SAULQUIN Clément

Année 2017-2018

Réalisation d'interface graphique en Python pour le contrôle d'appareil par bus GPIB

Rapport de stage
DUT Génie Electrique et Informatique Industrielle
Du 9 avril au 22 juin 2018

Institut des Nanotechnologie de Lyon
7 avenue Jean Capelle, bâtiment Blaise Pascale
69100 Villeurbanne

Tuteur entreprise : M. Nicolas BABOUX

Tuteur pédagogique : Mme. Sandra IZEM

IUT Lyon 1 – Département GEII

Sommaire

Tables des illustrations

Remerciement

Avant de commencer mon rapport, j'aimerais adresser mes remerciements aux personnes qui m'ont aidé tout au long de mon stage et de ma scolarité.

En premier lieux j'aimerai remercier M. Nicolas BABOUX qui a pris de son temps pour m'aider durant les moments difficiles en me donnant de bons conseils pour progresser et réussir au mieux ma mission. Je lui suis également reconnaissant pour sa confiance qu'il a placée en moi pour m'avoir donné un projet aussi intéressant.

Je remercie également toute l'équipe Dispositif Electronique de l'INL de m'avoir accueilli au sein de leur équipe et de m'avoir aidé tout au long de mon stage.

Enfin je tiens à remercier toute l'équipe pédagogique du département GEII de Lyon1 pour la formation de qualité fournis. Je remercie tout particulièrement M. Jean-François Chateaux sans qui je n'aurais pas trouvé mon stage. Je remercie également Mme. Izem pour m'avoir assistée et soutenue durant mon stage.

I. Introduction

Dans le cadre de ma formation à l'IUT Génie Electrique et Informatique industrielle (GEII) de l'Université Claude Bernard Lyon 1 situé sur le campus gratte-ciel à Villeurbanne, j'ai réalisé durant dix semaines un stage à caractère professionnalisant au laboratoire de l'Institut des Nanotechnologies de Lyon (INL). L'objectif étant d'une part d'acquérir de nouvelles connaissances aussi bien techniques que sur le monde du travail. Mais également de valider mon diplôme universitaire afin de pouvoir continuer mes études.

Durant ce stage j'ai pu découvrir les différents domaines d'activités du laboratoire de l'INL. Ce laboratoire effectue des recherches sur différentes branches des nanotechnologies comme les matériaux fonctionnels, l'électronique, les technologies photoniques et photovoltaïque ainsi que les biotechnologies et la santé.

Dans un premier temps je vais vous présenter plus en détails le laboratoire, puis présenter ma mission au sein de ce laboratoire. Je vous parlerai ensuite de la réalisation de cette mission durant ces semaines de stage. Je finirais enfin par faire un bilan sur l'état d'avancement de ma mission ainsi qu'un bilan personnel.

II. Présentation du laboratoire

Avant de parler de mon stage en lui-même je vais tout d'abord vous présentez le laboratoire dans lequel j'ai travaillé et vous donnez les chiffres clés de celui-ci

A. Présentation de l'INL

L'institut des nanotechnologies de Lyon, aussi appeler INL, est un laboratoire de recherche fondé en 2007 sous la tutelle du Centre National de la Recherche Scientifique (CNRS), de l'Ecole Centrale de Lyon (ECL), l'Institut National des Sciences Appliquées (INSA), de l'université Claude Bernard Lyon 1 et de l'école CPE Lyon. Le laboratoire est également assister dans son travail par la plateforme NanoLyon. L'INL est aujourd'hui sous la direction de Mme Catherine BRU-CHEVALIER.

L'objectif du laboratoire est de développer et d'encourager les recherches sur les micros et nanotechnologies à l'échelle mondiale aussi bien sur leurs développement que sur leurs applications. Les domaines de recherches se regroupent autour de quatre axes.

- La recherche sur les matériaux fonctionnels. Le développement d'application ou de méthode dans d'autres secteurs de recherche du laboratoire passe souvent par le développement de nouveaux matériaux ou de nouvelles méthodes pour élaborer ces matériaux. C'est là tout l'intérêt de ce domaine de recherche qui assure une certaine autonomie et un contrôle sur les nano matériaux.
- Le département électronique. Son objectif et de développer des technologies et de systèmes électroniques et de capteur pour des applications spécifiques. Le département développe aussi en parallèle de nouveaux procéder méthodologiques.
- La photonique et le photovoltaïque. La nano photonique se définie comme le contrôle de la propagation et de l'interaction des photons dans des volumes restreints. L'INL est quant à lui spécialisé dans les différents procédés technologiques allant de l'élaboration a la simulation.
- Le département Biotechnologie et santé. Son rôle d'apporter de nouvelles solutions aux différents problèmes du monde du vivant. Ces solutions peuvent aller de la miniaturisation de technologies au développement d'objet tel que des capteurs ou des vêtements

C'est grâce à cette pluralité scientifique et économique que l'INL est aujourd'hui devenu une plateforme scientifique incontournable dans le secteur des nanotechnologies à Lyon.

J'ai quant à moi évoluer dans l'équipe Dispositif Electronique (DE) diriger par Francis CALMON et Brice GAUTIER. Son objectif est de développer des technologies micro et nano

électronique dans le but de réaliser de nouvelles fonctions de base des nano composants, capteurs ou encore de microsystèmes. Le tout répondant aux besoins de notre société, notamment à ceux liés à la santé, l'environnement et le numérique. L'équipe met également un point d'honneur sur la consommation énergétique des composants comme par exemple des composants a ultra faible consommation.

B. Les chiffres clés

Pour les amateurs de chiffres, voici comment on peut résumer le laboratoire INL. L'INL est composé de plusieurs laboratoire, l'un présent sur à Ecully et l'autre sur le campus Lyon-Tech La Doua. Il regroupe au totale environ 200 salariés dont 121 postes permanents tous appartenant à différents corps de métier. On y retrouve notamment 65 enseignants chercheurs, 22 chercheurs du CNRS et 38 personnels techniques et administratifs. Le laboratoire comporte également des étudiants, 68 étudiants doctorants et 10 étudiants post doctorants.

Pour effectuer ses recherches, l'INL dispose d'un budget total s'élevant à 3.8 millions d'euros hors taxe et hors salaires. Ce budget permet aux chercheurs de l'INL de mener à bien leurs recherches et ainsi produire entre 100 et 120 publications scientifiques en moyenne par ans et d'organiser environ 120 conférences scientifiques par ans. L'INL dépose également une moyenne de 3 brevets par ans.

III. Présentation de la mission

Je vais maintenant prendre le temps de vous présenter la mission qui m'a était confié à mon arriver au laboratoire. Je vais tout d'abord placer le contexte de la mission avant de faire une rapide présentation de celle-ci.

A. Contexte de la mission

Le laboratoire utilise actuellement un banc de mesures dont le but principale est de faire des relever électriques sur des échantillons d'isolant ferroélectrique et d'en déduire certaines propriétés afin de les améliorer. Des résultats obtenus, les chercheurs peuvent ensuite trouver l'application idéale à ce matériau ou en découvrir de nouvelles propriétés intéressantes pour d'autres applications.

Le banc de mesures est composé de plusieurs appareils. Une générateur de fonction Nf Wf1996, un impédance-mètre HP-4284A, l'Agilent 4156 B, un oscilloscope Integra 40 et un matrice de commutation7008A. Je reviendrais sur l'utilité et les spécificités de chaque appareil plus tard.

Ce banc de mesures est géré par un bus General Purpose Interface Bus (GPIB) et relié au PC à l'aide d'un adaptateur GPIB-USB de National Instrument. Les appareils sont ensuite gérés des interfaces développées sous LabVIEW. C'est à ce niveau qu'arrive le problème. Pour envoyer les requêtes sur le bus GPIB, LabVIEW a besoin d'utiliser des drivers spéciaux pour utiliser l'adaptateur GPIB-USB. Cependant les drivers GPIB compatible avec LabVIEW ne fonctionne que sur d'ancienne version de LabVIEW qui elles-mêmes ne peuvent être utilisées que sur d'anciens systèmes d'exploitations limitant ainsi les performances du système d'acquisition et du traitement des données. Il est donc préférable de s'affranchir de ces contraintes.

B. Présentation de la mission

Ma mission a donc pour but de s'affranchir de la contrainte que représente LabVIEW et de rendre les programmes de gestion et de traitements plus adapté. Pour ce faire, le laboratoire a donc décidé de réécrire les scripts LabVIEW en langage Python. Je dois donc traduire les scripts LabVIEW en Python afin de pouvoir gérer les différents appareils sur le bus avec les

commandes GPIB adéquats. Mon travail consiste également à créer une interface graphique afin de rendre l'utilisation des appareils la plus complète et agréable possible.

Je dois cependant respecter quelques contraintes qui ont guidé mes choix sur le choix de certaines librairies et sur ma programmation.

- Développer un code portable sur tout type d'OS. C'est cette contrainte qui influence le plus la création du programme et de l'interface graphique. En effet même si Python reste portable, certaines librairie le sont moins que d'autre, notamment les librairies dérivées de librairies C ou C++.
- Utiliser Python. C'est un énorme avantage de pars la simplicité et la puissance du langage. Cependant l'aspect des fois transparents de ce que fais python rends parfois la programmation compliquer à aborder et à comprendre.
- Le design de l'interface graphique. L'interface graphique doit rendre l'utilisation du programme la plus complète et la plus agréable possible. Designer l'interface pour rendre la prise en main la plus rapide possible peut parfois poser problèmes.
- Le temps d'exécutions. Même si cette contrainte est plus au moins importante, j'apporte un point d'honneur à l'efficacité et la rapidité du programme fournis. Python étant de base un langage relativement lent, il n'est pas la peine de rendre les programmes encore plus lents.

Les enjeux de cette mission sont multiples mais l'enjeu majeur reste de pouvoir se débarrasser de LabVIEW afin de ne plus être bridé par l'utilisation d'ancien OS. Un autre enjeu est celui du contrôle bus GPIB. En effet avec un contrôle plus souple du bus et des appareils présents dessus il sera possible de faire des mesures ou configuration juste à partir de l'interpréteur Python. L'interpréteur est une console de commande permettant de coder en python sans faire de script. Le dernier enjeu est de rendre les scripts beaucoup plus portables. Ainsi, avec les bonnes librairies installées et les drivers GPIB, il sera alors possible pour une personne de faire des mesures avec son propre ordinateur sans passé par la machine du laboratoire.

Je vais à présent aborder l'aspect technique et la réalisation de ma mission durant ces dix semaines. Je commencerais par approfondir quelques points rapidement évoqués lors la présentation générale. Je vais ensuite expliquer la méthode que j'ai utilisée pour mener à bien ma mission puis je finirais avec la réalisation technique de la mission.

C. Présentation approfondie

J'ai abordé dans la présentation générale certains points qui méritent d'être approfondie pour une meilleure compréhension. Notamment le bus GPIB, les appareils présents sur le bus. J'éclaircirais également le cahier des charges imposé.

1. Bus GPIB

Tout d'abord le bus GPIB (General Purpose Interface Bus). C'est un bus de contrôle d'instrument de mesures électronique. Il est issu d'un bus de contrôle déjà existant, le bus HP-IB (Hewlet-Packard Interface Bus) créé en 1965 par Hewlet-Packard. Le bus devient ensuite le bus GPIB après la normalisation IEEE-488 au début des années 70.

Figure 1: exemple de câble GPIB. *Positron-libre*

La norme évolue ensuite en séparant la norme IEEE-488.1 qui normalise la couche physique (le câble) et la couche liaison (protocole de communication).

Le câble est constitué de 24 fils. Parmi ces fils 8 sont utilisés pour la transmission des données en parallèle (1 fils pour chaque bit de l'octet envoyer), ils sont notés de DIO1 à DIO8. 5 sont utilisés pour la gestion de l'interface et 3 pour le dialogue (préparation et fin de dialogue). Enfin les 8 restants sont des masses.

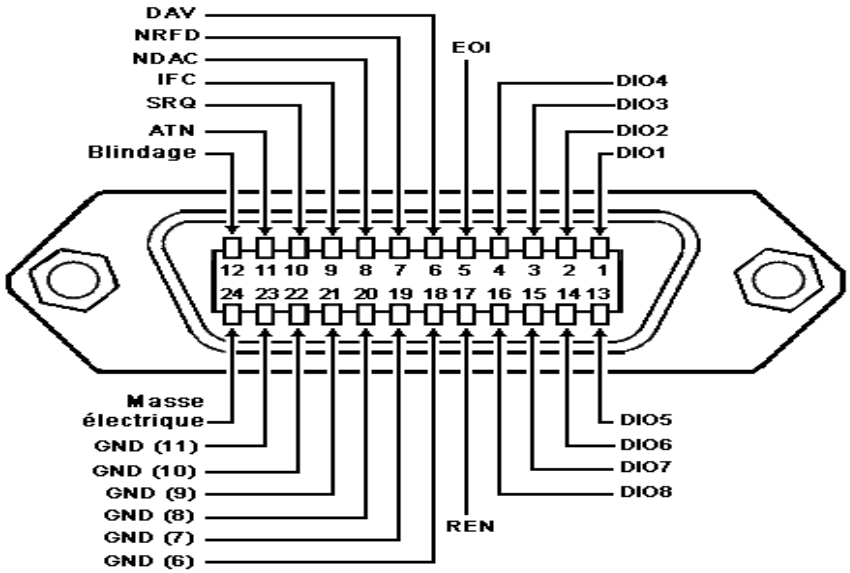


Figure 2: Pins des connecteurs GPIB. *Positron-libre*

La seconde norme, IEEE-488.2, apparut en 1990, définis quant à elle un langage de commande utilisé pour la communication entre appareil. Ce langage est appeler SCPI pour **Standard Commands for Programmable Instruments**. Voici quelques exemples de commandes :

- IDN ? : Demande à l'instrument d'envoyer ses paramètres comme le nom ou l'adresse.
- MEASure:VOLTage ? : Demande à l'instrument de retourner la tension mesurée. On notera l'importance de la case dans ces commandes. En effets les majuscules sont les caractères obligatoires tandis que les minuscules sont facultatives. Ainsi la requête peut également s'écrire : MEAS:VOLT

Les constructeurs utilisent donc ce langage pour définir les commandes de gestion de leurs appareils. Cependant certains appareils, notamment ceux fabriqué avant la normalisation des commandes, utilisent des commandes qui leurs sont propres rendant la programmation plus compliqué pour l'utilisateur.

Je terminerais par évoquer quelques caractéristiques spécifiques du bus GPIB. Tout d'abord le nombre d'appareils présents sur le bus ne peut pas dépasser 15. Cependant certaines carte de National Instrument peuvent accueillir jusqu'à 30 appareils de mesures. Il existe deux configurations possibles pour brancher les appareils sur le bus. La premières est la liaison linéaire où les instruments sont reliés les uns à la suite des autres. La seconde configuration est celle de l'étoile ou chaque instrument est relié à un même instrument qui lui est relié au poste de contrôle, un ordinateur le plus souvent.

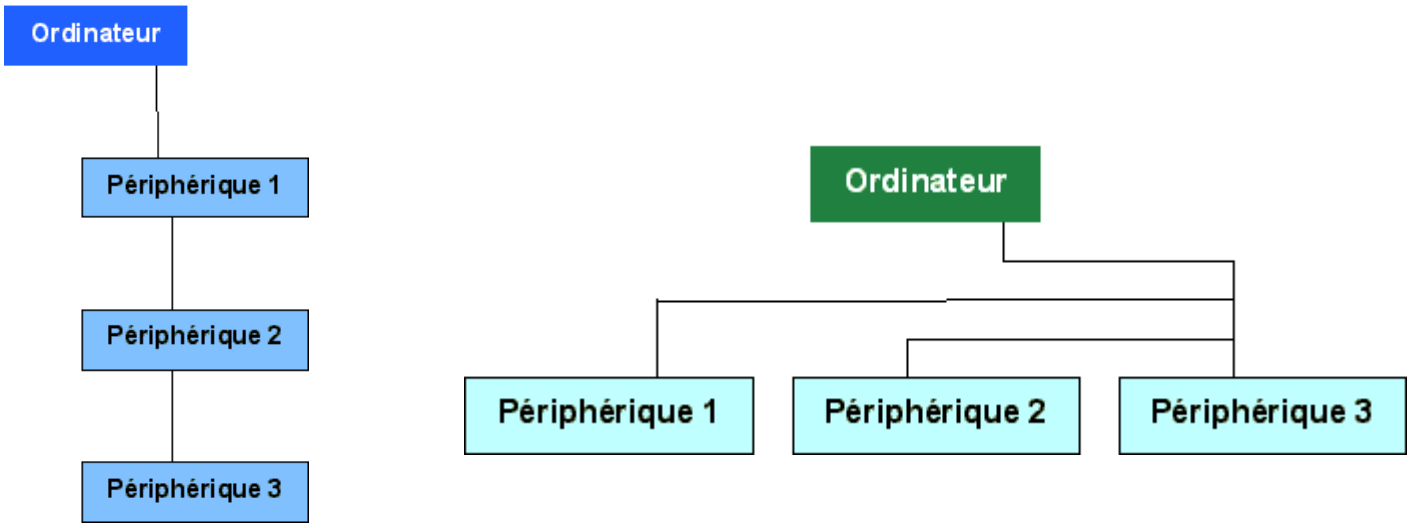


Figure 3: Configuration d'un bus GPIB. Linéaire à droite et en étoile à gauche *Nation Instrument*

Il est également possible de combiner ces deux configurations. Dans tous les cas il faut faire attention à ne pas dépasser le nombre d'instrument limite ainsi que la longueur totale du bus. En effet la longueur totale des câbles présents sur le bus ne peut pas excéder les 20 mètres. Il est cependant possible d'utiliser un extenseur GPIB pour allonger la longueur du bus.

Figure 4: Extenseur GPIB. *National Instrument*

De plus la distance entre chaque appareil du bus ne doit pas dépasser 4 mètres.

2. Appareils présents sur le bus

Je vais maintenant vous présenter les différents appareils présents sur le bus. Ce sont ces appareils que j'ai dû piloter avec les scripts pythons.

Le premier appareil que j'ai dû piloter est la matrice de commutation 7008A. Sont rôles est d'orienter des signaux électriques ou des signaux obtenues par mesures vers différents endroit du banc de mesures. La matrice permet donc d'éviter de re-câbler tout le banc de mesures à chaque changement de configuration et donc de simplifier les manipulations.

Le deuxième appareil auquel j'ai dû faire face est l'Agilent 4156B. Il permet d'effectuer de nombreuses mesures sur l'échantillon que les autres appareils ne peuvent pas faire. Sa principale qualité reste sa grande précision étant donné qu'il peut mesurer des courants de l'ordre du femto (10e-15) ampère.

Le bus contient également un impédance-mètres HP4284A. Il permet notamment d'effectuer les mesures d'impédances et de conductances sur l'échantillon permettant de déterminé si le matériau possède bien les propriétés attendu.

Un générateur de fonction est également présent sur l'appareil. Cet appareil permet de générer des fonctions ayants des formes précises pouvant être programmé directement par l'utilisateur. Ainsi les chercheurs peuvent stimuler leurs échantillons avec des courbes bien précises permettant de mettre en évidence certaines propriétés du matériau.

Le dernier instrument de mesure que j'ai commandé est un oscilloscope Nicolet Integra 40. Il permet de réaliser l'acquisition de courbes qui peuvent ensuite être récupérer et traiter par ordinateur.

Il y a également deux appareils supplémentaires. Un amplificateur de courant permettant d'amplifier les mesures de courant en sortie du banc de mesures afin de pouvoir rendre l'acquisition et le traitement plus simple. Le second est un amplificateur de tension. En effet un point faible du générateur de fonction est qu'il ne peut délivrer d'amplitude supérieure à 10 volts. C'est donc pour s'affranchir de ce point faible que l'amplificateur à était ajouter.

Figure 5: banc de mesures avec les appareils de mesures à gauche.

3. Cahier des charges

Le cahier des charges dépends de chaque appareils sur le bus étant donné qu'ils ont tous des utilisations différentes. Néanmoins on peut tirer des différents cahiers des charges des points communs et des lignes directrices communes.

Dans tous les cas l'objectif et retranscrire le code LabVIEW en code Python facilement utilisable. Le code fournis doit reprendre les parties importantes à la gestion du bus. Il doit permettre une utilisation la plus souple et la plus simple possible pour des utilisateurs n'ayant aucunes notions en programmations ou en Python.

La plupart des scripts python doivent être accompagnés d'une interface graphique permettant un réglage rapide des paramètres à envoyer sur le bus GPIB. L'interface doit être astucieusement réfléchie pour être la plus simple d'utilisation possible. Elle ne doit pas ralentir la manipulation en étant trop complexe. Cependant si l'appareil ne dispose pas d'énormément de paramètres ou que son utilisation est très simple, il est possible de laisser l'interface de côté si elle s'avère inutile.

Sur certains code, il m'a était imposer, en plus de l'interface graphique, de réaliser des commande à écrire directement dans l'interpréteur python. Cela permet de rendre l'utilisation encore plus souple et de pouvoir réaliser des commandes qui auraient était difficilement

développable en restant sur l'interface graphique. Cela permet également à l'utilisateur de saisir des requêtes GPIB qu'il a lui-même écrit pour faire des tests sur l'appareil.

Enfin dans certains cas et notamment pour les scripts où des commandes sont disponibles, une documentation est nécessaire. Le but étant de permettre à l'utilisateur de prendre en main le programme et d'utiliser les commandes disponibles sans à rentrer dans les lignes du script. La documentation s'adresse avant tout à des personnes sans expériences en programmation et ce doit donc d'être clair et illustrée d'exemple précis.

De ce cahier des charges généralistes, j'ai pu en déduire une méthode de travail que j'ai pu appliquer sur chacun des scripts.

D. Méthode de réalisation

Dans cette partie je vais vous expliquez la méthode que j'ai employé pour réaliser les différents programmes. Cette méthode c'est développer avec le temps et à évoluer à chaque fois que je voyais que la méthode que j'employais actuellement avait atteints ses limites. Je détaillerai certains points plus loin.

Je commence par l'analyse des scripts LabVIEW existant. Je détermine leurs utilités dans le programme général puis je détermine lesquels méritent une étude approfondie. Après cela je commence à étudier les scripts choisis plus en détails et d'en déduire la structure logique (boucles, conditions) du programme. Je récupère également les commandes GPIB qui me seront utiles pour la gestion de l'instrument.

Dans premiers temps je passe en revue l'ensemble des scripts LabVIEW utilisé pour le contrôle de l'appareil. Je relève ainsi leurs noms et surtout le but du script en question. S'il s'agit plus d'un script de récupération de données, d'écriture de données dans l'appareil ou encore de paramétrage de l'appareil.

Je passe ensuite à la phase de test des commandes GPIB pour vérifier leurs bon fonctionnement et déterminer la forme des résultats retournés pas cette commande. C'est une partie importante du travail car le bon fonctionnement du programme repose sur les requêtes GPIB.

Une fois toutes les requêtes vérifiées, je passe à la partie programmation. Je dessine d'abord l'interface graphique. Le but de mieux déterminé si elle sera simple d'utilisation et d'avoir un modèle facilitant la programmation. Je détermine ensuite les différentes classes du programme. Je reviendrais plus tard sur la définition d'une classe mais pour l'instant retenez qu'une classe correspond à une partie du programme. Une fois toute cette partie théorique terminée, je passe à la programmation à proprement parler. Je commence toujours par l'interface graphique puis j'implémente les requêtes GPIB.

Une fois le programme terminé je passe à la phase de test pour vérifier le bon fonctionnement du programme. Je commence par une utilisation normale du programme puis je pousse le programme à ses limites pour voir ses réactions. Je corrige ensuite les bugs et refaits des tests jusqu'à ce qu'il n'y ait plus de problèmes notable.

Si cela m'ai demandé, je passe ensuite au développement des commandes interpréteur. Je commence par lister toutes les commandes que je vais devoir coder en notant pour chacune d'elles la parties du programmes en jeux. Je passe ensuite au développement puis à la phase de test et à la correction des bugs.

Je présente ensuite le programme à M.Baboux pour vérifier si mon travail correspond à ses attentes. Pour finir si le besoin s'en fait sentir, je rédige un document d'utilisateur pour permettre une meilleure prise en main du programme.

Je vous ai donc présenté plus en détails la mission confiée. J'ai évoqué les détails du bus GPIB permettant de commander les appareils de mesures que dont j'ai expliqué le rôle de chacun. Puis après avoir regardé plus en détails le cahier des charges, j'en ai déduis une méthode de travail que j'ai appliqué tout au long de mon stage.

IV. Réalisation

Je vais maintenant parler de la réalisation en elle-même. Elle se décompose en cinq parties différentes, l'installation des outils puis une partie par scripts réaliser. Je vais donc commencer par vous parler de l'installation des différents outils informatiques que j'ai utilisés. Ensuite au lieu de parler de chacun des scripts je vais vous exposer plus en détails la méthode utilisée en parlant de certaines spécificités que les scripts ont imposées.

A. Installation des outils

La toute première étape de ma mission à étais d'installer les outils adéquats pour pouvoir ensuite travailler avec les instruments présent sur le bus GPIB. La structure logicielle d'un bus GPIB peut se représenter en trois couches. La couche drivers, la couche Visa et la couche LabVIEW ou Python dans mon cas.

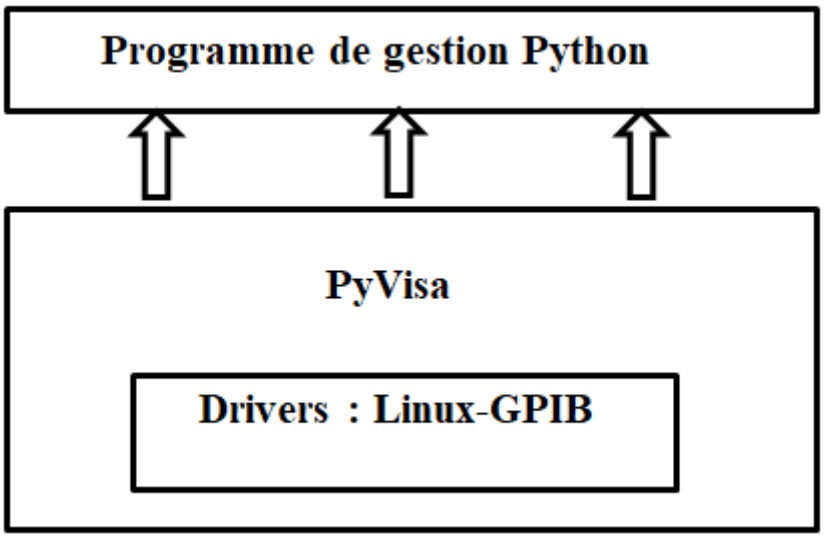


Figure 6: Couche logicielle d'un bus GPIB

La première couche correspond à la couche la plus basse, celle des drivers GPIB. Elle permet à l'ordinateur d'établir la communication entre le bus est la couche Visa. La plupart du temps ces drivers sont fournis par National Instrument et permettent de gérer le bus GPIB avec LabVIEW. Cependant ce sont ces pilotes que l'on veut enlever car ils ne sont plus disponibles sur les versions récentes du noyau linux.

Pour remplacer les pilotes de National Instrument, nous avons choisis d'utiliser le driver Linux-GPIB. Ce programme permet d'installer tous le nécessaire à la communication entre le bus GPIB et l'ordinateur. Son installation n'a pas était évidente car elle nécessitait de modifier de nombreux fichiers du driver afin qu'ils puissent reconnaître les adaptateurs GPIB-USB utiliser par le laboratoire. A cela c'est rajouter des tests de communication peu concluante qui a lever en moi des doutes quant à l'installation du driver. C'est après avoir regardé de nombreux forums sur internet et avec l'aide de mon tuteur que j'ai enfin réussis à installer et à faire fonctionner le driver GPIB. En effet il manquait une étape importante qui consiste à valider la configuration du driver. J'ai ensuite vérifié la communication avec un multimètre donné par mon tuteur pour faire les tests. Pour établis la communication j'ai utilisé un programme qui s'appelle *ibtest* fournis avec linux-GPIB.

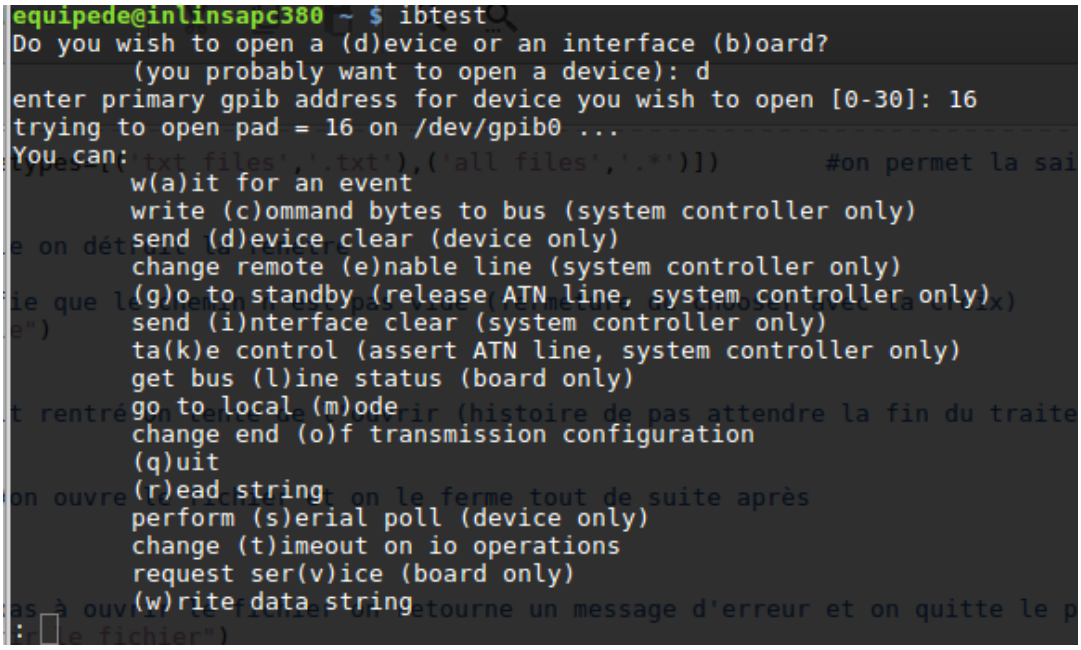


Figure 7: écran ibtest permettant de communiquer avec un appareil.

Viens ensuite l'installation de seconde couche, la couche Visa signifiant **Virtual Instrument Software Architecture** et qui englobe la couche driver. Cette couche, généralement fournis par LabVIEW, permet de communiquer avec le bus GPIB avec les requêtes de l'appareil. Une version python est disponible et il suffit d'installer PyVisa pour pouvoir utilisé les ressources Visa. Cependant deux problèmes se posent.

Le premier est que Visa est fourni par N.I et il utilise les drivers également fournis par N.I. Pour régler ce problème il faut installer une autre librairie qui s'appelle PyVisa-Py. Elle permet de configurer PyVisa afin qu'il puisse faire abstraction des drivers par défauts et utiliser ceux disponibles sur l'ordinateur comme linux-GPIB. La différence au niveau du code n'est pas flagrante. Pour ouvrir un gestionnaire de ressources visa, il suffit de taper la commande : *visa.ResourceManager('@py')* plutôt que *visa.ResourceManager()*. Cette légère différence dans le code permet néanmoins de préciser a Visa d'utiliser linux-GPIB.

Le second problème est la polyvalence de PyVisa. En effet cette bibliothèque permet de communiquer à l'aide de plusieurs types de bus comme un bus Ethernet ou USB. Il faut donc préciser le bus par défaut qui sera utilisé et pour ce faire, il faut installer la librairie gpib-ctypes. Elle permet d'initialiser le driver GPIB pour qu'il soit disponible par PyVisa. Il initialise également le bus GPIB comme bus par défauts détecté par PyVisa.
Pour résumer, le début de chaque programme python qui communiquera avec le bus GPIB commencera par les lignes suivantes :

```
>>> from gpib_ctypes import make_default_gpib
>>> make_default_gpib()
>>> import visa
>>> rm =visa.ResourceManager('@py')
```

Figure 8: Entête des programmes communiquant avec un bus GPIB.

L'installation des librairies en elles-mêmes n'a pas posé problèmes. Python dispose d'un installateur de librairies intégrer qui s'appelle PIP pour Pip Installs Packages. Il permet de télécharger les librairies et de les installer avec une simple ligne de commande :
pip install Nom

Enfin la dernière couche du modèle présenter un peu plus haut correspond à la partie LabVIEW que je dois remplacer par les programmes python. Je vais donc maintenant vous expliquer plus en détails comment je réalise un programme python pour gérer un instrument à partir des programmes LabVIEW existants.

B. Réalisation des programmes de gestion

Je vais maintenant vous expliquer de manières détaillée les étapes de la réalisation d'un programme. Même si le contenu de ces étapes divergent en fonction de l'appareil la méthode reste la même et c'est perfectionner au fil du temps. Elle se décompose en 4 à 5 étapes suivant le résultat attendu. On y trouve l'étude des scripts LabVIEW, les tests des commandes GPIB, le développement du programme, la phase de test et enfin si nécessaire la rédaction d'une documentation d'utilisation.

1. Etude des scripts LabVIEW

L'étude des scripts LabVIEW est la première étape lors de la réalisation d'un programme. Elle se décompose généralement en deux étapes même si dans certains cas la première étape peut être passée sous silence.

La première étape est l'analyse de l'utilité de chaque script. Cette étape permet de voir la structure générale du programme et de lister les scripts plus importants que d'autre. Cette liste permet de focaliser mon travail et ainsi ne pas perdre de temps avec des scripts qui ne

permettent pas de répondre au cahier des charges. Cette liste peut ensuite s'allonger dans certains cas, par exemple si une partie du programme utilise plusieurs scripts différents.

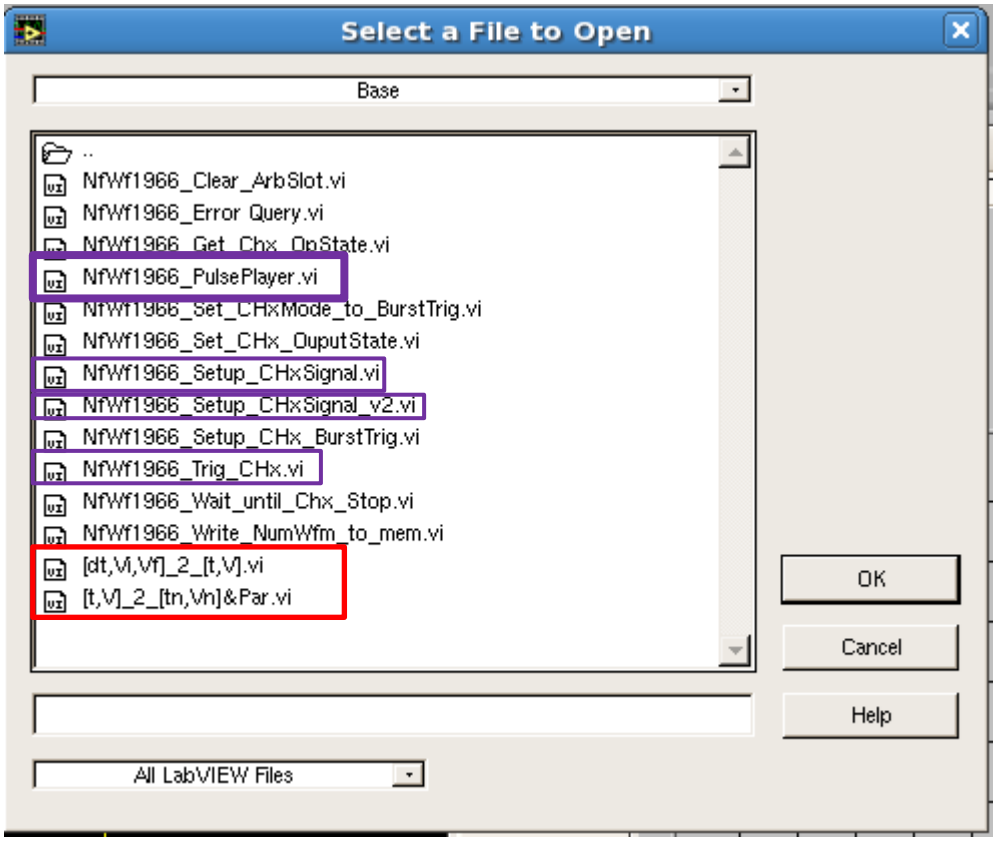


Figure 9: Liste des fichiers du programme du générateur de fonctions.

Dans cette situation par exemple, cette étape prends toutes sont importances et permet de se focaliser sur les scripts plus importants. Ici par exemple les deux fichiers entourés en rouges servent à générer des tableaux de points et ont un rôle majeur dans le bon fonctionnement du programme. Tandis que les deux premiers fichiers ne me permettent pas de répondre au cahier des charges. De plus l'un des fichiers important le PulsPlayer.vi utilise plusieurs scripts pour fonctionner qui viennent s'ajouter à la liste des fichiers à étudier.

La seconde étape consiste à étudier la structure logique du script en question. Ce que j'appelle structure logique, c'est la succession de chaque étape du programme ainsi que leurs particularités. Cette étapes est assez longue et demande souvent une grande concentration pour ne pas oublier d'étapes. Cependant la programmation très visuelle de LabVIEW facilite grandement cette étape. C'est également lors de cette étape que je récupère les requêtes GPIB qui me seront utiles pour la programmation. Voici un exemple de script LabVIEW issue du générateur de fonction.

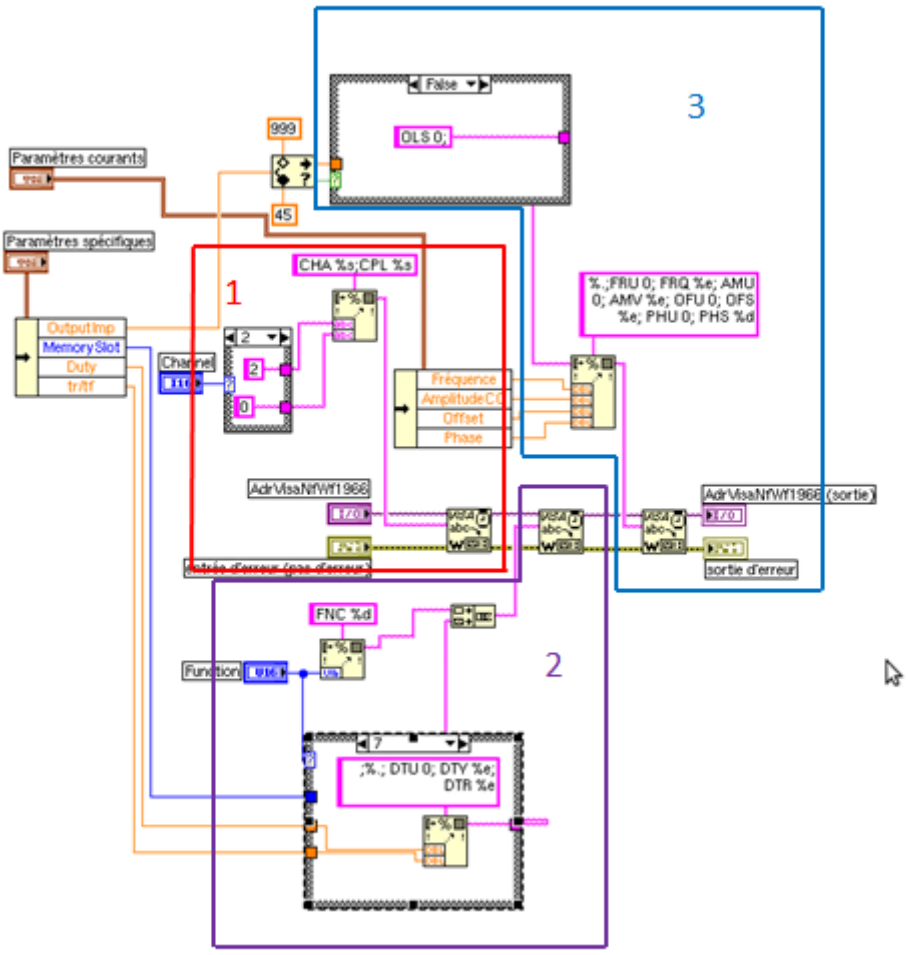


Figure 10: Exemples de script LabVIEW communiquant avec le bus GPIB.

Ce script permet de configurer les paramètres d'une fonction, comme la fréquence ou l'amplitude, que l'appareil devra envoyer. Grâce à la représentation visuelle du script, on peut voir qu'il se décompose en trois grandes étapes correspondants chacune a l'envoie d'une requête GPIB.

Je commence par noter toutes les variables d'entrées que ce script utilise. Le but étant de faciliter la programmation python plus tard. Ensuite je commence à étudier la structure du programme avec l'ordre d'envoie des requêtes et les particularités de chaque étape comme le formatage de chaine représenter par ces bloc :

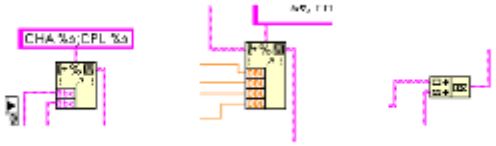


Figure 11: Bloc de formatage de chaines de caractères.

Ou encore les structures conditionnelles comme le if/else représenter par ce type de bloc :

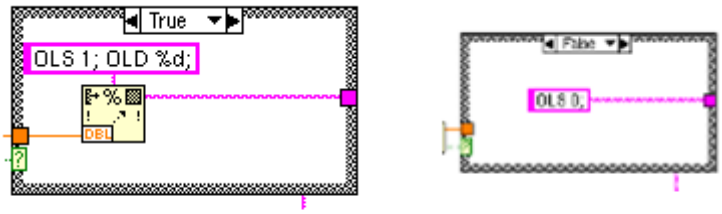


Figure 12: Bloc de structure if-else en LabVIEW.

Par exemple pour ce script j'obtiens le document suivant

Entrées : AmplitudeCC, fréquence, offset, phase, channel, function (sinusoid 1 ; triangle 2 ; Fsquare 3;Pramp 4 ; Nramp 5 ; user 6;Vsquare 7), Memory Slot, Duty, OutpuImp, tr/tf

Étape n°1 :

-Si **channel** différents de 0 : requête = CHA **channel**;CPL 0
-Si non : requête = CHA 1;CPL 1
-Envoie de la requête

Étape n°2 :

-Requête =FNC **function**
-Si **function** égale 3 : je rajoute à la requête ;DTR **tr/tf**
-Si non, si **function** égale 6 : je rajoute à la requête ;AFN **Memory Slot**
-Si non, si **function** égale 7 : je rajoute la requête ;DTU 0;DTY **Duty**;DTR **tr/tf**
-Si non je ne rajoute rien
-Envoie de la requête

Étape n°3 :

-Si 45<**OutpuImp**<999 : requête = OLS 1;OLD **OutpuImp**;
-Si non : requête = OLS 0;
-Je rajoute à la requête : FRU 0;FRQ **frequence**;AMU 0;AMV **amplitude CC**;OFU 0;OFS **offset**;PHU 0;HPS **phase**
-Envoie de la requête

Lors de l'étape deux, le numéro lors des tests de fonction correspondent a la place de la fonction dans la liste. La fonction 6 correspond donc à une fonction de type User qui correspondant à une fonction dont nous avons envoyé les points auparavant.

L'avantage de cette méthode est qu'ensuite la programmation du la fonction en question ne se résumera qu'à recopier les différentes étapes. Il faut cependant des fois adapté cette structure notamment en cas de problèmes avec les requêtes GPIB. Notamment sur les formats d'envoyer de données qui nécessite des traitements particuliers qui, sur LabVIEW, sont effectuer à partir de bloc dont le fonctionnement est transparents pour l'utilisateur. Ce sont ces blocs qui rendent le test de commande GPIB parfois compliqué.

2. Test des commandes GPIB

La vérification du bon fonctionnement des commandes GPIB est une étape cruciale dans la réalisation du programme. En effet c'est grâce à ces commandes que le programme peut communiquer avec les appareils présents sur le bus.

Ainsi je test les commandes une par une dans l'ordre d'arrivée dans le programme, cette démarche à plusieurs objectif. Le premier est surtout utiles pour les commandes qui permettent de lire des valeurs sur de l'appareil. Le but étant d'identifier le format de retours afin de récupérer des valeurs cohérentes et ensuite de les traiter de la meilleure manière possible. Un autre objectif est de vérifier que les commandes marchent bien sur python. Car même si LabVIEW et Python utilise Visa pour communiquer, certaines fonction de LabVIEW ne sont pas disponibles en python. De plus les méthodes PyVisa utilisées ont des critères d'envoie plus stricte qu'en LabVIEW. Pour illustrer ce propose je me dois de vous parler du générateur de fonction.

Une partie de mon travail sur ce script était d'envoyer les points correspondant à une courbe saisie par l'utilisateur. L'envoie sous LabVIEW est très simple. Chaque nombre compris entre -32767 et 32767 (entiers signés sur 16 bits) est convertis en caractère en gardant la même écriture hexadécimale : par exemple la valeur 65 (0x41) est coder en A dont le code hexa est 0x41. La chaine de caractère ainsi obtenue est envoyée sur le bus.

C'est là que les problèmes arrivent. Tout d'abord en python il n'est pas possible de transformer un nombre négatif en caractère. Il faut que je transforme les valeurs négatives en valeurs positives, donc des entiers non signés sur 16 bit, tout en conservant leurs écritures hexadécimales. Pour ce faire je fais l'opération suivante : 65535+val avec val : la valeur comprise entre 0 et -32767.

Une fois cette étape passer, je transforme le nombre en caractère ce qui ne pose aucun problèmes. C'est à l'envoie que le réel problème commencent : le format. Par défaut PyVisa envoie les caractères en ASCII qui ne peut pas coder des caractères dont la valeur décimale est inférieure à 128. Il faut trouver le bon format d'encodage pour envoyer les données. Après de multiple tentative j'ai décidé d'utiliser une méthode peut conventionnel. J'ai tout simplement testé chaque encodage supporter par python un par un. Bien sûr je ne m'attardais pas sur les formats 8 bits qui ne peuvent, en aucun cas, marcher. A ma grande surprise aucun format n'a pu encoder tous les caractères obtenus. J'ai alors eu l'idée de remplacer les caractères qui n'était pas supporté par d'autres d'une valeur proche qui pouvait être codé. Si vous le voulez, le code python est disponible en annexe n°2. La solution n'est surement pas la plus optimale mais c'est la seule que j'ai trouvé après plus d'une semaine de recherche. C'est

pour moi un parfais exemple d'une des contraintes que j'avais énoncé à savoir l'utilisation du langage python.
Enfin cette phase de test a pour but de vérifier la syntaxe des requête récupérer. Par exemple la requête MEAS:VOLT ? est valide tandis que MEAS: VOLT ? apporte une erreur. La seule différence entre ces deux requêtes est l'espace entre MEAS: et VOL ?

Une fois cette étape terminée je passe à la partie programmation.

3. Réalisation du programme

La partie programmation n'a pas en soit poser énormément de problèmes mais à pourtant était très enrichissante pour moi. Elle se décompose en plusieurs étapes qui sont apparues au fur et à mesure que les programmes à réaliser se sont complexifiés.

La première étape consiste à créer le visuel de l'interface graphique sur papier. Cette étape peut sembler optionnelle étant données que les programmes que je fais ne sont pas de gros logiciels. Ils se sont néanmoins imposer petit à petit comme nécessité pour plusieurs raison.
La première et la principale est que je ne suis pas l'utilisateur du programme. Je dois donc prendre en compte le fait que la personne derrière l'écran n'a pas la même vision que moi sur le programme. Ainsi ce que je trouve intuitif ne l'est pas pour quelqu'un d'autre. Avoir l'interface sur papier me permet alors de me mettre dans la peau d'un utilisateur lambda est de voir si ce que l'interface est intuitive ou non.
De plus avoir cette version papier permet de rendre le développement du code plus rapide. Mes idées sont déjà posées sur le papier et je n'ai donc pas à réfléchir sur les différents widgets (parties d'interface graphique comme les boutons) à positionner. En m'enlevant cette tâche je peux consacrer plus d'attention sur les différents liens logiques entre les widgets et la résolution des problèmes rencontrés.

Une fois l'interface graphique dessiné, je détermine les classes qui vont constituer mon programme. Ainsi chaque classe constitue une partie bien spécifique de l'interface graphique.
En Python et en langage orienté objet en générale, une classe est un outil de programmation qui regroupe plusieurs fonctionnalités et plusieurs variables. L'intérêt est d'une part de séparer le programme en plusieurs parties indépendantes mais également d'éviter les variables globales (variables qui sont disponibles dans tout le programme).

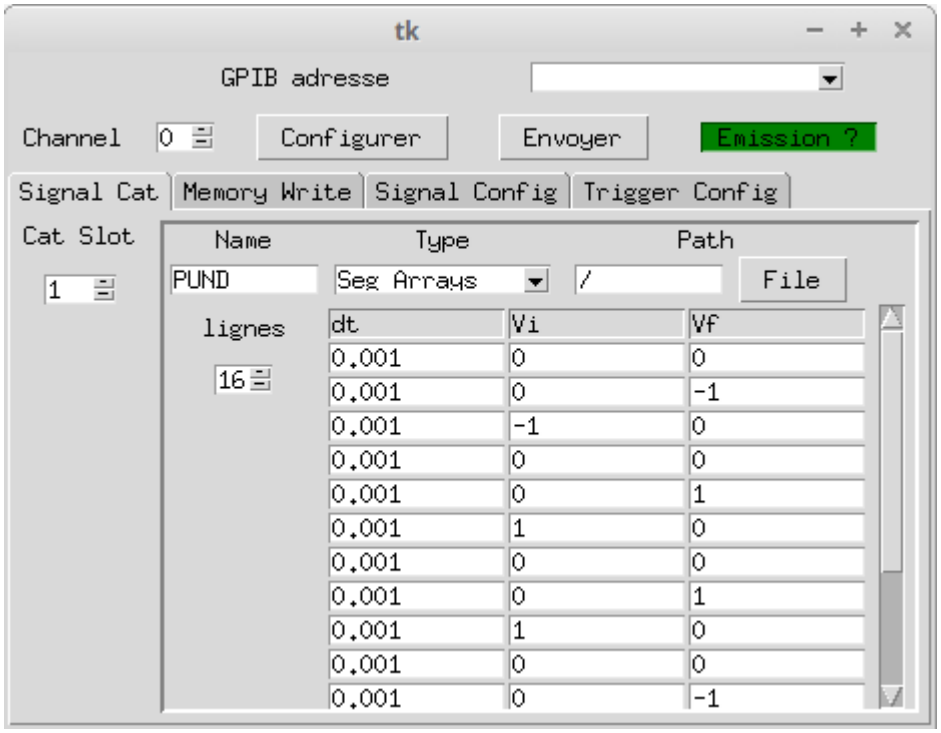


Figure 13: Exemple d'un tableau inséré dans une interface tkinter.

Dans l'exemple ci-dessus, le programme est composé des 5 classes différentes, quatre pour les différents onglets et une pour contenir les onglets et les paramètres en haut de l'interface. Cette disposition permet une programmation bien plus simple et bien plus souple.

Une classe s'articule autour de méthodes (fonctions à l'intérieur de classes) qui ont chacune un rôle à jouer.

Voici comment s'articule une classe de manière générale :

```
class NomClasse (object):
    """Présentation de la classe"""

    def __init__(self, arg1):
        """Constructeur de la classe"""

        ###initialisation de la classe###


    def methode1(self):
        """Première méthode de la classe"""

    #
    #
    #
    #
    #

    def methodeN(self):
        """Dernière méthode de la classe"""

maClasse =NomClasse(arg1 ="création de la classe")
```

Figure 14: Structure classe d'une classe en Python.

On initialise d'abord la classe en lui donnant un nom. La première méthode est le constructeur, il est appelé à la création de l'objet est permet d'initialiser tous les paramètres

nécessaires au bon déroulement fonctionnement de la classe. Viens ensuite les méthodes dont le nombre peut varier et qui permet de gérer la classe. Ensuite pour utiliser cette classe il suffit de l'appeler dans le corps principal du programme.

La variable arg1 est appeler paramètre de la classe. Il permet de passer un paramètre externe à la classe utile lors de l'initialisation.

Mes programmes ce sont donc structurées autour de ces classes chacune ayant sont rôles à jouer dans le programme. Cependant certaines classes ont étaient plus simple que d'autre à mettre en œuvre. Les classes les plus compliquées à développer sont celles contenant des tableaux. La librairie tkinter utiliser pour créer les interfaces graphiques ne propose aucun widgets de type tableau. Il faut donc trouver d'autre widget pour les remplacer.

Une solution souvent utilisée que j'ai retenue est de faire une succession de zone de saisie disposée en tableau. La création du tableau n'est pas un problème en soit mais la gestion de celui-ci deviens très vite un véritable casse-tête. Le nombre de variables et de listes ainsi que le nombre de méthodes devient très vite important. A titre d'exemple, une classe contenant un tableau peut très facilement atteindre les 400 lignes de codes tandis qu'une classe avec des widgets classiques font entre 150 et 250 lignes pour les plus grosses.

Une méthode que j'ai réussis à trouver pour faire face à ces classes est tout simplement de noter chaque variable et chaque méthode ainsi que leurs utilités dans la classe. Avec cette méthode la programmation est plus simple même si ça reste un véritable casse-tête. Si vous voulez voire à quoi peut ressembler ce genre de classe je vous réfère à l'annexe n°3

Une fois l'interface graphique terminé, je réalise les méthodes qui vont communiquer avec l'appareil GPIB en question. La première chose à faire lors de cette étape est d'ajouter à l'interface un widget de sélection de l'adresse GPIB de l'instrument. L'avantage est que si le laboratoire change d'appareil, l'interface graphique sera prête à l'emploie sans à modifier les paramètres GPIB du programme.

Ensuite je code les méthodes qui enverront les requêtes sur le bus GPIB. Je tiens à préciser qu'ici je parle de méthodes de la classe qui sont des sortes de fonctions propre à la classe. La composition de ces méthodes change en fonction du type de requête présentent dans le bus. Le premier type est l'écriture de paramètres dans l'appareil pour paramétrer un signal par exemple. Le deuxième est l'écriture de données dans l'appareil. C'est ce type de méthodes qui permet d'envoyer les données calculées dans l'exemple du générateur de fonction qui a posé tant de problèmes. Le dernier type regroupe les méthodes de mesures qui permettent l'acquisition de données.

Pour chacun de ces types on peut en tirer un organigramme global ne dépendant pas de la structure du programme qui détermine les grandes étapes. Par exemple l'organigramme d'une méthode d'écriture de paramètres est le suivant

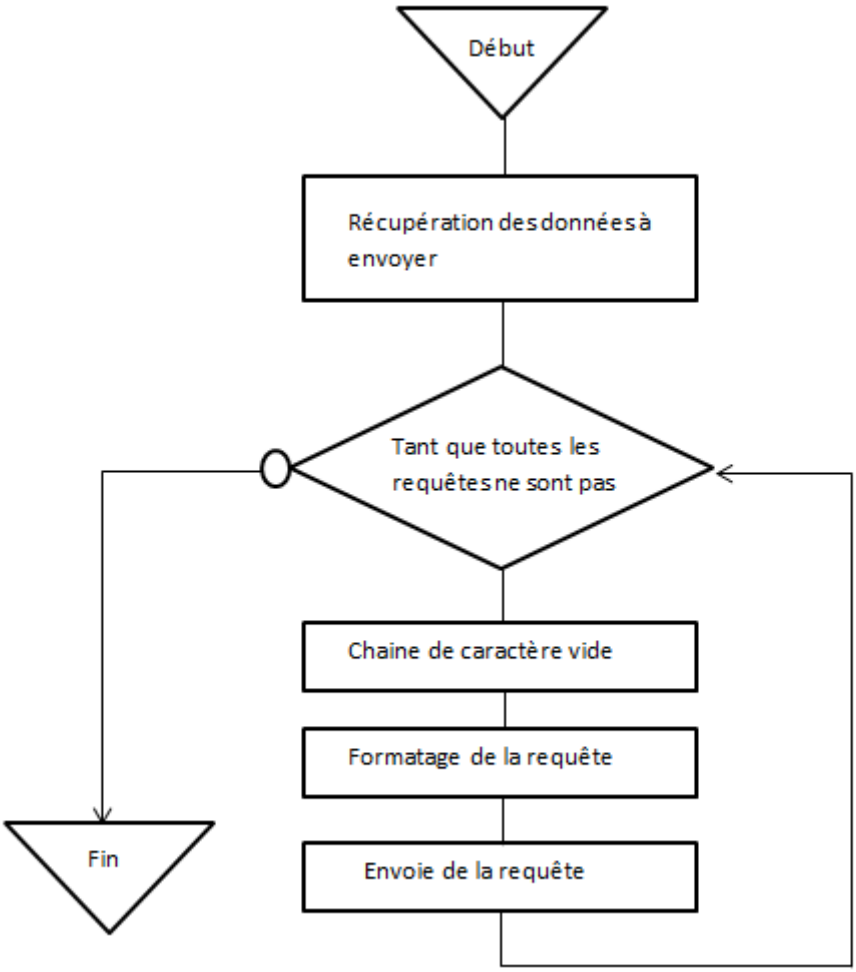


Figure 15: organigramme d'une méthode communiquant avec le bus GPIB

Une fois l'interface graphique totalement terminée, je passe au développement des commandes interpréteurs. Le but de ces commandes est de permettre à l'utilisateur de gérer l'appareil avec des options plus avancées qui aurait surchargé l'interface graphique et aurait complexifiée le code principal. C'est commandes sont généralement écrite dans un fichier à part. Elles sont ensuite importer en début de programme. Une référence sur la classe de gestion des commandes est créée pour permettre à l'utilisateur de saisir les commandes dans l'interpréteur python.

Les fichiers de commandes se décomposent de la même manière que les fichiers d'interface graphique. Les commandes sont regroupées en classe, ensuite une classe générale gère les autres classes. C'est cette classe qui permet d'accéder à toutes les commandes. Une référence sur la ressource GPIB est également disponible permettant, si c'est nécessaire, de piloter le bus directement avec les requêtes GPIB.

Une fois toutes c'est étapes réalisées, le programmes est prêt à être utilisé. Je peux ensuite passé à la phase de test plus ou moins longue suivant la taille du programme et le nombre de problème rencontrés.

4. Phase de test et de débogage

La phase de test et de correction des bugs est la dernière étape de la réalisation technique du programme. C'est durant cette phase que je corrige tous les défauts de programmation comme les erreurs de syntaxes des requêtes GPIB. La phase de test se décompose en deux parties. Chaque partie pouvant être interrompu par la correction des problèmes rencontrés.

La première partie consiste à tester le programme dans des conditions que j'estime « normale » c'est-à-dire sans valeurs extrêmes et sans pousser le programme à ses limites. C'est notamment durant cette phase que je corrige les requêtes GPIB qui déclenche des erreurs. Ensuite je fais plusieurs tests chacun avec des valeurs de paramètres différents pour déterminé si l'appareil réponds bien aux requêtes. Si le programme contient des commandes interpréteur, je les tests après la correction des problèmes de l'interface graphique.

Après avoir terminé cette première phase de test je passe à une utilisation plus extrême. J'envoie les valeurs limites disponibles par l'appareil, pour vérifier qu'il les prend bien en comptes. Je test aussi la fiabilité du programme et surtout de la structure logique.

5. Document d'utilisation

La documentation utilisateurs n'est pas demander à chaque programme. Si le programme est entièrement graphique ou qu'il ne comporte que quelques fonctions. En effet si l'interface graphique est bien faite, un simple coup d'œil permet à l'utilisateur de comprendre comment le programme s'utilise. Dans le cas où le programme n'a pas d'interface, c'est que celui-ci est assez simple pour n'être utilisé qu'avec des fonctions dans l'interpréteur python. Par exemple le programme de l'Agilent ne comporte que 2 fonctions. Un simple regard sur la doc string de la fonction, c'est-à-dire la présentation de la fonction, permet de comprendre comment elle s'utilise. En fait la documentation n'est réellement utile que lorsque des commandes interpréteur sont présentent. Le document permet alors d'expliquer comment fonctionne le programme et le commandes interpréteur. Elle se compose en générale de trois parties plus les annexes.

La première est l'introduction du programme qui présente le programme. Cette partie permet d'introduire le programme, ainsi que ses spécificités. Il explique le rôle global du programme et quel est l'appareil que contrôle ce programme. Il me permet également de données les consignes de mise en marche du programme qui ne sont pas forcément connu pour une personne n'ayant jamais touché au python. En effet il faut faire attention à la version

de python utilisée pour avoir à disposition les librairies utilisées par le programme. L'ouverture du programme se fait ensuite par importation : >>> *import Programme*

La deuxième partie concerne l'utilisation du programme avec l'interface graphique. Elle détails les étapes à réaliser pour effectuer certaines opérations comme des mesures ou l'écriture de certains paramètres. Même si l'interface est normalement assez bien construite pour rendre son utilisation simple, le but d'une documentation est de présenter le programme et son utilisation à l'utilisateur. L'interface graphique faisant partis du programme, il est donc normale de retrouver des explications de celle-ci.

C'est la dernière partie qui est la plus importante pour l'utilisateur. Elle donne des explications sur la partie commandes interpréteur du programme. Cette partie nécessitant des connaissances en python, elle doit donc être la plus claire possible. Dans cette partie je ne détails pas l'utilisation de chaque commandes disponibles, ce serai trop long. Je montre à l'utilisateur les commandes à utiliser pour effectuer des opérations basiques comme des mesures. Si cela vous intéresse vous pouvez regarder un extrait de cette partie à l'annexe n°4. Cela permet à l'utilisateur de se familiarisé avec le système de commande et de réaliser des actions simple avec celles-ci.

Je termine enfin ces documentations par quelques annexes. Dans ces annexes on y retrouve la liste exhaustive des commandes interpréteur avec la doc string de la commandes et les paramètres qu'elle prend en entré. La seconde annexe est la liste également exhaustive des liens des différentes documentations d'installation de librairies ou de drivers. On y retrouve également la documentation générale de Python3 qui est la version python utilisé pour faire tourner les programmes.

Je vous ai maintenant présenté ma méthode de travail que j'ai appliqué tout au long du stage pour mener à bien ma mission. Mon stage étant sur le point de se terminé, je peux maintenant faire un bilan de ces semaines de stage, même partiel.

V. Bilan de la mission

Je vais maintenant vous exposer le bilan du travail que j'ai réalisé durant ce stage au sein du laboratoire de l'INL. Je vais donc vous exposer l'état d'avancement de ma mission, les difficultés générales rencontrées durant mon stage. Je terminerai par vous donner mes idées qui pourront permettre d'améliorer le travail que j'ai fournis.

Aujourd'hui, les scripts du banc de mesures sont tous terminés pour la plupart. Le seul programme restant à faire est la gestion de l'oscilloscope Integra 40. Le programme a pris un peu de retard car un thésard a besoin du banc de mesure pour réaliser des manipulations sur ses échantillons. J'ai donc dû arrêter le test des commandes GPIB pour lui laisser la place. Cependant ce script sera terminé d'ici quelques jours si tous se passe comme prévus.
Il reste également un dernier programme de traitement des données à réaliser qui me semble à première vue relativement long à faire. Le programme est composé d'une interface graphique assez imposante et utilise de nombreux algorithmes mathématiques. La programmation de ce dernier programme risque donc d'être relativement longue et difficile avec tkinter.
Dans tous les cas, même si je ne sais pas trop si ce chiffre est le bon, celons moi le travail qui m'a était demandé est terminé à 85 %. J'accorde au 15 % restant, 5 % du script de l'oscilloscope et 10 % du programme de traitement des données.

Le retards sur l'avancement de la mission qui m'a était confié est dû à de nombreuses difficultés auxquelles j'ai fait face. De manières générales les problèmes rencontrés sont surtout dû aux problèmes d'encodage des caractères. Lors de mon apprentissage de Python, je ne m'étais pas attardé sur ce chapitre pensant que je n'aurai jamais à faire face à ce genre de problème et c'était une erreur. Les problèmes d'encodage peuvent apparaître dès que deux machines communiquent entre elles. Le script ayant posé le plus de problème étant celui du générateur de fonctions dont la conception a duré un peu plus de deux semaines. Le problème d'envoie des données que j'ai exposé durant la partie test des commandes GPIB et l'interface graphique conséquente à réaliser m'a forcer à faire évoluer mes méthodes et à persévérer dans l'effort.

Aux termes de ce stage, je pense avoir fournis un travail de bonne qualité et utilisable mais qui peut néanmoins être amélioré. Certaines pistes d'amélioration peuvent être intéressantes a creusé celons moi.
La première est la gestion des tableaux sur tkinter. Je pense qu'il doit y avoir des moyens plus simples que ceux que j'ai proposés pour gérer les tableaux de données. L'avantage d'une amélioration sur ce point serai de trouver un code plus lisible est plus simple. Il pourrait également être plus rapide à l'exécution.
Un autre point à améliorer sera l'insertion de graphique dans les interfaces. En effet l'insertion de graphique dans tkinter est possible mais néanmoins compliqué et offre des graphiques d'une qualité discutable.
Les interfaces graphiques peuvent donc être améliorées mais la gestion GPIB est quant à elle parfaitement fonctionnel. On pourra également envisager d'améliorer la gestion des appareils sur le banc de mesures en ajoutant de nouvelles fonctionnalités au programme.

Je viens donc de vous exposer le bilan technique de la mission qui m'a était confiée durant mon stage. Cependant, un autre bilan peut être fait, un bilan plus personnel sur ce que le stage m'a apporté.

VI. Bilan personnel

Après avoir établis un bilan de l'état actuel de ma mission, je tiens maintenant à vous faire part des différents apports durant ce stage et ma formation à l'IUT en générale.

La formation que j'ai suivis pendant deux ans à l'IUT GEEI de l'université Lyon1 m'a permis d'acquérir énormément de connaissance dans plusieurs domaines aussi bien technique que relationnelle. Ma formation m'a permis d'acquérir de solides connaissances techniques dans différents domaines. Me donnant ainsi un bagage technique s'étalant sur plusieurs domaines allant de l'énergie à l'informatique. Avec les nombreux projets réalisés pendant ma formation, j'ai également développé un solide esprit d'équipe qui me sera utiles durant toutes mes études et ma carrière.

Le stage quant à lui n'a fait que renforcer les connaissances déjà acquise durant ma formation et d'en développer d'autre. Ma plus grande évolution étant ma méthodologie. J'ai pu me confronter à des problèmes réels et chercher les solutions par moi-même. Le stage m'a permis de mettre mes connaissances à rude épreuve mais n'a fait que développer ma persévérance. Ce stage m'a également permis d'avoir une entrevue du monde de la recherche dans lequel j'aimerais travailler plus tard.

Au final ce stage m'a surtout servis, en plus de consolider les acquis de ma formation actuelle, de me confirmer dans ma poursuite d'étude. Il m'a montré qu'en informatique rien n'est jamais acquis ce qui me donne l'envie d'en apprendre plus. Mon projet de poursuite d'études en licence informatique puis en master puis en thèse c'est confirmer grâce à ce stage.

VII. Annexes

Sommaire

Annexe n°1 : Listes des Acronymes

- CNRS : Centre National de la Recherche Scientifique
- DE : Dispositif Electronique
- ECL : Ecole Centrale de Lyon
- GEII : Génie Electrique et Informatique Industriel
- GPIB : General Purpose Interface Bus
- IEEE : Institute of Electrical and Electronics Engineers
- INL : Institut des Nanotechnologies de Lyon
- INSA : Institut National des Sciences Appliquées
- NI : National Instrument
- PIP : Pip Installs Packages
- SCPI : Standard Commands for Programmable Instruments
- VISA : Virtual Instrument Software Architecture

Annexe n°2 : Code Python d'envoie

```
def sendConf(self):
        """Méthode qui permet de configurer un signal"""

        if self.device == None:
            print("Err: Aucun appareil GPIB connecter")
            return -1

        ###récupération des paramètres à envoyer###
        channel =str(int(self.SPch.get()))
        amp =str(float(self.sigConf.SPamp.get()))
        frq =str(float(self.sigConf.SPfreq.get()))
        off =str(float(self.sigConf.SPoff.get()))
        phs =str(float(self.sigConf.SPphs.get()))
        func =self.sigConf.fct.index(str(self.sigConf.CBfct.get()))+1
        memSlot =str(int(self.sigConf.SPmemSlot.get()))
        duty =str(float(self.sigConf.SPdut.get()))
        imp =str(float(self.sigConf.SPoutImp.get()))
        trtf =str(float(self.sigConf.SPtrtf.get()))

        ###envoie des paramètres sur le bus GPIB###

        ch =''
#création d'une chaine de caractères vide

        ##on configure la première requête en fcontion du numéros de voie##
        if channel == '0':
            ch ="CHA 1;CPL 1"
        else:
            ch ="CHA {0};CPL 0".format(channel)

        self.device.write(ch)
#écriture de la requête sur le bus

        ch =''                                                          #on
remet à zéros la requête

        ##configuration de la requête d'envoies des paramètres spécifique##
        ch ="FNC {0}".format(func)                                      #type
de fonction

        if func == 3:       #type Fsquare
            ch +=";DTR {0}".format(trtf)
        elif func == 6:     #type User
            ch +=";AFN {0}".format(memSlot)
        elif func == 7:     #type Vsquare
            ch +=";DTU 0;DTY {0};DTR {1}".format(duty, trtf)

        self.device.write(ch)                                           #envoie
de la requête

        ch =''                                                          #reset
de la chaine

        ##configuration de la derniere requête ##
        if 45<eval(imp) and eval(imp)<999:
#impédance de sortie
            ch ="OLS 1;OLD {0}".format(imp)
```

```
        else:
            ch ="OLS 0"

        ch +=";FRU 0;FRQ {0};AMU 0;AMV {1};OFU 0;OFS {2};PHU 0;PHS
{3}".format(frq, amp, off, phs)

        self.device.write(ch)                                    #envoie
de la requête sur le bus
```

Annexe n°3 : Exemple d'une classe de gestion d'un tableau en tkinter

```
class SigCat (Frame):
    """Classe de l'onglet n°1 qui correspondant a la listes des signaux et
a leurs tableaux de valeurs"""

    def __init__(self, boss =None):
        """Constructeur de la classe"""

        super().__init__()          #init de la mère


        self.data =[]                #tableau des données entrées

        ###récupération des données des anciens tableaus###
        with open("tab.txt", 'r') as file:
#on ouvre le fichier qui contiens les données des tableaux

            recup =file.readlines()
#on récupères les tableau lignes par lignes (1 ligne =1 tab)

        if recup == []:
#s'il n'y avait aucun tableau
            self.data =[ [ ['', 'Seg Arrays', 4], ['dt', 'Vi', 'Vf'] ],
#on crée le tableau de données
                         [ ['', 'Seg Arrays', 4], ['dt', 'Vi', 'Vf'] ],
                         [ ['', 'Seg Arrays', 4], ['dt', 'Vi', 'Vf'] ] ]
        else:
#sinon on reconstruit le tableau

            for tab in recup:
#on parcours les tableaux dans la listes des tableaux
                dta =[]
#on créer un tableau vide
                for case in tab.replace('\n', '').split('/!'):
#on parcours les cases dans le tableau récupérer

#(case séparer par des /!
                    c =[]
#on créer une liste des données de chaque case
                    for d in case.split('/'):
#on parcours les données dans chaque case

#(données séparer par des /
                        try:
                            c.append(eval(d))
#on essaie de transformer la chaine en son équivalent
                        except:
                            c.append(d)
#si on y arrive pas on rajoute la données brute (données str)
                    dta.append(c)
#on ajoute la case dans le tableau

                del(dta[len(dta)-1])
#on supprime la dernière case (vide jsp pk)
                self.data.append(dta)
```

```
#on ajoute la tableau a la liste des tableau


        self.case =[[]]                 #tableau qui sauvegarde les
références sur les cases du tableau
        self.currentTab =self.case[0]   #on initialise un tableau de départ
        self.infoTab =[]                #tableau qui contient les
informations sur les différents tableaux

        self.tabFr =Frame(self, borderwidth=2, relief=SUNKEN)   #on crée un
frame spéciale pour les informations du tableau actuel

        ###création des outils pour le scrolling###
        can =Canvas(self.tabFr, width =270, height =200)
#paramétrage du canvas qui contiendras le tableau
        can.grid(row =3, column =1, columnspan =10, sticky ="news", rowspan
=20)

        vsb =Scrollbar(self.tabFr, orient =VERTICAL, command =can.yview)
#création d'une scrollbar verticale
        vsb.grid(row =3, column =13, sticky =N+S, padx =5, rowspan =20)

        can.configure(yscrollcommand =vsb.set)
#on relie la scrollbarre au canvas


        ###sélection du tableau a afficher###
        Label(self, text ="Cat Slot").grid(column =0, row =0, padx =5,
columnspan =2)                    #texte indicatif

        self.SPtabl =Spinbox(self, from_ =1, to =20, increment =1, width
=3, command =self.changeTab)    #spinbox de sélection
        self.SPtabl.grid(column =0, row =1, padx =5, columnspan =2)
#affichage


        ###création d'une frame pour contenir le tableau a l'intérieur###
        self.fr =Frame(can)


        ###sélection du nombre de lignes dans le tableau###
        Label(self.tabFr, text ="lignes").grid(column =0, row =3)
#texte informatif

        self.SPligne =Spinbox(self.tabFr,from_ =1, to =50, increment =1,
#spinbox de sélection du nb de ligne
                              width =2, command =self.configTab)
        self.SPligne .grid(column =0, row =4, pady =2)
        self.SPligne.bind('<Return>', self.configTab)
#on permet la saisie par clavier

        self.SPligne.delete(0, 2)
        self.SPligne.insert(0, str(self.data[0][0][2]) )
#on insert 4 comme valeur par défaut


        ###sélection du nom du tableau###
        Label(self.tabFr, text ="Name").grid(column =0, row =0)     #texte
indicatif
```

```
        self.INname =Entry(self.tabFr, width =10)                    #zone
de saisie du nom du tableau
        self.INname.bind("<Return>", self.configTab)                 #on
autorise la réception des event type Return (touche Entrée)
        self.INname.insert(0, self.data[0][0][0])                    #on
insert le nom du tableau s'il en a un
        self.INname.grid(column =0, row =1, padx =2)


        ###sélection du chemin du fichier pour les fichier de points###
        Label(self.tabFr, text ="Path").grid(column =2, row =0, columnspan
=2)  #texte indicatif

        self.INpath =Entry(self.tabFr, width =10)
#zone de saisie
        self.INpath.insert(0, '/')
#on insert le caractère \ au début
        self.INpath.bind('<Return>', self.chargData)
        self.INpath.grid(column =2, row =1, padx =5)

        self.BTpath =Button(self.tabFr, text ="File", command
=self.recPath)    #on créer un bouton permettant de choisir
        self.BTpath.grid(column =3, row =1)
#ce fichier de points


        ###sélection du type de données dans le tableau###
        Label(self.tabFr, text ="Type").grid(column =1, row =0)
#texte indicatif
        self.CBtype =ttk.Combobox(self.tabFr, width =13)
#combobox de sélection du type
        ty =["Seg Arrays", "Points Arrays", "Points File"]
        self.CBtype["values"] =tuple(ty)
#on ajoute les valeurs dans la box
        self.CBtype.current(ty.index(self.data[0][0][1]))
#valeur courante
        self.CBtype.bind("<<ComboboxSelected>>", self.configTab)
#on ajoute une fonction lors de la validation d'un choix
        self.CBtype.grid(column =1, row =1, padx =5)


        self.infoTab.append([4])
        self.drawTab(self.data[0][0][2], 1)

        can.create_window(0, 0, window=self.fr)
        self.fr.update_idletasks()
        can.config(scrollregion=can.bbox("all"))

        self.tabFr.grid(column =7, row =0, rowspan =20, padx =5, pady =5)


    """Méthode de gestion de la classe"""

    def drawTab(self, nbLigne, nTab):
        """Méthode qui dessine le tableau"""

        i =0
#compteur

        for case in self.case[nTab-1]:
#parcours des cases correspondantes au numéros du tableau
```

```
            case.destroy()
#on supprime les case
        self.case[nTab-1] =[]
#on vide le tableau des références

        for name in self.data[nTab-1][1]:
#on parcours la liste des nom des colonnes du tableau
            entree =Entry(self.fr, width =12)
#on créer une case avec une zone de saisie
            entree.insert(0, name)
#on insert le nom de la colonne
            entree.config(state ='disabled', disabledforeground ="black")
            entree.grid(column =i, row =0)
#on affiche la case au bon endroit
            self.case[nTab-1].append(entree)
#ajout de la référence de la case dans la liste
            i +=1

        if nbLigne != self.data[nTab-1][0][2]:
#si le nombre de ligne a changer
            self.data[nTab-1][0][2] =nbLigne
#on remet à jour l'information

        for ligne in range(0, self.data[nTab-1][0][2]):
#parcours des lignes

            for colonne in range(0, len(self.data[nTab-1][1])):
#parcours des colonnes
                entree =Entry(self.fr, width =12)
#on créer la zone de saisie: case

                try:
                    entree.insert(0, self.data[nTab-1][ligne+2][colonne])
#on essai d'insérer une valeur dans cette case
                except:
                    entree.insert(0, '')
#si on peut pas on insert rien

                entree.grid(column =colonne, row =1+ligne)
#on affiche la case a la bonne position
                entree.bind("<Return>", self.recupData)
#on ajoute la détection
                entree.bind("<Tab>", self.recupData)
#on ajoute la détection des tabulation pour changer de case
                self.case[nTab-1].append(entree)
#on ajoute la case dans le tableau
        self.currentTab =self.case[nTab-1]
#on actualise la tableau actuel



    def configTab(self, event =None):
        """Méthode de configuration du tableau lors d'un changement de
paramètre"""

        nTabl =int(self.SPtabl.get())                    #on récupère le
numéros du tableau
        nLigne =int(self.SPligne.get())                  #on récupère le
nombre de ligne du tableau
        typ =self.CBtype.get()                           #on récupère le
```

```
type de tableau
        name =self.INname.get()                             #on récupère le nom
du tableau


        if nTabl>len(self.data):                            #si on a un nouveau
tableau
            self.data.append([ [' ', 'Seg Arrays', 4] ,['dt', 'Vi', 'Vf']
]) #on en créer un nouveau

        if typ != self.data[nTabl-1][0][1]:
#si on change de type de tableau

            if typ != "Seg Arrays":
#si le type n'est pas un tab de seg
                self.data[nTabl-1] =[ [' ', typ, self.data[nTabl-1][0][2]],
['t', 'V'] ]            #on reconfigure l'entête du tab

                if typ == "Points File":                    #si c'est un
fichier de point
                    self.chargData()                        #on entre dans la
fonction de chargement des données

            else:
                self.data[nTabl-1] =[ [' ', typ, self.data[nTabl-1][0][2]],
['dt', 'Vi', 'Vf'] ]     #si c'est un tab de seg

        self.data[nTabl-1][0][0] =name                      #on met à jour le
nom du tableau


        #print(len(self.data[nTabl-1]), nLigne)

        if nLigne<len(self.data[nTabl-1])-2:                #si on enlève des
lignes
            del(self.data[nTabl-1][nLigne+2])               #on supprime la
ligne en trop

        self.drawTab(nbLigne =nLigne, nTab =nTabl)          #on redessine le
tableau




    def changeTab(self, event =None):
        """Méthode qui permet de change le tableau afficher"""

        ###a modifier###

        self.SPligne.delete(0, 2)
        tab =int(self.SPtabl.get())

        typ =["Seg Arrays", "Points Arrays", "Points File"]

        if tab>len(self.data):
            self.data.append([ [' ', 'Seg Arrays', 4], ['dt', 'Vi', 'Vf']
])

        #print(tab.index(self.data[tab-1][0][1]))
```

```
        try:

            for case in self.currentTab:                                    #on
parcours les case du tableau pour les détruires
                case.destroy()

            self.SPligne.insert(0, str(self.data[tab-1][0][2]))             #on
insert le nombre de ligne dans la spinbox
            self.INname.delete(0, 15)                                       #on
elève l'ancien nom
            self.INname.insert(0, self.data[tab-1][0][0])
#pour mettre le nouveau
            self.CBtype.current(typ.index(self.data[tab-1][0][1]))          #on
actualise le type de tableau
            self.drawTab(self.data[tab-1][0][2], tab)                       #on
redessine alors le tableau

        except:     ###si l'une des étape précédente à échouer###

            for case in self.currentTab:    #on détruit les cases
                case.destroy()

            self.case.append([])            #on ajoute une nouvelle liste
de case
            self.drawTab(4, tab)            #on redessine le tableau



    def recupData(self, event):
        """Récupère est stocke les données saisie"""

        tab =int(self.SPtabl.get())                                         #on
récupère le numéros du tableau



        d =[ self.data[tab-1][0], self.data[tab-1][1] ]                     #on
formate l entête des données

        col =len(self.data[int(self.SPtabl.get())-1][1])                    #on
récupère le nombre de colonnes du tableau

        for case in range(col, len(self.currentTab), col):                  #on
parcours toute les cases de la première ligne a la dernière
            r =[]                                                           #on
créer une liste pour récupèrer les données de la ligne
            for i in range(0, col):                                         #on
parcours les case de la ligne
                try:
                    r.append(eval(self.currentTab[case+i].get()))           #on
ajoute la données de la case i
                    self.currentTab[case+i].configure(bg ="white")          #on
met le fond de la case en blanc
                except:                                                     #si
il n'y a pas de données a récupérer
                    r.append('')                                            #on
ajoute un vide
                    if 'Return' == event.keysym:
                        self.currentTab[case+i].configure(bg ="#FAB9B9")#on
configure le fond de la case en rouge pâle
```

```
            d.append(r)                                                    #on
ajoute les données de la ligne

        self.data[tab-1] =d                                                #on
remplace les anciennes données par les nouvelles

        ###sauvegarde des données dans le fichier###
        with open("tab.txt", "w") as file:      #ouverture du fichier
            ecri =''                            #on crée un chaine de
caractère vide
            for tab in self.data:               #on parcours les tableaux
dans la liste de stableaux
                writ =''                        #on crée une chaine vide
pour récupérer les données du tableau
                for t in tab:                   #on parcours les cases du
tableau
                    r =''                       #chaine vide pour contenir
les données dans les cases
                    for d in t:                 #on parcours les données
dans les cases
                        r +=(str(d)+'/')        #on récupères la donnée et
on rajoute le séparateur /

                    writ +=(r+'!')              #on rajoute ensuite la case
au tableau abec le séparateur !
                ecri +=(writ+'\n')              #on écrit le tableau
complet et on rajoute un saut de ligne
            file.write(ecri)                    #a la fin on écrit les
tableaux dans le fichier



    def recPath(self):
        """Méthode qui permet de récupérer le chemin d'un fichier"""

        #création du gestionnaire de fichier
        path =askopenfilename(title ="Point File", filetypes =[ ("text
file", ".txt"), ("all", ".*")])

        self.INpath.delete(0, len(self.INpath.get()))   #on supprime le
contenue dans la zone de saisie du chemin
        self.INpath.insert(0, path)                     #pour ajouter le
nouveau

        if self.CBtype.get() == "Points File":          #si le type de
tableau est déjà mis en tant que Points File
            self.chargData()                            #on charge les
données dans le tableau directement



    def chargData(self, event =None):
        """Méthode qui permet de charger les données d'un fichier dans le
tableau"""

        path =self.INpath.get()                                 #on
récupère le chemin du fichier
        tab =int(self.SPtabl.get())                             #et le
numéros du tableau
```

```
        data = [ self.data[tab-1][0], self.data[tab-1][1] ]      #on
configure l'entête des données récupérées

        if path =='' or path =='()' or path == "/":              #si le
chemin est invalide
            return                                               #on sort
directement

        try:
            with open(path, 'r') as file:                        #on tente
d'ouvrir le fichier
                dat =file.readlines()                            #on
récupères les données ligne par lignes

        except:                                                  #si on ne
peut pas ouvrir le fichier
            print("impossible d'ouvrir le fichier")              #on
préviens l'utilisateur
            return -1                                            #et on
quitte la méthode

        for d in dat:                                            #on
parcours les lignes de donées récupérées
            data.append(d.replace('\n', '').split('\t'))         #on
transforme chaque ligne en liste en enlevant les \n

        #print(data)

        self.data[tab-1] =data                                   #on met les
données dans la liste de données
        self.SPligne.delete(0, 2)                                #on
supprime le nombre de ligne actuelles
        self.SPligne.insert(0, len(dat))                         #pour le
remplacer par le nouveau

        self.drawTab(len(dat), tab)                              #on
redessine ensuite le tableau
```

Annexe n°4 : Extrait de documentation utilisateur

III. Commande shell

Il est également possible de réaliser des mesures grâce aux commandes shell. Deux mesures sont alors disponibles, une mesure à fréquence variable et a tension fixe et une mesure totalement personnalisée ou chaque points de mesures comporte un trio de valeurs Tension Courant Fréquence.

Avant d'aller plus loin sachez que même si les mesures se lancent par commande directement dans le shell, **la configuration de la mesure dans l'interface reste néanmoins nécessaire.**

Pour lancer une mesure en Fréquence, dans le shell taper la commande *commande.freqMeas(...)*. Cette méthode prend plusieurs paramètres :

- freq_start : fréquence de départ, doit être compris dans l'intervalle [20 Hz, 1 MHz]
- freq_stop : fréquence finale, doit être compris dans l'intervalle [20Hz, 1 MHz]

Attention freq_start doit être inférieur à freq_stop

- ptsPerDec : nombre de point de mesure par décade. Si une fréquence calculer est en dessous de freq_start ou au-dessus de freq_stop, le point de mesure est alors ignoré. PtsPerDec doit être compris dans l'intervalle [1, 10]
- voltage : Tension de mesures constante, doit être compris dans l'intervalle [-40 V, +40 V]

Pour lancer une mesure dont les fréquences sont comprises entre 500 et 785K Hz avec une tension de 20V et 8 points de mesures par décades, il faut donc taper la commande :
commande.freqMeas(freq_start =500, freq_stop =785000, ptsPerDec =8, voltage =20)
La mesure se lancera alors et les résultats seront directement envoyer dans le tableau de l'onglet Résultats

Le lancement d'une mesure personnalisé est quant à lui plus compliquer. Il faut auparavent créer un tableau contenant tous les points de mesures qui seront utiliser. Ce tableau doit être sous la forme [[V1, V2, ….. Vn], [I1, I2, ….., In], [F1, F2, ….., Fn]].
Exemple d'un tableau avec trois points de mesures créé dans l'interprète :

```
>>> tab =[ [1.0, 2.0, 3.0], [1.0e-3, 2.0e-3, 3.0e-3], [1.2e3, 1.35e3, 5e3] ]
```

On a dons ici trois points de mesures :
-1° point : 1.0V 1.0 mA 1.2 kHz
-2° point : 2.0V 2.0 mA 1.35kHz
-3° point : 3.0V 3.0 mA 5kHz

Vous pouvez afficher le tableau pour vérifier que les points de mesures sont correctes. Lancer alors la commande *commande.affTabMeas(tab)* avec tab est le nom de votre tableau. Vous obtenez alors le résultat suivant :

```
>>> tab =[ [1.0, 2.0, 3.0], [1.0e-3, 2.0e-3, 3.0e-3], [1.2e3, 1.35e3, 5e3] ]
>>> commande.affTabMeas(tab)
Tension          Courant          Frequence
1.0              0.001            1200.0
2.0              0.002            1350.0
3.0              0.003            5000.0
```

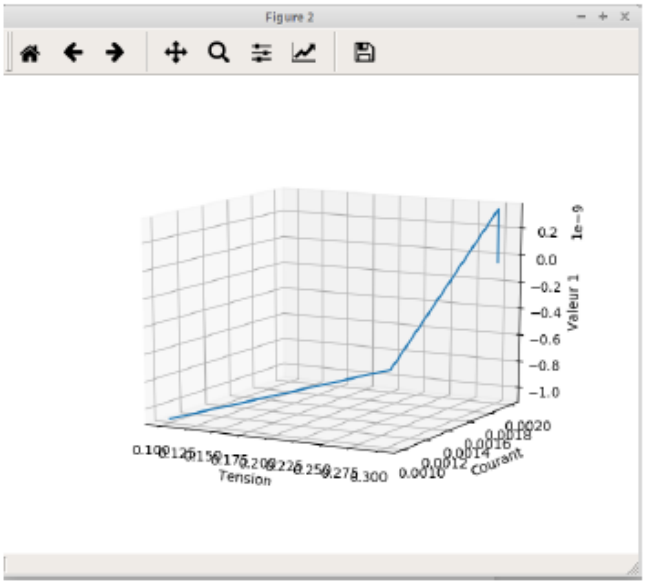
Une fois votre tableau prêt, il ne vous reste plus qu'à taper la commande *commande.measureTab(tab)* pour lancer l'acquisition. Les résultats seront ensuite envoyer dans le tableau de l'onglet Résultats.

Note : Si votre tableau ne comporte pas le même nombre de lignes partout, le programme vous demandera si vous voulez continuer la mesure ou pas. Si vous continuez la mesure, le programme remplacera les points manquant par les dernières valeurs en mémoire.

Il existe enfin une dernière commande permettant de tracer des graphiques en 3 dimensions pour suivre l'évolution d'une mesure en fonction de deux autres paramètres. Si vous souhaitez afficher ce graphique lancer alors la commande *commande.plotMeas(...)*. Cette méthode prends trois paramètres d'entrés : x, y et z représentants les valeurs à afficher sur l'axe x y et z. Ces paramètres peuvent prendre les valeurs suivantes : **Tension, Courant, Frequence, Valeur 1, Valeur 2.**

```
>>> commande.plotMeas(x ='Tension', y ='Courant', z ='Valeur 1')
```

Par exemple pour afficher la valeur 1 en fonction de la tension et du courant taper :
Vous obtiendrai alors la fenêtre avec le graphique.



Vous pourrez alors tourner le graphique en restant appuyer sur le clic droit de la souris.

Bibliographie

- Site de l'INL : http://inl.cnrs.fr/

- Explication du bus GPIB : https://www.positron-libre.com/electronique/protocole/ieee488/cable-ieee488-connecteur-gpib.php et http://www.ni.com/white-paper/12889/fr/

- Documentation des librairies Python utilisées :

    - Tkinter : https://docs.python.org/2/library/tkinter.html
    - PyVisa : http://pyvisa.readthedocs.io/en/stable/api/
    - Gpib_ctypes : https://media.readthedocs.org/pdf/gpib-ctypes/latest/gpib-ctypes.pdf
    - Linux-GPIB : https://linux-gpib.sourceforge.io/

Résumé

Ce rapport traite de la réalisation de programme Python destiné à contrôler différents instruments de mesures. Ces instruments sont reliés à un bus GPIB destiné au contrôle d'instrument. Le bus est ensuite relié à l'ordinateur par l'intermédiaire d'un adaptateur GPIB-USB. Actuellement ces instruments sont contrôlés à partir de scripts développés en LabVIEW. Le problème est que les drivers permettant de communiquer avec le bus ne sont plus/ supporter par les versions récentes de linux. Il est donc nécessaire de se détacher de LabVIEW pour améliorer la gestion du bus. La solution retenue consiste à réécrire les codes LabVIEW en codes Python plus portables. J'ai donc installé un driver GPIB indépendant : Linux-GPIB ainsi que les librairies de contrôle PyVisa-py. Les programmes sont généralement composer d'interface graphique permettant un contrôle simple et efficace de l'appareil. Certains programmes comportent des commandes interpréteur permettant une gestion plus poussé de l'appareil en question. A l'heure actuelle, il ne reste plus qu'un programme à terminer pour pouvoir gérer le bus en entier. Il restera ensuite à programmer le script de traitement des données récupérées sur le bus.

Mots-clés : Informatique, Programmation, Python, Interface graphique, Instrumentation, Réseaux