Génération d'un signal par modulation de largeur d'impulsion

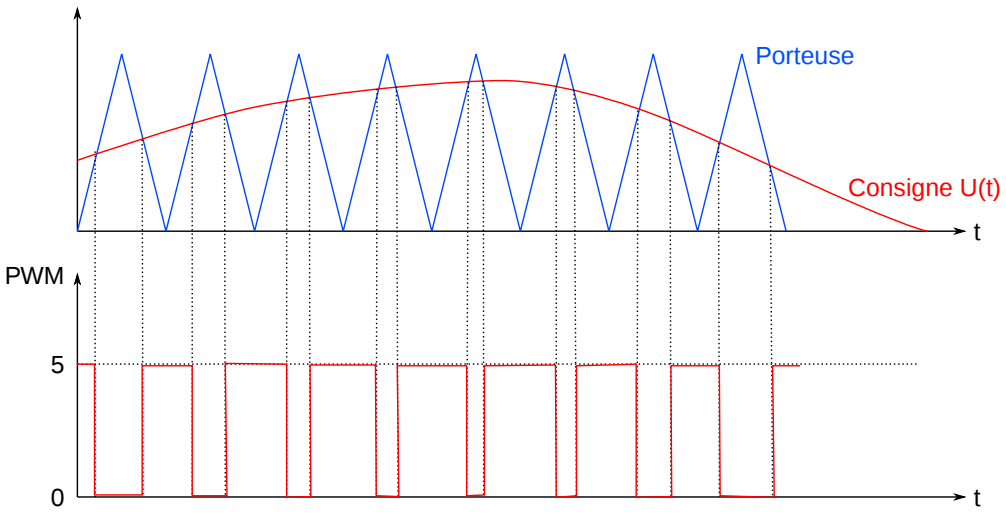
1. Introduction

La technique de modulation de largeur d'impulsion (Pulse Width Modulation PWM) consiste à générer un signal carré avec un rapport cyclique modulé en fonction d'un signal de commande. Le signal généré peut servir à commander un circuit de puissance à découpage (pont en H), associé à un filtrage passe-bas inductif, pour générer une onde sinusoïdale ou d'une autre forme. La technique est utilisée dans les onduleurs monophasés, diphasés ou triphasés [1]. Le même principe est utilisé dans les amplificateurs Audio de classe D.

Cette page montre comment générer un signal PWM avec un Arduino. Le programme présenté fonctionne sur les arduinos UNO, MEGA et YUN.

2. Modulation de largeur d'impulsion

La figure suivante montre le fonctionnement de la modulation de largeur d'impulsion (MLI). Une porteuse triangulaire est comparée à un signal de consigne, par exemple une sinusoïde. Le signal de consigne doit avoir une fréquence bien plus petite que la porteuse. Le signal de sortie est au niveau haut (disons 5 V) lorsque la consigne est supérieure à la porteuse, au niveau bas (0 V) dans le cas contraire. On considère le cas d'un signal de consigne à valeurs positives. Pour traiter un signal alternatif, il suffira de lui appliquer un décalage.



Le signal PWM obtenu doit subir un filtrage passe-bas pour en extraire le signal de consigne. Pour comprendre le principe de cette restitution, considérons le cas d'une consigne constante égale à $U(t) = U_0$. Le signal PWM est alors un signal carré dont le

rapport cyclique est $\alpha = U_0/m$, où m est la valeur maximale de la porteuse. La moyenne de ce signal carré est précisément égale à U_0.

Lorsque la consigne est lentement variable par rapport à la porteuse, il faudra appliquer un filtrage passe-bas pour restituer les variations de basses fréquences de la consigne. En pratique, le signal PWM est utilisé pour commander un circuit de puissance travaillant en commutation, et le filtrage passe-bas est assuré par une bobine en série avec la charge. Pour commander un pont en H, il faudra aussi disposer du signal complémentaire, obtenu avec une porte NON.

3. Programmation du chronomètre

Avec un microcontrôleur, la génération d'un signal PWM se fait avec un chronomètre-compteur (ou Timer). La plus grande précision sera obtenue avec un Timer 16 bits. L'arduino UNO (ATmega 328) possède un seul Timer 16 bits (Timer 1). L'arduino Yun et l'Arduino Leonardo (ATmega 32u4) possèdent deux Timers 16 bits (Timer 1 et 3). L'arduino MEGA (ATmega 2560) possède 4 Timers 16 bits (Timers 1,2,3,4).

Le compteur numéro n est un registre 16 bits, noté TCNTn, qui est incrémenté à chaque top d'horloge. La fréquence de l'horloge du compteur est soit celle de l'horloge principale du microcontroleur (16 MHz), soit un sous multiple. Voici les différents diviseurs disponibles et les bits de configuration :

- ▷ CSn2 = 0, CSn1 = 0, CSn0 = 0 : timer inactif
- ▷ CSn2 = 0, CSn1 = 0, CSn0 = 1 : fréquence f principale
- ▷ CSn2 = 0, CSn1 = 1, CSn0 = 0 : fréquence f/8
- ▷ CSn2 = 0, CSn1 = 1, CSn0 = 1 : fréquence f/64
- ▷ CSn2 = 1, CSn1 = 0, CSn0 = 0 : fréquence f/256
- ▷ CSn2 = 1, CSn1 = 0, CSn0 = 1 : fréquence f/1024

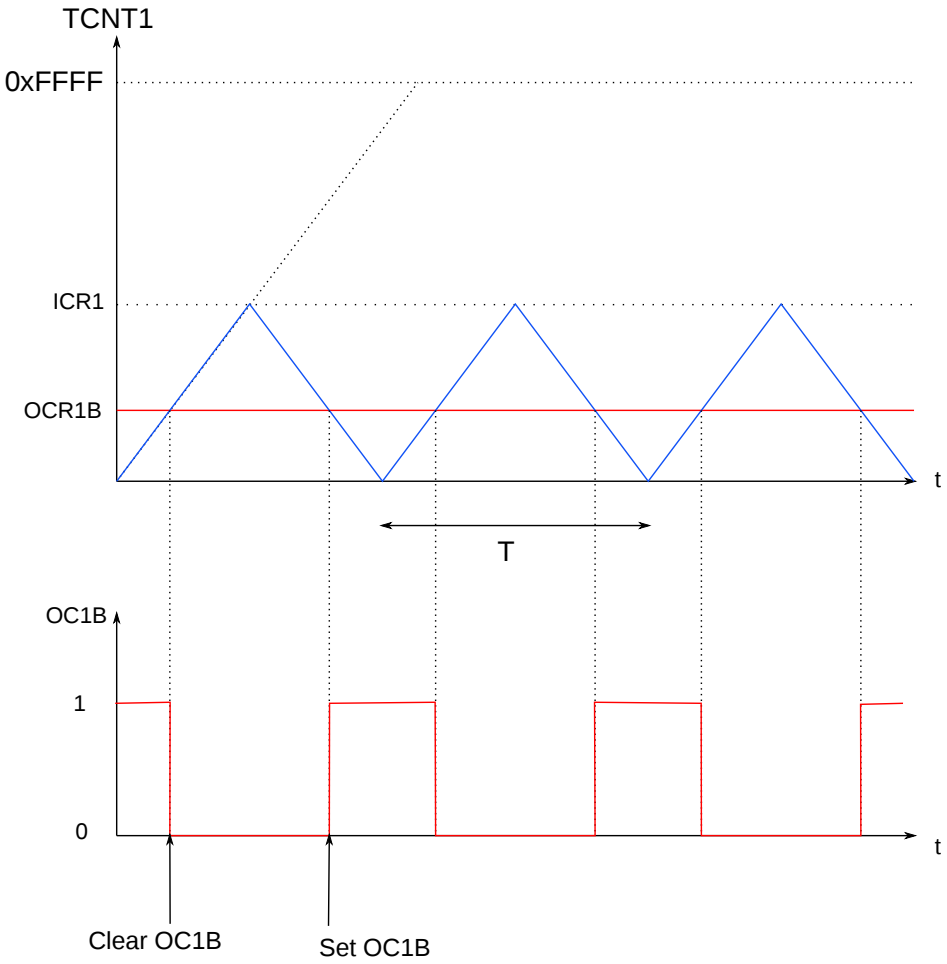
Si l'on choisit par exemple l'horloge 010, sa fréquence est $f/8 = 2\ MHz$, et le compteur parcourt un cycle complet en $32,768\ ms$.

Le Timer n se configure avec deux registres de contrôle 8 bits TCCRnA et TCCRnB (Timer Counter Control Register). Les bits CSn2, CSn1 et CSn0 sont les bits 2,1 et 0 du registre TCCRnB.

Le compteur comporte aussi trois registres 16 bits OCRnA, OCRnB et OCRnC (Output Compare Register) qui sont comparés au registre TCNT pour déclencher différentes actions. Les trois sorties associées sont COnA, COnB et COnC (Compare Output). Pour savoir à quelle borne de l'Arduino ces sorties sont reliées, il faut consulter la table de correspondance de l'Arduino Mega. Supposons que l'on souhaite générer un signal PWM sur la borne D12 de l'arduino. La table nous indique que cette borne est reliée à la sortie OC1B, c'est-à-dire la sortie B du Timer 1. Pour l'arduino UNO, la sortie OC1B est reliée à la borne D10.

Pour générer le signal PWM, on utilise le mode *PWM, Phase and Frequency Correct*, avec contrôle de la valeur maximale du compteur par le registre ICR1. Ce mode est sélectionné par les bits WGM13=1, WGM12=0, WGM13=0, WGM14=0 du registre TCCR1B. Le registre ICR1 (Input Control Register) est utilisé ici pour fixer la valeur maximale que prend le compteur (registre TCNT1). Lorsqu'il atteint cette valeur maximale, il est décrémenté jusqu'à revenir à 0. Il y a donc une phase croissante (*up-counting*)

suivi d'une phase décroissante (*down-counting*). Le registre OC1B contiendra la valeur du signal de consigne. On veut faire basculer la sortie OC1B à chaque fois que le registre TCNT1 est égal à OC1B, comme le montre la figure suivante :



Pour configurer la sortie B, il faut agir sur les bits COM1B1 et COM1B0 du registre TCCR1A. Nous choisissons la configuration COM1B1=1, COM1B0=0, qui conduit au comportement suivant : *clear OC1B on compare match when up-counting, set OC1B on compare match when down-counting.*

La période T du signal obtenu est le double de la période de l'horloge multipliée par la valeur de ICR1. Par exemple, si ICR1=800 et si l'on choisit la fréquence d'horloge 16 MHz, la fréquence du signal (la porteuse) est 10 kHz. On remarque que cette fréquence conduit à une bonne résolution pour définir les valeurs de la consigne, de l'ordre de 1/800. En augmentant la fréquence de la porteuse, on perd en résolution.

Pour mettre à jour le registre OCR1B avec la valeur du signal de consigne, on fait appel à une interruption déclenchée lorsque le compteur TCNT1 atteint la valeur maximale ICR1. Pour cela, il faut activer le bit TOIE1 (Timer Overflow Interrupt Enable) du registre TIMSK1 (Timer Interrupt Mask Register).

La mise à jour de la valeur de consigne se fait à partir d'une table contenant le signal de consigne et d'un accumulateur de phase. Supposons que la table comporte 128 éléments, définissant le signal de consigne sur une période. L'indice d'accès à cette table

est un entier 7 bits. L'accumulateur de phase est un entier 32 bits, qui représente un nombre décimal à virgule fixe comportant 7 bits pour sa partie entière et 25 bits pour sa partie fractionnaire. Si T est la période d'échantillonnage (la période de la porteuse), l'incrément à appliquer à l'accumulateur à chaque interruption pour obtenir un signal périodique de fréquence f est :

$$I = E(2^{32} f T) \tag{1}$$

L'indice d'accès à la table est obtenu en décalant la valeur de l'accumulateur de 25 bits vers la droite.

4. Programme arduino

Le programme fonctionne sur les arduinos UNO, YUN, LEONARDO et MEGA. Un signal PWM est généré avec le Timer 1 sur la sortie OC1B. Un second signal déphasé est généré sur la sortie OC1A et un troisième sur la sortie OC1C (non disponible sur le UNO). On pourra ainsi piloter un onduleur triphasé. Voici les bornes de sortie à utiliser suivant l'arduino :

- ▷ Arduino MEGA : OC1A : 11, OC1B : 12, OC1C : 13
- ▷ Arduino UNO : OC1A : 9, OC1B : 10
- ▷ Arduino YUN : OC1A : 9, OC1B : 10, OC1C : 11

generateurPWM.ino

```
#include "Arduino.h"
#define NECHANT 128
#define SHIFT_ACCUM 25

uint32_t icr;
uint32_t table_onde[NECHANT];
uint32_t accum1,accum2,accum3,increm;
uint16_t diviseur[6] = {0,1,8,64,256,1024};
```

La fonction suivante configure le Timer 1 avec une période donnée en microsecondes.

```
void init_pwm_timer1(uint32_t period) {
    char clockBits;
    TCCR1A = 0;
    TCCR1A |= (1 << COM1A1); //Clear OC1A on compare match when upcounting, set OC1A
    TCCR1A |= (1 << COM1B1);
#if defined(__AVR_ATmega2560__) || defined(__AVR_ATmega32U4__)
    TCCR1A |= (1 << COM1C1);
#endif
    TCCR1B = 1 << WGM13; // phase and frequency correct pwm mode, top = ICR1
    int d = 1;
    icr = (F_CPU/1000000*period/2);
    while ((icr>0xFFFF)&&(d<6)) { // choix du diviseur d'horloge
```

```
        d++;
        icr = (F_CPU/1000000*period/2/diviseur[d]);
    }
    clockBits = d;
    ICR1 = icr; // valeur maximale du compteur
    TIMSK1 = 1 << TOIE1; // overflow interrupt enable
    sei(); // activation des interruptions
    TCNT1 = 0; // mise à zéro du compteur
    TCCR1B |= clockBits; // déclenchement du compteur
}
```

Voici la fonction appelée lors de l'interruption. Elle incrémente les accumulateurs de phase et met à jour les registres OCR1A, OCR1B et OCR1C.

```
ISR(TIMER1_OVF_vect) { // Timer 1 Overflow interrupt
  accum1 += increm;
  accum2 += increm;
#if defined(__AVR_ATmega2560__) || defined(__AVR_ATmega32U4__)
  accum3 += increm;
  OCR1C = table_onde[accum3 >> SHIFT_ACCUM];
#endif
  OCR1A = table_onde[accum1 >> SHIFT_ACCUM];
  OCR1B = table_onde[accum2 >> SHIFT_ACCUM];

}
```

La fonction suivante remplit la table avec une forme d'onde en sinus. La valeur maximale est donnée par `icr`, qui est affectée dans la fonction `init_pwm_timer1`. L'amplitude est fournie en argument. Un décalage égal à la moitié de ICR est appliqué.

```
void set_sinus_table(float amp) {
  int i;
  float dt = 2*3.1415926/NECHANT;
  for(i=0; i<NECHANT; i++) {
    table_onde[i] = icr*0.5*(1.0+amp*sin(i*dt));
  }
}
```

Dans la fonction `setup`, les trois sorties utilisées sont configurées, la période de la porteuse et la fréquence de la sinusoïde sont choisies. Les accumulateurs sont initialisés de manière à donner les déphasages voulus. Enfin le Timer est configuré puis la table est remplie.

```
void setup() {
#if defined(__AVR_ATmega2560__)
    pinMode(11,OUTPUT);
```

```
    pinMode(12,OUTPUT);
    pinMode(13,OUTPUT);
#elif defined(__AVR_ATmega32U4__)
    pinMode(9,OUTPUT);
    pinMode(10,OUTPUT);
    pinMode(11,OUTPUT);
#else
    pinMode(9,OUTPUT);
    pinMode(10,OUTPUT);
#endif
    uint32_t period_pwm = 100; // en microsecondes
    uint32_t frequence = 100; // en Hz
    accum1 = 0;
    accum2 = ((uint32_t)(NECHANT * 0.25)) << SHIFT_ACCUM;
    accum3 = 0;
    increm = (uint32_t) (((float)(0xFFFFFFFF))*((float)(frequence)*1e-6*(float)(perio
    init_pwm_timer1(period_pwm);
    set_sinus_table(1.0);
}
```

La fonction `loop` ne fait rien. On pourra ultérieurement ajouter une communication avec un ordinateur pour modifier la fréquence du signal.

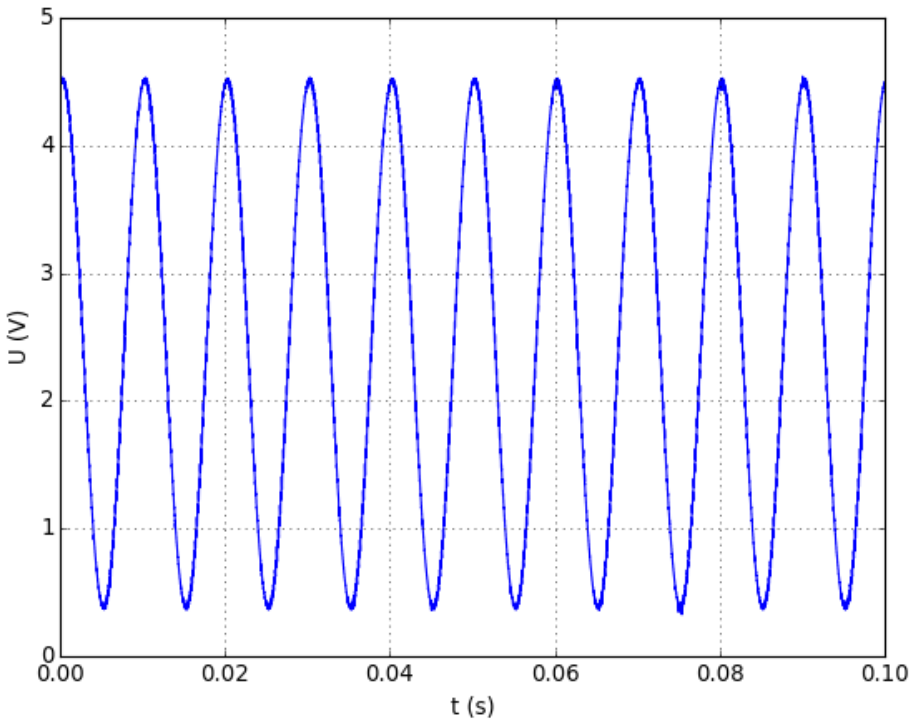
```
void loop() {

}
```

5. Test avec un filtre RC

Un filtre passe-bas RC est utilisé pour filtrer le signal PWM. Avec $R = 1\ k\Omega$ et $C = 1\ \mu F$, sa fréquence de coupure est $f_c = 160\ Hz$, ce qui convient pour des sinusoïdes jusqu'à environ $100\ Hz$. Avec une fréquence de porteuse à $10\ kHz$, le fondamental de la porteuse est à $1,2$ décade au dessus de la coupure, ce qui fait une atténuation de $-24\ dB$.

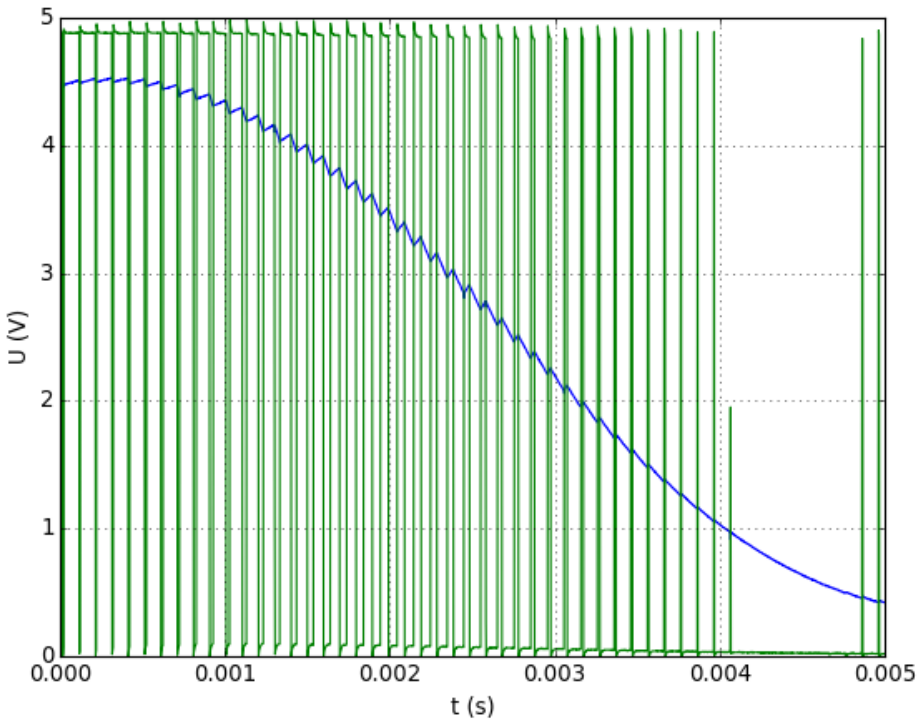
L'acquisition des signaux est faite avec la carte SysamSP5. On enregistre la sortie PWM et la sortie du filtre passe-bas. Voici un exemple avec un signal de fréquence $100\ Hz$, une porteuse à $10\ kHz$ et une amplitude de 1.

```
from matplotlib.pyplot import *
import numpy
[t,pwm,sortie] = numpy.loadtxt("sinus-pwm-1.txt")
figure()
plot(t,sortie)
xlabel("t (s)")
ylabel("U (V)")
grid()
axis([0,0.1,0,5])
```



Voici un détail du signal, avec le signal PWM :

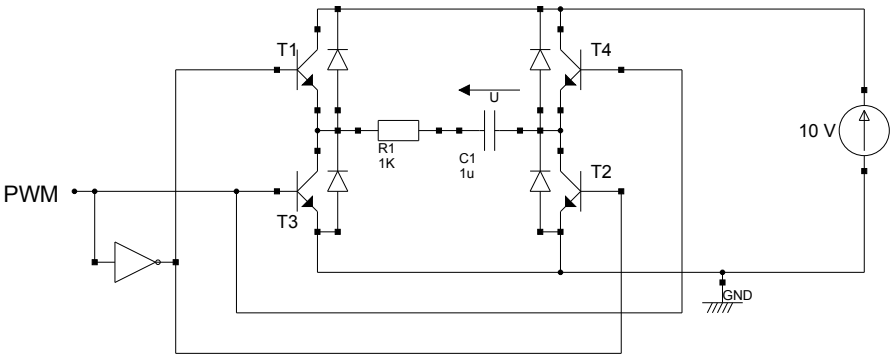
```
figure()
plot(t,sortie)
plot(t,pwm)
xlabel("t (s)")
ylabel("U (V)")
grid()
axis([0,0.005,0,5])
```



On voit bien les ondulations résiduelles sur le signal de sortie. Pour les réduire, il faut augmenter la fréquence de la porteuse. Il est possible d'augmenter la fréquence jusqu'à environ 20 kHz. Au delà, la fonction d'interruption n'a pas le temps d'être exécutée.

6. Test avec un pont en H et un filtre RC

Afin d'obtenir un signal alternatif, on utilise un pont en H commandé par le signal PWM. Voici par exemple le schéma de principe d'un pont en H à transistor bipolaire, comme le L298 de la carte MotorShield :



Le signal PWM commande les transistors T3 et T4 en commutation. Le signal complémentaire donné par la porte NON commande les transistors T1 et T2. Le courant passe donc dans la charge dans un sens ou dans l'autre selon le niveau de la commande.

Ce circuit est destiné à commander une charge inductive, une bobine ou un moteur, et peut fournir jusqu'à 2 A. Dans ce cas, le fitrage devra être réalisé soit par la charge inductive elle-même si son inductance est assez grande, soit par une bobine complémentaire ajoutée en série (filtre passe-bas LR). Dans le cas présent, la sortie est testée avec un filtre RC qui consomme très peu de courant. La tension U aux bornes du condensateur est acquise en mode différentiel par la carte SysamSP5. Une alimentation externe de 10 V est utilisée.

Le MotorShield possède deux ponts en H, notés A et B, ce qui permet de générer deux sinusoïdes en quadrature. Les bornes utilisées par le MotorShield sont les suivantes :

- ▷ BRAKE A : 9, BRAKE B : 8. Ses sorties désactivent les deux ponts. Elles doivent être mises à 0.
- ▷ PWM A : 3, PWM B :11. Ses sorties activent les ponts A et B. Elles doivent être mises à 1.
- ▷ DIR A : 12, DIR B : 13. Ses sorties sont normalement utilisées pour choisir la direction du courant dans le moteur. On applique donc les signaux PWM sur ces sorties.

Le programme précédent doit être modifié pour tourner sur l'Arduino MEGA, car la sortie OCR1A (borne 11) doit rester égale à 1. Pour cela, il suffit d'enlever la ligne de configuration de cette sortie dans la fonction `init_pwm_timer1`. D'autre part, il faut mettre les sorties ci-dessus au bon niveau dans la fonction `setup`. Voici le code modifié :

generateurPWMmotorShield.ino

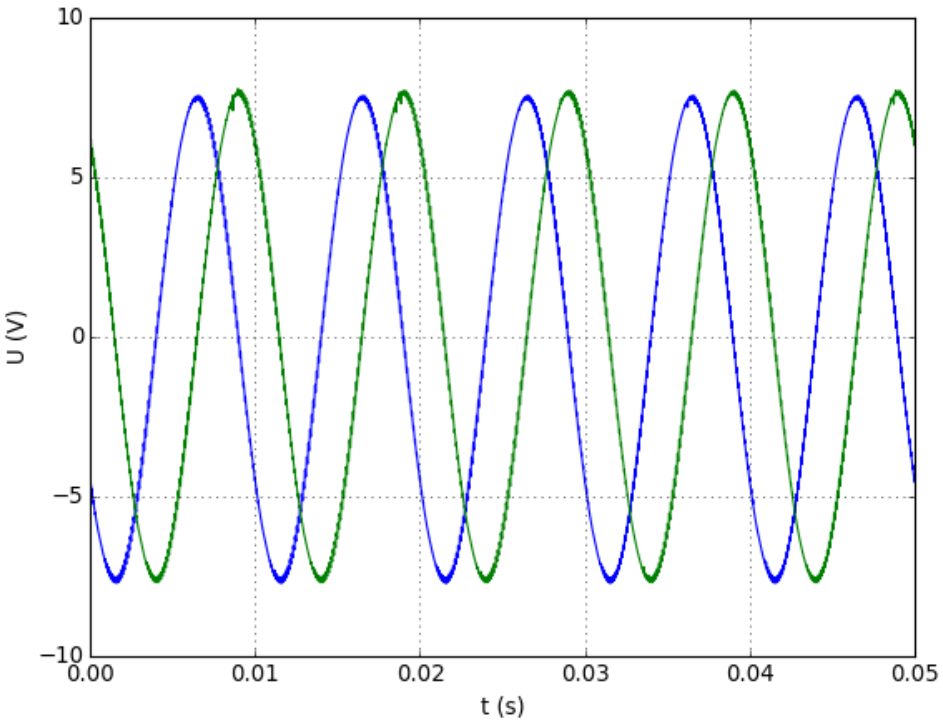
```
#include "Arduino.h"
#define NECHANT 128
#define SHIFT_ACCUM 25

uint32_t icr;
uint32_t table_onde[NECHANT];
uint32_t accum1,accum2,accum3,increm;
uint16_t diviseur[6] = {0,1,8,64,256,1024};

void init_pwm_timer1(uint32_t period) {
    char clockBits;
    TCCR1A = 0;
    TCCR1A |= (1 << COM1B1);
#if defined(__AVR_ATmega2560__) || defined(__AVR_ATmega32U4__)
    TCCR1A |= (1 << COM1C1);
#endif
    TCCR1B = 1 << WGM13; // phase and frequency correct pwm mode, top = ICR1
    int d = 1;
    icr = (F_CPU/1000000*period/2);
    while ((icr>0xFFFF)&&(d<6)) { // choix du diviseur d'horloge
        d++;
        icr = (F_CPU/1000000*period/2/diviseur[d]);
```

```
    }
    clockBits = d;
    ICR1 = icr; // valeur maximale du compteur
    TIMSK1 = 1 << TOIE1; // overflow interrupt enable
    sei(); // activation des interruptions
    TCNT1 = 0; // mise à zéro du compteur
    TCCR1B |= clockBits; // déclenchement du compteur
}

ISR(TIMER1_OVF_vect) { // Timer 1 Overflow interrupt
  accum2 += increm;
#if defined(__AVR_ATmega2560__) || defined(__AVR_ATmega32U4__)
  accum3 += increm;
  OCR1C = table_onde[accum3 >> SHIFT_ACCUM];
#endif
  OCR1B = table_onde[accum2 >> SHIFT_ACCUM];

}

void set_sinus_table(float amp) {
  int i;
  float dt = 2*3.1415926/NECHANT;
  for(i=0; i<NECHANT; i++) {
    table_onde[i] = icr*0.5*(1.0+amp*sin(i*dt));
  }
}

void setup() {
#if defined(__AVR_ATmega2560__)
    pinMode(11,OUTPUT); // A
    pinMode(12,OUTPUT); // B
    pinMode(13,OUTPUT); // C
#elif defined(__AVR_ATmega32U4__)
    pinMode(9,OUTPUT);
    pinMode(10,OUTPUT);
    pinMode(11,OUTPUT);
#else
    pinMode(9,OUTPUT);
    pinMode(10,OUTPUT);
#endif

    pinMode(3,OUTPUT);
    digitalWrite(3,HIGH); // pwm_a
    pinMode(11,OUTPUT);
    digitalWrite(11,HIGH); // pwm_b
    pinMode(9,OUTPUT);
    digitalWrite(9,LOW); // brake_a
    pinMode(8,OUTPUT);
```

```
    digitalWrite(8,LOW); // brakeèb


    uint32_t period_pwm = 100; // en microsecondes
    uint32_t frequence = 100; // en Hz
    accum1 = 0;
    accum2 = ((uint32_t)(NECHANT * 0.25)) << SHIFT_ACCUM;
    accum3 = 0;
    increm = (uint32_t) (((float)(0xFFFFFFFF))*((float)(frequence)*1e-6*(float)(perio
    init_pwm_timer1(period_pwm);
    set_sinus_table(1.0);
}

void loop() {

}
```

Voici les deux signaux obtenus par filtrage RC pour les deux ponts :

```
[t,UA,UB] = numpy.loadtxt("sinus-pwm-2.txt")
figure()
plot(t,UA)
plot(t,UB)
xlabel("t (s)")
ylabel("U (V)")
grid()
axis([0,0.05,-10,10])
```



Références

[1] L. Lasne, *Electronique de puissance*, (Dunod, 2015)