GNU LINUX MAGAZINE / FRANCE

DÉVELOPPEMENT SUR SYSTÈMES UNIX, OPEN SOURCE & EMBARQUÉ

N°205
JUIN 2017

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

L 19275 - 205 - F: 7,90 € - RD

Algo / Python

PRENEZ-VOUS POUR DIEU : CRÉEZ LA VIE ! p.24

- Comprenez l'algorithme du jeu de la vie de Conway
- Implémentez-le en Python avec une interface graphique
- Modifiez ses règles pour en faire un robot exécutant des tâches

Sécurité / Programmation

DIRTY COW : COMPRENEZ L'UNE DES FAILLES LES PLUS IMPORTANTES DE CES DERNIÈRES ANNÉES p.94

C / XML

CONCEVEZ VOS INTERFACES GRAPHIQUES AVEC GLADE ET UTILISEZ-LES EN C p.78

Compilation croisée / Embarqué

METTEZ EN PLACE UN CONTRÔLEUR CAN MCP2515 DANS UNE RASPBERRY PI p.42

Science / Julia

CONSTRUISEZ VOTRE MOTEUR PHYSIQUE 2D p.10

Bas niveau / Binaire

DÉCOUVREZ QU'ON PEUT AUSSI MANIPULER DES BITS EN PYTHON p.56

CLOUD COMPUTING - SCROLLING D'UN TEXTE SUR ÉCRAN DE LEDS WS2812...

* 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

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.61/62 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

On ne peut pas gagner à tous les coups ! C'est le constat qu'il m'a malheureusement fallu faire en essayant de lutter une énième fois contre l'obsolescence programmée... Comme toujours avec les tablettes et autres smartphones, la batterie est la première à donner des signes de faiblesse. En l'occurrence, sur une tablette de 5 ans, c'est tout à fait normal. Ce qui l'est moins, c'est de ne pas proposer de batterie amovible ! Alors bien sûr, sur une tablette on ne peut pas faire autrement (je ne connais aucun modèle avec batterie amovible). On peut se dire que le changement de batterie ne sera pas à la portée de tout le monde, mais que l'on peut raisonnablement penser qu'avec deux ou trois notions d'électronique on devrait arriver à démonter et remonter ladite tablette. La décision prise, on peut découper le scénario de la réparation en 4 phases :

Phase 1 - la planification : il faut trouver où acheter la batterie d'origine ou de remplacement en se méfiant des mauvaises imitations et s'équiper en outils permettant de démonter la tablette (de petits outils en plastique pour extraire la partie électronique de sa gangue de plastique sans tout abîmer).

Phase 2 - l'attente : il y a de gros risques pour que la batterie ait un long parcours en bateau à effectuer et il faudra donc être patient.

Phase 3 - l'action : vous avez reçu tous les éléments et vous prenez une demi-heure pour changer cette satanée batterie qui ne tient plus qu'une demi-heure et s'arrête sans avertissement. Premier constat : même pour des outils en plastique il faut choisir quelque chose de qualité ! À la moitié du démontage l'outil casse et il faut donc continuer... comme on peut...

Il suffit ensuite de retirer des vis et du scotch protégeant des nappes, et d'ouvrir les connecteurs de nappe pour retirer lesdites nappes. Oui, mais là aussi il y a un problème : après 5 ans d'utilisation, que devient une petite pièce en plastique située à côté d'un radiateur ? Elle devient cassante ! J'ai donc pu retirer trois nappes sur quatre, mais ce n'est malheureusement pas suffisant pour faire fonctionner la tablette.

Phase 4 - la résignation : il va falloir changer de tablette. Bien entendu il serait possible de passer du temps à bidouiller le connecteur, mais pour ça il faut justement avoir du temps... En attendant, la tablette sera donc remisée dans un coin jusqu'à ce que la prochaine la remplace (il ne faut pas être exagérément optimiste sur le temps dont on va disposer pour réparer une tablette que l'on n'utilise plus).

Bilan de l'opération : les constructeurs sont vraiment très forts et produisent des objets qui restent réparables le temps de la garantie, mais qui le sont beaucoup plus difficilement par la suite. J'en avais d'ailleurs fait également l'expérience avec une souris pour gamer qui au bout d'un an et demi d'utilisation a dû être mise au rebut (après une réparation de fortune qui a tenu 6 mois). Bilan connexe : finalement, je n'ai vraiment pas besoin d'une tablette ! :-)

Profitez donc de votre magazine, son contenu se déprécie beaucoup moins rapidement que tout le matériel électronique que vous utilisez !

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**

+

- **73 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 « GLMF205 » 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°205

CLOUD COMPUTING : DÉVELOPPER DES TECHNOLOGIES ET SERVICES NATIFS

RODRIGUE CHAKODE
[PhD en informatique, spécialiste en supervision IT et cloud, auteur de RealOpInsight]

MOTS-CLÉS : CLOUD COMPUTING, ÉCHANGE DE MEILLEURES PRATIQUES, INTEROPÉRABILITÉ, PASSAGE À L'ÉCHELLE, AUTO-SUPERVISION

Comme tendance, nous vous proposons de découvrir l'initiative The Cloud Native Computing Foundation, CNCF [1]. Cette initiative se construit avec pour ambition de créer et conduire l'adoption d'un nouveau paradigme de cloud, optimisé, interopérable, et capable de passer à l'échelle sur des infrastructures modernes composées de milliers de nœuds de traitements. Dans cet article, nous vous présenterons sa philosophie et ses moyens d'action. Mais avant cela, nous vous présenterons en introduction les enjeux et les facteurs qui ont motivé une telle initiative.

Avec l'émergence du *cloud computing* au cours de la dernière décennie, **Amazon Web Services** avec ses différentes offres de services **[2]**, a fortement contribué à la vulgarisation et l'adoption de ce paradigme. Beaucoup d'entreprises, petites et grandes, et même des particuliers, ont très rapidement suivi cette grande tendance. Qui dit tendance, dit une certaine célérité pour suivre le marché. Tant bien que mal. Car dans beaucoup de cas, il suffisait par exemple d'avoir des applications tournant dans une machine virtuelle chez Amazon ou sur une infrastructure de cloud privé, type **OpenStack [3]**, **OpenNebula [4]**, ou **VMware vCloud [5]**, pour se prévaloir d'une offre de services en mode *cloud*. Ce fut l'ère du buzz !

Le temps a passé, et on a pu se rendre compte qu'au-delà de la nouveauté et des slogans commerciaux qui l'ont accompagné, il n'est pas toujours simple de garantir certains éléments essentiels attendus d'une application *cloud*. C'est en particulier le cas lorsqu'on pense au

passage à l'échelle et l'interopérabilité *multicloud*. L'interopérabilité est le vrai maillon faible du *cloud* avec un risque élevé d'enfermement propriétaire (*vendor lock-in* **[6]**). Nous entendons par là la difficulté de pouvoir simplement migrer ses applications d'une infrastructure *cloud* à une autre avec un surcoût d'intégration faible, voire nul.

Place au réalisme, avec pour enjeu d'être capable de bâtir des solutions *clouds* dites « natives », qui répondent entièrement aux exigences du paradigme *cloud*, en ciblant en particulier la performance, l'interopérabilité et le passage à l'échelle. Il s'agit donc de concevoir des technologies et services *cloud* capables de fonctionner sur tout type d'infrastructures, sans risque d'enfermement propriétaire. C'est dans cette optique que l'initiative *The Cloud Native Computing Foundation* (**CNCF**) **[1]** est née. Pour atteindre cet objectif, les actions de la CNCF passent par différents aspects incluant entre autres, la mise en œuvre et l'évangélisation de meilleures pratiques en matière de *cloud*, l'incubation, le soutien et le financement de projets, la mise à disposition de ressources de tests à grande échelle pour les projets soutenus, etc. Nous y reviendrons en détail dans la suite de l'article.

1. LA CNCF ET SA CHARTE ARCHITECTURALE

Créée en 2015 par **Google**, **Twitter**, **Huawei**, **Intel**, **Cisco**, **IBM**, **Docker**, **Univa**, et VMware, la fondation CNCF s'est mise en place comme une organisation à but non lucratif, et visant à n'utiliser et promouvoir que des systèmes *open source* qui respectent une charte constituée des trois valeurs suivantes :

- Architecture à composant basée sur des conteneurs : en d'autres termes, chaque partie du système (applications, processus, etc.) doit être encapsulée dans un conteneur **[7]**, ceci afin de faciliter la reproductibilité, la transparence et l'isolation des ressources ;
- Orchestration dynamique : signifiant que les conteneurs encapsulant les différents composants du système doivent être perpétuellement et activement orchestrés et ordonnancés pour optimiser l'utilisation des ressources ;
- Architecture en micro-services où les composants du système sont segmentés en micro-services **[8]** afin d'accroître l'agilité tout en simplifiant significativement la maintenance des applications.

La fondation CNCF se veut être une référence pour promouvoir des systèmes respectant cette philosophie. Elle est née en 2015 avec le projet **Kubernetes [9],** un projet *open source* permettant d'orchestrer des conteneurs à grande échelle. Initialement développé en interne chez Google, Kubernetes a été ouvert et cédé à la communauté open source sous initiative CNCF, lui-même en tant que projet collaboratif de la Fondation Linux. Kubernetes sert de fondation à la philosophie CNCF. Depuis, l'initiative a soutenu et continue de soutenir différents autres projets comme nous le verrons dans la suite. Si vous avez un projet répondant à cette charte et souhaitez que ce dernier soit promu et/ou soutenu par la CNCF, vous pouvez soumettre votre projet sur le site internet de la fondation. Tous les projets soumis sont évalués collégialement par les membres d'un comité **[11]**. En conformité avec la charte de la CNCF et des critères bien définis, le comité accepte ou refuse des projets selon un principe de votation entre les membres. Une fois accepté, le projet suit un processus de maturation comme nous le verrons.

2. SOUTIEN AUX PROJETS

Lorsqu'un projet est accepté par le comité de sélection de la CNCF, le projet peut, selon le niveau de maturation, accéder à un programme de soutien bien défini. Avant tout, un projet candidatant pour un soutien par la CNCF doit respecter la charte de la fondation et, les auteurs doivent :

1. accepter de diffuser le projet selon les termes de la **Licence Apache 2.0 [18]**, ou à défaut une licence explicitement acceptée par le comité de gouvernance de la CNCF ;
2. accepter de céder les droits de marque à la fondation CNCF et aussi aider la fondation dans toute démarche de dépôt de marques liées.

2.1 Mode de soutien

Le soutien proposé au projet est divers et varié. Cela couvre aussi bien des dotations classiques pour financer les développements, la vulgarisation et la publicité via les moyens et les canaux de communication de la CNCF, et aussi l'accès au cluster de tests que la fondation met à disposition pour des tests à grande échelle. Pour information, ce cluster de test est actuellement composé de plus d'un millier de nœuds de traitement ultra puissants (jusqu'à 24 cœurs CPU et 250 Gio de RAM).

2.2 Sélection et maturation des projets

Lorsqu'un projet est accepté, il peut être admis à l'un des niveaux de maturation suivants **[11]** :

- Niveau de lancement (*inception project*) : ceci est destiné à des projets qui promeuvent et apportent une réelle valeur ajoutée concernant le développement de technologies et de services cloud natifs. Tous les douze mois, les projets en état de lancement sont évalués par les membres d'un comité de surveillance technique. Selon le résultat de l'évaluation, ce qui résulte notamment d'un système de vote, le comité peut décider, soit de maintenir le projet dans le programme, soit de le promouvoir à un niveau de maturité supérieur (voir ci-dessous). Le comité peut aussi simplement décider d'arrêter de soutenir le projet si les attentes ne sont pas au rendez-vous. Dans ce dernier cas, tous les droits et les marques sont retournés à leurs auteurs.
- Incubation de projet (*incubated project*) : à ce niveau, le projet doit démontrer un niveau de maturation significatif. En plus de remplir les critères des projets à l'étape de lancement, le projet doit entre autres démontrer : son utilisation en production par au moins trois utilisateurs indépendants, un flux continu et substantiel d'engagements et de contributions, un bon nombre de contributeurs permanents, etc.
- Projet mature et hébergé (*graduated project*) : c'est un projet qui a atteint un niveau de stabilité et de maturation satisfaisants. Il peut s'agir d'un projet fraîchement soumis ou d'un projet précédemment en phase de lancement ou d'incubation, qui a su démontrer qu'il possédait entre autres des contributeurs permanents et récurrents appartenant à moins de deux organisations, qu'il était capable d'atteindre et de continuer de maintenir un certain nombre de codes et pratiques en matière d'organisation et de gouvernance (voir *Best Practices Badge* et *Code of Conduct* **[11]**).

2.3 Quelques projets

La fondation CNCF soutient actuellement sept projets majeurs. Comme déjà indiqué, ces projets sont *open source* et gravitent autour des architectures avec des conteneurs. Ils couvrent tous les aspects d'un socle *cloud*, du développement, au déploiement, et à la supervision opérationnelle, en passant par l'intégration continue.

PAS BESOIN D'ÊTRE MEMBRE POUR CONTRIBUER

La fondation CNCF est organisée comme suit :

- **Un comité de gouvernance composé de collaborateurs désignés par les entreprises membres. Ce comité prend les grandes décisions au niveau marketing, de la stratégie et du budget, mais n'intervient pas dans la sélection des projets. Ce dernier point est confié au comité de surveillance technique décrit ci-après. Il existe plusieurs niveaux d'adhésion pour les entreprises ; ce qui est fonction d'un coût de cotisation annuelle et donne droit à plus ou moins de privilèges au sein du comité de gouvernance (ex : droit de vote).**
- **Un comité de surveillance technique dont un des rôles majeurs est de prendre les décisions concernant l'admission et l'accompagnement des projets soumis à la CNCF. Comme pour le comité de gouvernance, les membres de ce comité sont issus des organisations membres de la fondation et sont donc assujettis à des cotisations annuelles.**
- **La communauté des utilisateurs finaux : basée sur une adhésion payante également, cette communauté est formée d'organisations. En tant que membre de la communauté et selon son champ d'expertise, une organisation peut fournir un avis consultatif au comité de surveillance technique concernant les projets. Ainsi la communauté peut attirer l'attention du comité technique sur la pertinence et les priorités à donner aux projets. Ceci permet de soutenir les projets en accord avec les besoins de la communauté des utilisateurs.**
- **Les ambassadeurs : personnes physiques, les ambassadeurs sont des passionnés de la CNCF et/ou des projets que la fondation soutient. Le rôle des ambassadeurs est de promouvoir et/ou contribuer à des projets et technologies cloud natifs, de communiquer et vulgariser localement sur la CNCF, etc.**
- **Toute personne désireuse de contribuer peut librement apporter sa pierre à l'édifice, en organisant par exemple un *meetup*, une conférence, ou tout autre événement contribuant par quelques moyens que ce soit à promouvoir l'initiative et/ou les projets soutenus. D'une certaine façon, le simple fait d'écrire cet article pour *GNU/Linux Magazine* est une contribution pour la CNCF. Pour information, cet article est écrit à titre personnel (auteur de cet article) sans aucune relation à la la CNCF que j'ai découvert au hasard en fouinant sur Internet. J'ai trouvé l'initiative louable et décidé de la partager avec la communauté de *GNU/Linux Magazine*.**

Pour plus de détails sur l'organisation de la CNCF et les possibilités de contribution, bien vouloir consulter le site internet [1].

Les projets actuellement supportés incluent :

- **CoreDNS [12]** qui fournit un service de découverte DNS au sein d'un *cloud*. Le projet est soutenu à un état de commencement par la CNCF.
- **Fluentd [13]** est une solution de journalisation visant à unifier la collecte et l'analyse de données de logs provenant des applications dans une architecture en micro-services. Fluentd est un projet mature et hébergé par la CNCF.
- **gRPC [14]** est une plateforme haute performance visant à fournir un socle universel pour le développement d'applications en micro-services suivant une approche RPC. Le 1er mars 1017, le projet a été promu au rang de projet mature et il est donc hébergé par la CNCF.
- **Kubernetes [9]** offre un système d'orchestration et d'automatisation pour le déploiement et la gestion des applications basées sur des conteneurs à très grande échelle. Né avec la fondation CNCF, Kubernetes est le premier projet mature et hébergé par la CNCF en 2015.
- **Linkerd [15]** est un proxy transparent permettant de faire de la découverte de services, le routage, la détection et la prise en charge des pannes, ainsi que l'inspection des applications. Il fournit un maillage de tolérance aux pannes pour des applications *cloud* natives. Le projet est soutenu à l'état de commencement par la CNCF.
- **Prometheus [17]** fournit un service de supervision reposant sur des métriques. Il permet de définir des métriques et d'y associer un mécanisme d'alerte en cas de dépassement de seuil. Après avoir été intégré à un stade de commencement, Prometheus a été accepté comme second projet admis au rang de projet mature et il est donc hébergé par la CNCF en mai 2016.
- **OpenTracing [16]** est un socle ouvert de traçage et d'analyse des systèmes distribués. Il se veut indépendant et intégrable dans toute architecture à base de micro-services, là où les systèmes de traçage d'applications monolithiques deviennent inefficaces. OpenTracing a été intégré en octobre 2016 comme troisième projet mature et hébergé par la CNCF.

CONCLUSION

Le *cloud* a passé l'étape de buzz, son adoption est plus ou moins bien avancée dans les entreprises. L'interopérabilité, la résilience, de même que le passage à l'échelle sont des enjeux majeurs des applications et des infrastructures sous-jacentes. Nous avons vu dans cet article que l'initiative CNCF est en train de se mettre en route, et ambitionne de mettre ces problématiques au cœur des technologies *cloud* d'aujourd'hui et demain. Des projets comme Kubernetes, CoreDNS, et les autres que nous avons brièvement présentés plus haut dans cet article sont une première étape vers cette quête. La contribution de tous les acteurs du cloud, connus ou anonymes, vous en premier, est vivement encouragée pour soutenir et contribuer à cette belle dynamique. ■

RÉFÉRENCES

[1] Site Internet de *The Cloud Native Computing Foundation* : https://www.cncf.io/

[2] Site Internet d' *Amazon Web Services* : https://aws.amazon.com/

[3] Site Internet d'*OpenStack* : https://www.openstack.org/

[4] Site Internet d'*OpenNebula* : https://opennebula.org/

[5] Site Internet de *VMware vCloud* : http://www.vmware.com/fr/products/vcloud-suite.html

[6] CHAKODE R., *« Environnement d'exécution pour des services de calcul à la demande sur des grappes mutualisées »*, Thèse de doctorat, Université de Grenoble, 2012

[7] SOLTESZ S., PÖTZL H., FIUCZYNSKI M. E., BAVIER A., et PETERSON L., *« Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors »*, Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys '07). ACM, New York, NY, USA, 275-287, 2007. DOI= http://dx.doi.org/10.1145/1272996.1273025

[8] HASSELBRING W., *« Microservices for Scalability: Keynote Talk Abstract »*, Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering (ICPE '16). ACM, New York, NY, USA, 133-134, 2016. DOI : https://doi.org/10.1145/2851553.2858659

[9] Site Internet de *Kubernetes* : https://kubernetes.io/

[10] Site Internet de la Fondation Linux : https://www.linuxfoundation.org/

[11] Sélection et graduation des projets CNCF : https://www.cncf.io/projects/graduation-criteria/

[12] Site Internet de *CoreDNS* : http://coredns.io/

[13] Site Internet de *Fluentd* : http://fluentd.org/

[14] Site Internet de *gRPC* : http://grpc.io/

[15] Site Internet de *Linkerd* : http://linkerd.io/

[16] Site Internet d'*OpenTracing* : http://opentracing.io/.

[17] Site Internet de *Prometheus* : http://prometheus.io/

[18] Site web de la Licence Apache : http://www.apache.org/licenses/LICENSE-2.0

COMMENT RECRÉER NOTRE MONDE EN QUELQUES DIVISIONS ET ADDITIONS

GUILLAUME SAUPIN
[Responsable R&D QosEnergy, Data Scientist]

MOTS-CLÉS : PHYSICS ENGINE, SIMULATION, JULIA, MATH

Ce qui est formidable, avec l'informatique, c'est que c'est le jeu de construction ultime, un méta Lego, qui permet en assemblant de petites briques programmatiques, de créer de nouveaux univers imitant le nôtre.

Construire notre univers, cela permet de se l'approprier, de s'en faire une représentation mentale, et finalement de mieux le comprendre. C'est l'intuition qu'a eue Seymour Papert, l'inventeur du langage Logo, l'inspirateur des Lego Mindstorms, le père du constructionnisme. Quelques mois après sa mort, nous allons lui rendre hommage en construisant notre propre univers physique pour mieux nous approprier les lois de la physique. Ce moteur physique 2D, nous allons le développer à l'aide de Julia, un jeune langage de programmation, inspiré du Lisp, et donc parfaitement dans l'esprit des travaux de Papert, dont on a surnommé Logo le « Lisp sans parenthèse ».

1. SOUS LE CHARME DE JULIA

1.1 Un Lisp dans une robe à fleurs

Avant de rentrer dans le vif de notre sujet, détaillons les raisons qui ont présidées au choix de **Julia** : *« Au commencement était le Verbe, et le Verbe était en Dieu, et le Verbe était Dieu ».*

Quitte à jouer au démiurge, autant se doter d'outils puissants, en l'occurrence un langage qui ne bride pas notre créativité divine. Julia est un excellent candidat, car il est d'ascendance quasi divine, puisque dérivé du **Lisp**. Il offre en outre les fonctionnalités suivantes, qu'on retrouve finalement assez rarement au sein d'un même langage :

- Il est typé dynamiquement, mais permet de contraindre les types dans les prototypes des fonctions ;
- Il intègre un compilateur JIT, basé sur LLVM, qui lui offre des performances très proches de la référence constituée par le **C/C++** ;

- Il bénéficie du mécanisme de macro, dont l'absence dans nombre de langages atterre quotidiennement l'auteur de cet article. Comme la vie serait parfaite si elles existaient en **Go** !
- Son modèle objet supporte non seulement l'héritage, mais aussi le polymorphisme paramétrique et, tenez-vous bien, les *dispatches* multiples, connus aussi sous le nom de multi-méthodes en Lisp ;
- On y trouve aussi des coroutines, ou *green thread*, même si, à mon goût, elles restent bien moins facilement utilisables qu'en Go, ou **Erlang** ;
- C'est un langage fonctionnel, où les fonctions sont des objets de première classe, au même titre que tous les autres types. Elle peuvent donc être manipulées comme n'importe quel autre objet. Petit bémol, le *currying* n'est pas supporté nativement, mais on peut s'en sortir grâce aux macros ;
- Enfin, Julia est dotée d'une REPL, qui permet d'évaluer du code en mode console, outil indispensable pour un développement en mode incrémental.

Ce *medley* de fonctionnalités fait de Julia un langage très expressif, particulièrement performant, pour lequel il est difficile de résister à la tentation de le tester.

1.2 Aperçu

On vient de le voir, Julia ressemble quand même étrangement à son auguste ancêtre, le Common Lisp. Cependant, même si dans sa substance lui est très proche, elle s'en distingue néanmoins en surface : sa syntaxe est beaucoup plus proche du **Python**. C'est un choix délibéré des concepteurs de Julia, dont l'objectif est de substituer le Julia au Python, qui règne pour l'instant en maître dans le domaine du calcul scientifique *open source*.

Personnellement, je ne vois pas le souci avec les parenthèses :)

Voyons donc de quoi il retourne, en commençant par jouer avec le système objet de Julia :

```
abstract Body

type Ball <: Body
    shape::CircleShape
    trail::Vector{CircleShape}
    traj::History{Vector{Float64}}
    stateOffset::Int64
end
```

La première ligne déclare un type abstrait, c'est-à-dire qui n'est pas instanciable. C'est assez similaire aux *classes virtuelles pures* en C++, ou aux *interfaces* du go.

Vient ensuite la déclaration d'un type concret, ici une **Ball**, qui hérite de **Body**, et qui est dotée de propriétés propres. Notez la présence du type de chacune de ces propriétés, après les **::**, et qui aurait aussi bien pu être omis, comme l'illustre la déclaration de la classe ci-devant :

```
type Rectangle <: Body
    shape
    trail
    traj
    stateOffset
end
```

L'intérêt de renseigner le type est double : permettre au compilateur de générer du code plus efficace, mais aussi permettre de détecter plus rapidement des bugs en ayant l'assurance de savoir à qui l'on a à faire.

Ces deux types héritent du type abstrait **Body**, et il est donc possible d'écrire un code générique qui pourra s'appliquer indifféremment à ces derniers, et uniquement à eux. Par exemple :

```
function move(body::Body, pos::Vector{Float64})
    set_position(body.shape, Vector2f(pos[1], pos[2]))
    Hist.add(body.traj, pos)
    for i=1:length(body.trail)
        p = Hist.interpolate(body.traj, 0.25 * i)
        set_position(body.trail[i], Vector2f(p[1], p[2]))
    end
end

function draw(window, body::Body)
    SFML.draw(window, body.shape)
    for i=1:length(body.trail)
        SFML.draw(window, body.trail[i])
    end
end
```

Ces deux méthodes permettent respectivement de déplacer et d'afficher indistinctement une **Ball** ou un **Rectangle**. Là aussi, le type des paramètres a été spécifié, mais une fois encore, ce n'est pas obligatoire. Ne connaissant pas le type du paramètre **window**, je ne l'ai pas indiqué. Notez au passage l'utilisation de la bibliothèque **SFML**, qui va nous accompagner dans cet article pour la partie affichage.

Cela dit, la console en mode REPL permet de se renseigner sur le sujet :

```
julia> typeof(window)
SFML.RenderWindow
```

Voilà en quelques lignes un rapide aperçu du langage Julia. Nous aurons l'occasion d'approfondir d'autres aspects au cours de l'élaboration de notre moteur physique.

2. PHYSICS

Même si le but de cet article est ambitieux : écrire un simulateur physique, nous allons néanmoins travailler sur un périmètre très étroit de la physique, la mécanique newtonienne.

C'est une physique qui nous est familière, puisque c'est celle que nos sens appréhendent au quotidien. Elle est, en outre, culturellement bien implantée dans nos esprits, puisque cela fait plus de trois siècles que Newton en a énoncé les grands principes. En comparaison, son successeur, la théorie de la relativité restreinte, puis généralisée nous est bien moins familière.

Nous allons réduire encore le champ de notre étude, et nous contenter d'un modeste univers 2D. Le code que nous allons écrire sera cependant suffisamment générique pour passer à la dimension 3 sans modification significative.

2.1 Degrés de liberté

Dans le cadre de la mécanique newtonienne, l'état d'un corps est complètement décrit par ses seules position et vitesse. En l'absence d'action extérieure sur ce corps, ces deux données suffisent à décrire son mouvement.

Modifier cet état, c'est-à-dire changer et sa position et sa vitesse, implique d'appliquer une force. Cette force impactera la vitesse de l'objet, et cela d'autant plus fortement que la masse de l'objet est faible.

Imaginons maintenant, comme l'a si bien fait Abbott Abbott dans « *Flatland* », que nous évoluons dans un univers 2D. Quelle liberté a notre corps pour se déplacer ?

Il peut bien sûr se déplacer horizontalement, et aller de droite à gauche. Idem sur l'axe vertical, ce qui lui permet de monter et descendre. Mais ce n'est pas tout. Notre objet peut aussi tourner sur lui-même. Il a donc la liberté de se déplacer suivant trois axes : 2 axes de translation et un axe de rotation. Ce sont ses degrés de liberté.

Exprimé sous forme de code, voici de quoi il retourne :

```
type MechanicalState
    pos::Vector{Float64}
    velocity::Vector{Float64}
    inertia::Matrix{Float64}
    forces::Vector
end
```

L'état d'un corps peut donc se représenter par un vecteur de dimension 3, **pos**, encodant sa position, un vecteur de dimension 3, **velocity**, encodant sa vitesse. Si notre système contient deux corps, alors la dimension du vecteur **pos** est **6** de même que celle de **velocity**.

Nous ajoutons, par commodité, à la modélisation de notre corps une matrice d'inertie, **inertia**, contenant les informations relatives à la masse de l'objet. Nous reviendrons sur ce point un peu plus loin. Elle est carrée et de même dimension que le **velocity**.

Enfin, les forces s'appliquant sur notre corps sont aussi stockées dans cette modélisation. Notez que le type paramétrique **Vector**, n'a pas été ici spécialisé. C'est parce que les forces seront stockées sous forme de fonction. Or le type de ces fonctions n'est pas forcément connu à l'avance. Nous laissons le compilateur faire ce travail, souplesse qu'offre Julia.

2.2 Fiat body

Pour concrétiser, voyons maintenant comment créer *ex nihilo* un corps, et l'ajouter à notre monde :

```
world = World()

pos = [-2.0, 0., 0.]
vel = [0.0, 0.0, 0.]
mass = eye(3)
ball1 = Ball(ball_radius, 1)
Wd.addBody(world, ball1, pos, vel, mass)
```

Notre corps a donc une position définie par un vecteur de dimension 3, **[-2.0,0.0, 0.0]**. Il se trouve donc en **-2.0** sur l'axe x, **0.0** sur l'axe y, et est tourné de **0** degré.

Sa vitesse initiale est aussi de dimension 3, et est nulle sur ces trois dimensions.

Ces deux caractéristiques, position et vélocité, définissent l'état mécanique de notre corps, et s'intègrent à l'état modélisé par notre objet **MechanicalState** dans la fonction **addBody**, définie dans le module **Wd**. D'où le **Wd** dans la ligne **Wd.addBody(world, ball1, pos, vel, mass)**, qui indique que **addBody** est défini dans le module **Wd**.

Programme & Réservations sur paris.makerfaire.com

Make: makezine.com · LEROY MERLIN · ulule · Paris Tech festival · îledeFrance · MAIRIE DE PARIS · Le Parisien

Mais voyons ce que fait cette méthode **addBody** :

```
module Wd
...

function addBody(world::World, body::Body,
pos::Vector{Float64}, vel::Vector{Float64},
mass::Matrix{Float64})
    Physics.addBody(world.state, pos, vel,
mass)
    push!(world.bodies, body)
end
...
end
```

La substantifique moelle de cette méthode est dans :

```
module Physics
...
function addBody(state::MechanicalState,
pos::Vector{Float64}, vel::Vector{Float64},
mass::Matrix{Float64})
    append!(state.pos, pos)
    append!(state.velocity, vel)
    nr, nc = size(state.inertia)
    new = eye(length(state.pos))
    new[1:nr, 1:nc] = state.inertia
    new[nr+1:end, nc+1:end] = mass
    state.inertia = new
end
...
end
```

Ce module se contente simplement d'étendre les vecteurs position, vitesse et la matrice de masse de l'état mécanique de notre simulation.

Pour bien faire comprendre mon propos, détaillons le contenu de l'état mécanique après l'ajout de deux corps avec le code suivant :

```
pos = [-2.0, 0., 0.]
vel = [0.0, 0.0, 0.]
mass = eye(3)
ball1 = Ball(ball_radius, 1)
Wd.addBody(world, ball1, pos, vel, mass)

pos = [-1.0, 0., 0.]
vel = [0.0, 0.0, 0.]
mass = eye(3)
ball2 = Ball(ball_radius, 4)
Wd.addBody(world, ball2, pos, vel, mass)
```

Ceci devient :

```
world.state.pos == [-2.0, 0.0, 0.0, -1.0, 0.0, 0.0]
world.state.velocity == [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
world.state.inertia == [1 0 0 0 0 0
                        0 1 0 0 0 0
                        0 0 1 0 0 0
                        0 0 0 1 0 0
                        0 0 0 0 1 0
                        0 0 0 0 0 1]
```

La construction de cet état pour l'ensemble de notre système est appelée *l'assemblage*.

2.3 Première loi

Quelque chose a jailli du néant, et notre monde est désormais peuplé de *Bodies*, des *Balls* ou des *Rectangles* dont l'état mécanique est caractérisé par notre **mechanicalState**.

Ajoutons maintenant à cet univers une première loi, **la première loi de Newton**, qui ne dit pas autre chose que :

« Tout corps persévère dans l'état de repos ou de mouvement uniforme en ligne droite dans lequel il se trouve, à moins que quelque force n'agisse sur lui, et ne le contraigne à changer d'état. »

C'est terriblement simple (encore fallait-il le trouver) : un corps, se déplaçant à une vitesse donnée sur un axe donné, continue de se déplacer à la même vitesse, sur le même axe, tant qu'on n'applique aucune force dessus.

Imaginons par exemple que notre monde ne contienne qu'une balle, dont la position initiale serait à l'origine, et animée d'une vitesse de 1 m/s :

```
pos = [0.0, 0., 0.]
vel = [1.0, 0.0, 0.]
mass = eye(3)
ball1 = Ball(ball_radius, 1)
Wd.addBody(world, ball1, pos, vel, mass)
```

L'état mécanique initial, à **t0**, de notre monde serait alors :

```
world.state.pos == [0.0, 0.0, 0.0]
world.state.velocity == [1.0, 0.0, 0.0]
world.state.inertia == [1 0 0
                        0 1 0
                        0 0 1]
```

Pour pouvoir appliquer la première loi de Newton, il nous faut un moyen pour calculer, à partir de **t0**, les positions successives de notre corps.

Pour cela, on peut repartir de la définition de la vitesse. Notre corps est animé d'une vitesse de 1 m/s suivant l'axe horizontal, c'est-à-dire qu'il va parcourir en une seconde la distance d'un mètre. De manière générale, on peut calculer la vitesse moyenne sur la période **[t, t+Δt]** avec la formule suivante :

$$v_x(t) = \frac{x(t+\Delta t) - x(t)}{\Delta t}$$

Or, à un instant **t**, dans notre monde, nous connaissons **x(t)**, ainsi que v_x**(t)**. Calculer la nouvelle position **Δt** secondes plus tard se fait alors simplement : **x(t+Δt) = x(t) + v_x(t) * Δt**.

Si l'on généralise pour les deux dimensions **x(t)** et **y(t)**, on obtient, sous forme matricielle :

$$\begin{bmatrix} x(t+\Delta t) \\ y(t+\Delta t) \end{bmatrix} = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix} + \Delta t * \begin{bmatrix} v_x(t) \\ v_y(t) \end{bmatrix}$$

Grâce à cette simple formule, il est possible de déterminer une nouvelle position pour notre corps, et le faire ainsi évoluer dans le temps. En Julia, cela donne :

```
function integrate(state::MechanicalState,
dt::Float64)
    state.pos = state.pos + state.velocity * dt
end
```

En appelant dans une boucle sans fin cette méthode, notre corps va se déplacer selon la première loi de Newton. Seule la position évolue, la vitesse n'étant pas impactée, comme le stipule la première loi. Cette constance de la vitesse se matérialise d'ailleurs sur la figure 1 par un espacement constant entre les positions successives de notre corps test.

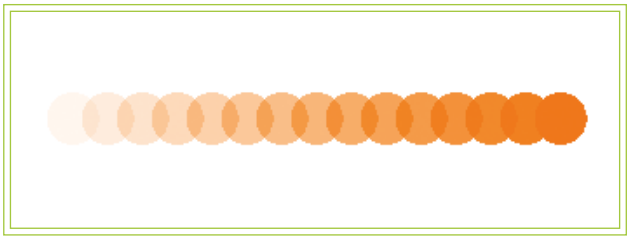


Fig. 1 : Un corps libre, suivant la première loi de Newton et se déplaçant donc en ligne droite.

Notez bien que la formule que nous utilisons fait l'hypothèse que la vitesse est constante entre **t** et **t+Δt**. Dans le cadre de la première loi de Newton, c'est toujours vrai, puisque la vitesse reste inchangée. Ce n'est généralement pas le cas, c'est pourquoi **Δt**, que l'on nomme « pas de temps » de la simulation, est généralement pris très court, de l'ordre de la fraction de millisecondes. Dans un laps de temps aussi court, dans les systèmes physiques qui nous intéressent, les vitesses n'ont pas le temps de varier significativement, et l'hypothèse de constance de la vitesse reste valable.

La boucle principale de notre simulateur ressemble à ce stade à :

```
while true
        clear(window, SFML.white) #clear the window
        update(world.state, world.bodies) #update
bodies position
        draw(window, world) #draw bodies
        display(window)
end
```

Avec la méthode **update** :

```
function update(state, balls)
    for i=1:20
        integrate(state, 1e-4)
    end
    for i=1:length(balls)
        ball = balls[i]
        move(ball, transform(window_width,
window_height, state.pos[ball.stateOffset:ball.
stateOffset+2]))
    end
end
```

Notez le pas de temps, **dt**, que nous avons pris égal à un dixième de milliseconde. Tant que nous n'appliquons que la première loi de Newton, ça n'a pas d'importance, mais cela en aura d'ici peu.

2.4 Seconde loi

2.4.1 Le principe

À ce point, notre monde n'est pas très intéressant d'un point de vue mécanique. Soit les objets existants sont immobiles, et le reste indéfiniment, soit ils sont animés d'un mouvement uniforme, et se déplacent donc indéfiniment dans la même direction, toujours à la même vitesse.

Ce qui nous manque maintenant, c'est un moyen pour altérer la vitesse des corps, pour leur donner des trajectoires plus complexes. **La seconde loi de Newton** nous renseigne précisément sur la façon dont se modifie la vitesse d'un corps :

« Soit un corps de masse m constante, l'accélération subie par un corps est proportionnelle à la résultante des forces qu'il subit, et inversement proportionnelle à sa masse».

Rappelons que l'accélération est à la vitesse ce que la vitesse est à la position. C'est-à-dire que l'accélération indique la variation instantanée de la vitesse. Il est donc possible de reformuler cette seconde loi, en disant simplement que la variation de vitesse d'un corps, soumis à une force, est d'autant plus grande que la force est grande, et d'autant plus marquée que la masse du corps est petite.

En substance, si je pousse comme une brute un rocher, il ne va pas beaucoup bouger, par contre, si je donne une pichenette dans ma fille de 2 ans, je vais la retrouver par terre.

Mais revenons à l'accélération, dont on peut, à l'instar de la vitesse, donner la valeur moyenne sur un pas de temps **Δt** à l'aide de cette formule, valable aussi suivant **y** :

$$a_x(t) = \frac{v_x(t+\Delta t) - v_x(t)}{\Delta t}$$

Ce qui permet, de la même manière que pour la position, de calculer la valeur de la vitesse au pas de temps suivant :

$$\begin{bmatrix} v_x(t+\Delta t) \\ v_y(t+\Delta t) \end{bmatrix} = \begin{bmatrix} v_x(t) \\ v_y(t) \end{bmatrix} + \Delta t * \begin{bmatrix} a_x(t) \\ a_y(t) \end{bmatrix}$$

2.4.2 Son application

En bon français, cela donne :

```
function integrate(state::MechanicalState,
dt::Float64, force::Vector2f)
    a = forceAcceleration(state, dt)
    state.velocity = state.velocity +  a * dt
    state.pos = state.pos + state.velocity * dt
end
```

Notre intégration dans le temps du mouvement de nos corps, se fait donc désormais en deux passes. D'abord, nous mettons à jour la vitesse de notre corps, puis, fort de cette nouvelle vitesse, nous mettons à jour la position.

En accord avec la formule ci-dessus, nous avons bien sûr besoin de l'accélération pour mettre à jour notre vitesse. Là encore, la seconde loi de Newton nous indique comment procéder pour un corps :

$$\begin{bmatrix} a_x(t) \\ a_y(t) \end{bmatrix} = M^{-1} * \begin{bmatrix} f_x(t) \\ f_y(t) \end{bmatrix}$$

L'accélération est proportionnelle à la force, et inversement proportionnelle à la masse, soit en quelques lignes :

```
function forceAcceleration(state::MechanicalState,
dt::Float64)
    M = state.inertia
    x = state.pos
    v = state.velocity
    forces = state.forces

    f = fill(0, length(state.pos))
    for i=1:length(state.forces)
        f += forces[i](x,v)
    end

    inv(M) * f
end
```

Où les forces s'appliquant dans notre monde sont stockées dans le **MechanicalState**, dans le vecteur **forces**, et sous la forme de fonctions de la vitesse et de la position.

L'ajout d'une force se fait par le biais de la fonction suivante :

```
function addForce(state::MechanicalState, f)
    push!(state.forces, f)
end
```

Notez aussi la présence de la ligne suivante :

```
inv(M) * f
```

Elle est le cœur de la seconde loi de Newton. **M** est ici la matrice de masse, tandis que **f** est le vecteur contenant la somme de forces s'appliquant à nos objets. L'opérateur **inv**, quant à lui, inverse la matrice de masse. L'accélération calculée est donc bien proportionnelle aux forces appliquées, et inversement à la masse.

2.4.3 Masse suspendue à un ressort

Donnons un peu de substance à cette loi et à ce code, en donnant un rapide exemple. Imaginons un corps, de 1 kg, suspendu à l'aide d'un ressort. En position initiale, il est en position **(0,0,0)** et le ressort qui le rattache à un point fixe ne lui applique aucune force.

Une fois libéré, ce corps va chuter suivant l'axe vertical, **y**, et osciller indéfiniment.

Simuler un tel système est dorénavant à notre portée. Les lignes suivantes suffisent :

```
world = World()
####
# Create Body
####
pos = [0.0, 0., 0.]
vel = [0.0, 0.0, 0.]
mass = eye(3)
ball = Rectangle(ball_radius, ball_radius, 7)
Wd.addBody(world, ball, pos, vel, mass)

s = spring(1e2,1.,0.,0.,2)
addForce(world.state, s)
```

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

Où :

```
function spring(stiffness::Float64,
damping::Float64, restLength::Float64,
restPos::Float64, coordIdx::Int64)
    function s(x::Vector{Float64},
v::Vector{Float64})
        f = fill(0.0, length(x))
        f[coordIdx] = - stiffness * ((x[coordIdx] -
restLength) - restPos) - damping * v[coordIdx]
        f
    end
    s
end
```

Cette fonction retourne une fonction. C'est toute l'élégance des langages fonctionnels. Cette fonction, `s`, est en plus une fermeture, puisqu'elle capture plusieurs paramètres de la fonction `spring`, dont notamment `offset`, qui indique quel degré de liberté de notre système se verra appliquer l'effort généré par le ressort.

2.5 Champ de pesanteur

2.5.1 Tout corps est accéléré

À ce point, les corps évoluant dans notre monde se comportent comme s'ils se déplaçaient dans l'espace, sans aucune forme d'interaction avec quoi que ce soit. Habitués que nous sommes à vivre sur Terre, il nous semble naturel qu'a minima, un corps tende à tomber quand rien n'est là pour le soutenir. Ce comportement est dû à la présence de la Terre, et du champ de pesanteur que son imposante masse génère.

La formulation classique de **la loi universelle de la gravitation**, qui gouverne l'action d'un corps massique sur un autre, s'énonce ainsi :

« Deux corps ponctuels de masses respectives M_a et M_b s'attirent avec des forces de mêmes valeurs (mais vectoriellement opposées), proportionnelles aux produits des deux masses, et inversement proportionnelles au carré de la distance qui les sépare. Cette force a pour direction la droite passant par les centres de gravité de ces deux corps. »

Il est possible d'utiliser cette loi pour calculer la force générée par l'attraction de la Terre, et l'intégrer dans notre simulation comme n'importe quelle autre force. Cependant, il est plus commode d'avoir recours à la notion de champ de pesanteur. Ce champ de pesanteur exerce sur tout objet doté d'une masse `m` une force $\vec{p}=m\vec{g}$, où $\vec{g}=\begin{bmatrix}0\\9.81\end{bmatrix}\frac{m}{s^2}$.

$\vec{g}$ s'exprime donc en $\frac{m}{s^2}$ et est de ce fait homogène à une accélération.

Inutile donc de calculer la force subie par un corps du fait de la présence du champ de pesanteur. Il suffit de réaliser que tout corps plongé dans ce champ gravitationnel est automatiquement accéléré.

Cela présente l'avantage d'être une modélisation plus en phase avec la théorie plus récente de la relativité généralisée, et simplifie grandement le code, puisque la prise en compte de la gravité se fait en ajoutant simplement un terme à notre accélération :

```
function integrate(state::MechanicalState,
dt::Float64, force::Vector2f)
    a = gravityAcceleration(state, dt) +
forceAcceleration(state, dt)
    state.velocity = state.velocity +  a * dt
    state.pos = state.pos + state.velocity * dt
end
```

Avec :

```
function gravityAcceleration(state::Mecha
nicalState, dt::Float64)::Vector{Float64}
    a = []
    for i=1:length(state.pos)/3
        a = append!(a,[0, -9.81, 0.])
    end
    a
end
```

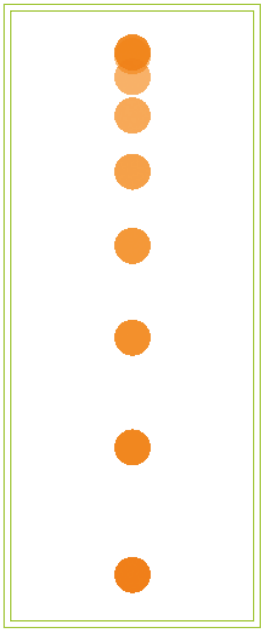


Fig. 2 : Le corps représenté par un disque jaune tombe librement sous l'effet de la gravité.

La figure 2 montre un corps chutant librement sous l'effet de la gravité. L'espacement croissant des points montre l'accélération quadratique du corps, contrairement à notre précédent exemple. C'est dû à l'introduction d'une accélération, qui est intégrée deux fois dans le temps, ce qui nous fait passer d'une constante à un terme en t^2.

2.5.2 Balistique

Illustrons la simulation du champ de pesanteur, en nous intéressant au cas de la balle de fusil, décrit par Richard Feynman dans ses cours de physique, dont je recommande chaudement la lecture.

La question que se pose M. Feynman, est la suivante :

- sachant que tout objet à la surface de la Terre est accéléré vers le centre de la Terre, sous l'effet de cette accélération de 9.81 m/s ;
- sachant que la Terre est ronde, et que son rayon est de 6900 km.

Quelle doit être la vitesse initiale d'une balle de pistolet pour qu'elle ne touche jamais le sol ? Richard Feynman donne la réponse à cette question dans son chapitre sur la théorie de la gravitation : environ 8 km/s.

Ce qu'on peut simuler comme suit :

```
world = World()
####
# Create Body
####
pos = [0.0, 0., 0.]
vel = [8e3, 0.0, 0.]
mass = eye(3)
ball = Rectangle(ball_radius, ball_radius,
7)
Wd.addBody(world, ball, pos, vel, mass)
```

2.6 Contraintes

Notre monde virtuel est encore très rudimentaire, mais commence néanmoins à ressembler pas mal à notre monde physique, tout au moins en ce qui concerne la physique du corps solide. Les objets simulés sont dotés d'une masse et se meuvent de manière physique sous l'effet de la gravitation, mais aussi de forces extérieures quelconques.

La prise en compte des forces et de l'accélération suffit à simuler un grand nombre de systèmes. Mais on ajoute généralement un autre outil aux simulateurs physiques : les contraintes.

Ces dernières permettent de modéliser différents types de liaisons physiques, telles que :

- les liaisons prismatiques, qui n'autorisent qu'un mouvement de translation ;
- les liaisons pivots, qui n'autorisent qu'une rotation entre les deux objets.

Bien d'autres types de liaisons existent, et nous allons concevoir le code permettant de les supporter de manière très générique. Notre objectif est de permettre la modélisation de n'importe quelle contrainte, du moment qu'on puisse l'exprimer sous la forme d'une fonction de la position et de la vitesse de notre corps.

2.6.1 Définition

Toujours dans l'optique de nous doter d'un moteur générique, nous allons poser le minimum de restriction sur la définition de nos contraintes. La forme la plus générique que l'on puisse donner à une contrainte est donc une fonction de l'état de notre système.

Et nous choisirons de considérer que cette contrainte est respectée quand sa fonction associée est nulle. Concrètement, si l'on souhaite forcer le déplacement d'un corps sur une trajectoire circulaire de rayon **l**, centrée sur l'origine, il suffit de définir la fonction :

$$c_\circ(x,y)=\frac{1}{2}(x^2+y^2-l^2)$$

Le point **(l,0)** vérifie par exemple cette contrainte, puisque $c_\circ(l,0)=\frac{1}{2}(l^2+0^2-l^2)=0$. Générer une telle fonction de contrainte en Julia est facile, puisque le langage est fonctionnel :

```
function distConstraint(l::Float64)
    f(x::Vector{Float64})::Float64 = 0.5 * ((x[1]^2 +
x[2]^2) - l^2)
    f
end
```

Avec ce moyen, vous pouvons définir tout type de contrainte, y compris les plus loufoques, du moment que l'on puisse les écrire sous forme de fonction de l'état de notre système.

2.6.2 Multiplicateurs de Lagrange

Nous voici rendus au point de cet article où il va nous falloir avouer que nous avons fait un peu de math. En effet, la formule $v_x(t)=\frac{x(t+\Delta t)-x(t)}{\Delta t}$ utilisée pour le calcul de la vitesse suivant **x**, à instant **t**, n'est autre qu'une version discrétisée de la définition de la dérivée :

$$v_x(t)=\frac{\partial x(t)}{\partial t}=\lim_{h\to 0}\frac{x(t+h)-x(t)}{h}$$

C'est-à-dire que nous avons estimé que nous pouvions approcher **vx(t)** en ne faisant pas tendre **h** vers **0**, mais en prenant plutôt une valeur suffisamment petite, ici **Δt** de l'ordre de quelques fractions de millisecondes.

Pour assurer le respect de notre contrainte, nous allons recourir aux mêmes outils. En effet, pour que notre contrainte soit respectée, il faut :

- qu'elle soit respectée à l'état initial, c'est-à-dire au premier pas de temps de notre simulation ;

• qu'elle soit respectée à chaque pas de temps.

Le premier point est simple à satisfaire. Il suffit de placer initialement nos corps à des positions en accord avec nos contraintes.

Le second est plus délicat à garantir. Imaginons que nous partions d'un instant **t** pour lequel notre contrainte **C** est respectée. Cela signifie que **C(x(t), y(t))=0**. Nous voulons que cette condition reste vraie à **t+Δt**, soit **C(x(t+Δt),y(t+Δt)=0**. Or pour passer de l'état **x(t)** à **x(t+Δt)**, nous avons décidé de travailler en accélération. Il va donc nous falloir trouver une accélération qui assure le respect de cette contrainte.

Voici comment. Nous commençons par dériver notre contrainte par rapport au temps, de manière à faire apparaître des vitesses :

$$\frac{\partial C(x(t),y(t))}{\partial t}=\frac{\partial C(x(t),y(t))}{\partial x(t)}*\frac{\partial x(t)}{\partial t}+\frac{\partial C(x(t),y(t))}{\partial y(t)}*\frac{\partial y(t)}{\partial t}$$
$$\frac{\partial C(x(t),y(t))}{\partial t}=\frac{\partial C(x(t),y(t))}{\partial x(t)}*v_x(t)+\frac{\partial C(x(t),y(t))}{\partial y(t)}*v_y(t)$$

Pour mémoire, **C(x(t),y(t))** étant la composition de la fonction **C** avec les fonctions **x** et **y**, nous avons utilisé la règle de dérivation en chaîne. La somme entre le terme fonction de **vx(t)** et **vy(t)** découle pour sa part du fait que **C** est fonction de **x** et **y**.

Pour simplifier la manipulation de ces objets mathématiques, on fait généralement appel à une notation matricielle, sous la forme $\frac{\partial C(x(t),y(t))}{\partial t}=JV$ avec $J=\left[\frac{\partial C(x(t),y(t))}{\partial x(t)}\ \frac{\partial C(x(t),y(t))}{\partial y(t)}\right]$ et $V=\begin{bmatrix}v_x(t)\\v_y(t)\end{bmatrix}$. **J** est alors appelé la **Jacobienne**.

Malheureusement, il nous faut aller plus loin dans les calculs pour obtenir ce que nous voulons : faire apparaître une accélération. Nous allons donc dériver à nouveau ce premier résultat par rapport au temps : $\frac{\partial C(x(t),y(t))}{\partial t^2}=\frac{\partial (JV)}{\partial t}=\frac{\partial J}{\partial t}V+J\frac{\partial V}{\partial t}$ où cette fois-ci, $\frac{\partial V}{\partial t}=\begin{bmatrix}\frac{\partial v_x(t)}{\partial t}\\\frac{\partial v_y(t)}{\partial t}\end{bmatrix}=\begin{bmatrix}a_x(t)\\a_y(t)\end{bmatrix}=A$ avec **A** le vecteur contenant l'accélération.

Rappelons que nous faisons tous ces calculs pour essayer de garantir notre contrainte **C(x(t), y(t))=0**. Si cette condition est assurée initialement, alors si la dérivée seconde est toujours nulle, alors la dérivée première sera elle aussi toujours nulle, et donc la condition restera toujours nulle. La contrainte sera ainsi toujours respectée si :

$$\frac{\partial J}{\partial t}V+JA=0$$

Or nous l'avons vu, la seconde loi de Newton nous indique que **A=M⁻¹*F**. Injectée dans l'équation précédente, cela donne :

$$\frac{\partial J}{\partial t}V+JM^{-1}F=0$$

Si maintenant on décompose les forces appliquées à notre système comme étant $F=F_e+F_c$, avec F_e les forces externes, et $F_c=J^T\lambda$ les forces générées par les contraintes que nous voulons assurer, alors nous aboutissons enfin à :

$$JM^{-1}J^T\lambda=\frac{-\partial J}{\partial t}V-JM^{-1}F_e$$

Notez bien que nous avons choisi de définir les forces appliquées par les contraintes comme étant des combinaisons linéaires du gradient des positions. C'est ce qu'exprime $F_c=J^T\lambda$. Or le gradient d'une surface est orthogonal à cette dernière, donc les forces F_c sont orthogonales au mouvement et ne génèrent donc pas de travail. Dit autrement, elles n'introduisent pas d'énergie dans notre simulation.

Les coefficients de λ sont ce qu'on appelle des **multiplicateurs de Lagrange**.

Une fois codé, ce n'est finalement pas si compliqué :

```
function multipliersAcceleration(
state::MechanicalState,
dt::Float64)
    a = gravityAcceleration(state,
dt) + forceAcceleration(state, dt)
    M = state.inertia
    x = state.pos
    v = state.velocity
    state.constraintJacobian =
jacobian(state.constraint, x)
    J = state.constraintJacobian
    Jprev = state.Jprev

    J_dt = (J - Jprev) / dt
    Jv = J_dt * v
    Minv = inv(M)
    l = -(J * Minv * J') \ (Jv +
(J * a))

    state.Jprev = J
    Minv * J' * l
end
```

À l'instar des méthodes **gravityAcceleration** et **forceAcceleration**, **multipliersAcceleration** calcule l'accélération résultant de l'application des forces liées aux contraintes.

La jacobienne est calculée par différentiation numérique avec le code suivant, en discrétisant sa définition :

```
function jacobian(f, x::Vector{Float64})
    eps = Float64(1e-8)
    fx = f(x)
    J = fill(Float64(0), length(fx),
length(x))
    for i=1:length(fx)
        for j=1:length(x)
            xp = copy(x)
            xp[j] = xp[j] + eps
            val = (f(xp) - fx)[i] / eps
            J[i,j] = val
        end
    end
    J
end
```

Vous aurez noté que $\frac{\partial J}{\partial t}$ est calculée suivant le même principe, mais en conservant `J(t-1)` et en la soustrayant de `J(t)` et en divisant le tout par `Δt`.

2.6.3 Contraignons

Doté de tous les outils nécessaires pour ajouter des contraintes, nous pouvons maintenant par exemple contraindre un corps à se déplacer sur un cercle. Revenons sur la contrainte déjà présentée ci-dessus :

```
function distConstraint(l::Float64)
    f(x::Vector{Float64})::Float64 = 0.5 * ((x[1]^2 +
x[2]^2) - l^2)
    f
end
```

Cette contrainte s'annule lorsque le corps concerné se trouve à une distance `l` de l'origine. Dans le même esprit, on peut contraindre un corps à suivre une trajectoire elliptique avec :

```
function elliConstraint(l::Float64)
    f(x::Vector{Float64}) = [x[1,1]^2/2^2 +
x[2,1]^2/1^2 - l^2]
    f
end
```

Ces deux précédentes contraintes portent sur les deux premiers degrés de liberté du premier corps, donc x et y. Il est bien sûr possible d'appliquer des contraintes sur plusieurs corps :

```
function multiConstraint(l::Float64)
    f(x::Vector{Float64}) = [x[1,1]^2/2^2 +
x[2,1]^2/1^2 - l^2 ; 0.5 * ((x[4]^2 + x[5]^2) - l^2)]
    f
end
```

Ici, le premier corps est invité à suivre une trajectoire elliptique, tandis que le second se déplace sur un cercle.

Pour illustrer cette gestion par contrainte, et mieux appréhender le rôle de la jacobienne, arrêtons-nous sur le cas de la contrainte circulaire.

Sur la figure 3 est représenté un corps, matérialisé par le point `D`, dont le mouvement circulaire est contraint par `C(x,y)=0`. La force f_c, combinaison linéaire des colonnes de J^T est orthogonale au mouvement.

Cette figure fait clairement apparaître le caractère orthogonal des colonnes de la transposée de la jacobienne par rapport à la contrainte `C(x,y)`. La force assurant le respect de la contrainte est bien radiale, et tend à rapprocher le point `D` dans le cas où il s'écarterait de la contrainte.

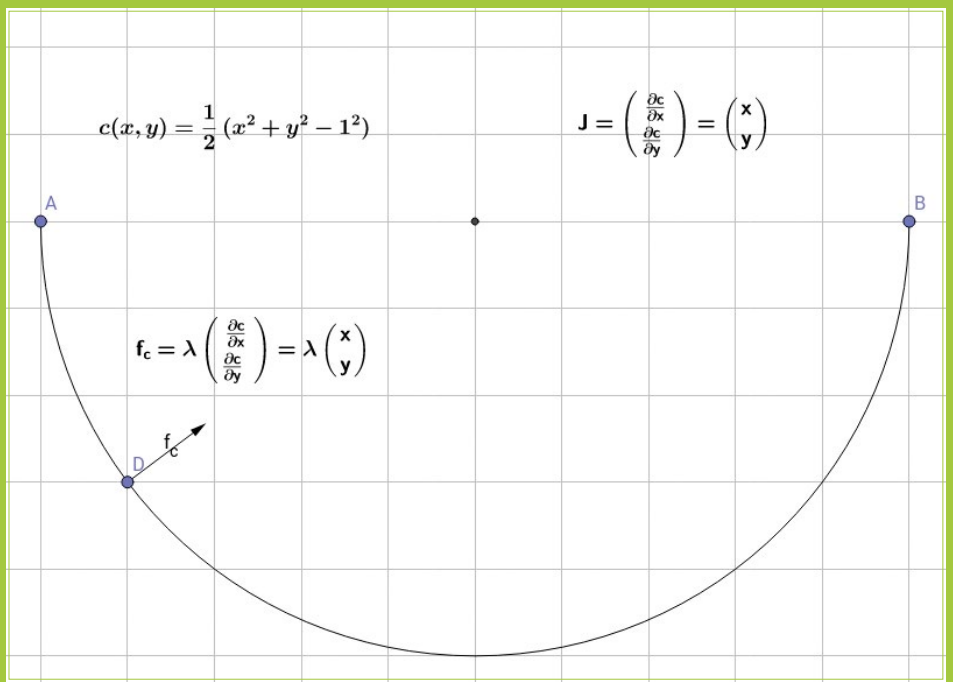


Fig. 3 : Le corps D se déplace suivant une trajectoire circulaire, selon la contrainte C(x,y)=0.

La définition globale de la scène donne alors tout simplement :

```
world = World()
constraintLength = 1.0
world.state.constraint = multiConstraint(constraintLength)
####
# Create Body 1
####
pos = [-2.0, 0., 0.]
vel = [0.0, 0.0, 0.]
mass = eye(3)
ball1 = Ball(ball_radius, 1)
Wd.addBody(world, ball1, pos, vel, mass)

####
# Create Body 2
####
pos = [-1.0, 0., 0.]
vel = [0.0, 0.0, 0.]
mass = eye(3)
ball2 = Ball(ball_radius, 4)
Wd.addBody(world, ball2, pos, vel, mass)
```

En image sur la figure 4, et grâce à la bibliothèque SFML, on observe que le corps rouge suit la trajectoire elliptique, tandis que le jaune suit une trajectoire circulaire.

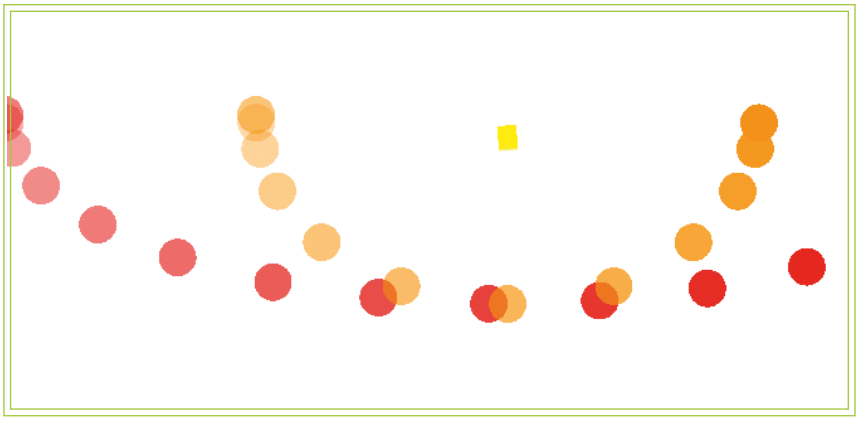


Fig. 4 : Deux corps évoluant sous l'effet de la gravité et d'une contrainte circulaire pour l'un, et elliptique pour l'autre.

CONCLUSION

Voici rapidement brossé comment créer un moteur physique 2D, très générique, avec support pour des forces de n'importe quel type, et mieux encore, pour n'importe quel type de contraintes.

Nous en avons profité pour découvrir Julia, qui est un langage parfaitement adapté pour faire des maths de manière efficace, même si actuellement je le trouve un peu jeune, et surtout lent à démarrer. Il lui manque aussi nativement le support pour le calcul symbolique, qui couplé au générateur de code LLVM pourrait faire un mélange détonnant.

Notamment, nous aurions pu nous en servir pour calculer non pas numériquement, mais symboliquement notre jacobienne et sa dérivée. Une autre possibilité pour le calcul de cette dernière, aurait été d'utiliser la différentiation automatique, qui est assez peu utilisée et que je vous invite à découvrir.

Manque aussi à notre panorama la gestion des contraintes unilatérale soit, en français, les contacts ! Mais c'est un sujet qui aurait nécessité encore plus de formules ! ■

NOTE

Le code de cet article a été développé sous Ubuntu, et est disponible sur GitHub : https://github.com/kayhman/alicia

POUR EN SAVOIR PLUS...

Si le sujet vous botte et que vous souhaitez en savoir plus, je vous recommande les tutoriels pour le SIGGRAPH de David Baraff.

Pour la partie purement physique, les cours de physique de Richard Feynman sont incontournables.

Enfin, pour ce qui est de la partie mathématique, je ne résiste pas au plaisir de vous faire découvrir les merveilleux cours vidéos du professeur Herbert Gross, du MIT, accessibles sur le site http://ocw.mit.edu. Ces cours sont en noirs et blancs, et ont été tournés en... 1970 ;-)

Le langage Julia se trouve pour sa part sur http://www.julialang.org, et la bibliothèque SFML utilisée pour l'affichage est ici : https://github.com/zyedidia/SFML.jl.

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

Webcast

4, 5, 6, 7 juillet 2017

Aix-Marseille Université, la Canebière

Information, programme, réservation et inscription :

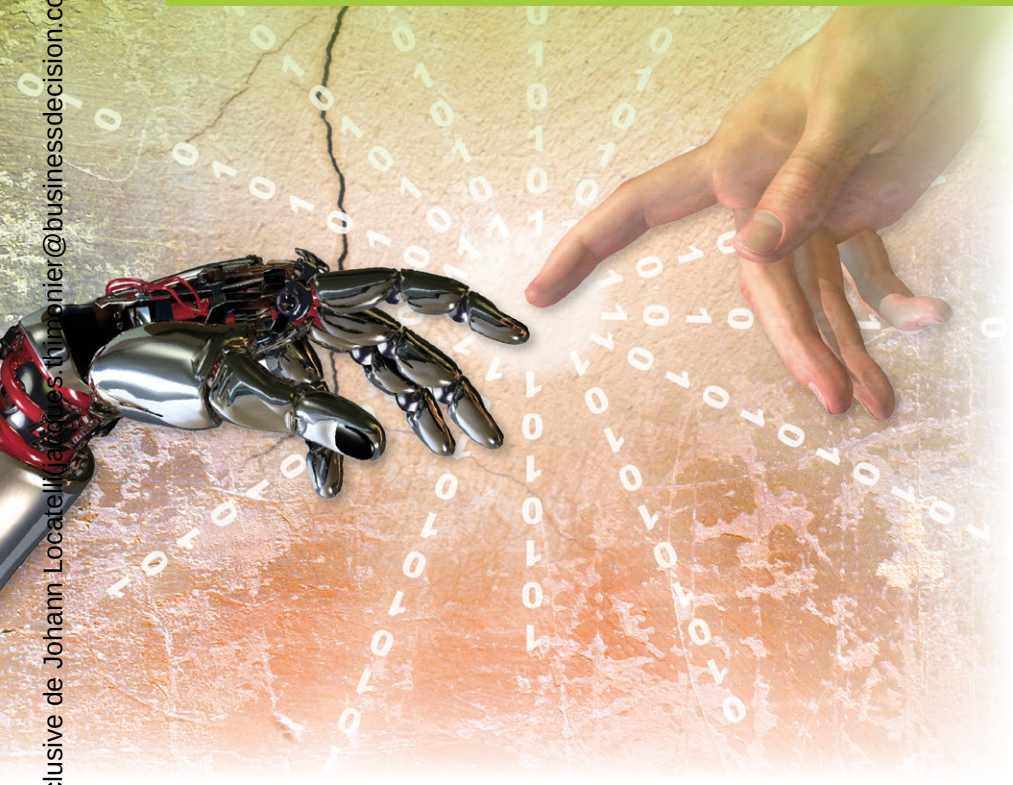
http://devlog.cnrs.fr/jdev2017

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

LE JEU DE LA VIE DE CONWAY : IMPLÉMENTATION ET PETITES ADAPTATIONS

TRISTAN COLOMBO

MOTS-CLÉS : JEU DE LA VIE, CONWAY, PYTHON, PYGAME, FICHIERS, ALGORITHME



Le jeu de la vie est un des grands classiques de l'algorithmie. Dans cet article, je vous propose de redécouvrir cet algorithme, basé sur les automates cellulaires et de l'implémenter en Python avant de créer des automates capables de réaliser des tâches pratiques.

Pour ceux d'entre vous qui ne le connaîtraient pas, le jeu de la vie n'est pas à proprement parler un jeu dans la mesure où vous n'aurez aucune action à effectuer. Il s'agit en fait d'un automate particulier, un **automate cellulaire**, dont les règles ont été définies par le mathématicien John Conway en 1970. Ce n'est pas un jeu/automate que son auteur aime particulièrement, car il a « masqué » toutes ses autres découvertes et il ne considère pas qu'il s'agisse vraiment de mathématiques. On peut aisément imaginer que si on lui parle de ce jeu depuis 47 ans, cela puisse devenir lassant... Je vous recommande vraiment de visionner deux vidéos d'interview (en anglais) sur **YouTube [1][2]** où il explique lui-même le jeu de la vie et comment il a eu l'idée de l'inventer. Bien entendu, vous n'êtes pas obligé de voir ces vidéos pour comprendre le jeu de la vie puisque c'est ce que nous allons faire dans la suite, mais nous avons là la chance de pouvoir avoir une explication fournie directement par l'inventeur d'un des « monuments » de l'informatique.

Dans le cadre de cet article, nous commencerons par définir ce qu'est un automate cellulaire, avant d'aborder le jeu de la vie et d'en réaliser une implémentation en Python. Pour finir, nous nous amuserons à créer des automates permettant d'obtenir des résultats pratiques : détecter des fichiers volumineux ou profiler un code.

1. LES AUTOMATES CELLULAIRES

Dans « automate cellulaire », il y a deux mots : « automate » et « cellulaire ». Pour pouvoir définir un automate cellulaire, il faut donc tout d'abord savoir ce qu'est un automate.

1.1 Les automates

Nous n'allons pas trop rentrer dans la théorie et nous nous limiterons ici à une catégorie précise d'automates : les **automates finis déterministes** ou **AFD**. Vous avez forcément déjà utilisé et implémenté un AFD. Par exemple, si un utilisateur saisit des chiffres (`0` ou `1`) et que vous voulez détecter dans cette saisie s'il a tapé une alternance stricte de `0` et de `1` (commençant indifféremment par `0` ou `1`), vous utiliserez un AFD. Le tableau suivant illustre quelques exemples de chaînes correctes et erronées :

Chaîne	Résultat
`0101010101`	Correct
`101010101`	Correct
`1010101010`	Correct
`1101010`	Erroné
`01010101021`	Erroné

On voit bien que notre chaîne ne doit contenir que les chiffres `0` ou `1`, qu'elle peut commencer par `0` ou `1` et qu'après un `0` on ne peut avoir qu'un `1` ou la fin de la chaîne, et après un `1` on ne peut avoir qu'un `0` ou la fin de la chaîne. On peut formaliser cela sous la forme d'un automate :

1. On ne peut utiliser que `0` ou `1` : c'est l'**alphabet**, l'ensemble des caractères autorisés dans l'automate. Ici `A = {0, 1}`.
2. Pour déterminer si un utilisateur a saisi `0` ou `1`, on utilise un système d'**états** (l'état « l'utilisateur a saisi `0` après un `1` ou rien », etc.). Cet ensemble d'états est noté `Q`. Dans le cadre de notre exemple, nous avons cinq états : `Q = {q0, q1, q2, q3, q4}`. Ces états représentent :
   - `q0` : état de départ, début de la saisie ;
   - `q1` : état où l'utilisateur vient de saisir un `0` « correct » (après un `1` ou rien) ;
   - `q2` : état où l'utilisateur vient de saisir un `1` « correct » (après un `0` ou rien) ;
   - `q3` : état où l'utilisateur vient de saisir un `0` « incorrect » (après un `0`) ;
   - `q4` : état où l'utilisateur vient de saisir un `1` « incorrect » (après un `1`).
3. L'état de départ (ou **état initial**) pour analyser la chaîne est l'état `q0`.
4. Les **états finaux** permettant de dire si le traitement est achevé sont `q1` et `q2` (corrects) et `q3` et `q4` (incorrects). On a donc un ensemble d'états finaux `F = {q1, q2, q3, q4}`.
5. Enfin il faut savoir comment l'on passe d'un état à l'autre. Cela est indiqué par une **fonction de transition** $\delta : Q \times A \rightarrow Q$ (en fonction d'un état de `Q` et d'un caractère de `A`, on obtient un état de `Q`). Pour notre automate, nous avons :
   - δ`(q0, 0) = q1` ;
   - δ`(q1, 1) = q2` ;
   - δ`(q1, 0) = q3` ;
   - δ`(q0, 1) = q2` ;
   - δ`(q2, 0) = q1` ;
   - δ`(q2, 1) = q4`.

On peut représenter tout cet automate `(A, Q, q0, F, δ)` sous la forme d'un schéma, comme le montre la figure 1.

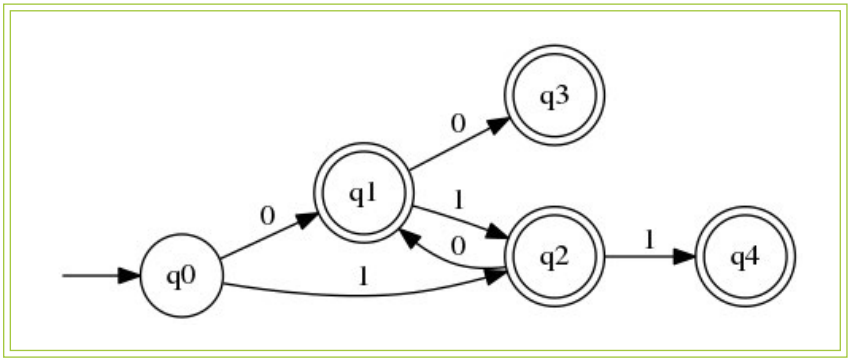


Fig. 1 : Automate Fini Déterministe détectant des suites de 0 et de 1.

GÉNÉRER DES AUTOMATES AVEC GRAPHVIZ

La figure 1 a été réalisée à l'aide de Graphviz. Comme cela peut servir, voici rapidement comment installer Graphviz et l'utiliser pour générer un automate (qui est un graphe orienté).

Sur les systèmes Debian et dérivés, Graphviz est disponible sous forme de paquet :

```
$ sudo apt install graphviz
```

Si vous ne le trouvez pas pour votre distribution dans le gestionnaire de paquets, vous pourrez le télécharger sur le site officiel [3].

On doit ensuite décrire l'automate en utilisant le langage de description dot (à ne pas confondre avec un langage informatique qui, lui, est doté d'instructions de boucle ou de conditions). Nous prendrons par exemple un fichier `figure_01.dot` :

```
01: digraph figure_01 {
02:     rankdir = LR;
03:     size = "8,5"
04:
05:     node [shape = point, color=white, fontcolor=white]; start;
06:     node [shape = doublecircle, color=black, fontcolor=black];
q1, q2, q3, q4;
07:     node [shape = circle]; q0;
08:
09:     start -> q0;
10:     q0 -> q1 [ label = "0" ];
11:     q1 -> q2 [ label = "1" ];
12:     q1 -> q3 [ label = "0" ];
13:     q0 -> q2 [ label = "1" ];
14:     q2 -> q1 [ label = "0" ];
15:     q2 -> q4 [ label = "1" ];
16: }
```

La description du graphe est comprise dans un bloc `digraph <nom_du_graphe>`, car il s'agit d'un graphe orienté (on peut utiliser `graph` pour un graphe non orienté).

En ligne 2, on spécifie l'orientation du graphe dans `rankdir` : de la gauche vers la droite (`LR` pour *Left to Right*) et en ligne 3, on donne une taille maximale au graphe (en pouces).

Les lignes 5 à 7 définissent comment seront représentés les nœuds du graphe :

- le nœud `start` sera un point et son nom sera affiché en blanc sur fond blanc (il sera donc invisible) ;
- les nœuds `q1`, `q2`, `q3`, et `q4` seront représentés par un double cercle (états finaux) ;
- le nœud `q0` sera représenté par un cercle simple (état initial).

Les lignes 9 à 15 sont ensuite des retranscriptions de la fonction de transition δ que nous avons définie au point 5 de la section 1.1. La syntaxe est : `noeud_depart -> noeud_arrivee [ label = "étiquette de l'arête" ]`.

Pour générer l'image, on fait appel au programme `dot` en spécifiant le format de sortie sous la forme `-T<format>` :

```
$ dot -Tpng figure_01.
dot -o figure_01.png
```

Le résultat de cette commande est bien sûr visible en figure 1.

1.2 On ajoute ensuite « cellulaire »

Passons maintenant aux automates cellulaires, introduits en parallèle dans les années 1940 par Stanislas Ulam et John von Neumann. Ulam s'intéressait à l'évolution de constructions graphiques engendrées à partir de règles simples dans un espace à deux dimensions divisé en « cellules » qui ne peuvent être que dans l'état « allumé » ou « éteint ». Ces cellules évoluent dans le temps en fonction des cellules voisines. Partant d'une grille de cellules et d'une configuration donnée, Ulam a pu constater qu'en débutant avec des règles très simples, on pouvait aboutir à des structures très complexes. Et en parallèle, Neumann travaillait sur les automates autoréplicateurs dans le but de créer une machine capable de créer n'importe quelle autre machine décrite dans son programme et de se « cloner ». Ulam lui a alors proposé d'appliquer ses recherches sur les « espaces cellulaires » à son problème, ce qui donna lieu à la création d'un automate cellulaire. Mais c'est en 1970 que le jeu de la vie de John Conway (un type d'automate cellulaire parmi une infinité) apporta aux automates cellulaires la notoriété qu'on leur connaît aujourd'hui.

Formellement, un **automate cellulaire** (**AC**) est défini à l'aide d'un quadruplet (d, A, V, δ) où :

- `d` est la **dimension** de l'automate (pour `d=1` l'espace de représentation est linéaire, pour `d=2` c'est une grille, etc.) ;
- `A` est l'alphabet. On peut employer `{0, 1}` où `0` désigne une cellule morte et `1` une cellule vivante ;
- `V` est le **voisinage**. Ce tuple indique quelles sont les cellules voisines d'une cellule `i`. Par exemple, sur une dimension `d=1`, une cellule `i` a pour voisines les cellules `i-1` et `i+1`. Si le voisinage prend en compte toutes ces cellules, alors on pourra le noter `(-1, 0, 1)`. Si on imagine que le voisinage ne tient compte que des cellules de gauche, alors ce sera `(-1, 0)`. Nous reviendrons plus longuement sur cette notion lors de la définition de l'automate cellulaire du jeu de la vie avec une dimension `d=2` ;
- $\delta : A^n \rightarrow A$ est la fonction de transition qui prend en compte l'état de la cellule `i` et des cellules de son voisinage.

Pour mieux comprendre de quoi il retourne, nous allons créer un automate cellulaire extrêmement simple, appelé **automate cellulaire élémentaire** (ou **ACE**). Cet automate est défini par `(d, A, V, δ)` tel que :

- `d=1` (représentation linéaire) ;
- `A={0, 1}` (on choisit généralement une représentation où `0` est une case vide et `1` une case colorée) ;
- `V=(-1, 0, 1)` : au temps `t+1` (ou à l'étape `t+1`), l'état de la cellule `i` dépend de l'état des cellules `i-1`, `i`, et `i+1` au temps `t` ;
- la fonction de transition δ est donnée sous la forme d'un tableau (voir tableau ci-dessous).

On utilise ce tableau de la manière suivante : au temps `t+1`, on cherche la valeur de la cellule `i` ; on regarde alors au temps `t` quelles sont les valeurs des cellules `i-1`, `i`, et `i+1`. Par exemple, si au temps `t` la cellule `i-1` contenait `0`, la cellule `i` contenait `1` et la cellule `i+1` contenait également `1` (configuration `011`), alors au temps `t+1` la cellule `i` vaut `1`.

La figure 2 montre une représentation de l'évolution de cet automate dans le temps en partant d'une configuration `010`. J'ai modifié la représentation que l'on en fait habituellement pour le rendre plus compréhensible (mais moins esthétique).

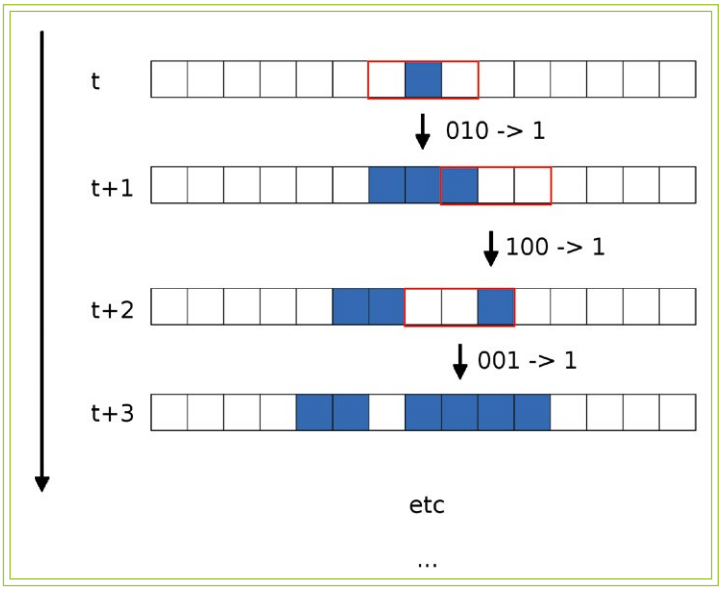


Fig. 2 : Évolution d'un ACE dans le temps avec détail du calcul de certaines cellules.

La représentation « classique » est donnée en figure 3 pour faire émerger le comportement global de l'automate, mais il faut garder à l'esprit que normalement on ne regarde qu'une seule ligne à un instant donné et que la ligne suivante représente l'évolution. On devrait donc obtenir une animation, comme dans le jeu de la vie que nous allons maintenant aborder.

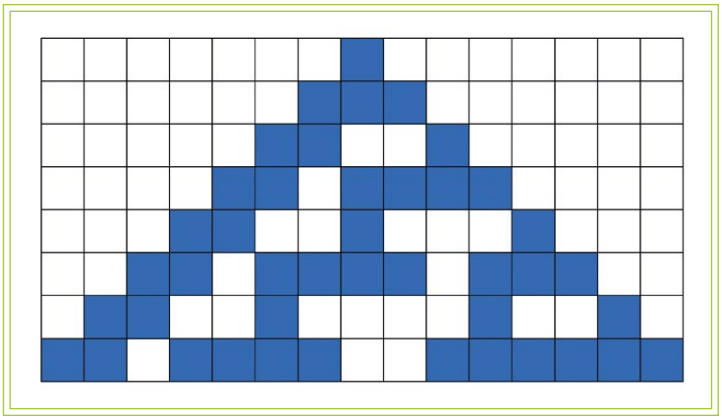


Fig. 3 : Représentation « classique » d'un ACE (temps croissant de haut en bas).

Temps	Configurations							
t	000	001	010	011	100	101	110	111
t + 1	0	1	1	1	1	0	0	0

2. LE JEU DE LA VIE

Le jeu de la vie est un automate cellulaire **(d, A, V, δ)** tel que :

- **d=2** (représentation sur une grille supposée infinie) ;
- **A={0, 1}** (pour **0**, une cellule est inactive ou morte et pour **1** une cellule est active ou vivante) ;
- **V=((-1, -1), (-1, 0), (-1, 1), (0, 0), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1))** : il s'agit du voisinage de Moore d'ordre 1, c'est-à-dire les 8 cellules qui entourent la cellule centrale (voir figure 4) ;

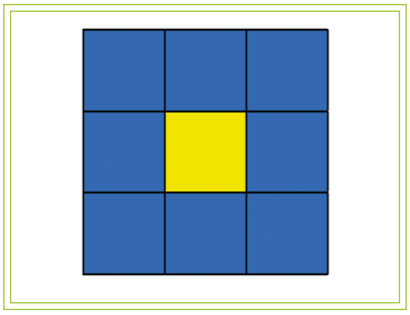


Fig. 4 : Voisinage de Moore (en bleu) d'une cellule (en jaune). Pour une cellule (x,y) en jaune, le voisinage sera composé des cellules (x-1, y-1), (x, y-1), (x+1, y-1), (x-1, y), (x+1, y), (x-1, y+1), (x, y+1), et (x+1, y+1).

- la fonction de transition **δ** répond aux règles suivantes :
  - → une cellule vivante avec moins de 2 voisins vivants meurt (sous-population) ;
  - → une cellule vivante avec 2 ou 3 voisins vivants reste vivante ;
  - → une cellule vivante avec plus de 3 voisins vivants meurt (surpopulation) ;
  - → une cellule morte avec exactement 3 voisins vivants devient vivante (naissance).

Nous allons maintenant voir comment implémenter ce jeu en Python.

2.1 La grille de jeu

Pour commencer, nous allons afficher une fenêtre contenant une grille représentant les différentes cellules. Notre programme s'appellera bien entendu **jeu_vie.py** :

```
01: import pygame
02: from pygame.locals import *
03:
04: def drawGrid(display, width, height, cellSize):
05:     for x in range(0, width, cellSize):
06:         pygame.draw.line(display, (0, 0, 0), (x, 0),(x, height))
07:     for y in range(0, height, cellSize):
08:         pygame.draw.line(display, (0, 0, 0), (0, y), (width, y))
09:
10: def initWindow(width=640, height=480, cellSize=10):
11:     if width % cellSize != 0:
12:         print('WARNING: width MUST be a multiple of cell size !')
13:         exit(1)
14:     if height % cellSize != 0:
15:         print('WARNING: height MUST be a multiple of cell size !')
16:         exit(2)
17:     pygame.init()
18:     display = pygame.display.set_mode((width, height))
19:     display.fill((255, 255, 255))
20:     pygame.display.set_caption('Jeu de la Vie')
21:     drawGrid(display, width, height, cellSize)
22:
23:
24: if __name__ == '__main__':
25:     initWindow()
26:
27:     while True:
28:         for event in pygame.event.get():
29:             if event.type == QUIT:
30:                 pygame.quit()
31:                 exit(0)
32:         pygame.display.update()
```

On commence par importer le module **pygame** dans les lignes 1 et 2 (**pygame.locals** pour avoir un accès direct à l'événement **QUIT**). La fonction **drawGrid()** des lignes 4 à 8 va nous permettre de tracer la grille du jeu en fonction d'une taille de fenêtre (déterminée par **width** et **height**), d'une taille de cellule (ou case) **cellSize**, et **display** qui sera un objet permettant d'accéder à la fenêtre pygame (voir la fonction **initWindow()**). Le tracé des lignes se fait simplement en traçant des traits verticaux (lignes 5 et 6) et horizontaux (lignes 7 et 8). Le principe est le suivant : pour **x** variant de **0** à la longueur de la fenêtre par pas de la taille de la cellule (pour une cellule de taille **10**, **x** vaudra **0**, **10**, **20**, etc.), on trace en noir (couleur **(0, 0, 0)**), la ligne partant de **(x, 0)** jusqu'à **(x, hauteur_de_la_fenetre)**. On obtient ainsi les traits verticaux et on adopte la même démarche pour les traits horizontaux.

La fonction **initWindow()** des lignes 10 à 21 permet d'afficher la fenêtre graphique du début du jeu. Pour cela, on doit spécifier une taille de fenêtre à l'aide des paramètres **width** et **height** (par défaut fenêtre de **640x480**), et une taille de cellule/case dans **cellSize**). Les lignes 11 à 16 permettent de s'assurer que la taille de fenêtre indiquée est compatible avec la taille des cellules (que l'on peut bien tracer un nombre entier de cellules dans cette fenêtre). On initialise ensuite le mode graphique en ligne 17, on définit la taille de la fenêtre (ligne 18), la couleur de fond de la fenêtre (blanc en rgb ou **(255, 255, 255)** en ligne 19) et le titre de la fenêtre en ligne 20. Enfin, on affiche la grille du jeu en appelant la fonction **drawGrid()** vue précédemment.

Pour terminer, dans le programme principal des lignes 24 à 32, on appelle la fonction d'initialisation **initWindow()** (ligne 25) puis on rentre dans la boucle événementielle infinie des lignes 27 à 32 (si l'événement **QUIT** est détecté, on quitte la fenêtre graphique).

En lançant ce code, on obtient pour l'instant simplement une grille dans une fenêtre graphique comme le montre la figure 5.

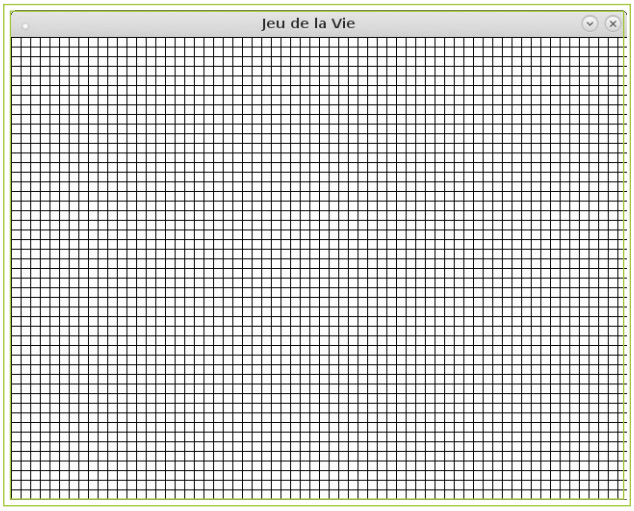


Fig. 5 : Affichage de la grille du jeu.

2.2 Peuplement aléatoire de départ

La représentation interne de notre grille sera assurée par un tableau **Numpy** contenant des **0** pour des cellules mortes et **1** pour des cellules vivantes. Nous allons donc ajouter une fonction qui va peupler aléatoirement notre grille interne de cellules vivantes et une fonction qui retranscrira cette grille interne dans la fenêtre graphique :

```
01: import pygame
02: from pygame.locals import *
03: import numpy
04: import random
...
12: def populate(width, height, cellSize):
13:     grid = numpy.zeros((width, height),
dtype=numpy.dtype('b'))
14:     for x in range(width // cellSize):
15:         for y in range(height // cellSize):
16:             grid[x, y] = random.randint(0, 1)
17:     return grid
18:
19: def displayCells(display, width, height,
cellSize, grid, color=(255, 155, 0)):
20:     for x in range(width // cellSize):
21:         for y in range(height // cellSize):
22:             if grid[x, y] == 1:
23:                 pygame.draw.rect(display, color, (x *
cellSize, y * cellSize, cellSize, cellSize))
24:             else:
25:                 pygame.draw.rect(display, (255, 255,
255), (x * cellSize, y * cellSize, cellSize, cellSize))
26:
27: def initWindow(width=640, height=480, cellSize=10):
...
38:     grid = populate(width, height, cellSize)
39:     displayCells(display, width, height, cellSize, grid)
40:     drawGrid(display, width, height, cellSize)
41:     return grid
42:
43:
44: if __name__ == '__main__':
...
```

Les modules **numpy** (ligne 3) et **random** (ligne 4) nous seront nécessaires pour travailler avec un **array** et avec des nombres aléatoires. La fonction **populate()** des lignes 12 à 17 renvoie un tableau de **width** x **height** cases de booléens (**'b'**) au sens **dtype** soit 8 bits (ligne 13) puis affectation aléatoire d'une valeur **0** ou **1** dans les lignes 14 à 16.

La fonction **displayCells()** va lire le tableau **grid** passé en paramètre et pour chaque cellule **grid[x, y]** notée à **1** afficher un carré orange (**color** qui vaut **(255, 155, 0)**) et blanc (**(255, 255, 255)**) sinon (lignes 22 à 25). Notez que l'on doit tout d'abord convertir la longueur et la hauteur en nombre de cases en divisant par **cellSize** les valeurs **width** et **height** (lignes 20 et 21) et que lors du tracé du carré (par **pygame.draw.rect()**) on doit effectuer la conversion inverse pour retrouver les coordonnées en fonction de la case en multipliant par **cellSize** (lignes 23 et 25).

On utilise ensuite ces fonctions **populate()** et **displayCells()** dans la fonction **initWindow()** (lignes 38 et 39) avant d'afficher la grille (ligne 40). En effet, nous traçons des cases de 10x10 pixels et si nous commençons par afficher la grille, nous allons la recouvrir et elle ne sera plus visible.

Le résultat obtenu avec ce code est visible en figure 6 (voir page suivante).

NOTE

L'utilisation de la division entière // en Python 3 permet d'éviter de convertir en entier le résultat d'une division effectuée par l'opérateur / :

```
>>> 10 / 2
5.0
>>> int(10 / 2)
5
>>> 10 // 2
5
```

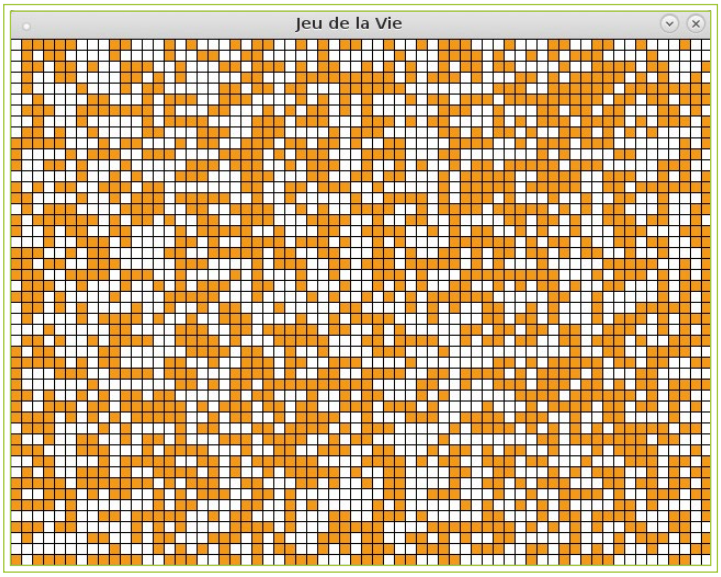


Fig. 6 : Grille du jeu peuplée aléatoirement de cellules vivantes.

2.3 Ajout des règles

Nous allons maintenant ajouter le calcul du voisinage d'une cellule qui nous permettra d'appliquer les quatre règles décrites précédemment :

```
01: import pygame
...
27: def initWindow(width=640, height=480, cellSize=10):
28:     if width % cellSize != 0:
29:         print('WARNING: width MUST be a multiple of
cell size !')
30:         exit(1)
31:     if height % cellSize != 0:
32:         print('WARNING: height MUST be a multiple of
cell size !')
33:         exit(2)
34:     pygame.init()
35:     clock = pygame.time.Clock()
...
42:     return (display, grid, clock)
43:
44: def getNeighbours(grid, cell, cellWidth, cellHeight):
45:     neighbours = 0
46:     for mod_x in range(-1, 2):
47:         for mod_y in range(-1, 2):
48:             if mod_x == 0 and mod_y == 0:
49:                 continue
50:             neighbourCell = (cell[0] + mod_x, cell[1]
+ mod_y)
51:             if (neighbourCell[0] < cellWidth  and
neighbourCell[0] >= 0) and \
52:                 (neighbourCell [1] < cellHeight and
neighbourCell[1] >= 0) and \
53:                 grid[neighbourCell[0],
neighbourCell[1]] == 1:
54:                                 neighbours += 1
55:     return neighbours
56:
57: def applyRules(grid, width, height,
cellSize, cellWidth, cellHeight):
58:     newGrid = numpy.zeros((width,
height), dtype=numpy.dtype('b'))
59:     for x in range(cellWidth):
60:         for y in range(cellHeight):
61:             nbNeighbours =
getNeighbours(grid, (x, y), cellWidth,
cellHeight)
62:             if grid[x, y] == 1:
63:                 if nbNeighbours < 2:
64:                     newGrid[x, y] = 0
65:                 elif nbNeighbours > 3:
66:                     newGrid[x, y] = 0
67:                 else:
68:                     newGrid[x, y] = 1
69:             else:
70:                 if nbNeighbours == 3:
71:                     newGrid[x, y] = 1
72:                 # else:
73:                     # newGrid[x, y] = 0
74:     return newGrid
75:
76: def gameOfLife(window=(640, 480),
cellSize=10, fps=10):
77:     (display, grid, clock) =
initWindow(window[0], window[1], cellSize)
78:     cellWidth = window[0] // cellSize
79:     cellHeight = window[1] // cellSize
80:
81:     while True:
82:         for event in pygame.event.
get():
83:             if event.type == QUIT:
84:                 pygame.quit()
85:                 exit(0)
86:
87:         grid = applyRules(grid,
window[0], window[1], cellSize, cellWidth,
cellHeight)
88:         displayCells(display,
window[0], window[1], cellSize, grid)
89:         drawGrid(display, window[0],
window[1], cellSize)
90:         pygame.display.update()
91:         clock.tick(fps)
92:
93:
94: if __name__ == '__main__':
95:     fps = 10
96:     window = (640, 480)
97:     cellSize = 10
98:
99:     gameOfLife(window, cellSize, fps)
```

On commence par ajouter une horloge à la fonction **initWindow()** en ligne 35. Cette horloge va nous permettre de rafraîchir l'affichage de la grille et de visualiser ainsi les différentes générations de cellules.

Ensuite, la fonction **getNeighbours()** nous renvoie le nombre de voisins (au sens du voisinage de Moore). Pour cela, on utilise deux boucles imbriquées où **mod_x** et **mod_y** prendront successivement pour valeur **-1**, **0** et **1**. Ceci permet de déterminer une cellule de voisinage (ligne 50), forcément différente de la cellule courante **cell** (les lignes 48 et 49 sont là pour ça). Ensuite, si cette cellule de voisinage reste dans la zone de la grille (lignes 51 et 52) et qu'elle est vivante (ligne 53), alors c'est un voisin de **cell** et on le compte dans la variable **neighbours** (ligne 54).

La fonction **applyRules()** permet de définir et d'appliquer les règles du jeu au tableau **grid**, générant par là même un nouveau tableau de la génération suivante de cellules **newGrid**. Pour appliquer les règles, on parcourt toutes les cellules de **grid** et on applique les règles :

- si la cellule est vivante :
  - → si elle a moins de 2 voisins, elle meurt (lignes 63 et 64) ;
  - → si elle a plus de 3 voisins, elle meurt (lignes 65 et 66) ;
  - → sinon elle reste vivante (lignes 67 et 68) ;
- si la cellule est morte :
  - → si elle a 3 voisins, une cellule naît (lignes 70 et 71) ;
  - → sinon elle reste morte (lignes 72 et 73, commentées car inutiles puisque les cases **newGrid** sont initialisées à **0**).

Enfin, tout le démarrage du jeu a été passé dans la fonction **gameOfLife()** dans les lignes 76 à 91 avec ajout du calcul du nombre de cases **cellWidth** et **cellHeight** (lignes 78 et 79), ce qui évite d'effectuer le calcul à de multiples reprises, puis le calcul de la nouvelle génération de cellules (ligne 87), l'affichage des cellules (ligne 88) et de la grille (ligne 89), le rafraîchissement de l'écran (ligne 90) et la mise à jour de l'horloge pour conserver un taux de *frames per second* (nombre d'images affichées par seconde) de **fps** (ligne 91).

Le programme principal se contente d'appeler **gameOfLife()** avec les bons paramètres pour lancer le jeu dont une capture est visible en figure 7.

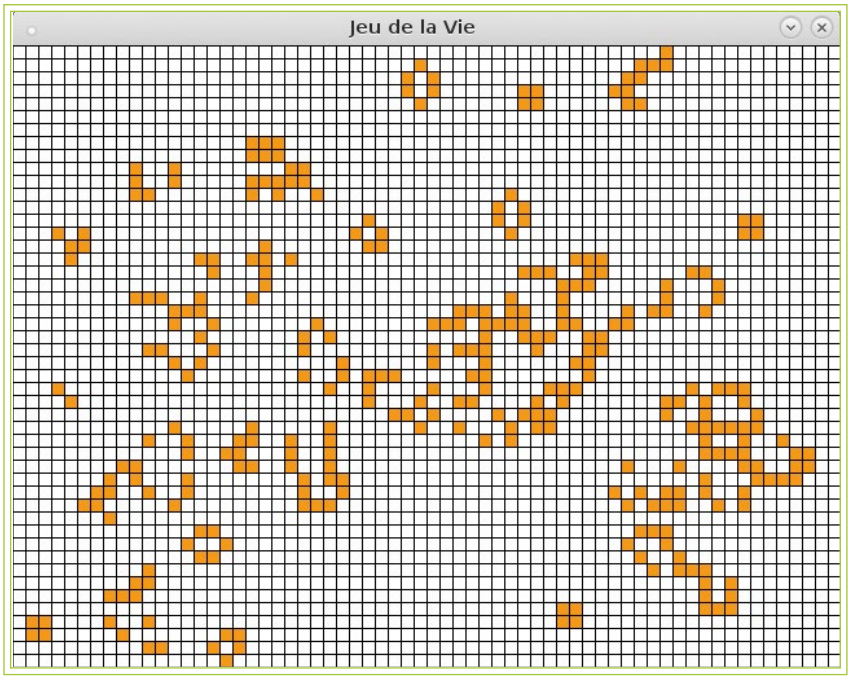


Fig. 7 : Exemple d'affichage du jeu de la vie.

Certaines des structures que vous obtiendrez seront statiques (comme le « block » en figure 8), d'autres oscilleront (comme le « blinker » en figure 9) et enfin d'autres se déplaceront sur la grille (comme le « glider » en figure 10).

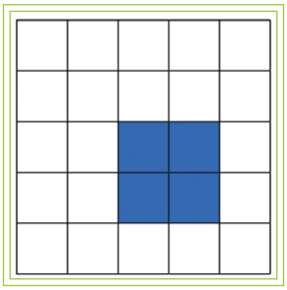


Fig. 8 : Le « block ».

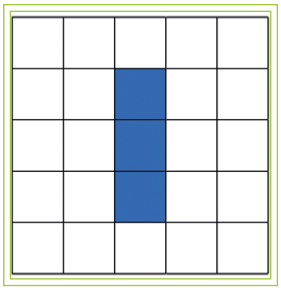


Fig. 9 : Le « blinker ».

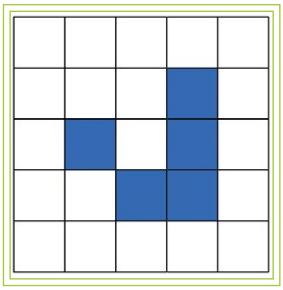


Fig. 10 : Le « glider ».

3. CAS PRATIQUES

Les automates cellulaires sont surtout employés pour effectuer des simulations comme la propagation d'un feu de forêt (la direction du vent modifie le calcul du voisinage) ou du clustering. Les cas pratiques d'applications ne sont pas légion, aussi ai-je essayé de trouver des exemples un peu plus originaux que ceux que l'on peut trouver d'habitude.

3.1 Détection de fichiers volumineux

Nous allons parcourir une arborescence de fichiers à la recherche des fichiers volumineux (ou tout autre critère que vous souhaiterez). Pour notre automate, nous allons nous retrouver dans le cadre d'un ACE dont nous allons légèrement modifier le fonctionnement :

- notre représentation sera linéaire (**d=1**), mais comme nous afficherons l'historique des générations, nous obtiendrons une grille ;
- notre alphabet sera composé de deux valeurs **A = {0, 1}** où **0** représentera un fichier « standard », et **1** représentera un fichier « cible » correspondant à la recherche ;
- **V=(0, 1)** : à l'étape **t+1**, l'état de la cellule **i** ne dépend de l'état que des cellules **i**, et **i+1** au temps **t** ;
- la fonction de transition δ est donnée sous la forme d'un tableau :

Temps	Configurations			
t	00	01	10	11
t + 1	0	1	0	1

Notre système se rapproche donc d'une machine de Turing même s'il n'en a pas exactement le formalisme. À chaque étape, nous allons décaler les **1** (les fichiers cibles) vers la droite et lorsqu'ils sortiront du ruban nous les afficherons. De plus, ce cas est hyper simple, car les règles de transition peuvent se résumer à analyser l'état des cellules **i+1** qui donneront leur valeur aux cellules **i**. Du point de vue de l'implémentation, on décale donc toutes les cellules du ruban vers la gauche... ce qui ne devrait pas être bien compliqué ! Nous n'utiliserons pas d'interface graphique pour ce cas trivial :

```
01: import os.path
02:
03: def populate(path, sizeMax):
04:     filesList = []
05:     for root, dirs, files in os.walk(path):
06:         for f in files:
07:             filename = os.path.join(root, f)
08:             if os.path.getsize(filename) > sizeMax:
09:                 cell = 1
10:             else:
11:                 cell = 0
12:             filesList.append((filename, cell))
13:     return filesList
14:
15: def applyRules(strip):
16:     current = strip.pop(0)
17:     if current[1] == 1:
18:         print('Hit :', current[0])
19:
20: def detectBigFiles(path, sizeMax=5000):
21:     strip = populate(path, sizeMax)
22:     while strip != []:
23:         applyRules(strip)
24:
25:
26: if __name__ == '__main__':
27:     detectBigFiles('/home')
```

La liste des fichiers est obtenue à l'aide du module **os.path** (importé en ligne 1) et au parcours récursif des répertoires (ligne 5) de la fonction **populate()**. La structure du ruban est une liste contenant des tuples **(nom_fichier, valeur)** où **valeur** vaut **1** si la taille du fichier est supérieure à **sizeMax** et **0** sinon (lignes 8 à 11).

Les règles ont été simplifiées au maximum et nous n'effectuons qu'un décalage à gauche de toutes les cellules dans la fonction **applyRules** (ligne 16) avec affichage de la cellule retirée si son état est **1** (lignes 17 et 18). Il ne reste plus qu'à appliquer cette fonction tant qu'il y a des données sur le ruban (lignes 22 et 23 de la fonction **detectBigFiles()**).

Cet exemple est intéressant non pas pour sa difficulté, mais pour la simplification de l'automate effectuée en amont. Sans réflexion nous aurions pu partir sur un voisinage **(-1, 0, 1)** avec une fonction de transition comportant 8 configurations et implémenter cela de manière brutale... ce qui aurait été beaucoup plus long et moins efficace.

3.2 Profilage de code

Nous allons maintenant afficher un profil graphique d'un code Python sous la forme d'une grille où chaque case correspond à un caractère :

- en blanc pas de caractère, la cellule est vide ;
- en gris un caractère faisant partie d'un commentaire (débutant par **#**) ;
- en jaune un caractère d'indentation ;
- en rouge un caractère faisant partie d'une ligne précédant un nouveau bloc ;
- et en bleu un caractère quelconque.

Pour le profilage de code, l'automate cellulaire se rapprochera de celui du jeu de la vie avec **(d, A, V, δ)** tel que :

- `d=2` ;
- `A={0, 1, 2, 3, 4, 5}` (pour `0`, pas de caractère, pour `1` commentaire, pour `2` début de bloc, pour `3` indentation, pour `4` caractère d'espacement et pour `5` caractère autre) ;

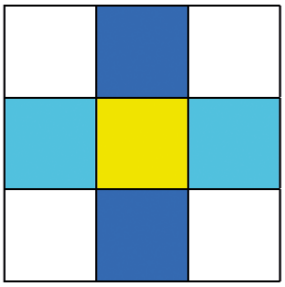


Fig. 11 : Voisinage de von Neumann d'ordre 1 (en bleu clair et bleu foncé) restreint à gauche et à droite (en bleu clair seulement).

- `V=((-1, 0), (0, 0), (0, 1))` : il s'agit du voisinage de von Neumann d'ordre 1 restreint à gauche et à droite (voir figure 11).
- la fonction de transition δ répond aux règles suivantes :
  - → une cellule caractère :
    - ▸ ayant pour voisin gauche une cellule commentaire devient une cellule caractère ;
    - ▸ ayant pour voisin droit une cellule bloc devient une cellule bloc ;
  - → une cellule espacement ayant pour voisin gauche une cellule indentation devient une cellule indentation ;
  - → une cellule bloc ayant pour voisin gauche une cellule commentaire devient une cellule commentaire.

Je n'ai pas traité ici absolument tous les cas, mais seulement le piège du commentaire contenant un caractère `:` et qui aurait pu être compté comme un bloc (il faudrait ajouter le cas où ce caractère est à l'intérieur d'une chaîne de caractères délimitée par `''` ou `""`). Au niveau du code, on repart de `jeu_vie.py` que l'on modifie légèrement :

```
001: import pygame
002: from pygame.locals import *
003: import numpy
004: import random
005: import sys
006: import math
```

```
007:
008: def drawGrid(display, width, height, cellSize):
...
014: def populate(code, width, height, cellSize):
015:     grid = numpy.zeros((width, height),
dtype=numpy.dtype('b'))
016:     for y, line in enumerate(code):
017:         for x, car in enumerate(line):
018:             if car == '#':
019:                 grid[x, y] = 1
020:             elif car == ':':
021:                 grid[x, y] = 2
022:             elif (car == ' ' or car == '\t' or car
=='\n') :
023:                 if x == 0:
024:                     grid[x, y] = 3
025:                 else:
026:                     grid[x, y] = 4
027:             else:
028:                 grid[x, y] = 5
029:     return grid
030:
031: def displayCells(display, width, height, cellSize,
grid, colors):
032:     for x in range(width // cellSize):
033:         for y in range(height // cellSize):
034:             if grid[x, y] != 0:
035:                 pygame.draw.rect(display,
colors[grid[x, y]], (x * cellSize, y * cellSize,
cellSize, cellSize))
036:             else:
037:                 pygame.draw.rect(display, (255,
255, 255), (x * cellSize, y * cellSize, cellSize,
cellSize))
038:
039: def initWindow(code, codeName, colors, width=640,
height=480, cellSize=10):
040:     if width % cellSize != 0:
041:         print('WARNING: width MUST be a multiple
of cell size !')
042:         exit(1)
043:     if height % cellSize != 0:
044:         print('WARNING: height MUST be a multiple
of cell size !')
045:         exit(2)
046:     pygame.init()
047:     clock = pygame.time.Clock()
048:     display = pygame.display.set_mode((width,
height))
049:     display.fill((255, 255, 255))
050:     pygame.display.set_caption('Profilage de ' +
codeName)
051:     grid = populate(code, width, height, cellSize)
052:     displayCells(display, width, height, cellSize,
grid, colors)
053:     drawGrid(display, width, height, cellSize)
054:     return (display, grid, clock)
055:
056: def applyRules(grid, width, height, cellSize,
cellWidth, cellHeight):
```

```
057:     newGrid = numpy.zeros((width, height),
dtype=numpy.dtype('b'))
058:     for x in range(cellWidth):
059:         for y in range(cellHeight):
060:             if grid[x, y] == 5 or grid[x,
y] == 4:
061:                 if x > 0 and grid[x - 1,
y] == 1:
062:                     newGrid[x, y] = 1
063:                 elif x > 0 and grid[x, y]
== 4 and grid[x - 1, y] == 3:
064:                     newGrid[x, y] = 3
065:                 elif x < cellWidth and
grid[x + 1, y] == 2:
066:                     newGrid[x, y] = 2
067:                 else:
068:                     newGrid[x, y] =
grid[x, y]
069:             elif grid[x, y] == 2 and x >
0 and grid[x - 1, y] == 1:
070:                 newGrid[x, y] = 1
071:             else:
072:                 newGrid[x, y] = grid[x, y]
073:     return newGrid
074:
075: def profile(code, codeName, colors,
window=(640, 480), cellSize=10, fps=10):
076:     (display, grid, clock) =
initWindow(code, codeName, colors, window[0],
window[1], cellSize)
077:     cellWidth = window[0] // cellSize
078:     cellHeight = window[1] // cellSize
079:
080:     while True:
081:         for event in pygame.event.get():
082:             if event.type == QUIT:
083:                 pygame.quit()
084:                 exit(0)
085:
086:         grid = applyRules(grid, window[0],
window[1], cellSize, cellWidth, cellHeight)
087:         displayCells(display, window[0],
window[1], cellSize, grid, colors)
088:         drawGrid(display, window[0],
window[1], cellSize)
089:         pygame.display.update()
090:         clock.tick(fps)
091:
092: def getWindowSize(code):
093:     height = len(code)
094:     width = 0
095:     for line in code:
096:         sizeLine = len(line)
097:         if sizeLine > width:
098:             width = sizeLine
099:
100:     return (width * 10, height * 10)
101:
102:
103: if __name__ == '__main__':
104:     if len(sys.argv) != 2:
```

```
105:         print('Syntax : profilage
<filename.py>')
106:         exit(3)
107:
108:     try:
109:         with open(sys.argv[1], 'r') as fic:
110:             code = fic.readlines()
111:     except:
112:         print('File error')
113:         exit(4)
114:
115:     fps = 10
116:     window = getWindowSize(code)
117:     cellSize = 10
118:     colors = [
119:         (255, 255, 255), # blanc
120:         (211, 211, 211), # gris
121:         (255, 0, 0), # rouge
122:         (255, 215, 0), # jaune
123:         (0, 191, 255), # bleu
124:         (0, 191, 255), # bleu
125:     ]
126:
127:     profile(code, sys.argv[1], colors,
window, cellSize, fps)
```

Les changements majeurs sont :

- le calcul de la taille de la fenêtre en fonction du nombre de caractères du fichier (fonction **`getWindowSize()`** dans les lignes 92 à 98) ;
- l'utilisation d'une liste de couleurs pour les différents états (lignes 118 à 125) ;
- la fonction **`populate()`** des lignes 14 à 29 qui associe un état différent en fonction du type de caractère ;
- la fonction **`applyRules()`** des lignes 56 à 73 qui est simplement la traduction des règles de transitions énoncées précédemment.

NOTE

J'ai conservé ici la boucle infinie du jeu de la vie, mais comme la tâche est bornée, il faudrait ajouter un compteur de modifications et arrêter la boucle lorsqu'il n'y a plus de modification entre la génération `t-1` et la génération `t`.

La capture de la figure 12 donne un exemple de début d'exécution de ce code sur un fichier Python et la figure 13 présente le résultat final.

CONCLUSION

Ce qui est remarquable avec les automates cellulaires c'est leur faculté de partir de règles simples au niveau élémentaire pour obtenir un comportement complexe au niveau global.

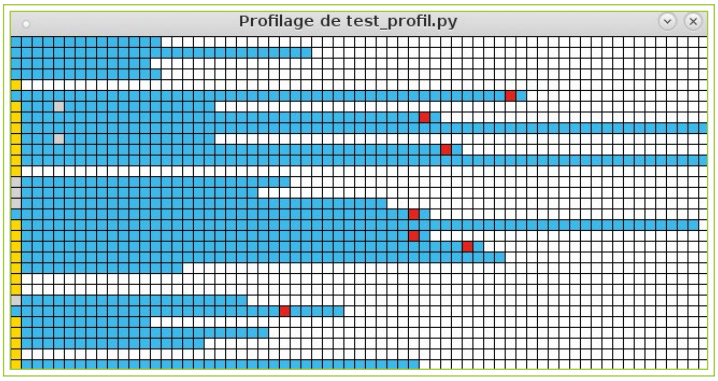


Fig. 12 : Début d'exécution sur un fichier Python.

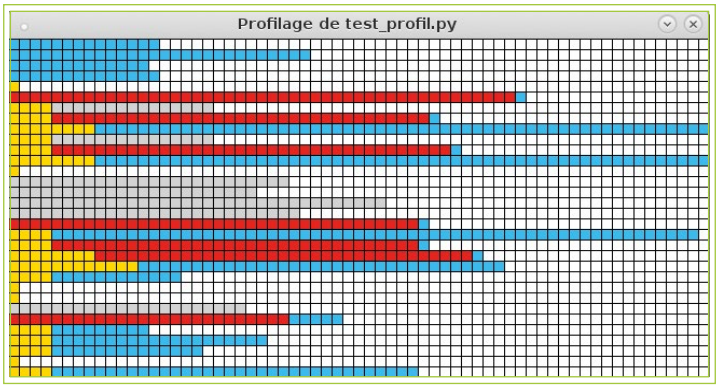


Fig. 13 : Résultat du profilage du fichier Python.

Nous avons pu en avoir un exemple en réalisant le Jeu de la Vie de Conway et nous avons utilisé ces automates pour résoudre des problèmes concrets. Par manque de place, je n'ai pas pu aborder un automate cellulaire particulier, pourtant digne d'intérêt : la fourmi de Langton. Si le sujet vous intéresse, vous pourrez consulter une vidéo introductive **[4]** présentant cet automate au comportement... assez particulier !

À vous maintenant d'expérimenter cela et de trouver des domaines d'application ! ■

RÉFÉRENCES

[1] Numberphile, interview de John Conway, *« Does John Conway hate his Game of Life ? »* : https://youtu.be/E8kUJL04ELA

[2] Numberphile, interview de John Conway, *« Inventing Game of Life »* : https://youtu.be/R9Plq-D1gEk

[3] Graphviz : http://www.graphviz.org/Download.php

[4] Science Étonnante, « La fourmi de Langton » : https://youtu.be/qZRYGxF6D3w

UTILISER (ENFIN) MIDNIGHT COMMANDER

TRISTAN COLOMBO

MOTS-CLÉS : MIDNIGHT COMMANDER, FICHIERS, CONFIGURATION, INTERFACE MODE TEXTE

Lorsque l'on veut naviguer dans l'arborescence de fichiers, il n'y a que trois solutions : le mode graphique avec des applications telles que Nautilus, le mode ligne de commandes dans un terminal ou le mode « hybride » avec Midnight Commander. Dans cet article, nous allons étudier la dernière solution dans le détail.

Midnight Commander fait partie de ces outils bien connus des administrateurs système, mais sans doute moins des développeurs. Pour ma part cela, fait des années que régulièrement je me dis « allez, cette fois je m'y mets »... et que finalement, après avoir navigué dans deux ou trois répertoires je me dis qu'effectivement l'outil a un fort potentiel, mais que ce n'est pas le moment, car il est trop long à prendre en main. Je pense que c'est la même réflexion que pour la personne qui souhaite se mettre à **Vim** : il faut du temps et on abandonne rapidement. J'ai donc décidé de sauter le pas et de relater mon expérience dans cet article.

1. INSTALLATION

Midnight Commander n'est pas installé par défaut. Sur un système **GNU/Debian**, cela se fait très simplement, à partir du moment où l'on connaît le nom du paquetage bien sûr :

```
$ sudo apt install mc
```

2. DÉMARRAGE ET USAGE DE BASE

Non, ne cherchez pas la doc... il existe bien https://midnight-commander.org/wiki/doc, mais il s'agit d'un brouillon. À la rigueur vous pouvez consulter la (longue) page de man, mais vous l'aurez compris, le plus simple sera de plonger dans cet outil tête la première et avec une visibilité plus que restreinte (nous verrons plus tard comment accéder à l'aide interactive qui, elle, est un peu plus complète). Pour effectuer le grand saut :

```
$ mc
```

Vous êtes maintenant face à un terminal en fond bleu comprenant un menu supérieur, un menu inférieur (sous lequel se trouve un Shell) et deux panneaux (voir figure 1).

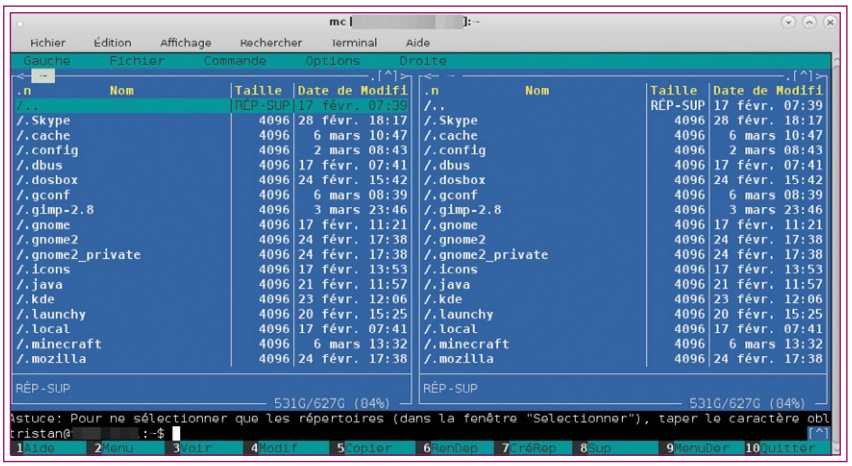


Fig. 1 : Démarrage de Midnight Commander.

Les deux panneaux sont des représentations de l'arborescence des fichiers contenus dans le répertoire d'où vous avez lancé votre commande **mc**. En utilisation basique, on va pouvoir se déplacer à l'intérieur d'un panneau à l'aide des touches **<Flèche_Haut>** et **<Flèche_Bas>** et passer d'un cadre à l'autre grâce à la touche **<Tab>**. Le menu du bas indique les actions possibles depuis les raccourcis **<F1>** à **<F10>**.

Ainsi, pour copier un fichier depuis le panneau de gauche vers l'onglet de droite, on va :

1. Naviguer dans le panneau de gauche à l'aide des flèches et de la touche **<Return>** pour rentrer dans les répertoires jusqu'à ce que l'on trouve le fichier à copier ;
2. Passer sur le panneau de droite en appuyant sur **<Tab>**, puis naviguer jusqu'au répertoire cible ;
3. Revenir dans le panneau de gauche (**<Tab>**) et se positionner sur le fichier à copier puis appuyer sur **<F5>**.

Une fenêtre de confirmation apparaîtra, comme sur l'exemple de la figure 2.

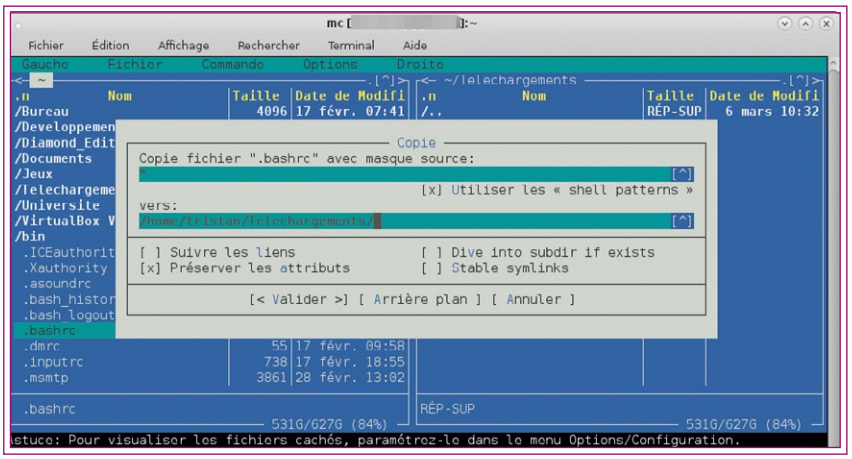


Fig. 2 : Exemple : copie du fichier ~/.bashrc dans ~/Telechargements.

NOTE

Vous avez également la possibilité de vous déplacer dans l'arborescence des panneaux en utilisant la commande cd depuis le Shell du bas.

Je ne rentrerai pas dans le détail sur la manière de renommer, déplacer, supprimer un fichier ou encore créer un répertoire, car tout cela suit la même logique que ce qui a été énoncé pour la copie. Par contre, puisque nous parlions de documentation au début de cette section, sachez qu'en appuyant sur la touche **<F1>** vous bénéficierez d'une aide interactive. Pensez à déplacer la zone active (flèche vers le bas) sur ***contents*** avant d'appuyer sur **<Return>** de manière à accéder au sommaire de l'aide (voir figure 3).

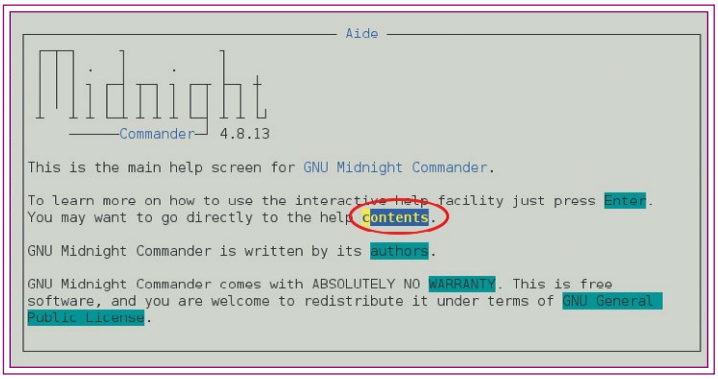


Fig. 3 : Accès au sommaire de l'aide.

Les présentations étant faites, nous allons aborder les questions qui ne manquent pas de se poser à l'usage.

NOTE

Vérifiez la configuration des touches de raccourcis de votre terminal... il est fort possible que la touche <F1> soit déjà affectée à la page d'aide de celui-ci et que par conséquent vous n'ayez pas accès à l'aide de Midnight Commander.

Sur Gnome-terminal, il faut également désactiver la touche d'accès au menu (<F10>) dans le premier onglet des préférences.

3. LES « COMMENT ?»

Dans cette section, je vous propose des réponses aux questions de configuration que je me suis posées.

3.1 Comment masquer les fichiers cachés (préfixés par un point) ?

On a souvent à travailler avec les fichiers cachés... mais lorsque l'on recherche un fichier « simple », la vue de tous les répertoires et fichiers cachés peut devenir très rapidement fatigante ! Le raccourci **<Alt>** + **<.>** (donc **<Alt>** + **<Shift>** + **<.>** en « vrai ») permet de basculer entre les modes d'affichage avec et sans fichiers cachés.

Pour régler le comportement par défaut de Midnight Commander vis-à-vis de ces fichiers, rendez-vous dans le menu ***Option des panneaux...*** (**<F9>** puis ***Options > Options des panneaux...***) et décochez la case ***Afficher les fichiers cachés*** comme le montre la figure 4. En désactivant l'option à ce niveau, à chaque démarrage de Midnight Commander les fichiers cachés ne seront pas affichés. Il suffira d'appuyer sur **<Alt>** + **<Shift>** + **<.>** pour les voir apparaître.

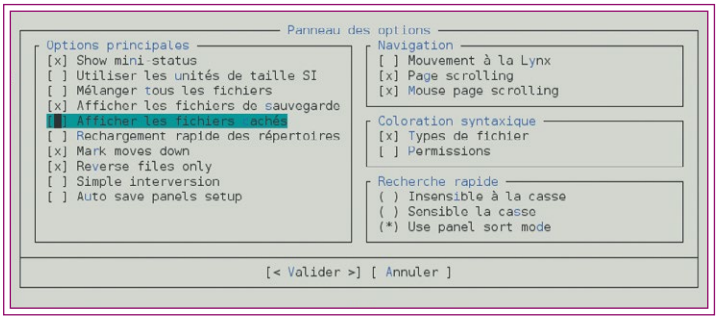


Fig. 4 : Modification de l'option d'affichage par défaut des fichiers cachés.

3.2 Comment rafraîchir le contenu d'un répertoire ?

Quand un nouveau fichier apparaît dans le répertoire d'un panneau, celui-ci n'est pas affiché automatiquement. Pour rafraîchir l'affichage d'un panneau, vous devez utiliser **<Ctrl>** + **<R>**.

3.3 Comment sélectionner plusieurs fichiers ?

La sélection/désélection de fichiers se fait avec la touche **<Insert>**. Il est possible d'utiliser une sélection par filtre :

- **<+>** : sélectionne les fichiers qui correspondent au filtre indiqué (une nouvelle fenêtre demandant ce filtre s'ouvre) ;
- **<\>** (ou plutôt **<Alt-Gr>** + **<\>**) : désélectionne les fichiers qui correspondent au filtre indiqué (il faut bien entendu que des fichiers aient déjà été sélectionnés) ;
- **<*>** : inverse la sélection.

Il est ainsi possible de copier, supprimer ou déplacer des lots de fichiers.

3.4 Comment utiliser le fichier sélectionné dans le Shell ?

En tapant une commande dans le shell, si vous appuyez sur **<Ctrl>** + **<Return>**, le nom du fichier actif dans Midnight Commander (le fichier surligné) apparaîtra dans votre commande. Plus fort, avec **<Ctrl>** + **<Shift>** + **<Return>** ce sera le fichier et son chemin complet qui apparaîtront dans le Shell.

NOTE

La complétion automatique du Shell utilise la touche <Tab> qui est elle-même employée par Midnight Commander... Pour utiliser la complétion, deux solutions s'offrent à vous :

- **soit vous utilisez <Esc> + <Tab> dans le Shell se trouvant sous Midnight Commander ;**
- **soit vous passez en vue shell par <Ctrl> + <O>, vous utilisez <Tab> normalement pour la complétion et vous revenez à Midnight Commander par <Ctrl> + <O>. Par contre, vous n'aurez plus accès au nom du fichier sélectionné par <Ctrl> + <Return>... on ne peut pas tout avoir !**

3.5 Comment créer un menu de raccourcis vers les répertoires les plus utilisés ?

Il existe un menu de raccourcis vers des répertoires définis par l'utilisateur. Ce menu est accessible par **<Ctrl>** + **<Alt-Gr>** + **<\>** ou par **<F9>** puis ***Commande > Répertoire hotlist***. Créez simplement des groupes de répertoires en sélectionnant ***Nouveau groupe*** et en attribuant un nom au groupe puis ajoutez des entrées par ***Nouvelle Entrée***. La sélection se fait ensuite à l'aide d'une liste déroulante (voir figure 5). C'est tout bête, mais ça permet de gagner beaucoup de temps quand on passe sa journée à aller d'un répertoire à l'autre et ça évite de garder des dizaines d'onglets ouverts...

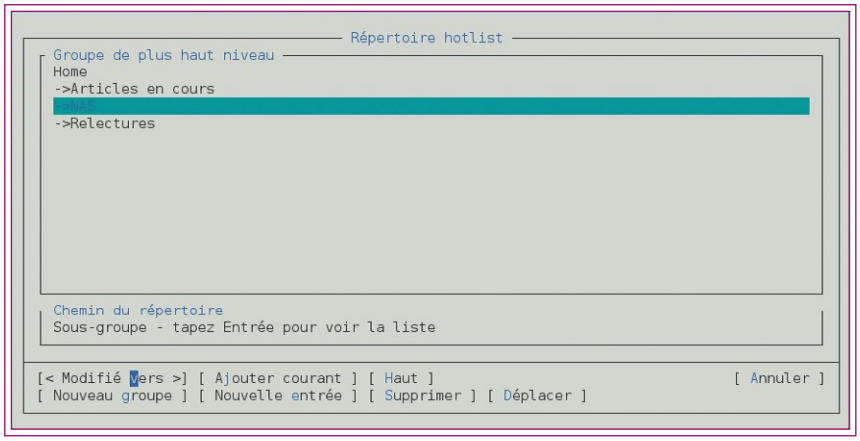


Fig. 5 : Le menu hotlist.

Si vous souhaitez modifier l'ordre des entrées de votre menu ou même concevoir votre menu un peu plus vite, éditez le fichier **~/.config/mc/hotlist**. Sa structure est très simple :

• Pour déclarer une entrée :

```
ENTRY "Home" URL "/home/tristan"
```

• Pour déclarer un groupe :

```
GROUP "Articles en cours"
  ENTRY "EN COURS" URL "/home/tristan/..."
  ...
ENDGROUP
```

• Pour déclarer un sous-groupe :

```
GROUP "Articles en cours"
  GROUP "Propositions Couv"
    ...
  ENDGROUP
  ...
ENDGROUP
```

• etc.

3.6 Comment configurer les applications à lancer pour ouvrir les fichiers ?

Il est possible que l'ouverture de certains fichiers ne se fasse pas avec votre application favorite ou même tout simplement qu'ils ne s'ouvrent pas... Heureusement, nous pouvons affecter un comportement particulier aux différents types de fichiers pour l'ouverture (appui simple sur **<Return>**), la vue (appui sur **<F3>**) ou l'édition (appui sur **<F4>**). Ceci se fait en créant un fichier **~/.config/mc/mc.ext**. Prenons l'exemple des fichiers png ou jpg : généralement je veux juste les voir (**GwenView**), mais parfois je dois les modifier (**Gimp** ou **Shutter**). Il faut alors déclarer les expressions régulières permettant de retrouver les fichiers concernés, les affecter à un groupe puis indiquer le comportement à adopter pour les fichiers du groupe :

```
# Traitement des images
regex/\.(png|PNG)$
    Include=image

regex/\.(jpg|JPG|jpeg|JPEG])$
    Include=image

include/image
    View=gimp %s
    Open=gwenview %s
    Edit=shutter %p
```

Avec ces règles, en appuyant sur **<F3>** nous lancerons Gimp (**View**), sur **<Return>** nous lancerons GwenView (**Open**) et sur **<F4>** nous lancerons Shutter (**Edit**). Le **%s** désigne la sélection en cours (nous aurions pu utiliser **%f** pour utiliser seulement le fichier en surbrillance). Si vous avez sélectionné trois fichiers **fic1**, **fic2** et **fic3**, **%s** contiendra la liste des fichiers séparés par un espace : **fic1 fic2 fic3**.

%p correspond au chemin absolu vers le fichier (utile pour certaines applications comme ici Shutter).

NOTE

Dans le cas d'une mise en pratique de cet exemple, l'utilisation d'un menu utilisateur personnalisé (voir dans la suite) est sans doute plus adaptée.

Il est également possible de ne pas utiliser de groupe si l'action porte sur une seule expression :

```
# Traitement des pdf
regex/\.(pdf|PDF)$
    Open=masterpdfeditor4 %f
```

Certaines applications devront être lancées en arrière-plan :

```
# Traitement des fichiers LibreOffice
regex/\.(odt|ods|odf|odp)$
    Open=(libreoffice %s &)
```

ATTENTION !

Il ne doit pas y avoir d'espace avant et après le signe égal définissant une règle.

Tout cela est bien pratique, car nous avons complètement la main sur la façon dont seront ouverts les fichiers... mais nous avons perdu les comportements par défaut ! Ceux-ci sont présents dans le fichier **/etc/mc/mc.ext** et je vous invite à copier les règles qui vous intéressent dans votre fichier **~/.config/mc/mc.ext**. A minima, vous aurez sans doute besoin de la règle permettant de visualiser le contenu des fichiers **tar.gz** :

```
# .tgz, .tpz, .tar.gz, .tar.z, .tar.Z, .ipk, .gem
regex/\.t([gp]?z|ar\.g?[zZ])$|\.ipk$|\.gem$
 Open=%cd %p/utar://
 View=%view{ascii} /usr/lib/mc/ext.d/archive.sh
view tar.gz
```

Celle-ci fait intervenir le *Virtual File System* de Midnight Commander : vous pouvez voir les fichiers de l'archive comme s'il s'agissait d'un simple répertoire...

NOTE

Au début du fichier /etc/mc/mc.ext, vous trouverez des indications sur les différents types de variables que vous pouvez utiliser.

3.7 Comment rechercher des fichiers ?

Avec **<Alt>** + **<Shift>** + **<?>**, vous accéderez à la fenêtre (voir figure 6) permettant de paramétrer une recherche de fichiers. Il s'agit d'un **find** / **grep** simplifié et graphique qui devrait correspondre à 90% des recherches... pour le reste il faudra utiliser la ligne de commandes ;-)

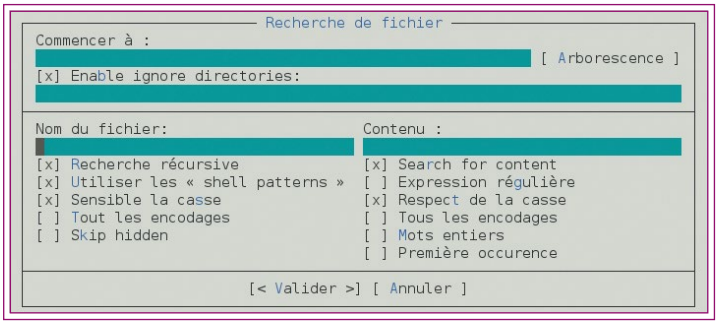


Fig. 6 : Recherche de fichiers dans Midnight Commander.

3.8 Comment compresser/décompresser des fichiers ?

Pour compresser ou décompresser des fichiers, il existe un menu spécial très pratique : le menu utilisateur, accessible en appuyant sur **<F2>** (voir figure 7). Les entrées de ce menu s'adaptent au type de fichier en surbrillance au moment de l'appui sur **<F2>** (si vous vous trouvez sur un fichier **tar.gz**, vous pourrez voir une entrée permettant de décompresser et d'extraire le contenu de l'archive).

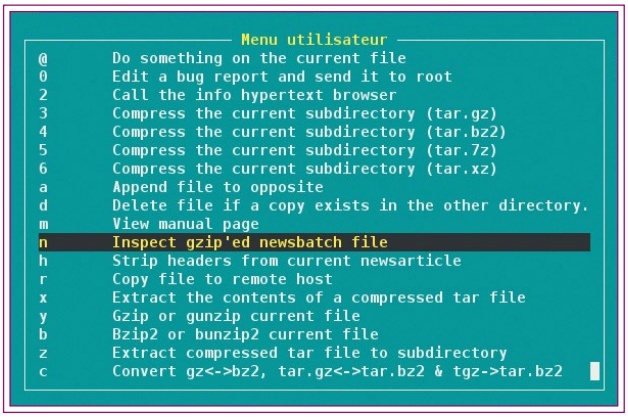


Fig. 7 : Aperçu du menu utilisateur.

3.9 Comment personnaliser le menu utilisateur ?

Les entrées du menu utilisateur par défaut sont définies dans le fichier **/etc/mc/mc.menu**. La personnalisation du menu se fait dans **~/.config/mc/menu** où l'on peut définir un nouveau jeu d'entrées en fonction de l'extension du fichier en surbrillance ou ajouter des entrées à celles données par défaut. Pour ne pas perdre tout le comportement par défaut du menu, je vous recommande de copier le fichier d'origine :

```
$ cp /etc/mc/mc.menu ~/.config/mc/menu
```

Voici un petit exemple de menu pour les fichiers Python :

```
shell_patterns=0

...

=+ f \.py$
rExecute Python file
 python3 %f

=+ f \.py$
pTest pep8
 pep8 %f | more
```

La première ligne est essentielle pour la reconnaissance des règles sur les fichiers sous forme d'expressions régulières. Celles-ci commencent ici par **=+** : si la condition qui suit est vraie alors l'entrée est ajoutée et devient l'entrée par défaut (attention toutefois si d'autres conditions sont respectées et définissent déjà une entrée par défaut, le résultat ne sera pas forcément celui attendu).

Ici **=+ f \.py$** crée une nouvelle entrée pour les fichiers d'extension **.py**.

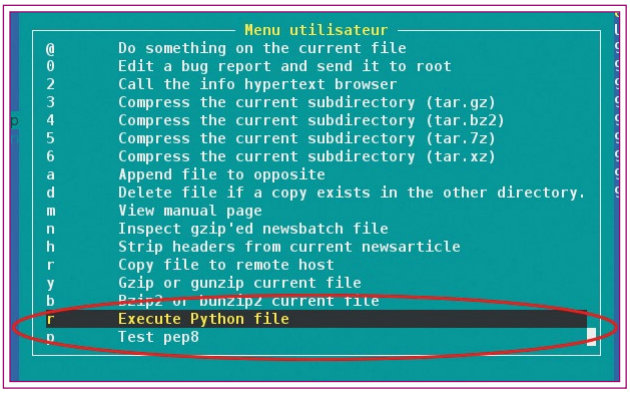


Fig. 8 : Le menu utilisateur personnalisé pour les fichiers Python.

Les entrées sont définies sur deux lignes :

```
<raccourci> Nom de l'entrée (affiché dans le menu)
   <commande> (%f désigne encore le fichier en
surbrillance)
```

Le résultat de ce petit menu est visible en figure 8. Pour rester homogène avec les autres entrées, vous devez utiliser des espaces et non des tabulations pour le décalage entre le raccourci et la description d'une entrée.

Vous trouverez plus d'informations sur ce menu dans l'aide : **<F1>** puis ***contents > Command Menu > Menu File Edit***.

NOTE

Vous pouvez créer des fichiers `.mc.menu` pour créer des menus utilisateurs qui ne se déclencheront que lorsque vous serez dans certains répertoires (ceux contenant les fameux `.mc.menu`). Pour ces fichiers, vous pouvez passer par le menu <F9> puis Commande > Édition du fichier de Menu et pour finir, sélectionnez local.

CONCLUSION

Pour beaucoup d'entre vous, cet outil n'est pas une grande découverte, mais pour ceux qui n'en ont jamais entendu parler ou qui n'ont tout simplement pas pris le temps de le tester, j'espère que cet article vous permettra d'utiliser Midnight Commander efficacement et rapidement. Comme avec Vim, il y a un petit effort à fournir au départ... mais on y gagne vraiment beaucoup (et même plus :-)) ! ■

NOTE

Après avoir utilisé de manière conjointe Midnight Commander et Nautilus pendant deux semaines, j'utilise maintenant exclusivement Midnight Commander depuis un mois et... je vais bien !

ASTUCE

Pour séparer vos entrées des entrées par défaut et les rendre plus visibles et lisibles (voir figure 9), vous pouvez placer une première règle qui insérera une démarcation :

```
=+ f \.py$
-       ------------------------------
```

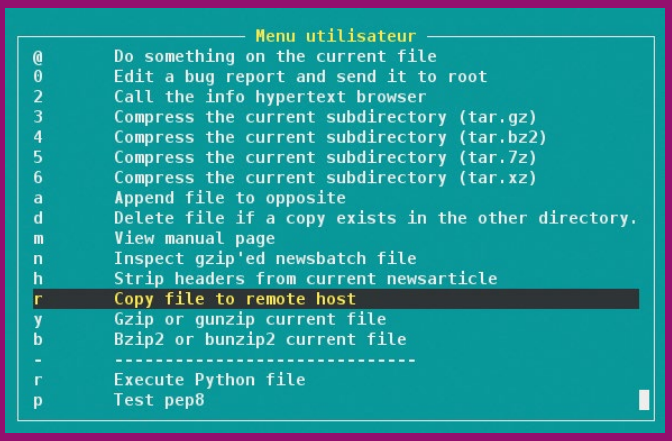
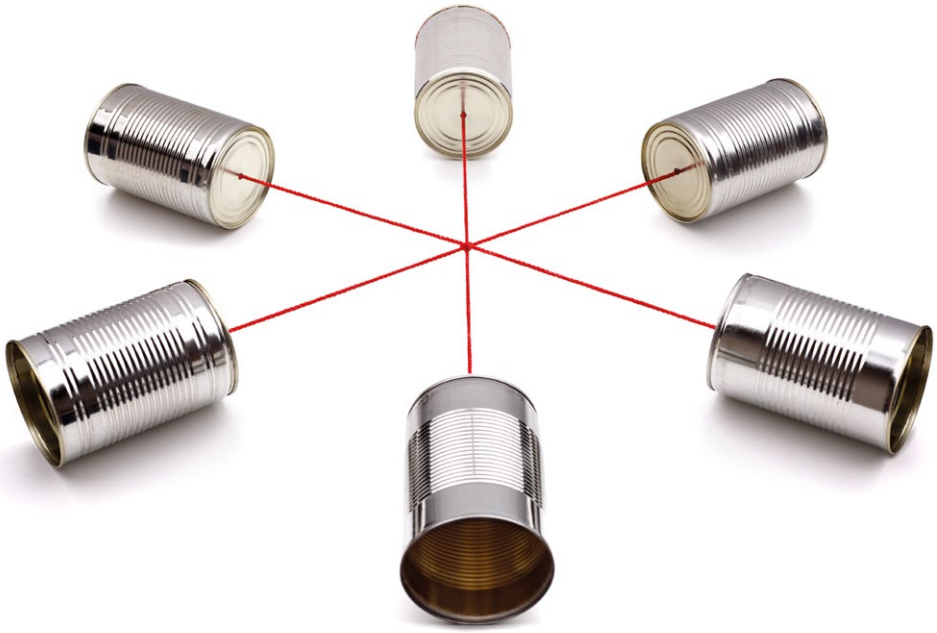


Fig. 9 : Ajout d'une ligne de démarcation dans un menu utilisateur.

CANOPEN
AVEC RASPBERRY PI

LAURENT DELMAS
[Ingénieur mécatronique, utilisateur GNU/Linux depuis 2001]

MOTS-CLÉS : CANOPEN, MULTIPLEXAGE, BUS CAN, RASPBERRY PI



Quel que soit le domaine d'application, automobile, industriel, médical ou aéronautique, l'utilisation du multiplexage est devenue une nécessité de par la complexité des systèmes. C'est alors que s'est standardisé le bus CAN (Control Area Network) qui n'est autre qu'une mise en œuvre de multiplexage. Nous allons voir comment implémenter le bus CAN avec un Raspberry Pi ainsi que la couche d'application CANOpen qui est de plus en plus utilisée par d'autres protocoles tels que l'OpenPowerLink.

Au début des années 2000, l'arrivée massive de l'électronique dans l'automobile a eu pour conséquence une augmentation des câbles avec les inconvénients que cela engendre (une augmentation du poids, un prix non négligeable, un encombrement important...). Une solution pour résoudre ces problèmes fut alors le multiplexage dont l'objectif est de faire passer l'ensemble des signaux via un même câble. C'est alors qu'apparaît le bus CAN (*Control Area Network*) développé principalement par l'entreprise **BOSCH**. Il trouve depuis son utilisation dans nombres de domaines : de l'automobile à l'aéronautique en passant par le médical et l'industrie et est devenu un standard incontournable.

D'autre part ces dernières années les nouvelles technologies, en particulier celles liées aux communications et au numérique, conduisent à une nouvelle révolution industrielle appelée Industrie 4.0 dite également « industrie intelligente ». S'ouvre alors à l'industrie traditionnelle une pléiade de possibilités : maintenance préventive, suivi temps réel, réalité augmentée, diagnostic à distance, etc.

Le bus CAN étant un standard des applications industrielles et afin que les systèmes actuels puissent bénéficier de cette révolution, il apparaît intéressant, voire indispensable, d'avoir une interface

qui permette de transformer les systèmes industriels traditionnels en systèmes connectés. Bien évidemment, il existe de telles interfaces dans le monde industriel. Nous allons, tout au long de cet article, voir comment réaliser ce type d'interface avec un **Raspberry Pi**.

Si vous effectuez une recherche rapide sur Internet à propos de l'implémentation du bus CAN avec le Raspberry Pi ou toutes autres cartes similaires (**Arduino**, **BeagleBone Black**, etc.), vous trouverez nombre d'articles plus ou moins récents et n'expliquant pas toutes les subtilités conduisant à un fonctionnement assuré. L'objectif de cet article est de synthétiser et d'expliciter toutes les étapes d'implémentation d'une interface CAN sur Raspberry Pi. Pour cela, nous présenterons et utiliserons le module **CAN SPI click 3.3V** de **Mikroelektronika** associé à un Raspberry Pi 3 modèle B **[1]**. Nous implémenterons enfin le protocole CANOpen.

1. RAPPEL À PROPOS DU RÉSEAU CAN

Le bus CAN est un protocole série, robuste, fiable qui s'accommode très bien avec des systèmes temps réels. Le protocole CAN comme tout protocole réseau utilise le modèle OSI. Toutefois, seules les deux premières couches du modèle OSI sont utilisées par le bus CAN, à savoir la couche physique et la couche liaison, respectivement au travers des normes ISO 11898-2 et ISO 11898-1 **[2]**.

Voici quelques caractéristiques principales du bus CAN :

- Transmission différentielle, ce qui lui confère une très bonne tenue aux perturbations ainsi qu'une immunité aux interférences électromagnétiques.
- Débit maximal pouvant atteindre 1Mbps sur une distance de 40m.
- Nombres de nœuds pouvant aller jusqu'à 30 sur un même bus. Ces nœuds peuvent être soit maîtres soit esclaves, le bus CAN fonctionnant en multi-maîtres.
- Mécanismes de détection et gestion des erreurs qui permettent d'obtenir une probabilité totale résiduelle d'erreur de 4,7. 10-11.
- Synchronisation des horloges de chaque nœud.
- Temps réel : pour garantir l'aspect temps réel, le bus CAN définit de façon précise le temps nominal de chaque bit (*Nominal Bit Time*) ainsi que le nombre de bits par seconde émis sans resynchronisation (*Nominal Bit Rate*). Le temps nominal de chaque bit correspond à un nombre fini de « quantum de temps » (*Time Quantum*). Ce quantum de temps est défini à partir de la fréquence d'horloge utilisée. Chaque bit CAN est composé de quatre parties :

  → Segment de synchronisation (SYNC) : laps de temps pendant lequel la synchronisation entre les nœuds a lieu.

  → Segment de propagation : période qui permet la compensation des retards physiques dans le réseau.

  → Segment phase 1 et Segment phase 2 : ces deux segments permettent de compenser les erreurs de synchronisation.

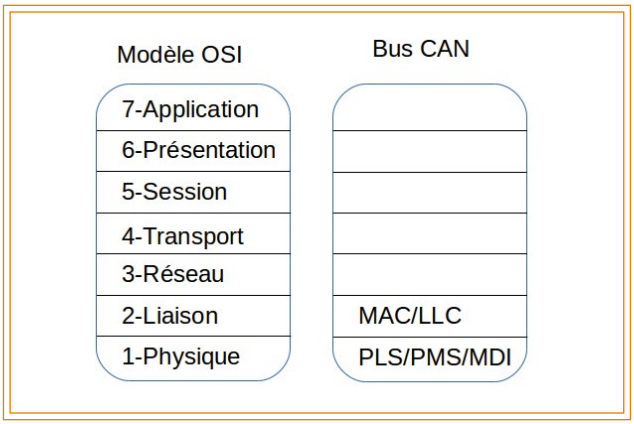


Fig. 1 : Modèle OSI et bus CAN.

Afin d'imager ces concepts, la Figure 2 représente les différents types de bits sur un même chronogramme. Pour le calcul détaillé du « quantum de temps » ainsi que le temps nominal de chaque bit, vous pouvez consulter la note d'application de NXP **[3]**.

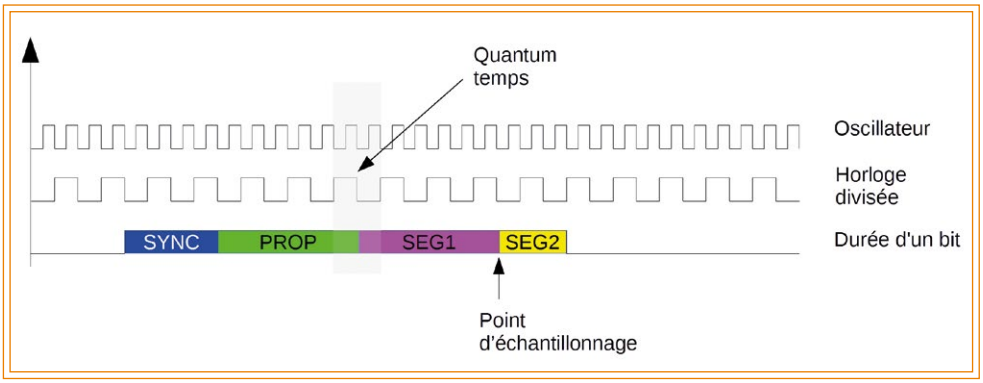


Fig. 2 : Composition d'un bit du bus CAN.

1.1 Topologie du bus CAN

La topologie du bus CAN est très simple puisque basée sur un protocole série. Le bus est obtenu à partir de deux fils torsadés. Chaque terminaison du bus est munie d'une résistance de 120Ω. De plus, afin de maintenir les caractéristiques du bus, chaque nœud doit être relié au plus proche du bus. Par exemple, la distance maximale préconisée dans la norme ISO 11898 pour un bus de 1Mpbs est de 30cm **[2]**. La figure 3 montre la topologie typique d'un bus CAN.

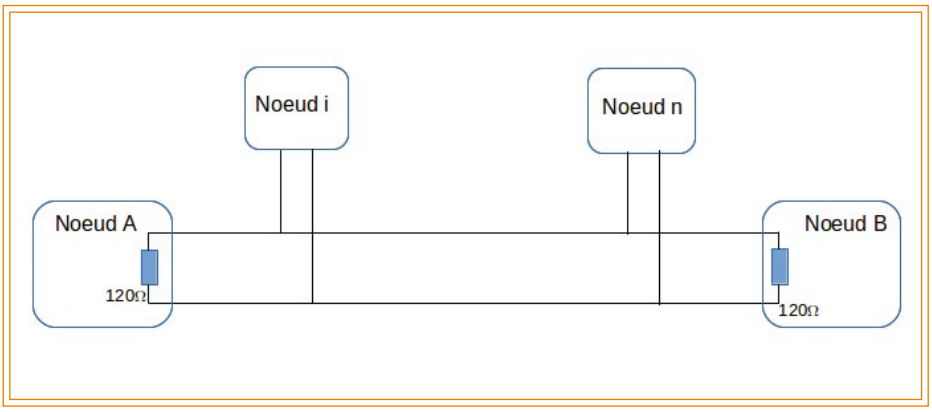


Fig. 3 : Topologie du bus CAN.

1.2. Architecture des nœuds CAN

Le débit du bus CAN est défini lors de l'initialisation du contrôleur CAN. Il dépend entre autres de la fréquence d'horloge utilisée. Pour cet article, j'ai choisi d'utiliser la carte CAN SPI click 3.3V de Mikroelektronika **[1]**. Le contrôleur CAN présent étant le **MCP2515** de **Microchip [4]**. Ce contrôleur est présent dans nombre d'autres cartes telles que la carte **PICAN2 [5]**. Le module CAN SPI utilise un quartz de 10MHz contrairement à la carte PICAN2 qui emploie un quartz de 16MHz. Comme nous le verrons un peu plus loin dans cet article, cela a une incidence importante pour la mise en place du driver et la détermination du *bit timing*.

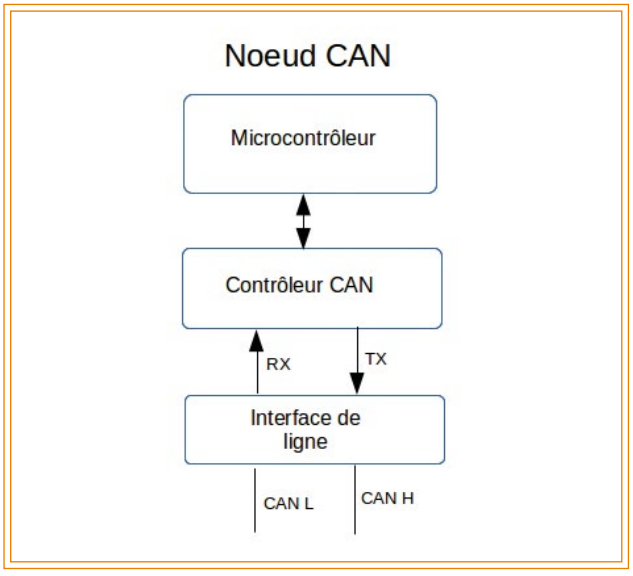


Fig. 4 : Architecture d'un nœud CAN.

1.3. Format des trames

Quelle que soit la couche d'application utilisée (CANOpen, KWP2000, UDS...), le format des trames CAN est toujours identique. Il est défini dans les spécifications CAN 2.0A et CAN2.0B respectivement avec des identifiants de 11bits et 29bits **[6]**. La figure 5 représente la composition typique d'une trame CAN.

Chaque trame commence avec un bit de début de trame (**SOF**) et se termine avec un bit de terminaison (**EOF**). Après le bit de début de trame vient l'identificateur qui est unique et spécifie l'information contenue dans la trame, auquel est associé un bit **RTR** précisant le type de trame (*Remote Transmission Request*). Un identifiant de 11bits permet 2^{11} = 2048 messages différents alors qu'un identifiant de 29bits autorise 2^{29} messages possibles. Dans le champ de contrôle est défini entre autres la longueur des données (**DLC**) contenues dans le champ **DATA**. La longueur maximale des données (**DATA**) est de 8 octets. Vient ensuite une somme de contrôle (**CRC**) permettant de garantir la validité du message transmis et le champ d'acquittement émis par la station réceptrice de la trame.

Fig. 5 : Format des trames CAN.

2. CARTE ÉLECTRONIQUE

Dans un numéro précédent de *GNU/Linux Magazine* a été présenté le protocole **MikroBUS** pour réaliser entre autres une carte relais commandée en Bluetooth **[7]**. Pour des raisons de facilité, nous allons utiliser dans cet article une carte qui respecte ce même protocole pour implémenter le module CAN. Il existe deux versions de cartes MikroBUS CAN, une

version 5V et une version 3.3V. La différence porte uniquement sur l'alimentation. Le Raspberry Pi ayant les deux tensions de disponibles sur le connecteur 2x20 broches, peut importe la version choisie. La carte est constituée d'un contrôleur CAN de Microchip MCP2515, commandée via une interface SPI et du composant **SN65HVD230** comme interface de ligne. Le schéma électrique de connexion du module CAN SPI au Raspberry Pi est représenté sur la figure 6. Les composants permettant de réaliser le montage sont disponibles chez **Radiospares**.

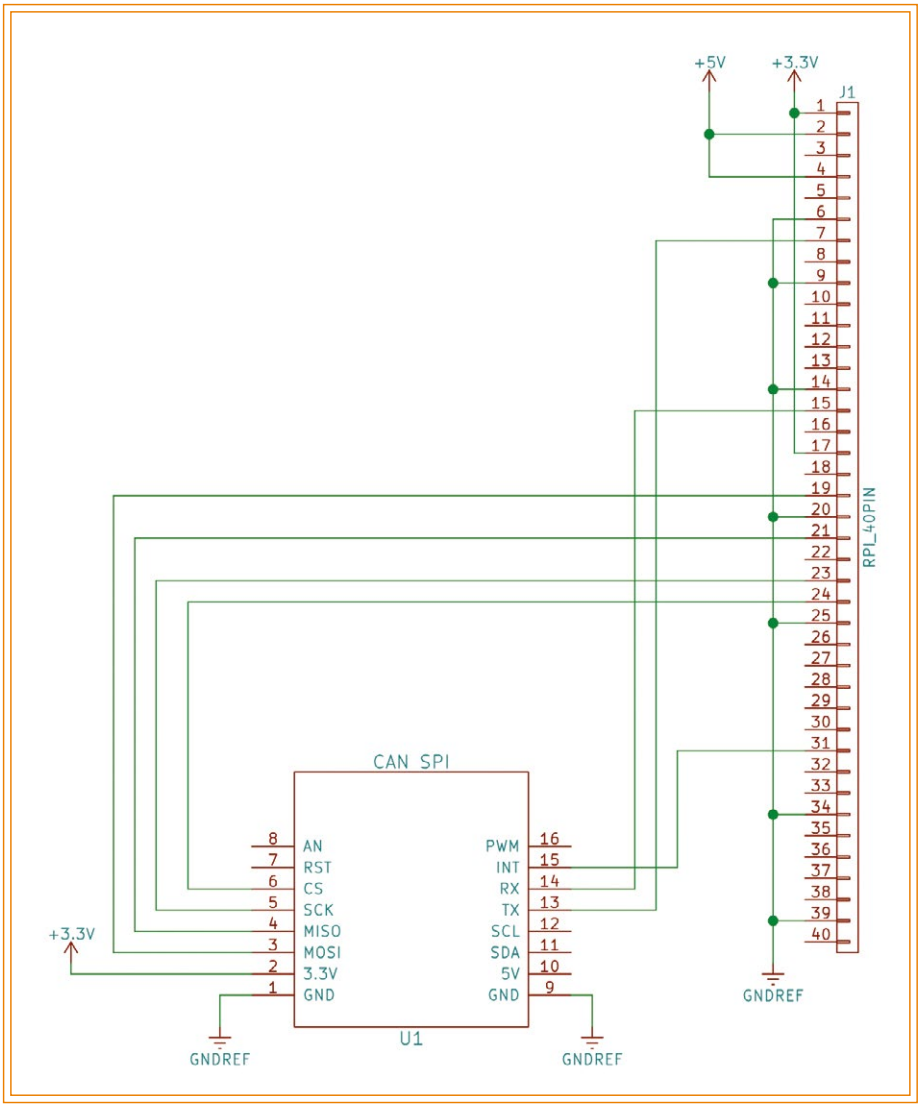


Fig. 6 : Schéma électrique du module bus CAN SPI relié au Raspberry PI.

3. ENVIRONNEMENT DE DÉVELOPPEMENT

Maintenant que l'aspect matériel est terminé, passons à la partie logicielle de la mise en place du bus CAN sur Raspberry Pi. Le contrôleur CAN (MCP2515) est accessible via le bus SPI, nous avons donc deux possibilités pour accéder et configurer le bus CAN : soit en écrivant un programme spécifique en utilisant le driver **SPIdev**, soit à partir d'un driver dédié pour le contrôleur CAN. Cependant cette dernière solution nécessite d'écrire un driver par implémentation du contrôleur CAN ayant une horloge différente. L'arrivée du *Device Tree* dans le noyau a permis de pallier ce type d'inconvénient. Depuis la version 4.x du noyau est disponible via le *Device Tree* ainsi que les *overlays*, le driver MCP2515 pour le contrôleur CAN **[8]**. Il suffit donc de spécifier les paramètres voulus au démarrage du noyau. Nous y reviendrons par la suite.

Pour ceux qui n'auraient pas la dernière version du noyau, nous allons tout d'abord télécharger et compiler le noyau avec les modules CAN puis définir les paramètres de démarrage associés. Nous partons toutefois du principe que vous avez à disposition une carte mémoire SD avec une version de **Raspbian** fonctionnelle. Cette dernière a deux partitions, l'une VFAT sur laquelle se trouvent le noyau ainsi que les fichiers de configuration, les fichiers du *Device Tree* et le *firmware*. La seconde partition ext4 correspondant au système racine de la Raspbian.

Si vous n'avez pas de carte mémoire SD de ce type, vous pouvez en créer une à partir de l'image disponible sur le site officiel. Téléchargez puis décompressez l'archive.

```
$ wget https://downloads.raspberrypi.org/
raspbian/images/raspbian-2017-03-03/2017-03-02-
raspbian-jessie.zip
$ unzip 2017-03-02-raspbian-jessie.zip
```

Insérez ensuite une carte SD sur laquelle vous souhaitez installer Raspbian. Attention de vous assurer d'avoir sauvegardé vos fichiers avant de procéder à l'écriture de la carte mémoire SD, car toutes les données seront effacées. Utilisez la commande **`df -h`** pour trouver le nom de votre carte mémoire SD. Dans mon cas, il s'agit du fichier **`/dev/sdd`** :

```
$ df -h
udev            2,9G       0  2,9G   0% /dev
tmpfs           588M    9,1M  579M   2% /run
/dev/sda2       193G     29G  155G  16% /
tmpfs           2,9G    632K  2,9G   1% /dev/shm
tmpfs           5,0M    4,0K  5,0M   1% /run/lock
tmpfs           2,9G       0  2,9G   0% /sys/fs/cgroup
/dev/sda5       357G    194G  145G  58% /home
cgmfs           100K       0  100K   0% /run/cgmanager/fs
/dev/sdd1       30G     27G  3,9G  88% /media/laurent/1D1C-11F7
```

Pour terminer, copiez l'image sur votre carte mémoire SD avec la commande **dd** :

```
$ sudo dd bs=4M if=2017-03-02-raspbian-jessie.img
of=/dev/sdd
```

3.1. Création de la chaîne de compilation

Pour compiler le noyau, il nous faut une chaîne de compilation croisée. Une fois n'est pas coutume, nous allons utiliser le générateur **Crosstool-NG** à l'instar de **Buildroot** pour créer notre propre chaîne de compilation **[9][10]**.

Tout d'abord, installons les paquetages ainsi que les dépendances nécessaires à la compilation de la chaîne de compilation :

```
$ sudo apt-get install bison cvs flex gperf
texinfo automake libtool libtool-bin gawk help2man
libncurses5-dev bc
```

Créons ensuite un répertoire de travail dans lequel nous allons créer notre chaîne de compilation :

```
$ mkdir -p raspberrypi/toolchain
$ cd raspberrypi/toolchain
```

Maintenant nous pouvons télécharger la dernière version de Crosstool-NG :

```
$ git clone https://github.com/crosstool-ng/crosstool-ng
Clonage dans 'crosstool-ng' ...
remote: Counting objects: 29870, done.
remote: Compressing objects: 100% (108/108), done.
remote: Total 29870 (delta 49), reused 0(delta 0), pack-
reused 29760.
Réception d'objects: 100% (29870/29870), 12,06 MiB |
806,00 KiB/s, fait.
Résolution des deltas: 100% (19679/19679), fait.
Vérification de la connectivité... fait.
```

Plaçons-nous dans le répertoire fraîchement créé puis faisons un **bootstrap** avant de procéder à la configuration de Crosstool-NG :

```
$ cd crosstool-ng
$./bootstrap
Running autoconf...
Done. You may now run:
    ./configure
```

Créons également un répertoire dans lequel sera installé Crosstool-NG puis compilons et installons ce dernier :

```
$ mkdir crosstool-ng
$ cd toolchain/crosstool-ng
$ ./configure --prefix=/home/laurent/
raspberrypi/crosstool-ng
$ make
$ makeinstall
```

Ajoutons à notre **PATH** le répertoire **bin** où est installé Crosstool-NG :

```
$ export PATH= "${PAHT}:/home/laurent/
raspberrypi/crosstool-ng/bin
```

Nous pouvons maintenant configurer notre chaîne de compilation croisée. Avant cela, créons de nouveau un répertoire de travail temporaire nommé **workspace** dans lequel sera configurée et construite la chaîne de compilation croisée elle-même. Démarrons ensuite la configuration à proprement parler, une interface graphique en mode texte similaire à celle du noyau est utilisée.

```
$ mkdir workspace
$ cd workspace
$ ct-ng menuconfig
```

Dans l'onglet ***Paths and misc options***, définissons le répertoire où seront téléchargées les sources des programmes et bibliothèques utilisées pour la construction de la chaîne de compilation. Activons également les caractéristiques expérimentales et précisions l'emplacement d'installation final de la chaîne de compilation.

Paths and misc options
→ ***Local tarballs directory (${home}/raspberrypi/src)***
→ ***Enable "Try features marked as EXPERIMENTAL"***
→ ***définir "Prefix directory" comme emplacement d'installation finale de la toochain (i.e. /home/user/x-tool/$[CT_TARGET}) soit ${HOME}/x-tools/${CT_TARGET}***

Dans l'onglet ***Target Options***, nous allons définir la cible pour laquelle va être construite la chaîne de compilation puis dans ***Operating system*** le système d'exploitation : en l'occurrence, il s'agit de Linux et du format binaire ELF.

ACTUELLEMENT DISPONIBLE

MISC HORS-SÉRIE N°15 !

NE LE MANQUEZ PAS

CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :

http://www.ed-diamond.com

Target Options
→ "Target architecture" : "ARM"
→ "Endianness" : "Little endian"
→ "Bitness" : "32-bit"
(→ "Architecture level" : "armv6zk")NO used
→ Use EABI

Operating system
→ "Target OS" : "Linux"

Binary utils
→ "Binary format" : "ELF"
→ Show Linaro version
→ "binutils version" : "2.28"

De même pour les librairies et compilateurs :

C-compiler
→ show linaro version
→ "gcc version" : "linaro -6.3-2017.02)"
(→ "gcc extra config" : "--with-float=hard")NO
→ "link libstdc++ statically into gcc binary"
→ additional supported languages:
→ C++
C-library
→ Show Linaro-version
→ "C library" : "glibc"
→ "glibc version" : "2.25"

Les outils résultants créés lors de la construction auront pour nom le format suivant **archi-cpu-system-libC**, soit dans notre cas : **arm-unknown-linux-gnueabi**. Si vous souhaitez changer le nom pour le cpu (**unknown**) par un autre, **rpi** par exemple, il faut le définir dans l'onglet ***Toolchain options*** :

Toolchain options
→ Tuple's vendor string : rpi

Nous pouvons enregistrer notre configuration dans le fichier par défaut **.config** et quitter la fenêtre de configuration.

Lançons ensuite la construction de la chaîne de compilation avec la commande suivante. Cela va prendre un certain temps ou temps certain pour la construction des différents outils et bibliothèques :

```
$ ct-ng build
```

Lorsque la construction est terminée, il ne nous reste plus qu'à ajouter le répertoire d'installation de la chaîne de compilation croisée dans notre **PATH**.

```
$ export PATH=${PATH}:/home/laurent/x-tool/arm-unknown-linux-gnueabi/bin
```

Afin que l'ajout du répertoire persiste au prochain redémarrage, ajoutez l'exportation dans le fichier **.bashrc** de votre répertoire courant.

Vérifions que notre chaîne de compilation croisée est bien disponible :

```
$ arm-unknown-linux-gnueabi-gcc -v
Using built-in specs.
COLLECT_GCC=arm-unknown-linux-gnueabi-gcc
COLLECT_LTO_WRAPPER=/home/laurent/x-tools/arm-unknown-linux-gnueabi/bin/../libexec/gcc/arm-unknown-linux-gnueabi/6.3.1/lto-wrapper
Target: arm-unknown-linux-gnueabi
Configured with: /home/fr46c2/raspberrypi/workspace/.build/src/gcc-linaro-6.3-2017.02/configure --build=x86_64-build_pc-linux-gnu --host=x86_64-build_pc-linux-gnu --target=arm-unknown-linux-gnueabi --prefix=/home/fr46c2/x-tools/arm-unknown-linux-gnueabi --with-sysroot=/home/fr46c2/x-tools/arm-unknown-linux-gnueabi/arm-unknown-linux-gnueabi/sysroot --enable-languages=c,c++ --with-arch=armv7-a --with-pkgversion='crosstool-NG crosstool-ng-1.23.0-rc2' --enable-__cxa_atexit --disable-libmudflap --disable-libgomp --disable-libssp --disable-libquadmath --disable-libquadmath-support --disable-libsanitizer --disable-libmpx --with-gmp=/home/fr46c2/raspberrypi/workspace/.build/arm-unknown-linux-gnueabi/buildtools --with-mpfr=/home/fr46c2/raspberrypi/workspace/.build/arm-unknown-linux-gnueabi/buildtools --with-mpc=/home/fr46c2/raspberrypi/workspace/.build/arm-unknown-linux-gnueabi/buildtools --with-isl=/home/fr46c2/raspberrypi/workspace/.build/arm-unknown-linux-gnueabi/buildtools --enable-lto --with-host-libstdcxx='-static-libgcc -Wl,-Bstatic,-lstdc++,-Bdynamic -lm' --enable-threads=posix --enable-target-optspace --disable-plugin --disable-nls --disable-multilib --with-local-prefix=/home/fr46c2/x-tools/arm-unknown-linux-gnueabi/arm-unknown-linux-gnueabi/sysroot --enable-long-long
Thread model: posix
gcc version 6.3.1 20170109 (crosstool-NG crosstool-ng-1.23.0-rc2)
laurent@ASUS:~/raspberrypi/workspace/$
```

Nous retrouvons bien la version de **gcc** que nous avions définie lors de la configuration avec Crosstool-NG ainsi qu'entre parenthèses les outils avec lesquels a été construite la chaîne de compilation croisée. Afin de libérer de l'espace et de ne pas encombrer notre disque dur, nous pouvons effacer le répertoire de travail temporaire **workspace**.

3.2. Compilation du noyau

Maintenant que nous disposons d'une chaîne de compilation croisée pour notre Raspberry Pi, nous allons procéder à la compilation du noyau avec les modules pour le CAN. Commençons par créer un répertoire de travail puis télécharger les sources du noyau :

```
$ mkdir kernel
$ cd kernel
$ git clone --depth=1 https://github.com/
raspberrypi/linux
Clonage dans 'linux'...
remote: Counting objects: 60207, done.
remote: Compressing objects: 100% (56585/56585),
done.
remote: Total 60207 (delta 5247), reused 16388
(delta 2726), pack-reused 0
Réception d'objets: 100% (60207/60207), 158.86
MiB | 60.00 KiB/s, fait.
Résolution des deltas: 100% (5247/5247), fait.
Vérification de la connectivité... fait.
Extraction des fichiers: 100% (56709/56709),
fait.
$ cd linux
```

Procédons à quelques préparatifs avant de configurer notre noyau. Nettoyons les sources, puis définissons une variable **KERNEL** :

```
$ make mrproper
$ KERNEL=kernel7
```

Configurons le noyau en prenant soin de définir l'architecture pour laquelle nous voulons construire le noyau ainsi que la chaîne de compilation croisée.

```
$ make ARCH=arm CROSS_COMPILE=arm-
unknown-linux-gnueabi- bcm2709_defconfig
$ make ARCH=arm CROSS_COMPILE=arm-
unknown-linux-gnueabi- menuconfig
```

Apparaît alors la traditionnelle fenêtre de configuration du noyau. Le fait d'avoir appelé la configuration **bcm2709_defconfig** a permis de définir la configuration par défaut pour un Raspberry Pi. Nous allons maintenant apporter quelques modifications supplémentaires afin que les drivers du contrôleur CAN (MCP2525) fonctionnent. Ajouter les options listées ci-dessous :

```
[*]Networking support --->
  <M> CAN bus subsystem support --->
      CAN Device drivers --->
     <M> Platform CAN drivers with
Netlink support
  [*] CAN bit-timing calculation
         CAN SPI interfaces --->
          <M> Microchip MCP251x SPI CAN controllers
  [*] CAN devices debugging messages

Device Drivers --->
  [*]SPI support --->
      <M> BCM2835 SPI controller
   <M> BCM2835 SPI auxiliary controller
   <M> User mode SPI device driver support
  -*-GPIO support --->
    [*] /sys/class/gpio/... (sysfs interface)
```

Effectuons ensuite une sauvegarde de la configuration dans le fichier par défaut **.config**. Puis nous pouvons sortir de la fenêtre de configuration pour débuter la compilation du noyau. Nous précisons que nous voulons compiler le noyau, les modules ainsi que les fichiers *device tree*. Pour accélérer la compilation, ajoutons le nombre de cœurs que nous souhaitons employer :

```
$ make ARCH=arm CROSS_COMPILE=arm-unknown-linux-
gnueabi- zImage modules dtbs -j4
```

3.3. Installation et configuration des modules CAN

La compilation du noyau terminée, nous pouvons donc installer ce nouveau noyau ainsi que mettre en place les modules pour le bon fonctionnement du bus CAN. Comme évoqué plus tôt dans cet article, nous partons d'une carte mémoire SD avec une version de Raspbian qui n'est pas nécessairement la dernière version. Dans le cas où le montage des systèmes de fichiers n'est pas automatique, la première tâche consiste à identifier puis monter les deux partitions pour pouvoir copier le nouveau noyau :

```
$ lsblk
NAME   MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
sda      8:0    0 30,5GG  0 disk
├─sda1   8:1    0   1,9G  0 part [SWAP]
├─sda2   8:2    0     1K  0 part
├─sda5   8:5    0  28,6G  0 part /
sdb      8:16   0  14,9G  0 disk
├─sdb1   8:17   0    63M  0 part
├─sdb2   8:18   0  14,8G  0 part
sr0     11:0    1  1024M  0 rom
$ mkdir /media/fat32
$ mkdir /media/ext4
$ sudo mount /dev/sdb1 /media/fat32
$ sudo mount /dev/sdb2 /media/ext4
```

Installons les nouveaux modules sur la carte mémoire SD :

```
$ sudo make ARCH=arm CROSS_COMPILATION=arm-
unknown-linux-gnueabi- INSTALL_MOD_PATH=media/
ext4 modules_install
```

Effectuons une sauvegarde de notre noyau actuel :

```
$ sudo cp media/fat32/$KERNEL.img media/
fat32/$KERNEL-backup.img
```

Finalement transférons le noyau fraîchement compilé ainsi que les fichiers *device tree* sur la carte :

```
$ sudo cp arch/arm/boot/zImage média/fat32/$KERNEL.img
$ sudo cp arch/arm/boot/dts/*.dtb media/fat32
$ sudo cp arch/arm/boot/dts/overlays/*.dtb* media/fat32/
overlays/
$ sudo cp arch/arm/boot/dts/overlays/README media/fat32/
overlays/
```

Il nous reste à activer le bus SPI et à configurer le module CAN via les paramètres passés au noyau lors du démarrage. La liste des paramètres est disponible dans le fichier README dans le répertoire **overlays** de la partition VFAT. Pour cela, nous allons éditer le fichier **config.txt** qui se trouve lui aussi sur la première partition de notre carte mémoire SD. Nous y ajoutons les lignes suivantes, puis redémarrons le Raspberry Pi :

```
# Activation du bus SPI
dtparam=spi=on
#Activation du controleur CAN MCP2515
# fréquence d'horloge de 10MHz
dtparam=mcp2515-can0, oscillator=10000000, interrupt=25
dtparam=spi0-hw-cs
```

Nous pouvons vérifier que l'interface est bien disponible soit avec la commande **ifconfig**, soit avec **dmesg** :

```
pi@raspberrypi:~$ ifconfig can
can0      Link encap:UNSPEC  HWaddr 00-00-00-00-
00-00-00-00-00-00-00-00-00-00-00-00
          NOARP  MTU:16  Metric:1
          RX packets:0 errors:0 dropped:0
overruns:0 frame:0
          TX packets:0 errors:0 dropped:0
overruns:0 carrier:0
          collisions:0 lg file transmission:10
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
pi@raspberrypi:~$ dmesg |grep -i can
[    4.651250] CAN device driver interface
[  693.208735] can: controller area network core
(rev 20120528 abi 9)
[  693.218134] can: raw protocol (rev 20120528)
[  704.026608] IPv6: ADDRCONF(NETDEV_CHANGE):
can0: link becomes ready
```

4. MISE EN PLACE ET TEST DU CAN AVEC WIRESHARK

Maintenant que les drivers SPI et le contrôleur CAN fonctionnent, nous allons pouvoir tester et utiliser le bus CAN. Le noyau Linux supporte le CAN via le *framework* **SocketCAN [11]**. Son implémentation consiste à ajouter un nouveau protocole à l'API réseau, PF_CAN à côté des autres protocoles comme PF_INET pour le protocole Internet. La communication CAN devient donc similaire à une communication classique qui utilise le protocole Internet via l'utilisation des sockets.

SocketCAN supporte différents protocoles tels que RAW pour l'envoi et la réception de trames CAN brutes, SAE J1939... Nombre de contrôleurs CAN fonctionnent avec cette API, dont le MCP2515 de Microchip que nous avons mis en place dans les sections précédentes.

Avant de pouvoir utiliser le bus CAN et en particulier le SocketCAN, il nous faut déclarer et associer l'interface CAN comme une interface réseau en lui précisant le *bitrate* voulu pour le bus CAN :

```
pi@raspberrypi:~$ sudo ip link set can0
up type can bitrate 500000
```

Pour modifier le *bitrate* ou autres paramètres du bus CAN, nous devons au préalable désactiver l'interface :

```
pi@raspberrypi:~$ sudo ip link set can0 down
```

Afin de réaliser nos premiers essais, soit vous avez un bus CAN à disposition (celui de votre voiture par exemple via la prise OBD II) , soit vous réalisez un bus CAN avec deux résistances de terminaison de 120 Ohms comme nous l'avons vu plus tôt dans cet article (Figure 3) sur lequel vous pouvez soit relier deux modules CAN via un ou deux Raspberry Pi. C'est cette dernière option que nous retiendrons. Nous avons deux Raspberry Pi identiques à celui que nous avons préparé.

Maintenant nous allons télécharger, compiler puis installer la boîte à outils **can-utils** sur nos deux Raspberry Pi **[12]** :

```
pi@raspberrypi:~$ cd raspberrypi
pi@raspberrypi:~/raspberrypi/$ git clone
https://github.com/linux-can/can-utils.git
pi@raspberrypi:~/raspberrypi/$  cd can-utils
pi@raspberrypi:~/raspberrypi/can-utils/$
./autogen.sh
pi@raspberrypi:~/raspberrypi/can-utils/$
./configure
pi@raspberrypi:~/raspberrypi/can-utils/$  make
pi@raspberrypi:~/raspberrypi/can-utils/$  sudo
make install
```

Cette boîte à outils fournit différents outils pour capturer et envoyer des messages sur le bus CAN. Par exemple, pour envoyer une trame ayant pour identifiant **183**, de longueur **8** octets et l'information **A0 13 00 00 00 00 00 00**, nous saisons la commande suivante :

```
pi@raspberrypi:~/raspberrypi/can-utils/$
cansend can0 183#A013000000000000
```

La capture de trame s'effectue avec la fonction **candump** suivie de l'interface **can0** :

```
pi@raspberrypi:~/raspberrypi/can-utils/$
candump can0
  can0  080   [0]
  can0  183   [8]  A0 13 00 00 00 00 00 00
  can0  314   [8]  12 24 48 36 11 AB CE FD
  can0  080   [0]
  can0  183   [8]  A0 13 00 00 00 00 00 00
  can0  314   [8]  12 24 48 36 11 AB CE FD
  can0  080   [0]
  can0  183   [8]  A0 13 00 00 00 00 00 00
  can0  314   [8]  12 24 48 36 11 AB CE FD
  can0  080   [0]
  can0  183   [8]  A0 13 00 00 00 00 00 00
  can0  314   [8]  12 24 48 36 11 AB CE FD
```

L'interface CAN étant reconnue et fonctionnant de façon similaire à une interface réseau, les outils habituels comme **Wireshark** fonctionnent. Si Wireshark n'est pas encore installé sur votre Raspberry Pi, il est temps de le faire. Ensuite, démarrez Wireshark avec les droits administrateur :

```
pi@raspberrypi:~/raspberrypi/can-utils/$ sudo
apt-get install wireshark wireshark-qt
pi@raspberrypi:~/raspberrypi/can-utils/$ sudo
wireshark-qt
```

S'ouvre alors la fenêtre principale de Wireshark où apparaissent les différentes interfaces disponibles. Nous retrouvons notre interface CAN dans la liste. Faisons un double clic dessus pour ouvrir ladite interface. Nous voyons alors défiler les diverses trames CAN disponibles sur le bus CAN (voir figure 7).

Le décodage des trames dépend de la couche applicative implémentée dans le système. Cela peut très bien être la couche applicative CANOpen très utilisée dans le milieu industriel, le protocole de diagnostic utilisé dans nombres de voitures KWP ou à venir UDS ou bien alors une couche propriétaire.

5. IMPLÉMENTATION DU CAN DANS VOS APPLICATIONS

Jusqu'à présent nous avons vu comment installer un contrôleur CAN et utiliser Wireshark ou la boîte à outils can-utils pour lire et envoyer des trames. Il est temps de voir un exemple d'implémentation d'un programme ayant la particularité d'utiliser le bus CAN. Par exemple à partir d'une commande CAN reçue, envoyer une autre trame CAN spécifique :

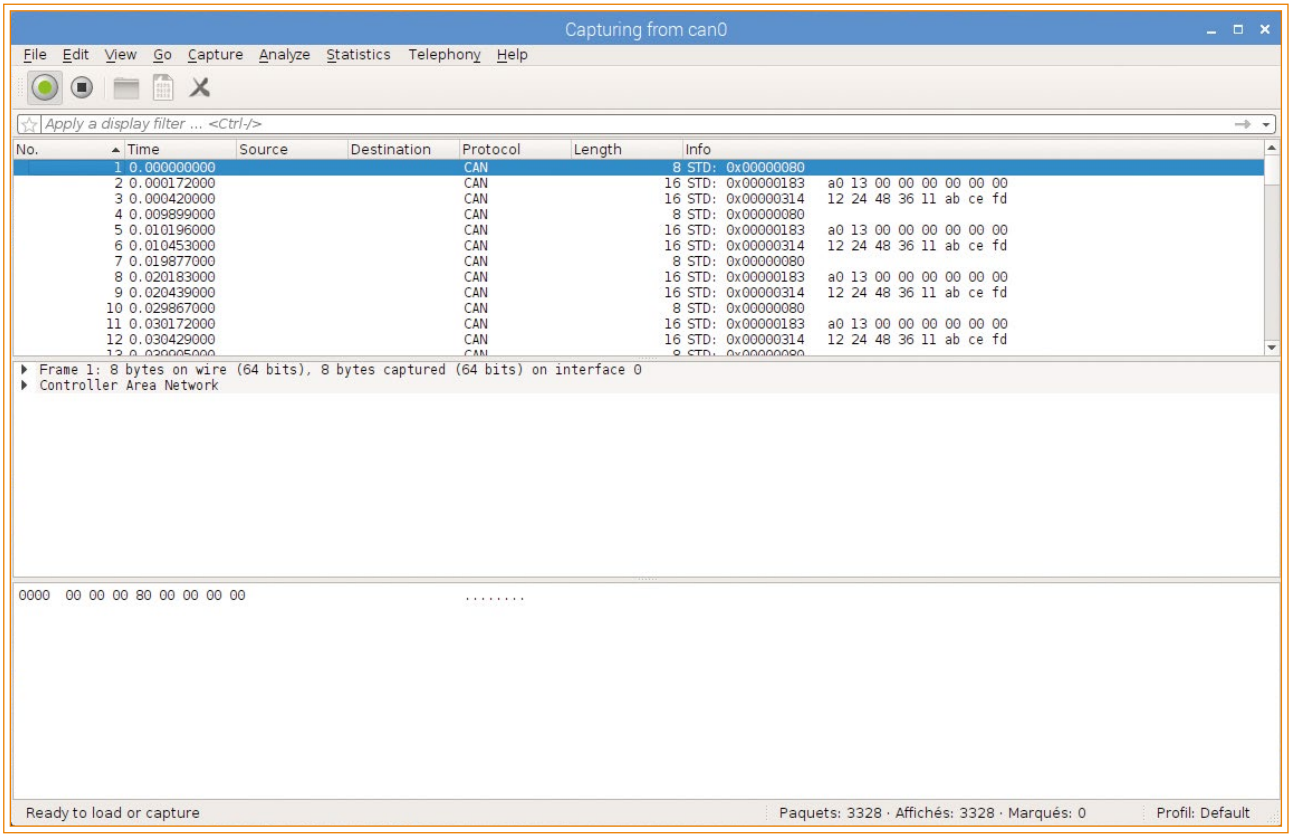


Fig. 7 : Capture des trames CAN avec Wireshark.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#include <net/if.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>

#include <linux/can.h>
#include <linux/can/raw.h>

int main(int argc, char** argv)
{
 int s;
 int nbytes;
 int cpt=0;
 struct sockaddr_can addr;
 struct can_frame frameA,frameB;
 struct ifreq ifr;

 if(argc!=2)
 {
  fprintf(stderr,"Usage : %s <device>\n",argv[0]);
  return -1;
 }

 if ((s=socket(PF_CAN,SOCK_RAW,CAN_RAW))<0)
 {
  perror("Erreur lors de l ouverture  du socket");
  return -1;
 }


 //différence entre strcpy vs strncpy
 //strcpy risque buffer overflow
 strcpy(ifr.ifr_name,argv[1]);
 ioctl(s,SIOCGIFINDEX,&ifr);

 addr.can_family=AF_CAN;
 addr.can_ifindex=ifr.ifr_ifindex;

 if(bind(s,(struct sockaddr *)&addr,sizeof(addr))<0)
 {
  perror("Erreur lors du binding");
  return -1;
 }

 //attend la reception de 100 trames CAN
 while(cpt<100)
 {
  //lecture puis affichage des trames CAN
  //nbytes=recv(s,&frameA,sizeof(struct can_frame),0);
  nbytes=read(s,&frameA,sizeof(struct can_frame));
  printf("%04x\t %d\n",frameA.can_id,frameA.can_dlc);
  printf("%x\n",frameA.can_id&CAN_SFF_MASK);
  int i;
  for(i=0;i<frameA.can_dlc;i++)
  {
   printf("%02x ",frameA.data[i]);
  }
  //si on recoit une trame ayant
l'identifiant 0x080
  if((frameA.can_id&CAN_SFF_MASK)==0x80)
  {
   // on envoie une seconde trame
   frameB.can_id=0x1F1;
   frameB.can_dlc=2;
   frameB.data[0]=0xFF;
   frameB.data[1]=0x15;
   nbytes=write(s,&frameB,sizeof(struct
can_frame));
  }

  cpt++;
 }

 return 0;
}
```

Nous compilons les quelques lignes de code de façon traditionnelle et pouvons lancer notre programme :

```
pi@raspberrypi:~/raspberrypi/can-utils/$
gcc CANbus.c -o CANbus
```

6. CANOPEN

Nous pouvons donc maintenant présenter la couche d'application **CANOpen**. La couche d'application a pour but de formaliser les échanges sur le bus CAN. La couche CANOpen quant à elle, spécifie les différents objets de communication, de services et la manière dont ils sont déclenchés. CANOpen définit quatre fonctions :

- la première fonction définie les échanges temps réels appelés communément **PDO** (*Process Data Object*), les objets sont envoyés soient de façon synchrone, soit sur un événement ou alors périodiquement ;
- la seconde fonction nommée **SDO** (*Service Data Object*) utilise le principe de type requêtes-réponses pour les échanges de données à la demande. Ce mode est utilisé principalement pour le paramétrage et la configuration des nœuds ;
- une troisième fonction est dédiée à l'administration du bus : démarrage du bus, surveillance... ;
- une quatrième fonction avec des objets prédéfinis : **SYNC**, **TIMESTAMPS**, **EMERGENCY**.

Chaque nœud après la mise sous tension ou bien une remise à zéro suivie de son initialisation passe en mode

PRE-OPERATIONNEL, c'est-à-dire en attente d'un ordre provenant du nœud maître. Le nœud fait connaître son état par l'envoi d'une trame spécifique appelée **HEARTBEAT**.

De plus, CANOpen standardise et prédéfinit les objets sur le bus CAN. Certes, à chaque nœud est attribué un identifiant unique (ID du nœud) par le concepteur du système correspondant à un nombre. Plus ce dernier est petit, plus le nœud est prioritaire. Le tableau ci-dessous synthétise les objets prédéfinis.

Le nœud maître, dès que sa phase d'initialisation est terminée, envoie la trame contenant l'objet module contrôle aux nœuds esclaves afin de leur demander de passer en mode OPERATIONNEL, ainsi que la trame SYNC pour synchroniser l'ensemble des messages du bus CAN. La figure 8 représente le diagramme d'état d'un nœud CAN.

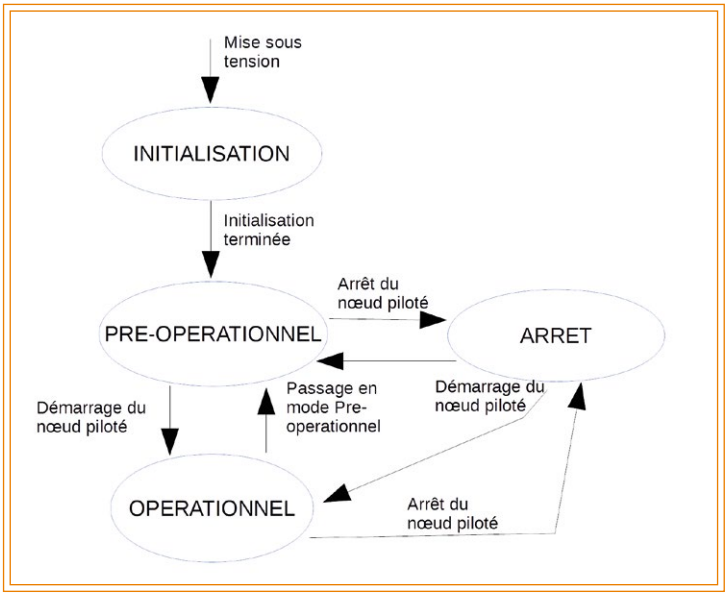


Fig. 8 : Diagramme d'état d'un nœud CAN.

Nous allons voir un exemple de mise en œuvre d'un esclave répondant au diagramme d'état présenté en Figure 8. Pour notre exemple, nous considérons que le nœud maître dès qu'il reçoit l'état PRE-OPERATIONNEL du nœud esclave envoie le module contrôle avec la commande de démarrage du nœud esclave. De plus, le nœud maître envoie l'objet de synchronisation SYNC toutes les 10ms.

La phase d'initialisation terminée, l'esclave envoie son HEARTBEAT avec son état et attend la demande de démarrage du nœud maître. Ensuite le nœud envoie de façon synchronisée le PDO 1 toutes les 20ms et le PDO 2 toutes les 100ms.

Objets	Identifiant (hexa)	Nœud esclave
NMT module contrôle	**000**	Reçoit uniquement
SYNC	**080**	Reçoit uniquement
EMERGENCY	**080** + ID du nœud	Transmet
PDO	**180** + ID du nœud	Transmission du PDO 1
	200 + ID du nœud	Réception du PDO 1
	280 + ID du nœud	Transmission du PDO 2
	300 + ID du nœud	Réception du PDO2
	380 + ID du nœud	Transmission du PDO 3
	400 + ID du nœud	Réception du PDO 3
	480 + ID du nœud	Transmission du PDO 4
	500 + ID du nœud	Réception du PDO 4
HEARTBEAT	**700** + ID du nœud	Transmet

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#include <net/if.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>

#include <linux/can.h>
#include <linux/can/raw.h>

//définition de ID du noeud
#define nodeID 0x05

//état du noeud
#define BOOTUP 0x0
#define OPERATIONAL 0x05
#define PREOPERATIONAL 0x7F
#define STOPPED 0x04

//CS module control
#define Start_Remote_Node 0x01
#define Stop_Remote_Node 0x02
#define Go_Pre_OP 0x80


int main(int argc, char** argv)
{
 int s;
 int nbytessend,nbytesread;
 int state=BOOTUP;
 struct sockaddr_can addr;
 struct can_frame frameA,
frameB,frameC,frameD;
```

```
struct ifreq ifr;

int syncPDO1=0;
int syncPDO2=0;

if(argc!=2)
{
 fprintf(stderr,"Usage : %s <device>\n",argv[0]);
 return -1;
}

//***********************
//initialisation du noeud

//initialisation du CAN
if ((s=socket(PF_CAN,SOCK_RAW,CAN_RAW))<0)
{
 perror("Erreur lors de l ouverture  du socket");
 return -1;
}

strcpy(ifr.ifr_name,argv[1]);
ioctl(s,SIOCGIFINDEX,&ifr);

addr.can_family=AF_CAN;
addr.can_ifindex=ifr.ifr_ifindex;

if(bind(s,(struct sockaddr *)&addr,sizeof(addr))<0)
{
 perror("Erreur lors du binding");
 return -1;
}
//fin initialisation du CAN
//autres initialisations possibles

//envoi du heartbeat
frameB.can_id=0x700+nodeID;
frameB.can_dlc=1;
frameB.data[0]=state;
nbytessend=write(s,&frameB,sizeof(struct can_frame));
printf("STATE:%04x\t%02x\n",frameB.can_id,frameB.
data[0]);
printf("Initialisation\n");

//fin de l'initialisation
//***********************
//passage au mode PRE OPERATIONNEL
state=PREOPERATIONAL;
printf("Fin initialisation\n");

while(1)
{
 //envoi du heartbeat
 frameB.can_id=0x700+nodeID;
 frameB.can_dlc=1;
 frameB.data[0]=state;
 nbytessend=write(s,&frameB,sizeof(struct
can_frame));
 printf("STATE:%04x\t%02x\n",frameB.can_id,frameB.
data[0]);
```

```
 //lecture des trames entrantes
 nbytesread=read(s,&frameA,sizeof(struct
can_frame));
 printf("%04x\t %d\t",frameA.can_id,frameA.
can_dlc);
 int i;
 for(i=0;i<frameA.can_dlc;i++)
 {
  printf("%02x ",frameA.data[i]);
 }
 printf("\n");

 //si trame de module controle
 if((frameA.can_id&CAN_SFF_MASK)==0x0)
 {
  //vérifie à qui est adressé la commande
  if ((frameA.data[1]==0x0)||(frameA.
data[1]==nodeID))
  {
   switch (frameA.data[0])
   {
     case Start_Remote_Node:
    state=OPERATIONAL;
    break;
     case Go_Pre_OP:
       state=PREOPERATIONAL;
       break;
     case Stop_Remote_Node:
    state=STOPPED;
    break;
   }

  }
 }
 //si trame SYNC
 if (((frameA.can_id&CAN_SFF_MASK)==0x80)&(
state==OPERATIONAL))
 {
  syncPDO1++;
  syncPDO2++;

  if((syncPDO1%2)==0)
  {
   //PDO1
   frameC.can_id=0x180+nodeID;
   frameC.can_dlc=4;
   //frameB.data=[0x00,0x11,0x22,0x33];
   frameC.data[0]=0x00;
   frameC.data[1]=0x11;
   frameC.data[2]=0x22;
   frameC.data[3]=0x33;
   nbytessend=write(s,&frameC,sizeof(stru
ct can_frame));
    printf("STATE:%04x\t%02x\n",frameC.can_
id,frameC.data[0]);
   syncPDO1=0;
  }

  if((syncPDO2%5)==0)
  {
   //PDO2
   frameD.can_id=0x280+nodeID;
   frameD.can_dlc=8;
```

```
    frameD.data[0]=0xA1;
    frameD.data[1]=0xB2;
    frameD.data[2]=0xC;
    frameD.data[3]=0xD4;
    frameD.data[4]=0xE5;
    frameD.data[5]=0xF6;
    frameD.data[6]=0xAA;
    frameD.data[7]=0xFF;
    //frameB.data=0xA1B2C3D4E5F6AAFF;
    nbytessend=write(s,&frameD,sizeof(st
ruct can_frame));
    printf("STATE:%04x\t%02x\n",frameD.
can_id,frameB.data[0]);
    syncPDO2=0;
   }
  }
 }


 return 0;
}
```

Dans la console suivante, nous constatons que notre nœud répond bien au module contrôle du nœud maître et que les trames sont envoyées de façon cyclique et synchronisée :

```
pi@raspberrypi:~/workspace $ ./CANOpen can0
STATE:0705 00
Initialisation
Fin initialisation
STATE:0705   7f
0000    2    01 00
STATE:0705   05
0080    0
STATE:0705   05
0080    0
STATE:0185   00
STATE:0705   05
0080    0
STATE:0705   05
0080    0
STATE:0185   00
STATE:0705   05
0080    0
STATE:0285   05
STATE:0705   05
0080    0
STATE:0185   00
STATE:0705   05
0080    0
STATE:0705   05
0000    2    01 00
STATE:0705   05
0080    0
STATE:0185   00
STATE:0705   05
0080    0
STATE:0705   05
0080    0
STATE:0185   00
STATE:0285   05
STATE:0705   05
i@raspberrypi:~/workspace $
```

CONCLUSION

Après un bref rappel sur le bus CAN, sa topologie et la constitution des nœuds CAN, nous avons mis en place un contrôleur CAN MCP2515 dans un Raspberry Pi 3. Pour cela, nous avons utilisé l'outil Crosstool-NG pour construire une chaîne de compilation croisée puis compiler les dernières sources du noyau qui utilisent le *Device Tree* facilitant ainsi la configuration du contrôleur CAN. Nous avons utilisé Wireshark pour faire la capture de trames CAN et avons également écrit un premier programme C permettant d'utiliser le bus CAN. Pour terminer, nous avons présenté et écrit un exemple d'utilisation de la couche applicative CANOpen. ■

RÉFÉRENCES

[1] CAN SPI click 3.3V : https://shop.mikroe.com/click/interface/can-spi-33v

[2] Normes ISO 11898, véhicules routiers - Gestionnaire de communication (CAN) : https://www.iso.org/standard/63648.html et https://www.iso.org/fr/standard/67244.html

[3] CAN bit timing requirements : http://www.nxp.com/assets/documents/data/en/application-notes/AN1798.pdf

[4] Spécifications du contrôleur CAN MCP2515 : http://ww1.microchip.com/downloads/en/DeviceDoc/21801d.pdf

[5] PiCAN2 carte CAN-Bus pour Raspberry Pi 2/3 : https://tronixlabs.com.au/raspberry-pi/pican2-can-bus-board-for-raspberry-pi-3-2-australia/

[6] CAN specification 2.0 part A and part B

[7] TEXIER P.-J., *« Le Relais 2 x 5V ...dans l'IoT ou l'art de piloter en BLE les périphériques de la WARP7 »* : http://connect.ed-diamond.com/GNU-Linux-Magazine/GLMF-200, janvier 2017

[8] PETAZZONI T., *« Introduction au Device Tree sur ARM »*, janvier 2016 : http://connect.ed-diamond.com/Open-Silicium/OS-017

[9] Crosstool-NG : http://crosstool-ng.github.io

[10] Buildroot : https://buildroot.org/

[11] SocketCAN : https://www.kernel.org/doc/Documentation/networking/can.txt

[12] can-utils : https://github.com/linux-can/can-utils

[13] CANOPEN : https://www.can-cia.org/canopen/

JOUONS AVEC LES BITS... EN PYTHON

TRISTAN COLOMBO

MOTS-CLÉS : PYTHON, BINAIRE, BITS, MÉMOIRE

Avez-vous déjà essayé de manipuler des données binaires en Python ? Ce n'est pas évident n'est-ce pas ? La question qui se pose est alors la suivante : la limitation provient-elle du langage lui-même ou simplement du manque de documentation à ce sujet ? Pour pouvoir y répondre, je me suis plongé dans une série de tests.

Bienvenue dans le monde merveilleux du binaire en Python, un monde où... beaucoup de gens se cassent les dents. Pourtant, venant d'un langage permettant d'effectuer de nombreuses opérations complexes très simplement, cela paraît complètement illogique.

Cet article présentera une suite de manipulations réalisables en Python pour manipuler des bits. Toutes les manipulations seront présentées sous la forme de tests dans des shell Python auxquels j'ai ajouté la coloration syntaxique pour en faciliter la lecture. Tout débute donc par un :

```
$ python3
```

1. POUR COMMENCER : LES OCTETS

Il existe bien un type permettant de définir des octets et comme on pouvait s'y attendre, son nom est **bytes** :

```
>>> octet = bytes(1)
>>> octet
b'\x00'
>>> type(octet)
<class 'bytes'>
```

La fonction **bytes()** nous permet donc de créer des chaînes d'octets en précisant en paramètre le nombre d'octets souhaité :

```
>>> octets = bytes(4)
>>> octets
b'\x00\x00\x00\x00'
```

ATTENTION !

Le type `bytes` est non mutable : vous ne pourrez donc pas modifier la valeur d'une variable de ce type.

Pour créer des chaînes d'octets non nulles, vous devez spécifier une liste de valeurs comprises entre **0** et **255** (puisque ces valeurs doivent tenir sur un octet) sous forme donc d'une liste ou d'un tuple :

```
>>> bytes((16,))
b'\x10'
>>> bytes((3,2,4))
b'\x03\x02\x04'
```

1.1 Convertir des octets en chaîne de caractères

Nous pouvons créer des octets... pour autant, nous souhaiterions les afficher de manière plus lisible (pour un humain standard). Ainsi, au lieu d'avoir **`b'\x00'`** pour un octet dont tous les bits sont à zéro, j'aimerais avoir **`'00000000'`**. Pour cela, nous allons devoir passer par plusieurs étapes :

1. Considérons que nous manipulons un unique octet :

```
>>> octet = bytes(1)
>>> octet
b'\x00'
```

2. Nous allons commencer par en obtenir la représentation entière :

```
>>> int.from_bytes(octet, byteorder='little', signed=False)
0
```

Nous avons utilisé ici la fonction **`from_bytes()`** **[1]** qui prend en premier paramètre un objet de type **`bytes`**, puis l'ordre des bits utilisé pour la conversion (**`'little'`** pour une lecture de droite à gauche - poids fort à droite (*little-endian*) - et **`'big'`** pour une lecture de gauche à droite - poids fort à gauche (*big-endian*)) et enfin le paramètre **`signed`** indique s'il faut utiliser un complément à deux pour représenter l'entier.

RAPPEL

Le complément à deux permet de représenter des entiers relatifs :

- **les entiers positifs sont écrits de manière traditionnelle sur 7 bits (on conserve un bit pour le signe qui sera ici `0` pour +) ;**
- **les entiers négatifs sont écrits sur 7 bits puis on prend leur inverse (on applique un `not` bit à bit) et on ajoute `1`.**

Exemple sur un octet :

- **`4` s'écrit `0 0000100` (le premier `0` a été décalé pour bien montrer qu'il s'agit du signe) ;**
- **`-4` s'écrit `1 1111100` (en inversant les bits de `00000100`, on obtient `11111011` puis en ajoutant `1` on obtient bien `11111100`).**

On mesure donc l'importance du paramètre `signed` de `int.from_bytes()` : pour `11111100`, avec `signed=True` , on obtient `-4` et avec `signed=False` on obtient `252`, ce qui peut être vérifié très facilement :

```
>>> relatif = b'\xfc'
>>> int.from_bytes(relatif, byteorder='little', signed=True)
-4
>>> int.from_bytes(relatif, byteorder='little', signed=False)
252
```

Pour savoir comment j'ai obtenu `b'\xfc'` pour `-4` vous devrez lire la suite de l'article...

NOTE

Il est possible de connaître l'ordre natif utilisé par le système pour la représentation des octets par :

```
>>> import sys
>>> sys.byteorder
'little'
```

3. Nous utilisons **`bin()`** pour obtenir la représentation binaire sous forme de **`1`** et de **`0`** :

```
>>> bin(int.from_bytes(octet, 'little', signed=False))
'0b0'
```

4. Nous éliminons le préfixe **`0b`** indiquant une représentation binaire :

```
>>> bin(int.from_bytes(octet, 'little', signed=False))[2:]
'0'
```

5. Pour que notre représentation des octets soit correcte, nous devons compléter la partie gauche par des zéros. La méthode **zfill()** remplit justement ce rôle. Elle s'applique sur une chaîne de caractères et la complète par des zéros jusqu'à ce que la chaîne ait une taille égale à la taille qui lui est passée en paramètre :

```
>>> mag = 'GLMF'
>>> mag.zfill(10)
'000000GLMF'
```

Il se trouve qu'une variable **bytes** passée en paramètre à la fonction **len()** permet d'obtenir le nombre d'octets de ladite variable :

```
>>> octets = bytes(4)
>>> len(octets)
4
```

Nous pouvons donc maintenant réaliser notre opération en appliquant à notre résultat la méthode **zfill()** à laquelle nous passons en paramètre le nombre de bits de notre variable **octet** (**len(octet) * 8**) :

```
>>> bin(int.from_bytes(octet, 'little',
signed=False))[2:].zfill(len(octet)*8)
'00000000'
```

Et voilà ! Bien sûr, idéalement, si vous devez utiliser de nombreuses conversions d'octets en chaîne de caractères, il est préférable de placer cette ligne dans une fonction :

```
>>> def bytes2str(octets):
...     return bin(int.from_bytes(octets,
'little', signed=False))[2:].zfill(len(octets)*8)
...
>>> bytes2str(b'\xfc')
'11111100'
```

1.2 Convertir des entiers en octets

Il existe la fonction réciproque de **int.from_bytes()** : il s'agit de **int.to_bytes()** qui accepte en paramètres la taille en octets de la conversion et bien sûr **byteorder** et **signed**. La conversion de l'entier **4** en octets se fait ainsi par :

```
>>> quatre = (4).to_bytes(1,
byteorder='little', signed=True)
>>> quatre
b'\x04'
>>> quatre = (4).to_bytes(4,
byteorder='little', signed=True)
>>> quatre
b'\x04\x00\x00\x00'
>>> quatre = (4).to_bytes(4, byteorder='big',
signed=True)
>>> quatre
b'\x00\x00\x00\x04'
```

1.3 Convertir des chaînes de caractères en octets

Il est possible de convertir simplement une chaîne de caractères de **0** et de **1** en entier :

```
>>> int('00000100', 2)
4
```

Il ne reste plus qu'à convertir l'entier en **bytes** :

```
>>> quatre = int('00000100', 2).to_bytes(1,
byteorder='little', signed=True)
>>> quatre
b'\x04'
```

Mais si l'on souhaite convertir une chaîne de caractères représentant des octets en entier, nous ne pourrons pas nous satisfaire d'un appel à **int()** car nous ne gèrerions ni l'ordre de représentation des bits ni l'aspect signé. Il faudrait effectuer une conversion en entier, puis en **bytes** pour revenir aux entiers :

```
>>> int.from_bytes(int('11111110', 2).to_
bytes(2, byteorder='little', signed=True),
byteorder='little', signed='True')
254
```

Tout cela était fort intéressant, mais nous n'avons pu manipuler que des octets... sans les modifier !

1.4 Un tableau d'octets pour pouvoir les modifier

En plus de **bytes**, il existe un type **bytearray** qui peut être vu comme un tableau de **bytes** modifiables.

bytearray ne porte pas vraiment bien son nom, car en fait ce ne sont pas des **bytes** qui sont stockés, mais des **int**... Nous pouvons par exemple y stocker les valeurs **0** et **4** puis effectuer un décalage à gauche de **1** sur la valeur **4** : **4** valant $(100)_2$, nous obtiendrons $(1000)_2$ soit **8**.

```
>>> array_bytes = bytearray(b'\x00\x04')
>>> for num, val in enumerate(array_bytes):
...     print(num, ':', val)
...
```

```
0 : 0
1 : 4
>>> array_bytes[1]
4
>>> type(array_bytes[0])
<class 'int'>
>>> array_bytes[1] << 1
8
```

Nous venons de voir un décalage de bits... penchons-nous donc maintenant sur la manipulation de bits.

2. PASSONS AUX BITS

Les opérateurs suivants sont disponibles pour travailler sur les bits :

Opérateur	Fonction
~	Complément à 1 (NOT bit à bit). Exemple : **~2 = -3** **2 = (00000010)$_2$** et **-3 = (11111101)$_2$** **~2 = 11111101**
&	ET bit à bit. Exemple : **2 & 3 = 22 = (10)$_2$** et **3 = (11)$_2$** **1 & 1 = 1** et **0 & 1 = 0** donc **10 & 11 = 10**
|	OU bit à bit. Exemple : **2 | 3 = 3** **2 = (10)$_2$** et **3 = (11)$_2$** **1 | 1 = 1** et **0 | 1 = 1** donc **10 | 11 = 11**
^	XOR bit à bit (OU exclusif qui n'est vrai que si l'un ou l'autre des opérandes est vrai et non les deux). Exemple : **2 ^ 3 = 1** **2 = (10)$_2$** et **3 = (11)$_2$** **1 ^ 1 = 0** et **0 ^ 1 = 1** donc **10 ^ 11 = 01**
<<	Décalage à gauche. Exemple : **3 << 1 = 6** **3 = (011)$_2$** et **6 = (110)$_2$**
>>	Décalage à droite. Exemple : **6 >> 2 = 1** **6 = (110)$_2$** et **1 = (001)$_2$**

Comme nous manipulons des entiers, il peut être intéressant de savoir combien de place ils occupent en mémoire. Pour cela, le module **sys** fournit la fonction **getsizeof()** :

```
>>> import sys
>>> sys.getsizeof(1)
28
>>> sys.getsizeof(156656456456)
32
```

Avec une architecture 64 bits, les entiers sont stockés sur au moins 28 octets (la taille s'adapte automatiquement à la taille de l'entier). Donc même pour stocker la valeur **1** qui tient sur un bit et donc sur un octet, nous utilisons 28 octets ! Finalement, heureusement que nous pouvons créer des objets de type **bytes** contenant par exemple un seul octet :

```
>>> a = bytes(1)
>>> sys.getsizeof(a)
34
```

Quoi ????? On stocke 1 octet dans 34 octets ? Là il y a un gros problème ! Avant de nous emballer, reprenons les choses calmement... La fonction **sys.getsizeof()** nous donne une approximation de la taille en octets de l'objet qui lui est passé en paramètre. Voilà ! Ce qui nous choque vient de là : les 34 octets correspondent à la taille d'un objet **bytes** qui traîne pas mal d'attributs et de méthodes (68 pour être précis)...

```
>>> dir(a)
['__add__', '__class__', '__
contains__', '__delattr__',
'__dir__', '__doc__', '__eq__',
'__format__', '__ge__', ...,
'splitlines', 'startswith',
'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
```

La taille obtenue nous indique donc la place occupée par l'ensemble des pointeurs utilisés par un objet. Donc la variable **a** qui est un octet (objet **bytes**) n'a rien à voir avec un octet au sens du **C** qui tiendrait sur un **char** donc 8 bits. Donc oui, on le savait, par rapport au C la consommation mémoire est énorme... mais on développe plus vite, il faut bien faire un choix...

NOTE

L'occupation mémoire en Python n'est qu'une estimation puisque de nombreux pointeurs entrent en jeu au niveau des attributs et des méthodes. Même en utilisant un module spécifique tel que `pympler` promettant un calcul récursif grâce à `pympler.asizeof`, ou même depuis Python 3.4 avec `tracemalloc` [2], nous resterons au stade de l'approximation.

Nous jetterons donc temporairement un voile pudique sur cette question de la mémoire (qui, il faut le reconnaître, fait très mal, même pour un Pythoniste convaincu). Attachons-nous plutôt aux opérations sur les bits, ça nous changera les idées. Avant de nous lancer, nous allons créer un petit module reprenant nos découvertes de la première section et qui nous permettra de visualiser plus simplement nos données. Ce module s'appellera **`easy_binary.py`** :

```
def bytes2str(octets, byteorder='little', signed=True):
    return bin(int.from_bytes(octets, byteorder=byteorder,
signed=signed))[2:].zfill(len(octets)*8)

def bin_int2str(integer, bytes=4, byteorder='little',
signed=True):
    return bytes2str((integer).to_bytes(bytes,
byteorder=byteorder, signed=signed))

def bin_str2int(string, bytes=4, byteorder='little',
signed=True):
    return int.from_bytes(int(string, 2).to_bytes(2,
byteorder=byteorder, signed=signed), byteorder=byteorder,
signed=signed)
```

Peut-être vous demandez-vous pourquoi pour convertir un entier en chaînes de caractères de **1** et **0** on ne fait pas simplement :

```
return bin(integer)[2:].zfill(bytes*8)
```

La réponse est simple : dans ce cas, il n'y a pas moyen de gérer la représentation (ordre des bits) ni l'aspect signé ou non.

Ce que nous allons voir maintenant dans la suite est déjà résumé dans le tableau précédent sur les opérateurs. Il s'agit seulement d'une application de ces opérateurs en Python sur des exemples concrets.

2.1 Décalage de bits

Pour le décalage de bits nous allons utiliser un entier inférieur à **255** de manière à ce qu'il puisse être représenté sur un octet (à condition qu'il soit non signé), ce qui sera plus lisible. Prenons donc au hasard la valeur **127** et effectuons quelques décalages :

```
>>> from easy_binary import bin_int2str
>>> val = 127
>>> bin_int2str(val, bytes=1)
'01111111'
>>> dec = val >> 2
>>> dec
31
>>> bin_int2str(dec, bytes=1)
'00011111'
```

Le décalage est bien celui attendu. Essayons maintenant un décalage de **1** vers la gauche. Partant de $(01111111)_2$ nous devrions obtenir $(11111110)_2$:

```
>>> val = 127
>>> bin_int2str(val, bytes=1)
'01111111'
>>> dec = val << 1
>>> dec
254
>>> bin_int2str(dec, bytes=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "easy_binary.py", line 5, in bin_int2str
    return bytes2str((integer).to_bytes(bytes,
byteorder=byteorder, signed=signed))
OverflowError: int too big to convert
```

Aïe, raté ! Nous serions-nous trompés ?

```
>>> bin_int2str(dec, bytes=2)
'0000000011111110'
```

Visiblement non, le résultat est correct ! Pourtant Python tient à passer sur deux octets pour la conversion... Serait-ce encore un « tour » de Python ? Non, rassurez-vous : nous avons affecté par défaut la valeur **`True`** au paramètre **`signed`** de **`bin_int2str()`** et donc nous ne pouvons représenter les entiers que jusqu'à **127** :

```
>>> bin_int2str(dec, bytes=1, signed=False)
'00000b10'
```

Ah, voilà un autre problème... C'est quoi encore cette représentation ? Reprenons notre fonction de conversion pas à pas :

```
>>> bdec = (dec).to_bytes(1, byteorder='little',
signed=False)
>>> bdec
b'\xfe'
>>> bin(int.from_bytes(bdec, byteorder='little',
signed=False))[2:].zfill(len(bdec)*8)
'11111110'
```

Apparemment, elle devrait fonctionner correctement. Ajoutons des sorties de contrôle à la fonction **`bin_int2str()`** :

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 !

```
>>> bin_int2str(dec, bytes=1, signed=False)
Entrée : b'\xfe'
True
-2
-0b10
'00000b10'
```

Le problème vient bien de cette fonction, ou plutôt de son appel... en fait nous avons oublié de passer en paramètre **byteorder** et **signed** et nous prenons donc les valeurs par défaut. La correction est la suivante :

```
def bin_int2str(integer, bytes=4, byteorder='little',
signed=True):
    return bytes2str((integer).to_bytes(bytes,
byteorder=byteorder, signed=signed), byteorder=byteorder,
signed=signed)
```

Et cette fois tout fonctionne correctement :

```
>>> bin_int2str(dec, bytes=1, signed=False)
'11111110'
```

Tout cela pour vous montrer combien vous devrez être vigilant sur la définition de la représentation et de l'aspect signé (ou non) de vos données...

2.2 Utilisation d'un masque

Le décalage fonctionne assez simplement tant que l'on travaille sur des entiers. Nous allons voir qu'il en est de même pour les masques. Prenons pour exemple les permissions sur les fichiers. Par défaut ces permissions sont réglées sur **666** avec un masque de **002**. Pour connaître la permission finale des fichiers, on applique l'inverse du masque sur les permissions initiales. Si nous faisons le calcul au préalable à la main :

- **666** est une notation octale qui donne en binaire $(110110110)_2$;
- **002** est également une notation octale qui donne $(000000010)_2$;
- l'inverse de **002** donc NOT **002** ou **~002** donne $(111111101)_2$;
- en appliquant le masque, on effectue **666** AND NOT **002** soit **666 & ~002** ce qui donne $(110110110)_2$ & $(111111101)_2$ = $(110110100)_2$ (soit pour la petite histoire les permissions **rw- rw- r--**).

Voici maintenant la même chose en Python :

```
>>> default = bin_int2str(0o666, bytes=2, signed=False)
>>> default
'0000000110110110'
>>> mask = bin_int2str(0o002, bytes=2, signed=False)
>>> mask
'0000000000000010'
>>> new_perms = 0o666 & ~0o002
>>> new_perms
436
>>> bin_int2str(new_perms, bytes=2, signed=False)
'0000000110110100'
```

La notation **0o** permet d'indiquer un encodage octal. Ainsi, **0o666** est égal à **6 x 82 + 6 x 81 + 6 x 80 = 6 x 64 + 6 x 8 + 6 X 1 = 438**. Pour le reste, nous ne faisons qu'employer des opérateurs logiques ainsi que notre fonction **bin_int2str()**.

2.3 Effectuer une opération logique et une affectation en une ligne

Tout comme pour **+=**, **-=**, etc., il existe les opérateurs **~=**, **&=**, **|=**, **^=**, **>>=** et **<<=**. Ainsi, en prenant pour exemple la ligne **i = i << 2**, il est possible de raccourcir son écriture en **i <<= 2** :

```
>>> i = 1
>>> i <<= 2
>>> i
4
>>> bin_int2str(i, bytes=1,
signed=False)
'00000100'
```

2.4 Conversion d'un entier sur 16 bits en deux caractères sur 8 bits

Sur 16 bits, on peut représenter les entiers non signés de **0** à 2^{16} **- 1**, soit **65535**. Si l'on considère des entiers signés, on ira de **-32768** à **32767** (**-**2^{15} à 2^{15} **- 1**).

NOTE

De manière générale, sur n bits on peut représenter :

- **les entiers non signés de 0 à 2^n - 1 ;**
- **les entiers signés de -2^{n-1} à 2^{n-1} - 1.**

Pour notre exemple, nous considérerons des entiers non signés et nous prendrons comme valeur **32555** :

```
>>> val = 32555
>>> bin_int2str(val, bytes=2, signed=False)
'0111111100101011'
```

Il s'agit ensuite d'une application des opérateurs logiques et du masquage. Pour obtenir les 8 bits de poids fort il suffit de faire un décalage à droite de 8 :

```
>>> high_val = val >> 8
>>> high_val
127
>>> bin_int2str(high_val, bytes=1, signed=False)
'01111111'
```

On obtient bien les 8 premiers bits de $(0111111100101011)_2$. Pour obtenir les 8 bits de poids faible, on applique le masque $(11111111)_2$ soit **255** :

```
>>> from easy_binary import bin_str2int
>>> bin_str2int('11111111', bytes=1, signed=False)
255
```

Nous avons ensuite nos bits de poids faible :

```
>>> low_val = val & 255
>>> bin_int2str(low_val, bytes=1, signed=False)
'00101011'
```

Il s'agit bien des 8 derniers bits de $(0111111100101011)_2$.

3. UN MINI XXD

Que fait la commande **xxd** ? Elle permet de faire des *dump* de fichiers en hexadécimal ou en binaire. Prenons l'exemple du fichier texte **fichier.txt** suivant :

```
Ceci est un petit exemple.
Il s'agit d'un simple fichier texte.
Avec des accents éàù...
```

Avec xxd nous pouvons générer un *dump* hexadécimal :

```
$ xxd fichier.txt
0000000: 4365 6369 2065 7374 2075 6e20 7065 7469  Ceci est un peti
0000010: 7420 6578 656d 706c 652e 0a49 6c20 7327  t exemple..Il s'
0000020: 6167 6974 2064 2775 6e20 7369 6d70 6c65  agit d'un simple
0000030: 2066 6963 6869 6572 2074 6578 7465 2e0a   fichier texte..
0000040: 4176 6563 2064 6573 2061 6363 656e 7473  Avec des accents
0000050: 20c3 a9c3 a0c3 b92e 2e2e 0a               ..........
```

Plus fort, nous pouvons faire du « reverse dumping » grâce à l'option **-r** :

```
$ xxd fichier.txt > fichier.hex
$ xxd -r fichier.hex
Ceci est un petit exemple.
Il s'agit d'un simple fichier texte.
Avec des accents éàù...
```

NOTE

Avec l'option -b vous pouvez obtenir le *dump* binaire, mais malheureusement le « reverse dumping » devra là se faire à la main :

```
$ xxd -b fichier.txt
0000000: 01000011 01100101 01100011 01101001 00100000 01100101  Ceci e
0000006: 01110011 01110100 00100000 01110101 01101110 00100000  st un
000000c: 01110000 01100101 01110100 01101001 01110100 00100000  petit
0000012: 01100101 01111000 01100101 01101101 01110000 01101100  exempl
0000018: 01100101 00101110 00001010 01001001 01101100 00100000  e..Il
000001e: 01110011 00100111 01100001 01100111 01101001 01110100  s'agit
0000024: 00100000 01100100 00100111 01110101 01101110 00100000   d'un
000002a: 01110011 01101001 01101101 01110000 01101100 01100101  simple
0000030: 00100000 01100110 01101001 01100011 01101000 01101001   fichi
0000036: 01100101 01110010 00100000 01110100 01100101 01111000  er tex
000003c: 01110100 01100101 00101110 00001010 01000001 01110110  te..Av
0000042: 01100101 01100011 00100000 01100100 01100101 01110011  ec des
0000048: 00100000 01100001 01100011 01100011 01100101 01101110   accen
000004e: 01110100 01110011 00100000 11000011 10101001 11000011  ts ...
0000054: 10100000 11000011 10111001 00101110 00101110 00101110  ......
000005a: 00001010                                               .
```

Pour utiliser sur un cas pratique tout ce que nous avons vu dans cet article, nous allons réaliser une sorte de mini xxd qui se présentera sous la forme d'un module disposant d'une fonction **dump()** qui génèrera un *dump* hexadécimal et **reverse_dump()** qui renverra les données texte correspondant à un *dump* hexadécimal. Voici le code de **xxd.py** :

```
01: import sys
02: import binascii
03:
04:
05: def dump(filename):
06:     try:
07:         with open(filename, 'rb') as fic:
08:             line = 0
09:             newline = True
10:             block = ''
11:             nb_block = 0
12:             strline = ''
```

```
13:             while True:
14:                 if newline:
15:                     print(str(line).zfill(7) + ': ', end='')
16:                     newline = False
17:                     line += 10
18:                 car = fic.read(1)
19:                 if car == b'':
20:                     break
21:                 if len(block) == 4:
22:                     print(block, end=' ')
23:                     block = ''
24:                     nb_block += 1
25:                 if nb_block == 8:
26:                     nb_block = 0
27:                     newline = True
28:                     print('  ' + strline)
29:                     strline = ''
30:                 hexa = binascii.hexlify(car).decode('utf-8')
31:                 block += hexa
32:                 int_val = int(hexa, 16)
33:                 if int_val < 32 or int_val > 127:
34:                     int_val = 46 # correspond à 0x2e, le
code ASCII du point
35:                 strline += chr(int_val)
36:             if block != '':
37:                 print(block, end=' ')
38:                 if len(block) == 2:
39:                     print('  ', end='')
40:                 add_block = 0
41:                 for i in range(9 - nb_block):
42:                     print('  ', end='')
43:                     add_block += 1
44:                     if add_block % 2 == 0:
45:                         print('  ', end='')
46:                 print(strline)
47:     except Exception as e:
48:         print('Erreur lors de l\'ouverture/lecture du
fichier {}'.format(filename))
49:         print(e)
50:         exit(1)
51:
52: def reverse_dump(filename):
53:     try:
54:         with open(filename, 'r') as fic:
55:             newline = ''
56:             for line in fic:
57:                 newline += line[9:][:-19].replace(' ', '')
58:             print(binascii.unhexlify(newline).
decode('utf-8'), end='')
59:     except Exception as e:
60:         print('Erreur lors de l\'ouverture/lecture du
fichier {}'.format(filename))
61:         print(e)
62:         exit(1)
63:
64:
65: if __name__ == '__main__':
66:     if len(sys.argv) != 3:
67:         print('Syntax: xxd.py [dump|reverse_dump]
<filename>')
68:         exit(2)
69:     if sys.argv[1] == 'dump':
70:         dump(sys.argv[2])
71:     elif sys.argv[1] == 'reverse_dump':
72:         reverse_dump(sys.argv[2])
```

Nous utiliserons ici le module **sys** (ligne 1) pour accéder aux arguments de la ligne de commandes et le module **binascii** (ligne 2) pour ses fonctions **hexlify()** et **unhexlify()** **[3]** permettant respectivement d'obtenir la valeur hexadécimale de la chaîne de caractères passée en paramètre ou la chaîne de caractères issue des valeurs hexadécimales passées en paramètre.

La fonction **dump()** des lignes 5 à 50 permet d'obtenir le *dump*, donc la même représentation que la commande **xxd** sans option. Le fichier passé en paramètre est ouvert en lecture en mode binaire (ligne 7) et cinq variables sont définies :

- **line** permet de compter le nombre de « lignes » : chaque ligne contient 16 codes hexadécimaux donc le nombre de codes augmente de **0x10** (en hexadécimal) toutes les lignes. Comme il s'agit d'une dizaine par ligne, nous pourrons considérer qu'il s'agit d'entiers et ajouter simplement **10** à chaque nouvelle ligne (ligne 17) ;
- **newline** justement est un booléen permettant de savoir s'il faut afficher le compteur de début de ligne. Si tel est le cas, on affiche la valeur de **line** en complétant par des zéros de manière à avoir sept caractères (ligne 15) puis on désactive **newline** en la passant à **False** et on incrémente **line** de **10** ;
- **block** définit un « bloc » composé de deux codes hexadécimaux. Lorsque deux codes sont stockés dans bloc, alors sa taille est de 4 (ligne 21) : on affiche le bloc et on le vide. On en profite pour compter dans **nb_block** le nombre de blocs affichés sur la ligne (ligne24).
- **nb_block** contient le nombre de blocs de la ligne courante. Au bout de 8 blocs (ligne 25) on passe à une nouvelle ligne, ce qui entraîne une mise à zéro du nombre de blocs (ligne

26), le passage à **True** de **newline** (ligne 27) et l'affichage puis la remise à vide de **strline** (lignes 28 et 29) ;

- **strline** contient les données de la ligne sous forme de caractères ASCII. À chaque ajout d'un code hexadécimal correspondant au caractère lu (lignes 30 et 31), on ajoute la représentation ASCII dans **strline** (ligne 35 avec **chr()** après avoir converti la valeur hexadécimale en base **10** avec **int(<valeur_hexa>, 16)** en ligne 32). Pour les caractères non affichables, on les remplace par des points (lignes 33 et 34).

En étudiant ces cinq variables, nous avons pu voir la quasi-totalité du code de cette fonction. Il manque seulement les lignes 36 à 46 qui permettent d'afficher les données de la dernière ligne si celle-ci n'est pas complète (moins de 16 codes).

La fonction **reverse_dump()** est beaucoup plus rapide à écrire : on lit le fichier en mode texte (ligne 54) et pour chaque ligne on ne conserve que les codes hexadécimaux en retirant les numéros de ligne et la représentation ASCII (lignes 56 et 57). Cela correspond à un slice sur **[9:]** et sur **[:-19]** et à la suppression des caractères espace. Une fois que l'on a récupéré tous les codes hexadécimaux, on procède à la conversion (ligne 58).

ATTENTION !

La conversion doit être effectuée en une seule fois, car en utf-8 un caractère peut occuper de un à quatre octets...

Le programme principal des lignes 65 à 72 gère les paramètres de la ligne de commandes de manière minimaliste pour pouvoir appeler au choix **dump** ou **reverse_dump**. À l'exécution, nous obtenons le même résultat que les commandes **xxd** et **xxd -r**.

4. PARCE QU'ON NE PEUT PAS SE SATISFAIRE DU STOCKAGE D'UN OCTET DANS 34 OCTETS...

Non, il n'est pas normal d'avoir besoin de 34 octets pour stocker un seul octet et vous n'achèverez pas cet article sans un début de réponse.

Tout d'abord, vous pourriez être tentés de régler rapidement le problème, simplement en utilisant des types C dans Python. C'est effectivement une bonne idée... mais d'après vous, un type C c'est quoi ? Un objet ! Du coup, un **char** ne sera pas un octet :

```
>>> import sys
>>> import ctypes
>>> a = ctypes.c_char(65)
>>> a
c_char(b'A')
>>> sys.getsizeof(a)
136
>>> a.value
b'A'
>>> sys.getsizeof(a.value)
34
```

Attention, cela ne signifie nullement qu'utiliser **ctypes** est une mauvaise idée ! En interne la donnée proprement dite sera effectivement stockée sur un octet. Mais la solution est peut-être un peu trop « brutale » et nous pouvons sans doute trouver mieux.

Il y a une petite expérience que nous n'avons pas réalisée... En Python tout est objet et donc, comme nous l'avons vu précédemment, il est normal d'obtenir une occupation mémoire supérieure à celle du C. En partant de ce constat, un octet « coûtant » 34 octets, la place mémoire de deux octets ne devrait pas augmenter de manière conséquente (on devrait même avoir logiquement 5 octets). Testons cela :

```
>>> import sys
>>> a = bytes(1)
>>> sys.getsizeof(a)
34
>>> a = bytes(2)
>>> sys.getsizeof(a)
35
```

Voilà qui est plus rassurant. Donc normalement, pour **n** octets nous devrions avoir besoin de **33 + n** octets. Testons avec 25 octets :

```
>>> a = bytes(25)
>>> sys.getsizeof(a)
58
```

58 est bien le résultat de **33 + 25** ! Cela est rassurant, mais si l'on doit travailler sur de nombreux octets, le « gaspillage » des 33 octets sera multiplié par le nombre de valeurs utilisées. Suivons donc la même logique que précédemment en utilisant un tableau d'octets (**bytearray**) :

```
>>> a = bytearray(1)
>>> a[0]
0
>>> sys.getsizeof(a)
58
>>> a = bytearray(2)
>>> sys.getsizeof(a)
59
```

Sur un **bytearray**, la taille pour **n** octets stockés sera de **57 + n**, quel que soit le nombre de valeurs différentes. Le tableau suivant montre l'occupation mémoire en fonction du type et du nombre de valeurs utilisées pour stocker à chaque fois un octet ;

Type	Nombre de valeurs	Occupation mémoire (en octets)
bytes	1	34
bytes	2	68
bytes	5	170
bytes	15	850
bytearray	1	58
bytearray	2	59
bytearray	5	62
bytearray	25	82

En figure 1, un petit graphe nous permet de résumer cela en mettant en parallèle l'occupation mémoire du C.

Suivant le programme développé il est donc important de bien choisir le type de ces variables, car cela peut avoir très rapidement des répercussions sur l'exécution du code.

Mais nous avons vu que finalement il pouvait être pratique de travailler avec des entiers... la question qui se pose alors est de savoir comment économiser de la mémoire en stockant des entiers.

En Python3, comme vous devez le savoir il n'y a plus de différences entre entiers et entiers dits longs : il n'y a plus que des entiers longs et le type est ajusté automatiquement :

```
>>> a = 255
>>> sys.getsizeof(a)
28
>>> a = 3434532222
>>> sys.getsizeof(a)
32
>>> a = 33445353465346654343242
>>> sys.getsizeof(a)
36
```

Si nous voulons imposer une taille à nos entiers, la meilleure solution reste d'utiliser **Numpy** qui dispose des types **int8**, **int16**, **int32**, **int64** et de leurs équivalents non signés, préfixés par un **u** (**uint8**, etc.) **[4]**. Ici malheureusement l'estimation de l'occupation mémoire ne nous apportera strictement rien, on supposera donc que « logiquement », d'après ce que nous avons vu précédemment l'utilisation de tableaux numpy dont le type est fixé devrait être la meilleure solution :

```
>>> import numpy as np
>>> a = np.array([0, 100, 200, 25],
dtype=np.uint8)
```

ATTENTION !

Numpy ne vous renverra pas de message d'erreur en cas de dépassement de capacité. Si vous voulez stocker 266 dans un `uint` qui ne peut contenir des entiers que jusqu'à 255, on ne conservera que les 8 derniers bits : 266 vaut $(100001010)_2$ et donc 10 en conservant les 8 derniers bits $(00001010)_2$.

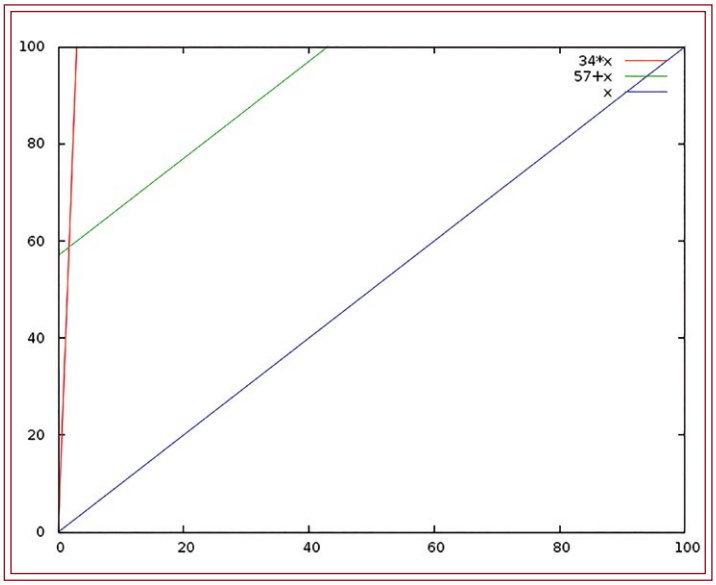


Fig. 1 : Occupation mémoire (ordonnées) en fonction du nombre de variables de 1 octet (abscisse). En rouge en Python avec le type bytes, en vert en Python avec le type bytearray et en bleu en C.

CONCLUSION

Pour la rédaction de cet article, j'ai parcouru le code source de Python, de ctypes et de numpy espérant y trouver une explication lumineuse, mais malheureusement ce ne fut pas le cas : les structures contiennent des pointeurs qui pointent vers des structures de pointeur, etc. J'ai également recherché comment analyser et profiler l'occupation mémoire de code Python et je n'ai rien trouvé de concluant bien que des efforts aient été réalisés de ce côté avec **tracemalloc**. Il ne reste donc que la déduction et donc normalement, numpy ayant été particulièrement optimisé, étant basé sur ctypes, c'est cette solution qui devrait utiliser le moins de mémoire (en passant par des tableaux). En tout cas une chose est sûre : il est possible de travailler en binaire en Python... et ce n'est pas très compliqué ! ■

RÉFÉRENCES

[1] Fonction from_bytes() : https://docs.python.org/3/library/stdtypes.html

[2] Module tracemalloc : https://docs.python.org/3/library/tracemalloc.html

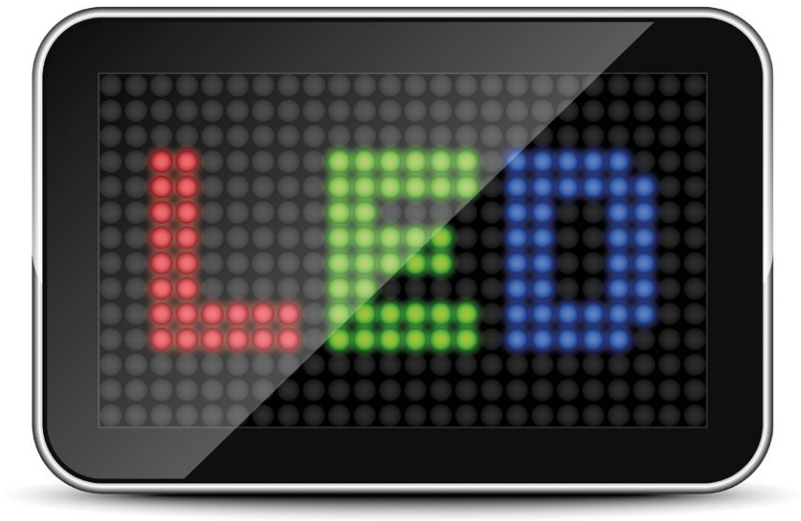
[3] Fonctions hexlify() et unhexlify() du module binascii : https://docs.python.org/3/library/binascii.html

[4] Les types de Numpy : https://docs.scipy.org/doc/numpy/user/basics.types.html

AFFICHER DU TEXTE SUR UN ÉCRAN DE LEDS WS2812

TRISTAN COLOMBO

MOTS-CLÉS : PYTHON, POLICE, PILLOW, WS2812, NUMPY



Ça n'a l'air de rien, mais afficher un texte sur un écran de 8x8 leds demande un minimum de réflexion ; surtout si l'on a pas envie de redéfinir entièrement une police...

Il n'y a rien de plus naturel que d'afficher un texte sur un écran. Pourtant, lorsque l'on modifie les contraintes et que l'écran ne peut plus afficher que 64 pixels (écran de 8x8 leds), si l'on ne veut pas passer une journée à redéfinir tous les caractères d'une police au format souhaité, il va falloir faire preuve d'un minimum d'imagination. Je vous propose dans cet article d'explorer une solution en **Python** permettant de parvenir au résultat souhaité en un minimum de lignes. Nous irons même un peu plus loin en ajoutant un scrolling permettant de faire défiler un texte complet.

NOTE

Cet article vient d'une problématique rencontrée lors de tests sur un écran de 8x8 leds WS2812. La solution proposée pourra bien sûr être adaptée à n'importe quel type d'écran et j'utiliserai d'ailleurs ici l'émulateur vrtneopixel [1] :

```
$ sudo pip3 install
vrtneopixel
```

En cas de problème pour l'installation de pygame, tapez la commande suivante avant de relancer l'installation de vrtneopixel :

```
$ sudo pip3 install
--upgrade pip
```

1. L'IDÉE

Créer une police minimale, contenant donc les 26 lettres de l'alphabet en majuscule plus les 10 chiffres et le caractère espace (ce n'est pas le caractère le plus compliqué :-)), nous amènerait à créer 37 listes de coordonnées. Pour avoir un aperçu du travail nécessaire, nous allons créer et afficher seulement la lettre **A** (si vous voulez faire les autres, libre à vous, pour ma part je les génèrerai...).

Sur un écran de 8x8 leds la lettre **A** pourra avoir la forme présentée en figure 1.

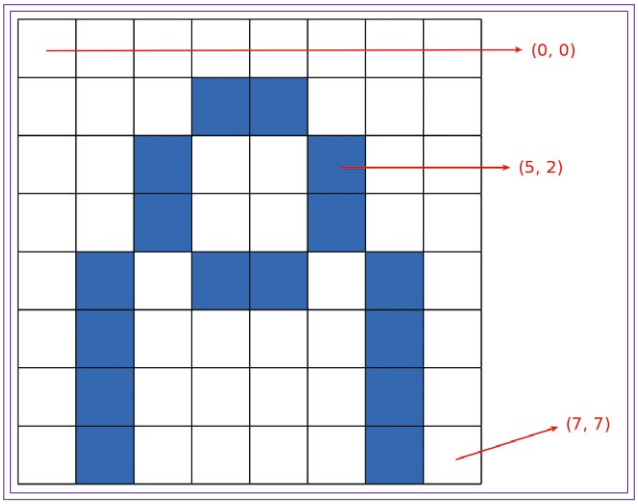


Fig. 1 : Lettre A dans un quadrillage.

Une fois la lettre dessinée, il faut noter la liste des coordonnées de tous les pixels qui la composent. Dans le cas du **A** de la figure 1, on obtient la liste **[(3, 1), (4, 1), (2, 2), (5, 2), (2, 3), (5, 3), (1, 4), (2, 4), (3, 4), (4, 4), (5, 4), (6, 4), (1, 5), (6, 5), (1, 6), (6, 6), (1, 7), (6, 7)]**.

Utilisons cette liste dans un script **A.py** pour afficher la lettre sur l'écran :

```
01: from vrtneopixel import *
02: import time
03:
04: # Variables de configuration de l'écran
05: LED_COUNT = 64
06: LED_PIN = 18
07: LED_FREQ_HZ = 800000
08: LED_DMA = 5
09: LED_BRIGHTNESS = 8
10: LED_INVERT = False
11:
12: def clearScreen(leds):
13:     for i in range(64):
14:         leds[i] = Color(0, 0, 0)
15:
16: def displayScreen(strip, leds):
17:     for i in range(64):
18:         strip.setPixelColor(i, leds[i])
19:     strip.show()
20:
21: if __name__ == '__main__':
22:     # Définition de la lettre A
23:     A = [(3, 1), (4, 1), (2, 2), (5, 2),
(2, 3), (5, 3), (1, 4), (2, 4), (3, 4), (4,
4), (5, 4), (6, 4), (1, 5), (6, 5), (1, 6),
(6, 6), (1, 7), (6, 7)]
24:     # Initialisation des leds en noir
25:     leds = [Color(0, 0, 0)] * 64
26:
27:     # Création d'un élément permettant de
"manipuler"
28:     # l'écran de leds
29:     strip = Adafruit_NeoPixel(LED_COUNT,
LED_PIN, LED_FREQ_HZ, LED_DMA, LED_INVERT, LED_
BRIGHTNESS)
30:
31:     # Initialisation de l'écran
32:     strip.begin()
33:
34:     # Ajout de la lettre sur l'écran
35:     for (x, y) in A:
36:         leds[x + y * 8] = Color(255, 0, 0)
37:
38:     # Affichage de l'écran
39:     displayScreen(strip, leds)
40:
41:     # Attente de 10s
42:     time.sleep(10)
43:
44:     # Effacement de l'écran
45:     clearScreen(leds)
46:     displayScreen(strip, leds)
```

Les lecteurs de (l'excellent) hors-série n°2 de *Hackable Magazine « Débutez en programmation sur Raspberry Pi »* auront reconnu la structure du code permettant d'afficher le serpent. Pour les autres, comme l'affichage des leds sur un écran de leds WS2812 se fait de manière linéaire (pour un écran de 8x8 leds de la led n°1 à la led n°64), il faut représenter l'écran sous la forme d'une liste de 64 pixels. Pour effacer l'écran (lignes 12 à 14), on affiche 64 pixels noirs (donc éteints) dans la fonction **clearScreen()**. L'affichage de l'écran (lignes 16 à 19) nécessite le passage en paramètre de **displayScreen()** d'une instance de **Adafruit_NeoPixel** pour convertir les données de la liste **leds** (passée également en paramètre) en pixels à l'écran. Le programme principal définit ensuite les coordonnées des pixels représentant la lettre **A** (ligne 23), crée **strip**, l'instance de **Adafruit_NeoPixel** qui permet d'accéder à l'écran (ligne 29), l'initialise (ligne 32), ajoute la lettre **A** dans la représentation des pixels de l'écran **leds**, et affiche la lettre à l'écran (ligne 39) avant de faire une pause de 10 secondes (ligne 42) et de tout effacer (lignes 45 et 46). Le résultat de l'exécution de ce code est visible en figure 2.

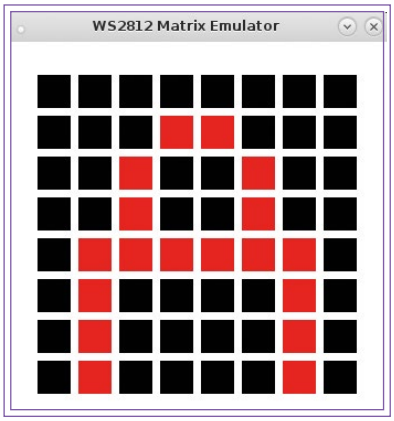


Fig. 2 : Affichage de la lettre A sur l'émulateur vrtneopixel.

Vous constatez donc que le travail permettant d'afficher nos 37 caractères est titanesque ! Il faut trouver une autre méthode !

Nous allons utiliser la bibliothèque **PIL** (*Python Imaging Library*) qui permet de générer une image bitmap en noir et blanc. Il suffit ensuite de parcourir l'image pixel à pixel et de détecter les pixels noirs pour obtenir la lettre à afficher. Le processus est résumé dans la figure 3.

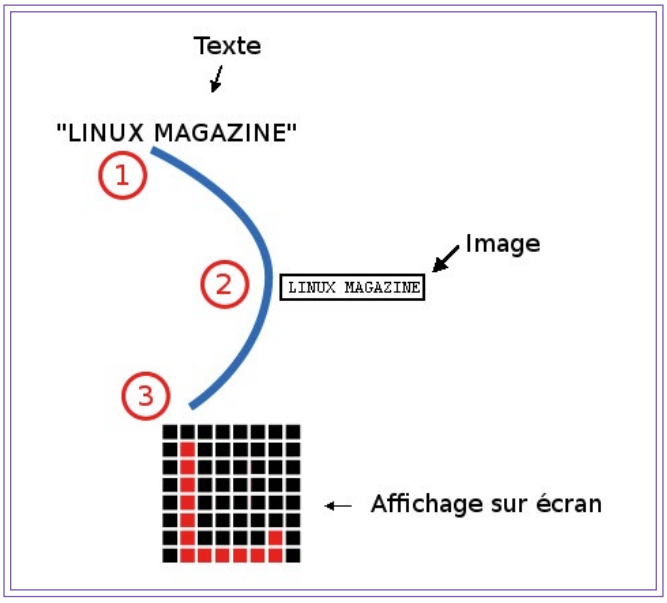


Fig. 3 : Conversion d'une chaîne de caractères (1) en bitmap noir et blanc (2) pour affichage sur écran de leds (3).

Si vous n'avez pas encore installé PIL (qui est en fait **Pillow**, le fork actif de PIL), vous devrez taper :

```
$ sudo apt install python3-pillow
```

L'utilisation d'une police à chasse fixe (donc de largeur constante) simplifiera grandement les choses. Et même mieux, en utilisant une police monospace (donc avec une hauteur et une largeur constantes), nous aurons encore moins de problème ! J'ai utilisé la police **Hack** (**Hack-Regular.ttf** que vous pourrez retrouver avec les codes sources de cet article), téléchargée sur https://www.fontsquirrel.com/fonts/list/classification/monospaced.

2. AFFICHAGE DE CARACTÈRES AVEC DES

Pour procéder par étapes, nous allons commencer par générer une image depuis une chaîne de caractères et afficher les caractères sous forme de **#** qui représenteront les leds de notre écran. Nous utiliserons les tableaux du module **Numpy** qui autorisent une souplesse supérieure aux listes :

```
$ sudo apt install python3-numpy
```

Voici le code de **text_char.py** qui convertit (pour l'instant) un texte en image puis en suite de **1** et de **0** :

```
01: from PIL import Image, ImageFont,
ImageDraw
02: import numpy as np
03:
04: def text2image(text, fontName='Hack-
Regular.ttf', pt=11, saveName=None):
05:     font = ImageFont.
truetype(fontName, pt)
06:     (w, h) = font.getsize(text)
07:     image = Image.new('L', (w, h), 1)
08:     draw = ImageDraw.Draw(image)
09:     draw.text((0, 0), text, font=font)
10:     if not saveName is None:
11:         image.save(saveName)
12:     return image
13:
14:
15: if __name__ == '__main__':
16:     image = text2image('LINUX
MAGAZINE')
17:     arr = np.asarray(image)
18:     arr = np.where(arr, 0, 1)
19:     arr = arr[(arr != 0).any(axis=1)]
20:     print(arr[:, 0:7])
```

La fonction **text2image()** des lignes 4 à 12 prend en paramètre une chaîne de caractères **text**, une police **fontName** (par défaut, j'utilise **Hack-Regular.ttf**), la taille de police **pt** et un nom de fichier de sauvegarde **saveName** si l'on souhaite exporter l'image générée (pour vérification). En ligne 5, on définit la police utilisée et sa taille dans **font** puis, en ligne 6, on calcule la taille occupée par le texte en utilisant cette police : **w** pour la largeur et **h** pour la hauteur. On crée ensuite une image en ligne 7 en utilisant le mode **L** **[2]** (8 bits par pixel en noir et blanc), en donnant le tuple définissant la taille de l'image et la couleur d'initialisation de l'image **[3]** (ici tous les pixels de l'image seront à **1**). En ligne 8, on crée une zone de dessin **[4]** dans laquelle on écrit la chaîne de caractères **text** avec la police **font** (ligne 9). Si un nom de fichier a été spécifié, on enregistre l'image (lignes 10 et 11) puis on renvoie l'image.

Le programme principal appelle la fonction **text2image()** pour convertir la chaîne **'LINUX MAGAZINE'** en image (ligne 16). En ligne 17, nous convertissons l'image en tableau Numpy puis nous inversons les valeurs en ligne 18 avec **where()** **[5]** :

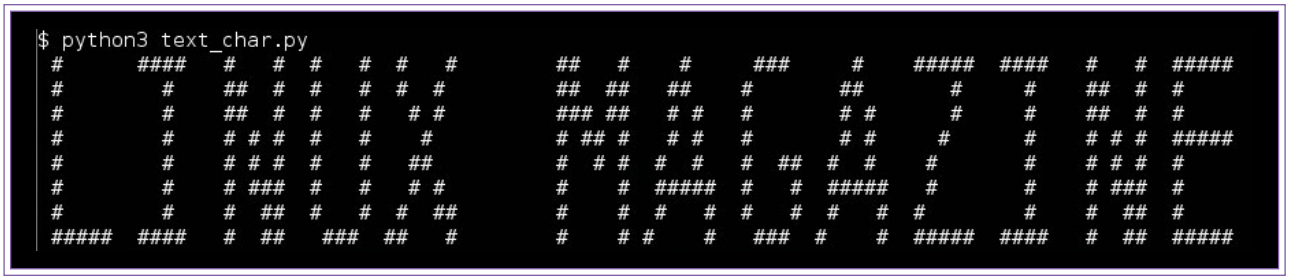


Fig. 4 : Affichage d'un texte dont les caractères sont composés par des #.

quand un élément de **arr** renvoie la valeur **True** (donc également **1**), on place la valeur **0** et, sinon (**False** donc également **0**), on place la valeur **1**. Pour finir, nous « nettoyons » le tableau en retirant les lignes qui ne contiennent pas d'information (tous les pixels sont à **0**). Ceci est réalisé en ligne 19 : **arr != 0** renvoie un tableau de booléens (pour tous les éléments de **arr**, **True** pour **1** et **False** pour **0**) et l'application de **any(axis=1)** **[6]** teste les éléments sur une dimension (nous obtiendrons un booléen par ligne : si une ligne ne contient que des **False**, nous obtiendrons **False** et **True** sinon). Ainsi ce mécanisme nous permet d'obtenir une liste du type **[False, False, ..., True]** qui, appliquée à **arr**, conserve uniquement les lignes qui contiennent au moins une valeur. La ligne 20 permet simplement d'afficher les éléments du tableau définissant le premier caractère (les 7 premières colonnes) comme on peut le voir à l'exécution :

```
$ python3 text_char.py
[[0 1 0 0 0 0 0]
 [0 1 0 0 0 0 0]
 [0 1 0 0 0 0 0]
 [0 1 0 0 0 0 0]
 [0 1 0 0 0 0 0]
 [0 1 0 0 0 0 0]
 [0 1 0 0 0 0 0]
 [0 1 1 1 1 1 0]]
```

On voit (ou on devine) bien un **L**. Pour afficher la chaîne de caractères complète avec des **#**, il suffit de remplacer les **0** par des espaces et les **1** par des **#**. On ajoute seulement les trois lignes suivantes à la fin du script (en supprimant la ligne 20) :

```
...
20:     arr = np.where(arr, '#', ' ')
21:     for row in arr:
22:         print(''.join(row))
```

On utilise à nouveau le **where()** (ligne 20) puis en parcourant les lignes nous concaténons les caractères **' '** et **'#'** (**join()** avec **''** en ligne 22). On peut voir en figure 4 que le résultat obtenu est bien celui attendu.

3. CONVERSION DES # EN LEDS

Maintenant que nous sommes capables d'afficher une chaîne de caractères sous forme de **#**, la conversion en leds ne sera guère complexe.

NOTE

Dans l'exemple proposé, la police Hack-Regular en 11pt génère des lettres de 8 x 7 pixels, à condition d'utiliser uniquement des caractères majuscules. Je n'ai pas automatisé la recherche de taille de police optimale en fonction des lettres et de la police utilisées, mais on peut tout à fait imaginer développer une fonction qui effectuerait cette tâche.

Tout d'abord, comme nous voulons effectuer un scrolling horizontal, nous allons ajouter un caractère espace avant et après la chaîne de caractères de manière à ce que le premier et le dernier caractère suivent le même mouvement que les autres caractères. La figure 5 illustre ce mécanisme.

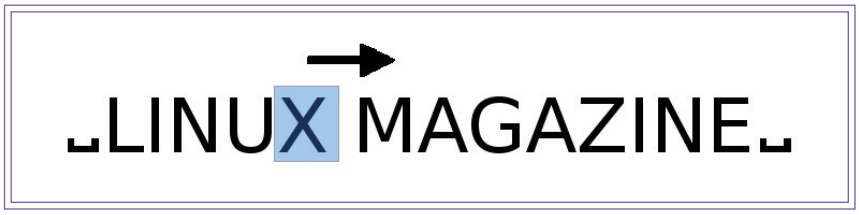


Fig. 5 : Scrolling horizontal sur une chaîne de caractères.

Deux solutions sont possibles : soit nous modifions le tableau **arr** en y concaténant deux tableaux de 8 x 7 composés uniquement de **0**, soit, avant de lancer la conversion de la chaîne en image, nous lui ajoutons des caractères espace en début et en fin. L'une est plus mathématique, l'autre plus astucieuse... à vous de choisir celle que vous préférez...

3.1 Ajout des « fenêtres » vides

3.1.1 Solution 1 : Concaténation de tableaux

Numpy dispose d'une fonction **concatenate()** **[7]** qui permet de concaténer deux tableaux suivant un axe (à condition que les dimensions soient compatibles) :

```
...
14: if __name__ == '__main__':
15:     image = text2image('LINUX
MAGAZINE')
16:     arr = np.asarray(image)
17:     arr = np.where(arr, 0, 1)
18:     arr = arr[(arr != 0).any(
axis=1)]
19:     arr = np.concatenate((np.
zeros((8, 7)), arr), axis=1)
20:     arr = np.concatenate((arr,
np.zeros((8, 7))), axis=1)
...
```

3.1.2 Solution 2 : Concaténation d'espaces

Pour concaténer des espaces à une chaîne de caractères, cela se passe de tout commentaire...

```
...
04: def text2image(text,
fontName='Hack-Regular.ttf',
pt=11, saveName=None):
05:     text = ' ' + text + ' '
...
```

3.2 Affichage caractère à caractère

Avant de pouvoir mettre en place le *scrolling* nous allons passer par une étape transitoire en affichant la chaîne de caractères, caractère à caractère. Pour cela, nous allons parcourir le tableau **arr** par fenêtre de 7 colonnes (en considérant toutes les lignes). Nous détacherons ainsi chaque lettre de la chaîne. Ici encore deux solutions sont possibles : soit en bornant les colonnes dans une boucle **for**, soit à l'aide du *slicing*.

3.2.1 Bornes dans un for

Dans cette première solution, je donnerai le code complet puisque nous passons à un affichage sur écran de leds et que nous devons donc « mixer » le code de nos deux premiers programmes :

```
01: from PIL import Image, ImageFont, ImageDraw
02: import numpy as np
03: from vrtneopixel import *
04: import time
05:
06: # Variables de configuration de l'écran
07: LED_COUNT = 64
08: LED_PIN = 18
09: LED_FREQ_HZ = 800000
10: LED_DMA = 5
11: LED_BRIGHTNESS = 8
12: LED_INVERT = False
13:
14: def clearScreen(leds):
15:     for i in range(64):
16:         leds[i] = Color(0, 0, 0)
17:
18: def displayScreen(strip, leds):
19:     for i in range(64):
20:         strip.setPixelColor(i, leds[i])
21:     strip.show()
22:
23: def text2image(text, fontName='Hack-Regular.ttf', pt=11,
saveName=None):
24:     font = ImageFont.truetype(fontName, pt)
25:     (w, h) = font.getsize(text)
26:     image = Image.new('L', (w, h), 1)
27:     draw = ImageDraw.Draw(image)
28:     draw.text((0, 0), text, font=font)
29:     if not saveName is None:
30:         image.save(saveName)
31:     return image
32:
33: def image2array(image):
34:     arr = np.asarray(image)
35:     arr = np.where(arr, 0, 1)
36:     arr = arr[(arr != 0).any(axis=1)]
37:     arr = np.concatenate((np.zeros((8, 7)), arr), axis=1)
38:     arr = np.concatenate((arr, np.zeros((8, 7))), axis=1)
39:     return arr
40:
41: def draw_letter(leds, array, size, num, color):
42:     y = 0
43:     for row in range(size[0]):
44:         x = 0
45:         for col in range(size[1] * (num - 1), size[1] *
num):
46:             if array[row, col] == 1:
47:                 leds[x + y * 8] = color
48:             else:
```

```
49:                     leds[x + y * 8] =
Color(0, 0, 0)
50:                 x += 1
51:             y += 1
52:         return leds
53:
54:
55: if __name__ == '__main__':
56:     # Initialisation des leds en noir
57:     leds = [Color(0, 0, 0)] * 64
58:
59:     # Création d'un élément
permettant de "manipuler"
60:     # l'écran de leds
61:     strip = Adafruit_NeoPixel(LED_
COUNT, LED_PIN, LED_FREQ_HZ, LED_DMA,
LED_INVERT, LED_BRIGHTNESS)
62:
63:     # Initialisation de l'écran
64:     strip.begin()
65:
66:     image = text2image('LINUX
MAGAZINE')
67:     arr = image2array(image)
68:
69:     for i in range(int(np.shape(arr)
[1] / 7)):
70:         leds = draw_letter(leds, arr,
(8, 7), i + 1, Color(255, 0, 0))
71:         # Affichage de l'écran
72:         displayScreen(strip, leds)
73:         # Attente de 1s
74:         time.sleep(1)
75:
76:     # Effacement de l'écran
77:     clearScreen(leds)
78:     displayScreen(strip, leds)
```

Dans la fonction **draw_letter()** des lignes 41 à 52, on voit bien le parcours par lignes et par colonnes avec déplacement progressif des colonnes vers la droite de la chaîne de caractères. Toutefois, **col** correspondant au numéro de colonne dans le tableau **array**, nous devrons utiliser une variable **x** pour rester dans l'intervalle **[0, 7]** (et une variable **y** pour rester cohérent dans la notation). En ligne 69, on boucle sur le nombre de caractères de la chaîne (caractères sur 7 pixels de long dans le tableau **arr**) et on appelle **draw_letter()** en ligne 70 pour mettre à jour **leds** et déclencher l'affichage en ligne 70.

3.2.2 Slicing

En appliquant le *slicing* dans la fonction **draw_letter()**, nous pouvons économiser les deux variables **x** et **y** :

```
...
41: def draw_letter(leds, array, size, num, color):
42:     array = array[:, size[1] * (num - 1) : size[1] * num]
43:     for row in range(size[0]):
44:         for col in range(size[1]):
45:             if array[row, col] == 1:
46:                 leds[col + row * 8] = color
47:             else:
48:                 leds[col + row * 8] = Color(0, 0, 0)
49:     return leds
...
```

Pour rappel, l'écriture **array[:, size[1] * (num - 1) : size[1] * num]** signifie que nous récupérons toutes les lignes du tableau **array** et que nous ne prenons que les colonnes ayant un indice compris entre **size[1] * (num - 1)** et **size[1] * num**.

3.3 Scrolling horizontal

Pour le scrolling horizontal, nous allons adapter la fonction **draw_letter()** de manière à spécifier une fenêtre quelconque sur la chaîne de caractères puis nous allons définir la fonction de *scroll* proprement dite :

```
...
51: def draw_part(leds, array, start, stop, color):
52:     array = array[:, start : stop]
53:     for row in range(8):
54:         for col in range(8):
55:             if array[row, col] == 1:
56:                 leds[col + row * 8] = color
57:             else:
58:                 leds[col + row * 8] = Color(0, 0, 0)
59:     return leds
60:
61: def scroll_h(strip, leds, text, color):
62:     image = text2image(text)
63:     arr = image2array(image)
64:
65:     for start in range(0, np.shape(arr)[1] - 7):
66:         leds = draw_part(leds, arr, start, start +
8, color)
67:         # Affichage de l'écran
68:         displayScreen(strip, leds)
69:         # Attente de 0,2s
70:         time.sleep(0.2)
...
```

On voit bien que **draw_part()** des lignes 51 à 59 est adaptée de **draw_letter()** avec la modification portant sur la section de tableau à traiter à l'aide des paramètres **start** et **stop** (ligne 52) et le parcours du tableau résultant qui est forcément de taille 8 x 8 (lignes 53 et 54). Pour le *scroll* proprement dit (lignes 61 à 70), il suffit de parcourir le tableau représentant les leds de la chaîne de caractères avec un pas d'une led (ligne 65). La fonction **shape()** **[8]** de Numpy (toujours ligne 65) permet de récupérer la longueur du tableau (elle renvoie un tuple composé du nombre de lignes et du nombre de colonnes). L'appel à la fonction de *scroll* se fera dans le programme principal sous la forme :

```
scroll_h(strip, leds, 'LINUX MAGAZINE',
Color(255, 0, 0))
```

3.4 Et pour un scrolling vertical ?

Pour un scrolling vertical, notre « bandeau » de lettres n'est pas dans la bonne orientation. Comme notre écran fait 8 pixels de largeur (et les caractères en faisant 7, ils pourront parfaitement être affichés), nous allons créer un tableau de **n x 8** pixels où **n** correspond au nombre de lettres multiplié par 8 (puisque chaque lettre occupe 8 lignes).

```
...
072: def get_letter(array, size, num):
073:     letter = array[:, size[1] * (num - 1) :
size[1] * num]
074:     return letter
075:
076: def verticalize(array, size):
077:     new_array = np.zeros((size[1],
size[0]))
078:     for i in range(int(np.shape(array)
[1] / 7)):
079:         print(i, ' ', get_letter(array,
(8, 7), i + 1)) # Vérification
080:         new_array[i * 8 : (i + 1) * 8,
0:7] = get_letter(array, (8, 7), i + 1)
081:     return new_array
082:
083: def draw_part_v(leds, array, start,
stop, color):
084:     array = array[start:stop, :]
085:     for row in range(8):
086:         for col in range(8):
087:             if array[row, col] == 1:
088:                 leds[col + row * 8] =
color
089:             else:
090:                 leds[col + row * 8] =
Color(0, 0, 0)
091:     return leds
092:
093: def scroll_v(strip, leds, text, color):
094:     image = text2image(text)
095:     arr = image2array(image)
096:     arr = verticalize(arr, (np.
shape(arr)[0], (len(text) + 2) * 8))
097:
098:     for start in range(0, np.shape(arr)
[0] - 8):
099:         leds = draw_part_v(leds, arr,
start, start + 8, color)
100:         # Affichage de l'écran
101:         displayScreen(strip, leds)
102:         # Attente de 0,2s
103:         time.sleep(0.2)
...
```

La fonction **letter()** des lignes 72 à 74 permet d'extraire un sous-tableau de **array** correspondant à la lettre en position **num**. La fonction **verticalize()** des lignes 76 à 81 se charge ensuite de représenter le tableau de lettres « verticalement ». Pour cela, on crée un nouveau tableau **new_array** que l'on remplit de zéros (ligne 77). La taille de ce tableau est spécifiée par le tuple **size** passé en paramètre. Ensuite on parcourt le tableau **array** autant de fois qu'il contient de lettres (ligne 78 avec encore une utilisation de **shape()**), puis on insère chaque lettre dans le tableau **new_array** (ligne 80). L'écriture **new_array[i * 8 : (i + 1) * 8, 0:7]** permet de spécifier un sous-tableau dont l'ensemble des valeurs sera écrasé par les valeurs du tableau renvoyé par **get_letter(array, (8, 7), i + 1)** (bien entendu les deux tableaux doivent être strictement de même dimension). Pour y voir plus clair, voici un exemple en shell Python :

```
$ python3
>>> import numpy as np
>>> array = np.ones((2, 3))
>>> array
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> array[0:2, 0:2] = np.array([[2, 3],
[3, 4]])
>>> array
array([[ 2.,  3.,  1.],
       [ 3.,  4.,  1.]])
```

Nous passons ensuite à la fonction **draw_part_v()** des lignes 83 à 91. Par rapport à **draw_part()**, seule la ligne 84 change pour le découpage du tableau ; il y aura peut-être un travail de refactorisation à effectuer par la suite... Enfin, **scroll_v()** des lignes 93 à 103 ressemble quant à lui énormément à **scroll_h()** : on a simplement ajouté la « verticalisation » en ligne 96, changé la borne de la boucle **for** en ligne 98, et appelé **draw_part_v()** en lieu et place de **draw_part()**.

L'appel se fait comme pour **scroll_h()** :

```
scroll_v(strip, leds, 'LINUX MAGAZINE',
Color(255, 0, 0))
```

Mais en fait ça ne marche pas si bien que cela, car les lettres sont collées les unes aux autres : il faut ajouter après chaque lettre une ligne de pixels vide. Il faut donc apporter les modifications suivantes à notre code :

```
...
072: def get_letter(array, size, num):
073:     letter = array[:, size[1] * (num - 1) :
size[1] * num]
074:     letter = np.concatenate((letter, np.zeros((1,
7))), axis=0)
075:     return letter
076:
077: def verticalize(array, size):
078:     new_array = np.zeros((size[1], size[0]))
079:     for i in range(int(np.shape(array)[1] / 7)):
080:         print(i, ' ', get_letter(array, (8, 7),
i + 1))
081:         new_array[i * 9 : (i + 1) * 9, 0:7] =
get_letter(array, (8, 7), i + 1)
082:     return new_array
...
093: def scroll_v(strip, leds, text, color):
094:     image = text2image(text)
095:     arr = image2array(image)
096:     arr = verticalize(arr, (np.shape(arr)[0],
(len(text) + 2) * 8 + len(text) + 2))
...
```

En ligne 74, on retrouve la concaténation de tableaux que nous avions déjà utilisé. Cette fois-ci nous concaténons une ligne de zéros (création du vecteur par **np.zeros((1, 7))** et choix des lignes (première dimension) par **axis=0**). Comme nous avons ajouté une ligne, nous avons modifié les dimensions du tableau retourné et il faut donc modifier la ligne 81 pour agrandir la zone de **new_array** impactée par l'ajout de données. Enfin, en ligne 96, lorsque nous indiquons la taille du nouveau tableau à créer, il faut ajouter **len(text) + 2** lignes qui correspondent aux lignes ajoutées après chaque caractère.

Cette fois-ci nous obtenons un résultat satisfaisant.

4. AJOUTER DES EFFETS DE COULEUR

Pour ajouter des effets de couleur sur le texte, je vous propose une amélioration très simple de notre code consistant à ajouter un filtre : une liste de 64 éléments (correspondant à 64 leds) et contenant des codes couleur. Lorsqu'un pixel sera allumé, il prendra la couleur indiquée par le filtre.

Voici les modifications portées sur le *scrolling* horizontal :

```
...
051: def draw_part(leds, array, start, stop, color,
filter_effect=None):
052:     array = array[:, start : stop]
053:     for row in range(8):
054:         for col in range(8):
055:             if array[row, col] == 1:
056:                 if filter_effect is None:
057:                     leds[col + row * 8] = color
058:                 else:
059:                     leds[col + row * 8] = filter_
effect[col + row * 8]
060:             else:
061:                 leds[col + row * 8] = Color(0, 0, 0)
062:     return leds
063:
064: def scroll_h(strip, leds, text, color, filter_
effect=None):
065:     image = text2image(text)
066:     arr = image2array(image)
067:
068:     for start in range(0, np.shape(arr)[1] - 7):
069:         leds = draw_part(leds, arr, start, start +
8, color, filter_effect)
070:         # Affichage de l'écran
071:         displayScreen(strip, leds)
072:         # Attente de 0,2s
073:         time.sleep(0.2)
...
```

```
109: if __name__ == '__main__':
110:     # Initialisation des leds
en noir
111:     leds = [Color(0, 0, 0)] * 64
112:
113:     # Initialisation du filtre
d'effet
114:     filter_effect = [Color(120 +
2 * i, i, i) for i in range(64)]
115:
116:     # Création d'un élément
permettant de "manipuler"
117:     # l'écran de leds
118:     strip = Adafruit_
NeoPixel(LED_COUNT, LED_PIN, LED_
FREQ_HZ, LED_DMA, LED_INVERT, LED_
BRIGHTNESS)
119:
120:     # Initialisation de l'écran
121:     strip.begin()
122:
123:     scroll_h(strip, leds, 'LINUX
MAGAZINE', Color(255, 0, 0), filter_
effect)
...
```

Pour que le code puisse fonctionner sans filtre, le paramètre **filter_effect** est mis à **None** par défaut (lignes 51 et 62). Dans les lignes 56 à 59, si le filtre est actif alors on utilise sa couleur pour éclairer la led courante, sinon on utilise la couleur **color** passée en paramètre. Ici le filtre employé est très simple, il s'agit d'une variation progressive autour du rouge (ligne 114). Le résultat (sans le *scrolling*...) est présenté en figure 6.

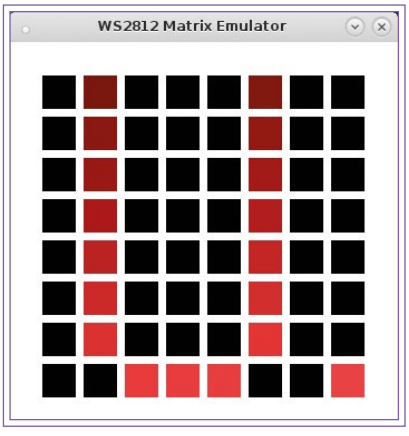


Fig. 6 : Exemple de lettre sur laquelle on a appliqué un filtre.

CONCLUSION

Je n'alourdirai pas inutilement cet article avec une version un peu plus épurée du code, mais il faut être conscient que quelques modifications seraient nécessaires :

- ce code est adapté aux écrans 8 x 8 leds, mais on pourrait le généraliser en rendant la taille paramétrable ;
- quelques fonctions peuvent être refactorisées de manière à diminuer la taille du code et à en simplifier la maintenabilité ;
- la création des filtres pourrait être facilitée : il faudrait sérialiser un tableau de leds pour en avoir une meilleure perception ;
- et bien d'autres améliorations...

Utiliser un écran de leds nous a obligés à manipuler des tableaux Numpy et à découvrir toute la souplesse de ces derniers. Nous avons également pu voir combien le module Pillow pouvait être utile pour convertir des chaînes de caractères en liste de pixels. Bref, merci Python... ;-) ■

RÉFÉRENCES

[1] COLOMBO T., *« Conception d'un émulateur de leds WS2812 »*, *GNU/Linux Magazine n°202*, mars 2017, p. 68 à 75 : http://connect.ed-diamond.com/GNU-Linux-Magazine/GLMF-202/Conception-d-un-emulateur-de-NeoPixels.

[2] Les modes disponibles : http://pillow.readthedocs.io/en/3.4.x/handbook/concepts.html#concept-modes

[3] La méthode Image.new() : http://pillow.readthedocs.io/en/3.4.x/reference/Image.html#PIL.Image.new

[4] Objet ImageDraw : http://pillow.readthedocs.io/en/3.1.x/reference/ImageDraw.html

[5] where() de Numpy : https://docs.scipy.org/doc/numpy/reference/generated/numpy.where.html

[6] any() de Numpy : https://docs.scipy.org/doc/numpy/reference/generated/numpy.any.html

[7] concatenate() de Numpy : https://docs.scipy.org/doc/numpy/reference/generated/numpy.concatenate.html

[8] shape() de Numpy : https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.shape.html

RMLL

SAINT-ETIENNE 2017

Rencontres Mondiales
du Logiciel Libre

LIBRE *ET* CHANGE

du 1er au 7 juillet 2017

Ateliers/Conférences
Village associatif
Salon professionnel
Concerts/Animations

2017.rmll.info

Saint-Étienne, L'expérience design
Saint-Etienne métropole
Loire Le Département
La Région Auvergne-Rhône-Alpes
Université de Lyon
Université Jean Monnet Saint-Étienne
Jean Louis Gagnaire, Député de la Loire, Réserve parlementaire
Adista

DÉVELOPPEMENT RAPIDE D'APPLICATIONS GTK+ AVEC GLADE

CHRISTOPHE BORELLY
[Professeur de l'ENSAM – IUT de Béziers]

MOTS-CLÉS : GLADE, RAD (RAPID APPLICATION DEVELOPMENT), XML, GTK+, PROGRAMMATION C

Glade [3] est un outil de développement rapide d'applications GTK+ et il permet de réaliser graphiquement à la souris l'essentiel de l'interface utilisateur. Je vais donc vous présenter son fonctionnement et ce qu'il apporte au niveau programmation.

Si vous avez suivi les épisodes précédents **[1][2]** présentant les bases de la programmation en **C** d'une application graphique avec la librairie **GTK+**, vous vous êtes sans doute aperçu que cela peut devenir un peu fastidieux parfois et qu'il fallait utiliser 5, 6 voire 10 lignes pour paramétrer un élément graphique. Bien sûr il y a toujours la possibilité d'écrire des fonctions génériques ou bien des macros, mais cela prend également un peu de temps pour faire les choses comme il faut. Une autre alternative quand elle existe (et là ça tombe bien, c'est le cas !) est d'utiliser un outil qui va générer le paramétrage que l'on a besoin.

Vous allez donc découvrir dans ce qui suit, le rôle et l'intérêt de **Glade** qui existe malgré tout depuis 1998.

Au moment où j'écris ces lignes, la dernière version de glade est la 3.20.0 du 22 mars 2016 et nécessite au moins la version 3.20 de GTK+. Mais il faut savoir que si vous voulez pouvoir utiliser les *widgets* de GTK+3, il vous faudra une version de glade supérieure à la version 3.10. Par exemple, avec **Mint** 17.3, il n'y a pas de problèmes, car la version de glade est la 3.16.1. Ce qui donne accès aux éléments graphiques des versions inférieures ou égales à GTK+ 3.10.

Un autre exemple sur mon système **Slackware** 64 « current », la version de `glade` est la 3.8.5 et je n'ai accès qu'aux éléments des versions inférieures à GTK+ 2.24 (voir menu préférences ou **<Ctrl>** + **<P>**). Seulement ce système comporte bien une version de GTK+3 (la 3.18.9 pour être précis). Il m'a donc fallu recompiler glade en version 3.19.0 (l'avant dernière mouture, pour éviter d'avoir aussi GTK+3 à mettre à jour) pour obtenir le support de GTK+3 jusqu'à la version 3.16.

NOTE

Tous les exemples de cet article sont disponibles sur GitHub. Vous trouverez donc pour chaque section un fichier cbXX-y.glade ainsi que le programme associé glade-XX-y.c. Seulement à partir de la section 3, comme on commence à programmer l'application finale, le fichier tictactoe.glade ne change pas et les différentes évolutions du code source sont dénommées tictactoe-y.c pour la section 3.y.

1. PREMIÈRE UTILISATION DE GLADE

Pour lancer **glade**, il suffit de taper la commande dans une console, ou bien d'utiliser le menu de lancement des applications de votre système. Vous devez alors obtenir une fenêtre subdivisée en plusieurs parties avec sur la gauche des boites d'outils (voir figure 1) : ***Actions***, ***Niveaux supérieurs***, ***Conteneurs***, ***Contrôle*** et ***Affichage***, puis sur la droite : ***Recherche de composants*** et ***Propriétés***.

Ensuite, tout va se passer avec la souris au centre de la fenêtre. Pour commencer, on peut créer la fenêtre de l'application (**GtkApplicationWindow**) en cliquant sur le troisième élément de la zone ***Niveaux supérieurs*** (voir figure 1).

Vous pourrez alors modifier la taille de la fenêtre à la souris et tout un tas de propriétés dans la zone en bas à droite. Par exemple, on peut modifier l'identifiant de la fenêtre qui par défaut est **applicationwindow1** (j'ai choisi de le raccourcir à **window1** dans cet exemple).

Si vous enregistrez votre travail, cela créera un fichier texte au format XML avec l'extension **.glade**. Voilà ce que cela donne :

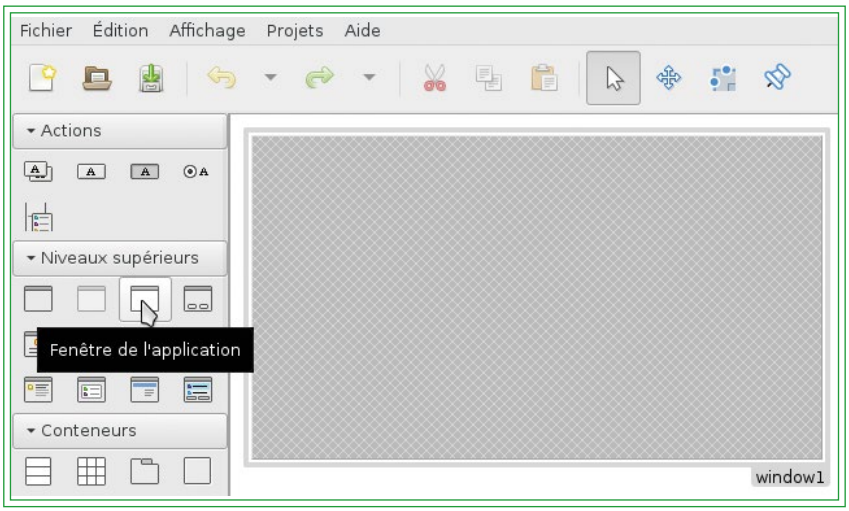


Fig. 1 : Ajout d'une fenêtre dans l'interface de glade.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Generated with glade 3.19.0 -->
<interface>
  <requires lib="gtk+" version="3.10"/>
  <object class="GtkApplicationWindow" id="window1">
    <property name="can_focus">False</property>
    <child>
      <placeholder/>
    </child>
  </object>
</interface>
```

C'est assez facile à lire et on devine facilement l'organisation et les propriétés de chaque élément.

On peut dès à présent tester cet embryon d'interface graphique avec un petit bout de code en C. Il faut se servir de **GtkBuilder** pour pouvoir utiliser le fichier XML généré par glade. On charge donc le fichier précédent à la ligne 7 (fonction disponible depuis GTK+ version 3.10, car avant il fallait utiliser 2 fonctions : **gtk_builder_new()** et **gtk_builder_add_from_file()**) puis on recherche l'identifiant **window1** pour obtenir la fenêtre à la ligne 9. Ensuite, on ajoute cette fenêtre à l'application et on affiche tous ses composants (lignes 11 et 12).

```
01:  #include <gtk/gtk.h>
02:
...
06:  static void startApplication(GtkApplication* app,gpointer data) {
07:    GtkBuilder *builder=gtk_builder_new_from_file("cb01.glade");
08:    if (builder!=NULL) {
09:      GtkWidget *window=(GtkWidget *)gtk_builder_get_object(builder,
"window1");
10:      if (window!=NULL) {
11:        gtk_application_add_window(app,GTK_WINDOW(window));
12:        gtk_widget_show_all(window);
13:      }
```

```
14:    }
15:  }
...
19:  int main(int argc,char *argv[]) {
20:    GtkApplication *app=gtk_application_new("fr.iutbeziers.glade-01",
21:                                            G_APPLICATION_FLAGS_NONE);
22:    g_signal_connect(app,"activate",G_CALLBACK(startApplication),NULL);
23:    int status=g_application_run(G_APPLICATION(app),argc,argv);
24:    g_object_unref(app);
25:    return status;
26:  }
```

Pour compiler rapidement les exemples de cet article, je vous propose d'utiliser **cmake** avec le fichier **CMakeLists.txt** suivant qui permet de créer un exécutable pour chaque fichier source C (voir lignes 9 à 13).

```
01:  project(cb-glade C)
02:  cmake_minimum_required(VERSION 2.6)
03:  find_package(PkgConfig REQUIRED)
04:  pkg_check_modules(GTK3 REQUIRED gtk+-3.0>=3.10)
05:  include_directories(${GTK3_INCLUDE_DIRS})
06:  link_directories(${GTK3_LIBRARY_DIRS})
07:  add_definitions(${GTK3_CFLAGS_OTHER})
08:  file( GLOB SRC RELATIVE "${CMAKE_SOURCE_DIR}" "${CMAKE_
SOURCE_DIR}/*.c" )
09:  foreach( cSource ${SRC} )
10:    string( REPLACE ".c" "" prgName ${cSource} )
11:    add_executable(${prgName} ${cSource})
12:    target_link_libraries(${prgName} ${GTK3_LIBRARIES})
13:  endforeach( cSource )
```

Comme les programmes devront utiliser des fichiers du répertoire courant, on utilise **cmake** en mode « in-source ». On tape donc dans une console :

```
$ cmake .
-- The C compiler identification is GNU 5.3.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Found PkgConfig: /usr/bin/pkg-config (found version "0.29.1")
-- Checking for module 'gtk+-3.0'
--   Found gtk+-3.0, version 3.18.9
-- Configuring done
-- Generating done
-- Build files have been written to: /home/cb/articles/cb-glade-
tictactoe
$ make
...
$ ./glade-01
```

Cela donne une petite fenêtre vide en haut de l'écran. Nous apprendrons dans la section suivante comment changer cela en modifiant les propriétés de la fenêtre.

Maintenant, que nous avons un petit aperçu de l'ensemble du processus de développement d'une application GTK+ en utilisant glade, il reste à ajouter et paramétrer tous les éléments graphiques désirés et à finaliser le fonctionnement des actions utilisateur dans le programme.

2. AJOUT DE WIDGETS DANS LA FENÊTRE

Afin de servir d'exemple, je vous propose donc de réaliser une interface permettant de jouer au jeu « tic-tac-toe » **[4]**. Le but du jeu étant d'aligner 3 symboles sur une grille de 9 cases.

Nous ajouterons donc à la fenêtre : une grille de 9 boutons, une barre d'état, un menu textuel classique, une barre d'outils et une boite d'informations « à propos » (voir figure 2).

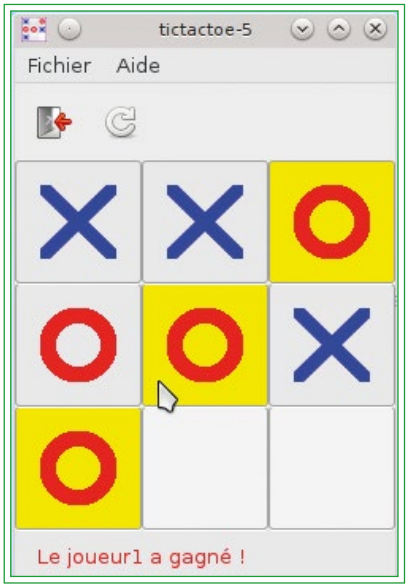


Fig. 2 : Aperçu de l'application finale.

2.1 Paramétrage de la fenêtre

Comme nous l'avons vu précédemment, la fenêtre s'affiche par défaut en haut de l'écran. Si l'on veut changer cela, on peut modifier les attributs de position et de gravité et les fixer à la valeur **Centre** et dans la partie ***Apparence***, un peu plus bas, indiquer une largeur et une hauteur par défaut.

On peut aussi définir une icône particulière pour l'application en utilisant la

case « nom de l'icône », laissant le système chercher la meilleure résolution dans les sous-répertoires de **/usr/share/icons/hicolor/**, mais il faut y avoir préalablement copié les icônes de l'application (voir le **Makefile** du répertoire **icons** fourni).

Au final, cela ajoute les 5 propriétés suivantes à l'objet **window1** dans le fichier XML.

```
<property name="window_position">center</property>
<property name="default_width">200</property>
<property name="default_height">200</property>
<property name="icon_name">tic-tac-toe</property>
<property name="gravity">center</property>
```

2.2 Ajout des éléments graphiques principaux

La toute première chose à faire pour étoffer l'application est d'ajouter un conteneur à la fenêtre. J'ai choisi comme base, l'élément boite (**GtkBox**) de la partie conteneurs. J'ai ensuite indiqué qu'il y aurait 4 éléments dans cette boite :

- le menu (**GtkMenuBar**), la 8ème icône de la partie conteneurs ;
- la barre d'outils (**GtkToolBar**), la 9ème icône de la partie conteneurs ;
- la grille de jeu (**GtkGrid**), la 2ème icône de la partie conteneurs ;
- la barre d'état (**GtkStatusBar**), la 31ème icône de la partie contrôle et affichage.

Cela donne l'arborescence de la figure 3.

On peut ensuite ajouter les 9 boutons (1ère icône de la partie contrôle et affichage) dans la grille.

À l'exécution du programme, on obtient la figure 4 avec déjà pas mal d'éléments graphiques ajoutés. La barre d'outils est vide pour l'instant, vous verrez au paragraphe 2.5 comment ajouter des outils à cette barre.

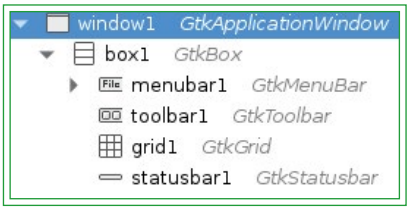


Fig. 3 : Arborescence du contenu de la fenêtre de l'application.

Fig. 4 : Premier contenu de la fenêtre de l'application.

2.3 Améliorations visuelles

Si l'on veut que la taille des boutons s'adapte à la taille de la fenêtre, il faut cocher la case ***homogène*** dans les propriétés des colonnes et des lignes de la grille, puis activer le mode ***étendre*** dans la partie agencement. On peut également fixer une taille « préférée » de 240 par 240 pixels pour la grille (dans la partie commune), ce qui permet de limiter la taille minimale des boutons à 80x80.

Le label par défaut des boutons (***button***) peut aussi être supprimé, car il n'y a pas besoin d'utiliser du texte dans les cases de la grille de jeu.

Au niveau du fichier XML généré par glade, on voit donc apparaître les balises **property** suivantes :

```
<object class="GtkGrid" id="grid1">
  <property name="width_request">240</property>
  <property name="height_request">240</property>
  <property name="visible">True</property>
  <property name="can_focus">False</property>
  <property name="row_homogeneous">True</property>
  <property name="column_homogeneous">True</property>
  <child>
    <object class="GtkButton" id="button1">
      <property name="visible">True</property>
      <property name="can_focus">True</property>
      <property name="receives_default">True</property>
    </object>
    ...
  </child>
  ...
</object>
<packing>
  <property name="expand">True</property>
  <property name="fill">True</property>
  <property name="position">2</property>
</packing>
```

2.4 Gestion des actions de l'utilisateur

Dans le jeu, l'utilisateur va devoir interagir avec l'application de deux façons différentes.

La première interaction consiste à sélectionner une case libre de la grille tout simplement en cliquant dessus (gestion des clics sur un bouton). Pour cela, il suffit de modifier la partie signaux de chaque bouton et de donner un nom

de fonction à exécuter pour le signal **clicked** (j'ai choisi **on_button_clicked** pour les 9 boutons).

Il faut noter que cela ajoute la balise signal à chaque bouton dans le fichier XML créé par glade :

```
<object class="GtkButton" id="button1">
  ...
  <signal name="clicked" handler="on_
button_clicked" swapped="no"/>
</object>
```

Ensuite, il faut permettre à l'utilisateur de recommencer une partie, de terminer l'application ou bien d'afficher la boite d'informations. Pour cela, nous allons définir un groupe d'actions de type **GtkActionGroup** (avec la première icône de la partie actions).

L'intérêt de définir des actions, va porter sur le fait de pouvoir y associer une icône et plus tard des raccourcis clavier (voir paragraphe 2.6). Il est important ici, pour ne pas avoir de messages d'erreurs à l'exécution, d'ajouter également un groupe de raccourcis de type **GtkAccelGroup** dans l'interface (icône 11 de la partie Divers ou clic sur le bouton « Nouveau » après avoir cliqué sur l'icône à la fin de la case groupe d'accélérateurs du groupe d'actions).

```
  <object class="GtkAccelGroup"
id="accelgroup1"/>
  <object class="GtkActionGroup"
id="actiongroup1">
    <property name="accel_
group">accelgroup1</property>
    ...
```

Pour ajouter des actions au groupe d'actions, il faut utiliser le menu contextuel ***Edit*** puis l'onglet ***Hiérarchie***. On fixe l'identifiant à **actionNewGame** pour la première action, puis on indique par exemple **gtk-refresh** pour l'icône prédéfinie. Si l'on veut anticiper la prochaine section, on peut remplir la case infobulle avec le texte « Nouvelle partie », puis cliquer sur la case « toujours afficher l'image ». Il reste enfin à spécifier la fonction **on_actionNewGame_activate** pour le signal **activate**. On réitère ce paramétrage pour l'action **actionAbout** en spécifiant cette fois-ci l'icône **gtk-about** et pour l'action **actionQuit** avec l'icône **gtk-quit**.

Il reste à associer ces actions aux éléments du menu, en profitant de l'occasion pour enlever les parties inutiles créées automatiquement par glade dans de ce dernier. J'ai choisi de ne garder que 2 menus (le **menuitem1** et le **menuitem4**) avec seulement 2 éléments dans la partie **menuitem1** (**imagemenuitem1** et **imagemenuitem5**).

Graphiquement, pour créer le lien logiciel entre l'élément de menu et l'action voulue, on sélectionne un élément **imagemenuitem**, puis on clique sur ***Action liée*** et on choisit l'action désirée.

Cela se concrétise ainsi au niveau du fichier XML :

```
<object class="GtkImageMenuItem" id="imagemenuitem1">
  ...
  <property name="related_action">actionNewGame</property>
</object>
...
<object class="GtkImageMenuItem" id="imagemenuitem5">
  ...
  <property name="related_action">actionQuit</property>
</object>
...
<object class="GtkImageMenuItem" id="imagemenuitem10">
  ...
  <property name="related_action">actionAbout</property>
</object>
```

Dans le code C, il suffit de définir les 4 fonctions (1 pour les boutons et 3 pour les actions) et d'appeler la fonction de connexion des signaux de **GtkBuilder** (ligne 21). On peut noter que le second paramètre fourni à cette fonction est aussi la variable **builder**, ce qui permet d'y avoir accès dans les 4 fonctions de rappel par l'intermédiaire du pointeur **data** (voir lignes 3, 6, 9 et 12).

```
03 : void on_button_clicked(GtkWidget* src,gpointer data) {
04 :   printf("on_button_clicked...\n");
05 : }
06 : void on_actionNewGame_activate(GtkAction* action,gpointer data) {
07 :   printf("on_actionNewGame_activate...\n");
08 : }
09 : void on_actionAbout_activate(GtkAction* action,gpointer data) {
10 :   printf("on_actionAbout_activate...\n");
11 : }
12 : void on_actionQuit_activate(GtkAction* action,gpointer data) {
13 :   printf("on_actionQuit_activate...\n");
14 : }
...
18 : static void startApplication(GtkApplication* app,gpointer data) {
19 :   GtkBuilder *builder=gtk_builder_new_from_file("cb02-4.glade");
20 :   if (builder!=NULL) {
21 :     gtk_builder_connect_signals(builder,builder);
...
```

À l'exécution du code **glade-02-4**, on remarque bien les icônes dans le menu (voir figure 5) et pour l'instant , suivant les actions réalisées par l'utilisateur, on ne voit que des messages s'afficher dans la console. Nous expliquerons plus tard, comment implémenter réellement les fonctionnalités désirées.

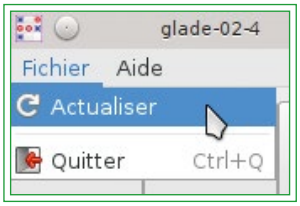


Fig. 5 : Menu intégrant les 3 actions définies dans glade.

Fig. 6 : Application avec 2 icônes dans la barre d'outils.

2.5 Ajout de boutons dans la barre d'outils

On va pouvoir maintenant ajouter des boutons à la barre d'outils en utilisant les actions créées précédemment.

Il faut d'abord sélectionner l'élément **toolbar1** puis avec le bouton droit de la souris aller dans le menu ***Edit*** et la partie ***Hiérarchie*** pour ajouter deux **GtkToolButton**. On indique comme précédemment l'action liée à chaque bouton.

À l'exécution, on voit bien apparaître les deux icônes avec l'infobulle associée à chaque action (voir figure 6).

NOTE

Si l'on veut voir apparaître un texte (le label) juste après l'icône d'une action dans la barre d'outils, il faut cocher « est important » dans les paramètres du GtkToolButton. On peut aussi utiliser le caractère _ dans le texte pour préciser le mnémonique associé au bouton (voir section 1 de l'article [2]).

2.6 Ajout de raccourcis clavier

Il est également possible avec glade de définir les raccourcis clavier utilisables pour chaque action. Vous avez peut-être déjà remarqué que pour le menu ***Quitter***, il s'affiche **<Ctrl> + <Q>** en fin de menu sur la figure 5, mais rien ne se passe jusqu'à présent si on appuie sur ces 2 touches du clavier.

Pour définir des raccourcis clavier personnalisés pour les actions de notre groupe d'actions, il suffit en fait de cliquer sur l'icône de la case ***Raccourci*** puis de cliquer sur la combinaison désirée. J'ai choisi **<Ctrl> + <N>** pour **actionNewGame**, pas de raccourci pour **actionAbout** et **<Ctrl> + <Q>** pour **actionQuit**, ce qui se traduit par **<Primary>N** et **<Primary>Q** dans le logiciel glade et par le modificateur **GDK_CONTROL_MASK** dans le fichier XML.

La dernière chose importante pour que les accélérateurs soient pris en compte par la fenêtre de l'application, est de définir le même groupe d'accélérateurs (**accelgroup1**) au niveau de la fenêtre **window1**.

On obtient donc un fichier XML avec le contenu suivant :

```
...
  <object class="GtkActionGroup" id="actiongroup1">
    ...
    <child>
      <object class="GtkAction" id="actionNewGame">
      ...
      </object>
      <accelerator key="n" modifiers="GDK_CONTROL_
MASK"/>
    </child>
    ...
    <child>
      <object class="GtkAction" id="actionQuit">
      ...
      </object>
      <accelerator key="q" modifiers="GDK_CONTROL_
MASK"/>
    </child>
  </object>
  <object class="GtkApplicationWindow"
id="window1">
  ...
    <accel-groups>
      <group name="accelgroup1"/>
    </accel-groups>
...
```

À l'exécution (**glade-02-6**), on voit bien apparaître les raccourcis dans le menu et cette fois-ci, ils fonctionnent !

2.7 Ajout de la boite d'informations

En utilisant l'icône 5 de la partie ***Niveaux supérieurs***, on peut ajouter un élément **GtkAboutDialog** qui permet de donner toutes les informations sur l'application, le nom des développeurs, le type de licence, etc. Pour cet exemple (voir figure 7), j'ai rempli les valeurs nom du programme, version, icône, URL, commentaires, licence et auteur. Cela crée les balises suivantes dans le fichier XML :

```
  <object class="GtkAboutDialog"
id="aboutdialog1">
    <property name="can_focus">False</property>
    <property name="type">popup</property>
    <property name="type_hint">dialog</
property>
```

```
    <property name="program_name">Tic-
Tac-Toe</property>
    <property name="version">Version
1.0</property>
    <property name="comments"
translatable="yes">Un petit jeu de
réflexion...</property>
    <property name="website">http://www.
borelly.net/</property>
    <property name="authors">Christophe
BORELLY</property>
    <property name="logo_icon_name">tic-
tac-toe</property>
    <property name="license_type">gpl-3-0
</property>
    ...
```

Même si on définit toutes ces propriétés, on ne peut pas encore visualiser la boite de dialogue dans notre application, car il faut ajouter tout de même un peu de code en C. Cela sera détaillé en section 3.5.

Fig. 7 : Aperçu de la boite d'informations.

3. PROGRAMMATION DES FONCTIONNALITÉS

L'interface graphique est maintenant finie (**tic-tac-toe.glade**), il ne reste plus qu'à programmer le détail des fonctions de rappel pour la gestion des clics sur les boutons et les différentes actions utilisateur (**Quit**, **NewGame** et **About**).

3.1 Programmation de l'action actionQuit

La documentation GTK+ indique que pour terminer une application GTK+3, il suffit d'enlever la fenêtre principale de l'application. On va donc se servir à la ligne 16, du paramètre **data** (comme expliqué précédemment) pour obtenir le **GtkBuilder** créé au début de la fonction **startApplication()**. À partir de là, on peut récupérer la fenêtre **window1** définie dans notre interface graphique (ligne 18) et ensuite obtenir l'application correspondante (ligne 20). Il ne reste enfin qu'à enlever la fenêtre de l'application (ligne 22). Cela donne le code suivant :

```
14:  void on_actionQuit_activate(GtkAction*
action,gpointer data) {
15:    printf("on_actionQuit_activate...\n");
16:    GtkBuilder *builder=GTK_BUILDER(data);
17:    if (builder!=NULL) {
18:      GtkWindow *window=(GtkWindow *)gtk_
builder_get_object(builder,"window1");
19:      if (window!=NULL) {
20:        GtkApplication *app=gtk_window_get_
application(window);
21:        if (app!=NULL) {
22:          gtk_application_remove_window(app,
window);
23:          printf("Bye bye !\n");
24:        }
25:      }
26:    }
27:  }
```

Vous devez maintenant pouvoir fermer le programme avec le premier bouton de la barre d'outils, ou bien avec le menu ***Fichier*** + ***Quitter*** ou encore avec le raccourci **<Ctrl>** + **<Q>**.

3.2 Clic sur les boutons de la grille de jeu

Dans le jeu de « tic-tac-toe » **[4]**, chaque joueur choisi chacun à son tour, une case disponible de la grille et marque cette case avec un symbole donné (par exemple, un cercle rouge pour le joueur 1 et une croix bleue pour le joueur 2).

Pour réaliser cela, nous utiliserons un style CSS différent pour chaque joueur à l'aide de 2 images de fond.

```
.joueur1 {
  background-image:url('j1.png');
  background-repeat:no-repeat;
  background-position:center;
}
.joueur2 {
  background-image:url('j2.png');
  background-repeat:no-repeat;
  background-position:center;
}
```

Nous allons donc charger le fichier de style à l'initialisation du programme à la ligne 95 avec la fonction **loadCSS()**. Puis lors d'un clic sur un bouton donné, nous ajouterons à celui-ci la classe **joueur1** ou **joueur2** en fonction du numéro de joueur (ligne 44) et nous désactiverons le bouton (ligne 41) pour qu'il ne puisse pas être utilisé par le joueur suivant.

Pour simplifier le programme, j'ai opté pour l'utilisation de trois variables globales (lignes 3 et 4). La première (**joueurId**) permettant de sauvegarder le numéro d'index du joueur en cours (**0** pour le joueur 1 et **1** pour le joueur 2), la deuxième (**nbCoup**) permettant de connaître le nombre de coups dans la partie et la dernière (**joueurs**) représentant un tableau de deux chaînes de caractères pour le nom des deux classes CSS.

On peut noter aussi l'utilisation de la fonction **displayJoueur()** à la ligne 48 pour afficher le nom du prochain joueur dans la barre d'état. Je vous laisse regarder le détail de cette fonction dans les sources.

Voici donc les parties importantes des adaptations du code :

```
03:  int joueurId=0,nbCoup=0;
04:  const gchar *joueurs[2]={"joueur1","joueur2"};
...
38:  void on_button_clicked(GtkWidget* src,gpointer data) {
39:    GtkBuilder *builder=GTK_BUILDER(data);
40:      if (builder!=NULL) {
41:        gtk_widget_set_sensitive(src,FALSE);
42:        GtkStyleContext *ctx=gtk_widget_get_
style_context(src);
43:        if (ctx!=NULL) {
44:          gtk_style_context_add_class(ctx,
joueurs[joueurId]);
45:        }
46:        nbCoup++;
47:        joueurId=(joueurId==0)?1:0; // Joueur
suivant
48:        displayJoueur(builder);
49:      }
50:   }
...
94:   static void startApplication(GtkApplicati
on* app,gpointer data) {
95:     loadCSS();
96:     GtkBuilder *builder=gtk_builder_new_from_
file("tictactoe.glade");
...
```

3.3 Programmation de la fin de partie

Voilà la partie algorithmique la plus intéressante de ce jeu.

Pour gagner, un joueur doit aligner 3 symboles en vertical, en horizontal ou en diagonale. Il y a donc 8 cas différents. La structure de donnée la plus adaptée à une grille de 3x3 cases est naturellement une matrice, mais par souci de simplicité et de rapidité, j'ai choisi d'utiliser un tableau de 9 cases représentant la grille en partant du point en haut à gauche et en allant à droite puis vers le bas. Ce qui donne la numérotation de la figure 8 pour les cases de la grille.

Donc pour gagner en horizontal, un joueur devra donc avoir coché les cases 0, 1 et 2 ou bien 3, 4 et 5 ou encore 6, 7 et 8. En vertical, cela donne 0, 3 et 6 ou bien 1, 4 et 7 ou 2, 5 et 8. Les deux diagonales sont 0, 4 et 8 et 2, 4 et 6.

J'ai alors choisi d'ajouter la fonction **verificationVictoire()** qui va renvoyer TRUE si un des 2 joueurs a gagné, mais il y a 2 sous-parties à réaliser :

- Créer un tableau temporaire de résultats appelé **data** (où l'on notera **1**, si la case en question a été cochée par le joueur 1, **2** pour le joueur 2 et **0** si la case n'est pas encore utilisée). Cette étape n'est pas très optimisée, car on la refait à chaque coup, mais cela était plus rapide à faire ainsi.
- Tester les différents cas de victoire que l'on vient d'énumérer précédemment (voir lignes de 70 à 85). Afin d'automatiser un peu cela, j'ai défini cette fois-ci, une matrice de 8 par 3 contenant tous les indices des cases gagnantes (voir lignes 48 à 50).

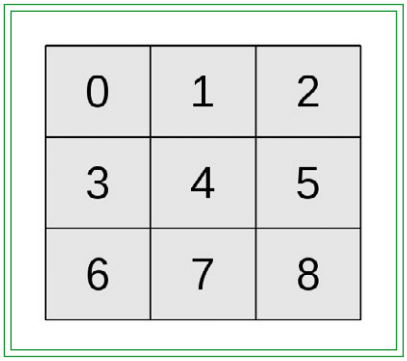


Fig. 8: Numérotation des cases de la grille.

Le parcours de la liste des « enfants » (les boutons) de la grille de jeu ne sera pas détaillé ici, mais je vous invite à aller voir dans les sources entre les lignes 54 et 68...

Ensuite, pour qu'un joueur soit donc déclaré vainqueur, il faut que les 3 cases à tester du tableau **data** soient égales et différentes de **0** (Le joueur 1 aura 3 cases avec des **1**, et le joueur 2, 3 cases avec des **2**). On obtient donc l'expression de la ligne 73, sachant que les indices **a**, **b** et **c** contiennent les numéros des 3 cases à tester (voir ligne 72).

Afin de clarifier un peu plus le code en cas de victoire, j'ai aussi créé 2 petites fonctions, **markWin()** et **disableAllButtons()**, permettant :

- d'ajouter la classe CSS **win** à un élément de la liste (ceci rendant plus visuelle la ligne réalisée, voir lignes de 74 à 76) ;
- de désactiver tous les boutons de la grille (très utile pour terminer la partie, voir ligne 77).

Dernière modification faite à la ligne 117 dans la fonction **on_button_clicked** (qui je le rappelle, s'exécute à chaque coup d'un joueur) concerne l'appel de la fonction de vérification que l'on vient d'écrire. Cela permet de tester à chaque tour si un des 2 joueurs a gagné et de plus, on ne change l'index **joueurId** que si le joueur précédent n'a pas gagné et donc l'affichage du nom du joueur des lignes 78 et 79 correspond bien au joueur en train de gagner (pas besoin de regarder le contenu des cases du tableau **data** pour savoir qui a vraiment gagné).

Voici donc le code final :

```
...
45:  int verificationVictoire(GtkBuilder *builder) {
46:    gboolean res=FALSE;
47:    int i,j;
48:    int win[8][3]={{0,1,2},{3,4,5},{6,7,8},// Horizontal
49:                   {0,3,6},{1,4,7},{2,5,8},// Vertical
50:                   {0,4,8},{2,4,6}};       // Diagonales
51:    if (builder!=NULL) {
...      // Création et remplissage du tableau temporaire data
69:        // Teste les différents cas de victoire
70:        for (i=0;i<8;i++) {
71:          // Indice des 3 cases à tester
72:          int a=win[i][0],b=win[i][1],c=win[i][2];
73:          if (data[a]!=0&&data[a]==data[b]&&data[b]==data[c]) {
74:            markWin(liste,a);
75:            markWin(liste,b);
76:            markWin(liste,c);
77:            disableAllButtons(liste);
78:            gchar *msg=g_markup_printf_escaped("Le %s a gagné !",
79:                                               joueurs[joueurId]);
80:            displayStatus(builder,msg,FALSE);
81:            g_free(msg);
82:            res=TRUE;
83:            break;
84:          }
85:        }
86:        if (liste!=NULL) g_list_free(liste);
87:      }
88:    }
89:    return res;
90:  }
...
109:  void on_button_clicked(GtkWidget* src,gpointer data) {
...
117:      if (verificationVictoire(builder)!=TRUE) {
...
```

3.4 Programmation de l'action actionNewGame

Dans cette partie, il va falloir fixer à zéro l'identifiant du numéro de joueur et le nombre de coups, puis afficher un message dans la barre d'état et enfin réinitialiser l'état de tous les boutons de la grille de jeu. Pour cette dernière opération, on fait une boucle sur tous les éléments de la liste comme précédemment (voir ligne 137) et on appelle les fonctions **gtk_widget_set_sensitive()** en fixant l'état à TRUE (ligne 138) pour activer le bouton et **gtk_style_context_remove_class()** pour effacer toutes les classes CSS (lignes 141 à 143).

On peut aussi appeler cette fonction au tout début du programme dans la fonction **startApplication()** pour bien initialiser la grille de jeu et la barre d'état. Il suffit de fixer à **NULL** le pointeur vers l'action à la ligne 199.

Cela donne le code suivant :

```
126:  void on_actionNewGame_activate(GtkAction*
action,gpointer data) {
...
134:      if (GTK_IS_CONTAINER(grid)) {
135:        GList *liste=gtk_container_get_
children(GTK_CONTAINER(grid));
136:        GList *child;
137:        for (child=liste;child!=NULL;child=
g_list_next(child)) {
138:          gtk_widget_set_sensitive(child->data,
TRUE);
139:          GtkStyleContext *ctx=gtk_widget_get_
style_context(child->data);
140:          if (ctx!=NULL) {
141:            gtk_style_context_remove_
class(ctx,"joueur1");
142:            gtk_style_context_remove_
class(ctx,"joueur2");
143:            gtk_style_context_remove_
class(ctx,"win");
144:          }
145:        }
146:        if (liste!=NULL) g_list_free(liste);
147:      }
...
190:  static void startApplication(GtkApplication*
app,gpointer data) {
...
198:        gtk_widget_show_all(window);
199:        on_actionNewGame_activate(NULL,builder);
...
```

3.5 Programmation de l'action actionAbout

La dernière chose à faire consiste à afficher la boite d'informations dans la fonction **on_actionAbout_activate()**. Il faut pour cela récupérer les widgets de la boite **aboutdialog1** et de la fenêtre **window1**, puis appeler la fonction **gtk_window_set_transient_for()** à la ligne 160 pour indiquer au système que la boite de dialogue doit être placée au-dessus de la fenêtre principale de l'application. On lance enfin la fonction **gtk_dialog_run()** qui va attendre que l'utilisateur ait cliqué sur le bouton « Fermer » avant de cacher la boite d'informations à la ligne 162.

```
154:  void on_actionAbout_activate(GtkAction*
action,gpointer data) {
155:    GtkBuilder *builder=GTK_BUILDER(data);
156:    if (builder!=NULL) {
157:      GtkWidget *about=(GtkWidget *)gtk_
builder_get_object(builder,"aboutdialog1");
158:      if (about!=NULL) {
159:        GtkWindow
*window=(GtkWindow *)gtk_builder_get_
object(builder,"window1");
160:        gtk_window_set_transient_
for(GTK_WINDOW(about),window);
161:        gtk_dialog_run(GTK_
DIALOG(about));
162:        gtk_widget_hide(about);
163:      }
164:    }
165 }
```

Voilà, maintenant lorsque l'on clique sur le menu « À propos », la boite de dialogue s'affiche à l'utilisateur.

CONCLUSION

Voilà terminé notre exemple de développement d'application GTK+ à l'aide de l'outil glade.

Il reste cependant à internationaliser l'application **[1]** avec l'outil **xgettext** et l'option **--keyword=translatable** pour pouvoir convertir les chaînes définies dans le fichier XML généré par glade.

Une autre évolution possible est de rendre l'exécutable « portable », car en effet, si on lance l'application depuis un autre répertoire, on obtient un joli message d'erreur indiquant que le fichier CSS et le fichier **.glade** ne peuvent pas être ouverts. Une solution à ce problème est d'intégrer plutôt directement dans le code source du programme, le contenu de ces 2 fichiers en se servant des fonctions **gtk_css_provider_load_from_data()** et **gtk_builder_new_from_string()** par exemple.

Je vous laisse essayer de tester tout cela à titre d'exercices... ■

RÉFÉRENCES

[1] BORELLY C., « *Programmer en GTK+* », *GNU/Linux Magazine n°194*, page 58

[2] BORELLY C., « *Des menus dans une application GTK+* », *GNU/Linux Magazine n°198*, page 68

[3] Site de Glade : https://glade.gnome.org/

[4] Page du jeu « tic-tac-toe » sur Wikipédia : https://fr.wikipedia.org/wiki/Tic-tac-toe

[5] COLOMBO T., « *Le Tic Tac Toe un jeu simple à développer ?* », *GNU/Linux Magazine n°196*, page 18

ACTUELLEMENT DISPONIBLE

GNU/LINUX MAGAZINE HORS-SÉRIE N°90 !

PROGRAMMATION RÉSEAU EN PYTHON

NE LE MANQUEZ PAS

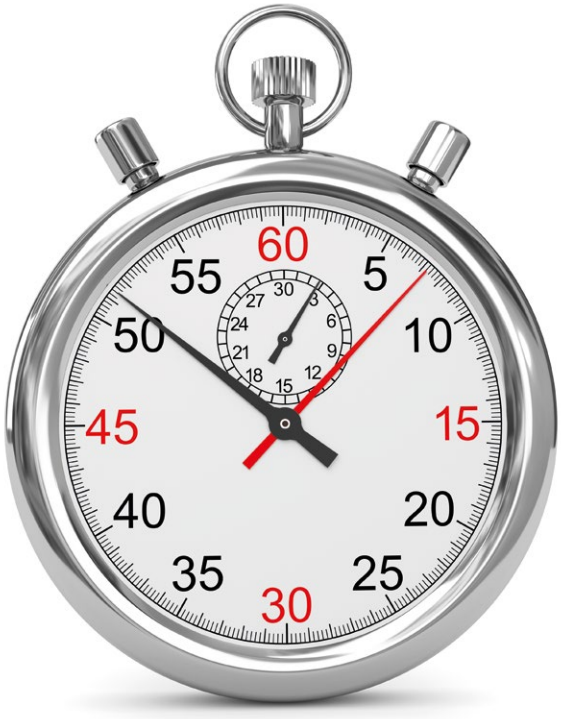
CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :

http://www.ed-diamond.com

DE BEAUX CHRONOGRAMMES AVEC WAVEDROM

FABIEN MARTEAU
[Responsable FPGA chez Armadeus Systems, membre actif du Front de libération des FPGA]

MOTS-CLÉS : CHRONOGRAMME, WAVEDROM, JAVASCRIPT, JSON



WaveDrom est un outil de dessin de chronogrammes écrit en JavaScript/HTML/CSS. Il permet de décrire ses chronogrammes avec une syntaxe simple en JSON. Une bibliothèque JavaScript va ensuite convertir la structure JSON en une image SVG affichable dans un navigateur. Il est également possible d'utiliser une application « hors-ligne » pour générer ses images en SVG, PNG, JPG, etc.

Le chronogramme est l'outil de prédilection de l'électronicien pour documenter le fonctionnement de son système numérique ou analogique. Un simple chronogramme vaut souvent mille discours et descriptions.

Hélas, le dessin de chronogrammes se révèle vite être une activité dévoreuse de temps, pour un rendu souvent discutable. Habituellement pour dessiner des chronogrammes on utilise **LibreOffice Draw**, **DIA** ou **Xfig** qui peuvent permettre de faire des rendus relativement corrects au prix d'un long effort de dessin. Certain se servent même d'un tableur en utilisant astucieusement l'option d'encadrement des cellules pour faire les créneaux de leurs signaux, d'autres abandonnent l'ordinateur pour le papier la règle et le stylo. Pour versionner, bonjour les dégâts...

Les formats de fichiers de tous ces logiciels étant du genre « binaire », le gestionnaire de version ne fera que réenregistrer le fichier à chaque commit, et il sera impossible de visualiser correctement les modifications effectuées.

Partant de ce constat, Aliaksei Chapyzhenka a cherché à développer un langage permettant de décrire ses chronogrammes au format texte qui soit convertible en une belle image. Pour convertir cette description

texte en image, plutôt que de créer une application *standalone*, qui aurait dû être portée sur chaque plateforme utilisée (**Linux**, **BSD**, **Haiku**, **Windows**, **Android**...), l'outil a été développé en **JavaScript**/**HTML**. Ce choix permet à quiconque qui possède un navigateur internet de tester **WaveDrom**.

1. MISE EN ROUTE

Commençons par un simple exemple de page web HTML, le code suivant décrit un « handshake » wishbone :

```
<html>
  <head>
    <script src="http://wavedrom.com/skins/
default.js" type="text/javascript"></script>
    <script src="http://wavedrom.com/wavedrom.
min.js" type="text/javascript"></script>
    <script>
       window.addEventListener('DOMContentLoa
ded', WaveDrom.ProcessAll, false);
    </script>
  </head>
  <body>

    <script type="WaveDrom">
      { signal: [
        { name: "CLK_I",  wave: "p..." },
        { name: "STB_O",  wave: "01.0"},
        { name: "ACK_I",  wave: "0.10" },
      ]}
    </script>

  </body>
</html>
```

La description du chronogramme en tant que telle se trouve entre les balises **<script type="WaveDrom">** et **</script>** ; elle se résume à une structure JSON. Chaque signal est muni d'un nom **name** et d'une description dans le temps du signal **wave**.

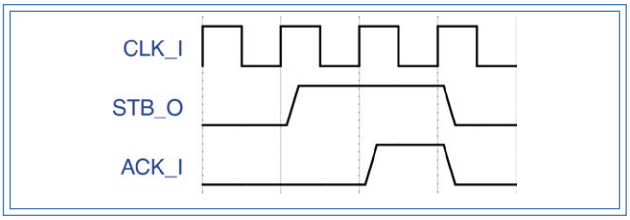


Fig. 1 : Un premier chronogramme.

Si l'on ouvre ce code dans un navigateur web (connecté à Internet pour récupérer les bibliothèques JavaScript WaveDrom), nous obtenons le joli chronogramme de la figure 1.

2. UN PEU DE SYNTAXE

Le lecteur trouvera une description très complète des possibilités offertes par la syntaxe WaveDrom sur la page Hitchhiker's Guide to the WaveDrom **[1]**. Chaque caractère de la chaîne « wave » représente une période de temps. Le symbole **.** étend l'état du signal à la période suivante.

```
{ signal: [{ name: "Données", wave: "01.
zx=ud.23.45" }] }
```

Le résultat de ce premier exemple est visible en figure 2.

Il est possible d'ajouter du texte dans les signaux « de bus » au moyen de l'entrée **data** :

```
{ signal: [{ name: "Données", wave: "01.
zx=ud.23.45", data:"un deux trois 4 5" }] }
```

La figure 3 montre le résultat obtenu dans ce deuxième exemple.

Les signaux « de bus » sont les signaux représentant plus d'un bit. Ces signaux sont décrits par les caractères **=2345** et ne peuvent pas être représentés par les seuls états **0** ou **1**, d'où la possibilité de leur ajouter un label donnant leurs valeurs.

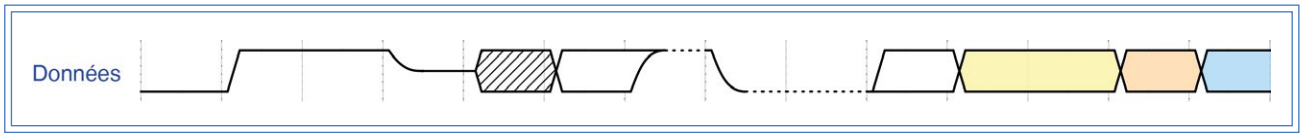


Fig. 2 : Premier exemple.

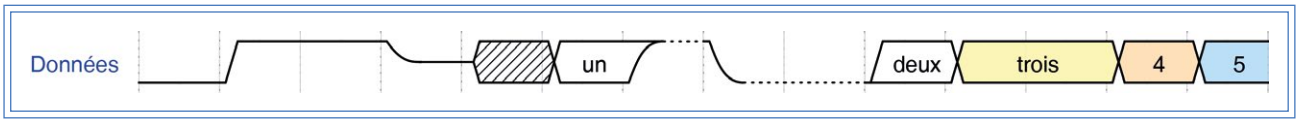


Fig. 3 : Deuxième exemple.

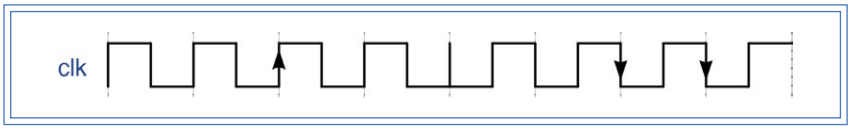


Fig. 4 : Troisième exemple.

Les signaux d'horloges changent d'état au cours d'une période de temps. C'est la raison pour laquelle il est prévu des symboles particuliers (voir figure 4) :

```
{ signal: [{ name: "clk", wave: "p.Ppn.N."}] }
```

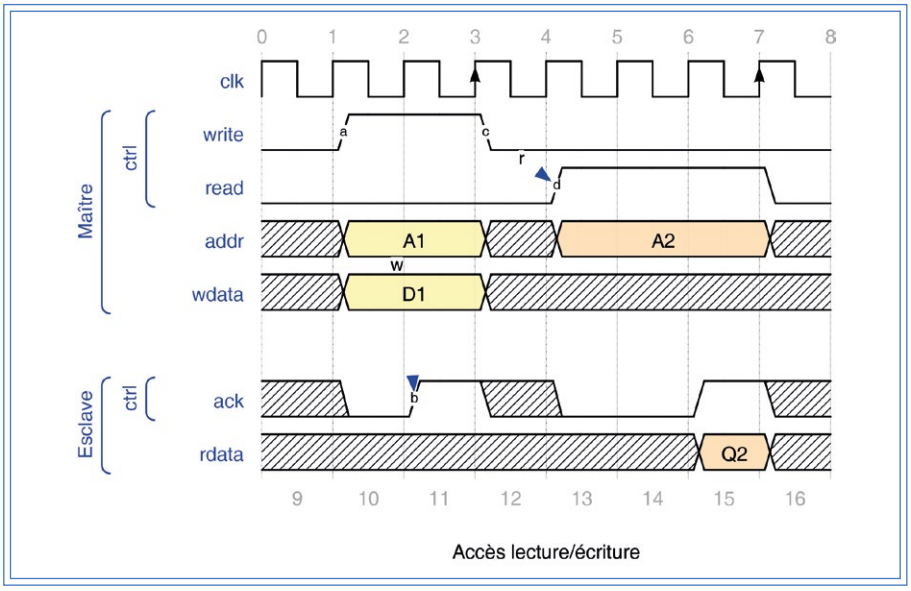


Fig. 5 : Exemple un peu plus complet de ce qu'il est possible de faire avec WaveDrom.

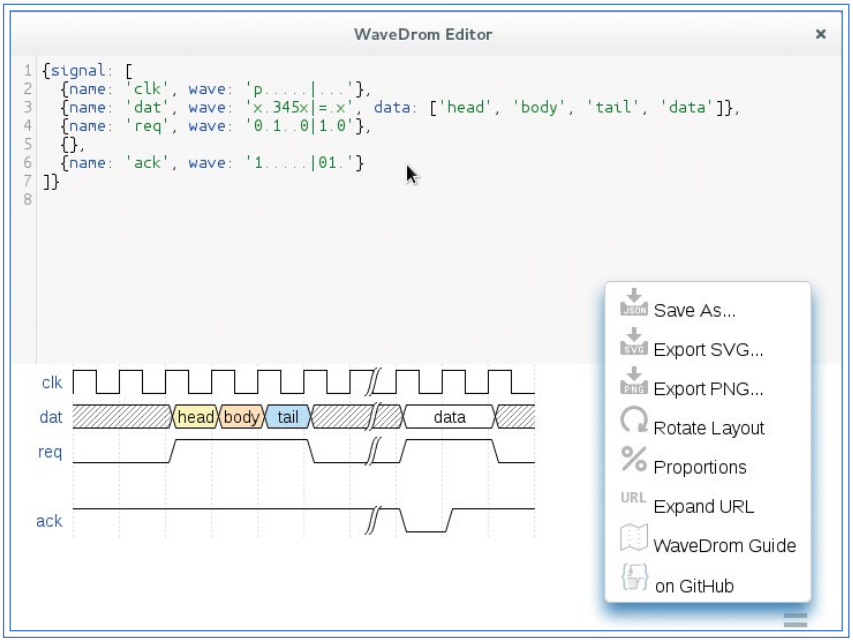


Fig. 6 : Exécution de l'application WaveDrom.

Le **p** représente une période d'horloge active sur front montant (positif), et le **n** sur front descendant (négatif). Il est possible d'ajouter une flèche pour indiquer le front en utilisant la majuscule et l'utilisation du **.** répétera la période d'horloge décrite au caractère précédent.

Nous n'avons ici qu'effleuré la syntaxe WaveDrom, il est aussi possible d'ajouter des titres, sous-titres et labels à des groupes de signaux. Il est également possible de tracer des flèches, d'intégrer des formules LaTeX et de rendre dynamique ou paramétrable le chronogramme grâce à l'insertion de JavaScript.

3. DIFFÉRENTES MANIÈRES D'ÉDITER

Pour le chaland voulant simplement tester rapidement WaveDrom, il suffit de se rendre sur le site officiel du projet, une zone d'édition/rendu s'y trouve tout simplement **[2]**.

Maintenant, si l'on veut faire des rendus WaveDrom en mode déconnecté, une application est aussi disponible **[3]** sous la forme d'un paquet **tar.gz** ou **zip** qu'il suffit de décompresser et de lancer avec la commande (voir figure 6):

```
$ ./wavedrom-editor
```

Une extension développée par votre serviteur est également disponible pour **mediawiki**. Une fois installée, il suffit d'encadrer sa prose WaveDrom par des balises :

```
<wavedrom></wavedrom>
```

Et enfin, la base, quand on veut intégrer ses chronogrammes dans ses documents LaTex et autres : une version en ligne de commandes nommée **wavedrom-cli [4]** est bien sûr disponible pour convertir un fichier **texte.js** en png/svg/jpg/...

Cette version en ligne de commandes demeure en JavaScript pur (**wavedrom-cli.js**), et il est donc nécessaire d'utiliser

un logiciel tiers pour l'exécuter (voir **phantomjs [5]**). Par exemple, pour convertir un fichier WaveDrom nommé `wb_ack.js` en image au format PNG on utilisera la commande suivante :

```
$ phantomjs /opt/wavedrom/wavedrom-cli.js
-i wb_ack.js -p wb_ack.png
```

ATTENTION !

Pour la version ligne de commandes de WaveDrom, il est nécessaire d'exécuter du JavaScript en mode « console ». Le site officiel recommande phantomjs :

```
$ wget https://bitbucket.org/ariya/phantomjs/
downloads/phantomjs-2.1.1-linux-x86_64.tar.bz2
$ tar -jxvf phantomjs-2.1.1-linux-x86_64.tar.bz2
$ vim ~/.bashrc
#phantomjs
export PATH="/opt/phantomjs-2.1.1-linux-x86_64/
bin":$PATH
```

CONCLUSION

WaveDrom permet de réaliser de beaux chronogrammes relativement facilement. Tous les amateurs de Latex/Markdown/... approuveront la description au format texte. Wavedrom est peu dépendant de logiciels installés sur sa machine dans la mesure où l'on utilise la version web, et il s'insère très bien dans les documents web ou les présentations du type **impress.js**. Et enfin, WaveDrom permet d'avoir une excellente finition de ses illustrations dans sa documentation. Ce principe de description graphique via une structure JSON a également été étendu pour réaliser des schémas de logique combinatoire **[6]**. ■

RÉFÉRENCES

[1] Le tutoriel officiel de wavedrom : http://wavedrom.com/tutorial.html

[2] Un éditeur en ligne permettant de tester wavedrom avec un simple navigateur et une connexion internet : http://wavedrom.com/editor.html

[3] Version « application » de wavedrom, à installer sur son ordinateur : https://github.com/wavedrom/wavedrom.github.io/releases

[4] Version ligne de commandes, pour intégrer dans un Makefile par exemple : https://github.com/wavedrom/cli

[5] Pour la ligne de commandes, il est nécessaire d'utiliser un logiciel pour exécuter le JavaScript : http://phantomjs.org/

[6] Comment utiliser WaveDrom pour faire des schémas : http://wavedrom.com/tutorial2.html

LA FAILLE DIRTY COW

SYLVAIN NAYROLLES
[Security Software Engineer]

MOTS-CLÉS : LINUX, NOYAU, DIRTY COW, MEMORY MANAGER, RACE CONDITION

Dirty COW a défrayé la chronique lors de sa découverte en 2016. Nous allons voir que cet engouement est justifié, car Dirty COW présente toutes les qualités d'une faille critique : stable, discrète, simple et impactant un grand nombre de versions du noyau Linux.

Derrière Dirty COW se cache une des failles, affectant le noyau Linux, les plus importantes de ces dernières années. À première vue, le nom peut paraître un peu exotique, mais nous verrons qu'il cache une réelle signification. Son nom officiel est **CVE-2016-5195**. Le bug à l'origine de la faille fût introduit par Linus Torvald en personne, suite à la correction malheureuse d'un bug similaire. C'est suffisamment rare pour être noté. On doit la découverte de cette faille à Phil Oester, un ingénieur en sécurité informatique californien. Elle fut difficile à mettre en exergue, car son exploitation repose sur une *race condition* du système d'exploitation, et plus particulièrement le module de gestion mémoire, réputé pour sa complexité.

1. RACE CONDITION

Une *Race Condition*, ou situation de compétition, définit une situation du système non prévue par le programmeur suite à un changement dans l'ordonnancement des états. Ce type de bug est bien connu lorsque l'on pratique la programmation concurrentielle. Il faut, dans ce cas, prendre en compte l'ordre d'exécution de nos tâches, la gestion

de l'accès concurrent à nos données, etc. Cette complexité augmente naturellement la surface d'attaque de n'importe quelle application.

Les *Races Conditions* constituent une classe de failles de sécurité reposant sur la recherche de l'état permettant l'exploitation. On se repose souvent sur l'appel d'un grand nombre de fois à une API en espérant que l'intervention de l'ordonnanceur des tâches entraînera l'état tant espéré.

Dans le cadre de l'exploitation de Dirty COW, c'est le système de gestion de la mémoire, ou *memory manager*, du noyau Linux qui est faillible. Nous allons voir qu'il existe un état non pris en compte entre la gestion du **Dirty** bit et la fonctionnalité de **C**opy **O**n **W**rite.

2. DIRTY...

Dirty COW affecte le système de gestion mémoire du noyau Linux. Ce dernier repose sur un système dit de pagination. La pagination est un mécanisme offert par la plupart des architectures processeurs modernes, et permet de répondre à une problématique de gestion de ressource mémoire. En effet, l'espace mémoire virtuel théorique adressable pour un processeur x86_64 est de 2^{64} bits, soit un peu plus de 2 000 000 de terabytes.

NOTE

En fait, il y a beaucoup moins d'espace d'adressage possible que les 2^{64} bits annoncés, car le bus d'adressage n'est pas entièrement exploité pour l'adressage mémoire. Mais cela reste grandement suffisant pour une utilisation commune.

Ceci ne correspond à aucun système physique. Afin de simuler un espace virtuel aussi grand, les fondeurs et les développeurs de systèmes ont mis au point des algorithmes tels que la pagination. Ces algorithmes permettent de réaliser un chargement partiel des données ou du code d'un programme, un chargement à la demande. Ces blocs partiels de la mémoire d'un processus sont appelés **pages**. Il existe donc différentes stratégies de gestion des pages, afin de sélectionner quelle page doit être déchargée afin de laisser sa place en mémoire physique à une autre page dont le processus a besoin. Pour cela, le système se base sur différents critères, dont le **Dirty** bit.

Le Dirty bit est un mécanisme présent dans les processeurs de la famille x86 permettant de notifier aux systèmes d'exploitation que la page mémoire associée a été modifiée depuis son chargement, ou bien depuis la dernière fois que le bit fût réinitialisé. Ce bit est souvent utilisé par le système d'exploitation pour gérer le chargement ou le déchargement des pages mémoires. Il est renseigné dans la **Page Table Entry**. La *Page Table Entry*, ou **PTE**, permet de faire le *mapping* entre l'adressage virtuel et l'adressage physique des pages mémoires d'un processus.

Le Dirty bit intervient aussi dans le mécanisme de cache des fichiers de Linux. Quand on lit un fichier sous Linux, il est mis en cache avant écriture. Cette problématique se retrouve souvent quand l'on veut transférer des données sur une clé USB, et que l'on arrache la clé ; souvent les données transférées sont corrompues et incomplètes. En interne, Linux utilise le Dirty bit pour notifier au cache d'écrire sur le média correspondant que le fichier a été modifié et qu'il nécessite une mise à jour.

En règle générale, un processus ne se préoccupe pas de la manière dont ces pages sont gérées par le système d'exploitation. Mais dans certains cas, souvent des processus nécessitant une phase importante d'optimisation, ou bien des exploits dans notre cas, il peut être intéressant d'indiquer au système d'exploitation comment gérer certaines d'entre elles. L'appel système `madvise` permet de fournir des indices au système sur l'utilisation ou non de certaines zones mémoires. En l'occurrence, il est possible de lui dire qu'une zone mémoire ne sera plus utile dans le futur du processus via le paramètre `MAP_DONTNEED`.

3. ...COW

L'acronyme COW signifie **C**opy **O**n **W**rite. Cela décrit un algorithme de gestion de page au sein du système d'exploitation. Tant qu'une page est en lecture, plusieurs processus peuvent y accéder de façon concurrente. Ce mécanisme est très utile afin de partager des segments exécutables, non modifiables, entre plusieurs processus, d'une même bibliothèque par exemple (bibliothèques dynamiques). Ce mécanisme prend tout son sens dans le cadre de l'appel système `fork`. En effet, juste après un `fork`, les deux processus possèdent tous deux les mêmes pages. Sauf quand une de ces pages est accédée en écriture, ceci provoque une faute de page que le noyau va intercepter. Suite à cela, il va provoquer la copie de cette dernière dans l'espace virtuel du processus demandant l'écriture. La PTE (*Page Table Entry*) associée à cette nouvelle page possède le Dirty bit. Cette page ne possède plus de lien avec la précédente, toute modification n'entraîne aucune modification sur la page source.

4. BUG

En fait, ceci n'est pas un nouveau bug. Une première correction de ce dernier fût réalisée il y a de cela onze ans par Linus Torvald en personne, mais malheureusement cette correction ne fut pas suffisante. C'est ce qui est décrit dans le message de *commit* du patch **[1]** :

```
This is an ancient bug that was actually attempted to
be fixed once (badly) by me eleven years ago in commit
4ceb5db9757a ("Fix get_user_pages() race for write
access") but that was then undone due to problems
on s390 by commit f33ea7f404e5 ("fix get_user_pages
bug").
```

Si nous analysons le patch, nous pouvons constater qu'il est relativement simple, dans sa forme bien sûr, car il faut quand même être relativement sensibilisé à la programmation noyau. De plus, la partie *memory management* est considérée comme étant la plus complexe.

Pour comprendre le bug, nous allons reprendre les notions précédentes.

Concrètement le bug de Dirty COW permet de réaliser une écriture dans une page en lecture seule. En effet, il existe un état qui permet de contourner les sécurités mises en place par le noyau.

Quand un processus possède une page en lecture seule et qu'il tente une écriture, une opération de Copy On Write est réalisée. Une fois que le processus n'a plus besoin de cette page, elle sera *flushée* et détruite.

Mais il ne faut pas oublier qu'une copie en informatique a toujours un coût temporel. Le bug consiste donc à exploiter un état où la copie n'est pas encore terminée, et le noyau autorise l'écriture dans la page en lecture seule.

Il existe donc une situation où l'on tente une écriture dans une page en lecture seule, ceci entraîne une opération de COW suite à une faute de page. Durant cette faute, le *thread* appelant va mettre à jour la *Page Table Entry* dont le Dirty Bit. Une nouvelle opération d'écriture arrive alors que l'opération de COW n'est pas terminée, et que les modifications de la PTE autorisent l'écriture. Le *thread* appelant arrive donc à écrire dans une page en lecture seule.

On commence à deviner le potentiel d'un tel bug, mais comment le rendre exploitable et stable pour en faire un CVE de qualité ?

5. EXPLOITATION

L'art de l'exploitation laisse souvent admiratif par l'ingéniosité que les développeurs mettent en place afin de tirer parti d'un bug noyau en espace utilisateur.

5.1 Tout est fichier

Il existe plusieurs méthodes d'exploitations de Dirty COW dans la nature. Nous allons décrire la plus courante et la plus simple qui se nomme `/proc/self/mem`.

`/proc/self/mem` est un pseudo fichier représentant la mémoire virtuelle du processus courant.

NOTE

Les systèmes de fichier `/proc` et `/sys` sont des systèmes de fichiers permettant d'accéder à des informations noyau pour chaque processus. On peut y trouver par exemple la ligne de commandes d'appel du programme, les zones mémoires, l'occupation de la mémoire physique, des informations d'ordonnancement... Ces deux pseudo systèmes de fichiers regorgent d'informations très utiles sur l'état de vos processus. D'ailleurs beaucoup d'outils d'audit tels que `ps` se basent sur ces systèmes de fichiers.

Dans le monde Linux, tout est fichier. En fait, derrière cette phrase se cache une homogénéité des API de programmation système sous Linux. Un processus peut donc adresser ses propres pages via le pseudo fichier `/proc/self/mem`.

Il est aussi possible de *mapper* un fichier, au sens système cette fois-ci, au sein de l'espace virtuel d'un processus, via l'appel système `map`.

5.2 Cible

Nous n'allons pas nous contenter de tenter d'écrire dans une page en lecture de notre processus, même si l'on pouvait envisager de réaliser, via cette technique, un binaire se modifiant lui-même, en tentant de modifier son segment exécutable. Au lieu de ça, nous allons tenter d'écrire dans un fichier pour lequel l'exploit possède des droits en lecture, mais bien sûr pas en écriture.

Le but d'un tel exploit est de modifier par exemple un fichier de configuration

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

d'un serveur web, un `authorized_keys`, qui liste les clés autorisées à se connecter en SSH à un utilisateur particulier, ou bien modifier un fichier exécutable avec un sticky bit.

NOTE

Le sticky bit permet de donner des privilèges élevés à un exécutable lancé avec un utilisateur non privilégié. L'utilitaire `ls` permet de visualiser ce type de permission via le caractère s. Il est possible de lister tous les fichiers possédant cette permission via la commande find suivante :

```
find / -type d -perm -1000 -exec ls -ld
{} \;
```

5.3 Mise en œuvre

L'exploit **[2]** va donc essayer de reproduire l'état de concurrence mettant à défaut les protections du système d'exploitation. Pour cela, il va créer deux *threads* dans lesquels il va répéter un grand nombre de fois deux opérations en espérant exploiter le bug décrit précédemment.

Avant de lancer les hostilités, il va au préalable *mapper* le fichier ciblé, disponible uniquement en lecture, dans son espace virtuel via la commande `map`.

```
map=mmap(NULL,st.st_size,PROT_READ,MAP_
PRIVATE,f,0);
```

L'option `MAP_PRIVATE` permet d'exploiter le mécanisme de Copy On Write vulnérable.

5.3.1 Thread 1

Le premier *thread* va se contenter de faire des tentatives d'écriture sur la zone mémoire correspondant au fichier ciblé. Pour cela, il va débuter en ouvrant le pseudo fichier `/proc/self/mem` :

```
int f=open("/proc/self/mem",O_RDWR);
```

Puis, se positionner sur la page correspondant à la zone mappée pointant vers le fichier :

```
lseek(f,(uintptr_t) map,SEEK_SET);
```

Et enfin écrire la donnée que l'on souhaite :

```
c+=write(f,str,strlen(str));
```

5.3.2 Thread 2

Le *thread* 2 va se contenter de notifier au système d'exploitation de libérer les pages associées au *mapping* du fichier via l'appel système `madvice` :

```
c+=madvise(map,100,MADV_DONTNEED);
```

À la fin de l'exécution du programme, il y a de grandes chances que le fichier cible contienne ce que vous désirez.

NOTE

Cette faille affecte toutes les versions du noyau Linux, sans oublier le système Linux le plus déployé de nos jours : Android. Il existe plusieurs PoC démontrant l'exploitabilité de Dirty Cow sous Android. Google a publié un patch correctif fin décembre. Mais la plupart des constructeurs ne diffusent pas ces patchs, si critiques soient-ils. Donc beaucoup de terminaux sont encore vulnérables à cette faille. Rassurez-vous, les dernières versions d'Android basées sur SELinux limitent considérablement la surface d'attaque.

CONCLUSION

Nous avons donc démontré la simplicité de mise en œuvre de Dirty COW ainsi que l'immense impact qu'elle peut avoir sur un système cloisonné. Malheureusement, beaucoup de versions du noyau sont impactées, et bien sûr l'ensemble des distributions actuellement en support a immédiatement intégré le patch dans une mise à jour. Le problème reste notable pour toutes les distributions dont le support est terminé. Si Dirty COW ne symbolisait pour vous qu'un engouement incompréhensible, j'espère qu'à la lecture de cet article vous ne douterez plus de l'importance de s'en protéger. ■

RÉFÉRENCES

[1] Patch correctif de Dirty COW : https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=19be0eaffa3ac7d8eb6784ad9bdbc7d67ed8e619

[2] Exploit de Dirty COW : https://github.com/dirtycow/dirtycow.github.io/blob/master/dirtyc0w.c

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***

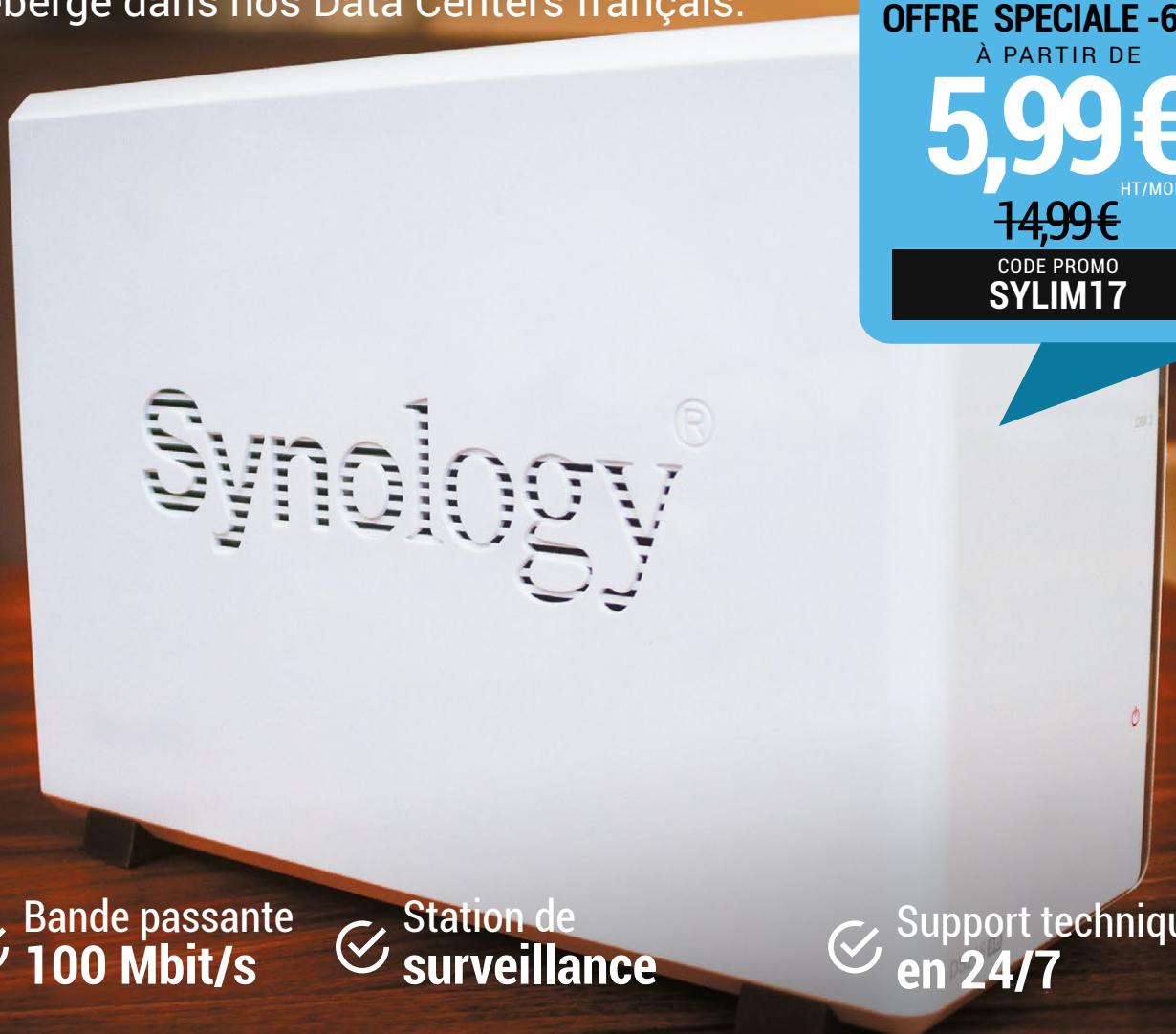
OFFRE SPÉCIALE -60 %

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

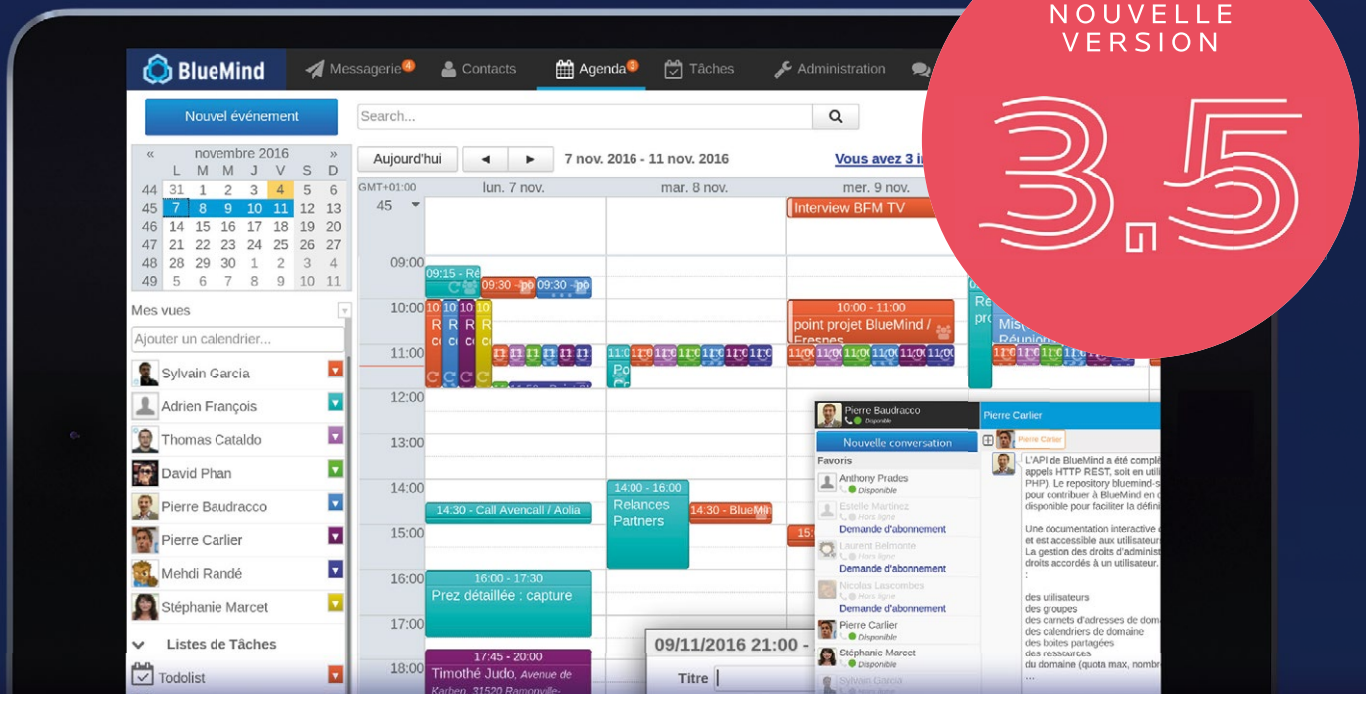
NOM DE DOMAINE | HÉBERGEMENT WEB | SERVEUR VPS | SERVEUR DÉDIÉ | CLOUD PUBLIC | MESSAGERIE | STOCKAGE | CERTIFICATS SSL

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