Mémo Arduino du Fablab : Pour débuter

Cette page est à peu près finie... néanmoins, je sais qu'elle reste à parfaire donc pour toute remarque, suggestion ou question : justin.cano@centrale-marseille.fr

Bonne lecture !

Justin Cano

I – Introduction :

Qu'est ce que c'est ?

Les cartes Arduino sont des cartes électroniques programmables, qui ont la particularité de lire et de générer à la fois des signaux numériques et analogiques. Les sorties étant élaborées en fonction des valeurs d'entrées qui sont les variables du programme.

Quel est l'intérêt alors ? Cela ressemble étrangement à des langages informatiques traditionnels !

Hé bien, chers lecteurs, c'est qu'une Arduino est capable de lire directement des tensions (comme si c'était vous derrière le multimètre !) , de générer du courant analogique par le biais d'un hacheur (PWM) et évidemment des variables booléennes (0 ou 5V) tout cela en restant autonome et au cœur d'un système embarqué (typiquement, un robot). D'autres arguments en sa faveur ? Hé bien le langage de programmation est simple , convivial et surtout en open source ! Et qu'un seul logiciel et un petit câble USB>MiniUSB suffit !

Qu'est-ce qu'on attend donc pour programmer ?

Ça , je vous le demande !! Mais téléchargez d'abord le logiciel :

http://arduino.cc/en/main/software

II – Comment programmer :

A/ Les bases de la syntaxe en Arduino :

On dit qu'un exemple vaut mille mots, donc voici un programme typique, Blink ou clignotement de LED :

```
int led = 13;

void setup() {
  pinMode(led, OUTPUT);
}


void loop() {
  digitalWrite(led, HIGH);
  delay(1000);
  digitalWrite(led, LOW);
  delay(1000);
}
```

On remarque que chaque instruction est composé d'une fonction (*int,pinMode*...) et se termine par un point-virgule (semicolon).

On peut découper le programme en trois grandes structures :

Structure de définition :

La première partie du programme définit comment vous voulez **nommer** les entrées-sorties. « int » en dehors des "*voids*" le permet...

« la patte (ou pin) digitale 13 de mon Arduino s'appellera désormais led par les lois sacrées du programme 😎 *»*

Structure "Void Setup" (initialisation) :

Cette structure entre accolades ne sera exécutée **qu'une seule fois** à la mise sous tension de l'Arduino.

Elle contient également les **déclarations de sortie** : en effet, toutes les pin (pattes) de l'Arduino

sont à défaut **des entrées** et la syntaxe *pinMode(« NomDuPin », OUTPUT)* permet de transformer l'entrée en sortie (respectivement, *INPUT* permet le contraire). L'obligation de déclaration de sortie s'explique par le fait qu'il faut que la carte se débloque de son mode courant faible (en gros vous lui ordonnez d'envoyer de la « puissance », toute relative car les sorties sont limitées à 30mA 😐).

« Led sera considérée comme une sortie digitale (état haut ou bas) »

Structure "Void Loop" (boucle) :

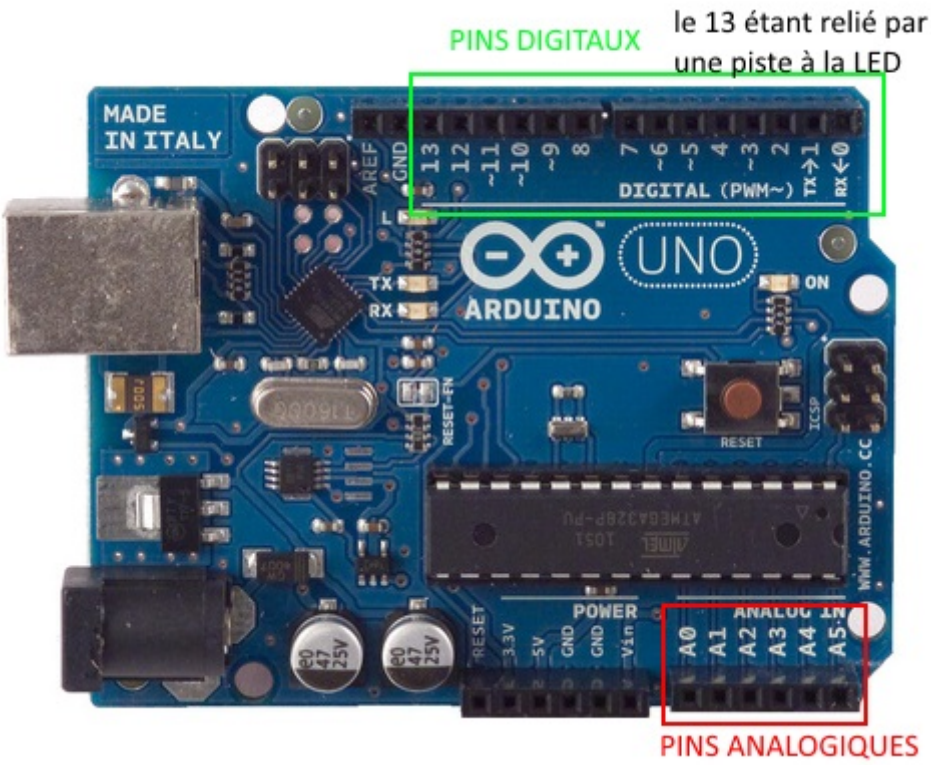
Structure clé de votre programme, cette dernière sera **exécutée en boucle** (loop) jusqu'à la mise hors tension de l'Arduino ou bien de sa reprogrammation.
Ici, on effectue deux fois la commande *delay(1000)* et deux fois la commande *digitalWrite(pin,ETAT)* qui permet de faire basculer une sortie digitale (ou numérique) a un état 1 ou 0...

« [Tartempion] La LED s'allume car digitalWrite(led,HIGH) est synonyme d'état haut pour notre chère diode...
- puis rien ne se passe durant 1000ms = 1s
- ensuite la LED s'éteint : digitalWrite(led,LOW) équivaut à un état bas de la diode
- puis rien ne se passe durant 1s
le programme repart de "[Tartempion]"... et ainsi de suite : c'est un joli clignotant qu'on a là !! 🙂 *»*

B/ Les deux types de pins :

Comme dit ci-dessus, les cartes Arduino servent à être implémentées **dans** des circuits électroniques. Il est donc nécessaire que notre chère carte communique avec le reste du circuit, c'est pour cela que savoir se servir des pins est très important.
Il existe **deux types** de pins servant à cela dans une carte Arduino, quel que soit le modèle :

Les pins analogiques : notés A1, A2... An (n et m dépendent du modèle)
Les pins numériques (ou digitaux): notés tout simplement 1,2,3... m
Une petite localisation de ces derniers sur **l'Arduino Uno** :



Les pins **analogiques** permettent une lecture de tension, une lecture d'état booléen ou une écriture d'état booléen (exemple : mettre la pin led à l'état HIGH).
Les pins **digitaux** quant à eux permettent de lire et d'écrire des états booléens et d'écrire des états analogiques (pour certains dotés de la fonction *PWM*)
NB= nous reviendrons sur ces quatre fonctions d'écriture ultérieurement.

C/ Types de variables :

NB: ceci n'est pas une liste exhaustive, ni rigoureuse, il en existe d'autres, mais pour débuter on se contentera de ces dernières :

int :

Cette catégorie de variable, que vous avez déjà entr'aperçue au B/ sert à définir une variable entière relative (la taille dépend de la carte utilisée, mais l'intervalle est au moins par défaut [-32,768;32,767]).

Comme les **pins** sont repérés par des entiers (oui, 1, **A0** et 3 sont des entiers pour Arduino) elle peut servir à affecter un nom à un pin
exemple: *int led=13*

Mais elle sert à bien d'autres choses...
exemple : un compteur : “
void setup () { int i=0; } mise à zéro du compteur et déclaration de variable
void loop () {
“instruction P dont le nombre d'exécution est à compter” (ex: un demi-tour de moteur pas à pas)

i++ (incrémentation) }
Bref, à chaque fois que l'instruction P sera exécutée, la variable i s'incrémentera, en d'autre termes *i=NombreDExécutions(P)*

Il existe en réalité deux types de variables entières (modes de **int**) une variable int sera par défaut déclarée (suivant le modèle de carte Arduino utilisée) dans l'un des deux :

- **long** qui permettent de définir des entiers relatifs volumineux (inclus dans [-2,147,483,648 ; 2,147,483,647])

- **short** qui définit des entiers inclus dans [-32,768;32,767]

NB: si on veut forcer une variable à être **long** ou **short** il suffit de remplacer **int** dans la déclaration (par **short** par exemple), la syntaxe est la même

float

Son principe d'utilisation est le même sauf qu'on y stocke des réels en point flotant (compris dans [3.4028235E+38 ;-3.4028235E+38]) elle est très utile pour stocker des valeurs de quotients (**ex:** conversion de mesures de tension, que l'on traitera plus tard)

Les calculs en point flotants prennent beaucoup plus de temps. En effet, il faut que le compilateur envoie au programme que des instructions intelligible au microprocesseur.

- Ce dernier est idiot, il ne comprend que les additions/soustractions/multiplications/divisions en nombre entier.
- Il faut donc lui faire replacer la virgule à chaque opération, et cela demande vingt fois plus de temps (ou quarante, enfin bref vous m'avez compris)

Les floats font 32 bits (1 mot simple) :

- 1 pour indiquer le signe du nombre représenté **S**
- 8 pour représenter l'exposant **E** (entier relatif)
- 23 pour représenter la fraction **F** (qui permet
- Le nombre réel vaut alors **(-1)^S * F * 2^E**

A noter que cette représentation mène à des erreurs de précision, la fraction n'étant qu'une approximation du nombre...

double

Pareil que le précédent sauf qu'ici, on travaille avec deux fois plus de bits, soit un double mot de 64 bits. La précision est ainsi doublée, tout comme le temps de calcul hélas...

Pour en savoir plus sur l'encodage des réels, vous pouvez consulter ce site web :
http://www.zentut.com/c-tutorial/c-float/

char

A été crée initialement pour coder un caractère (d'où son nom) en un nombre entier compris entre 0 et 255 (ou entre -128 et 127) :
ex: *char LaLettreMysterieuse='Z'*
est équivalent à *char LaLettreMysterieuse=90*
La ressource pour l'encodage (tableau) est disponible ici
Sinon, il s'agit d'une variable entière stockée sur 8 bits soit 1 octet.

boolean

Permet d'enregistrer une variable booléene :

qui ne peut prendre que les valeurs “HIGH” ou “LOW”

Très utiles pour les expressions logiques !! 😎

D/ Opérateurs élémentaires

Voici un rappel succinct des différents **opérateurs** mathématiques élémentaires utilisés en Arduino :
NB j'en ai “oublié” volontairement certains dont l'emploi nécessite un besoin (trop) spécifique ou/et

1mg de paracétamol... au choix ! 😎

```
= (affectation)
+  (addition)
- (soustraction)
* (multiplication)
/ (division)
% (modulo)
```

opérateurs de comparaison

```
== (égal à)
!= (différent de)
< (strictement inférieur à)
```

```
> (strictement supérieur à)
<= (inférieur ou égal à)
>= (supérieur ou égal à)
```

opérateurs booléens

```
&& (et)
|| (ou)
! (non)
```

opérateurs de composition

Leur emploi est synonyme de paresse ou d'astuce  !

```
++ (incrément)
-- (décrément)
```

ex: *i++; équivaut à i=i+1;*

```
+= (addition composée)
-= (soustraction composée)
*= (multiplication composée)
/= (division composée)
```

ex: *i+=N; équivaut à i=i+N;*

opérateurs bitwise

Ce sont des opérateurs permettent une opération bit à bit et non pas de la valeur globale.

Exemple : un caractère **char** peut être vu comme un tableau de booléen, par exemple un char qui vaudrait 42 en décimal, vaut 0x2A en hexadécimal, et il vaut 0b00101010 en binaire. Maintenant, je veux savoir si ce dernier est divisible par 4 (la réponse est non, on le sait), comment faire pour le prouver ? Il suffit de récupérer le dernier et l'avant dernier bit, si tous les deux valent 0, on a gagné, c'est un multiple de 4 ! Mais... comment on fait pour récupérer les valeurs ? Eh oui cher public, on va utiliser les opérateurs bitwise !

Les shifts (décalage de bits)

```
<< shift vers la gauche
>> shift vers la droite
```

Exemple : 0b10001011«2 = 0b00101100 et 0b11110000»6 = 0b00000011

Shifter « revient à multiplier par deux la valeur représentée (rajouter un zéro, donc ce serait décupler en décimal, ici c'est doubler car on est en binaire) et shifter » revient à diviser (retirer un zéro, en décimal décimer)

On peut supprimer des valeurs avec ces derniers : il faut faire attention à la taille en bits de la variable, un shift de 8 sur un char (de 8 bits) peut être vu comme un écrasement total de sa valeur et ce quel que soit son sens 

```
| OU bitwise, ex : 0b1010 | 0b0001 = 0b1011
& ET bitwise, ex : 0b0101 & 0b1100 = 0b0100
^ NOR (NON OU) bitwise, ex : 0b1010 ^ 0b1011 = 0b0001
~ NON bitwise, ex : ~0b1011 = 0b0100
```

Operations composées :

```
|= OU bitwise appliqué à lui même
^= NOR bitwise appliqué à lui même (inverse l'état de tous les bits)
&= ET bitwise appliqué à lui même
```

Pour revenir à notre exemple : 42 est-il divisible par 4 ? on peut faire la chose suivante en langage C sous Arduino :

```
  char i = 42;
  char bits0et1 = i & 0x00000011; //on selectionne les deux bits finaux
```

Puis, on sépare le bit 0 du bit 1, et on fait une comparaison logique sur le dernier bit des deux chars ainsi obtenus (un OU bitwise ou logique font autant bien l'affaire)

```
  boolean OUbits0et1 = bits0et1 | (bits0et1>>1);
  return ~OUbits0et1;
```

La propriété de divisibilité par 4 est vraie si «les deux derniers bits sont tous deux nuls», c'est à dire si «le bit 1 OU le bit 0 n'est PAS à VRAI».

E/ Fonctions classiques :

Ceci n'est qu'un résumé des **syntaxes** de ces différentes fonctions : si il subsiste certains doutes quant à leur emploi, je vous invite à visiter la page suivante

Les tests

***if** : "Si la condition est respectée alors les instructions seront exécutées **une** fois"*
if (condition) { instructions }

if (variante avec else **(sinon)) :** Si le test est vrai alors "instructions si vrai" sinon "instructions si faux" est exécuté
if (condition) {instructions si vrai} else {instruction si faux}

NB : Les points-virgules ne sont présents qu'a la fin **d'instructions** pas des **tests** et **boucles** !

Les boucles

while (tant que): "Tant que la condition sera respectée alors la boucle d'instruction tournera"
while(condition) {instructions }

for : Forme générale
for(initialisation ; test ; incrément\décrément)

exemple : "Je veux que ma boucle d'instructions s'exécute 5 fois.
for(int i = 0 ; i<5 ; i++) { instructions}

Fonctions utiles (mais spécifiques)

constrain: Contraint une variable à rester dans la norme (très utile en commande analogique PWM)
y = constrain(x, m, M);

*y = x si m < **x** < M*
*y = m si **x** < m*
*y = M si M < **x***

map : Permet d'ajuster une variable en fonction de paramètres constants
A = map(x, in_min, in_max, out_min, out_max);
est strictement équivalent à :
A = (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;

goto: 

à utiliser avec modération 

Cette fonction permet **de sortir** d'une boucle dès qu'elle est exécutée et de "sauter" de boucles en boucles, ce qui peut se révéler **dangereux** pour l'exactitude du programme (qui peut ne pas s'arrêter) et occasionner un (gros) bug.
Si on devait résumer goto en quelques mots, c'est la **téléportation** intrinsèque au programme !
[instructions1;]
:ptdr

[instructions2;]
goto ptdr;
[instructions3;]
à chaque fois que *"goto ptdr;"* est relue le programme revient à l'étape *":ptdr"* ; les instructions2 sont donc exécutés perpétuellement et les instructions3 jamais 

delay : Cette fonction permet de retarder de N millisecondes le programme.
delay(N); delayMicroseconds(n); variante où n s'exprime en microsecondes

Fonctions mathématiques

Eh, oui ! Arduino les connait... en ce qui concerne la syntaxe, je vous conseille la chose suivante :
float x ; x est un réel
float y ; idem pour y
float z ; y = f(x);
Où f(x) est remplacé par :

```
min(x,z)  < retourne le maximum de (x,z)
max(x,z)  < idem pour le minimum
abs(x)    < valeur absolue
pow(x,z)  < retourne x à la puissance z (réel)
sqrt(x)   < racine carrée
sin(x)
cos(x)
tan(x)
```

Conversion de données

```
char()  < convertit un nombre en mot
int()   < fonction "partie entière"
long()  < convertit un nombre en type "long"
float() < convertit un nombre en réel
```

F/ Les quatre fonctions à action externe spécifiques à Arduino :

digitalRead :

Permet de contrôler si la tension à laquelle est soumise un pin est haute (5V) ou basse (0V) retourne la valeur sous une variable booléenne ayant pour valeurs *{HIGH;LOW}*
I=digitalRead(pin) on affecte à I HIGH ou LOW

digitalWrite :

Sans doute la fonction la plus simple et utile des quatre, elle a permis de réaliser le *programme Blink* ci-dessus. Quand vous l'appliquez, deux configurations sont possibles : état haut (5V en sortie) ou état bas (0V en sortie) , les **syntaxes** respectives sont :
mise à l'état bas : *digitalWrite("Pin",LOW);*
mise à l'état haut: *digitalWrite("Pin",HIGH);*

analogRead : Réservée aux pins analogiques

Cette fonction permet de lire sur 1024 points de mesure une tension de 0 à 5V . En français pur et simple cela signifie que si l'on dédie le pin A0 à cette mesure, que l'on le nomme *voltmetre* (*int voltmetre = A0 ;*) alors lorsqu'on exécutera
MesureBrute=analogRead(voltmetre)
MesureBrute prendra alors la valeur d'un entier compris entre 0 et 1024 proportionnel à la valeur de la tension (1024 pour 5V, 512=1024/2 pour 2.5V etc...). Il est donc nécessaire de convertir cette mesure brute en mesure réelle, en millivolts :

On peut donc écrire ***MesureReelle=analogRead(voltmetre)*5000.0/1024.0 ;***

A ceux qui ont remarqué les .0 après les précédents chiffres : ce n'est pas nécessaire de les mettre mais conseillé, en effet les mesures réelles sont rarement entières après de telles divisions, alors pour ne pas perdre de l'information, il est conseillé de les enregistrer dans des variables de type **float** (nombres décimaux)

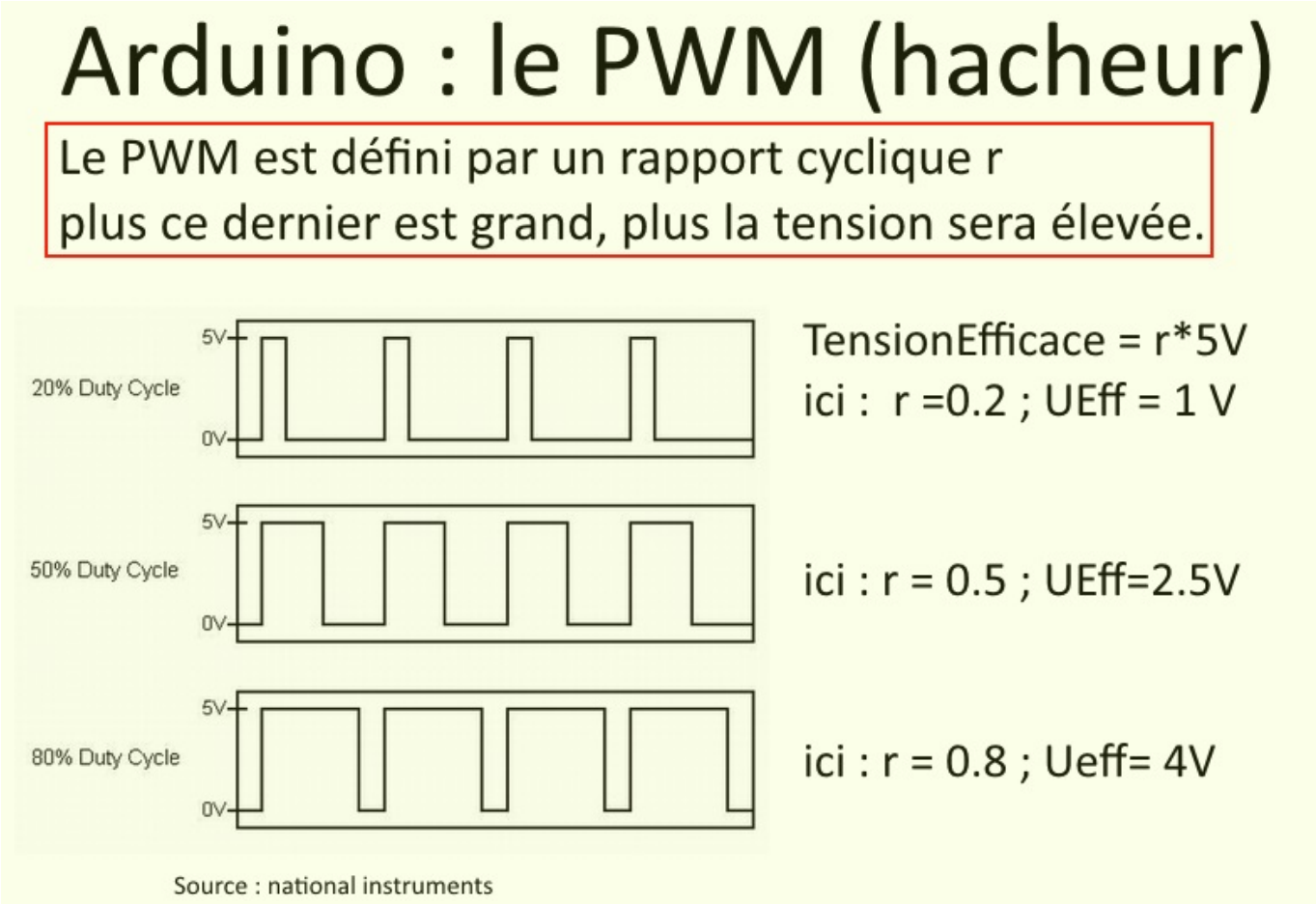
AVERTISSEMENT :
Ne **JAMAIS** dépasser les 5 volts en entrée d'une Arduino, cela pourrait lui être fatal. Mais, on peut **ruser** en utilisant des **ponts diviseurs de tension** en vue d'adapter cette dernière à notre chère carte . Avis aux "stressés" de la manip' = on peut caler en série une résistance de 10Kohms en entrée de la pin, cela n'altèrera pas la mesure (de l'ordre du microampère) mais préservera la carte d'une éventuelle maladresse...

Réalisation : Testeur de batterie !

analogWrite (PWM ou hacheur en VF) :

Les Arduinos ne délivrent **PAS** directement du courant continu analogique : ce serait trop facile pour vous, hein ?
Mais pas de panique ! I existe des moyens de retranscrire en analogique ce que dit notre Arduino, en

effet elle est dotée de **pins marqués PWM** c'est à dire des hacheurs.
Le **PWM** est un signal rectangulaire (fréquence de l'ordre de 500HZ) qui est pondéré par un **rapport cyclique r** . En fait pour une période de **T** secondes, on aura **r*T** secondes de signal haut à 5V et **(1-r)*T** secondes de signal bas à 0V.
donc **r = DuréeSignalHaut/T**
Bon, assez parlé voici une image explicative



En fait, le rapport cyclique r est codé dans une Arduino entre 0 et 255, c'est à dire qu'il existe 255 niveaux d'ajustement.

Exemple : je veux obtenir une tension en moyenne égale à 2V sur le pin *alim2v* . Je calcule r : **r = 2/5 = 0.4** *puis je calcule le codage correspondant :* ***CODAGE = r*255 = 0.4*255 = 102*** *(si le nombre n'était pas entier il aurait fallu arrondir)*
puis j'applique la commande : analogWrite(alim2v, 102);

On pourra utiliser la formule générale : **analogWrite("Pin", PartieEntière[(TensionVoulue/5)*255]);**

Remarque : Pour des valeurs de *TensionVoulue* qui **varient** il est judicieux d'utiliser une variable entière **int** .
Par exemple : *int k = TensionVoulue(t)*255/5 ;*

Une **note spécifique** relative à la **conversion numérique-analogique** sera bientôt mise en ligne

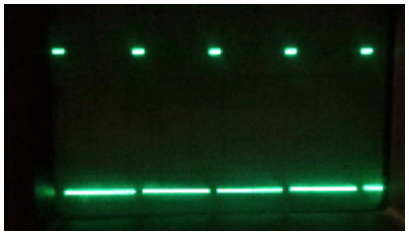
Attention : afin que votre programme reste **dans les cordes**, je vous conseille d'utiliser la fonction **constrain** afin de limiter la valeur de commande PWM entre 0 et 255. En effet, votre Arduino ne sait pas ce que signifie *analogWrite(pin, 267);* et va sortir un beau 0V à la place de 5V !

Petit aperçu de ce que ça donne à l'oscillo :

Voici une boucle de commande PWM visualisée sur l'oscillo et codée comme suit :

void loop () {

```
 for( int i; i<256; i++) { \\
  AnalogWrite( Sortie, i); \\
  delay(10); */ -> valeur arbitraire mais il faut un petit delay pour y voir
quelque chose /* \\
  } \\
\\
```



Réalisation : un variateur pour ampoule à incandescence ou moteur CC le principe est le même...

G/ Définir une fonction

Lorsqu'on ne veut pas recopier le même code plusieurs fois dans le programme, on peut utiliser une **fonction**. Elle doit se placer en dehors de ***void*** *loop() { }* et ***void*** *setup{ }* (qui sont d'ailleurs elle-mêmes des fonctions.)

Une fonction doit comporter.

- des arguments (ou variables)
- une image (optionnel)
- un nom
- un type
- une structure de définition

Types de fonctions

- void → Sans doute le plus courant, cette fonction **n'a pas d'image** (ie: elle ne renvoie aucun nombre ou booléen). Cela ne signifie pas qu'elle ne fait rien, elle peut agir sur les entrées/sorties (digital/analog|Read/Write) mais ces fonctions ne font pas sortir d'informations directement.
- int → Permet de renvoyer **un entier** [CFR plus haut]
- float → Permet de renvoyer **un réel**
- char → Permet de renvoyer une **liste de caractères**
- boolean → Permet de renvoyer une **variable booléene**
- etc…

Syntaxe

typeDeFonction nomDeVotreFonction(typeDeVariable nomDeVariable, …, …) { instructions; }

NB : On peut avoir autant de variable que l'on veut mais il faut préciser le type au préalable !

Exemples

Clignotant

void blink() {

digitalWrite(led, HIGH);
delay(1000);
digitalWrite(led, LOW);
delay(1000);
}
Cette fonction est sensé faire clignoter une diode à 0.5Hz !

Correction proportionnelle

On se donne un système asservi par un capteur donnant une tension Ucapteur on veut une tension Uconsigne. Une correction **proportionnelle** est en fait la multiplication de l'erreur fois une constante d'intégration K. On veut que cette correction nous ressorte un **entier** compris entre 0 et 255 (commande applicable par PWM|analogWrite) et que tous les paramètres soient réglables.

float proportionnele(float Ucapteur, float Uconsigne, float K) *{*

float Erreur= (Uconsigne-Ucapteur)/Uconsigne;
*float CorrectionProportionnelle=Erreur*K;*
CorrectionProportionnelle=constrain(CorrectionProportionnelle, 0, 255);
return CorrectionProportionnelle;

}

H - Le port serie

par Pierre Salles, que je remercie chaleureusement !

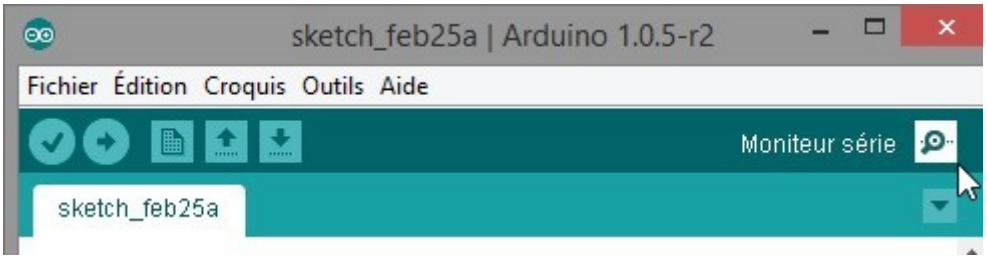
Présentation

“Ce qui se passe dans l'Arduino reste dans l'Arduino”
Ou pas.

Votre PC ne sert pas juste à envoyer une bout de code sur votre carte préférée.

Ces deux lascars peuvent entretenir des relations plus que fusionnelles. 😎

La liaison USB sert d'alimentation pour la cartes et les petits projets, mais avec quelques lignes de codes, vous pouvez accéder à toutes vos variables en temps réels. Vous pouvez aussi envoyer des instructions à l'Arduino, en utilisant le clavier, la souris. (mais on verra ça un peu plus tard).



Le code :

```
void setup() {
Serial.begin(9600);
Serial.print(“Salut, ”);
Serial.println(“ça va ?”);
Serial.print(“il fait beau aujourd'hui”);
}
void loop() { }
```

rajouter image du serial avec ce prog.

Il faut d'abord initialiser la liaison dans le void setup, avec le Serial.begin(). La valeur entre paranthèse est une valeur en bauds.
Cela correspond à un nombre de bits transmis par seconde. Les valeurs de référence sont : 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, et 115200

Ensuite, le Serial.print() permet d'afficher un texte ou une grandeur sur le moniteur.
Le Serial.println() permet d'afficher le prochain texte à la ligne suivante.

A quoi ça sert?

Le but n'est bien sur pas d'afficher seulement bonjour.
On peut l'utiliser pour vérifier son programme et essayer de repérer les erreurs.

Exemple :

Mon programme ne marche pas!! Au secours! Que se passe t'il??? 
Aide Dora à trouver la solution...

```
int x = 0;
int y = 0;
void setup() {}

void loop () {
while (x<100) {
y++;
}
*partie utile du programme qui visiblement ne fonctionne pas*
}
```

Bon, pas besoin de chercher loin, mais c'est juste pour se faire une idée. Si vous ne trouvez pas, initialisez la liaison Serial comme expliqué précédemment (Serial.begin), et placez correctement les instructions pour afficher les 2 variables du programme dans le void loop.

Correction :

```
int x = 0;
int y = 0;
void setup() {
Serial.begin(9600);
}
void loop () {
while (x<100) {
y++;
Serial.print("x= ");
Serial.print(x);
Serial.print(" y= ");
Serial.println(y);
delay(500);
}
*reste du programme*
}
```

Le delay(u) est une pause de u millisecondes. Si vous avez essayé de faire le programme seuls, sans mettre cette ligne, vous avez peut être eu un peu peur 

Vous pouvez donc expliquer à Dora que la condition de sortie de la boucle n'est jamais vérifiée, et vous pouvez lui montrer les données du moniteur série pour justifier vos propos.

Mais encore...

On peut quand même faire plus intéressant avec toutes ces valeurs. Vous pouvez les enregistrer automatiquement dans un fichier Excel par exemple.
PLX-DAQ
Téléchargez le .zip.
Lancez le .exe
Un nouveau dossier va apparaitre, avec le fichier Excel. L'utilisation d'une macro fait en général apparaitre un avertissement relatif à la sécurité. Passez outre. Une nouvelle fenetre s'ouvre.

Bon, la base est donnée ici. Cette liaison a beaucoup de potentiel, à découvrir dans la partie Processing pour les intéressés

III - Utilisation du logiciel

Cette partie est essentiellement pratique : munissez vous d'un PC, d'une carte Arduino et d'un câble USB

A/ Compilateur

Maintenant que vous savez parfaitement comment se structure un programme, il faudrait maintenant **l'exporter** sur la carte. Pour cela, un **check-up complet** de votre programme va être réalisé par le logiciel, je vous propose de coder les programmes suivants afin de cataloguer les erreurs récurrentes (et énervantes)

Ma première compilation

Tout d'abord, copiez le programme suivant dans la fenêtre du logiciel :

```
int led = 13;

void loop () {
  digitalWrite(led,HIGH)
```

```
  delay(1000);
  digitalWrite(led,LOW);
}
```

Bien, maintenant que vous avez copié le programme, vérifiez-moi ce dernier (compilation) en cliquant sur le bouton **en forme de coche**.

Plein d'écritures oranges apparaissent en bas de l'écran, c'est la panique !
Rassurez-vous, vous avez juste copié des erreurs de compilation basiques et vous allez y remédier.
Le logiciel vous affiche quelque chose comme ça:

core.a(main.cpp.o): In function `main':
C:\Program Files (x86)\Arduino\hardware\arduino\cores\arduino/main.cpp:40: undefined reference to `setup'

Le compilateur vous a donc dit qu'il n'existait pas de **setup** dans votre programme, je l'ai oublié 🤦
Bref, insérez au **bon endroit** la ligne de code suivante :

```
void setup() {
  pinMode(led,OUTPUT);
}
```

Ma deuxième compilation...

Voici la version corrigée...

```
int led = 13;

void setup() {
  pinMode(led,OUTPUT);
}
void loop () {
  digitalWrite(led,HIGH)
  delay(1000);
  digitalWrite(led,LOW);
}
```

Recompilez donc ledit programme et... paf ! Un deuxième écran d'erreur 🔧 Fix Me!
Le message dit : *sketch_nov18a.ino: In function 'void loop()':*
sketch_nov18a:8: error: expected `;' before 'delay'

Oups ! Un point-virgule manque à l'appel (l'erreur de loin la plus fréquente) et le logiciel vous dit à quel endroit c'est...

Ma troisième sera la bonne !

```
int led = 13;

void setup() {
  pinMode(led,OUTPUT);
}
void loop () {
  digitalWrite(led,HIGH);
  delay(1000);
  digitalWrite(led,LOW);
}
```

Recompilez le programme, qui cette fois sera **validé** en terme de syntaxe Arduino, mais cela ne signifie pas toujours qu'il fonctionne correctement.

B/ Téléverser un programme

Munissez vous de votre câble **USB**, et si votre ordinateur ne reconnait pas la carte, allez dans le gestionnaire des périphériques et installez le driver (se trouve dans le dossier de l'application Arduino, sous-dossier “drivers”) ou consultez les ressources logiciel du site Arduino.cc
Copiez le programme précédent, il est sensé faire clignoter la LED de votre Arduino.
Puis, appuyez sur le bouton téléverser (flèche vers la gauche) en ayant sélectionné au préalable le **modèle de la carte** .

Le téléversement implique une compilation de votre programme, on peut ainsi faire les deux étapes d'un coup !

Eh bien quel beau clignotant vous avez là !

Comment ça ? La LED de l'Arduino s'allume continuellement ? 😮

En fait, elle s'éteint puis s'allume instantanément car j'ai (encore) oublié un truc... une instruction delay(1000) en bout de programme pour temporiser **l'état bas** de la LED. Ceci pour vous montrer qu'un programme peut être correct au sens du logiciel mais pas pour l'utilisateur...

Rajoutez *delay(1000);* après le deuxième digitalWrite() puis téléversez une seconde fois et **ADMIREZ !** votre travail...

C/ Liens, support logiciel et références

Site officiel d'Arduino

http://arduino.cc/

Références du langage : très utiles pour ceux qui veulent aller plus loin

http://arduino.cc/en/Reference/HomePage

Toute la gamme Arduino...

http://arduino.cc/en/Main/Products

Téléchargements utiles

http://arduino.cc/en/Main/Software

Ma référence absolue en matière d'électronique

Avis aux amateurs de transistors
Avis aux amateurs d'AOP
Avis aux amateurs de CMOS
Avis aux amateurs de musique
Ce site risque bien de vous plaire : http://www.sonelec-musique.com

Site d'un "arduinnien" confirmé

https://battomicro.wordpress.com/

Site d'un électronicien que vous reconnaîtrez...

Je sais, se faire de la pub c'est mal  ... surtout que j'ai lancé le site il y a peu 
Néanmoins : http://nitraced.neowordpress.fr

IV – Une réalisation "sérieuse" :

Maquette d'onduleur 12VDC vers 110VAC utilisant un asservissement commandé Arduino Nano

ATTENTION : Cette section a été rédigé par un passionné d'électronique et ne parle pas beaucoup d'Arduino. Il lui a fallu près de quatre mois pour réaliser tout ce qui va

suivre, et il en est ressorti traumatisé à vie ^^
Si vous n'avez pas peur… continuez !

A/ Présentation du projet

En fait, j'avais l'intention de réaliser un onduleur 230VAC dans le cadre de mon TIPE. Or, en CPGE PSI, on ne voit que les composants linéaires (condensateurs, résistances, AOP en régime insaturés, transfos…) donc j'ai dû construire un système quasiment linéaire. Je dis quasiment, car une Arduino s'insérait dans le montage

Qu'est ce qu'un onduleur ?

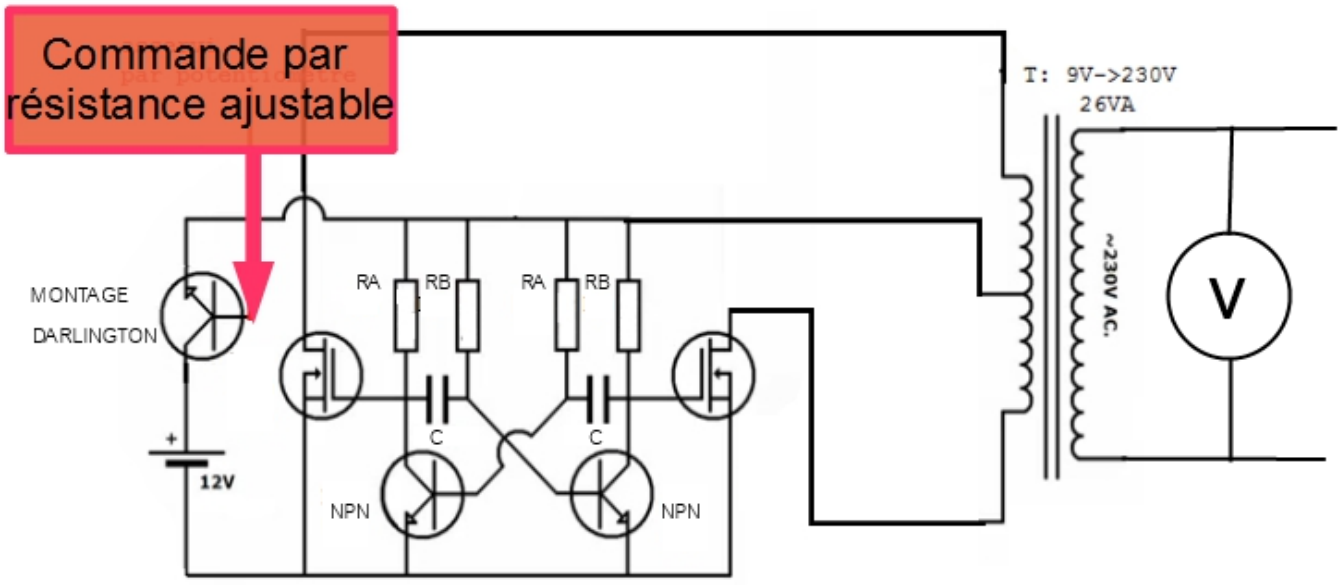
Un onduleur est un dispositif permettant de passer d'un courant de basse tension continu (ex: batterie 12VDC) à un courant alternatif de haute tension (230VAC).

Comment construire un onduleur ?

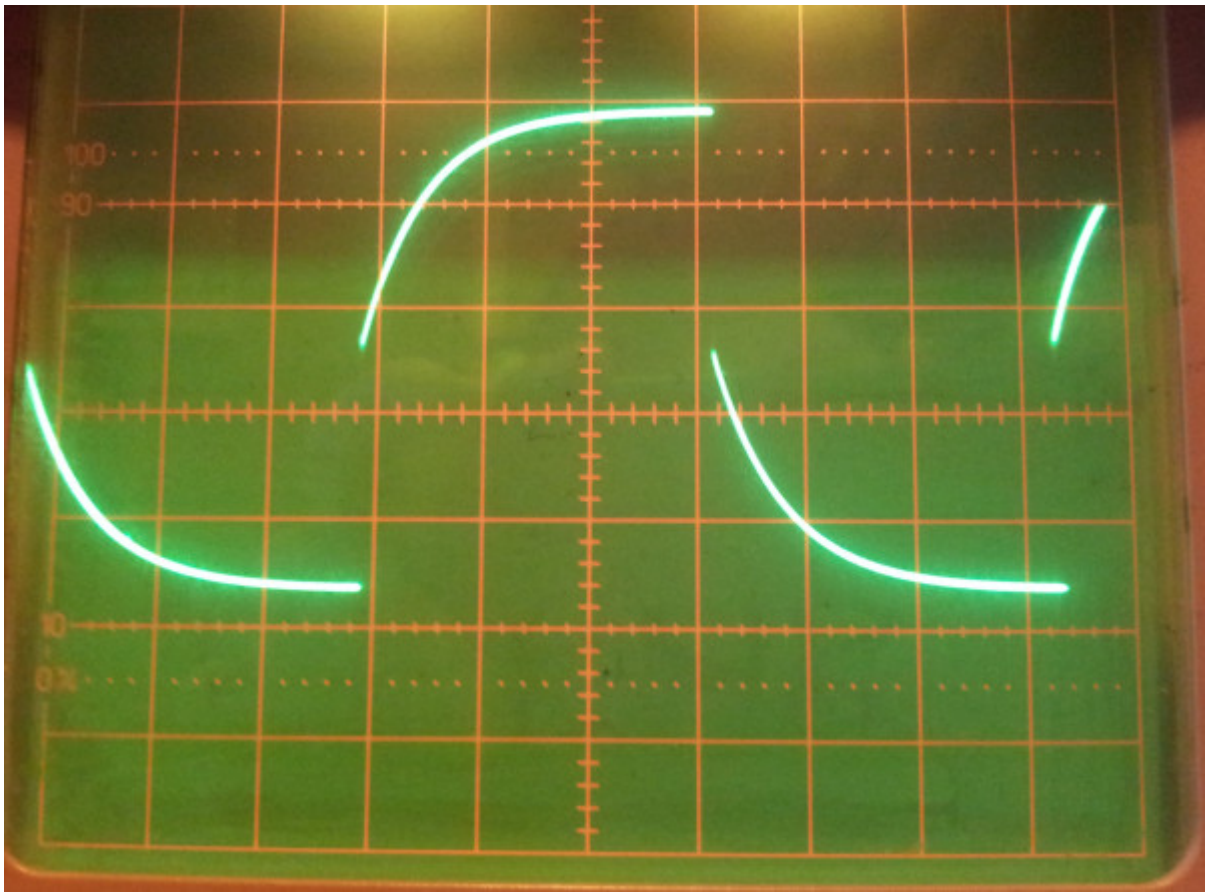
Le schéma générique est assez simple, il faut **générer une onde sinusoïdale de fréquence 50Hz** (réseau E.D.F) à partir du 12VDC, que cette dernière soit **insaturée en intensité** (en clair qu'elle ait assez de patate !) et la faire rentrer dans un **transformateur bien choisi** afin qu'il élève la tension suffisamment **mais pas trop**.
Deux problématiques se dressèrent donc face à moi…

La qualité du signal

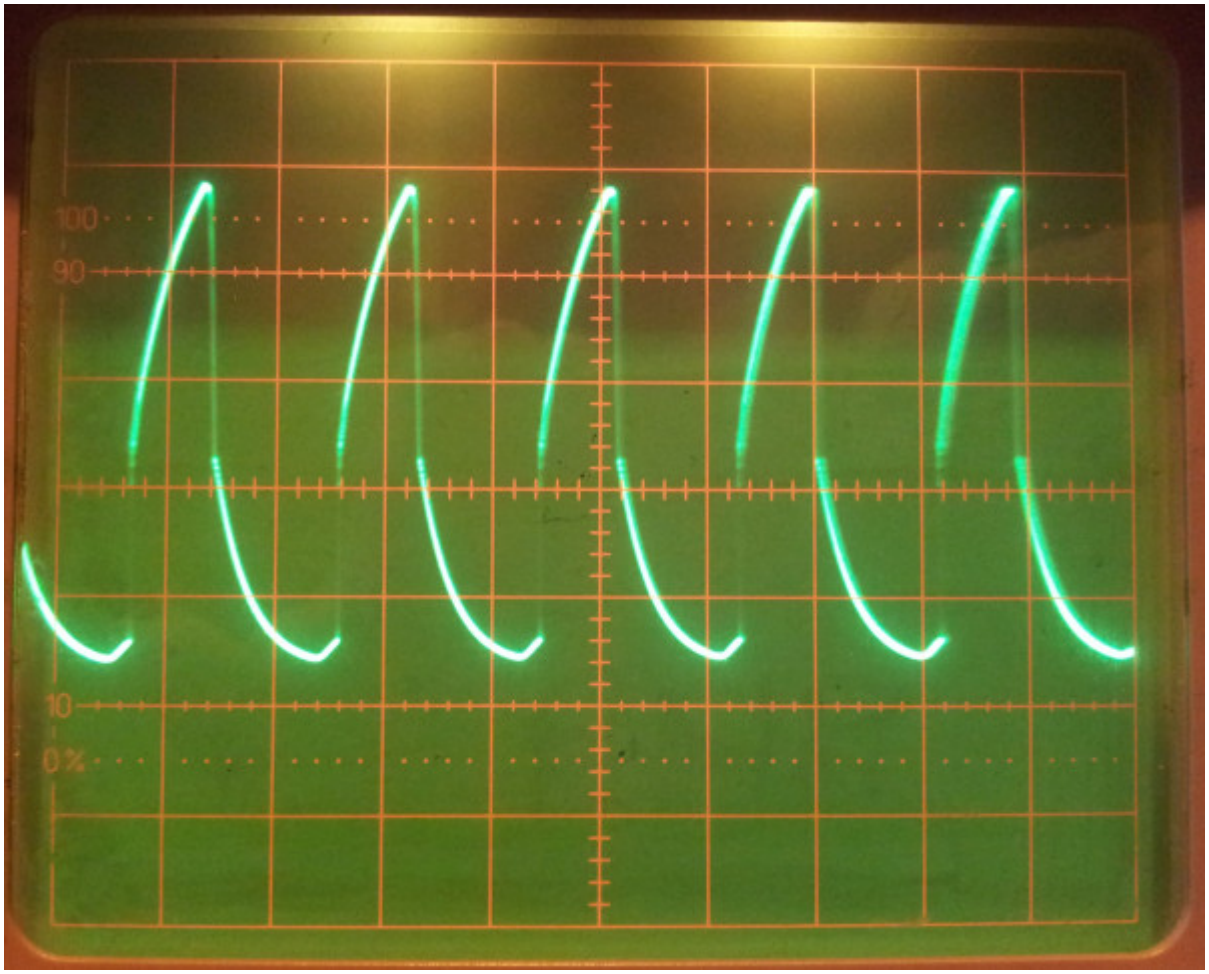
On peut obtenir facilement des tensions alternatives avec des montages simples :



Ici, on remarque que le montage de l'oscillateur (**multivibrateur astable**) est simple (5 transistors, 2 condensateurs, 4 résistances...)
Sans trop chipoter on peut considérer en **sortie de l'oscillateur** que le signal est presque sinusoïdal :

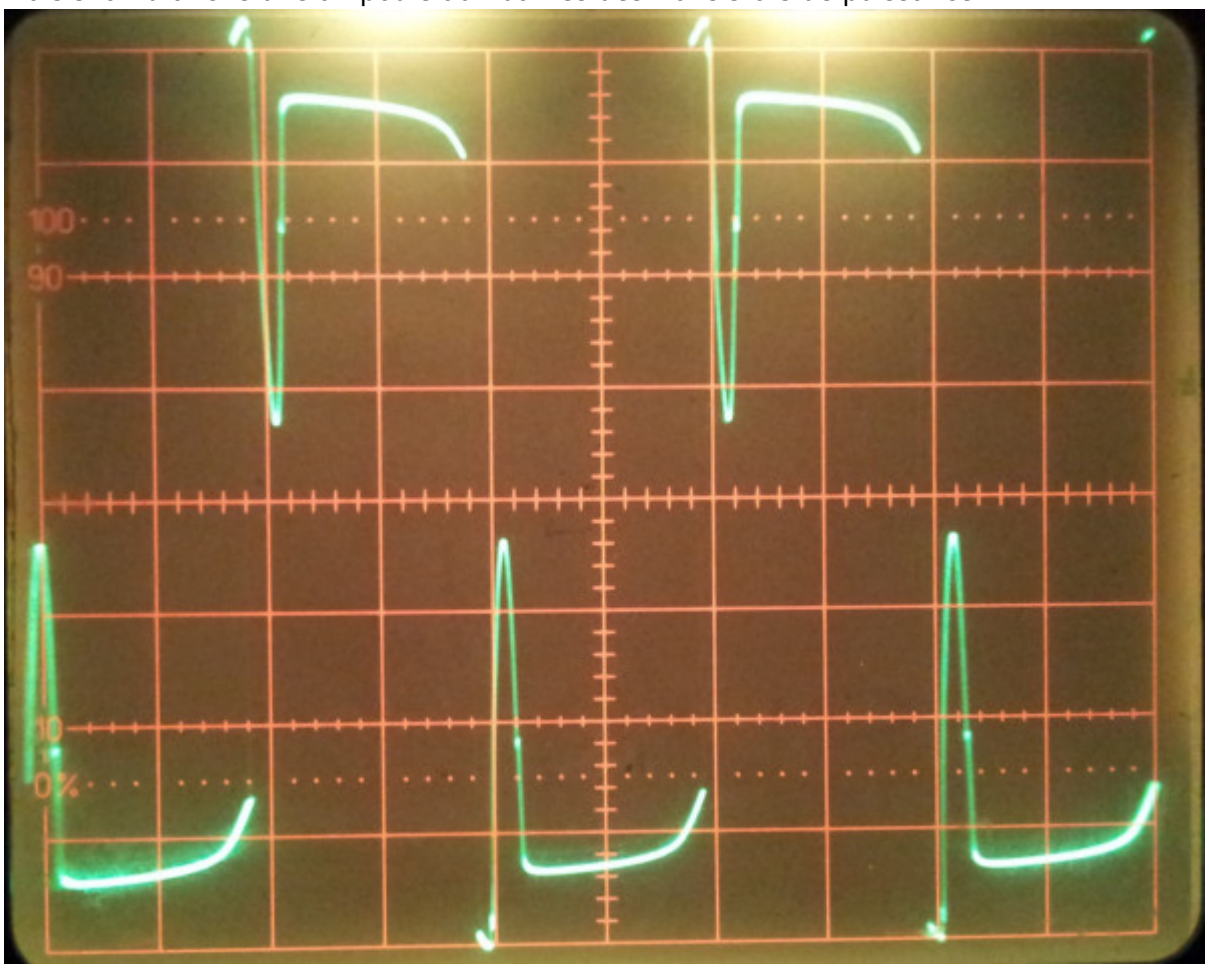


Ruse de sioux : on inverse la voie 2...



et bim !

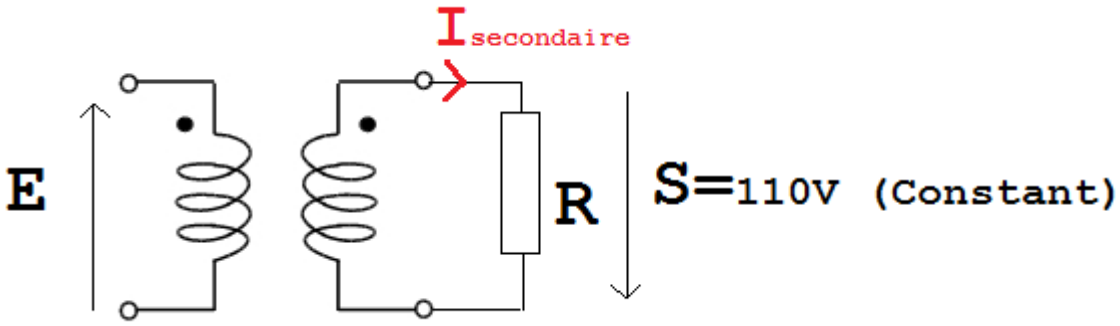
Mais si on branche une ampoule aux bornes des transistors de puissance...



patatras !

L'asservissement en tension

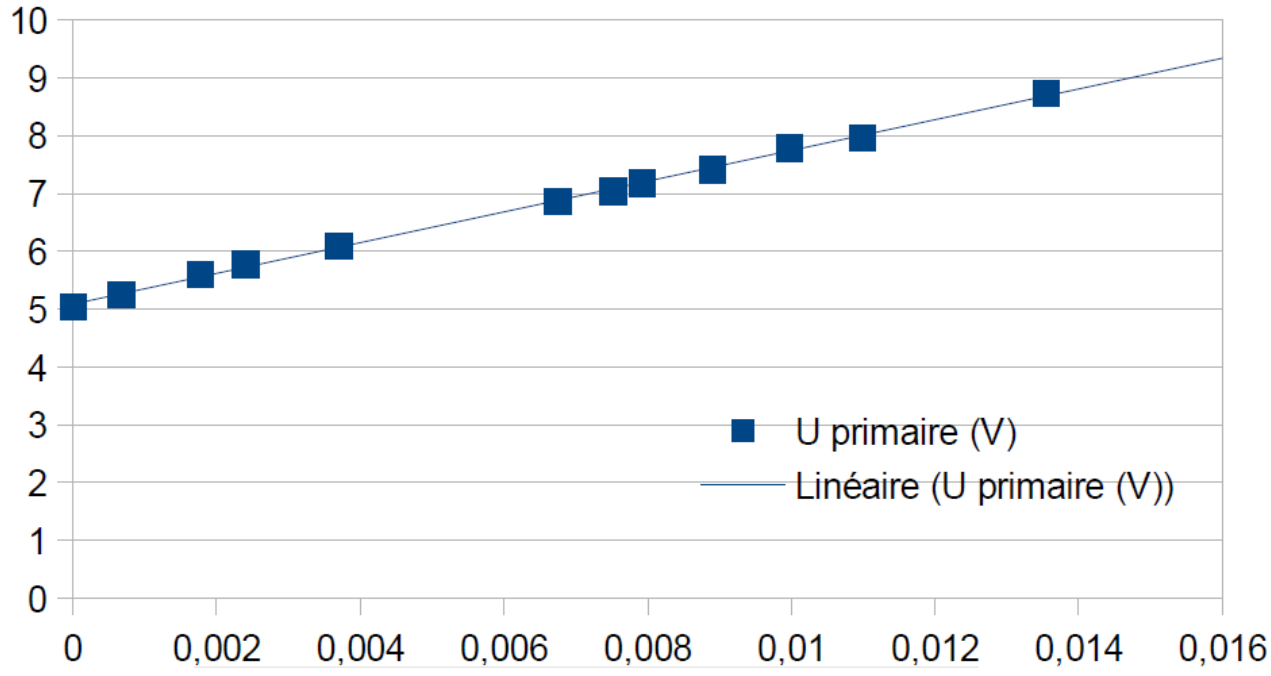
Un jour, mon professeur de SI m'a dit que le rapport d'un transformateur était constant... ce même jour j'ai mesuré 500V au lieu de 230V en sortie de ce dernier. Comme quoi, asservir en tension un onduleur est primordial si vous voulez que votre alimentation de smartphone soit encore valide après une première utilisation 😉
Voici quelques images illustrant le problème : on fait varier la charge en sortie (ou impédance) et on se rend compte que la tension d'entrée doit être inversement proportionnelle à l'intensité demandée (sans dépasser Imax du transfo tout de même !)
En effectuant les mesures dans les conditions suivantes :



On observe :

f(x) = 265,6655216533x + 5,0814888651
R² = 0,9987393519

U primaire = f (I secondaire)



D'où R=265 Ohms m.= 21,6

B/ Que vient faire Arduino dans ce montage ?

J'ai utilisé une Arduino nano en vue d'asservir ma tension en sortie selon la **philosophie suivante** : je vais mesurer quelle tension il y a en sortie, et si c'est trop fort l'arduino va faire en sorte de faire baisser l'entrée et si c'est trop faible le contraire.

Dans un pur souci de sécurité & de moyens, j'ai réalisé toutes mes expériences en 110V mais la problématique est quasi-similaire en 230V...

Mais trois problèmes ésotériques se dessinèrent contre l'utilisation de l'Arduino...

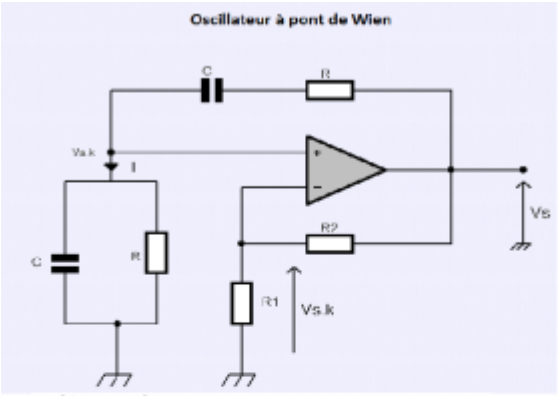
Mesurer...

La tension était en 110V alternatif alors que Umax<5V **continus** pour mon Arduino.
Il a donc fallu créer un montage redresseur et diviseur de tension sans trop perdre en **rapidité & précision**.

Générer le signal sinusoïdal

Arduino ne peut pas générer sans “shield” un signal sinusoïdal, et comme je ne voulais pas payer

10000000€ pour réaliser ce qu'aurait pu faire un montage analogique... j'ai utilisé des AOP un **oscillateur à pont de Wien** pour être plus précis :



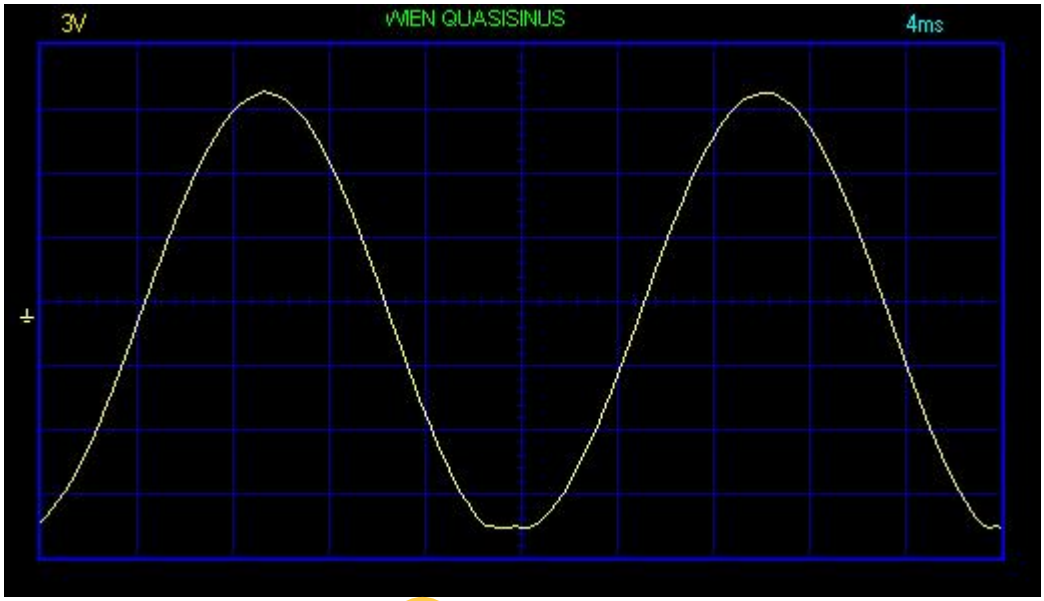
$$i = \frac{k.V_s}{R} + k.C\frac{dV_s}{dt}$$

$$\Rightarrow (1-k)\frac{dV_s}{dt} = R\frac{d}{dt}\left(\frac{k.V_s}{R} + k.C\frac{dV_s}{dt}\right) + \frac{1}{C}\left(\frac{k.V_s}{R} + k.C\frac{dV_s}{dt}\right)$$

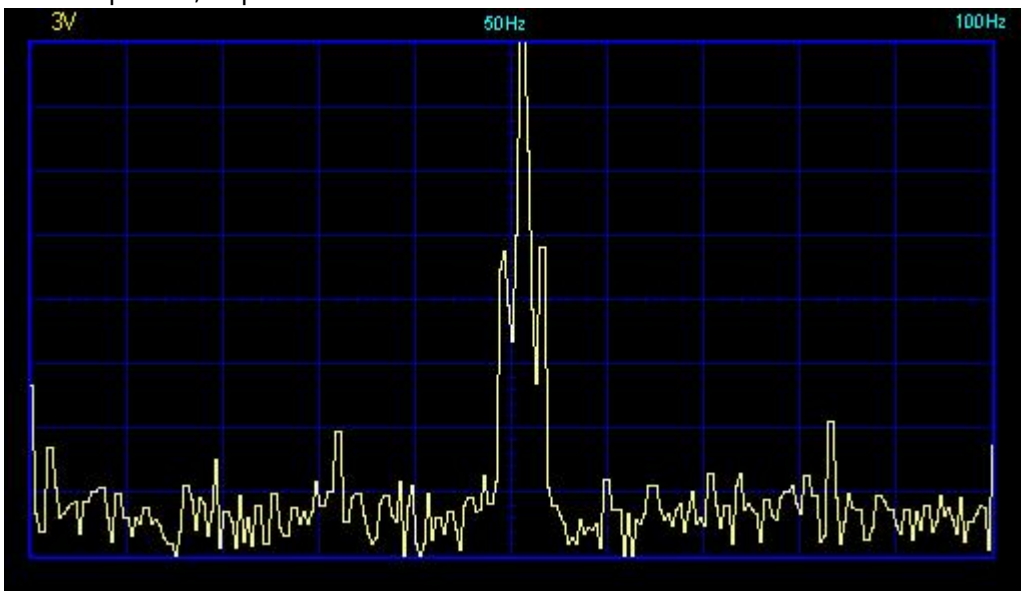
$$\Leftrightarrow (1-k)\frac{dV_s}{dt} = k.\frac{dV_s}{dt} + kRC\frac{d^2V_s}{dt^2} + \frac{kV_s}{RC} + k\frac{dV_s}{dt}$$

$$\Leftrightarrow kRC\frac{d^2V_s}{dt^2} + (3k-1)\frac{dV_s}{dt} + \frac{kV_s}{RC} = 0$$

Voici le signal en sortie de ce nouvel oscillateur :



Et son spectre, super centré

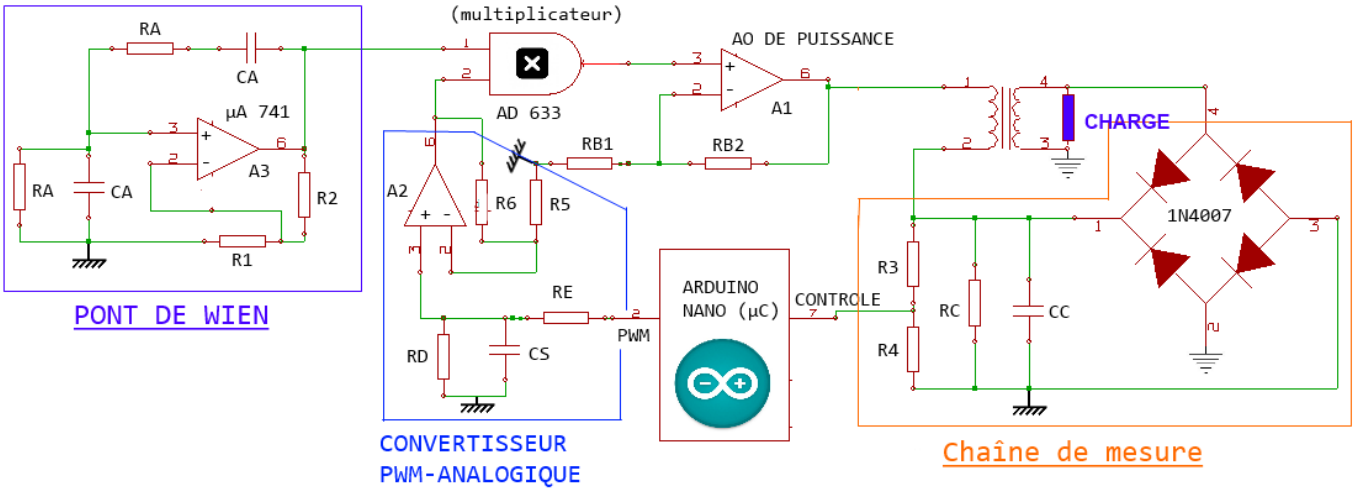


Contrôler le tout en entrée, grâce à la Nano

J'ai fait une chose assez stupide pour le contrôle de l'entrée par l'Arduino (je voulais à tout prix faire de l'asservissement linéaire, voilà pourquoi) j'ai réalisé le tout en trois étapes :

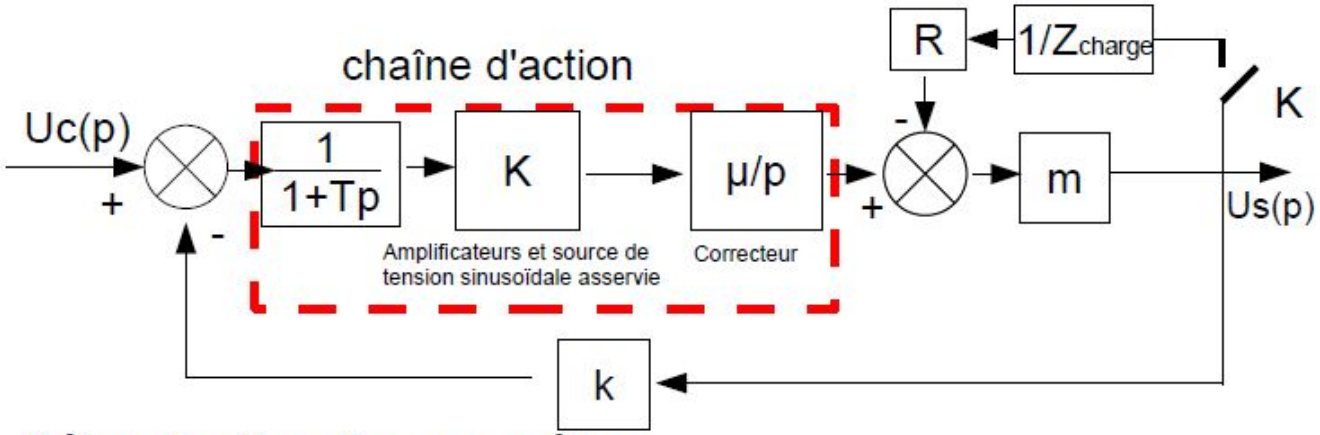
1. Lissage du PWM (signaux rectangulaires en sortie de l'Arduino)
2. Comme il ne restait plus rien, première amplification (A2)
3. Multiplication par le signal **constant en amplitude**à 50 Hz du pont de Wien
4. Amplification de puissance
5. Transformation du signal ("amplification" inductive)
6. Redressage d'une partie infime du courant, lissage et adaptation en vue de contrôler la valeur de l'amplitude en sortie.

Tout se lit dans le schéma suivant :



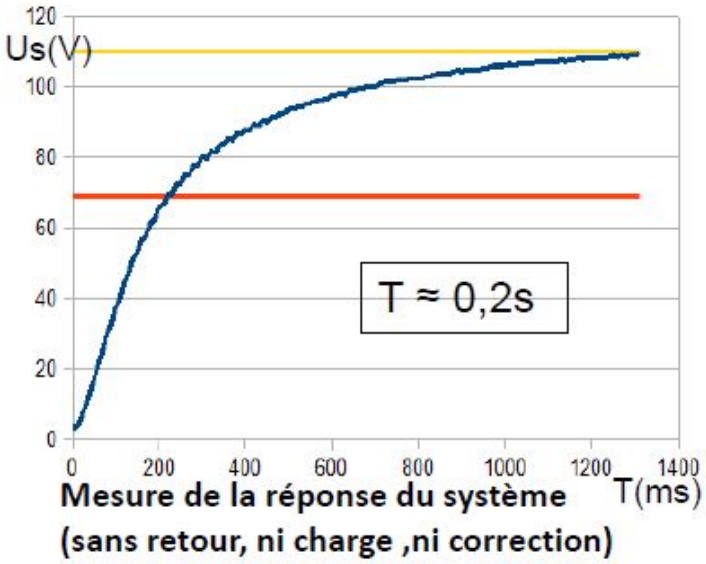
C/ Correction et codage adapté

Une étude de l'asservissement m'a révélé qu'une **correction intégrale** suffisait (je ne cherchais pas une rapidité folle (50HZ, c'est faible et mes oscillos étaient à la ramasse côté mesures/secondes^^). Le schèma-blocs était le suivant (les constantes étant obtenues en régime permanent ou temporel expérimentalement (tests de réponses en boucle ouvertes, mesures d'impédance par ampéremètre/voltmètre...) :



Détermination des paramètres

* 1ier Ordre :



Mesure de la réponse du système
(sans retour, ni charge ,ni correction)

* Facteur K :

GA1 = 11 GA2 = 3

$$K = \frac{12\ GA1\ GA2}{\sqrt{2}} \approx 187$$

* Facteur k = 0,0055

Divers diagrammes pour prouver la stabilité de mon système faits sous Scilab :

Diagramme de black

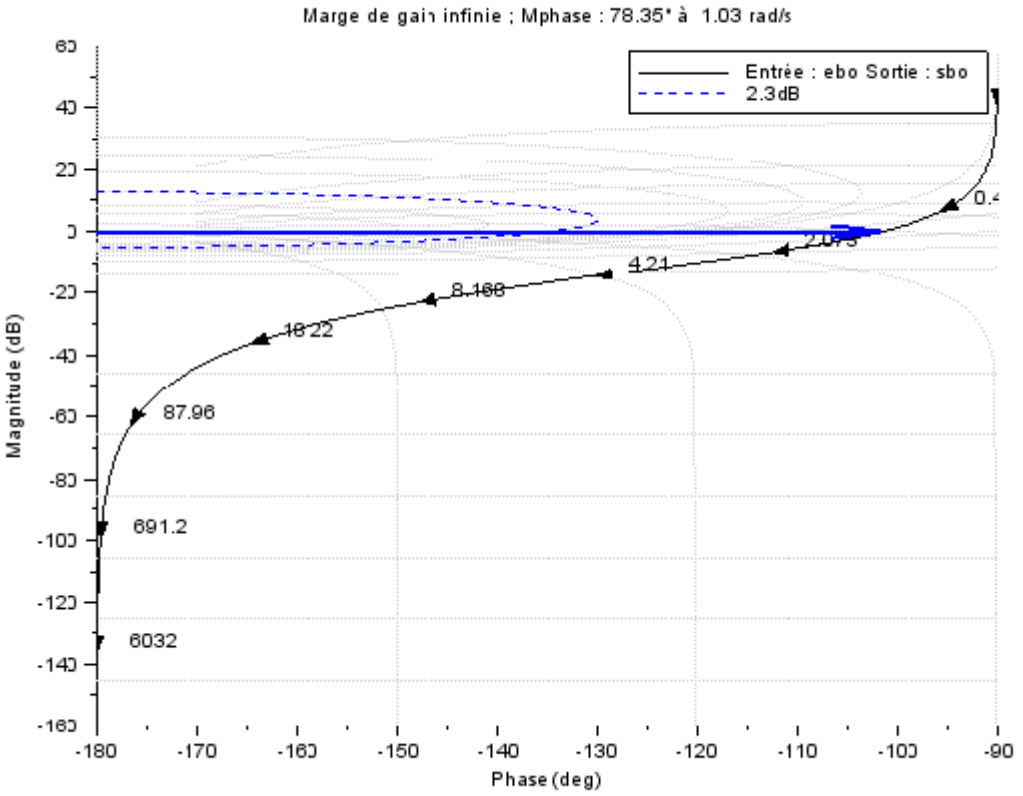
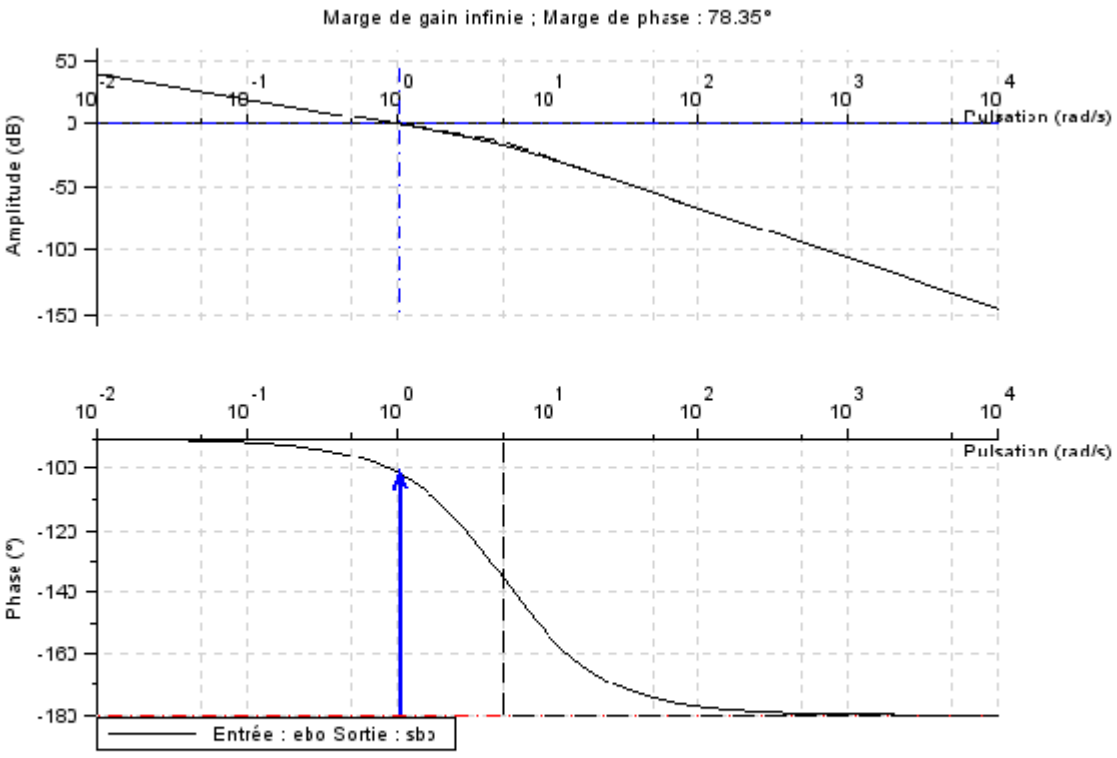


Diagramme de Bode



Codage Arduino

Ce programme ultra-court suffisait (ceci prouve l'efficacité d'Arduino 😉 :

```
int commande = 3;
int test = A0;
int led = 13;

float Uc = 600;
float K = 0.01;
int totalEcart=0;
int consigne = 90;

void setup() {
  pinMode( commande, OUTPUT);
  pinMode( led , OUTPUT);
  digitalWrite( led, HIGH);
  delay(10000);
  digitalWrite( led, LOW);

}
void loop () {
  float mesure = 0;
  float erreur = 0;
  float MesureTot = 0;
      for (int i = 0; i<10;i++) {
      float M=0;
      M=analogRead(test)*5000.0/1024.0;
      MesureTot+=M;
      delay(1);
      }
  mesure=MesureTot/10;
  erreur=Uc-mesure;
  consigne+=int(erreur*K);
  consigne= constrain(consigne,90,255);
  analogWrite(commande, consigne);
  delay(1);
}
```

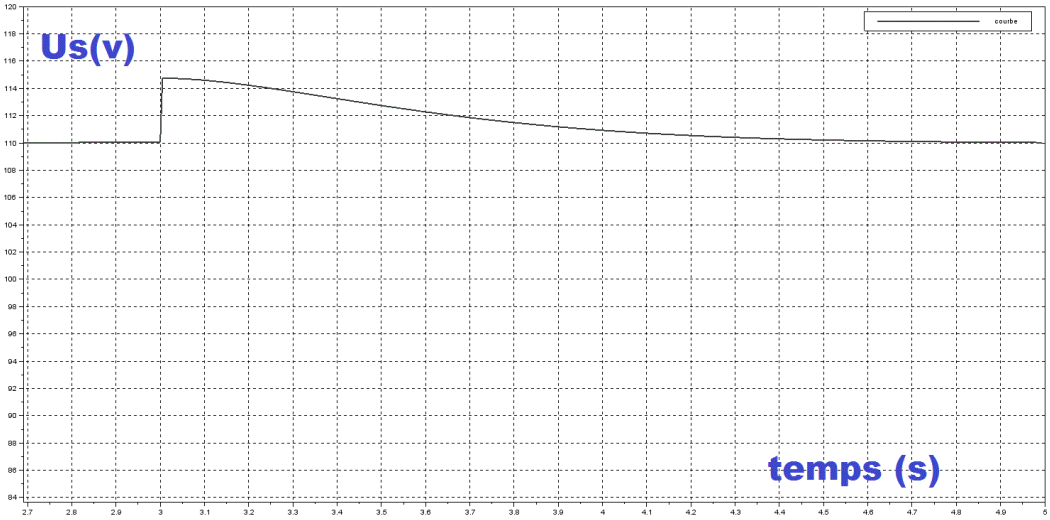
D/ Résultats (positifs !!)

Je vous laisse juger tout cela grâce à des images :

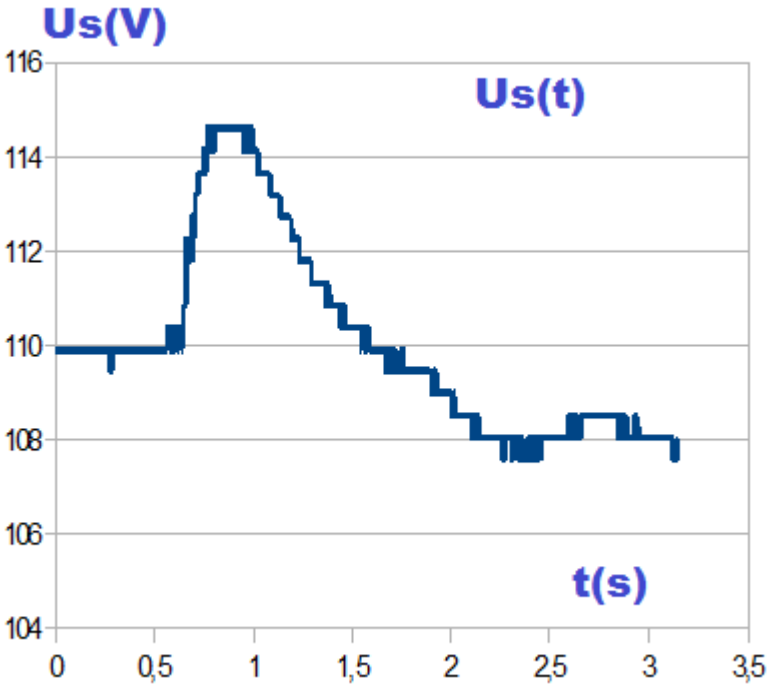
Fidélité du modèle

Scilab :

(vérifiez le temps de réponse et les asymptotes 😎)



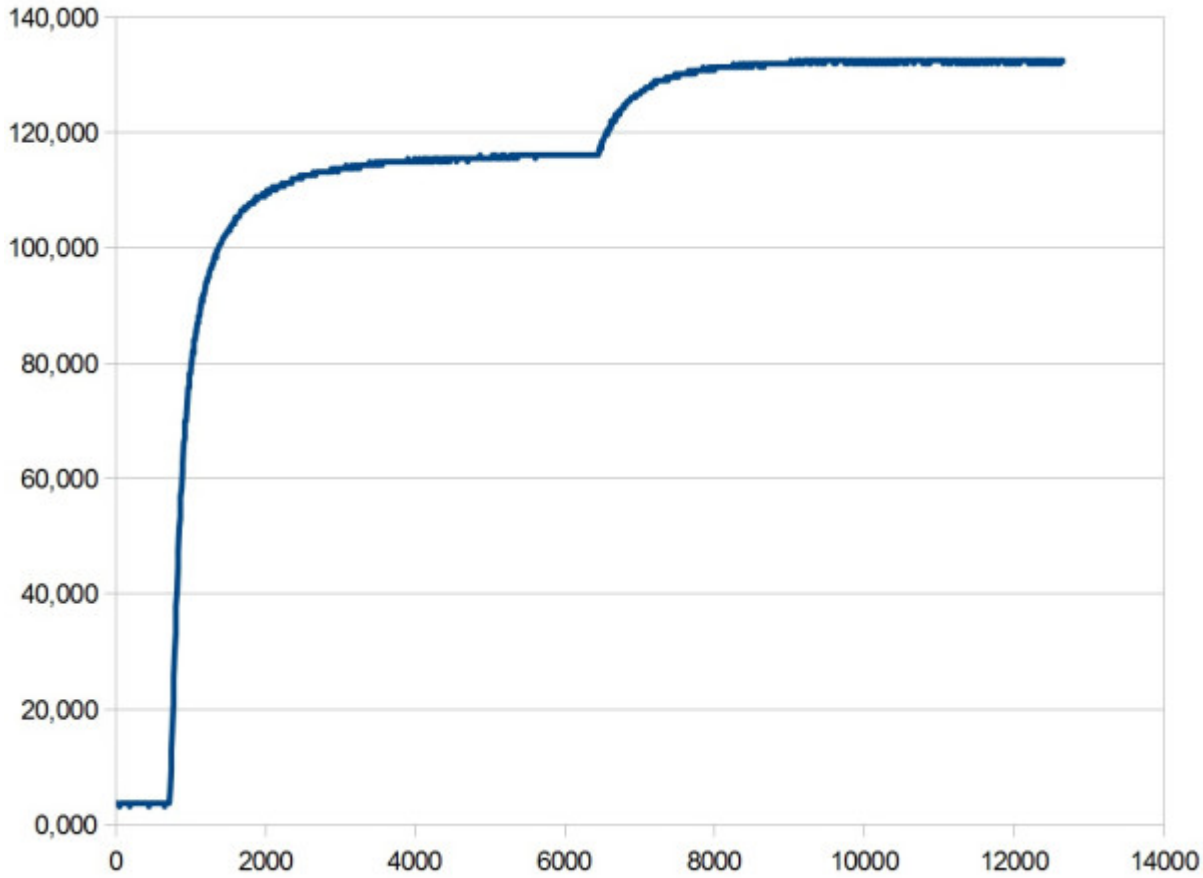
Réalité :



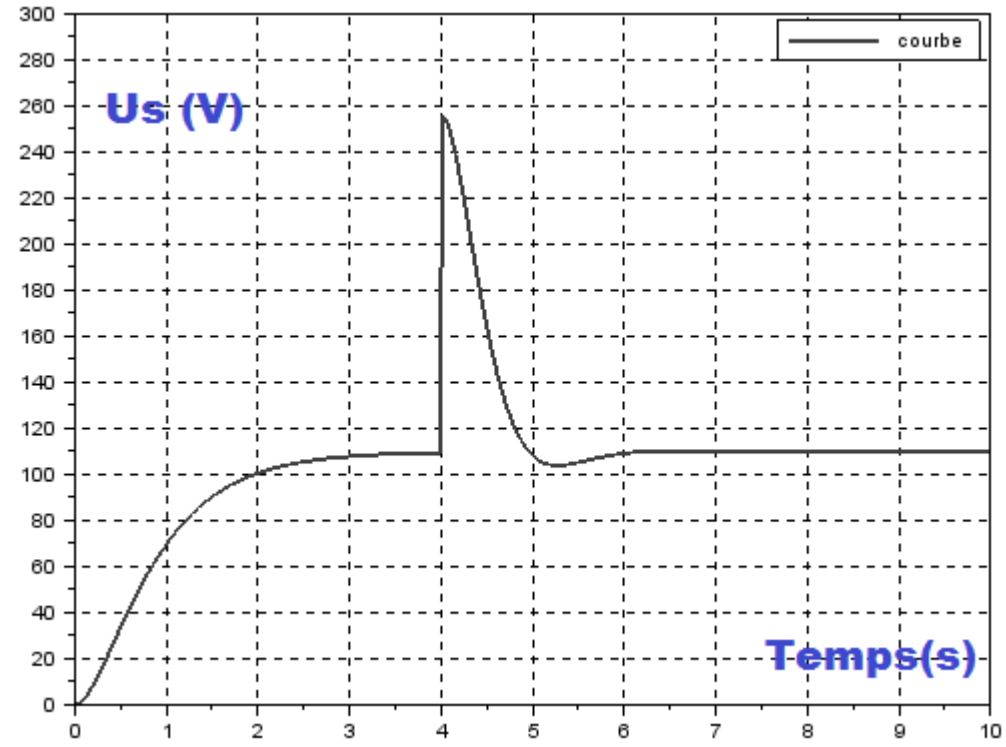
L'écart est dû au faible encodage de l'Arduino à 600mV et à mon oscillo... aussi !

Utilité de l'asservissement

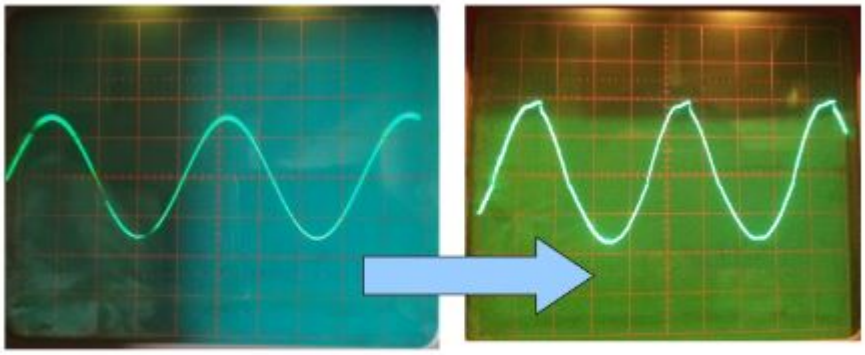
Sans lui : le smartphone grille...



Avec lui : il survit...



Qualité du signal



Signal d'entrée (Wien) Signal de sortie (Transfo)

MERCI D'AVOIR TOUT LU