Comment Otto se déplace ?

J. Lemaire

(Pierrefeu Mai 2021)

1 Motivations

Il y a quelques temps, ma dernière petite fille (8 ans) a fait part de son enthousiasme, suite à une initiation à la robotique qu'elle venait de suivre dans son école. Ceci m'a incité à lui offrir un robot OttoDIY+[1], pour lui permettre d'en savoir un peu plus… et éventuellement épater les copines !

Mais, tout en l'aidant à l'assembler, j'avoue avoir été totalement bluffé par ce petit Robot, réellement génial par sa simplicité et ses possibilités, notamment par sa démarche n'utilisant que 4 micros servos.

Ayant été enseignant en environnement informatique[2] dans une autre vie (je suis maintenant à la retraite), je me suis pris au jeu et j'ai essayé de comprendre son fonctionnement.

Par ailleurs, ayant constaté comme d'autres utilisateurs que le firmware proposé pour commander OttoDIY+ en Bluetooth était limite pour un Arduino Nano qui ne dispose que de 2K de mémoire SRAM, je me suis demandé s'il n'était pas possible d'optimiser un peu le code C++ fourni, en me limitant ici au seul problème des mouvements de base d'Otto, pour simplifier.

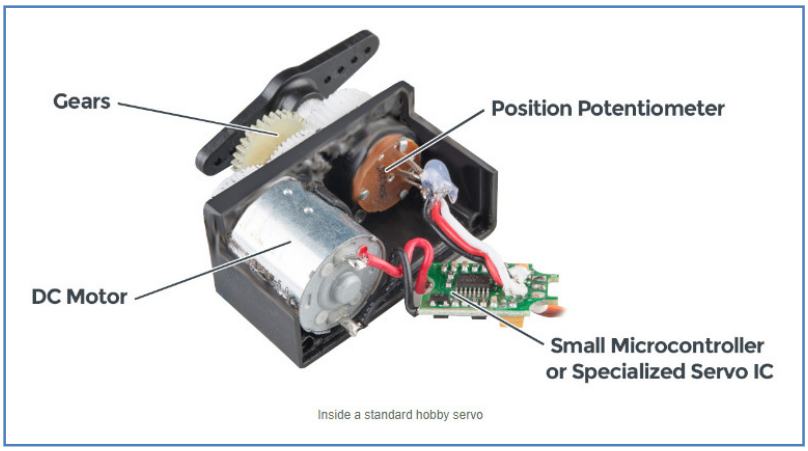
Ce document expose ce petit travail. Mais avant d'entrer dans les détails, je tiens à bien préciser que je ne cherche nullement ici à critiquer le remarquable travail des concepteurs d'Otto , mais juste à en comprendre son fonctionnement « locomoteur », évoquer les questions que je me suis posées à l'occasion, et très modestement, présenter les réponses apportées. Ce faisant, j'ai bien conscience du caractère très limité de cette étude eu égard à toute la gestuelle d'Otto !

[1] Dénommé Otto par la suite, pour simplifier.
[2] Département Informatique de l'Institut Universitaire de Technologie de Nice – Côte d'Azur, notamment.

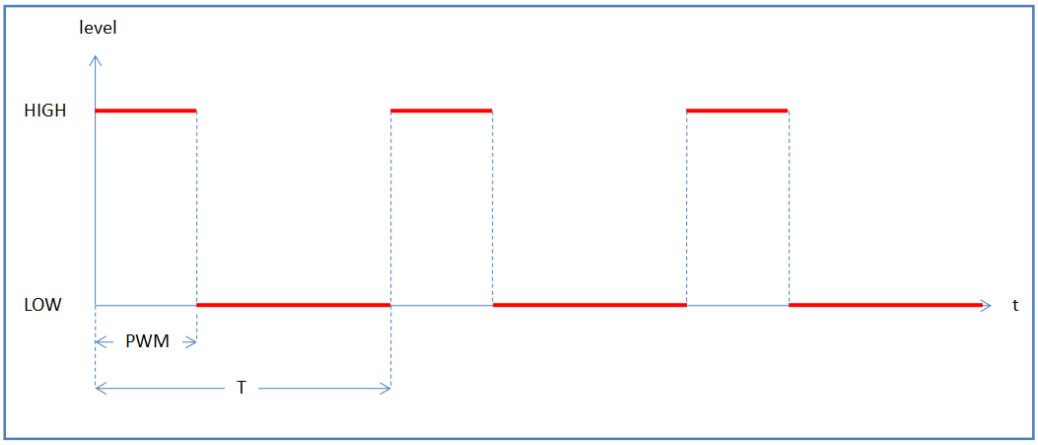
2 Servos

Les déplacements d'Otto sont réalisés par 4 micro-servos de type MG90S[3].

Ils sont chacun constitués d'un petit moteur à courant continu qui, moyennant une série d'engrenages, peut faire tourner un axe d'un angle compris approximativement entre 0 et 180°. Un circuit intégré pilote le moteur, le contrôle de l'angle étant assuré par un potentiomètre[4] :



Ce circuit intégré peut être commandé par un signal PWM[5] de fréquence 50Hz (période T = 20ms), dont la valeur, notée PWM également pour simplifier, sera définie par la durée de l'état HAUT sur chaque période :

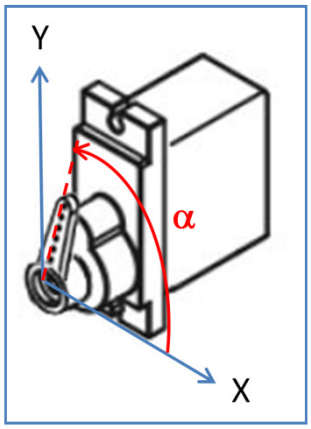


Par défaut pour un servo :

- PWM_{min} = 544µs => angle ≅ 0°
- PWM_{max} = 2400µs => angle ≅ 180°

L'angle **α** de rotation, également appelé **position** du servo, sera mesuré comme suit, le demi-bras étant monté de telle sorte que cet angle puisse varier entre 0 et 180° :

[3] Micro Servo Motor MG90S - Tower Pro (electronicoscaldas.com)
[4] Servos Explained - SparkFun Electronics
[5] Pulse-width modulation - Wikipedia



Pour commander un servo avec un Arduino et sur n'importe quelle broche digitale, il suffit d'utiliser la bibliothèque **Servo** réalisée par Michael Margolis[6], qui définit la classe Servo :

```
class Servo
{
public:
  Servo();
  uint8_t attach(int pin);           // attach the given pin to the next free channel, sets
pinMode, returns channel number or 0 if failure
  uint8_t attach(int pin, int min, int max); // as above but also sets min and max values for
writes.
  void detach();
  void write(int value);             // if value is < 200 its treated as an angle, otherwise as
pulse width in microseconds
  void writeMicroseconds(int value); // Write pulse width in microseconds
  int read();                        // returns current pulse width as an angle between 0 and 180
degrees
  int readMicroseconds();            // returns current pulse width in microseconds for this servo
(was read_us() in first release)
  bool attached();                   // return true if this servo is attached, otherwise false
private:
   uint8_t servoIndex;               // index into the channel data for this servo
   int8_t min;                       // minimum is this value times 4 added to MIN_PULSE_WIDTH
   int8_t max;                       // maximum is this value times 4 added to MAX_PULSE_WIDTH
};
```

Voici un exemple d'utilisation dans un sketch Arduino, pour un servo connecté sur la broche 2 :

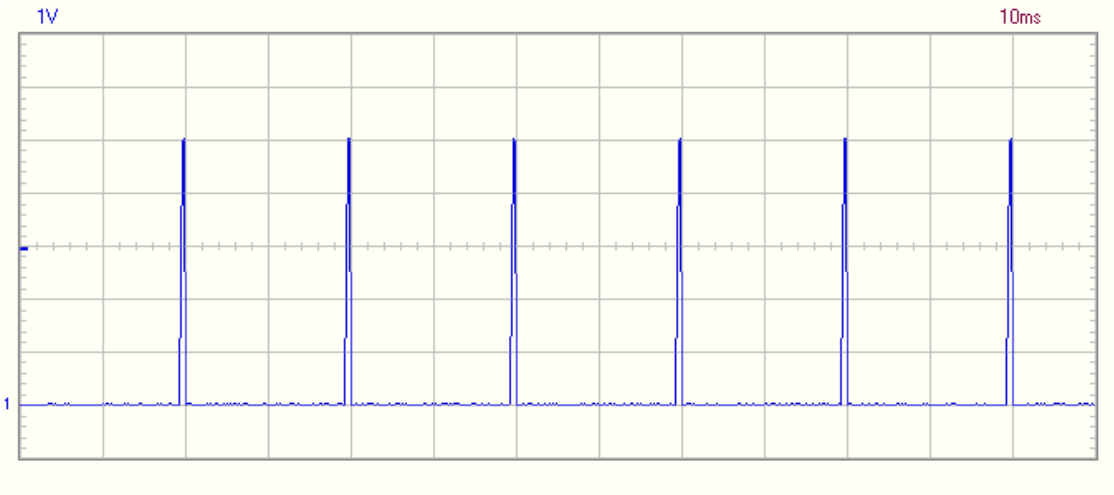
```
#include <Servo.h>

Servo s;          // Servo object

void setup()
{
  s.attach(2);  // Attaches the servo to pin 2
  s.write(0);   // Turns 0°
}

void loop()
{
}
```

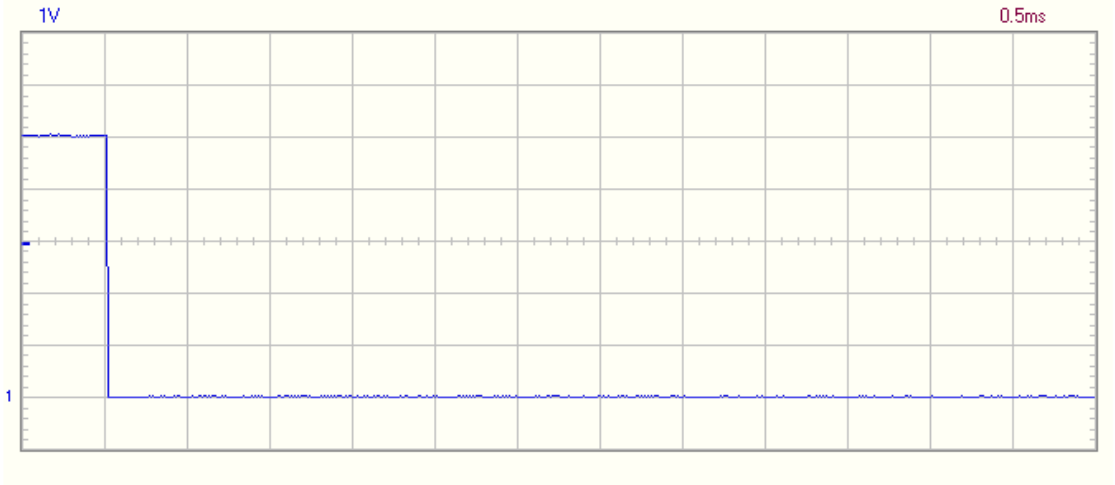
Dans le cas d'un Arduino Nano, on peut utiliser 12 servos en parallèle et l'implémentation de cette bibliothèque met à profit **Timer1** pour générer de tels signaux, en mode interruption[7]. Voici les signaux observés avec un oscilloscope[8] sur la broche 2, quand on lance le sketch précédent :

[6] GitHub - arduino-libraries/Servo: Servo Library for Arduino

[7] On rappelle qu'un Arduino Nano, avec un ATMega328P, dispose de 3 timers programmables, **timer0** (8bits), **timer1** (16bits) et **timer2** (8bits), qui contrôlent par défaut les sorties PWM sur les broches 5/6 (avec timer0 à 980Hz), 9/10 (avec timer1 à 490Hz) et 3/11 (avec timer2 à 490Hz), celles-ci pouvant être générées en appelant la méthode **analogWrite(…)**.

[8] Avec zoom temporel.





Ce mode de commande par interruption permet d'envoyer un signal PWM à **50Hz** sur n'importe quelle broche à sortie digitale et de faire tourner des servos en parallèle, sans bloquer le microcontrôleur. Par contre, la bibliothèque **Servo** ne permet pas de contrôler leur vitesse de rotation[9] ni même de les arrêter[10] sur un angle précis. Notons enfin qu'un servo n'est pas aussi précis qu'un moteur pas à pas : il est donc souvent nécessaire de le trimer pour faire coïncider l'angle effectif du bras avec l'angle transmis à la méthode **write**, par exemple pour 90° ; cette possibilité n'est pas non plus prise en compte dans la bibliothèque **Servo**. Par contre, on peut redéfinir les valeurs PWM_{min} et PWM_{max}, ce qui est une autre façon, plus juste mais pas toujours possible, de résoudre le problème.

Le sketch suivant montre que la méthode **read** renvoie le dernier angle transmis à **write** et non la position courante du servo :

```
#include <Servo.h>

Servo s1,s2;

void setup()
{
  s1.attach(2);
  s2.attach(3);
  s1.write(0);
  s2.write(0);
  Serial.begin(115200);
```

[9] De l'ordre de 300ms pour 180° avec une alimentation 5V.
[10] **detach()** arrête la rotation, sans maintenir un angle.

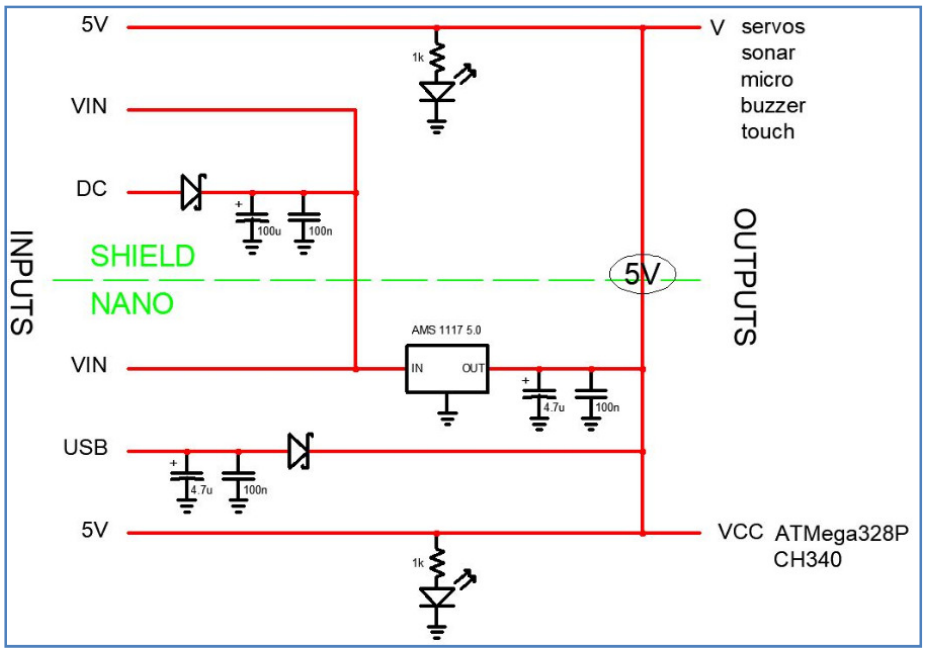
```
  delay(1000);
  Serial.println("start");
  Serial.println(millis());
  Serial.print("s1="); Serial.print(s1.read());
  Serial.print(" s2="); Serial.println(s2.read());
  s1.write(180);
  s2.write(180);
}

void loop()
{
  Serial.println(millis());
  Serial.print("s1="); Serial.print(s1.read());
  Serial.print(" s2="); Serial.println(s2.read());
  delay(10);
}
```

Affichage sur le moniteur série :

```
start
999
s1=0 s2=0
1000
s1=180 s2=180
1010
s1=180 s2=180
1021
s1=180 s2=180
1032
s1=180 s2=180
…
```

Pour terminer cette petite présentation, il convient de rappeler qu'un servo, même « micro », peut engendrer un courant non négligeable : celui-ci peut dépasser 200mA, en fonctionnement sous contraintes. Dans le cas d'Otto, il est donc vivement conseillé d'alimenter séparément ses 4 micro-servos, et non uniquement via un port série USB de type 2, qui ne peut pas fournir plus de 500mA. Cela protègera aussi le microcontrôleur Arduino Nano et les autres composants du robot contre les variations de courant, quand on l'alimente via son entrée VIN, même si son régulateur 5V (AMS 1117 5.0) semble ici assez tolérant et réactif.

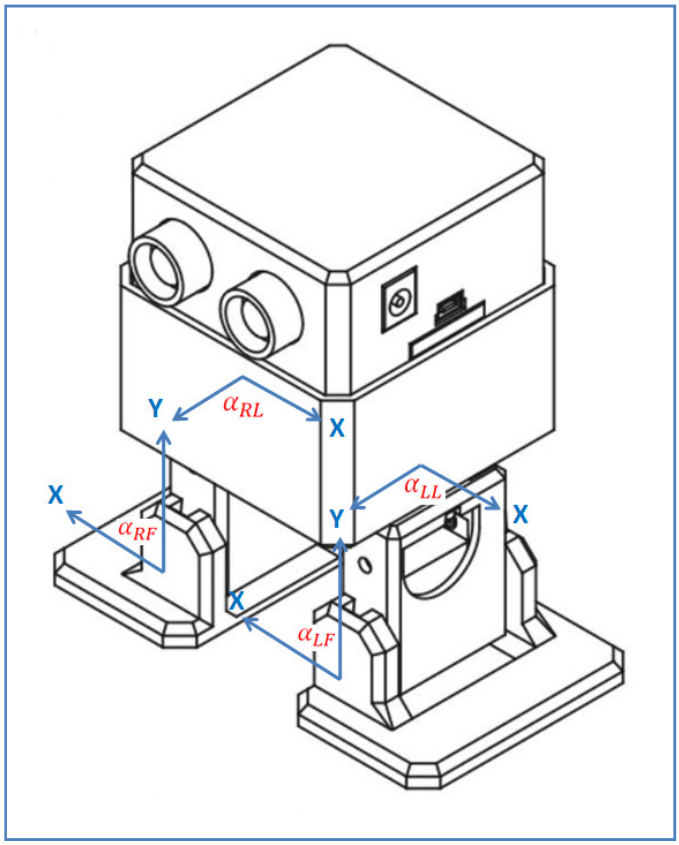
Pour bien comprendre et mettre en œuvre une solution efficace, voici le schéma du circuit d'alimentation par défaut d'Otto :



3 La solution retenue dans Otto

La convention précédente de mesure de l'angle de rotation d'un servo conduit à ceci, pour les 4 servos d'Otto, montés conformément aux instructions d'assemblage :

- α_{LL} => jambe gauche
- α_{RL} => jambe droite
- α_{LF} => pied gauche
- α_{RF} => pied droit

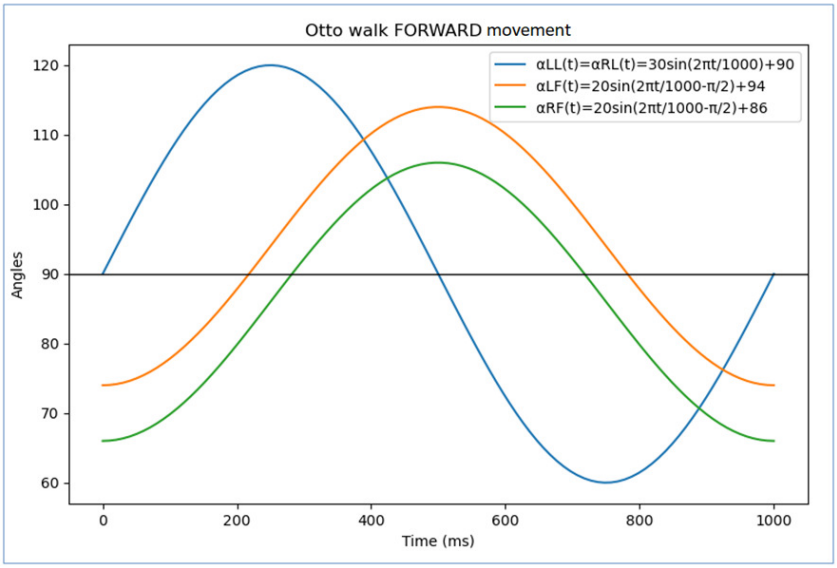


Ainsi :

- $\alpha_{LL} > 90°$ ou $\alpha_{RL} > 90°$ ⇔ jambes tournées vers la droite
- $\alpha_{LF} > 90°$ ⇔ pied gauche abaissé (vers l'extérieur)
- $\alpha_{RF} < 90°$ ⇔ pied droit abaissé (vers l'extérieur)

Un mouvement d'Otto pendant une durée T ms sera précisé par la donnée de 4 fonctions angulaires, encore notées $\alpha_{LL}, \alpha_{RL}, \alpha_{LF}$ et α_{RF} pour simplifier, définies sur l'intervalle $[0, T]$ et à valeurs dans $[0,180]$.

Voici par exemple les fonctions appelées par la méthode **walk(1, 1000, FORWARD)**, pour faire avancer Otto pendant 1000ms, avec un pas à gauche puis un pas à droite :

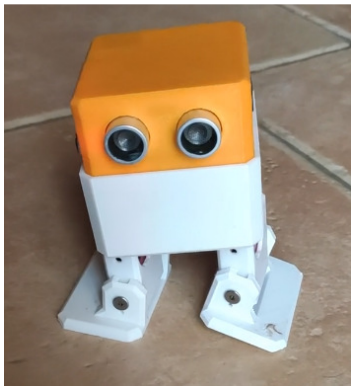


[11]

Ici, pour $0 \leq t \leq 1000$:

- $\alpha_{LL}(t) = \alpha_{RL}(t) = 30 \sin\left(\frac{2\pi t}{1000}\right) + 90$
- $\alpha_{LF}(t) = 20 \sin\left(\frac{2\pi t}{1000} - \frac{\pi}{2}\right) + 94$
- $\alpha_{RF}(t) = 20 \sin\left(\frac{2\pi t}{1000} - \frac{\pi}{2}\right) + 86$

Celles-ci provoquent un avancement du robot avec un balancement fluide où il s'appuie successivement sur sa jambe gauche puis sur sa jambe droite, tout en les faisant tourner vers la droite, puis vers la gauche (c'est la résistance du sol sous le pied à plat qui permet cet avancement) :



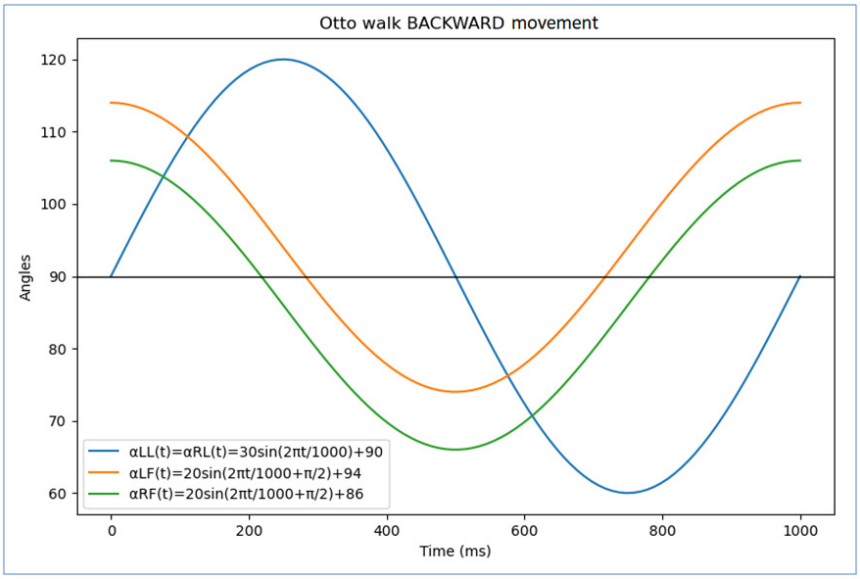
(Clic-G pour voir la vidéo)

---

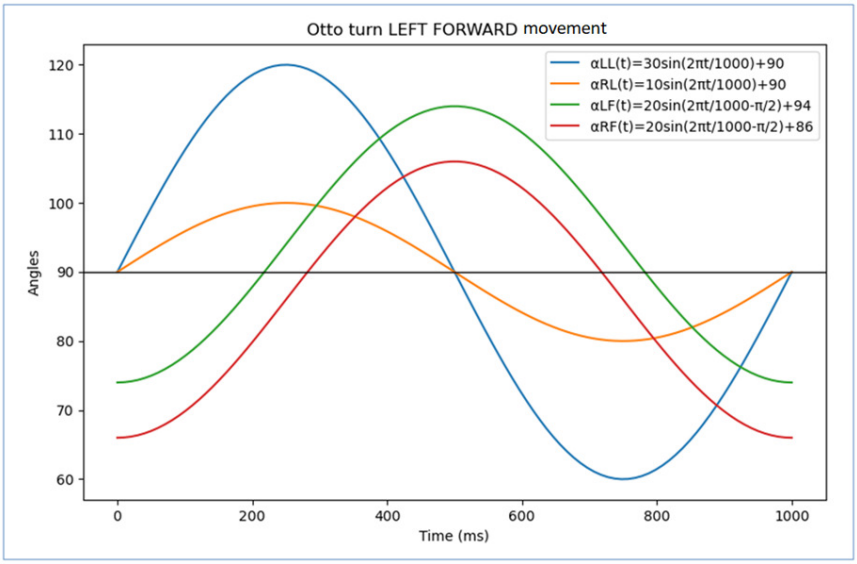
[11] Graphique réalisé avec **WinPython** (WinPython download | SourceForge.net) et le programme suivant :

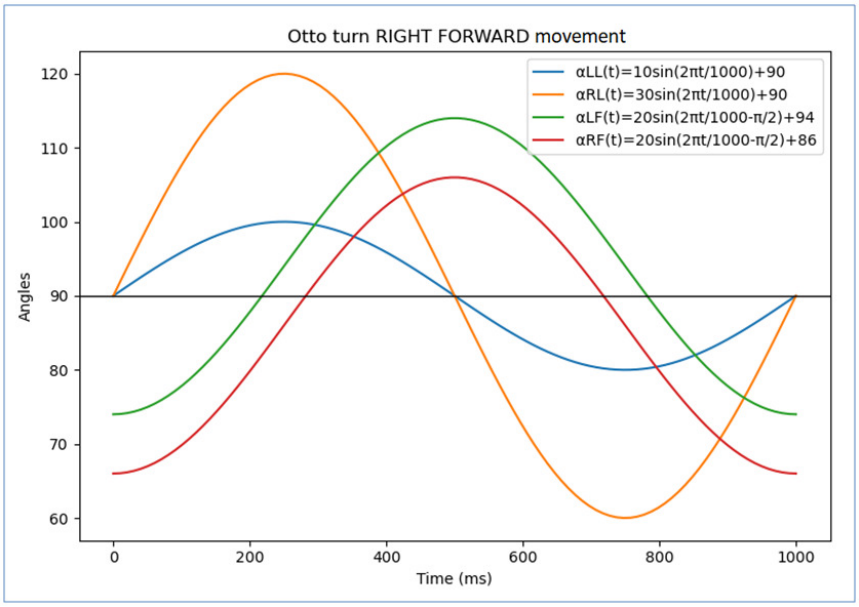
```
# Plot.py

# Libraries
from numpy import pi, sin, linspace
import matplotlib.pyplot as plt

# Time values
t = linspace(0, 1000, 100)

#Plots
plt.plot(t, 30*sin(2*pi*t/1000)+90, label="αLL(t)=αRL(t)=30sin (2πt/1000)+90")
plt.plot(t, 20*sin(2*pi*t/1000-pi/2)+94, label="αLF(t)=20sin (2πt/1000-π/2)+94")
plt.plot(t, 20*sin(2*pi*t/1000-pi/2)+86, label="αRF(t)=20sin (2πt/1000-π/2)+86")
plt.axhline(y=90, linewidth=1, color='k')
plt.xlabel("Time (ms)")
plt.ylabel("Angles")
plt.title("Otto walk FORWARD Movement")
plt.legend()
plt.show()
```

Pour le faire reculer, il suffit d'inverser les déphasages des pieds :

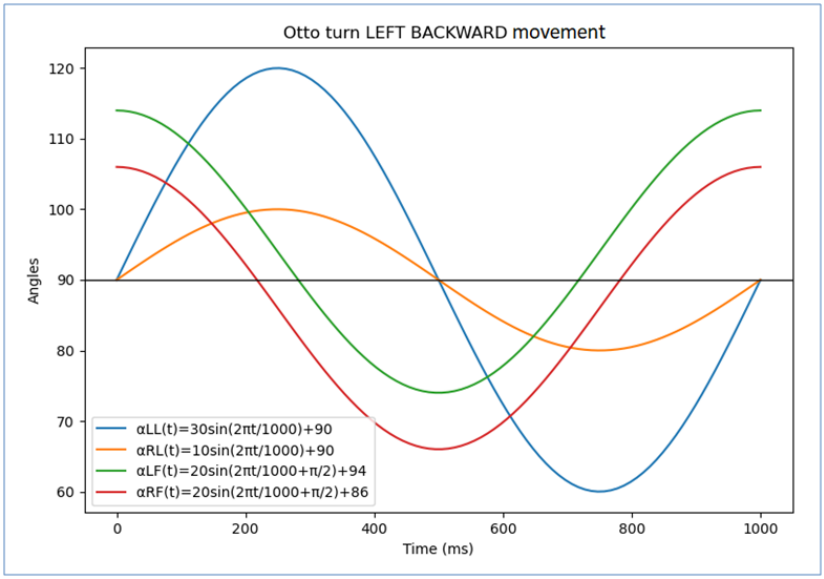


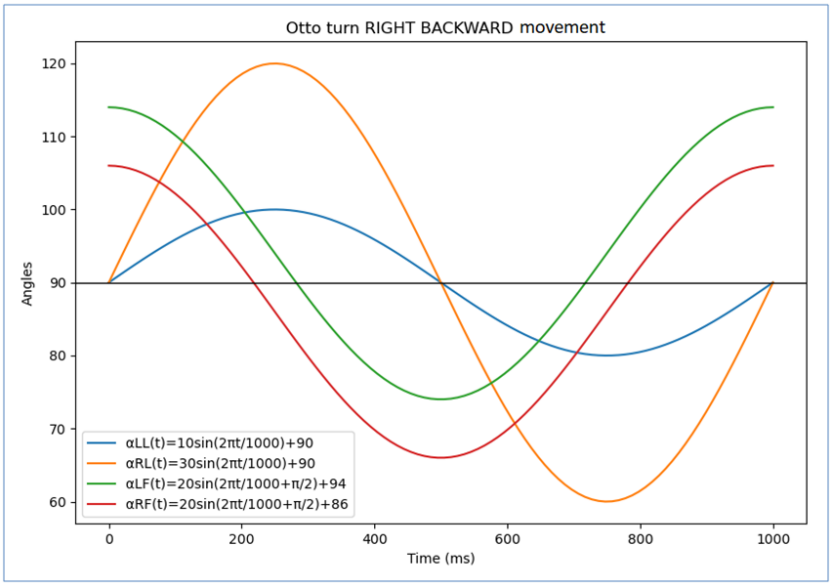
Pour le faire tourner en avançant, il suffit de réduire l'amplitude de la rotation de la jambe, du coté opposé vers lequel on veut tourner ; par exemple :



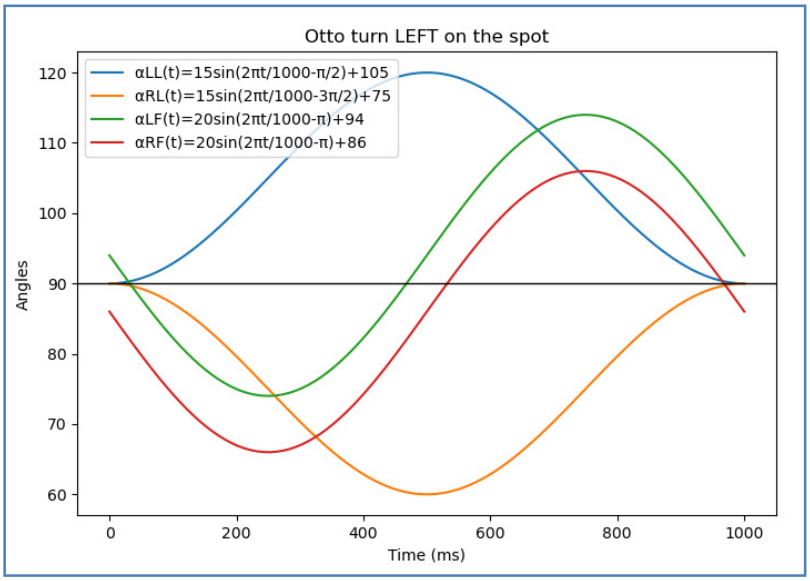


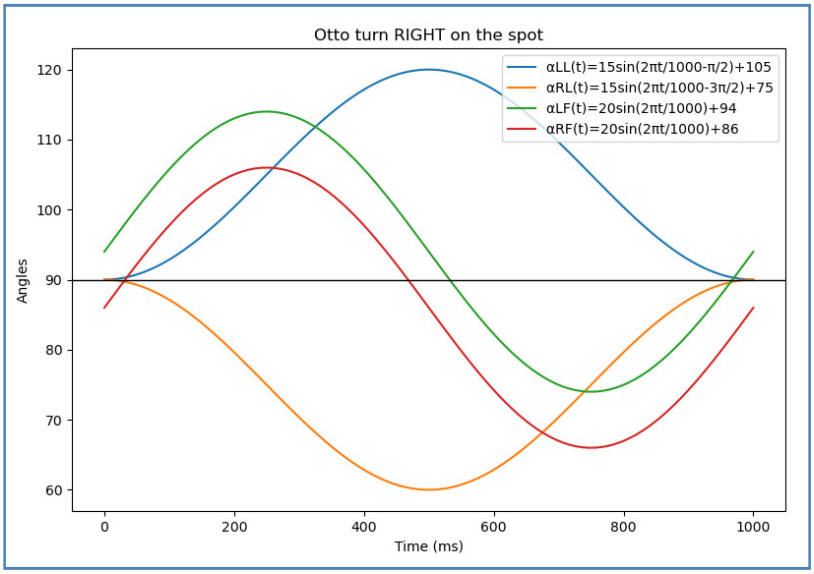
On peut dès lors imaginer comment le faire tourner, en reculant cette fois :





Enfin, si on veut le faire tourner sur place, il suffit d'engendrer un mouvement contradictoire des 2 jambes, comme pour un engin à chenilles :





Toutes ces fonctions angulaires sont de la forme :

$$\alpha(t) = a \sin\left(\frac{2\pi t}{T} + \varphi\right) + b$$

Avec

- a = amplitude (°)
- b = offset (°)
- φ = phase (rad)[12]
- T = durée (ms)

Ayant constaté que plusieurs autres mouvements pouvaient également se réaliser avec de telles fonctions (**turn**, **updown**, **swing**, **tiptoeSwing**, **jitter**, **ascendingTurn**, **moonwalker**, **crusaito**, **flapping**)[13], les concepteurs d'Otto ont utilisé des objets « oscillateurs », instances de la classe **Oscillator**[14], quasiment identique à la classe **ServoOsc** définie dans la bibliothèque du même nom[15]. Elle a été développée initialement par **Juan González Gómez**, dans le cadre de sa thèse de Ph.D. en robotique, **Robótica Modular y Locomoción: Aplicación a Robots Ápodos (2008)**[16], qui a clairement démontré toute l'importance de ce concept, avec une mise en œuvre Arduino[17], utilisée dans les premières versions d'Otto[18]:

```
class Oscillator
{
  public:
    Oscillator(int trim=0) {_trim=trim;};
    void attach(int pin, bool rev =false);
    void detach();

    void SetA(unsigned int A) {_A=A;};
    void SetO(unsigned int O) {_O=O;};
    void SetPh(double Ph) {_phase0=Ph;};
    void SetT(unsigned int T);
    void SetTrim(int trim){_trim=trim;};
    int getTrim() {return _trim;};
```

[12] Radians.
[13] Cf. **Otto9.h** et **Otto9.cpp** dans GitHub - OttoDIY/OttoDIYLib: Latest Libraries for Otto DIY robots
[14] Cf. **Oscillator.h** et **Oscillator.cpp** dans GitHub - OttoDIY/OttoDIYLib: Latest Libraries for Otto DIY robots
[15] GitHub - fitzterra/ServoOsc: Servo oscillator library for Arduino
[16] Juan Gonzalez:Tesis - WikiRobotics (iearobotics.com)
[17] GitHub - Obijuan/ArduSnake: Arduino modular snake robots library. Generate the locomovement of snake robots easily!
[18] GitHub - bqlabs/zowi: An open-source and fully hackable biped robot.

```
    void SetPosition(int position);
    void Stop() {_stop=true;};
    void Play() {_stop=false;};
    void Reset() {_phase=0;};
    void refresh();

  private:
    bool next_sample();

  private:
    //-- Servo that is attached to the oscillator
    Servo _servo;

    //-- Oscillators parameters
    unsigned int _A;  //-- Amplitude (degrees)
    unsigned int _O;  //-- Offset (degrees)
    unsigned int _T;  //-- Period (miliseconds)
    double _phase0;   //-- Phase (radians)

    //-- Internal variables
    int _pos;         //-- Current servo pos
    int _trim;        //-- Calibration offset
    double _phase;    //-- Current phase
    double _inc;      //-- Increment of phase
    double _N;        //-- Number of samples
    unsigned int _TS; //-- sampling period (ms)

    long _previousMillis;
    long _currentMillis;

    //-- Oscillation mode. If true, the servo is stopped
    bool _stop;

    //-- Reverse mode
    bool _rev;
};
```

Cette classe **Oscillator** est fondamentalement associée à un servo qu'elle peut faire « osciller ». Elle permet de définir les paramètres a, b, φ et T d'une fonction α de la forme précédente, en appelant les méthodes **SetA**, **SetO**, **SetPh** et **SetT**. La méthode **refresh** provoque la mise à jour du mouvement obtenu en appliquant cette fonction ; il suffit de l'appeler périodiquement dans la boucle **loop** et cet appel n'est pas bloquant. On peut aussi trimer le servo en appelant la méthode **SetTrim**[19], l'arrêter à tout moment, en appelant la méthode **Stop** et réduire sa consommation en appelant la méthode **detach**.

Du point de vue de ses mouvements, un objet **Otto**, ici instance de la classe **Otto9**[20], est donc essentiellement défini par les 4 oscillateurs stockés dans le tableau :

```
Oscillator servo[4];
```

Toutes les fonctions de mouvement appellent[21] in fine la méthode **oscillateServos**, en lui passant des tableaux contenant les valeurs appropriées des paramètres précités ; après les avoir pris en compte, cette dernière fait appel à la méthode **refresh** des 4 oscillateurs[22] :

```
void Otto9::oscillateServos(int A[4], int O[4], int T, double phase_diff[4], float cycle=1) {
  for (int i=0; i<4; i++) {
    servo[i].SetO(O[i]);
    servo[i].SetA(A[i]);
    servo[i].SetT(T);
    servo[i].SetPh(phase_diff[i]);
  }
  double ref=millis();
   for (double x=ref; x<=T*cycle+ref; x=millis()) {
     for (int i=0; i<4; i++) {
```

[19] On lui passera la valeur de l'angle (lue dans l'EEPROM) résultant de la procédure de calibration du robot.
[20] Cf. **Otto9.h** dans GitHub - OttoDIY/OttoDIYLib: Latest Libraries for Otto DIY robots
[21] Via la méthode **_execute**
[22] Cf. **Otto9.cpp** dans GitHub - OttoDIY/OttoDIYLib: Latest Libraries for Otto DIY robots

```
            servo[i].refresh();
        }
    }
}
```

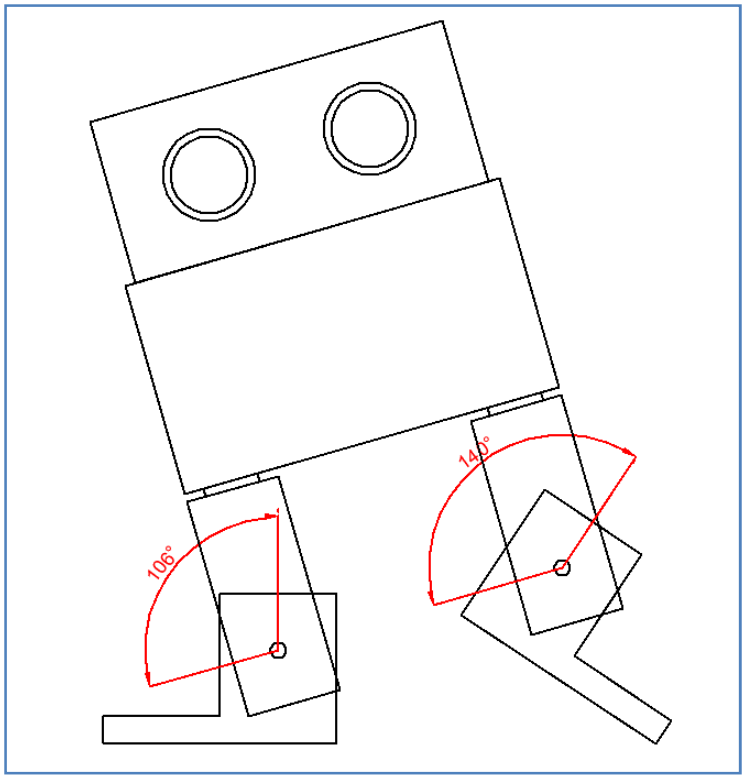
Tout ceci implémente donc de manière très claire cette notion d'oscillation, centrale dans la thèse de **Juan González Gómez**.

Sans vouloir prétendre corriger cette mise en œuvre, il est tentant toutefois de mieux répartir les différentes fonctionnalités considérées ici :

- La notion de mouvement d'abord, qui pourrait utiliser d'autres fonctions que des oscillations, avec des possibilités d'échantillonnage, d'interruption et de reprise, le tout dans un fonctionnement non bloquant ;
- Son application à un servo, en étendant par héritage les fonctionnalités de la classe Servo, avec une possibilité de trim, comme dans OttoDIYLib ;
- Le fonctionnement de plusieurs servos en parallèle ;
- La mise en œuvre de ceci dans Otto, avec la prise en compte de tous les mouvements évoqués précédemment, à différentes vitesses et de manière fluide ;
- La commande d'Otto via un port série et en particulier celui créé par son module Bluetooth.

C'est ce que nous allons tenter de faire dans le paragraphe suivant.

Mais avant de l'aborder, nous souhaiterions poser une question aux concepteurs d'Otto : comment ont été déterminés les paramètres des différents mouvements oscillatoires, et notamment ceux des pieds ? Par exemple, si on considère la méthode **walk** où les jambes sont toujours parallèles, on peut calculer l'angle d'inclinaison d'un pied pour que l'autre soit à plat et ceci peut se faire graphiquement avec un logiciel de dessin (Autocad ici) et une représentation à l'échelle d'Otto :



Ainsi, lorsque le pied droit est relevé avec un angle de 106° (quand les 2 jambes ont un angle de 90° et sont donc écartées au maximum), le pied gauche devrait être abaissé avec un angle de 140° pour que le pied droit soit à plat. Or, ce dernier est fixé à 114° dans l'oscillation. Pour quelle raison ?

4 Classes Movement, ServoE et TServoE

4.1 Fonctions de mouvement

Pour faire varier l'angle d'un servo dans le temps, on a vu qu'il suffisait de définir une fonction α sur un intervalle $[0, T]$; si le mouvement débute au temps t_0, l'angle sera égal à $\alpha(t)$ au temps $t_0 + t$.

Cette fonction α sera généralement caractérisée par des **paramètres** ; c'est le cas, par exemple, pour une oscillation, précisée par les paramètres a, b, φ et T :

$$\alpha(t) = a \sin\left(\frac{2\pi t}{T} + \varphi\right) + b \text{ pour } 0 \leq t \leq T$$

En C++, si les temps sont évalués en millisecondes, on peut donc naturellement considérer le type **<double(double)>** pour une fonction de mouvement.

Si l'on dispose d'une version au moins égale à 11 de ce langage, il y a une façon très directe pour implémenter de manière générique une telle fonction, avec des paramètres : il suffit de considérer une **fonction Lambda** dont les paramètres de la liste de capture (entre [ ]) correspondent aux paramètres précités, transtypée avec **std::function** pour obtenir in fine une fonction du type recherché ; par exemple, pour une oscillation :

```
std::function<double(double)> [a,b,phi,T](double t){return a*sin(2*PI*t/T+phi)+b;};
```

Malheureusement ce puissant mécanisme ne semble pas possible avec un Arduino Uno ou Nano, pour lesquels l'IDE Arduino, qui utilise le compilateur avr-G++7.3.0, ne connaît pas **std::function**. Cette restriction s'explique par la très faible mémoire SRAM disponible sur leur processeur **ATMega328P** : 2KB seulement ![23]

On a donc mis en œuvre une autre technique, moins compacte mais beaucoup plus économe en ressources processeur, basée sur une classe de fonctions paramétriques et un pointeur sur l'une d'entre elles. Ces fonctions, bien que classiques, ressemblent à des fonctions Lambda.

*Classe **Movement** (fichiers **Movement.h** et **Movement.cpp**) :*

```
#define TS 20            // Sampling period(ms) to update Movements

enum StateMovement {active, stopped, inactive};

class Movement
{
  typedef double (Movement::*PtrMovementF)(double);
  // Type of pointer on a Movement function

  public:
    // Constructor
    Movement();

    // Selection methods (NB : speed (unit/s) >0)
    void selAff(double a, double b, double T, unsigned nRep = 0);
    void selOsc(double a, double b, double phi, double  T, unsigned nRep = 0);

    // Access to private fields
    double T() const {return _T;}
    void setT(double T);
    StateMovement stateMov() const {return _stateMov;}
    void setStateMov(StateMovement stateMov) {_stateMov = stateMov;}
```

[23] Les libraires Arduino STL, telles que **ArduinoSTL** et **StandardCplusplus,** ne prennent pas non plus en charge **std ::function**. A noter néanmoins l'existence d'une librairie STL qui implémente **std::function** (GitHub - fopeczek/arduino-function_objects), mais pour **platformIO**. Préférant rester dans le cadre de l'IDE Arduino, nous ne l'avons pas testée.

```
    double operator()(double t) {return ((*this).*_pMovF)(t);} // Selected function

    // Sampling with TS period
    void start();
    bool update();
    void stop();
    double val() const {return _val;}


  private:
    // Movement functions
    double Aff(double t) {return _a*(1-t/_T)+_b*t/_T;}       // Affine
    double Osc(double t) {return _a*sin(2*PI*t/_T+_phi)+_b;} // Oscillation

    // Parameters
    double _a;
    double _b;
    double _phi; // (radians)
    double _T;   // Period(ms) > 0 (no control)

    // Other fields
    PtrMovementF _pMovF;        // Pointer on the selected function
    StateMovement _stateMov;    // State Movement
    unsigned int _nRep;         // Number of repetitions
    unsigned int _iUse;         // Movement use number (0, 1, ..., _nUse)
    unsigned long _startTime, _endTime, _previousTime, _stopTime;
    double _val;                // Sampled value
};
```

Elle n'implémente ici que 2 fonctions de mouvement, fournies à titre d'exemple, qui seront toutes deux utilisées plus loin :

- $\alpha(t) = a(1 - \frac{t}{T}) + b\frac{t}{T}$ (**affinité**)
- $\alpha(t) = a\sin\left(\frac{2\pi t}{T} + \varphi\right) + b$ (**oscillation**)

Mais il est très facile d'en ajouter d'autres !

Pour sélectionner l'une d'entre elles, il suffit d'appeler la méthode **Sel…** correspondante, en lui passant ses paramètres, ainsi qu'un éventuel facteur de répétition[24].

Dès lors, en utilisant **l'opérateur()**, on peut obtenir les valeurs de cette fonction pour $0 \leq t \leq T$, ou au-delà si elle est répétée.

Cette classe permet aussi d'échantillonner un mouvement avec une période fixe **TS**, fixée ici à 20ms : il suffit pour cela d'appeler au départ la méthode **start**, puis ensuite **update** dans la boucle loop ; cette dernière est NON BLOQUANTE et renvoie **true** quand un nouvel échantillon est disponible. Dans les 2 cas, la méthode **val** fournit la valeur du mouvement.

On peut aussi interrompre le mouvement, en appelant **stop**, puis le reprendre en appelant à nouveau **start**. A noter que la méthode **update** renvoie **true** pendant l'interruption et que **val** ne change pas.

*Sketch **TestMovement.ino** :*

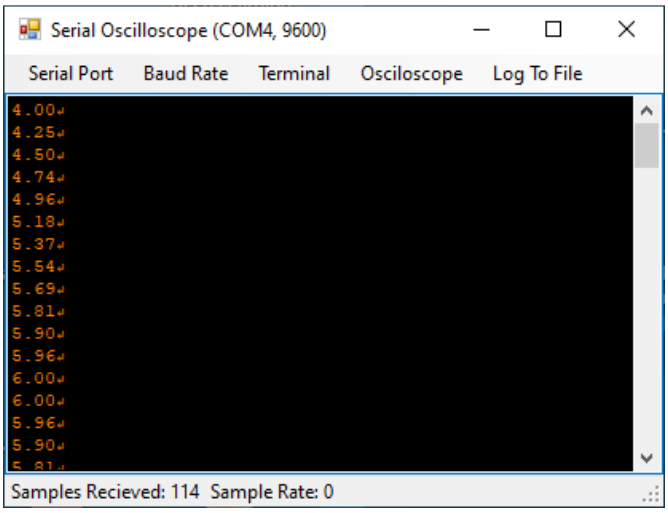
```
// TestMovement.ino

#include <Movement.h>

Movement m;
unsigned long oldTime, newTime;
byte state;

void print(double x)
{
```

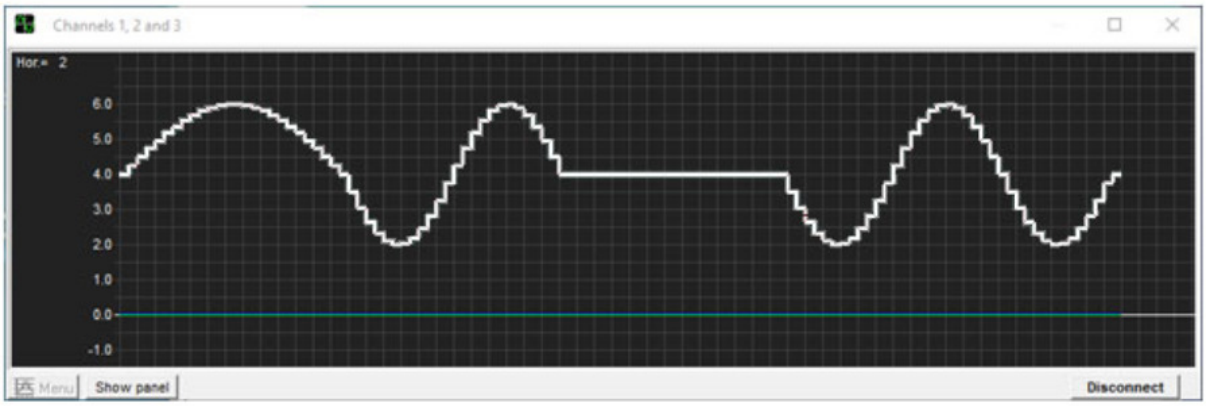
[24] NB : 0 répétition => le mouvement est réalisé 1 fois.

```
  Serial.print(x); Serial.print('\r');
}

void setup()
{
  Serial.begin(9600);
  delay(100);
  m.selOsc(2, 4, 0, 1000, 2);  // Osc[2,4,0,1000] 3 times
  m.start();
  print(m.val());
  oldTime = millis();
  state = 0;
}

void loop()
{
  if (m.update()) print(m.val());
  newTime = millis();
  if (newTime-oldTime >= 500)
  {
    oldTime = newTime;
    if (state==0) {m.setT(500); state++;}       // Period/2
    else if (state==1) {m.stop(); state++;}     // Stop
    else if (state==2) {m.start(); state++;}    // Restart
  }
}
```

On considère ici une oscillation définie par **Osc[2, 4, 0, 1000]** répétée 3 fois, et échantillonnée à **20ms**. On commence par diviser par 2 la période de 1000ms au bout de 500ms, puis on arrête le mouvement au bout de 500ms, et on le reprend enfin au bout de 500ms à nouveau.

*Affichage des résultats avec **Serial Oscilloscope***[25] :





(NB : unité horizontale = 40ms)

[25] https://x-io.co.uk/serial-oscilloscope/ (NB : la couleur des graphes a été changée pour les rendre plus lisibles).

On peut donc ainsi, très facilement, sélectionner, échantillonner, interrompre et reprendre un mouvement, sans avoir à gérer le timing.

4.2 Extension de la classe Servo

On va étendre, par héritage, les possibilités de la classe **Servo** en lui offrant de pouvoir contrôler le mouvement d'un servo dans le temps, avec l'une des fonctions paramétriques précédemment définies, et ceci de la manière la moins bloquante possible.

Procéder ainsi présente néanmoins un petit inconvénient : celui de ne pas connaître exactement la position (= angle) du servo, si on interrompt le mouvement, et ceci peut-être utile dans certaines fonctions de mouvement. On va donc stocker en permanence dans le champ **_pos** cette position et implémenter une méthode **move** analogue à **Servo::write**, pour la mettre à jour. Ceci qui nous permettra également d'intégrer la correction de trim :

```
void move(double pos);
```

Initialement, nous voulions introduire un délai d'attente systématique à la fin de cette méthode, pour garantir que la position visée soit bien atteinte. Mais nous y avons finalement renoncé car cela perturbait l'échantillonnage. Il conviendra donc d'apprécier le risque que cette situation se produise, et si nécessaire faire appel à **delay** juste après, quand on l'appelle directement. Pour cette raison, il vaudra mieux utiliser à la place :

```
void move(double pos, double T);
```

qui contrôlera le mouvement avec une durée égale à **T** ms, en utilisant **Aff[_pos, pos, T]** comme fonction de mouvement. Il faudra choisir **T** assez grand pour qu'à chaque échantillonnage (toute les TS = 20ms), on ait bien le temps de terminer la rotation du bras, sachant qu'en 20ms on peut tourner au maximum de 12°.

Evidemment, comme cette classe pourra utiliser les fonctions paramétriques **Aff** et **Osc** de la classe **Movement** pour contrôler les mouvements du servo dans le temps. Ceci pourra être réalisé de manière non bloquante avec les méthodes suivantes qui surchargent les méthodes du même nom de la classe **Movement** et s'utilisent de manière identique :

```
void start();
bool update();
void stop();
```

*Classe **ServoE** (fichiers **ServoE.h** et **ServoE.cpp**) :*

```
class ServoE : public Servo, public Movement
{
  public:
    // Constructors
    ServoE();
    ServoE(unsigned pin, double trim = 0.);

    // Access to private fields (only necessary)
    void setPin(unsigned pin) {_pin = pin;};
    void setTrim(double trim) {_trim = trim;};
    void setPos(double pos) {_pos = pos;};
    double pos() const {return _pos;};

    // Methods
    void attach();
    void detach();
    // Blocking Movement
    void move(double pos);           // Uncontrolled
    void move(double pos, double T); // Controlled
    // Non-blocking controlled Movement
    void start();
    bool update();
    void stop();
    bool isInactive();

  private:
```

```
    unsigned _pin;          // Arduino pin where the servo is connected
    double _trim;           // Trim value on writing (°)
    double _pos;            // Current position (°)
};
```

*Sketch **TestServoE.ino** :*

```
// TestServoE.ino

#include <ServoE.h>

ServoE s(3,-5);  // Servo on pin 3 with trim -5
unsigned long oldTime, newTime;
unsigned sta;

void print(double x)
{
  Serial.print(x); Serial.print('\r');
}

void setup()
{
  Serial.begin(9600);
  delay(100);
  s.attach();

  // Oscillation (x4) with 3000ms period
  s.selOsc(90, 90, 0, 3000, 3);
  s.start();
  print(s.pos());
  oldTime = millis();
  sta = 0;
}

void loop()
{
  if (s.update()) print(s.pos());
  newTime = millis();
  if (newTime-oldTime >= 2000)
  {
    oldTime = newTime;
    if (sta == 0) {s.setT(1000); sta++;}    // Period/3 at 2s
    else if (sta == 1) {s.stop(); sta++;}   // Stop at 4s
    else if (sta == 2) {s.start(); sta++;}  // Restart at 6s
  }
  if (s.isInactive()) s.detach();
}
```

On considère un servo sur la broche 3, trimé à -5°. Son mouvement est une oscillation définie par **Osc[90, 90, 0, 3000]**, répétée 3 fois et échantillonnée à **20ms**. On commence par diviser par 3 sa période à 2s, puis on arrête le mouvement à 4s, puis on le reprend à 6s.

*Affichage des résultats avec **Serial Oscilloscope** :*

(NB : unité horizontale = 100ms)

4.3 Mouvement de plusieurs servos en parallèle

La classe **TServoE<N>** répond à ce problème en définissant des méthodes classiques sur un tableau de **N** servos, tous de type **ServoE**.

*Classe **TServoE<N>** (fichier **TServoE.hpp**)*

```
/*******************************************************************************
Class TServoE<N>
*******************************************************************************

Table of N extended servos (template)

Public methods :
  TServoE() : constructor
  TServoE(unsigned tPin[], double tTrim[] = NULL) : constructor
  ServoE &operator[](unsigned i) : access to servos
  void attach() : attach
  void detach() : detach
  void move(double tPos[], double T) : blocking controlled parallel movements
  void start() :  start or restart non-blocking controlled parallel movements
  bool update() : update non-blocking controlled parallel movements
  void stop() : stop non-blocking controlled parallel movements
  bool isInactive() : test if all the servos are inactive
*/
```

Comme c'est une classe **template**, elle est déclarée et implémentée dans un unique fichier, **TServoE.hpp**, dont nous ne fournissons ici que les commentaires.

Un opérateur **[]** permet d'obtenir une référence sur chaque servo du tableau. En l'utilisant, on peut par exemple sélectionner pour lui, une fonction de mouvement.

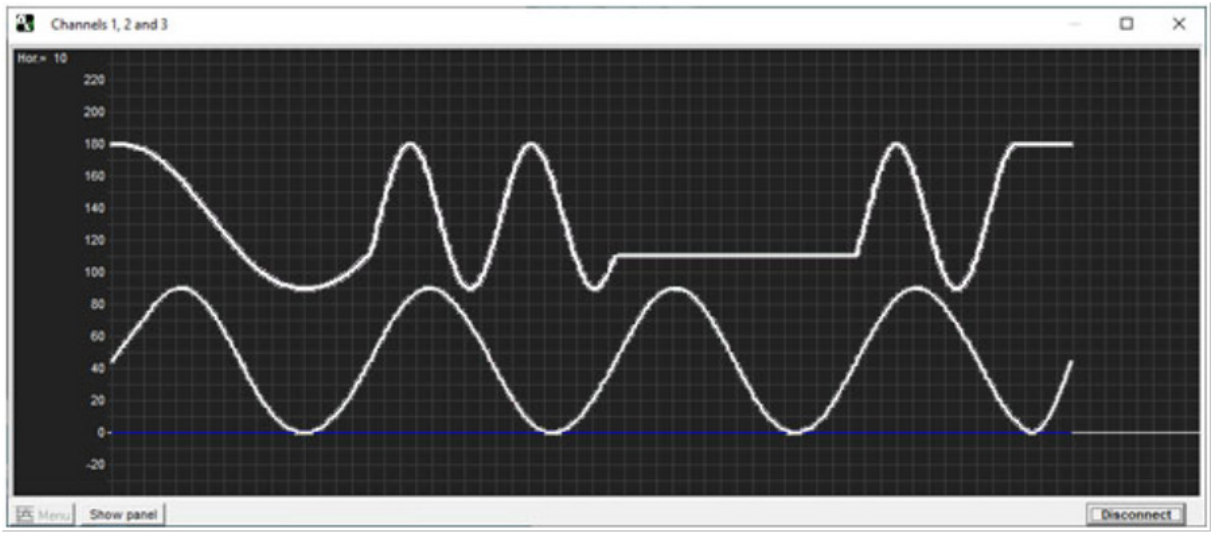
Toutes les méthodes de mouvement font tourner les **N** servos en parallèle en les contrôlant et là encore, on dispose de méthodes **start**, **update** et **stop**, pour le faire de manière non bloquante.

*Sketch **TestTServoE.ino** :*

```
// TestTServoE.ino

#include <TServoE.hpp>

unsigned tPin[] = {2,3};
double tTrim[] = {-12,-5};
TServoE<2> tS(tPin, tTrim);
unsigned long oldTime, newTime;
unsigned sta;

void print(double x, double y)
{
  Serial.print(x); Serial.print(','); Serial.print(y); Serial.print('\r');
}

void setup()
{
  Serial.begin(9600);
  delay(100);
  tS.attach();

  // Blocking controlled parallel Movements to start positions
  double tPos[] = {45, 180};
  tS.move(tPos, 500);

  // Non-blocking controlled parallel Movements
  tS[0].selOsc(45, 45, 0, 2000, 3);
  tS[1].selOsc(45, 135, PI/2, 3000, 3);
  tS.start();
  print(tS[0].pos(), tS[1].pos());
  oldTime = millis();
  sta = 0;
}
```

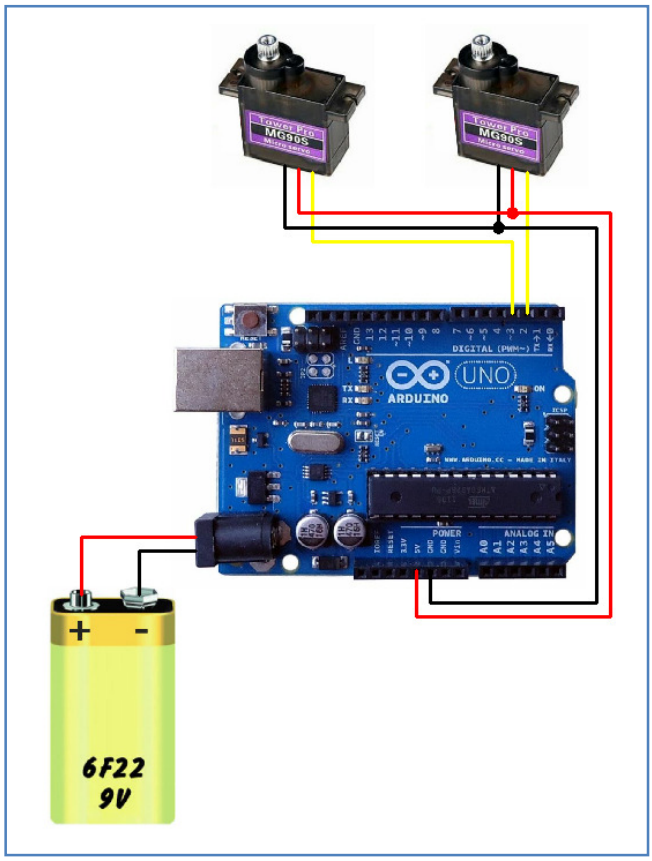
```
void loop()
{
  if (tS.update()) print(tS[0].pos(), tS[1].pos());
  newTime = millis();
  if (newTime-oldTime >= 2000)
  {
    oldTime = newTime;
    if (sta == 0) {tS[1].setT(1000); sta++;}   // Period/3 at 2s
    else if (sta == 1) {tS[1].stop(); sta++;}  // Stop at 4s
    else if (sta == 2) {tS[1].start(); sta++;} // Restart at 6s
  }
  if (tS.isInactive()) tS.detach();
}
```

*Affichage des résultats avec **Serial Oscilloscope** :*



(NB : unité horizontale = 200ms)

*Schéma de connexion avec un **Arduino Uno** :*



5 Mise en œuvre avec Otto

Avec ces 3 classes **Movement**, **ServoE** et **TServoE<N>**, il est alors très facile de définir une classe **Robot** pour modéliser les mouvements d'Otto : il suffit de la faire hériter de **TServoE<4>** !

*Classe **Robot** (fichiers **Robot.h** et **Robot.cpp**) :*

```
enum Part {LL, RL, LF, RF};
enum Direction {FORWARD, BACKWARD};
enum Side {LEFT, RIGHT};
enum Speed {SLOW, NORMAL, RAPID};

/*********************************************************************************
Class Robot
*********************************************************************************

Robot class for OttoDIY with only legs and feet Movement functionnalities
*/

class Robot : public TServoE<4>
{
  public:
    Robot(unsigned pinLL, unsigned pinRL, unsigned pinLF, unsigned pinRF, bool calibrated = true);
// Constructor
    void setSpeed(Speed spd);                       // To change the Movement speed
    void move(double tPos[]);                       // Blocking controlled parallel Movements to
tPos
    void home();                                    // Blocking controlled parallel Movements to
{90,90,90,90}
    void walk(Direction dir, unsigned int nRep = 0); // To start a walk non-blocking Movement
    void turn(Side sid, unsigned int nRep = 0);      // To start a turn non-blocking Movement
    void walk_turn(Direction dir, Side sid, unsigned int nRep = 0); // To start a walk_turn non-
blocking Movement

  private:
    Speed _spd;
    void setMov(double tA[], double tB[], double tPhi[], unsigned int nRep);
};
```

Otto ayant 4 servos, on a défini un constructeur auquel il faut indiquer les numéros des broches utilisées : 2, 3, 4, 5 par défaut. Comme dans OttoDIYLib, son dernier argument, **calibrated**, indique si on peut aller lire dans l'EEPROM les valeurs des trims.

3 méthodes de mouvements sont fournies à titre d'exemple :

- Les méthodes **walk** et **walk_turn** sont quasiment identiques aux méthodes **walk** et **turn** de la classe **Otto9**, avec en plus la possibilité de tourner en reculant pour la seconde.
- La méthode **turn** correspond ici à une rotation sur place du robot, à gauche ou à droite.

Leurs implémentation sont analogues à celles de la classe **Otto9** ; par exemple :

```
void Robot::turn(Side sid, unsigned int nRep = 0)
{
  double phi;
  switch (sid)
  {
    case LEFT: phi = -PI; break;
    case RIGHT: phi = 0; break;
  }
  float tA[4] = {15, 15, 20, 20};
  float tB[4] = {105, 75, 94, 86};
  float tPhi[4] = {-PI/2, -3*PI/2, phi, phi};
  setMov(tA, tB, tPhi, nRep);
}
```

Avec :

```
void Robot::setMov(double tA[], double tB[], double tPhi[], unsigned int nRep)
{
  double tPos[4];
  for (unsigned i=0; i<4; i++) tPos[i] = tA[i]*sin(tPhi[i])+tB[i];
  move(tPos);
  double x = 30.;
  double T = 4000*x/tSpeed[_spd];
  for (unsigned i=0; i<4; i++) (*this)[i].selOsc(tA[i], tB[i], tPhi[i], T, nRep);
  start();
}
```

On notera que de tels mouvements se décomposent donc en :

- Un mouvement affine bloquant pour faire tourner les servos jusqu'aux positions de départ du mouvement oscillatoire, ceci pour augmenter la fluidité du mouvement ;
- Un mouvement oscillatoire non bloquant, avec les paramètres adéquats.

Ces 2 mouvements sont contrôlés par un paramètre de vitesse **_spd**, dont on déduit les périodes **T**, communes pour les 4 servos du robot, qui sont donc ainsi bien synchronisés. La méthode **setSpeed** permet de la changer « à chaud ».

*Sketch **TestRobot.ino** :*

```
// TestRobot.ino

#include <Robot.h>

Robot r(2, 3, 4, 5);

void setup()
{
  r.attach();
  r.home();
  r.walk(FORWARD, 5);
}

void loop()
{
  r.update();
  if (r.isInactive())
  {
      r.home();
      r.detach();
  }
}
```

6 Commande du robot via une liaison Bluetooth

6.1 Classes SerialCommand et RobotSC

La chose à bien comprendre ici est que problème consiste essentiellement à commander Otto via une liaison série TTL[26], comme par exemple celle qu'on établit avec une connexion USB, en utilisant l'objet **Serial**, instance de la classe **HardwareSerial**.

En effet, le module Bluetooth d'Otto est connecté sur les broches 12 (RX) et 11 (TX) de l'Arduino Nano, avec établissement d'une liaison série TTL sur ces dernières, en utilisant une instance de la classe **SoftwareSerial**[27].

[26] Ici niveau logique = 5V pour l »Arduino Nano, la ligne étant au repos à l'état BAS (0V).
[27] Un Arduino Uno ou Nano ne dispose que d'un seul port série matériel, connecté au broches 0 (RXD) et 1 (TXD) et utilisé lors d'une connexion USB. C'est pour cette raison que ces broches ne sont pas reliées à un composant du Robot.

Or, si l'on regarde les définitions de ces 2 classes[28] :

```
class HardwareSerial : public Stream
class SoftwareSerial : public Stream
```

on constate qu'elles héritent toutes les deux de la classe **Stream** et implémentent notamment ses méthodes virtuelles :

```
virtual int available() = 0;
virtual int read() = 0;
```

Grace au **polymorphisme** du C++, on pourra traiter les deux cas de manière générique, en considérant une instance (en fait une référence pour éviter une copie inutile) de la classe **Stream**, qui pourra donc accepter aussi bien un objet instance de la classe **HardwareSerial** comme **Serial**, qu'un objet instance de la classe **SoftwareSerial** (cf. plus loin), et bien sûr appeler les méthodes précédentes.

C'est ce que l'on va faire dans la classe **SerialCommand** définie comme suit.

*Classe **SerialCommand** (fichiers **SerialCommand.h** et **SerialCommand.cpp**) :*

```
class SerialCommand
{
  public:
    SerialCommand(Stream& serial, int nCmd, Command tCmd[]);
    char typ() const {return _typ;}
    unsigned opt() const  {return _opt;}
    void resetParse();
    bool parseCommand();

  private:
    Stream& _serial;              // Serial port used for commands
    unsigned _nCmd;               // Number of acceptable commands
    Command *_tCmd;               // Array of acceptable commands (as reference)
    bool _okCommand;              // True if a command as been correctly parsed
    char _typ;                    // Type of this command
    unsigned _opt;                // Option of this command
    unsigned _nParse;             // Number of parsed characters
};
```

Le but de cette classe est de d'analyser les commandes reçues sur le Stream **serial** passé à son constructeur. Celles-ci devront respecter la syntaxe et la sémantique suivantes :

```
Syntax :
  <command> = <typ><opt>#
  <typ> = <letter>
  <letter> = A|B|...|Z
  <opt> = <digit>|<digit><digit>
  <digit> = 0|...|9

Semantic :
  (<typ>,<opt>) in tCmd
```

Le tableau **tCmd** des commandes acceptables, paramètre du constructeur, est ici défini par :

```
Command tCmd[] =    // Acceptable commands
{
  {'C',2,{1,0}},    // Contact : 1=>on, 0=>off
  {'M',9,{0,1,2,3,4,13,14,23,24}},
```

Cependant, la librairie **SoftwareSerial** (Arduino - SoftwareSerial) permet d'étendre, par logiciel, cette fonctionnalité en créant d'autres ports série sur d'autres broches, mais avec des restrictions.

[28] Dans le dossier d'installation de l'IDE Arduino :

- <Arduino>/hardware/arduino/avr/cores/arduino/HardwareSerial.h
- <Arduino>/hardware/arduino/avr/libraries/SoftwareSerial/src/HardwareSerial.h

```
  // Movement : 0=>home, 1/2=>walk FORWARD/BACKWARD, 3/4=>turn LEFT/RIGHT, 13=>walk FORWARD and
turn LEFT, etc.
  {'I',2,{1,0}},    // Interruption : 1=>on, 0=>off
  {'S',3,{1,2,3}}   // Speed : 1=>SLOW, 2=>NORMAL, 3=>RAPID
};
```

chacune étant de type **Command**[29] :

```
struct Command {char typ; unsigned nOpt; unsigned tOpt[MAX_OPTIONS];};  // Acceptable command
```

Par exemple :

- M0# => home
- M1# => avancer
- M3# => tourner à gauche (sur place)
- M13# => avancer et tourner à gauche
- Etc.

Comme on ne maitrise pas toujours la production de '\n' (saut de ligne) ou '\r' (retour charriot), ces caractères seront ignorés et on a préféré utiliser '**#**' pour indiquer la fin d'une commande.

La principale méthode de cette classe se charge de décoder les commandes reçues :

```
bool parseCommand();
```

Son implémentation est celle d'un interpréteur classique, qui renvoie **true** dès réception de l'une d'elles ; il suffit alors d'appeler les méthodes **typ** et **opt** pour en connaître le type et l'option.

Et c'est ce que l'on fait dans l'unique méthode **processCommand** de la classe **RobotSC**, qui hérite de la classe **Robot** et de la classe précédente.

*Classe **RobotSC** (fichiers **RobotSC.h** et **RobotSC.cpp**) :*

```
class RobotSC : public Robot, SerialCommand
{
  public:
    RobotSC(int pinLL, int pinRL, int pinLF, int pinRF, Stream& serial, bool calibrated = true);
    void processCommand();
};

void RobotSC::processCommand()
{
  if (parseCommand())
  {
    resetParse();
    unsigned _opt = opt();
    switch (typ())
    {
      case 'C':
        if (_opt==1) attach();
        else if (_opt==0) detach();
        break;
      case 'M':
        if (_opt==0) home();
        else if (_opt==1) walk(FORWARD, REP);
        else if (_opt==2) walk(BACKWARD, REP);
        else if (_opt==3) turn(LEFT, REP);
        else if (_opt==4) turn(RIGHT, REP);
        else if (_opt==13) walk_turn(FORWARD, LEFT, REP);
        else if (_opt==14) walk_turn(FORWARD, RIGHT, REP);
        else if (_opt==23) walk_turn(BACKWARD, LEFT, REP);
        else if (_opt==24) walk_turn(BACKWARD, RIGHT, REP);
        break;
      case 'I':
```

[29] Ici MAX_OPTIONS = 9.

```
          if (_opt==1) stop();
          else if (_opt==0) start();
          break;
        case 'S':
          if (_opt==1) setSpeed(LOW);
          else if (_opt==2) setSpeed(NORMAL);
          else if (_opt==3) setSpeed(RAPID);
          break;
      }
    }
  }
```

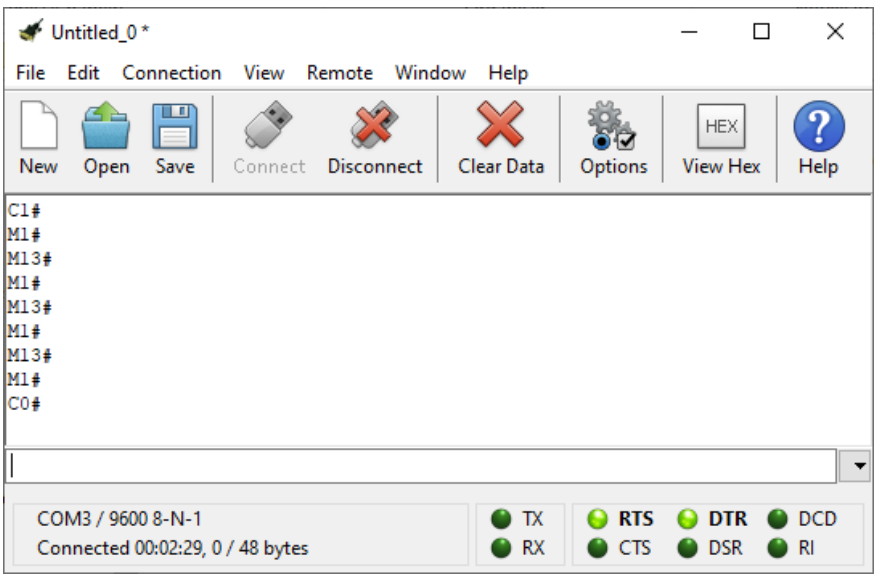
A noter ici l'appel de **resetParse** pour réinitialiser l'interpréteur, ainsi que la valeur constante du paramètre **REP**, fixée à 100 pour simplifier l'envoi des commandes.

6.2 Commande du Robot avec un moniteur série, via une liaison USB

*Sketch **TestRoboSC.ino** :*

```
// TestRobotSC.ino

#include <RobotSC.h>

RobotSC r(2, 3, 4, 5, Serial);

void setup()
{
  Serial.begin(9600);
  delay(100);
}

void loop()
{
  r.processCommand();
  r.update();
}
```

Voici un exemple de commandes transmises avec le moniteur série **Coolterm**[30], pour faire tourner le robot autour d'un cube :



[30] http://freeware.the-meiers.org/ avec les options **Line Mode** et **Local Echo**.

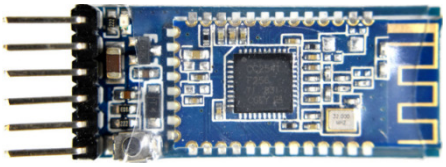
6.3 Commande du Robot avec un Smartphone, via une liaison Bluetooth

Pour le sketch Arduino, comme annoncé, il suffit juste de remplacer **Serial** par une instance, **BTSerial** ici, de la classe **SoftwareSerial**.

*Sketch **TestRobotBT.ino** :*

```
// TestRobotBT.ino

#include <RobotSC.h>
#include <SoftwareSerial.h>

SoftwareSerial BTSerial(11,12);
RobotSC r(2, 3, 4, 5, BTSerial);

void setup()
{
  BTSerial.begin (9600);
  delay(100);
}

void loop()
{
  r.processCommand();
  r.update();
}
```

Il faut ensuite établir une connexion Bluetooth entre le module ici fourni et le Smartphone.



Ce module Bluetooth « Low Energy 4.0 » est compatible avec un **MLT-BT05** et architecturé autour d'un composant **CC2541** de Texas Instrument (ou clone…). Il est paramétrable avec des **commandes AT**[31] [32].

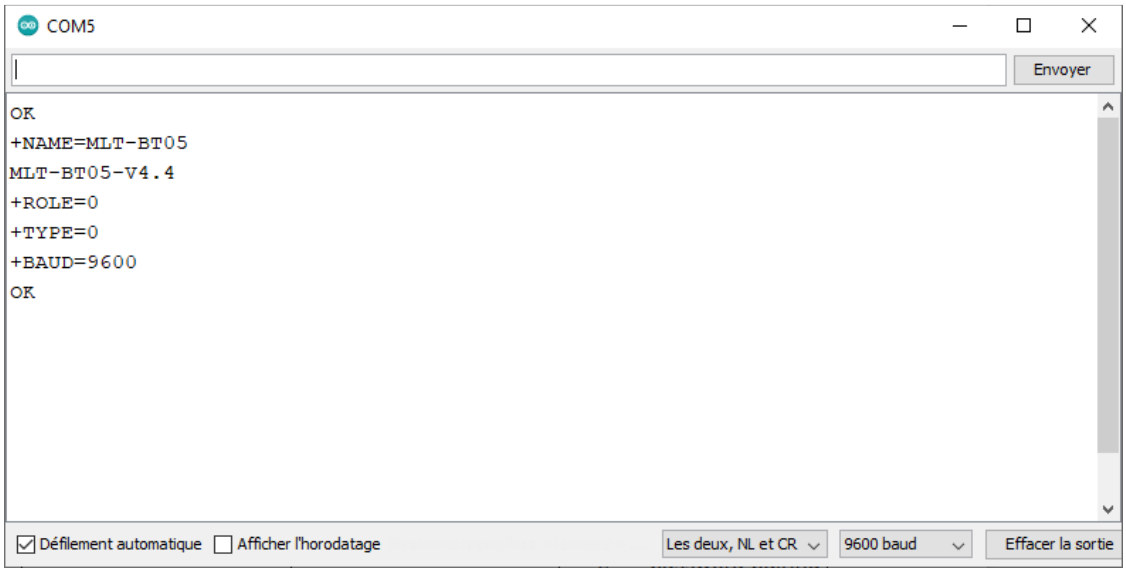
* AT	Check if the command terminal is working
* AT+DEFAULT	Restore factory default
* AT+BAUD	Get/Set baud rate
* AT+RESET	Software reboot
* AT+ROLE	Get/Set current role.
* AT+DISC	Disconnect connection
* AT+ADVEN	Broadcast switch
* AT+ADVI	Broadcast interval
* AT+NINTERVAL	Connection interval
* AT+POWE	Get/Set RF transmit power
* AT+NAME	Get/Set local device name
* AT+LADDR	Get local bluetooth address
* AT+VERSION	Get firmware, bluetooth, HCI and LMP version
* AT+TYPE	Binding and pairing settings
* AT+PIN	Get/Set pin code for pairing
* AT+UUID	Get/Set system SERVER_UUID .
* AT+CHAR	Get/Set system CHAR_UUID .
* AT+INQ	Search from device
* AT+RSLV	Read the scan list MAC address
* AT+CONN	Connected scan list device
* AT+CONA	Connection specified MAC
* AT+BAND	Binding from device
* AT+CLRBAND	Cancel binding
* AT+GETDCN	Number of scanned list devices
* AT+SLEEP	Sleep mode
* AT+HELP	List all the commands

[31] Par exemple : Bluetooth Module with Arduino (AT-09, MLT-BT05, HM-10) — Maker Portal (makersportal.com)
[32] MLT-BT05-AT-commands-TRANSLATED.pdf (wlu.ca)

Pour le paramétrer, le plus simple est d'utiliser le moniteur série de l'IDE Arduino, connecté à 9600 bauds, et le sketch **Otto_BlueTest.ino**, fourni dans **OttoDIYLib**[33].

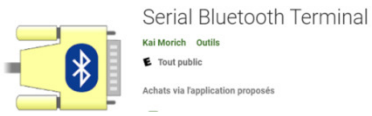
Voici les réponses obtenues en envoyant les commandes AT suivantes (avec les réglages d'usine, seule la vitesse du port série doit être changée) :

- **AT+DEFAULT** // Restauration des paramètres d'usine
- **AT+NAME** // Affichage du nom
- **AT+VERSION** // Affichage version
- **AT+ROLE0** // Fixer le rôle (0 => Esclave)
- **AT+TYPE0** // Fixer le type (0 => pas de mot de passe)
- **AT+BAUD4** // Fixer la vitesse du port série : 4 => 9600 bauds

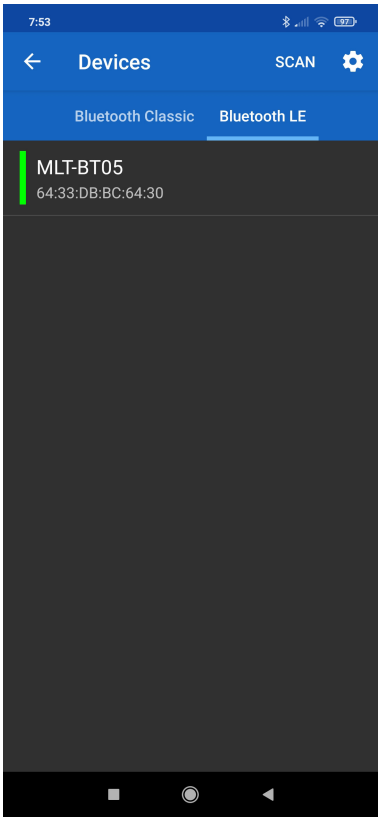


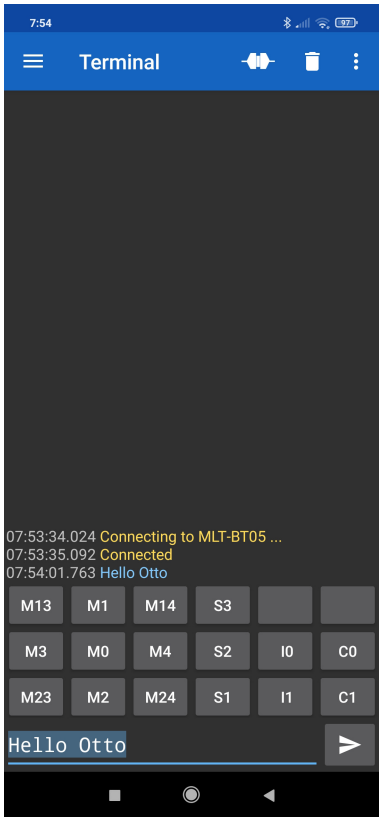
Le module peut alors être reconnu sur un Smartphone[34] acceptant le protocole Bluetooth Low Energy 4.0.

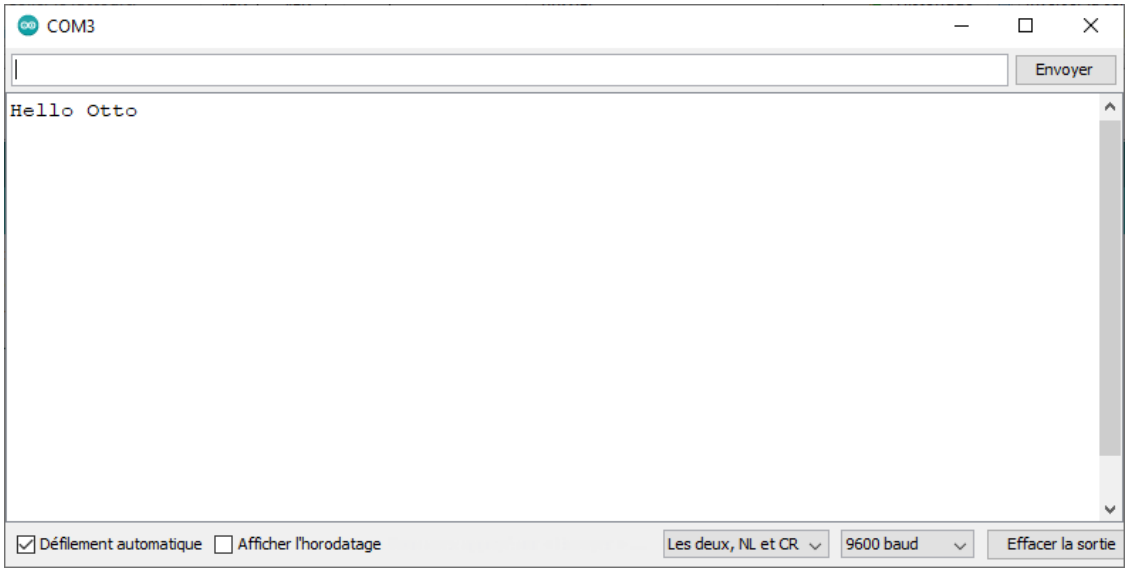
Pour s'en convaincre, après avoir activé ce protocole, il suffit d'y installer l'application Android **Serial Bluetooth Terminal**, disponible sur Google Play[35] :



[33] GitHub - OttoDIY/OttoDIYLib: Latest Libraries for Otto DIY robots
[34] Xiaomi MI9 SE avec Android 10 (MIUI 12.0.4) ici.
[35] Serial Bluetooth Terminal – Applications sur Google Play

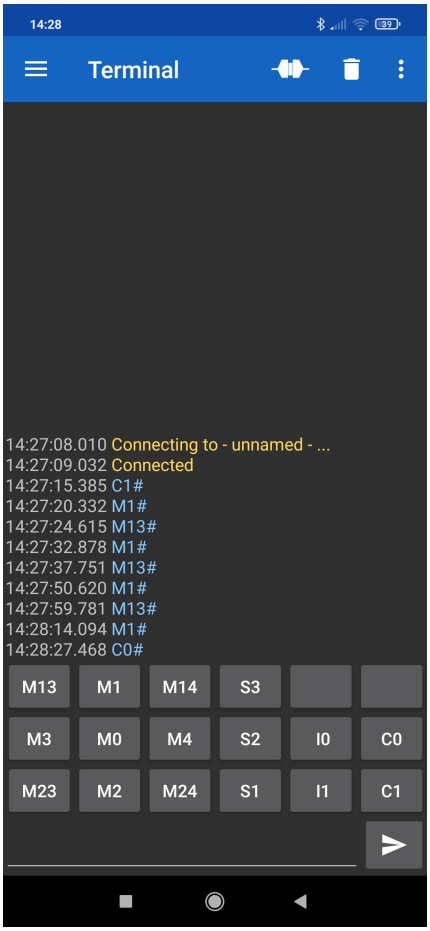


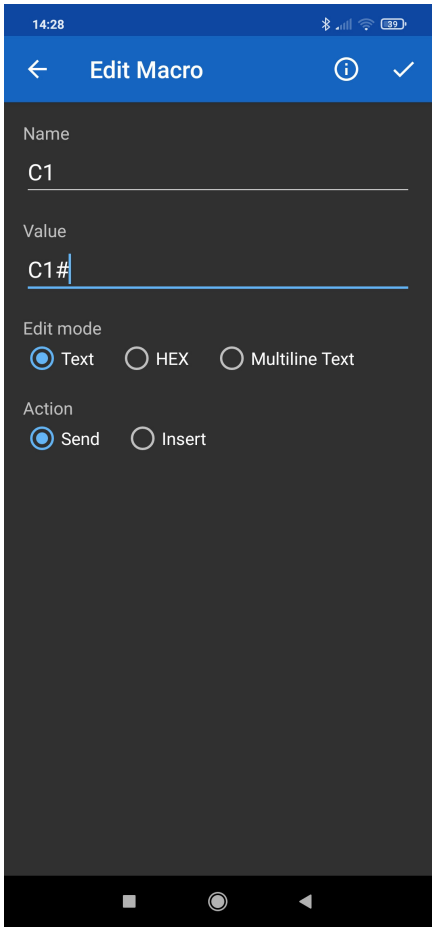




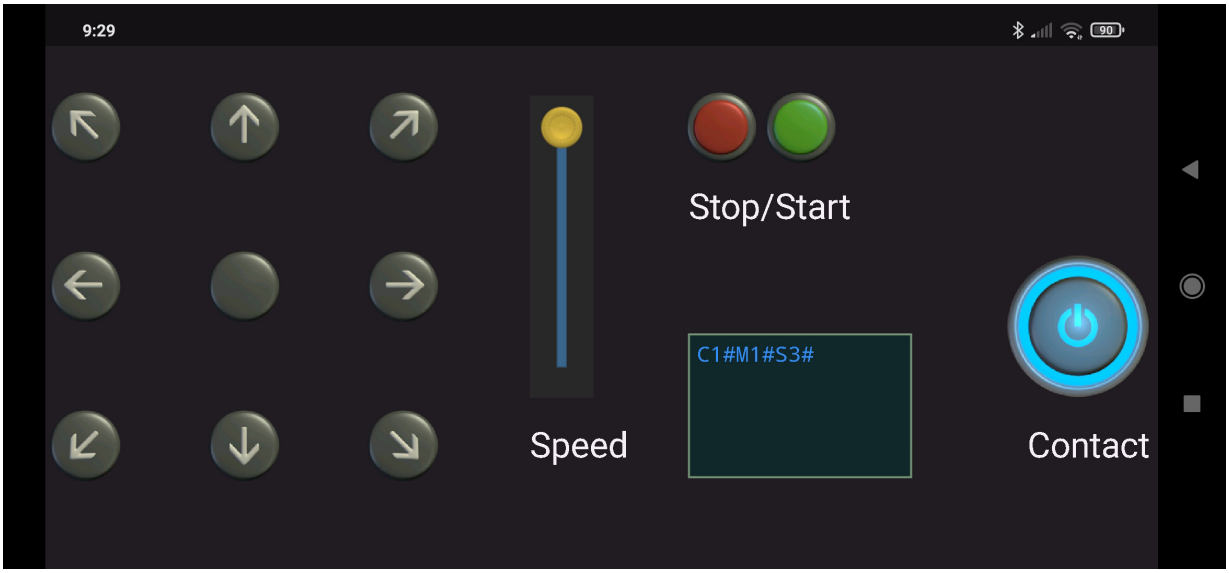
Si on installe maintenant le sketch **TestRobotBT.ino** dans le robot, on peut le commander comme on l'a fait précédemment avec un moniteur série.

Pour faciliter l'envoi des commandes, on peut affecter des macros et des labels aux boutons de l'application.



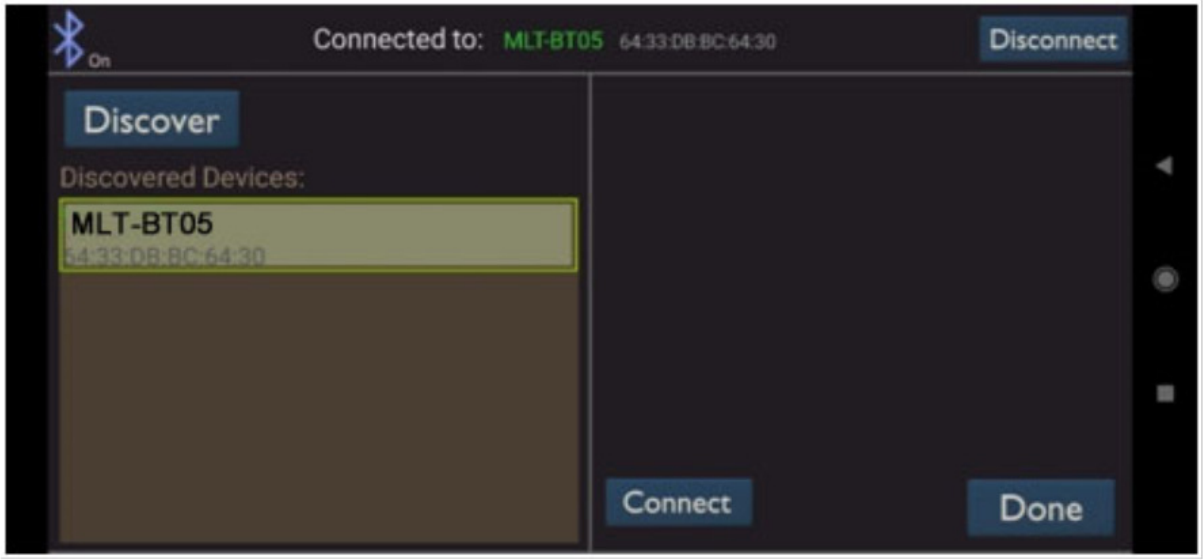


Si on veut une interface un peu plus « fun », on peut utiliser l'application **Bluetooth Electronics**[36]**,** en construisant par drag and drop une interface de commande :

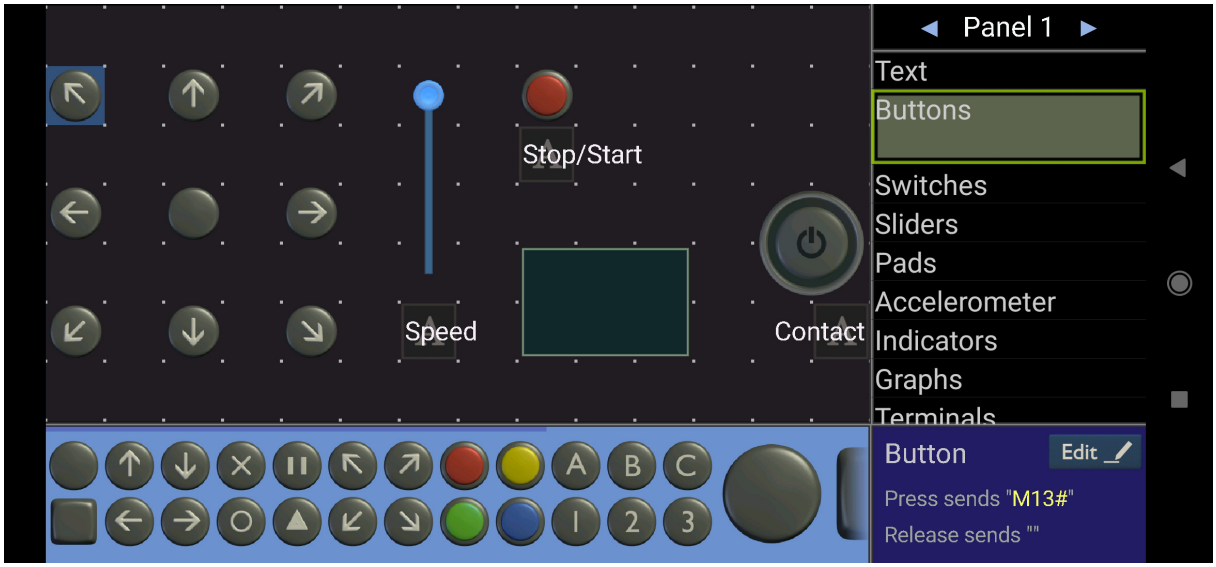


Là encore, le module Bluetooth est reconnu quand on active le Bluetooth dans le Smartphone[37] :

[36] Bluetooth Electronics (keuwl.com)
[37] Avec la Localisation… Bug ?



Et il suffit alors de composer un écran en mode « Edition » où, là encore, on peut affecter des macros aux éléments de l'interface :



Puis de passer en mode « Run » pour commander le robot.

A noter néanmoins quelques bugs, en particulier si on utilise '\n' comme fin de ligne : c'est pour cette raison que nous avons préféré '#'.

Remarques

- Les exécutables produits avec les 2 sketches précédents sont largement compatibles avec les possibilités mémoires d'un Arduino Nano : ils n'occupent que 30% environ de la mémoire flash et de la mémoire SRAM disponible.
- Il y a donc de la marge pour prendre en compte d'autres capteurs : microphone, capteur capacitif (sensible au toucher), capteur de distance (sonar), etc. Ils peuvent s'intégrer comme on vient de le faire avec un port série, puisque ce dernier est utilisé ici comme « capteur » de caractères.
- On peut aussi étendre cette fonctionnalité au cas d'un fichier de texte ou d'un flux réseau de commandes, puisque ceci reviendrait encore à mettre en œuvre des objets instances de classes héritant de la classe **Stream**. Mais ici, l'approche utilisant **Blockly** proposée avec Otto

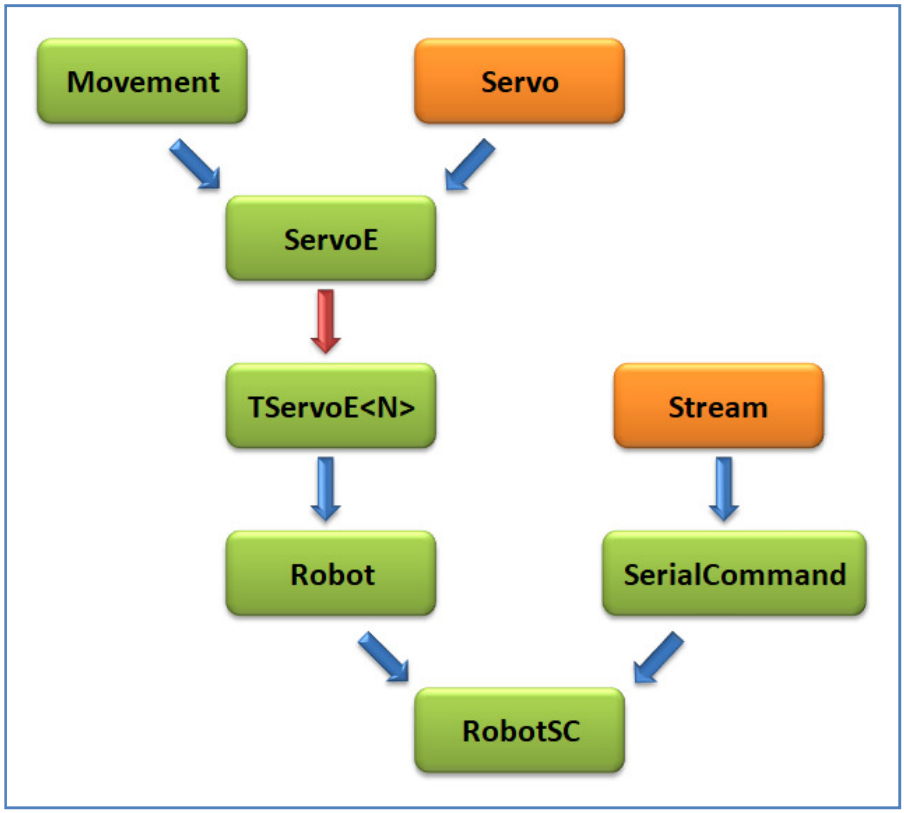
est bien plus séduisante et il ne serait pas bien difficile de l'adapter pour produire du code compatible avec la classe **Robot**.

- La simplicité de cette mise en œuvre n'est possible qu'avec des processus non bloquants, tant au niveau des servos qu'au niveau des capteurs. Dans le cas contraire, il faudrait recourir à des traitements par interruption, ce que nous avons voulu éviter ici pour ne pas perturber le bon fonctionnement des timers et en particulier **timer1**, utilisé dans la bibliothèque Servos.

7 Conclusion

Cette étude nous a permis de comprendre comment Otto se déplace et en particulier toute l'importance des mouvements oscillatoires étudiés par **Juan González Gómez**.

Pour répondre aux objectifs énoncés à la fin du paragraphe 3, nous avons défini une bibliothèque Arduino de classes C++ (en vert ci-dessous), avec les dépendances suivantes, notre préoccupation première étant toujours d'implémenter dès que possible les fonctionnalités visées, de manière à simplifier d'autant les classes dépendantes et ce, de la manière la plus générale possible.



Sur ce schéma simplifié[38], on constate l'importance de la classe **Movement** dont toutes les autres classes dépendent. Elle définit de manière générique la notion de mouvement, sans préciser la fonction spécifique choisie et elle « encapsule » tout le processus d'échantillonnage d'un mouvement, notion clé pour réaliser ensuite des mouvements contrôlés, non bloquants et en parallèle sur plusieurs servos.

[38] Non UML ! Les flèches bleues => héritage et la flèche rouge => composition.

Nous avons mis à profit toute la puissance du C++[39] et de la programmation orientée objet (héritage, composition, polymorphisme, généricité, pointeur sur une méthode de classe, opérateurs, template) pour implémenter simplement les fonctionnalités visées. Ceci nous a permis de produire des exécutables très légers, bien adaptés aux faibles capacités mémoire de l'Arduino Nano contenu dans Otto.

Des outils tels que **WinPython** et **Serial oscilloscope** ont été utilisés pour produire des graphiques : ils méritent d'être connus !

Nos objectifs ne concernant que les mouvements de jambes et de pieds d'Otto, nous n'avons pas voulu prendre en compte toutes les fonctionnalités d'Otto et en particulier celles liées à ses capteurs. Mais, avec les concepts mis en œuvre, ceci ne serait pas difficile.

Nous tenons enfin à remercier tous les concepteurs d'Otto pour nous avoir inspiré ce travail et en particulier **Camilo Parra Palacio** pour les informations qu'il nous a aimablement transmises.

[39] Mais limitées à celles prises en compte par un le compilateur de l'IDE Arduino pour un Arduino Nano.