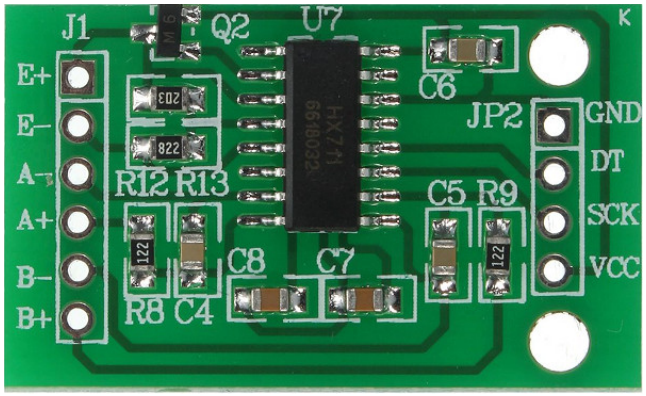
# Jauge de déformation

## 1. Introduction

Ce document montre comment utiliser un capteur à jauge de déformation (ou jauge de contrainte) à l'aide d'un amplificateur HX711 (Avia Semiconductor) relié à une platine Arduino.
Une jauge de déformation est constituée d'un circuit métallique (en forme de grille) gravé sur une plaque fine. Lorsque la plaque est soumise à une déformation, la résistance du circuit varie légèrement, proportionnellement à la déformation. Dans le domaine de déformation élastique, la déformation est proportionnelle à la force appliquée. On peut donc, après étalonnage, utiliser la jaune de déformation pour mesurer une force (ou une contrainte, qui est une force par unité de surface).
Un capteur de force est constitué de quatre jauges de déformation, dont les résistances sont reliées pour former un pont de Wheatstone (voir schéma plus loin).
L'amplificateur HX711 est un convertisseur 24 bits muni d'un amplificateur à fort gain (128, 64 ou 32), spécialement conçu pour mesurer la faible tension délivrée par un capteur de force. L'interface numérique comporte une borne SCK (pour le signal d'horloge) et une borne DT (pour la transmission des bits de donnée). La fréquence de conversion est de 10 Hz ou de 80 Hz, en fonction de la tension appliquée sur la borne RATE. Voici une photo de la platine que nous utilisons et le brochage du HX711 :

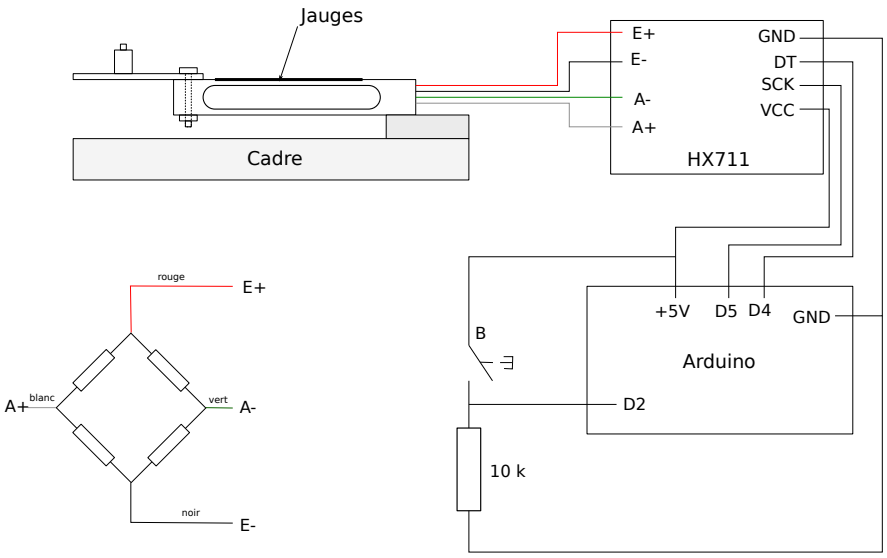



| | | | | | |
|---|---|---|---|---|---|
| Regulator Power | VSUP | 1 | 16 | DVDD | Digital Power |
| Regulator Control Output | BASE | 2 | 15 | RATE | Output Data Rate Control Input |
| Analog Power | AVDD | 3 | 14 | XI | Crystal I/O and External Clock Input |
| Regulator Control Input | VFB | 4 | 13 | XO | Crystal I/O |
| Analog Ground | AGND | 5 | 12 | DOUT | Serial Data Output |
| Reference Bypass | VBG | 6 | 11 | PD_SCK | Power Down and Serial Clock Input |
| Ch. A Negative Input | INNA | 7 | 10 | INPB | Ch. B Positive Input |
| Ch. A Positive Input | INPA | 8 | 9 | INNB | Ch. B Negative Input |

Sur cette platine, la borne RATE n'est pas connectée, ce qui implique que la conversion se fait à une fréquence de 10 Hz. Cette fréquence d'échantillonnage est suffisante pour des applications quasi statiques de type pesée ou mesure de force lentement variable.

## 2. Montage mécanique et électrique

Nous utilisons un capteur de force en forme de barre, à laquelle on applique une déformation de flexion. La barre est fixée à une extrémité sur un cadre. Sur l'autre extrémité, on fixe un petit plateau horizontal sur lequel on viendra poser des masses pour l'étalonnage et pour le test. Le capteur comporte quatre fils. Les fils rouge et noir permettent d'alimenter le pont et sont reliés aux bornes E+ et E- de la platine, elles-même reliées à un régulateur de tension (la platine est alimentée par l'arduino). Les fils vert et blanc délivrent la tension à mesurer. On les branche soit sur les entrées A- et A+ (Voie A), soit sur les entrées B- et B+ (Voie B). La voie A permet d'utiliser un gain d'amplification de 128 ou 64 alors que la voie B fontionne avec un gain de 32. Nous utilisons la voie A. Les bornes GND et VCC sont reliées aux bornes GND et +5V de l'arduino. Les bornes SCK et DT sont reliées aux entrées D5 et D4 de l'arduino. Le montage comporte aussi un bouton poussoir (optionnel) qui permettra de piloter manuellement l'étalonnage. Lorsqu'on appuie sur le bouton, une tension +5V est appliquée sur l'entrée D2 (cette entrée doit pouvoir déclencher une interruption).




## 3. Programme arduino

Le programme arduino récupère les nombres de 24 bits délivrés par le HX711, à une fréquence de 10 Hz, et effectue (éventuellement) un filtrage par convolution afin de lisser le bruit. Par défaut, la valeur obtenue est affichée sur la console et le bouton poussoir permet de faire l'étalonnage : une première pression permet de régler le zéro, une deuxième pression permet de faire l'étalonnage avec une masse connue, une troisième pression permet de revenir à l'état initial. Il est aussi possible de faire un étalonnage au démarrage en définissant la macro `CALIB_START`. Dans ce cas, il faut ouvrir une console série et suivre les instructions : laisser sans poids pendant 10 s puis poser un poids de 50 g et le laisser 10 s (les mesures se font pendant les 2 dernières secondes). La valeur d'offset et d'échelle sont alors affichées ; il faut les reporter avec les macros `OFFSET` et `ECHELLE` avant de recompiler et recharger le

programme. Ces valeurs son prises en compte si la macro `NO_CALIB` est définie. Dans le cas contraire, l'étalonnage est fait à chaque utilisation, par pression sur le bouton. Un pilotage peut aussi être effectué depuis un programme python (décrit plus loin), ce qui permet de récupérer le flux des valeurs échantillonnées à 10 Hz.
Remarque : en principe, la détermination de l'échelle doit être faite une seule fois pour une jauge donnée. Le réglage du zéro (valeur de l'offset) dépend en revanche de l'orientation de la jauge (verticale ou horizontale). Le plus souvent, il n'est pas nécessaire de faire le réglage du zéro (si l'échelle est déjà connue) car on peut appliquer un décalage en post-traitement en considérant que la force est nulle au début de l'enregistrement. Si la force mesurée n'est pas verticale, il faudra tout de même prévoir un moyen de placer le plateau horizontalement pour faire l'étalonnage avec un poids.

arduinoHX711.ino

```
#define DOUT 4
#define SCK 5
#define BUTTON 2
#define TRIGGER 3
#define GAIN 1 // 1 = 128 channel A, 3 = 64 channel A, 2 = 32 channel B
#define NVAL 21 // taille du tampon (impair)
#define MCAL 50000.0 // poids de calibration en mg
//#define FILTRAGE  // filtrage passe-bas
#define FCOUP 0.05 // fréquence de coupure du filtre (relative à 10 Hz)
#define START_TRANSMISSION 100
#define STOP_TRANSMISSION 101
#define ETALONNAGE 102
#define FACT_TRANS 100
//#define CALIB_START // étalonnage au démarrage
#define NO_CALIB // pas d'étalonnage, valeurs de offset et echelle codées en dur (déterm
#define OFFSET 35743.81 // offset et échelle déterminés par un étalonnage au démarrage
#define ECHELLE 0.07561

volatile uint8_t d[3];
volatile unsigned long valeur;
volatile long data[NVAL];
volatile float force[NVAL];
volatile uint16_t indice;
volatile uint8_t filler = 0x00;
volatile float accum;
volatile int8_t k,j;
volatile float offset, echelle;
uint8_t npress;
bool etalbout;
volatile uint8_t reduc;
volatile int16_t compteur_reduc;
volatile bool transmission;
volatile int32_t transbuffer[0];
volatile float b[NVAL]; // coefficients du filtre
bool triggered;
```

La fonction `init_pwm_timer1` configure le Timer 1 pour qu'il génère des interruptions périodiques. La période est donnée en microsecondes.

```
void init_pwm_timer1(uint32_t period) {
    char clockBits;
    uint32_t icr;
    uint16_t diviseur[6] = {0,1,8,64,256,1024};
    cli();
    TCCR1A = 0;
    TCCR1A |= (1 << COM1A1); //Clear OC1A on compare match when upcounting, set OC1A on
    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<5)) { // 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
   TCNT1 = 0; // mise à zéro du compteur
   OCR1A = icr*0.5;
   TCCR1B |= clockBits; // déclenchement du compteur
   sei(); // activation des interruptions
}
```

La fonction `ISR(TIMER1_OVF_vect)` est appelée lors des interruptions générées par le Timer 1 (toutes les 100 ms). La première opération effectuée par cette fonction est la communication avec le HX711, qui consiste à attendre que DOUT soit au niveau haut puis à lire les 24 bits de données. La lecture de 8 bits se fait avec la fonction `shiftIn`, qui envoie pour chaque bit une impulsion d'horloge sur SCK et lit le bit renvoyé par le HX711. Après lecture des 24 bits (stockés dans le tableau `d`), on doit appliquer sur SCK un nombre d'impulsions correspondant au gain d'amplification souhaité. Le nombre obtenu est mis sous la forme d'un entier 32 bits (`valeur`) puis stocké dans un tampon (`data`). Si le filtrage est activé, ce tampon, contenant NVAL échantillons, permet d'effectuer un filtrage par convolution. Le résultat de la convolution est un flottant (`accum`), auquel on applique un décalage et un facteur d'échelle, pour obtenir une valeur en milligrammes (ou une autre unité). Le résultat est stocké dans un second tampon (`force`), qui sera utilisé pour l'étalonnage. Si la transmission avec le PC est désactivée, la valeur est simplement affichée sur la console. Si la transmission est activée, elle est transmise sous la forme d'un entier 32 bits, après avoir été multipliée par `FACT_TRANS`. La transmission peut se faire avec une réduction de la fréquence d'échantillonnage, définie par `reduc`. Par exemple, si `reduc=10`, un échantillon est transmis toutes les secondes. Si la force mesurée varie très lentement, on pourra ainsi choisir la fréquence finale en fonction de l'application, tout en gardant un échantillonnage de base à 10 Hz pour effectuer le filtrage du bruit. On notera que le bruit peut avoir plusieurs origines : le bruit dans la tension mesurée (d'origine thermique), le bruit introduit par l'amplificateur, le bruit de quantification, et le bruit dans le phénomène physique étudié. Pour ce dernier, il pourra être nécessaire d'effectuer un filtrage passe-bas avec une fréquence de coupure très basse (par rapport à 10 Hz), ce qui nécessitera éventuellement d'augmenter la taille du tampon (`NVAL`), si l'arduino utilisé possède assez de mémoire.

```
ISR(TIMER1_OVF_vect) {
   while (digitalRead(DOUT)==HIGH) {;}
   d[2] = shiftIn(DOUT,SCK,MSBFIRST);
   d[1] = shiftIn(DOUT,SCK,MSBFIRST);
   d[0] = shiftIn(DOUT,SCK,MSBFIRST);
   for (unsigned int i=0; i<GAIN; i++) {
      digitalWrite(SCK,HIGH);
      delayMicroseconds(1);
      digitalWrite(SCK,LOW);
      delayMicroseconds(1);
   }
  if (d[2] & 0x80) {
    filler = 0xFF;
  } else {
    filler = 0x00;
  }
   valeur = ( static_cast<unsigned long>(filler) << 24
     | static_cast<unsigned long>(d[2]) << 16
      | static_cast<unsigned long>(d[1]) << 8
      | static_cast<unsigned long>(d[0]) );
   data[indice] = static_cast<long>(valeur);

#ifdef FILTRAGE
   accum = 0.0;
   j = indice;
   for (k=0; k<NVAL; k++) {
      accum += b[k]*data[j];
      j -= 1;
      if (j<0) j = NVAL-1;
   }
#else
  accum = data[indice];
#endif

   accum = (accum-offset)*echelle;
   force[indice] = accum;

   indice += 1;
   if (indice==NVAL) indice = 0;
   if (transmission&&triggered) {
     compteur_reduc += 1;
     if (compteur_reduc==reduc) {
        compteur_reduc = 0;
        transbuffer[0] = accum * FACT_TRANS;
        Serial.write((uint8_t *)transbuffer,4);
     }
   }
}
```

La fonction `etalonnage` effectue l'étalonnage. L'action réalisée dépend de l'ordre de l'appel. Le premier appel effectuer le réglage du zéro, le deuxième appel effectue le réglage de l'échelle avec un poids de 50 g, le troisième appel initialise l'étalonnage. Cette fonction était initialement prévue pour être appelée par une interruption déclenchée par un bouton poussoir. Elle n'est pas utilisée lorsque l'étalonnage est fait au démarrage. Le premier appel calcule la moyenne des valeurs stockées dans le tampon `force`, afin d'en déduire le décalage à appliquer (réglage du zéro). Ce réglage doit être fait dans une configuration mécanique où l'on

souhaite avoir une valeur de force nulle, par exemple la position horizontale du plateau vide dans notre montage. Le deuxième appel calcule le facteur d'échelle en fonction de la valeur de `MCAL`. Ce réglage doit être effectué lorsqu'une masse étalon de valeur MCAL est posée sur le plateau. Le troisième appel initialise le décalage et le facteur d'échelle.

```
void etalonnage() {
   delay(300);
   if (etalbout) if (digitalRead(BUTTON)==LOW) return;
   npress += 1;
   float m;
     for (k=0; k<NVAL; k++) {
        m += force[k];
     }
    m /= NVAL;
   if (npress==1) {
     offset = m;
   }
   else if (npress==2) {
      echelle = MCAL/m;
   }
   else if (npress==3) {
      echelle = 1.0;
      offset = 0.0;
      npress = 0;
   }
}
```

La fonction `calcul_filtre` calcule les coefficients d'un filtre RIF passe-bas (filtre de convolution) dont la fréquence de coupure (relative à la fréquence d'échantillonnage) est donnée.

```
void calcul_filtre(float fc) {
  int P = (NVAL-1)/2;
  int k;
  float u;
  int i;
  for (i=0; i<NVAL; i++) {
      k = i-P;
      u = 2*PI*k*fc;
      if (u==0) b[i] = 2*fc;
      else b[i] = 2*fc*sin(2*PI*k*fc)/(2*PI*k*fc);
      b[i] *= 0.54+0.46*cos(2*PI*k/(2*P)); // Hamming
  }
}
```

Si la macro `TRIGGER` est définie, sa valeur donne l'entrée permettant de déclencher l'enregistrement avec un front montant. Ce front déclenche une interruption qui appelle la fonction suivante :

```
void start() {
  if (triggered==false) {
```

```
    triggered = true;
  }
}
```

La fonction `setup` initialise la liaison série, configure les entrées et sorties, initialise les tampons et les variables, calcule les coefficients du filtre et déclenche le Timer. La pression sur le bouton poussoir déclenche une interruption qui exécute la fonction `etalonnage`. Sachant que `FCOUP=0.05` et `NVAL=21`, le filtre passe-bas possède 21 coefficients et a une fréquence de coupure de 0,5 Hz. Le facteur de réduction est `reduc=5` donc la valeur de la force est affichée sur la console toutes les demi secondes.

```
void setup() {
  char c;
  Serial.begin(115200);
  Serial.setTimeout(0);
  c = 0;
  Serial.write(c);
  c = 255;
  Serial.write(c);
  c = 0;
  Serial.write(c);
  pinMode(BUTTON,INPUT);
  attachInterrupt(digitalPinToInterrupt(BUTTON),etalonnage,RISING);
  pinMode(DOUT,INPUT);
  pinMode(SCK,OUTPUT);
#ifdef TRIGGER
  pinMode(TRIGGER,INPUT);
  triggered = false;
#else
  triggered = true;
#endif
  attachInterrupt(digitalPinToInterrupt(TRIGGER),start,RISING);
  digitalWrite(SCK,LOW);
  for (indice=0; indice<NVAL; indice++) {
      data[indice] = 0.0;
      force[indice] = 0.0;
  }
  indice = 0;
#ifdef NO_CALIB
  offset = OFFSET;
  echelle = ECHELLE;
#else
  offset = 0;
  echelle = 1;
#endif
  npress = 0;
  reduc = 1;
  transmission = false;
  etalbout = true;
  calcul_filtre(FCOUP);
  init_pwm_timer1(100000); // échantillonnage à 10 Hz
  delay(3000);
#ifdef CALIB_START
  offset = 0;
```

```
  echelle = 1;
  etalbout = false;
  Serial.println("Mise à zéro ...");
  delay(10000);
  etalonnage();
  Serial.println("Placer une masse de 50 g ...");
  delay(10000);
  etalonnage();
  Serial.print("offset = "); Serial.println(offset);
  Serial.print("echelle = "); Serial.println(echelle,5);

#endif
}
```

La fonction `start_transmission` répond à une demande du programme python de démarrer une transmission des données avec un facteur de réduction choisi. La fonction `stop_transmission` stoppe cette transmission.

```
void start_transmission() {
  uint32_t c1,c2;
  while (Serial.available()<2) {};
  c1 = Serial.read();
  c2 = Serial.read();
  reduc = ((c1<<8) | c2);
  compteur_reduc = 0;
  transmission = true;
  Serial.flush();
  char c;
  c = 0;
  Serial.write(c);
  c = 255;
  Serial.write(c);
  c = 0;
  Serial.write(c);
}

void stop_transmission() {
  transmission = false;
}
```

La fonction `loop` scrute le port série pour savoir si le programme python demande à effectuer un démarrage de transmission, un arrêt de transmission, ou un échantillonnage. Il est possible d'ajouter d'autres opérations dans cette fonction loop. L'acquisition des données du capteur et leur traitement se font entièrement dans la fonction `ISR(TIMER1_OVF_vect)`, appelée périodiquement tous les dixièmes de secondes. Le temps restant entre deux interruptions est largement suffisant pour effectuer d'autres opérations, par exemple déclencher une action en fonction des valeurs de force mesurées (stockées dans le tampon `force`). D'une manière générale, il est important que les fonctions de communication avec un périphérique ne bloquent pas l'exécution du code localisé dans la fonction `loop`. Dans le cas présent, cette communication se fait dans la fonction d'interruption. L'instruction `while (digitalRead(DOUT)==HIGH) {;}`, qui boucle tant que le HX711 n'est pas disponible, ne doit pas être exécutée dans la fonction `loop`, car cela aurait pour effet de boucler pendant une durée de l'ordre de 100 ms. Elle est

exécutée dans la fonction d'interruption appelée toutes les 100 ms, ce qui fait que la sortie DOUT est très probablement déjà au niveau HIGH lorsqu'elle est exécutée.

```
void loop() {
   char com;
   if (Serial.available()>0) {
      com = Serial.read();
      if (com==START_TRANSMISSION) start_transmission();
      else if (com==STOP_TRANSMISSION) stop_transmission();
      else if (com==ETALONNAGE) {
        etalbout = false;
        etalonnage();
        etalbout = true;
      }
   }
   // autres actions à placer ici, en fonction de la force mesurée
}
```

## 4. Programme python

Le programme python permet de récupérer les valeurs de la force ou de déclencher les opérations d'étalonnage. Il comporte une classe `Arduino`.

arduinoHX711.py

```
import numpy
import serial
import time

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.START_TRANSMISSION = 100
        self.STOP_TRANSMISSION = 101
        self.ETALONNAGE = 102
        self.FACT_TRANS = 100
        self.FECHANT = 10.0

    def close(self):
        self.ser.close()
    def write_int8(self,v):
        char = int(v&0xFF) # nécessaire pour les nombres négatifs
        self.ser.write((char).to_bytes(1,byteorder='big'))
    def write_int16(self,v):
        v = numpy.int16(v)
```

```
        char1 = int((v & 0xFF00) >> 8)
        char2 = int((v & 0x00FF))
        self.ser.write((char1).to_bytes(1,byteorder='big'))
        self.ser.write((char2).to_bytes(1,byteorder='big'))
    def write_int32(self,v):
        v = numpy.int32(v)
        char1 = int((v & 0xFF000000) >> 24)
        char2 = int((v & 0x00FF0000) >> 16)
        char3 = int((v & 0x0000FF00) >> 8)
        char4 = int((v & 0x000000FF))
        self.ser.write((char1).to_bytes(1,byteorder='big'))
        self.ser.write((char2).to_bytes(1,byteorder='big'))
        self.ser.write((char3).to_bytes(1,byteorder='big'))
        self.ser.write((char4).to_bytes(1,byteorder='big'))

    def start_transmission(self,reduc=1):
        self.write_int8(self.START_TRANSMISSION)
        self.write_int16(reduc)
        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)
    def stop_transmission(self):
        self.write_int8(self.STOP_TRANSMISSION)
    def etalonnage(self):
        self.write_int8(self.ETALONNAGE)

    def lecture(self):
        buf = self.ser.read(4)
        x = buf[3]*0x1000000 + buf[2]*0x10000 + buf[1]*0x100 + buf[0]
        if buf[3]&0x40: #bit de signe = 1
            x = x-0x100000000
        x = x/self.FACT_TRANS
        return x
```

Le script suivant permet d'effectuer un étalonnage puis de tracer en temps réel la force pendant une durée choisie. À la fin de l'acquisition, les données sont enregistrées dans une fichier texte.

arduinoHX711-animate.py

```
import numpy
from matplotlib.pyplot import *
import matplotlib.animation as animation
from arduinoHX711 import *

ard = Arduino("COM5")
calib = False # étalonnage déjà fait
if calib:
    input('Mise à zéro ...')
    time.sleep(3)
    ard.etalonnage()
```

```
    input('Placer une masse 50 g pour étalonnage ...')
    time.sleep(3)
    ard.etalonnage()

fe = 10.0
duree = 60
reduc = 1
te = reduc/fe
N = int(duree/te)

t = numpy.linspace(0,duree,N)
F = numpy.zeros(N)

fig,ax = subplots()
line0, = ax.plot(t,F)
ax.grid()
ax.set_xlabel("t (s)")
ax.set_ylabel("F (mg)")


ard.start_transmission(reduc)
n=0
temps = numpy.array([])
force = numpy.array([])
t = 0.0
def animate(i):
    global ax,line0,ard,n,N,te,temps,force,t
    if n<N:
        f = ard.lecture()
        temps = numpy.append(temps,t)
        force = numpy.append(force,f)
        t += te
        n += 1
        line0.set_xdata(temps)
        line0.set_ydata(force)
        ax.axis([0,temps[n-1],force.min(),force.max()])

ani = animation.FuncAnimation(fig,animate,N,interval=te*1000*0.8)
show()
ard.stop_transmission()
ard.close()
figure()
plot(temps,force)
grid()
xlabel("t")
ylabel("F")
numpy.savetxt("force.txt",numpy.array([temps,force]).T,delimiter="\t",fmt="%.6e",header=
figure()
plot(temps,force)
grid()
xlabel("t")
ylabel("F")
```