GNU LINUX MAGAZINE / FRANCE

DÉVELOPPEMENT SUR SYSTÈMES UNIX, OPEN SOURCE & EMBARQUÉ

N°204
MAI 2017

FRANCE
MÉTRO. : 7,90 €
DOM/TOM : 8,50 €
BEL/LUX/PORT.
CONT. : 8,90 €
CH : 13 CHF
CAN : 14 $CAD

L 19275 - 204 - F: 7,90 € - RD

Python / Statistiques

PRÉDISEZ LES SAISIES DE VOS UTILISATEURS !

... avant qu'ils ne sachent eux-mêmes quoi écrire p.80

- Regroupez les documents d'un domaine pour créer votre corpus
- Calculez les probabilités d'apparition des mots
- Prédisez les prochains mots dans une phrase

Bas niveau / C++

UTILISEZ LA PROGRAMMATION FONCTIONNELLE EN C++ p.56

Traitement de signaux / Radio logicielle

RÉCEPTIONNEZ ET DÉCODEZ DES TRAMES RDS AVEC GNU/OCTAVE p.12

IoT / Programmation

IMPLÉMENTEZ VOTRE OBJET CONNECTÉ EN PYTHON SUR RASPBERRY PI p.44

Bash / Bizarre

CONSTRUISEZ UNE STRUCTURE OBJET « WITH »... EN SHELL ! p.70

Sécurité / Kernel

DÉCOUVREZ L'ASLR, LE MÉCANISME D'ALLOCATION MÉMOIRE ALÉATOIRE INTÉGRÉ AU NOYAU LINUX p.92

DÉPLOYEZ KUBERNETES SUR RASPBERRY PI - CRÉEZ UN PATCH POUR WEBOOB...

* WARGAME * CONFERENCES * CHALLENGES * WORKSHOPS *

NUIT DU HACK

XV

24->25 Juin 2017

New York Hôtel Convention Center

Disneyland Paris

www.nuitduhack.com

@hackerzvoice

GNU/Linux Magazine France
est édité par Les Éditions Diamond

10, Place de la Cathédrale - 68000 Colmar - France
Tél. : 03 67 10 00 20 – **Fax :** 03 67 10 00 21
E-mail : lecteurs@gnulinuxmag.com
Service commercial : abo@gnulinuxmag.com
Sites : www.gnulinuxmag.com – www.ed-diamond.com

Directeur de publication : Arnaud Metzler
Chef des rédactions : Denis Bodor
Rédacteur en chef : Tristan Colombo
Résponsable service infographie : Kathrin Scali
Réalisation graphique : Thomas Pichon
Responsable publicité : Valérie Fréchard,
Tél. : 03 67 10 00 27 – v.frechard@ed-diamond.com
Service abonnement : Tél. : 03 67 10 00 20
Impression : pva, Druck und Medien-Dienstleistungen GmbH, Landau, Allemagne

Distribution France : (uniquement pour les dépositaires de presse)
MLP Réassort : Plate-forme de Saint-Barthélemy-d'Anjou.
Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier. Tél. : 04 74 82 63 04

IMPRIMÉ en Allemagne - PRINTED in Germany
Dépôt légal : À parution, **N° ISSN :** 1291-78 34
Commission paritaire : K78 976

Périodicité : Mensuel
Prix de vente : 7,90 €

SUIVEZ-NOUS SUR :

https://www.facebook.com/editionsdiamond

@gnulinuxmag

p.21/22 DÉCOUVREZ TOUS NOS ABONNEMENTS MULTI-SUPPORTS !

LES ABONNEMENTS ET LES ANCIENS NUMÉROS SONT DISPONIBLES !

EN VERSION PAPIER ET PDF :
www.ed-diamond.com

Codes sources sur https://github.com/glmf

ÉDITO

Il est parfois édifiant de constater à quel point une idée simple peut nous faciliter la vie. Cette idée apparaît comme par enchantement et l'on se demande comment on a pu vivre jusque-là sans elle... et surtout comment on a fait pour ne pas l'avoir avant, cette satanée idée ! Prenons un exemple tout simple : lorsque l'on travaille sur un Raspberry Pi et que l'on écrit un article sur un PC, on a deux claviers, deux souris et deux écrans. Immanquablement, lorsque l'on souhaite écrire sur le Raspberry Pi on utilise le clavier... du PC ! Ce qui conduit à des résultats quelque peu aléatoires... Lorsque l'on souhaite déplacer la souris du PC, on utilise la souris du Raspberry Pi ce qui l'amène généralement, de rage, à prendre son envol sur quelques dizaines de centimètres (ce qui permet de comprendre l'utilité du câble qui relie la souris à l'ordinateur...), etc. Alors qu'en fait, on pourrait utiliser un bête commutateur USB et un commutateur HDMI qui existent depuis bien longtemps (les plus fortunés pouvant même se tourner directement vers des *switchs* KVM pour *Keyboard-Video-Mouse*) ! Il fallait juste avoir l'idée au bon moment... Les machines ne peuvent malheureusement (ou plutôt heureusement) pas anticiper, prédire ces idées qui nous font défaut. Par contre, il est possible statistiquement de déterminer le prochain mot auquel vous allez penser. En effet, en analysant un ensemble de documents, on peut calculer la probabilité pour un mot d'apparaître connaissant le mot précédent (ou les *n* précédents). La qualité des documents utilisés pour « l'apprentissage » est donc essentielle ! Vous n'obtiendrez pas la même précision suivant que vous allez employer :

- des documents hétérogènes avec par exemple des textes en anglais du XVIIIe siècle, des textes en français du XVIe siècle et des textes en italien du XVe siècle ;
- ou des documents homogènes issus de la même époque et du même domaine.

Dans le premier cas, vous allez réaliser une « grosse soupe » en mélangeant des époques, des langues et certainement des domaines (articles de presse, poésie, théâtre, etc.) et vous allez l'utiliser pour déterminer le prochain mot d'une phrase en français du XXIe siècle : un tirage purement aléatoire vous demandera moins de travail pour un taux de réussite à peu près équivalent.

Par contre dans le second cas, en travaillant de manière précise à l'élaboration de la base de documents, vous pourrez obtenir des résultats intéressants qui ne devront rien au hasard.

Ce mois-ci nous vous proposons d'étudier ce processus de prédiction dans son ensemble, depuis la création de la « base d'apprentissage » jusqu'à la proposition d'un mot devant compléter une phrase.

Vous pourrez également découvrir dans ce magazine comment réceptionner et décoder des trames RDS (mais oui, le RDS : les données numériques transmises sur les ondes FM), comment fonctionne le mécanisme d'allocation mémoire aléatoire du kernel et bien d'autres choses encore !

Je vous laisse prendre connaissance de ce contenu assez dense, je vous souhaite une bonne (lecture | année | soirée | journée)* et je vous retrouverai avec plaisir le mois prochain !

* Avec certains historiques de mots (comme « je vous souhaite une bonne »), la prédiction peut être difficile à effectuer... ;-)

Tristan Colombo

CONNECT ÉVOLUE !

LISEZ CE NUMÉRO ET PLUS DE 150 AUTRES EN LIGNE !

ACTUELLEMENT SUR CONNECT :

- **CE NUMÉRO**
- **et + de 150 autres numéros de GNU/Linux Magazine**

+

- **72 numéros Hors-Séries de GNU/Linux Magazine**

TOUT CELA À PARTIR DE 199 € TTC*/AN !

* Tarif France Métropolitaine

OFFRE DÉCOUVERTE CONNECT

1 MOIS GRATUIT, RÉSERVÉE AUX PROFESSIONNELS

Appelez le 03 67 10 00 28 et donnez le code « GLMF204 » pour découvrir Connect gratuitement pendant 1 mois !

Pour tous renseignements complémentaires, contactez-nous via notre site internet : **www.ed-diamond.com**, par téléphone : **03 67 10 00 28** ou envoyez-nous un mail à **connect@ed-diamond.com** !

SOMMAIRE

GNU/LINUX MAGAZINE FRANCE

N°204

PRINCIPAUX CHANGEMENTS ET AMÉLIORATIONS DE DJANGO 1.10

SÉBASTIEN CHAZALLET
[Ingénieur Logiciels libres]

MOTS-CLÉS : DJANGO, ROADMAP, LTS

Django, le framework pour les perfectionnistes avec des deadlines est dans une phase de grande mutation et améliore rapidement ses fonctionnalités dans la perspective d'une version 2.0. Au beau milieu de l'été 2016, sortait la version 1.10 de Django. Cet article se propose de faire le tour des nouveautés et d'anticiper ce qui nous attend pour le futur.

Le *framework* **Django** est en train de s'imposer comme le *framework* web de référence, à la fois pour l'écosystème Python, mais aussi de manière plus générale. Il est aujourd'hui la raison numéro un pour laquelle de plus en plus de développeurs apprennent le langage Python et il est omniprésent dans les offres d'emplois de nombre de sociétés innovantes.

Au-delà du *framework* lui-même, dont la qualité et les performances sont irréprochables, il faut surtout voir les possibilités qui sont données d'étendre ses fonctionnalités et de construire des applications ultra-modulaires. C'est ainsi que de très nombreuses extensions à Django ont vu le jour et se sont imposées. On peut citer pêle-mêle `django-model-utils`, `django-hstore`, `django-crispy-forms`, `django-geojson` ou encore `django-polymorphic`.

Un autre indicateur du succès du *framework* est le nombre, la taille et la qualité de projets écrits en Django et qui prennent de plus en plus d'importance. Nous pouvons citer **Wagtail**, **django-cms**, **LFS** (boutique en ligne) et surtout le magnifique **Taiga** qui fait travailler Django avec **AngularJS**.

La version 1.10 de Django est sortie durant l'été 2016 et a apporté de très nombreuses modifications parmi lesquelles on peut distinguer des ajouts de fonctionnalités et la suppression d'autres qui avaient été dépréciées.

Tout comme la version LTS précédente, Django supporte Python 2.7, 3.4 et 3.5.

1. PROCESSUS INTERNES

À l'instar de la plupart des projets *open source* d'une certaine importance, Django a mis au point un processus de gestion de version lui permettant de gérer plus facilement l'ajout de fonctionnalités et surtout sa maintenance.

Une nouvelle version sera ainsi publiée tous les huit mois et supportée pendant autant de temps. Une version toutes les trois versions sera une LTS (*Long Term Support*), ce qui signifie qu'elle bénéficiera de mises à jour de sécurité pendant au moins 3 ans. Ce seront donc des versions idéales pour une mise en production.

Voici un tableau récapitulatif ci-dessous.

Savoir quelle version utiliser dépend du statut de l'application.

Si elle est en développement actif, on souhaitera utiliser de nouvelles fonctionnalités. On va donc réaliser la migration de Django à chaque nouvelle sortie et l'on aura à chaque fois 8 mois pour mener cette migration à bien et bénéficier de ces nouvelles fonctionnalités.

Si elle est en exploitation, on souhaitera alors effectuer les migrations d'une version LTS à une autre, ce qui est rendu possible grâce au support étendu de trois ans. On aura alors à chaque fois une année pour effectuer la migration.

Le support standard va corriger toutes les anomalies, quelles qu'elles soient tandis que le support étendu ne corrigera que les problèmes liés à la sécurité ou pouvant engendrer des pertes de données.

L'ajout de fonctionnalités ne se fera que lors de la sortie d'une nouvelle version et le cycle de sortie de toute nouvelle version sera le suivant :

- Propositions (en anticipation) ;
- Développement :
  - → se termine par la sortie d'une – et une seule – version alpha ;
  - → tout ce qui n'a pas pu être terminé à cet instant est reporté à la version suivante ;
- Corrections :
  - → un mois pour publier la bêta ;
  - → un mois de plus pour une RC1 :
  - → 15 jours pour publier la version finale ou, si nécessaire, une RC2 et 15 jours de plus pour la version finale ;
- Phase de support standard :
  - → les nouvelles versions feront varier le troisième numéro (par exemple 1.10.1, puis 1.10.2).

Enfin, notons qu'une politique de dépréciation a été mise en place. La toute première version après une LTS supprimera les fonctionnalités dépréciées dans les versions non LTS tandis que la version suivante supprimera celles de la version LTS précédente. Aucune suppression d'une fonctionnalité dépréciée n'aura lieu dans la LTS.

Ainsi, la version 2.0 supprimera les fonctionnalités dépréciées dans les versions 1.9 et 1.10 tandis que la version 2.1 supprimera les fonctionnalités dépréciées dans la version 1.11.

Version	Sortie	Fin du support standard	Fin du support étendu	Dépréciation
1.8 (LTS)	Avril 2015	Décembre 2015	Août 2016	
1.9	Décembre 2015	Août 2016	Avril 2017	
1.10	Août 2016	Avril 2017	Décembre 2017	
1.11 (LTS)	Avril 2017	Décembre 2017	Avril 2020	
2.0	Décembre 2017	Août 2018	Avril 2019	1.9, 1.10
2.1	Août 2018	Avril 2019	Décembre 2019	1.11
2.2 (LTS)	Avril 2019	Décembre 2019	Avril 2022	
3.0	Décembre 2019	Août 2020	Avril 2021	2.0, 2.1
3.1	Août 2020	Avril 2021	Décembre 2021	2.2
3.2 (LTS)	Avril 2021	Décembre 2021	Avril 2024	

2. AJOUT DE FONCTIONNALITÉS

2.1 Gestion du middleware

Django étant une application hautement extensible, elle définit un canal de traitement et permet à toute extension ou application d'intervenir à toute étape de ce canal de traitement.

Ceci est fait par la gestion de *middleware* avec une méthode **process_request** pour gérer des actions au moment où l'on reçoit une requête, puis une méthode **process_response** pour gérer des actions à exécuter au moment où la réponse est construite. En cas d'exceptions, un autre type de réponse est à construire pour signifier cette exception à l'utilisateur, ce qui est généré par la méthode **process_exception**.

Dans un monde parfait, tous les *middlewares* exécuteraient une fois **process_request**, puis en sortie, soit **process_response**, soit **process_exception**. Or, ce n'est pas ce qu'il se passe. On constate que certains *middlewares* renvoient une réponse dans **process_request**, court-circuitant le fonctionnement des *middlewares*.

C'est pour éviter ceci et pour imposer un fonctionnement cadré que le fonctionnement des *middlewares* a été revu. Maintenant, tous les *middlewares* auront une structure similaire à ce qui suit (extrait directement de la documentation) :

```
class SimpleMiddleware(object):
  def __init__(self, get_response):
     self.get_response = get_response
     # One-time configuration and initialization.

  def __call__(self, request):
     # Code to be executed for each request before
     # the view is called.

     try:
         response = self.get_response(request)
     except Exception as e:
         # Code to handle an exception that wasn't caught
         # further up the chain, if desired.
         ...

     # Code to be executed for each request/response after
     # the view is called.

       return response
```

Chaque *middleware* sera maintenant construit au moment du lancement du serveur (par conséquent, la méthode d'initialisation sera appelée une et une seule fois) et sera utilisé à chaque requête (via la méthode d'appel).

Comme Django utilise plusieurs *middlewares* (et vous permet de créer les vôtres), ces derniers sont chaînés. Ainsi, l'objet **get_response** peut être une vue ou le prochain *middleware* dans la chaîne, mais ce détail n'est pas important à ce niveau. Tout ce qui compte est de savoir qu'il s'agit d'une fonction qui traite une requête et renvoie une réponse.

Si d'aventure vous êtes sollicité pour écrire votre propre *middleware* (et cela est un excellent moyen trop souvent ignoré pour gérer des opérations transverses ou étant dupliquées dans des vues dans laquelle elles représentent un travail d'aspect secondaire), vous saurez que vous pouvez agir avant ou après l'utilisation de la vue.

En ce qui concerne les impacts qui seront plus visibles, ils concernent l'aspect configuration. Ainsi, on remplacera **MIDDLEWARE_CLASSES** – qui devient déprécié – par **MIDDLEWARE** :

```
MIDDLEWARE = [
    'django.middleware.security.
SecurityMiddleware',
    'django.contrib.sessions.middleware.
SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.
AuthenticationMiddleware',
    'django.contrib.messages.middleware.
MessageMiddleware',
    'django.middleware.clickjacking.
XFrameOptionsMiddleware',
]
```

Ce dernier contiendra un contenu similaire, mais les classes citées auront été modifiées.

Si vous utilisez des extensions de Django qui nécessitent l'inclusion d'un *middleware*, il faudra vous assurer que ce dernier ait migré (il est possible de le migrer d'une manière qui le rende rétrocompatible).

Pour rappel, si votre but n'est que de transmettre des variables à la vue, il est préférable d'utiliser des *templates processor* plutôt que des *middlewares*, car c'est à la fois plus simple et mieux adapté.

2.2 Sécurité

Argon2 [1] est une nouvelle méthode permettant de *hasher* les mots de passe et elle est relativement performante. À tel point que Django recommande de l'utiliser en lieu et place des autres. Pour ce faire, il est cependant nécessaire d'installer une extension :

```
$ pip install django[argon2]
```

Pour l'ajouter à la liste des méthodes permettant de *hasher* les mots de passe, il faut modifier ainsi la configuration :

```
PASSWORD_HASHERS = [
    'django.contrib.auth.hashers.
Argon2PasswordHasher',
```

```
    'django.contrib.auth.hashers.
PBKDF2PasswordHasher',
    'django.contrib.auth.hashers.
PBKDF2SHA1PasswordHasher',
]
```

Pour en faire l'outil de *hashage* par défaut, il suffit de le mettre en premier. En effet, par défaut, **PBKDF2** est celui que Django choisi parce qu'il ne nécessite pas l'installation d'une bibliothèque supplémentaire. Ce dernier a par ailleurs été amélioré.

Notez que si vous changez de méthode de *hashage*, vous devrez mettre à jour les mots de passe déjà stockés. Pour que ceci se produise, il est important de garder toutes les méthodes de *hashage* utilisées dans la base de données dans la liste **PASSWORD_HASHER**. Le changement du mot de passe peut se faire naturellement au moment de la connexion d'un utilisateur, il se connecte avec succès, on a le mot de passe en clair et il vérifie l'ancienne empreinte, on va donc pouvoir calculer la nouvelle empreinte.

L'autre méthode consiste à faire une empreinte nouvelle non pas à partir du mot de passe, mais à partir de l'empreinte ancienne. Vous trouverez, comme d'habitude, toutes les informations utiles dans la documentation **[2]**.

À noter que les anciennes méthodes de *hashage* considérées comme faibles ont été retirées de la liste.

Django intègre un *middleware* CSRF qui permet de gérer les problématiques de sécurité CSRF (*Cross Site Request Forgery*) ; il a été amélioré sur deux points : la vue par défaut en cas d'échec de représentation du jeton accepte maintenant un attribut **template_name** optionnel facilitant sa personnalisation et surtout, dans le but d'améliorer la sécurité, le mécanisme change le *token* à chaque requête.

On notera également que les vues **login** et **logout** ont été légèrement amendées pour simplifier leur utilisation.

La commande **./manage.py clearsessions** supprime maintenant toutes les sessions basées sur des fichiers.

2.3 Données, champs et modèles

Django 1.10 supprime le support de **PostgreSQL** 9.1, au moment où cette dernière est sur le point de voir son support supprimé et où la version 9.5 est sortie (ce que vous avez pu lire dans *GMLF n°194*).

Cette version rajoute le support de la recherche *full text* pour PostgreSQL. On peut maintenant faire une recherche dans plusieurs champs et utiliser différentes méthodes pour classer les résultats par pertinence, avec la possibilité de faire dépendre le choix de la méthode à celui de la langue.

Il existe maintenant trois méthodes permettant le support des trigrammes. Cette méthode permet de retrouver des données ressemblantes, c'est-à-dire des données proches de ce que l'on recherche, la notion de proximité reposant sur une mesure de similarité basée sur le nombre de correspondances lorsque l'on prend les caractères trois par trois (d'où le terme de trigramme).

Voici un exemple tiré de la documentation :

```
>>> City.objects.filter(name__trigram_
similar="Middlesborough")
['<City: Middlesbrough>']
```

On voit que la donnée a été trouvée malgré le fait que l'orthographe du mot ait été approximative. Notons que cette fonctionnalité est disponible pour PostgreSQL et qu'elle nécessite que l'extension de PostgreSQL nommé **pg_trgm** soit activée **[3]**.

Au-delà de cette énorme amélioration, il existe également de plus modestes changements qui ont néanmoins un impact certain : on peut citer le fait que les champs **HSTORE** voient leurs clés et valeurs converties en chaînes de caractères pour simplifier et homogénéiser leur utilisation.

De plus, pour les bases de données le supportant, il est possible de récupérer les identifiants attribués par la base de données lors d'une opération **bulk_insert**.

Enfin, il faut également citer un travail important sur **django.contrib.gis**, la bibliothèque permettant de traiter les informations géographiques ou spatiales. Citons l'ajout des méthodes **unary_union** ou **covers** pour **GEOSGeometry**, l'ajout de la méthode **statistics** ou l'amélioration de la méthode data pour **GDALBand** et quelques améliorations spécifiques pour MySQL ou PostGis.

Le modèle **User** dispose maintenant de validateurs explicites pour n'autoriser que les caractères ASCII pour Python2 et les caractères Unicode pour Python3. Ce comportement peut cependant être reconfiguré, les validateurs n'étant pas dépendants de la version du langage Python.

Le modèle **Site** (**django.contrib.sites**, permettant de gérer plusieurs sites utilisant un seul applicatif Django) supporte maintenant les clés naturelles (une clé naturelle permet de retrouver un enregistrement d'une manière unique et plus élégante que l'utilisation d'un identifiant - ceci est particulièrement utile pour l'import ou l'export de données).

Il a été rajouté le support de la sérialisation des objets de type **Enum**.

La procédure de migration va maintenant, avant de s'appliquer, rechercher des inconsistances dans l'historique des migrations et générer une erreur s'il en est trouvé. De plus,

elle supporte maintenant les migrations non atomiques et elle génère les signaux `pre_migrate` et `post_migrate`.

3. AMÉLIORATIONS

3.1 Interface d'administration

L'interface d'administration subit un certain nombre de changements mineurs :

- le lien vers le site n'est plus `/`, mais pointe vers `request.META['SCRIPT_NAME']`, ce qui permet de fonctionner également lorsque `/` n'est pas la racine du site ;
- tout le code **JavaScript** est maintenant embarqué, ce qui auparavant, nous empêchait d'activer l'entête `Content-Security-Policy` ;
- le message de succès après ajout ou modification tout comme les objets sélectionnés dans `ModelAdmin.raw_id_fields` dispose maintenant d'un lien vers le formulaire de modification de l'objet ;
- la classe `InlineModelAdmin` dispose maintenant d'un nouvel attribut `classes` qui permet de spécifier les classes des champs en ligne, ce qui peut permettre de choisir `collapse` pour les cacher (avec un lien *show* pour les faire apparaître seulement à la demande) ;
- le `block` contenant les icônes d'actions est tout le temps visible, y compris si l'on n'a pas la permission d'ajout (l'icône d'ajout n'apparaît cependant pas), ce qui permet de faciliter la personnalisation de ce dernier ;
- pour `DateFieldListFilter`, il a été rajouté les choix *Pas de date* et *a une date* lorsque le champ est `nullable`.

3.2 Scripts

Les différents scripts accessibles depuis `manage.py` ont été améliorés. On peut rapidement citer quelques exemples :

- le script `makemigrations` affiche maintenant le chemin des fichiers de migration générés ;
- le script `shell` dispose d'une option `--interface` (ou `-i`) qui accepte maintenant `python` comme option. Au passage, `shell --plain` est maintenant dépréciée ;
- le script `shell` dispose d'une option `--command` qui permet de lancer une commande avec l'environnement Django, puis de quitter ensuite ;
- ajout d'un attribut `BaseCommand.requires_migration_checks` qu'il faut mettre à `True` pour vérifier si la base de données correspond aux migrations disponibles (avant, c'était systématiquement réalisé) ;
- le script `inspectdb` permet de spécifier une table particulière à inspecter en la passant en paramètre ;
- la commande `call_command` accepte maintenant un objet en premier argument, pour faciliter les tests.

3.3 Divers

On peut citer pour terminer de nombreux petits changements qui vont cependant avoir leur importance :

- l'instruction de gabarit `static` utilise maintenant en sous main `django.contrib.staticfiles` si cette dernière est dans les `INSTALLED_APPS`, ce qui fait qu'on peut maintenant toujours utiliser `{% load static%}` sans avoir à s'inquiéter de savoir si l'application `staticfiles` est ou n'est pas installée ;
- les formulaires et widgets *Media* utilisent maintenant `django.contrib.staticfiles` ;
- on peut plus facilement gérer la manière dont le script `collectstatic` va fonctionner en personnalisant `AppConfig` ;
- les formulaires utilisent maintenant l'attribut `required` ;
- la classe `View` peut être importée depuis `django.views`.

3.4 Dépréciations

Avec cette nouvelle version, de nouveaux comportements sont dépréciés et vont donc bientôt être supprimés. Ainsi, la version 1.9 avait ajouté la possibilité de faire ceci :

```
e.related_set.set([obj1, obj2])
```

Ce qui permettait d'écrire ceci, moins élégant :

```
e.related_set = [obj1, obj2]
```

La nouvelle étape consiste à ce que Django 1.10 déprécie cette ancienne façon de faire, laquelle ne sera donc bientôt plus possible.

Les méthodes `accessed_time`, `created_time` et `modified_time` (permettant de récupérer les informations temporelles non liées à un fuseau horaire) sont dépréciées au profit de, respectivement, `get_accessed_time`, `get_created_time` et `get_modified_time`.

La bibliothèque `django.contrib.gis` subit aussi des changements, puisque les méthodes `get_srid` et `set_srid` de `GEOSGeometry` sont dépréciées au profit de la propriété `srid`, tout comme `get_x`, `set_x`, `get_y`, `set_y`, `get_z`, `set_z`, `get_coords` et `set_coords` de la classe `Point` le sont au profit des propriétés `x`, `y`, `z` et `tuple`.

Enfin, la propriété `cascaded_union` de la classe `MultiPolygon` est dépréciée en faveur de la propriété `unary_union` et la fonction `django.contrib.gis.utils.precision_wkt` est dépréciée en faveur de `WKTWriter`.

Comme on le voit, l'utilisation des propriétés se généralise et cela est vrai partout. Ainsi, les méthodes `User.is_authenticated` et `User.is_anonymous` sont maintenant des propriétés (qui seront comparables en utilisant `==` et `!=` dans Django 1.10.1 **[4]**).

La liste exhaustive peut être consultée sur le site de la documentation, mais on a cité ici les choses les plus utilisées.

3.5 Suppressions

Toutes les fonctionnalités dépréciées dans *Django 1.8* sont maintenant supprimées. Il y en a un très grand nombre, mais on notera surtout la suppression de la possibilité d'utiliser `reverse` pour récupérer l'URL d'un chemin utilisant des points et l'impossibilité d'utiliser ces mêmes chemins pour `LOGIN_URL` et `LOGIN_REDIRECT_URL`.

De plus, les modules `django.core.context_processors`, `django.db.models.sql.aggregates` et `django.contrib.gis.db.models.sql.aggregates` sont supprimés tout comme les méthodes ou propriétés de `django.db.sql.query.Query` nommées `aggregates`, `aggregate_select`, `add_aggregate`, `set_aggregate_mask` et `append_aggregate_mask`.

Un autre point important concerne les méthodes dépréciées suite à la refonte de l'API META, lesquelles sont maintenant supprimées, soit :

- `get_field_by_name` ;
- `get_all_field_names` ;
- `get_fields_with_model` ;
- `get_concrete_fields_with_model` ;
- `get_m2m_with_model` ;
- `get_all_related_objects` ;
- `get_all_related_objects_with_model` ;
- `get_all_related_many_to_many_objects` ;
- `get_all_related_m2m_objects_with_model`.

Enfin, au niveau de la configuration, il faut configurer tout ce qui se rapporte aux gabarits dans `TEMPLATES`, les éléments suivants n'étant plus pris en compte :

- `ALLOWED_INCLUDE_DIRS` ;
- `TEMPLATE_CONTEXT_PROCESSORS` ;
- `TEMPLATE_DEBUG` ;
- `TEMPLATE_DIRS` ;
- `TEMPLATE_LOADERS` ;
- `TEPLATE_STRING_IF_INVALID`.

Là encore, la liste complète est à lire dans la documentation, ainsi que les notes de dépréciations de *Django 1.8*, un peu plus détaillées **[5]**.

CONCLUSION

En peu de temps, Django est devenu une solution de référence et ceci bien au-delà du cercle restreint des Pythonistes. Il est d'ailleurs l'une des raisons principales pour laquelle les développeurs apprennent Python.

Il faut dire qu'il jouit d'une image de simplicité et de performance qui n'est absolument pas usurpée, mais pour pouvoir s'améliorer de manière continue sans tomber dans le risque de devenir l'usine à gaz, ce que de très nombreuses applications prometteuses sont devenues malgré elles dans le passé, il lui faut continuer d'innover et de grandir sans renier ses principes et en gardant une cohérence très forte et une exigence très élevée.

Force est de constater que c'est exactement ce qu'il se passe, puisqu'à chaque cycle, la solution s'améliore, gagne en cohérence, améliore ses pratiques et corrige ses petits défauts, avant que ceux-ci ne deviennent trop grands pour pouvoir se le permettre.

Django est, à mon sens, devenue la solution idéale pour faire du développement web et est celle qui a le plus brillant avenir devant elle. Cette dernière mouture améliore grandement la sécurité de programmation, facilite l'application de bonnes pratiques cohérentes et apporte son lot de nouveautés qui vont devenir assez rapidement indispensables. ■

RÉFÉRENCES

[1] Argon2, sur Wikipédia : https://en.wikipedia.org/wiki/Argon2 et sur GitHub : https://github.com/p-h-c/phc-winner-argon2.

[2] Mise à jour de mots de passe : https://docs.djangoproject.com/en/dev/topics/auth/passwords/#wrapping-password-hashers.

[3] Extension PostgreSQL pg_trgm : https://www.postgresql.org/docs/current/static/pgtrgm.html.

[4] Corrections : https://docs.djangoproject.com/en/1.10/releases/1.10.1/.

[5] Suppressions : https://docs.djangoproject.com/en/1.10/releases/1.10/#features-removed-in-1-10.

RADIO DATA SYSTEM (RDS)

JEAN-MICHEL FRIEDT
[Institut FEMTO-ST, dpt. Temps-fréquence, Besançon]

MOTS-CLÉS : RDS, DÉMODULATION, DÉCODAGE, CORRECTION D'ERREURS

RDS – Radio Data System – est le mode numérique de communication exploité par les stations FM de la bande commerciale 88–108 MHz pour indiquer à l'utilisateur des informations telles que le nom de la station reçue, du texte libre tel que les informations en cours de diffusion ou un titre de musique, ainsi que l'heure ou la nature du programme diffusé. Nous nous proposons, à partir du signal analogique reçu par un récepteur de télévision numérique terrestre (DVB-T) exploité comme récepteur radiofréquence généraliste, d'analyser les diverses étapes de démodulation et de décodage, pour finalement conclure sur une exploration des méthodes de détection et de correction d'erreurs.

La bande commerciale FM, comprise entre 88 et 108 MHz, est divisée pour allouer une bande de 200 kHz à chaque station (Fig. 1, en haut). Chaque station redivise chaque tranche du spectre radiofréquence qui lui est allouée en trois sous-segments : le son, avec d'abord la somme des signaux destinés à l'oreille droite et l'oreille gauche, puis si la transmission est en stéréo, la différence entre oreille droite et oreille gauche (afin qu'un récepteur mono puisse recevoir une station stéréo), et finalement un signal numérique – RDS (*Radio Data System*, Fig. 1 en bas) – comportant des informations telles que le nom de la station (Fig. 2), ou du texte libre tel que le titre d'une émission ou d'un morceau de musique. Un récepteur est informé qu'une émission est en stéréo par la présence d'un pilote – un signal périodique continu – à 19 kHz. La sous-porteuse de la transmission numérique se fait à 57 kHz générée comme trois fois le pilote si l'émetteur est stéréo, hypothèse que nous ne ferons pas au cours de nos traitements dans lesquels nous nous efforcerons de reproduire une copie locale de la sous-porteuse à 57 kHz. La bande passante du signal numérique est de l'ordre de 5 kHz.

Un lecteur simplement désireux de lire le contenu des informations numériques ainsi transmises pourra se contenter

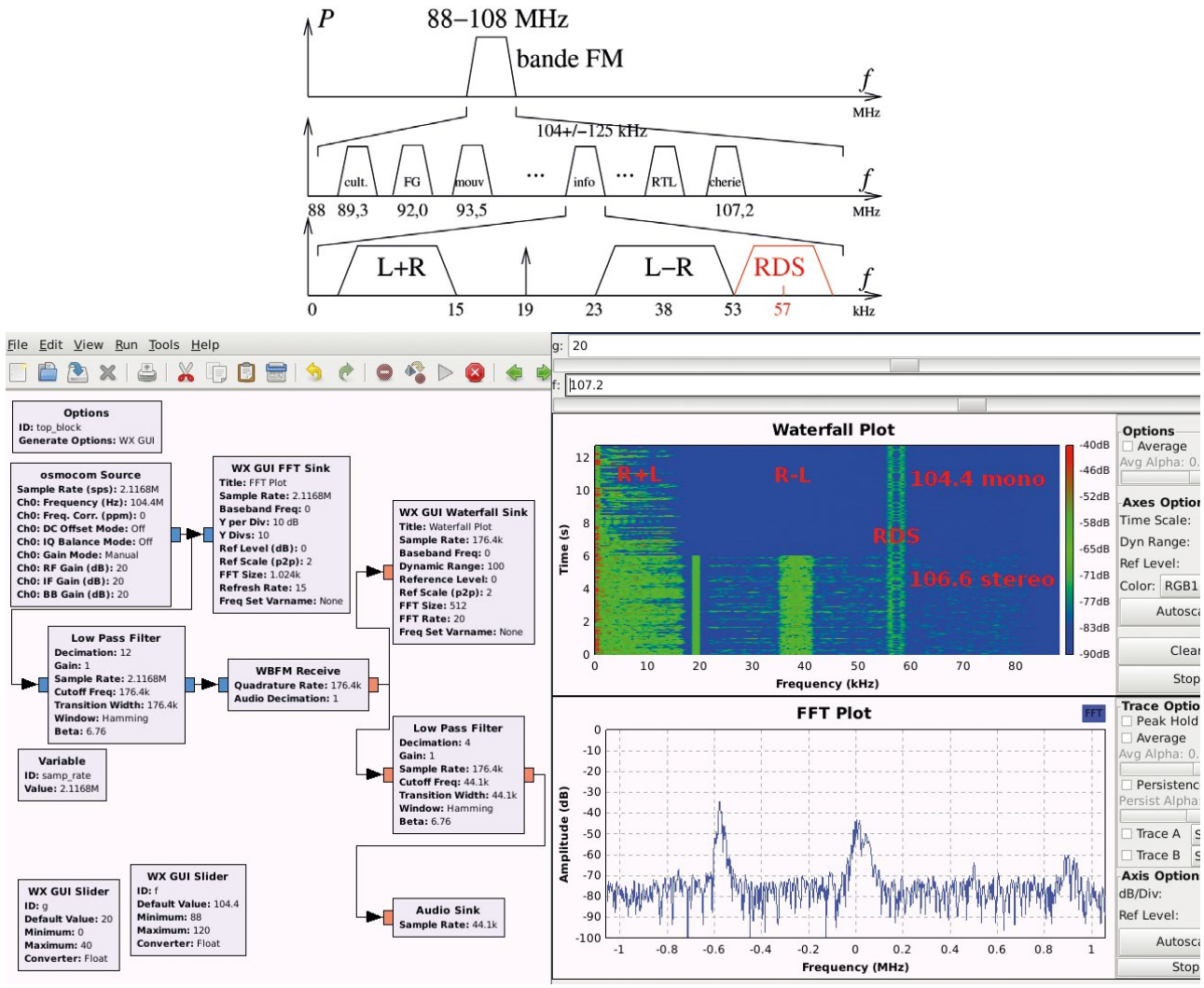


Fig. 1 : En haut : la bande de fréquence qui nous intéresse – RDS – se trouve à 57 kHz de la porteuse de chaque station FM : elle est donc accessible par transposition de fréquence, pour ramener le signal numérique en bande de base (autour de la fréquence nulle), après la démodulation FM de la station écoutée.
En bas : les divers signaux émis par une station de la bande commerciale FM sont visibles sur le mode waterfall après le démodulateur WBFM. Nous avons changé la fréquence en milieu d'acquisition entre une station stéréo et une station mono, qui toutes deux émettent un signal d'identification RDS.

d'utiliser un composant intégré qui se charge de tout le travail, par exemple chez **ST** le **TDA7330 [1]** ou **TDA7478** (*single chip RDS decoder*), voire le récepteur FM intégré **RDS5807** de **RDA MicroElectronics. [2]** exploite dans la même veine un **TEA5764** pour synchroniser un réseau de capteurs, bien que toutes ces références semblent obsolètes et inaccessibles chez les principaux fournisseurs. Ce faisant, nous n'aurons rien appris du mode d'encodage, de la nature des informations transmises, mais surtout de la méthode utilisée pour retrouver des bits corrompus lors de la transmission. Tous ces concepts seront appréhendés lors du décodage logiciel des trames, avec analyse étape par étape de la séquence suivie pour passer d'un signal radiofréquence modulé en fréquence (FM commerciale) à des phrases intelligibles. Comme d'habitude, cette compréhension permettra de mieux appréhender des utilisations malicieuses du protocole ou de le détourner de son usage initial **[1][3][4]**. Tous les prototypages s'aideront de l'outil d'implémentation de traitement du signal **GNURadio [5]** pour acquérir le signal, suivi d'une mise en œuvre des algorithmes de décodage sous **GNU/Octave [6]** en s'efforçant de revenir aux bases, et sans passer par des bibliothèques de haut niveau telles que la *communication toolbox* **[7]** qui nous cacheraient la compréhension détaillée du flux de traitement.

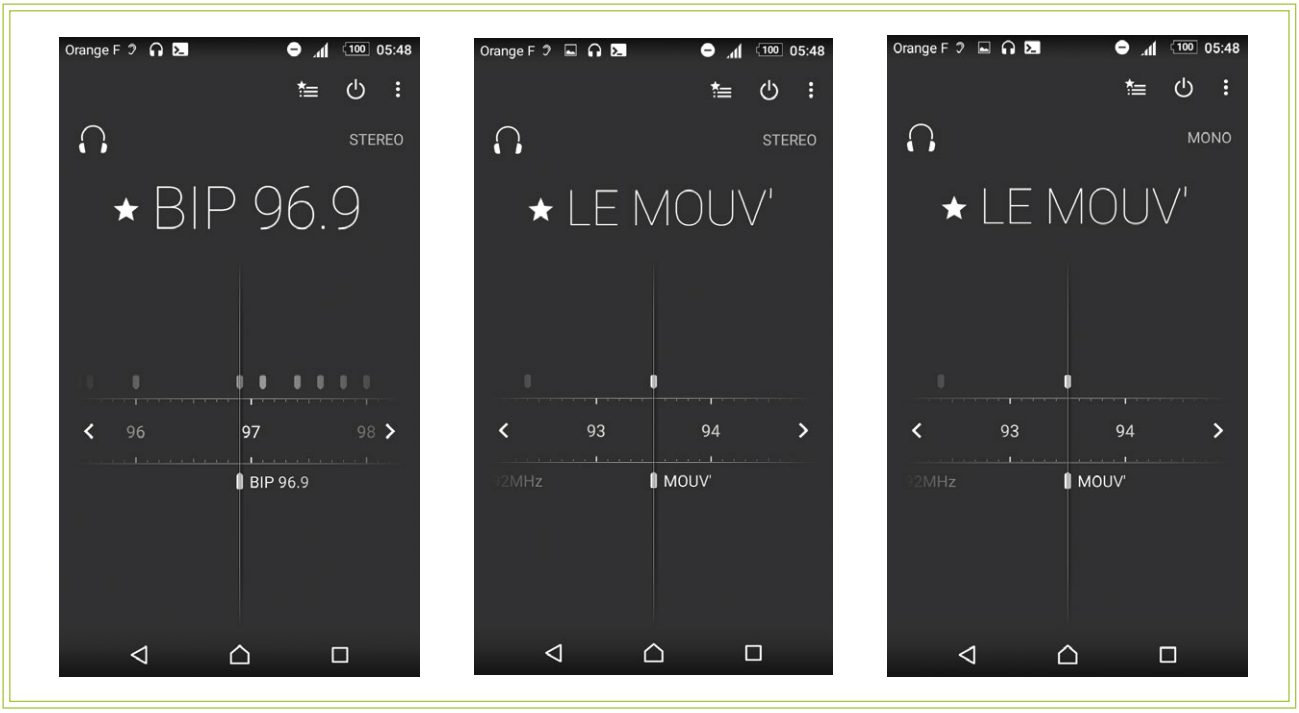


Fig. 2 : Réception de diverses stations FM commerciales, avec affichage du nom de la station tel que transmis par RDS. Noter que Le Mouv' est parfois perçu comme stéréo, parfois comme mono, sans que cela empêche d'afficher son identifiant.

Le décodage d'un flux de communication numérique sur porteuse radiofréquence nécessite toujours de résoudre les problèmes de synchronisation d'oscillateurs distants, à savoir l'oscillateur radiofréquence qui génère la porteuse sur laquelle l'information est transmise, et le débit de communication. Le récepteur transpose le signal radiofréquence en bande de base par le mélange avec un oscillateur local LO (Fig. 3). La phase qui encode l'information sur la porteuse ne sera exploitable que si une copie de l'oscillateur de l'émetteur – RF – est générée en local : LO est asservi sur RF selon des mécanismes qui seront décrits plus bas (section 2). Une fois que des bits seront obtenus en sortie de la démodulation, l'information numérique est échantillonnée périodiquement pour extraire l'état de chaque bit et former des trames. Ici encore, le rythme du décodage n'a aucune raison d'être synchronisé avec le rythme de génération des données : même si la valeur nominale du débit de communication est connue, un écart entre l'oscillateur qui génère les données numériques sur l'émetteur et celui qui cadence l'échantillonnage sur le récepteur finira au bout d'un moment par induire une désynchronisation. Ici encore, un asservissement sera nécessaire, qui sera décrit en section 7.

Dans la séquence d'acquisition par GNURadio proposée en figure 4 qui vise à démoduler un signal dans la bande FM, en extraire la sous-porteuse comportant les informations numériques, et stocker le résultat dans un fichier binaire pour post-traitement, deux caractéristiques déterminent la qualité de la démodulation et la puissance de calcul nécessaire : les fréquences de coupure pour isoler la bande qui nous intéresse par filtrages successifs, et les décimations (ne prendre qu'un point sur **N** pour une décimation d'un facteur **N**) pour réduire le débit de données. Un récepteur de télévision numérique terrestre (DVB-T) basé sur le convertisseur analogique-numérique **RTL2832U** travaille nécessairement avec une fréquence d'échantillonnage comprise entre 1,5 et 2,4 MHz, soit bien plus que l'encombrement spectral d'une bande FM. Notre première tâche consiste donc à décimer ce flux de données pour n'avoir qu'environ 200 kéchantillons/s, respectant ainsi l'encombrement spectral d'une unique station FM. Cependant, la décimation ne peut se faire sans avoir atténué le signal des autres stations se trouvant à plus de 100 kHz de la bande qui nous intéresse, sinon leurs signaux se ramèneront dans la bande de base par

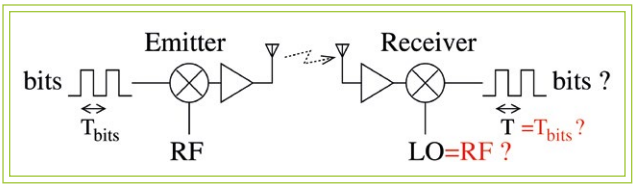


Fig. 3 : Problème de synchronisation des oscillateurs distants qui portent le signal et cadencent le débit de donnée. Le décodage ne sera fonctionnel que si les deux oscillateurs du récepteur – LO et échantillonnage de l'état de l'information numérique – sont asservis sur leurs homologues du côté de l'émetteur.

repliement spectral **[8]** lors de la décimation. Par conséquent, nous commençons par un filtre passe-bas qui isole la station d'intérêt, puis décimons suffisamment pour réduire le flux à un débit de l'ordre de 200 kéchantillons/s. L'autre intérêt de la décimation est de permettre des transitions des filtres en aval plus nettes avec un même nombre de coefficients : en effet, la bande de transition d'un filtre est de l'ordre de la fréquence d'échantillonnage divisée par le nombre de coefficients. Ainsi, il est très lourd de calculer un filtre présentant une bande étroite à taux d'échantillonnage élevé, tandis que le même résultat s'obtient à ressource de calcul réduite en décimant judicieusement auparavant. Les 200 kéchantillons/s que nous venons d'obtenir portent toutes l'information qui encode la station FM commerciale qui nous intéresse : le bloc WBFM démodule le signal et fournit un flux de données qui peut lui aussi être décimé si seule la composante audiofréquence nous intéressait. Cependant, nous désirons décoder le signal RDS qui se trouve autour d'une sous-porteuse à 57 kHz, donc nous ne pouvons pas encore décimer, mais devons attendre de transposer la sous-porteuse de 57 kHz vers la bande de base par un **Xlating FIR Filter [9]** pour une fois de plus décimer, les 200 kéchantillons/seconde étant bien trop rapides pour les quelques kHz occupés par le signal numérique. Ainsi, le filtre FIR (filtre à réponse impulsionnelle finie) est conçu pour isoler le signal numérique sur une bande de quelques kHz et rejeter les autres composantes spectrales avant la décimation qui fournit un débit de données raisonnable pour décoder le signal numérique. Attention : la fréquence d'échantillonnage qui définit le FIR doit être la fréquence d'échantillonnage en entrée du Xlating FIR Filter, donc tenir compte de la décimation du premier filtre passe-bas et éventuellement du démodulateur FM. Dans notre cas, ce filtre (variable `taps` qui sera appelée dans le champ du même nom dans le Xlating FIR Filter) est défini par l'expression Python : `filter.firdes.low_pass_2(1, samp_rate/8, 2000, 500, 60)` pour dire que le filtre passe-bas a décimé d'un facteur 8, pas de décimation sur la démodulation WBFM, 2 kHz de fréquence de coupure et une transition sur 500 Hz. À l'issue de ce traitement, nous avons isolé la bande du signal RDS que nous désirons décoder.

RDS est actuellement accessible pour les utilisateurs de GNURadio au travers du module **gr_rds**, initié par Dimitrios Symeonidis **[11]** et maintenu à ce jour par Bastian Bloessl **[12]**. Reprendre un code existant peut fournir de l'inspiration, voire valider le bon fonctionnement de l'installation matérielle de réception, mais n'amène rien à la compréhension de la mise en œuvre des diverses étapes de démodulation et de décodage, que nous tenterons d'appréhender ici. Les divers contributeurs à gr_rds ne semblent pas avoir cru bon de détailler le fonctionnement de leur code dans une documentation ou publication associée, rendant la lecture du code source quelque peu fastidieuse, surtout pour les novices que nous sommes à

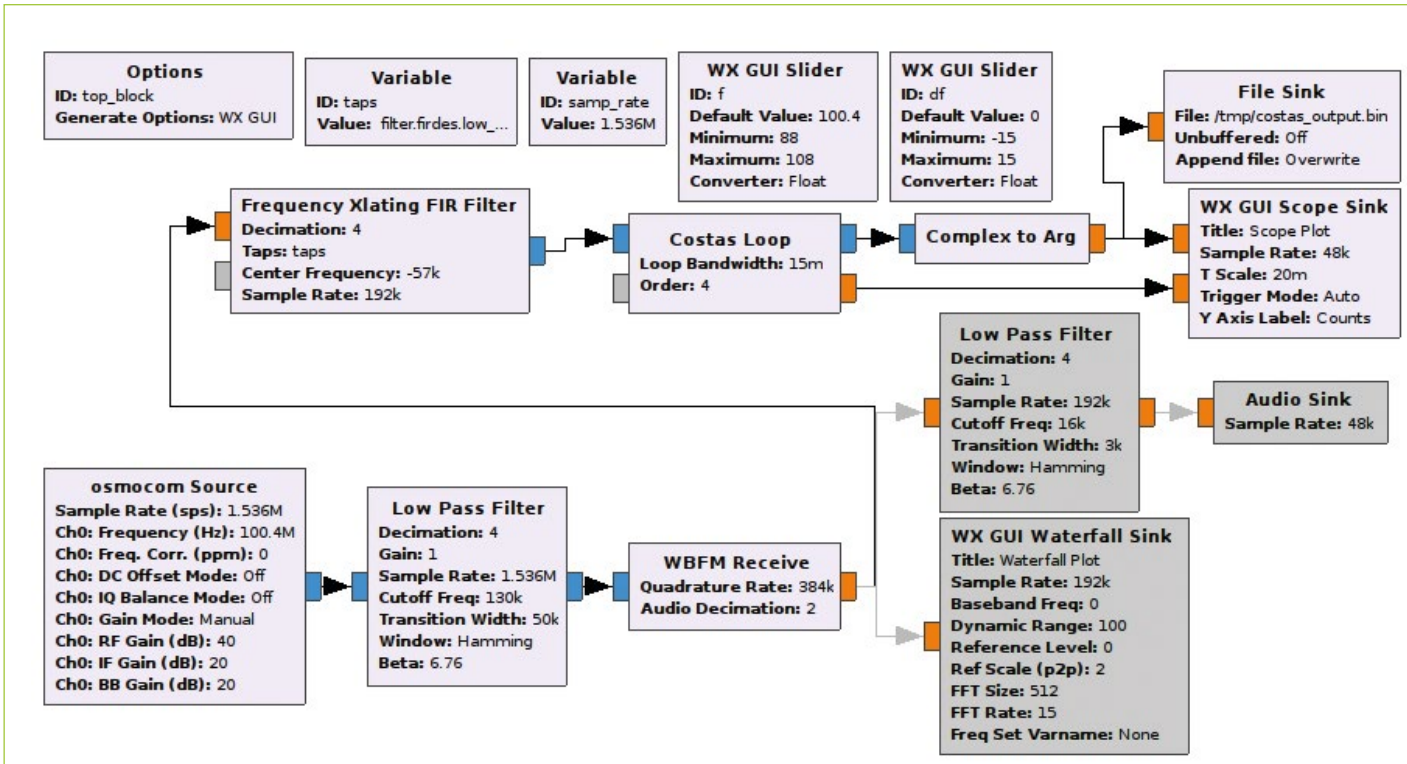
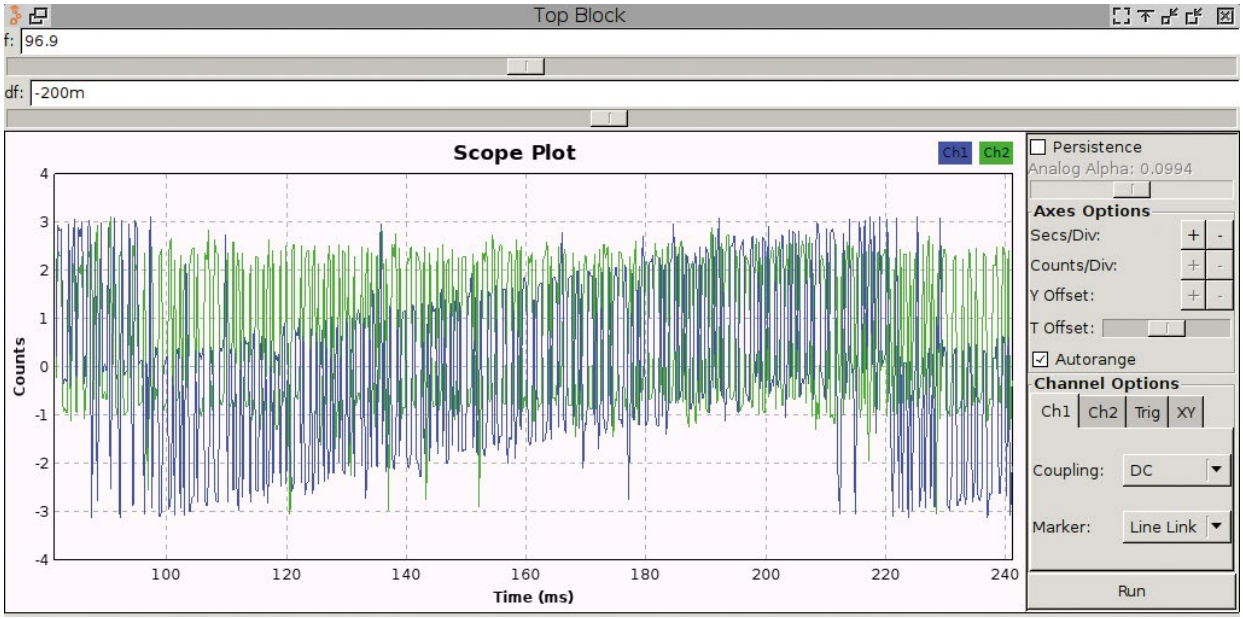


Fig. 4 : À gauche : séquence de traitements d'un signal acquis dans la bande FM commerciale. Le mode waterfall (en bas à droite) montre clairement le pilote à 19 kHz et le RDS autour de 57 kHz. À droite : en l'absence de synchronisation sur la porteuse (bleu), la phase qui encode le signal comporte encore une dérive linéaire de pente égale à l'écart entre la fréquence nominale de 57 kHz et la fréquence de l'oscillateur local numérique. L'application de la boucle de Costas (vert) élimine cet écart de fréquence : la phase représente des bits exploitables (en bas à droite en bleu).

ce stade de l'analyse du protocole. En effet, nous verrons que plusieurs couches OSI devront être parcourues avant d'obtenir un message intelligible, et cette séparation claire dans la documentation technique n'en reste pas moins confuse au premier abord par le report en annexe de la couche 2 tandis que le texte principal aborde les couches 1 et 3. Par ailleurs, tous les codes consultés sur le Web s'inspirent d'une implémentation des codes correcteurs d'erreur sous forme de registre à décalage, une solution naturelle pour l'électronicien, mais peu intuitive pour le traiteur de signaux qui exprime plus naturellement un problème sous forme matricielle. Nous proposerons donc une implémentation qui nous semble originale de la synchronisation de trames et de la correction d'erreurs que nous n'avons pas trouvée dans les documentations consultées au cours de cette étude, mais dont la concision semble favoriser la compréhension de concepts un peu complexes à appréhender au niveau de la porte logique.

Le mode de modulation est décrit de façon quelque peu confuse dans les documents techniques décrivant la norme RDS **[13]** (www.interactive-radio-system.com/docs/EN50067_RDS_Standard.pdf), en expliquant qu'une modulation d'amplitude sans porteuse **[14]** est générée par deux signaux en opposition de phase **[13, section 1.4]**. L'alternative, de considérer l'information comme portée par une modulation en phase de ±90°, change complètement notre stratégie de démodulation (voir annexe A). Alors qu'une modulation d'amplitude ne nécessite qu'une estimation grossière de la porteuse et un redressement/filtrage dans une bande suffisamment large pour inclure tout écart entre fréquence de l'émetteur et fréquence de l'oscillateur local du récepteur, une modulation de phase nécessite une stratégie pour *reconstruire une copie locale de l'oscillateur* de l'émetteur avant modulation, qui sera l'objet de la première partie de ce document.

Ayant obtenu des signaux – phase du signal – représentatifs d'une séquence de bits, nous nous efforcerons de regrouper ces bits en trame (Fig. 5). Étant donné que les bits sont transmis continuellement, nous devrons trouver une stratégie pour identifier un début de trame dans ce flux continu de bits. Finalement, ayant synchronisé notre récepteur sur le flux de bits, nous en interprèterons le contenu et verrons que des résultats cohérents sont obtenus. Nous constatons sur la figure 4 - qui consiste en un flux de traitement dans GNURadio pour démoduler une bande FM, en extraire l'information à 57 kHz de la porteuse et en extraire phase et amplitude - que



la phase ne présente pas une structure visuellement associée à une séquence de bits. Le spectre de la phase nous indique que la piste de la modulation de phase à deux états séparés de 180° (BPSK – *Binary Phase Shift Keying* **[15][16]**) est la bonne : le spectre est étalé autour de 1200 Hz par la modulation de phase, mais la raie fine à 2375 Hz confirme que tout processus non-linéaire qui a produit le carré du signal introduit un signal spectralement pur, comme on peut s'y attendre avec le BPSK.

1. TRANSPOSITION ET REPRODUCTION DE LA PORTEUSE

Nous avions déjà vu en étudiant les signaux GPS **[15]** que le mode de modulation BPSK, caractérisé par deux états de la phase 0 et 180° pour représenter deux états des bits 0 et 1 par exemple, s'exploite en produisant une copie locale de l'oscillateur de l'émetteur sans modulation. Pour ce faire, nous avions considéré diverses approches incluant une extraction de phase à partir des données complexes `I` et `Q` issues du démodulateur en exploitant une fonction insensible aux rotations de phase de 180° (`atan` de GNU/Octave qui ne tient pas compte du quadrant dans lequel se trouvent `I` et `Q` individuellement, mais ne s'intéresse qu'à `Q/I`, contrairement à `atan2` qui exploite chaque composante séparément et restitue la phase en tenant compte des rotations de 180°). Une alternative avait été de passer le signal reçu au carré (le multiplier par lui-même) afin de retirer la modulation de phase, puisque le passage au carré d'un signal harmonique génère le double de son argument, et `2 x 180° = 360 = 0 [360]` (Fig. 6 en bas à droite). Le résultat, de fréquence double de la fréquence de la sous-porteuse, produit une copie locale de la source émise avant modulation par passage dans un compteur qui divise par deux sa fréquence. Cette dernière méthode est implémentée dans le bloc de traitement dit de la boucle de Costas **[15]**, qui fournit d'une part le signal corrigé de l'erreur entre l'oscillateur émetteur et l'oscillateur local du récepteur, et d'autre part une estimation de cette erreur.

Notre stratégie de traitement consiste donc à :

1. transposer le signal à 57 kHz de la sortie du démodulateur FM pour amener l'information numérique en bande de base, par exemple avec **Xlating FIR Filter** de GNURadio, en exploitant un oscillateur local libre. Alors que nous devons habituellement expérimenter avec la fréquence de transposition de ce bloc avant de nous rappeler s'il faut indiquer + ou - la fréquence pour amener le signal en bande de base (autour de la fréquence nulle), le problème ne se pose exceptionnellement pas dans ce cas où la sortie du démodulateur FM est un signal réel, donc dont le module de la transformée de Fourier est pair. L'information qui nous intéresse

est autour de +57 kHz, mais aussi autour de -57 kHz : les deux solutions sont acceptables et fournissent le même résultat ;

2. extraire la phase du signal résultant, phase qui présente deux composantes : l'information encodée dans la phase à ±90°, et la dérive linéaire due à l'écart de fréquences des oscillateurs de l'émetteur et du récepteur **Δf** ;

3. fournir ce signal à une boucle de Costas qui estime **Δf** et la compense.

La modulation se fait à 1187,5 bits/s, et nous verrons plus loin qu'il s'agit d'un mode d'encodage différentiel qui nécessite donc au moins **1187,5 x 2 = 2375** Hz, déterminant donc la largeur du filtre du **Xlating FIR Filter** ainsi que son facteur de décimation. Nous viserons d'avoir au moins 5 points par période, soit au moins 11875 échantillons/s.

2. DU SIGNAL EN BANDE DE BASE AUX BITS

Nous supposons maintenant avoir un flux représentatif d'un signal numérique. Nous voulons extraire de la phase une séquence de bits. Dans un premier temps, nous apprenons **[13, section 1.6]** que le signal est encodé de façon différentielle (s'apparentant aussi à un codage Manchester différentiel **[16]**), confirmé par notre observation que la phase varie deux fois plus vite que le taux attendu de 1187,5 Hz. Nous allons donc seuiller la phase après en avoir retranché la valeur moyenne – en considérant qu'une phase inférieure à **0** est une valeur nulle sinon la valeur du bit est **1** (*bit slicer* dans GNURadio), et une fois la valeur saturée obtenue, nous appliquons la condition que deux phases adjacentes de phase de même valeur se traduisent par un bit à **0**

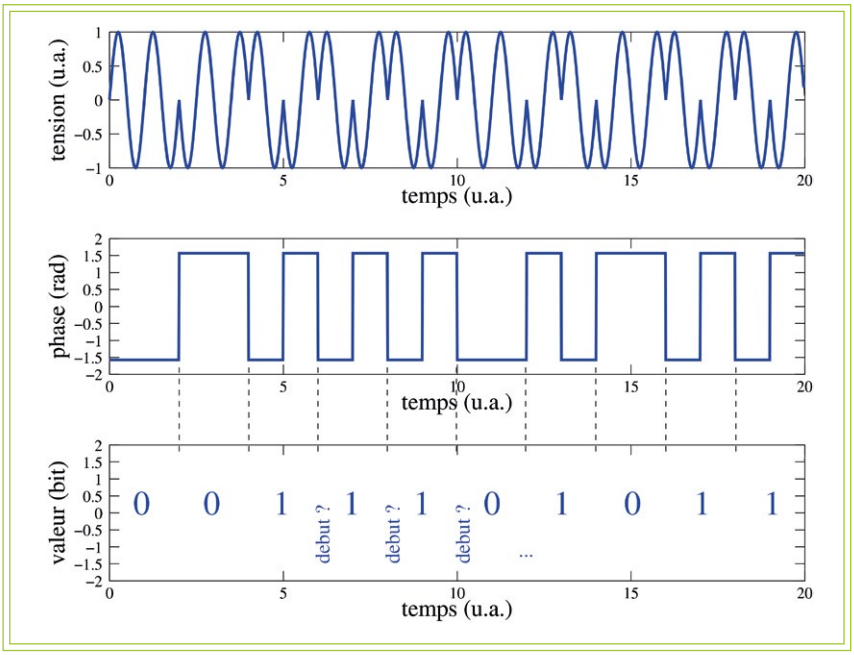


*Fig. 5 : Séquence de décodage des trames : le passage du milieu au bas est conforme au Tab. 2 de **[13]** avec une conversion respectant le codage de Manchester différentiel.*

et si une transition est observée, le bit résultant est égal à **1**. Les deux subtilités ici tiennent d'une part à se tromper de front lors de la recherche du demi-bit permettant de commencer l'analyse – dans ce cas toutes les paires adjacentes présentent une transition (cas du codage Manchester) – et d'autre part de recaler le débit de communication si le rythme des données n'est pas calé sur notre horloge locale. Nous recherchons donc la première transition (**debut**), puis avançons dans la séquence de données en recherchant au quart de la période après la transition et aux trois quarts (second état). En fonction de l'égalité ou l'inégalité de ces deux états (**s1** et **s2**), nous déduisons **so** le bit de sortie, comme OU exclusif de ces deux bits, ce qui correspond aussi à une somme modulo **2**, expression plus naturelle pour GNU/Octave il nous semble.

```
fe=24000;              % freq. d'echantillonnage -- cf sink gnuradio
bitlength=fe/1187.5  % 1 bit = 2 transitions
bitsur2=bitlength/2;
bitsur4=bitlength/4; % half transition width
r=read_complex_binary('costas_output100p4_24kHz.bin');% Virgin

p=angle(r);
p=conv(p,ones(4,1)/4);p=p(2:end-2); %filtre passe-bas de puissance
unitaire
p=p(2000:end);                    %time needed for costas to lock
p=p-mean(p);
k=find(diff(p(11:1000))>0.5);debut=k(1)+10; % indice premiere
transition
s=(p>0);s=s-mean(s);                % binary slicer
l=debut;
for k=1:length(s)/bitlength-bitlength
```

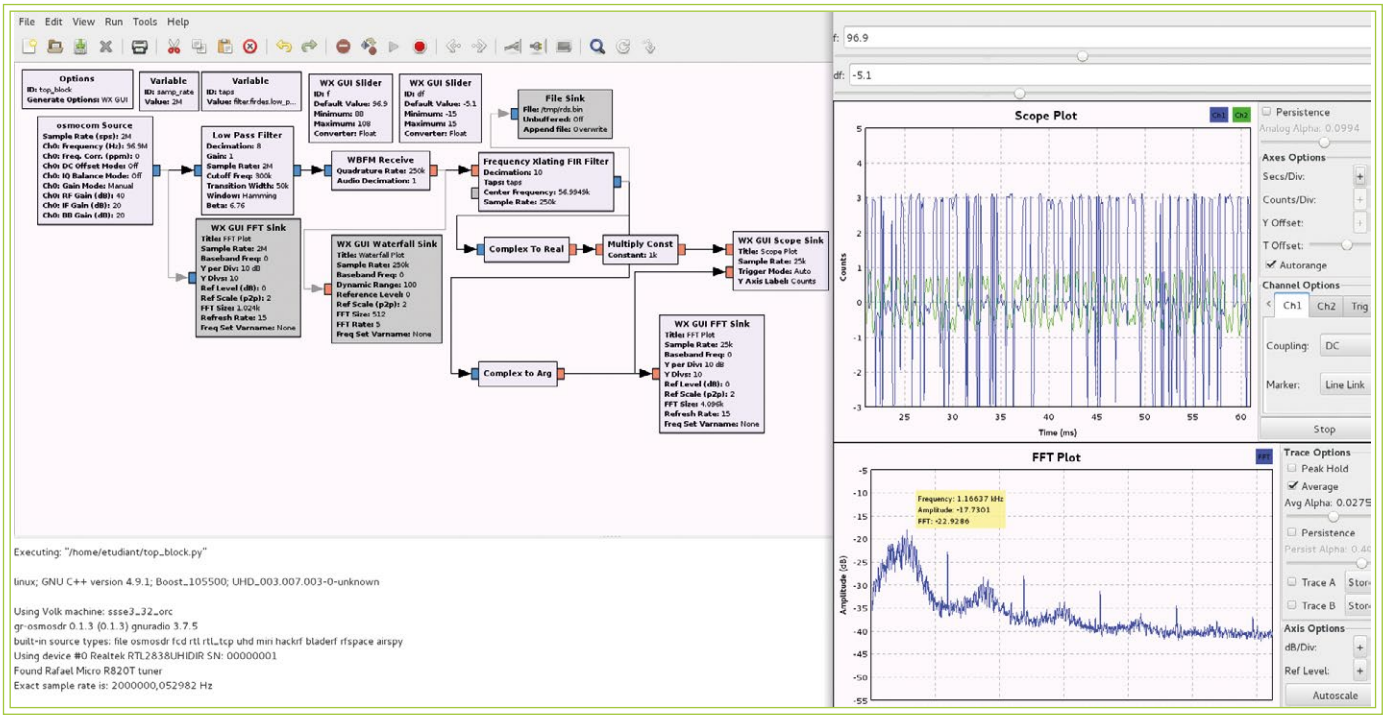


Fig. 6 : À gauche : réception FM, suivi de l'extraction de la sous-bande RDS. En l'absence de synchronisation sur la porteuse, la phase varie trop vite pour ressembler à un signal numérique.

```
    s1(k)=s(floor(l+bitsur4));          % premier etat
    s2(k)=s(floor(l+bitsur2+bitsur4)); % 2e etat 1/2 periode
plus tard
    l=l+bitlength;                      % on avance d'une periode
    transition=abs(diff(s(floor(l-2):floor(l+1))));
    [val,pos]=max(transition);
    if (val>0)                          % tentative de
synchronisation
       if (pos==3) l=l+1;end            % on est un peu en avance
       if (pos==1) l=l-1;end            % on est un peu en retard
    end
end
s1=(s1>0); s2=(s2>0);
so=mod(s1+s2,2);                        % 2 bits -> 1 etat
```

La chaîne de traitement GNURadio fournit un signal synchronisé sur la porteuse radiofréquence, mais ne traite pas le problème de la synchronisation du débit de données numérique (intervalle de temps entre deux bits). Dans l'exemple ci-dessus, une approche naïve consiste à observer si la transition d'un bit à l'autre se produit une période avant ou après le moment attendu, et si c'est le cas de décaler un peu le compteur qui s'incrémente le long de la trame (variable `l`). Une façon plus rigoureuse, mais plus complexe, est discutée en annexe B, en exploitant les blocs de traitement adéquats de GNURadio que sont le **MPSK Decoder** ou **Clock Recovery MM**, tels que présentés dans http://gnuradio.4.n7.nabble.com/Clock-Recovery-MM-documentation-td55117.html. Bien que cette méthode de synchronisation du flux numérique n'ait pas été exploitée au cours de la rédaction de cet article, induisant les études sur la capacité de correction des bits corrompus lors du décodage des trames par le code correcteur d'erreur qui vont suivre, sa mise en œuvre tardive rend le décodage extrêmement efficace tel qu'illustré en fin d'annexe B.

3. DES BITS AUX MESSAGES : SYNCHRONISATION

Nous observons une séquence continue de bits. Cependant, une trame commence et finit à certaines frontières que nous devons déterminer : nous avions vu par exemple que dans ACARS **[17]** chaque phrase commence par un préambule qui permet par ailleurs de synchroniser le récepteur avec le débit de données de l'émetteur. Dans le cas de RDS, le mode de recherche du début est décrit dans la norme **[13, annexe C]** : placer ce sujet aussi loin dans la documentation rend sa lecture quelque peu complexe puisqu'elle encourage à se lancer dans le décodage des trames **[13, section 2]** avant de s'être assuré de l'intégrité des trames. Nous apprenons entre ces deux sections que toute trame RDS est formée de 16 bits de données suivies d'un code correcteur d'erreur de 10 bits (pour une alternative, voir l'annexe C).

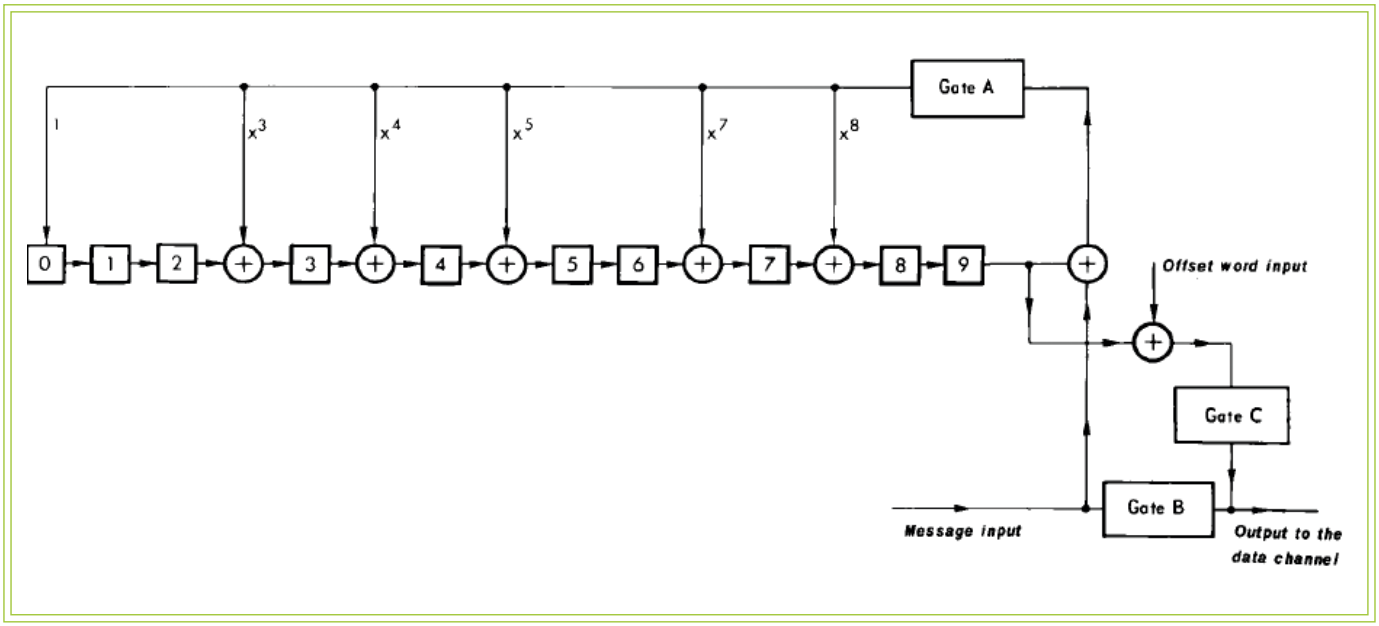


*Fig. 7 : Implémentation sous forme de registre à décalage du code correcteur d'erreur. Nous proposons l'alternative de précalculer les états possibles du code correcteur d'erreur pour l'implémenter sous forme d'opération matricielle, plus naturelle sous GNU/Octave. Chaque symbole « ⊕ » se comprend au sens binaire, comme un OU exclusif. Ce schéma est issu de **[13, p.62]**.*

L'information de base est donc un bloc de 26 bits, et 4 blocs successifs de types différents – nommés **A**, **B**, **C**, **D** – se suivent pour former une trame. Le mode de synchronisation **[18, chapitre 12]** consiste donc à :

1. prendre 26 bits adjacents dans la séquence acquise ;
2. calculer le code correcteur d'erreur (voir ci-dessous) pour ces 26 bits en supposant qu'il s'agit d'un bloc **A** ;
3. si les 10 derniers bits de la séquence considérée correspondent au code correcteur d'erreur, il est envisageable que nous ayons trouvé la synchronisation, que nous validons en calculant le code correcteur du bloc suivant (**B**, 26 bits suivants) puis **D** (26 bits séparés de 78 bits de l'indice courant d'analyse). Si ces trois conditions sont vérifiées, nous avons très probablement trouvé un cas de synchronisation et donc le premier bit de la trame. Le bloc **C** a été sauté, car deux cas peuvent se présenter – **C** et **C'** selon la nature du message – avec des syndromes différents, multipliant les cas à traiter ;
4. si le calcul du code correcteur sur les 16 premiers bits ne donne pas la séquence des 10 derniers bits, il n'y a pas synchronisation : nous avançons d'un bit dans la séquence et nous recommençons la procédure.

Cette séquence doit finir par converger vers une condition où tous les 10 bits de fin de chaque bloc correspondent au code correcteur d'erreur des 16 bits qui précèdent. Si ce n'est pas le cas, le décodage des bits (étape précédente) a échoué, et il faut reconsidérer la conversion des données brutes de phase vers les données numérisées. Ce succès du décodage est la source de la synchronisation en temps proposée par **[2]**. Une fois le début du bloc identifié, le contenu de chaque bloc dépend de la nature du message : alors que le bloc A contient toujours un identifiant de la station (code **PI**, annexe C), le contenu des autres blocs change pour chaque type d'information transmise, tel que documenté dans **[13, section 3]**.

Un point clé de cette discussion tient au calcul du code correcteur d'erreur. Toutes les implémentations que nous avons trouvées exploitent un registre à décalage avec rétroaction (*Linear Feedback Shift Register – LFSR*), une représentation naturelle pour un FPGA ou un microcontrôleur programmé en assembleur **[19]**, mais peu approprié à GNU/Octave qui exprime les problèmes sous forme matricielle (voir note). Afin de ne pas risquer d'être qualifié de plagiat – tous les codes libres consultés sur Internet et implémentés pour calculer le code correcteur sont du même acabit que https://github.com/bastibl/gr-rds/blob/master/lib/decoder_impl.cc#L69-L80 (Fig. 7) – nous proposons de calculer le code correcteur par son expression matricielle telle que proposée dans **[13, section B.2.1]** : une matrice **H** de **26 x 10** fournit la relation entre les données et chacun des bits du code correcteur d'erreurs.

Abonnez-vous !

C'est simple... c'est possible sur :

http://www.ed-diamond.com

... OU SÉLECTIONNEZ VOTRE OFFRE DANS LA GRILLE AU VERSO ET RENVOYEZ CE DOCUMENT COMPLET À L'ADRESSE CI-DESSOUS !

Voici mes coordonnées postales :	
Société :	
Nom :	
Prénom :	
Adresse :	
Code Postal :	
Ville :	
Pays :	
Téléphone :	
E-mail :	

❑ Je souhaite recevoir les offres promotionnelles et newsletters des Éditions Diamond.

❑ Je souhaite recevoir les offres promotionnelles des partenaires des Éditions Diamond.

En envoyant ce bon de commande, je reconnais avoir pris connaissance des conditions générales de vente des Éditions Diamond à l'adresse internet suivante : http://**boutique.ed-diamond.com/content/3-conditions-generales-de-ventes** et reconnais que ces conditions de vente me sont opposables.

LINUX MAGAZINE / FRANCE

Les Éditions Diamond
Service des Abonnements
10, Place de la Cathédrale
68000 Colmar – France
Tél. : + 33 (0) 3 67 10 00 20
Fax : + 33 (0) 3 67 10 00 21

Vos remarques :

.....

.....

Bon d'abonnement

CHOISISSEZ VOTRE OFFRE !

Prix TTC en Euros / France Métropolitaine*

Offre	SUPPORT / ABONNEMENT	PAPIER Réf	PAPIER Tarif TTC	PAPIER + BASE DOCUMENTAIRE (1 connexion BD) Réf	PAPIER + BASE DOCUMENTAIRE (1 connexion BD) Tarif TTC
LM	11n° GNU/Linux Magazine France	LM1	69,-	LM13	245,-
LM+	11n° GNU/Linux Magazine France + 6n° Hors-Série	LM+1	125,-	LM+13	299,-
LES COUPLAGES AVEC NOS AUTRES MAGAZINES					
A	11n° GNU/Linux Magazine France + 6n° Linux Pratique	A1	105,-	A13	399,-
A+	11n° GNU/Linux Magazine France + 6n° Hors-Série + 6n° Linux Pratique + 3n° Hors-Série	A+1	189,-	A+13	489,-
B	11n° GNU/Linux Magazine France + 6n° MISC	B1	109,-	B13	499,-
B+	11n° GNU/Linux Magazine France + 6n° Hors-Série + 6n° MISC + 2n° Hors-Série	B+1	185,-	B+13	629,-
C	11n° GNU/Linux Magazine France + 6n° Linux Pratique + 6n° MISC	C1	149,-	C13	669,-
C+	11n° GNU/Linux Magazine France + 6n° Hors-Série + 6n° Linux Pratique + 3n° Hors-Série + 6n° MISC + 2n° Hors-Série	C+1	249,-	C+13	769,-
J	11n° GNU/Linux Magazine France + 6n° Hackable	J1	105,-	J13	399,-
J+	11n° GNU/Linux Magazine France + 6n° Hors-Série + 6n° Hackable	J+1	159,-	J+13	459,-
LA TOTALE DIAMOND !					
L	11n° GLMF + 6n° HK* + 6n° LP + 6n° MISC	L1	189,-	L13	839,-
L+	11n° GLMF + 6n° HS + 6n° HK* + 6n° LP + 3n° HS + 6n° MISC + 2n° HS	L+1	289,-	L+13	939,-

Les abréviations des offres sont les suivantes : LM = GNU/Linux Magazine France | HS = Hors-Série | LP = Linux Pratique | HK = Hackable

Particuliers = CONNECTEZ-VOUS SUR :

http://www.ed-diamond.com

pour consulter toutes les offres !

*Les tarifs hors France Métropolitaine, Europe, Asie, etc. sont disponibles en ligne !

Professionnels = CONNECTEZ-VOUS SUR :

http://proboutique.ed-diamond.com

pour consulter toutes les offres !

*Les tarifs hors France Métropolitaine, Europe, Asie, etc. sont disponibles en ligne !

NOTE

Le passage de l'expression polynomiale du code correcteur d'erreur à l'expression matricielle ne saute pas forcément aux yeux au premier abord. Le principe de l'algèbre linéaire (expression matricielle) tient à décomposer le problème sur chaque élément de la base et de déduire la solution comme combinaison linéaire de chaque solution obtenue sur la décomposition du problème sur chaque élément de la base. Dans notre cas, la base du problème est un bit dans le message à 1 à la nième position et les autres bits à 0. Nous appliquons donc le calcul du code correcteur d'erreur à chaque vecteur `[0 ... 0 1 0 ... 0]` en déplaçant le 1 dans le vecteur, ce qui en expression polynomiale s'écrit x^n. Le code correcteur d'erreur s'obtient comme reste de la division polynomiale [20] de x^n, $n \in [0..25]$ (les 26 positions possibles du bit dans le message émis) avec le polynôme représentant le code BCH [21, p. 251] $x^{10} + x^8 + x^7 + x^5 + x^4 + x^3 + 1$ qui s'écrit sous GNU/Octave `[1 0 1 1 0 1 1 1 0 0 1]` (puissance la plus forte à gauche). Pour rappel, une division polynomiale s'effectue comme une division classique, avec le dénominateur multiplié par la puissance adéquate pour éliminer le terme de plus haute puissance du numérateur, et le reste calculé par soustraction de ces deux termes. La procédure se poursuit sur le reste jusqu'à atteindre une puissance inférieure à celle du dénominateur. Par exemple :

$$\underbrace{x^4+x^2+x+1}_{\text{numérateur}} = \underbrace{(x^2+1)}_{\text{dénominateur}} \times \underbrace{x^2}_{\text{quotient}} + \underbrace{(x+1)}_{\text{reste}}$$

Le reste de la division polynomiale s'écrit sous GNU/Octave (et Matlab) comme `[b, r] = deconv(N,D);` avec `N` et `D` les vecteurs représentant respectivement les polynômes du numérateur et dénominateur, et `r` le reste de la division qui nous intéresse. Dans l'exemple précédent, `[d, r] = deconv([1 0 1 1 1],[1 0 1])` donne `d = 1 0 0` et `r = 0 0 0 1 1`. Ainsi la matrice `G` de [13] s'obtient par le script GNU/Octave suivant :

```
vecteur=[1 zeros(1,10)];          % premier element de la
base : 1*x^{10}
polynome=[1 0 1 1 0 1 1 1 0 0 1]; % x^10+0+x^8+x^7+0+x^5+x^
4+x^3+0+0+x
for k=1:16                        % position du bit a 1
  [a,b]=deconv(vecteur,polynome); % division polynomiale
  mod(abs(b(end-9:end)),2)        % modulo x^10
  vecteur=[vecteur 0];            % element suivant de la
base : ajout d'un 0
end
```

Comme la matrice identité à gauche de `G` place le bit de poids faible en bas, ce script calcule les lignes de la droite de `G` de bas en haut.

```
% p.75 US, 64 EU
sA2=[1 1 1 1 0 1 1 0 0 0]; % A=  [0 0 1 1 1 1 1 1 0 0];
sB2=[1 1 1 1 0 1 0 1 0 0]; % B=  [0 1 1 0 0 1 1 0 0 0];
sC2=[1 0 0 1 0 1 1 1 0 0]; % C=  [0 1 0 1 1 0 1 0 0 0];
Cp= [1 1 0 1 0 1 0 0 0 0];
% sCp=[1 0 1 1 1 0 1 1 0 0];
sCp=[1 1 1 1 0 0 1 1 0 0]; % an495
D=  [0 1 1 0 1 1 0 1 0 0];
sD2=[1 0 0 1 0 1 1 0 0 0];
% incoherent avec an495
% sD2=[0 1 0 1 0 1 1 0 0 0]; % an495

% p.74
H=[
 1 0 0 0 0 0 0 0 0 0 ; % cette
 0 1 0 0 0 0 0 0 0 0 ; % sous-
 0 0 1 0 0 0 0 0 0 0 ; % matrice
 0 0 0 1 0 0 0 0 0 0 ; % identite
 0 0 0 0 1 0 0 0 0 0 ; % sera
 0 0 0 0 0 1 0 0 0 0 ; % remplacee
 0 0 0 0 0 0 1 0 0 0 ; % par
 0 0 0 0 0 0 0 1 0 0 ; % eye()
 0 0 0 0 0 0 0 0 1 0 ; % dans la
 0 0 0 0 0 0 0 0 0 1 ; % suite
 1 0 1 1 0 1 1 1 0 0 ;
 0 1 0 1 1 0 1 1 1 0 ;
 0 0 1 0 1 1 0 1 1 1 ;
 1 0 1 0 0 0 0 1 1 1 ;
 1 1 1 0 0 1 1 1 1 1 ;
 1 1 0 0 0 1 0 0 1 1 ;
 1 1 0 1 0 1 0 1 0 1 ;
 1 1 0 1 1 1 0 1 1 0 ;
 0 1 1 0 1 1 1 0 1 1 ;
 1 0 0 0 0 0 0 0 0 1 ;
 1 1 1 1 0 1 1 1 0 0 ;
 0 1 1 1 1 0 1 1 1 0 ;
 0 0 1 1 1 1 0 1 1 1 ;
 1 0 1 0 1 0 0 1 1 1 ;
 1 1 1 0 0 0 1 1 1 1 ;
 1 1 0 0 0 1 1 0 1 1 ;
];

texte="";
station="";
debut=0
so=(1-so);
for k=1:length(so)-104
  data1=(so(k:k+25));             % A
  data2=(so(k+26*1:k+25+26*1)); % B
  data3=(so(k+26*2:k+25+26*2)); % C(')
  data4=(so(k+26*3:k+25+26*3)); % D
  HI1=mod(data1*H,2);
  HI2=mod(data2*H,2);
  HI3=mod(data3*H,2);
  HI4=mod(data4*H,2);
  pa=findstr((sA2),(HI1));
  pb=findstr((sB2),(HI2));
  pc=findstr((sC2),(HI3));
  pcp=findstr((sCp),(HI3));
  pd=findstr((sD2),(HI4));
  if (!isempty(pa) && !isempty(pb) &&
!isempty(pd))
     printf("synchronisation\n");
  end
end
```

Cet exemple, un peu volumineux compte tenu de la matrice de décodage `H`, effectue les opérations suivantes :

1. pour chaque séquence de 104 bits consécutifs, nous découpons 4 blocs adjacents de 26 bits chacun (ll.47–50). Chaque bloc est supposé être formé de 16 bits de données suivis de 10 bits de code de correction d'erreur auquel ont été ajoutés les bits d'identification du bloc ;

2. nous calculons le syndrome de 10 bits de chaque bloc de 26 bits par application du produit matriciel avec **H** (ll. 51–54) ;

3. nous recherchons si le syndrome ainsi calculé correspond au syndrome du bloc correspondant (ll. 55–59). Si cette condition est vérifiée, le bloc est validé, et nous considérons être synchronisés si cette condition est vérifiée pour les 4 blocs consécutifs ;

4. si un bloc ne vérifie pas un syndrome du code correcteur d'erreur égal à la valeur prévue, nous supposons ne pas être synchronisés : nous avançons d'un cran dans la séquence de bits acquise pour redéfinir une nouvelle séquence de 104 bits, et reprenons le calcul.

NOTE

Le passage de G à H tel que fourni dans [13] n'est lui non plus pas trivial. Alors que le passage de G à une forme de H telle que l'intégrité d'un message x reçu se vérifie par H · x = 0 tel que proposé dans [21, p. 244] ou [22, p. 70] ne nécessite qu'une transposition de la partie non-identité de G, l'expression de la matrice de contrôle d'erreur fournie dans [13, p.63] ne respecte pas cette expression, mais place l'identité à gauche de la matrice de décodage. Le passage d'une expression à l'autre est illustré sur http://www.di-mgt.com.au/cgi-bin/matrix_stdform.cgi qui exploite l'algorithme de [22, p. 52, 71] pour passer de l'une (issue de la matrice d'encodage) à l'autre (dite standard) par combinaison linéaire et permutation des lignes ou colonnes. D'après le site web sus-cité, nous passons de l'expression de H fournie dans [13, p. 63]

```
Input:
1 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1 1 1 0 1 1 0 0 1 1 1
0 1 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1 1 1 0 1 1 0 0 1 1
0 0 1 0 0 0 0 0 0 0 1 0 1 1 1 0 0 0 1 0 1 1 1 1 1 0
0 0 0 1 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 1 1 1 0 0 0
0 0 0 0 1 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 1 1 1 0 0
0 0 0 0 0 1 0 0 0 0 1 0 1 0 1 1 1 1 1 0 1 0 1 0 0 1
0 0 0 0 0 0 1 0 0 0 1 1 0 0 1 0 0 0 1 0 1 1 0 0 1 1
0 0 0 0 0 0 0 1 0 0 1 1 1 1 1 1 0 1 1 0 0 1 1 1 1 0
0 0 0 0 0 0 0 0 1 0 0 1 1 1 1 1 1 0 1 1 0 0 1 1 1 1
0 0 0 0 0 0 0 0 0 1 0 0 1 1 1 1 1 1 0 1 1 0 0 1 1 1
```

à celle issue de [21, p. 244] où le morceau à droite de l'identité dans G est transposé (la colonne de gauche ci-dessous est effectivement la fin de la première ligne de G par exemple)

```
0 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0
0 0 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0
0 1 1 0 0 0 1 1 0 0 1 1 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0
1 1 0 0 1 1 0 1 1 0 1 0 0 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0
1 1 1 0 0 1 1 0 1 1 0 1 0 0 1 1 0 0 0 0 1 0 0 0 0 0 0 0
1 0 0 0 1 1 1 1 0 1 0 1 0 1 1 1 0 0 0 0 0 1 0 0 0 0 0 0
0 0 1 1 1 0 1 1 1 0 0 1 0 1 0 1 0 0 0 0 0 0 1 0 0 0 0 0
1 1 1 0 0 0 0 1 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0
1 1 1 1 0 0 0 0 1 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 1 0 0 0
1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 1 0 0
1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 1 0
1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 1
```

par cette série de transformations (noter le passage de la sous-matrice identité de gauche à droite de la matrice H).

4. INTERPRÉTATION DES MESSAGES

Le plus difficile est fait : nous avons une séquence de bits, dont nous sommes garantis de l'intégrité par calcul du code correcteur d'erreur, il ne reste plus qu'à interpréter les divers messages, en remplaçant le contenu des lignes 60–62 de l'exemple précédent. Toutes les transactions se font avec le bit de poids fort (MSB) transmis en premier. Nous nous sommes limités à quelques exemples simples et représentatifs pour garder le code court : nom de la station, horloge (jour/heure/minute) ou texte libre. Comme nous voulions pouvoir identifier la station en cours de réception, le premier type de bloc décodé est **0A** ou **0B** tel que défini par les 4 premiers bits du bloc **B** : si les 4 premiers bits du bloc **B** sont nuls, alors le bloc **D** contient des caractères ASCII qui contiennent le nom de la station FM émettrice. Le bloc **B** est le deuxième bloc dont les données sont stockées dans la variable **data2** dont nous interprétons le contenu comme deux caractères ASCII, MSB en premier, consécutifs :

```
puissances=2.^[7:-1:0]'; % 128 64 32 6 8
4 2 1
  if (!isempty(pa) && !isempty(pb) &&
!isempty(pd))
     if ((data2(1:4)*puissances(end-3:
end))==2)  % texte RDS
       texte=[texte
char(data3(1:8)*puissances)] % 2chars en C
       texte=[texte
char(data3(9:16)*puissances)]
       texte=[texte
char(data4(1:8)*puissances)] % 2chars en D
       texte=[texte
char(data4(9:16)*puissances)]
     end
     if (sum(data2(1:4))==0)
% nom station
       station=[station
char(data4(1:8)*puissances)];
```

```
      station=[station char(data4(9:16)*puissances)]
    end
    if ((data2(1:5)*puissances(end-4:end))==2)  % 1A
       day=data4(1:5)*puissances(end-4:end);
       heure=data4(6:10)*puissances(end-4:end);
       minute=data4(11:16)*puissances(end-5:end);
       printf("temps1A %d %d %d\n",day,heure,minute);
    end
    if ((data2(1:5)*puissances(end-4:end))==8)  % 4A
       day=data2(15:16)*2.^[16:-1:15]'+data3(1:15)*2.^[14:-1:0]';
       heure=data3(16)*2^4+data4(1:4)*puissances(end-3:end);
% UTC
       minute=data4(5:10)*puissances(end-5:end);
       offset=data4(12:16)*puissances(end-4:end); % *30 min->
local
       printf("temps4A %d %d %d %d\n",day,heure,minute,offset);
    end
  else % printf(".");
  end
end
printf("\n");
```

On notera la petite subtilité dans la conversion du tableau de bits en nombre par le produit scalaire (et non terme à terme) de data et puissance, ce dernier contenant les puissances de **2** de **128** à **1** (bit de poids le plus fort en premier).

Le résultat de ce traitement, pour des fréquences reçues à Besançon, est de la forme

```
station = IRN  VIRGIGIN GIGIGIN GIN GIIRGIIRGIN  VGI VIR
```

pour 100,4 MHz,

```
texte =  ES (FEAT. GUCCI MANE)      MOUVAE SREMMURD - BLACK
BEAT(FEAT. GECCI MANE)      MOUV',
RAE SREMMURD - BLACK BEAT(FEAT. G]CCI MANE)          MOUVAE
SREMMERD - BLACK BEATLES (FEAT. G MAN    MOUV'( RAE
station = LEOUOU MLEV'V' MOULE MOUV'OUV'LE MV'LEOUV'LE
MOUV' MOUV'LE MOUV'LEV'LE MOUV'LEOUV'LE MOULE MOULEV' MOUV'
MOUV'LE MOUV'LE MV'LE MOUV'LE MOUV'LE MV'LE
MOUV'LEOUV'LEOULE MV'LE MOUV
```

pour 93,5 MHz, et

```
station = .9.9P BI.996.9BIP BI96BIP .9BI96.9P
96.9BIBI.9BI.996BIP .9BI.9BIP
```

pour 96,9 MHz, qui laisse penser que nous avons enregistré des signaux de Virgin, Le Mouv' et BIP. Finalement, Rire & Chanson sur 91.0 se traduit par :

```
station = ELLI& E &  RIR RE &  RIRE &  RE &  RIR&  R SLI
SELLIG  SELLIG  SELLIG  SELLI RIRE &  RIRE &  RIRE &
RIRE &  RIRE &  RIRE  RIRE  R& IRE & IR R
```

Sellig semble être un humoriste donc son nom dans le titre de la chaîne n'est pas improbable. Plus intéressant, France Info nous donne :

```
texte = N MOCH - SUR(LA CART FRAFRANCE
INFO  14  : JULIEN MOCH - SUR LA CARTE
DE FRAFRANCE INFO - LE| 17 : JULIEN
MOCH - SUR LA CARTE DE FRANCE FRANKE
INFO - LE 14 | 17 : JULIEC@ - SUR LA
CARTE DE FRANCE - LE 14 | 17 : JN MOCH -
SUR LA Ci£RTE DE FRACE INFO - LE 14 |
17 : JULIE
station =       FO    INFO    FO
INFOININININFO    INFO    FO  INFO
FO    INFO  INFO    FO    INFO
INFO    IN    INFO    INFO    INFO
INFO  INFO    INFO      INFO    IN
INFO    INFO    INFO    FO    INFO
FO  INFO    INFO    INFO  IN  INFO
INFO    INFO    FO
```

qui diffuse, en plus du titre de la chaîne, un texte libre annonçant l'émission en cours.

Sans être parfaites, ces trames démontrent que le concept est convenablement implémenté, en accord avec les résultats fournis par gr_rds (Fig. 8, page suivante).

5. CORRECTION D'ERREUR

Nous avons décodé les messages issus de RDS après nous être synchronisés pour aligner les phrases sur le flux continu de bits, que pouvons-nous vouloir de plus ? Le code correcteur d'erreur de 10 bits ajouté en fin de chaque message de 16 bits n'a pour le moment été exploité que pour la synchronisation et garantir l'intégrité des transactions. Le signal RDS est bruité, et un certain nombre de trames sont éliminées de l'étude parce que leur code correcteur ne correspond pas aux attentes. Pouvons-nous améliorer le rendement de la réception en tentant de corriger les erreurs grâce à la redondance introduite par le code correcteur d'erreur **[23]**, démarche qui a contribué à la croissance du débit de communication dans les sondes spatiales (Fig. 9) **[24]** ? Cela nous ramène en marche arrière par rapport au paragraphe précédent, à la couche 2 de la

hiérarchie OSI, mais donne l'opportunité de se confronter à une étude empirique de l'implémentation des codes correcteurs d'erreur.

Le code implémenté permet non seulement d'identifier une corruption du message, mais en plus d'identifier quel bit a été modifié : un code de Hamming **[15, chapitre 8]** est capable d'une telle correction pour une unique erreur. Un code BCH **[21, page 252][22, chapitre 11]** est une extension qui permet de corriger plus d'un bit : ici, le code proposé permettra d'aller jusqu'à la correction de deux bits corrompus lors de la transmission (voir figure 9).

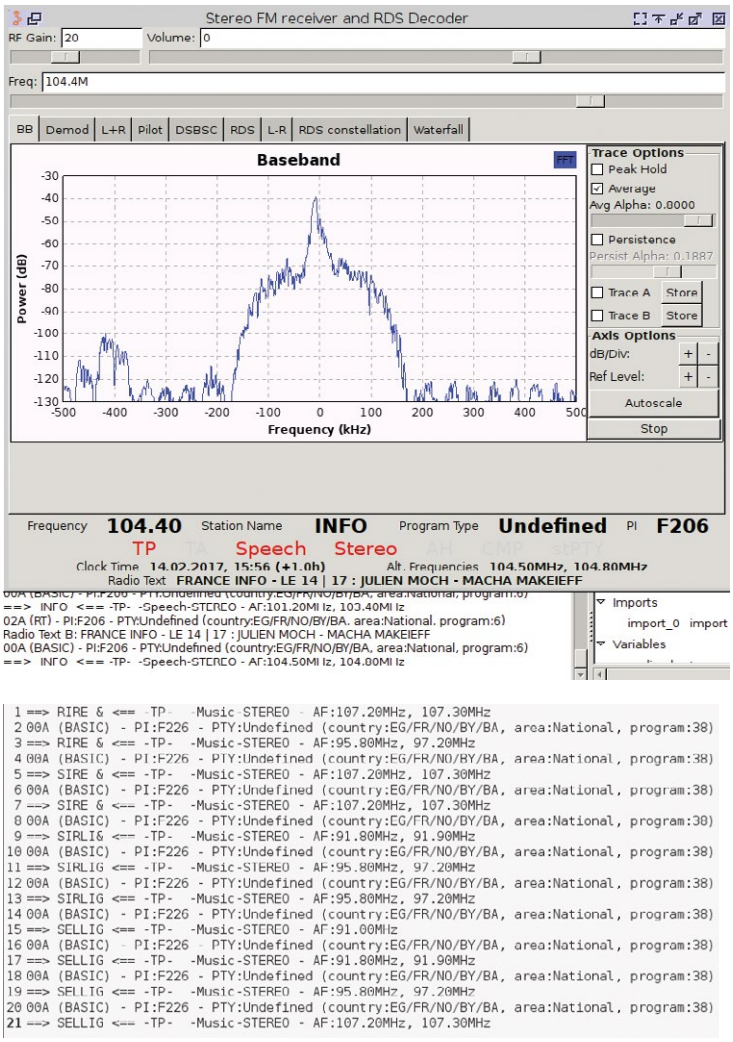


Fig. 8 : En haut, gr_rds en action sur France Info : la présentation est à peine plus attractive que l'exécution d'un script dans GNU/Octave et les données affichées sont sensiblement les mêmes. Nous notons à l'exécution que le nom de la station et le texte libre s'accumulent petit à petit, en accord avec la difficulté que nous observons d'obtenir une trame complète valide. En bas : la sortie de gr_rds sur Rire & Chanson, confirmant l'inclusion de l'auteur de l'émission en cours dans le titre de la station.

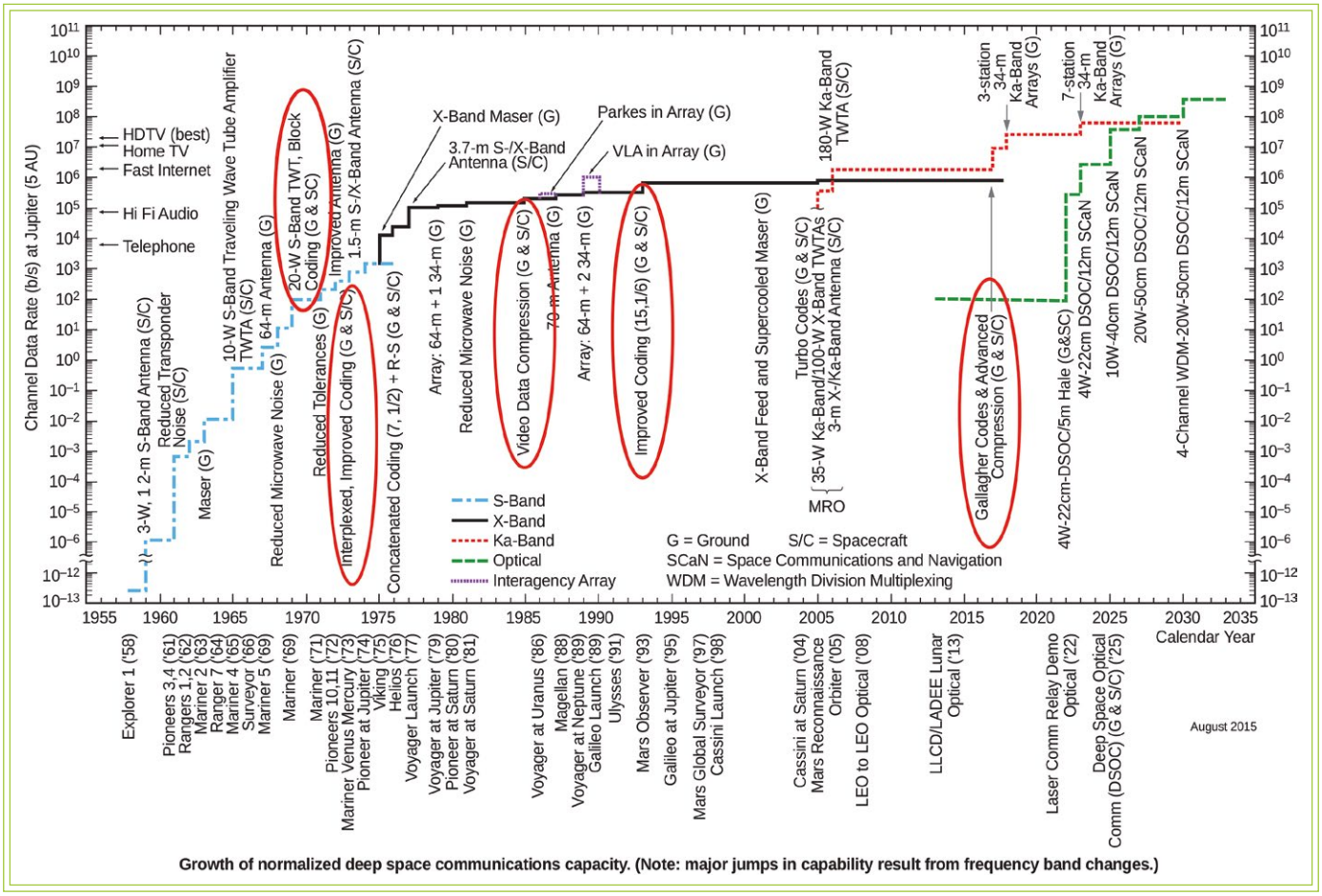
5.1 Une erreur

L'annexe C de **[13]** nous explique que le code correcteur d'erreur est implémenté, lors de l'émission, comme une combinaison linéaire (somme modulo 2, ou XOR) des bits à émettre.

Une transformation linéaire s'implémente soit de façon logique par un registre à décalage avec des points de ponction pour alimenter les portes XOR (Fig. 7), mais comme nous prototypons sous GNU/Octave, nous continuons à exploiter l'approche matricielle **[21, p. 244]** : une matrice de **16 x 10** calcule les 10 bits de sortie compte tenu des 16 bits d'entrée. Nous avons vu que nous ajoutons un identifiant de bloc pour la synchronisation à ce code de correction d'erreur : le message transmis comprend donc les 16 bits de données suivis des 10 bits de correction sommés au code de bloc. En terme GNU/Octave, cela s'écrit :

```
A=  [0 0 1 1 1 1 1 1 0 0];
% A block emission
data=[0 1 0 0 1 0 1 0 0 1 0 0
1 1 0 1]; % JM
G=[0 0 0 1 1 1 0 1 1 1;
% 16x10: message emis
   1 0 1 1 1 0 0 1 1 1;
   1 1 1 0 1 0 1 1 1 1;
   1 1 0 0 0 0 1 0 1 1;
   1 1 0 1 0 1 1 0 0 1;
   1 1 0 1 1 1 0 0 0 0;
   0 1 1 0 1 1 1 0 0 0;
   0 0 1 1 0 1 1 1 0 0;
   0 0 0 1 1 0 1 1 1 0;
   0 0 0 0 1 1 0 1 1 1;
   1 0 1 1 0 0 0 1 1 1;
   1 1 1 0 1 1 1 1 1 1;
   1 1 0 0 0 0 0 0 1 1;
   1 1 0 1 0 1 1 1 0 1;
   1 1 0 1 1 1 0 0 1 0;
   0 1 1 0 1 1 1 0 0 1];
mI=mod(data*G,2)
mI=mod(mI+A,2);  % ajout de A
envoi=[data mI]
```

Si tout se passe bien, ce message **0 1 0 0 1 0 1 0 0 1 0 0 1 1 0 1 0 1 0 0 1 0 1 0 1 0** qui contient encore les deux caractères ASCII **JM** est transmis, et reçu sans corruption de bit, donc le décodage que nous avons vu s'obtient côté récepteur par :



Growth of normalized deep space communications capacity. (Note: major jumps in capability result from frequency band changes.)

Fig. 9 : Évolution de la bande passante de communication des sondes spatiales.

```
% p.75 US, 64 EU
sA2=[1 1 1 1 0 1 1 0 0 0];
% syndrome de reception
% p.74
H=[
 eye(10); % matrice identite
 1 0 1 1 0 1 1 1 0 0 ;
 0 1 0 1 1 0 1 1 1 0 ;
 0 0 1 0 1 1 0 1 1 1 ;
 1 0 1 0 0 0 0 1 1 1 ;
 1 1 1 0 0 1 1 1 1 1 ;
 1 1 0 0 0 1 0 0 1 1 ;
 1 1 0 1 0 1 0 1 0 1 ;
 1 1 0 1 1 1 0 1 1 0 ;
 0 1 1 0 1 1 1 0 1 1 ;
 1 0 0 0 0 0 0 0 0 1 ;
 1 1 1 1 0 1 1 1 0 0 ;
 0 1 1 1 1 0 1 1 1 0 ;
 0 0 1 1 1 1 0 1 1 1 ;
 1 0 1 0 1 0 0 1 1 1 ;
 1 1 1 0 0 0 1 1 1 1 ;
 1 1 0 0 0 1 1 0 1 1 ;
];

mIr=abs(mod(envoi*H,2)-sA2)
```

et le message reçu **mIr** doit donner **0** après soustraction du syndrome correspondant au bloc A, nommé ici sA2. Cette petite expérience numérique valide le bon fonctionnement du décodage des messages.

Supposons maintenant qu'entre l'émission et la réception, une erreur survienne :

```
N=3;envoi(N)=1-envoi(N); % introduction d'erreurs
```

Pouvons-nous faire mieux que rejeter le paquet ? En affichant le résultat du calcul du syndrome, nous constatons qu'en insérant une erreur sur le troisième bit, le syndrome nous indique :

```
mIr = 0   0   1   0   0   0   0   0   0   0
```

qui dit bien que le troisième bit est corrompu. Cela correspond au fait que les 10 premières lignes de H forment une matrice identité. De façon générale, une erreur sur le **N**ième bit se détecte par un syndrome égal à la ligne **N** de **H**. En effet, une erreur peut survenir sur n'importe quel bit des 26 bits du message : si une erreur survient sur le bit 25 :

```
N=25;envoi(N)=1-envoi(N); % introduction d'erreurs
```

nous obtenons :

```
mIr = 1  1  1  0  0  0  1  1  1  1
```

qui est bien l'avant-dernière ligne de **H**. La recherche du bit corrompu se généralise donc par

```
[num,ligne]=ismember(mIr,H,'rows')
% si non nul, c'est le motif de la ligne de la matrice ou' se trouve l'erreur
```

avec **num** nous indiquant si le syndrome a été trouvé comme ligne de **H**, et si c'est le cas, **ligne** nous indique quel bit a été corrompu. Nous améliorons donc notre capacité de décodage de RDS, avec par exemple 24 caractères en plus des 680 déjà acquis sur l'identification de Le Mouv' (+4%), 9 caractères en plus des 79 déjà acquis pour Virgin (+11%) et 10 en plus des 60 pour BIP (+17%).

Quelques expérimentations avec les deux expressions de la matrice de décodage **H** constatées auparavant nous convainquent qu'elles fonctionnent de la même façon, puisque les propriétés du code sont conservées par permutation des lignes et colonnes. Ainsi, nous observons par

```
A=  [0 0 1 1 1 1 1 1 0 0]; % A block emission
sA2=[1 1 1 1 0 1 1 0 0 0];
% syndrome de reception

data=[0 1 0 0 1 0 1 0 0 1 0 0 1 1 0 1]; % JM
G=[0 1 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 0 ;
   0 0 1 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 ;
   0 1 1 0 0 0 1 1 0 0 1 1 0 0 0 1 ;
   1 1 0 0 1 1 0 1 1 0 1 0 0 1 1 0 ;
   1 1 1 0 0 1 1 0 1 1 0 1 0 0 1 1 ;
   1 0 0 0 1 1 1 1 0 1 0 1 0 1 1 1 ;
   0 0 1 1 1 0 1 1 1 0 0 1 0 1 0 1 ;
   1 1 1 0 0 0 0 1 1 1 1 1 0 1 0 0 ;
   1 1 1 1 0 0 0 0 1 1 1 1 1 0 1 0 ;
   1 1 1 1 1 0 0 0 0 1 1 1 1 1 0 1
];

mI=mod(data*G',2);    % on a pris transposee de G pour prendre moins de place
```

```
mI=mod(mI+A,2);              % encodage du message
envoi=[data mI]              % message + code (identit'e a gauche de G')
N=5;envoi(N)=1-envoi(N);     % introduction d'une erreur

H1=[    % forme fournie dans document RDS => on en deduit le syndrome de A
1 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1 1 1 0 1 1 0 0 1 1 1  ;
0 1 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1 1 1 0 1 1 0 0 1 1  ;
0 0 1 0 0 0 0 0 0 0 1 0 1 1 1 0 0 0 1 0 1 1 1 1 1 0  ;
0 0 0 1 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 1 1 1 0 0 0  ;
0 0 0 0 1 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 1 1 1 0 0  ;
0 0 0 0 0 1 0 0 0 0 1 0 1 0 1 1 1 1 1 0 1 0 1 0 0 1  ;
0 0 0 0 0 0 1 0 0 0 1 1 0 0 1 0 0 0 1 0 1 1 0 0 1 1  ;
0 0 0 0 0 0 0 1 0 0 1 1 1 1 1 1 0 1 1 0 0 1 1 1 1 0  ;
0 0 0 0 0 0 0 0 1 0 0 1 1 1 1 1 1 0 1 1 0 0 1 1 1 1  ;
0 0 0 0 0 0 0 0 0 1 0 0 1 1 1 1 1 1 0 1 1 0 0 1 1 1 ];

H2=[    % forme issue de [I | tG] => on en deduit A (matrice de controle)
0 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0  ;
0 0 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 0 1 0 0 0 0 0 0 0  ;
0 1 1 0 0 0 1 1 0 0 1 1 0 0 0 1 0 0 1 0 0 0 0 0 0 0  ;
1 1 0 0 1 1 0 1 1 0 1 0 0 1 1 0 0 0 0 1 0 0 0 0 0 0  ;
1 1 1 0 0 1 1 0 1 1 0 1 0 0 1 1 0 0 0 0 1 0 0 0 0 0  ;
1 0 0 0 1 1 1 1 0 1 0 1 0 1 1 1 0 0 0 0 0 1 0 0 0 0  ;
0 0 1 1 1 0 1 1 1 0 0 1 0 1 0 1 0 0 0 0 0 0 1 0 0 0  ;
1 1 1 0 0 0 0 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 1 0 0  ;
1 1 1 1 0 0 0 0 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 1 0  ;
1 1 1 1 1 0 0 0 0 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 1 ];

res=mod(envoi*H2'-A,2)    %
[tmp,col]=ismember(res,H2',"rows")
res=mod(envoi*H1'-sA2,2) % sA
[tmp,col]=ismember(res,H1',"rows")
```

qu'en encodant un message, y compris l'identifiant (optionnel dans ces expérimentations) de bloc, nous sommes capables d'une part de valider le transfert d'information sans erreur : commenter l'introduction de l'erreur en ligne 19 pour constater que le résultat des deux décodages, avec **H1** ou **H2**, sont tous deux nuls. Par ailleurs, en cas d'introduction d'une erreur, le résultat est bien le contenu de la ligne d'indice égal au numéro du bit erroné.

5.2 Deux erreurs

Retrouver un bit corrompu fonctionne bien, et nous avons compris comment identifier quel bit est corrompu en recherchant la ligne de **H** qui contient le syndrome après décodage d'un message avec un bit qui a été modifié. Cependant, que se passe-t-il si deux bits sont corrompus ?

Une rapide petite expérience numérique nous met sur la voie :

```
N=2;envoi(N)=1-envoi(N); % introduction d'erreurs
N=5;envoi(N)=1-envoi(N); % introduction d'erreurs
```

se traduit par

```
mIr = 0   1   0   0   1   0   0   0   0   0
```

Deux bits sont désormais à **1** après décodage du syndrome, celui de la deuxième et de la cinquième ligne. L'analyse est donc triviale tant que deux des dix premiers bits ont changé : le numéro des bits modifiés apparaissent dans le syndrome calculé en réception, toujours parce que les 10 premières lignes de **H** forment la matrice identité. Ce concept se généralise en citant le cours **[25]** : *« Suppose we wish to correct all patterns of* **t** *errors. In this case, we need to pre-compute more syndromes, corresponding to 0, 1, 2,...* **t** *bit errors. Each of these should be stored by the decoder. ».*

Nous devons donc calculer une nouvelle matrice, déduite de **H**, que nous noterons **m2r**, qui contient tous les syndromes de 10 bits correspondant aux sommes de toutes les combinaisons possibles de 2 lignes de **H**. Une approche par boucle **for**, pas nécessairement la plus élégante pour GNU/Octave, mais qui a le bon goût de fonctionner, donne :

```
m2r=[0 0 0 0 0 0 0 0 0 0]; % seed
l=1;
for k=1:length(H)-1        % compute all
combinations
  for j=k+1:length(H)
     m2r=[m2r ; mod(H(k,:)+H(j,:),2)];
     solution(l,1)=k;       % which line
is to be corrected ?
     solution(l,2)=j;
     l=l+1;
  end
end
m2r=m2r(2:end,:);           % remove seed
```

Évidemment, **m2r** est beaucoup plus grande que **H** puisque cette fois nous avons tous les couples d'erreurs possibles représentés dans chaque ligne de la nouvelle matrice. Nous nous assurons que toutes les lignes sont uniques – propriété du code capable de corriger deux erreurs de communication :

```
for k=1:length(m2r)
  [num,ligne]=ismember(m2r(k,:),m2r,'rows');
  if (num>1)
     printf("doublon %d ",num)
% never twice the same row
  end
end
```

ne renvoie par de réponse (question : pourquoi **[m2runiq, m,n]=unique(m2r,'rows');** renvoie moins de lignes dans **m2runiq** qu'il n'y en a dans **m2r**, alors que toutes les lignes sont uniques ?).

Nous pouvons maintenant rechercher quelle paire de bits a changé lors de la transmission par

```
[num,ligne]=ismember(mIr,m2r,'rows')
if (num>0) solution(ligne,:),end
% display which bits have flipped
```

et comme nous avions pris soin de placer dans **solution** le couple **j** et **k** des lignes de **H** qui ont été sommées pour donner la ligne de **m2r**, nous connaissons l'indice **k** et **j** des bits modifiés. En effet,

```
N=21;envoi(N)=1-envoi(N); % introduction
d'erreurs
N=25;envoi(N)=1-envoi(N); % introduction
d'erreurs
```

se traduit bien par

```
mIr = 0   0   0   1   0   1   0   0   1   1
num = 0
ligne = 0
num =  1
ligne =  314
ans = 21   25
```

dans laquelle nous constatons que la recherche du syndrome **mIr** dans la matrice **H** (une modification) ne donne pas de résultat (**num=0**), mais que la recherche dans **m2r** donne un résultat, avec le syndrome présent dans la 314ème ligne de **m2r**, qui a été calculée lorsque **j=21** et **k=25**. Le décodage fonctionne bien. RDS n'annonce que la capacité à corriger 1 ou 2 bits corrompus lors de la transmission, nous arrêtons donc là notre étude du code correcteur d'erreurs.

Le nombre de bits qui peut ainsi être corrigé par un code est déterminé par le concept de distance de Hamming, qui se représente graphiquement comme la distance entre le « bon » message et le message corrompu reçu, en deçà de laquelle la correction est possible (http://fr.wikipedia.org/wiki/Code_correcteur).

La capacité de corriger des transmissions corrompues est au cœur du gain de débit de communication (Fig. 10), que ce soit dans une liaison avec les sondes spatiales lointaines (*deep space network*) **[24]** ou en liaison courte portée à faible consommation **[26]**. Ce survol rapide d'un exemple simple ouvre donc des perspectives d'approfondissement qui restent d'actualité : sans prétendre comprendre comment générer ces codes, l'expérimentation sur des données réelles avec un résultat concret fournit la motivation à approfondir les ouvrages plus théoriques sur le sujet tels que **[21]** et son excellente mise en relation de compression, cryptographie et correction d'erreurs.

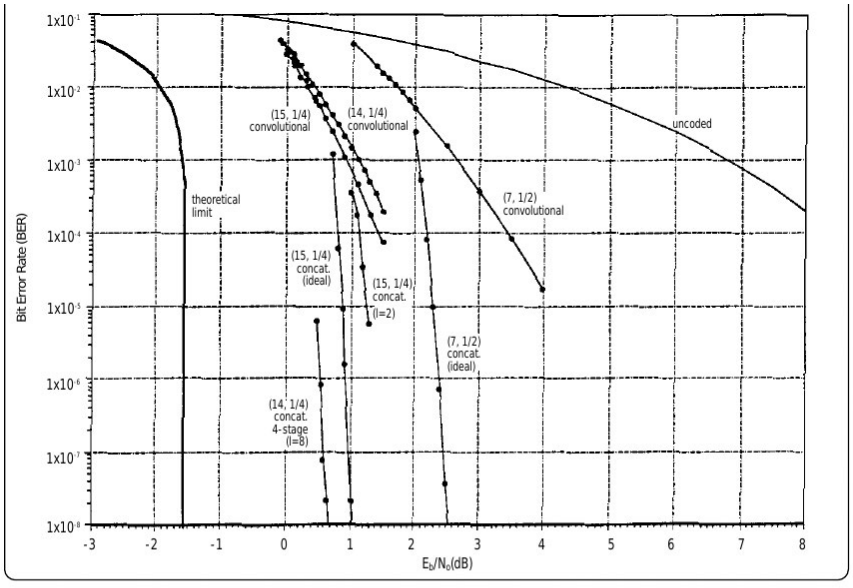


Figure 7-11. Telemetry communication channel performance for various coding schemes. The first set of curves shows the *Voyager* k = code, both alone and in combination with the Reed-Solomon code. The second set of codes illustrates the k = 15 code, which was to be demonstrated with Galileo's original high-rate channel, shown alone and in combination with the Reed-Solomon code, either as constrained by the *Galileo* spacecraft data system (I = 2) or in ideal combination. The third set shows the k = 14 code, devised by the Advanced Systems Program researchers for the actual Galileo low-rate mission, both alone and in combination with the selected variable-redundancy Reed-Solomon code and a complex four-stage decoder. The added complexity of the codes, which has its greatest effect in the size of the ground decoder, clearly provides increased reliability for correct communication.

Fig. 10 : Évolution du taux d'erreur en fonction du rapport signal à bruit, pour divers types de codes correcteurs d'erreurs, reproduit de [24, Fig. 7-11, p. 477].

comme robuste pour un environnement en production, en particulier sur la synchronisation du décodage des trames transmises à 1187,5 bps de façon asynchrone, mais cela se ferait en alourdissant le code proposé au détriment de sa lisibilité. Il nous semble que la démonstration des principes de base de ce mode de modulation est explicite, avec des messages intelligibles et cohérents avec les informations transmises par chaque station. Cet exercice a nécessité environ 2 mois de travail pour aboutir : il s'agit donc d'un bon exercice pour se familiariser avec les diverses couches d'un protocole de communication, en allant de la couche matérielle à la couche session.

Un point décevant est l'absence de transfert précis de temps par ce médium. Nous n'avons reçu que peu de trames de datation (CT, groupe 4A) qui nous permettent de connaître la date et l'heure qu'avec une précision de ±0,1 s au début de chaque minute. Cette déficience sera bientôt corrigée avec le décodage logiciel de DCF77 que nous présenterons dans un autre numéro.

CONCLUSION

Nous avons analysé le protocole RDS, exploité pour la transmission numérique d'informations associée aux chaînes FM de la bande commerciale incluant le nom de la station ou la nature du programme, en partant de l'acquisition de l'information analogique depuis un récepteur de télévision numérique terrestre, puis extraction de l'information numérique sur la sous-porteuse à 57 kHz, avant d'extraire les trames (en ayant résolu la synchronisation des phrases) et d'en interpréter le contenu. Nous avons finalement abordé la correction d'erreurs pour constater que nous pouvions simuler des erreurs connues et les identifier. En appliquant cette méthode de traitement aux données reçues expérimentalement, nous avons pu retrouver plus de 10% de signaux exploitables additionnels par rapport au cas où nous ne traitions que les informations transmises dans leur intégralité.

Bien des développements seraient nécessaires avant que cette démonstration de principe ne puisse être considérée

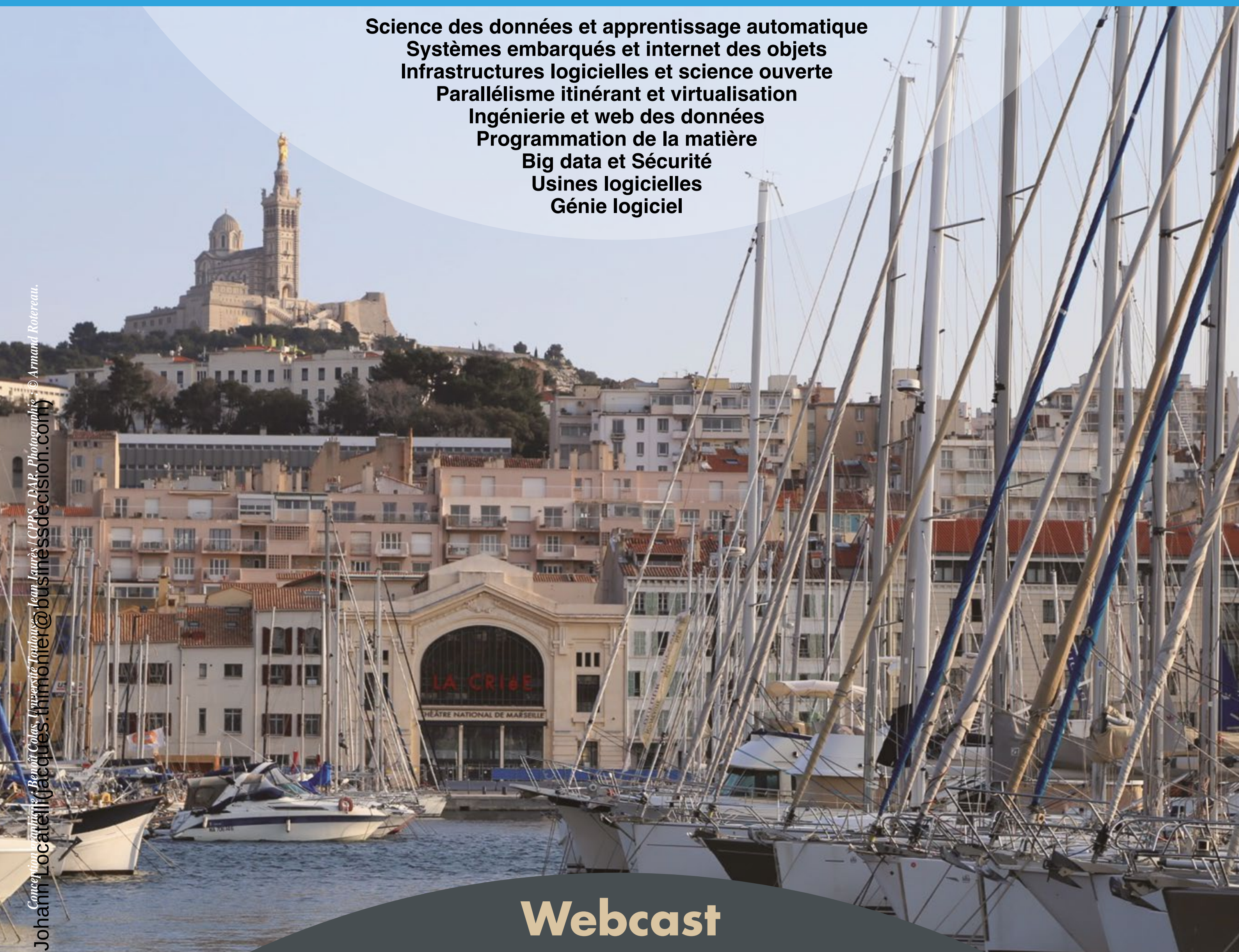
Une archive contenant les scripts GNU/Octave, configuration de GNURadio Companion et fichiers d'exemples, est disponible à http://jmfriedt.free.fr/lm_rds.tar.gz.

REMERCIEMENTS

Je remercie Thomas Lavarenne pour avoir motivé cette étude par ses interrogations et des échanges constructifs, et Bastien Bloessl pour avoir exprimé l'opinion que le décodage de RDS est trivial **[27]**, un commentaire inacceptable tant que ce protocole avait résisté à nos investigations. Ce défaut de compréhension est maintenant corrigé. G. Cabodevila a amélioré la qualité du code GNU/Octave lors de sa relecture, et É. Carry a clarifié un certain nombre de points. Bien que nous ayons acquis une copie papier de l'ouvrage de **Dunod [21]**, les autres références ont été obtenues auprès de **Library Genesis**, http://gen.lib.rus.ec, une source indispensable pour nos recherches.

JDEV 2017

Journées Développement Logiciel



Science des données et apprentissage automatique
Systèmes embarqués et internet des objets
Infrastructures logicielles et science ouverte
Parallélisme itinérant et virtualisation
Ingénierie et web des données
Programmation de la matière
Big data et Sécurité
Usines logicielles
Génie logiciel

Conception graphique : Benoît Colas, Université Toulouse - Jean Jaurès / CPPS - DAP. Photographie : © Armand Rotereau.

Webcast

4, 5, 6, 7 juillet 2017

Aix-Marseille Université, la Canebière

JDEV2017

Information, programme, réservation et inscription :

http://devlog.cnrs.fr/jdev2017

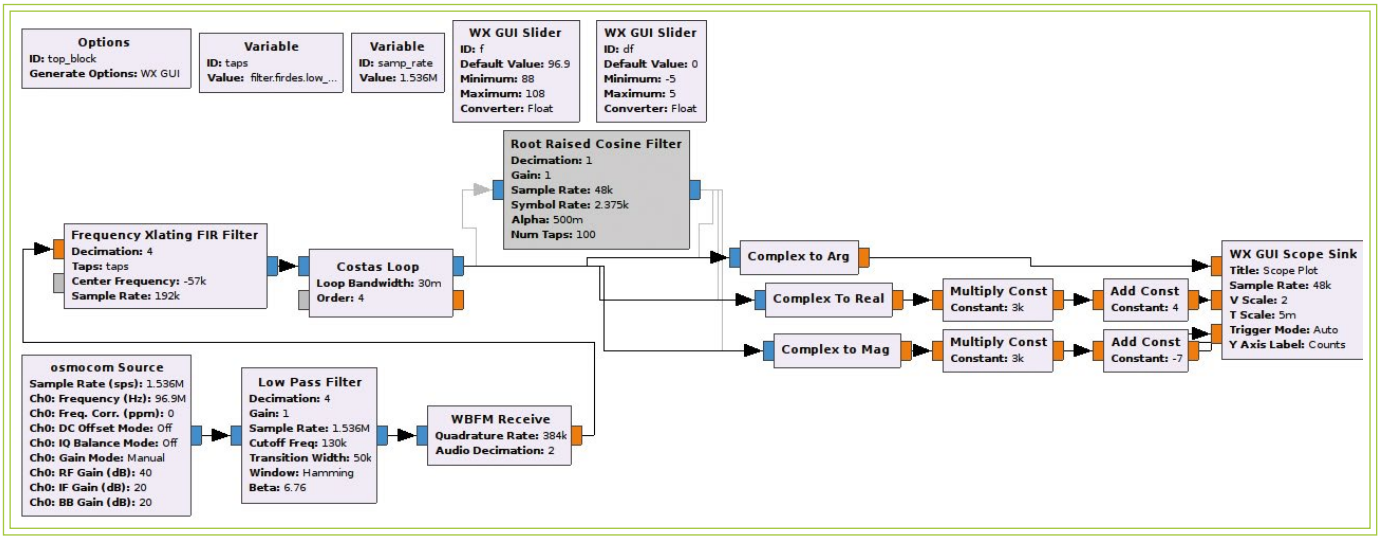


Fig. 11a : graphique pour l'extraction des diverses composantes du signal dans la sous-porteuse RDS après asservissement de l'oscillateur par la boucle de Costas. La phase s'affichera en bleu, la partie réelle en vert, et le module en rouge.

ANNEXE A : PHASE OU AMPLITUDE ?

Nous avons pris le parti au cours de cette étude de considérer le signal transmis par RDS comme une information modulée en phase, car nos premières tentatives consistant à afficher l'amplitude du signal filtré (Band Pass Filter autour de **57 ± 2,5** kHz) se traduisaient par un signal inexploitable. Bien que la boucle de Costas pour asservir en phase reste nécessaire, afficher la partie réelle au lieu de la phase donne aussi un signal parfaitement exploitable (Fig. 11). Quelques petites oscillations sont visibles sur les symboles les plus longs, qui pourraient se traduire par une fausse détection d'un symbole court si on n'y prend garde : un filtre passe-bas optimisé retire cette fluctuation et garantit une meilleure détection de la nature du signal transmis. En effet, filtrer par une fonction de transfert spectrale quasi-rectangulaire se traduit, par transformée de Fourier de la porte rectangulaire, par une convolution par un sinus cardinal (**`sin(x) / x`**) et donc un étalement de chaque symbole sur ses voisins compte tenu du temps de décroissance lent du filtre. Quel que soit le filtrage, tracer l'amplitude du signal (module du complexe) donne une information inexploitable pour la détection des bits : seules la phase ou la partie réelle sont exploitables.

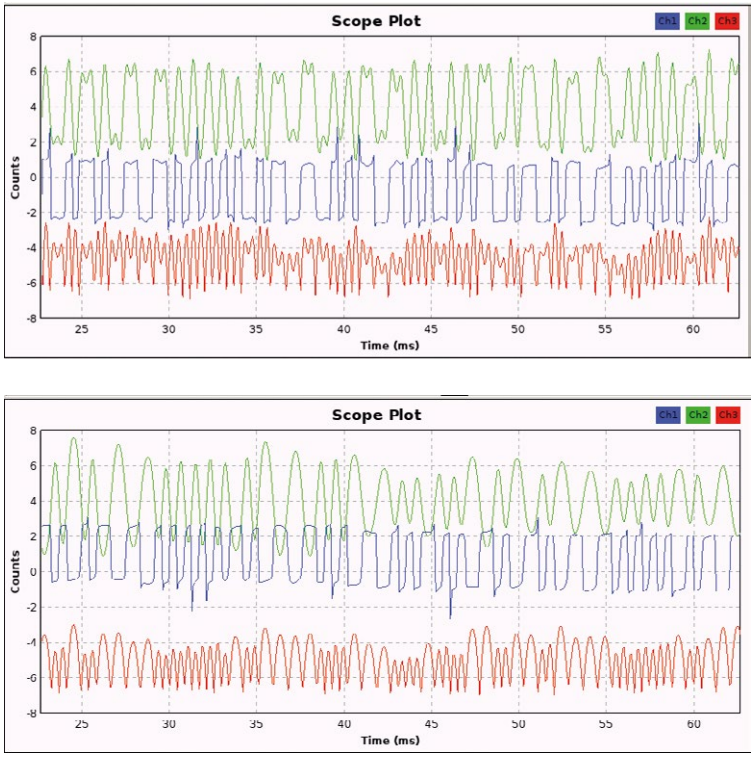


Fig. 11b. En haut : sans filtre passe-bas RRC, la partie réelle présente quelques oscillations sur les symboles les plus longs. En bas : un filtre RRC permet d'atténuer ces artefacts et de rendre la partie réelle du signal exploitable. Dans tous les cas, le module (rouge) est inexploitable.

NOTE

Le filtre Root Raised Cosine (RRC [28]) a été conçu pour pallier à cette déficience : un filtre passe-bande réduisant à la fois l'encombrement spectral et temporel, permettant de filtrer finement un signal tout en évitant un temps de décroissance trop long. Alors qu'un filtre rectangulaire dans le domaine spectral (bleu sur Fig. 12, en haut, cas $\alpha = 0$) se traduit par un excellent filtrage spectral, mais une réponse temporelle en sinus cardinal très longue avec un zéro aux positions des symboles adjacents, la moindre erreur sur la date d'émission d'un symbole (Fig. 12, cas du haut de la courbe en temps) se traduit par une contribution de ce symbole sur ses voisins, d'autant plus importante que les symboles voisins sont loin. Le RRC présente aussi des zéros sur les symboles adjacents, mais sa décroissance est bien plus rapide (Fig. 12, en bas, cas $\alpha = 1$), évitant la pollution des voisins en cas d'erreur de synchronisation [29, p.136][30]. Le paramètre α fournit un levier pour passer continûment du meilleur filtrage spectral à l'optimum entre filtrage spectral et étalement temporel de l'impulsion. Le RRC vise donc à optimiser à la fois l'occupation du spectre en réduisant l'encombrement du canal utilisé (réduction de la bande passante B) pour transmettre le flux numérique, tout en maximisant le débit de communication en réduisant l'intervalle de temps 1/B entre les symboles successifs.

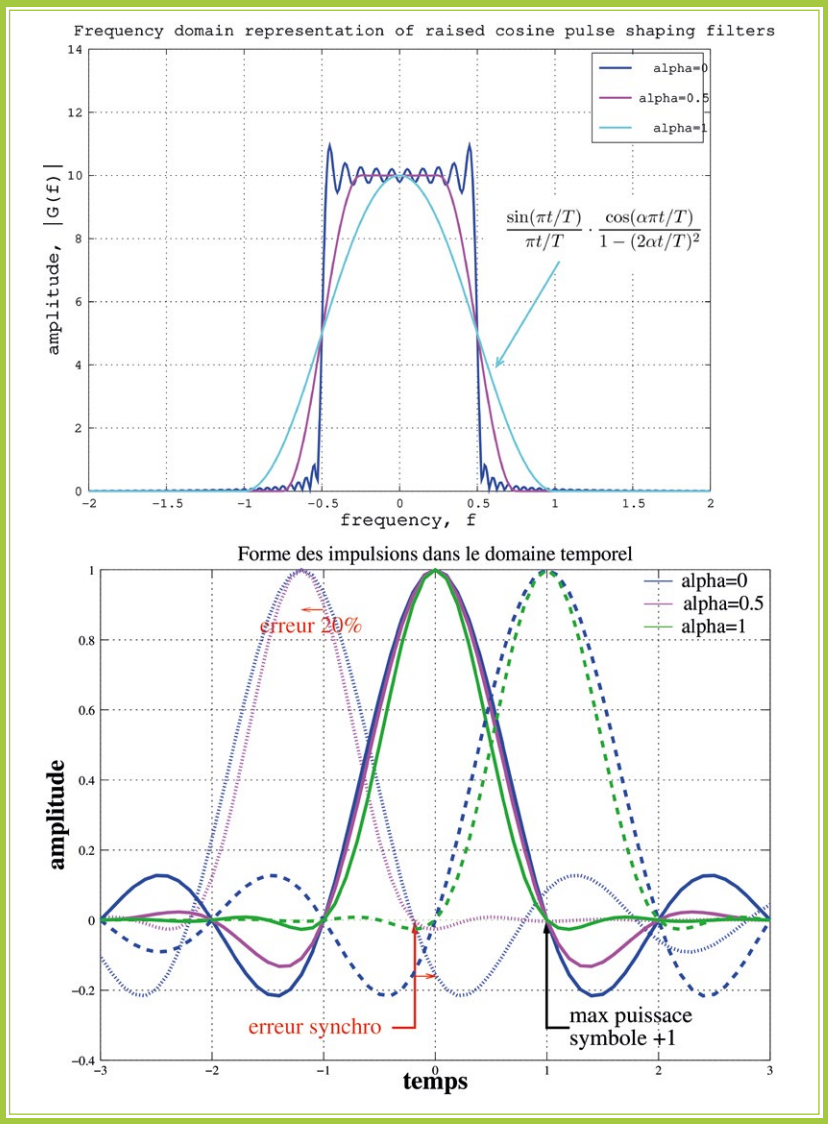


Fig. 12 : En haut : réponse spectrale des filtres RRC avec diverses formes, du rectangle (meilleure sélectivité spectrale) au segment de fonction trigonométrique. En bas : plusieurs impulsions successives dans le temps, filtrées avec diverses formes de RRC, avec l'impulsion de gauche décalée de 20% par rapport à sa date nominale pour une transmission périodique.

ANNEXE B : AJUSTEMENT DE L'HORLOGE DE LA TRAME NUMÉRIQUE

La modulation de phase nécessite une copie locale de la porteuse du signal radiofréquence sans modulation, une tâche résolue par la boucle de Costas qui élimine la modulation de phase. Une fois les bits visibles sous forme de phase à 0 ou π (pour BPSK), il reste le problème de connaître le rythme auquel les bits sont transmis. L'horloge qui cadence la transmission numérique (par exemple un microcontrôleur sur l'émetteur) n'a aucune raison d'être la même source de fréquence que la porteuse radiofréquence, et il faut donc de nouveau retrouver une copie locale de l'horloge qui cadence le signal numérique pour le décoder (Fig. 13). Diverses stratégies sont disponibles, telles que maximiser le nombre de transitions dans le signal transmis pour permettre l'asservissement de l'horloge locale (cas du codage Manchester qui garantit au moins une transition à chaque bit). GNURadio

fournit divers blocs pour la synchronisation de l'horloge cadençant le flux numérique, tels que **Clock Recovery** (selon l'algorithme publié dans **[27]**) ou **MPSK Receiver**. Ces blocs s'alimentent de la sortie de la boucle de Costas qui a éliminé l'erreur grossière de fréquence, et manipulent maintenant les symboles numériques issus du traitement précédent pour ajuster le flux de données. Ces blocs fournissent un échantillon par symbole, rendant le traitement ultérieur très simple (saturation par comparaison – nommé **Binary Slicer** dans GNURadio – puis analyse de la séquence numérique ainsi formée). Nous validons le bon fonctionnement de ces blocs en affichant le diagramme de constellation, qui comporte en ordonnée la partie imaginaire de la sortie du bloc de traitement et en abscisse la partie réelle. Puisque dans le plan complexe l'angle à l'axe des abscisses est la phase et la distance à l'origine la magnitude, un nuage de points à angle constant et distance constante indique une synchronisation efficace. Un nuage de points « qui danse » ou forme un cercle indique l'absence de synchronisation. La

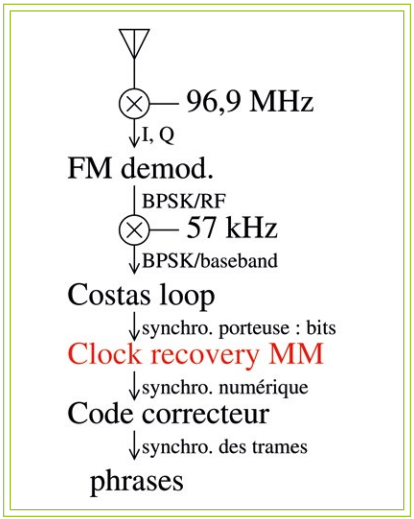


Fig. 13 : Chaîne de traitement des informations acquises depuis l'antenne pour retrouver le message (phrases) : dans cette annexe, nous nous intéressons à la synchronisation du message numérique, en rouge.

sortie directe de la boucle de Costas fournit une illustration du second point (Fig. 14, gauche) tandis que le passage dans un bloc chargé de synchroniser l'horloge cadençant le flux numérique donne bien deux nuages (Fig. 14, droite) qui sont les deux états possibles attendus en BPSK. Nous constatons ici que le filtre RRC que nous avons vu auparavant, même s'il ne change pas de façon significative la forme du signal d'un point de vue visuel, assiste la synchronisation de l'horloge en réduisant la diffusion de puissance dans un état des symboles numériques vers l'autre. Un filtre passe-bas atteint le même résultat, mais moins efficacement.

La configuration des blocs de synchronisation telle que décrite dans http://gnuradio.org/redmine/projects/gnuradio/wiki/Guided_Tutorial_PSK_Demodulation nous informe que nous devons tenter de minimiser le nombre d'échantillons par symbole, tout en le maintenant au-dessus de **2**. Pour ce faire, un RRC est inséré entre la boucle de Costas et le bloc de synchronisation d'horloge, avec un facteur de décimation qui permet d'atteindre un facteur **Omega** (ratio de la fréquence d'échantillonnage au débit du flux numérique) proche, mais au-dessus de **2**. Dans notre exemple, **Omega** vaut **samp_rate/128/(1187.5*2)** avec **128** le produit des décimations des divers blocs de traitements entre la source et la synchronisation d'horloge, **1187,5 x 2** le débit de bits en encodage Manchester, et **samp_rate** le débit de données issues de la source. Les autres paramètres du bloc de synchronisation restent inchangés. Le RRC est quant à lui configuré avec un débit d'entrée égal au débit de la source divisé du produit des divers facteurs de décimations dans la chaîne de traitement qui le précède, et un débit de symboles égal à **1187,5 x 2** de nouveau, déterminant donc sa fréquence de coupure.

Le fichier issu d'une synchronisation de la porteuse (Costas) et du flux numérique (Clock Recovery ou MPSK) se décode parfaitement avec les scripts proposés dans le texte (section 3), en retirant les 4 premières lignes puisque nous avons désormais 1 échantillon/symbole, et en ne gardant des lignes 8 à 24 que **p=angle(r);p=p-mean(p);s=(p>0);s=s-mean(s);s1=s(2:2:end);s2=s(1:2:end-1);**. En effet, **s** est la version saturée de la phase **p**, dont le décodage par Manchester différentiel considère les valeurs successives des paires d'échantillons initiaux. La suite du traitement (synchronisation sur le syndrome des séquences de 16 bits) reste identique, pour par exemple donner

```
station = P 96.9BIP 96.9BIP 96.9BIP 96.9BIP 96.9BIP
96.9BIP 96.9BIP 96.9BIP
```

qui est cette fois parfaite, ou

```
station = EUROPE 1EUROPE 1EUROPE 1EUROPE 1EUROPE 1EUROPE
1EUROPE
```

de nouveau sans corruption de données, voir

```
station = RIRE &  RIRE  RIRE &  RIRE &  RIRE &  RIRE
JEAN    JEAN    JEAN    JEAN    JEAN
JEDUJARDINDUJARDINDUJARDINDUJARDINDUJARDINDUJARD
temps4A 57811 7 51 2
```

ou

```
texte =  FRANNFO - LE 9 : FABIENN - 8APHATIE          FRANCE
INFO - LE 7 | 9 : FABIENNE SINTES - 8H30 APHATIE
```

NE MANQUEZ PAS LA NOUVELLE FORMULE !

LINUX PRATIQUE N°101

PROTÉGEZ VOS MOTS DE PASSE AVEC KEEPASS & CHIFFREZ AVEC VERACRYPT !

ACTUELLEMENT DISPONIBLE

CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :

http://www.ed-diamond.com

```
  FRANCE INFO - LE 7 | 9 :IENNE SINTES - 8H30 APHATIE          FRANCE
INFO - LE 7 | 9 : FABIENNE SINTES - 8H30 APHATIE
  FRANCE INFO - LE 7 | 9 : FABIENNE SINTES - 8H30 APHATIE
station =FO    INFO    INFO    INFO    INFO    INFO    INFO    INFO
INFO    INFO    INFO    INFO    INFO    INFO
  INFO    INFO    INFO    INFO    INFO    INFO    INFO    INFO
INFO    INFO    INFO    INFO    INFO    INFO    INFO
temps4A 57811 7 54 2
```

qui est proche de la perfection. La date est en accord avec nos attentes : la date est le jour julien modifié 57811, que http://www.csgnetwork.com/julianmodifdateconv.html convertit en 27 Février 2017, l'heure est 7h5{1,4} décalée de 2 demi-heures, qui est bien 8h5{1,4}, heure de l'enregistrement. La trame **4A** **[13, p.28]** est donc convenablement décodée et permet de recevoir l'heure par RDS, avec une mise à jour chaque minute annoncée comme exacte à ±0,1 s près. Ce n'est qu'après avoir inclus la synchronisation d'horloge de la trame numérique que nous obtenons des trames de datation de la forme **4A** qui ne sont émises qu'une fois chaque minute. Une

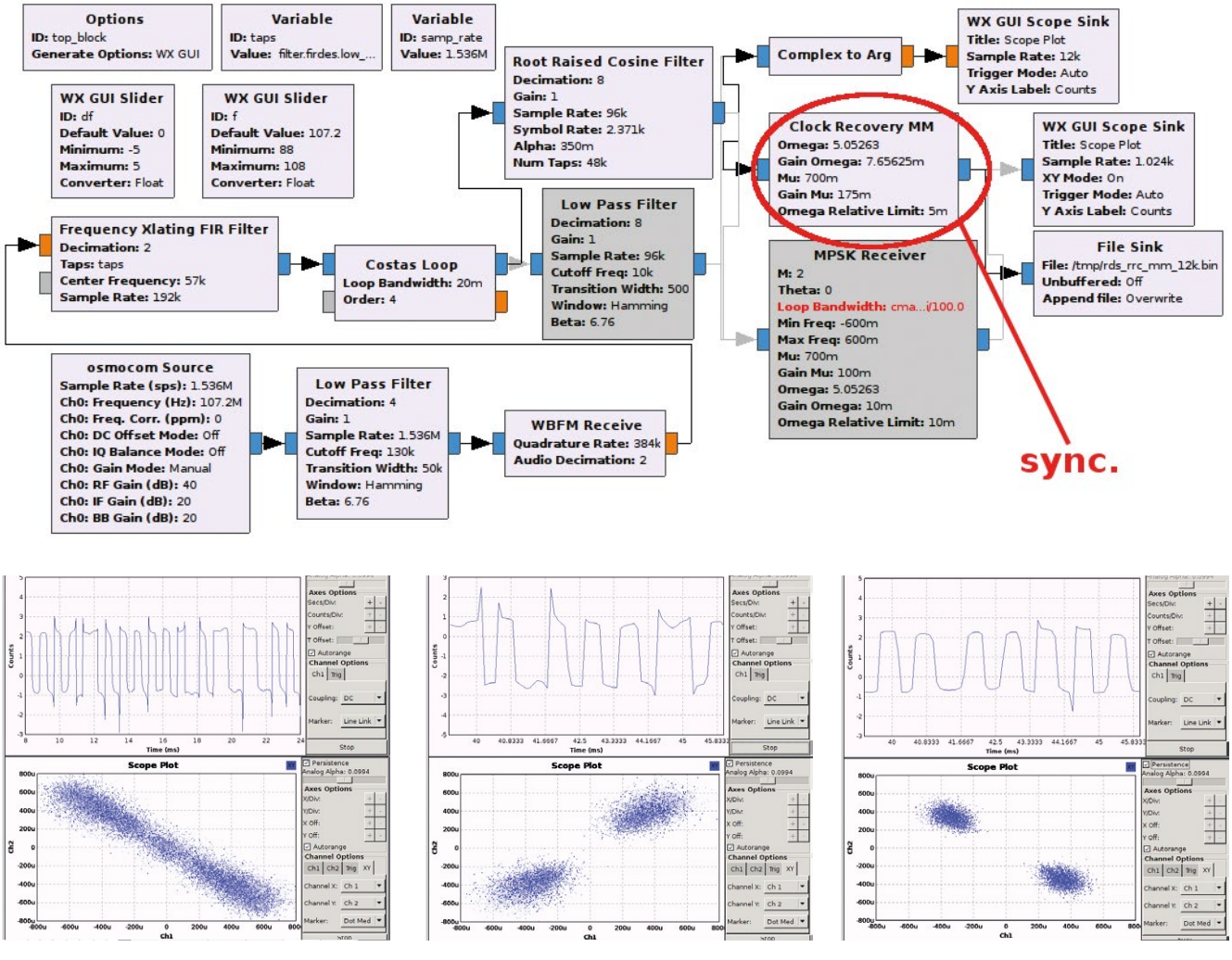


Fig. 14 : En haut : flux de traitement, ajoutant la synchronisation du flux numérique après la synchronisation sur la porteuse (Costas). En bas : diagramme de constellation (mode XY d'un oscilloscope recevant le flux de données complexes) dans les diverses conditions – de gauche à droite, sortie de la boucle de Costas (sans synchronisation de la séquence numérique), synchronisation de l'horloge numérique après un filtre passe-bas, et finalement après un filtre RRC. Deux nuages de points distincts garantissent la capacité à séparer les symboles.

proportion trop faible de trames convenablement décodées a toutes les chances de nous faire rater cette séquence exceptionnelle de bits transmis :

```
station = RT  RTL2    RT  RTL2    RT
RTL2    RT  RTL2    RT  RTL2    RT  RTL2
RT  RTL2    RT  RTL2    RT  RTL2
temps4A 57811 8 3 2
```

Tous les exemples de cette annexe ont été acquis au rythme d'un échantillon complexe (2 x 4 octets) par symbole, tel que le propose la sortie du bloc Clock Recovery MM. Au rythme de 1187,5 x 2 = 2375 Hz, les fichiers d'enregistrement sont typiquement de quelques centaines de kilo-octets pour des acquisitions d'une dizaine de secondes (19000 kB/s).

ANNEXE C : CODE PI

Chaque trame RDS transmise contient, dans son premier paquet (bloc A **[13, p. 15]**), le code PI de la station. Il a été suggéré **[13, p. 66]** que connaissant ce code PI, unique à chaque station radio, il serait possible de se synchroniser sur cette séquence de bits qui se répète tous les 104 bits (4 blocs de 26 bits), au lieu de se battre avec le code correcteur d'erreur et calculer pour chaque séquence de 26 bits reçus si les 10 derniers bits correspondent au syndrome des 16 premiers bits tel que nous l'avons implémenté. Nous aurions pu réécrire l'histoire de cet article en démontrant ce concept avant d'entreprendre l'implémentation du code correcteur d'erreur, mais la vérité est que la liste des codes PI n'a été trouvée qu'une fois cette prose achevée. En effet, le site http://www.csa.fr/maradiofm/radiords_tableau fournit la liste des codes **PI** des diverses stations autorisées. Par exemple pour Radio Campus à Besançon, nous apprenons que son code **PI** est **FC3A**. En ajoutant dans la boucle de décodage des trames, au même titre que le texte libre ou le nom de la station, le décodage du **PI** par

```
if (PI==0) PI=dec2hex(data1(1:16)*2.^[15:
-1:0]')  % PI
```

en ayant pris soin d'initialiser PI=0 en dehors de la boucle, notre script GNU/Octave de décodage identifie bien **PI = FC3A** lorsque nous écoutons 102,4 MHz. Le code est bien identifié de cette façon. ■

Retrouvez toutes les notes et références de cet article sur le blog de GNU/Linux Magazine :
http://www.gnulinuxmag.com/

DÉPLOYEZ KUBERNETES SUR VOS RASPBERRY PI AVEC KUBEADM

STÉPHANE BEURET
[Principal Platform Architect à Data Essential]

MOTS-CLÉS : KUBERNETES, KUBEADM, RASPBERRY PI, HYPRIOT, DOCKER

C'est un fabuleux tour de force de faire tourner Kubernetes (K8s) sur Raspberry Pi (Rpi), mais les gens brillants sont nombreux dans le monde de l'open source, et la somme de leur travail fait qu'on peut y arriver aujourd'hui presque sans le moindre effort. Personnellement ce n'est pas la première fois que je m'y essaye, mais jusqu'à présent, c'était toujours plus ou moins... du bricolage. Jusqu'à ce que je tombe sur Kubeadm ! Kubeadm est un tout nouvel outil (en alpha) qui permet de déployer Kubernetes aussi bien sur CentOS 7, Ubuntu 16.04 ou HypriotOS v1.0.1+. Pas d'autre prérequis (d'accord : du réseau et 1GB de mémoire vive, la belle affaire !). Tout ça donne envie de se lancer !

1. PRÉPARATION MATÉRIELLE

Pour cet article, j'utilise quatre Raspberry Pi 3 (que je n'ai plus besoin de vous présenter), bien plus véloces que leurs précédentes itérations. Néanmoins si votre budget est plus restreint, deux Pi feront tout aussi bien l'affaire. J'ai pris des câbles arc-en-ciel, c'est-à-dire que chacun a une couleur différente, pour relier les cartes Ethernet à un petit hub, ce qui constitue mon réseau privé (`192.168.1.0/24`), accessible uniquement entre les Raspberry, et depuis mon laptop. Pour le réseau public (`192.168.100.0/24`), j'utilise la carte wifi. Si vous n'avez qu'une seule interface réseau, ça marchera tout aussi bien.

Cependant, je vais vous faire une confidence, pour la carte SD, j'ai peur de ne pas avoir fait le meilleur choix... et j'ai dû recourir à une petite astuce que je dévoilerai plus loin pour réduire les latences, et que mon processeur ne passe pas son temps à attendre que les données soient écrites sur « disque ». Si votre choix de carte SD n'est pas encore fait, ou si voulez en changer, mais que vous ne savez pas laquelle est la mieux adaptée, alors laissez-moi vous diriger

vers ce wiki qui est de très bon conseil : http://elinux.org/RPi_SD_cards. Pour ma part, ma carte SD doit faire environ du 10MB/s en écriture, ce qui est un peu juste. J'ai donc retrouvé dans mes tiroirs trois clés USB sur lesquelles j'ai « strippé » un volume LVM, ce qui fait un net gain par rapport à la carte SD même si je ne l'ai pas objectivement mesuré ; voici comment j'ai procédé :

```
# pvcreate /dev/sda /dev/sdb /dev/sdc
# vgcreate vg_stripped /dev/sda /dev/sdb /dev/sdc
# lvcreate -i 3 -l 100%FREE -n stripped_lv  stripped_vg
# mkfs.ext4 /dev/mapper/stripped_vg-stripped_lv
# mount /dev/mapper/stripped_vg-stripped_lv /mnt
```

J'ai ensuite fait un **rsync** de mon **/var** sur le nouveau volume :

```
# rsync -ar /var/* /mnt
# sync
# umount /mnt
```

J'ai modifié **/etc/fstab** en ajoutant cette ligne à la fin du fichier :

```
/dev/mapper/stripped_vg-stripped_lv /var /ext4 defaults,
noatime 1 2
```

Et j'ai redémarré la Pi. **Docker** tournant dans **/var/lib/docker**, c'est bien dans mon **/var** que j'ai besoin de performances. Bien sûr, pour appliquer cette procédure, il nous faut d'abord installer les Pi, alors nous y venons.

2. MISE EN PLACE DU SYSTÈME D'EXPLOITATION

Côté système d'exploitation (SE), aucune d'hésitation : **HypriotOS**. Pour celles et ceux qui ne connaîtraient pas encore Hypriot, c'est un petit groupe de hackers (dans le meilleur sens du terme) qui s'est fixé comme mission de faire de Raspberry Pi LA plateforme de choix pour faire tourner des conteneurs ; Docker étant actuellement la technologie dominante dans ce domaine, ils se sont concentrés sur celui-ci pour créer HypriotOS. Ce dernier est aussi facile à installer qu'à utiliser. N'hésitez pas à aller faire un tour sur leur blog qui est une mine d'informations (http://blog.hypriot.com/). Hypriot a, par ailleurs, mis au point un petit utilitaire pour simplifier l'installation d'HypriotOS sur une carte SD : **flash**. Je vais vous montrer comment l'utiliser, et je vous invite à consulter sa page **GitHub** (https://github.com/hypriot/flash) si vous envisagez de l'utiliser vous-même.

Pour l'installation de flash, pas de difficulté particulière, il n'y a qu'à suivre les instructions ; poursuivons donc directement sur la création de la carte SD. Je commence par télécharger la dernière version d'HypriotOS qui se trouve ici : http://blog.hypriot.com/downloads/. Pour cet article, j'utilise la v1.1.1 qui embarque un Docker 1.12.3. Une fois cette étape réalisée, je prépare mes fichiers de configuration yaml. Ils ressemblent à ceci :

```
hostname: Pi8ma
wifi:
  interfaces:
    wlan0:
      ssid: "hotspot"
      password: "t@pS3krt"
```

J'en prépare 4 de la sorte, dans lesquels je n'ai qu'à changer le nom d'hôte. HypriotOS est un peu opiniâtre et ne permet pas de changer de nom après coup, donc pensez bien à différencier vos Pi. Il ne me reste plus qu'à lancer la commande :

```
$ flash --config Pi8ma.yaml
hypriotos-rpi-v1.1.1.img
```

Et en moins de 5 minutes, mes quatre Rpi sont prêts, voire déjà démarrés ! Il ne reste plus qu'à configurer le réseau. Niveau interface publique, comme je l'ai dit, j'utilise le wifi. Et ici, ô magie, je n'ai rien à configurer sur mes Pi, car j'ai un petit serveur **hostapd** couplé à un **dnsmasq** qui ronronne sur une autre framboise (eh oui, encore une), et qui va se charger de tout, mais un simple DHCP (celui de votre box internet par exemple) fera tout aussi bien l'affaire... Pour la partie privée, un tout petit peu de configuration manuelle dans **/etc/network/interfaces.d/eth0**, car là ce réseau ne bénéficie pas de DHCP :

```
allow-hotplug eth0
iface eth0 inet static
  address 192.168.1.233
  netmask 255.255.255.0
```

Terminons cette section par un petit aperçu de ce à quoi ressemble cette topologie réseau en figure 1.

3. PETIT RAPPEL SUR KUBERNETES

Avant de se lancer dans l'installation proprement dite, un mot peut-être sur **Kubernetes**. D'abord, il est open source, et sert à orchestrer des conteneurs... par milliers ! C'est à l'origine le projet **Borg** de **Google**, qui a plus de dix ans maintenant, que Google a donné à la **Cloud Native Computing Foundation** (https://www.cncf.io/). La version 1.0 de Kubernetes date du 21 juillet 2015.

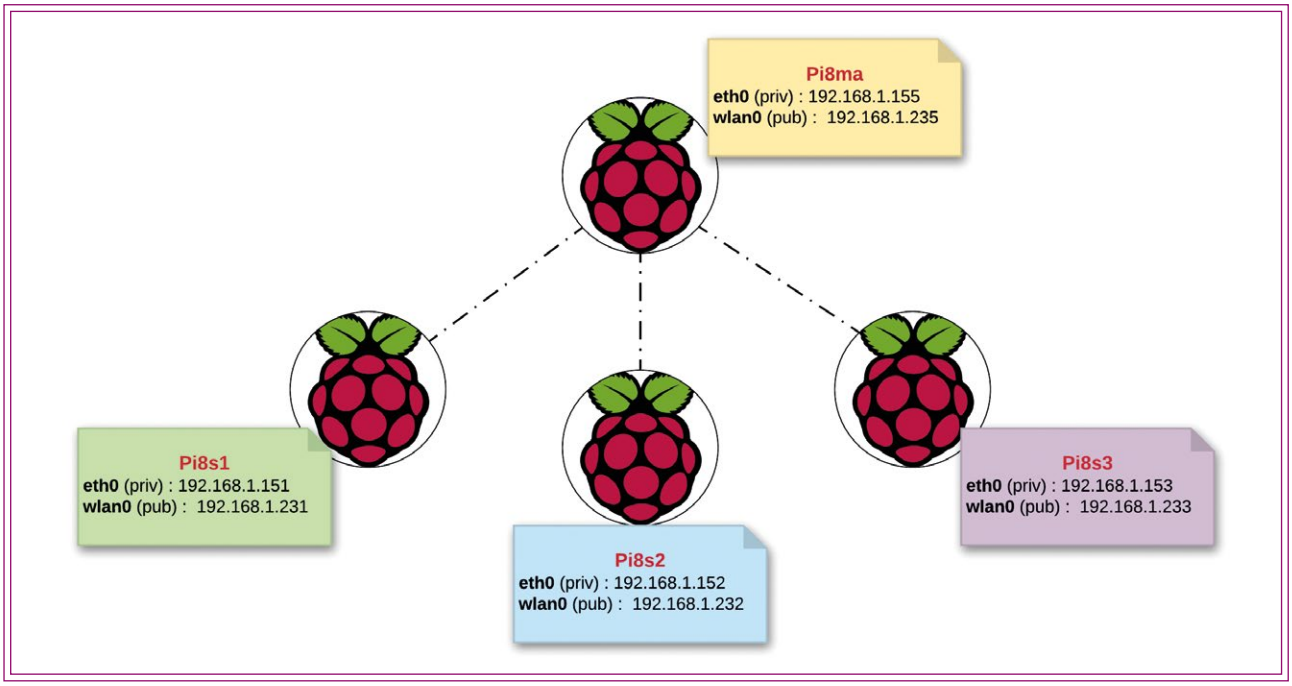


Fig. 1 : Schéma réseau des Rapsberry Pi.

Il se compose d'un ou plusieurs maîtres (*masters*), et de **n** *minions* (*worker nodes*). Sur le maître, on va trouver les composants suivants :

- **etcd**, le stockage clé-valeur de **CoreOS** ;
- **API serveur**, qui est l'interface principale de K8s (Kubernetes), aussi bien pour les services externes qu'internes ;
- **Scheduler**, qui va décider sur quel minion va tourner un pod (voir encadré) en fonction des ressources de l'hôte ;
- **Controller manager**, qui crée, met à jour et détruit les ressources qu'il gère.

Quant aux minions, ils hébergent les composants suivants :

- **kubelet**, responsable du bon fonctionnement du minion, il vérifie aussi que tous les conteneurs sont en bonne santé ;
- **kube-proxy**, qui est aussi bien un proxy qu'un équilibreur de charges ; c'est lui qui dirige le trafic réseau vers le bon pod ;
- **cAdvisor** est un agent de monitoring qui collecte différentes métriques sur les conteneurs.

À cette liste déjà longue de services de base, ajoutons-en un autre qui est indispensable, mais qui ne fait pas partie de K8s **core** : le réseau. Mon propos n'étant pas ici de faire un cours sur Kubernetes, je finis simplement en disant que pour ce service il existe un grand nombre d'addons, et que le plus commun est **flannel** (CoreOS).

LA NOTION DE POD

Les pods sont spécifiques à Kubernetes. Un pod est un groupe d'un ou plusieurs conteneurs, le stockage partagé pour ces conteneurs, et les options concernant la manière dont ils doivent être exécutés. Un pod peut être vu comme un hôte logique pour une application. S'il n'y avait pas les conteneurs, l'application ou les applications étroitement liées de ce pod auraient été exécutées sur la même machine physique ou virtuelle.

4. C'EST PARTI : DÉPLOYONS K8S !

Première étape, installer tous les binaires nécessaire à K8s sur tous les rpi :

```
# curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
# cat << EOF > /etc/apt/sources.list.d/
# curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
```

```
# cat << EOF > /etc/apt/
sources.list.d/kubernetes.list
deb http://apt.kubernetes.io/
kubernetes-xenial main
EOF
# apt-get update
# apt-get install -y kubelet
kubeadm kubectl kubernetes-cni
```

Seconde étape, configurer le maître. Il n'y a qu'une commande à lancer, **kubeadm init**, tellement simple ! Cependant, nous devons lui ajouter quelques arguments :

- **--pod-network-cidr=10.244.0.0/16**, c'est un prérequis pour utiliser flannel, ce qui est mon cas. Mais j'expliquerai les raisons de ce choix quand nous en viendrons au réseau ;
- **--api-advertise-addresses=192.168.1.235**, comme j'ai deux interfaces réseau, je spécifie par laquelle je veux passer pour accéder à l'API.

Alors allons-y :

```
# kubeadm init
--api-advertise-
addresses=192.168.1.235
--pod-network-
cidr=10.244.0.0/16
```

Après quelques minutes (le temps que toutes les images soient téléchargées et que tout se mette en place), on a le message suivant :

```
Kubernetes master
initialised successfully!

You can now join any number
of machines by running the
following on each node:

kubeadm join
--token=2bccc8.
ec9402a7861c7533
192.168.1.235
```

Gardez bien au frais le *token*, il servira chaque fois qu'on souhaitera ajouter un minion.

Un petit coup d'œil pour vérifier que tout est bien en place :

```
$ kubectl get pods --all-namespaces
NAMESPACE     NAME                                READY   STATUS              RESTARTS   AGE
kube-system   dummy-2501624643-7d34m              1/1     Running             0          10m
kube-system   etcd-pi8ma                          1/1     Running             1          9m
kube-system   kube-apiserver-pi8ma                1/1     Running             0          9m
kube-system   kube-controller-manager-pi8ma       1/1     Running             0          9m
kube-system   kube-discovery-2202902116-zb36j     1/1     Running             0          10m
kube-system   kube-dns-2334855451-i2dl1           0/3     ContainerCreating   0          10m
kube-system   kube-proxy-8asmf                    1/1     Running             0          10m
kube-system   kube-scheduler-pi8ma                1/1     Running             0          9m
```

Et ça *roxxe* ! Pas tout à fait, allez-vous dire, et notre pod DNS alors ? Celui-ci est en effet en attente du réseau. C'est quelque chose que j'ai survolé plus tôt, et je vais aller un peu plus dans les détails cette fois. Kubernetes permet l'utilisation de différents *addons*, que ce soit pour le réseau ou pour le tableau de bord, afin d'étendre ses capacités. Quelques exemples vaudront mieux qu'un long discours :

- flannel, que j'ai déjà cité, crée un réseau commun entre les minions ;
- **Calico**, qui fait beaucoup parler de lui, en plus du réseau, permet de faire des règles de sécurité ;
- **Weave Net**, qui amène lui aussi des règles de sécurité au niveau du réseau.

Rendez-vous sur cette page pour plus de détails : http://kubernetes.io/docs/admin/addons/.

Si j'ai arrêté mon choix sur flannel pour cet article, c'est pour deux raisons :

- Flannel n'utilise qu'une seule image, ça fait donc moins à télécharger (les choix sont parfois dictés par la plateforme) ;
- les règles de réseaux amènent nécessairement une charge supplémentaire au niveau de la mémoire et du CPU ; alors si ça n'est pas sensible sur de très gros serveurs, par contre on le ressent vite sur des Raspberry.

Pour l'installation de flannel, là encore, c'est très simple. Rendez-vous à cette adresse sur GitHub : https://github.com/coreos/flannel/blob/master/Documentation/kube-flannel.yml, et téléchargez le fichier yaml. Une fois posé sur votre maître, il ne reste plus qu'à changer l'image initiale, qui est de l'amd64 pour de l'arm :

```
$ sed -i s/amd64/arm/g kube-flannel.yml
```

Puis, pour créer les pods :

```
$ kubectl apply -f kube-flannel.yml
configmap "kube-flannel-cfg" created
daemonset "kube-flannel-ds" created
```

Si tout se passe bien, une fois que le pod flannel est lancé, le pod DNS suit :

```
$ kubectl get pods --all-namespaces
NAMESPACE     NAME                          READY   STATUS    RESTARTS   AGE
kube-system   kube-dns-2334855451-i2dl1     3/3     Running   0          32m
kube-system   kube-flannel-ds-h47b2         2/2     Running   0          1m
```

5. AJOUT DES MINIONS

NOTE

Par souci de sécurité, les maîtres ont pour unique rôle de piloter la ferme de minions. Pour un lab ou une démo, on peut bien entendu ajouter le maître lui-même à cette ferme, tant il est vrai que vous pouvez implémenter et configurer K8s à votre guise. Cependant, gardez à l'esprit que, du point de vue de la sécurité, ce n'est pas une bonne pratique. Par ailleurs, dites-vous aussi que notre serveur est un tout petit arm64, et que s'il est maître et minion, il va être bien difficile pour lui d'accueillir en plus des services !

Vous allez voir que l'ajout des minions se fait lui aussi sans peine. Sur l'un d'entre eux, entrez simplement le résultat de la commande obtenu sur le maître lors du **`kubeadm init`** :

```
# kubeadm join --token=2bccc8.ec9402a7861c7533
192.168.1.235
Running pre-flight checks
<util/tokens> validating provided token
<node/discovery> created cluster info
discovery client, requesting info from
"http://192.168.1.235:9898/cluster-info/v1/?token-
id=2bccc8"
<node/discovery> cluster info object received,
verifying signature using given token
<node/discovery> cluster info signature and
contents are valid, will use API endpoints
[https://192.168.1.235:6443]
<node/bootstrap> trying to connect to endpoint
https://192.168.1.235:6443
<node/bootstrap> detected server version v1.4.4
<node/bootstrap> successfully established connection
with endpoint https://192.168.1.235:6443
<node/csr> created API client to obtain unique
certificate for this node, generating keys and
certificate signing request
<node/csr> received signed certificate from the API
server:
Issuer: CN=kubernetes | Subject:
CN=system:node:Pi8s1 | CA: false
Not before: 2016-11-24 20:01:00 +0000 UTC Not After:
2017-11-24 20:01:00 +0000 UTC
<node/csr> generating kubelet configuration
<util/kubeconfig> created "/etc/kubernetes/kubelet.
conf"

Node join complete:
* Certificate signing request sent to master and
response
  received.
* Kubelet informed of new secure connection details.

Run 'kubectl get nodes' on the master to see this
machine join.
```

Répétez l'opération sur chaque minion, puis allez sur le maître vérifier que tous vos minions ont bien rejoint la grappe :

```
$ kubectl get nodes
NAME      STATUS    AGE
pi8ma     Ready     22h
pi8s1     Ready     2m
pi8s2     Ready     15s
pi8s3     Ready     1s
```

Merveilleux, ils sont tous là ! Nous allons pouvoir les utiliser tout de suite.

NOTE

Petite astuce au cas où quelque chose s'est mal passé : à tout moment vous pouvez démonter votre Kubernetes avec la commande `kubeadm reset`. Une fois lancée, la seule chose qui restera sur les hôtes, ce seront les images.

6. NOTRE PREMIÈRE APPLICATION

Maintenant que nous avons terminé notre déploiement, il est temps de le mettre à l'épreuve. Mon test consistera à lancer un *deployment* de nginx, puis de l'exposer au travers d'un *service*.

LA TERMINOLOGIE DE KUBERNETES

Si j'utilise les termes anglo-saxons de *deployment* ou *service*, ce n'est certainement pas pour faire de la peine aux amoureux de la langue française, mais parce que Kubernetes a son propre vocabulaire. Voici un petit rappel de certains termes, pour les autres, je vous invite à consulter cette page : http://kubernetes.io/docs/user-guide/ :

- ***Deployment*** **: un déploiement est la déclaration de l'état d'un service. Il s'appuie sur un *replication controller* ;**
- ***Replication controller*** **: il maintient le nombre de répliques désiré de pods pour un déploiement, et s'assure de leur bonne santé ;**
- ***Service*** **: un service définit le moyen d'accéder aux pods, comme une adresse IP stable et un nom DNS correspondant.**

Fig. 2 : Notre serveur nginx est accessible depuis n'importe où.

Je choisis sur dockerhub la première image de nginx pour armhf, et je lance mon déploiement :

```
$ kubectl run nginx --image=werwolfby/armhf-alpine-nginx
--replicas=3 --port=80
deployment "nginx" created
```

Je vérifie que tout s'est bien déroulé :

```
$ kubectl get deployment
NAME    DESIRED  CURRENT  UP-TO-DATE  AVAILABLE   AGE
nginx   3        3        3           3           1m
```

Je regarde si mes pods sont bien distribués :

```
$ kubectl get pods -o wide
NAME                 READY  STATUS   RESTARTS  AGE  IP          NODE
nginx-304267959-14qu5  1/1    Running  0         3m   10.244.2.2  pi8s2
nginx-304267959-7cxwp  1/1    Running  0         3m   10.244.3.2  pi8s3
nginx-304267959-u9ud0  1/1    Running  0         3m   10.244.1.2  pi8s1
```

Ensuite, j'expose Nginx pour qu'il soit accessible, aussi bien à l'intérieur qu'à l'extérieur de mon cluster :

```
$ kubectl expose deployment nginx --port=80
--type=LoadBalancer
service "nginx" exposed
```

Je récupère les informations sur mon service afin de connaître l'adresse IP de mon équilibreur de charge, ainsi que le port par lequel nginx est accessible sur les minions :

```
$ kubectl describe service nginx
Name:   nginx
Namespace:  default
Labels:  run=nginx
Selector:  run=nginx
Type:   LoadBalancer
IP:   10.103.121.55
Port:   <unset> 80/TCP
NodePort:  <unset> 30983/TCP
Endpoints:  10.244.1.7:80,10.24
4.2.10:80,10.244.3.9:80
Session Affinity: None
No events.
```

Depuis n'importe quel minion, nginx est à présent accessible :

```
$ curl 10.103.121.55
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the
nginx web server is successfully
installed and
working. Further configuration
is required.</p>
</body>
</html>
```

Et depuis l'extérieur, mon service est accessible sur le port **30983** de tous mes minions (voir figure 2).

C'est maintenant votre tour de faire vos propres déploiements !

CONCLUSION

La fin de cet article annonce le début d'une grande aventure : vous êtes maintenant en mesure d'orchestrer votre propre cluster Kubernetes, que ce soit pour assouvir une simple curiosité, pour tester des développements ou pour acquérir un meilleur niveau dans cette technologie. Et je rends ici un hommage ému à la communauté open source pour avoir créé et enrichi de telles technologies, que ce soit Kubernetes, Hypriot, CoreOS ou Docker, sans oublier bien sûr la fondation Raspberry ! ■

PROGRAMMEZ VOTRE OBJET CONNECTÉ RPI AVEC PYTHON

PATRICE KADIONIK
[Maître de Conférences HDR à l'ENSEIRB-MATMECA, Bordeaux-INP]

MOTS-CLÉS : IOT, PYTHON, RASPBERRY PI, THREADS, SOCKETS

Cet article présente la programmation avec le langage Python d'un objet connecté conçu autour d'une carte Raspberry Pi 3 B.

L'Internet des Objets (IoT, *Internet of Things*) est un domaine en pleine ébullition. Le domaine n'est pas novateur du point de vue technique. C'est la rencontre au bon moment d'un ensemble de techniques pour répondre à un besoin du moment. C'est par exemple ce qui s'est produit avec le Web au début des années 1990...

Le domaine est en pleine expansion avec une prévision de près de 80 milliards d'objets connectés en 2020 ! Dans ce cadre, concevoir un objet connecté pour répondre à un besoin doit se faire de façon simple et rapide pour pouvoir fournir une offre le plus rapidement possible.

Le prototypage rapide peut être une réponse à cette contrainte, prototypage rapide du point de vue matériel, mais aussi du point de vue logiciel.

Pour cela, la carte **Raspberry Pi** répond bien à cette contrainte : un matériel bon marché et fiable, l'accès simple à des entrées/sorties (E/S), mais aussi un environnement logiciel prêt à l'emploi avec comme système d'exploitation Linux embarqué. De plus, les concepteurs de la carte Raspberry Pi promeuvent l'usage du langage **Python** avec leur carte pour former une nouvelle génération de programmeurs aux joies de la programmation. D'ailleurs la carte Raspberry Pi n'aurait-elle pas dû s'appeler à l'origine carte Raspberry Py comme l'a signalé Eben Upton, fondateur de la « Raspberry Pi Foundation » : *« we came up with this name 'Raspberry Pi' in the end - the 'Pi' in 'Raspberry pi' is the 'Py' in 'Python', um, misspelled, so I've had to defend my spelling for the last 4-and-a-half years »* ?

L'usage du langage Python pour programmer un objet connecté à base de carte Raspberry Pi est donc naturel et répond à la contrainte du prototypage rapide...

Nous avons pu voir dans les articles **[1]** et **[2]** parus dans *Open Silicium* la conception d'un objet connecté pour la mise en œuvre de **Java** embarqué. Cet objet connecté a été conçu autour d'une carte Raspberry Pi 3 B et d'une carte d'E/S spécifique. Ce matériel n'est pas lié à un langage de programmation particulier et

il est parfaitement possible de le réutiliser comme base matérielle pour une programmation en langage Python. Nous allons donc partir de cette base matérielle dont la réalisation est parfaitement décrite dans l'article **[1]** pour réaliser notre objet connecté programmé en langage Python.

L'auteur supposera enfin que le lecteur sait programmer en langage Python...

1. CARTE CIBLE RPI-PYTHON

La carte cible présentée dans l'article **[1]** est construite autour d'une carte Raspberry Pi 3 B (RPi). Appelée initialement carte rpi-java, elle est l'association de la carte RPi et d'une carte d'E/S spécifique. Elle est bien sûr indépendante d'un langage de programmation particulier et elle devient par un coup de baguette magique la carte rpi-python pour les besoins de cet article... La réalisation de la carte rpi-python ne pose aucun problème particulier et l'on pourra se référer à l'article **[1]** et aux ressources en ligne de l'article **[3]** pour sa réalisation.

La carte d'E/S est connectée à l'aide d'un connecteur 2x20 broches au connecteur d'E/S de la carte RPi.

La carte d'E/S rpi-python possède :

- 5 leds : LED1 à LED5 ;
- 4 boutons poussoirs : BP1 à BP4 ;
- Un afficheur LCD 2x16 caractères interfacé sur le bus I2C de la carte Rpi ;
- Un capteur de température numérique DS1624 interfacé sur le bus I2C de la carte Rpi.

On notera sur le tableau suivant la correspondance entre le numéro de broche du connecteur 2x20 broches de la carte RPi (BOARD), le numéro de broche notation BCM et le nom du signal de la carte d'E/S.

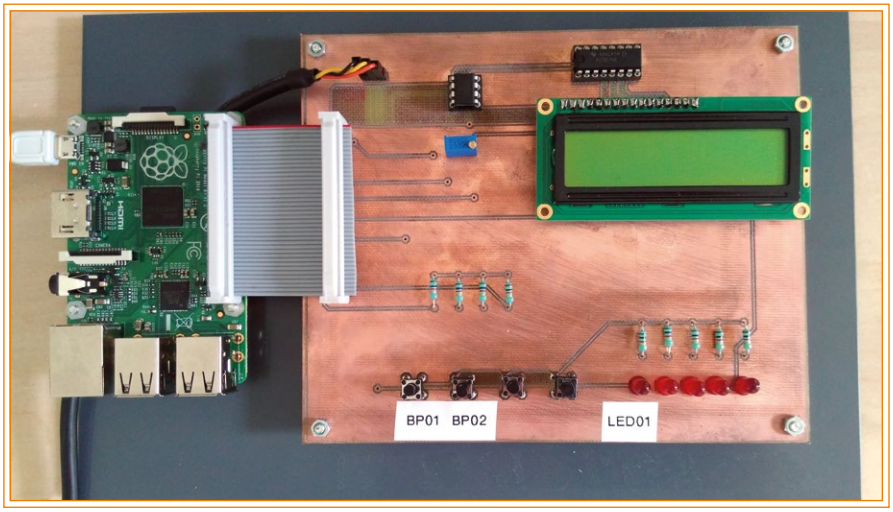


Fig. 1 : Carte cible rpi-python.

Numéro de broche BOARD du connecteur RPi	Numéro de broche BCM correspondant	Fonction
1		3,3 V
2		5 V
3	2	Bus I2C
5	3	Bus I2C
6		RS232
7	4	BP1
8	14	RS232
10	15	RS232
11	17	BP2
12	18	LED2
13	27	BP4
15	22	BP3
16	23	LED3
18	24	LED4
22	25	LED5
25		Masse
26	7	LED1

Connecteur 2x20 broches et signaux de la carte rpi-python.

Par la suite, on adoptera le fait qu'une commande Linux embarqué sur la carte cible rpi-python sera écrite :

```
rpi-python% commande Linux
```

2. INSTALLATION ET CONFIGURATION DE RASPBIAN

Nous allons installer une distribution Linux embarqué sur la carte SD de la carte Raspberry Pi. Le choix se porte naturellement sur la distribution **Raspbian [4]** que l'on installera sur la carte SD. L'installation est classique et décrite ici **[5]**. La version installée est la version **2017-02-16-raspbian-jessie**.

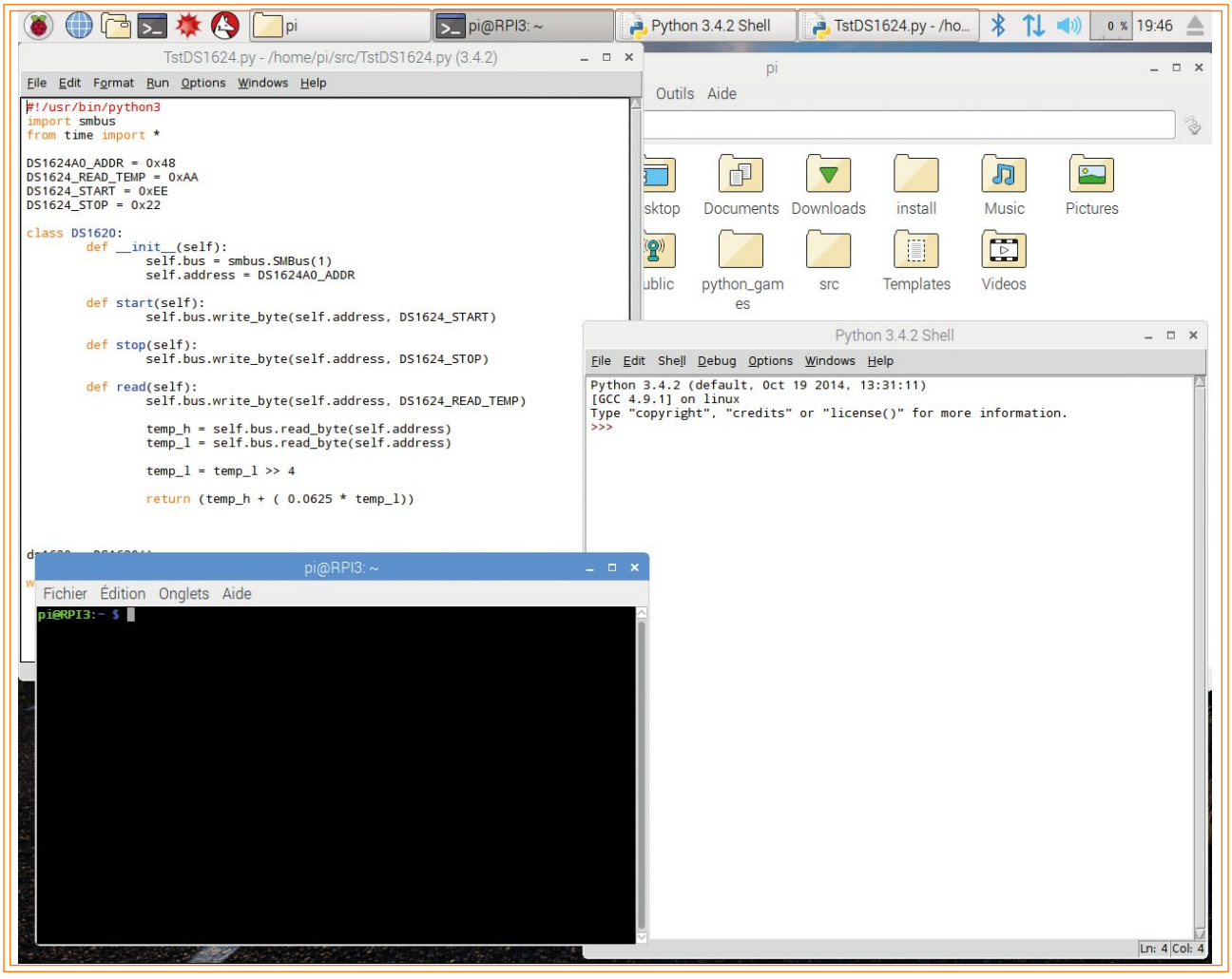


Fig. 2 : Environnement de développement Python IDLE3.

L'environnement de développement ne sera pas pour une fois croisé, mais natif, c'est-à-dire que l'environnement hôte de développement et la cible se confondent. Cela facilitera le développement et le test.

Pour cela, on utilisera l'environnement graphique fourni par Raspbian en branchant son écran au connecteur HDMI de la carte RPi ainsi qu'un clavier et une souris USB.

On retrouve alors l'environnement de bureau comme sur la figure 2.

On retrouve ainsi un environnement graphique **LXDE** classique avec un atelier de développement Python IDLE. On utilisera Python 3 et il faudra donc utiliser **IDLE3**.

Avant de passer au développement logiciel, il faudra au préalable configurer la carte Rpi. On ajustera le fichier de configuration réseau **/etc/network/interfaces** pour coller à sa topologie (ici adresse statique pour l'interface Ethernet) :

```
rpi-python% cat  /etc/network/interfaces
...
auto eth0
iface eth0 inet static
 address xx.xx.xx.xx
 netmask yy.yy.yy.yy
 gateway zz.zz.zz.zz
...
```

On ajustera aussi le serveur de nom DNS du fichier de configuration **/etc/network/resolvconf.conf** :

```
rpi-python% cat  /etc/network/resolvconf.
conf
...
name_servers=tt.tt.tt.tt
...
```

On ajustera le fichier de configuration **/boot/config.txt** pour activer le bus I2C :

```
rpi-python% cat /boot/config.txt
...
# Uncomment some or all of these to
enable the optional hardware interfaces
dtparam=i2c_arm=on
#dtparam=i2s=on
#dtparam=spi=on
...
```

Il est possible d'utiliser l'outil **raspi-config** pour l'activation du bus I2C, comme le montre la figure 3, après exécution de :

```
rpi-python% sudo raspi-config
```

On ajustera le fichier de configuration **/etc/modules** pour charger les modules Linux I2C :

```
rpi-python% cat /etc/modules
# /etc/modules: kernel modules to load at
boot time.
#
# This file contains the names of kernel
modules that should be loaded
# at boot time, one per line. Lines
beginning with "#" are ignored.

i2c-dev
i2c_bcm2708
```

On pourra vérifier que le bus I2C est bien actif après redémarrage de la carte RPi :

```
rpi-python% lsmod|grep i2c
i2c_bcm2708             4834  0
i2c_dev                 5859  0

rpi-python% dmesg|grep i2c
[    1.971246] i2c /dev entries driver
[    1.986707] bcm2708_i2c 3f804000.i2c: BSC1
Controller at 0x3f804000 (irq 83) (baudrate
100000
```

On pourra aussi vérifier que les 2 périphériques de la carte rpi-python sont bien visibles :

```
 rpi-python% i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: 20 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- 48 -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

Pour la partie programmation Python, on vérifiera que le paquetage **python-smbus** est bien installé (il l'est par défaut). Si ce n'est pas le cas, on l'installera par la commande :

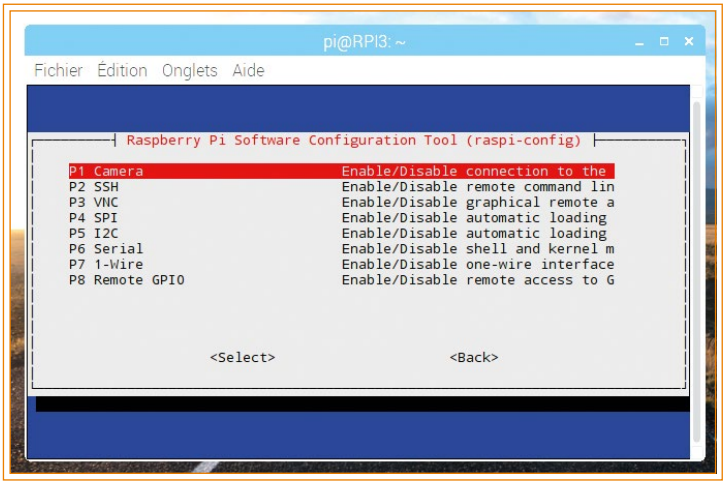


Fig. 3 : Configuration avec l'outil raspi-config.

```
rpi-python% sudo apt-get install python-smbus
```

De même, on vérifiera que les paquetages Python **RPi.GPIO** et **CharLCD** sont bien installés (**RPi.GPIO** l'est par défaut). Si ce n'est pas le cas, on les installera par la commande :

```
rpi-python% sudo pip3 install RPi.GPIO
rpi-python% sudo pip3 install CharLCD
```

La bibliothèque Python **RPi.GPIO** permet d'accéder aux E/S de la carte RPi donc aux leds et boutons-poussoirs de la carte.

La bibliothèque Python CharLCD permet d'accéder à l'afficheur LCD de la carte d'E/S.

L'environnement logiciel est enfin prêt pour une programmation avec Python...

3. ACCÈS PYTHON AUX GPIO

Nous utiliserons pour cela la bibliothèque Python **RPi.GPIO**. L'importation de la bibliothèque se fera par la directive :

```
import RPi.GPIO as GPIO
```

On a alors à disposition les principales fonctions suivantes :

- **GPIO.setmode(GPIO.BOARD)** : numérotation des broches en mode BOARD ;
- **GPIO.setmode(GPIO.BCM)** : numérotation des broches en mode BCM ;
- **GPIO.setup(x, GPIO.IN)** : configuration de la broche **x** en entrée ;

- **GPIO.setup(x, GPIO.OUT)** : configuration de la broche **x** en sortie ;
- **GPIO.input(x)** : lecture de l'état courant de l'entrée **x** ;
- **GPIO.output(x, GPIO.LOW)** : positionnement à l'état bas (**0**) de la sortie **x** ;
- **GPIO.output(x, GPIO.HIGH)** : positionnement à l'état haut (**1**) de la sortie **x**.

La bibliothèque RPi.GPIO permet aussi :

- d'utiliser une sortie en mode PWM (*Pulse Width Modulation*) ;
- de récupérer la valeur d'une entrée non pas par scrutation, mais par événement de la même façon qu'avec un *listener* en langage Java.

On pourra se référer au document **[6]** pour plus d'informations.

La réalisation d'un chenillard à la « K2000 » sur la carte rpi-python se fera très simplement avec le code source Python suivant :

```
#!/usr/bin/python3
import RPi.GPIO as GPIO
from time import *

LED1 = 26
LED2 = 12
LED3 = 16
LED4 = 18
LED5 = 22

GPIO.setwarnings(False)
GPIO.setmode(GPIO.BOARD)

GPIO.setup(LED1, GPIO.OUT)
GPIO.setup(LED2, GPIO.OUT)
GPIO.setup(LED3, GPIO.OUT)
GPIO.setup(LED4, GPIO.OUT)
GPIO.setup(LED5, GPIO.OUT)

while True:
    GPIO.output(LED1, GPIO.LOW)
    GPIO.output(LED2, GPIO.HIGH)
    sleep(0.1)

    GPIO.output(LED2, GPIO.LOW)
    GPIO.output(LED3, GPIO.HIGH)
    sleep(0.1)

    GPIO.output(LED3, GPIO.LOW)
    GPIO.output(LED4, GPIO.HIGH)
    sleep(0.1)

    GPIO.output(LED4, GPIO.LOW)
    GPIO.output(LED5, GPIO.HIGH)
    sleep(0.1)

    GPIO.output(LED5, GPIO.LOW)
    GPIO.output(LED4, GPIO.HIGH)
    sleep(0.1)

    GPIO.output(LED4, GPIO.LOW)
    GPIO.output(LED3, GPIO.HIGH)
    sleep(0.1)

    GPIO.output(LED3, GPIO.LOW)
    GPIO.output(LED2, GPIO.HIGH)
    sleep(0.1)

    GPIO.output(LED2, GPIO.LOW)
    GPIO.output(LED1, GPIO.HIGH)
    sleep(0.1)
```

La bibliothèque RPi.GPIO est donc très souple et simple à mettre en œuvre pour piloter les GPIO de la carte rpi-python.

4. ACCÈS PYTHON AU BUS I2C

Deux périphériques I2C sont connectés sur le bus I2C de la carte rpi-python. Nous trouvons ainsi :

- un afficheur LCD 2x16 caractères en mode 4 bits par l'intermédiaire d'un circuit adaptateur I2C/GPIO (identificateur I2C 0x20) ;
- un capteur de température numérique **DS1624** (identificateur I2C 0x48).

L'accès au capteur de température se fait très simplement avec la bibliothèque I2C native de Python. Il faudra au préalable importer la bibliothèque **smbus** par la directive :

```
import smbus
```

On a alors à disposition les principales fonctions suivantes :

- **bus = smbus.SMBus(x)** : choix du bus I2C numéro **x** ;
- **bus.write_byte(address, command)** : écriture I2C de la commande **command** pour le périphérique de numéro I2C **address** ;
- **bus.read_byte(address)** : lecture I2C de la donnée 8 bits du périphérique de numéro I2C **address**.

Dès lors, il est facile de piloter le capteur de température. Nous créons un objet Python **DS1624** qu'il suffit d'instancier.

Nous avons alors à disposition les méthodes suivantes :

- **DS1624()** : constructeur ;
- **start()** : lancement de l'acquisition de la température ;
- **stop()** : fin de l'acquisition de la température ;
- **double read_temp()** : lecture de la température.

Un exemple d'usage du capteur de température est donné par le code source Python suivant :

```
#!/usr/bin/python3
import smbus
from time import *

DS1624A0_ADDR = 0x48
DS1624_READ_TEMP = 0xAA
DS1624_START = 0xEE
DS1624_STOP = 0x22

class DS1624:
        def __init__(self):
             self.bus = smbus.SMBus(1)
             self.address = DS1624A0_ADDR

        def start(self):
             self.bus.write_byte(self.address, DS1624_START)

        def stop(self):
             self.bus.write_byte(self.address, DS1624_STOP)

        def read_temp(self):
             self.bus.write_byte(self.address, DS1624_READ_TEMP)

             temp_h = self.bus.read_byte(self.address)
             temp_l = self.bus.read_byte(self.address)

             temp_l = temp_l >> 4

             return (temp_h + (0.0625 * temp_l))



ds = DS1624()

while True :
        ds.start()
        sleep(1)

        temperature = ds.read_temp()
        print ("T1=%02.1f" % temperature)
```

Le pilotage de l'afficheur LCD se fait à l'aide de la bibliothèque Python **CharLCD [8]**. Nous voyons ici la force de Python et de ses bibliothèques. Il y a sur le site pypi.python.org un nombre important de paquetages Python **[7]** fournissant des bibliothèques directement utilisables avec la carte RPi. Alors, pourquoi réinventer la roue ? Tout cela va donc dans le sens d'un prototypage... rapide !

Il faudra au préalable importer les bibliothèques RPi.GPIO et charlcd par les directives :

```
import RPi.GPIO as GPIO
from charlcd import direct as lcd
from charlcd.drivers.gpio import Gpio
from charlcd.drivers.i2c import I2C
```

La bibliothèque Python CharLCD permet de piloter un afficheur LCD en mode 8 bits ou 4 bits, un afficheur I2C ou bien via un circuit adaptateur I2C/GPIO PCF8574 comme utilisé par la carte rpi-python.

Elle fournit pour notre cas les objets Python I2C et CharLCD qu'il suffit d'instancier avec les bons paramètres. Pour la carte rpi-python, nous avons un afficheur 2x16 caractères à l'adresse I2C **0x20** sur le bus I2C numéro **1**. Cela donne l'initialisation suivante :

```
i2c = I2C(0x20, 1)
i2c.pins = {
    'RS': 4,
    'E': 6,
    'E2': None,
    'DB4': 0,
    'DB5': 1,
    'DB6': 2,
    'DB7': 3
}

lcd = lcd.CharLCD(16, 2, i2c)
```

Nous avons alors à disposition les principales méthodes suivantes :

- **init()** : initialisation de l'affichage ;
- **write(string)** : écriture d'une chaîne de caractères string à la position courante ;
- **write(string, li, co)** : écriture d'une chaîne de caractères string à la ligne **li** et colonne **co** ;
- **lcd.set_xy(li, co)** : positionnement à la ligne **li** et colonne **co**.

Un exemple d'usage de l'afficheur LCD est donné par le code source Python suivant :

```
#!/usr/bin/python3
import RPi.GPIO as GPIO
from charlcd import direct as lcd
from charlcd.drivers.gpio import Gpio
from charlcd.drivers.i2c import I2C

GPIO.setmode(GPIO.BCM)


i2c = I2C(0x20, 1)
i2c.pins = {
    'RS': 4,
    'E': 6,
    'E2': None,
    'DB4': 0,
    'DB5': 1,
    'DB6': 2,
    'DB7': 3
}

lcd = lcd.CharLCD(16, 2, i2c)
lcd.init()

lcd.write("Hello world!")
lcd.set_xy(0, 1)
lcd.write("Raspberry Pi")
```

5. PROGRAMMATION DE THREADS PYTHON

Il est important de pouvoir créer une application Python multitâches. Python fournit tout ce qu'il faut pour créer des *threads*.

On peut implanter un *thread* sous forme d'un objet Python implémentant l'« interface » **threading.Thread**. Il faudra au préalable importer la bibliothèque **threading** par la directive :

```
import threading
```

Un exemple de création de *threads* est donné par le code source Python suivant :

```
import threading
import time

class Affiche(threading.Thread):
    def __init__(self, nom = ''):
        threading.Thread.__init__(self)
        self.nom = nom
        self.Terminated = False
    def run(self):
        while not self.Terminated:
            print (self.nom)
            time.sleep(2.0)
    def stop(self):
        self.Terminated = True


a = Affiche("Thread A")
b = Affiche("Thread B")

a.start()
b.start()
time.sleep(10)
a.stop()
b.stop()
```

Les méthodes suivantes peuvent être alors utilisées pour :

- **run()** : lancement du *thread* ;
- **stop()** : arrêt du *thread*.

6. API SOCKETS PYTHON

Si l'on veut contrôler à distance son objet connecté, au plus simple, il faut pouvoir créer un serveur TCP. Là encore, Python est notre ami et fournit une bibliothèque **socket** qui permet d'implémenter des clients et des serveurs. Elle ressemble bien sûr dans sa syntaxe et son usage à l'API sockets en langage C sous *NIX...

Il faudra au préalable importer la bibliothèque **socket** par la directive :

```
import socket
```

On trouvera alors l'enchaînement classique **socket()**, **bind()**, **listen()**, **accept()** et **recv()**/**send()** bien connu de tout programmeur sous *NIX.

Un exemple de mini-serveur web itératif écoutant sur le port **8080** est donné par le code source Python suivant :

```
import socket

socket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
socket.bind(('', 8080))
socket.listen(1)

while True:
```

```
        client, address = socket.accept()

        request = client.recv(1024)

        client.send(str.encode("HTTP/1.1
200 OK\n"))
        client.send(str.encode("Content-
Type: text/html\n"))
        client.send(str.encode("\n"))

        client.send(str.encode("<meta
http-equiv=\"Refresh\" content=\"1\">"))
        client.send(str.
encode("<CENTER><H1>Welcome to the rpi-
python Web Server</H1></CENTER>"))

        client.send(str.encode("Hello
World!"))

        client.close()
```

Le serveur itératif renvoie à chaque requête HTTP la même réponse HTTP suivante :

```
HTTP/1.1 200 OK
Content-Type: text/html

<meta http-equiv="Refresh" content=1>
<CENTER><H1>Welcome to the rpi-python Web
Server</H1></CENTER>
Hello World!
```

La page web est rafraîchie toutes les secondes grâce à la balise HTML `<meta http-equiv="Refresh" content="1">`.

On peut apprécier alors la puissance et la simplicité du langage Python.

7. PROGRAMMATION PYTHON DE L'OBJET CONNECTÉ

Nous avons maintenant tout ce qu'il faut pour programmer notre objet connecté sous Python. Notre objet connecté représenté par la carte rpi-python possède les fonctionnalités suivantes :

- il est interrogeable à distance et intègre un serveur web qui fournit la température courante et l'heure courante ;
- il affiche localement sur l'afficheur LCD de la carte rpi-python la température courante et l'heure courante ;
- il réalise un chenillard sur les leds de la carte.

On peut bien sûr imaginer implanter d'autres fonctionnalités. Tout reste possible et est affaire de périphériques intégrés sur la carte d'E/S et d'imagination...

Le code source Python (fichier **iot-rpi.py**) de notre objet connecté est disponible dans les ressources de l'article **[3]** et est reproduit ci-après :

```
#!/usr/bin/python3
import smbus
import threading
import socket

from time import *

import RPi.GPIO as GPIO

from charlcd import direct as lcd
from charlcd.drivers.gpio import Gpio
from charlcd.drivers.i2c import I2C

DS1624A0_ADDR = 0x48
DS1624_READ_TEMP = 0xAA
DS1624_START = 0xEE
DS1624_STOP = 0x22

# Notation BCM
LED1 = 7
LED2 = 18
LED3 = 23
LED4 = 24
LED5 = 25

temp_www = 0

class DS1624:
    def __init__(self):
        self.bus = smbus.SMBus(1)
        self.address = DS1624A0_ADDR

    def start(self):
        self.bus.write_byte(self.address, DS1624_
START)

    def stop(self):
        self.bus.write_byte(self.address, DS1624_
STOP)

    def read_temp(self):
        self.bus.write_byte(self.address, DS1624_
READ_TEMP)

        temp_h = self.bus.read_byte(self.address)
        temp_l = self.bus.read_byte(self.address)

        temp_l = temp_l >> 4

        return (temp_h + (0.0625 * temp_l))

class task_led(threading.Thread):
    def __init__(self, nom = ''):
        threading.Thread.__init__(self)
        self.nom = nom
        self.Terminated = False
        GPIO.setup(LED1, GPIO.OUT)
        GPIO.setup(LED2, GPIO.OUT)
        GPIO.setup(LED3, GPIO.OUT)
        GPIO.setup(LED4, GPIO.OUT)
        GPIO.setup(LED5, GPIO.OUT)

    def run(self):
        while not self.Terminated:
            GPIO.output(LED1, GPIO.LOW)
            GPIO.output(LED2, GPIO.HIGH)
            sleep(0.1)

            GPIO.output(LED2, GPIO.LOW)
            GPIO.output(LED3, GPIO.HIGH)
            sleep(0.1)

            GPIO.output(LED3, GPIO.LOW)
            GPIO.output(LED4, GPIO.HIGH)
            sleep(0.1)

            GPIO.output(LED4, GPIO.LOW)
            GPIO.output(LED5, GPIO.HIGH)
            sleep(0.1)

            GPIO.output(LED5, GPIO.LOW)
            GPIO.output(LED4, GPIO.HIGH)
            sleep(0.1)

            GPIO.output(LED4, GPIO.LOW)
            GPIO.output(LED3, GPIO.HIGH)
            sleep(0.1)

            GPIO.output(LED3, GPIO.LOW)
            GPIO.output(LED2, GPIO.HIGH)
            sleep(0.1)

            GPIO.output(LED2, GPIO.LOW)
            GPIO.output(LED1, GPIO.HIGH)
            sleep(0.1)

    def stop(self):
        self.Terminated = True


class task_lcd(threading.Thread):
    def __init__(self, nom = ''):
        threading.Thread.__init__(self)
        self.nom = nom
        self.Terminated = False

    def run(self):
        global temp_www
        while not self.Terminated:
            temp = 100.0

            ds = DS1624()
            lcd.init()

            while (True):
                ds.start()
                ds.stop()
                temp = ds.read_temp()
                temp_www = temp

                lcd.set_xy(0, 0)
                t = strftime('%H:%M:%S')
                lcd.write(t)
```

```
                lcd.set_xy(0, 1)
                lcd.write("TEMP=%02.1f oC" % temp)

                sleep(0.01)

    def stop(self):
        self.Terminated = True


class task_www(threading.Thread):
    def __init__(self, nom = ''):
        threading.Thread.__init__(self)
        self.nom = nom
        self.Terminated = False

        self.socket = socket.socket(socket.AF_INET, socket.
SOCK_STREAM)
        self.socket.bind(('', 8080))
        self.socket.listen(1)

    def run(self):
        global temp_www
        while not self.Terminated:
            while (True):
                client, address = self.socket.accept()

                request = client.recv(1024)

                client.send(str.encode("HTTP/1.1 200 OK\n"))
                client.send(str.encode("Content-Type: text/
html\n"))
                client.send(str.encode("\n"))

                client.send(str.encode("<meta http-
equiv=\"Refresh\" content=\"1\">"))
                client.send(str.encode("<CENTER><H1>Welcome
to the rpi-python Web Server</H1></CENTER>"))

                t = strftime('%H:%M:%S')
                client.send(str.encode(t))

                client.send(str.encode("<BR>"))

                client.send(str.encode("TEMP=%02.1f °C"
% temp_www))

                client.close()

    def stop(self):
        self.Terminated = True

####
# Debut
####


# Initialisation GPIO
GPIO.setwarnings(False)
GPIO.setmode(GPIO.BCM)

# Initialisation LCD
i2c = I2C(0x20, 1)
i2c.pins = {
    'RS': 4,
    'E': 6,
    'E2': None,
    'DB4': 0,
    'DB5': 1,
    'DB6': 2,
    'DB7': 3
}
lcd = lcd.CharLCD(16, 2, i2c)

# Lancement threads
t1 = task_lcd("task_lcd")
t1.start()
t2 = task_led("task_led")
t2.start()
t3 = task_www("task_www")
t3.start()
```

Le code source ne pose pas de difficultés de compréhension. On trouve ainsi la création de trois *threads* Python :

- **Thread task_led** : gestion des leds de la carte rpi-python, réaliser un chenillard à la K2000 ;
- **Thread task_lcd** : gestion de l'afficheur LCD avec affichage de l'heure et de la température courantes de la carte rpi-python ;
- **Thread task_www** : implémentation d'un serveur web sur le port d'écoute **8080** qui renvoie l'heure courante et la température courante. La page web est rafraîchie toutes les secondes.

En forgeant la requête web @IP_carte_rpi-python:8080, on accède avec un navigateur web à la page d'accueil « *Welcome to the rpi-python Web Server* » qui fournit l'heure courante et la température courante de notre objet connecté.

La figure 4 montre ce qui s'affiche dans le navigateur web lorsque l'on interroge à distance l'objet connecté.

Le lancement de l'application **iot-rpi** peut se faire à travers l'IDE IDLE3 ou bien directement en ligne de commandes comme suit :

```
rpi-python% sudo python3 iot-rpi.py
```

Il est dès lors facile de créer un shell script pour que cette application soit automatiquement lancée au démarrage de la carte que l'on configurera pour être dans l'init level 3 (sans environnement graphique).

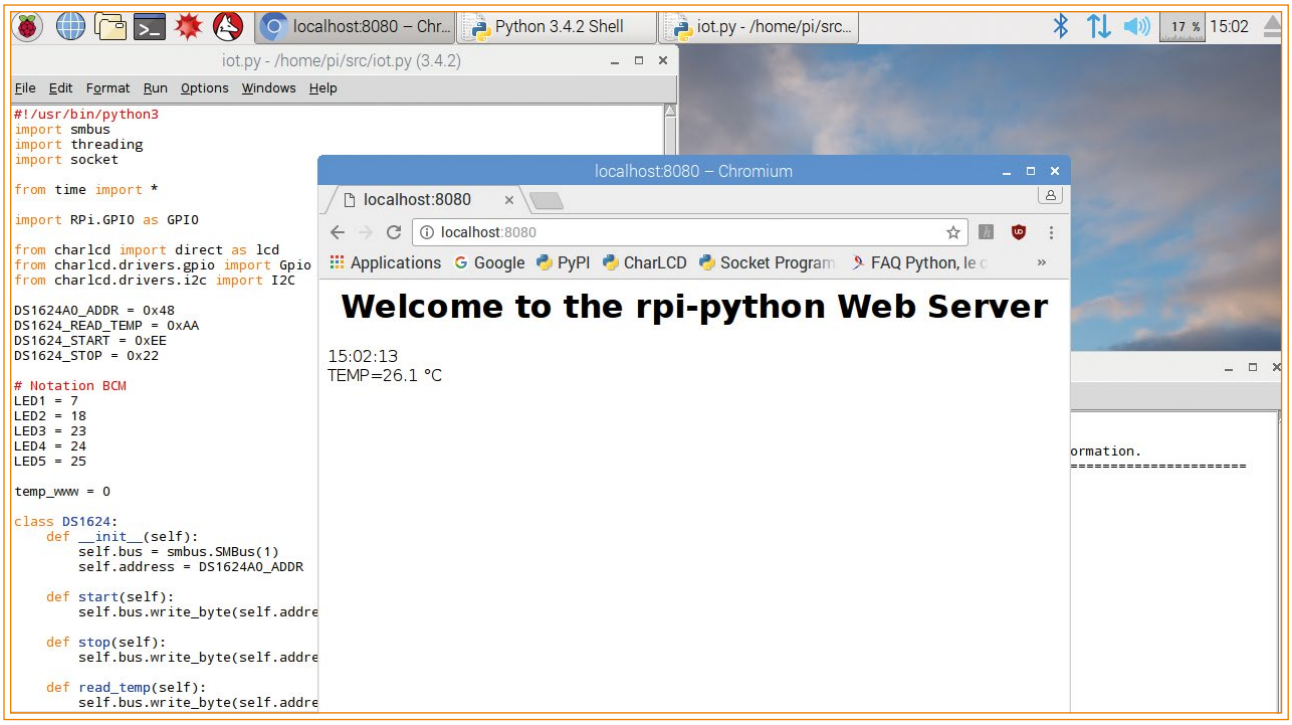


Fig. 4 : Interrogation à distance par le Web de l'objet connecté.

CONCLUSION

Nous avons pu voir la mise en œuvre du langage Python sur un objet connecté réalisé autour d'une carte Raspberry Pi 3 B et d'une carte d'E/S spécifique fournissant divers périphériques. La carte facilement réalisable **[1]** peut être conçue par tout amateur s'il se rapproche d'un « fablab ».

Le langage Python est un langage puissant et élégant très facile à apprendre même pour un débutant (une pensée pour le **BASIC** des années 80...).

Nous pouvons apprécier la carte Raspberry Pi pour un prototypage matériel rapide. Elle est faite pour cela. Mais nous pouvons aussi apprécier le langage Python pour un prototypage logiciel bien plus rapide encore. Le temps de développement de l'application IoT est bien plus réduit qu'avec le langage Java ou le langage **C**. De plus les bibliothèques Python offertes par la communauté sont innombrables notamment celles pour la carte Raspberry Pi, ce qui évite de réinventer la roue !

Le langage Python reste néanmoins un langage interprété, cela peut être sa limite par rapport à un langage exécuté comme le langage C surtout si l'on a des contraintes temporelles. Mais sa force est ailleurs notamment avec la possibilité de sortir très rapidement un prototype *« proof of concept »* d'un objet connecté par exemple. Tout est là affaire d'imagination... ■

RÉFÉRENCES

[1] « IoT : conception d'un objet connecté Java ME », Open Silicium n°19, p 21-27, juillet 2016

[2] « IoT : programmation d'un objet connecté Java ME », Open Silicium n°20, p 56-65, octobre 2016

[3] Ressources de l'article : http://kadionik.vvv.enseirb-matmeca.fr/pub/rpi-python/

[4] Distribution Raspbian : https://www.raspberrypi.org/downloads/raspbian/

[5] Guide d'installation de Raspbian : https://www.raspberrypi.org/documentation/installation/installing-images/linux.md

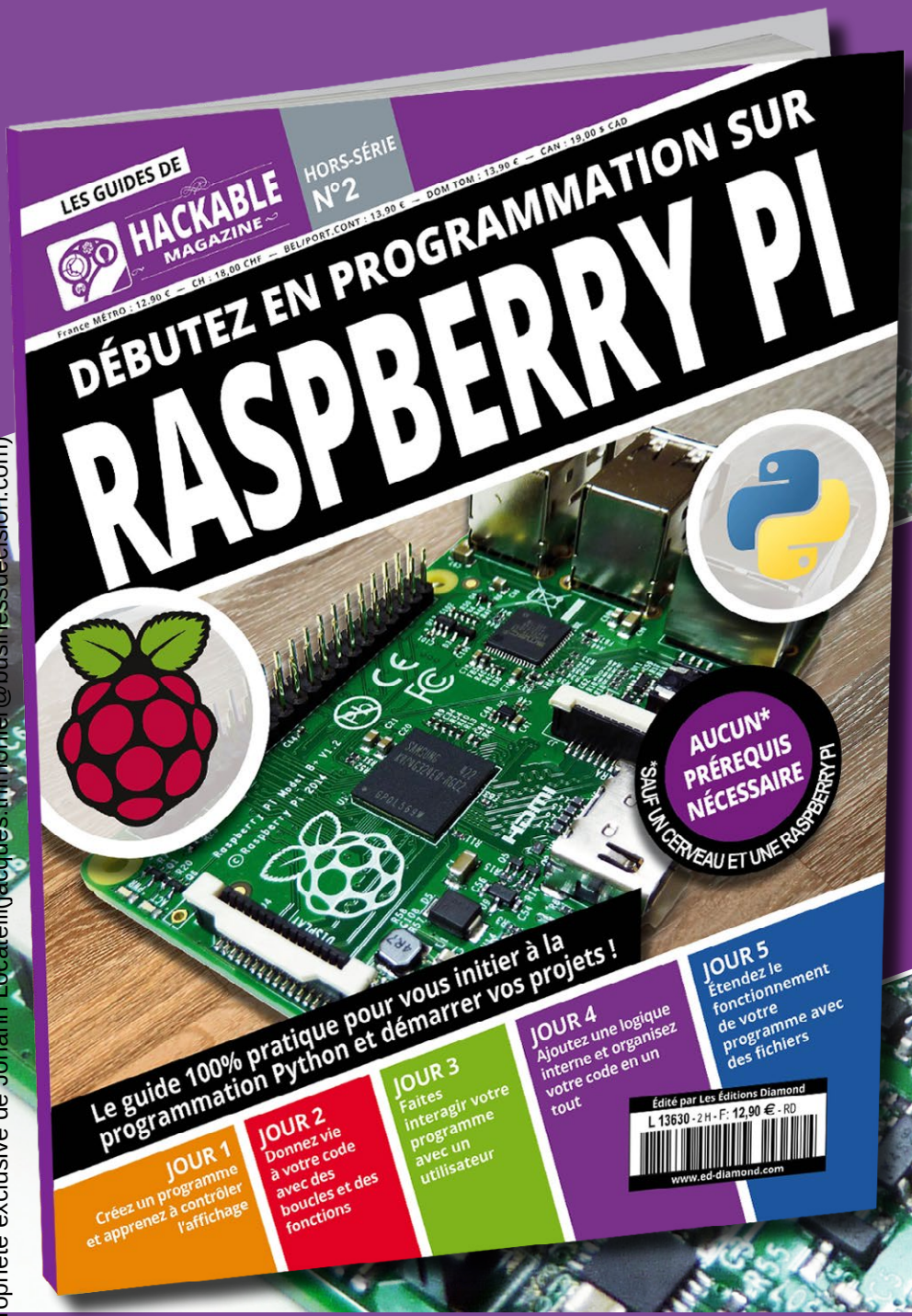
[6] GALODE Alexandre, « Raspberry Pi. Python et le port GPIO » : http://deusyss.developpez.com/tutoriels/RaspberryPi/PythonEtLeGpio/

[7] Paquetages pip pour la carte Raspberry Pi : https://pypi.python.org/pypi?action=search&term=raspberry&submit=search

[8] Bibliothèque Python CharLCD : https://pypi.python.org/pypi/CharLCD/

ACTUELLEMENT DISPONIBLE

HACKABLE HORS-SÉRIE N°2 !



DÉBUTEZ EN PROGRAMMATION SUR RASPBERRY PI !

NE LE MANQUEZ PAS

CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :

http://www.ed-diamond.com

PROGRAMMATION FONCTIONNELLE EN C++

SÉBASTIEN CHAZALLET
[Développeur logiciels libres]

MOTS-CLÉS : C++, PROGRAMMATION FONCTIONNELLE, FERMETURE, FONCTION ANONYME, ITÉRATEUR

Si C est le langage de référence de la programmation impérative, C++ est celui du paradigme objet, et on n'insistera jamais assez sur sa qualité. Pour autant, il y a plus à découvrir dans ce langage, c'est pourquoi on se propose de décrire ses possibilités en programmation fonctionnelle.

Le paradigme fonctionnel ne doit pas se voir comme concurrent aux autres, mais plutôt comme complémentaire. Il est conçu pour rendre le développement de certains cas concrets d'application plus simple, court et performant.

Les cas d'utilisations en questions impliquent le travail sur une séquence de données. L'état d'esprit est qu'au lieu de penser à l'algorithme à écrire, il faut plutôt penser à l'action à mener sur une des données. Puis, on utilisera la méthode appropriée pour décliner cette action sur toutes les données.

1. INTRODUCTION

1.1 Concepts

Pour entrer progressivement dans la programmation fonctionnelle, il faut partir d'un cas concret simple et commencer par l'écrire de manière impérative :

```
#include <iostream>
#include <array>

void affiche_
tableau(std::array<int,4>
seq) {
    auto first = seq.begin();
    auto last = seq.end();

    cout << "La séquence
contient:";
    while (first != last) {
        std::cout << ' ' <<
(*first);
        ++ first;
    }
    std::cout << ".\n";
}

int main() {
    std::array<int,4> seq =
{2, 4, 6, 8};
    affiche_tableau(seq);

    return 0;
}
```

L'ordre des opérations est très classique : on déclare les itérateurs, on commence

à écrire la phrase et l'on parcourt la séquence pour écrire chaque nombre. Enfin, on termine la phrase.

Voici ce que cela donnerait en utilisant le paradigme fonctionnel :

```
#include <iostream>
#include <algorithm>
#include <array>

void affiche_nombre (int i) {
    std::cout << ' ' << i;
}

void affiche_tableau(std::array<int,4> seq)
{
    auto first = seq.begin();
    auto last = seq.end();

    cout << "La séquence contient:";
    for_each(first, last, affiche_nombre);
    std::cout << ".\n";
}

int main() {
    std::array<int,4> seq = {2, 4, 6, 8};
    affiche_tableau(seq);

    return 0;
}
```

Plusieurs différences peuvent être constatées, même si ce n'est pas flagrant dans cet exemple très simple : la longueur du code, sa lisibilité et enfin sa performance. Manipuler les itérateurs soi-même est en général légèrement plus lent.

De plus, il est possible d'améliorer légèrement cette fonction. D'une part en ne déclarant pas les itérateurs en avance (ce qui a été fait par mimétisme avec l'exemple précédent, mais n'est pas réellement utile), mais aussi en utilisant une fonction anonyme :

```
#include <iostream>
#include <algorithm>
#include <array>

void affiche_tableau(std::array<int,4> seq) {
    cout << "La séquence contient:";
    for_each(seq.begin(), seq.end(), [](int i)
{std::cout << ' ' << i;});
    std::cout << ".\n";
}

int main() {
    std::array<int,4> seq = {2, 4, 6, 8};
    affiche_tableau(seq);

    return 0;
}
```

La fonction **affiche_tableau** a ainsi été réduite à 3 lignes seulement, restant cependant assez lisible.

Cependant, vous n'êtes peut-être pas familier avec les fonctions anonymes, on va donc les présenter en détail.

1.2 Fonctions anonymes

Aussi appelée fonction lambda, une fonction anonyme est simplement une fonction qui n'a pas de nom. La conséquence est qu'elle ne peut être appelée ailleurs qu'à l'endroit où elle est déclarée. Elle peut être aussi complexe que souhaité et avoir autant de lignes que nécessaire. Pour la présentation, on peut tout garder sur une seule ligne, s'il n'y a qu'une seule instruction :

```
for_each(seq.begin(), seq.end(), [](int i)
{std::cout << ' ' << i;});
```

Ou encore sur plusieurs lignes (si plusieurs instructions), en procédant ainsi :

```
for_each(seq.begin(), seq.end(), [](int i){
    std::cout << ' ';
    std::cout << i;
});
```

Les instructions sont, comme les fonctions classiques, situées entre des accolades et précédées par le prototype de la fonction qui ne contient que les paramètres précédés de leurs types respectifs, et positionnés entre parenthèses. Il n'y a pas de nom de fonction, bien entendu, mais il n'y a pas non plus de type de retour (il est en réalité **auto**, de manière implicite).

Notons cependant qu'il est possible de préciser ce type à titre d'information (ceci n'est pas contraignant au sens classique d'une déclaration de fonction qui, comme le langage est statique, sera vérifiée à la compilation) :

```
for_each(seq.begin(), seq.end(), [](int i)
-> void {std::cout << ' ' << i;});
```

L'ironie est qu'une fonction anonyme peut tout de même être nommée, tout simplement en l'assignant à une variable, comme on le ferait de n'importe quel résultat d'opération :

```
std::function<void (int)> action = for_each(seq.
begin(), seq.end(), [](int i) -> void
{std::cout << ' ' << i;});
```

Le type de la variable est une fonction, elle ne renvoie rien et prend un seul paramètre qui est entier. Son type peut avantageusement être remplacé par **auto**.

```
auto action = for_each(seq.begin(), seq.end(),
[](int i) -> void {std::cout << ' ' << i;});
```

La dernière chose, mais non la moindre, à retenir est la présence de ces crochets. Ceux-ci font appel à une autre notion : celle de fermeture (*closure*).

1.3 Fermetures

Une fermeture est une fonction qui peut capturer les variables à l'extérieur d'elle-même. Mais ceci ne se fait pas de manière automatique, c'est le développeur qui contrôle les variables extérieures auxquelles il est possible d'accéder.

La présence des crochets, sans rien à l'intérieur, précise qu'aucune variable extérieure n'est capturée. Une variable peut être capturée par valeur ou par référence, en utilisant la syntaxe usuelle. Ainsi, **[a, &b]** signifie que la variable **a** est capturée par valeur et la variable **b** par référence. C'est la seule chose à retenir.

La présente du signe **=** va permettre de déclarer qu'elles sont toutes capturées par valeur, ce qui est toujours une mauvaise idée, tandis que la présence de **&** permet de déclarer qu'elles le sont par référence, ce qui reste néanmoins une très mauvaise idée.

Une écriture telle que **[&, a]** signifie que toutes les variables sont capturées par référence, sauf **a** qui l'est par valeur. À l'inverse, **[=, &b]** signifie que toutes les variables sont capturées par valeur, sauf **b** qui l'est par référence. Et au risque de paraître insistant, c'est permis par le langage, mais cela reste une mauvaise idée.

Voici quelques exemples très classiques qui vous permettront de mieux saisir l'intérêt ou non de tout ceci :

```
int resultat = 0;
std::function<void (int, int)> ajout = [&resultat]
(int d, int u) {
    int s = 10 * d + u;
    cout << "Ajout de : " << d << " * 10 + " << u <<
" (soit " << s << ") dans " << resultat << ".\n";
    resultat += s;
};
auto affiche = [=]() { cout << "Le resultat est " <<
resultat << ".\n"; }
int main() {
    affiche();       // Affiche "Le resultat est 0."
    somme(4, 2);     // Ajout de : 4 * 10 + 2 (soit
42) dans 0."
    affiche();       // Affiche "Le resultat est 42."
    somme(36, 18);   // Ajout de : 36 * 10 + 18 (soit
378) dans 42."
    affiche();       // Affiche "Le resultat est 420."
    return 0;
}
```

2. FONCTIONS USUELLES

2.1 Logique

Maintenant que l'on a vu le principe général de la programmation fonctionnelle, on va partir sur un exemple concret et simple : écrire une fonction qui va me dire si tous les nombres d'une séquence sont pairs.

Avant de commencer, quelques précisions sur les entêtes dont nous aurons besoin :

```
#include <iostream>      // std::cout
#include <algorithm>     // std::all_of ,
std::any_of , std::none_of ,
#include <array>         // std::array
```

Dans un premier temps, nous allons déclarer une séquence de données (ici, la séquence est un tableau), ainsi que les itérateurs qui vont bien :

```
std::array<int,4> seq = {2, 4, 6, 8}
auto first = seq.begin()
auto last = seq.end()
```

Afin de comparer, une fois encore, le paradigme fonctionnel à ce que l'on connaît déjà, voici ce que l'on pourrait faire en utilisant la programmation impérative :

```
while (first != last) {
    if (*first % 2 == 1) std::cout << "Faux";
    ++ first
}
std::cout << "Vrai";
```

Voici ce que cela donnerait en utilisant le paradigme fonctionnel. Le principe est de réfléchir au niveau de la donnée, et d'écrire une fonction qui va dire si la donnée est valide ou non :

```
bool estPair (int i) { return (i % 2) ==
1; }
```

Puis, on peut appliquer cette fonction à l'ensemble de la séquence, ici directement dans une condition :

```
if (all_of(first, last, estPair))
    std::cout << "Faux";
else
    std::cout << "Vrai";
```

Dans le cas d'une fonction complexe et/ou utilisée plusieurs fois, cette façon de procéder est idéale. S'il s'agit comme ici d'une fonction simpliste, on peut se permettre cette écriture :

```
if (all_of(first, last, [](int i){return
(i%2)==1;}))
    std::cout << "Faux";
else
    std::cout << "Vrai";
```

Cette écriture présente l'avantage d'être plus dense, d'éviter de déclarer la fonction **estPair**, mais elle peut également être ressentie comme moins lisible.

Comme vous l'aurez deviné, ma fonction **all_of** va permettre de vérifier que la fonction appliquée à chaque élément va tout le temps renvoyer **True**. Si un seul élément renvoie **False**, alors le résultat global est **False**.

On peut également citer **any_of** qui va renvoyer **True** si au moins un élément dans la liste renvoie **True** ou **none_of** qui renverra **True** si aucun élément dans la liste ne renvoie **True**, donc si tous renvoient **False**.

Dans le même état d'esprit, on peut aussi présenter **is_partitioned** qui renverra vrai si et seulement si toutes les valeurs pour lesquelles la valeur renverra vrai sont placées avant celles renvoyant faux. Voici un exemple :

```
#include <iostream>      // std::cout
#include <algorithm>     // std::is_partitioned
#include <array>         // std::array

bool negatif (int i) { return i < 0; }

void est_partitionne(array<int> nombres) {
    std::cout << "nombres: ";
    for (int& x:nombres) std::cout << ' ' << x;

    if ( std::is_partitioned(nombres.
begin(),nombres.end(),negatif) )
        std::cout << " : partitionné\n";
    else
        std::cout << " : non partitionné\n";
}

int main () {
    est_partitionne({-2,-1,0,1,2}); // Affiche
-2 -1 0 1 2 : partitionné
   est_partitionne({-2,1,0,-1,2}); // Affiche
-2 1 0 -1 2 : non partitionné

    return 0;
}
```

Jusqu'à présent, nous avons utilisé la programmation fonctionnelle pour lire des séquences. La présentation de cette dernière fonction va nous emmener tout naturellement à voir comment on peut aussi les modifier.

2.2 Paradigmes fonctionnels et itérateurs

Comme nous venons de le voir, le paradigme fonctionnel fait une utilisation intensive des itérateurs. Ils sont passés, tout comme la fonction prédicat, en paramètres. Mais ils peuvent également être renvoyés comme le résultat d'une fonction.

Lorsque l'on a une séquence partitionnée, c'est comme si l'on avait deux séquences distinctes mises l'une à la suite de l'autre, la première vérifiant toujours le prédicat et la seconde ne le vérifiant jamais. il est assez simple de positionner un itérateur à l'endroit de la rupture et ainsi de ne parcourir que la première ou la seconde partie. Ceci se fait naturellement ainsi :

```
#include <iostream>     // std::cout
#include <algorithm>    // std::partition,
std::stable_partition
#include <array>        // std::array

bool negatif (int i) { return i < 0; }

void parcourir(array<int> nombres) {
    std::cout << "Nombres: ";
    for (int& x:nombres) std::cout << ' ' << x;
    std::cout << ".\n";

    std::array<int>::iterator rupture;
    rupture = std::partition(nombres.
begin(),nombres.end(),negatif)

    std::cout << "Avant la rupture : ";
    for (std::array<int>::iterator it=nombres.
begin(); it!=rupture; it++)
        std::cout << ' ' << *it;
    cout << ".\n";

    std::cout << "Après la rupture : ";
    for (std::array<int>::iterator it=rupture;
it!=nombres.end(); it++)
        std::cout << ' ' << *it;
    cout << ".\n";
}

int main () {
    parcourir({-2,-1,0,1,2});

    return 0;
}
```

Ce qui va afficher :

```
Nombres: -2 -1 0 1 2
Avant la rupture : -2 -1 0 1 2
Après la rupture : 0 1 2
```

L'idée est donc que cette fonction va nous donner l'itérateur nécessaire pour terminer d'itérer sur les éléments vrais ou pour commencer à itérer sur ceux dont le prédicat n'est pas validé.

En réalité, il y a beaucoup plus que cela, parce que la fonction ne se contente pas de chercher le point de rupture. Elle s'assure que la séquence est partitionnée en faisant la répartition elle-même si nécessaire (en échangeant les éléments mal placés elle-même si nécessaire).

Le code précédent marche donc avec une séquence non partitionnée :

```
int main () {
    parcourir({-2,1,0,-1,2});

    return 0;
}
```

Il est à noter que la séquence est modifiée durant le processus. On peut également préciser l'existence de **stable_partition** qui produit la même chose, à l'exception du fait qu'elle garantit que, dans chaque partition, les éléments sont dans le même ordre que dans la séquence d'origine.

2.3 Min et max

Un des cas d'utilisation où la programmation fonctionnelle se démarque de la programmation impérative est la recherche du min ou du max. Voici un exemple concret :

```
#include <iostream>     // std::cout
#include <algorithm>    // std::min_element ,
std::max_element , std::minmax_element
#include <array>        // std::array

void affiche_minimum(array<int> nombres) {
    std::cout << "Nombres: ";
    for (int& x:nombres) std::cout << ' ' << x;
    std::cout << ".\n";

    std::cout << "min=" << *std::min
element(nombres.begin(),nombres.end()) << ".\n";
}

int main () {
    affiche_minimum({-2,-1,0,1,2});

    return 0;
}
```

Cette méthode permet de trouver le plus petit nombre d'une séquence en une seule ligne, de manière très performante, sans avoir besoin d'écrire une boucle en la parcourant soi-même.

Bien évidemment, ceci fonctionne exactement de la même manière pour **max_element**, mais on peut préciser ici qu'il existe également la fonction **minmax_element** qui permet de renvoyer à la fois l'élément le plus petit ainsi que le plus grand.

```
#include <iostream>     // std::cout
#include <algorithm>    // std::min_element ,
std::max_element , std::minmax_element
#include <array>        // std::array

void affiche_minimum_et_maximum(array<int>
nombres) {
    std::cout << "Nombres: ";
   for (int& x:nombres) std::cout << ' ' << x;
    std::cout << ".\n";
    auto minmax = std::minmax_element(nombres.
begin(),nombres.end());
    std::cout << "min=" << *result.first << <<
", " << *result.second ".\n";
 }

int main () {
    affiche_minimum_et_maximum({-2,-1,0,1,2});

    return 0;
}
```

Cette méthode renvoie un objet de type pair qui contient les deux itérateurs vers les deux valeurs que l'on souhaite retrouver. Bien entendu, il n'est à utiliser que lorsque l'on a besoin des deux et il est à préférer à l'utilisation des deux fonctions **min_element** et **max_element**, car cela reviendrait à parcourir deux fois la liste.

3. OPÉRATIONS USUELLES

3.1 Transformer

L'opération de transformation consiste à modifier tous les éléments d'une liste de la même manière : par exemple, mettre tous les éléments au carré :

```
#include <iostream>     // std::cout
#include <algorithm>    // std::partition,
std::stable_partition
#include <array>        // std::array

bool carre (int i) { return i * i; }

void transformer(array<int> nombres) {
   array<int> resultat;
   resultat.resize(nombres.size());

   std::cout << "Nombres: ";
   for (int& x:nombres) std::cout << ' ' << x;
   std::cout << ".\n";

   std::transform(nombres.begin(),nombres.
end(),resultat.begin(),negatif);

   std::cout << "Résultat: ";
   for (int& x:resultat) std::cout << ' ' << x;
   std::cout << ".\n";
}

int main () {
   transformer({-2,-1,0,1,2});  // Résultat: 4
1 0 1 4

   return 0;
}
```

Cette fonction est assimilable à la fonction **map** en Python, par exemple.

3.2 Remplacer

Le remplacement permet de remplacer par une valeur déterminée, tous les éléments d'une séquence qui remplissent le prédicat :

```
#include <iostream>     // std::cout
#include <algorithm>    // std::partition,
std::stable_partition
#include <array>        // std::array
void remplacer(array<int> nombres) {
    std::cout << "Nombres: ";
    for (int& x:nombres) std::cout << ' ' << x;
    std::cout << ".\n";

    std::replace_if(nombres.begin(), nombres.
end(),impair, 0);

    std::cout << "Résultat: ";
    for (int& x:nombres) std::cout << ' ' << x;
    std::cout << ".\n";
}

int main () {
    remplacer({-2,-1,0,1,2});
// Renvoie -2 0 0 0 2

    return 0;
  }
```

Contrairement à la fonction de transformation, cette fonction modifie l'objet initial au lieu d'aller écrire une nouvelle séquence, comme c'est le cas de la transformation.

Laisser intact la séquence d'origine pour en créer une nouvelle est possible en utilisant **replace_copy_if**.

Comme pour la transformation, il faudra glisser en troisième argument l'itérateur vers l'objet résultat. Si on reprend l'exemple plus haut, on aurait :

```
std::replace_copy_if(nombres.begin(),nombres.
end(),resultat.begin(),impair, 0);
```

Il est également à noter que la taille du résultat est identique à celle de départ.

3.3 Filtrer

Le filtrage permet de ne garder dans une liste que les éléments pour lesquels la fonction **filtre** renvoie vrai. Autrement dit, les éléments qui ne passent pas le prédicat sont supprimés. La conséquence directe est que la séquence peut contenir moins d'éléments qu'initialement.

Pour cette raison, elle renvoie un itérateur, lequel marque la nouvelle fin de la séquence :

```
#include <iostream>     // std::cout
#include <algorithm>    // std::partition,
std::stable_partition
#include <array>        // std::array

bool pair (int i) { return i % 2 = 0; }

void filtrer(array<int> nombres) {
    std::cout << "Nombres: ";
    for (int& x:nombres) std::cout << ' ' << x;
    std::cout << ".\n";
```

```
    std::array<int>::iterator nouvelle_fin;
    nouvelle_fin = std::remove_if(nombres.
begin(),nombres.end(),pair);

    std::cout << "Résultat : ";
    for (std::array<int>::iterator it=nombres.
begin(); it!=nouvelle_fin; it++)
        std::cout << ' ' << *it;
    cout << ".\n";
}

int main () {
    filtrer({-2,-1,0,1,2});  // Résultat : -2 0
2 (seulement trois éléments)

    return 0;
}
```

Cette fonction est assimilable à la fonction **filter** que l'on peut trouver en Python, par exemple.

3.4 Mélanger

Mélanger une liste, c'est-à-dire disposer ces éléments à des emplacements aléatoires est relativement aisé :

```
#include <iostream>     // std::cout
#include <algorithm>    // std::partition,
std::stable_partition
#include <array>        // std::array

bool impair (int i) { return i % 2 = 1; }

void melanger(array<int> nombres) {
    std::cout << "Nombres: ";
    for (int& x:nombres) std::cout << ' ' << x;
    std::cout << ".\n";

    std::random_shuffle(nombres.begin(),nombres.end());

    std::cout << "Résultat: ";
    for (int& x:nombres) std::cout << ' ' << x;
    std::cout << ".\n";
}

int main () {
    melanger({-2,-1,0,1,2});  // Renvoie -2 1 2 -1 0,
par exemple

    return 0;
}
```

Le fait que cette méthode fasse partie du paradigme fonctionnel est dû au fait qu'elle peut prendre optionnellement une fonction permettant de gérer l'aléatoire :

```
int ma_methode (int i) { return
std::rand()%i;}
std::random_shuffle(nombres.begin(),nombres.
end(), ma_methode);
```

3.5 Trier

Pour terminer, après avoir mélangé notre séquence, nous allons la trier :

```
#include <iostream>     // std::cout
#include <algorithm>    // std::partition,
std::stable_partition
#include <array>        // std::array

bool impair (int i) { return i % 2 = 1; }

void melanger(array<int> nombres) {
    std::cout << "Nombres: ";
    for (int& x:nombres) std::cout << ' ' << x;
    std::cout << ".\n";

    std::sort(nombres.begin(),nombres.end());

    std::cout << "Résultat: ";
    for (int& x:nombres) std::cout << ' ' << x;
    std::cout << ".\n";
}

int main () {
    melanger({-1,1,0,2,-2});  // Renvoie -2 -1
0 1 2

    return 0;
}
```

Et nous préciserons que, là encore, il est possible de donner une fonction permettant de décrire la méthode pour comparer des objets deux à deux. De même, il existe **stable_sort** qui permet de s'assurer que des éléments qui sont comparables (égaux au sens de la fonction de comparaison) seront disposés entre eux dans le même ordre que dans la séquence de départ.

On notera également qu'il existe **partial_sort** qui trie les X premiers éléments de la séquence et **partial_sort_copy** qui permet de prendre les X premiers éléments les plus petits (au sens de la fonction de comparaison) et de les mettre dans la séquence de résultat.

Enfin, il existe **is_sorted** qui permet de savoir si la séquence est triée ou non et **is_sorted_until** qui permet de récupérer l'itérateur à partir duquel la séquence n'est plus ordonnée.

CONCLUSION

La programmation fonctionnelle est un aspect relativement séduisant de C++, mais qui reste encore un peu méconnu. On notera cependant que c'est un axe de développement important du langage, puisque de nouvelles fonctionnalités dans ce domaine viennent enrichir le langage. On notera, par exemple, l'introduction d'une des fonctions les plus connues du paradigme fonctionnel dans C++17, à savoir **reduce**. Il y est également introduit **accumulate** ou encore **transform_reduce**. ■

ikoula
HÉBERGEUR CLOUD

PRÉSENTE

CLOUD**IKOULA**ONE

Le succès est votre prochaine destination

MIAMI SINGAPOUR PARIS
AMSTERDAM FRANCFORT ___

*CLOUD**IKOULA**ONE* est une solution de Cloud public, privé et hybride qui vous permet de déployer **en 1 clic et en moins de 30 secondes** des machines virtuelles à travers le monde sur des infrastructures SSD haute performance.

www.ikoula.com sales@ikoula.com 01 84 01 02 50

ikoula
HÉBERGEUR CLOUD

NOM DE DOMAINE | HÉBERGEMENT WEB | SERVEUR VPS | SERVEUR DÉDIÉ | CLOUD PUBLIC | MESSAGERIE | STOCKAGE | CERTIFICATS SSL

« PAIE TON PATCH !™ » : WEBOOB

FRANÇOIS REVOL
[Serial Patcheur, contributeur de plusieurs modules (europarl, gdcvault, vimeo) et de leurs icônes (oui, j'ai honte)]

MOTS-CLÉS : PATCH, CONTRIBUER, UPSTREAM, SCRAPING, PYTHON

Combien de fois vous êtes-vous dit « Pourquoi c'est pas corrigé ça ? » ou « faudrait patcher ce truc » sans oser le faire ? Voici une occasion !

Web Outside of Browsers **[1]** est un ensemble d'outils modulaires en ligne de commandes écrits en **Python**, ainsi que quelques applications graphiques **Qt**. Son but est de pouvoir utiliser des sites web comme l'on utilise d'autres ressources sous Unix, à l'aide d'outils simples composables et scriptables. Parmi les outils de *scraping* existants, il s'agit probablement du plus complet, et décrire ses possibilités nécessiterait plusieurs articles. Ses compétences vont de la récupération de vidéos de sites web (et non-web en **Flash**) à l'émission de virements bancaires, en passant par l'édition de tickets dans un bugtracker, ou la recherche d'emploi. C'est cette dernière fonction que nous testerons, en contribuant au support du site **LinuxJobs.fr [2]**.

Sur le canal IRC **[3]** de **Weboob**, il est de tradition lorsque quelqu'un râle sur une fonctionnalité manquante de dire « PTP », c'est-à-dire « Paie Ton Patch », comme invitation à contribuer, certes pas toujours efficace. L'expression, consacrant le patch comme monnaie officielle de la *do-ocratie*, est donc toute désignée comme titre.

1. ARCHITECTURE

Weboob est constitué de nombreux modules, chacun implémentant une ou plusieurs *capacités* (*Capability*) telles que `CapVideo`, `CapMessages`, `CapJob`, etc. et de nombreuses applications utilisant ces différentes capacités. Un module est composé de plusieurs fichiers, chacun déclarant une ou plusieurs classes :

- `__init__.py` : exporte la classe de base `Module` ;
- `module.py` : définit une sous-classe de `Module` indiquant son nom, ses capacités, etc. ;
- `browser.py` : dérive la classe `Browser` qui implémente l'interaction avec le site web ;
- `pages.py` : implémente plusieurs classes décrivant chaque type de page que le site retourne et comment y récupérer les informations désirées ;
- `test.py` : le truc que tout le monde oublie ;

- **favicon.png** : une icône PNG de 64x64 avec transparence. Les icônes dans Weboob sont volontairement caricaturales pour contourner un éventuel problème de droit des marques. Donc lâchez-vous !

Enfin, le *framework* propose tout ce qu'il faut pour gérer les connexions HTTP(S), et parser le HTML ou le JSON. Quelques modules datent encore des « heures les plus sombres de l'histoire de Weboob », avant l'avènement du **Browser2**. Un module important **weboob.deprecated.browser** est une invitation à contribuer à une mise à jour.

Le tout est documenté sur le site de développement **[4]**. Nous utiliserons l'application **handjoob** pour tester notre module.

2. CODE SOURCE ET INSTALLATION

Weboob documente **[5]** plusieurs méthodes d'installation, suivant votre distribution et vos besoins. Pour tester notre code plus simplement, on peut installer en mode développeur, ou même lancer directement les commandes sans installation à l'aide d'un script.

Weboob a quelques dépendances, sur **Debian** par exemple :

```
# apt install python-html2text python-prettytable python-dateutil python-lxml python-mechanize python-yaml python-cssselect python-requests
```

Le code source officiel est disponible sur un serveur Gitlab dédié **[6]**, où les principaux contributeurs ont aussi leur propre dépôt. Récupérons le dépôt de développement, dans **~/src/** par exemple :

```
$ git clone https://git.weboob.org/weboob/devel.git weboob
```

2.1 Installation « développeur »

Un script est proposé pour installer sans toucher au système (passez **--deps** au script si vous n'avez pas trouvé toutes les dépendances dans votre distro, il les installera par **pip**) :

```
$ mkdir ~/bin
$ tools/local_install.sh $HOME/bin
$ echo 'export PATH=$PATH:$HOME/bin' >> ~/.bashrc
```

Ensuite il nous faut déclarer notre dépôt local comme source de modules, puis forcer une mise à jour :

```
$ echo "file://$HOME/src/weboob/modules" >> ~/.config/weboob/sources.list
$ weboob-config update
```

2.2 Pas d'install

Le script **local_run.sh** dans le dépôt permet d'éviter l'installation :

```
$ tools/local_run.sh handjoob search python
```

Si vous optez pour cette méthode, il vous suffira donc de lancer les commandes par ce script.

3. GÉNÉRATION D'UN SQUELETTE DE MODULE

Vous avez probablement maintenant déjà fouillé le code source et vous vous apprêtez à copier un module existant vers **modules/linuxjobs** suivi d'un coup de **sed**. Je sais, on l'a tous fait, mais pour une fois on vous propose une méthode plus propre **[7]**. Depuis la racine du dépôt, créons une branche, puis invoquons l'outil magique :

```
$ git checkout -b linuxjobs
$ tools/boilerplate.py cap "linux jobs" CapJob
Created ...modules/linuxjobs/__init__.py
...
```

Si vous avez déjà configuré git proprement, il utilisera même vos nom et email pour l'attribution. Forçons ensuite une mise à jour du cache des modules, et vérifions qu'il trouve bien notre bébé :

```
$ git checkout -b linuxjobs
$ weboob-config update
...
$ weboob-config info linuxjobs
...
| License          | AGPLv3+
| Description      | linuxjobs website
| Capabilities     | CapJob
...
```

4. CHOISIR LA SOURCE DES DONNÉES

Avant de foncer tête baissée, il convient d'étudier un peu le site : comment se fait une recherche ? Les données sont-elles bien balisées ? Y a-t-il une source plus fiable et plus

stable que le code **HTML** de la page, via des requêtes AJAX récupérant du **XML** ou du **JSON** ? Le *scraping* reste un dernier recours fragile, sensible au moindre changement des pages.

Ici le site est récent, et les annonces sont présentées très simplement, il faudra donc valider proprement les données. Si le site ne fait pas de requêtes AJAX, il existe par contre des flux **RSS** pour chaque catégorie, où la description est en **Markdown** au lieu de HTML. Par contre, rechercher une annonce nécessiterait de savoir dans quel flux chercher, donc *parser* la catégorie dans la page HTML de toute façon. Et puis nous voulons un exemple bien *gore* pour l'article, donc restons sur le *hache-thé-aime-elle*.

5. REMPLISSONS LES BLANCS

Nous allons maintenant remplir naïvement le squelette généré avant de réaliser un premier test, en nous appuyant sur la documentation de **CapJob** et d'autres modules comme **adecco** (mais il utilise l'ancienne API) ou **popolemploi** (si, le module existe !). L'exemple reste simple, mais parfois les interactions entre les classes dans certains modules complexes peuvent nécessiter une aspirine. N'hésitez pas à demander aux gens bons sur IRC !

5.1 module.py

La classe **LinuxJobsModule** définit trois méthodes non implémentées que nous devons donc remplir :

- **advanced_search_job**, qui effectue une recherche sur des critères configurés au préalable. Le site visé ne proposant pas de recherche multicritère il nous faudrait filtrer les résultats, et gérer aussi la configuration du module. Nous allons donc laisser cette méthode de côté. Mais vous pourrez proposer un patch !
- **get_job_advert** doit retourner un objet décrivant l'annonce correspondant à l'identifiant unique en paramètre. Le site semble utiliser un nombre croissant monotone dans l'URL pour cet usage. Nous passons simplement la requête au **browser** comme font les autres modules :

```
    def get_job_advert(self, _id, advert=None):
        return self.browser.get_job_advert(_id, advert)
```

- **search_job** doit itérer sur une recherche par mots-clefs. Ici aussi, c'est le **browser** qui se chargera de la requête :

```
    def search_job(self, pattern=None):
        for job_advert in self.browser.search_job(pattern):
            yield job_advert
```

5.2 browser.py

La classe **LinuxJobsBrowser** est chargée de l'interaction avec le site, et utilise différentes Pages en fonction de l'URL demandée.

Plutôt que **Page1** et **Page2**, nous utiliserons **SearchPage** et **AdvertPage** en indiquant les *regexp* idoines, l'une récupérant l'ID de l'annonce, l'autre indiquant où spécifier la chaîne à rechercher. Corrigeons au passage l'URL de base, par défaut en HTTP et .com :

```
from .pages import SearchPage, AdvertsPage
import urllib

class LinuxJobsBrowser(PagesBrowser):
    BASEURL = 'https://www.linuxjobs.fr'

    advert_page = URL('/jobs/(?P<id>.+)', AdvertPage)
    search_page = URL('/search/(?P<job>)', SearchPage)
```

Nous aurions également pu spécifier un **PROFILE** de navigateur (Weboob se fait passer pour Firefox par défaut), ce qui est utile entre autres pour récupérer des vidéos HLS à la place du Flash sur un site codé avec les pieds. Il est également possible de spécifier un **TIMEOUT** différent des 10s par défaut.

En lieu et place du **get_stuff** proposé, nous ajouterons les méthodes que nous appelons depuis **LinuxJobsModule** :

```
    def get_job_advert(self, _id, advert):
        self.advert_page.go(id=_id)

        assert self.advert_page.is_here()
        return self.page.get_job_advert(obj=advert)

    def search_job(self, pattern=None):
        if pattern is None:
            return []
        self.search_page.go(job=urllib.quote_plus(pattern.
encode('utf-8')))

        assert self.search_page.is_here()
        return self.page.iter_job_adverts()
```

Dans les deux cas, nous appelons la méthode **go()** pour envoyer la requête après avoir préparé les paramètres. Si l'**id** n'a pas vraiment besoin de traitement, la chaîne de recherche doit être encodée en UTF-8 et les espaces remplacés par des signes plus.

Comme suggéré dans **get_stuff**, un **assert** s'assure que nous sommes bien passés sur la bonne page, dont une instance sera alors assignée à **self.page**. Puis nous appelons une méthode de la classe correspondante pour remplir l'objet décrivant l'annonce, ou itérer sur les résultats de la recherche.

5.3 pages.py

C'est le gros du boulot puisque ces classes vont rechercher les éléments utiles dans le contenu de la page. Ce sont celles que l'on doit patcher lorsqu'un webmaster a un accès de créativité. Heureusement, cette tâche s'est considérablement simplifiée depuis **Browser2**.

```
class AdvertPage(HTMLPage):
    @method
    class get_job_
advert(ItemElement):
        klass = BaseJobAdvert
```

Alors là, ça ressemble à de la magie pour qui n'est pas encore à l'aise avec les décorateurs de classe en Python. Nous déclarons une classe **AdvertPage** qui dérive de **HTMLPage**. Le décorateur **@method** indique que la classe déclarée en dessous (**get_job_advert**) sera appelée comme une méthode, et qu'elle sera au final une **BaseJobAdvert** dont nous remplirons les champs juste après. Les autres modules font ainsi, donc ça devrait fonctionner...

Contentons-nous pour l'instant de récupérer l'**id**, l'**url** (que l'on reconstruit à partir de l'**id** et de la *regexp* de l'URL) et l'intitulé du poste (**title** et **job_name** sont identiques) qui ô joie est le contenu de la balise **title** :

```
        obj_id = Env('id')
        obj_url = BrowserURL('advert_page', id=Env('id'))
        obj_title = CleanText('//title')
        obj_job_name = CleanText('//title')
```

Rassurez-vous, ça se corsera pour les autres champs. Le filtre **CleanText** permet de nettoyer le texte des tabulations et autres retours à la ligne, et d'être sûr que le résultat est bien de l'Unicode. Nous lui passons en paramètre un sélecteur XPath indiquant la balise **title**.

Passons à la recherche. Les résultats sont groupés dans un **div** par catégorie, chacun contenant des balises **a** vers les annonces. Dans le lien nous trouvons par contre des infos mieux balisées que sur la page de l'annonce elle-même avec des **span** de classe **job-title** et **job-company**...

De plus, les résultats sont retournés sur une page unique, nous n'aurons donc pas à gérer la pagination.

Ici aussi nous utiliserons un peu de magie décorative :

```
class SearchPage(HTMLPage):
  @method
  class iter_job_adverts(ListElement):
      item_xpath = '//a[@class="list-group-item "]'

      class item(ItemElement):
          klass = BaseJobAdvert

          obj_id = Regexp(Link('.'), '.*fr/jobs/(\d+)/.*')
          obj_title = CleanText('h4/span[@class="job-title"]')
          obj_society_name = CleanText('h4/span[@class="job-company"]')
```

Là encore c'est une page HTML, mais des classes existent pour gérer le JSON, CSV, XML... et même des fichiers XLS et PDF !

La magie va opérer et sélectionner tous les éléments répondant au sélecteur **item_xpath**. Au passage, on me dit que l'on peut user de **$x('//a...')** dans la console de Firefox pour tester les sélecteurs XPath en direct, pratique !

Notez l'espace dans le nom de classe du lien, elle doit être strictement égale. Une recherche dans le contenu aurait pu être demandée par **a[contains(@class, "list-group-item")]**, ou **a[has-class("list-group-item")]**, mais c'est spécifique à Weboob. Incidemment, la correspondance stricte nous filtre des liens « Voir toutes les annonces pour... » qui elles n'ont pas l'espace en bout de chaîne...

Pour chaque **item** trouvé, un objet **BaseJobAdvert** est retourné par l'itérateur avec les propriétés récupérées par les filtres déclarés ici. L'**id** numérique est retrouvé dans l'URL de l'annonce par une **Regexp**.

Sans oublier les imports nécessaires au début du fichier, pompés sur **popolemploi**.

6. ÇA MARCHE ?

Une petite mise à jour du cache des modules :

```
$ weboob-config update
```

Tentons une recherche :

```
$ handjoob -b linuxjobs search python
Warning: there is currently no configured
backend for handjoob
Do you want to configure backends? (Y/n):
```

Nous n'avons pas encore configuré de *backend*, c'est-à-dire une instance du module. Répondons-**y** pour sélectionner celui qui nous intéresse, et relançons la recherche avec l'option de **debug** et sans limiter le nombre de réponses :

```
$ handjoob -b linuxjobs --debug -n 0 search python
...
2016-10-01 04:24:38,860:DEBUG:backend.linuxjobs.
browser:1.2:browsers.py:662:internal_callback
Handle https://www.linuxjobs.fr/search/python with
SearchPage
566@linuxjobs — DevOps
 Society : OXALIDE
573@linuxjobs — H/F Ingénieur support produit
 Society : Linagora
```

Ça marche \o/ Enfin un boulot intéressant ?

```
$ handjoob -b linuxjobs info 566@linuxjobs
DevOps
url: https://www.linuxjobs.fr/jobs/566
Job name : DevOps
```

Un peu succinct ! C'est normal, nous n'avons pas encore rempli tous les champs depuis la page de l'annonce. La commande **info** accepte la notation **id@backend**, ainsi que l'**id** seule puisque l'on force le *backend* avec **-b**.

7. PLUS DE CHAMPS

C'est maintenant que la facilité d'utilisation du *framework* va s'exprimer, sur la partie la plus tordue. Ajoutons la lecture de la date (en français et en toutes lettres) à **iter_job_adverts** :

```
        obj_publication_date =
Date(CleanText('h4/span[@class="badge pull-
right"]'), parse_func=parse_french_date)
```

Nous utilisons le filtre **Date** sur le texte nettoyé du **span** choisi, en précisant **parse_french_date** importé depuis **weboob.tools.date** pour lire notre date pas très standard. Et ça marche !

Passons à la page d'annonce. Les champs importants sont dans des balises **h2**, **h4** ou même **small** dans quelques **div** imbriqués pas vraiment identifiés hormis par leur ordre d'apparition dans la page. Vous êtes dispensés de lire la documentation de XPath, voici les spoilers :

```
        obj_society_name = CleanText(
'//div[2]/div[@class="col-md-9"]/h4[1]')
        obj_publication_date =
Date(CleanText('//div[2]/div[@class="col-
md-9"]/small', replace=[(u'Ajoutée le',
'')]), parse_func=parse_french_date)
        obj_place = Regexp(CleanText(
'//div[2]/div[@class="col-md-9"]/h4[2]'),
'(.*) \(.*\)')
        obj_description = CleanHTML(
'//div[4]/div[@class="col-md-9"]')
```

La société est en effet dans le premier **h4** du lot. La date, elle, est préfixée par un « Ajouté le » sans espace à la fin... Nous précisons donc au filtre **CleanText** le remplacement à effectuer, puis nous utilisons encore une fois le *parseur* de date française. Le lieu est contenu dans le second **h4**, mais nous devons nous débarrasser de la catégorie entre parenthèses... Une **Regexp** s'en charge très bien. Quant à la description, c'est le 4ème **div** à droite, 1er **div** à gauche, on nettoie le HTML bien sûr, et c'est bon !

NOTE

Les commandes Weboob ne se contentent pas d'afficher le résultat en texte brut. L'option -f permet de spécifier un formater, pour effectuer vos propres traitements. Par exemple :

```
$ handjoob -b linuxjobs -f json -n 0
search python
[{"id": "566@linuxjobs", "publication_
date": "2016-08-22", ...
$ handjoob -b linuxjobs -f csv -n 0 search
python > jobs.csv && loffice --calc
--infilter="csv:59,34,0,1" jobs.csv
```

8. UN TEST

Histoire de montrer que l'on n'oublie pas d'écrire le **test.py** (ben oui, on cherche du boulot...) :

```
class LinuxJobsTest(BackendTest):
    MODULE = 'linuxjobs'

    def test_linuxjobs_search(self):
        l = list(self.backend.search_
job('linux'))
        assert len(l)
        advert = self.backend.get_job_
advert(l[0].id, l[0])
        self.assertTrue(advert.url,
'URL for announce "%s" not found: %s'
% (advert.id, advert.url))
```

Ce serait un comble de ne pas trouver d'annonce parlant de Linux ! Vérifions donc le test, ainsi que le style de codage :

```
$ tools/run_tests.sh linuxjobs
$ tools/pyflakes.py
```

9. PTP !

C'est le moment de payer, on crée un commit :

```
$ git add modules/linuxjobs
$ git commit -m "Add linuxjobs.fr module"
```

En vrai le **commit-log** serait plus long, mentionnant les fonctionnalités manquantes dans un second paragraphe. Weboob disposant maintenant d'une instance de Gitlab, il nous suffit de créer un compte dessus, d'éventuellement y déposer une clef ssh, de forker le dépôt, puis de l'ajouter comme remote :

```
$ git remote add mine https://git.weboob.
org/mmuman/devel.git
$ git fetch --all
```

On pousse la branche sur notre dépôt :

```
$ git push mine
```

Il ne nous reste plus qu'à cliquer sur le bouton « Merge Request » de la branche sur le site et remplir le formulaire. Bien qu'il ne soit plus nécessaire d'envoyer le patch sur la liste de diffusion [8], il est recommandé de s'y inscrire si vous comptez contribuer régulièrement. Le canal IRC étant très actif c'est un bon endroit pour discuter de votre code avant de le soumettre pour fusion.

CONCLUSION

Nous avons écrit et contribué à notre premier patch sur Weboob, ajoutant un module de recherche d'emploi presque complet, en utilisant les nouvelles possibilités du *framework*. Pour avoir déjà écrit quelques modules avant **Browser2**, la concision des filtres et leurs possibilités de composition rendent le code bien plus court et lisible. Notez que Weboob est utilisé par plusieurs entreprises dans leur logiciel (**Budgea**, **Cozy Cloud**, etc.), et qu'il dispose même d'un support professionnel **[9]**.

Enfin, sachez qu'en plus de contribuer au code, vous pouvez également soutenir les développeurs en devenant membre de l'Association Weboob **[10]** pour la modique somme de cinq euros.

POUR ALLER PLUS LOIN

Pour les anglophones voulant apprendre à contribuer sérieusement à des projets libres, il existe l'*Upstream University* : http://upstream-university.org/ ■

RÉFÉRENCES

[1] Site officiel de *Weboob* : http://weboob.org/

[2] *« Le job board du Logiciel Libre et de l'open source »* : https://www.linuxjobs.fr/

[3] Canal IRC officiel de Weboob : irc://irc.freenode.net/#weboob

[4] Documentation développeurs : http://dev.weboob.org/

[5] Procédures d'installation : http://weboob.org/install

[6] Serveur git (dépôts officiels et personnels) : https://git.weboob.org/

[7] BACHELIER L., *« Faster module creation for Weboob »* : http://laurent.bachelier.name/2013/02/faster-module-creation-for-weboob/

[8] Liste de diffusion : http://lists.symlink.me/mailman/listinfo/weboob

[9] Support professionnel pour Weboob : http://weboob.com/

[10] Association Weboob : http://association.weboob.org/

CONCTRUCTIONS « WITH » EN LANGAGE... BASH !

ETIENNE DUBLÉ
[Ingénieur de Recherche CNRS au LIG]

RELU PAR H.-J. AUDÉOUD

MOTS-CLÉS : SHELL, BASH, PYTHON, ROBUSTESSE, DÉBOGAGE, OS LIVE

Dis-moi petit syntax checker, qu'as-tu à me dire sur mon code source ? Comment ça, tu me rends la main tout de suite ? Tu n'as rien trouvé ?? Pas le moindre petit warning ??? Allez, ne sois pas timide, je t'ai mis l'option ultra-verbose je te signale !! T'es pas censé être timide !! Écoute, tu vois bien que je travaille seul là-dessus... si tu ne me dis rien, qui d'autre va me faire la conversation ?? Bon. Tu l'auras voulu. Voyons si tu resteras de marbre quand j'aurai ajouté cette structure exotique à mon code source...

INTRODUCTION

Quelques mois plus tôt...

En ce moment je travaille sur **debootstick**, un outil pour générer des systèmes bootables. Mon but final est de pouvoir enchaîner les commandes suivantes sur un système **Debian** (ou **Ubuntu**...) :

```
$ debootstrap jessie os_tree
$ debootstick os_tree usb.img
```

L'outil **debootstrap** (à ne pas confondre avec **debootstick** donc !) utilisé en première ligne est un grand classique des systèmes Debian. Il permet de générer dans le répertoire donné en paramètre (ici **os_tree**) une arborescence minimale d'OS Debian (ici en version **jessie**). Mon outil **debootstick** se chargera alors, dans un deuxième temps, de convertir cette arborescence en image bootable (**usb.img** dans cet exemple).

On pourra ensuite écrire l'image bootable obtenue sur une clé USB (ici **/dev/sdb**) :

```
$ dd bs=10M if=usb.img
of=/dev/sdb
```

Et l'OS pourra alors être démarré sur n'importe quelle machine.

Vous vous dites peut-être qu'on pourrait simplifier la chose, côté utilisateur, en intégrant dans **debootstick** l'appel préliminaire à la commande **debootstrap**. C'est d'ailleurs ce que font, en général, les outils de ce genre. De mon côté, j'ai préféré suivre la philosophie UNIX et restreindre l'outil à une seule fonction « atomique ». Cela rendra l'outil plus souple et, paradoxalement, plus puissant. En effet, on peut imaginer toute une gamme d'autres scénarios pour générer ou customiser notre arborescence **os_tree**. Par exemple :

```
$ docker run --name jessie-container -it
debian:jessie /bin/sh
[docker-shell]$ [... cutomize ...]
[docker-shell]$ exit
$ mkdir os_tree
$ docker export jessie-container | tar xf
- -C os_tree
$ debootstick os_tree usb.img
```

Et voilà. Après transfert sur une clé USB, on peut faire booter sur une machine physique ce qui n'était jusqu'à présent qu'un conteneur virtuel. Sympa, non ?

Le souci, c'est que pour l'instant cet outil n'existe que dans mon imagination. J'en suis dans les premières phases de développement, et je n'avance pas bien vite. Ce que je vous ai présenté comme une simple « conversion » cache en réalité une certaine complexité : il faut créer un fichier image, le partitionner, demander au noyau de créer des *devices* virtuels pour manipuler ces partitions, initialiser les systèmes de fichiers, faire des **mount** et des **chroot**, copier l'arborescence, installer les éventuels paquets manquants (noyau linux, *bootloader*, etc.), installer les *bootloaders* (BIOS et UEFI), puis tout redéfaire. Et je simplifie pas mal de choses (par exemple, pour obtenir une image de taille minimale, il faut ruser). Le plus ennuyeux, c'est quand on rencontre une erreur au beau milieu de l'exécution. Le cas typique, c'est le « **no space left on device** ». Le script s'arrête, et il laisse votre machine dans un état imprévisible, avec des montages sur un fichier temporaire qui n'existe plus, etc. Si on essaie de faire le ménage à la main, on y passe un bon quart d'heure. Jusque-là, j'optais donc pour la méthode Windows : je rebootais la machine. Mais je perds clairement trop de temps.

Pour résoudre ce souci, il faudrait que **debootstick** enregistre cet empilement de commandes délicates. En cas d'imprévu, il serait alors capable de tout défaire, en dépilant. S'il était écrit en **Python** par exemple, on pourrait utiliser des blocs **with** imbriqués. Mais **debootstick** est clairement orienté « système », donc le shell est plus approprié. Et en shell, pas de bloc **with** !

Pardon ? On me dit dans l'oreillette que vous ne seriez pas contre un petit rappel sur ce fameux bloc **with**. Faisons donc le point sur cette construction, telle qu'elle apparaît en Python.

1. BLOC « WITH » EN PYTHON

Prenons pour exemple la maxime : « Avec ce marteau, tous les problèmes ressemblent à des clous » :

```
problemes = [ 'clou1', 'clou2' ]
with prendre_marteau() as marteau:
    for probleme in problemes:
        marteau.clouer(probleme)
```

On voit déjà que le bloc **with** met en évidence l'outil (ici le **marteau**), ainsi que le périmètre de son utilisation, le tout de manière assez élégante. Mais allons un peu plus loin. En fait, il y a du code caché qui s'exécute à l'entrée et à la sortie du bloc. Complétons notre exemple :

```
#!/usr/bin/env python
# -*- coding: utf8 -*-
import sys

class Marteau(object):
    def __enter__(self):
        print('marteau en main')
        return self
    def __exit__(self, type, value, tb):
        print('marteau rangé')
    def clouer(self, objet):
        assert (objet.startswith('clou')), \
            "Je ne peux pas clouer '%s'!" % objet
        print(objet + ' cloué.')

def prendre_marteau():
    return Marteau()

problemes = sys.argv[1:]
with prendre_marteau() as marteau:
    for probleme in problemes:
        marteau.clouer(probleme)
print('Au revoir!')
```

Voilà ce que ça donne à l'exécution :

```
$ ./test.py clou1 clou2
marteau en main
clou1 cloué.
clou2 cloué.
marteau rangé
Au revoir!
$
```

Et voici quelques explications.

1) Quand l'interpréteur Python a rencontré notre bloc **with**, il exécute **prendre_marteau()**.

2) S'agissant d'un bloc **with**, le résultat de **prendre_marteau()** doit être un objet qui implémente les méthodes spéciales **__enter__()** et **__exit__()**. On appelle cet objet un *context manager*. L'interpréteur appelle alors la méthode **__enter__()** du *context manager*.

3) La méthode **__enter__()** renvoie un résultat, ici **self** (c'est souvent le cas), et ce résultat est assigné à la variable spécifiée après l'instruction **as** (ici **marteau**).

4) L'interpréteur exécute le contenu du bloc **with**.

5) En sortie du bloc, l'interpréteur appelle la méthode **__exit__()** du *context manager*.

6) L'exécution reprend après le bloc **with**.

Et que se passe-t-il si on essaie de clouer autre chose que des clous ? Essayons :

```
$ ./test.py clou1 vis clou2
marteau en main
clou1 cloué.
marteau rangé
Traceback (most recent call last):
  File [...]
AssertionError: Je ne peux pas clouer 'vis'!
$
```

Pas de problème pour clouer **clou1**, en revanche la même tentative sur **vis** a levé une exception. Cette exception a visiblement interrompu la suite du bloc (puisque **clou2** n'a pas non plus été cloué), et a provoqué une sortie prématurée du programme (puisque le message « Au revoir! » n'apparaît pas).

En revanche, on voit bien le message « marteau rangé », ce qui prouve que la fonction **__exit__()** a bien été appelée, pour faire le ménage en sortie du bloc **with**. Et ceci malgré l'exception !!

Les connaisseurs me diront qu'on pourrait obtenir le même genre de comportement en utilisant la clause **finally** d'un bloc **try**. En fait, la sémantique est quand même un peu différente, car avec un bloc **with**, l'exception n'est *pas* capturée, elle continue donc son chemin en dehors du bloc. Souvent, c'est ce qu'on attend du programme : en cas d'imprévu, on préfère sortir au plus tôt, pour limiter les dégâts. D'autre part, dans un programme où on doit « faire » puis « défaire » quelque chose, les méthodes **__enter__()** et **__exit__()** proposent une sémantique parfaite. Meilleure à mon avis qu'un bloc **try...finally**.

Le langage python fournit en standard des objets faisant office de *context manager*. C'est le cas par exemple des « objets fichiers ». Ainsi, sans connaître le détail des méthodes **__enter__()** et **__exit__()**, on peut écrire :

```
with open('mon_fichier.txt') as f:
    print(f.read()) # ou tout autre traitement sur f
```

Ici, l'avantage, c'est qu'on s'assure que le fichier sera fermé en sortie de bloc, quoi qu'il arrive.

Je ne vais pas m'étaler beaucoup plus, mais sachez qu'on peut très bien imbriquer plusieurs blocs **with**. L'idée, c'est qu'on va empiler des traitements via les fonctions **__enter__()** et les dépiler via les fonctions **__exit__()**. Et le dépilage interviendra donc dans tous les cas (exécution normale ou exception).

Vous comprenez donc pourquoi j'aimerais pouvoir appliquer le même principe à mon script shell **debootstick**. Donc essayons !

2. BLOC « WITH » EN BASH (??)

2.1 Implémentation naïve

On va commencer simplement et faire notre test à partir du code bash suivant, proche de la version Python :

```
#!/usr/bin/env bash
source with_bloc.sh

clouer() {
    objet="$1"
    if [ "${objet:0:4}" != "clou" ]
    then
        echo "Je ne peux pas clouer '$objet'!"
        return 1  # non-nul = echec
    else
        echo "$objet cloué."
    fi
}

prendre_marteau() {
    echo 'marteau en main'
}

defaire_prendre_marteau() {
    echo 'marteau rangé'
}

problemes="$@"
with prendre_marteau
    for probleme in $problemes
    do
        clouer "$probleme"
    done
end_with
echo 'Au revoir!'
```

À la fin du code, on voit ce qui ressemble à un bloc **with**, entre deux balises **with** et **end_with**. Afin qu'elles soit réutilisables, je les ai définies dans un fichier **with_bloc.sh** séparé, importé en deuxième ligne.

Maker Faire® Paris

9 > 11 juin 2017

Cité des Sciences et de l'Industrie

Appel aux Makers

JUSQU'AU 30 AVRIL

Maker Faire est à la fois une fête de la science, une foire populaire et un événement de référence pour l'innovation. Ce concept regroupe stands de démonstration, ateliers de découverte, spectacles et conférences autour des thèmes de la créativité, de la fabrication, et des cultures Do It Yourself.
Aujourd'hui, plus de 200 éditions réunissent dans 38 pays **des communautés de passionnés, experts ou débutants, qui partagent l'envie de créer, fabriquer et apprendre les uns des autres.**
La 4ème édition de **Maker Faire Paris** se tiendra à la Cité des Sciences et de l'Industrie.

Vous souhaitez participer ?
Rejoignez la communauté des Makers en vous inscrivant à notre appel !

Clôture des inscriptions le 15 avril 2017 à minuit !

Un événement
Make: makezine.com

Présenté par
LEROY MERLIN ...et vos envies prennent Vie !

Partenaire
cité des sciences & de l'industrie

Inscription sur
paris.makerfaire.com

La deuxième chose à remarquer dans ce code, c'est la convention que j'ai adoptée : pour pouvoir utiliser une fonction **<f>()** dans un bloc **with**, il faudra au préalable définir la fonction **defaire_<f>()**. Comme vous vous en doutez, cette fonction sera appelée automatiquement au moment du **end_with** (avec les mêmes arguments).

Avant de vous dévoiler ce que j'ai mis dans **with_bloc.sh**, voyons déjà si ça marche :

```
$ ./test.sh clou1 clou2
marteau en main
clou1 cloué.
clou2 cloué.
marteau rangé
Au revoir!
$
```

Apparemment, oui, en tout cas dans ce cas simple.

Voici donc le contenu de **with_bloc.sh** :

```
 1  EOL="
 2  "
 3  a_depiler=""
 4
 5  with() {
 6      cmd="$*"
 7      $cmd
 8      a_depiler+="${EOL}defaire_$cmd"
 9  }
10
11  end_with() {
12      cmd_defaire="$(echo "$a_depiler" |
tail -n 1)"
13      $cmd_defaire
14      a_depiler="$(echo "$a_depiler" |
head -n -1)"
15  }
```

Comme vous voyez, il n'y a pas des masses de code ; **with** et **end_with** sont en fait de simples fonctions. L'idée générale, c'est qu'à chaque appel **with()**, on exécute bien sûr la commande donnée en paramètre, mais on doit aussi stocker quelque part la commande **defaire_<qqchose> <args>** qui sera appelée au moment du **end_with**. Et pour que ça fonctionne quand on a plusieurs **with** imbriqués, on doit stocker ces commandes sur une *pile* : on empile à chaque appel **with()**, on dépile à chaque **end_with()**.

Pour implémenter la pile **[2]**, j'ai choisi de simplement chaîner des éléments dans une chaîne de caractères (variable **a_depiler**). Chaque élément est de la forme **\n<cmd>**. J'ai défini le séparateur **\n** via la variable **EOL** (lignes 1 et 2), parce qu'en bash si on écrit **\n** on écrit en réalité les 2 caractères **backslash** et **n**. En revanche, et c'est la raison pour laquelle j'ai choisi ce caractère, en matière de traitement ligne par ligne on est plutôt bien outillé, en shell. Ce codage permet ainsi, de manière assez évidente :

- d'obtenir le dernier élément (= le haut de la pile) avec un simple **tail -n 1** (ligne 12) ;
- de dépiler ce dernier élément via **head -n -1** (ligne 14).

Pour ce qui est d'empiler, dans la fonction **with()** donc, on a juste à utiliser la concaténation de chaîne (via l'opérateur **+=**, ligne 8). Vous noterez que sur cette ligne on ajoute le préfixe **'defaire_'** à la commande originale, dans l'esprit de la convention décrite plus haut.

En dehors de cette gestion de pile, la ligne 7 permet d'exécuter la commande donnée en paramètre du **with**, et la ligne 13 permet d'exécuter la commande préfixée par **'defaire_'**.

2.2 Gestion des exceptions

Bon. Tout ça c'est bien beau, mais qu'est-ce qui se passe si on tente de clouer des vis ?

```
$ ./test.sh clou1 vis clou2
marteau en main
clou1 cloué.
Je ne peux pas clouer 'vis'!
clou2 cloué.
marteau rangé
Au revoir!
$
```

Mouais, c'est pas idéal... On est censé sortir du bloc **with** dès qu'une erreur survient, et là au contraire le programme a continué comme si de rien n'était. Il faudrait générer une exception quand on arrive sur la vis. Le problème c'est que... euh... en fait il n'y a pas de notion d'exception, en bash ! **[3]**

Bon. Mais à bien y réfléchir, on a quelque chose qui s'en rapproche. Un truc qu'on trouve dans tous les articles sur les « bonnes pratiques » en bash. C'est l'instruction **set -e**. Rajoutons-la en haut du script :

```
#!/usr/bin/env bash
set -e
source with_bloc.sh
[...]
```

Relançons le même test :

```
$ ./test.sh clou1 vis clou2
marteau en main
clou1 cloué.
Je ne peux pas clouer 'vis'!
$
```

Avouez qu'on a l'impression de sortir sur une exception ! En fait, quand on lance la commande **set -e**, on indique à l'interpréteur du script qu'il doit vérifier le code de retour de chaque commande. Et si ce code de retour est non nul (ce qui est un signe d'erreur), l'exécution est interrompue. Dans notre exemple, la commande **return 1** de la fonction **clouer()** a donc suffi pour interrompre l'exécution.

Le souci dans ce cas, c'est que le marteau n'a pas été rangé. Cela implique que le code du **end_with** n'a pas été exécuté, contrairement à ce qui se passe avec un vrai bloc **with**, comme en Python.

Pour forcer l'appel du **end_with** en cas de sortie prématurée, il faudrait déclencher un traitement juste avant de sortir du script. Pour ce genre de choses, on utilise une fonctionnalité annexe de la commande **trap**. L'utilité principale de cette commande est d'associer un traitement à la réception d'un signal. Par exemple :

```
...
on_sigint() {
    # ... traitement SIGINT
}

trap on_sigint SIGINT
...
```

Mais en réalité, en deuxième paramètre, on n'est pas restreint aux seuls identifiants de signaux : la commande **help trap** (voir encadré) nous indique quelques possibilités supplémentaires. En particulier, on peut spécifier **EXIT** pour détecter la sortie du script, et donc exécuter un code adéquat juste avant de sortir.

Rajoutons donc le code suivant à la fin de **with_bloc.sh** :

```
on_exit() {
    while [ "$a_depiler" != "" ]
    do
        end_with
    done
}

trap on_exit EXIT
```

Voilà, je crois que ce code est assez évident : en sortie du script, on referme les éventuels blocs **with** restés ouverts. Testons :

```
$ ./test.sh clou1 vis clou2
marteau en main
clou1 cloué.
Je ne peux pas clouer 'vis'!
marteau rangé
$
```

Cette fois-ci, ça fonctionne !

POURQUOI « HELP TRAP » ET PAS « MAN TRAP » ?

Certaines commandes, comme trap, sont internes au shell. La preuve :

```
$ which trap
$
```

Il n'y a pas d'exécutable nommé trap sur le système. C'est juste bash qui l'interprète. Par conséquent, la documentation de la commande trap s'obtient par... man bash ! Le souci, c'est que la page de manuel de bash est un peu longuette. C'est pour ça que je vous proposais d'utiliser l'aide en ligne de bash, en tapant help <cmd>.

Pour corser un peu le tout, sachez qu'il y a aussi des commandes qui sont à la fois disponibles sur le système ET implémentées en interne par bash. C'est le cas de la commande echo par exemple :

```
$ which echo
/bin/echo
$ help echo
echo: echo [-neE] [arg ...]
    Write arguments to the standard
output.
[...]
```

Quel intérêt me direz-vous ? La plupart du temps, c'est une question de performance. Si le shell utilise la commande système, alors il lui faut créer un processus fils à chaque fois pour la lancer, ce qui est quand même très coûteux pour une commande aussi basique. Le problème, c'est qu'il peut y avoir quelques différences subtiles entre les deux implémentations. Et au final, si vous tapez man echo, vous ne regardez sûrement pas la bonne documentation ! Il y a d'ailleurs une indication dans cette page de manuel qui devrait vous alerter à ce sujet.

2.3 Le test de trop

Il ne reste plus qu'une chose à tester : les bloc **with** imbriqués. Mais normalement, avec notre gestion basée sur une pile, ça devrait rouler.

Bon voilà, j'ai réécrit la fin de **test.sh**. Voilà les dernières lignes :

```
[...]
with ouvrir_tiroir
```

```
    with prendre_marteau
        for probleme in $problemes
        do
            clouer "$probleme"
        done

with prendre_telephone
    echo "allo, chef, c'est fait!"

echo '** fin du script **'
```

Je vous laisse imaginer le code de **ouvrir_tiroir()**, **defaire_ouvrir_tiroir()**, **prendre_telephone()** et **defaire_prendre_telephone()**. Voyons ce que ça donne à l'exécution, juste avec 2 clous pour commencer :

```
$ ./test.sh clou1 clou2
tiroir ouvert
marteau en main
clou1 cloué.
clou2 cloué.
telephone en main
allo, chef, c'est fait!
** fin du script **
telephone sur son socle
marteau rangé
tiroir fermé
$
```

?? Qu'est-ce que c'est que ce bazar ? Y aurait-il un souci dans ma gestion de pile ?[1]

Un bon quart d'heure plus tard...

Suis-je idiot ! J'ai juste oublié d'écrire les **end_with** !! Je pouvais toujours chercher un bug dans ma gestion de pile !

Trois balises end_with rajoutées plus tard...

C'est reparti pour le test :

```
$ ./test.sh clou1 clou2
tiroir ouvert
marteau en main
clou1 cloué.
clou2 cloué.
marteau rangé
tiroir fermé
telephone en main
allo, chef, c'est fait!
telephone sur son socle
** fin du script **
$
```

C'est mieux, beaucoup mieux ! Et avec la vis :

```
$ ./test.sh clou1 vis clou2
tiroir ouvert
marteau en main
clou1 cloué.
Je ne peux pas clouer 'vis'!
marteau rangé
tiroir fermé
$
```

Impeccable, on a juste refermé les **with** ouverts, avant de quitter à cause de l'exception.

3. UNE INTÉGRATION PLUS POUSSÉE DANS LE LANGAGE

Cette petite mésaventure d'oubli des **end_with** m'a fait prendre conscience d'une limite importante de mon implémentation. En Python, la fermeture du bloc **with** est implicite, elle s'effectue automatiquement quand on quitte le bloc indenté. De ce fait, il n'y a pas de mot-clé comme **end_with** à indiquer (ou à oublier, en l'occurrence). En bash, l'indentation n'influe pas sur la syntaxe, mais on a quand même des vérifications de syntaxe bien utiles sur certaines constructions. Par exemple, dans une boucle **while**, si jamais on oublie le **done** final, on aura une erreur de syntaxe, ce qui a l'avantage d'aiguiller directement le programmeur sur le bon diagnostic.

Avec mon implémentation actuelle, il paraît compliqué de déclencher une telle erreur de syntaxe. Mais si on la fait évoluer un peu, c'est peut-être possible. Je crois d'ailleurs que j'ai une petite idée qui se dessine, un genre de détournement de la boucle **while**...

3.1 Une boucle while contrôlée

Commençons par un petit test avec le script suivant, **boucle_controlee.sh** :

```
#!/bin/bash

boucle_controlee() {
    if [ $entree_boucle = 1 ]
    then
        echo entree du bloc $*
        entree_boucle=0
        return 0    # condition ok, re-boucler
    else
        echo sortie du bloc $*
        return 1    # condition pas ok, sortir
    fi
}
```

[1] L'avis de mon relecteur est plutôt que le script a développé une intelligence propre et déduit qu'avec deux mains un humain peut très bien prendre le téléphone sans lâcher le marteau...

```
entree_boucle=1
while boucle_controlee
do
    echo dans le bloc
done
```

Si on identifie les itérations du **while** (variable **entree_boucle**), on peut la « contrôler » : on décide de passer le premier test, donc d'exécuter le contenu du bloc, mais d'arrêter au deuxième test. Voilà ce que ça donne à l'exécution :

```
$ ./boucle_controlee.sh
entree du bloc
dans le bloc
sortie du bloc
$
```

3.2 Un alias pour cacher ce que vous n'êtes pas censés voir

Si vous ne voyez pas où je veux en venir, modifions légèrement la chose en ajoutant la définition d'un alias :

```
#!/bin/bash

boucle_controlee() {
    [... voir plus haut ...]
}

shopt -s expand_aliases
alias with='entree_boucle=1; while boucle_controlee'

with ici_une_commande
do
    echo dans le bloc
done
```

Là c'est clair, non ? Il est pas beau ce bloc **with...do... done** ? Et là, si on oublie de fermer le bloc avec le **done**, on aura effectivement une erreur de syntaxe !

On peut vérifier, à l'exécution ça marche toujours :

```
$ ./boucle_controlee.sh
entree du bloc ici_une_commande
dans le bloc
sortie du bloc ici_une_commande
$
```

Il reste deux points à éclaircir. Le premier, c'est l'emploi bizarre de la variable **entree_boucle**, initialisée avant la boucle **while** (dans l'alias) et ensuite modifiée dans la fonction **boucle_controlee()**. En fait, on est tenté d'implémenter ce contrôle de la boucle dans la fonction uniquement, en alternant entre deux états, pour gérer respectivement l'entrée et la sortie du bloc. En réalité, ce serait une erreur, car l'entrée et la sortie correspondent à deux appels distincts de cette fonction. Or, en cas d'imbrication de blocs **with**, on casse cette alternance entre entrée et sortie de bloc. Au final, je me suis donc contenté de simplement repérer l'entrée dans la boucle.

Le deuxième point concerne l'activation de l'option **expand_aliases**. Il faut savoir que, par défaut, bash n'interprète les alias que lors d'une session *interactive*. Ici, s'agissant de l'exécution d'un script, ce n'est pas le cas. Il faut donc activer cette option pour que l'alias soit pris en compte.

3.3 Implémentation finale

Notre implémentation finale de **with_bloc.sh** va reprendre les éléments de notre première implémentation, ainsi que l'idée développée ci-dessus.

```
EOL="
"
a_depiler=""

start_with() {
    cmd="$*"
    $cmd
    a_depiler+="${EOL}defaire_$cmd"
}

end_with() {
    cmd_defaire="$(echo "$a_depiler" | tail -n 1)"
    $cmd_defaire
    a_depiler="$(echo "$a_depiler" | head -n -1)"
}

on_exit() {
    while [ "$a_depiler" != "" ]
    do
        end_with
    done
}

trap on_exit EXIT

boucle_controlee() {
    if [ $entree_boucle = 1 ]
    then
        start_with $*
        entree_boucle=0
        return 0    # condition ok, re-boucler
    else
        end_with
        return 1    # condition pas ok, sortir
    fi
}

shopt -s expand_aliases
alias with='entree_boucle=1; while boucle_controlee'
```

La fonction **with()** de notre première implémentation a été renommée en **start_with()**. Excepté ce renommage, elle est inchangée. La fonction **end_with()** est également inchangée, tout comme le code qui ferme les blocs restés ouverts en cas d'exception (**on_exit()** et commande **trap**). La fonction **boucle_controlee()** a été très légèrement modifiée pour appeler les fonctions **start_with()** et **end_with()**. Enfin, la définition de l'alias reprend ce qu'on a écrit ci-dessus.

Pour tester cette version, on pourra adapter notre script de test :

```
#!/usr/bin/env bash
set -e
source with_bloc.sh

[... mêmes fonctions que précédemment ...]

problemes="$@"
with ouvrir_tiroir; do
    with prendre_marteau; do
        for probleme in $problemes; do
            clouer "$probleme"
        done
    done
done

with prendre_telephone; do
    echo "allo, chef, c'est fait!"
done

echo '** fin du script **'
```

En testant, avec ou sans la vis, on retrouve des résultats identiques à notre première implémentation. Par contre, si on oublie de fermer un des blocs avec **done**, l'interpréteur détecte le souci de syntaxe :

```
$ ./test.sh clou1 vis clou2
./test.sh: ligne 56: erreur de syntaxe :
fin de fichier prématurée
$
```

Pour tout vous dire, c'est bien la première fois que je suis content de voir s'afficher une erreur de syntaxe. :)

ÉPILOGUE

Deux ans plus tard...

Après avoir implémenté cette gestion dans **debootstick** **[6]**, j'ai pu débuguer plus rapidement. Parce qu'un outil qui met le bazar sur votre machine à chaque plantage, c'est plutôt pénible à débuguer ! D'ailleurs, phase de débugage ou pas, cela reste un outil qui enchaîne les opérations bas niveau sur votre système. Ce « filet de sécurité » est donc plutôt bienvenu.

Peu de temps après, j'ai donc pu fournir une première version, capable de prendre en charge les scénarios décrits au début de l'article. Et quelques autres joyeusetés. L'outil est maintenant disponible dans l'OS Debian (testing), depuis l'été 2015. Vous pourrez donc le trouver dans la prochaine version stable de Debian **[7]**, « stretch », qui sera peut-être sortie quand vous lirez ces lignes.

Pour la petite histoire, quand on construit un paquet pour Debian on doit faire tourner **lintian**, un outil qui vérifie les sources. Comme je l'ai laissé entendre au tout début de l'article, cette construction bizarre n'est alors pas passée inaperçue ! J'ai dû ajouter ce qu'on appelle un « *lintian override* », une annotation indiquant à l'outil que ce qu'il a détecté n'est pas un souci.

Merci à Henry-Joseph pour la relecture ! ■

NOTES ET RÉFÉRENCES

[1] Vous avez pu entrevoir dans un article récent **[5]** cette suite d'opérations à effectuer. Notons toutefois que debootstick vise à créer des systèmes live pérennes sur le long terme, la structure de l'OS est donc plus simple (pas de squashfs), ce qui permet des mises à jour complètes (*bootloader* & noyau compris). D'autre part, debootstick ne travaille pas directement sur votre clé USB, il construit une image ; vous pourrez donc la tester au préalable avec kvm. En effet, flasher un OS sur une clé USB n'est pas un acte anodin vis-à-vis de la durée de vie de celle-ci.

[2] Dans la plupart des langages, pour implémenter une pile, on se base sur un tableau. Mais à mon avis, en bash, l'usage des tableaux est plutôt cryptique, surtout pour des opérations comme « supprimer le dernier élément du tableau ». Alors je ne vais pas souiller votre magazine préféré avec ce genre d'incantations.

[3] Si c'était un « article dont vous êtes le héros », ici il y aurait une première fin possible ;-)

[4] L'avis de mon relecteur est plutôt que le script a développé une intelligence propre et déduit qu'avec deux mains un humain peut très bien prendre le téléphone sans lâcher le marteau...

[5] ENDRES F., *« Live-System from Scratch »*, *GNU/Linux Magazine* n°202, mars 2017, p. 54 à 61.

[6] Page GitHub de debootstick : https://github.com/drakkar-lig/debootstick

[7] Package debootstick dans Debian Stretch : https://packages.debian.org/stretch/debootstick

INNOROBO EVENT

16-18 MAI 2017

PARIS FRANCE

WHERE INNOVATION GROWS

Plongez au **cœur** de l'**innovation robotique**

Chercheurs, créateurs et utilisateurs vous dévoilent le monde de demain

UNE APPROCHE HUMAINE DE LA ROBOTIQUE

SMART HOMES | INDUSTRY 4.0 & SUPPLY CHAIN SERVICES | MEDICAL & HEALTH | TECHNOLOGIES & FORESIGHT | SMART CITIES | FIELD ROBOTICS | HOSPITALITY RETAIL & TOURISM

www.**innorobo**.com

YouTube theinnorobo

ET SI LA PRÉDICTION N'ÉTAIT PLUS LE DOMAINE RÉSERVÉ DES ORACLES ?

TRISTAN COLOMBO

MOTS-CLÉS : PRÉDICTION, PYTHON, CORPUS, MODÈLE DE LANGUE, SÉQUENCE DE MOTS

Lorsque vous effectuez une recherche sur le Web, on vous propose automatiquement la fin de votre phrase. Lorsque vous tapez un texte, le traitement de texte vous propose des corrections. Lorsque vous écrivez un SMS, les mots sont automatiquement complétés. Non, ce n'est pas de la magie… il y a un truc !

Il peut être parfois assez impressionnant de constater à quel point un moteur de recherche semble être en capacité de lire dans notre esprit pour compléter notre requête. Dans l'Antiquité, c'était les oracles qui étaient chargés de ce travail : prédire ce qui allait arriver. Mais de là à connaître le prochain mot que vous souhaitiez écrire… cela devient trop complexe, il faut laisser œuvrer la science ! Science qui, je tiens à le préciser dès le début de cet article, a tout de même ses limites : tout le monde vérifie la correction automatique des SMS sous peine d'envoyer par inadvertance un message pervers à son patron pour signaler une simple absence ou à sa voisine pour lui demander si elle peut aller chercher les enfants à l'école. Pour éviter toute situation fâcheuse, le plus simple reste de désactiver la correction automatique.

Quoi qu'il en soit la technique permettant de prédire le prochain mot que vous allez écrire ou de corriger un mot est la même et se base simplement sur des statistiques. Dans cet article, nous allons concevoir un moteur basique de prédiction (de mots bien entendu, vous n'aurez pas la combinaison du prochain Loto :-)).

1. LE CORPUS

Avant toute chose, nous devons disposer d'un corpus, c'est-à-dire d'un ensemble de documents textuels ciblant le domaine dans lequel nous souhaitons intervenir. Plus ce corpus sera important, plus nos résultats pourront être justes (ce n'est quand même pas une certitude...). Si vous voulez prédire le prochain mot tapé par un utilisateur dans un texte en français, il vous faudra des textes français (de préférence datant de la même époque que l'utilisateur, la langue évoluant au cours des siècles, et du même niveau de langue, un rappeur ne s'exprimant pas de la même manière qu'un auteur classique). Si vous êtes dans un domaine plus technique, vous pouvez constituer un corpus représentatif d'un langage de programmation, etc.

Le **CNRTL** (Centre National de Ressources Textuelles et Lexicales) fournit un corpus basé sur des textes libres de droits, le corpus **Frantext**. Ce corpus est constitué de 500 œuvres de la littérature française du XVIIIème au XXème siècle. C'est donc une alternative possible pour récupérer un corpus conséquent (un formulaire permet d'effectuer une sélection parmi les auteurs, genres, etc.).

Pour notre moteur, il serait plus intéressant de créer notre propre corpus, ce qui nous permettra de cibler précisément le domaine d'application. Pour cela, nous allons travailler sur des fichiers ePub puisque le format est très répandu. Nous utiliserons la commande **ebook-convert** du paquet **calibre** (sous **GNU/Debian**) pour effectuer une conversion du format ePub vers le format texte :

```
$ sudo apt install calibre
$ ebook-convert fichier.epub fichier.txt
```

L'intérêt de l'ePub, c'est que l'on peut en trouver un peu n'importe où soit sous licence libre, soit à l'achat. N'étant pas un grand lecteur de numérique, j'ai créé mon corpus à partir du pack « Fantastique et Merveilleux » du site ebookenbib.net qui propose des ePubs de livres libres de droits (malheureusement, le projet a l'air mort). Alors certes, il ne s'agit pas d'œuvres très récentes et certaines sont des traductions, mais cela permet d'avoir un bon exemple de création de corpus.

```
$ mkdir -p corpus/epub
$ mkdir -p corpus/txt
$ cd corpus/epub
```

Téléchargez ensuite vos *ebooks* puis placez-les dans le répertoire **epub** :

```
$ mv ~/Telechargement/*.epub .
```

Il reste à les convertir :

```
$ for fic in *.epub; do ebook-convert "${fic}" "${fic}.txt"; done
```

Puis à créer le corpus :

```
$ mv *.txt ../txt
$ cd ../txt
$ cat *.txt > ../corpus.txt
```

Le fichier **corpus.txt** contient donc notre corpus personnalisé !

2. LES STATISTIQUES

Nous allons maintenant générer les statistiques d'apparition liées des mots de notre corpus (par exemple la probabilité de voir apparaître le mot « aboie » après le mot « chien »).

2.1 Les n-grammes

Les *n-grams* (en français on parle de n-grammes) sont des sous-séquences de **n** symboles (pour **n = 1** on parle d'unigramme, pour **n = 2** de bigramme et pour **n = 3** de trigramme). Pour calculer la probabilité d'apparition d'un n-gramme, on se base sur un **alphabet** définissant les symboles possibles et un **corpus d'apprentissage** de taille **L**. La vraisemblance d'apparition du prochain symbole va dépendre de l'historique d'apparition des symboles de taille **n - 1**. Prenons un exemple pour y voir plus clair.

Considérons :

- l'alphabet **{a, b, c}** ;
- le corpus d'apprentissage (volontairement très restreint) de taille **L = 9** : **aababcaab** ;

Dénombrons maintenant les n-grammes jusqu'à **n = 3** (les trigrammes ou 3-grammes) :

- 1-grammes : **a** (5 fois), **b** (3 fois) et **c** (1 fois) ;
- 2-grammes : **aa** (2 fois), **ab** (3 fois), **bc** (1 fois), **ba** (1 fois) et **ca** (1 fois) ;
- 3-grammes : **aab** (2 fois), **aba** (1 fois), **bab** (1 fois), **abc** (1 fois), **bca** (1 fois), et **caa** (1 fois).

Nous pouvons en déduire quelques proba→bilités :

- Nous avons dénombré **5 + 3 + 1 = 9** 1-grammes, donc **P(a) = 5/9**, **P(b) = 3/9 = 1/3** et **P(c) = 1/9** ;
- Nous avons **2 + 3 +1 +1 +1 = 8** 2-grammes, donc **P(aa) = 2/8 = 1/4**, etc. ;
- Nous avons dénombré **2 + 1 + 1 + 1 + 1 + 1 = 7** 3-grammes, donc **P(aab) = 2/7**, etc. ;
- Et surtout, nous pouvons déterminer des probabilités conditionnelles **P(x|y)** (lire « probabilité d'obtenir **x** si **y** auparavant ») :
  - → **P(b|a) = 3/5** (dans les 2-grammes nous avons 3 fois **ab** et 5 grammes commençant par **a** (les **aa** et les **ab**)) ;
  - → **P(b|aa) = 2/2 = 1** ;
  - → etc.

2.2 Implémentation en Python

2.2.1 Calcul des 1-grammes

Nous allons commencer par générer la liste des unigrammes (1-grammes) d'après un corpus qui sera passé en paramètre. Ici les unigrammes sont des mots (on ne va pas compter les lettres...). Voici le code de **predict.py** :

```
01: import sys
02: import re
03: import collections
04:
05:
06: def clearLine(line):
07:     line = re.sub(r"[^\w\s]", ' ', line)
08:     line = re.sub(r'\s+', ' ', line)
09:     return line.lower()
10:
11: def unigrams(infile, outfile='unigrams.txt'):
12:     print('Calcul des unigrammes...', end='',
flush=True)
13:     unigrams = {}
14:     try:
15:         with open(infile, 'r') as fic:
16:             for line in fic:
17:                 for word in clearLine(line).
split(' '):
18:                     if word != '':
19:                         if word in unigrams:
20:                             unigrams[word] += 1
21:                         else:
22:                             unigrams[word] = 1
23:     except Exception as e:
24:         print('\nUne erreur est survenue lors
de la lecture de', infile)
25:         print(e.strerror)
26:         exit(1)
27:
28:     try:
29:         with open(outfile, 'w') as fic:
30:             unigrams = collections.
OrderedDict(sorted(unigrams.items(),
key=lambda t : t[1], reverse=True))
31:             for word in unigrams:
32:                 fic.write(word + ' ' +
str(unigrams[word]) + '\n')
33:     except Exception as e:
34:         print('\nUne erreur est survenue
lors de l\'écriture dans', outfile)
35:         print(e.strerror)
36:         exit(1)
37:
38:     print(' Ok')
39:
40:
41: if __name__ == '__main__':
42:     if len(sys.argv) != 2:
43:         print('Syntax : predict <corpus_
file.txt>')
44:         exit(2)
45:     unigrams(sys.argv[1])
```

Dans les lignes 1 à 3 nous procédons aux imports de **sys** pour accéder aux arguments passés en paramètre lors de l'appel du programme, **re** pour manipuler les expressions régulières, et **collections** pour pouvoir utiliser des dictionnaires ordonnés (**OrderedDict**). La fonction **clearLine()** des lignes 6 à 9 « nettoie » la ligne qui lui est passée en paramètre : suppression des caractères autres que des lettres et des espaces (ligne 7), remplacement des suites d'espaces (combinaison d'espaces, tabulations, etc.) en ligne 8, et passage en minuscule de tous les caractères grâce à **lower()** en ligne 9.

La fonction **unigrams()** porte bien son nom puisque c'est elle qui calcule les unigrammes à partir du corpus **infile** (le résultat de la fonction, une liste de mots et d'occurrences dans le corpus, est écrit dans le fichier **outfile**). Notez en ligne 12 l'utilisation du paramètre **flush=True** dans la fonction **print()** pour forcer l'affichage (le temps de calcul étant long, sans ce paramètre le message apparaîtrait après le calcul).

Les unigrammes seront stockés dans un dictionnaire **unigrams** (défini en ligne 13). Nous parcourons ensuite le fichier du corpus et, pour chaque ligne (ligne 16), nous la « nettoyons » à l'aide de **clearLine()**, nous la découpons par mot (suivant les espaces à l'aide de **split()**) et nous parcourons chaque mot qui sera stocké dans **word** (ligne 17). Nous ajoutons ensuite ce mot (s'il n'est pas vide) au dictionnaire **unigrams** tout en comptant le nombre d'occurrences (lignes 18 à 22).

Il faut ensuite exporter le dictionnaire dans le fichier **outfile** dont le nom est passé en paramètre en ligne 11 (valeur par défaut **'unigrams.txt'**). Le traitement est loin d'être optimisé ici car, pour des raisons de facilité de lecture, un nouveau traitement est appliqué pour trier les mots par nombre d'occurrences décroissant. Pour cela, nous allons trier les **items()** d'**unigrams** et les placer dans un **OrderedDict** (ligne 30). Il n'y a plus ensuite qu'à parcourir le dictionnaire pour écrire les données dans le fichier (lignes 31 et 32).

Lors de l'exécution de ce code nous obtenons le fichier **unigrams.txt** :

```
$ python3 predict.py corpus/corpus.txt
$ more unigrams.txt
de 145410
et 92181
la 86397
le 77989
...
```

Nous pouvons voir que le mot « de » est le mot le plus utilisé de notre corpus. Nous ne pouvons pas faire grand-chose des unigrammes, si ce n'est de trouver les mots les plus utilisés du corpus.

2.2.2 Calcul des 2-grammes, 3-grammes, ... n-grammes

Les n-grammes à partir de **n = 2** sont un peu plus informatifs puisqu'ils se basent sur un historique (probabilité d'obtenir un mot connaissant les n-1 mots précédents). Nous allons immédiatement généraliser le calcul des n-grammes en utilisant une structure très proche du code précédent :

```
...
40: def ngrams(infile, n, outfile=''):
41:     if n < 2:
42:         print('Fonction ngrams() :
Valeur minimale de n : 2')
43:         exit(3)
44:     if outfile == '':
45:         outfile = str(n) + '_grams.txt'
46:     print('Calcul des
{}-grammes...'.format(n), end='', flush=True)
47:     ngrams = {}
48:     try:
49:         with open(infile, 'r') as fic:
50:             for line in fic:
51:                 words = clearLine(line).
split(' ')
52:                 words = [ elt for elt in
words if elt != '' ]
53:                 words = [ ' '.join(name)
for name in zip(*[words[i:] for i in
range(n)])]
54:                 for word in words:
55:                     if word in ngrams:
56:                         ngrams[word] += 1
57:                     else:
58:                         ngrams[word] = 1
59:     except Exception as e:
60:         print('\nUne erreur est survenue lors
de la lecture de', infile)
61:         print(e.strerror)
62:         exit(1)
63:
64:     try:
65:         with open(outfile, 'w') as fic:
66:             ngrams = list(ngrams.items())
67:             ngrams.sort(key=lambda elt :
elt[1], reverse=True)
68:             for word, val in ngrams:
69:                 fic.write(word + ' ' +
str(val) + '\n')
70:     except Exception as e:
71:         print('\nUne erreur est survenue lors
de l\'écriture dans', outfile)
72:         print(e.strerror)
73:         exit(1)
74:
75:     print(' Ok')
...
```

Le paramètre **n** de la fonction **ngrams()** permet d'indiquer quels n-grammes nous souhaitons calculer. Le code étant vraiment très proche du précédent je ne vais concentrer mes commentaires que sur les lignes de calcul proprement dites qui ne sont d'ailleurs pas forcément triviales. En ligne 51, **words** contient la liste des mots de la ligne qui vient d'être lue, puis en ligne 52 on supprime de cette liste les mots vides (**''**). Vient ensuite la ligne 53 qui va réaliser pratiquement tout le travail à elle seule. Pour bien la comprendre, je vous propose de la décortiquer pas à pas dans un shell Python en prenant une liste **words** d'exemple :

```
$ python3
>>> words = ['petite', 'phrase', 'd', 'exemple']
>>> zip(words, words[1:])
<zip object at 0x7f88751c75c8>
>>> list(zip(words, words[1:]))
[('petite', 'phrase'), ('phrase', 'd'), ('d',
'exemple')]
>>> [ ' '.join(name) for name in zip(words, words[1:])]
['petite phrase', 'phrase d', 'd exemple']
```

La fonction **zip()** crée les tuples constitués par les éléments (groupés par indice) des listes qui lui sont passées en paramètres. Ainsi pour une liste **['a', 'b', 'c']** et une autre liste **[1, 2, 3]**, nous obtiendrons les tuples **('a', 1)**, **('b', 2)** et **('c', 3)**. **words[1:]** correspond à la liste **words** à laquelle on ôte le premier élément : nous réalisons donc un décalage d'un mot ce qui correspond à un appel à :

```
>>> list(zip(['petite', 'phrase', 'd', 'exemple'],
['phrase', 'd', 'exemple']))
[('petite', 'phrase'), ('phrase', 'd'), ('d', 'exemple')]
```

Nous obtenons donc bien les 2-grammes. Voyons comment obtenir les 3-grammes :

```
>>> [words[i:] for i in range(3)]
[['petite', 'phrase', 'd', 'exemple'],
['phrase', 'd', 'exemple'], ['d', 'exemple']]
```

En augmentant progressivement le décalage de mots sur notre liste **words**, on obtient une liste de trois listes : sans décalage, avec un mot de décalage puis avec deux mots de décalage.

```
>>> list(zip(['petite', 'phrase', 'd', 'exemple'],
['phrase', 'd', 'exemple'], ['d', 'exemple']))
[('petite', 'phrase', 'd'), ('phrase', 'd', 'exemple')]
```

On obtient les 3-grammes. Par contre, le passage de paramètre dans la fonction **zip()** devra se faire en utilisant l'opérateur *star* (*) qui fera que chaque élément de la liste sera considéré comme un paramètre (sans cet opérateur la fonction ne voit qu'une seule liste). Voici un exemple d'application de *star* :

```
>>> list(zip([words[i:] for i in range(3)]))
[(['petite', 'phrase', 'd', 'exemple'],), (['phrase',
'd', 'exemple'],), (['d', 'exemple'],)]
>>> list(zip(*[words[i:] for i in range(3)]))
[('petite', 'phrase', 'd'), ('phrase', 'd', 'exemple')]
```

Enfin, pour obtenir une clé unique nous effectuons un simple **join()** sur le résultat obtenu.

```
>>> [ ' '.join(name) for name in zip(['petite',
'phrase', 'd', 'exemple'], ['phrase', 'd',
'exemple'], ['d', 'exemple'])]
['petite phrase d', 'phrase d exemple']
```

On retrouve enfin la ligne 53 complète :

```
53: words = [ ' '.join(name) for name in
zip(*[words[i:] for i in range(n)])]
```

Ensuite, pour écrire les données dans un fichier, dans les lignes 66 à 67, on convertit le dictionnaire **ngrams** en liste pour trier par nombre décroissant d'occurrences. Le calcul des 2-grammes se fera en appelant :

```
ngrams(sys.argv[1], 2)
```

On obtient alors le fichier **2_grams.txt** :

```
de la 17824
qu il 14098
de l 9706
à la 9141
d un 8404
d une 7823
que je 7564
...
```

En fonction des besoins vous pourrez donc générer les n-grammes avec la longueur d'historique souhaitée. Passons maintenant à l'utilisation de ces n-grammes.

3. LA PRÉDICTION

Pour pouvoir effectuer des prédictions, nous allons calculer les probabilités d'apparition d'un mot connaissant les n mots précédents. Pour cela, nous lirons un fichier de n-grammes et pour un historique donné nous calculerons pour chaque mot suivant possible le rapport du nombre d'occurrences de ce mot par le nombre total d'occurrences pour l'historique ciblé :

```
...
077: def generate_stats(infile):
078:     print('Lecture du fichier des
n-grammes...', end='', flush=True)
079:     ngrams = {}
080:     try:
081:         with open(infile, 'r') as fic:
082:             for line in fic:
083:                 words = line.split(' ')
084:                 value = int(words.pop())
085:                 next_word = words.pop()
086:                 key = ' '.join(words)
087:                 if key in ngrams:
088:                     ngrams[key][next_
word] = value
089:                 else:
090:                     ngrams[key] = { next_
word : value }
091:     except Exception as e:
092:         print('\nUne erreur est survenue
lors de la lecture de', infile)
093:         print(e.strerror)
094:         exit(1)
095:
096:     for words in ngrams:
097:         n = 0
098:         for next_word in ngrams[words]:
099:             n += ngrams[words][next_word]
100:         ngrams[words] = [ (next_word,
round(ngrams[words][next_word] / n, 3)) for
next_word in ngrams[words]]
101:         ngrams[words].sort(key=lambda elt
: elt[1], reverse=True)
102:
103:     print(' Ok')
104:     return ngrams
...
```

Là encore le code est assez simple. Pour chaque ligne lue dans le fichier des n-grammes (ligne 82), on découpe la ligne en mots (ligne 83). **words** est donc une liste contenant **n** mots (**n-1** mots d'historique et **1** mot suivant) et une valeur. En ligne 84, on récupère le dernier élément de la liste (la valeur) grâce à **pop()** qui renvoie et efface la dernière valeur de la liste. En ligne 85, on recommence la même opération pour le nouveau

dernier élément qui est donc le mot suivant et enfin en ligne 86 on crée la clé de notre dictionnaire en liant les mots restants par des espaces. Dans les lignes 87 à 90, on constitue le dictionnaire qui associe à chaque « historique » les différents mots suivants possibles ainsi que leur nombre d'occurrences.

Il faut ensuite créer le dictionnaire final qui contient les probabilités d'apparition d'un mot suivant l'historique (la clé). On parcourt chaque clé de notre dictionnaire pour compter le nombre total d'occurrences pour un historique donné (lignes 98 et 99) puis on associe à chaque historique une liste de tuples contenant le prochain mot et sa probabilité d'apparition tronquée à trois décimales (ligne 100). Pour finir, la liste associée à un historique est triée par ordre décroissant de probabilité d'apparition (ligne 101). Ainsi on peut facilement retrouver le mot ayant la plus forte probabilité d'apparaître (**ngrams[words][0][0]**) ou utiliser un seuil pour obtenir les meilleurs candidats.

Une fois que l'on dispose de ce dictionnaire la fonction de prédiction est une simple formalité :

```
...
106: def predict(stats, term, threshold):
107:     result = []
108:     if term in stats:
109:         for words, value in stats[term]:
110:             if value > threshold:
111:                 result.append((words, value))
112:     if result == []:
113:         return [('', 1)]
114:     else:
115:         return result
...
```

La fonction **predict()** attend un dictionnaire contenant les probabilités, un terme (un ou plusieurs mots en fonction des n-grammes utilisés) et un seuil. Si **term** est présent dans le dictionnaire (ligne 108) alors on construit la liste des résultats satisfaisant la contrainte de seuil (lignes 109 à 111). Chaque résultat est de la forme **(mot_suivant, probabilité)**. Si rien n'est trouvé, en ligne 113 on renvoie la liste **[('', 1)]** qui indique une probabilité de 100% d'avoir quelque chose... :-)

À titre exploratoire, on peut ajouter une fonction qui liste les plus fortes probabilités d'apparition :

```
...
117: def high_predict(stats, threshold):
118:     result = []
119:     for term in stats:
120:         for words, value in stats[term]:
121:             if value > threshold:
122:                 result.append((term, words, value))
123:     return result
...
```

Il n'y a plus qu'à tester notre programme. Au premier lancement il faudra créer le fichier des n-grammes puis ce ne sera plus nécessaire :

```
...
125: if __name__ == '__main__':
126:     if len(sys.argv) != 2:
127:         print('Syntax : predict <corpus_file.txt>')
128:         exit(2)
129:     ngrams(sys.argv[1], 2)
130:     ngrams = generate_stats('2_grams.txt')
131:     print(predict(ngrams, 'aujourd', 0.5))
```

Ici le test porte sur le mot suivant « aujourd » en se basant sur un 2-grammes avec une probabilité d'apparition supérieure ou égale à 50%. Le tableau suivant présente quelques résultats obtenus avec ce programme :

Historique	N-grammes	Mot(s) suivant(s)
aujourd	2-grammes	**hui** : 100 %
van	2-grammes	**helsing** : 93 %
n y	3-grammes	**a** : 35 %, **avait** : 20 %
déclama contre	3-grammes	**les** : 100 %
pourquoi ne pas	4-grammes	**le** : 16 %, **faire** : 12,5 %

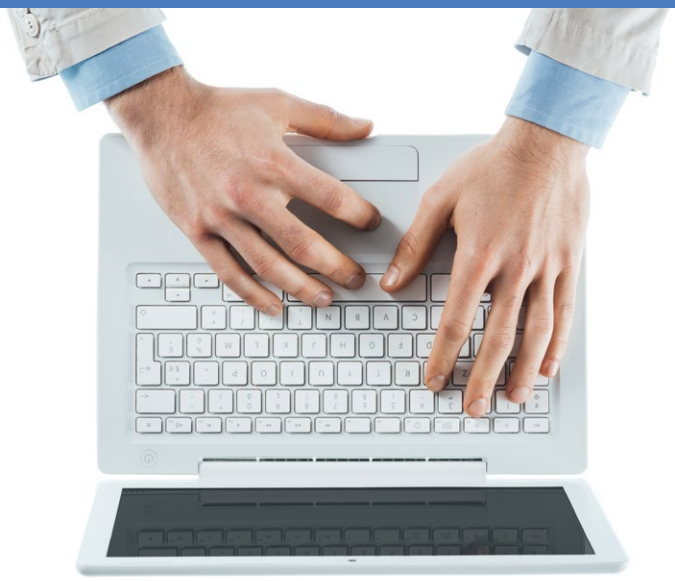
CONCLUSION

Il ne s'agit là que d'une présentation sommaire d'un domaine très vaste et il existe de nombreuses améliorations tant au niveau du modèle qu'algorithmique. En restant sur notre petit programme, on peut toutefois encore étendre le corpus pour obtenir des résultats plus précis ou encore sauvegarder les dictionnaires de probabilités en binaire de manière à ne pas avoir à les reconstruire à chaque lancement. On peut aussi appliquer le même principe sur la prédiction de mots complets (comme pour les SMS) en se basant sur des n-grammes de lettres et prédire la ou les lettres suivantes. Sachez enfin que **Google** propose sous licence Creative Commons des ensembles de n-grammes générés en 2012 pour différentes langues : http://storage.googleapis.com/books/ngrams/books/datasetsv2.html. Vous avez désormais toutes les clés en main pour effectuer différents tests... ■

CRÉEZ VOTRE PROPRE SAVEUR MARKDOWN

STÉPHANE MOUREY
[Mousse sur le Seeraiwer]

MOTS-CLÉS : SAVEUR, HTML, PHP, MARKDOWN, PERSONNALISATION



Markdown est un langage brillant tant par sa concision que par sa lisibilité. Mais il arrive parfois de le trouver un peu trop limité ou étroit. Du coup, on voit fleurir sur le Web de multiples saveurs qui tentent de répondre à certains besoins. Voyons comment développer la vôtre.

Créé par Aaron Swartz et John Gruber, ce langage remarquable de simplicité, destiné à être converti en **HTML**, a rencontré un succès remarquable. Voyons comment l'utiliser depuis **PHP** afin d'offrir à vos utilisateurs cette méthode de saisie que beaucoup considèrent bien plus élégante et pratique qu'un quelconque RTE, mauvaise imitation d'un traitement de texte.

Une des qualités du langage **Markdown**, que certains voient comme un défaut, est qu'il a donné lieu à de multiples implémentations apportant chacune leur lot de particularités. Ces variantes sont souvent appelées des *saveurs*. Au-delà de la base commune, un effort d'apprentissage est à chaque fois nécessaire pour apprendre chaque nouvelle saveur rencontrée, en constatant parfois des différences menant à des fonctionnalités identiques. Pourtant, cette versatilité du langage permet de l'adapter à son contexte, qui a bien souvent des exigences ignorées du contexte général. Aujourd'hui, nous verrons tout d'abord comment implémenter le langage de base ainsi que l'une ou l'autre de ses saveurs, ensuite comment ajouter de nouvelles fonctionnalités et enfin comment modifier les fonctionnalités existantes.

Notre implémentation sera purement serveur, réalisée en PHP sans se préoccuper aucunement de l'interface utilisateur. Nous nous appuierons sur **Cebe Markdown [1]**, un *parser* Markdown particulièrement souple. Projet de Carsten Brandt, un des développeurs du fameux **Yii framework**, Cebe Markdown a été développé pour produire la documentation de ce *framework*. Ce *parser* supporte d'emblée le Markdown traditionnel, la saveur **GitHub** et partiellement la saveur **PHP Markdown Extra [1]**.

1. INSTALLATION

L'installation s'opère à l'aide de **Composer** avec cette simple commande :

```
$ composer require cebe/
markdown "~1.0.1"
```

Cela provoquera le téléchargement de Cebe Markdown, ainsi que la création ou la mise à jour du fichier `composer.json` de votre projet.

2. PARSER LE MARKDOWN

Pour la suite de notre article, nous supposerons que la chaîne de caractères formatée en Markdown à convertir en HTML est reçue dans la variable **$markdown**. Le résultat de la conversion sera également une chaîne, reçue dans **$html**.

Afin de convertir notre chaîne, il suffit de créer un objet capable de cette tâche, et d'appeler sa méthode **parse** en lui passant la chaîne en paramètre : la méthode nous renverra l'objet converti. La classe de l'objet dépend de la saveur que vous souhaitez utiliser : **Mardown** pour la saveur classique, **GithubMarkdown** pour la saveur GitHub, et enfin **MarkdownExtra** pour le parfum Markdown Extra. Voici ce que cela donne :

```
$parser = new \cebe\markdown\Markdown();
echo $parser->parse($markdown);

$parser = new \cebe\markdown\
GithubMarkdown();
echo $parser->parse($markdown);

$parser = new \cebe\markdown\
MarkdownExtra();
echo $parser->parse($markdown);
```

Notez qu'il est possible d'ignorer les éléments de type bloc pour traiter seulement les éléments en ligne, ce qui peut se révéler utile dans certaines situations :

```
echo $parser-
>**parseParagraph**($markdown);
```

L'objet **parser** supporte quelques options :

```
$parser->html5 = true;
```

Par défaut, le HTML produit suit les spécifications de la version 4. Le passage de la propriété **html5** à **true**, permet de passer à la version 5 :

```
$parser->keepListStartNumber = true;
```

La propriété **keepListStartNumber** permet de maintenir la numérotation indiquée depuis la chaîne d'origine jusque dans le HTML produit (par défaut, un *parser* Markdown est censé incrémenter sans se préoccuper de la numération effectivement présente).

```
$parser->enableNewlines = true;
```

Par défaut en Markdown, un retour de chariot ne provoque pas de retour à la ligne en HTML. L'idée est que le retour chariot est utilisé pour la lisibilité du code source et ne doit donc pas interférer avec le HTML produit. Pour faire effectivement un retour à la ligne dans le HTML, on sautera une ligne, ce qui provoquera un nouveau paragraphe. Pour un simple retour de ligne **
, le retour chariot doit être suivi de deux espacements. La propriété **enableNewlines modifie ce comportement, en convertissant tous les retours à la ligne en balise **
**.

Enfin, sachez qu'en plus de la classe, il est possible d'utiliser le *parser* depuis la ligne de commandes :

```
$ bin/markdown monfichier.md
```

La ligne de commandes supporte également plusieurs saveurs grâce à l'option **--flavor** qui accepte deux valeurs : **gfm** pour GitHub et **extra** pour Markdown Extra. Enfin, l'option **--full** permet de produire un fichier HTML complet, avec en-tête et pied, le comportement par défaut étant de produire une conversion sans ajout :

```
$ bin/markdown --flavor=gfm --full monfichier.md
```

3. AJOUTEZ UN PEU DE PIMENT

Voyons maintenant comment cuisiner Markdown à notre propre sauce en lui ajoutant quelques fonctionnalités. Cebe Markdown reconnaît deux types d'éléments : les éléments en ligne et ceux en bloc. Cette distinction est la même qu'en HTML. Cebe Markdown utilise une approche différente pour chacun d'eux, vous devez donc commencer par décider si votre ajout va produire l'un ou l'autre.

3.1. Ajoutez des éléments de type bloc

Illustrons cela avec un exemple. Imaginons que nous décidions d'améliorer le traitement des blocs de code du Markdown. Comme pour la saveur GitHub, nous allons marquer le début et la fin du bloc par trois guillemets obliques ```. Deux mises en forme seront possibles, l'une par défaut que nous appellerons « source » et la seconde sera nommée « console ». On peut imaginer qu'il s'agit dans la présentation du document de distinguer des codes sources de programme d'une part, d'une copie de session en console, d'autre part, ainsi que nous le faisons dans ce magazine. Ajoutons une option **n** qui permettra d'indiquer que les lignes doivent être numérotées. Cette option acceptera un paramètre qui indiquera le nombre par lequel commencer la numérotation.

Voici un exemple du code Markdown amélioré que nous pourrions traiter ainsi :

````
```
print 'Hello world !';
```
``` console
$ echo "Hello world !"
```
``` source:n
$ echo "Hello world !"
```
``` console:n:100
print 'Hello world !';
```
````

Voici le HTML correspondant que nous souhaitons obtenir :

```
<pre><code class="source">print 'Hello world !';
</code></pre>
<pre><code class="console">$ echo "Hello world !"
</code></pre>
<pre><code class="source">1 $ echo "Hello
world !"<br/></code></pre>
<pre><code class="console">100 print 'Hello
world !';<br/></code></pre>
```

Pour y parvenir, il nous faut étendre la classe **cebe\markdown\Markdown**. Notez qu'il est possible de partir des classes **cebe\markdown\GithubMarkdown** et **cebe\markdown\MarkdownExtra**. Pour chaque nouveau type de bloc, il faudra d'abord lui choisir un nom **<nombloc>**, puis ajouter à notre nouvelle classe trois méthodes : **identify<nombloc>**, **consume<nombloc>** et **render<nombloc>**.

3.1.1. La méthode identify

Appelons notre nouveau type de bloc « source ». La méthode **identifySource** que nous allons donc écrire, permet d'identifier le début du bloc. Cette méthode est appelée pour chaque ligne qui n'est pas encore contenue dans un autre bloc déjà identifié et elle revoit une chaîne de caractères indiquant le type identifié. Elle reçoit trois arguments : le premier contient la ligne en cours d'analyse, le second un tableau contenant toutes les lignes, et le dernier, un entier correspondant à la clé de la ligne courante dans le précédent tableau. Presque toujours, vous n'aurez besoin que du premier, les autres paramètres étant fournis pour le cas où vous auriez besoin d'analyser le contexte environnant la ligne.

Voici ce que donne notre méthode **identifySource** :

```
06: protected function identifySource($line){
07:   if (strncmp($line, '```', 3) === 0) {
08:     return 'source';
09:   }
10: }
```

strncomp est une fonction PHP qui permet de comparer le début de deux chaînes de caractères sur une longueur limitée et qui renvoie **0** si elles ne diffèrent pas sur cette longueur. Donc si la ligne commence par trois guillemets obliques, notre méthode renvoie la chaîne de caractères **source** identifiant ainsi la suite du bloc.

3.1.2. La méthode consume

Le but d'une méthode **consume** est d'analyser le bloc de façon à en extraire d'éventuels paramétrages ainsi que d'en identifier la fin. Elle reçoit deux paramètres : le premier est le tableau de toutes les lignes du contenu, le second est l'index de la prochaine ligne à analyser. Elle renvoie un tableau contenant deux éléments : le contenu qui a été identifié comme étant celui appartenant au bloc sous forme d'un tableau de lignes, et la clé de la ligne où le bloc finit.

Il nous faut donc réaliser deux tâches dans cette méthode : tout d'abord, analyser la première ligne afin d'en extraire les paramétrages que nous avons défini plus haut ; ensuite, parcourir toutes les lignes qui suivent et les agréger dans un tableau jusqu'à découvrir la fin du bloc. Concrètement, dans notre cas, nous pouvons l'écrire ainsi :

```
12: protected function
consumeSource($lines,$current){
13:   $line = rtrim($lines[$current]);
14:   $params = explode(':',substr($line, 4));
15:   if (!in_array($params[0],$this->
allowedSourceTypes))
16:     $params[0]='source';
17:   $block = array(
18:     'source',
19:     'params'  => $params,
20:     'content' => array()
21:   );
22:   $maxIdx = count($lines);
23:   while (++$current < $maxIdx
24:          && ($line =
rtrim($lines[$current])) != '```'){
25:     $block['content'][] = $line;
26:   }
27:   return array($block,$current);
28: }
```

L'analyse des paramétrages a lieu des lignes 13 à 17. Après avoir éliminé les espaces inutiles en fin de chaîne et supprimé les premiers caractères indiquant le début du bloc, nous explosons la chaîne de caractères restante autour du caractère double-point dans un tableau recevant chacun de nos paramètres. Nous vérifions tout de même si le type de source indiqué correspond à un type supporté, en vérifiant s'il se trouve dans un tableau, propriété que nous avons déclarée au début de la classe :

```
03: class Edmarkdown extends \cebe\
markdown\Markdown {
04: protected $allowedSourceTypes =
array('source','console');
```

Aux lignes 17 à 21, nous préparons l'objet que nous allons renvoyer. La première entrée de ce tableau, qui ne contient que la valeur `'source'`, doit correspondre au nom de notre type de bloc. La troisième entrée, qui a pour clé `'content'`, doit recevoir les lignes de contenu qui vont être traitées comme faisant partie de ce bloc. Pour le moment, ce tableau est vide, mais nous allons bientôt le remplir. Pour le reste, nous sommes libres, et j'ai ajouté l'entrée `'params'` pour recevoir les paramètres que nous venons d'analyser sur la première ligne.

Des lignes 22 à 26, nous parcourons la liste des lignes que nous avons reçues en les ajoutant chacune dans notre tableau des lignes à traiter, jusqu'à ce que nous rencontrions le marqueur de fin de bloc, ou jusqu'à épuisement des lignes. Il faut prendre garde à incrémenter le compteur à chaque boucle, car le *parser* attend qu'on lui renvoie le numéro de la ligne à laquelle nous avons interrompu le traitement.

3.1.3. La méthode render

Une fois le paramétrage défini et le contenu à traiter identifié, il ne nous reste plus qu'à passer au traitement proprement dit. Pour cela, il nous faut écrire la méthode `renderSource`. La méthode ne reçoit qu'un seul argument : le tableau que nous avons élaboré au paragraphe précédent, contenant les paramètres et le contenu à remplacer. Elle renvoie en retour une chaîne de caractères qui doit prendre la place du contenu dans le contexte d'origine.

Dans notre cas, voici ce qu'elle donne :

```
30: protected function renderSource($block){
31:   $content ='<pre><code
class="'.$block['params'][0].'">';
32:   if (array_key_exists(1,$block['params'])
&&$block['params'][1]=='n'){
33:     $start = array_key_
exists(2,$block['params']) ? $block['params']
[2] : 1;
34:     $lineNumberLength = strlen(count($block
['params'])+$start);
35:     $i=$start;
36:     foreach ($block['content'] as $line) {
37:       $content.=str_
pad($i,$lineNumberLength,'0',
STR_PAD_LEFT).' '.$line.'<br/>';
38:       $i++;
39:     }
40:   }else{
41:     $content.= implode('<br/>',$block
['content']);
42:   }
43:   $content.='</code></pre>';
44:   return $content;
45: }
```

À vrai dire, il n'est pas toujours nécessaire de faire un traitement si compliqué : en effet, une bonne partie de notre travail ici consiste à gérer la numérotation optionnelle des lignes (lignes 32 à 40). Sans cela, la ligne 41 aurait été suffisante. Mis à part cela, notre travail a consisté à placer deux balises HTML **<pre><code>** permettant l'affichage correct d'un code source, de réassembler les différentes lignes de notre contenu en une seule chaîne de caractères et de concaténer le tout avec les balises fermantes nécessaires. Le tour est joué.

3.2. Ajouter des éléments en ligne

Là encore, prenons un exemple. Imaginons que nous voulions rajouter un élément de nom **red**. Nous utiliserons deux parenthèses pour le marquer. Partons du code Markdown suivant :

```
Un ((élément red))
```

Il sera transformé en :

```
Un <span style="red"> élément red</span>
```

Pour ajouter des éléments en ligne, nous n'aurons cette fois besoin que de deux méthodes : **parse<nombloc>** et **render<nombloc>**, la première étant chargée d'identifier le contenu et de réaliser le prétraitement, la seconde renvoyant le contenu à substituer.

3.2.1. La méthode parse

Notre méthode va donc s'appeler **parseRed**. Il est important de noter que cette méthode nécessite un bloc de documentation pour fonctionner. Ce bloc doit comprendre au moins le tag **@marker** et il prend la valeur de notre marqueur d'ouverture, **((** dans notre cas.

parseRed reçoit un unique argument, une ligne de Markdown à analyser. Elle renvoie un tableau dont la première entrée est également un tableau à deux entrées, et la seconde un entier. La première entrée du second tableau est le nom de notre type d'élément ; la seconde est le texte qui va remplacer cet élément si plus aucun traitement n'est nécessaire pour lui, ou encore un autre tableau dont nous allons reparler plus loin. Enfin, l'entier est la longueur de l'élément à remplacer...

Cela vous semble confus ? Tâchons d'éclaircir les choses. Notre méthode **parseRed** étant appelée dès la présence du marqueur d'ouverture, il est nécessaire de traiter le cas où aucune fermeture n'interviendrait dans la ligne. Il nous faut alors renvoyer des données pour maintenir le texte en l'état, c'est-à-dire remplacer nos deux parenthèses ouvrantes par deux parenthèses ouvrantes. Le tableau qu'il nous faut renvoyer est alors :

```
61:   return array(
62:    array('text','(('),
63:    2
64:   );
```

Il a donc deux entrées, la première étant elle-même un tableau de deux éléments. Le premier est **'text'**, le second est **'(('**. La seconde entrée du premier tableau est **2**. Nous indiquons donc par là que, à l'endroit où se trouve notre marqueur d'ouverture, il faut remplacer deux caractères par le texte **'(('**.

Voyons maintenant ce que donne notre méthode au complet :

```
47: /**
48:  * @marker ((
49:  */
50:
51: protected function
parseRed($markdown) {
52:  if (preg_match('/^\(\
((.+?)\)\)/', $markdown, $matches)) {
53:   return array(
54:    array(
55:     'red',
56:     $this->parseInline($matches[1])
57:    ),
58:     strlen($matches[0])
59:   );
60:  }else{
61:   return array(
62:    array('text','(('),
63:    2
64:   );
65:  }
66: }
```

Le plus efficace pour obtenir le texte concerné par notre marqueur est d'utiliser la fonction **preg_match** qui nécessite d'utiliser une expression régulière, toujours désagréable à lire ou à écrire lorsque l'on n'en a pas l'habitude. Ici, sa syntaxe est rendue un peu plus complexe encore du fait qu'il nous a fallu utiliser le caractère d'échappement **** pour parvenir à repérer les parenthèses. Bref, nous recevons ici dans **$matches[0]** le texte à remplacer, y compris les marqueurs d'ouverture et de fermeture, et dans **$matches[1]**, le même texte, sans les marqueurs. Si aucune occurrence n'est trouvée, nous passons à la ligne 61, où nous retrouvons le tableau que nous avons déjà analysé. Le tableau que nous renvoyons si un résultat est obtenu a la même structure. Le changement le plus important est que le texte de substitution a été remplacé par **$this->parseInline($matches[1])** à la ligne 56. Ceci nous renvoie un tableau, dont nous n'avons pas à étudier le contenu ici. Mais cela nous permet de poursuivre l'analyse du Markdown à l'intérieur de notre élément.

Si nous souhaitions que le contenu de notre élément reste intact, nous aurions dû écrire :

```
53:    return array(
54:      array(
55:       'text',
56:       '<span style="red">'.$matches[1].
'</span>'
57:      ),
58:       strlen($matches[0])
59:    );
```

Dans ce cas-là, la méthode **renderRed** que nous allons étudier maintenant ne serait pas nécessaire.

3.2.2. La méthode render

La méthode **renderRed** est étonnamment simple au regard de ce qui précède :

```
68: protected function renderRed($element){
69:   return '<span style="red">'.$this->
renderAbsy($element[1]).'</span>';
70: }
```

Il y a peu de choses à en dire. Si ce n'est que la variable **$element** contient l'*AST*, l'*Abstract Syntax Tree*, l'arbre abstrait de l'analyse syntaxique, sous forme de tableaux imbriqués, à partir de notre élément. L'élément d'index **1** du plus haut tableau contient le niveau inférieur : le fait de le passer à la méthode **renderAbsy** permet l'analyse récursive du Markdown dans notre contenu.

CONCLUSION

Vous voilà maintenant armé pour faire rager les puristes en multipliant les parfums de Markdown à l'envi. Sachez tout de même que vous pouvez pousser les choses encore plus loin avec Cebe Markdown : en effet, il est possible de panacher les différentes saveurs. Les classes de base sont en effet construites à partir de *traits*, solution de PHP au problème de l'héritage multiple, et il vous est possible d'importer dans votre propre classe telle méthode depuis **GithubMarkdown** et telle autre depuis **MarkdownExtra**. Apprenez-en plus dans la documentation officielle **[2]**. ■

RÉFÉRENCES

[1] PHP Markdown Extra : https://michelf.ca/projects/php-markdown/extra

[2] Cebe Markdown : https://github.com/cebe/markdown

ADDRESS SPACE LAYOUT RANDOMIZATION

SYLVAIN NAYROLLES
[Security Software Engineer]

MOTS-CLÉS : SÉCURITÉ, RELOCATION, ELF, LINK, PIE, PIC

Brad Spengler, le leader du projet grsecurity, exprimait, lors de son intervention au SSTIC 2016, son agacement à lier la sécurité d'une application à son implémentation. Pour lui, la sécurisation d'une application ne doit pas se faire via la recherche de bugs dans cette dernière, mais doit être assurée par la plateforme sur laquelle elle s'exécute. L'ASLR fait partie de ces évolutions proposées par les systèmes d'exploitation, dans le but de complexifier l'exploitation de failles applicatives.

Au milieu des années 2000, l'exploitation de failles applicatives en *userland* était relativement simple, car il n'existait que très peu de mécanismes de protection. Ces dernières sont venues initialement des fondeurs avec le DEP, *Data Protection Execution*, visant à partitionner l'espace mémoire en pages exécutables ou non d'une application. Notamment, les zones mémoires hébergeant la pile ou le tas ne sont plus exécutables. Pour contourner ces protections, une nouvelle génération d'exploit a vu le jour, visant à utiliser le code exécutable présent dans l'application défaillante en modifiant, par exemple, le flux d'exécution vers des fonctions connues de la **libc**. Cette technique d'exploitation, connue sous le nom de **return to libc**, repose sur le fait que l'adresse de la fonction visée est prédictible à l'exécution de l'application. L'*Address Space Layout Randomization*, ou ASLR constitue une réponse intéressante du système d'exploitation, en positionnant de façon aléatoire l'adresse des segments exécutables de l'application et de ses bibliothèques.

1. MODÈLE EXÉCUTABLE

Pour comprendre les mécanismes mis en jeu dans l'ASLR, il est important de comprendre le modèle d'un exécutable. Lors de la compilation, un exécutable est divisé en sections. Chaque section symbolise un type de donnée. Les principales sections sont :

- **.text** qui regroupe l'ensemble du code exécutable de l'application ;
- **.rodata** pour *read only data*, regroupant les constantes du programme ;
- **.bss** qui regroupe toutes les variables globales du programme ;
- **.got** pour *Global Offset Table* sur laquelle nous reviendrons dans la section suivante ;
- **.plt** pour *Procedure Linkage Table* sur laquelle nous reviendrons également dans la section suivante.

Il en existe bien d'autres, chacune avec leurs usages spécifiques. Pour lister les sections d'un exécutable, il suffit d'utiliser **readelf**, un logiciel, qui comme son nom l'indique, permet de lire le format ELF. Nous allons lui demander de lire les sections de l'exécutable **ls** :

NOTE

Le terme de segment peut ici nous induire en erreur. Le modèle de sécurité mémoire repose sur deux concepts fondamentaux, rarement associés, souvent opposés : la segmentation et la pagination. Certains systèmes utilisent intelligemment une combinaison des deux. Pour des raisons de portablilté, Linux a fait le choix de ne pas utiliser la segmentation, et de privilégier la pagination, essentiellement pour des raisons de portabilité entre processeurs. Ici donc, le terme de segment ne se réfère en aucun cas à ce mécanisme de protection mémoire.

```
$ readelf --sections /bin/ls
Section Headers:
[Nr] Name              Type            Address          Offset
     Size              EntSize         Flags  Link  Info  Align
[12] .plt              PROGBITS        0000000000402190  00002190
     0000000000000710  0000000000000010  AX       0     0     16
[13] .text             PROGBITS        00000000004028a0  000028a0
     000000000000fc29  0000000000000000  AX       0     0     16
[15] .rodata           PROGBITS        00000000004124e0  000124e0
     000000000000699c  0000000000000000   A       0     0     32
[22] .got              PROGBITS        000000000061bff8  0001bff8
     0000000000000008  0000000000000008  WA       0     0     8
[23] .got.plt          PROGBITS        000000000061c000  0001c000
     0000000000000398  0000000000000008  WA       0     0     8
[24] .data             PROGBITS        000000000061c3a0  0001c3a0
     0000000000000254  0000000000000000  WA       0     0     32
[25] .bss              NOBITS          000000000061c600  0001c5f4
     0000000000000d60  0000000000000000  WA       0     0     32
```

Ici, j'ai volontairement supprimé certaines sections pour ne garder que celles qui nous intéresseront par la suite. Ces sections sont ensuite regroupées par segments. Ces derniers vont posséder des droits particuliers en lecture, écriture ou exécution. Lors du chargement de l'exécutable, le *loader* va allouer à chacun un *pool* de page auquel il appliquera les droits associés.

Pour lister les segments, et l'association section segment, nous allons une nouvelle fois utiliser **readelf** :

```
$ readelf --segments /bin/ls

Elf file type is EXEC (Executable file)
Entry point 0x404870
There are 9 program headers, starting at offset 64

Program Headers:
 Type           Offset             VirtAddr           PhysAddr
                FileSiz            MemSiz              Flags  Align
 PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                0x00000000000001f8 0x00000000000001f8  R E    8
 INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238
                0x000000000000001c 0x000000000000001c  R      1
     [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
 LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                0x000000000001bc2c 0x000000000001bc2c  R E    200000
 LOAD           0x000000000001bdf0 0x000000000061bdf0 0x000000000061bdf0
                0x0000000000000804 0x0000000000001570  RW     200000
 DYNAMIC        0x000000000001be08 0x000000000061be08 0x000000000061be08
                0x00000000000001f0 0x00000000000001f0  RW     8
 NOTE           0x0000000000000254 0x0000000000400254 0x0000000000400254
                0x0000000000000044 0x0000000000000044  R      4
 GNU_EH_FRAME   0x0000000000018e7c 0x0000000000418e7c 0x0000000000418e7c
                0x0000000000000734 0x0000000000000734  R      4
 GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                0x0000000000000000 0x0000000000000000  RW     10
 GNU_RELRO      0x000000000001bdf0 0x000000000061bdf0 0x000000000061bdf0
                0x0000000000000210 0x0000000000000210  R      1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym
.dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt .init .plt .text
.fini .rodata .eh_frame_hdr .eh_frame
```

```
   03     .init_array .fini_array .jcr .dynamic .got
.got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
   08     .init_array .fini_array .jcr .dynamic adelf
--relocs /bin/cat.got
```

Nous pouvons donc visualiser les droits associés aux segments ainsi que leurs adressages virtuels. Si nous prenons l'exemple du segment **02** (attention, la numérotation commence à **0**), son adresse virtuelle est **0x40000**, sa taille est de **0x1bc2c** et ses droits sont **R** (*Read*) et **E** (*Execute*). Si nous regardons quelles sections lui sont associées, nous retrouvons la section **.text** regroupant l'ensemble du code. D'ailleurs, le point d'entrée du programme se situe à l'adresse **0x404870**, soit en plein dans ce segment, ce qui confirme notre analyse précédente. Si maintenant nous examinons le segment **03**, nous constatons qu'il possède les droits **R** (*Read*) et **W** (*Write*). Nous retrouvons la section **.data** de notre exécutable.

Sans oublier les segments alloués dynamiquement, tels que le tas et la pile, qui eux aussi initialisent le **NX bit** (*Non executable bit*), des pages associées à leur espace mémoire.

Ce modèle fut donc introduit dans les années 2000 sous le terme de DEP pour *Data Execution Prevention*, permettant de limiter la surface d'exploitation des programmes en *userland*. En effet, grâce à cette protection, il n'était plus possible d'injecter un shellcode dans l'espace mémoire d'un processus.

Une nouvelle génération d'exploit fût donc initiée, reposant principalement sur l'exécution de codes déjà présents dans les segments exécutables : les *heap overflow*, *stack overflow*, tous permettant de modifier le flux d'exécution du programme vers une adresse connue de l'espace exécutable du logiciel défaillant. La plus connue d'entre elles est la technique dite du *return to libc*, afin d'exécuter des fonctions de type *system*.

Toutes ces techniques reposent sur l'aspect prédictible des adresses des sections exécutables. Une solution fut donc d'introduire de l'aléa dans le chargement des bibliothèques tierces.

2. ASLR

L'ASLR, pour *Adress Space Layout Randomization*, fut une des avancées majeures des systèmes d'exploitation, concernant la sécurisation des applications dont ils avaient la charge. Il permet donc de charger un segment à une adresse non connue de l'application en amont. Nous distinguerons dans la suite, les segments appartenant aux bibliothèques chargées dynamiquement, des segments appartenant au programme principal.

2.1 Relocation

L'ASLR repose essentiellement sur le concept dit de relocation. La relocation est un mécanisme permettant au *loader* de recalculer l'adresse d'un symbole dans l'espace d'adressage virtuel de certaines sections. Il existe différents types de relocation, et donc de calculs associés. Nous verrons dans la suite de cet article que certaines d'entre elles sont compatibles avec un chargement aléatoire. Pour connaître les symboles et les types de relocation associés, il suffit d'utiliser soit **objdump**, un utilitaire permettant de *dumper* les sections d'un exécutable et de produire le code assembleur associé, soit **readelf** permettant aussi de *dumper* les sections d'un ELF, mais cette fois sans désassembler :

```
$ readelf --relocs /bin/cat
```

La notion de relocation de code est antérieure à l'ASLR, car la relocation est très utilisée aussi lors de l'étape de *link* de l'exécutable, et donc de la résolution des symboles externes d'une application. N'oublions pas que nous pouvons résoudre un symbole de façon statique. Nous pouvons nous référer à la spécification de l'**ABI SystemV [1]** pour explorer les différentes techniques de relocation.

2.2 Position Independent Code

Le *Position Independent Code*, ou PIC, permet de générer une bibliothèque dynamique, dont certains segments critiques, tel que celui contenant le code exécutable, pourra être chargé n'importe où au sein de l'espace d'adressage de l'exécutable principal.

Afin d'illustrer notre propos, nous allons observer l'espace d'adressage des différents segments utilisés lors de l'exécution de notre commande courante. Pour cela, nous allons utiliser la richesse du procfs et du pseudo fichier **maps** :

```
$ cat /proc/self/maps
```

En réalisant cette commande, nous allons afficher les segments de notre processus **cat** qui nous sert à afficher...

Si nous observons le résultat de la commande pour les segments du tas, de la pile et du segment exécutable de la **libc** :

```
019a6000-019c7000 rw-p 00000000 00:00 0
[heap]
7f66a9b84000-7f66a9d44000 r-xp 00000000
08:07 528906                          /lib/
x86_64-linux-gnu/libc-2.21.so
7fffea3d5000-7fffea3f6000 rw-p 00000000
00:00 0                               [stack]
```

Nous allons refaire notre commande :

```
020a1000-020c2000
rw-p 00000000 00:00 0
[heap]
7fef1f4ba000-
7fef1f67a000 r-xp
00000000 08:07 528906
/lib/x86_64-linux-gnu/
libc-2.21.so
7ffe1ba01000-7ffe1ba22000
rw-p 00000000 00:00 0
[stack]
```

Les espaces d'adressage sont totalement différents et notre programme tourne toujours. Ceci grâce à la magie du PIC et de l'ASLR.

2.2.1 Résolution locale

Nous allons essayer de comprendre comment ça marche. Pour cela, nous allons construire notre analyse autour d'un code source minimaliste, composé d'une bibliothèque dynamique avec une fonction exportée, et d'un programme principal appelant cette dernière.

Nous allons créer un fichier ne contenant que la fonction **maps** permettant d'afficher la commande précédente dans le contexte de notre processus :

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
void maps(void) {
 char command[1024] = "";
 snprintf(command, 1024, "cat
/proc/%d/maps", getpid());
 // print maps
 system(command);
}
```

Nous allons compiler le fichier via les commandes :

```
$ gcc -c maps.c -o maps.o
$ gcc -shared -o libmaps.so
maps.o
/usr/bin/ld: maps.o:
relocation R_X86_64_32
against '.rodata' can not
be used when making a shared
object; recompile with -fPIC
```

Le *linker*, ici **ld** nous interdit de créer notre bibliothèque dynamique à partir de notre fichier objet, et nous indique qu'il faut utiliser l'option **-fPIC** pour la générer. L'erreur provient de l'utilisation de la constante **"cat /proc/%d/maps"**. Cette chaîne de caractères se situe dans la section **.rodata**.

Prenons un peu le temps pour comprendre le code généré. Pour cela, nous allons analyser la table de relocation de notre fichier :

```
$ readelf --relocs maps.o
Relocation section '.rela.text' at offset 0x338 contains 5 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
00000000003d  000a00000002 R_X86_64_PC32    0000000000000000 getpid - 4
00000000004d  00050000000a R_X86_64_32     0000000000000000 .rodata + 0
00000000005f  000b00000002 R_X86_64_PC32    0000000000000000 snprintf - 4
00000000006e  000c00000002 R_X86_64_PC32    0000000000000000 system - 4
000000000083  000d00000002 R_X86_64_PC32    0000000000000000 __stack_chk_fail - 4
```

Nous voyons apparaître notre symbole **.rodata**, référençant la section associée, qui possède un type de relocation **R_X86_64_32**. Si nous consultons la spécification de l'ABI SystemV **[1]**, celle qui spécifie aussi le format ELF, nous constatons que ce type de relocation est simple, et consiste juste à remplacer la valeur à l'offset **0x4d** dans le code par la valeur de l'adresse de la section **.rodata**.

Si nous analysons le code assembleur généré :

```
$ objdump -D maps.o
...
4c:   ba 00 00 00 00          mov    $0x0,%edx
```

En **0x4d**, en observe la valeur zéro. La phase de relocation sera donc appliquée lors de la résolution des symboles par le *linker*.

Cette adresse doit donc être connue lors du *link*, ce qui rend incompatible le fait que cette adresse peut être aléatoire comme dans le cas de l'ASLR.

Nous allons donc regénérer notre bibliothèque cette fois-ci avec l'option **-fPIC** :

```
$ gcc -c -fPIC maps.c -o maps.o
$ gcc -shared -o libmaps.so maps.o
```

Si nous procédons à la même analyse que précédemment :

```
$ readelf --relocs maps.o
Relocation section '.rela.text' at offset 0x368 contains 5 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
00000000003d  000b00000004 R_X86_64_PLT32   0000000000000000 getpid - 4
00000000004f  000500000002 R_X86_64_PC32    0000000000000000 .rodata - 4
000000000061  000c00000004 R_X86_64_PLT32   0000000000000000 snprintf - 4
000000000070  000d00000004 R_X86_64_PLT32   0000000000000000 system - 4
000000000085  000e00000004 R_X86_64_PLT32   0000000000000000 __stack_chk_fail - 4
```

Les types de relocation associés à nos symboles ont totalement changé.

Si nous analysons le code généré dans le fichier objet :

```
$ objdump -D maps.o
...
4c:   48 8d 15 00 00 00 00    lea    0x0(%rip),%rdx
```

Pour retrouver l'adresse du segment **.rodata**, nous utilisons maintenant un offset relatif à la valeur du registre **rip** (*instruction pointer*).

Réalisons maintenant l'analyse non plus sur le fichier objet, mais sur la bibliothèque dynamique associée :

```
$ objdump -D libmaps.so
...
82c:    48 8d 15 42 00 00 00     lea     0x42(%rip),%rdx
```

Nous voyons donc que l'adresse de notre constante est calculée par rapport à la valeur du pointeur d'instruction. En effet, les sections **.text** et **.rodata** appartiennent au même segment et il est donc possible d'utiliser un offset :

```
$ readelf --segments libmaps.so

Elf file type is DYN (Shared object file)
Entry point 0x6e0
There are 7 program headers, starting at offset 64

Program Headers:
 Type   Offset             VirtAddr           PhysAddr
        FileSiz            MemSiz              Flags  Align
 LOAD   0x0000000000000000 0x0000000000000000 0x0000000000000000
        0x000000000000090c 0x000000000000090c  R E    200000

 Section to Segment mapping:
  Segment Sections...
   00     .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.
version .gnu.version_r .rela.dyn .rela.plt .init .plt .text
.fini .rodata .eh_frame_hdr .eh_frame
```

Théoriquement, l'adresse de notre constante se calcule de la façon suivante :

adresse de la donnée = RIP + offset

Lorsque le segment exécutable est chargé à l'adresse **0**, la valeur du **RIP** à ce moment de l'exécution vaut :

RIP = 0x82C + 0x7

En effet, le **RIP** pointe toujours vers l'instruction suivante, donc nous ajoutons **7** qui est la taille de l'instruction courante.

offset = 0x42

Ce qui donne comme adresse **RIP + offset = 0x875**. Observons ce qui se trouve en **0x875** :

```
$ objdump -s libmaps.so
Contents of section .rodata:
 0875 63617420 2f70726f 632f2564 2f6d6170  cat
/proc/%d/map
 0885 7300
```

Nous retrouvons bien notre chaîne de caractères. Notre section peut donc être chargée n'importe où dans l'espace d'adressage virtuel, la relocation appliquée est compatible.

NOTE

Mais pourquoi ne pas réaliser ce type de relocation à chaque fois ? Tout simplement pour des histoires d'optimisation. Le fait de calculer une adresse par rapport à la valeur du pointeur d'instruction coûte plus cher en terme de cycle CPU.

Ceci va nous permettre de résoudre nos symboles de façon interne à notre bibliothèque, mais qu'en est-il de ceux qui doivent être appelés depuis un exécutable, qui ne connaît donc pas l'adresse de chargement du segment de la bibliothèque ?

2.2.2 Lazy Binding

Nous allons voir que sous Linux la résolution des symboles dynamiques se fait de façon très pertinente.

Reprenons notre bibliothèque compilée avec l'option **-fPIC**, et appelons-la de la façon la plus simple :

```
#include "maps.h"
int main(int argc, char** argv) {
 maps();
}
```

```
$ gcc main.c -o aslr -L. -lmaps
```

Si nous désassemblons la section de code de notre exécutable, responsable de l'appel à la fonction **maps** :

```
4006a5:       e8 e6 fe ff ff          callq
400590 <maps@plt>
```

Ce dernier appelle un code se situant dans la section **.plt** pour *Procedure Linkable Table*. Cette section porte mal son nom, car si nous l'examinons de près nous constatons que cette dernière est une section exécutable. Nous allons donc la désassembler à son tour :

```
Disassembly of section .plt:

0000000000400560 <__libc_start_main@plt-0x10>:
  400560:       ff 35 a2 0a 20 00       pushq
0x200aa2(%rip)        # 601008 <_fini+0x2008d4>
  400566:       ff 25 a4 0a 20 00       jmpq
*0x200aa4(%rip)        # 601010 <_fini+0x2008dc>
  40056c:       0f 1f 40 00             nopl   0x0(%rax)

...

0000000000400590 <maps@plt>:
  400590:       ff 25 92 0a 20 00       jmpq
*0x200a92(%rip)        # 601028 <_fini+0x2008f4>
  400596:       68 02 00 00 00          pushq  $0x2
  40059b:       e9 c0 ff ff ff          jmpq   400560
<_init+0x20>
```

L'instruction pointée par la commande **call**, fait directement un *jump* relatif vers une autre section, la **.got.plt**. Cette dernière est une section *Read Write* non exécutable. La valeur à l'offset **0x601028**, référencée par la **.plt**, est **0x400596** soit l'adresse suivante à notre *jump* dans la section **.got.plt**. Si nous continuons l'analyse du code de la **.plt**, nous voyons qu'il appelle le code de la première entrée de la **.plt** en poussant la valeur **2** sur la pile.

La première entrée de la **.plt** est en fait le code responsable de la résolution de l'adresse de la fonction.

Mais alors pourquoi faire un saut vers la section **.got.plt** avant ?

Nous allons donc réaliser une analyse dynamique de ce qui se passe, avec **gdb**. Ce qui nous intéresse c'est la valeur présente dans la **.got** à l'adresse **0x601028**. Nous allons donc positionner un point d'arrêt avant l'appel à la fonction **maps** et un juste après, et nous allons afficher la valeur de la **.got**.

```
$ gdb ./aslr
(gdb) b *0x4006a5
(gdb) b* 0x4006aa
(gdb) run
(gdb) x /g 0x601028
0x601028 <maps@got.plt>:
0x0000000000400596
(gdb) continue
(gdb) x /g 0x601028
0x601028 <maps@got.plt>:
0x00007ffff7bd77e0
```

Nous constatons que cette dernière a changé. Mais à qui appartient l'adresse **0x00007ffff7bd77e0** ?

```
(gdb) info files
...
0x00007ffff7bd76e0 -
0x00007ffff7bd786b is
.text in ./libmaps.so
```

C'est une adresse appartenant directement à la section de code associée à notre bibliothèque. Si nous poussons notre analyse, nous verrons que c'est bien l'adresse de notre fonction **maps**.

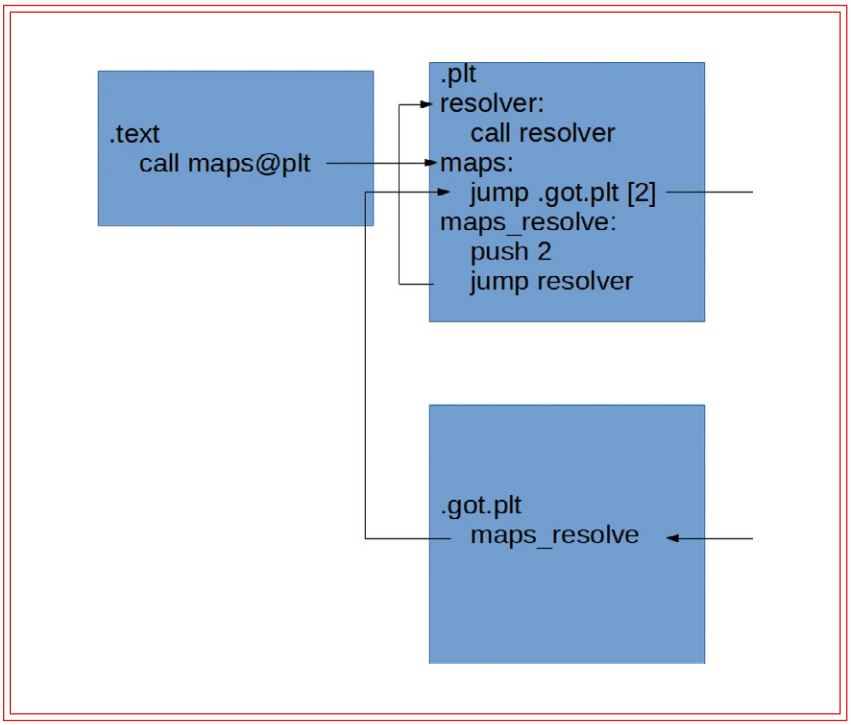


Fig. 1 : Schéma des sections .text .plt et .got.plt lors du chargement de l'exécutable avant résolution du symbole maps.

Lors de la résolution, le *loader* a remplacé la valeur initiale présente dans la **.got** par la vraie valeur de la fonction ciblée. Les prochains appels à **maps** ne feront donc pas appel au résolveur, d'où l'appellation de *Lazy Binding* ou résolution feignante.

Si nous revenons au code présent dans la section **.plt**, nous avons observé que juste avant d'appeler le code du résolveur, on a poussé une valeur sur la pile, en l'occurrence la valeur **2** pour la fonction **maps**. Cette valeur est en fait un index dans la table de relocation :

```
$ readelf --relocs aslr
Relocation section '.rela.plt' at offset 0x4f8 contains 3 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000601018 000200000007 R_X86_64_JUMP_SLO 0000000000000000 __libc_start_main + 0
000000601020 000300000007 R_X86_64_JUMP_SLO 0000000000000000 __gmon_start__ + 0
000000601028 000400000007 R_X86_64_JUMP_SLO 0000000000000000 maps + 0
```

Nous retrouvons comme valeur d'index l'adresse de la **.got** qui va permettre de réaliser le *lazy bindings* (voir figure 2, page suivante).

Mais ce modèle n'est pas encore parfait, car certaines sections du programme principal, comme la section **.text**, ont toujours une adresse prédictible, et donc vulnérable à des attaques plus sophistiquées comme des exploitations à base de ROP (*Return Oriented Programming*).

2.3 Position Independent Executable

Le PIE est le pendant du PIC, mais cette fois-ci pour les segments du programme principal. Si nous reprenons notre exemple précédent, nous devons cette fois-ci spécifier l'option **-pie** et **-fPIC** :

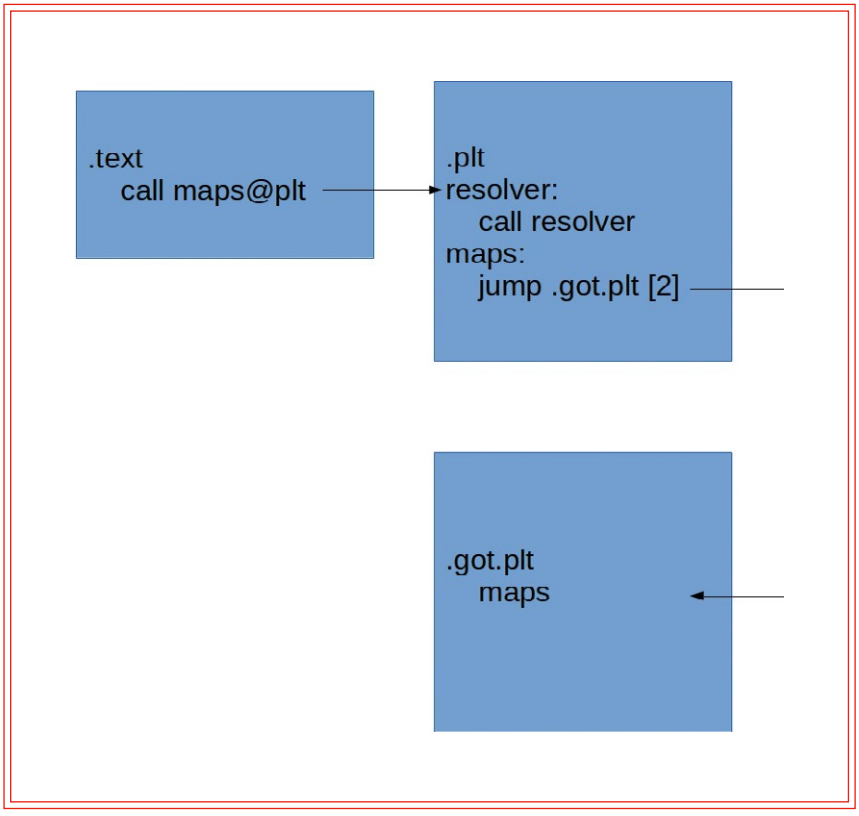


Fig. 2 : Schéma des sections .text .plt et .got.plt après résolution du symbole maps.

```
$ gcc main.c -o aslr -L. -lmaps -pie -fPIC
```

Si nous exécutons notre simple programme :

```
$ ./aslr
560a35cf3000-560a35cf4000 r-xp 00000000 08:07 7083250
/home/sylvain/dev/articles/Linux Magazine/ASLR/src/aslr
560a35ef3000-560a35ef4000 r--p 00000000 08:07 7083250
/home/sylvain/dev/articles/Linux Magazine/ASLR/src/aslr
560a35ef4000-560a35ef5000 rw-p 00001000 08:07 7083250
/home/sylvain/dev/articles/Linux Magazine/ASLR/src/aslr
7f079dc9b000-7f079dc9c000 r-xp 00000000 08:07 7083147
/home/sylvain/dev/articles/Linux Magazine/ASLR/src/libmaps.so
7f079dc9c000-7f079de9b000 ---p 00001000 08:07 7083147
/home/sylvain/dev/articles/Linux Magazine/ASLR/src/libmaps.so
7f079de9b000-7f079de9c000 r--p 00000000 08:07 7083147
/home/sylvain/dev/articles/Linux Magazine/ASLR/src/libmaps.so
7f079de9c000-7f079de9d000 rw-p 00001000 08:07 7083147
/home/sylvain/dev/articles/Linux Magazine/ASLR/src/libmaps.so
...
```

Nous constatons cette fois-ci que tous les segments sont bien situés à des emplacements aléatoires. Si nous analysons les sections de code présentes dans le ELF, nous constatons qu'elles ont toutes une valeur nulle comme adresse de chargement.

Mais alors, pourquoi ne pas compiler tous les programmes avec cette option magique ?

Eh bien, cette option a un coût non négligeable lors du chargement de l'exécutable. En effet, le chargeur doit réécrire les adresses présentes dans les sections « relogeables ». Ceci constitue un surcoût de 30 % environ lors du chargement de chaque exécutable.

C'est pour cela que la majorité des programmes standards présents dans une distribution Linux ne sont pas compilés avec une telle option (environ 5 % pour une distribution comme **Debian**). Par exemple, un programme comme **sshd** lui est bien compilé avec PIE.

CONCLUSION

Il est important de comprendre comment notre programme est compilé pour savoir comment il s'exécute. Si l'option PIC est incontournable, car obligatoire, pour la génération de bibliothèques dynamiques, la génération d'un programme PIE demande une certaine expertise. Par exemple, est-il vraiment pertinent de compiler un utilitaire tel que `ls` avec le PIE, sachant que ce dernier développe une surface d'attaque limitée pour une fréquence de chargement élevée ? Par contre, dans le cas d'un logiciel *daemon* qui n'est chargé qu'une seule fois, le PIE apporte une réelle protection. L'ASLR apporte une protection efficace contre les attaques de type *Return Oriented Programming* (ROP), à condition que le programme ciblé soit bien protégé. ■

RÉFÉRENCE

[1] Documentation de *l'ABI SystemV* : http://refspecs.linuxbase.org/

SERVEURS DÉDIÉS Synology®

AVEC ikoula HÉBERGEUR CLOUD

Votre serveur dédié de stockage (NAS) hébergé dans nos Data Centers français.

POUR LES LECTEURS DE **LINUX MAG***

OFFRE SPÉCIALE -60 %

À PARTIR DE

5,99 € HT/MOIS

~~14,99 €~~

CODE PROMO **SYLIM17**

- Bande passante **100 Mbit/s**
- Station de **surveillance**
- Support technique **en 24/7**
- Trafic réseau **illimité**
- Système d'exploitation **DSM 6.0**
- Hébergement dans **nos Data Centers**

*Offre spéciale -60 % valable sur la première période de souscription avec un engagement de 1 ou 3 mois. Offre valable jusqu'au 31 décembre 2017 23h59 pour une seule personne physique ou morale, et non cumulable avec d'autres remises. Prix TTC 7,19 €. Par défaut les prix TTC affichés incluent la TVA française en vigueur.

CHOISISSEZ VOTRE NAS

https://express.ikoula.com/promosyno-lim

ikoula HÉBERGEUR CLOUD — /ikoula — @ikoula — sales@ikoula.com — 01 84 01 02 50

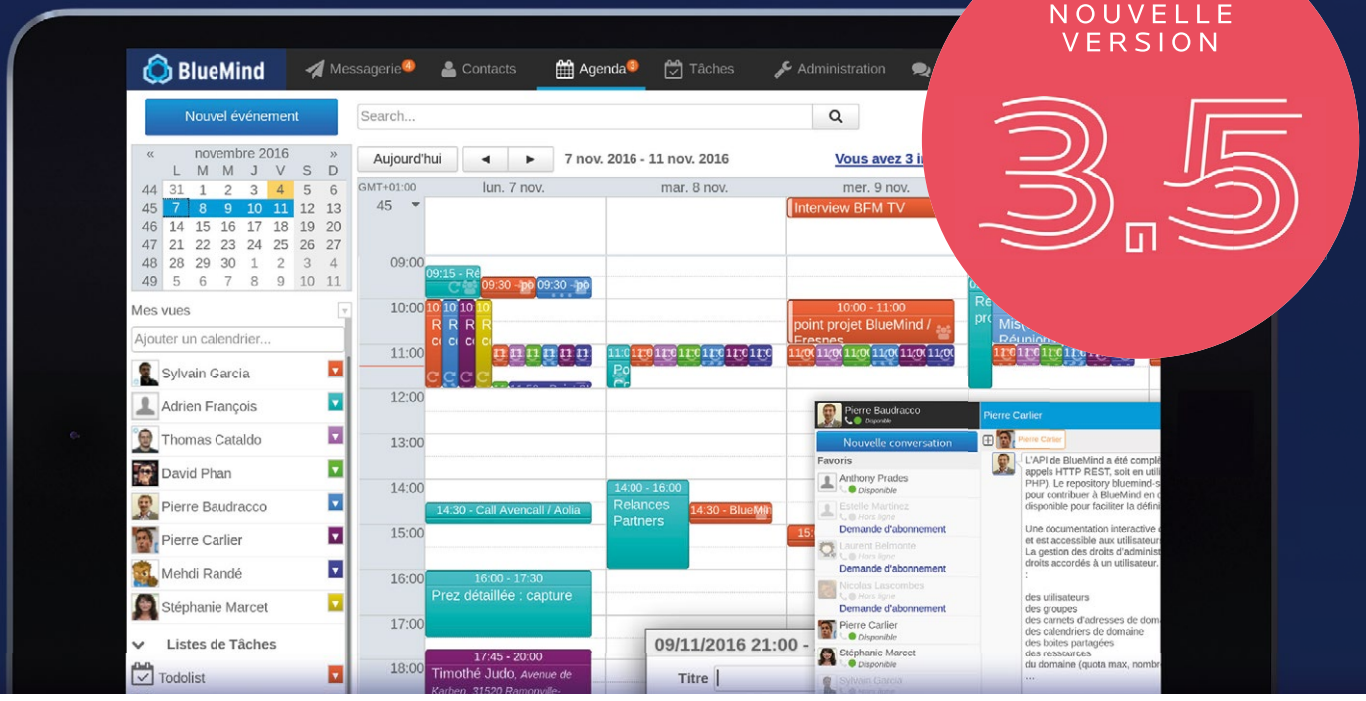
NOM DE DOMAINE | HÉBERGEMENT WEB | SERVEUR VPS | SERVEUR DÉDIÉ | CLOUD PUBLIC | MESSAGERIE | STOCKAGE | CERTIFICATS SSL

BlueMind

SOLUTION OPENSOURCE
PROFESSIONNELLE DE MESSAGERIE
COLLABORATIVE

LIBÉREZ VOTRE MESSAGERIE

EMAILS
AGENDAS
CONTACTS
TÂCHES
CHAT
MOBILES
DOCUMENTS & PIÈCES JOINTES
ENVOI FICHIERS VOLUMINEUX
...
...



NOUVELLE VERSION
3.5

FRANCAIS / NOMBREUSES RÉFÉRENCES / ERGONOMIQUE / ÉVOLUTIF / ÉCONOMIQUE

Découvrez l'écosystème BlueMind et toutes les fonctionnalités sur
www.bluemind.net