Si vous avez bien assimilé tout ce qui a été dit précédemment, vous devez être sûr de deux choses. Une LED ne peut pas, en principe, voir son intensité lumineuse varier. Ce composant demande le passage d'un certain courant et ce faisant s'allume avec une certaine intensité, point. Deuxièmement, les broches d'une carte Arduino comme la UNO ne savent prendre que deux états : haut ou bas. Là non plus, il n'y a guère de place pour des demis mesures. Les sorties de la carte sont donc clairement numériques (ou digitales selon vos préférences linguistiques).

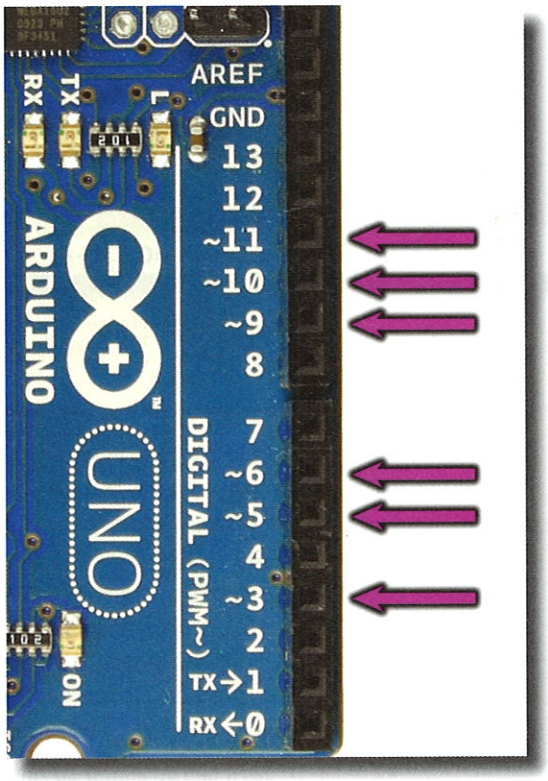
Notre objectif cependant sera ici de faire progressivement s'allumer une LED jusqu'à arriver à son intensité maximale pour ensuite la faire décroître à nouveau jusqu'à s'éteindre totalement. En d'autres termes, nous voulons obtenir un comportement typiquement analogique avec des broches et un composant qui ne le sont pas du tout. Le secret, c'est la PWM.

1. DE L'ANALOGIQUE AVEC DU NUMÉRIQUE, OU PRESQUE

En observant attentivement les inscriptions sur une carte Arduino, vous remarquerez sans doute que certains des numéros associés aux broches sont précédés d'un symbole « ~ ». Ceci indique que ces broches disposent d'une fonctionnalité particulière. En plus de pouvoir être utilisées en entrée ou en sortie comme les autres pour lire ou imposer une tension de 0V (masse) ou +5V, elles peuvent être utilisées avec la fonction `analogWrite()`. Le nom est trompeur, car il ne s'agit pas réellement d'une sortie analogique, mais d'une technique permettant de s'approcher d'un tel comportement.

Cette technique est appelée PWM pour *Pulse-Width Modulation* ou modulation de largeur d'impulsions (MLI). Mais derrière ce terme qui peut paraître très impressionnant se cache un mécanisme relativement simple. Vous avez déjà sans doute regardé au travers d'un ventilateur, une VMC, une hélice d'avion ou n'importe quel dispositif rotatif quelconque en marche. Vous pouvez voir au travers, mais ce que vous percevez n'est pas aussi clair que ce qui se trouve autour. Vous avez l'impression que l'objet est translucide alors qu'en réalité les pales qui tournent sont toujours parfaitement opaques. Il se trouve simplement qu'elles sont séparées les unes des autres par un vide, qui lui est effectivement totalement transparent. C'est la succession de pleins et de vides, très rapide, qui donne l'impression de transparence partielle.

La PWM et les sorties « analogiques » d'une carte Arduino fonctionnent de la même manière. Ces broches ont la capacité d'allumer et d'éteindre une LED de façon si rapide que vous n'arrivez pas, avec



Certaines broches d'une carte Arduino disposent d'une fonction spéciale appelée PWM, mais qui de façon générique sont considérées comme des sorties analogiques. Les numéros des broches en question sont précédés d'un tilde pour les différencier, car les autres broches ne peuvent pas être utilisées de la sorte. Une carte Arduino UNO par exemple possède six de ces broches et une Arduino Mega 2560 en a douze.

vos pauvres yeux, à faire la distinction entre un état et un autre. Ce que vous percevez alors, c'est une intensité lumineuse proportionnelle au temps où la LED est allumée par rapport à celui où elle est éteinte. Cette relation est ce qu'on appelle un **rapport cyclique**. Si la LED est allumée la moitié du temps et éteinte l'autre moitié, le rapport cyclique est de 50%. Un quart contre trois quarts, donne 25%, un dixième contre neuf dixième donne 10%, etc.

Dans ce mode de fonctionnement, la vitesse de clignotement n'est pas importante à partir du moment où elle est suffisante et ce n'est pas ce sur quoi il faut influer pour varier l'intensité de la LED. C'est le rapport cyclique qui détermine l'intensité et c'est précisément ce que permet de faire la fonction **analogWrite()** en prenant en argument le numéro de la broche concernée et une valeur entre 0 et 255 permettant de changer ce rapport : 255 = 100%, 127 = 50%, 64 = 25%, etc.

Mettons sans attendre cela en pratique avec la broche 5 où sera connectée une LED via une résistance de 330 ohms. Notre croquis sera le suivant :

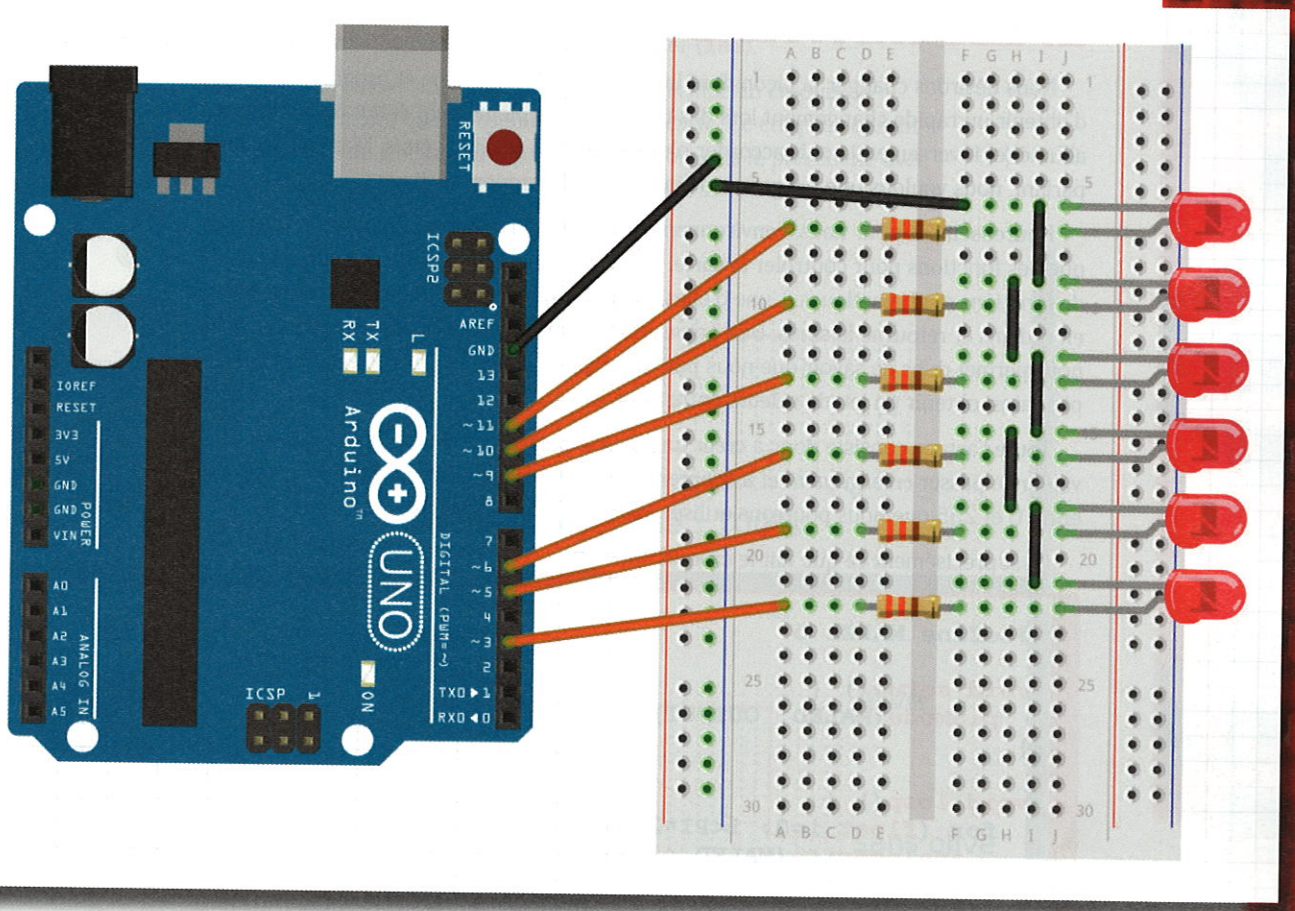
```
#define MALED 5

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

void loop() {
  for(int i=0; i<256; i++) {
    analogWrite(MALED,i);
    delay(5);
  }
  for(int i=255; i>=0; i--) {
    analogWrite(MALED,i);
    delay(5);
  }
}
```

Petite nouveauté ici dès la première ligne. La directive **#define** nous permet de définir une macro. La syntaxe n'est pas la même qu'une déclaration et une initialisation de variable. Cette ligne se lit ainsi : « je définis une macro MALED qui correspond à 5 ». Pas de signe « = » et pas de « ; », il ne s'agit pas d'une instruction, mais d'une macro. À partir de cette ligne, dès que nous utiliserons **MALED**, cela correspondra au fait de spécifier **5**. Le comportement du croquis serait le même en remplaçant cette ligne par **const int MALED=5;**, mais nous économisons ici un tout petit peu de mémoire. Utiliser une macro ou une constante est une question de préférence personnelle plus qu'autre chose (tant que vous n'êtes pas à cours de mémoire). Le logiciel Arduino (le compilateur pour être précis) optimisera de toute façon le croquis au moment de la vérification. Personnellement, j'aime utiliser des macros pour désigner les broches. C'est une question d'habitude.

La partie intéressante réside dans la fonction **loop()**. Nous avons deux boucles **for** parcourant les valeurs de 0 à 255 puis de 255 à 0. Ces valeurs, stockées dans des variables locales nommées **i** sont ensuite utilisées avec **analogWrite()** pour définir le rapport cyclique que nous voulons pour la broche 5.



Notre montage nous permettant de découvrir ces nouvelles fonctionnalités est très similaire au précédent. Nous n'avons cependant que six LEDs correspondant aux 6 sorties « analogiques ».

Après vérification et enregistrement du croquis dans la carte, la LED connectée l'illumine progressivement puis s'éteint doucement avant de recommencer, encore et encore. Vous l'avez compris, ceci n'est pas une réelle sortie analogique, mais une succession d'états discrets (0V et 5V) qui donne l'impression d'avoir affaire à un comportement analogique. Il existe des solutions permettant d'avoir une vraie sortie analogique, via ce qu'on appelle des DAC pour *Digital-to-Analog Converter*. La plupart des cartes Arduino n'en possèdent cependant pas. Seules les Due et les Zero intègrent cette fonctionnalité, mais il reste toujours possible d'utiliser un montage, un module ou un shield disposant d'un DAC si votre projet en a besoin.

Le résultat que nous obtenons ici est acceptable, mais pas pleinement satisfaisant. En effet, la LED semble rester à une intensité importante bien plus de temps qu'elle n'est éteinte alors que notre croquis est pourtant parfaitement fonctionnel. Ceci provient à la fois du fonctionnement même de la LED et de notre perception de la lumière.

En représentant graphiquement la progression des valeurs de `i` dans nos boucles, nous obtenons une forme en dent de scie. Mais pour obtenir une pulsation telle que celle utilisée sur certains ordinateurs pour indiquer une mise en veille, il nous faut quelque chose de plus harmonieux : un sinus.

2. LES MATHS S'EN MÊLENT !

Nous désirons changer la façon dont le rapport cyclique évolue et progresse. L'idée est d'obtenir un rapide changement lors des transitions de pleinement éteint à pleinement allumé (et inversement), mais accorder plus de temps à ces deux états. Mathématiquement parlant, nous voulons une progression sinusoïdale du rapport cyclique.

Heureusement pour nous, l'environnement Arduino met à notre disposition bien plus que des fonctions pour contrôler la carte. Nous avons également sous la main une tripotée de fonctions mathématiques. L'une d'elles, **sin()**, prend en argument une valeur d'angle en radians et retourne le sinus. Une brève remémoration de nos cours de trigonométrie nous permet donc de savoir que nous pouvons passer un argument entre zéro et deux fois pi et ainsi obtenir un jeu de valeurs entre -1 et 1.

Toute l'astuce consiste donc à créer une boucle parcourant un certain nombre de ces valeurs à passer en argument et à adapter le résultat pour obtenir un rapport cyclique entre 0 et 255 que nous pourrons utiliser avec **analogWrite()**.

C'est précisément ce que fait le croquis suivant :

```
#define MALED 5

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

void loop() {
  for (float i=0; i<PI*2; i=i+PI*2/300) {
    analogWrite(MALED,sin(i)*127.5+127.5);
    delay(5);
  }
}
```

Nous utilisons ici un nouveau type de variable. Jusqu'à présent nous n'avons joué qu'avec des valeurs entières, **float** nous permet de déclarer une variable destinée à contenir une valeur en virgule flottante (un nombre à virgule en somme).

Pour obtenir l'ensemble des valeurs qui nous intéressent, nous devons fournir aux appels de la fonction **sin()** une série de valeurs allant de zéro à tau (2 fois pi). Pour ce faire nous utilisons donc une boucle **for** débutant à 0, se poursuivant tant que nous ne sommes pas à tau et s'incrémentant de 1/300 de tau à chaque tour.

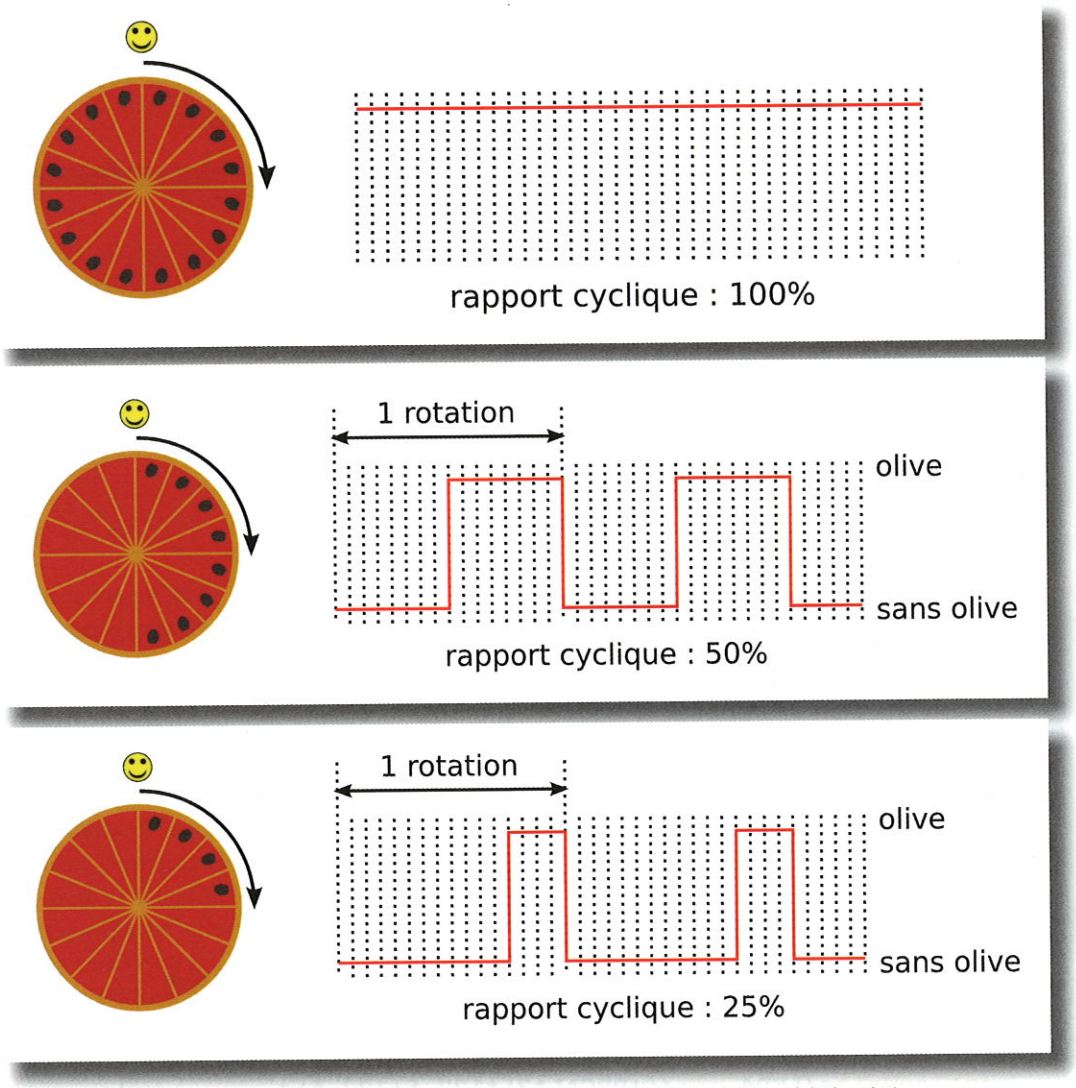
Il nous suffit alors, dans le corps de la boucle, d'appeler la fonction avec la variable contenant la valeur appropriée. Nous ne pouvons pas directement utiliser la valeur retournée, car elle se situe entre -1 et 1. Nous devons donc la multiplier et l'augmenter pour obtenir un résultat entre 0 et 255. Et tout cela est ensuite directement utilisé en argument de la fonction **analogWrite()**.

Notez que, dans un tel calcul, les règles de priorité classiques s'appliquent : d'abord les multiplications et divisions, et ensuite les additions et soustractions. Nous avons bien le résultat retourné par **sin()** multiplié par 127,5, **puis** augmenté de 127,5.

Remarquez également que le calcul porte sur une valeur de type **float**, mais que **analogWrite()** prend en argument un **int**. Comme précédemment avec **random()**,

nous avons là une conversion implicite. Notez toutefois qu'il ne s'agira pas d'un arrondi, mais que la valeur initiale (`float`) est tronquée pour entrer dans un entier (`int`).

Une dernière précision avant de passer à la suite. Lorsque vous vérifiez un croquis, le logiciel Arduino vous indique la mémoire qu'il consommera dans la carte. Ce croquis consommera ici 2304 octets de la mémoire de stockage des programmes alors que le précédent, avec ces deux boucles `for`, n'en consomme que 1310.



La technique appelée PWM consiste à alterner les états haut et les états bas sur des périodes de temps déterminées. On peut aisément illustrer cela avec, comme ici, un disque ou une pizza qui tourne sur elle-même et où la présence d'olive sur certaines parts définit un état. Ce qu'il faut bien comprendre, c'est que la pizza tourne toujours à la même vitesse, mais que c'est l'alternance de parts avec et sans olive qui change : le rapport cyclique.

C'est étrange, le croquis comporte moins de lignes, mais est finalement plus gros ? En effet, l'utilisation d'une fonction mathématique implique que le croquis final, tel qu'enregistré dans la carte, intègrera du code qui n'était jusqu'alors pas nécessaire. Vous n'avez pas de prise sur ce phénomène, le compilateur analyse votre croquis et détermine automatiquement ce qu'il doit

intégrer pour assurer son fonctionnement. Ce n'est pas un problème avec un croquis aussi simple mais, parfois, il peut être intéressant de ne pas utiliser certaines fonctions de calcul, mais plutôt d'embarquer uniquement des résultats dans vos croquis. Comme par exemple les 150 valeurs de rapport cyclique, qui prendraient moins de place que d'intégrer une fonction mathématique complexe comme **sin()**...

3. ET AVEC 6 LEDS ALORS ?

Et si nous combinions tout cela avec le croquis précédent ? Nous disposons de 6 broches sur une carte Arduino UNO qui peuvent utiliser la PWM, nous pouvons donc nous en servir de concert pour créer un effet lumineux intéressant.

Il ne s'agit ici que d'associer les différents concepts que nous venons de voir pour produire un nouveau croquis :

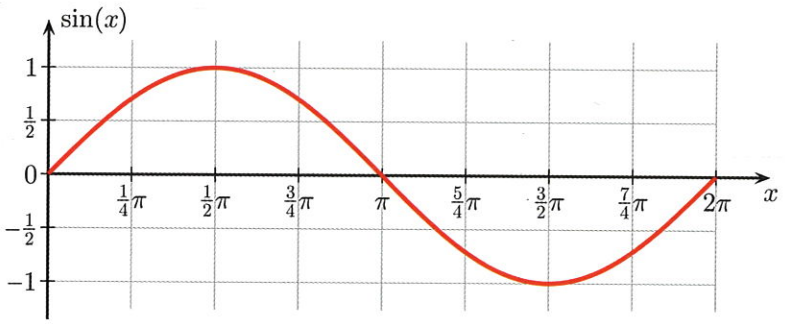
```
// les sorties avec PWM
#define LED1 11
#define LED2 10
#define LED3 9
#define LED4 6
#define LED5 5
#define LED6 3

// une valeur pour décaler le sinus
#define DECAL PI*2/6

// configuration des sorties
void setup() {
  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  pinMode(LED3, OUTPUT);
  pinMode(LED4, OUTPUT);
  pinMode(LED5, OUTPUT);
  pinMode(LED6, OUTPUT);
}

void loop() {
  // parcours des valeurs
  for (float i=0; i<PI*2; i=i+PI*2/1000) {
    // application du rapport cyclique sur chaque sortie
    analogWrite(LED1,sin(i)*127.5+127.5);
    analogWrite(LED2,sin(i+DECAL)*127.5+127.5);
    analogWrite(LED3,sin(i+DECAL*2)*127.5+127.5);
    analogWrite(LED4,sin(i+DECAL*3)*127.5+127.5);
    analogWrite(LED5,sin(i+DECAL*4)*127.5+127.5);
    analogWrite(LED6,sin(i+DECAL*5)*127.5+127.5);
  }
}
```

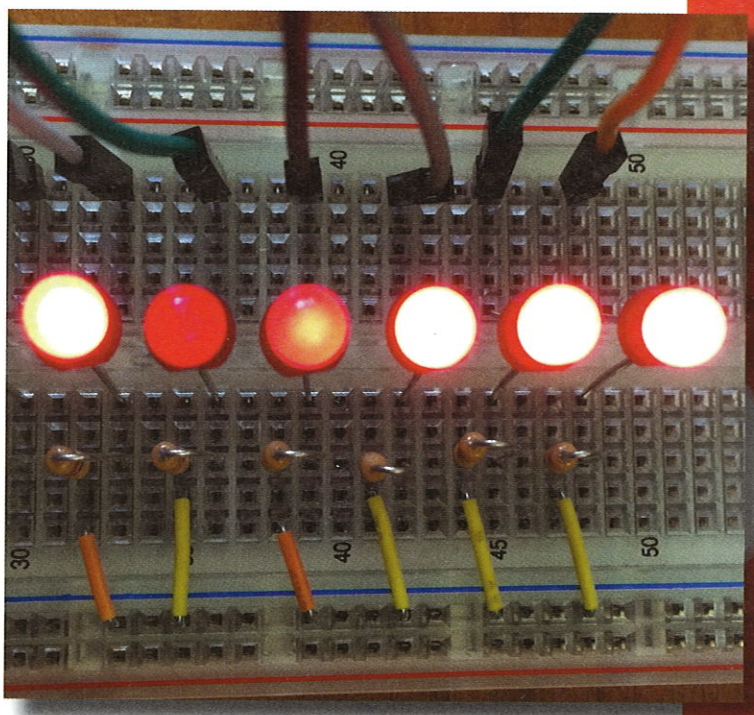
Je vais me passer, cette fois, de donner des explications, car cela reviendrait à me répéter inutilement. La seule subtilité ici réside dans la définition de la macro **DECAL** qui, comme vous pouvez le constater, peut contenir un calcul. Le sinus de pi fois 2 donne zéro, comme le sinus de zéro ou encore de pi fois 4, pi fois 6, pi fois 8, etc. Pour décaler le sinus au travers



Pour faire pulser une LED de façon fluide et agréable, l'appel à des fonctions mathématiques est une solution donnant de bons résultats, mais qui implique de se remémorer quelques cours, parfois anciens pour certains, de trigonométrie et des rapports entre un cercle, les sinus et les radians. De plus, en utilisant de telles fonctions dans vos croquis vous en augmenterez la taille dans la mémoire de la carte Arduino.

des LEDs, il nous suffit d'ajouter un sixième de la valeur de tau (pi x 2) à la valeur passée à `sin()` pour chaque rapport cyclique.

Nous obtenons ainsi une « vague » qui parcourt les six LEDs ou, autrement dit, une représentation lumineuse du sinus.



Le résultat obtenu avec notre dernier croquis fera pulser les différentes LEDs alternativement, donnant ainsi un effet de mouvement progressif et tout en douceur.

CONCLUSION

Nous venons ici de découvrir les sorties dites analogiques d'une carte Arduino, qui sont en réalité le résultat d'une technique appelée PWM. Celle-ci nous permet de régler l'intensité lumineuse d'un composant qui ne le permet pas normalement. La PWM peut également être utilisée pour régler la vitesse d'un moteur de la même manière. Plutôt que d'utiliser la persistance rétinienne propre à nos yeux, c'est un effet mécanique ou plutôt cinétique qui est utilisé. Le moteur change de vitesse, mais c'est finalement le rapport entre le nombre d'à-coups et d'absence d'à-coup qui provoque cet effet.

Comme précédemment, je vous invite à jouer avec ces croquis et les tripatouiller dans tous les sens pour bien en appréhender le fonctionnement. Parmi les modifications et évolutions possibles, nous avons en premier lieu l'intégration de la fonction `millis()` pour remplacer `delay()`. Remplacer les boucles `for` par des équivalents avec `while` est également un excellent exercice même s'il est d'usage pour ce genre de choses de préférer `for`, car plus compact et clair.

Vous pouvez enfin, avec le dernier croquis, réduire le nombre de tours dans la boucle et donc le nombre de changements de valeurs du rapport cyclique, en remplaçant la division par 1000 par une division par 10, par exemple et en ajoutant un `delay(100)`. L'effet sera sensiblement différent, moins fluide. Pourquoi ? ■