui s'apparente au RADAR à bruit dans lesquels une source large bande *a priori* inconnue est enregistrée. Pour rappel, la mesure de retard induit par l'aller-retour du signal entre l'émetteur, la cible supposée statique dans ces exemples, et le récepteur est alors obtenue par intercorrélation entre le signal de référence ref et le signal dit de surveillance sur

$$xcorr(\tau) = \int_{-\infty}^{+\infty} ref(t) \cdot sur(t+\tau) dt \tag{1}$$

et nous avions montré qu'en passant par le théorème de convolution, cette expression s'implémente efficacement dans le domaine spectral grâce à la transformée de Fourier rapide FFT (et son inverse $iFFT$) par

$$xcorr(\tau) = iFFT(FFT(ref) \cdot FFT^*(sur))$$

avec * le complexe conjugué. Cette expression nous avait permis d'implémenter la corrélation dans GNU Radio grâce aux blocs FFT tel que nous l'avons vu en Fig. 1. Cette fois, au lieu de deux récepteurs de télévision numérique terrestre, nous avons 4 voies d'oscilloscope, une de référence et trois de surveillance, qui en plus sont supposées être toutes synchrones. Ainsi, nous pourrons ajouter à la mesure de temps de vol et de vitesse de la cible une mesure de direction d'arrivée en mesurant la phase entre les corrélations issues des diverses voies.

Le seul degré de liberté dans cette équation est la durée d'intégration : l'infini étant relativement grand, il est courant de le tronquer en discrétisant 1 sous forme de

$$xcorr_n = \sum_{k=0}^{N} ref_k \cdot sur_{k+n}$$

et il faut choisir N "judicieusement". Dans l'exemple de la Fig. 1, nous avons un *a priori* sur le fait que la longueur d'un cable coaxial ne fait pas plus de (choix arbitraire) 10 m, ou un retard de 50 ns. Si nous échantillonnons à 5 GS/s (200 ps de période d'échantillonnage), alors ce retard se traduit par $50/0,2 = 250$ points. Il ne sera donc pas utile de faire des corrélations sur plus de 500 points pour rechercher des retards de l'ordre de ± 10 m. D'un autre côté, le rapport signal à bruit s'améliore comme le produit $B \cdot T$ de la bande passante du signal mesuré B par le temps d'intégration T : dans un contexte de faible rapport signal à bruit tel qu'observé dans les RADARs, il peut être judicieux d'augmenter $N = B \cdot T$ (traduisant l'augmentation de T), ici en supposant que la bande passante du signal (source de bruit large bande) est uniquement limitée par la fréquence d'échantillonnage.

Ces concepts généraux étant acquis et visant à démontrer l'utilité d'une source de signaux radiofréquence large bande discontinue, nous allons nous efforcer de les mettre en œuvre comme bloc source intégré dans GNU Radio, d'abord dans sa version 3.7 encore disponible dans la majorité des distributions Linux, puis dans sa version 3.8 fraîchement sortie au mois de Juillet 2019.

2 Bloc source pour GNU Radio 3.7

La transition de GNU Radio 3.7 à 3.8 vient d'être annoncée : nos développements initiaux se sont portés sur la première version que nous allons aborder ici, sans douter que cette version restera active dans les années à venir avant que la transition vers 3.8 ne soit effective.

2.1 Premiers pas : `gr_modtool`

Les blocs de traitement sont écrits en Python ou en C++. Par soucis d'efficacité, nous nous focaliserons sur le second langage que nous maîtrisons mieux. L'arborescence d'un squelette de bloc de traitement GNU Radio est initialisée par l'outil `gr_modtool` dans lequel nous déclarons tout d'abord le nom de l'ensemble des blocs que nous allons déclarer, et ensuite le nom de chaque bloc individuel (par exemple nous pourrions vouloir créer l'ensemble des blocs "arithmétiques" qui contiendrait les divers blocs "addition", "multiplication", "division" et "soustraction"). Dans notre cas les choses seront simples : la classe sera `oscilloscope` contenant un unique bloc de traitement nommé lui aussi `oscilloscope` :

```
$ gr_modtool newmod oscilloscope
Creating out-of-tree module in ./gr-oscilloscope... Done.
$ cd gr-oscilloscope/
$ gr_modtool add oscilloscope
```

avec comme type de bloc `source`, comme langage `cpp`, pas d'argument et pas de code de gestion de qualité. Nous voila donc équipé de l'arborescence pour travailler, avec dans l'ordre d'importance :

1. `lib` le répertoire contenant le code source en C++ du bloc ainsi que le fichier d'entête associé déclarant les variables privées et publiques de la classe,
2. `include` contenant le prototype du constructeur de la classe,
3. `grc` contenant la description, au format XML, des entrées et sorties ainsi que les paramètres modifiables par l'utilisateur du bloc pour informer l'interface graphique `gnuradio-companion` de ces interfaces.

Nous ne toucherons pas pour le moment aux autres répertoires (`cmake` pour les scripts de compilation, `python` pour le code d'instanciation du bloc dans une chaîne de traitement, `swig` pour exporter les symboles du C++ vers Python [8], ou `examples` et `doc` dont l'utilité est évidente) de l'arborescence.

En commençant par regarder `lib/oscilloscope_impl.cc`, nous voyons un certain nombre de motifs de la forme `<+MIN_OUT+>`, `<+MAX_OUT+>` ou `<+OTYPE+>` qu'il faudra remplacer par le nombre minimum et maximum d'interfaces – dans notre cas entre 1 et 4 dans l'hypothèse qu'au moins une voie de l'oscilloscope est utilisable et au maximum 4 sont accessibles – et la nature des interfaces – GNU Radio manipule classiquement des nombres à virgule flottance compris entre -1 et +1 donc nous remplaçons `<+OTYPE+>` par `float`. Comme c'est un bloc source, il ne prend aucune entrée et `gr_modtool` a déjà déclaré `gr::io_signature::make(0, 0, 0)` indiquant l'absence de données lues.

Il est judicieux à ce point de valider son installation de GNU Radio en compilant ce bloc de traitement totalement inutile puisque nous n'avons que déclaré les interfaces, mais néanmoins fonctionnel. Comme dans tout projet faisant appel à `cmake`, nous créons `build`, y entrons et lançons `cmake ../` qui génère un `Makefile` permettant la compilation par `make`. Si l'environnement de développement est fonctionnel, la compilation s'achève en quelques secondes par `[100%] Built target doxygen_target` : nous voilà avec une arborescence exploitable. **Attention :** le bloc que nous venons de compiler **n'est pas visible de GNU Radio ni de** `gnuradio-companion`. Pour ce faire, il nous faudra `make install` afin de copier les fichiers utiles dans l'arborescence de GNU Radio.

Noter que pour le moment nous n'avons pas renseigné les interfaces dans le fichier XML disponible dans le répertoire `grc` pour décrire les interfaces et les paramètres du bloc, opération que nous complèterons ultérieurement. On pourra toutefois déjà se convaincre de la facilité de générer ce fichier XML en se plaçant dans le répertoire `gr-oscilloscope` et en lançant `gr_modtool makexml` : après avoir autorisé à écraser le contenu de l'exemple initial (qu'il peut être bon de conserver dans un coin pour référence), nous constatons que `grc/oscilloscope_oscilloscope.xml` contient une description fidèle de notre bloc :

```
<category>[OSCILLOSCOPE]</category>
<import>import oscilloscope</import>
<make>oscilloscope.oscilloscope()</make>
<source>
  <name>out</name>
  <type>float</type>
  <nports>4</nports>
</source>
```

donc notre bloc se trouvera dans le menu `Oscilloscope` de `gnuradio-companion`, se charge dans Python 2.7 par `import oscilloscope`, ne contient que des sorties au nombre de 4 communiquant des nombres à virgule flottante. Nous verrons plus tard comment rendre le nombre d'interfaces paramétrable. Le squelette de fichier XML de configuration issu de l'analyse des fichiers d'entête de l'implémentation du bloc devra de toute façon être corrigé pour répondre à toutes nos attentes (nombre variable de voies de mesure, fonctions appelées pour modifier dynamiquement les paramètres d'acquisition).

Notre travail se limite donc à remplir le constructeur (`oscilloscope_impl::oscilloscope_impl()`), éventuellement le destructeur (`oscilloscope_impl::~oscilloscope_impl()`) mais surtout la fonction principale qu'est la méthode `oscilloscope_impl::work()`. Tous ces prototypes de fonctions sont décrits dans le fichier d'entête dans le répertoire `lib` et l'implémentation de ces fonctions se trouve dans le fichier d'extension `.cc` dans ce même répertoire.

2.2 Paramétrage du bloc

Un oscilloscope est paramétré par trois valeurs qui vont nous intéresser ici : l'amplitude des signaux (*range*), la fréquence d'échantillonnage (*sampling rate*) et la durée de l'acquisition (*duration*). Le produit de la fréquence d'échantillonnage par la durée d'acquisition détermine le nombre de points acquis, limité par la profondeur mémoire de l'instrument. Plus la durée est longue plus la cible peut être loin (équivalent du taux de répétition de l'impulsion – PRR pour *Pulse Repetition Rate* – dans un RADAR impulsionnel) mais plus le nombre de points à traiter sera important à bande passante (double de la fréquence d'échantillonnage) donnée. Par ailleurs, nous voulons pouvoir définir l'adresse IP de l'oscilloscope avec lequel nous communiquons, et le nombre de voies acquises, et ce évidemment sans avoir à recompiler le bloc de traitement. Nous avons ainsi établi la liste des paramètre communiqués au constructeur :

```
oscilloscope_impl::oscilloscope_impl(char *ip,float range,float rate,float duration,int channels)
      : gr::sync_block("oscilloscope",
              gr::io_signature::make(0, 0, 0),
              gr::io_signature::make(1, 4, sizeof(float))), // min max channels
              _range(range),_rate(rate),_duration(duration),_channels(channels)
```

qui est donc une source n'acceptant aucune entrée (signature de l'entrée avec 0 voies) et propose entre 1 et 4 sorties (signature de la sortie (1,4)) de type nombre à virgule flottante. Les cinq paramètres sont la chaîne de caractère représentant l'adresse IP, les trois paramètres de configuration de l'oscilloscope, et le nombre de voies mesurées. Le prototype de la fonction dans le fichier d'entête `include/oscilloscope/oscilloscope.h` est mis à jour en accord avec cette définition : `static sptr make(char*,float,float,float,int);`. On notera que cette syntaxe, valable en C, ne plaît pas à `gr_modtool` qui désire avoir le nom des variables associées à chaque paramètre, ou tout au moins un espace après chaque type de variable sans laquelle l'analyseur syntaxique échoue.

Le lien entre fichier de configuration XML et C++ est décrit dans le bout de code ci-dessous :

```
<make>oscilloscope.oscilloscope($ip,$range,$rate,$duration)</make>
<param>
  <name>Rate</name>
  <key>rate</key>
  <type>int</type>
</param>
```

GNU Radio companion sait ainsi qu'un paramètre `rate` est ajustable par l'utilisateur dans le bloc source, qu'il s'agit d'un entier, et que ce paramètre est fourni au constructeur comme troisième argument.

De la même façon, la fonction principale de travail est adaptée à cette configuration avec

```
    oscilloscope_impl::work(int noutput_items,
      gr_vector_const_void_star &input_items,
      gr_vector_void_star &output_items)
    {float *out0 = (float *) output_items[0]; // 1 a 4 sorties
     float *out1 = (float *) output_items[1];
     float *out2 = (float *) output_items[2];
     float *out3 = (float *) output_items[3];
...
```

dans lequel `noutput_items` informera du nombre de données que nous renvoyons – en accord avec la demande de l'ordonnanceur qui indique dans cette variable le nombre de données qu'il voudrait recevoir – dans le tableau `output_items`. Ce tableau de tableau pointe sur quatre sous-tableaux qui contiendront chacun les données des voies de mesure de l'oscilloscope : nous avons nommé ces tableaux `out`i, $i \in [0..3]$, et ces pointeurs visent les entrées de `output_items`.

Nous complétons `lib/oscilloscope_impl.h` avec les variables privées et publiques de la classe, en proposant dès à présent un découpage du programme dans lequel les fonctions définissant les paramètres ajustables par l'utilisateur sont séparées, découpage dont nous verrons l'intérêt plus bas. Ainsi lorsque nous voulons modifier par exemple la fréquence d'échantillonnage, nous appellerons la fonction publique `void set_rate(float)` ; qui stocke dans la variable privée `float _rate` la valeur fournie par l'utilisateur en vue de la distribuer aux autres fonctions de la classe.

2.3 Remplissons le bloc de traitement : communication par VXI11

La majorité des instruments récents communique sur bus Ethernet au travers du protocole VXI11 (ou LXI, `http://www.lxistandard.org/`). Ainsi, notre constructeur sera principalement chargé d'initialiser la `socket` pour ouvrir le port de communication et de configurer l'instrument. Pour ce faire, nous nous inspirons de n'importe quel exemple de `https://github.com/applied-optics/agilent_vxi11/blob/master/library/agilent_vxi11.c` par exemple, avec l'ouverture de la communication par `vxi11_open_device` qui prend en argument la chaîne de caractères de l'adresse IP et renvoit la structure `VXI11_CLINK *` qui sera utilisée au cours de toutes les transactions. Nous avons encapsulé les fonctions de lecture et d'écriture vxi11_receive() et `vxi11_send()` de façon à les rendre facilement adaptable à tout autre protocole de communication. Adapter le programme à un nouveau type d'instrument se résume donc à appréhender la liste des instructions GPIB

fournies dans le manuel de l'utilisateur (par exemple `https://scdn.rohde-schwarz.com/ur/pws/dl_downloads/pdm/cl_manuals/user_manual/1326_1032_01/RTE_UserManual_en_15.pdf` avec sa section 17 intitulée "Remote Control Commands"). Nous faisons bien entendu l'hypothèse que l'utilisateur a convenablement configuré les interfaces réseau de son ordinateur, en particulier en définissant l'adresse IP de l'interface Ethernet sur le même sous-réseau que l'oscilloscope.

Serveur TCP simulant une source de données

Tous les lecteurs n'ont pas forcément un oscilloscope radiofréquence à leur disposition. Par ailleurs, il peut être intéressant d'exploiter `gr-oscilloscope` avec des sources de données autres qu'un oscilloscope – par exemple transmis sur Internet par TCP/IP – voir pour synthétiser des signaux simulant divers types de bruits ou de sources difficiles à observer expérimentalement. Nous nous proposons donc de fournir un serveur TCP/IP générant un flux de données alimentant `gr-oscilloscope`. Ce dernier appliquera le protocole implémenté par ce serveur si l'adresse IP est 127.0.0.1 (localhost), et le protocole VXI11 d'un oscilloscope sinon.

Notre serveur TCP/IP reçoit, à la connexion du client, le nombre de voies attendues, et se contente ensuite d'envoyer le nombre de données attendues par le client.

```
#define MY_PORT         9999

int main()
{int sockfd;
 struct sockaddr_in self;
 float *buffer;
 struct sockaddr_in client_addr;
 int clientfd;
 socklen_t addrlen=sizeof(client_addr);
 int taille,k,c;
 long channels;

 sockfd = socket(AF_INET, SOCK_STREAM, 0);  // socket type (TCP blocking)
 bzero(&self, sizeof(self));
 self.sin_family = AF_INET;
 self.sin_port = htons(MY_PORT);
 self.sin_addr.s_addr = INADDR_ANY;
 bind(sockfd, (struct sockaddr*)&self, sizeof(self));
 listen(sockfd, 20);
 while (1) {
  clientfd = accept(sockfd, (struct sockaddr*)&client_addr, &addrlen);
  recv(clientfd, &channels, sizeof(long), 0);channels=ntohl(channels);
  while (taille!=-1)
    {recv(clientfd, &taille, sizeof(long), 0);
     taille=ntohl(taille); printf("request: %d values\n",taille);
     if (taille>0)
        {buffer=(float*)malloc(sizeof(float)*taille);
         for (c=0;c<channels;c++)
            {for (k=0;k<taille;k++)
                 buffer[k]=sin(2*M_PI*(float)k/(float)taille*5*(float)(c+1));
             send(clientfd, buffer, taille*sizeof(float), 0);
            }
         free(buffer);
        }
    }
  close(clientfd);
  taille=0;    // when disconnect and reconnect
 }
 close(sockfd);return(0);  // Clean up (should never get here)
}
```

Il s'agit de l'archétype du serveur TCP/IP sans aucune subtilité : une *socket* est créée en mode connecté (TCP, `SOCK_STREAM`) compatible Internet (IP, `AF_INET`), associée (`bind()`) au port 9999, et attend (`listen()`) une connexion d'un client. Lors de la connexion, nous récupérons le descripteur de fichier associé (`accept()`) et échangeons (`recv()` et `send()`) jusqu'à interruption des échanges lorsque le client requiert un nombre négatif d'échantillons. Une fois la connexion rompue, le serveur attend une nouvelle connexion.

Dans cet exemple nous remplissons séquentiellement les tampons contenant les données de chaque voies de sinusoïdes de fréquence croissante, mais il serait très simple de fournir des séquences aléatoires décalées dans le temps pour simuler le retard introduit par le temps de vol du signal entre l'émetteur, la cible et le récepteur.

Dans le cas du serveur TCP/IP qui émule l'oscilloscope, nous ferons explicitement appel à la séquence de connexion du client par l'habituel

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);          // IPv4, TCP
adresse.sin_family=AF_INET;
adresse.sin_addr.s_addr = inet_addr("127.0.0.1"); // serveur sur localhost
adresse.sin_port =htons(9999);                     // port du serveur
bzero(&(adresse.sin_zero),8);
longueur = sizeof(adresse);
connect(sockfd, (struct sockaddr *)&adresse, longueur);
```

en prenant soin de bien passer tous les paramètres avec l'*endianness* du réseau

```
longueur=htonl(_channels);
write(sockfd,&longueur,sizeof(long)); // number of channels
```

Nous pouvons immédiatement remplir le destructeur qui libère les ressources : de nouveau dans le cas du serveur TCP/IP, cela se résume par

```
    oscilloscope_impl::~oscilloscope_impl() // Our virtual destructor.
    {int val;
     val=htonl(-1);
     write(sockfd,&val,sizeof(int));
     close(sockfd);
    }
```

qui informe le serveur de la fin des transactions (requête d'un nombre négatif de données) et ferme la socket.

L'initialisation étant achevée, la méthode `work` est celle sollicitée systématiquement par l'ordonnanceur GNU Radio et donc la plus importante. Notre choix a été de collecter une trace sur l'oscilloscope répondant aux attentes de l'utilisateur en terme de durée de la mesure, et de distiller ces points à l'ordonnanceur selon ses besoins. En effet, l'ordonnanceur GNU Radio requiert un nombre variable de points selon le débit des traitements et la fréquence d'échantillonnage, et nous ne saurions prédire cette valeur ou voir si elle est compatible avec les paramètre d'acquisition d'un oscilloscope. Par ailleurs, nous voulons limiter le temps perdu en transactions sur le bus Ethernet et l'acquisition de gros paquets de données limitera les latences induites par les communications.

Le code ci-dessous contient la fonction de collecte des informations : si une variable `_num_values` indexant le nombre de données restant en réserve dans le tampon a atteint 0, il faut acquérir un nouveau jeu de points en transmettant la commande GPIB adéquate (`RUNSINGLE` chez Rohde & Schwarz), attendre la fin de l'acquisition (`OPC?`) et lire les mesures (`CHANx:WAV:DATA?` pour la voie `x`) pour les stocker dans des tableaux temporaires `_tab`x, $x \in [0..3]$.

```
int oscilloscope_impl::work(int noutput_items,
    gr_vector_const_void_star &input_items,
    gr_vector_void_star &output_items)
   {float *out0 = (float *) output_items[0]; // tableaux de donnees
    float *out1 = (float *) output_items[1];
    float *out2 = (float *) output_items[2];
    float *out3 = (float *) output_items[3];
    long int k,val,offset;
    int chan_count;
    char mystring[256];
    char buffer[256];
    if (_num_values==0)             // plus de points en reserve : recharger
       {_num_values=_sample_size;   // nombre de points a collecter
        _position=0;
        sprintf(buffer,"FORM:DATA␣INT,16\n");envoi(dev,buffer); // LSB first by default
        sprintf(buffer,"RUNSINGLE\n");envoi(dev,buffer);
        sprintf(buffer,"*OPC?"); envoi(dev,buffer); relit(dev,buffer,256);

        for (chan_count=1;chan_count<=_channels;chan_count++)
          {sprintf(mystring,"CHAN%d:WAV:DATA?",chan_count);
           vxi11_send_and_receive(dev, mystring, _data_buffer, (2*_sample_size+100), 100*VXI11_READ_TIMEOUT);
           offset=_data_buffer[1]-'0';     // ASCII -> dec : indexe de debut
           for (k=0;k<_sample_size;k++)    // rm # and header
              {if (chan_count==1) _tab1[k]=(float)(*(short*)(&_data_buffer[2*k+offset+2]))/65536.; // suppose littl
[... traite 2, 3 ou 4 voies en remplissant _tabi[k] ...]
              }
          } // end of chan_count
    } // end of _num_values==0
```

Dans le cas contraire où assez de points sont encore disponibles en mémoire pour répondre aux demandes de l'ordonnanceur GNU Radio, nous transférons des tampons vers les tableaux renvoyant les mesures pour poursuivre la chaîne de traitements :

```
    for (k=0;((k<noutput_items) && (_position<_sample_size));k++) // tant qu'il reste des points ...
      {out0[k]=_tab1[_position];
[... traite 2, 3 ou 4 voies en remplissant outi ...]
       _num_values--;
       _position++;
      }
    if (_num_values==0) return(k); // on ne renvoit que ce qui restait
    return noutput_items;          // sinon on annonce avoir satisfait l'ordonnanceur
   }
```

Cette séquence de mesures est encore plus simple avec le serveur TCP/IP qui renvoit les informations à la demande sans contrainte physique de configuration de l'instrument. Encore une fois nous prendrons simplement soin de convertir tous les échanges en *endianness* du réseau IP (*host to network*) par `ntohl()`. Un exemple d'une telle transaction est illustré en Fig. 2.

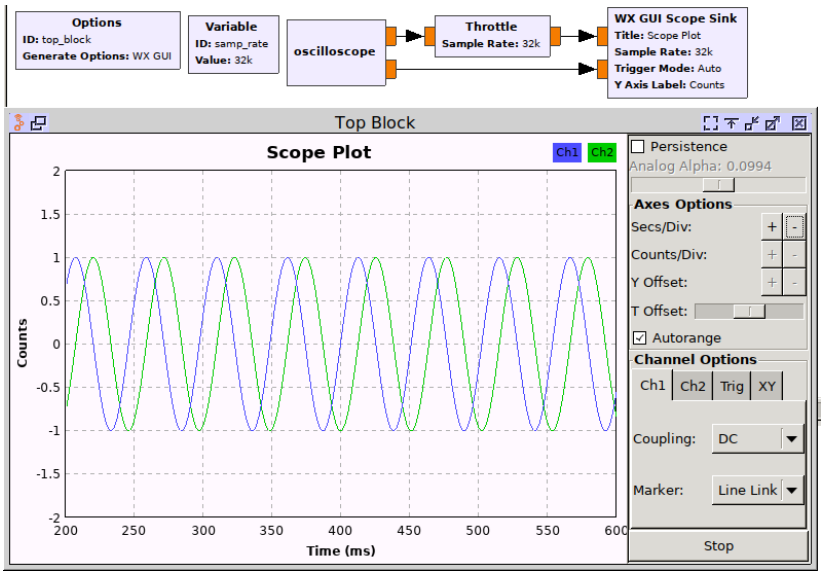


FIGURE 2 – Le serveur TCP/IP émule un oscilloscope pour les transactions avec le client de `gr-oscilloscope`. Cet exemple de GNU Radio 3.7 utilise volontairement l'interface WX pour différencier de l'implémentation GNU Radio 3.8 que nous verrons plus loin : cette interface graphique est à proscrire puisque ne sera pas maintenue dans les évolutions de GNU Radio faute de portabilité et de support.

Les grandes lignes du code étant appréhendées, il nous reste à compiler le programme. Alors que RPC est au cœur de nombreux protocoles (incluant NFS), Sun RPC n'est plus installé par défaut depuis la version 2.26 de glibc [10]. Pour continuer à accéder à ces fonctionnalités, la bibliothéque doit être compilée avec l'option `-enable-obsolete-rpc`. Sur certaines distributions, telle que Debian, cette option semble active puisque `/usr/include/rpc/rpc.h` est toujours présent. Dans d'autres, comme Gentoo, `rpc.h` n'est disponible qu'après avoir installé `libtirpc`, avec comme effet de bord que `rpc.h` ne se trouve plus dans `/usr/include/rpc/` mais dans `/usr/include/tirpc/rpc/`. Dans ce cas, il faut aider `cmake` à trouver les bons chemins pour accéder à l'entête et à la bibliothèque. Nous avons donc ajouté dans `cmake/Modules` un fichier `Findlibtirpc.cmake` (le nom est imposé par `cmake`) :

```
find_package(PkgConfig)
pkg_check_modules(PC_LIBTIRPC QUIET libtirpc)
set(TIRPC_DEFINITIONS ${PC_LIBTIRPC_CFLAGS_OTHER})
find_path(TIRPC_INCLUDE_DIR rpc/rpc.h
    HINTS ${PC_LIBTIRPC_INCLUDEDIR} ${PC_LIBTIRPC_INCLUDE_DIRS}
    PATH_SUFFIXES libtirpc )
find_library(TIRPC_LIBRARY NAMES tirpc libtirpc
    HINTS ${PC_LIBTIRPC_LIBDIR} ${PC_LIBTIRPC_LIBRARY_DIRS} )
include(FindPackageHandleStandardArgs)
find_package_handle_standard_args(libtirpc DEFAULT_MSG
    TIRPC_LIBRARY TIRPC_INCLUDE_DIR)
mark_as_advanced(TIRPC_INCLUDE_DIR TIRPC_LIBRARY )
set(TIRPC_LIBRARIES ${TIRPC_LIBRARY} )
set(TIRPC_INCLUDE_DIRS ${TIRPC_INCLUDE_DIR} )
add_library(TIRPC::TIRPC INTERFACE IMPORTED)
set_target_properties(TIRPC::TIRPC PROPERTIES
    INTERFACE_INCLUDE_DIRECTORIES "${TIRPC_INCLUDE_DIRS}"
```

```
    INTERFACE_LINK_LIBRARIES "${TIRPC_LIBRARY}"
)
```

Ce fichier a pour rôle de chercher les répertoires où se trouvent `rpc.h` et `libtirpc.so` pour ensuite ajouter ceux-ci dans les variables utilisées par `cmake` et Makefile lors de la compilation.

Pour utiliser ce fichier, et ajouter les chemins lors de la compilation, le fichier `lib/CMakeLists.txt` doit être modifié afin d'ajouter

```
find_package(libtirpc REQUIRED)
```

et changer la ligne :

```
target_link_libraries(gnuradio-oscilloscope gnuradio::gnuradio-runtime)
```

par

```
target_link_libraries(gnuradio-oscilloscope PUBLIC
    gnuradio::gnuradio-runtime
    TIRPC::TIRPC)
```

RPC étant accessible, il reste à ajouter les bibliothèques de communication VXI11 que nous avons clonées depuis `https://github.com/applied-optics/vxi11` pour les placer dans le répertoire `lib` de notre projet – les sources nécessaires sont donc dans `lib/vxi11/library` – en complétant `CMakeLists.txt` du répetoire `lib/` par

```
list(APPEND oscilloscope_sources
    oscilloscope_impl.cc
    vxi11/library/vxi11_clnt.c
    vxi11/library/vxi11_user.c
    vxi11/library/vxi11_xdr.c
   )
```

(noter que les deux premières lignes existaient déjà et nous n'avons fait que rajouter les sources implémentant VXI11).

Le programme résultant permet de collecter continûment des mesures de plusieurs voies d'un oscilloscope pour les traiter en temps réel par GNU Radio (Fig. 3), en particulier pour des traitements un peu plus subtiles que le simple affichage, permettant de mettre en évidence des défauts de l'instrument.

En effet, la corrélation de deux voies alimentées par le *même signal périodique* présente des sauts de phase qui ne peuvent être attribuées qu'à une désynchronisation d'un échantillon des flux de données transmis par l'instrument (Fig. 4), un effet catastrophique pour un traitement cohérent des mesures.

3 Fonctions de *callback*

Le programme de base est fonctionnel, nous sommes capables de collecter des mesures d'un oscilloscope sur 1 à 4 voies et alimenter ainsi une chaîne de traitement GNU Radio définie graphiquement dans GNU Radio Companion. Cependant, en l'état toute modification des paramètres d'acquisition nécessite d'interrompre l'exécution de la chaîne de traitement, modifier le paramètre dans le bloc source, et relancer la chaîne, une situation peu confortable qui ne répond pas nécessairement aux attentes de l'utilisateur qui désire interagir dynamiquement avec son instrument.

GNU Radio propose un mécanisme dans lequel les modifications de paramétrage du bloc source sont détectées et transmises à des fonctions spécifiquement associées à chaque paramètre : il s'agit des fonctions de *callback* que nous avions pris soin de séparer de la fonction principale `work` lors de la conception du programme d'acquisition des données (e.g. la fonction `set_rate()` que nous avions mentionnée). Ce lien est créé dans le fichier XML de configuration de GNU Radio Companion en cohérence avec les fonctions implémentées en C++. Ainsi, quatre fichiers sont mis à jour pour implémenter les fonctions de *callback* :

1. `gr-oscilloscope/grc/oscilloscope_oscilloscope.xml` définit la nature (entier, nombre à virgule flottante, complexe, chaîne de caractères ...) et le nom des paramètres
2. `gr-oscilloscope/include/oscilloscope/oscilloscope.h` définit dans le prototype du constructeur les paramètres fournis au C++ au lancement du bloc,
3. `gr-oscilloscope/lib/oscilloscope_impl.h` définit les variables privées qui contiennent les valeurs des paramètres qui seront mises à jour par les fonctions de *callback*,
4. `gr-oscilloscope/lib/oscilloscope_impl.cc` définit l'implémentation en C++ des fonctions de *callback* et en particulier les actions à prendre sur la configuration de l'oscilloscope par commande VXI11 en réponse aux changements de paramètres requis par l'utilisateur.

Nous avons mis en œuvre les fonctions de *callback* pour changer les paramètres de mesure tels que la fréquence d'échantillonnage, la durée de la mesure ou la gamme de tensions acquises. À titre d'illustration, le changement de fréquence d'échantillonnage se traduit par

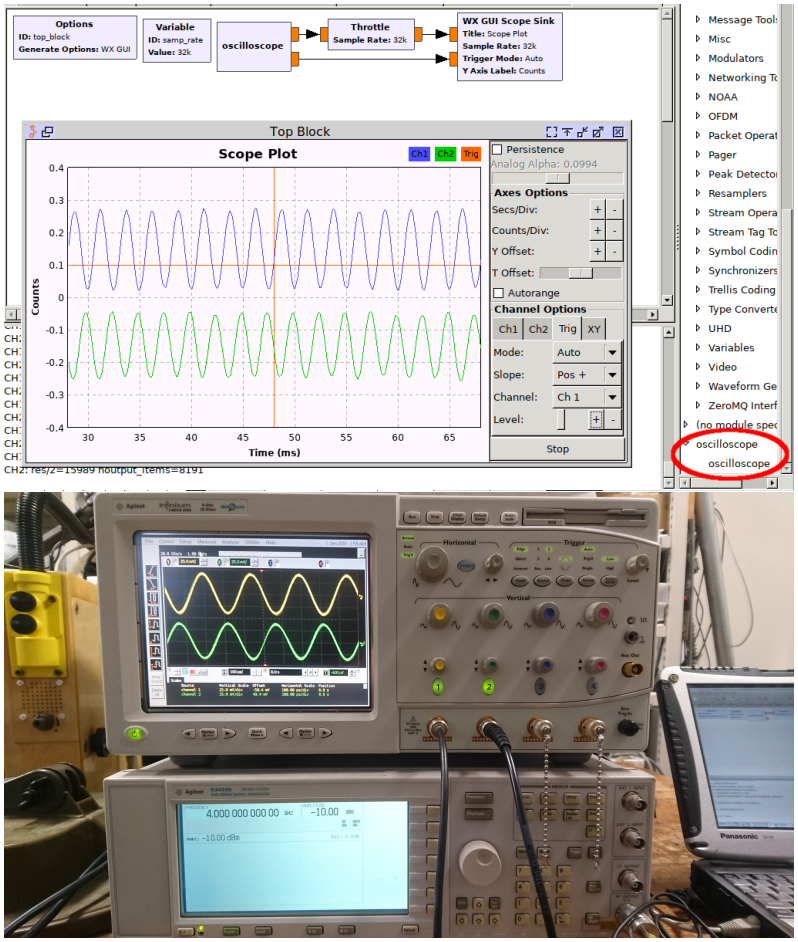


FIGURE 3 – Haut : acquisition du flux de données issu d'un oscilloscope radiofréquence par notre bloc de traitement `gr-oscilloscope`. Bas : montage expérimental.

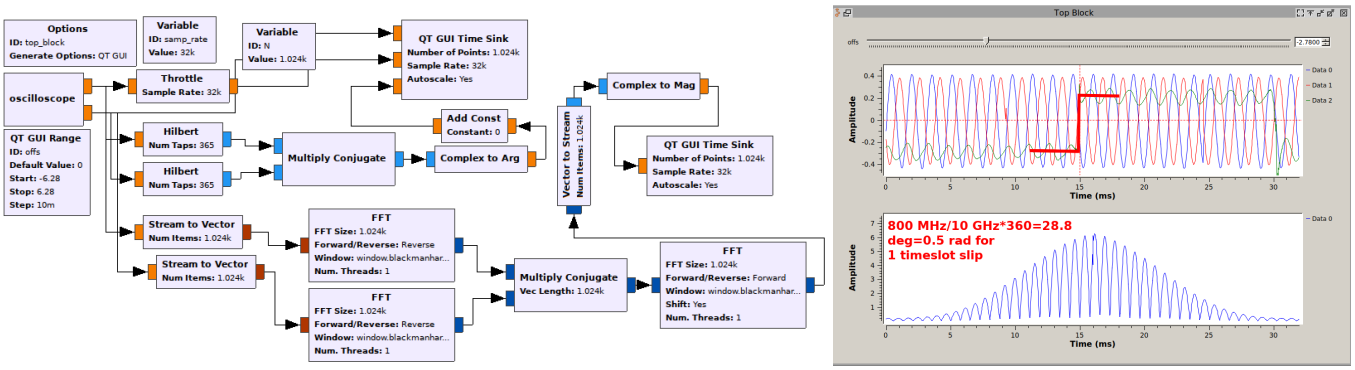


FIGURE 4 – Deux voies d'un oscilloscope Agilent 54855A sont alimentées par un même signal radiofréquence à 800 MHz et la phase de la corrélation est analysée. Un saut aléatoire d'un échantillon entre les deux voies est parfois observé, effet fort ennuyeux pour une accumulation cohérente de mesures.

```
1  void oscilloscope_impl::set_rate(float rate)
2  {char buffer[256];
3   _sample_size = (int)(_duration*rate);
4   sprintf(buffer,"ACQ:SRATE␣%f\n",rate);
5   envoi(dev,buffer);
6   _data_buffer=(char*)realloc(_data_buffer,2*_sample_size+100);
7   _tab1=(float*)realloc(_tab1,_sample_size*sizeof(float)); // realloc=malloc si _tab1==NULL
8  [... idem pour les autres voies ...]
9   _rate=rate;
10 }
```

Ainsi, changer la fréquence d'échantillonnage, à durée de mesure donnée, modifie le nombre de points acquis et nécessite donc de re-allouer la mémoire occupée par les divers tableaux contenant les points de mesures. Cette fonction reste valable lors de l'initialisation puisque `realloc()` se comporte comme `malloc` si le pointeur fourni en argument est NULL. La commande GPIB configurant la fréquence d'échantillonnage de l'oscilloscope est transmise par VXI11, et finalement la nouvelle valeur de la fréquence d'échantillonnage est stockée dans la variable privée `_rate` de la classe pour être disponible par les autres méthodes.

Il nous reste à faire le lien entre Python et C++ : les fonctions de *callback* sont déclarées comme publiques dans `oscilloscope_impl.h` du répertoire `lib`

```
1     class oscilloscope_impl : public oscilloscope
2     {private:
3        int _rate;
4      public:
5       void set_rate(int);
6       oscilloscope_impl(char*,float,int,float);
```

et le fichier de configuration XML de GNU Radio Companion est renseigné du nom de la fonction de *callback* associée au paramètre modifiable (la déclaration du paramètre restant la même que vue précédemment) :

```
1 <make>oscilloscope.oscilloscope($ip,$range,$rate,$duration)</make>
2 <callback>set_rate($rate)</callback>
```

À l'issue de ces modifications, les paramètres sont désormais modifiable dans la chaîne de traitement en cours d'exécution, par exemple au moyen d'un menu déroulant (`QT GUI Chooser` pour limiter les choix à quelques valeurs prédéfinies) ou d'un curseur glissant (`QT GUI Range`).

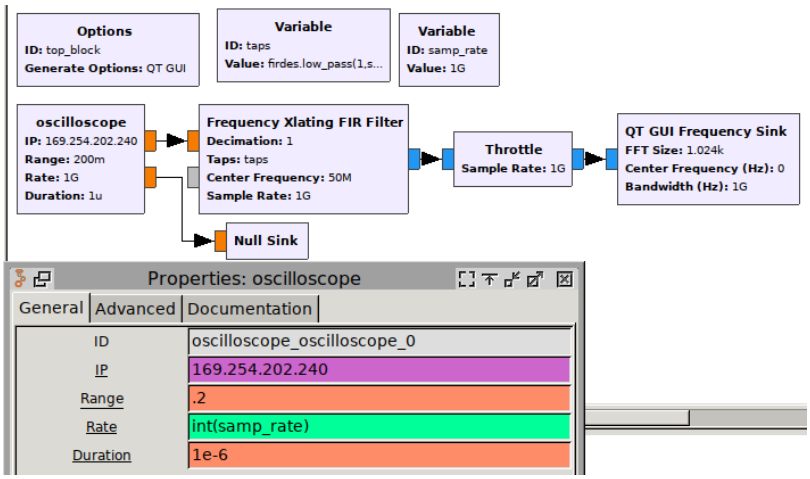


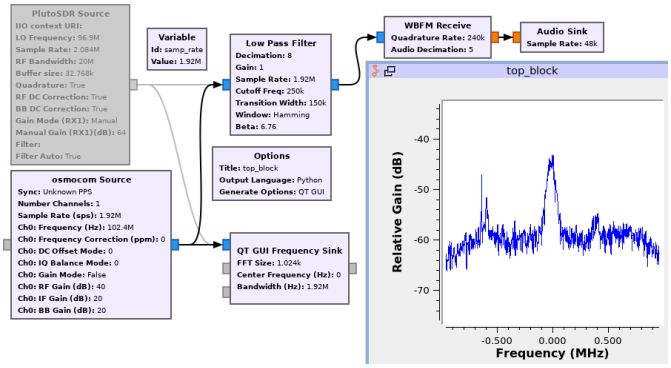
FIGURE 5 – Le bloc `gr-oscilloscope` muni de ses fonctions de *callback* : les paramètres dynamiquement ajustables sont soulignés.

Nous voici donc munis d'un bloc fonctionnel pour traiter un flux de données issu d'une ou plusieurs voies d'oscilloscope pour GNU Radio 3.7.

4 Passage à GNU Radio 3.8

GNU Radio 3.8 (`https://github.com/gnuradio/gnuradio/releases/tag/v3.8.0.0`) est en pleine effervescence. Compte tenu du nombre de modifications encore apportées pour stabiliser cette version, il semble judicieux de compiler depuis les sources au lieu de s'appuyer sur une distribution binaire qui aura toutes les chances d'être obsolète face à la version de développement. Néanmoins, nombre de blocs source (osmosdr, PlutoSDR) ou de traitement n'ont pas encore été portés à 3.8, donc conserver une version de 3.7 fonctionnelle pour ses activités quotidiennes reste utile.

PyBOMBS fournissant les recettes pour GNU Radio 3.7, il faut mettre à jour les recettes `.lwr` (voir ci-dessous) pour installer `gr-iio` qui fournit les blocs de communication avec la PlutoSDR et `gr-osmosdr` pour les récepteurs DVB-T. `pybombs install gr-iio` indique que `libad9361` et `libiio` vont être installés mais dépend de `gnuradio`. Nous devons donc modifier la recette `gr-iio.lwr` en modifiant `gnuradio` par `gnuradio38` et modifier `gitrev: tags/v0.3` par `gitbranch: upgrade-3.8`. De la même façon, `gr-iqbal.lwr` est modifié pour remplacer la dépendance `gnuradio` par `gnuradio38`. Pour OsmoSDR, nous modifions `gr-osmosdr.lwr` pour remplacer `gnuradio` par `gnuradio38` et `gitbranch: master` par `gitbranch: gr3.8`. Nous avons aussi du fournir comme `source: https://github.com/mvaenskae/gr-osmosdr.git`. La figure ci-contre montre le bon fonctionnement de ces blocs.



Réception de Radio Campus à Besançon par PlutoSDR sous GNU Radio 3.8.

La compilation de GNU Radio par PyBOMBS (`https://github.com/gnuradio/pybombs`) permet d'isoler tout l'environnement de développement dans un répertoire donné sans risquer de polluer son installation "stable" : PyBOMBS est un utilitaire écrit en Python qui fait usage de fichiers `.lwr` pour fournir les régles pour compiler l'ensemble d'un environnement (liste des paquets) ou pour compiler un paquet spécifique (dépendances dudit paquet, url du dépôt à télecharger, options de compilation, quel est le paquet équivalent dans la distribution hôte, ...).

La première étape est d'installer PyBOMBS lui même. Nos tests sur une installation à l'aide de `pip3` (version dédiée à Python3) ayant montré des soucis il semble, en l'état, préférable de se baser sur le dépôt.

```
1 git clone https://github.com/gnuradio/pybombs
2 cd pybombs
3 python3 setup.py install --user
```

(noter le `-user` pour installer PyBOMBS dans `$HOME/.local` et non dans `/usr/local`. Il est nécessaire de s'assurer que `$HOME/.local/bin` est dans le `$PATH`)

Ensuite nous devons configurer PyBOMBS par

```
1 pybombs auto-config
```

pour générer le fichier $HOME/.pybombs/config.yml adapté à la distribution (principalement le gestionnaire de la distribution).

Finalement la commande :

```
1 pybombs recipes add-defaults
```

va télécharger dans `$HOME/.pybombs/recipes` les deux dépôts de recettes `gr-etcetera` et `gr-recipes`, fournissant les règles `*.lwr`.

Avant de continuer, et de lancer la compilation de gnuradio 3.8, plusieurs étapes doivent être réalisées (qui ne seront, à terme, plus nécessaires dès lors que `https://github.com/gnuradio/gr-recipes/pull/155` aura été appliqué). La première est d'installer manuellement les paquets `click-plugins` et `pyqtgraph` avec le gestionnaire de paquets de sa distribution. En effet GNU Radio dépend de ceux-ci mais ces derniers ne sont pas listés comme dépendances.

La recette `gnuradio-default.lwr` pointe sur `master`. Selon [11], depuis la sortie de la première version stable de gnuradio 3.8, cette branche est dédiée à la future 3.9 et une branche spécifique (`maint-3.8`) a été créée pour suivre les évolutions de 3.8 (exclusivement sur un aspect corrections et optimisations). Il est donc nécessaire de modifier, dans `$HOME/.pybombs/recipes/gr-recipes/gnuradio38.lwr` la ligne

```
1 gitbranch: master
```

par, si l'on souhaite être conservatif et s'assurer d'une reproductibilité, en se basant sur le dernier `tag` stable (3.8.0.0 à la date de rédaction) :

```
1 gitrev: v3.8.0.0
```

ou si l'on souhaite profiter des dernières corrections (et des dernières régressions) par :

```
1 gitbranch: maint-3.8
```

Finalement, nous avons précisé que les fichiers `.lwr` fournissent les régles pour la construction d'un paquet ou d'un environnement. Il existe dans `gr-recipes` une régle pour construire gnuradio-3.8, mais aucune pour la génération de l'environnement comportant ce paquet. Deux solutions sont possibles pour contourner ceci. La première consiste à créer son propre fichier de recette. Pour cela il faut se placer dans `$HOME/.pybombs/recipes/gr-recipes`, copier `gnuradio-default.lwr` en `gnuradio-3.8.lwr` et éditer ce fichier pour remplacer (ligne 22) `gnuradio` par `gnuradio38`.

Enfin, lancer la compilation par :

```
1 pybombs prefix init ~/gnuradio38 -a gr38 -R gnuradio-38
```

en supposant que nous voulions installer GNU Radio 3.8 dans `gnuradio38` de sa maison. Noter que les utilisateurs de la distribution Fedora (s'il y en a) risquent de rencontrer une difficulté lors de cette compilation tel que décrit à `https://github.com/gnuradio/gnuradio/issues/1444`, nécessitant d'ajouter l'option `-DENABLE_CTRLPORT_THRIFT=OFF` dans la recette `gnuradio38.lwr`. Les utilisateurs de l'interface graphique IceWM pourront lier `pkexec` à `sudo` pour installer les paquets des dépendances en super-utilisateur, l'agent d'authentification ne semblant pas disponible pour ce gestionnaire de fenêtres.

La seconde solution consiste à créer un prefix (environnement), puis de lancer la compilation/installation de gnuradio explicitement par :

```
1 pybombs prefix init ~/gnuradio38 -a gr38
2 pybombs -p gr38 install gnuradio38
```

et d'attendre un (très) long moment, avant d'avoir dans `$HOME/` un répertoire `gnuradio38` construit en suivant la régle gnuradio-38 que nous avons créée.

Une ultime modification doit être faite au niveau du `$HOME/gnuradio38/setup_env.sh` : même si PyBOMBS et GNU Radio ont bien détecté l'utilisation de Python3 lors de la compilation, la variable d'environnement `PYTHONPATH` pointe sur python2.7 et non python3.x : modifier toutes les occurrences du premier par sa version de Python3. À l'issue de la compilation d'une version donnée de GNU Radio, et 10 GB pour 3.8.0 de moins sur son disque dur, cette version sera utilisée en chargeant les variables d'environnement associées fournies par PyBOMBS au moyen de `source setup_env.sh`.

ATTENTION : GNU Radio Companion charge toujours la même configuration, à savoir les chaînes de traitement précédemment en cours d'édition, qu'il s'agisse de 3.7 ou de 3.8. Or la description des chaînes de traitement n'est pas compatible entre 3.7 et 3.8 : toute chaîne (fichier `grc`) ouverte en 3.8 devient illisible en 3.7. On pensera donc à bien fermer tous les onglets avant de quitter GNU Radio Companion tant que les deux versions de GNU Radio Companion doivent cohabiter.

L'environnement de travail de GNU Radio 3.8 ne diffère pas significativement de celui de 3.7 à quelques subtilités près [11]. la plus visible étant probablement le passage d'une description des fichiers de configuration de GNU Radio Companion de XML vers YAML, brisant tout espoir de compatibilité avec les anciennes versions. Néanmoins, le passage d'un format à l'autre est simplifié par la méthode `gr_modtool makeyaml` qui se charge de créer le fichier de configuration de GNU Radio Companion en analysant le fichier d'entête de nos blocs. Évidemment ce générateur automatique de code ne peut pas savoir que nous désirons dynamiquement modifier le nombre de sorties du bloc en fonction de la configuration de l'oscilloscope – entre 1 et 4 voies de mesure – et nous devons modifier le fichier YAML généré en conséquence :

```
$ vi oscilloscope_oscilloscope.block.yml
- label: out
  domain: stream
  dtype: float
  multiplicity: ${channels}
```

où nous avons modifié le champ `multiplicity` de sa valeur initiale (4 dans notre configuration) par la variable `${channels}` qui définit dans notre interface le nombre de voies de mesure (paramètre aussi fourni à l'implémentation du bloc en C++).

ATTENTION : il est tentant de `cp -r` un bloc GNU Radio 3.7 pour le porter à 3.8. Parmi les déboirs que nous avons rencontrés par cette approche, une subtile modification du comportement de Python 3 par rapport à Python 2 porte sur la syntaxe de SWIG : nous prendrons soin de vérifier que `python/__init__.py` importe la classe du nouveau bloc par `from .oscilloscope_swig import *` : la version Python 2 utilisée par GNU Radio 3.7 omettait le "point" devant `oscilloscope_swig`, se traduisant par une erreur d'objet introuvable lors de l'exécution du bloc. La création d'un bloc par `gr_modtool` de GNU Radio 3.8 ne souffre pas de ce problème en créant un `python/__init__.py` avec le bon format d'appel à l'objet.

4.1 Modification dynamique des paramètres : *callbacks*

Pour conclure ce tour d'horizon de l'écriture d'un bloc pour GNU Radio 3.8, nous désirons ajouter la capacité à dynamiquement modifier certains paramètres de l'oscilloscope comme nous le ferions avec un vrai instrument, tel que la durée de la trace, la fréquence d'échantillonnage ou la gamme de tensions. Nous avons déjà pris soin de séparer les fonctions de configuration du programme principal : il nous suffit de déclarer ces fonctions comme *callback* appelées chaque fois que GNU Radio se rendra compte que nous voulons modifier un paramètre depuis un ascensceur (*QT GUI Range*) ou une série d'onglets (*QT GUI Chooser*). Nous nous inspirons de `gnuradio38/gr-trellis` des sources de GNU Radio en observant les appels à la fonction `set_0()` pour constater que 4 fichiers sont à adapter pour implémenter une fonction de callback :

- dans `include/block/block.h` qui est inclus par SWIG pour déclarer les liens entre Python et C++, nous déclarons le prototype de la fonction de callback comme un `virtual void`,
- dans `lib/block_impl.h` nous déclarons le prototype de la fonction de callback qui est implémentée dans ...
- ... `lib/block_impl.cc`.

— Finalement GNU Radio Companion est informé dans le fichier de configuration YAML du lien entre une variable et cette fonction de callback par le mot clé `callbacks` après les motifs définis dans le sous ensemble `template` :

```
templates:
  imports: import block
  make: block.block(${ip}, ${myrange}, ${myrate}, ${myduration}, ${channels})
  callbacks:
  - set_rate(${myrate})
```

par exemple pour modifier la fréquence d'échantillonnage (noter qu'il est peu judicieux d'utiliser `rate` qui semble être un mot clé de GNU Radio Companion).

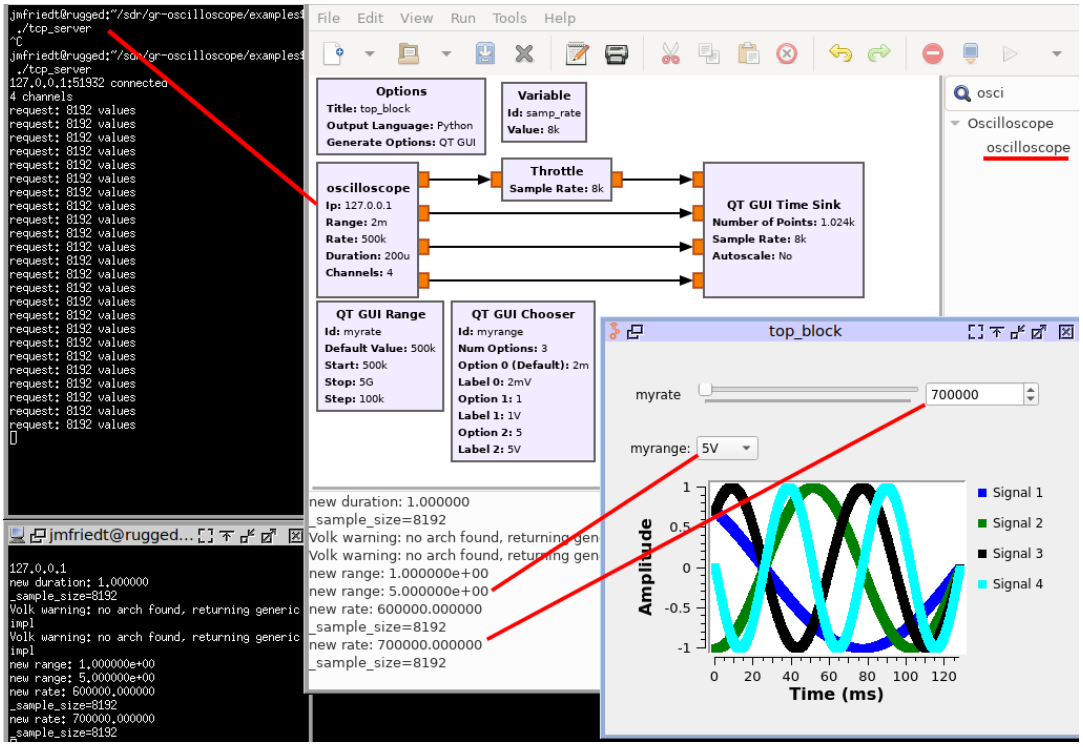


FIGURE 6 – Alimentation du bloc `gr-oscilloscope` par le serveur TCP/IP configuré sur 4 voies (gauche). Nous avons modifiés deux fois la gamme de tensions et une fois la fréquence d'échantillonnage au moyen des outils de sélection visibles dans la fenêtre en bas à droite. Chaque modification s'est traduite par l'appel à la fonction de *callback* correspondante (milieu).

5 Quelques applications

Nous avons introduit l'article par la démonstration du temps de vol d'un signal électromagnétique dans une série de cables coaxiaux pour en retrouver la longueur par corrélation d'un bruit large bande enregistré sur les divers canaux d'un oscilloscope radiofréquence. Nous proposons ici quelques applications plus courantes d'analyse spectrale.

5.1 Affichage de spectre

Un spectre n'a aucune prétention de mesure continue du flux de données acquises à un débit radiofréquence. Ainsi, en collectant des paquets de données, nous pourons identifier les diverses sources radiofréquences émettant à divers instants : moins la latence entre deux paquets de données est importante, meilleure est notre chance de capturer un émetteur intermittent.

L'application est certes un peu trivial mais permet de valider le bon fonctionnement de l'application dans un contexte bien connu (Fig. 7).

5.2 Spectre de bruit

Le corrolaire de l'affichage d'un spectre est d'afficher un spectre de bruit d'un composant. Il s'agit d'une mesure classique pour établir la limite de détection d'un instrument. Ici l'intérêt de l'analyse porte sur le passage d'une mesure de tension en unité arbitraire (valeurs sur 16 bits) en valeurs quantitatives avant de passer au spectre des fluctuations de tension. Pour

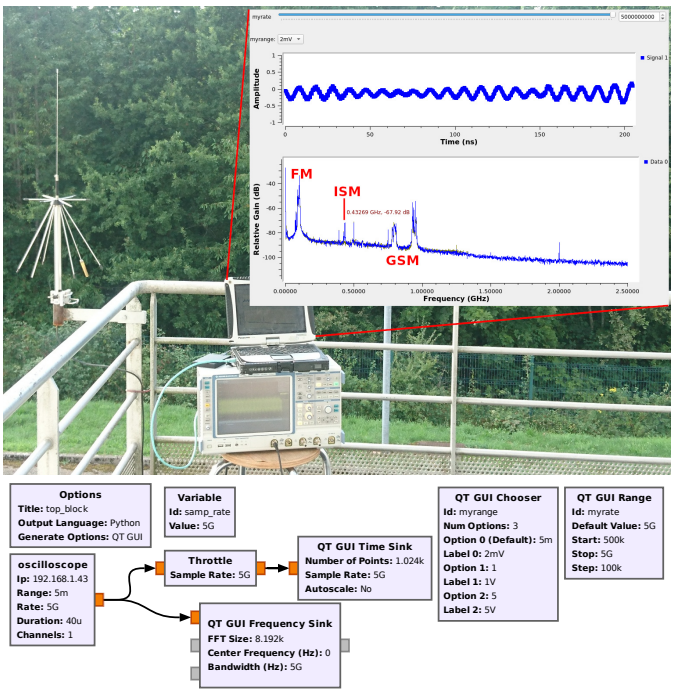


FIGURE 7 – Haut : contexte expérimental, avec une antenne discone large bande directement connectée sur l'entrée de l'oscilloscope configurée en 50 Ω. Bas : chaîne de traitement triviale avec affichage d'une voie d'oscilloscope dans le domaine temporel (haut) et spectral (bas) mettant en évidence les émissions puissantes dans la bande FM, notamment par Radio Campus proche du site de réception, les deux bandes GSM à 800 et 900 MHz ainsi qu'une émission ponctuelle dans la bande Industrielle, Scientifique et Médicale (ISM) lors de l'ouverture à distance d'un véhicule.

rappel, on passe d'une tension V à une puissance V^2/Z en divisant par l'impédance de charge Z (habituellement 50 Ω en radiofréquence), et l'affichage dans le domaine spectral au lieu du domaine temporel s'obtient par $|FFT(V)|^2/Z$. Enfin, la densité spectrale de bruit s'obtient en normalisant par la largeur RBW d'un intervalle de FFT soit la fréquence d'échantillonnage f_s divisée par le nombre de points N sur lequel est calculée la transformée de Fourier : $RBW = f_s/N$ permet d'obtenir $S = 10\log_{10}(|FFT(V)|^2/Z) - 10\log 10(f_s/N) + 30$ dBm/Hz (le +30 vient de la conversion de W en mW).

Le passage de GNU Radio 3.7 vers 3.8 affecte aussi les développeurs de systèmes embarqués : GNU Radio 3.7 est disponible depuis 2015 dans Buildroot et une série de patchs pour le passage à la version 3.8 vient d'être envoyée. Porter GNU Radio à Buildroot n'est pas en soi complexe : il est principalement nécessaire de bien évaluer les dépendances globales et celles spécifiques à chaque module pour les spécifier dans les fichiers du paquet. Une des seules difficultés était en relation avec FFTW qui, à l'époque, ne pouvait être compilée qu'avec un seul seul type de précision. Ainsi tous les blocs dépendant de cette bibliothèque ne pouvaient être activables que si FFTW était sélectionné en mode "single precision" (cette limitation est obsolète depuis le début de l'année). Le second point ayant posé quelques difficultés a été, dans le cas des processeurs ARM disposant d'un FPU NEON, de fournir explicitement les options au compilateur : il a été nécessaire d'ajouter `-mfpu=neon` pour VOLK de GNU Radio 3.8 sans quoi les tests qui permettent de déterminer si le compilateur supporte ce jeu d'instructions échouaient, donnant comme résultat l'absence de support pour le FPU dans VOLK. Dans tous les cas, nous rappelons ici les précepts de développement sur système embarqué qui proscrivent de compiler sur la plateforme cible aux ressources réduites, mais de toujours développer sur la plateforme hôte, puissante et munie d'un environnement de développement convivial, et cross-compiler des binaires à destination de la cible tel que le propose Buildroot.

6 Conclusion

Nous avons pris le prétexte d'acquérir des signaux radiofréquences d'un oscilloscope pour présenter la séquence de compilation de blocs de traitement dédiés pour GNU Radio 3.7 et 3.8 permettant notamment d'alimenter la chaîne d'analyse par des signaux synthétiques issus d'un serveur TCP/IP. Cette exploration a été l'occasion d'identifier le dysfonctionnement de certains instruments qui perdent des échantillons entre leurs voies supposées synchrones. L'application envisagée de réflectométrie, aussi applicable aux cas des RADARs passifs ou RADARs à bruit, démontre qu'un flux de données large bande discontinu peut avoir un intérêt. Pour rappel, les dépôts de `gr-oscilloscope` pour GNU Radio 3.7 et 3.8 se trouvent respectivement à `http://github.com/jmfriedt/gr-oscilloscope` et `http://github.com/jmfriedt/gr-oscilloscope38`.

Remerciement

J.-MF a découvert l'utilisation de l'oscilloscope radiofréquence comme source de données dans les applications de RADAR passif dans le laboratoire du Pr. M. Sato au CNEAS à Sendai, Japon (voir par exemple Fig.1 de [9]).

Références

[1] J.-M Friedt, *Matériel pour la radio logicielle*, GNU/Linux Magazine France **224** (Mars 2019)

[2] J.-M Friedt, *Contrôle d'instruments scientifiques : les protocoles GPIB, VXI11 et USBTMC*, GNU/Linux Magazine France **124** (Férier 2010)

[3] `https://www.rtl-sdr.com/gr-oscilloscope-using-an-oscilloscope-as-a-software-defined-radio/`

[4] J. Bloomer, *Power Programming with RPC*, UNIX Network Programming, O'Reilly & Associates (1992)

[5] S.D. Sharples, *VXI11 Ethernet Protocol for Linux* à `http://optics.eee.nottingham.ac.uk/vxi11/` (mis à jour 30 Juillet 2015, accédé Aout 2019) et `https://github.com/applied-optics/vxi11`

[6] D. Bodor, *Mesurez la vitesse de la lumière dans les câbles !*, Hackable **25** pp. 64–71 (Juillet-Aout 2018)

[7] J.-M Friedt, *RADAR passif par intercorrélation de signaux acquis par deux récepteurs de télévision numérique terrestre*, GNU Linux Magazine France **212**, pp.36– (2018)

[8] W. Daniau, *Interfaçage de code C++ pour Ruby et Python avec SWIG*, GNU/Linux Magazine **226** (Mai 2019)

[9] W. Feng, J.-M. Friedt, G. Cherniak, Z. Hu, M. Sato, *Direct path interference suppression for short range passive bistatic SAR imaging based on atomic norm minimization and Vandermonde decomposition*, IET Radar, Sonar and Navigation (2019), DOI : 10.1049/iet-rsn.2018.5214

[10] `https://sourceware.org/ml/libc-alpha/2017-08/msg00010.html`

[11] `https://wiki.gnuradio.org/index.php/GNU_Radio_3.8_OOT_Module_Porting_Guide`