Mesures courant-tension de puissance

1. Introduction

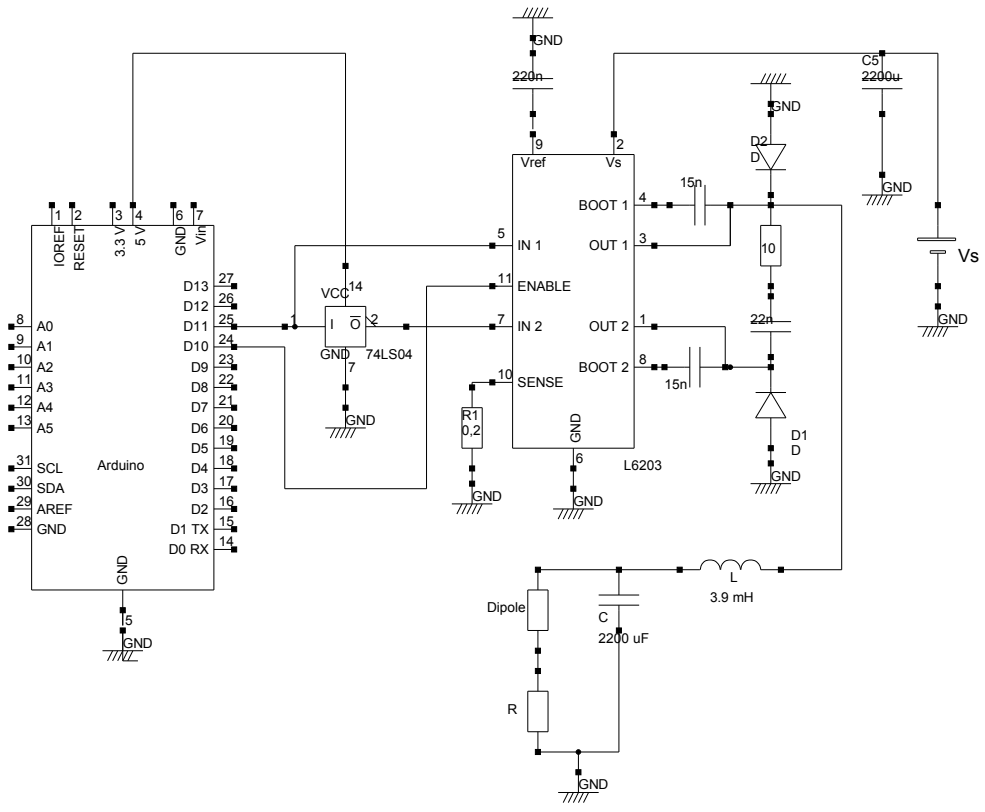
Ce document présente la réalisation d'un traceur de caractéristique courant-tension (en DC) pour un dipôle d'une puissance de quelques watts. La tension peut varier de 0 à 30 V, le courant de 0 à 3 A.

Le dispositif utilise le convertisseur DC-DC à découpage décrit dans Convertisseur DC-DC pour arduino. La technique d'utilisation du convertisseur analogique-numérique est décrite dans Conversion analogique-numérique multivoies avec échantillonnage.

2. Dispositif électronique

2.a. Convertisseur à découpage

Le convertisseur DC-DC à découpage est décrit en détail dans Convertisseur DC-DC pour arduino. On rappelle son schéma avec la connexion à l'arduino :



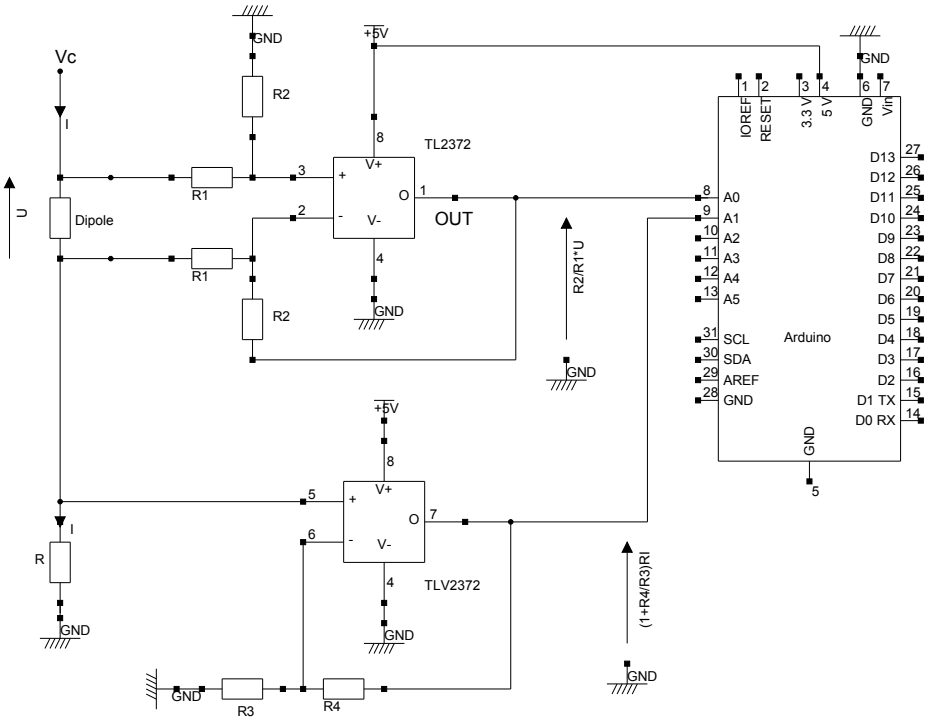
La tension constante V_s fournie par la source d'énergie est au minimum de 12 V. Le convertisseur fournit à la charge une tension V_c constante inférieure à V_s.

2.b. Circuit de mesure

La charge est constituée du dipôle à étudier et d'une résistance R servant à mesurer le courant. Si l'on veut alimenter le dipôle avec une source de tension, cette résistance

doit être faible par rapport à celle du dipôle. Si à l'inverse ou souhaite plutôt une source de courant, on doit prendre une grande résistance.

Le circuit de mesure permet d'une part de faire une mesure différentielle de la tension U aux bornes du dipôle, d'autre part d'amplifier ou de réduire la tension Ri.



On utilise un amplificateur double *rail-to-rail*, que l'on alimente par la tension 5 V de l'arduino.

L'amplificateur différentiel fournit à l'entrée A0 de l'arduino la tension :

$$V_0 = \frac{R_2}{R_1} U \tag{1}$$

Le rapport R_2/R_1 est choisi pour que la gamme de tension U souhaitée donne en sortie la gamme $[0, 5]$ V (ou un peu moins).

L'amplificateur non inverseur fournit à l'entrée A1 la tension :

$$V_1 = \left(1 + \frac{R_4}{R_3}\right) RI \tag{2}$$

Le rapport R_4/R_3 est choisi pour que la gamme de courant donne une tension dans la gamme $[0, 5]$ V en sortie.

Par exemple, les tests ci-dessous sont faits avec les valeurs $R_1 = 18$ $k\Omega$, $R_2 = 47$ $k\Omega$, $R_3 = 10$ $k\Omega$, $R_4 = 47$ $k\Omega$. La résistance en série avec le dipôle est $R = 1$ Ω (résistance de forte puissance, si possible avec radiateur).

Pour ces valeurs, la tension U peut varier de 0 à 12 V (la tension de l'alimentation) et le courant I peut varier de 0 à 850 mA. Le pont MOSFET L6203 peut fonctionner jusqu'à 4 A. Pour accéder à des courants plus forts, il faudra réduire le rapport R_4/R_3. Pour accéder à des tensions plus élevées, il faudra augmenter la tension V_s de l'alimentation et réduire le rapport R_2/R_1.

3. Programme Arduino

Ce programme est chargé de recevoir de l'ordinateur un rapport cyclique à appliquer au hacheur, de programmer la génération du signal PWM, de programmer une numérisation échantillonnée des voies A0 et A1 pendant une durée définie, et de renvoyer à l'ordinateur les valeurs moyennes des valeurs.

Les constantes de temps sont fixées dans l'entête : période de découpage, période d'échantillonnage pour les mesures, nombre d'échantillons pour le calcul de la moyenne.

mesuresCourantTensionPuissance.ino

```
#include "Arduino.h"

#define LECTURE_COURANT_TENSION 100

uint32_t period_pwm = 20; // période de découpage en microsecondes
#define TE_MESURES 1000 // période d'échantillonnage en microsecondes
#define N_MESURES 1000 // nombre d'échantillons pour un cycle de mesure


uint32_t compteur_adc;
uint8_t multiplex[2] = {0,1}; // voie 0 : U, voie 1 : I
float v0_mult, v1_mult;
float somme_v0,somme_v1;
uint8_t data[8];
char timer3_clockBits;
uint16_t diviseur[6] = {0,1,8,64,256,1024};
```

La fonction `init_pwm_timer1_fixe` programme le Timer 1 pour qu'il délivre un signal PWM de fréquence et de rapport cyclique choisis sur la sortie OCR1A (sortie D11 sur l'arduino MEGA, sortie D9 sur les arduinos UNO et YUN). La période est donnée en microsecondes, le rapport cyclique est un flottant entre 0 et 1.

```
void init_pwm_timer1_fixe(uint32_t period, float ratio) { // rapport cyclique fixe
    char clockBits;
    TCCR1A = 0;
    TCCR1A |= (1 << COM1A1); //Clear OCnA/OCnB/OCnC on compare match, set OCnA/OCnB/O
    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;
    uint32_t 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
    OCR1A = ratio*icr;
    TCNT1 = 0; // mise à zéro du compteur
    TCCR1B |= clockBits; // déclenchement du compteur
}
```

La fonction `init_interrupt_timer3` programme le Timer 3 pour qu'il délivre des interruptions périodiques afin de faire fonctionner le convertisseur analogique-numérique (ADC). La période est donnée en microsecondes.

```
void init_interrupt_timer3(uint32_t period) { // interruptions pour l'ADC
    TCCR3A = 0;
    TCCR3B = (1 << WGM32); // mode CTC avec OCR3A pour le maximum
    uint32_t top = (F_CPU/1000000*period);
    timer3_clockBits = 1;
    while ((top>0xFFFF)&&(timer3_clockBits<5)) {
      timer3_clockBits++;
      top = (F_CPU/1000000*period/diviseur[timer3_clockBits]);
    }
    OCR3A = top; // période
    TIMSK3 = (1 << OCIE3A); // interruption lorsque TCNT3 = OCR3A
    TCCR3B |= timer3_clockBits;
}
```

La fonction suivante configure l'ADC. Le `prescaler` permet de fixer la vitesse de la conversion. une valeur de 7 convient bien pour notre usage.

```
void adc_init(uint8_t prescaler) {
  ADCSRA = 0;
  ADCSRA |= (1 << ADEN);
  ADCSRA |= prescaler;
  ADCSRB = 0;
}
```

La fonction suivante est appelée lors des interruptions générées par le Timer 3. Elle effectue les conversions analogiques-numériques des entrées A0 et A1 et met à jour les sommes des valeurs. Ses sommes sont des flottants pour éviter les débordements. Lorsque le nombre de mesures effectuées atteint la valeur maximale, la valeur moyenne de chaque voie est transmise à l'ordinateur. La valeur moyenne est un flottant compris entre 0 et 1024 (convertisseur A/N 10 bits). On le transmet sous la forme d'un entier 32 bits après l'avoir multiplié par 0xFFFF, selon le principe du nombre à virgule fixe. Le cycle de mesures se répète périodiquement.

```
ISR(TIMER3_COMPA_vect) {
    ADMUX = 0b01000000 | multiplex[0];
    ADCSRA |= 0b01000000; // start ADC
```

```
   while (ADCSRA & 0b01000000); // wait ADC
   uint16_t v0 = (ADCL | (ADCH << 8));
   ADMUX = 0b01000000 | multiplex[1];
   ADCSRA |= 0b01000000; // start ADC
   while (ADCSRA & 0b01000000); // wait ADC
   uint16_t v1 = (ADCL | (ADCH << 8));
   somme_v0 += v0;
   somme_v1 += v1;
   compteur_adc++;
   if (compteur_adc==N_MESURES) {
     TCCR3B &= ~7; // arrêt des interruptions
     uint32_t v0_moy = somme_v0/N_MESURES*0xFFFF;
     uint32_t v1_moy = somme_v1/N_MESURES*0xFFFF;
     data[0] = v0_moy >> 24;
     data[1] = v0_moy >> 16;
     data[2] = v0_moy >> 8;
     data[3] = v0_moy;
     data[4] = v1_moy >> 24;
     data[5] = v1_moy >> 16;
     data[6] = v1_moy >> 8;
     data[7] = v1_moy;
     Serial.write(data,8);
     compteur_adc = 0;
     somme_v0 = somme_v1 = 0.0;
     TCCR3B |= timer3_clockBits; //reprise des interruptions
   }

}
```

La fonction `setup` effectue l'initialisation de la liaison série avec l'ordinateur. La sortie D10 est mise au niveau haut pour activer le pont L6203 (entrée ENABLE). Le signal PWM est généré sur la borne D11 dans le cas de l'arduino MEGA, la borne D9 dans le cas de l'arduino UNO ou de l'arduino YUN. Ces bornes doivent être configurées en sortie.

```
void setup() {
   Serial.begin(115200);
   char c;
   Serial.setTimeout(0);
  c = 0;
  Serial.write(c);
  c = 255;
  Serial.write(c);
  c = 0;
  Serial.write(c);
   pinMode(10,OUTPUT);
   digitalWrite(10,HIGH); // port ENABLE du pont
#if defined(__AVR_ATmega2560__)
```

```
    pinMode(11,OUTPUT);
#else
    pinMode(9,OUTPUT);
#endif
   somme_v0 = somme_v1 = 0.0;
   adc_init(7);
}
```

La fonction suivante lit une commande provenant de l'ordinateur. Le rapport cyclique est lu sous la forme d'un entier 16 bits, puis converti en flottant. Les deux timers sont déclenchés.

```
void lecture_courant_tension() {
   uint32_t c1,c2;
   float rapport;
   while (Serial.available() < 2) {};
   c1 = Serial.read();
   c2 = Serial.read();
   rapport = 1.0/0xFFFF*((c1 << 8) | c2);
   init_pwm_timer1_fixe(period_pwm,rapport);
   init_interrupt_timer3(TE_MESURES);

}


void lecture_serie() {
  char com;
  if (Serial.available() > 0) {
    com = Serial.read();
    if (com==LECTURE_COURANT_TENSION) lecture_courant_tension();
  }
}

void loop() {
  lecture_serie();

}
```

4. Programme python

Ce programme est constitué d'une classe `Arduino`. Dans le constructeur, la communication série avec l'arduino est établie. Les résistances des amplificateurs de mesures sont définies dans le constructeur, ce qui permet de calculer le facteur multiplicatif à appliquer aux valeurs numériques transmise par l'arduino pour les convertir en la tension U en Volts et le courant I en Ampères. La constante AREF est la tensions lue au

voltmètre sur la broen AREF de l'arduino, qui est la tension de référence utilisée par le convertisseur A/N.

mesuresCourantTensionPuissance.py

```
# -*- coding: utf-8 -*-
import serial
import numpy
import time
from matplotlib.pyplot import *
import threading

class Arduino():
    def __init__(self,port):
        self.ser = serial.Serial(port,baudrate=115200)
        c_recu = self.ser.read(1)
        while ord(c_recu)!=0:
            c_recu = self.ser.read(1)
        c_recu = self.ser.read(1)
        while ord(c_recu)!=255:
            c_recu = self.ser.read(1)
        c_recu = self.ser.read(1)
        while ord(c_recu)!=0:
            c_recu = self.ser.read(1)
        self.LECTURE_COURANT_TENSION = 100
        AREF = 4.83
        R1 = 47.67
        R2 = 18.05
        R3 = 9.93
        R4 = 47.67
        R = 1.0
        GAIN0 = R2/R1
        GAIN1 = 1+R4/R3
        self.v0_mul = AREF*1.0/(1024*0xFFFF)/GAIN0
        self.v1_mul = AREF*1.0/(1024*0xFFFF)/GAIN1/R

    def close(self):
        self.ser.close()

    def write_int16(self,v):
        v = numpy.int16(v)
        char1 = (v & 0xFF00) >> 8
        char2 = (v & 0x00FF)
        self.ser.write(chr(char1))
        self.ser.write(chr(char2))

    def read_int32(self):
        b1 = ord(self.ser.read(1))
        b2 = ord(self.ser.read(1))
        b3 = ord(self.ser.read(1))
```

```
        b4 = ord(self.ser.read(1))
        return b1*0x1000000 + b2*0x10000 + b3*0x100 + b4

    def lecture_courant_tension(self,rapport_cyclique,n=2):
        self.ser.write(chr(self.LECTURE_COURANT_TENSION))
        self.write_int16(rapport_cyclique*0xFFFF)
        for k in range(n):
            u = self.read_int32()*self.v0_mul
            i = self.read_int32()*self.v1_mul
        print("U = %f, I = %f"%(u,i))
        return (u,i)
```

La fonction `lecture_courant_tension` envoit à l'arduino un rapport cyclique pour le signal de commande du hacheur. L'argument `n` est le nombre de cycles de mesure à faire avant de renvoyer la tension moyenne et le courant moyen. Dans la plupart des cas, une valeur de 2 convient. Pour des dipôles dont les effets thermiques sont importants, il peut être nécessaire d'augmenter le nombre de cycles.

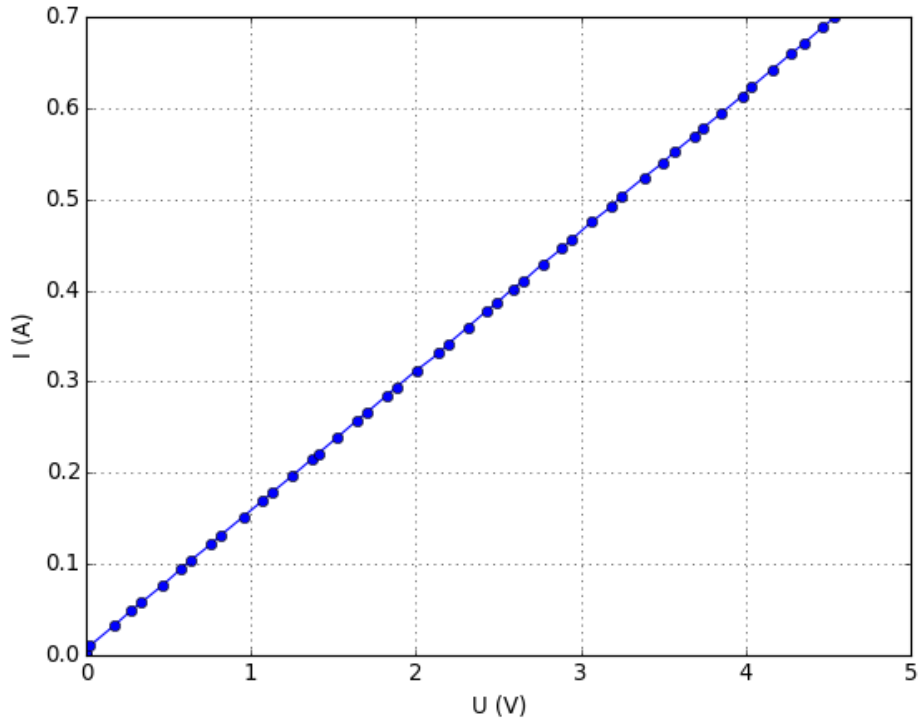
Voici un exemple d'utilisation, consistant à faire appliquer 100 valeurs de rapport cyclique échelonnées entre 0 et 0.5. À la fin des mesures, on applique un rapport 0 pour annuler le courant dans le dipôle.

```
def test():
    ard = Arduino(4)
    liste_u = []
    liste_i = []
    rapports = numpy.linspace(start=0.0,stop=0.5,num=100)
    for r in rapports:
        (u,i) = ard.lecture_courant_tension(r,2)
        liste_u.append(u)
        liste_i.append(i)
    ard.lecture_courant_tension(0)
    numpy.savetxt("data.txt",[liste_u,liste_i])
    figure()
    plot(liste_u,liste_i,"o")
    ylabel("I (A)")
    xlabel("U (V)")
    grid()
    show(block=True)
    ard.close()
```

On commence par faire un test avec une résistance de puissance $6,8\ \Omega$:

```
import numpy
from matplotlib.pyplot import *
```

```
[U,I] = numpy.loadtxt("resistance6.8.txt")
figure()
plot(U,I,"o-")
ylabel("I (A)")
xlabel("U (V)")
grid()
```

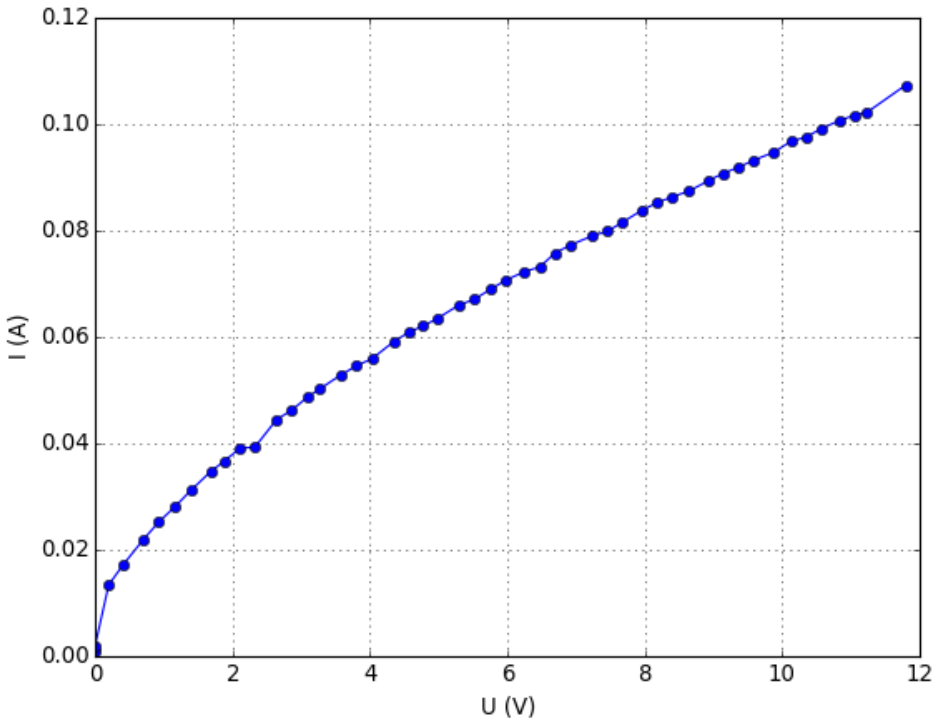


5. Applications

6. Caractéristique d'une lampe à incandescence

Il s'agit d'une petite lampe à incandescence fonctionnant sous 12 V avec un courant de 100 mA. Le nombre de cycles de mesures pour chaque point est $n = 10$.

```
[U,I] = numpy.loadtxt("lampe-n10.txt")
figure()
plot(U,I,"o-")
ylabel("I (A)")
xlabel("U (V)")
grid()
```

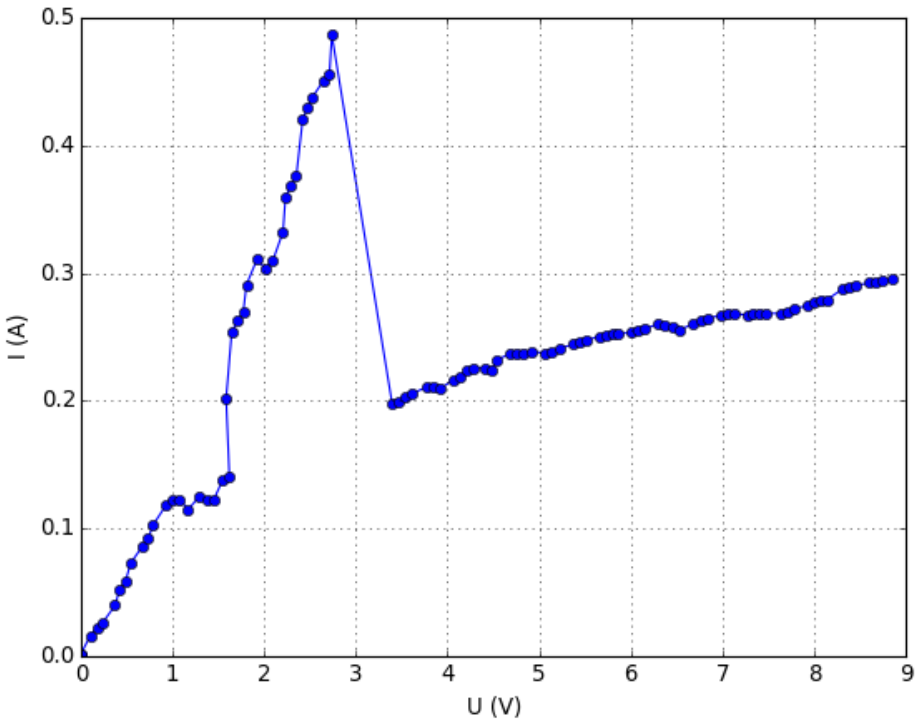


Une lampe à incandescence est souvent utilisée dans les circuits en raison de sa caractéristique non linéaire, particulièrement marquée dans les courants faibles (alors que le filament n'émet pas dans le visible).

7. Caractéristique d'un moteur

Il s'agit d'un petit moteur à courant continu de tension nominale 6 V.

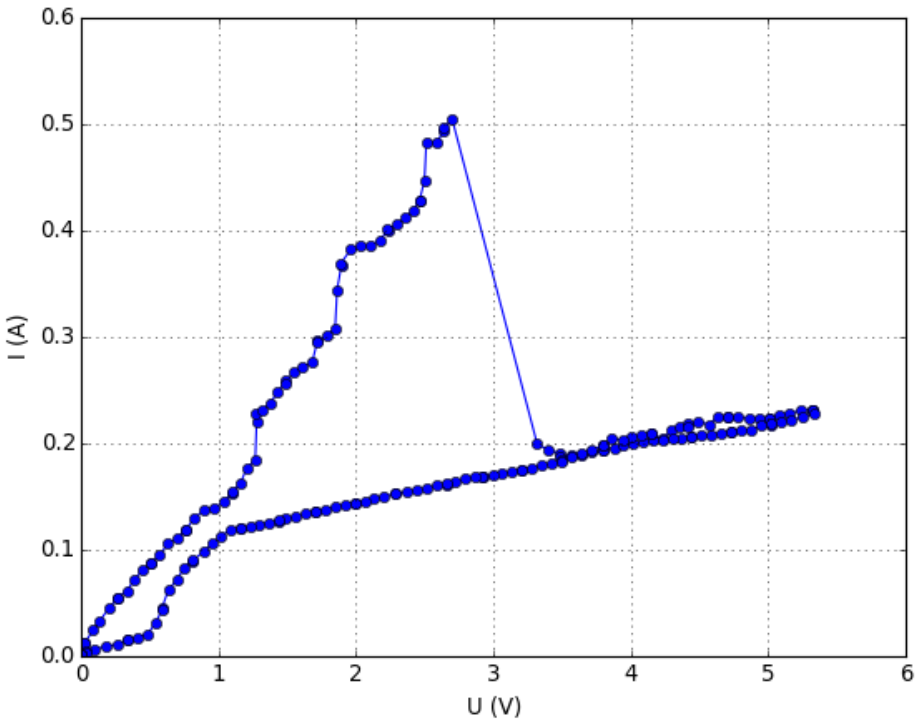
```
[U,I] = numpy.loadtxt("moteur.txt")
figure()
plot(U,I,"o-")
ylabel("I (A)")
xlabel("U (V)")
grid()
```



Dans un premier temps, le moteur reste à l'arrêt et le courant augmente jusqu'à environ 500 mA. Le démarrage du moteur se traduit par une chute du courant (le couple diminue) mais on remarque aussi un saut de la tension. Cela montre que le convertisseur ne parvient pas dans cette phase à imposer la tension. Lorsque la tension augmente après le démarrage du moteur, sa vitesse augmente et le courant augmente car le couple augmente (le moteur est branché sur un réducteur).

Il est intéressant de réduire la tension après la phase de montée :

```
[U,I] = numpy.loadtxt("moteur-2.txt")
figure()
plot(U,I,"o-")
ylabel("I (A)")
xlabel("U (V)")
grid()
```



Dans la phase descendante, le moteur continue de tourner jusqu'à une vitesse très faible. Il s'arrête de tourner en dessous de $0,5\ V$, alors qu'il faut appliquer au moins $2,5$ sur le moteur au repos pour le faire tourner.