

MicroPython

Que faire avec un ESP32-C3 mini ?

Anumby nov-2024

Python

Python 2 → Python 3

Langage interprété (Basic, Javascript, ...)

Interpréteur CPython → shell Python, prompt >>>

Programme Python = « Script » Python

Micropython

Interpréteur CPython « allégé » pour microcontrôleur (SOC)

Toutes les fonctionnalités de Python 3

Quelques librairies standards

Firmwares sur le site Micropython : <https://micropython.org/download/>

Python - Librairie/module

Import simple

```
>>> import math  
>>> math.sin(math.pi/2)  
1.0
```

Import avec alias

```
>>> import math as mt  
>>> mt.sin(mt.pi/2)  
1.0
```

Import d'éléments isolés

```
>>> from math import pi, sin, cos, exp  
>>> cos(pi/2)  
6.123233995736766e-17
```

Python - Librairie/module

Afficher le contenu d'une librairie

```
>>> import math
>>> dir(math)
['__doc__', '__file__', '__loader__', '__name__', '__package__',
 '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'ceil',
 'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'erfc',
 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'gamma',
 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf',
 'lcm', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf',
 'nextafter', 'perm', 'pi', 'pow', 'prod', 'radians', 'remainder',
 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc', 'ulp']
```

Librairie time

```
>>> import time
>>> time.sleep(5)    # temps en s
```

Python - Fonctions

Ecriture d'une fonction

```
def moyenne(arg1, arg2):  
    m = (arg1 + arg2)/2  
    return m
```

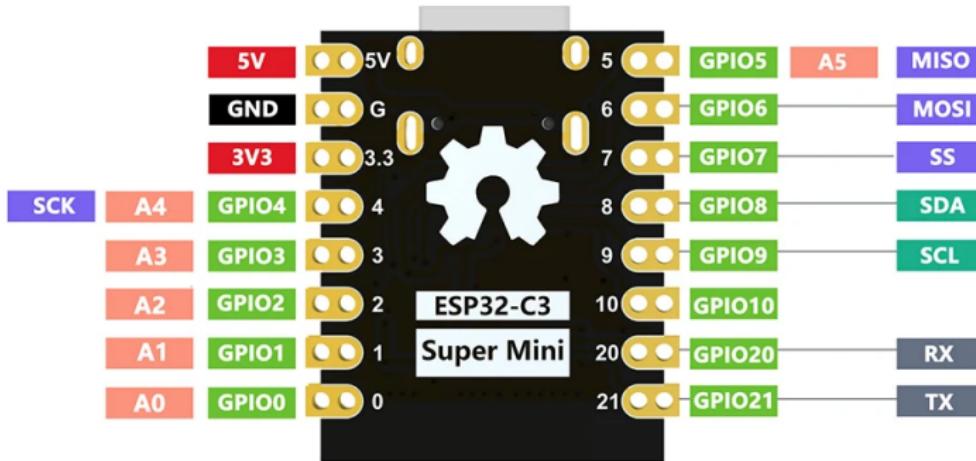
Appel de la fonction

```
>>> moyenne(1.5, 7)  
4.25
```

Important

⚠ Respecter l'indentation !

MicroPython - ESP32-C3 mini



■ Pin No. ■ Power ■ ADC ■ SPI ■ GND ■ UART ■ Digital

⚠ Circuit vu de dessous !

Micropython - ESP32-C3 mini

The screenshot shows the Thonny IDE interface with a MicroPython session running on an ESP32-C3 mini. The top status bar indicates "Thonny - <untitled> @ 1:1". The left sidebar shows the file structure under "This computer / Users / gilles / ESP32-C3 mini" and lists "boot.py" as the selected file. The main window has two tabs: "untitled" and "Shell". The "untitled" tab contains a single line of code: "1". The "Shell" tab displays the MicroPython REPL output:

```
MPY: soft reboot
MicroPython v1.23.0 on 2024-06-02; LOLIN_C3_MINI with ESP32-C3FH4
Type "help()" for more information.

>>> help('modules')

  __main__      btree      initsetup      ssl
  asyncio      builtins    io           struct
  boot        c3mini     json          sys
  espnow       collections machine      time
  onewire      cryptolib  mip/_init_
  _thread      deflate     neopixel    tls
  webrepl     dht         network     uasyncio
  aioespnow   ds18x20    ntptime     uctypes
  apal06      errno       onewire    umqtt/robust
  array        esp         os           umqtt/simple
  asyncio/_init_ esp32      platform   upysh
  asyncio/core espnow     random     urequests
  asyncio/event espnow     framebuffer vfs
  asyncio/funcs flashbdev re
  asyncio/lock  framebuf  requests/_init_
  asyncio/stream gc         select
  binascii     hashlib   socket
  bluetooth   heapq
  Plus any modules on the filesystem

>>> |
```

The "machine", "math", "os", "platform", and "random" modules are highlighted with red boxes, indicating they are currently being typed or selected. The bottom status bar shows "MicroPython (ESP32) - USB ITAC(serial)dubw.usbunit@localhost:14401" and "MicroPython Que faire avec un ESP32-C3 mini ? Anumby nov-2".

Librairie machine

Fonctions liées à la gestion du hardware, en particulier les GPIOs

```
>>> import machine  
>>> dir(machine)  
['__class__', '__name__', 'ADC', 'ADCBlock', ...]
```

GPIO

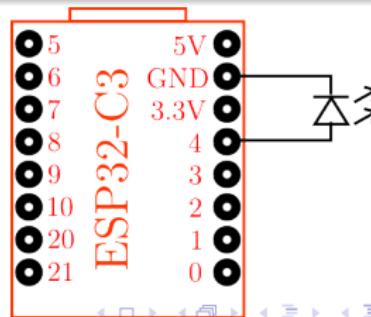
- module Pin : contrôle des pins en entrée/sortie
- module PWM : Pulse Width Modulation
- module Timer : contrôle des timers hardware
- module UART : bus de liaison série
- module I2C : bus de liaison I2C
- module ADC : conversion analogue → digital
- ...

Module Pin : output

```
>>> from machine import Pin  
>>> p4 = Pin(4, Pin.OUT)    # pin 4 utilisée en sortie  
>>> dir(p4)  
['__class__', 'value', ...  
..., 'board', 'init', 'irq', 'off', 'on']  
>>> p4.value(0)          # p4 à 0V, identique à p4.off()  
>>> p4.value(1)          # p4 à 3.3V, identique à p4.on()
```

Exercice

Ecrire un script qui fait clignoter la LED 10 fois avec une période de 1s et un rapport cyclique de 0.5.
Idem avec la LED 'onboard' (GPIO 8)



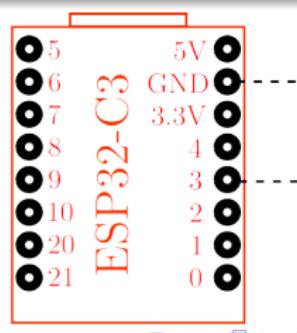
Module Pin : input

```
>>> from machine import Pin  
>>> p3 = Pin(3, Pin.IN)    # pin 4 utilisée en entrée  
>>> p3.init(pull = Pin.PULL_DOWN)    # 0 si pas d'input  
ou  
>>> p3.init(pull = Pin.PULL_UP)      # 3.3V si pas d'input  
puis  
>>> p3.value()    # lecture de la valeur  
1
```

Exercice

Configurer p3 en mode
PULL_UP et afficher sa valeur
dans la console toutes les
secondes.

Idem avec le bouton BOOT de
la carte (GPIO 9).



Module Pin : interruption

Principe : le changement d'état de la pin déclenche l'appel (quasi)instantanée d'une fonction ('handler')

transition 0 → 1  Pin.IRQ_RISING

transition 1 → 0  Pin.IRQ_FALLING

Exemple

```
def mon_handler(p):
    print('pin 3')

>>> p3 = Pin(3, Pin.IN, Pin.PULL_DOWN)
>>> p3.irq(trigger=Pin.IRQ_RISING, handler=mon_handler)
<IRQ>
```

Librairie time

sleep(2)	pause de l'exécution pendant 2 s
sleep_ms(10)	pause de l'exécution pendant 10 ms
sleep_us(100)	pause de l'exécution pendant 100 μ s
time()	valeur du compteur des secondes (origine arbitraire)
ticks_ms()	valeur du compteur des ms
ticks_us()	valeur du compteur des μ s

Exercice

En utilisant la fonction `ticks_ms` de la librairie `time`, modifier la fonction `my_handler` pour supprimer les rebonds à la fermeture de l'interrupteur (debouncing).

Librairies os et sys

- `os` : commandes Linux du système de fichiers
 - `remove`, `chdir`, `getcwd`, `listdir`, `mount`, `rmdir`, ...
- `sys` : fonctions système spécifiques
 - `sys.path` : liste des chemins d'accès aux librairies (commande `import`)
à mettre à jour lorsqu'on ajoute des répertoires avec des nouvelles librairies

Scripts de démarrage

2 scripts lancés successivement au boot :

- `boot.py` : script de configuration
contient toutes les options de configuration de l'utilisateur (imports, définition de fonctions, de variables, chemin vers les librairies, ...)
- `main.py` : script de lancement de la tâche principale
boucle de lecture des capteurs, commande des actionneurs, ...

Lancement d'un script au démarrage

Plusieurs options :

- copier le code du script dans `main.py`
- dans `main.py`, ajouter la ligne :

```
import mon_script
```

où `mon_script.py` est le nom du fichier script. S'il n'est pas dans le répertoire racine, son chemin d'accès doit être dans la liste `sys.path`

- dans `main.py`, ajouter la ligne :

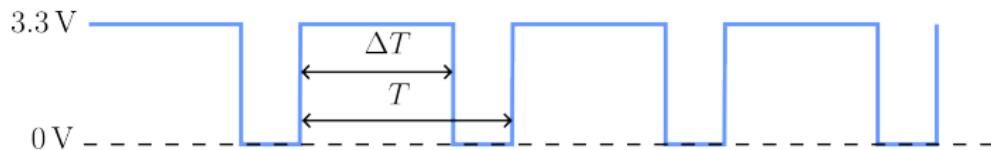
```
execfile(filename)
```

où `filename` est une chaîne de caractères contenant le nom complet du fichier (avec le répertoire et l'extension `.py`)

Exercice

Ecrire un script qui fait clignoter la LED (GPIO 8) et le lancer au boot.

Module PWM : Pulse Width Modulation

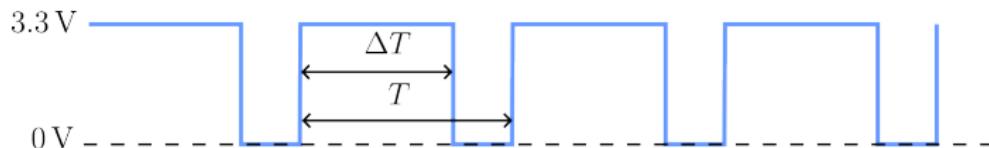


$$fréquence = 1/T$$

$$duty = \frac{\Delta T}{T} \in [0\%; 100\%]$$

$$moyenne \in [0V ; 3.3V]$$

Module PWM : Pulse Width Modulation



$$fréquence = 1/T$$

$$duty = \frac{\Delta T}{T} \in [0\%; 100\%]$$

$$moyenne \in [0V ; 3.3V]$$

Applications :

- commande de moteur à cc
- commande de moteur pas à pas
- commande de moteur brushless
- commande de servomoteur
- commande intensité LED

...

Exemple

```
>>> from machine import PWM  
>>> p4 = PWM(4, freq=2000, duty=800) # duty: 0 -> 1023 (100\%)  
>>> p4.freq(1000) # changer la fréquence  
>>> p4.duty(350) # changer le rapport cyclique  
>>> p4.freq() # renvoie la fréquence  
1000  
>>> p4.duty() # renvoie le rapport cyclique  
350
```

On peut augmenter la résolution sur le rapport cyclique

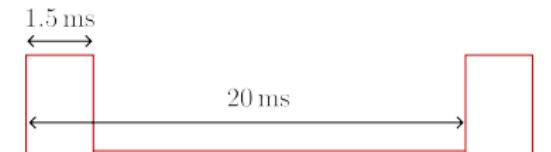
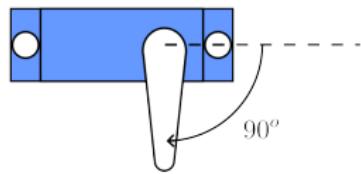
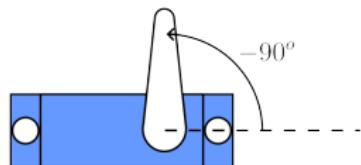
```
>>> p4.duty_ns(50000) # rapport cyclique en ns (50000ns = 50us)  
>>> p4.duty_u16(15000) # duty : 0 -> 65535 = 2**16-1(100\%)
```

⚠ Pour l'ESP32, fréquence $\geq 5 \text{ Hz}$ (période $\leq 0,2 \text{ s}$)

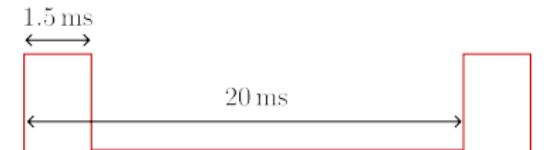
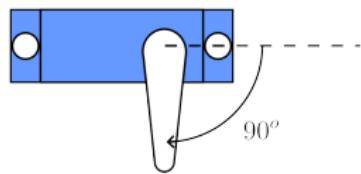
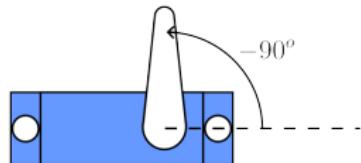
Exercice

Une LED alimentée par une tension PWM clignote à la fréquence du signal. Pour les fréquences supérieures à quelques dizaines de hertz, l'oeil ne perçoit pas le clignotement et voit un éclairement moyen.
Ecrire un script qui fait passer progressivement l'éclairement de la LED de 0 à 100% en 2 s, puis éteint la LED.

Application : commande de servomoteur SG90



Application : commande de servomoteur SG90



Exercice

Ecrire un script qui envoie un signal PWM sur la commande (cmd), et qui définit une fonction angle(x) qui positionne le servo à la position x $\in [-90^\circ; 90^\circ]$

Module Timer

Un timer provoque l'appel, périodiquement ou une seule fois, d'une fonction définie par l'utilisateur (ISR ou callback). Si un script est en cours d'exécution, il est interrompu et il reprend à la fin de l'appel.

Applications

- surveillance d'un capteur
- rafraîchissement d'un affichage
- commande d'un moteur pas à pas
- ...

Exemple

```
def compteur(t):
    global n
    print(n)
    n += 1

>>> from machine import Timer
>>> n = 0
>>> tim = Timer(0, period=1000, callback=compteur)
>>>
0
1
...
>>> tim.deinit()    # pour désactiver le timer
```

Attention

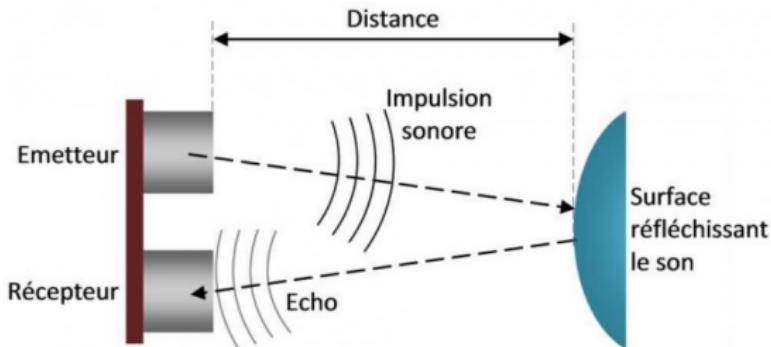
- la fonction callback prend un argument (timer t)
- pour l'ESP32C3 mini, les numéros de Timer sont toujours pairs : 0, 2, 4, ...



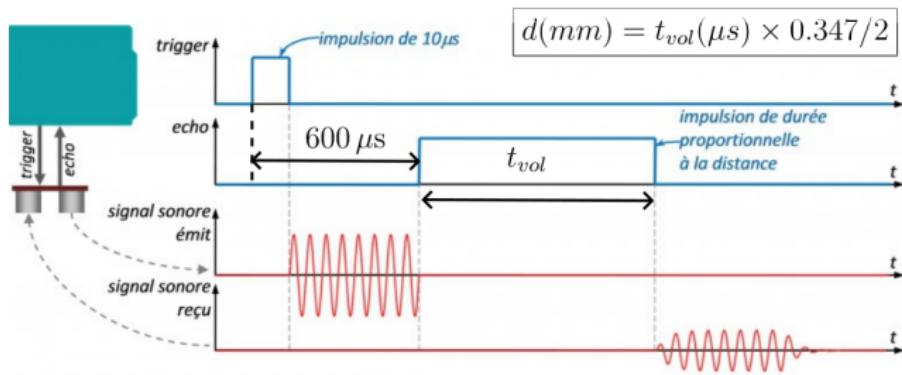
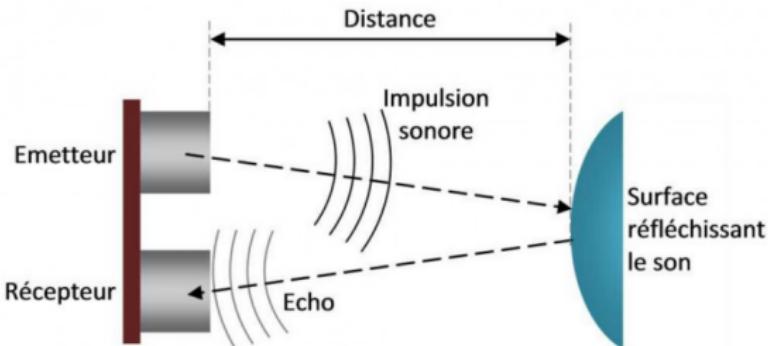
Exercice

Ecrire une fonction `blink` qui change l'état de la LED onboard (GPIO 8), puis créer un timer qui utilise cette fonction comme callback pour faire clignoter la LED.

Capteur de distance ultrasons HC-SR04



Capteur de distance ultrasons HC-SR04



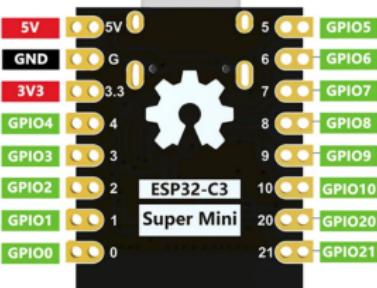
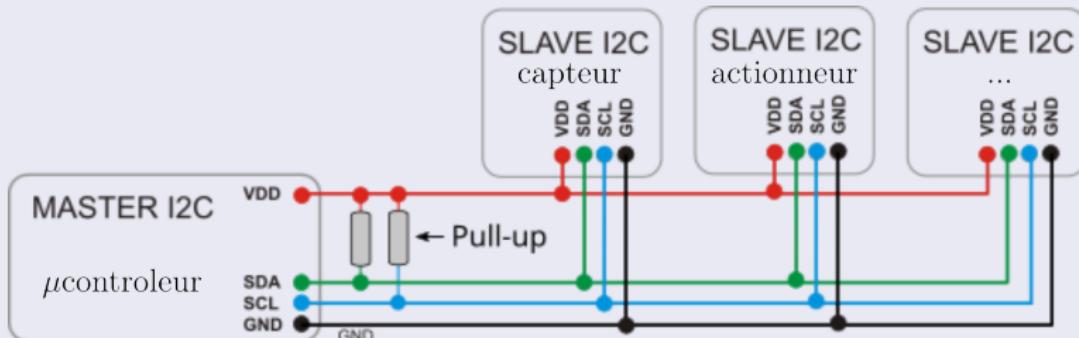
Fonction mesure de distance

```
from machine import Pin
from time import sleep_us, ticks_us

# initialisation pins et constantes
trig = Pin(...)
echo = Pin(...)
cson = 0.347      # vitesse du son en mm/us

# fonction de lecture du capteur
def mesure()
    génération de l'impulsion trigger
    attente front montant echo
    déclenchement chrono
    attente front descendant echo
    arrêt chrono
    conversion temps -> distance
    renvoyer le résultat
```

Module I2C (Inter Integrated Circuit)



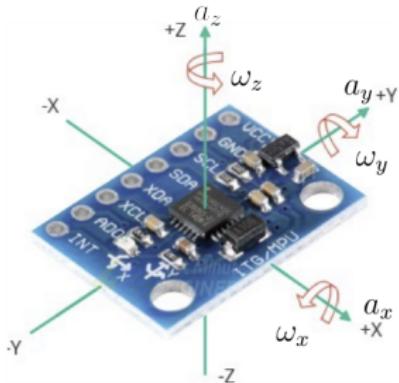
adressage 7 bits → 128 périphériques
débit ~ 400 kB/s
sda et scl : 2 pins quelconques

Applications : capteur de température/pression, accéléromètre, centrale inertie (IMU), driver moteur/servomoteurs, afficheur OLED, ...

Remarque : adresse I2C de l'esclave imposée par le fabricant → impossible de mettre 2 périphériques identiques sur le même bus.

```
>>> from machine import SoftI2C  
>>> i2c = SoftI2C(sda=4, scl=3)  
>>> i2c.scan()  
[104]      (liste des adresses des périphériques trouvés  
           sur le bus)
```

MPU6050 - Présentation



Algorithme de fusion

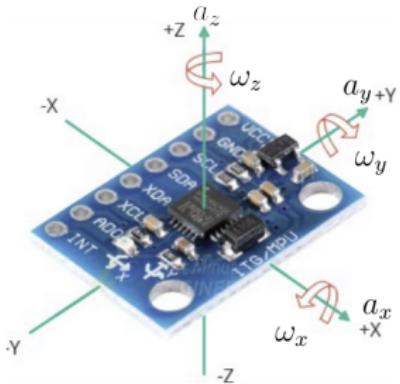
accélération linéaire \vec{a}

champ de gravité \vec{g}

position angulaire :

- roulis, tangage, lacet
- angles d'Euler

MPU6050 - Présentation



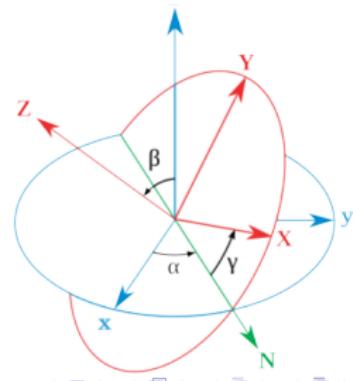
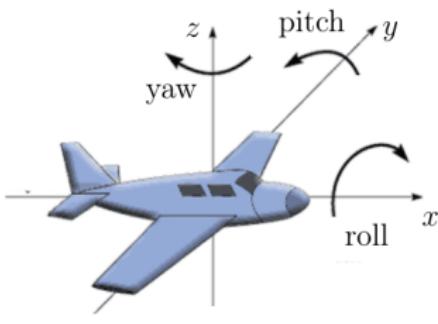
Algorithme de fusion

accélération linéaire \vec{a}

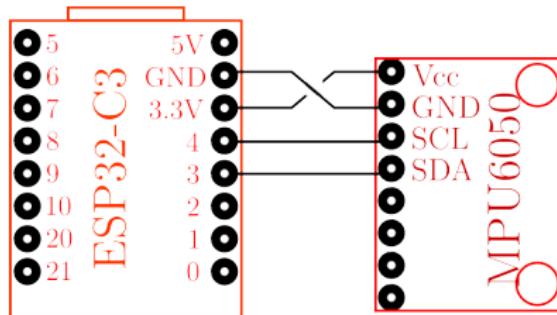
champ de gravité \vec{g}

position angulaire :

- roulis, tangage, lacet
- angles d'Euler



MPU6050 - Librairie MPU6050dmp20



```
>>> from machine import SoftI2C  
>>> from MPU6050dmp20 import *  
>>> i2c = SoftI2C(sda=3, scl=4)  
>>> mpu = MPU6050dmp(i2c)  
***** MPU6050dmp init : MPU6050 found address 0x68  
>>>
```

Mesure des offsets

```
>>> mpu.calibrate()
***** MPU6050 calibration init - Hit CTRL-C to stop
axOff=-4523, ayOff=507, azOff=620, gxOff=130, gyOff=24, gzOff=39
...
***** MPU6050 calibration interruption
Offsets set to axOff=-4452, ayOff=511, azOff=628, gxOff=130, gy
>>>
```

Mesure des offsets

```
>>> mpu.calibrate()
***** MPU6050 calibration init - Hit CTRL-C to stop
axOff=-4523, ayOff=507, azOff=620, gxOff=130, gyOff=24, gzOff=39
...
***** MPU6050 calibration interruption
Offsets set to axOff=-4452, ayOff=511, azOff=628, gxOff=130, gy
>>>
```

Rebooter l'ESP32 (Ctrl-D)

```
>>> from machine import SoftI2C
>>> from MPU6050dmp20 import *
>>> i2c = SoftI2C(sda=3, scl=4)
>>> mpu = MPU6050dmp(i2c, axOff=-4452, ayOff=511, azOff=628,
                     gxOff=130, gyOff=26, gzOff=39)
***** MPU6050dmp init : MPU6050 found address 0x68
>>> mpu.dmpInitialize()
>>> mpu.setDMPEnabled(True)
>>> mpu.getIntStatus()
```

Après initialisation du dmp, données mises à jour toutes les 10 ms
(100 Hz)

```
def getData():
    mpu.resetFIFO()
    mpu.getIntStatus()
    while mpu.getFIFOCount() != 42:
        if mpu.getFIFOCount() > 42:
            mpu.resetFIFO()
    buf = mpu.getFIFOBytes(42)
    quat = mpu.dmpGetQuaternion(buf) # quaternion
    acc = mpu.dmpGetFifoAccel(buf)   # accel - grav
    gyro = mpu.dmpGetFifoGyro(buf)   # vitesse angulaire
    grav = mpu.dmpGetGravity(quat)   # gravité
    linac = mpu.dmpGetLinearAcc(grav, acc) # accélération
    yaw, pitch, roll = mpu.dmpGetYawPitchRoll(quat, grav)
    theta, phi, psi = mpu.dmpGetEuler(quat)
    return yaw, pitch, roll
```

Système de fichiers

Deux types de fichiers

- fichier texte : contient des chaînes de caractères encodées en utf-8.
Les lignes se terminent par '\r\n' (retour chariot et nouvelle ligne).
- fichier binaire : peut contenir n'importe quelle suite d'octets.

Création d'un fichier texte

```
>>> fd = open('test.txt', 'w') # 'w'= fichier texte ouvert
                                # en écriture
>>> fd.write('debut\r\n')          # 1ère ligne
8
>>> fd.write('ceci est un test\r\n') # 2e ligne
18
>>> fd.write('fin\r\n')           # 3e ligne
5
>>> fd.close()                  # fermeture du fichier
```

Lecture d'un fichier texte existant

```
>>> fd = open('test.txt', 'r') # 'r'= fichier texte ouvert
                                # en lecture
>>> fd.readline()           # on lit la 1ère ligne
'debut\r\n'
>>> fd.readline()           # on lit la 2e ligne
'ceci est un test\r\n'
>>> fd.readline()           # on lit la 3e ligne
'fin\r\n'
>>> fd.close()              # fermeture du fichier
```

ou bien

```
>>> fd.readlines(). # on lit toutes les lignes d'un coup
['début\r\n', 'ceci est un test\r\n', 'fin\r\n']
>>> fd.close()
```

Exercice

Ecrire un script qui enregistre dans un fichier texte, chaque seconde et pendant 10s, la date, l'heure et la température du microcontrôleur (une ligne par enregistrement).

Pour l'horodatage, utiliser la fonction `localtime` de la librairie `time`.

La température est donnée par la fonction `mcu_temperature` (sans accent !) de la librairie `esp32`.

module UART

Protocole de liaison série full duplex

Chaines de caractères <class 'str'>

```
>>> s = 'a b_c-d'      # ou s = "a b_c-d"
>>> len(s)        # longueur
7
>>> s[0]          # 1er caractère
'a'
>>> s[1]          # 2e caractère
' '
>>> s[-1]         # dernier caractère
'd'
>>> s[2:5]        # sous-chaine
'b_c'
>>> s[2] = 'x'
TypeError: 'str' object does not support item assignment
```

⚠ une chaîne de caractères est un objet *non mutable*



Chaines d'octets <class 'bytes'>

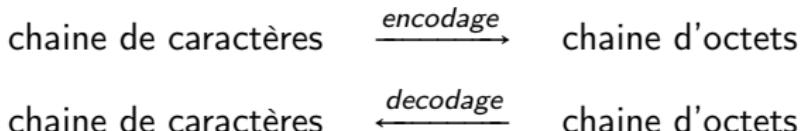
```
>>> r = b'\x00\x01\x02'  
>>> len(r)      # longueur  
3  
>>> r = b'\x61\x0d\x0a'  
b'a\r\n'  
>>> r[0]  
97      (0x61)  
>>> r[1]  
13      (0x0d)  
>>> r[2]  
10      (0x0a)
```

Une chaine d'octets est une **liste d'octets** → chaque élément de la liste est **un entier compris entre 0 et 255**

```
>>> r[1] = 10
TypeError: 'bytes' object does not support item assignment
```

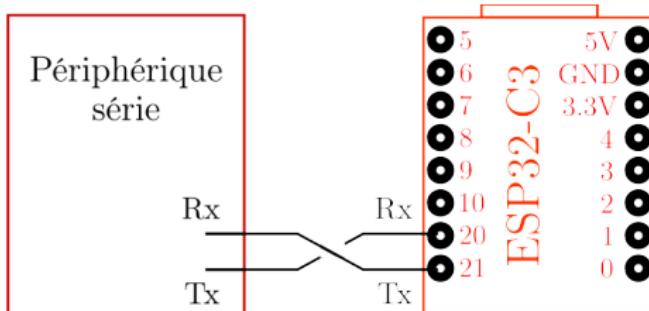
⚠ une chaîne de bytes est un objet *non mutable*

Encodage et décodage (utf-8)



```
>>> s = 'première ligne\r\n'
>>> s.encode()
b'premi\xc3\xaa8re ligne\r\n'
>>> r = b'\x0d\x0a'
>>> r.decode()
'\r\n'
```

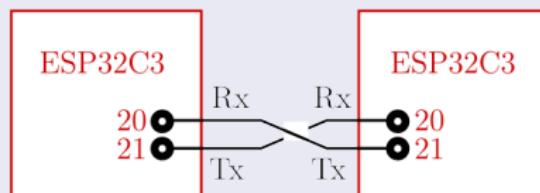
Module UART



```
>>> from machine import UART
>>> u = UART(1, rx=20, tx=21)
>>> u
UART(1, baudrate=115211, bits=8, parity=None, stop=1, tx=21,
      rx=20, rts=-1, cts=-1, txbuf=256, rxbuf=256,
      timeout=0, timeout_char=0)
>>> u.write('chaine de caractères envoyée')
30          (nombre de caractères envoyés)
```

```
>>> u.read()      # lit le contenu du buffer d'entrée  
b"chaine d'octets re\xc3\xaeue"  
>>> u.read(10)   # lit au plus 10 octets dans le buffer d'entrée  
>>> u.any()      # nombre d'octets en attente de lecture
```

Exercice : communication série entre deux ESP32C3

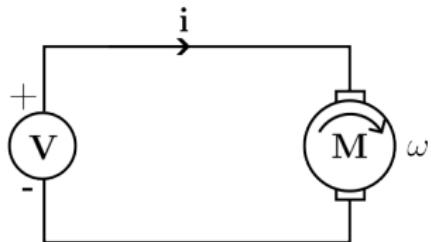
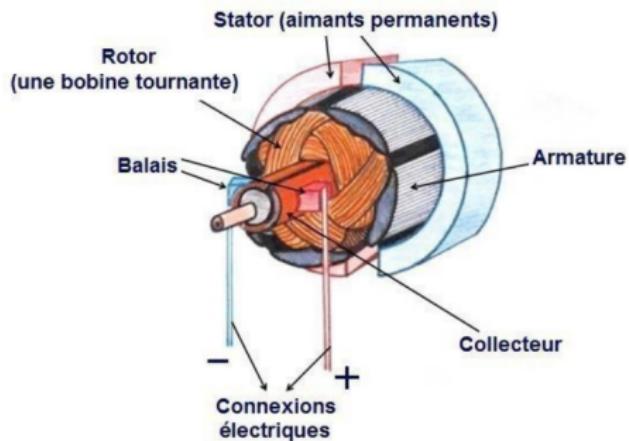


Etablir une connexion série entre les deux ESP32 et tester l'émission et la réception des deux côtés.

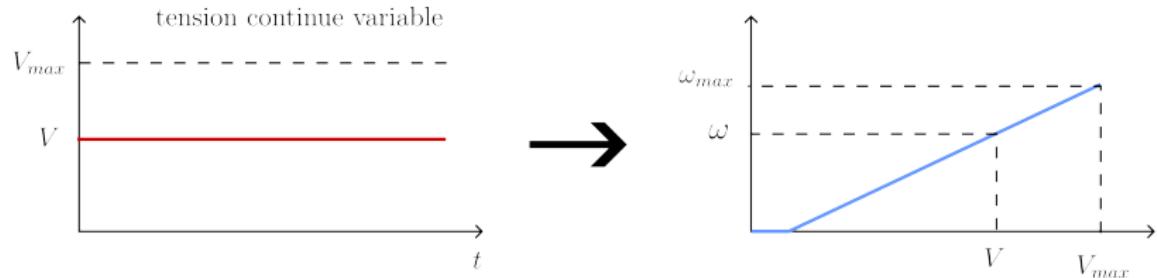
Allumer/éteindre la diode GPIO8 d'un ESP32 à partir de l'autre.

`exec(cmd)` → execute la commande cmd (chaine de caractères ou d'octets)

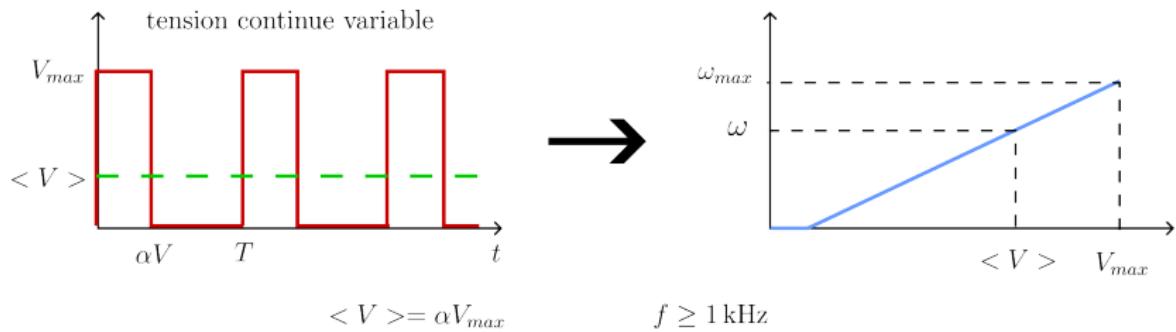
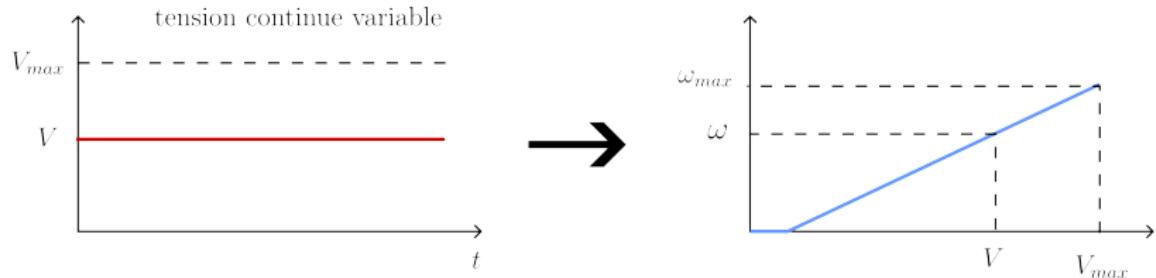
Moteur à courant continu



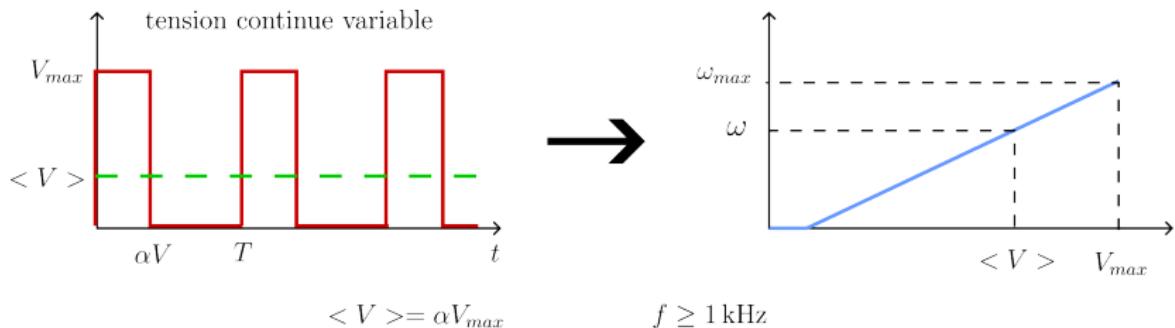
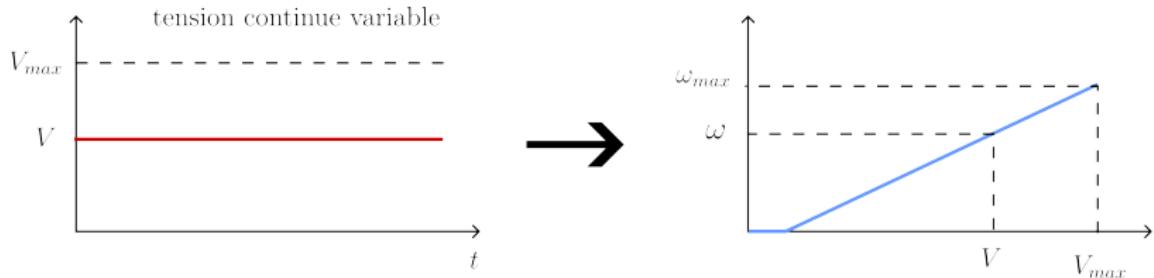
Moteur à courant continu



Moteur à courant continu



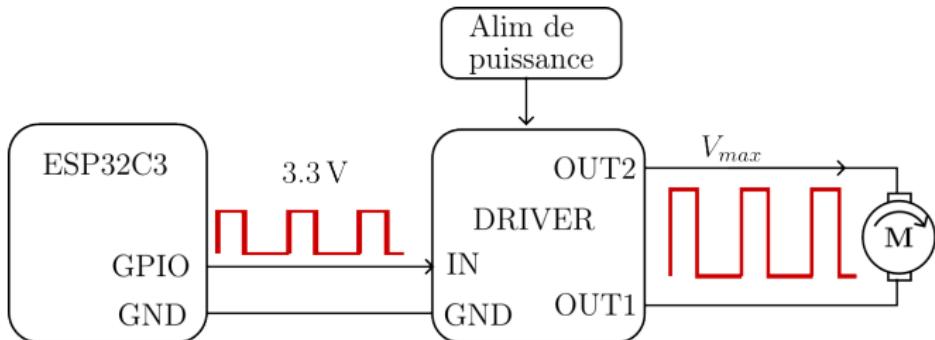
Moteur à courant continu



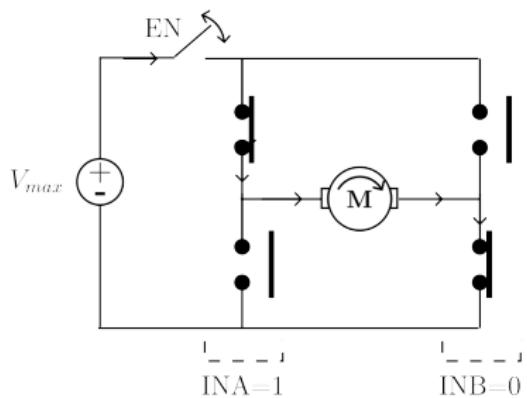
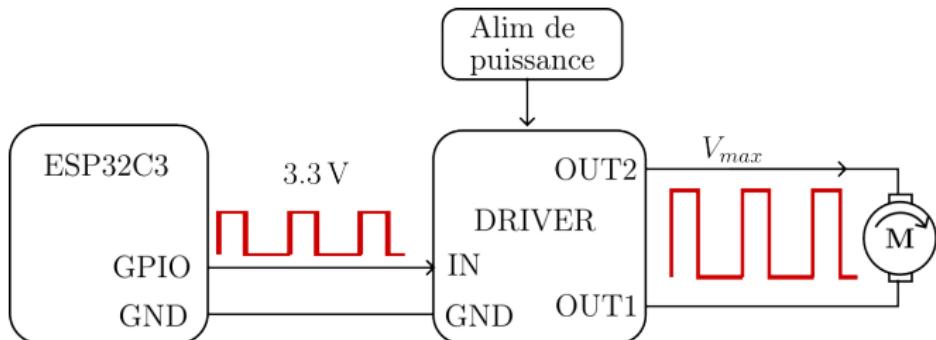
Pour les GPIOs : $V_{max} = 3,3 \text{ V}$ $i_{max} < 50 \text{ mA}$

⇒ utiliser un driver

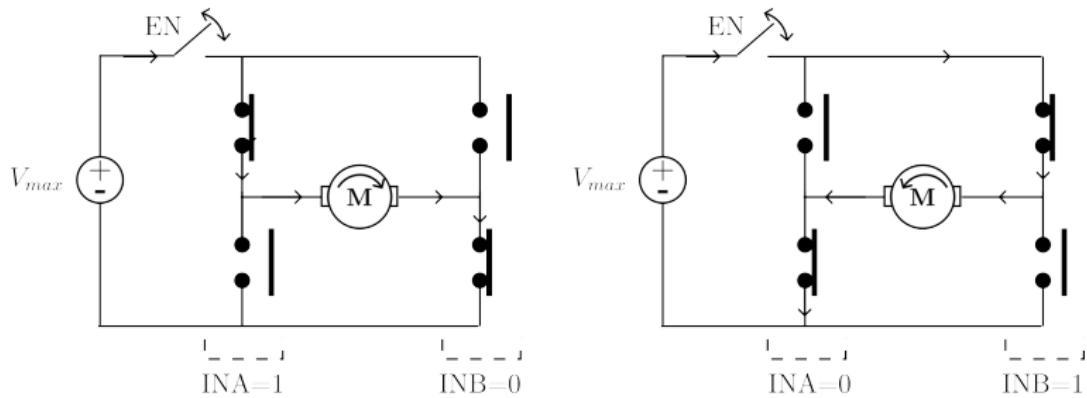
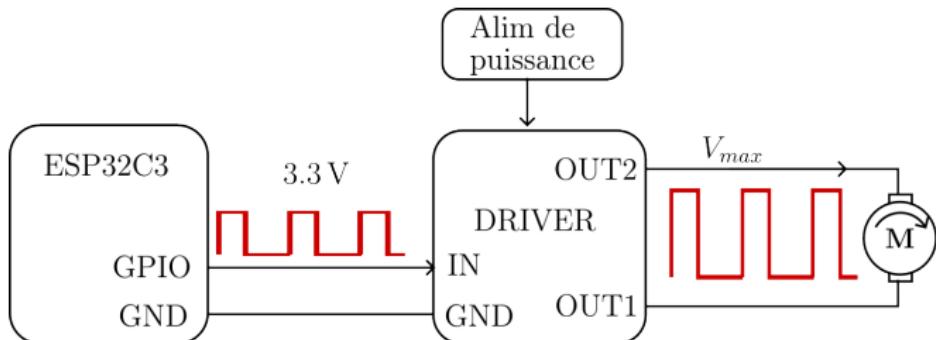
Moteur à courant continu



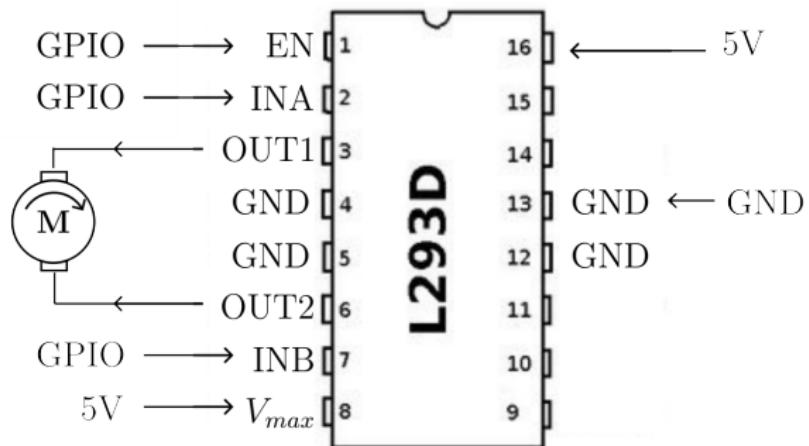
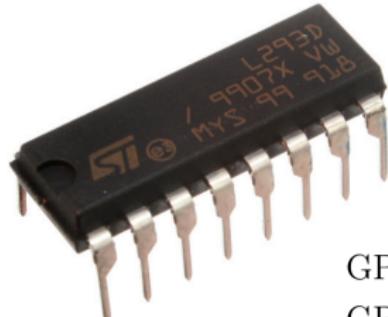
Moteur à courant continu



Moteur à courant continu



Moteur à courant continu



Exercice : Ecrire un script Commandes_moteur.py qui :

- importe les modules Pin et PWM
- initialise les pins INA et INB (sens) et EN (pwm de fréquence 1 kHz)
- définit une fonction avance(vit) et une fonction recule(vit) qui prennent comme argument une vitesse $vit \in [0\%; 100\%]$
- définit une fonction stop() qui arrête le moteur
- définit une fonction vitesse() qui renvoie la vitesse du moteur

Driver

```
>>> from machine import Pin  
>>> p0 = Pin(0, Pin.OUT) → p0 = objet de type Pin  
>>> p0.value(1)  
>>> p0.off()  
>>> p0  
Pin(0)  
>>> p1 = Pin(1, Pin.IN)  
>>>
```

Driver

```
>>> from machine import Pin
>>> p0 = Pin(0, Pin.OUT) → p0 = objet de type Pin
>>> p0.value(1)
>>> p0.off()
>>> p0
Pin(0)

>>> p1 = Pin(1, Pin.IN)
>>>
>>> from machine import PWM
>>> monpwm = PWM(p0, freq=500, duty=20) → monpwm = objet de type PWM
>>> monpwm.freq(100)
>>> monpwm.duty(85)
>>> monpwm
PWM(Pin(0), freq=100, duty=85, resolution=14, (duty=8.30%, resolution=0
```

Driver

```
>>> from machine import Pin
>>> p0 = Pin(0, Pin.OUT)           → p0 = objet de type Pin
>>> p0.value(1)
>>> p0.off()
>>> p0
Pin(0)

>>> p1 = Pin(1, Pin.IN)
>>>
>>> from machine import PWM
>>> monpwm = PWM(p0, freq=500, duty=20) → monpwm = objet de type PWM
>>> monpwm.freq(100)
>>> monpwm.duty(85)
>>> monpwm
PWM(Pin(0), freq=100, duty=85, resolution=14, (duty=8.30%, resolution=0

>>> from machine import Timer      → tim = objet de type Timer
>>> tim = Timer(2, period=15, callback=cb)
>>> tim.value()

0

>>> tim.deinit()
>>> tim

Timer(0, mode=PERIODIC, period=15)
```

Driver

```
>>> from machine import SoftI2C
>>> i2c = SoftI2C(scl=0, sda=1)
>>>
>>> from MPU6050dmp20 import MPU6050dmp
>>> mpu = MPU6050dmp(i2c) → mpu = objet de type MPU6050dmp
***** MPU6050dmp init : MPU6050 found address 0x68
>>> mpu.calibrate()
***** MPU6050 calibration init - Hit CTRL-C to stop
ax:-4096 ay: 515 az: 686 gx: 137 gy: 23 gz: 37
***** MPU6050 calibration interruption
Offsets set to ax:-4096 ay: 515 az: 686 gx: 137
>>> mpu.addr
104
>>> mpu.i2c
SoftI2C(scl=0, sda=1, freq=500000)
>>> mpu
<MPU6050dmp object at 3fcab400>
```

Driver

```
>>> from monfichier import type_objet  
>>> mon_objet = type_objet(args)
```

méthodes

attributs

```
>>> mon_objet.fonction1(args)  
>>> mon_objet.fonction2(args)
```

```
>>> mon_objet.attribut1  
>>> mon_objet.attribut2
```

Driver

```
>>> from monfichier import type_objet  
>>> mon_objet = type_objet(args)  
      ↓  
    méthodes           attributs  
>>> mon_objet.fonction1(args)      >>> mon_objet.attribut1  
>>> mon_objet.fonction2(args)      >>> mon_objet.attribut2
```

Driver moteur à courant continu

```
>>> from Driver_moteurCC import moteurCC  
>>> mg = moteurCC(pin1=0, pin2=1, pinpwm=2, freq=1000)  
>>> md = moteurCC(pin1=3, pin2=4, pinpwm=5, freq=1000)  
>>> mg.arriere(vitesse=30)  
>>> md.avant(vitesse=50)  
>>> mg.stop()  
>>> md.stop()
```

