Université du Littoral – Côte d'Opale – Calais
Unité d'ouverture, licence MSPI et SV, 2me et 3me année
Licence MSPI non-spécialisée, 1re année S1
Département de physique

Enseignement autour de micro-contrôleur *Arduino Uno*

Robin Bocquet, montages éléctroniques, rédaction de l'énoncé 2017-2020
Arnaud Cuisset, carnet des charges et gestion du projet 2017-2019
Pierre Kulinski, réalisateur montages Arduino, soutien technique, Dk 2017
Pascal Masselin, montages éléctroniques, intervention en TP L1 S1, Calais 2021
Wilfried Montagnier, soutien technique, Calais
Dmitrií Sadovskií *, développement et traitement informatique, rédaction de l'énoncé

Calais, automne 2025

1 Introduction

Le but de ces travaux pratiques est de vous initier à l'utilisation des microcontroleurs qui ont littéralement envahis le monde technologique d'aujourd'hui. Un microcontrôleur n'est rien d'autre qu'un microprocesseur à jeu d'instructions limité spécialisé dans les communications avec l'extérieur. On retrouve ce type de composants dans les voitures par exemple où ils gèrent l'ensemble des capteurs et éléments de sécurité du véhicule, dans les drones où ils gèrent les capteurs de vitesse, d'altitude, de lacet, roulis et tangage, dans les robots où bien souvent des radars anti-collisions sont mis en place ou bien encore dans les imprimantes 3D où les moteurs ainsi que les positionnements sont gérés par ce type de composant. Les microcontrôleurs ont été développés dans les années 80 mais ont réellement diffusés dans le grand public depuis 2005 grâce au travail d'un groupe d'italiens dans le monde du logiciel libre qui a développé une plateforme logiciel simplifiant l'utilisation de ces composants. Il s'agit des développements connus sous la bannière ARDUINO et repris maintenant sous le nom GENUINO, adhérant à la charte du développement «libre». L'ensemble des documentations sur la carte ARDUINO, ainsi qu'une mine d'exemples sont consultables sur son site. Enfin, dans cette introduction, notez que vous trouvez nombre de tutoriaux sur le « net » pour vous aider à découvrir le matériel et le logiciel.

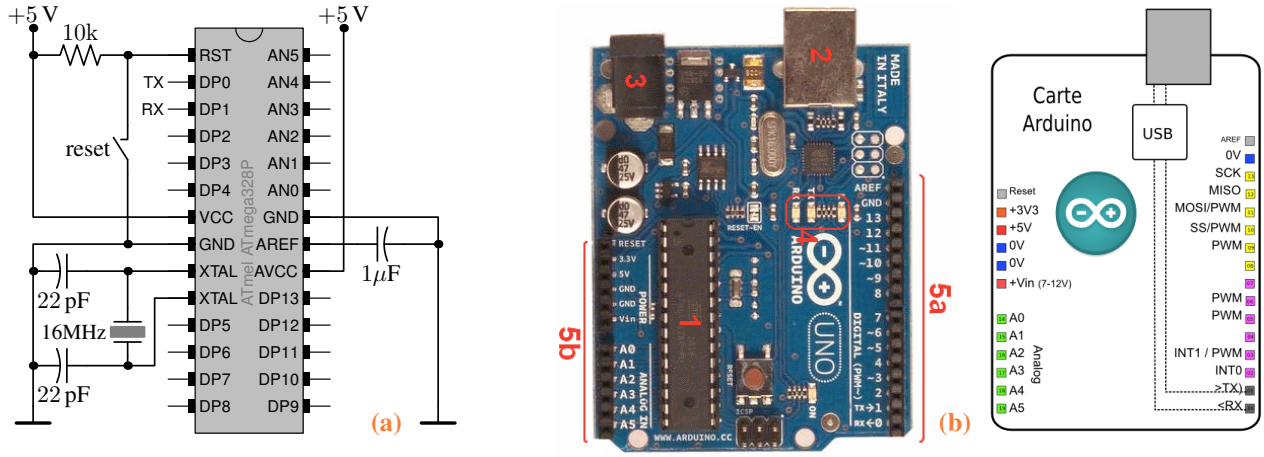


FIG. 1 – Brochage principal du microcontrôleur ATMEGA328P-PU (a) et carte ARDUINO UNO (b) où on distingue (1) microcontrôleur ATMEGA328, (2) connecteur USB pour relier l'ordinateur, (3) jack de connection d'une alimentation extérieure (≤12V DC), (4) diodes de fonctionnement de la carte dont TX et RX pour visualiser le passage de données dans le port série, et (5) connecteurs d'entrée/sortie pour «le monde extérieur», avec (5a) digitales $D0$–$D13$ et masse GND fonctionnant en niveaux TTL 5V et (5b) analogiques $A0$–$A5$ accompagnés par les broches de sortie d'alimentation (3.3 V stabilisée, 5 V, et la masse GND) dont la VIN permet également entrer une alimentation extérieure de 5 V DC à la carte.

* sadovski@univ-littoral.fr, responsable du projet BQE HTLC (Hight Tech Low Cost) ULCO 2017

High performance, low power AVR® 8-bit microcontroller

- Advanced RISC architecture
  - 131 powerful instructions, most single clock cycle execution
  - 32×8 general purpose working registers
  - fully static operation
  - Up to 20 MIPS Throughput at 20 MHz
  - On-chip 2-cycle Multiplier
- High endurance non-volatile memory segments
  - 4/8/16/32K Bytes of in-system self-programmable flash program memory
  - 256/512/512/1K bytes EEPROM
  - 512/1K/1K/2K bytes internal SRAM
  - Write/Erase cycles : 10,000 Flash/100,000 EEPROM
  - Data retention : 20 years at 85˚C/100 years at 25˚C (1)
  - Optional Boot Code Section with Independent Lock Bits
  - In-System Programming by On-chip Boot Program
  - True Read-While-Write Operation
  - Programming Lock for Software Security
- Operating Voltage : 1.8 - 5.5V
- Temperature Range : −40˚C to 85˚C
- Speed grade : 0 - 4 MHz@1.8 - 5.5V, 0 - 10 MHz@2.7 - 5.5.V, 0 - 20 MHz @ 4.5 - 5.5V
- Power consumption at 1 MHz, 1.8V, 25˚C
  - Active Mode : 0.2 mA
  - Power-down Mode : 0.1 μA
  - Power-save Mode : 0.75 μA (Including 32 kHz RTC)
- Peripheral Features
  - Two 8-bit timer/counters with separate prescaler and compare mode
  - One 16-bit timer/counter with separate prescaler, compare mode, and capture mode
  - Real Time Counter with separate oscillator
  - Six PWM channels
  - 8-channel 10-bit ADC in TQFP and QFN/MLF package temperature measurement
  - 6-channel 10-bit ADC in PDIP package temperature measurement
  - Programmable serial USART
  - Master/Slave SPI serial interface
  - Byte-oriented 2-wire serial interface (PHILIPS I2C compatible)
  - Programmable watchdog timer with separate on-chip oscillator
  - On-chip analog comparator
  - Interrupt and wake-up on pin change
- Special microcontroller features
  - Power-on Reset and Programmable Brown-out Detection
  - Internal Calibrated Oscillator
  - External and Internal Interrupt Sources
  - Six Sleep Modes : Idle, ADC Noise Reduction, Power-save, Power-down, Standby, and Extended Standby
- I/O and packages
  - 23 Programmable I/O Lines
  - 28-pin PDIP, 32-lead TQFP, 28-pad QFN/MLF and 32-pad QFN/MLF

TAB. 1 – Caractéristiques générales du microcontôleur ATMEGA328P

2 Initiation à ARDUINO

Nous utiliserons le microcontôleur[1] (μCU) ATMEGA328 dont le brochage est montré dans la figure 1a. Ses caractéristiques générales se trouve sur le site du son fabricant ATMEL. Rassurez vous cependant, que vous n'aurez à utiliser le μCU seul que dans les phases finales d'intégration de vos développements, lors des projets libres, si vous en avez le temps. Vous utiliserez la plupart du temps le μCU dans son environnement de développement ARDUINO. La documentation complète et les explications des différentes broches et entrées sorties sont données sur le site de ATMEL.

D'ores et déjà vous pouvez noter (table 1) l'existence d'un port de communication série (RXD et TXD, broches 2 et 3), de convertisseurs analogiques-digitaux (broches ADCi), d'un bus I2C (SDA et SDC) très utilisé avec les capteurs, d'une interface série synchrone dénommée SPI (MISO, MOSI et SCK) et de broches dénommées digitales à collecteur ouvert. Enfin vous disposez d'une mémoire flash d'une capacité de 32 kO pour stocker votre programme de manière permanente mais, bien évidemment, effaçable. Notez que les broches digitales sur la carte ARDUINO, (fig. 1, centre, 5b) sont des niveaux TTL 5 V, ce qui signifie, que le niveau logique 1 (HIGH) correspond à une tension comprise entre 2.4 et 5 V, et que le niveau logique bas (LOW) correspond à une tension comprise entre 0 et 1.4 V. Chaque sortie numérique est limitée en courant à 20 mA.

2.1 Programmation de ARDUINO

Talk is cheap. Show me the code.
Linus Torvalds

Le MCU ARDUINO est programmé en C++ dans un environnement spécialement dédié (sec. 2.1.1). Nous allons couvrir les bases de cette programmation et de l'utilisation de cet environnement en développant un petit code qui nous permettra de façon évolutive :

[1] on utilisera certaines formes abréviative anglaises devenues très courantes, ainsi le microcontôleur est le μCU = *micro controller unit*

1. brancher et reconnaître votre carte, faire clignoter la LED propre à Arduino (Uno)
2. modifier le code pour faire clignoter le signal SOS avec cette LED
3. allumer les 4 LED's externes une par une en séquence dans le `setup`
4. passer des messages via port série/usb (aka «Terminal» ou «Console»)
5. communiquer avec l'Arduino via son Terminal par les touches à 1 caractère
6. interpréter les touches, retourner leur code ASCII
7. pour les touches 0..9, a..f, ou A..F les interpréter comme des chiffres hexadécimaux
8. donner la valeur 0..16 du chiffre correspondant
9. donner sa représentation binaire (en 4 bits), utiliser opérations «bitshift» et «bitmask»
10. allumer les 4 LED's externes (voir point 3) selon cette représentation binaire
11. (option) interpréter l'entrée analogique d'un potentiomètre pour allumer les 4 LED's selon la tension obtenue (aka «barographe»)

2.1.1 Installer et utiliser le logiciel ARDUINO (IDE)

Un microcontrôleur ARDUINO est en fait très sommaire avec beaucoup moins de possibilités qu'un ordinateur. Pour s'en convaincre, il suffit de regarder sa quantité de mémoire. Aussi, sa programmation se fait séparément sur un ordinateur (*offline*) et le programme est ensuite transféré dans le MCU pour son exécution. Pour cela, on utilise un environnement de développement intégré, le «desktop IDE» (basé sur l'interface Java). Pour son installation sous Linux, voir `https://www.arduino.cc/en/Guide/Linux`.

```
> tar xvf arduino-1.8.3-linux32.tar.xz
> ln -s arduino-1.8.3/ arduino
> cd arduino
```

Taper la commande `arduino` ouvre la fenêtre de l'IDE (sous Java).

2.1.2 Brancher de la carte ARDUINO à l'ordinateur

Vous disposez de

- un ordinateur sur lequel le programme de développement ARDUINO est installé
- une carte ARDUINO UNO ou équivalent avec son câble USB idoine
- une plaquette d'essais électronique à trous pour les montages électroniques
- des fils dénudés à chaque bout ainsi que les composants électroniques pour les montages envisagés

Lorsque l'on insère le câble USB (type B coté carte, type A coté ordi), on observe en utilisant la commande

```
crw-rw---- 1 root dialout 188,   0 Aug  3 10:51 ttyUSB0
crwx-w---- 1 dima tty       4,   1 Aug  3 12:30 tty1
crw------- 1 root root    189, 513 Aug  3 12:48 usbdev5.2
crw-rw---- 1 root dialout 166,   0 Aug  3 12:48 ttyACM0
```

que les liaisons série–USB sont reconnus par l'ordinateur et les interfaces usb `ttyUSB0` et série `ttyACM0` sont établis. Pour communiquer avec l'ARDIUNO, à partir de l'IDE, nous sélectionnons le port série `ttyACM0` via TOOLS ▷ SERIAL PORT. Ensuite, nous pouvons voir avec TOOLS ▷ GET BOARD INFO que la carte ARDUINO est bien reconnu :

```
BN: Arduino/Genuino Uno
VID: 2A03
PID: 0043
SN: 95536333830351807052
```

Enfin, dans les OUTILS ▷ TYPES DE CARTES on choisit/confirme la carte GENUINO UNO.

Langage de programmation

Il est basé sur `c++`, consulter le site de ARDUINO pour les informations sur ses principales fonctions. A noter la syntaxe `c,c++` de base, tel que la terminaison obligatoire de ligne par `;` et les commentaires. Tout codage ARDUINO présente la même structure :

- les définitions des constantes, des fonctions, et des variables qui seront utilisées dans tout le programme
- la partie `setup(){...}` qui n'est exécutée qu'une seule fois au tout début du programme.
- le programme principal `loop(){...}` qui est exécuté dans une boucle infinie. Repérez sur votre carte le bouton de redémarrage, dit «reboot», qui permet d'interrompre cette boucle et de revenir au `setup(){...}`.

2.1.3 Programme Blink

Nous allons tester le matériel à disposition en faisant l'expérience d'allumage de la diode électroluminescente (LED) de la carte ARDUINO. Parmi les exemples de codage les plus basiques, le programme se trouve dans FILE ▷ EXAMPLES ▷ BASICS ▷ BLINK

```
/*
  Blink: Turns on/off the onboard LED on for 200 msec
*/

void setup() {                      // runs once when you press reset or power on
  pinMode(LED_BUILTIN, OUTPUT);     // initialize digital pin LED_BUILTIN
}

void loop() {                       // runs over and over again forever
  digitalWrite(LED_BUILTIN, HIGH); // turn the LED on (set HIGH voltage level)
  delay(200);                       // wait for a 200 milliseconds
  digitalWrite(LED_BUILTIN, LOW);  // turn the LED off (set the voltage LOW)
  delay(200);
}
```

On charge ce programme FILE ▷ OPEN dans le logiciel IDE. Pour compiler et vérifier ce programme, nous utilisons SKETCH ▷ COMPILE AND VERIFY

```
Sketch uses 928 bytes (2%) of program storage space. Maximum is 32256 bytes.
Global variables use 9 bytes (0%) of dynamic memory, leaving 2039 bytes for local variables. Maximum is 2048
    bytes.
```

et pour le faire tourner on passe par COMPILE-UPLOAD (téléverser le programme dans la carte) ou on utilise le raccourci clavier CTRL-U. La LED de la carte doit clignoter à une fréquence de $1/(2 \times 0,200)$ Hz confirmant le fonctionnement de la carte et du logiciel IDE. Vous êtes alors prêts à utiliser l'environnement ARDUINO.

2.1.4 Programme BlinkSOS

On modifie le programme précédant pour faire clignoter «SOS» en morse. Ce signal de détresse consiste de trois appels courts (lettre S) suivis par trois appels longues (lettre O) puis encore 3 courts. Pour cela, on essaie d'utiliser une boucle `for` :

```
 for(i=3; i==0; i--) {
    digitalWrite(LED_BUILTIN, HIGH);   // turn the LED on (HIGH is the voltage level)
    delay(200);                        // wait for a 200 milliseconds
    digitalWrite(LED_BUILTIN, LOW);    // turn the LED off by making the voltage LOW
    delay(200);
  }
  delay(200);
```

ici `i` est déclarée auparavant comme une variable du type `int` par une commande du type `int i;` placée soit avant le `setup` (déclaration globale), soit au debut de `loop` (locale). L'instruction `for` réalise une boucle et prend trois arguments separés par point-virgule. Ici `i = 3` initialise la valeur de compteur `i` à 3 et la condition `i == 0` permets de continuer la boucle tant que `x` reste positif. La dernière partie est exécutée à la fin de la boucle, apres les instructions groupées par `{ }`. Ici `i--` décrémente la valeur de `i` par pas de 1. En fait, l'instruction `i--` est équivalente à `i = i-1`, mais se fasse plus efficacement en etant une operation de processeur, et prend moins de place en mémoire.

Pour coder «SOS», il nous faudra trois boucles `for` successives. Alternativement, en plus de `loop` et `setup`, nous pouvons définir une petite *fonction* qui effectuera cette boucle avec le delay variable demandé comme son parametre.

```
void blink(unsigned char n, int d, int dd, unsigned char pin=LED_BUILTIN) {
  while(n--) {                  // blink n times
    digitalWrite(pin, HIGH);    // turn the LED on (HIGH is the voltage level)
    delay(d);                   // wait for d ms
    digitalWrite(pin, LOW);     // turn the LED off by making the voltage LOW
    delay(d);                   // wait for d ms
  }
  delay(dd);                    // mark the end of sequence by extra dd msec
}
```

Maintenant, dans le `loop`, notre code se simplifie à

```
  blink(3, 200, 250);           // letter S
  blink(3, 500, 250);           // letter O
  blink(3, 200, 500);           // letter S and final pause
```

2.1.5 Montage d'une LED

Les pins digitaux (DP's), initiés (dans le `setup`) comme `OUTPUT`, peuvent servir à passer et à commuter les tensions de 5 (V_{CC}) et 0 V (GND) signifiant les niveaux logiques de «oui» et «non», respectivement (voir la sec. 2.1.7). Ainsi notre pin `LED_BUILTIN`, étant déjà initié pour cette utilisation, peut commuter une LED[2] externe. On branche celle-ci (sec. 2.1.6) par son anode sur le pin 13 correspondant à `LED_BUILTIN` de ARDUINO UNO, et une résistance de 220 Ω entre son cathode et GND. Le code reste inchangé, mais au présent, la LED externe double les clignotements de la LED sur la carte.

2.1.6 Remarque : Connexion de LED's

Pour ceux qui n'ont pas suivi d'enseignement d'électronique, une LED est une diode qui s'éclaire lorsqu'un courant la parcourt. Elle sert beaucoup pour visualiser de façon très simple l'état logique d'une ligne (haut/bas ou vrai/faux). La diode est un dipôle asymétrique constitué d'une anode et d'une cathode. Lorsque elle est polarisée dans le sens direct (+V sur l'anode et 0 V sur la cathode), elle est passante, et un courant circule. Lorsqu'elle est polarisée en inverse, elle est bloquée, et aucun courant ne circule. La figure 2a donne les caractéristiques courant-tension de LED de diverses couleur. Vous remarquerez que lorsque la tension à leur bornes est inférieure à $\sim 1,5$ V, le courant est nul : la diode est bloquée. Il faut que la tension dépasse une certaine valeur appelée «tension de coude» pour que le courant puisse passer. Typiquement, pour les LED que vous avez en TP, la tension de coude est de 1,8 V. Au delà de cette valeur, une petite variation de la tension induit une forte variation du courant.

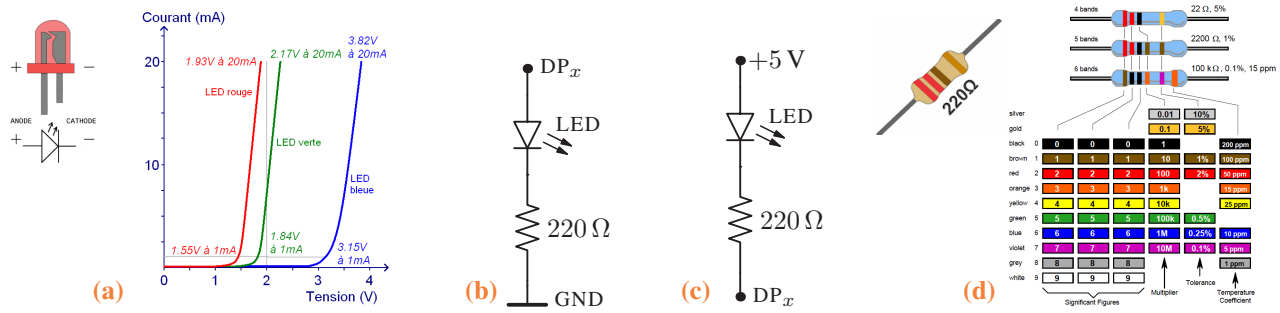


FIG. 2 – Caractéristiques courant-tension de diodes électroluminescentes (LED's) dont la cathode est répérée par la patte la plus courte (a), et leur montage de base de type *pullup* (b) et *pulldown* (c). Les résistances sont marguées par des bandes colorées (d).

Afin de connecter une LED à un pin digital DP$_x$ de ARDUINO ($x = 2..13$, tension de sortie V_{CC} en mode `OUTPUT`), on place une résistance R en série avec la LED pour limiter le courant à 25 mA, cf la sec. 2.2.1. Calculons la valeur de R. La tension d'alimentation V_{CC} est la somme de tensions V_R aux bornes de la résistance et V_{LED} aux bornes de la LED (fig. 2b). Une LED rouge typique doit avoir une différence de potentiel V_{LED} de 1.6-1.8V entre ses pins pour un courant, approximativement, de 25 mA. Donc

$$V_R = V_{CC} - V_{LED} = 5\,\text{V} - 1.8\,\text{V} = 3.2\,\text{V}.$$

Par la loi d'Ohm, nous obtenons pour $I = 25$mA une résistance de $R = V_R/I = 128\,\Omega$. Cependant, les tutoriels recommandent d'utiliser 220 Ω. Pourquoi ? La raison est dans la pratique : en regardant les résistances disponibles couramment, on trouve une de 100 Ω. Inférieure à 128 Ω, que nous avons besoin, celle-là est risquée parceque le courant $I = 3.2/100 = 32$ mA serait alors trop important. La valeur usuelle suivante est de 220 Ω, ce qui nous donne un courant $I = 3.2/220$ d'environs 14 mA. Les DEL's sont des composants non-linéaires, opérationnelles pour des courants de 10–25 mA, et la différence de courant entre 14 et 25 mA n'est pas nécessairement proportionnelle à la différence de luminosité. Dans la plupart des cas, on ne sera même pas capable d'apercevoir cette différence.

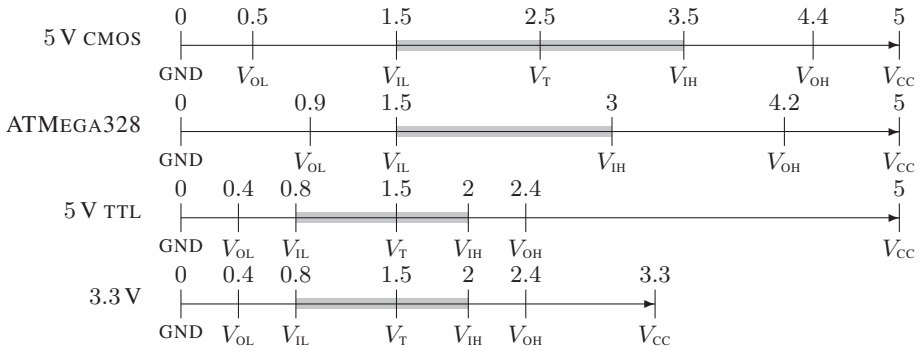


FIG. 3 – Niveaux logiques

[2] L'abréviation omniprésente anglaise LED = *light emitting diode* désigne une diode électroluminescente parfois abrégé DEL en français

2.1.7 Remarque : Niveaux logiques de ARDUINO

En regardant le datasheet de l'MCU ATMEGA328 de la carte ARDUINO UNO, on peut conclure que les niveaux de tension sont légèrement différents de standard CMOS (voir la fig. 3). Ceci fait la plateforme ARDUINO, avec son intervalle des tensions invalides 1.5..3.0 V raccourcie coté V_{IH}, un peu plus robuste, offrant une marge plus large pour la bruit, et possédant une tolérance élargie pour les signaux d'entrée `HIGH`. En conséquent, la conception des interfaces et les connexions aux autres équipements sont simplifiées.

2.1.8 Capteur de lumière comme interrupteur

Les pins digitaux peuvent servir également à lire les signaux logiques à leur entrée après être déclarés (dans le `setup`) en mode `INPUT`. Comme indiquée dans la sec. 2.1.7, les plages de voltage que ARDUINO tolère comme «oui» et «non» (aka `HIGH` et `LOW`) sont assez larges. Nous allons exploiter ces plages pour brancher une photorésistance (LDR), un capteur de lumière décrit dans la sec. 3.2. Considérez le branchement en série d'une résistance ohmique R coté V_{CC} et une LDR coté masse, avec l'entrée DP_2 correspondant au point entre R et LDR de resistance r. La tension d'entrée v sur DP_2 égale à la tension au bornes de la LDR,

$$v = \frac{r}{r+R}\, V_{\text{CC}}.$$

La résistance r de la LDR dépende de l'intensité de la lumière qu'elle reçoit. Elle varie entre $r = 100\,\text{K}$ dans le noir, et 2 à 5 K en plein jour. Par conséquent, si on choisit $R = 10\,\text{K}$, le signal v variera entre 4.5 V et 0.8 à 1.6 V (1.3 V pour $r = 3.5\,\text{K}$). En regardant les niveaux logiques dans la fig. 3, on conclut que dans ce montage, DP_2 donne «oui» dans le noir, et «non» si la LDR reste bien illuminée. Maintenant, nous pouvons réagir aux niveaux de luminosité. Les changements dans notre code sont minimes. On initialise le pin DP_2

```
#define pin_LDR 2                       // photo-resistor pin
void setup() {
  pinMode(LED_BUILTIN, OUTPUT);         // for onboard LED
  pinMode(pin_LDR, INPUT);              // for LDR state
}
```

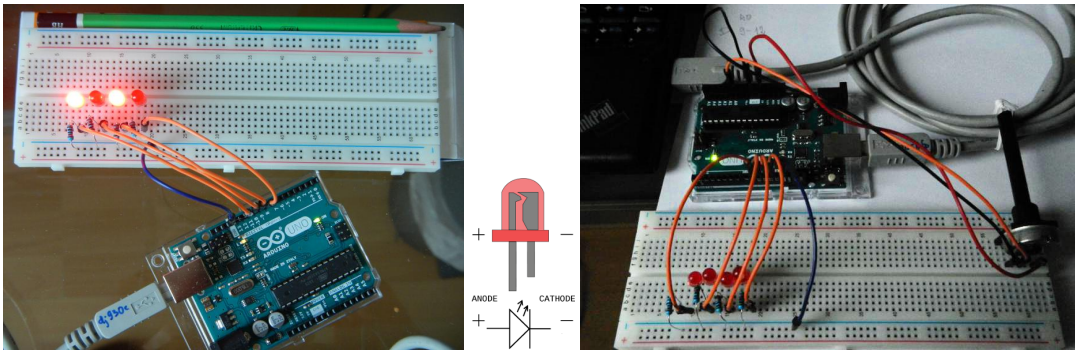
et il suffit de modifier une seule ligne dans notre fonction `blink` dans la sec. 2.1.4

```
    if(digitalRead(pin_LDR))    // if no ambient light is present
      digitalWrite(pin, HIGH); // turn the LED on (HIGH is the voltage level)
```

pour permettre les clignotements «SOS» seulement en absence de la lumière ambiante (ou la LDR reste couverte).

2.2 Montage à 4 LED's

Par la suite, on monte quatres LED comme montré sur la figure ci-dessous (les résistances entre les cathodes de LED's et GND sont de 220 Ω). Dans la partie `setup()`, nous pouvons initialiser les quatre sorties digitales avec une petite boucle :




```
  for(j=PIN_BASE,i=4; i--; pinMode(j++, OUTPUT));
```

où l'utilisation des opérations ++ et −− est à noter. Et par la suite, nous pouvons aussi allumer nos LED's une par une

```
for(j=PIN_BASE,i=4; i--; j++) {
   digitalWrite(j, HIGH);    // turn the LED on (HIGH is the voltage level)
   delay(200);               // wait for a 200 milliseconds
   digitalWrite(j, LOW);     // turn the LED off by making the voltage LOW
   delay(200);
 }
```

pour tester notre montage.

Câblage : dans les montages de prototypage on utilise les cavaliers avec les embouts Dupont M/M ou M/F, ou tout simplement les fils de diamètre 0.51 mm, section $0.2\,\text{mm}^2$, et résistance linéique 0.084 Ω/m, connu aussi comme 1×AWG24.

2.2.1 Remarque : Le courant maximal qu'on peut tirer des pins digitaux de ARDUINO à 5 V

La carte ARDUINO est conçue pour être alimentée de deux manières (sec. 2.3) : par le port USB ou par son connecteur externe. Pour USB, la limite maximale du courant est de 500 mA. Pour une alimentation externe qui utilise le régulateur +5 VDC de la carte, la limite est plutôt fixée par la dissipation de chaleur de ce régulateur et en conséquence, est déterminée par la valeur spécifique de la tension sur le connecteur d'alimentation externe. Une limite de 300 mA sera certainement correcte pour les deux types de sources.

Cependant, une autre limitation est le courant que les broches d'entrée/sortie (E/S) peuvent gérer à la fois individuellement et la consommation totale de courant de toutes les broches d'E/S ensemble. 200 mA est une limite totale et 40 mA est une limite maximale individuelle. Ainsi, 200 mA pour toutes les broches de sortie est la limite à laquelle vous vous heurtez le plus souvent, et il est préférable de faire fonctionner les broches d'E/S en toute sécurité à environ 20 mA maximum.

2.2.2 Interaction sur le port série

Le port série classique est émulé par l'ARDUINO à travers de son interface USB. Pour initier ce port à la vitesse de transfert de 9600 baud (la vitesse typique des anciens terminaux et modems sur le port série RS232), nous ajoutons dans l'initialisation `setup()`

```
Serial.begin(9600);
```

Pour communiquer les informations par ce port, une fois que notre carte a été identifiée (comme `ttyACM0` dans ce document, ou `COM1`, `COM2`, etc sous WINDOWS), il faut ouvrir le terminal de l'environment Arduino IDE. Notez, qu'à l'ouverture du terminal, ainsi qu'après le téléchargement d'un nouvel exécutable, la carte se redémarre en exécutant le `setup`.

Sortir les informations : le texte (string) Nous pouvons sortir un message de démarrage, par exemple

```
Serial.println("\nSOS blinking and Serial interaction via keys");
```

Notez, qu'à la différence de `Serial.print` la commande `Serial.println` y ajoute automatiquement le changement de ligne. Pour ces deux commandes, le texte, dit «string», doit être entouré par les signes double de citation `"`.

Le caractère, le byte, le bit Chaque string consiste en un ou plusieurs caractères, et chaque caractère est représenté par un nombre 0..255 (où les caractères imprimables/lisibles démarrent à 32). Un tel nombre occupe un *byte ou octet*, c. à d., *huit* registres binaires 0/1 dits *bits* ($255 = 2^8 - 1$). Dans le langage `c, c++`, on déclare ces nombres avec le type `char` ou plus précisément `unsigned char`. Il existe plusieurs façons de définir la correspondance (codage) entre les caractères et ces nombres. Actuellement, le plus répandu et simple est le codage ASCII.

NB : les types représentant les nombres entiers A la différence de `unsigned char`, le type `char` définit les nombres entiers dans l'intervalle $-127 \ldots 127$ qui occupent aussi un byte dont les 7 bits inférieures gardent la valeur ($127 = 2^7 - 1$) et le $8^{\text{ième}}$ bit sert à donner le signe. Les nombres entiers, quant à eux, sont définis par la classe `int` ou encore `unsigned int` et occupent *deux* bytes. Par conséquent, ces nombres vont jusqu'à $\pm(2^{15} - 1)$ ou bien $2^{16} - 1$.

Entrée d'information Pour envoyer des informations à la carte, nous les entrons dans la ligne de commande du terminal suivies par la touche «Enter» qui les envoie. Les informations sont transmises comme un texte qui est mis en attente dans le tampon de l'interface série/usb dit «buffer». Pour savoir si les informations nous attendent à l'entrée, on appelle `Serial.available()`

2.2.3 Montage du potentiomètre et le convertisseur A/N (ADC)

On branche les extrémités d'un potentiomètre de $R = 10$ à 100 kΩ entre GND et la sortie 5 V coté analogique ; le connecteur amovible se branche à l'entrée analogique, par exemple $A0$ (voir fig. 1). Le courant consommé de $V/R = 5/10^4 = 5\,\text{mA}$ est inférieur à celui que ARDUINO peut soutenir, voir sec. 2.2.1. La programmation est simple, voir l'exemple «AnalogInput» dans FILE▷EXAMPLES▷ANALOG. La partie essentielle du code est

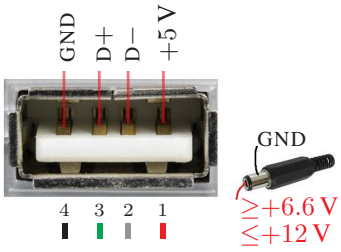
```
#define sensorPin A0
int sensorValue = 0;
sensorValue = analogRead(sensorPin);
```

où nous appelons `analogRead` pour lire la tension présente sur la broche $A0$ (l'entrée analogique). Cette fonction effectue la conversion de la tension analogique en signal numérique (A/N) avec un convertisseur à 10 bits pour la plupart des MCU ARDUINO. Cela signifie que le signal numérisé peut prendre $2^{10} = 1024$ valeurs comprises entre 0 et 1023, 1023 correspondant à 5 V.

On transforme ainsi la tension présente sur l'entrée $A0$ en une valeur lisible et exploitable par le μCU. La conversion consiste à comparer cette tension à une valeur de référence fixée à +5 V par défaut. Il est possible de comparer par rapport à une autre valeur de tension comprise entre 0 et 5 V qui est appliquée à la borne A_{ref} de la carte. Il faut alors le spécifier lors de l'utilisation de la commande `analogReference(reftype)`.

Programme final (avec option potentiomètre) On sauvegarde les modifications dans `BlinkSOS.ino` (voir appendice A).

2.3 Options d'alimentation



La carte ARDUINO[3] possède plusieurs possibilités d'alimentation. Jusqu'au présent, nous avons exploité les $+5\,V_{CC}$ et la masse GND de son cable USB (connecteur de type A coté ordinateur, voir ci à gauche). Cette option est la seule dans le cas où on ne possède pas d'une alimentation DC satabilisée, ni d'une batterie, et on l'utilise souvant au départ d'un prototypage. Cependant, ces $+5\,V_{CC}$ sont fournis par (le *hub* de) l'ordinateur, et par conséquent, (a) ils sont très «pollués» par les différents signaux logiques qui sautent fréquement entre 0 et $+5\,V$; et (b) la puissance (l'ampèrage) maximale de cette source est spécifique à l'ordinateur utilisé et elle n'est pas garantie d'être suffisant. Ces limitations rendent les $+5\,V_{CC}$ du cable USB particulièrement mal adaptés pour les mesures analogiques précises (avec un ADC interne ou externe, voir les sec. 2.5, 3.1, 3.3, ou bien 4.1.3). Dans ce cas, il est fortement recommandé de fournir une alimentation stable *externe* et supérieure à 5 V sur le jack dans la fig. 1-3. Ce jack est «centre-positive» (voir ci à gauche) et est protégé par une diode de la polarité inversée. Il connecte à un régulateur linéaire NCP1117 pour produire la tension V_{CC} stabilisée de $+5\,V$. A son entrée (donc sur le jack) ce type de régulateur requit une tension supérieure d'au moins 1 V.

> **Jack et cable USB :** Dans le cas où le cable USB reste également connecté à l'ordinateur, la carte ARDUINO UNO bascule en alimentation externe fourni par le jack (fig. 1-3) ou le pin VIN si elle y sense une tension supérieure à $+6.6\,V$.

Bien évidemment, la tension externe ne doit pas dépasser une valeur maximale, qui dans notre cas est de 12 V. Enfin, il existe une troisième option de connecter $+5\,V$ *directement* sur le pin 5 V (donc V_{CC}, à ne pas confondre avec VIN !). On rencontre cette option parfois dans les montages embarqués avec une source de $+5\,V$ stabilisées commun. Autrement, elle est à éviter comme dangereuse.

2.4 Traiter les interruptions et programmer les boutons

Cet exemple simple nous apprend l'idée très importante, celle d'interruptions externes et leur traitement par une `ISR` (= interrupt service routine). On apprend aussi le branchement et l'utilisation d'une bouton entre +5V et, à travers d'une résistance de $R = 10\,k\Omega$, la masse. La broche 2 est branchée à la bouton, avant la résistance, pour détecter la montée de la tension (dit `RISING`). Une LED branchée sur pin 10 (et la résistance de $220\,\Omega$ comme nous avons déjà appris dans la sec. 2.2) sert pour la visualisation. En option, un condensateur de $C = 10\,nF$ sert à couper les perturbations aux périodes $T \geq 2\pi RC \approx 0.63$ msec et fréquences supérieures à $T^{-1} \approx 1.6kHz$. On se rappelle d'une alternative à une simple bouton dans la sec. 2.1.8, où on utilise une LDR (voir et 3.2) pour commuter la broche 2. Ce montage est couramment utilisé comme un compteur de passage d'un obstacle, connu comme *photogate*.

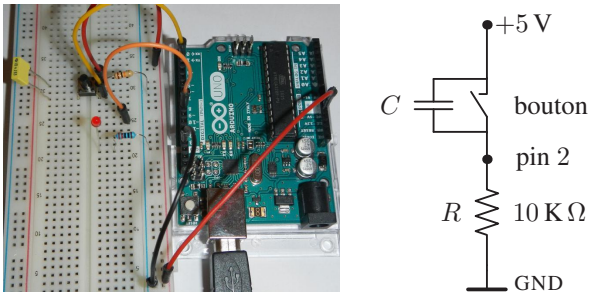


FIG. 4 – Montage d'une bouton de type «pull down switch» pour demander l'interruption sur le pin 2.

```
/*
  Button Switch using an external interrupt. DS 2021-11-29

  This code is in the public domain and may be used without restriction and
  without warranty. The code is largely inspired by tutorials of Ron Bentley

  https://create.arduino.cc/projecthub/ronbentley1/button-switch-using-an-external-interrupt-7879df
  https://create.arduino.cc/projecthub/ronbentley1/understanding-and-using-button-switches-2ffe6c

  the button status is flagged as 'switched' AFTER the button is pressed
  AND then released, AND a debounce period has elapsed

  The button is wired in a standard configuration with an external 10K pull
  down resistor, which ensures that the digital interrupt pin is kept LOW until
  the button is pressed and raises it to HIGH (+5V).

  Operation of the button is demonstrated by toggling a LED on and off.
*/

// ATTN. on Arduino Uno and other 328-based boards, digital pins 2 and 3
```

[3] Spécifiquement, nous considérons ici la UNO Rev.3. Pour les autres cartes, consultez leur datasheet.

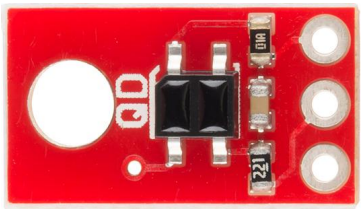
```
// are the only ones that are usable for programmable external interrupts
#define BUTTON_PIN  2    // external interrupt pin trigerred by the button
#define LED_PIN     9    // digital pin connected to the LED indicator
#define LED_STATE   LOW  // initial LED indicator state (code testing only)

#define INT_TRIGGER_TYPE     RISING // interrupt triggered on a RISING input
#define DEBOUNCE             30     // time in msec to wait

// declare volatile to assure safe ISR access at any time
// 0 at first, counts subsequent incoming interrupts after
volatile unsigned int int_proc_stat = 0;
// begin with no switch press pending, i.e. false (not triggered)
volatile bool     switching_pending = false;
volatile long int elapse_timer = 0; // store the time of the interrupt

bool init_complete =         false; // inhibit the ISR during setup

// ISR for handling interrupt triggers arising from associated button switch
void button_int_handler() {
 if (init_complete && ((int_proc_stat+=digitalRead(BUTTON_PIN)) == 1)) {
     digitalWrite(LED_BUILTIN,HIGH);// the onboard LED on signals "busy"
     switching_pending = true;      // begin handling new button press
     elapse_timer = millis();       // reset DEBOUNCE elapse timer
   }
}

int button_state() {
 int k;
 if ((k=int_proc_stat)
     // interrupt flag k>0 has been raised on this button and requires handling
     && switching_pending && !digitalRead(BUTTON_PIN)
     && millis() - elapse_timer >= DEBOUNCE) {
   // switch was pressed, now released, and DEBOUNCE time has elapsed
   Serial.print(k);
   Serial.println(" ints");
   digitalWrite(LED_BUILTIN, LOW);  // the onboard LED off signals "done"
   switching_pending = false;       // clear for new interrupt service cycle
   int_proc_stat = 0;               // reset interrupt counter
   elapse_timer = 0;
   return k;  // k>0 signals that switch has been pressed "k times" and released
 }
 return 0;    // either no press has occured or DEBOUNCE period has not elapsed
}

void setup() {
 int i;
 pinMode(LED_BUILTIN, OUTPUT);
 digitalWrite(LED_BUILTIN, HIGH);   // turn the onboard LED on = "busy"
 Serial.begin(115200);              // 115200 bps = 14400 bytes/sec, 70 usec/byte
 pinMode(LED_PIN, OUTPUT);          // initialize switch indicator,
 digitalWrite(LED_PIN, LED_STATE);  // and trigger it four times
 for(i=8; i--; delay(500)) digitalWrite(LED_PIN, !digitalRead(LED_PIN));
 while (!Serial) {
   ; /* wait for serial port to connect (native USB port only) */
 }
 pinMode(BUTTON_PIN, INPUT);        // declare and set interrupt pin
 attachInterrupt(digitalPinToInterrupt(BUTTON_PIN), // pins 2,3 give 0,1 on UNO
     button_int_handler, INT_TRIGGER_TYPE);
 Serial.println("# button interrupt handler");
 init_complete = true;              // open interrupt processing for business
 digitalWrite(LED_BUILTIN, LOW);    // turn the onboard LED off = "done"
} // end setup

void loop() {
 interrupts(); /* re-enable interrupts (if/after disabled by noInterrupts() */
  if (button_state()) {             // test if the button is pressed
    digitalWrite(LED_PIN, !digitalRead(LED_PIN)); // toggle LED state
  } else {                          // do other things....
    if(digitalRead(BUTTON_PIN)) Serial.println("# button ON");
    delay(200);
  }
}
```

2.4.1 Réduire la consommation inutile : ARDUINO SLEEP MODE

Par rapport à notre code, on observe pour, que la plupart de temps le MCU ARDUINO reste en attente des appuis sur le bouton et donc tourne «à vide» dans son `loop()`, tout en consommant approximativement 30 mA ! Pour éviter ceci, il souffit de le faire «endromir», c.à.d de passer en mode SLEEP, dans lequel le MCU est à l'arrêt, il ne suit plus son programme, et ne peut être remis en marche que par un signal extérieur. Pour activer ce mode, apportez des modifications suivants à notre sketch.

```
#include <avr/sleep.h> // AVR library for sleep modes
. . . . . .
void button_int_handler() {
   . . . . . .
   sleep_disable();     // disable sleep mode
}
void setup() {
  . . . . . .
  set_sleep_mode(SLEEP_MODE_PWR_DOWN); // full sleep mode
} // end setup
. . . . . .
void loop() {
 interrupts(); /* re-enable interrupts (if/after disabled by noInterrupts() */
 sleep_enable();   // enable sleep mode
  if (button_state()) {             // test if the button is pressed
    digitalWrite(LED_PIN, !digitalRead(LED_PIN)); // toggle LED state
  } else {                          // do other things....
    if(digitalRead(BUTTON_PIN)) Serial.println("# button ON");
    delay(500);     // wait 500 msec before going to sleep
    digitalWrite(LED_BUILTIN,LOW);  // turn onboard LED off
    delay(200);     // wait 200 msec to allow the led to turn off
    sleep_cpu();    // activate full sleep mode
    Serial.println("Woke up!");     // first command after interrupt
    digitalWrite(LED_BUILTIN,HIGH); // turning onboard LED on
  }
}
```

2.4.2 Detecteurs d'obstacles et suiveurs de ligne



Comme une situation très courante (sec. 4.11.3), où les interruptions externes sont indispensables, considérez la détection des objets avec une *photocellule* (l'obstacle opaque passe entre l'émetteur et le récepteur de la lumière visible ou infrorouge), ou un «radar» (l'obstacle réfléchissant passe devant un émetteur-récepteur). Ainsi un *suiveur de ligne* QRE1113 est un variant d'un radar infrarouge (IR) à courte distance, typiquement de 3−5 mm. Les couleurs claires réfléchissent d'avantage. Ceci permets de détecter une ligne blanche sur le fond noir et vice versus. Le QRE1113 est composé d'un DEL-IR (émetteur) et un photo-transisteur (récepteur). On le trouve prémonté sur les petites cartes («breakout boards») ROB-09453 (analogique) et ROB-09454 (digital, ci à gauche) de SPARKFUN[4]. La première carte possède une résistance de 10 K en série à l'entrée du transisteur constituant un diviseur de tension avec la sortie V_{OUT} : sans réflexion (couleur noir), le transisteur est fermé et V_{OUT} reste HIGH. Ainsi le QRE1113 peut y être employé en continu comme un bouton du style «pull-up» aka «low-end». La deuxième carte possède un condensateur de 10 nF en série avec le transisteur et une résistance de 200 Ω protégeant la V_{OUT}. Nous pouvons reconvertir cette carte en détecteur analogique (continu) en ajoutant une resistance de 10 K (ou[4] 47 K) entre la V_{CC} et la V_{OUT}. Concrètement, sur un ARDUINO UNO avec $V_{CC} = 5$ V, une telle carte modifiée donne 0.5 à 0.7 V pour une réflexion par une plaque métallique coloré blanche située à une distance de ≈ 4 mm, et 4.9 V sans aucune réflexion (le détecteur pointe dans le vide). Ces valeurs suffisent largement pour déclencher les évènements HIGH/LOW logiques sur les pins digitales[5].

Ayant connecté la sortie VOUT de ROB-09453 ou de ROB-09454 modifiée au pin DP2 de ARDUINO, et afin de compter le nombre des instances de détection `button_count` et connaître le temps depuis la première instance, on utilise le service d'interruption suivant.

```
#define BUTTON_PIN  2       // external interrupt trigger pin: 2 or 3 on UNO (interrupts 0 or 1)
#include <time.h>           // modified C header file for avr-libc and AVR-GCC
volatile time_t        button_start=0, button_timer=0; // time in ms of the first and last trigger events
volatile unsigned int  button_count=0;                 // number of button trigger instances
#define BUTTON_INT digitalPinToInterrupt(BUTTON_PIN)
#define BUTTON_LED LED_BUILTIN
void button_int_handler() { // ISR for handling interrupt triggers arising from associated button switch
 button_timer = millis();   // time of the event and number of events
 if(!button_count++) button_start=button_timer; // reset time of first event
#ifdef BUTTON_LED           // toggle LED indicator
 digitalWrite(BUTTON_LED, !digitalRead(BUTTON_LED));
#endif
}
```

[4] Appelés souvent *reflectance sensors* ou *encoders* en anglais, ces détecteurs sont fournis également par Pololu comme QTR-1A ou QTR-1RC, voir la réf. 0J13.
[5] Les tolérances de cartes ARDUINO de $V_{CC} = 5$ et 3.3 V pour les niveaux logiques HIGH/LOW sont élargies par rapport aux standards TTL et CMOS.

Se service[6] doit être declaré (=installé) dans le `setup()` avec `attachInterrupt` :

```
void setup() {
 pinMode(LED_BUILTIN, OUTPUT);
 digitalWrite(LED_BUILTIN, HIGH);    // turn the onboard LED on = "busy"
#ifdef BUTTON_LED
#if BUTTON_LED != LED_BUILTIN
 pinMode(BUTTON_LED, OUTPUT);        // special LED indicator of the trigger
#endif
 digitalWrite(BUTTON_LED, LOW);      // flush LED indicator of the trigger
#endif
 pinMode(BUTTON_PIN, INPUT);         // pulldown push button or switch
 attachInterrupt(BUTTON_INT, button_int_handler, FALLING); // the mode can be LOW, CHANGE, RISING, FALLING

 . . . .

 digitalWrite(LED_BUILTIN, LOW);    // turn the onboard LED off = "done"
}
```

Ici le mode FALLING est choisi pour marquer exclusivement l'instance de passage HIGH → LOW du signal V_{OUT} de capteur QRE1113.

> Ainsi, dans le cas de pendule de *Pohl* (sec. 4.11.3), LOW correspond à sa position de l'équilibre (où une petite flèche blanche réfléchit la lumière du capteur). L'interruption aura lieu seulement à l'entrée dans cette position ; le départ ne la provoquera pas. Cela permet de lancer une acquisition «en attente» d'un premier passage à l'équilibre, décaler le pendule à la main pour attribuer les conditions initiales (amplitude, phase) au système, et lâcher. L'enregistrement commencera au premier arrivée à l'équilibre avec une phase initiale $\varphi_0 = 0$ ou π selon la direction de lancement.

Maintenant, le temps de la première détection sera sauvegardé dans `button_start`, et le numéro consécutif et le temps de la détection la plus récente se trouveront respectivement dans `button_count` et `button_timer` à tout moments au cours de l'execution de `loop()`. Ceci permet, par exemple, estimer l'intervalle entre instances périodiques :

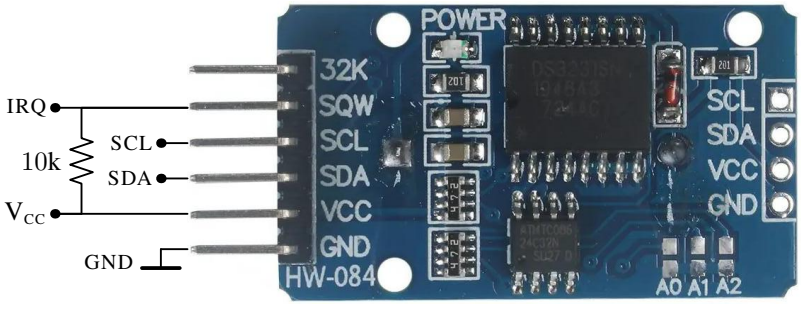
```
 if(button_count>1) Serial.println( floor((float)(button_timer-button_start)/(button_count-1)), 0);
```

Notez, que pour réinitialiser le comptage, on remet `button_count` à 0. On peut également accéder à l'état momentané (HIGH ou LOW) du capteur avec `digitalRead(BUTTON_PIN)`. Par ailleurs, on peut choisir suspendre l'activité et attendre l'instance de détection :

```
#include <avr/sleep.h>      // AVR library for sleep modes
#include <avr/power.h>
void loop() {

 . . . .

    Serial.flush();                       // clear all output on serial line
    button_count=0;                       // reset trigger count
    set_sleep_mode(SLEEP_MODE_PWR_DOWN);  // select full sleep mode
    sleep_enable();                       // (re)enable sleep mode
    sleep_cpu();                          // ***** ACTIVATE SLEEP *****
    sleep_disable();                      // on comeback: disable the sleep mode

 . . . .

}
```

2.4.3 Utilisation d'une horloge RTC pour reveiller ARDUINO

Le mode SLEEP est évidemment très utile dans des nombreuses situations courantes, où le MCU agit aux moments de temps precis et ne travaille que quelques centaines de msec (par ex., pour prendre des mesures) par minute (heure, jour, mois, année, etc).



Pour des intervalles de temps assez courts, inférieurs ≈ 4 sec, nous pouvons nous servir de l'un de compteurs de temps de 8 bit (`timer0` et `2`) et 16 bit (`1`) internes standards que possède l'IC ATMEGA328P, et qui sont pilotés par l'horloge système de 16 Mhz de la carte ARDUINO UNO (en version 5V, et 8 MHz en version 3.3V) ou un horloge externe. Par ailleurs, en utilisant le WDT (watchdog timer) de ATMEGA328P nous pouvons aller jusqu'à ≈ 8 sec, tout en minimisant la consommation. Ayant un oscillateur interne dédié de 128 kHz pour sa source, le WDT est le seul capable de fonctionner en mode de sommeil profond `SLEEP_MODE_PWR_DOWN`, et sa cadence minimale est de 60 Hz (16 msec),

[6] La procédure de service d'interruption ou ISR (*interrupt service routine*) est engagé à l'instance d'interruption et bloque tout autre activité du μCU. Par conséquent, une ISR doit être très rapide afin de ne pas empêcher le fonctionnement normal interrompu.

Les intervalles de temps plus longues demandent un «montre» que le MCU ARDUINO ne possède pas d'office. Il faudra ajouter au MCU un module externe RTC (= Real Time Clock) capable de passer le signal TTL d'un reveil programmable. Ainci nous allons utiliser une mini-carte (au dessus à gauche) basée sur le IC chip DS3231 (2 alarmes, capteur temperature interne), puce mémoire AT24C32 (32K), et l'interface de communication I2C (adresse `0x68`). Ces pins SCL et SDA sont connectés[7] aux pins horloge ($A5$) et données ($A4$) I2C de ARDUINO, voir la fig. 1b, tout en gardant les fils de liaison I2C au plus court. Le pin SQW de DS3231 communique l'alarme via le pin D3 (IRQ1) de ARDUINO par une interruption externe[8]. Ce pin est du type «open drain» et en cas d'alarme, il est mis à la masse (LOW). Par conséquent, cela demande un montage «pull-up», avec V_{CC} passé à SQW via une résistance[9] de 10 kΩ. On ajoute les et definit/initialise la structure `dt` pour garder le temps courant bibliothèques

```
#include <Wire.h>   // I2C
#include <DS3231.h> // RTC
DS3231 myRTC;                                // RTC data object
DateTime dt = DateTime(2022,10,21,12,30,0); // set default date
```

L'interruption d'alarme RTC passé sur `RTC_PIN` est servie de façon très simple : on signale l'alarme par un «flag» et réveille le μCU.

```
// declare volatile for safe ISR r/w access at any time
// RTC alarm flag: bits 0 and 1 signal DS3231 alarms 1 and 2
volatile unsigned char alarm_flag = 0;
// an ISR cannot talk via I2C (to the RTC) because interrupts are disabled!
// so, in particular, we cannot check RTC alarm flags with checkIfAlarm
void rtc_int_handler() {
 alarm_flag |= 0b1000;              // signals a new alarm interrupt (bit 3)
}
// bits 3,2 and 1,0 are set when alarms 2,1 are enabled and call an interrupt
// NB: the interrupt flags are automatically cleared, see checkIfAlarm
// WARNING: RTC interrogations use I2C and interrupts must be enabled
unsigned char rtc_alarm_state() {
 unsigned char i,k=0;
 for(i=2; i; k<<=1) k |= myRTC.checkAlarmEnabled(i--); // enabled bits
 for(i=2; i; k<<=1) k |= myRTC.checkIfAlarm(i--);      // interrupt flag
 return (k>>1);
}
```

On initialise le module d'horloge dans le `setup` et demarre ces deux alarmes (une chaque seconde, l'autre chaque minute à 0 secondes)

```
#define BUTTON_PIN  2    // external interrupt pin trigerred by the button
#define RTC_PIN     3    // external interrupt pin trigerred by the RTC alarm
#define RTC_INT    digitalPinToInterrupt(RTC_PIN)
void setup() {
  int i;
  . . .
  Wire.begin();  // initialize I2C
  myRTC.setClockMode(false);                 // 24h clock mode
  myRTC.setEpoch( dt.secondstime(),false);   // default RTC date
  for(i=2; i; myRTC.turnOffAlarm(i--));      // disable all RTC alarms
  myRTC.enable32kHz(false);                  // just in case it is in awrong state
  myRTC.enableOscillator(false,false,0);     // alarm interrupts on the INT/SQW pin
  myRTC.setA1Time(1,10,30,0, 0b1111111, true,false,false); // every second
  myRTC.setA2Time(1,10,30,   0b1111111, true,false,false); // every minute
  // INPUT_PULLUP involvs an internal resistor of approx 50k, which is too large
  // use INPUT and an extenal 10k pull-up resistor between 5V VCC and SQW
  pinMode(RTC_PIN,INPUT);                    // receive alarm(s) from RTC pin SQW
  attachInterrupt(RTC_INT, rtc_int_handler, FALLING);
  rtc_alarm_state();                         // clear stale alarm flags, just in case
  for(i=2; i; myRTC.turnOnAlarm(i--));       // enable both RTC alarms
  . . .
}
```

Le reste est fait dans le `loop` parceque en mode «deep sleep» le bus I2C ne fonctionne pas et nous ne pouvons pas interroger le RTC. Ici, pour un teste, on imprime les secondes chaque seconde et le temps/date complets et la temperature (de RTC) chaque minute.

```
// actions that should be taken in response to the raised RTC alarm flag
// because the MCU should never go to sleep with stale RTC alarm interrupts
// this should either be executed repeatedly and/or right before going to sleep
unsigned char alarm_action() {
 unsigned char t;
 if (t=alarm_flag) {                // find and store which RTC alarm was called
   // the value HIGH=1 of the DS3231 pullup interrupt pin means no interrupt
   alarm_flag |= (digitalRead(RTC_PIN)^1)<<2;
   alarm_flag |= rtc_alarm_state() & 3; // autoclear DS3231 interrupt flags
```

[7] Les cartes UNO récentes (Rev. 3) possèdent deux entrées dédiées à cote de pin AREF, qui sont connectées à $A5$ et $A4$ et sont référencées ainsi dans le code.
[8] ATMEGA328P possède *deux* pins D2 et D3 programmables pour les interruptions externes IRQ0 et IRQ1, et dans la sec. 2.4, nous réservons le D2 pour le bouton.
[9] Ainsi limitant le courant à 0.5 mA. Rappelons (sec. 2.2.1) que chaque I/O pin D_i de ARDUINO peut source/sink un maximum de 40mA (donc on vise $\leq$20mA).

```
 }
 dt = RTClib::now();                  // update current time
 if (alarm_flag&1) Serial.println(dt.second());
 if (alarm_flag&2) {
   Serial.print(dt.year());    Serial.print("-");
   Serial.print(dt.month());   Serial.print("-");
   Serial.print(dt.day());     Serial.print(F(" "));
   Serial.print(dt.hour());    Serial.print(":");   Serial.print(dt.minute());
   Serial.print(F(" tRTC="));
   Serial.print(myRTC.getTemperature()); Serial.println(F("C "));
 }
 alarm_flag = 0;                      // clear all alarm flags
 return t;                            // return the flag acted upon
}

void loop() {
 interrupts();     // re-enable interrupts (if/after disabled by noInterrupts()
 set_sleep_mode(SLEEP_MODE_PWR_DOWN);   // full sleep mode
#if defined(DEBUG) && DEBUG > 2
 Serial.print(F("--loop-- "));
#endif
 sleep_enable();                      // (re)enable sleep mode
 digitalWrite(LED_BUILTIN,LOW);       // turn onboard LED off
 delay(10);                           // wait to allow the led to turn off
 rtc_alarm_state();                   // just in case alarms went off unattended
 sleep_cpu();                         // ***** activate sleep *****
 sleep_disable();                     // on comeback: disable the sleep mode
 power_all_enable();                  // re-enable the peripherals, do we need it?
 digitalWrite(LED_BUILTIN,HIGH);      // turn onboard LED on to signal "busy"
#if defined(DEBUG) && DEBUG > 2
 Serial.print(F("Woke up! "));
#endif
 alarm_action();
}
```

2.5 Échantillonnage rapide avec ARDUINO ADC

L'échantillonnage en temps réel avec des intervalles de temps équidistants est une tache très courante dans de nombreuses applications[10]. Le temps de mesure d'un échantillon par l'ADC de microcontrôleur ARDUINO UNO étant de 100 μsec, le taux maximal théorique peut atteindre 10 kHz. Cependant, le MCU possède peu de mémoire (2K) pour sauvegarder les données acquises. L'alternative est de transférer les mesures immédiatement par son port série. Dans ce cas, on doit compter le temps de transfert. A la vitesse de 115200 baud, ce temps est estimé à 70 μsec/byte. Ainsi, si on utilise le mode texte (ASCII), un nombre inférieur à 1024 est représenté par quatre caractères, chacun étant codé sur un byte. Il nous faudra donc *quatre* bytes maximum en format décimal, ou bien *trois* bytes en format hexadécimal pour les valeurs inférieures à 1024[11]. A ceci on ajoute le caractère de terminaison de ligne (LF sous Linux). Donc en tout, nous allons avoir besoin de quatre bytes minimum, et le temps pour un échantillon s'élève à $100 + 4 \cdot 70 \approx 400$ μsec. En conclusion, dans cette méthode, il nous sera possible d'échantillonner aux fréquences inférieures ou égales à 2.5 kHz. Pour le cas de transfert binaire, avec seulement *deux* bytes envoyés par échantillon, ce taux remonte à 4 kHz.

Nous allons échantillonner un signal AC de 0..5V et de fréquence 100 Hz avec l'ADC. On peut utiliser un GBF comme source, mais à l'occasion ici, pour prendre un exemple, nous avons pris un ancien chargeur des piles NiCd et un potentiomètre variable, voir la figure 5a, de 10 kΩ pour régler la tension de sortie V_{out} à 0..5 V. Le chargeur utilise un transfo et deux diodes (two diode rectifier), les bornes A0 (entrée analogique 0) et GND de ARDUINO sont branchées sur le potentiomètre. Une fois téléchargé dans le MCU, le

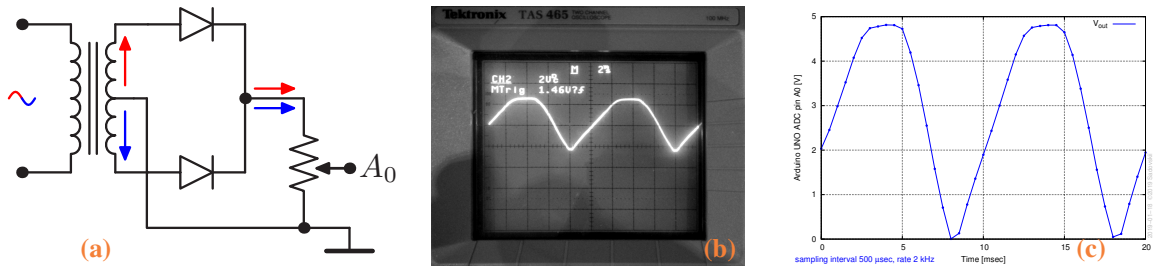


FIG. 5 – Montage pour tester l'échantillonneur rapide en teps réel (a), la tension passée sur le pin A0 (b), les données (c).

programme `~/Arduino/adcsampler/adcsampler.ino` (voir l'appendice B) permet de communiquer avec lui[12] en utilisant les

[10] comme une application possible, voir TP «réaction oscillant»
[11] il s'agit ici du nombre maximal de bytes nécessaires, pour les petites valeurs ce nombre décroît
[12] en utilisant le moniteur série de IDE, la vitesse du port série est réglée à 115200 baud

clés T, N, H, et S, pour définir l'intervalle de l'échantillonnage (`READ_PERIOD`), le nombre des échantillons (`npnt`), ainsi que le format (décimal ou hex), puis démarrer les mesures. Par exemple, avec `T500N100HS` on définit l'intervalle de 500μsec, 100 échantillons et format hex. La figure 5c montre le résultat sous forme graphique.

2.6 Simple DAC

Le μCU ATMEGA328 de la carte ARDUINO ne possède pas d'un convertisseur digital vers analogique (dit DAC[13]) permettant de sortir de façon programmable des différentes fractions de son V_{CC} de 5 ou 3.3 V. Cependant, avec peu de composantes, nous pouvons exploiter sa sortie PWM pour créer un circuit DAC simple pour les applications à précision et le temps de réaction basses à moyennes. Dans le cas le plus simple, on applique le filtre RC passe bas (22 μF, 3.3 k) au signal PWM et on protège la sotie avec un amplificateur opérationnel[14] en mode «tampon». On reconnaît qu'il s'agit en effet d'un filtre actif passe bas de *Sallen-Key*. Ainsi, une fréquence de coupure de ≈250 Hz (pour la fréquence porteuse de PWM à phase correcte de 490 Hz qu'on obtient avec `analogWrite` sur les pins digitaux $DP_{9,10}$) peut être achevée avec $R_1 = R_2 = 6.2\,\text{k}$ et $C1 = C2 = 100\,\text{nF}$. Par ailleurs, pour diminuer le temps de réponse, on peut utiliser un filtre RLC en mode critique,

mais encore mieux—augmenter la fréquence porteuse de PWM. Ainsi, le ATMEGA328 à 16 MHz (carte ARDUINO UNO) peut sortir sur ses pins $DP_{9,10}$ (timer 1) le signal PWM de résolution 10 bit (0.1% duty) et fréquence max 15.6 (fast) ou 7.8 (phase correct) kHz. Une bonne base pour un DAC assez versatile avec un simple filtre RC.

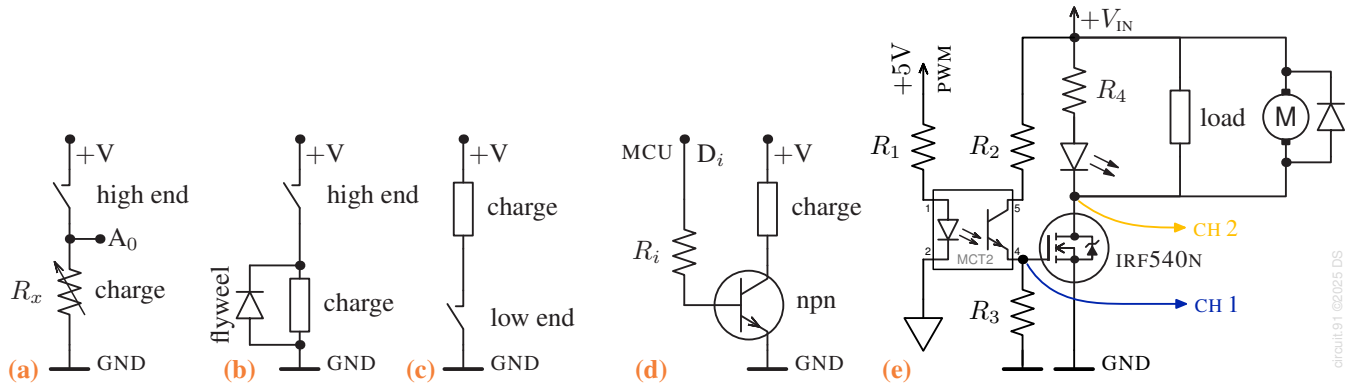


FIG. 6 – Les possibilités différentes de commander l'alimentation d'une charge à forte puissanse (dit «load») : (a) interrupteur côte alimentation (dit «high end power switch») permettant des mesures de tension (pin A_0 de μCU) aux bornes d'une charge ohmique ; (b) le même, mais avec une charge inductive (moteur, bobine d'un relais) et une diode (dit «flywheel» ou «snubbing») protégeant l'interrupteur au moments de déconnexion ; (c) interrupteur déconnectant la masse (dit «low end») ; (d) le cirquit (c) dont l'interrupteur est remplacé par un transistor npn commandé par le pin digital D_i de μCU (0 pour ouvrir et $+V_{DD}$ pour enclencher). Notez, que la tension +V dans (d) peut parvenir d'une source d'alimentation dédiée à forte consommation avec sa masse *connectée* à celle de μCU, et peut être différente (supérieure !) de (à) la $V_{CC} = V_{DD}$ operée par le μCU. Au contraire, le curquit (e) utilise un optocoupleur et un N-mosfet pour commander les appareils «externes» dont V_{IN} et la masse GND sont *isolées* de μCU.

2.7 Commande des appareils à forte puissance

On est parfois obligé de commander des instruments ou des équipements (dit *charge* ou *load*) qui peuvent consommer des intensités incompatibles avec les sorties du μCU. On est amené à *découpler* le circuit d'utilisation (forte intensité) et de commande (faible intensité). Pour cela on utilise classiquement un transistor en commutation, soit un transistor bipolaire BJT[15] ou un MOS-FET[16]. Ce sont des éléments électroniques à 3 broches dénommées comme indiqué ci-dessous.

BJT	Base	– Collecteur (collector) –	Emetteur (emitter)	TIP120
MOSFET	Grille (Gate) –	Drain	– Source	IRF520 (5V), FQP30N06L (3.3V)
Connexion	sortie μCU	– power through load –	masse (ground)	

Pour modéliser la charge (fig. 6), nous prendrons une LED, une résistance ohmique, ou un moteur DC. La masse de la source d'alimentation à commander et la masse de μCU sont connectées.

In the case of a DC motor, we add a 1N400x power diode in parallel with the collector and emitter of the transistor, pointing away from ground. The diode protects the transistor from back voltage generated when the motor shuts off, or if the motor is turned in the reverse direction. Used this way, the diode is called a protection diode or a snubber diode. You can omit the diode if you don't have one, as the transistors recommended here all have a built-in protection diode

[13] DAC = digital to analog converter
[14] mais à ce point, on entre de nouveau dans un «trade-off»—dépenser 40+ centimes pour le IC tampon, ou 70 centimes pour un DAC minuscule à résolution 8-bit ?
[15] BJT : *bipolar junction transistor*, transistor bipolaire.
[16] MOSFET : *metal-oxide-semiconductor field-effect transistor*, transistor à effet de champ à grille isolée.

3 Les capteurs

Le microcontrôleur est réellement adapté pour l'utilisation de capteurs au sens large. Il s'agit bien souvent de mesurer une grandeur électrique (tension ou courant) ou un temps qui sont proportionnels à la grandeur physique que mesure le capteur. Nous vous proposons d'utiliser et de mettre en oeuvre un capteur de température basé sur une thermistance et un capteur de lumière basé sur l'utilisation d'une photo-résistance. Vous mettrez en place la liaison série vers le moniteur série pour visualiser les données de votre capteur.

3.1 La thermistance

On connaît plusieurs méthodes de mesurer la température qui exploitent les phénomènes d'expansion, l'éffet thermoélectrique, et la radiation thermique. Les capteurs à la base de semi-conducteurs ont été développés en 20me siècle. Ils répondent rapidement et précisément aux variations de la température par changements de leur conductivité (éffet thermoélectrique), mais manquent de la linéarité[17]. La thermistance[18] dont vous disposez est une CTN (Coefficient de Température Négative), c'est à dire que sa résistance diminue avec l'augmentation de la température. Il existe également des CTP (Coefficient de Température Positif). La résistance d'une CTN varie en fonction de la température de façon *non-linéaire* suivant la relation de *Steinhart-Hart* définie par un polynôme de degré 3

$$T^{-1} = a + b\,\log(R_T) + c\,\left[\log(R_T)\right]^3$$

avec trois coefficients phénoménologiques (a, b, c). Pour une plage de T restreinte, on peut également utiliser une approximation

$$\log\frac{R_t}{R_x} = B\,\frac{x-t}{(T_0+t)(T_0+x)} = B\left(\frac{1}{T_0+t} - \frac{1}{T_0+x}\right) \text{ où } t \approx x \text{ en °C, } B = B_x = \frac{\alpha_x}{100}\,(T_0+x)^2 \text{ et } T_0 = 273.15\,\text{°K.}$$

Ici t et x représentent la température de référence (typiquement 25 °C) et celle mesurée actuellement. Voici la réalisation C++.

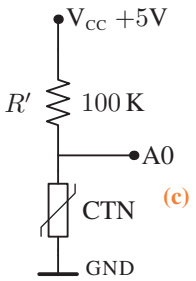
```
#define ADC_MAX   1023
/* 1023 vs 1024, cf this https://skillbank.co.uk/arduino/adc.htm#adc4 */
#define ABS_T0  273.15   // 0C in Kelvin
/* parameters for the NTC thermistor TDK B57164K104J K164 100, 100K at 25C */
#define NTC_t  25        // reference temperature in Celsius (25)
#define NTC_B 4600       // value of parameter B [K] at reference temperature
#define NTC_Rt 100000    // [Ohm] resistance at t = 25
#define NTC_Rref 100000  // [Ohm] fixed resistor on the V divider
#define NTC_r ((float) NTC_Rref/NTC_Rt)
/* get thermistor temperature reading using simplified Steinhart-Hart eqn */
float therm2temp(float r) {
 r/=(ADC_MAX+1)-r;       // resistance Rx relative to Rref
 return 1 / ( log(r * NTC_r)/NTC_B + 1.0/(ABS_T0 + NTC_t) ) - ABS_T0; }
```

La thermistance dont vous disposez (fig. 7a) présente une résistance de 100 kΩ à 25 °C. On la branche dans un «diviseur de tension»

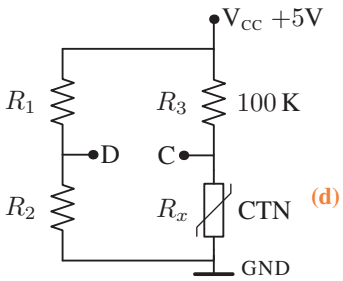
NTC TDK B57164K104J K164 100
$R_{25} = 100\,\text{k}\Omega$
$B_{25/100} = 4600 \pm 3\%$ K **(a)**

x°C	R_x/R_{25}	α_x (%K)
0.0	3.6781	5.6
5.0	2.7944	5.4
10.0	2.1391	5.3
15.0	1.6507	5.1
20.0	1.2823	5.1
25.0	1.0000	5.0
30.0	0.78393	4.8
35.0	0.61822	4.7
40.0	0.49053	4.6

(b)



(c)



(d)

FIG. 7 – La thermistance (a), ses paramètres (b), et son montage dans un diviseur de tension (c) et sur le pont de Wheatstone (d).

comme montré dans la fig. 7c, où on utilise le V_{CC} de la carte, la resistance $R' = R_{25} = 100\,\text{k}\Omega$, et l'entrée analogique A0 du μCU pour mesurer la tension à ses bornes. Notez que ce montage est déjà rencontré dans la sec. 2.2.3.

Diviseur de tension : On étudie des petites variations $\mathrm{d}R$ de la resistance $R = R_0 + \mathrm{d}R$ autour de R_0 en utilisant un diviseur de tension (cf la fig. 7c) avec une resistance R', et en mesurant u_R. Pour déterminer la valeur optimale de R', on considère le branchement en série (loi des mailles), exprime u_R,

$$U_{\text{cc}} = u_R + u_{R'} = I\,(R+R') \Rightarrow u_R = U_{\text{cc}}\,\frac{R}{R+R'} \approx U_{\text{cc}}\,\frac{R_0}{R_0+R'}\left[1 + \frac{R'}{R_0\,(R_0+R')}\,\mathrm{d}R\right],$$

et cherche à maximiser le coefficient devant $\mathrm{d}R$. On trouve que ce coefficient (sensibilité) est au maximum pour $R' = R_0$.

[17] Les circuits intégrés contemporains corrigent la non-linéarité de la réponse de la thérmistance, et sont très faciles et peu coûteux à déployer. Ainsi, le signal de la sortie analogique d'un TMP36 de 0 à 1.5 V couvre les températures de −50 à 100°C avec la précision de ±1°C.

[18] en anglais on dit thermistor NTC ou PTC

Si les variations de R sont larges : dans ce cas, pour déterminer $R = R_0 \pm \Delta R$, on mesure u_R et applique la formule

$$R = u_R\, u_{R'}^{-1}\, R' = u_R\,(U_{CC} - u_R)^{-1} R' = v\,(1023 - v)^{-1} R', \quad \text{où } 0 \le v \le 1023 \text{ est mesurée par ARDUINO.}$$

En remplaçant le potentiomètre de l'ancien montage par la CTN, on exploite le code de la sec. 2.2.3, qu'on modifie pour afficher les valeurs de T. Afin de tester vos mesures, calculez les valeurs de la résistance R_{CTN} attendues (cf. les données dans la fig. 7b référencées par son fabricant) et les valeurs de la tension correspondantes sur A0 pour les températures T entre 25 et 30 °C.

Pont de Wheatstone : Ce montage est employée couramment pour étudier des *petites variations* de résistance d'un capteur, tel qu'un sonde de température, une jauge de déformation, etc. La tension d'entrée $V_{AB} \equiv V_{CC}$ est appliquée à sa «diagonale», voir la fig. 7d. A l'équilibre du pont, les produits en croix des résistances sont égaux : $R_1\, R_x = R_2\, R_3$, et la tension $V_{CD} =: V$ est nulle. Si on déséquilibre légèrement les résistances, on trouve que

$$\frac{V}{V_{CC}} = \frac{k}{(1+k)^2}\left(+\frac{\mathrm{d}R_1}{R_1} - \frac{\mathrm{d}R_2}{R_2} + \frac{\mathrm{d}R_3}{R_3} - \frac{\mathrm{d}R_x}{R_x}\right), \quad \text{avec } k = \frac{R_2}{R_1} \Rightarrow \max_k \frac{k}{(1+k)^2} = \frac{1}{4} \text{ pour } R_1 = R_2.$$

On mesure $\epsilon := V/V_{CC} \ll 1$ à l'aide d'un amplificateur différentiel, ou mieux, d'un amplificateur de mesure (sec. 3.4).

Pour la thermistance sur le pont de Wheatstone (fig. 7d) avec les résistances constants $\mathrm{d}R_{1,2,3} = 0$ de même valeur $R' \approx R_t$, et vu que

$$\log R_x = \log R_t - B_t\,(x-t)\,(T_0+x)(T_0+t)^{-1} \approx \log R_t - B_t\,(x-t)\left[1 - (T_0+t)^{-1}\,(x-t) + \dots\right] \Rightarrow \mathrm{d}\big(\log R_x\big) = -B_t\,\mathrm{d}x,$$

on obtient $4\,\epsilon = -\mathrm{d}\big(\log R_x\big) = B_t\,\mathrm{d}x$. Dans un plus grande intervalle de $x \approx t$, nous pouvons exploiter la formule exacte

$$(T_0+x)^{-1} = (T_0+t)^{-1} + B_t^{-1} \log\frac{R'}{R_t} + B_t^{-1} \log\frac{1-2\,\epsilon}{1+2\,\epsilon}\,. \tag{3.1}$$

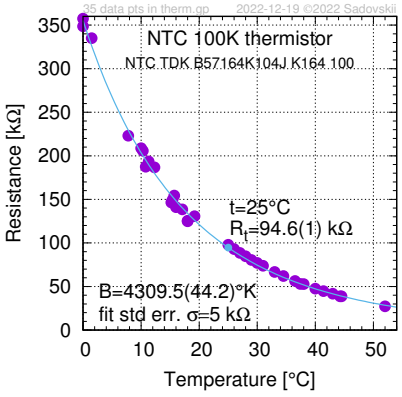


FIG. 8 – Calibration d'une thermistance CTN

Calibration d'une thermistance : Vous utilisez la CTN sur la fig. 7a, trois resistances $R_{1,2,3}$ de 100 kΩ, deux multimètres pour mesurer V et V_{CC}, un thermomètre, un calorimètre muni de deux résistances de chauffage en série et un mélangeur (le matériel de TP mesures calorimétriques), la source de tension DC à deux sorties, dont une simple pour le chauffage et une stabilisée pour le V_{CC} de 5 V, et les glaçons. La thermistance est protégée hermétiquement (vernis, peinture) pour pouvoir être plongée dans l'eau sans provoquer le court-cirquit. Elle est branchée dans le pont de Wheatstone (fig. 7d), un diviseur de tension (7c), ou, tout simplement, au bornes d'un ohm-mètre. Vous chauffez et refroidissez l'eau dans le calorimètre pour prendre les mesures de V dans l'intervalle de température $x = 0 \dots 50$ °C avec votre ARDUINO. Vous pouvez également mesurer les valeurs de R_x directement avec l'ohm-mètre et les tabuler. Par la suite, vous vérifiez l'eq. (3.1) avec les valeurs de B (±138°) et R_t de réference dans la fig. 7a. Si cela est nécessaire, vous ajoustez (re-calibrez) ces valeurs ; voir la figure 8 ci à gauche.

3.2 La photo-résistance

La conductivité d'un matériau et donc sa résistance, est fonction du matériau lui-même mais également de son environnement : la température comme dans le cas de la thermistance (sec. 3.1) ou encore de la luminosité. C'est cette propriété que nous utilisons ici avec la photo-résistance. La luminosité ambiante peut être mesurée en lux. 1 lux correspond à un niveau de lumière en pleine nuit et 1000 lux

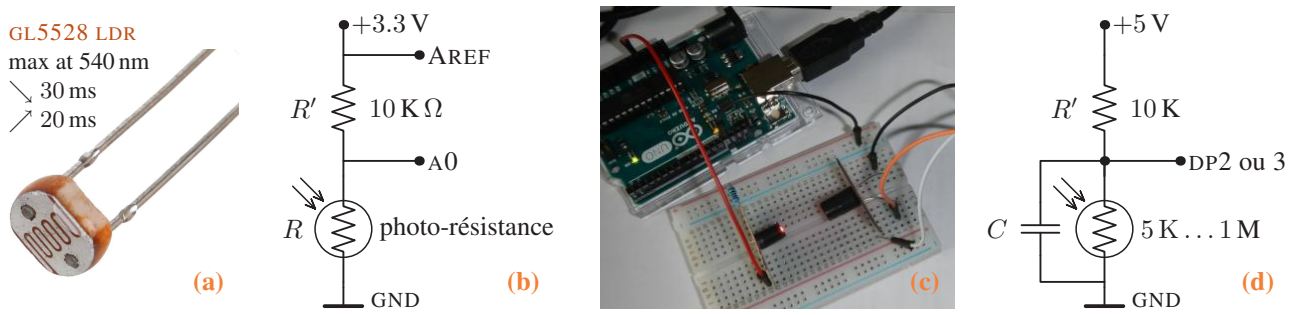


FIG. 9 – La photo-résistance (a) ; cablage en «diviseur de tension» avec V_{CC} de 3.3 V (b) ; mesure de R_{min} (c) ; comme capteur digital (d).

en pleine journée. La valeur de la résistance R d'une photo-résistance[19] suit une loi exponentielle en fonction de la luminosité L :

$$R(L) = R_0\, L^{-k}, \quad \log R = \log R_0 - k\, \log L, \quad \text{où } R_0 > 0 \text{ et } k > 0 \text{ sont des constantes.}$$

[19] appelée aussi LDR, pour *light dependent resistor*, ou encore *photoresistor*

Ainsi la courbe $\log R(L)$ est une droite avec une pente négative si bien que la valeur de la résistance diminue avec L depuis quelques centaines de kΩ ou même 1 MΩ dans l'obscurité jusqu'à 10-20 kΩ à 10 lux ou quelques centaines d'Ω en plein jour, ceci, bien évidemment, en fonction des modèles. Comme les thermistances, les photorésistances sont le plus souvent utilisées dans un pont diviseur de tension (fig. 9). Notez que pour gagner en stabilité, on utilise la source interne de 3.3 V comme V_{CC} (plus filtré et plus stable que les 5 V fournis par l'ordi sur son port USB, cf la sec. 2.3) pour alimenter la LED et surtout le diviseur de tension de la photo-résistance $R + R'$. Pour R' de 10 K, les valeurs typiques attendues sur A0 sont entre $0.9\,V_{CC}$ et $\leq 0.5\,V_{CC}$.

> **La source de 3.3V :** Les UNO et MEGA possèdent un régulateur linéaire séparé de tension 3.3 V/150 mA qui n'est pas utilisé par la carte et peut servir, par exemple, pour alimenter les capteurs, ou autres modules utilisant cette tension.

Alternativement, attachée à un pin digital, la photo-résistance sert comme un capteur «ON/OFF» de la présence de lumière dans des nombreuses applications style «photo cellule» ou «passerelle encodeur» dont le déploiement ressemble celui d'un bouton (sec. 2.4).

3.3 Capteur de force et poids

Les capteurs de force produisent des petites variations de résistance et sont montés typiquement sur un pont de Wheatstone (cf. sec. 3.1). Ce dernier est alimenté (dit excité) par une tension $u = V_{CC}$ et sorte une petite tension δ due au déséquilibre des résistances (on dénote les connexions $E_\pm$ pour l'excitation du pont, et $S_\pm$ pour la sortie/signal).

> **Jauges de contrainte résistives** de capteurs de force (dit *load cells*) sont caractérisés par leur sensibilité $s = \delta_{max}/u$, dont les valeurs typiques restent autour de 1 à 3 mV/V. Prenons $u = 3$ V. Dans ce cas, une cellule de 2 mV/V chargée à son poids maximal rend seulement 6 mV. Pour poids max de 50 kg, cela done 120 μV/kg ou 8 g/μV.

La fig. 10 ci-dessous montre le schéma standard de branchement de 4 capteurs de force à 3 fils. Chaque capteur de ce type possède deux

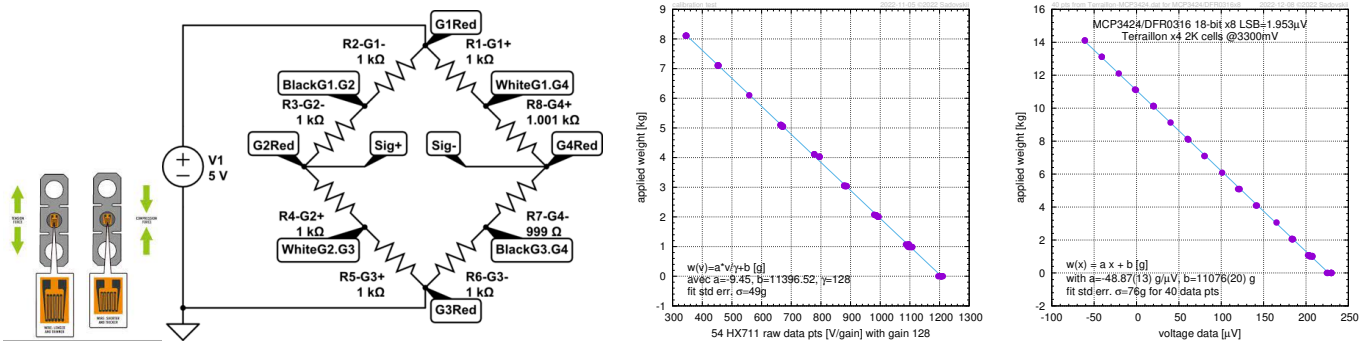


FIG. 10 – Montage de 4 cellules de force à 3 fils et leur calibration

résistances de, typiquement, 1K en série, un fil donnant leur milieu, deux autres les extrémités. On trouve celui du milieu (rouge dans la fig. 10) à l'aide d'un ohmmètre. Les deux résistances sont différents : soit l'une est fixe et l'autre est active (jauge de contrainte), soit les deux sont actives, une en *compression* et l'autre en *élongation*, avec les signes de leur ΔR opposés. Les couleurs des fils distinguent ces résistances. Ceci explique pourquoi on les branche «en cercle» et utilise les 4 fils du milieu comme les entrées/sorties du pont. La résistance du pont est de $\approx$ 2K.

Pour mesurer la très faible sortie différentielle $S_\pm$ (de quelques mV) avec une précision nécessaire, deux cartes/cirquits peuvent servir : HX711 (on la trouve souvent utilisée dans les balances) et MCP3424. Les deux possedent un ampli différentiel, un ADC, et communiquent leur données sur une ligne série. La carte HX711 propose en plus son propre alimentation stabilisée de 4.2V pour les $E_\pm$ du pont. La carte possède un ampli de $\times 64$ ou $\times 128$, et l'ADC de 24-bit. Cependant, le bruit est tel que cela revient plutôt à 16-bit «noise-free»[20]. La plage d'entree pour $\times 128$ est de 20 mV. Cote interface, on la trouve un peu rudimentaire. HX711 peut rendre une précision de 1 g avec les 4 cellules à 50 kg provenant d'une «balance de la salle de bain» qu'on trouve facilement ou on récupère d'un vieux appareil, mais on constate le problème de stabilité en usage continu : au fil de temps qu'elle reste allumée, sa ligne de base flotte.

Le circuit MCP3424 possède un vrai interface I2C (donc il s'attache facilement aussi à un RASPBERRY PI). Du point de vue des applications, il est plus versatile, mais on a besoin d'alimenter le pont, en utilisant, par exemple, les 3.3V stabilisés de ATMEGA328. Son ampli à gaine $\times\gamma$ de $\times 8$ (maxi), $\times 4$, ou $\times 2$, son ADC va jusqu'à 18-bit, et la plage d'entrée est de $V_{max} = \pm 2048/\gamma$ mV, donc 256 mV en $\times 8$. On peut facilement interpréter les données de MCP3424 en mV (μV et même nV). Elles correspondent aux valeurs de multimètre avec 10 μV d'écart systématique (pb impédance ?). Pour un LSB[21] de 1.95μV, pas pire ... L'autre cote attractive de MCP3424 est son multiplexeur à 4 canaux integré. Ainsi, dans le cas de TP Clément-Désormes (sec. 4.4), par exemple, cela permet de mesurer Δp et ΔT avec la même carte. Le problème pour les applications de MCP3424 est, dans des certains cas, sa résolution insuffisant (face aux bruits). Ainsi, selon la courbe de calibration dans la fig. 10, la sensibilité est de 50 g/μV, donc c'est logique de n'avoir que 100 g de

[20] Pour un signal en courant continu (DC), noise-free bits (noise-free resolution) $= \ln(\text{FS/p-p})/\ln 2$, où FS (*full signal*) est la plage du signal et p-p (*peak-to-peak*) est le bruit crête-à-crête qu'on peut estimer avec l'ecart type (*rms noise*) σ du signal comme $6.6\,\sigma$.

[21] La valeur de *least significant bit* (LSB) pour une résolution de r-bit et gaine γ est donnée par $2^{1-r}\,|V_{max}|/\gamma$

précision avec un LSB de 2 μV ! Le cumule des échantillons ne semble pas d'aider enormement. En plus, si la ligne de base de MCP3424 reste assez stable, qques LSB y flottent quand même... Ce que semble logique est d'essyer placer un pre-ampli d'une gaine de 10..100 devant l'entrée $S_\pm$ pour remonter de 1 a 10..100 mV et exploiter ainsi tout la plage de la carte.

3.4 Amplificateurs opérationnels

ARDUINO ne posséde que l'alimentation positive V_{CC} (dit «single rail») et ne mesure directement que les signaux analogques V positives et inférieures à son V_{CC}. Par ailleurs, avec son ADC de bord, la précision varie entre 1 mV si on utilise la référence interne $V_{REF} \approx 1$ V et $10^{-3}\, V_{CC}$. Ainsi on ne peut pas mesurer directement les signaux faibles à signe inconnu comme c'est souvent le cas dans les mesures sur le pont de Wheatstone (sec. 3.1). Les amplificateurs opérationnels (AOP, ampli-op), étant indispensables pour conditionner de tels signaaux, font fréquemment une partie essentielle de la chaîne d'acquisition. Il existe un nombre des cirquits intégrés AOP qu'on puisse déployer avec ARDUINO en monde «single rail», i.e., avec une seule V_{CC} de $+5$ ou $+3.3$ V et *sans alimentation negative*. Ainsi John Errington nous recommende un MCP6001/2/4 à technologie CMOS ou si non—les bons vieux LM324 et LM358 à technologie bipolaire. Les ampli-op's sont étudiés en cours d'électronique, on trouve également plein des informations et tutoriels sur internet. Un bref sommaire de leurs applications typiques en mode courant continu est donné ci-dessous.

Des faibles signaux électriques issus de capteurs de mesure sont traités à l'aide d'un amplificateur de mesure,[22] qui est constitué par plusieurs ampli-op's permettant d'augmenter l'impédance d'entrée et le taux de réjection du mode commun de l'ensemble.

Amplificateurs Pour mesurer les faibles différences V avec une précision elevée, il existe des cirquits integrés et des mini-cartes «breakout» tout prêtes, notamment HX711, ADS1115, ou encore MCP3424, qui combinent un INA avec réduction de bruit (et souvent à gaine programmable) avec un ADC à une résolution programmable et plus importante que celle de ARDUINO, et parfois avec un commutateur (multiplexer) 2×4, et qui communiquent par I2C.

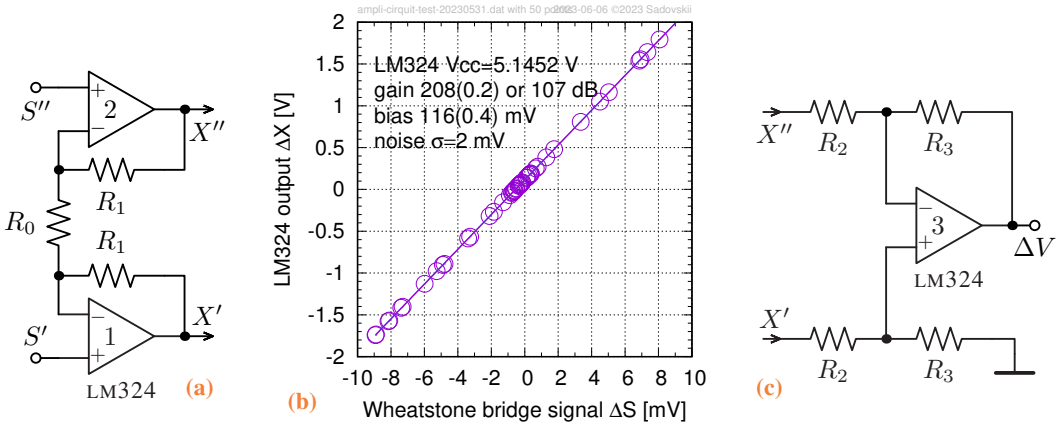


FIG. 11 – Amplificateur de mesure : (a) partie tampon, (b) résultats de sa réalisation (voir texte), et (c) partie differentielle.

Un amplificateur de mesure (INA) est présenté dans la fig. 11. Soit (S', S'') les tensions au bornes de sortie d'un pont de Wheatstone, avec la partie (mode) commune S et la différence ΔS, spécifiquement

$$S' = S + \frac{\Delta S}{2}, \quad S'' = S - \frac{\Delta S}{2}, \quad \text{où} \quad S = \frac{S' + S''}{2} \quad \text{et} \quad \Delta S = S' - S'' ,$$

qu'entrent dans cet INA. Idéalement, à la sortie de sa parie tampon (fig. 11a) on expecte les signaux

$$X' = S + g\,\frac{\Delta S}{2} \quad \text{et} \quad X'' = S - g\,\frac{\Delta S}{2} \quad \text{avec le gaine en mode différentiel} \quad g = 1 + \frac{2\,R_1}{R_0} .$$

En d'autres mots, alors que la différence ΔS est amplifié g fois et $X' - X'' = \Delta X = g\,\Delta S$, le mode commun S reste inchangé.

Réalisation : Dans la figure 11b on voit les résultats d'une réalisation de ce montage sur la base d'un LM324 avec $R_1 = 1$ M, et $R_0 = 10$ k. Ici, l'ampli-op et le pont de Wheatstone ont été alimenté par V_{CC} de 5.1452(2)V provenant d'un port USB du laptop LENOVO X230 en mode dormant. Les mesures de ΔS et ΔX ont été effectuées (en c.c.) respectivement par les TRMS-millimètres GOSSEN METRA 29S à grande précision de 1 μV et TEKTRONIX DMM916 à précision de 0.1 mV, plus standard mais suffisant pour ΔX. La stabilité de la source était approximativement 2 μV, celle de la sortie ΔX était d'ordre de 0.2 mV. On constate que l'ampli reste parfaitement linéaire et très peu bruyant dans l'intervalle d'au moins de ± 10 mV avec les tensions de sortie ΔX dans l'intervalle de ± 2 V. La gaine $g = 208$ est très proche à la valeur théorique de 201, mais on observe un bias assez important de 116 mV. Celui-ci est dû aux imprécisions de résistances $R_1' \approx R_1'' \approx R_1$ d'ordre de 1%, ainsi que aux imperfections de LM324.

[22] ou amplificateur d'instrumentation, appelé en anglais *Instrumentation Amplifier*, *in-amp*, ou encore INA

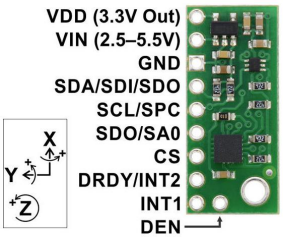
L'amplitude de ΔX à la sortie de la partie tampon de l'INA peut être déjà (comme dans notre exemple concrète) suffisamment grande pour être mésurée avec ARDUINO. Seulement, il nous faut effectuer la soustraction $X' - X''$. Pour cela, on utilise un troisième ampli-op en mode différentiel (fig. 11c) sortant une tension

$$\Delta V = \frac{R_3}{R_2}\,\Delta X\,.$$

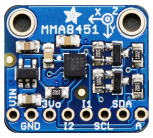
Evidemment, vu que cet ampli-op est en mode «single rail», il faut s'assurer impérativement, que $X' > X''$ et donc ΔV reste positive avec un marge. Et il faut également qu'elle reste inférieure à V_{CC}. En d'autre mots, on ne peux mesurer ainsi que les $0 < \Delta S < V_{\text{CC}}/g$. Si cela n'est pas le cas, on doit ajouter à X' une tension positive connue et fixe à l'aide d'un ampli-op supplémentaire (quatrième).

Estimons la précision du montage concret décrit ci-dessus et illustré dans la fig. 11. Assumant que la partie différentielle (fig. 11c) n'augmente pas le bruit et d'autres erreurs d'ordre de 0.1 mV déjà présents dans la fig. 11b, et que la ΔV maximale est de 1V, l'imprécision de 1 mV dans les valeurs de ΔV mesurés sera largement déterminée par la résolution du ADC de ARDUINO. On conclue qu'on pourra ainsi mesurer les signaux $\Delta S < 5$ mV avec une incertitude absolue de $1\,\text{mV}/g = 5\,\mu$V, i.e., à quatres chiffres significatifs maximum. Supposons par ailleurs qu'il s'agit des mesures de poids (sec. 3.3) entre 0 et 50 kg. Dans ce cas, l'imprécision de nos mesures monte à 50 g. Peut on l'améliorer ? Théoriquement si, car notre ampli-op sorte ΔX avec un bruit de 0.2 mV (déterminé entre autre, par les instabilités de V_{CC} dans les μV) et notre incertitude peut potentiellement être 10 g. Cependant, une mesure de 1 V avec l'incertitude de 0.1 mV nécessitera un ADC à 20 registres du type HX711. C'est deux fois plus que possède ARDUINO UNO, et très probablement bien plus lent. En restant avec ARDUINO, on peut assurer l'incertitude maximale de 10 g seulement dans l'intervalle de 0 a 5 kg et avec un gain additionnel d'ordre $R_3/R_2 = 5$ au stade différentiel.

3.5 Accéléromètre et gyroscope



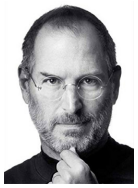
Les capteurs MEMS[23] à six degrés de liberté, tels qu'un MPU6050, combinent un accéléromètre 3D avec un gyroscope 3 axes. Il existe également des modules gyroscope seuls, tels que L3GD20H ci à gauche, aux caractéristiques similaires (16bit, plages `FS` de ± 250 / 500 / 2000°/s). Pour utiliser ce module (au plus simple, en I2C), on installe la bibliothèque `l3g-arduino` (voir leurs commentaires dans `L3G.cpp` et `L3G.h` et l'exemple `l3g-arduino-master/examples/Serial/Serial.ino`) ; pour l'alimenter, on passe la sortie $+5\,V_{\text{CC}}$ de la carte ARDUINO (fig. 1b) et sa masse sur VIN et GND respectivement ; pour communiquer, on connecte[24] les broches SDA/SCL (données/horloge) du bus I2C (adresse[25] `0x6b` par défaut, ou `0x6a`, si SDO est mise à la masse). Attention, la carte L3GD20H ci à gauche, possède des resistances «pull-up» de 4.7 K entre ces lignes SDA/SCL et VIN, ce que puisse affecter le fonctionnement des autres cartes sur I2C. Ainsi, une combinaison de deux telles cartes descende la résistance «pull-up» du bus à 2.35 K. Notez que `enableDefault()` sélectionne la plage `FS` de ± 250°/s (degrés par seconde, ou dps). Sur cette plage et avec la résolution de 16 bit, on estime la précision du gyroscope comme $\text{LSB} = 2\times\texttt{FS}/2^{16} = 0.00763$°/s, soit 7.63 mdps. Pour plus de renseignement, on consulte la page 10 du datasheet de L3GD20H, où la sensitivité `So` de 8.75 / 17.50 / 70.00 mdps/digit selon `FS` est indiquée. Donc une valeur «brut» (dit *raw*) de 114 corresponde approximativement à 1°/s, soit $\approx 114 \times$ LSB.



On utilise également la petite carte d'accéléromètre 3-axes MMA8451 (plages de ± 2 / 4 / 8 g), qu'on branche, comme dans le cas du gyroscope ci-dessus, sur Vin/GND et SDA/SCL (addresse[25] I2C `0x1D` par défaut, ou `0x1C` si la broche A est connectée à la masse), voir ci à gauche. Le codage nécessite deux bibliothèques, `Adafruit_MMA8451` pour lire les données «*raw*» et `Adafruit_Sensor` pour les traduire en unités physiques (m/s^2). A noter, que ce capteur possède également une détection de ses inclinaison et orientation (par rapport au pesanteur) consultables avec `getOrientation()`. Les capteurs accéléromètre-gyroscope se trouvent couramment dans les téléphones mobiles ou tablettes ; quelques applications sont présentées sec. 4.11.

[23] MEMS = micro-electro-mechanical system
[24] Dans le cas d'un montage mobile, par exemple sur un pendule, souder les fils légers. Pour identifier les broches I2C sur la carte ARDUINO, voir la note 7.
[25] Pour détecter au démarrage toutes adresses actives sur le bus I2C, consulter appendice D.

4 Projets UO ARDUINO 2025



Creativity is just connecting things.
When you ask creative people how they did something,
they feel a little guilty because they didn't really do it, they just saw something.
Steve

Lors de la dernière séance TP ARDUINO, à la fin de la séance d'initiation, on se met en groupes de travail (de 2, 3, 4) et détermine un projet. On peut sélectionner un projet scientifique parmi ceux qui vous sont proposés, mais tout autre projet utilisant Arduino est possible. Il est donc indispensable d'étudier cette section, consulter de nombreux sites internet et toutes autres sources, échanger entre vous, avant la séance d'initiation (la première rencontre).

Vous devrez étudier votre projet d'un point de vue théorique et en discuter avec l'enseignant «mentor» pour commander éventuellement les éléments qui vous seront utiles pour son développement, idéalement à la fin de la 1ère séance.

Vous devrez réaliser le projet ou tout au moins aller le plus loin possible, en concertation avec l'enseignant responsable et présenter vos travaux à l'issue des séances de projets. Vous aurez la possibilité d'emporter vos montages pour travailler chez vous, en dehors de l'université. Nous vous rappelons que vous trouverez sur le réseau la majorité des informations qui vous seront nécessaires à la compréhension de votre projet. Vous aurez ensuite dans la majorité des cas à adapter un programme à votre cas particulier.

Parmi les projets que nous avons recherché (et donc nous pensons qu'ils seront à la fois faisables et intéressants dans le cadre de ce module) nous pouvons envisager les projets suivants.

Traitement des données Dans des certains projets de style «scientifique», on aura besoin de transférer un grand volume des données collectées par ARDUINO et les traiter dans un ordinateur (un PC, MAC, ANDROID). Sous LINUX ou MAC OS, le plus simple est de passer directement par un scripte `bash` pour récupérer les données via un port série dans un fichier texte : rien à compiler, installer etc, s'est un scripte ! Sous WINDOWS, il existe une procédure similare via un port `COM`.

Par la suite, les étudiants traiteront leurs données comme et avec quoi ils veulent. En particulier, on peut suggérer regressi. C'est un util interactif, un peu style tableur, qui peut avaler tout fichiers de données au format texte. Il est possible même de le coupler avec ARDUINO. L'ancien version de ce logiciel existe pour windows, il a ete utilise beaucoup à la fac et dans les lycees, mais ses versions actuelles (en `python`) n'existent que pour MAC OS, LINUX, et ANDROID.

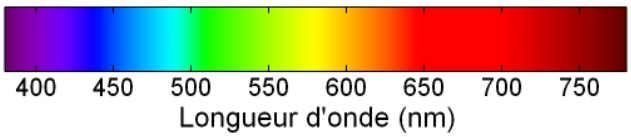
Cote ARDUINO, on exploite le programme `adcsampler` (voir l'appendice B). Il possède un protocole permettant de définir le nombre des échantillons, le taux, etc, et de lancer la collecte des données (sec. 2.5). Il est facilement modifiable selon le projet. L'example complet, avec les scriptes `bash` se trouve dans le topo réaction oscillant. Il est exploité aussi dans le colorimètre (sec. 4.1).

4.1 Colorimétrie

Nous combinons les expériences et le code développé dans les sec. 2.2, 2.5, et 3.2 pour construire et tester un colorimètre (spectrophotomètre monochromatique) permettant de détecter des molécules spécifiques et de mesurer leurs concentrations en temps réel.

4.1.1 Absorption de la lumière, loi de *Beer-Lambert*

Considérons un faisceau de lumière de jour traversant un milieux actif tel qu'une solution aqueuse *colorée*. La lumière de jour étant blanche, elle possède toutes les couleurs (toutes les longueurs d'onde λ) du spectre visible ci-dessous.



On conclut que le milieux retient ou *absorbe* la partie du spectre complémentaire à sa couleur : un objet transparent vert laissera passer le vert et arrêtera toutes les autres couleurs. Par ailleurs, il est clair que l'intensité I de la lumière absorbée diminue avec l'épaisseur l du milieux. La relation entre I et l est donnée par la loi *Beer-Lambert*.

Sous sa forme différentielle, cette loi prédit que la quantité de la lumière absorbée $\mathrm{d}I$ par l'épaisseur $\mathrm{d}l$ est

$$\mathrm{d}I = I(l+\mathrm{d}l) - I(l) = -\varepsilon\, C\, I(l)\, \mathrm{d}l\ . \tag{4.1a}$$

On constate que l'absorption est un phénomène forcé, provoqué, induit par l'intensité I de la lumière traversant le milieux : plus on a de la lumière, plus il y a de la lumière $\mathrm{d}I$ absorbée. Le coefficient de proportionnalité comprend naturellement la concentration C [Mol/L] des molécules actives et leur caractéristique individuelle ε [L/Mol/cm] appelée le *coefficient d'absorption* ou l'*absorbance*[26]. La forme (4.1a) s'applique aux petites épaisseurs $\mathrm{d}L \approx \Delta l$. Sinon, on doit utiliser la loi intégrale

$$\log I(l) - \log I(0) = -\varepsilon\, C\, l \quad \Rightarrow \quad I(l) = I_0 \exp(-\varepsilon\, C\, l)\ . \tag{4.1b}$$

[26] selon les domaines d'applications spécifiques, on rencontre aussi l'*absorptivité*, la densité optique du milieu, l'opacité ou l'extinction.

Cependant, pour des petites variations de $x = l\,C$, la forme (4.1b) est excessive et difficilement applicable, surtout en présence d'erreurs de mesure. Ainsi, si x varie entre $x_{\min}$ et $x_{\max}$, nous utiliserons plutôt une *linéarisation* de (4.1b)

$$I(l) \approx I_0\,\exp(-\varepsilon\,\bar{x})\big(1-\varepsilon\,(x-\bar{x})+\dots\big) \quad \text{avec } \bar{x}=(x_{\max}-x_{\min})/2. \tag{4.1c}$$

Enfin, dans une étude avec une épaisseur fixe et une concentration C qui est fonction de I, on réécrit (4.1c) sous forme

$$C(I) = a_0 + a_1\,I \tag{4.1d}$$

4.1.2 Régression linéaire

Considérons les mesures (x_i, y_i) avec $i = 1, \dots, N$ pour chaque valeur de x. Dans notre cas (sec. 4.1.1), y et x représentent l'intensité I et la concentration C. On estime les incertitudes de y_i égales à Δy_i (ou à ces écart types σ_i), et on suppose dans le cas de la *régression*, que les valeurs de x sont "exactes". Ainsi x est appelée *variable de contrôle*. On étude le phénomène décrit par la loi théorique $y = f(x;\alpha)$ avec des paramètres phénoménologiques à déterminer $\alpha = \alpha_1, \alpha_2, \dots$. Dans notre cas, il s'agit de l'approximation linéaire (4.1d) avec ses deux paramètres $\alpha = (a_0, a_1)$. En d'autres termes, nous cherchons à faire passer une *ligne droite* appelée aussi «courbe de calibration» par les points $(x_i, y_i) = (C_i, I_i)$ à partir de mesures de I_i pour des échantillons de concentration C_i connue (voir l'appendice E). Ces échantillons de concentration connue sont appelés les «standards». Bien entendu, leur nombre N doit être égale ou être supérieur au nombre des paramètres α (donc pour nous $N \geq 2$). Par la suite, en connaissant les α, nous pourrons déterminer la valeur de C pour toute valeur de I dans l'intervalle $I_{\min} \dots I_{\max}$ couvert par (4.1d).

4.1.3 Colorimètre avec ARDUINO

Nous utilisons une photo-résistance (sec. 3.2), le branchement et les contrôles de la LED (sec. 2.2.1), et les principes de l'échantillonnage ADC rapide aux intervalles de temps fixes (sec. 2.5) pour construire un colorimètre. Ce dispositif sera capable de mesurer des concentrations avec les périodes de 300 μsec (plus prudemment 500 μsec, voir sec. 2.5) et donc au taux maximal de 2-3 kHz. Les formules (E.1b) seront implémentées dans le code de calibration de colorimètre.

Matériel : une carte ARDUINO UNO et un ordinateur pour lui communiquer en TTY, une LED rouge générique (de λ approximativement 630 ou 660 nm), des résistances de 220Ω et $R' = 10\text{k}\Omega$, une photo-résistance de $R_{\min} = 3.1\text{k}\Omega$ exposée par la LED rouge, voir la fig. 9, et $R_{\max} \approx 0.35\text{M}\Omega$ sans lumière, un potentiomètre de 10kΩ en option. Carton, colle, scotch, feutre noir ...

Le principe : La LED et la photo-résistance sont montées sur le breadbord à une distance ≈ 15 mm (fig. 12). Elles sont branchés selon les schémas dans la fig. 2a et 9 avec VCC de 3.3V. Pour pouvoir commuter les +5V de la LED, on utilise la pin digitale 10 (voir `LED_PIN` dans le code). Le potentiomètre peut servir à adapter la valeur de la tension de référence ($V_{\max}$ anticipée sur la R) inférieure à 3.3V. La LED envoie sa lumière vers la R, éventuellement, à travers une cuvette. On évite toutes lumières parasites (caches).

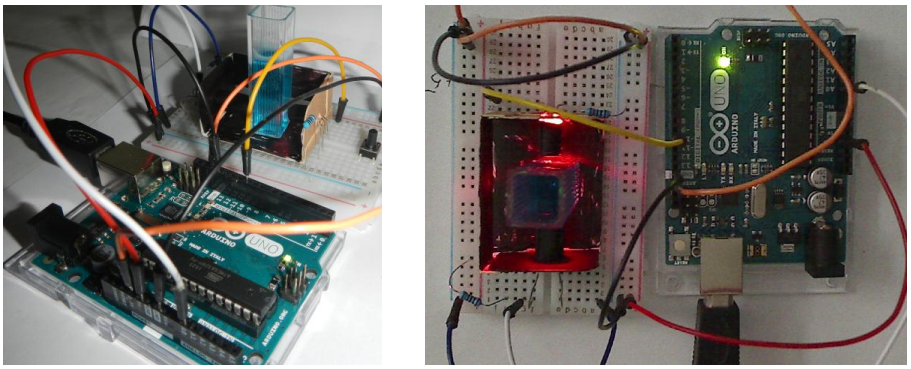


FIG. 12 – Montage du colorimètre avec $CuSO_4$ dans une cuvette.

Le code et l'interface : Une fois téléchargé dans le micro-contrôleur, le programme `~/Arduino/cmeter/cmeter.ino` (voir l'appendice C) permet d'opérer et de communiquer avec le colorimètre (voir également la sec. 2.5 et le code). Ainsi `A5N10` préconise d'afficher deux mesures de 5 échantillons moyennés chaque (lisez le code !). Le taux de l'échantillonnage est réglé par la période `T` en μsec. Les mesures sont démarrées par `S`. On allume/éteint la LED avec `X`. Les points de calibration successifs sont introduits par `d`, et on peut réinitier l'ensemble de ces points avec `D`. Pour effectuer la calibration et déterminer les valeurs de a_0 et a_1 on utilise `C`. Par exemple, la commande `d500S` ajoute le point dont la valeur de concentration est 500. A noter, qu'on cible l'intervalle d'affichage à 4 chiffres. Pour cela on implémente des facteurs dans l'échelle de C. Ainsi 0.5M devient 500 mM ou même 5000 10^{-4}M. Par défaut, les valeurs 0 ... 1023 de l'ADC sont affichées. Pour les convertir en concentration C, utilisez `c`. Par ailleurs, la commande `r` permet d'afficher la valeur de la résistance de R.

4.1.4 TP Mesures de concentration

Objectif : examiner la relation entre l'absorbance et la concentration du sulfate de cuivre (II) $CuSO_4$ dans une solution aqueuse (la loi de *Beer*) et déterminer la concentration dans des échantillons de concentration inconnue. Sur le spectre d'absorption d'une solution de 0,5M de $CuSO_4$ dans l'eau à température ambiante, on peut voir que l'absorption est maximale à la longueur d'onde $\lambda_{max} = 780$ nm, avec un coefficient d'absorption molaire ε de ≈ 12.5 mol^{-1}cm^{-1} ; à 635 nm il est de 2.8 mol^{-1}cm^{-1}.

Matériel : des petites cuvettes plastiques génériques pour les colorimètres et spectrophotomètres (4 ml, $12.55\times12.65\times44.55$ mm) avec couvercles hermétiques pour les solutions standardisés de $CuSO_4$ de la concentration n [10^{-1} M] avec $n = 0, 1, 2, 3, 4, 5$ et 2-3 solutions de contrôle.

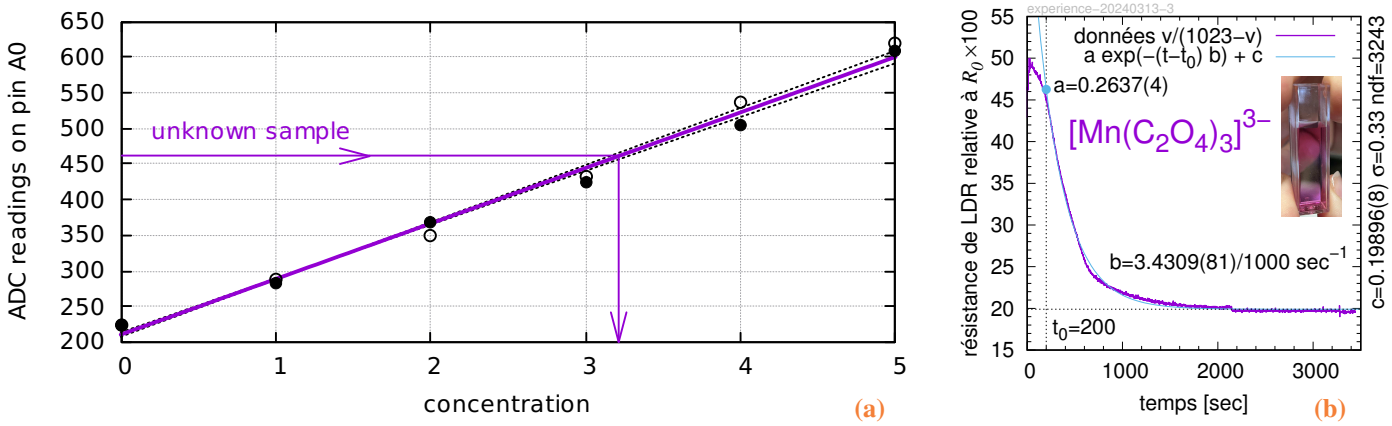


FIG. 13 – Etudes colorimétriques : (a) courbe de calibration de colorimètre ; (b) observations en temps réel.

Mode opératoire : Calibrez votre appareil avec les solutions standards. Représentez les données de calibration et la courbe sur un graph (tableur ou papier millimétrée, voir la fig. 13a) et confirmez la linéarité (cf. appendice E) et le fonctionnement du code `linreg()`. Mesurez la solution inconnue et en déduisez sa concentration.

4.1.5 TP Mesures de la constante cinétique

Objectif : utiliser le mini-colorimètre à la base de ARDUINO UNO dans la sec. 4.1.3 pour étudier *en temps réel* la dissociation du Mn(III) *tris-oxalate* (avec une durée des observations entre 10 min et 1 h) et déterminer la constante de la vitesse de cette réction ainsi que, optionnellement, modéliser sa dépendance en température (pour différentes agitations et intensités de la lumière ambiante).

Exposé : Le Mn(III) *tris-oxalate* $[Mn(C_2O_4)_3]^{3-}$ est un anion complexe unstable, qu'on peut obtenir[27] en ajoutant la solution du permanganate de potassium (violet) à une solution aquatique incolorée du sulfate de manganèse et de l'acide oxalique :

$$4\,MnSO_4 + 15\,HOOC{-}COOH + KMnO_4 \rightleftharpoons 5\,H_3[Mn(C_2O_4)_3]^{3-} + 3\,H_2SO_4 + 4\,H_2O, \tag{4.2a}$$

$$4\,Mn^{2+}(aq) + 15\,(COO^-)_2(aq) + MnO_4^-(aq) \rightleftharpoons 5\,[Mn(C_2O_4)_3]^{3-}(aq) + 4\,H_2O. \tag{4.2b}$$

Cet anion est photosensible, et étant exposé à la lumière du jour ($h\nu$), il se décompose rapidement même aux temperatures ambiantes

$$[Mn(C_2O_4)_3]^{3-}(aq) \xrightarrow{h\nu} 2\,MnC_2O_4(aq) + 3\,H_2C_2O_4(aq) + 2\,CO_2\uparrow.$$

Bien que son mécanisme précis est assez compliqué, dans une première approximation, la dissociation suit la loi cinétique des réactions d'ordre 1. Elle est relativement lente, parce que une énergie supplémentaire (lumière) est requise pour exciter son état de transition. En plus, la stabilité thermique de $[Mn(C_2O_4)_3]^{3-}$ est baisse, et la réaction est affectée par la température de façon très significative.

Mode opératoire : on suive le mode opérationnel simplifié[28] Deux solutions, acide oxalique (sans additif de $MnSO_4$) et permanganate de potassium à 0.02 M, 1.5 mL chaque, sont ajoutées (de préférence, sans exposer excessivement à la lumière) dans une cellule de photométrie (acrylique, $1\times1\times4$ cm, voir la fig. 13b), laquelle est secouée, et placée dans le colorimètre. Une solution marronne contenant le Mn(III) *tris-oxalate* se forme. Notez que le maximum d'absorption de $KMnO_4$ est à 450–470 nm. Donc pour la colorimétrie, on

[27] P. A. Nikolaychuk et M. M. Vayner, The decomposition of Tris-(Oxalato)-Manganate(III) complex ion as the reaction suitable for the laboratory practice on chemical kinetics, *Int. J. Therm. Chem. Kin.* **1**(2) 50–9(2016) ; Practical exercises in physical chemistry. Photometry. I. Shenderovich, ed, Freie Universität Berlin (2006).

[28] Pour tout réalisations de ces réactions, se rapprocher à Mme *Caroline Duhr*, responsable des laboratoires de chimie organique et biochimie à ULCO, Calais, qui les a testé et simplifié en 2024 (projets L2 S4). Cependant, on note que l'ajout de $MnSO_4$ comme une source des cations de Mn^{2+}(aq) reste indispensable pour former le *tris-oxalate* selon (4.2). En absence de $MnSO_4$ l'interpretation des résultats est plus complexe.

doit utiliser soit une LED bleue (assez rare), soit une blanche avec un filtre bleu, comme on fait pour la réaction oscillant. Par la suite, les mesures (avec un taux de 1 ou 0.5 Hz) démarrent immédiatement. La solution devient graduellement (15 à 45 min) incolorée. Optionnellement, plaçez votre colorimètre et les solutions dans une étuve thermostatique (sur une plaque chauffante du labo) pour contrôler et varier leur températures.

Exploitation des mesures : Selon la loi d'absorption sous sa forme linéarisée (sec. 4.1.1, confirmer les conditions de linéarité), la concentration $c(t)$ de Mn(III) *tris-oxalate* et proportionnelle à la résistance $R(t)$ de la photorésistance (LDR). Cette dernière est donnée par la formule $v/(1023-v)$ de diviseur de tension dans la fig. 9b (cf la sec. 3.1, le cas d'une large $\Delta R/R$), où v est le signal sur le pin `A0` digitalisé par l'ADC de votre ARDUINO. On ne s'interesse qu'au taux de décroissance b [sec^{-1}] de $c(t) \propto R(t)$ (surtout que la concentration initiale est difficilement contrôlable, et la constante R' peut être ajustée afin de maximiser la plage de réponse de colorimètre). Pour déterminer b, modélisez $R(t)$ par la fonction $a\exp(-(t-t_0)\,b)+c$, où $t_0 \geq 0$ représentent, respectivement, la période initiale à ignorer et la «ligne de base», i.e., la limite $\lim R(t)$ pour $t \to \infty$ (cf la fig. 13b). Au premier temps, estimez t_0 et c en traçant le graphe de $R(t)$. Par la suite, tracez $\log(R(t)-c)$ pour $t \geq t_0$ et trovez son coefficient directeur.

4.1.6 Développement

On peut envisager des projets d'amélioration de certains éléments de notre petit appareil, notamment

Op Amp Afin d'exploiter mieux la plage de résolution de l'ADC de μCU et d'augmenter ainsi la précision, insérer un ampli opérationnel pour (a) déduire la tension de base $V_{\min}$ (aka *debias*) et (b) amplifier le signal par facteur de $\times 2$ ou même 3.

EEPROM Sauvegarder les résultats de calibration (et certains autres réglages) dans la mémoire non-volatile de ARDUINO pour éviter leur perte. Au présent, l'appareil doit être recalibré chaque fois que il est rebranché (mis sous tension) !

Ecran Ajouter un écran (et quelques boutons de contrôle) pour afficher la concentration instantanée en toute autonomie.

GUI Développer une application sur PC pour communiquer avec l'appareil en utilisant Java ou Tcl/Tk.

Bluetooth Installer une carte («module») bluetooth et développer la même application pour ANDROID.

Température Ajouter une thermistance (sec. 3.1) pour contrôler la température des échantillons.

Spectre Étudier les possibilités d'utiliser des LED multicolores RGB et obtenir ainsi un «spectrophotomètre» à trois λ.

Enregistrement en temps réel afin d'étudier, par exemple, la cinétique des réactions chimiques, voir la sec. 4.1.5.

4.2 Spectrophotomètre (DS)

Connecter ARDUINO à un ancien spectrophotomètre à réseau pour effectuer des mesures automatisées.

4.3 Mesure de la chaleur massique de l'eau et de la chaleur latente de fusion de la glace (DS)

On se propose d'informatiser le TP mesures calorimétriques de L2 physique-chimie. Les mésures de la température sont détaillées dans la sec. 3.1. L'échantillonnage en temps réel est décrit dans la sec. 2.5, d'où on récupère le code pour prendre les mesures de température. Le courant I (on se renseigne sur le IC ACS758 et/ou les mesures ampèremétriques avec ARDUINO) et la tension U_{CC} (avec le ADC) de chauffage sont stabilisés et peuvent être mésurés juste une fois au départ et une fois à la fin de l'intervalle de temps des mesures de la température. On improvise un petit mélangeur (un électro-aimant vertical reglée par le μCU en PWM à travers certainement d'un mosfet). Finalement, on mésure le poids avec un jauge de contrainte (pont de Wheatstone) et la carte HX711 (voir sec. 3.3). A noter que cette carte possède deux entrées A et B, dont une peut servir pour les mésures de température et l'autre pour le poids. Pour l'interaction avec un ordi (la RASPBERRY 3 sous LINUX dans la classe que vous utilisez pour vos études de ARDUINO à Calais) on s'inspire du scripte pour `bash` dans le TP «réaction oscillante» et on traite les données par l'informatique (MAXIMA, GNUPLOT).

4.4 Expérience Clément–Désormes (DS)

Il s'agit d'un autre TP de L2 physique-chimie qui a pour but la détermination de la constante adiabatique γ de l'air. Dans ce cas, l'utilisation de l'μCU ARDUINO est même plus interessant que dans la sec. 4.3, car elle permet des observations du procès en fonction de temps. Ce qu'est impossible autrement. On équipe la bonbonne Clément–Désormes avec les capteurs (différentiels et totales) de la pression p et de la température T (sec. 3.1) et on récupère leur données en temps réel (sec. 2.5) pendant la période de l'établissement de la température après la détente. Pour mesurer précisément les petites variations de T, on utilise le montage en pont de Wheatstone (fig. 7d, cf. sec. 3.3). On utilise également un capteur de pression différentielle MPXV5010DP, MPXV7002, ou encore un simple MPX53DP pour observer directement la surpression Δp dans la bonbonne.

4.5 Pèse-ruches à distance

On se propose d'échantillonner l'évolution de poids d'une ruche en temps réel. Ceci serve aux apiculteurs pour déterminer si les abeilles sont en train de produire le miel (et donc tout va bien) ou de le consommer (et il faut les alimenter en supplément avec un sirop du sucre). Pour une ruche éloignée (en montagne...), les échantillons peuvent être envoyées par SMS en permettant ainsi à l'apiculteur de ne pas se déplacer inutilement. Une ruche pèse entre 10 et 50 kg ; son poids varie en fonction de la présence des abeilles dont le poids peut aller jusqu'à 3 kg (soit environ une colonie de 30 000 abeilles). Les changements nets journaliers aillent de 100 g à 1 kg. Ceci suggère un échantillonnage une fois par heure, cumulé en 24 h, référencé une fois, par exemple au minuit (quand tout les abeilles dorment dans la ruche), et reporté par SMS. On utilise, évidemment, les capteurs de force. La solution économique serait d'employer les capteurs à 3 fils (formant un démi-pont de Wheatstone, sec. 3.3) qu'on trouve couramment dans les pieds des pèse-personnes de commerce. Alimenté par un accu au Pb 12 ou 6 V ou un accu LiPo de 4.2 V, avec, si on veut, un système de recharge solaire, l'appareil doit économiser sa consommation. Ceci implique le mode «`POWER DOWN`» interrompu par un horloge RTC (sec. 2.4.1). On y ajoute un bouton, une carte GSM style SIM800L, et la gestion d'alimentation (sec. 2.7) des capteurs activée elle aussi que pour les courts moments des mésures. Notons que la carte GSM permets de déterminer l'heure et la date au démarrage et que la carte RTC possède un thermomètre qui peut être utile pour compenser la variation de la sensibilité des capteurs en fonction de °C. En option, on ajoute des autres capteurs environnementaux.

4.6 Capteur de vitesse de vent par sonde Pitot (DS)

On propose de fabriquer un capteur autonome de vitesse relative (résistance) du vent sur la base de ARDUINO et d'une sonde (tube) Pitot (Henri Pitot en 1732, voir le cours de hydrodynamique). Ce capteur peut être déployé dans une voiture (les voitures de F1 possèdent une sonde de ce type), drone, etc. L'objectif minimum sera d'avoir un prototype capable de mesurer la vitesse avec une précision et une stabilité raisonnables et les renvoyer par la connexion série/usb et/ou sur un écran lcd. En option, on peut envisager une connexion via bluetooth vers des smartphones et une application pour piloter le capteur. La sonde peut être trouvée d'occasion ou fabriquée au laboratoire. Pour les mesures, il s'agit de piloter plusieurs capteurs de pression (statique, dynamique) et de température, utiliser leur données pour calculer la vitesse, puis tester et calibrer l'appareil. Pour les plus avancés, une correction pour la vitesse réelle mesurée par un GPS peut y être ajoutée.

4.7 Capteurs environnementaux

Dans différentes applications d'évaluation de la pollution, d d'installations chimiques (fuites), de combustion ou autres, on cherche à mesurer les concentrations des gaz (NO_2, CO, CO_2, O_3, SO_2 ...) ainsi que la pression, la température et les particules fines. On propose de développer un appareil portatif capable d'effectuer et d'enregistrer ces mesures (mode traceur) en temps réel et en autonomie pendant une certaine période du temps. Ce genre d'appareil peut être monté sur un drone ou un robot pour travailler dans les zones dangereuses, voir inaccessibles aux hommes. Ce projet est naturellement d'importance pour les sites de Calais et Dunkerque.

Ce projet ainsi que le projet Pitot fait partie du projet initial de BQE 2017. On envisage de piloter plusieurs capteurs avec un enregistrement des données sur une carte CF en temps réel, par exemple chaque seconde. En option, on peut envisager une communication wifi ou RF, ou même GSM.

Si la partie électronique n'est pas trop lourde, une calibration des capteurs peut être envisagée (en collaboration avec les chimistes ? sur Dk ? chambre des essays). On peut avoir plusieurs binômes faisant différents capteurs et autres parties de l'appareil, notamment un système d'enregistrement universel (avec aussi données GPS) à déployer ici et dans le capteur Pitot.

4.8 Viscosimètre avec un capteur inductif de déplacement/vitesse (DS)

Pour mesurer la viscosité d'un liquide (glycerol) on utilise une petite bille en acier du rayon r descendant le long de l'axe d'un cylindre en plexiglas de rayon $\rho \gg r$ et de longueur $l = 50$ cm rempli du liquide et placé verticalement. Ce viscosimètre fait parti des TP en L1 et L3. Le régime stationnaire (vitesse constante) s'installe après 10 premiers centimètres de descente. On détecte le passage de la bille à 10, 20, 30, et 40 cm ; le temps de passage entre les points du repère donne la vitesse stationnaire qui dépend de la viscosité du liquide (loi de Stokes). A noter que le même dispositif peut être testé (sans liquide) pour mesurer la constante gravitationnelle g.

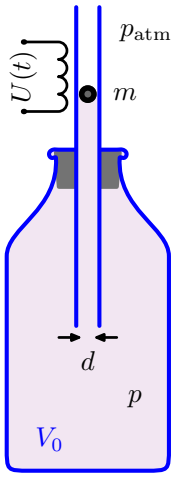
La détection utilise le principe d'un détecteur de métal. On entoure le cylindre par un câble fin isolé en faisant 8 tours à chaque point de repère (donc $N = 32$ en tout) et on mesure le changement d'inductance L de l'ensemble provoqué par le passage de la bille. Pour $N = 32$ tours de câble et $\rho = 25$ mm (5m de câble) on obtient

$$L \approx 5\, N^2\, \rho = 128\, \mu\,\text{H}.$$

Pour mesurer L en temps réel on passe des impulsions de courte durée (quelques μsec). On peut utiliser Arduino lui même pour donner les impulsions, voir aussi d'autres projets, et les capteurs industriels. Dans le circuit, le câble fait partie d'un filtre LR, où on doit avoir au moins $R = 220\Omega$ pour protéger le circuit d'Arduino de surcharge. La résistance du câble étant de quelques Ω, donc bien inférieure à R. On peut ainsi anticiper le temps caractéristique du filtre $\tau = L/R = 0.58\mu$sec qui limite la durée utile des impulsions. On note que la fréquence du contrôleur ATmega326 est de 16MHz et son cycle de 0.06μsec est seulement 10 fois inférieur à τ. Ainsi pour former les impulsions de quelques μsec il est préférable d'utiliser l'accès direct aux registres au lieu de la procédure `digitalWrite` de la bibliothèque, trop gourmande en opérations. Le délai de lecture de son ADC avec `analogRead` est de 100μsec peut aussi être diminué.

Autrement, pour le même $L = 100\mu$H il existe aussi un circuit intégré CS209A avec son propre oscillateur dont le fonctionnement reste à étudier.

4.9 Expérience de Rüchardt



Une bouteille de volume V_0 est munie d'un tube de petit diamètre interne d. Une balle en acier de la masse m et de même diamètre glisse facilement dans le tube en faisant des petits déplacements Δy tout en gardant la bouteille enfermée hermétiquement. Le volume V_0 est tel que ces déplacements ne l'affectent pratiquement pas. La pression p dans la bouteille dévie légèrement de la pression atmosphérique p_{atm}. Ce dispositif permet de mesurer la valeur de la constante adiabatique γ (coefficient de Laplace) du gaz dans la bouteille. Une bobine placée en haut du tube et alimentée par une source de la tension $U(t)$ sert à ajuster la position de la balle ou à forcer ses oscillations. Dans le but de développer un TP moderne pour les L3 et Master, on propose d'utiliser un microcontrôleur Arduino pour échantillonner en temps réel (sec. 2.5) les mesures simultanées de déplacement $\Delta y(t)$, de pression $p(t)$, de température du gaz $T(t)$, et de tension $U(t)$. En particulier, les déplacements sont mesurés avec un capteur laser 850nm *VL6180X/VL53L0X* (Time of Flight Micro-LIDAR Distance Sensor) capable d'un taux d'échantillonnage maximal de $\approx 200Hz$ (voir sec. 2.7.1 de sa notice). Pour une pression atmosphérique ($p_{atm} \approx 100$ kPa), une masse $m = 10$ de 20 g, $V_0 = 10$ L, $d = 10$ à 20 mm, $\gamma = 7/5$ (gaz diatomique) à la température 25°C, les fréquences propres sont de l'ordre de 1 à 10 Hz.

4.10 Magnétomètre

On propose d'utiliser les modules «breakout» de SparkFun (le circuit intégré de HMC5883L de Honeywell, ±1 à ±8 G, 12-bit, LSB 1 à 4 mG, à 75 Hz), de Pololu 2737 (LIS3MDL par ST ±4 à ±16 G, 16-bit, LSB 0.5 mG, RMS 4 mG, 150 Hz, 10×23 mm), de Sparkfun 19921 (MMC5983, FSR ±8 G, 18-bit, 0.06 mG LSB, bruit RMS 0.4 mG, 8×19 mm), de Adafruit 5579 (MMC5603 ±30 G, 20-bit, LSB 0.06 mG, jusqu'à 1 kHz), 4488 (LIS2MDL, jusqu'à ±50 G), tout avec un interface I2C, ou, enfin, un capteur simple analogique lineaire à effet Hall KY-035 (circuit AH49E) pour mesurer le champ magnétique en temps réel. Par la suite, on peut utiliser une imprimante 3D pour scanner le champ dans un volume, ou avoir un système de moteurs. Applications : TP physique, entre autres. Une autre application fortement intéressante est de construire un pendule afin de observer et mesurer la vitesse de rotation de la Terre[29].

Notez que la sensibilité du capteur ne doit pas être trop élevée si les champs magnétiques B visés sont plus forts que celui de la Terre $B_{\oplus} \leq 60\,\mu$T ou 0.6 G[30]. Estimons nos besoins. Un courant continu circulaire I de rayon r autour de 0 dans le plan xy, crée le champ

$$B_r(z) = \mu_0 \frac{I}{2} \frac{r^2}{(r^2+z^2)^{3/2}} \quad \text{avec } \mu_0 = 4\pi\times10^{-7}\text{ H/m} \quad \Rightarrow \quad \mu_0 \frac{I}{2r} \text{ au centre à } z = 0.$$

Pour N spires de même rayon on compte $NB_r(z)$. Par ailleurs, le champ sur l'axe d'un solénoïde infinie ($l \to \infty$) est

$$B = \mu_0\, n\, I, \quad \text{avec } n = N\, l^{-1} \text{ le nombre des spires par mètre.}$$

Ainsi, en TP L2, à l'intensité de courant de $I = 5$ A (maximale), le champ varie entre 1 et 0.5 G dans les anneaux simples de $r = 3$ et 6 cm, et monte jusqu'à 63 G dans le solénoïde de $n = 10^3$ qu'on dispose. En même temps, le capteur linéaire à effet Hall AH49E sorte un signal analogique de 1 à 4 V pour -1000 à $+1000$ G et $V_{CC} = 5$ V (recommendée). Par conséquent, la sensibilité typique du AH49E égale 1.5 mv/G tandis que son bruit annoncé reste au dessous de 0.1 mV, soit 66 mG. Pour les valeurs maximales de ±100 G attendues, le signal sera dans l'intervalle de 2350 à 2650 mV On peut tenter d'utiliser ce capteur au condition de pouvoir bien mesurer les 0.1 mV. Pour cela, il vaut pré-amplifier $\times3$ et soustraire la base de 2350 mV avec un ampli-op et atteindre 0.5 G/mV et le bruit de 0.3 mV. L'ADC de ARDUINO de 10 bit dans le diapason $[0, 1]$ V (référence interne AREF = 1 V) nous assure le LSB = 1 mV. En cumulant une cetaine de mesures, on peut espérer de poivoir mesurer B avec une incertitude de 0.1 G, ce que suffit marginalement nos besoins en TP L2.

4.11 Projets autour de accéléromètre et gyroscope

On considère les différents applications des cartes accéléromètre–gyroscope (sec. 3.5).

4.11.1 Rotation de la Terre

Les MPU6050 et L3GD20H peuvent servir à mesurer directement la rotation de la Terre (voir la fig. 14 et cf la note 29). On pointe l'axe z du gyroscope verticalement vers le haut, et on oriente son axe x vers le nord (N, voir la fig. 14a avec x et $\vec{\Omega}$ alignés) ou sud (S). Pour passer de N à S, le gyro est retourné de 180° autour de la verticale : N $\to$ S : $(x, y, z) \mapsto (-x, -y, z)$. Pour chaque orientation, les données [degrés/s] sont collectées pendant *une minute* à $\approx$100 échantillons/s dans la plage de 250°/s (la plus lente et sensible). Chaque mesure représentera donc une moyenne de $N = 6000$ échantillons, et leur différence égale à

$$\Delta = \omega_N - \omega_S = 2\,\Omega\,\cos\theta = 2\,(360^\circ\,T^{-1})\,\cos\theta, \text{ avec la période } T = 24\times60^2 = 86\,400 \text{ s et } \Omega = 0.0042^\circ\text{/s, ou } 1^\circ \text{ en 240 secondes.}$$

[29] D. B. Plewes, Magnetic monitoring of a small Foucault pendulum, *Rev. Sci. Instr.*, **89**(6) :065112 (2018)

[30] 1 gauss (G) = 0.1 mT, l'unité du système CGS, est utilisée couramment pour les champs magnétiques faibles.

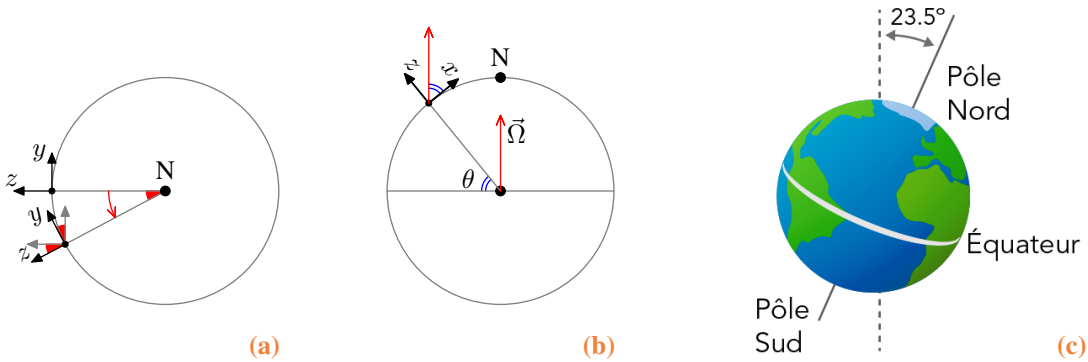


FIG. 14 – Rotation de la Terre et du gyroscope. Le vecteur $\vec{\Omega}$ représente la vitesse de rotation de la Terre ; le pesenteur $\vec{g}$ pointe vers le centre ; (x, y, z) marquent les axes du gyroscope. (a) La hémisphère nord avec deux positions successives du gyroscope sur l'équateur ; z est aligné avec $-\vec{g}$ et pointe «verticalement vers le haut», y donne le ouest, tandis que $\vec{\Omega}$ et x (orienté N) sont pointés vers nous. (b) Le gyro à la lattitude $\theta = 50.7°$ avec la même orientation des axes. (c) L'orientation de $\vec{\Omega}$ par rapport à l'orbite de la Terre autour du Soleil (axe horizontal).

Ici, l'origine du facteur $\cos\theta \leq 1$ parvient du fait, qu'à la latitude θ, le vecteur $\vec{\Omega}$ est incliné par rapport à la verticale : dans l'hémisphère N, l'angle entre $-\vec{g}$ et $\vec{\Omega}$ égale à $\frac{1}{2}\pi - \theta$, voir la fig. 14b. Ainsi $\Delta = 0.0053°/s$ à notre latitude $\theta = 50.768°$. Afin d'éviter ce facteur et de maximiser Δ nous pouvons soit (a) monter le gyro «en pente» d'angle θ orientée N-S et aligner son axe x avec $\vec{\Omega}$, soit (b) combiner les axes x et z. Dans le cas (b), on doit passer de N à S en tournant autour de l'axe y : $\mathrm{N} \rightarrow \mathrm{S} : (x, y, z) \mapsto (-x, y, -z)$, et on mesure

$$\Omega = |\vec{\Omega}| = \sqrt{\Omega_x^2 + \Omega_z^2}, \quad \text{où } 2\,(\Omega_x, \Omega_y, \Omega_z) = \big(\omega_{\mathrm{N}x} - \omega_{\mathrm{S}x},\ 0,\ \omega_{\mathrm{N}z} - \omega_{\mathrm{S}z}\big).$$

Notez que le bruit de gyroscopes MPU6050 et L3GD20H est estimé à 10 LSB, et nous devrons faire avec une incertitude $\sigma = 0.08°/s$ de chaque échantillon. Par conséquent, selon la «loi des grands nombres», nous obtendrons une $\sigma/\sqrt{N}$ pour chaque mesure ω, et $2\,\sigma/\sqrt{N} = 0.0002$ pour Δ. Ceci suffira clairement pour détecter sa valeur anticipée de 0.005, à condition, bien sûr, que la ligne de base de gyroscope n'évolue guère durant les 2 minutes d'enregistrement.

4.11.2 Indicateur de freinage style F1

On se propose d'utiliser un accéléromètre + gyroscope (avec, en option, un GPS) pour détecter, enregistrer et réagir à toutes accélérations du véhicule. Ainsi un clignotant rouge feux arrière peut être activé quand le véhicule décélère au frein moteur. Une autre application est un indicateur automatique des virages dans une moto où le pilote n'a pas le temps d'activer/désactiver le clignotant.

4.11.3 Pendule de *Pohl*

Le pendule de *Pohl* est un oscillateur de torsion (voir la fig. 16 et la sec. 4.3.1 du cours de vibrations) constitué de : (a) un disque assez massif en cuivre qui peut tourner autour de son centre par un angle θ avec $|\theta| \leq \theta_{\max}$; (b) un pointeur (une flèche blanche) placé sur le disque permettant de repérer l'angle de rotation θ sur le rapporteur ; (c) un ressort spiral, qui exerce un couple mécanique avec la tendance à ramener le disque vers sa position d'équilibre à $\theta = 0$; (d) un système de freinage électromagnétique (par les courants de *Foucault* et la force de *Laplace*) qui permet d'amortir les oscillations de façon douce et contrôlée ; (e) un moteur, relié au ressort par un levier. Lorsque le disque tourne d'un angle θ, le ressort s'enroule (ou se déroule), ce qui fait apparaître une force de rappel dont le moment mécanique s'oppose au moment du disque. Si on lâche le disque, le système retrouve son état d'équilibre en oscillant : c'est le régime d'oscillations *libres*. Le moteur permet d'exercer une force excitatrice sinusoïdale sur le pendule. dont la fréquence $\Omega/(2\pi)$ corresponde à la fréquence de rotation du moteur et peut être ajustée. Le système se met au régime sinusoïdal *forcé* avec la même fréquence.

Les oscillations libres et forcées du pendule de *Pohl* peuvent être étudiées en effectuant *deux* enregistrements simultanés : l'un pour les mouvements rotatifs du pendule et l'autre pour les mouvements de la poussée produite par le moteur (force externe). Au lieu des capteurs pour les déplacements angulaires θ, on utilise deux *gyroscopes* L3GD20H (Pololu 0J7997, sec. 3.5) pour mesurer $\dot{\theta}$. On les installe sur le dispositif TP PHYWE ref. P2132705. On y ajoute un *détecteur de ligne* QRE1113 (sec. 2.4.2) pour enregistrer les moments de passage du pendule à son point d'équilibre, et, en option, pour commencer l'enregistrement à ce moment precis (donc avec une phase initiale φ_0 de 0 ou π). Il est possible également de mesurer le courant dans la bobine de freinage Faucault avec un détecteur de courant DC à effet Hall ACS712, un plus récent ACS723, ou même un LTSR-6-NP. Enfin, on peut piloter le moteur électique[31] en PWM via un transisteur NPN ou un MOSFET de puissance capables de commuter 1 A, dont le cirquit est découplé, pour plus de sécurité, à l'aide d'un optocoupleur (voir la sec. 2.7 ainsi qu'un cirquit tout fait par RobotDYN[32]). Les enregistrements peuvent être traités sous LINUX (RASPBERRY) avec un script en `gnuplot` et `gawk`, et possiblement d'autres utilités standards, tels que `ps2pdf`.

[31] FAULHABER *Schöneich* 2230V024S (148) avec un reducteur 81:1 22B (298) à engrenage droit. Selon l'étiquette de PHYWE, le moteur requiert Antrieb 24 V 650 mA, Motorspannung an Prüfbuchsen 2−16 V, in Resonanznähe 8 V.

[32] à la base d'un N-channel MOSFET de puissance IRF540N (4.5-24 V (peak 36 V) 30 A 50 W, une diode Schottky intégrée) et le photocoupleur EL817.

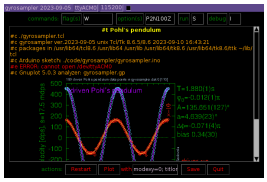
TP pendule de *Pohl* Matériel requis :

1. le pendule PHYWE ref. P2132705 soit (a) nu, (b) amenagé avec capteurs pour ARDUINO, ou (c) muni d'une grille ;
2. les étudiants ramènent leur propre clé USB pour sauvegarder leur résultats dans les cas (b) et (c) ;
3. alimentation fixe 24 V/1 A DC ou AC pour le moteur[31] ;
4. alimentation DC stabilisée $\leq$1 A et $0-10$ V pour la bobine de freinage *Foucault* ;
5. ampèremètre $0-1$ A DC (bobine) et voltmètre $0-24$ V (moteur) ;
6. (a) chronomètre pour les études manuels
7. (b) coffret avec une carte ARDUINO UNO pré-chargé avec le programme `gyrosampler` et fils connecteurs M/F ;
   (b) ordinateur-écran RASPBERRY PI3 muni de cables pour ARDUINO (USB-B) et la clé (USB-A) ;
8. (c) module GTI, photo-cellule, et un PC sous WINDOWS XP avec logiciels GTI et RÉGRESSI.

Montage :
mettre les sources d'alimentation à 0 V/0 A et les connecter ; brancher le voltmètre (sortie UX) ; brancher l'ampèremètre en série avec la bobine de freinage. Selon le dispositif : (b) Brancher ARDUINO à PI3 avec le câble USB B ; «à l'arrière» du pendule PHYWE, repérer une petite plaque portant 5 connecteurs *Dupont* M avec fils *jaune*, *vert*, *noir*, *rouge*, et *blanc* ; câbler le pin digitale DP2 au fil *blanc*, pins SCL et SDA (à l'aurte extremité de la même ligne de pins digitaux de la carte UNO, face aux LEDs) au fils *jaune* et *vert*, et GND et 5V (ligne opposée) aux fils *noir* et *rouge*. Vérifier que rien n'empêche les mouvements du pendule. (c) à préciser avec le prof
Expérience :

1. Observer les oscillations libres (sans amortissement) et déterminer la pulsation propre ω_0 du pendule.
   (a) Lancer le pendule, compter une dizaine de démi-périodes (passage de la flèche à 0 du rapporteur qui marque la position verticale d'équilibre) et mesurer le temps en lançant le chronomètre au premier passage à 0. Par la suite, selon le dispositif :
   (b) Sur le compte `arduino`, ovrir le «terminal» du logiciel `arduino` à la vitesse 115200 bps. Le μCU répond avec des informations sur son état. Envoyer la commande `W`. Décaler et lancer le pendule. La LED sur la carte marque chaque démi-période. Envoyer la commande `Z` et noter les informations.
   (c) Lancer le pendule et enregistrer ces mouvements avec le logiciel `GTI`. Exporter vers `Régressi` et réproduire les données avec la fonction théorique $\theta(t) = A\,\exp(-\gamma\,t)\,\sin(\omega_0\,t+\varphi)+\theta_0$ en ajustant ses paramètres. À voir avec le prof. et les TP L1.
2. Étudier les oscillations libres amorties et déterminer le coefficient γ pour différents courants I=0.25 à 0.5 A dans la bobine.
   (a) Lancer le pendule, repérer sur le rapporteur les amlitudes max/min A_i décroissantes par leur valeur absolue à chaque démi-période. Pour convertir les graduations du rapporteur en degrés, se référer à l'appendice 4.11.3A. Tracer une ligne droite dans les coordonnés $(i, \log(|A_i|))$ et déterminer son coeff. directeur. En trouver γ en [1/sec].
   (b) En sa version avancé, le programme `gyrosampler` détermine la quasi-période T, l'amplitude initiale A, et le coefficient d'amortissement γ après avoir échantillonné $\dot\theta(t)$ pendant au moins $2-3$ démi-périodes, voir 1(b). Cependant, la précision de ces estimations dépend du rapport de taux de l'échantillonnage et T. Pour obtenir les valeurs plus précises, et pour estimer les incertitudes, enregistrer et traiter la courbe $\dot\theta(t)$ comme expliqué dans l'appendice 4.11.3C.
   (c) suivre la même approche que dans même que 1(c)
3. Étudier les oscillations forcées. Allumer l'alimentation du moteur (24 V fixe). Régler la tension U_x aux bornes du moteur à l'aide de deux potentiomètres rotatifs «Grob» (réglage approximatif) et «Fein» (fin). Contrôler U_x au voltmètre. La pulsation des oscillations égale à $\Omega = 2\,\pi/T$, où T est la période de rotation du moteur. Pour faciliter les manipulations, on peut déduire la relation $T(U_x)$. À noter que, selon PHYWE, la condition de résonance $\Omega \approx \omega_0$ correspond à $U_x \approx 8$ V. Mesurer la pulsation Ω, l'amplitude $A(\Omega)$, et le déphasage $\varphi(\Omega)$ pour différents valeurs de $\Omega(U_x)$ dans l'intervalle $[0, 4\,\omega_0]$ et pour $2-3$ valeurs de $\gamma(I)$.
   (a) Estimer Ω en suivant l'approche de 1(a) ; mesurer T en chronométrant les tours du moteur (encodeur blanc sur sa roue, face «arrière», compter plusieurs tours) ; confirmer la relation entre T et Ω. Sur le rapporteur noir, repérer les points de retour $\theta_{\max} = -\theta_{\min}$ (plusieurs fois, prendre le moyen, estimer l'erreur) ; en déduire l'amplitude A (en degrés, voir 2(a)) correspondant à la fréquence Ω. Pour mésurer le déphasage, chronométrer le temps entre les passages consécutifs du pendule et de la roue encodeur du moteur à leurs repères respectifs (l'équilibre pour le pendule, où sa phase égale 0 ou π), et convertir en fraction de T.
   (b) Les estimations de la période T sont accessibles directement par la commande `Z`. Pour mesurer A et avoir plus de précision, enregistrer simultanément et traiter les données $\dot\theta(t)$ (pour les mouvements du pendule) et $\dot f(t)$ (pour la poussée du moteur) avec «flags» `FW` (voir l'appendice 4.11.3C). Tracer les courbes paramétriques de *Lissajous* $t \to \big(\dot\theta(t), \dot f(t)\big)$ et retrouver le déphasage φ par la méthode des «ellipses»[33], employée couramment dans les études des circuits éléctriques aux courants/tensions sinosoïdales.
   (c) avec GTI, l'impossibilité d'enregistrer $\theta(t)$ et $f(t)$ simultanément fait la mesure de déphasage irréalisable.
4. Représenter les résultats comme courbes de résonance[34] $A(\Omega/\omega_0)$ et de déphasage $\varphi(\Omega/\omega_0)$ pour différents ammortissements γ.

[33] Pour une ellipse $t \to \big(a\,\sin(\omega\,t),\ b\,\sin(\omega\,t+\varphi)\big) = (x, y)$ avec $a>0$ et $b>0$ les amplitudes de $x(t)$ et $y(t)$, on trouve $|\sin(\varphi)| = h/B$, où h donne son intersection supérieure avec l'axe y (vertical). Cette méthode *ne permet pas de déterminer le signe* de φ. Pour cela, il faut connaître la direction de parcours.
[34] voir l'exercice 4.3 de la section 4 du cours «Vibrations et ondes»



Appendice 4.11.3C : logiciels `gyrosampler` On utilise trois logiciels `gyrosampler` avec extentions `.ino`, `.tcl`, et `.gp` pour, respectivement, échantillonner (acquérir), collecter, et traiter les données. Par manque de dévéloppeurs et testeurs, les deux derniers (faisant appel aux autres utilitaires, tels que `awk`, `netpbm`, et `gs`) ne sont opérationnels en ce moment que sous LINUX. Facilement adaptatables pour MACOS, ou autres UNIX'es, leur exploitation sous WINDOWS reste problématique mais possible en principe.

`gyrosampler.ino` est un sketch en `c++` pour ARDUINO (voir la sec. F). Au plus simple, on peut l'installer dans le μCU et l'interroger avec l'outil habituel ARDUINO IDE (sec. 2.1.1) disponible sur tout plateformes. Son fonctionnement ressemble à celui de `adcsampler` (cf. sec. 2.5 et 4.1.3) : ouvrir et configurer le moniteur à 115200 bauds, envoyer des commandes au sketch, copier avec `Ctrl-A` les mesures qui apparaissent en réponse, et les coller (`Ctrl-V` ou souris) dans un fichier text `gyrosampler.dat` en utilisant tout éditeur de texte. Par la suite, ces données peuvent être traitées dans un tableur (tel que EXCEL ou RÉGRÉSSI sous WINDOWS) ou avec `.gp`. Autrement, on s'en sert de `.tcl` pour tout faire. Voici la liste des commandes.

Les cavaliers (flags) agissent comme des boutons d'un interrupteur on/off qui basculent leur paramètre $1/0$ à chaque fois :

`F` distingue entre les régimes *forcé* et *libre* avec l'acquisition des données de deux et un gyroscope(s), respectivement ;

`W` distingue le déclenchement immédiate et au premier passage à l'équilibre (voir l'action `S`) ;

Les options sont suivis par des valeurs numériques x (un nombre entier positif à plusieurs chiffres décimaux), qu'elles passent aux paramètres du scripte. Leur appel répétitive (avec les mêmes valeurs) n'affecte guère le fonctionnement du scripte :

`T`x définit l'intervalle `READ_PERIOD` entre les échantillons en μsec. Par default, `READ_PERIOD` est fixé à 20000, donnant le taux de 50 Hz, soit approximativement 100 points par la période du pendule ;

`N`x contrôle le nombre maximal des échantillons `npts` ;

`P`x définit le nombre maximal `pcnt` des démi-périodes à observer (si $x > 0$, voir l'action `S`) ;

Les actions sont des boutons-poussoir déclenchant une réponse :

`Z` affiche le compteur de passages à l'équilibre (c.p.é.), voir la sec. 2.4.2, estime la démi-période T, et remet le c.p.é. à zéro ;

`S` demarre l'acquisition après avoir mis le c.p.é. à zéro ; L'enregistrement commence soit immédiatement, soit au premier passage à l'équilibre (selon le flag `W`), et procède jusqu'à l'épuisement de `npts` (voir option `N`) ou `pcnt`, si ce dernier est positif (option `P`). Pour chaque échantillon, la/les valeur(s) de la vitesse angulaire reportée(s) par le/les gyroscope(s) en mdps est/sont affichées (option `F`) sous forme de nombres entiers avec le signe, suivie(s) par la valeur momentanée du c.p.é.

`I` affiche des informations sur la carte, le sketch, et ses paramètres actuels.

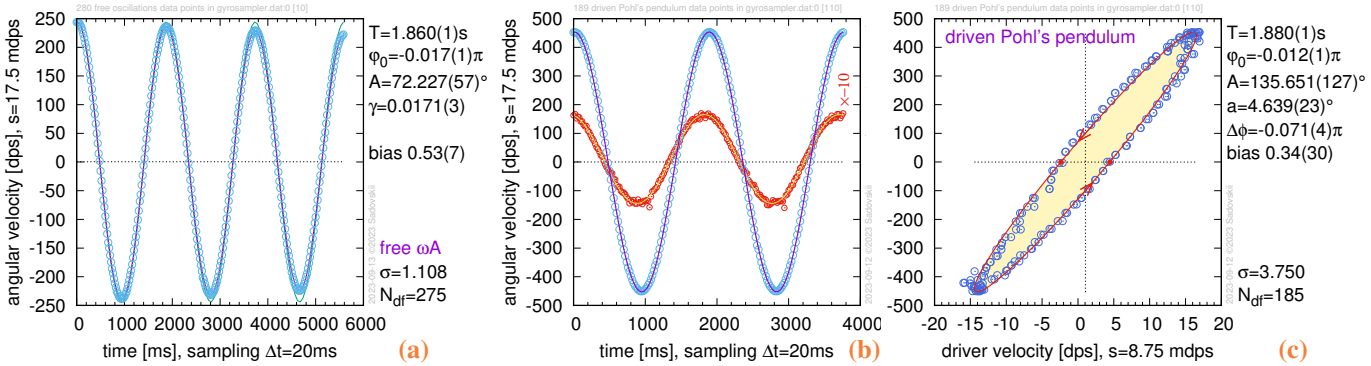


FIG. 15 – Pendule de *Pohl* : la présentation et l'analyse des mesures avec l'utilitaire `gyrosampler.gp` (scripte GNUPLOT) des oscillations (a) libres, (b) forcées, et (c) forcées sous forme de l'ellipse de Lissajous, dit «mode x-y». Les graphes représentent les données de gyroscopes `0x6a` et `0x6b` sur les vitesses $\mathrm{d}\theta/\mathrm{d}t$ du pendule et de la force avec les amplitudes respectives sont ωA et $\omega\, a$.

`gyrosampler.gp` est un scripte pour le logiciel graphique GNUPLOT. Pour l'executer, il souffit de saisir un terminal et lancer la commande `gyrosampler.gp`. Il analyse les données dans `.dat` et produit le fichier graphique `.pdf`. Les exemples de ce dernier sont illustrés dans la la fig. 15. La commande reconnaît quelques options `nom` sous forme `nom=val`, spécifiquement

titlon avec `val` = 0 ou 1 gère l'apparition du title sur le graphique ;

modexy avec `val` = 0 ou 1 sélectionne le mode d'affichage classique (avec l'axe de temps t, fig. 15a et 15b) ou, en régime forcé, le mode paramètrique dit «x-y», voir la fig. 15c ;

dset avec valeurs 0, 1, etc définit l'indice du jeu des données à analyser, les jeux sont separés par un double changement du ligne ;

scl (`.75`) ajuste l'hauteur du graphe dans le `.pdf` en unités de la taille de réference de 3 in sans contrôler les polices.

Ses options sont soumises au scripte avec `gyrosampler.gp -e "nom1=val1; nom2=val2"`.

`gyrosampler.awk` est un scripte auxiliaire en langage `awk` qui permet à `gyrosampler.gp` d'extraire des informations du fichier `.dat` issu typiquement de l'interaction avec `gyrosampler.ino`. Il est possible d'engager ce scripte (dans un terminal) par la commande `gyrosampler.awk gyrosampler.dat` et vérifier ainsi toutes ces informations.

`gyrosampler.tcl` est une appli TCL/TK assurant une interface graphique (GUI, voir l'entête de cet appendice) et une alternative à ARDUINO IDE (pour `.ino`) et `gyrosampler.gp`. Son utilisation est assez intuitive. Saisir une fenêtre de terminal et lancer la commande `gyrosampler.tcl`. Au démarrage, l'appli tente de connecter à la carte ARDUINO. Si la connexion est établie, les «commandes» à `.ino` peuvent être soumises, et les réponses sont montrées dans la fenêtre principale de l'appli. Par la suite, les «actions» sont proposées. Ainsi «Plot» engage `gyrosampler.ino` et «Save» permet de sauvegarder les fichiers `.dat` et `.pdf` actuels sur une clé usb (si présente) ou dans le répertoire courant (`./`) avec leur nom prolongé par une indice unique. A noter, que s'ils ne sont pas effacés explicitement avec «Clear», les résultats des appels successives à `.ino` sont cumulés et sont exploités ensemble par «Plot» avec l'indice défaut de jeu des données (dataset `dset`) correspondant au résultat le plus récent.

Appendice 4.11.3A : calibrer la plage de sensibilité de gyroscopes Le gyroscope L3GD20H possède *trois* plages de sensibilité (*full scale* ou FS) : ± 250 (défaut L3G), ± 500, et ± 2000 dps (degrés/sec). Les amplitudes de la poussée de moteur sont faibles et on garde ± 250 pour le gyroscope `0x6b`. L'amplitude maximale $A_{\max} = \theta_{\max}$ du pendule même (gyroscope `0x6a`) et limitée à 20 grandes divisions (g.d.) de circle rapporteur du dispositif PHYWE P2132705. On remarque que 10 g.d. (l'angle entre deux rayons du corps de pendule en cuivre) correspondent à $1/5$ me du cercle complet. Par consequent,

$$\frac{2\,\pi}{5}\,\text{rad} = 10\,\text{unités g.d.} \Rightarrow \frac{\pi}{25}\,\text{rad} = 7.2^\circ = 1\,\text{g.d.}, \quad \frac{\pi}{5}\,\text{rad} = 36^\circ = 5\,\text{g.d.}, \quad \text{et} \quad A_{\max} = 20\,\text{g.d.} = 144^\circ.$$

La periode T_0 des oscillations libres du pendule et sa pulsation propre ω_0 sont, respectivement, ≈ 1.8 s et $2\,\pi/1.8$. C'est autour de la fréquence $1/T_0$, dans les conditions de résonance, qu'on anticipe les grosses amplitudes de θ et $\dot\theta$. Ainsi pour $\dot\theta$ maximale on estime

$$\dot\theta_{\max} = A_{\max}\,\omega_0 = 144\,\frac{2\,\pi}{1.8} = 160\,\pi = 502.65\,\text{dps}.$$

Par conséquent, il faut choisir la plage FS $= \pm 500$ dps pour le gyroscope «avant» `0x6a` (indice `0` dans `gyro[]`) et les deux bits FS1-FS0 de son registre REG4 `0x23` (sec. 7.5, pp. 39-40 du datasheet de L3GD20H) doivent être mis à 0-1 (au lieux de 0-0 par défaut).

```
gyro[0].writeReg(L3G::CTRL_REG4,0x10); // FS=500
```

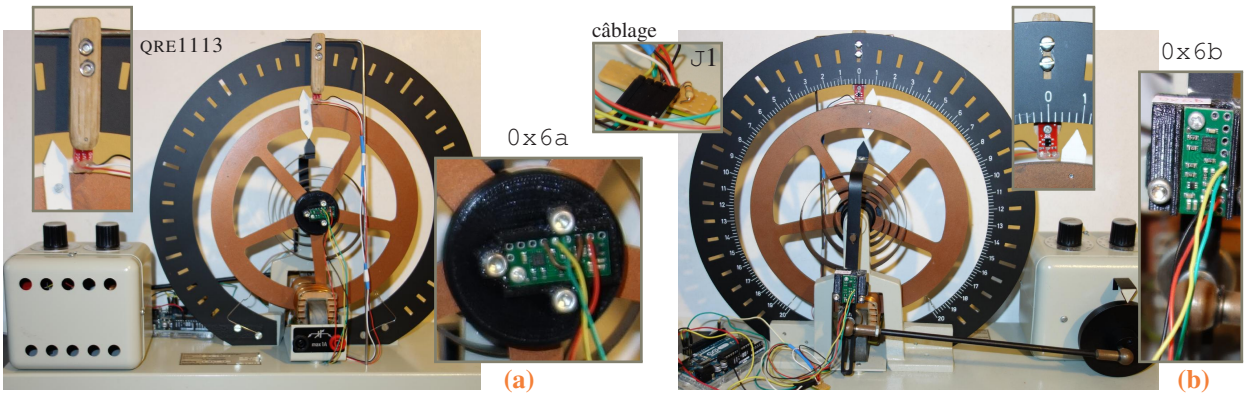


FIG. 16 – Pendule de *Pohl : montage des capteurs à l'avant (a) et à l'arrière (b) du dispositif* PHYWE P2132705, voir l'appendice 4.11.3B.

Appendice 4.11.3B : montage des capteurs

- Souder quatre fils (AWG30 souples multibrins 0.05 mm^2 avec isolant en silicone) de longueur 25 à 30 cm à chaque carte gyroscope L3GD20H : *jaune* à SCL (horloge du bus I2C), *vert* à SDA (données I2C), *noir* à GND (masse), et *rouge* à VIN (tension d'entrée V_{cc}).
- Les fils doivent rester libre afin de ne pas empecher les mouvements du pandule et de la poussée. Pour organiser le passage des fils et les rassembler «à l'arrière», où on placera une petite plaque de montage, mettre une gaine 3/32 thermorétractable 2.4 → 1.2 mm de 3 cm.
- Sur une des cartes L3GD20H, mettre (souder) un cavalier entre SDO et GND pour lui attribuer l'adresse I2C `0x6a`. Cette carte sera destinée à mesurer les mouvements du pendule et sera dénommée «avant» (AV) et/ou par son adresse. L'autre carte sera destinée à mesurer les (petites) poussées du moteur et sera dénommée «arrière» (AR) et/ou par son adresse `0x6b` (voir la fig. 16).
- Souder trois fils de ≈ 45 cm à la carte QRE1113 (détecteur IR d'obstacles) : *noire* à GND, *blanc* à OUT, et *rouge* à VCC. Protéger ces soudures par la colle thermique pour éviter un court-circuit accidentel provoqué par des contactes de la carte avec les parties métalliques du pendule.
- Sur une petite plaque (breadboard) de montage de 3×4 cm (4×12 trous, la plaque peut être élargie de 1 cm de chaque côté pour y accommoder deux trous de fixation de diamètre 2 mm), souder 5 connecteurs Dupont coudé males pour SCL, SDA, GND, VCC, et VOUT (sortie QRE1113). Y ramener, respectivement, deux fils jaunes, deux verts, trois noires, trois rouges, et le fil blanc. Noter que 2 fils AWG30 peuvent être insérés et soudés ensemble dans un seul trou de la plaque. Souder une résistance de 10 K entre VCC et VOUT. Protéger les soudures par la colle thermique.
- Descendre les fils de QRE1113 le long du rapporteur côte «avant» (serres câbles nylon) ; ranger ces fils et les fils du gyroscope AV `0x6a` au-dessous de la bobine de l'aimant éléctrique (gaine) avant de les ramener tous vers «l'arrière» à la plaque de montage.
- Imprimer les supports de deux gyroscopes «avant» et «arriere» avec sa couverture (fichiers `stl` par *Pierre Kulinski* et *Pascal Masselin*). Pour leur fixation, utiliser 3 vis M3.20 (av) et 2 mêmes vis avec 2 écrous M3 (ar). Pour installer les cartes L3GD20H dans leur supports, utiliser 2 vis M2.5 avec leurs ecrous. Alternativement, remplir les trous avec la colle thermique et utiliser 2 vis bois VB2.5 (diamètre 2 mm, longueur 5 mm).

- Fabriquer (imprimer) le petit support de la carte QRE1113. Utiliser 2 vis M3.10 avec ecrous pour l'installer dans la fente du rapporteur noir au point 0 g.d. (l'équilibre en haut) de côté «avant» (côté flèche blanche sur la fig. 16). La carte même s'y fixe avec une vis VB2.5. Vérifier que la distance entre QRE1113 et la flèche reste de $2-3$ mm, et que le pendule peut osciller sans empêchement (voir la fig. 16).
- Installer un «étai» du rapporteur de côté «avant» afin de gâcher tout ses vibrations latéraux (devenant très nuisibles et provocant fausses signaux de détecteur QRE1113, surtout pour les grandes amplitudes en régime forcé proche de la résonance). L'étai constitue l'hypoténuse de $\approx$28.5 cm d'un triangle de 5×28 cm avec le rapporteur. Il fait l'angle de 80° avec la base, et se fixe par insertion (eventuellement collée) en haut dans le trou horizontal du support de QRE1113, et en bas, dans le trou percé verticalement dans la plateforme-base (en bois MDF) du pendule PHYWE. Son bout inférieur de 15 mm est incliné donc legérement de 10°, tandis que son bout supérieur de 5-6 cm est horizontal et perpendiculaire au triangle. Dans notre prototype, l'étai a été fabriqué en fil d'alliage alu assez rigide de diamètre 2.5 mm et de longueur $\approx$36 cm. Après son insertion, il est déformé légèrement afin d'obtenir la distance optimale de $2-3$ mm entre le détécteur QRE1113 et la flèche blanche.

4.12 PCR et qPCR thermal cycler (DS)

De nombreux tests modernes de diagnostic moléculaire ciblant les acides nucléiques sont généralement limités aux pays développés ou aux laboratoires de référence nationaux des pays en développement. La capacité de rendre les technologies de diagnostic rapide des maladies infectieuses dans un format portable et peu coûteux constituerait une avancée révolutionnaire dans le domaine de la santé mondiale. De nombreux tests moléculaires moléculaires sont également élaborés sur la base de réactions en chaîne par polymérase (PCR), qui nécessitent des thermocycleurs relativement lourds (de l'ordre de 10 à 20 kg) et nécessitant une une alimentation électrique continue. La vitesse de montée en température de la plupart des thermocycleurs les plus économiques est relativement lente (2 à 3 °C/s), de sorte qu'une réaction en chaîne par polymérase peut prendre 1 à 2 heures. Par-dessus tout, ces thermocycleurs sont encore trop chers (de 2 000 à 4 000 dollars) pour être utilisés dans des environnements à faibles ressources.

Les projets OpenPCR et PS-PCR sont déjà très développés et testés, il y a une doc très détaillée pour les reproduire et tout le logiciel nécessaire (y comprit côté PC où on les branche via port usb). Comme cerise sur le gâteau, on pourrait peut être y ajouter un pilote par iPhone... évolution vers la détection en temps réel avec fluorescence voir Open qPCR alternative robotique à l'élément Peltier, voir rapid and low-cost PCR thermal cycler for low resource settings

En bref : il s'agit de réplication de gènes (DNA). À la base, c'est un bon élément Peltier piloté par Arduino. On mesure/contrôle aussi la température et (en option) la fluorescence. Il y a une grande partie biologique et un peu de cinétique chimique. Les tests sont à faire en collaboration avec les biologistes (Jean-Christophe Devedjan)

4.13 Capteur déplacement/vitesse (DS)

On récupère une vieille souris pour connecter ces capteurs au microcontrôleur Arduino et en faire un capteur de déplacement linéaire (1D et/ou 2D), angulaire (rotation), vitesse, et accélération. Autrement, on utilise le détecteur infrarouge QRE1113 ou une photocellule décrits dans la sec. 2.4.2. Dans le cas d'une photocellule ou «photogate», l'idée est de reproduire le système «picket fence». Ceci est utilisé pour enregistrer les déplacements linéaires dans les anciens TP oscillations (GTI+RÉGRESSI) Voir le prototype et la théorie. Il y a des nombreuses applications dans les labos de TP, notamment oscillations et autres TP L1 de mécanique, pendule de Pohl (ancien TP L3, sec. 4.11.3), goniomètre (L1 optique), mesures en 2D, robotique.

4.14 Anémomètre et girouette à effet Doppler (R. Bocquet)

Le but du projet est de réaliser un anémomètre à effet Doppler. Il s'agit d'utiliser cet effet bien connu, dans la gamme des ultrasons, pour mesurer la vitesse du vent ainsi que sa direction. Cette gamme de sons se situe à des fréquences au-dessus des 20 kHz, limite audible de tout être humain normalement constitué. Vous aurez à votre disposition des émetteurs et des récepteurs accordés certainement à 30 kHz mais qu'il faudra vérifier. L'idée de la mesure est assez simple : l'onde sonore est une onde mécanique portée par l'air, la vitesse de propagation dépendra donc de la vitesse de l'air. Vous devrez :

- comprendre le principe physique de l'effet Doppler dans l'air
- mettre en évidence cet effet dans une expérience
- proposer un montage permettant de donner également la direction
- proposer un montage (électronique + ARDUINO) pour mesurer des vents allant de 0,1 noeud à 50 noeuds[35]
- réaliser le montage si vous avez le temps et le tester.

Ne négligez pas le travail préparatoire dans ce projet.

4.15 Cinémomètre à GPS (R. Bocquet)

Le but du projet est de réaliser un instrument de la taille d'une grosse montre pour mesurer la vitesse et la direction de déplacement d'un bateau à voile ou d'une bicyclette. Pour cela on vous propose d'utiliser un composant (u-blox 6 GLONASS GPS) qui n'est autre qu'un GPS et un tout petit écran de visualisation de 4 caractères en bus I2C. On utilisera le format de données NMEA, très utilisé dans l'industrie du nautisme et disponible sur le GPS que vous avez à votre disposition. Vous devrez :

[35] 1 noeud = 1 mile nautique par heure

- réaliser le montage GPS et visualisation avec carte ARDUINO
- réaliser le montage GPS et visualisation sur écran
- réaliser le montage avec un micro-contrôleur Atiny 45 ou 85

* développer une carte électronique autonome (pile 3,3 V bouton) et son boitier

4.16 Optimisation de la consommation d'un micro-contrôleur ATtiny 85 (R. Bocquet)

Les microcontrôleurs que vous utilisez ont la possibilité de fonctionner avec de très faibles consommations et sous des tensions de 3,3 V. Il s'agit dans ce projet de mettre en place une programmation d'un micro-contrôleur ATtiny 85 qui est un composant électronique enfichable sur la plaquette d'essais et de faire un montage permettant de mesurer la consommation du composant. Ce projet demande des notions d'électronique et de très bonnes connaissances de la langue anglaise. En effet vous devrez lire la notice du micro-contrôleur et programmer directement les ports du micro-contrôleur.

4.17 Centrale météorologique – 2 groupes de travail (R. Bocquet)

Il s'agit de mettre en place une centrale de mesure météorologique avec un boîtier extérieur embarquant les capteurs et un boîtier intérieur pour réception, traitement et stockage des données. Les deux boîtiers seront reliés par une transmission radiofréquence à 433 MHz. Dans le cas où un seul groupe est constitué, chaque partie peut -être traitée séparément.

Groupe 1 : capteurs et émetteur. réaliser et mettre au point un montage permettant à minima de mesurer la température, la pression, l'humidité relative et la luminosité. Vous définirez un protocole de données et mettrez en place une liaison RF pour transmettre les données. Vous étudierez les possibilités de transfert sur carte de votre montage en utilisant un micro-contrôleur ATtiny 45 ou 85.

Groupe 2 : récepteur. réaliser et mettre au point un récepteur des données météo du groupe 1 avec un moyen de sauvegarde. Vous programmerez une carte ARDUINO pour visualiser sur un écran LCD graphique les données et développerez un code pour réaliser une prévision météo, fonction des données enregistrées.

Groupes 1 et 2 : liaison RF. établir la liaison RF entre les 2 boîtiers et tester les distances possibles de transmission, ainsi que les difficultés qui pourraient nuire à la qualité de la transmission.

A Branchement de LED's et barographe `~/Arduino/BlinkSOS/BlinkSOS.ino`

```
/*
  SOS signal Blink (a derivative of Blink in the Arduino example library)
  Repeatedly turn the onboard LED on/off for 200 msec three times,
  then for 500 msec three times, and then again 200 msec for three times.

  Most Arduinos have an on-board LED you can control. On the UNO, MEGA and ZERO
  it is attached to digital pin 13, on MKR1000 on pin 6. LED_BUILTIN is set to
  the correct LED pin independent of which board is used.
  If you want to know what pin the on-board LED is connected to on your Arduino model, check
  the Technical Specs of your board  at https://www.arduino.cc/en/Main/Products

  This example code is in the public domain.

  modified 8 May 2014  by Scott Fitzgerald
  modified 2 Sep 2016  by Arturo Guadalupi
  modified 8 Sep 2016  by Colby Newman

  modified  3 Aug 2017  by Dima Sadovskii
  modified 18 Nov 2021  by Dima Sadovskii
*/

// NB: use a PWM (~) pin to see analog fading effect, pin 8 is not suitable
#define PIN_BASE 9  // starting digital pin number for 4 external LED's
#define PIN_ANLG A0 // pin number for digital entry
// the setup function runs once when you press reset or power the board
void setup() {
  int j,i;
  // initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);
  // initialize 4 external digital pins starting in sequence from PIN_BASE as output
  for(j=PIN_BASE,i=4; i--; pinMode(j++, OUTPUT));
  // light the LED's in a sequence
  for(j=PIN_BASE,i=4; i--; j++) {
    digitalWrite(j, HIGH);   // turn the LED on (HIGH is the voltage level)
    delay(200);              // wait for a 200 milliseconds
```

```
    digitalWrite(j, LOW);    // turn the LED off by making the voltage LOW
    delay(200);
  }
  // Open serial communications and wait for port to open:
  Serial.begin(9600);
  while (!Serial) {
    ; // wait for serial port to connect. Needed for native USB port only
  }
  // send an intro:
  Serial.println("\nSOS blinking and Serial interaction via keys");
  // Serial.println();
}

// parse a single byte character key as a hexadecimal digit
unsigned int hexToByte (char c) {
  if ( (c >= '0') && (c <= '9') ) return (int)(c - '0');
  if ( (c >= 'A') && (c <= 'F') ) return (int)(c - 'A')+10;
  if ( (c >= 'a') && (c <= 'f') ) return (int)(c - 'a')+10;
}

// issue a triple blink on pin unless the analog input on PIN_ANLG is non-zero
// or there is a byte waiting on the serial input line to be examined
void blink3(int d, int dd, int pin) {
  int i;
  for(i=3; i; i--) {           // repeat three times
    if(analogRead(PIN_ANLG) || Serial.available()) return;
    digitalWrite(pin, HIGH);   // turn the LED on (HIGH is the voltage level)
    delay(d);                  // wait for d milliseconds
    digitalWrite(pin,  LOW);   // turn the LED off by making the voltage LOW
    delay(d);
  }
  delay(dd);                   // wait for extra dd msec
}

int aold=-1; // last read analog input

// Arduino runs this loop function over and over again forever
void loop() {
  int i,j,hex,aval=0;
  char key;
  aval = analogRead(PIN_ANLG);
  if(aval>0) {                          // deal with nonzero analog input
    Serial.print(aold);
    Serial.print(" V=");
    Serial.println(aval);
    aold = aval;                        // keep track of last analog input
    for(i=4; i--; digitalWrite(i+PIN_BASE,(aval>=i*256)?HIGH:LOW));
  }
  else { // show SOS signal
    if(aold>0) {
      aold=0;
      Serial.println("stop analog entry");
      for (i=255 ; i >= 0; i -= 5) {     // fade out from max to min in increments of 5 points (range from 0 to
          255):
        analogWrite(PIN_BASE, i);        // NB: only possible for a PWM pin, e.g. 9 (but not 8!)
        delay(20);                       // wait for 20 msec to see the dimming effect
      }
      //digitalWrite(PIN_BASE,LOW);
    }
    blink3(200, 250, LED_BUILTIN);  // letter S
    blink3(500, 250, LED_BUILTIN);  // letter O
    blink3(200, 500, LED_BUILTIN);  // letter S and final pause
  }
  // Read serial input:
  if(Serial.available() > 0) {
    key = Serial.read();
    Serial.print("key ");
    Serial.print(key);
    Serial.print(" with ASCII ");
    Serial.print(int(key));
    Serial.print("=0x");
    Serial.print(key,HEX);
    if(isHexadecimalDigit(key)) {     // use isDigit for decimal input
      //if Hexadecimal display the numerical value of the key
      Serial.print(" gives HEX value ");
      Serial.print(hex=hexToByte(key));
```

```
      Serial.print(" and BIN code ");
      for(i=8,j=PIN_BASE+4;i>0;i=i>>1) {
        Serial.print(((hex&i)?1:0));
        digitalWrite(--j,((hex&i)?HIGH:LOW));
      }
    }
    Serial.println();
  }
}
```

B Échantillonnage de ADC ~/Arduino/adcsampler/adcsampler.ino

Programme pour échantillonner l'ADC en temps réel sans interruptions horloge ni bufferisation (envoi direct sur le port série)

```
/* -*- mode: c++; coding: latin-1; current-input-method: latin-1-prefix; ispell-local-dictionary: "english";
    fill-column: 79; comment-column: 0; eval: (local-set-key "\C-c\C-\M-u" 'browse-url-firefox); -*-
  This simple precise fixed time interval ADC sampler
  can be used for periods greater than 300 usec (see code, safer 500 usec)
  and thus has maximal theoretical sampling rate of 3kHz (more like 2kHz)
  see https://playground.arduino.cc/Interfacing/LinuxTTY on interfacing
  with linux serial port, which is normally /dev/ttyACM0
  stty -F /dev/ttyACM0 cs8 115200 ignbrk -brkint -icrnl -imaxbel -opost -onlcr -isig -icanon -iexten -echo -
      echoe -echok -echoctl -echoke noflsh -ixon -crtscts
  stty -F /dev/ttyACM0 115200 cs8 cread clocal
  screen /dev/ttyACM0 115200
*/
#define PIN_ANLG A0 /* pin A0..A5 for analog 5V max 10-bit ADC entry */
unsigned long READ_PERIOD = 4000;  // 4000 us gives 250 Hz sampling rate
unsigned long lastRead=0;
unsigned int npts=0; // number of samples

void setup() {
  /* Open faster serial communication (instead of usual speed 9600bps) */
  Serial.begin(115200); // 115200 bps = 14400 bytes/sec, 70 usec/byte
  while (!Serial) {
    ; /* wait for serial port to connect (for native USB port only) */
  }
  Serial.println("# fixed time interval ADC sampler");
}

void loop() {
  static unsigned int ncnt=0;
  static char key='Q';
  static long val=0;
  static char hex_format=0;
  char r;
  while(Serial.available()) { // read settings
    r = Serial.read();
    if(isDigit(r)) {
      val*=10;
      val+= r-'0';
    }
    else {
      switch(key) {  // keys that preceed numerical values
        case 'T':    // set sampling period (usec)
          if(val) READ_PERIOD = val;
          break;
        case 'N':    // set number of samples
          if(val) npts=val;
      }
      val=0;
      key=r;
      switch(key) {  // switches
        case 'H':    // hex format toggle
          hex_format^=1;
          break;
        case 'S':    // start sampling
          ncnt=npts;
        default:
          Serial.print("# period=");
          Serial.print(READ_PERIOD);
          Serial.print(" usec, samples=");
          Serial.print(npts);
          Serial.print(" at ");
```

```
          Serial.print(1000000.0 / READ_PERIOD);
          Serial.print(" Hz");
          Serial.println();
        case 'N':
        case 'T':
        case 'Q':
          break;
      }
    }
  }
  if(ncnt) { /* on 16MHz boards the time resolution is 4 usec, overrun in approx 70 min */
    for(ncnt=0, lastRead=micros()+8; ncnt<npts; ){  /* acquire and display npts samples */
      while(micros()<lastRead);
      lastRead += READ_PERIOD;
      val=analogRead(PIN_ANLG);  // 0..5V->0..1024 takes about 100 usec (10kHz)
      if(hex_format) Serial.println(val,HEX); // at least 200 usec for 3 bytes
      else Serial.println(val);
      ncnt++;
    }
    ncnt-=npts;
  }
}
```

C Colorimètre ~/Arduino/cmeter/cmeter.ino

```
/*
  Simple precise fixed-time-interval ADC sampler used as colorimeter;
  can sample at periods larger than 300 usec (safer 500 usec, see code)
  and thus has maximal theoretical sampling rate of 3kHz (more like 2kHz)

  TP: examine the relationship between the absorbance and concentration
  of a copper (II) sulfate CuSO4 solution (Beer's law) and measure
  the concentration of unknown copper (II) sulfate solution samples.
  The molar absorptivity of CuSO4 at lambda_max=635 nm is 2.81/M/cm.
  see more in http://dvoirah.ovh/etudes/Arduino/BQE-Arduino.pdf

  minimum parts required (in addition to Arduino UNO and computer):
    generic red LED (around 630 or 660 nm), 220 and 10k Ohm resistors
    photoresistor (3.1k Ohm exposed by red LED, closed at 0.35M Ohm)
    the LED-photoresistor distance on the breadbord is about 15mm
    use Arduino's 3.3V supply as VCC (more stable) to power the photoresistor
    use generic colorimeter 12.55x12.65x44.55 plastic cuvettes (with caps)
    for CuSO4 standard solutions of n/10 M with n=0,1,2,3,4,5 and 2-3 controls

  On interfacing with linux serial port, normally at /dev/ttyACM0, see
  https://playground.arduino.cc/Interfacing/LinuxTTY
  stty -F /dev/ttyACM0 cs8 115200 ignbrk -brkint -icrnl -imaxbel -opost -onlcr -isig -icanon -iexten -echo -
      echoe -echok -echoctl -echoke noflsh -ixon -crtscts
  stty -F /dev/ttyACM0 115200 cs8 cread clocal
  screen /dev/ttyACM0 115200

  When doing analog readings, especially with a 'noisy' board like the Arduino,
  we suggest two tricks to improve results. One is to use the 3.3V voltage pin
  as an analog reference, and the other is to take several readings in a row
  and average them. The 3.3V goes through a secondary filter/regulator stage
  and is less noisy to use it, connect 3.3V to AREF (next to digital GND).
  Several measurements can be averaged in time-independent or slow systems.

*/
#define _DEBUG_                   /* debugging, undefine to save space */
#include <EEPROM.h>               /* for nonvolatile memory access */

#define ADC_PIN A0                /* pins A0..A5 for analog 10-bit ADC */
#define LED_PIN 10                // digital pin for controlling LED source
// NB: use a PWM (~) pin to use analog fading effect, e.g. pin 8 is not suitable
// TODO: use PWM to control LED intensity
#define DIVISOR_R 10000           // 10K Ohm resistor in series with photo-R
#define STD_MAX 8                 // max number of calibration standards (>=2)

/* TODO: keep calibration coeffs and settings in nonvolatile memory */
double a1_coeff=.1;               // linear regression coefficients
double a0_coeff= 0;
double linreg_stderr=0;           // error of linear regression approximation
```

```
unsigned int  npts = 0;            // number of samples to measure
uint8_t apts = 2;                  // number of samples for averaging each point
unsigned long READ_PERIOD = 4000;  // 4000 us gives 250 Hz sampling rate

unsigned long lastRead = 0;        // time in us of the durrent point sequence
uint8_t hex_format=0;              // default raw integer output format
uint8_t std_num=0;                 // num of calibration data points (standards)
unsigned int std_C[STD_MAX], std_R[STD_MAX];

double Rval(unsigned int adc) {    // ADC data -> resistance of photo resistor
 return( DIVISOR_R / (1024 / ((double) adc) - 1) ); /* 10K / (1023/ADC - 1) */
}

double Cval(unsigned int adc) {    // ADC data -> concentration
 return(a1_coeff * ((double) adc) + a0_coeff);
}

double get_stderr() {              /* compute std error of linear regression */
 unsigned int *x=std_R, *y=std_C;  /* for the current calibration data set   */
 int k; double s=0,f;
 if(std_num>1) {                   /* check if there's enough data */
   for(k=std_num; k--; s+=f*f) f = Cval(*x++) - *y++;
   s /= (double) std_num;          /* unshifted normalized standard error   */
 }
 return(sqrt(s));                  /* sqrt( (sum_i (f(x_i) - y_i)^2) / N ) */
}

double linreg() {                  /* linear regression of current std data */
 unsigned int *x=std_R, *y=std_C;
 int k; double xx,_y,_x,xy;
 if(std_num>1) {                   /* check if there's enough standard data */
   for(k=std_num, _x  = _y = xx = xy = 0; k--; _x += *x++, _y += *y++) {
     xx += ((double) *x) * *x;                /* average of x^2 */
     xy += ((double) *x) * *y;                /* average of x*y */
   }
   a1_coeff = (xy*std_num - _x*_y) / (xx*std_num - _x*_x); /* slope of y(x) */
   a0_coeff = _y - _x * a1_coeff;
   a0_coeff/= (double) std_num;               /* shift of y(x) */
   linreg_stderr = get_stderr();              /* resulting std error */
  /* TODO: store calibration coeffs in nonvolatile memory */
 }
 return(linreg_stderr);
}

unsigned long min_val=0, max_val=1023;         /* 10-bit ADC value bracket */
void setup() {
  /* LED_BUILTIN is set to the correct LED digital pin for the board in use;
     it is 13 on the UNO, MEGA and ZERO, 6 on MKR1000. */
  pinMode(LED_BUILTIN, OUTPUT);
  digitalWrite(LED_BUILTIN, HIGH);  // turn the onboard LED on = "busy"
  /* set up faster serial communication (instead of default 9600bps) */
  Serial.begin(115200); // 115200 bps = 14400 bytes/sec, 70 usec/byte
  /* determine actual bracket of ADC values with light source on/off */
  pinMode(LED_PIN, OUTPUT);          // LED light-source controlling pin
  digitalWrite(LED_PIN,  LOW);       // make sure the LED is off
  delay(50);                         // wait for a 50 msec and measure
  max_val = analogRead(ADC_PIN);     // Vmax for maximum resistance (dark)
  digitalWrite(LED_PIN, HIGH);       // turn the LED on
  delay(200);                        // wait for a 200 msec and measure
  min_val = analogRead(ADC_PIN);     // Vmin for minimum resistance (max light)
  while (!Serial) {
    ; /* wait for serial port to connect (for native USB port only) */
  }
  // connect AREF to 3.3V and use that as VCC, because it is less noisy!
  analogReference(EXTERNAL);
  /* TODO: restore calibration coeffs and settings from nonvolatile memory */
  Serial.print("# real time colorimeter ");
  Serial.print(min_val);
  Serial.print("..");
  Serial.print(max_val);
#ifdef _DEBUG_
  Serial.print(" (debug)");
#endif
  Serial.println();
  digitalWrite(LED_BUILTIN, LOW);   // onboard LED off (end of setup)
}
```

```
void loop() {
 static unsigned int ncnt=0;
 static uint8_t acnt=0;
 static char key='Q';
 static unsigned long val=0,sval=0;
 char r,cflag=0;
  while(Serial.available()) { // read settings
    r = Serial.read();
    if(isDigit(r)) {
      val*=10;
      val+= r-'0';
    }
    else {
      switch(key) {  // keys that preceed numerical values
        case 'T':    // set sampling period (usec)
          if(val) READ_PERIOD = val;
          break;
        case 'D':    // concentration value for calibration data (standards)
    std_num=0; // reset calibration data storage
  case 'd':    // prepare new calibration point
    if(std_num >= STD_MAX) std_num--; // prevent storage overflow
    std_C[std_num] = val;             // enter calibration data y(x) >= 0
    std_R[std_num] = 0;               // clear x data (raw ADC values)
    cflag = 1;                        // calibration flag on
          break;
        case 'A':    // set number of averaged points
          if(val) apts = val;
          break;
        case 'N':    // set number of samples
          if(val) npts = val;
          /* TODO: store READ_PERIOD and/or npts to nonvolatile memory */
      }
      val=0;
      key=r;
      switch(key) {  // switches
        case 'h':    // set hex format
    hex_format =0;
        case 'H':    // toggle hex format (legacy)
          hex_format^=1;
          break;
        case 'c':    // concentration format toggle
          hex_format^=2;
          break;
        case 'r':    // resistance format toggle
          hex_format^=4;
          break;
        case 'C':    // (re)calibrate using currently stored data
          if(std_num>1) {
      Serial.print("# N=");    // number of points (>=2)
      Serial.print(std_num);
      Serial.print(" sigma="); // standard error
      Serial.print((unsigned int) floor(linreg()+.5));
      Serial.print(" a0 = ");  // shift
      Serial.print(a0_coeff);
      Serial.print(" a1 = ");  // slope
      Serial.print(a1_coeff);
      Serial.println();
#ifdef _DEBUG_
      for(acnt=0; acnt < std_num; acnt++) {
        Serial.print("# (V,C)_");
        Serial.print(acnt+1);
        Serial.print(" = (");
        Serial.print(std_R[acnt]);
        Serial.print(",");
        Serial.print(std_C[acnt]);
        Serial.print(") -> ");
        Serial.print((int) round(Cval(std_R[acnt])-std_C[acnt]) );
        Serial.println();
      }
#endif
    }
    else
      Serial.println("# Insufficient data for calibration");
          break;
  case 'X':    // toggle the LED on/off by bringing the voltage level HIGH/LOW
```

```
    digitalWrite(LED_PIN, !digitalRead(LED_PIN));
    delay(50);                      // wait for a 20 milliseconds
    break;
        case 'S':    // start sampling by rewinding npts
          ncnt=npts;
    acnt=apts;
    sval=0;
    if(!digitalRead(LED_PIN)) {     // make sure the LED is on
      digitalWrite(LED_PIN, HIGH);  // turn the LED on
      delay(50);                    // wait for a 50 milliseconds
    }
        default:
          Serial.print("# T=");
          Serial.print(READ_PERIOD);
          Serial.print(" usec, samples=");
          Serial.print(npts);
          Serial.print("/");
          Serial.print(apts);
          Serial.print(" @ ");
          Serial.print(1000000.0 / READ_PERIOD);
          Serial.print(" Hz, err=");
    Serial.print((unsigned int) floor(linreg_stderr+.5));
          Serial.println();
        case 'D':
        case 'd':
        case 'A':
        case 'N':
        case 'T':
        case 'Q':
          break;
      }
    }
  }
  if(ncnt) {
    /* acquire and display npts samples. ATTN: on 16MHz boards, the time
       resolution is 4 usec, and ncnt is overrun in approx 70 min */
    digitalWrite(LED_BUILTIN,HIGH);// onboard LED on = "busy"
    for(sval=ncnt=0, acnt=apts, lastRead=micros()+8; ncnt<npts; ncnt++){
      while(micros()<lastRead);     // wait for the next read
      lastRead += READ_PERIOD;
      val = analogRead(ADC_PIN);    // 1024 ADC takes about 100 usec (10kHz)
      acnt--; sval+=val;            // accumulate data
      if(!acnt) {                   // display every apts values only
  val = (unsigned long) (sval / apts); // average of apts measurements
  sval=0; acnt=apts;
  if(cflag) {
    std_R[std_num]=val;       // store raw value of x for calibration
    std_num++;              // add new point
  }
  switch(hex_format) {        // modify the raw value if necessary
    case 2:                   // concentration from linear regression
    case 3:                   // (may be negative when close to 0)
      val = (unsigned long) abs(round(Cval(val)));
      break;
    case 4:
    case 5:                   // resistance in Ohm
      val = (unsigned long) Rval(val);
      break;
  }
  if(hex_format&1)
    Serial.println(val,HEX); // at least 200 usec for 3 bytes
  else
    Serial.println(val);
  if(cflag) {
    cflag=0;
    ncnt=npts-1;              // disable any subsequent sampling
#ifdef _DEBUG_
    Serial.print("# C[");
    Serial.print(std_num);
    Serial.print("] = ");
    Serial.print(std_C[std_num-1]);
    Serial.println();
#endif
  }
      }
    }
```

```
    ncnt -= npts;
    digitalWrite(LED_BUILTIN, LOW);// onboard LED off = "idle"
  }
}
```

D Scanner le bus I2C avec ~/Arduino/scanI2C.h

Le bus I2C (dit *two-wire*) accepte plusieurs cartes, ou «slaves», à condition que leurs 7-bit adresses sont distinctes. Il est parfois très utile de découvrir tout les cartes qui sont actives sur ce bus. Pour ceci, on essaye de communiquer à 127 adresses dans la plage de `0x01` à `0x7F` et consulte le code d'erreur. Si ce code est 0 (sans erreur), on reporte l'adresse active détectée et, en option, l'interprète selon sa définition fournie alleurs.

```
// simple I2C bus scan for valid addresses
unsigned char scanI2C() {
 unsigned char err, i, n = 0;
#ifdef DEBUG
 Serial.print(F("I2C scan:"));
#endif
 for (i = 1; i < 0x7F; i++ ) {
     Wire.beginTransmission(i);
     // use the return value to see if device i responded to the query
     switch(err = Wire.endTransmission()) {
       case 4:
#ifdef DEBUG
   Serial.print(F(" ERROR 0x"));
   Serial.print(i,HEX);
#endif
       case 0:
#ifdef DEBUG
   Serial.print( (n ? F(", ") : F(" ")) );
   switch(i) { // display recognized devices
#ifdef MCP3424_ADDR
     case MCP3424_ADDR: Serial.print(F("MCP3224 "));
       break;
#endif
#ifdef DS3231_ADDR
     case DS3231_ADDR: Serial.print(F("DS3231 "));
       break;
#endif
#ifdef RTC_EEPROM_ADDR
     case RTC_EEPROM_ADDR: Serial.print(F("EEPROM "));
#endif
#ifdef _SPARKFUN_MMC5983MA_
     case I2C_ADDR: Serial.print(F("MMC5983 "));
#endif
#ifdef RTC_ADDR
     case RTC_ADDR: Serial.print(F("RTC "));
#endif
#ifdef MMC56X3_DEFAULT_ADDRESS
#ifdef MMC56X3_ADDR
     case MMC56X3_ADDR:
#else
     case MMC56X3_DEFAULT_ADDRESS:
#endif
       Serial.print(F("MMC56X3 ")); // 0x30
       break;
#endif // end for MMC56X3_DEFAULT_ADDRESS
     default:
       break;
   }
#endif // end for DEBUG, TODO: print space for all recognized addresses
#ifdef DEBUG
   Serial.print(F("0x"));
   if(i<0x10) Serial.print(F("0"));
   Serial.print(i,HEX);
#endif
   n++;
       default:
   break;
     }
   }
#ifdef DEBUG
 if(n) {
```

```
   Serial.print(F(" -> "));
   Serial.print((int) n);
   Serial.print(F(" device"));
   if(n>1) Serial.print(F("s"));
 }
 Serial.print(F("\n")); // avoid DOS-style \r\n that println implements
#endif
 return n;
}
```

Pour ne pas encombrer votre fichier principal avec extension `.ino` (le sketch), placez le code ci-dessus dans un fichier `scanI2C.h` (ou faites un link symbolique à ce fichier situé ailleurs) dans le même répertoire que votre `.ino` et engagez le avec la directive `#include`, voir l'example ci-dessous.

```
#define DEBUG 1
#define MCP3424_ADDR 0x6C // MCP3424 diff amp ADC (0x68 is reserved for the RTC)
#include <Wire.h>         // I2C bus library
#if defined(DEBUG) && DEBUG>0
#include "scanI2C.h"      // auxiliary code for I2C address scan
#endif

void setup() {
 Wire.begin();            // enable I2C communications (RTC,MCP)
#if defined(DEBUG) && DEBUG>0
 scanI2C();
#endif
}
```

E Méthode des moindres carrés. Régression linéaire

L'idée centrale de cette méthode (appelée *least squares fit* en anglais) est de trouver les valeurs de paramètres α, pour lesquelles la déviation moyenne quadratique entre la théorie et l'expérience

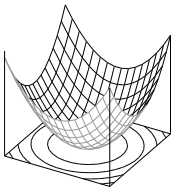
$$F(\alpha) = \sum_{i=1}^{N} \left(\frac{y_i - f(x_i, \alpha)}{\sigma_i} \right)^2$$

devient minimale. Ici les écarts types σ_i de chaque y_i jouent le rôle de «'poids» permettant de donner plus «d'importance» aux mesures plus précises. Dans le cas d'une régression *linéaire* où

$$y = f(x) = a_1 x + a_0, \quad \alpha = \{a_1, a_0\}$$

on a *deux* paramètres, et on cherche

$$\min_{a_1,a_0} F(a_1, a_0) = \min_{a_1,a_0} \sum_{i=1}^{N} \left(\frac{y_i - (a_1 x_i + a_0)}{\sigma_i} \right)^2 .$$



En introduisant les «moyennes pondérées»

$$[g] := \frac{1}{S} \sum_{i=1}^{N} \frac{g_i}{\sigma_i^2}, \quad \text{où} \quad S = \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \quad \text{et} \quad g_i = x_i, y_i, x_i y_i, x_i^2 \tag{E.1a}$$

(devenant les moyennes $[g] = \bar{g}$ dans le cas simple où tous $\sigma_i \equiv \sigma$), on trouve les valeurs (voir le code `linreg` dans la sec. 4.1.3)

$$a_1 = \frac{[xy] - [x][y]}{[xx] - [x][x]}, \quad a_0 = [y] - [x]\, a_1 \tag{E.1b}$$

qui minimisent $F(a_1, a_0)$ avec leurs écarts types correspondant[36]

$$\sigma_{a_1}^2 = \frac{S^{-1}}{[xx] - [x][x]}, \quad \sigma_{a_0}^2 = \sigma_{a_1}^2 [xx]. \tag{E.1c}$$

[36] Dans le cas simplifié $S^{-1} \approx \sigma^2 N^{-1}$, et par conséquence, les incertitudes Δa_1 et Δa_0 décroissent comme $1/\sqrt{N}$. On retrouve ainsi un résultat fondamental.

Démonstration. Au minimum de $F(a_1, a_0)$, on a $dF = 0$. On obtient le système de deux équations linéaires de variables (a_1, a_0)

$$\begin{cases} \dfrac{\partial F}{\partial a_1} = 0 \\ \dfrac{\partial F}{\partial a_0} = 0 \end{cases} \Rightarrow \begin{cases} \displaystyle\sum_{i=1}^{N} x_i(y_i - (a_1 x_i + a_0))\sigma_i^{-2} = 0 \\ \displaystyle\sum_{i=1}^{N} (y_i - (a_1 x_i + a_0))\sigma_i^{-2} = 0 \end{cases} \Rightarrow \begin{cases} [xy] - a_1[xx] - a_0[x] = 0 \\ [y] - a_1[x] - a_0 = 0 \end{cases}$$

dont la solution est donnée par (E.1b). □

```
struct {float a0,a1;} linreg;       // linear regression coefficients (global)
float linreg_fun(float x) { linreg.a0 + linreg.a1*x; }
/* compute linear regression coefficients for n integer data set y(x) */
void linreg_define(unsigned int n, unsigned int *x, unsigned int *y) {
 int k; float xx,_y,_x,xy;
 if(n>1) {                          /* check if there's enough standard data */
   for(k=n, _x  = _y = xx = xy = 0; k--; _x += *x++, _y += *y++) {
     xx += ((float) *x) * *x;                 /* average of x^2 */
     xy += ((float) *x) * *y;                 /* average of x*y */
   }
   linreg.a1 = (xy*n - _x*_y) / (xx*n - _x*_x); /* slope of y(x) */
   linreg.a0 = _y - _x * linreg.a1;
   linreg.a0/= n;                            /* shift of y(x) */
 }
 else linreg.a1 = linreg.a0 = 0;
}
/* compute standard unshifted error (sigma) for n integer data set y-f(x) */
float linreg_stderr(unsigned int n, unsigned int *x, unsigned int *y) {
 int k; float s=0,f;
 if(n>1) {                          /* check if there's enough data */
   for(k=n; k--; s+=f*f) f = linreg_fun(*x++) - *y++;
   s /= n;                          /* unshifted normalized standard error   */
 }
 return(sqrt(s));                   /* sqrt( (sum_i (f(x_i) - y_i)^2) / N ) */
}
```

F Échantillonnement de gyroscopes du pendule du *Pohl* `gyrosampler.ino`

Voir les sections 2.4.2 (interruptions), 4.11, 4.11.3, et son appendice 4.11.3C.

```
/* -*- mode: c++; coding: latin-1; current-input-method: latin-1-prefix; ispell-local-dictionary: "english";
    fill-column: 79; comment-column: 0; eval: (local-set-key "\C-c\C-\M-u" 'browse-url-firefox); -*-
  simple precise fixed-time-interval 2-channel gyro sampler (2x pololu L3GD20)
  see the gyro datasheet at https://www.pololu.com/file/0J731/L3GD20H.pdf and
  https://www.st.com/resource/en/application_note/an4505-l3gd20-3axis-digital-output-gyroscope-
      stmicroelectronics.pdf
  with QRE1113 based phase locking aka zero crossing detector aka "button"
  intended for the Pohl's pendulum student lab setup, see PHYWE ref. P2132705
  https://www.phywe.com/experiments-sets/university-experiments/forced-oscillations-pohl-s-pendulum_10989_12022/
  DS 2023-08-20
*/
#define _PROGRAM_ "gyrosampler"  // program name
#define DEBUG 1                  // set to 0 to suppress extra msg
#define _VER_ "Aug 30 2024"      // program version
#define TIME_DEFAULT "12:30:00"
#if !defined(__DATE__)           // normally these are set by compiler
#define __DATE__ _VER_
#endif
#if !defined(__TIME__)
#define __TIME__ TIME_DEFAULT
#endif
#define F_CPU_SCL (16000000/F_CPU) // CPU frequency scaling (2 for 3.3V 8MHz)

/* instead of the usual speed of 9600 bps, a faster serial communication line,
   with speed of 115200 bps = 14400 bytes/sec, 70 usec/byte can be used */
#define SERIAL_BPS 115200
// ATTN: on Arduino Uno and other 328p-based boards, digital pins 2 and 3
// are the only ones that can be used for programmable external interrupts 0,1
#define BUTTON_PIN  2            // external interrupt trigger pin
/*
  NB: the role of the button can be also provided (in parallel or solely) by
  the infrared line sensor such as SparkFun ROB-09453 Breakout (Analog) for
```

```
  QRE1113 (https://www.sparkfun.com/datasheets/Robotics/QR_QRE1113.GR.pdf), see
  https://www.sparkfun.com/products/9453, which has a 10K resistor in series
  with the IR photo-transistor making a voltage divider for Vout so that
  without reflection, Vout is high, i.e., the photo-cell makes a low-end
  switch. Note that the SparkFun ROB-09454 Breakout (Digital) in has a 10nF
  capacitor instead and an additional 200 Ohm protecting the Vout line, see
  https://www.sparkfun.com/products/9454. This board can be converted to the
  analog (continuous) sensor by adding a 10K resistor between Vcc and Vout. The
  concrete results for UNO Vcc=5V are 0.5 to 0.7V for reflection off a white
  painted metal plate at approx 4mm distance, and 4.9V without any reflection
  (the sensor pointed in the open). These are sufficient to trigger HIGH/LOW
  logic events, see https://docs.arduino.cc/learn/microcontrollers/5v-3v3 and
  especially
  https://learn.sparkfun.com/tutorials/logic-levels/arduino-logic-levels and
  https://forum.arduino.cc/uploads/short-url/2rCNgicUzd2hDpinV0C9C8RpWdx.pdf .
  The schematics for the abovementioned breakout boards are available at
  http://cdn.sparkfun.com/datasheets/Sensors/Proximity/QRE1113-Digital-Breakout-v11.pdf.
  http://cdn.sparkfun.com/datasheets/Sensors/Infrared/QRE1113%20Line%20Sensor%20Breakout%20-%20Analog.pdf
*/
/*
  https://www.arduino.cc/reference/en/language/functions/analog-io/analogwrite/
  PWM on Uno, Nano, Mini: 490 Hz on pins 3, 9, 10, 11,  980 Hz on pins 5 and 6

  NB: PWM outputs on pins 5 and 6 may have higher-than-expected duty cycles
  because of interactions with functions millis() and delay(), which use the
  same internal timer. This will be noticed mostly on low duty cycle settings
  (e.g. 0-10) and may result in a value of 0 not fully turning off the output.

  so better choose 490 MHz because the sampler code relies heavily on millis()
  ATTN: since this may lead to perturbations on the external interrupt pin 2
  https://forum.arduino.cc/t/pwm-appears-to-be-conflicting-with-external-interrupts/117351
  do not use pin 3 but rather one of 9,10,11

*/
#define PWM_PIN 9        // PWM @490 Hz (away from DP2 and not used for SPI)

#define AMP_PIN A0       // analog input from the current sensor

// 20000us gives 50Hz sampling rate => 50 samples per 1/2-period of the pendulum
// 40000us gives 25Hz sampling rate => 50 samples per one period of the pendulum
unsigned long READ_PERIOD = 40000;
unsigned long lastRead=0;
unsigned int  npts=0;    // number of collected samples
unsigned char gyros=0;   // bits 0,1: gyroscope board presence and usage
#define GYRO_ALIGN -1    // should be -1 for opposite axis alignment

#include <limits.h>
#include <math.h>        // some math constants
#include <avr/sleep.h>   // AVR library for sleep modes
#include <avr/power.h>
#include <avr/wdt.h>     // WDT timer can be potentially used if thigs go south

#include <Wire.h>        // I2C bus, DS3231 0x68, mcp3424 (DFR0316) 0x6A
#include <L3G.h>         // L3GD20, L3GD20H, and L3G4200D gyros on Pololu boards
L3G gyro[2];             // prowide for two-gyroscope-board configuration

#include "printlf.h"     // awkward ad-hoc replacement for Serial.println()
#if defined(DEBUG)
// see /opt/arduino/hardware/arduino/avr/libraries/Wire/src/Wire.cpp
#if DEBUG>0 && defined(TwoWire_h)
#include "scanI2C.h"     // auxiliary code for I2C address scan
#endif
#include "freeRAM.h"     // free RAM information
#include "boardinfo.h"   // extract board information from chip signature
#endif
#ifdef AMP_PIN
#define ACS712_TYPE 5    // our chip is ACS712T ELC-05B 22121 (5A max)
#define AMP_SIGN -1
#include "acs712.h"      // ACS_712 current sensor IC
#define readVcc_nsamples 16
// NB: with 16 samples, the sampling accuracy is about 10mV but it is fast
#include "readVcc.h"     // internal voltage reference
#endif

#ifdef BUTTON_PIN
#include <time.h>        // modified C header file for avr-libc and AVR-GCC
```

```
// time in ms of the first and last trigger events
volatile time_t        button_start=0, button_timer=0;
// number of button trigger instances
volatile unsigned int  button_count=0;

#define BUTTON_INT digitalPinToInterrupt(BUTTON_PIN)
// cf https://forum.arduino.cc/t/does-the-serial-communication-interfere-with-usage-of-external-interrupts
    /1085101
// boxcar wait interval (ms) during which subsequent interrupts will be ignored
// as spurious to reduce jitter
#define BUTTON_GAP 2
#define BUTTON_LED LED_BUILTIN
// ISR for handling interrupt triggers arising from associated button switch
void button_int_handler() {
#ifdef BUTTON_GAP
 time_t t = millis();             // bail out if the events get too close
 if (button_count && t-button_timer < BUTTON_GAP) return;
 button_timer = t;                // new event is outside the gap
#else
 button_timer = millis();         // time of the event and number of events
#endif
 if(!button_count++) button_start=button_timer;
#ifdef BUTTON_LED                 // toggle LED indicator
 digitalWrite(BUTTON_LED, !digitalRead(BUTTON_LED));
#endif // button led
}
#endif // button pin

#ifdef PWM_PIN
// https://www.arduino.cc/reference/en/language/functions/analog-io/analogwrite/
// analogWrite values go from 0 (always off) to 255 (always on)
// (the PWM mode which is pre-cofigured on the UNO for all pins is 8 bit)
// analogWrite has nothing to do with the analog pins and analogRead
// the pin will generate a steady rectangular wave of the specified duty cycle
// until the next analogWrite(), digitalRead(), or digitalWrite() on this pin

// https://passionelectronique.fr/pwm-arduino/
// https://passionelectronique.fr/wp-content/uploads/Atmel-ATmega328P-datasheet.pdf?pseSrc=pgPwmArduino
// The ATmega168P/328P chip has three PWM timers, controlling 6 PWM outputs.
// Timer   Timer   Arduino
// compare output -> pin
// OCR0A   OC0A       6    Fast PWM Mode 8-bit @976,56 Hz, 255 steps
// OCR0B   OC0B       5    Fast PWM Mode 8-bit @976,56 Hz, 255 steps
// OCR1A   OC1A       9    Phase Correct PWM Mode 8-bit @490,20 Hz, 510 steps
// OCR1B   OC1B      10    Phase Correct PWM Mode 8-bit @490,20 Hz, 510 steps
// OCR2A   OC2A      11    Phase Correct PWM Mode 8-bit @490,20 Hz, 510 steps
// OCR2B   OC2B       3    Phase Correct PWM Mode 8-bit @490,20 Hz, 510 steps

// NB: by default, on the UNO, the pwm is configured so that

#define PWM_BOTTOM 0x0000

// DP5 and 6 use the 8-bit Timer 0 and can only work with 8-bit resolution.
// They are configured by default for Fast (simple) PWM with frequency
// f = 16 MHz / 64 / 256 = 976,5625 (the simple duity cycle of 255+1 steps)
// (here notice that 16*2^20/64/256 = 1024)

#if PWM_PIN==5          // Fast PWM Mode 8-bit @976,56 Hz
#define PWM_OCR OCR0A   // Timer 0 register A
#define PWM_OC  OC0A
#define PWM_COM COM0A0  // lsb of the 2 compare output bits in TCCR0A
#endif
#if PWM_PIN==6          // Fast PWM Mode 8-bit @976,56 Hz
#define PWM_OCR OCR0B   // Timer 0 register B
#define PWM_OC  OC0B
#define PWM_COM COM0B0  // lsb of the 2 compare output bits in TCCR0A
#endif
#if PWM_PIN==5 || PWM_PIN==6
#define PWM_MAX 0x00FF  // MAXimum of the Timer 0 counter
#define PWM_TOP PWM_MAX // highest value in the Timer 0 count sequence, 8-bit
#define PWM_CYCLE 256   // number of steps in the period
#define PWM_TCCRA TCCR0A
#define PWM_TCCRB TCCR0B
#endif
/*
  DP3 and 11 use Timer 2 (8-bit) and can only work with 8-bit resolution; DP9
```

```
  and 10 use Timer 1 (16-bit) and support more precise 9- and 10-bit pwm modes
  all DP9, 10, 11, and 3 are configured for Phase Correct PWM with frequency

  f = 16 MHz / 64 / 510 = 490,196 (510 = 2*255 steps in the duty interval)

  NB: The phase-correct PWM mode and the phase-and-frequency-correct PWM mode
  are based on a dual-slope operation with counter going repeatedly from BOTTOM
  (0) to TOP and back to BOTTOM.  Due to the symmetry of the dual-slope PWM
  modes, the latter are preferable in motor control applications.
*/
#if PWM_PIN==9         // Phase Correct PWM Mode 8-bit @490,20 Hz
#define PWM_OCR OCR1A  // Timer 1 register for channel A
#define PWM_OC  OC1A
#define PWM_COM COM1A0 // lsb of the 2 compare output bits in TCCR1A
#endif
#if PWM_PIN==10        // Phase Correct PWM Mode 8-bit @490,20 Hz
#define PWM_OCR OCR1B  // Timer 1 register for channel B
#define PWM_OC  OC1B
#define PWM_COM COM1B0 // lsb of the 2 compare output bits in TCCR1A
#endif
#if PWM_PIN==9 || PWM_PIN==10
#define PWM_MAX 0xFFFF // MAXimum of the Timer 1 counter (decimal 65535)
#define PWM_TOP 0x00FF // upper value in the Timer 1 count sequence, 8-bit
// depending on the operation mode, Timer 1 TOP equals 0x00FF (default 8-bit
// PWM), 0x01FF, 0x03FF, or the value stored in the OCR1A or ICR1 register.
#define PWM_CYCLE 510  // number of steps in the default fcPWM period 2*255
#define PWM_TCCRA TCCR1A
#define PWM_TCCRB TCCR1B
#define PWM_TCCRC TCCR1C
// Atmel ATmega328P Waveform Generation Modes (WGM)
// bits _BV(WGMx3)!_BV(WGMx2)  and _BV(WGMx1)|_BV(WGMx0) for Timer 1
#define PWM_WGM ((PWM_TCCRB&0x18)>>1 | (PWM_TCCRA&3))
const char *pwm_wgm[] = { // datasheet, 15-5 p.109
  "-",     "pc~8",  "pc~9", "pc~10", "CTC-O", "f~8", "f~9", "f~10",
  "pfc~I", "pfc~O", "pc~I", "pc~O",  "CTC-I",   "?", "f~I", "f~O" };
#else
// bits _BV(WGMx2)  and _BV(WGMx1)|_BV(WGMx0) for Timers 0 and 2
#define PWM_WGM ((PWM_TCCRB&0x08)>>1 | (PWM_TCCRA&3))
const char *pwm_wgm[] = { // datasheet, 14-8 p.86
  "-",  "pc~",  "CTC", "f~", "?", "pc~O", "?", "f~O"};
#endif

#if PWM_PIN==11        // Phase Correct PWM Mode 8-bit @490,20 Hz
#define PWM_OCR OCR2A  // Timer 2 8-bit register A
#define PWM_OC  OC2A
#define PWM_COM COM2A0  // lsb of the 2 compare output bits in TCCR2A
#endif
#if PWM_PIN==3         // Phase Correct PWM Mode 8-bit @490,20 Hz
#define PWM_OCR OCR2B  // Timer 2 8-bit register A
#define PWM_OC  OC2B
#define PWM_COM COM2B0  // lsb of the 2 compare output bits in TCCR2A
#endif
#if PWM_PIN==3 || PWM_PIN==11
#define PWM_MAX 0x00FF  // MAXimum of the Timer 2 counter
#define PWM_TOP PWM_MAX // highest value in the Timer 0 count sequence, 8-bit
#define PWM_CYCLE 510   // default number of steps in the period
#define PWM_TCCRA TCCR2A
#define PWM_TCCRB TCCR2B
const unsigned char pwm_divisors[] = {0,0,3,5,6,7,8,10}; // Timer 2
#else
const unsigned char pwm_divisors[] = {0,0,3,6,8,10};     // Timer 0 and 1
#endif
/*
    Arduino timer scaling:
    common prescaler: 1, 2, 4, 8, 16, 32, 64, 128, 256
                      0  1  2  3   4   5   6    7    8   (value of CLKPR & 0xF)
    timer0 or timer1: 0, 1, 8, 64, 256, 1024
                      0  1  2   3    4     5   (value of TCCR0B or TCCR1B & 7)
    timer2:           0, 1, 8, 32,  64,  128, 256, 1024
                      0  1  2   3    4     5    6,    7  (value of TCCR2B & 7)
*/

#define PWM_ON   (PWM_TCCRA>>PWM_COM)&0x3
#define PWM_DIV  (1<<pwm_divisors[PWM_TCCRB & 7])
#define PWM_FREQ ((float) F_CPU / (1<<(CLKPR & 0xF)) / PWM_DIV / PWM_CYCLE)
```

```
// per mille (Latin meaning 'in each thousand') indicates parts per thousand
// aka promille and permille, the symbol being o/oo (the usual percentage o/o)

// read the appropriate PWM timer control register (the raw value of pwm)
// and return as per mille parts 0..1000 of PWM_TOP
unsigned int pwm_get() {
#ifdef PWM_OCR
 return map(PWM_OCR, 0, PWM_TOP, 0, 1000);
#else
// analogRead(PWM_PIN) with values of 0..1023 is silly: it cannot read DP's !
 return digitalRead(PWM_PIN); // paranoid for screw-ups
#endif
}
// assuming v=0..1000 per mille parts, convert percents to the 0..PWM_TOP range
// and set the PWM signal on the PWM_PIN
unsigned int pwm_set(unsigned int v) {
 v = map(v, 0, 1000, 0, PWM_TOP);
 analogWrite(PWM_PIN, v);
 return pwm_get();
}
// https://en.wikipedia.org/wiki/Pulse-width_modulation
// the average voltage for PWM with duty cycle D equals D*Vmax + (1-D)*Vmin
// compare this code to the map function available in arduino lib
#endif

#ifdef AMP_PIN
/*
  The DC current in the electromagnet of the Eddy current brake or dumper
  or retarder, also known induction, Faraday, or electromagnetic brake.
  French: ralentisseur électromagnétique / frein de Foucault
*/
#define AMP_SAMPLES_MAX 64          // TODO: use integers for A and dA ?!
float A=0, dA=0;                    // running mean value and std error
unsigned int nA=0;                  // number of collected samples
int amp_sample(int i) {             // acquire i samples x_k of AMP_PIN
 float d; int x;
 while(i--) {
   x = analogRead(AMP_PIN)-512;     // load x with k-th value x_k, k=nA+1
#if defined(AMP_SIGN) && AMP_SIGN<0
   x*=AMP_SIGN;                     // legacy for wrongly connected acs712
#endif
   // Serial.println(x);
   if(nA++) {           // for k>1
     d=x-A;             // x_k - A_{k-1}
     A+=d/nA;           // A_k = A_{k-1} + (x_k-A_{k-1})/k
     dA+=d*(x-A);       // Q_k = Q_{k-1} + (x_k-A_{k-1})*(x_k-A_k)
   }
   else {               // for k=1 (since A_0 = Q_0 = 0)
     A=x;               // A_1 = x_1
     dA=0;              // Q_1 = 0
   }
 }
 return (int) A; // ACS712_mA(A);
}

#endif

// NB: the F() moves strings to flash memory instead of wasting SRAM
void print_info(unsigned char typ, unsigned char flags, unsigned char pcnt) {
 unsigned char i;
#if defined(DEBUG) && DEBUG>=0
 if(typ&1) {
   Serial.print(F("# "));
   Serial.print(F(_PROGRAM_));
#if defined(ARDUINO_ARCH_AVR)
   Serial.print(F(" AVR:"));        // compiled for avr architecture (16-bit)
#else
   Serial.print(F(" "));
#endif // end arch
   Serial.print(F(_VER_));  Serial.print(F(", MCU "));
   Serial.print(BOARD_INFO,HEX);
   Serial.print(F("@")); Serial.print(F_CPU/1000000); Serial.print(F("MHz "));
   Serial.print(BOARD_TYPE);        // Arduino board name
   Serial.print(F(" SRAM:")); Serial.print(freeRAM()); // free data RAM available
   printlf();                       // end first header line
#if DEBUG>1
```

```
   Serial.print(F("# compile date "));
   Serial.print(F(__DATE__));  Serial.print(F(", ")); Serial.print(F(__TIME__));
   Serial.print(F(" from ")); Serial.print(F(__FILE__));
   printlf();
#endif // end DEBUG>1
 }
#endif // end DEBUG>=0
#if defined(TwoWire_h) && defined(DEBUG) && DEBUG>0
 if(typ&2) {                        // if I2C library is loaded
   Serial.print(F("# "));
   scanI2C();                       // scan and report all devices on I2C
 }
#endif // end I2C
#ifdef DEBUG
 // if asked for info on gyro[i] and it is available
 // i=0 for 0x6a (front, pendulum axis, large amplitude)
 // i=1 for 0x6b (rear, driving force, small ampllitude)
 for(i=0; i<=1; i++) {
   if(typ&(4<<i) && gyros&(1<<i)) {
     Serial.print(F("# gyro[")); Serial.print(i);  Serial.print(F("] "));
     // device id from register 0x0F and from L3G identification
     Serial.print(gyro[i].readReg(L3G::WHO_AM_I), HEX);
     Serial.print(F("(")); Serial.print((int) gyro[i].getDeviceType());
     // I2C address  0x6a+i (in L3G.cpp D20_SA0_LOW_ADDRESS+i)
     Serial.print(F(")@0x")); Serial.print(0x6a+(i?L3G::sa0_high:L3G::sa0_low),HEX);
     Serial.print(F(" t="));
     // temperature data (-1LSB/deg, 8 bit) from register 0x26
     // NB: the L3GD20 includes an on-chip temperature sensor that is suitable
     // for differential temperature measurements: updated @1Hz, the register
     // provides a (negative?) temperature relative to an uncalibrated /
     // unspecified offset to account for temperature variations over time.
     Serial.print( (int) gyro[i].readReg(L3G::OUT_TEMP) );
     // active axes
     Serial.print(F(" R1,4="));
     Serial.print(gyro[i].readReg(L3G::CTRL_REG1),HEX);
     // FS1-FS0 bits define full scale (FS) in dps
     Serial.print(F(","));
     Serial.print(gyro[i].readReg(L3G::CTRL_REG4),HEX);
     printlf();
   }
 }
#endif // end gyro info
#if defined(DEBUG) && defined(BUTTON_PIN)
 if(typ&0x10) {
   Serial.print(F("# trig="));      // current trigger state (0=equilibrium)
   Serial.print(digitalRead(BUTTON_PIN));
#ifdef BUTTON_GAP
   Serial.print(F(" gate="));       // boxcar gate to suppress jittering
   Serial.print(BUTTON_GAP);
   Serial.print(F("ms"));
#endif
   printlf();
 }
#endif
 if(typ&0x20) {                     // current sampling parameter values
   Serial.print(F("# time="));
   Serial.print(READ_PERIOD);
   Serial.print(F(" usec, "));
   Serial.print(npts);
   Serial.print(F(" samples at "));
   Serial.print(1000000.0 / READ_PERIOD);
   Serial.print(F(" Hz"));
   if(pcnt) {
     Serial.print(F(" or "));
     Serial.print((float) pcnt / 2);
     Serial.print(F(" periods"));
   }
   Serial.print(F(" flags "));
   Serial.print(flags,BIN);
#ifdef PWM_PIN
   Serial.print(F(" V="));          // voltage mV in the motor driver circuit
   Serial.print(pwm_get(), DEC);
#endif
#ifdef AMP_PIN
   /*
     The DC current in the electromagnet of the Eddy current brake /
```

```
     dumper / retarder, aka induction, Faraday, or electromagnetic
     brake. French: ralentisseur électromagnétique / frein de Foucault
   */
   Serial.print(F(" J="));          // DC current in the dumper electromagnet
   Serial.print(ACS712_mA(A));
   Serial.print(F("("));
   Serial.print((nA>0?ACS712_mA(sqrtf(dA/nA)):0),DEC);
   Serial.print(F(")mA"));         // Serial.print(nA);
#ifdef readVcc_nsamples
   Serial.print(F(" Vcc="));
   Serial.print(Vcc);              // current value of Vcc
#endif
#endif
   printlf();
 }
 if(typ&0x40) {
   Serial.print(F("#BEGIN flag=")); Serial.print(flags,BIN);
   for(i=0; i<=1; i++)             // loop on gyros i=0,1
     if(gyros&(1<<i)) {
       Serial.print(F(" s")); Serial.print(i);  Serial.print(F("="));
       Serial.print(gyro[i].readReg(L3G::CTRL_REG4),HEX);
     }
#ifdef GYRO_ALIGN
   Serial.print(F(" sgn="));        // gyroscope alignment correction
   Serial.print(GYRO_ALIGN);
#endif
#ifdef PWM_PIN
   Serial.print(F(" V="));          // driver motor voltage mV
   Serial.print(pwm_get(), DEC);
#endif
#ifdef AMP_PIN
   Serial.print(F(" J="));          // current in the electromagnetic dumper
   Serial.print(ACS712_mA(A));
#endif
 }
 if(typ&0x80) {
   Serial.print(F("# scl="));       // overall timer frequency prescaler
   Serial.print( 1<<(CLKPR & 0xF), DEC);
   // " @" F_CPU / (1<<(CLKPR & 0xF)) / 1000000 "MHz"
#ifdef PWM_PIN                      // pwm setup information
   Serial.print(F(" DP"));
   Serial.print(PWM_PIN);           // digital pin used
#ifdef PWM_COM                      // examine pwm compare output mode bits
   Serial.print((i=PWM_ON)?"~":"=");
   if(!i)                           // if pwm is off, show pin status 0/1
     Serial.print(digitalRead(PWM_PIN));
#endif
#ifdef PWM_WGM
   Serial.print(F(" "));            // Wave Generation Mode
   Serial.print(pwm_wgm[i = PWM_WGM]);
#if defined(DEBUG) && DEBUG>0
   Serial.print(F("["));            // content of the WGM bits (aka WG=)
   Serial.print( i, BIN);
   Serial.print(F("]"));
#endif
#endif
   Serial.print(F(" CS="));         // timer clock select (divisor) bits
   i=PWM_TCCRB&0x07;                // _BV(CSx2)|_BV(CSx1)|_BV(CSx0)
   Serial.print(1<<pwm_divisors[i],DEC);
#if defined(DEBUG) && DEBUG>0
   Serial.print(F("["));            // content of the CS bits
   Serial.print(i,BIN);
   Serial.print(F("]"));
#endif
   if(i) {
     Serial.print(F(" @"));         // frequency
     Serial.print(PWM_FREQ,0);
     Serial.print(F("Hz"));
     }
   else Serial.print(F(" off"));    // pwm timer is off
   Serial.print(F(" "));            // x2 for phase (and frequency) correct WGM
   if(*pwm_wgm[PWM_WGM]=='p') Serial.print(F("2x"));
   Serial.print(PWM_OCR, DEC);      // duty cycle [clock ticks]
   Serial.print(F("/"));
   Serial.print(PWM_CYCLE);         // full pwm pulse duration (aka period)
#endif
```

```
   printlf();
 }
}

void setup() {
 unsigned char i;
 pinMode(LED_BUILTIN, OUTPUT);
 digitalWrite(LED_BUILTIN, HIGH);   // turn the onboard LED on = "busy"
#ifdef PWM_PIN
 pinMode(PWM_PIN, OUTPUT);          // activate pwm on pin PWM_PIN
 digitalWrite(PWM_PIN, LOW);        // turn the pwm pin off (V=0)
#endif
#ifdef BUTTON_PIN
#ifdef BUTTON_LED
#if BUTTON_LED != LED_BUILTIN
 pinMode(BUTTON_LED, OUTPUT);       // special LED indicator of the trigger
#endif
 digitalWrite(BUTTON_LED, LOW);     // flush LED indicator of the trigger
#endif
 /* declare and set interrupt pins: on UNO, pins 2,3 give interrupt 0,1  */
 pinMode(BUTTON_PIN, INPUT);        // pulldown push button or switch
 attachInterrupt(BUTTON_INT, button_int_handler, FALLING);
 // the mode can be LOW (whenever the pin is low), CHANGE, RISING, FALLING
 // https://www.arduino.cc/reference/en/language/functions/external-interrupts/attachinterrupt/
 // NB: FALLING triggers when the sensor signal goes from HIGH to LOW, which
 // means when the reflecting obstacle comes into sensor's sight. In the Pohl's
 // pendulum setup with the sensor placed at the equilibrium position, this
 // avoids an artifact interrupt occuring when the pendulum is pulled out of
 // its equilibrium manually and then launched.
#endif
 Serial.begin(SERIAL_BPS);          // open serial communication port
 while(!Serial);                    // wait for it to connect (native USB only)
 print_info(1,0,0);
#if defined(TwoWire_h)              // if I2C library is loaded
 Wire.begin();                      // enable I2C communications (RTC,MCP)
 print_info(2,0,0);
#endif // end I2C
 // i=0 for 0x6a (front, pendulum axis, large amplitude)
 // i=1 for 0x6b (rear, driving force, small ampllitude)
 for(i=0; i<=1; i++) {
   /* The I2C Slave ADdress (SAD) associated to the L3GD20H is binary 110101x.
      The SDO/SA0 pin can be used to modify the less significant bit (LSB) x of
      SAD: if it is connected to Vcc, then x=1 (SAD = 1101011 or 0x6b) else if it
      is connected to the ground then x=0 (SAD = 1101010 or 0x6a). This permits
      to connect and address two different gyroscopes to the same I2C bus. */
   // is the board D20H or D20 ? it returns 0xD7, so D20H
   // D20H_WHO_ID     0xD7
   // D20_WHO_ID      0xD4
   // L3G4200D_WHO_ID 0xD3
   if(!gyro[i].init(L3G::device_auto, (i?L3G::sa0_high:L3G::sa0_low) )) continue;
   gyros |= 1<<i; // availability flag
   gyro[i].enableDefault();
   /* Estimate the maximal angular velocity amplitude of the Pohl's pendulum:
      ----------------------------------------------------------------------
      The protractor circle of the apparatus is graduated in somewhat arbitrary
      units (au), such that 10 au correspond to 1/5th of the full circle, i.e.,
      2pi/5 = 10 units => pi/25 = 7.2 deg = 1 unit and pi/5 = 36 deg = 5 units.
      The maximal deviation angle A is (restricted to) about +/-20 au = 144
      deg.  The period of free oscillations is approximately 1.8 sec, and the
      proper (resonant) circular frequency w0 is, therefore, about
      2pi/1.8. This means that the maximal amplitude of the angular velocity of
      the pendulum equals A w0 = 144 2pi/1.8 = 160 pi = 502.65 dps (deg/sec). */

   // The default L3G gyro's full scale is set to +/-250 dps (FS1-FS0=0x00 in
   // register 0x23, see sec. 7.5, pp. 39-40 of the L3GD20 datasheet),
   // the call below selects the 500 dps scale (0x10).
   if(!i) gyro[0].writeReg(L3G::CTRL_REG4,0x10);
   // z-axis data only in register 0x20, see sec. 7.2 on p.36 of the datasheet
   gyro[i].writeReg(L3G::CTRL_REG1, gyro[i].readReg(L3G::CTRL_REG1)&0xFC);
   print_info(4<<i,0,0);
 } // end of loop on boards 0,1
#ifdef readVcc_nsamples
 Vcc=readVcc();                     // check the actual value of Vcc
#endif
 print_info(0x10,0,0);
 digitalWrite(LED_BUILTIN, LOW);    // turn the onboard LED off = "done"
```

```
}

void loop() {
 static unsigned int  ncnt=0;       // number of samples
 static unsigned char pcnt=0;       // number of full half-periods
 static unsigned char flags=0;      // flag bits: 0 hex format, 1 trigger wait
 static unsigned char key=';';      // initilize the stale key at first pass
 static long val=0;
 // because r is used as index in gyro[r], it should rather be nonnegative!
 // https://stackoverflow.com/questions/18502257/array-subscript-has-type-char-wchar-subscripts
 unsigned char r;
 // Assuming no line endings present, just plain stream of characters.  Number
 // arguments refer to the last non-number key (legacy mode). Consequently,
 // commands containing number arguments should terminate with a command that
 // doesn't use such an argument, for example 'i' or ';' (big headache...)
 while(Serial.available()) {        // read settings and commands
    r = Serial.read();
    if(isDigit(r)) {                // swallow the possible numerical value
      val*=10;
      val+= r-'0';
    }
    else {
      switch(key) {  // handle the PREVIOUS key preceeding a numerical value
        case 'T':    // sampling interval (usec)
          if(val) READ_PERIOD = val;
          break;
        case 'N':    // number of samples
          if(val) npts=val;
    break;
  case 'P':    // number of full half-periods to span (0..255)
          if(val) pcnt = val;
    break;
#ifdef PWM_PIN
  case 'V':    // set PWM external voltage control (0..1023)
    switch(val) {
      case 0:
        digitalWrite(PWM_PIN, LOW);    // bring PWM down (V=0)
        break;
      case 1000:
        digitalWrite(PWM_PIN, HIGH);   // bring PWM down (V=100%)
        break;
      default:
        pwm_set(val);
    }
    print_info(0x80,flags,pcnt);       // detailed info on pwm
    break;
#endif
      }
      val=0;
      key=r;          // === LAST READ byte becomes CURRENT key ===
      switch(key) {  // handle switches
  case 'Z':    // reset trigger count and display avg trigger time (period)
#ifdef BUTTON_PIN
          Serial.print(F("# "));
          Serial.print(button_count);
          Serial.print(F(" counts"));
    if(button_count>1) {
      Serial.print(F(" with T="));
      Serial.print( floor((float) (button_timer-button_start) / --button_count), 0);
      Serial.print(F("ms"));
    }
          printlf();
    button_count=0;
#endif
    break;
        case 'H':    // toggle hex format
          flags^=1;
          break;
        case 'W':    // toggle waiting for trigger
          flags^=2;
          break;
        case 'F':    // toggle sampling of second gyro, if available
          flags^=(gyros&2)<<1;
          break;
  case 'I':    // full startup header and current parameter values
    print_info(0x3F,flags,pcnt);
```

```
    break;
        case 'S':    // reset sample counter to start sampling
          ncnt=npts;
    /* The sampling time is the time interval between successive samples,
       also called the sampling interval or the sampling period, and
       denoted T. The sampling rate is the number of samples per
       second. It is the reciprocal of the sampling time, i.e. 1/T, also
       called the sampling frequency, and denoted Fs */
        default:     // general information on sampling parameters
#ifdef AMP_PIN
  case 'J':    // resample current in the electromagnetic dumper cirquit
    nA=0;
#ifdef readVcc_nsamples
    Vcc=readVcc();  // NB: the Vcc sampling may take a bit of time
#endif
    amp_sample(AMP_SAMPLES_MAX);
#endif
  case 'i':    // brief header and current parameter values
    print_info(0x20,flags,pcnt);
        case 'N':    // NB: commands that accept numerical values are postpended
  case 'P':    // and CANNOT be used to display information at this point
        case 'T':
#ifdef BUTTON_PIN
  case 'V':
#endif
        case 'Q':
        case ';':    // line ending, helpful for commands with numerical args
          break;
      }
    }
  }
  /* time resolution on 16MHz boards (such as 5V UNO) is 4 usec (1/16*64), and
     at this min sampling time ncnt will be overrun in approx 70 min, which is
     an ample amount of time ... */
  if(ncnt) {          // proceed to acquire and display npts samples
    print_info(0x40,flags,pcnt);
    Serial.flush();  // clear all output (i.e., wait) before going to sleep
    button_count=0;  // reset trigger count (aka number of 1/2 periods)
    //#if defined(DEBUG) && DEBUG > 2
    //#endif
    if(flags&2) {    // postpone sampling until the first trigger event
      interrupts();  // re-enable interrupts (if/after disabled by noInterrupts()
      set_sleep_mode(SLEEP_MODE_PWR_DOWN);   // full sleep mode
      sleep_enable();                    // (re)enable sleep mode
      sleep_cpu();                       // ***** ACTIVATE SLEEP *****
      sleep_disable();                   // on comeback: disable the sleep mode
      power_all_enable();                // re-enable peripherals, is it needed?
    }
    printlf();
    for(ncnt=0, lastRead=micros()+8; ncnt<npts && ! (pcnt && button_count>pcnt); ncnt++){
      while(micros()<lastRead);
      lastRead += READ_PERIOD;
      for(r=0; r<=((flags>>2)&1); r++) {
  // take a reading from the gyro number r=0,1 and store the value
  gyro[r].read();
        /*
        NB: these are raw 16-bit values obtained by concatenating the 8-bit
        high and low gyro data registers.  They can be converted to units of
        dps (degrees per second) using the conversion factors specified in the
        datasheet for your particular device and full scale setting (gain).
        Specifically, the L3GD20H datasheet on p.10 gives conversion factors
        for Sensitivity (So) 8.75, 17.50, and 70.00 mdps/digit (aka mdps/LSB)
        corresponding to full scale (FS) ranges of +/- 245, 500, and 2000 dps,
        respectively.  So, for example: if an L3GD20H with its default FS
        setting of 245 gives a reading of 345, this corresponds to 345 * 8.75 =
        3020 mdps = 3.02 dps, see examples of the L3G library
        https://github.com/pololu/l3g-arduino.
        */
  val = (int) gyro[r].g.z;
#ifdef GYRO_ALIGN
  if(r) val*=GYRO_ALIGN;
#endif
  // NB: at 115200 bps it takes at least 200 usec for 3 bytes
  // HEX is more economic but less readable, especially for negaives
  // TODO: use HEX as binary format (max 5 bytes)
  if(flags&1)
```

```
    if(val>0) Serial.print(val,HEX);   // positive hexadecimal
    else {
      Serial.print(F("-"));            // negative hexadecimal
      Serial.print(-val,HEX);
    }
  else Serial.print(val);              // decimal
  Serial.print(F(" "));
      }
      Serial.print(button_count); printlf(); // number of periods
    }
    Serial.print(F("#END n=")); Serial.print(ncnt); printlf();
    if(pcnt) ncnt=0; else ncnt-=npts;
  }
}
```

G Gestion des projets (AC)

Objectifs : Acquérir les méthodes et les outils fondamentaux de la gestion de projet pour piloter un projet avec succès et se doter d'une boîte à outils. A l'issue de la formation, les apprenants seront capables de :

1. S'approprier les notions clés de la gestion de projet ;
2. Identifier le rôle et les responsabilités des pilotes ;
3. Identifier les étapes clés d'un projet et le processus de mise en œuvre ;
4. Conduire un projet en mettant en œuvre une méthode et des outils opérationnels ;
5. Identifier les ressources pour la réussite d'un projet ;
6. Débloquer les situations difficiles dans la gestion de projet.

Compétences et savoirs Les étudiants devront ensuite valider les compétences et savoirs suivants :

1. Appliquer un QQOQCP ; (méthode couramment utilisée en entreprise à appliquer ici sur un «petit projet»)
2. Définir un plan d'actions ; (décomposition en tâches et sous-tâches, attribution des rôles, identifications des ressources, risques et parades. . .)
3. Organiser ce plan d'actions dans le temps et définir des livrables (diagramme de Gantt) ;

H Organisation de projets, année 2020

L'UO Arduino est offert à ULCO depuis 2019 et se distingue par son mode de travail. Il s'agit principalement des *projets* autour de micro-contrôleur Arduino que les étudiants réalisent en *quasi-autonomie* en groupes de 3-4, et, au moins en partie, en dehors de ses 20h heures «officielles». Cet approche engage non seulement les connaissances en informatique, électronique, physique, mathématique, chimie, et biologie—en fonction de la nature du projet, mais les aspects logistiques, relationnels et organisationnels, et elle demande une grande motivation de la part des étudiants. Alors tous ceux qui cherchent une note «facile» pour comforter leur moyenne, tous ceux qui anticipent les cours-TD-TP classiques où on peut dormir tranquillement, copier les notes et comptes rendus de son voisin, et attendre que l'enseignant effectue les manips du TP feront mieux d'abstenir et de libérer plus de nos ressources aux autres étudiants.

Dans ce contexte, le rôle principale des enseignants devient l'accompagnement. Après une courte introduction (4h), nous tiendrons 4 séances (16h au maximum) de «permanence» pour permettre aux étudiants d'avancer leur projets et de résoudre tous problèmes. Les groupes se rencontrent en dehors de ses heures «officielles» et restent en contact par mél avec leur enseignants «mentors». Vu la nature des projets et le niveau des étudiants très variées, l'énseignant ne peut pas s'occupper de 3 (au maximum 4) groupes à la fois. Cette année nos intervenants «mentors» seront :

- Dimitri Sadovskií (DS), professeur à ULCO, département de physique, sadovski@univ-littoral.fr
- Robin Bocquet (RB), professeur à ULCO, département de physique, robin.bocquet@univ-littoral.fr
- Arnaud Cuisset (AC), professeur à ULCO, département de physique, arnaud.cuisset@univ-littoral.fr
- Marc-Alexandre CARPENTIER (MAC), étudiant en L1 math Calais, marcal.carpentier@gmail.com
  après son IUT info, il a enseigné à l'IUT et a créé son propre entreprise : l'appli qu'il a développé (une base des données pour un site des rencontres) a déjà 150000 utilisateurs inscrits et cela lui paye une rente..
- Sébastien MENIERE (SM), étudiant en L3 math Calais, sebastien.meniere@gmail.com
  Sébastien a eu son propre entreprise d'électronique il est expert en Arduino et son projet de l'an passe était excellent (générateur des codes aléatoires Arduino pour porte monnaie bitcoin)

Pour tout commandes de matériel et composantes, des petites repartions ou d'autres aides (par exemple soudures), on peut s'adresser à

– Wilfried Montagnier (WM), soutien technique de TP physique Calais, wilfried.montagnier@univ-littoral.fr

Les 34 étudiants de l'année 2025 formiront 8 groupes de projet de 4 étudiants en moyen par groupe. Il est, bien sur, envisageable qu'ils ne sont pas tous sérieux, mais on peut, quand même, anticiper d'avoir 4 à 6 projets à suivre réellement. Comme l'équipement, nous donnons 1-2 coffrets Arduino par groupe/projet. Certains composants nécessaires se trouvent dans nos kits. Pour les composants spécifiques aux projets, il sera indispensable de choisir son projet au plus vite afin de pouvoir les commander et recevoir au temps. Tout demandes sont à adresser au sérvice TP physique (WM).

On peut commencer soit 24/1 soit 31/1 avec une séance d'introduction (4h) où tout le monde vient ensemble. La séance est découpée en deux parties : 2h dans une amphi (de 8h30 à 10h15) pour discuter des questions générales (RB) et «Gestion de projets» (AC) suivis par la présentation des projets possible (RB+DS+ ?) de 10h30 à 12h30. On choisit son projet. On prépare son choix en regardant la sec. 4, l'internet, en discutant par mél avec nous, et suite à la présentation des projets et aux échanges dans l'amphi. On forme les groupes (binômes, trinômes, 4 max) pour chaque projet. Les groupes choisissent leur «mentors». A ce point les aspects relationnels et la capacité de l'orientation rapide jouent un rôle important : il faut choisir ceux avec qui on peut travailler et surtout fuir ceux qui cherchent à ne rien faire. Par la suite, on fait 2h dans les classes info en deux groupes séparés (RB et DS) pour réaliser certains manips éducatifs dans la sec. 2. On a l'occasion de tester la cohésion des groupes.

Dès que les étudiants se départagent en groupes de projet, SM et MAC prendront chaque 1-2 groupes des étudiants motivés. RB et DS garderons les autres 4 groupes. SM et MAC organisent leur travail comme ils peuvent. Ils sont présents sur site. RB et DS tiendrons les permanences les vendredis. Ainsi pour ceux qui auront besoin de plus d'introduction à Arduino, nous ferons une séance d'entraînement de plus (voir sec. 2.1.1), mais, peut être, déjà pas pour tout le monde. On décidera après la formation des groupes, choix de projets, etc.

A la fin du semestre, ou même après les examens, on se revoit *tous* en amphi pour une *soutenance publique* (4h), où les mentors forment le jury pour donner la note à chaque groupe/projet, et où des autres enseignants seront invités pour donner leur avis.