Formation Juin 2019 :

⋆ ⋆ ⋆

Python pour l'enseignant de Physique-Chimie au lycée

⋆ ⋆ ⋆

Bonus 2

Table des matières

— *Le lien vers cet énoncé en ligne est sur le "padlet" (Bonus 2 formation) ; téléchargez-le et utilisez-le pour accéder aux liens cliquables de l'énoncé.*

Vous pouvez aussi télécharger l'énoncé à cette adresse :
`https://github.com/formationPythonPC-Juin/enonce-donnees/blob/master/bonus2.pdf`

— *Par rapport aux l'énoncés précédents, celui-ci ne propose pas de fichiers d'aides.*

Quoiqu'il en soit, les fichiers corrections sont donnés en lien à la fin de chaque partie.

Vous réalisez donc les exercices via le niveau 1 (pas d'aide) ou le niveau 3 (correction donnée, il vous faut alors commenter les lignes de code pour expliquer à quoi elles servent).

9 Résolutions numériques d'équations différentielles

Au vu de ce qui est fait en classe de 2nde et de 1ère, on peut imaginer qu'en classe de Terminale (Spécialité), les élèves pourront rencontrer des équations différentielles.

Python propose des méthodes pour approximer numériquement les solutions d'équations différentielles (ordre 1, 2 ou plus, linéaires ou non linéaires).

Donc, même sans connaître les solutions exactes dans des cas restreints (ordre 1 ou 2, linéaires et sans second membre), on peut "facilement" avoir une solution numérique grâce à la méthode `odeint()` du paquet `scipy.integrate`.

De plus, les modèles numériques permettent très rapidement d'implémenter des méthodes, encore il y a peu dans les programmes, telle la méthode d'Euler explicite.

Pour la suite, on s'intéressera à deux cas :
- chute libre avec frottements quadratiques : équation différentielle d'ordre 1 non linéaire ; l'équation est résolvable mais au prix de calculs relativement longs.
- pendule simple hors de l'approximation des petits angles : équation différentielle d'ordre 2 non linéaire ; pas de solution exacte.

La suite de ce document se propose de voir ces deux cas à travers 3 méthodes de résolution numérique :
- la méthode d'Euler explicite
- la méthode d'Euler implicite
- l'utilisation d'une autre méthode numérique plus évoluée : la fonction `odeint()`.

REMARQUE : Dans l'hypothèse où les élèves de Terminale ne connaîtront pas les solutions classiques d'équations différentielles, il peut tout de même être intéressant d'utiliser les méthodes d'Euler ou la méthode `odeint` pour résoudre des cas classiques (chute libre sans frottement, avec frottements linéaires, pendule simple dans l'approximation des petits angles ...)

Il faut noter qu'il existe encore de nombreuses méthodes de résolution numérique (on peut citer la méthode de Verlet, la méthode de Runge-Kutta, la méthode des éléments finis ...).

On étudie ici les méthodes les plus simples (méthodes d'Euler) ainsi qu'une méthode hybride fournie par Python (via la fonction `odeint`).

9.1 Démonstrations générales pour les méthodes d'Euler

On suppose une équation différentielle du type $\frac{dy}{dt} = f(t, y)$

9.1.1 Première démonstration

Avec celle-ci, on comprend bien le passage vers Euler explicite
- discrétisation du temps : on coupe notre intervalle de temps avec un pas de temps h ; les temps sont discrétisés : $t_0, t_1, t_2, \dots t_n, t_{n+1}$
- on peut écrire (développement de Taylor) que $y(t_{n+1}) = y(t_n + h) = y(t_n) + h \times \frac{\mathrm{d}y}{\mathrm{d}t}(t_n) + \frac{h^2}{2!} \times \frac{\mathrm{d}^2y}{\mathrm{d}t^2}(t_n) + o(h^3)$
- Soit, si h est assez petit : $y(t_{n+1}) \approx y(t_n) + h \times \frac{\mathrm{d}y}{\mathrm{d}t}(t_n)$
- Et finalement, $\boxed{\frac{\mathrm{d}y}{\mathrm{d}t}(t_n) \approx \frac{y(t_{n+1}) - y(t_n)}{h}}$ où h est le pas de temps.

$\Longrightarrow$ on remplacera toutes les dérivées par cette formule dans les méthodes d'Euler.

Soit : $\boxed{y(t_{n+1}) = y(t_n) + h \times f(t, y)}$: **on a le schéma d'Euler explicite.**

Le problème c'est qu'on ne voit pas vraiment le schéma d'Euler implicite ; voilà une seconde démonstration pour le faire apparaître clairement.

9.1.2 Seconde démonstration

$\frac{dy}{dt} = f(t, y)$: Intégrons cette équation entre t_n et t_{n+1} soit sur l'intervalle h :

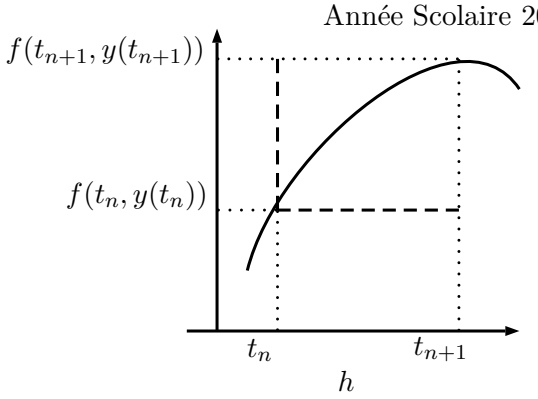
$\int_{t_n}^{t_{n+1}} \frac{dy}{dt} \cdot dt = \int_{t_n}^{t_{n+1}} f(t, y) \cdot dt$ soit $y(t_{n+1}) - y(t_n) = \int_{t_n}^{t_{n+1}} f(t, y) \cdot dt$

Le second membre pose problème, elle représente l'aire sous la courbe de la fonction f :

D'après le schéma ci-contre, on peut approximer cette aire de deux façons :

soit en la minorant (si f est croissante) et dans ce cas, l'aire sera : $h \times f(t_n, y(t_n)) \Longrightarrow$ **Euler explicite**

soit en la majorant (si f est croissante), et dans ce cas, l'aire sera : $h \times f(t_{n+1}, y(t_{n+1})) \Longrightarrow$ **Euler implicite**



9.1.3 Euler explicite

Reprenons notre équation de départ :
$\frac{dy}{dt} = f(t, y)$
Avec la méthode d'Euler explicite, on se base sur le comportement de dérivée, position... au point n :
$$\frac{\mathrm{d}y}{\mathrm{d}t}(t_n) \approx \frac{y(t_{n+1}) - y(t_n)}{h} \approx f(t_n, y(t_n))$$

9.1.4 Euler implicite

Reprenons notre équation de départ :
$\frac{dy}{dt} = f(t, y)$
Avec la méthode d'Euler implicite, on se base sur le comportement de dérivée, position... au point $n+1$:
$$\frac{\mathrm{d}y}{\mathrm{d}t}(t_n) \approx \frac{y(t_{n+1}) - y(t_n)}{h} \approx f(t_{n+1}, y(t_{n+1}))$$

Euler implicite diverge généralement moins qu'Euler explicite.

9.2 Utilisation de la fonction `odeint()`

Celle ci est utilisable via l'importation du paquet `scipy.integrate` :
from scipy.integrate import odeint
Le principe consiste :

1. à donner l'équation différentielle à Python, sous la forme d'une définition de fonction.
2. Puis de définir l'intervalle de temps et le pas de temps utilisé pour réaliser l'approximation de la solution.
3. De définir bien sûr les conditions initiales.
4. D'enfin faire appel à la méthode `odeint()` pour résoudre l'équation ; les paramètres donnés à cette fonction étant l'équation différentielle (1), les CI (3), le tableau de temps utile à la résolution (2).

Nous verrons cette méthode à travers les exemples.

9.3 Chute libre avec frottements quadratiques

Dans ce cas-là (frottements proportionnels à v^2, la solution exacte est longue à trouver. Le recours à des solutions numériques peut être envisagé.

La force de frottement peut se mettre sous la forme : $\vec{F} = -f \times v^2 \cdot \dfrac{\vec{v}}{|v|}$

L'équation différentielle donne :
$\begin{cases} m \cdot \frac{\mathrm{d}v}{\mathrm{d}t} = & -f \cdot v^2 + m \cdot g \\ \vec{v} = & \frac{\mathrm{d}\vec{r}}{\mathrm{d}t} \end{cases}$ soit en projection sur l'axe : $\begin{cases} m \cdot \frac{\mathrm{d}v}{\mathrm{d}t} = & -f \cdot v^2 + m \cdot g \quad (1) \\ v = & \frac{\mathrm{d}y}{\mathrm{d}t} \quad (2) \end{cases}$
Dans ce cas, on constate que les variables y et v ne sont pas liées, ce qui va nous simplifier la tâche.
Les CI sont $v(t=0) = 0$ et $y(t=0) = 0$.

Les valeurs des grandeurs entrant en jeu :
— $m = 80$ kg
— $f = 0,17$ SI

— $g = 9,81$ SI
— L'axe Oy est choisi positif vers le bas

9.3.1 Méthode d'Euler explicite

Il suffit de traduire la dérivée par le taux d'accroissement entre 2 instants proches séparés de h :
$m \cdot \frac{\mathrm{d}v}{\mathrm{d}t} \approx m \cdot \frac{v_{n+1} - v_n}{h} \approx -f \cdot {v_n}^2 + m \cdot g$
Soit encore : $\boxed{v_{n+1} = v_n - \frac{h}{m} \times (f \cdot {v_n}^2 - m \cdot g)}$ avec h le pas de temps, $h = dt$
De la seconde équation, on tire $\boxed{y_{n+1} = y_n + h \times v}$ avec h le pas de temps, $h = dt$
Ce sont ces deux équations qu'il va nous falloir traduire dans notre programme, en tenant compte des C.I.

Vous pourrez créer un programme `chute-euler-explicite.py`.

1. Donner les valeurs des grandeurs entrant en jeu dans l'étude dans des variables aux noms explicites.
2. De même, intégrer vos conditions initiales à votre programme. Par la même occasion, on fixera un temps initial $t = 0$ et un pas de temps pour les calculs d'une seconde : $dt = 1$ ou $h = 1$ selon vos notations.
3. Entrez alors les formules des forces intervenant : poids et frottements.
4. Initialisez des listes vides pour la position (Y1), la vitesse (V1) et le temps (T1).
5. Créez alors une boucle qui permet sur les 30 premières secondes de chute libre de :
   — implémenter vos listes Y1, V1 et T1 des valeurs de y, v et t au moment considéré
   — calculer à nouveau la force de frottements, le poids et la somme des forces
   — en déduire la nouvelle position en fonction de la précédente
   — en déduire la nouvelle vitesse en fonction de la précédente
   — vous n'oublierez pas d'augmenter alors le temps actuel du pas de temps choisi (dt ou h)
6. Représentez dès lors la vitesse de l'objet en fonction du temps.

► `lien vers la correction de cet exercice : chute-euler-explicite.py` ◄

9.3.2 Méthode d'Euler implicite

L'idée est la même sauf que cette fois-ci :
$m \cdot \frac{\mathrm{d}v}{\mathrm{d}t} \approx m \cdot \frac{v_{n+1} - v_n}{h} \approx -f \cdot {v_{n+1}}^2 + m \cdot g.$
L'ensemble est moins trivial car on voit se dessiner un polynôme de degré 2 en v_{n+1}.

On cherche son discriminant et la solution positive en v_{n+1} qui nous donne :
$v_{n+1} = \frac{-1 + \sqrt{1 + 4 \times (f \cdot h/m) \cdot (v_n + h \cdot g)}}{2 \cdot (f \cdot h/m)}$

Vous pourrez créer un programme `chute-euler-implicite.py`.

$\Longrightarrow$ Se servir de cette formule pour implémenter un code fournissant la représentation de la solution (vitesse en fonction du temps) de l'équation différentielle par la méthode Euler implicite (il n'y a quasiment rien à changer par rapport au code précédent).

► `lien vers la correction de cet exercice : chute-euler-implicite.py` ◄

9.3.3 Méthode `odeint()` du paquet `scipy.integrate`

La méthode `odeint()` fournit une résolution numérique plus précise que les deux méthodes vues précédemment.

Vous pourrez créer un programme `chute-odeint.py`.

$\Longrightarrow$ Interprétez chaque ligne du code fourni ici (`chute-odeint.py`) ; tâchez de comprendre à partir de ce dernier comment fonctionne la méthode odeint.

9.3.4 Comparaison des trois méthodes

$\Longrightarrow$ Sur un même graphe, représentez les solutions (vitesse en fonction du temps) du problème de la chute libre avec frottements quadratiques par les méthodes d'Euler explicite, Euler implicite et par la méthode odeint.

Vous pourrez créer un programme `chute-3methodes.py`.

► `lien vers la correction de cet exercice : chute-3methodes.py`◄

9.4 Pendule simple en dehors de l'approximation des petits angles

L'équation différentielle du mouvement est : $\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} = -\frac{g}{\ell} \cdot \sin\theta$. On ne connaît pas de solution exacte pour cette situation.

Posons $\omega_0^2 = \frac{g}{\ell}$ alors $\ddot{\theta} = -\omega_0^2 \cdot \sin\theta$ où θ représente l'angle entre la verticale et l'axe du pendule, ℓ la longueur du fil, g l'accélération de la pesanteur.

9.4.1 Rappels mathématiques

Comme on a pu s'en rendre compte, les méthodes employées jusqu'ici ne concernent que des équations différentielles d'ordre 1.

Dès lors, une question se pose : Comment gérer des équations différentielles d'ordre supérieur ?

Les mathématiques nous apprennent que toute équation différentielle d'ordre n peut se mettre sous la forme de n équations différentielles d'ordre 1 ; on pourra résoudre toutes ces équations par les méthodes vues plus haut.

La conséquence indirecte et malheureuse de cela est qu'en cherchant à diminuer l'ordre de l'équation différentielle, on va créer des inconnues matricielles (vectorielles) comme on va le voir dans le cas du pendule simple (on diminue l'ordre des équations différentielles mais on augmente la dimension de l'espace des fonctions solutions).

9.4.2 Mise en équation dans le cas du pendule simple

On pose $\Omega = \dot{\theta}$ (pulsation) et on crée une variable vectorielle : $X = \begin{pmatrix} \theta \\ \Omega \end{pmatrix}$

et on obtient alors l'équation différentielle suivante :

$\dot{X} = \begin{pmatrix} \dot{\theta} \\ \ddot{\theta} \end{pmatrix} = \begin{pmatrix} \Omega \\ -\omega_0^2 \cdot \sin\theta \end{pmatrix}$

soit $\dot{X} = \begin{pmatrix} X[1] \\ -\omega_0^2 \cdot \sin X[0] \end{pmatrix}$

Comme on le constate dans cet exemple, nous nous sommes ramenés à 2 équations différentielles d'ordre 1 dans lesquelles les variables sont couplées. On a diminué l'ordre (ordre 1) mais on s'est alors ramené à des inconnues matricielles (vectorielles) et non plus scalaires.

Pour la suite on considère un pendule simple avec :

— $\ell = 0,25$ m

— $g = 10$ m.s^{-2}

— $\theta(t=0) = 0$

— $\dot{\theta}(t=0) = \Omega(t=0) = 2$ rad.s^{-1}

On limite l'étude aux 8 premières secondes d'évolution. On choisira un pas de temps $dt = 0,02$ s pour toutes les méthodes.

On cherche à représenter la pulsation en fonction du temps par les 3 méthodes vues plus haut.

9.4.3 Méthode d'Euler explicite

Vous pouvez créer un programme `pendule-euler-explicite.py`

L'équation différentielle vectorielle ci-dessus se traduit en approximant le développement de Taylor à l'ordre 1 par :

— $\theta_{n+1} - \theta_n = \Omega_n \times dt \Longrightarrow \theta_{n+1} = \theta_n + \Omega_n \times dt$

— $\Omega_{n+1} - \Omega_n = -\omega_0^2 \cdot dt \cdot \sin\theta_n \Longrightarrow \Omega_{n+1} = \Omega_n - \omega_0^2 \cdot dt \cdot \sin\theta_n$

1. Définir les constantes et les C.I.
2. Définir de même 3 listes vides `angle1`, `puls1` et `T1` vides qui contiendront les valeurs d'angles, de pulsations et de temps au cours du mouvement.
3. Créez une boucle pour les 10 premières secondes d'évolution au sein de laquelle :
   — vous implémentez les 3 listes créées vides précédemment
   — vous traduisez les deux équations vues précédemment
   — vous n'oublierez pas d'incrémenter le temps en fin de boucle
4. Représentez l'évolution de la pulsation au cours de ces 10 premières secondes ; conclusions.

▶ `lien vers la correction de cet exercice : pendule-euler-explicite.py`◀

9.4.4 Méthode d'Euler implicite

Vous pouvez créer un programme `pendule-euler-implicite.py`

La solution précédente diverge totalement ; on va essayer de stabiliser les solutions en modifiant la référence à l'angle dans le calcul de la pulsation :

— $\theta_{n+1} = \theta_n + \Omega_n \times dt$

— $\Omega_{n+1} = \Omega_n - \omega_0^2 \cdot dt \cdot \sin\theta_{n+1}$

L'ordre des calculs est ici essentiel, puisqu'on a besoin de la première équation pour résoudre la seconde.

$\Longrightarrow$ Se servir de ces formules pour implémenter un code fournissant la représentation de la solution (pulsation en fonction du temps) de l'équation différentielle par la méthode Euler implicite (il n'y a quasiment rien à changer par rapport au code précédent).

▶ `lien vers la correction de cet exercice : pendule-euler-implicite.py`◀

9.4.5 Méthode `odeint()` du paquet `scipy.integrate`

La méthode `odeint()` fournit une résolution numérique plus précise que les deux méthodes vus précédemment.

Vous pourrez créer un programme `pendule-odeint.py`.

$\Longrightarrow$ Trouvez par la méthode `odeint()` la représentation graphique de la pulsation en fonction du temps au cours des 10 premières secondes d'évolution.

AIDE :

Ici, il faudra choisir pour résoudre l'équation différentielle par `odeint()` une inconnue vectorielle qu'on a notée X plus haut.

D'après la définition de X, il faudra ensuite représenter $X[1]$ en fonction du temps.

▶ `lien vers la correction de cet exercice : pendule-odeint.py`◀

9.4.6 Comparaison des trois méthodes

$\Longrightarrow$ Sur un même graphe, représentez les solutions (pulsation en fonction du temps) du problème du pendule simple par les méthodes d'Euler explicite, Euler implicite et par la méthode odeint.

Vous pourrez créer un programme `pendule-3methodes.py`.

▶ `lien vers la correction de cet exercice : pendule-3methodes.py`◀

10 Communication Python ⟷ microcontrôleurs

Quand on souhaite faire des acquisitions via un capteur en utilisant un microcontrôleur (Arduino^{TM}, MicroPython, ...), on est souvent tenté de vouloir enregistrer les données d'acquisitions.

Deux cas peuvent se présenter alors :

— Votre microcontrôleur dispose d'une mémoire suffisante pour enregistrer les données ; dans ce cas, pas de problème.

— Votre microcontrôleur dispose de très peu d'espace mémoire (carte micro :bit) ou ne dispose pas de mémoire de stockage du tout (Arduino^{TM}) et dans ce cas : comment fait-on ?

10.1 La méthode de base

10.1.1 Présentation

Les logiciels utilisés pour programmer votre microcontrôleur (IDE pour Arduino^{TM}, Mu-Editor pour microPython, Programming Editor pour PICAXE ...) vont vous permettre de communiquer avec la carte.

Ces logiciels vont permettre de communiquer avec la carte dans les deux sens via le **port série** reliant l'ordinateur au microcontrôleur et généralement "symbolisé" par un câble USB.

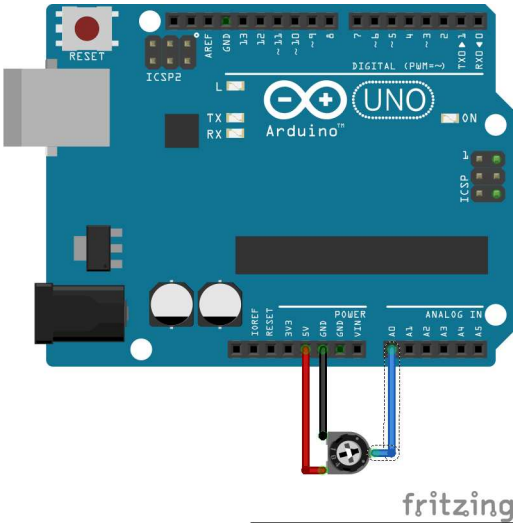
Le sens ordinateur → microcontrôleur est essentiel puisqu'il va permettre de programmer la carte.

Mais l'autre sens existe aussi : on peut faire passer des informations du microcontrôleur à l'ordinateur via le port série.

Ces logiciels disposent donc d'**une interface** permettant de voir ce qu'envoie le microcontrôleur. On lui donne le nom de moniteur série sous Arduino^{TM}, REPL dans les déclinaisons microPython, terminal Série chez PICAXE....

10.1.2 Cas pratique sous l'IDE Arduino^{TM}

Nous souhaitons acquérir les valeurs analogiques renvoyées par une carte microcontrôleur lors de la manipulation d'un potentiomètre.



Le potentiomètre est alimenté ici entre 0 et 5 Volts, et la broche centrale est placée sur une entrée analogique de la carte (ici A0 pour le microcontrôleur utilisé).

Le microcontrôleur transformera cette tension analogique en valeur numérique qu'on note `valeur` (entre 0 et 1023).

On souhaite voir afficher les valeurs que retourne le microcontrôleur sur l'interface série de l'ordinateur toutes les 500 ms.

Un code possible pourrait être :

Sous l'IDE Arduino :

Le premier bloc est un bloc de réglages, il n'est lu qu'une fois par Arduino ; on définit la broche A0 comme une entrée (de toute façon elle ne peut être autre chose), et puis surtout on définit la vitesse de communication avec le port série à 9600 bauds.

On constate ici, comme en Python qu'on utilise une bibliothèque appelée Serial. Cette bibliothèque est importée d'office par Arduino^{TM}, nous n'avons pas eu à le faire dans le préambule.

Puis on rentre dans une boucle infinie (loop).

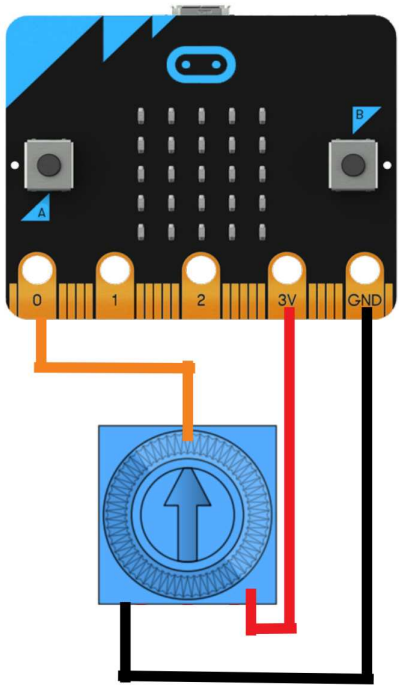
On définit tout d'abord une variable `valeur` comme un décimal (float) et qui vaut la valeur analogique lue sur la broche A0 ; donc l'image de la tension au centre du pont diviseur du potentiomètre.

Puis on refait appel à la bibliothèque Serial pour afficher (print) cette variable à travers le port série.

Il suffit alors d'envoyer le code sur la carte puis d'ouvrir le moniteur série (avec la petite loupe en haut à droite) et s'affichent toutes les 500 ms les valeurs analogiques correspondant à la tension au centre du pont diviseur.

Quand on a assez de mesures, il suffit alors de faire un copier-coller vers un fichier, qu'on pourrait appeler `donnees.csv` et faire ensuite un traitement par tableur ou par Python comme en 1.5 page 3.

10.1.3 Cas pratique sous Mu-Editor pour microPython (carte micro :bit)



Cette fois-ci, la tension délivrée par la carte vaut 3,3 Volts. Les branchements sont là encore basiques. L'entrée notée 0 peut être une entrée numérique ou analogique.

Le microcontrôleur transformera la tension analogique lue sur l'entrée 0 en valeur numérique qu'on note `valeur` (entre 0 et 1023).

On souhaite voir afficher les valeurs que retourne le microcontrôleur sur l'interface série de l'ordinateur toutes les 500 ms.

Un code possible pourrait être :

Sous Mu-Editor :

On importe tout d'abord la bibliothèque microbit qui contient quasiment toutes les modules utiles pour faire fonctionner la carte ; cette bibliothèque est spécifique à cette carte.

Puis on rentre dans une boucle infinie (while True) ; on définit alors une variable `valeur` par l'appel de la fonction `read_analog` de la broche 0 gérée par le module `pin0` de la bibliothèque microbit.

On affiche cette valeur. (Ici, pas d'appel à une bibliothèque Serial ; microPython "cache" la manipulation, et pourtant, les données vont bien voyager à travers le port série.

Puis on fait une pause de 500 ms grâce à la fonction sleep de la bibliothèque microbit.

Et la boucle infinie recommence.

Il suffit alors d'envoyer le code sur la carte puis d'ouvrir le REPL (icône au dessus), de relancer la carte et s'affichent toutes les 500 ms les valeurs analogiques correspondant à la tension au centre du pont diviseur.

Quand on a assez de mesures là encore, il suffit alors de faire un copier-coller vers un fichier, qu'on pourrait appeler `donnees.csv` et faire ensuite un traitement par tableur ou par Python comme en 1.5 page 3.

Des microcontrôleurs sont à votre disposition si vous désirez tester ces manipulations

10.2 Le microcontrôleur communique avec Python (sur l'ordinateur)

Il existe une autre méthode, plus compliquée mais aussi beaucoup plus puissante. En fait Python (de votre ordinateur) est capable de communiquer avec un microcontrôleur dans les deux sens. Il peut :

— **commander un microcontrôleur** : par exemple, une acquisition est lancée via le microcontrôleur quand l'utilisateur le décide. (Ce cas-là bien que très intéressant pour des manipulations où on ne souhaite pas toucher à la carte ni au programme implanté dessus, ne sera pas étudié car il varie selon la carte utilisée et fait appel à des fonctions parfois "exotiques" propres à chaque microcontrôleur).

— **Recevoir directement des informations d'un microcontrôleur** : dans ce cas-là, tout est automatique, et Python peut avec uniquement un seul programme, récupérer les données envoyées sur le port série, les enregistrer dans un fichier, les traiter pour représenter des graphes...

C'est ce deuxième cas que nous allons étudier. Il permet déjà de prendre conscience de la puissance de Python pour gérer des situations courantes dans notre discipline.

10.2.1 Compléments quant aux communications entre Python et les entrées / sorties

Pour gérer la communication avec le port série, Python dispose de la bibliothèque `pyserial` appelée par `import serial`.

Une première chose à savoir est que Python possède 2 "types" pour les chaînes de caractères :

— le "type" `str` ou `string` qui est présenté sur le tuto disponible sur le site académique

— le "type" `byte` (octet) qui traite les séquences d'octets ; les chaînes de caractères sont des suites d'octets ; Python les traduit lors d'affichage écran par des caractères à peu près compréhensibles par l'utilisateur.

$\Longrightarrow$ Quand vous utilisez un programme Python, tout ce qui rentre ou sort de votre programme est typé "`bytes`".

Cela signifie que si Python arrive à lire des données provenant par exemple du port série de l'ordinateur, elles seront de type `bytes` et il faudra donc pouvoir les transformer en format lisible (`string`) : cela se fait par la méthode `.decode("utf8")` ; l'utf-8 étant l'encodage utilisé par Python3.

10.2.2 Notre objectif

— Un microcontrôleur envoie des données sur le port série ; ces données ne sont pas lues par le moniteur série (Arduino^{TM}) le REPL (MicroPython), le terminal série (PICAXE)...

— Il est essentiel dans cette hypothèse de ne pas ouvrir l' interface série, sinon il y a conflit entre Python et cette interface.
— Nous souhaitons que notre programme Python puisse récupérer les données en provenance de la carte et les copie dans un fichier `donnees.csv`.
— Les codes utilisés pour les microcontrôleurs sont les mêmes que précédemment : on souhaite donc enregistrer la valeur lue par l'entrée analogique de la carte dans un fichier par l'intermédiaire de Python.
— Pour ce qui suit, la carte est déjà programmée par les codes vus précédemment ; elle fait son travail : elle envoie les données sur le port série ; voyons à présent comment Python peut les lire et les copier dans un fichier enregistré sur le disque dur de l'ordinateur.

10.2.3 Le code Python

On souhaite ici récupérer 10 valeurs des données renvoyées par la carte et les enregistrer dans un fichier `donnees.csv`.

Le code Python permettant de récupérer les données de la carte est donné ici :

► `lien vers le code : port-serie.py`◄

— Entrez le code sur l'éditeur de la carte ;
— Compilez ce code pour que la carte commence à envoyer les données ;
— Créez un fichier Python que vous pourrez appeler `port-serie.py` ;
— Dans ce fichier, copiez-collez l'intégralité du code proposé ci-dessus ;
— Lisez les différentes lignes, tâchez de comprendre leur rôle ;
— Compilez ce code ; que se passe-t-il ?
— Un nouveau fichier s'est créé à l'endroit où vous avez créé votre fichier Python. Ouvrez-le ; conclusions.

10.2.4 Conclusions et ouverture

— On peut imaginer envoyer depuis la carte un couple de données : `temps;valeur_analogique`
— Une fois le fichier .csv créé, on peut alors le traiter par Matplotlib et obtenir par exemple un graphe ; ce traitement peut se réaliser à la suite du code vu plus haut ; ainsi, la réception des données et leur traitement sont automatisés
— La méthode vue précédemment ne présente que peu d'intérêt pour enregistrer une dizaine de lignes de données ; mais imaginez la puissance de cette méthode quand il s'agit d'enregistrer quelques centaines de lignes et de tracer un graphe à la suite ; dans ce cas-là, Python est imbattable
— Lors d'un futur possible traitement par Matplotlib du fichier .csv, il faudra bien penser à convertir les données en nombres décimaux (pour l'instant, ce sont toujours des chaînes de caractères)
— Le code Python donné a été voulu très court, mais c'est une base pour pouvoir réceptionner ce qu'on veut, pendant le temps qu'on veut depuis un microcontrôleur. Libre à chacun(e) de le modifier pour en tirer toute sa "substantifique moelle"
— Comme expliqué plus haut, Python est aussi capable de commander un microcontrôleur, c'est à dire pour l'exemple que nous avons traité, de décider le moment où on démarre l'acquisition des données ; cela peut être très pratique lors de l'utilisation d'une carte en dispositif embarqué (sur un pendule par exemple), alors qu'on ne veut pas toucher à la carte et qu'on souhaite démarrer l'acquisition de façon décentrée. L'utilisation de Python prend alors une autre dimension puisqu'on peut dès lors réaliser une double communication `Python` → `microcontrôleur` suivie de `microcontrôleur` → `Python`.