TP de Physique-Chimie

PSI

P.E LEROY

19 mai 2026

Résumé

Vous trouverez dans ce document l'ensemble des TP de Physique-Chimie réalisés durant la seconde année PSI, classés dans l'ordre de la progression des TP.

Bon courage !
PE LEROY

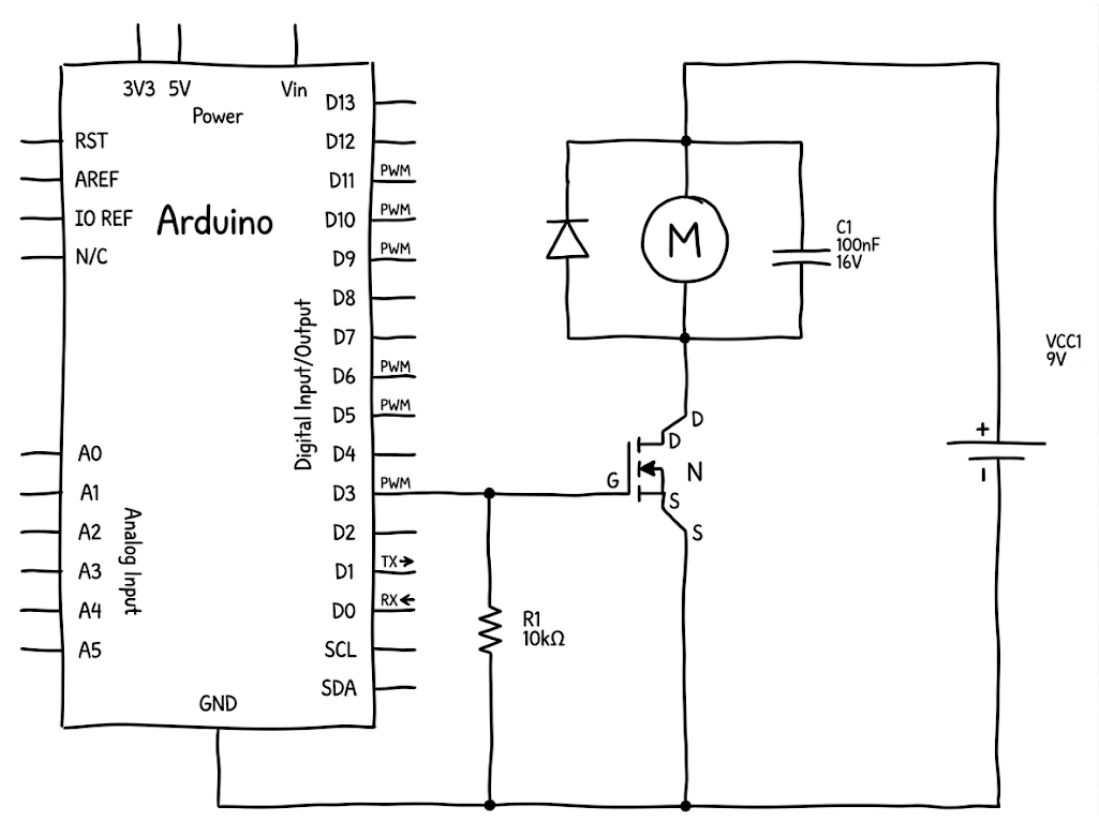


Table des matières

TP filtrage analogique

PE. LEROY

> Capacités exigibles du programme :
>
> **Filtrage analogique d'un signal périodique**
>
> - Mettre en évidence l'action d'un filtre linéaire sur un signal périodique dans les domaines fréquentiel et temporel.

Liste du matériel :

- Boîtes de résistance/capacité/inductance variables
- Oscilloscope
- GBF
- Multimètre Metrix rouge

On commence par importer les bibliothèques nécessaires :

> La ligne suivante `%matplotlib notebook` est une commande magique qui sert à l'affichage des graphes dans le notebook.

Entrée[ ]:
```
%matplotlib notebook
```

Entrée[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Réalisation et caractérisation d'un filtre passe-bas d'ordre 1

Nous allons réaliser un filtre passe-bas d'ordre 1 de fréquence de coupure $f_c \simeq 720$ Hz. Le filtre passe-bas le plus simple est un filtre RC.

Déterminer (au brouillon) l'expression de la fonction de transfert de ce filtre et la fréquence de coupure f_c en fonction de R et C. Sachant que habituellement $R \sim 1$ à $10\ \mathrm{k\Omega}$ (valeur à choisir), déterminer la valeur de C correspondante.

Mesurer précisément les valeurs de R et C choisies au multimètre, puis en déduire la valeur exacte de f_c :

Entrée[ ]:
```
# Valeurs à mesurer au multimètre
R=
C=

# Valeur à calculer
fc=
print(fc)
```

Réaliser le filtre, puis en l'alimentant avec une tension d'entrée sinusoïdale vérifier que :

- **l'amplitude du signal de sortie diminue lorsque la fréquence augmente ;**
- **à la fréquence de coupure f_c, l'amplitude du signal de sortie est environ égale à l'amplitude du signal d'entrée divisée par $\sqrt{2}$:**

> Attention l'amplitude mesurée à l'oscilloscope est souvent une amplitude crète à crète (<< pic to pic >>), on a donc $E = E_{cc}/2$ et S=$S_{cc}/2$.

Entrée[ ]:
```
# À fc :
E=
S=
# Pour comparer :
print(S)
print(E/np.sqrt(2))
```

On peut aussi faire le calcul de l'écart normalisé. Pour comparer 2 mesures, on utilise **l'écart normalisé** ou **z-score** :

$$E_N = \frac{|m_2 - m_1|}{\sqrt{u(m_1)^2 + u(m_2)^2}}$$

Par convention, on qualifie deux résultats de **compatibles** si leur écart normalisé vérifie la propriété :

$$\boxed{E_N \lesssim 2}$$

> Ce seuil à 2 est d'origine historique. On le retrouve dans de nombreux champs scientifiques, comme la médecine, la pharmacie, la biologie, la psychologie, l'économie, l'écologie, etc. Ce seuil peut différer selon le domaine : par exemple pour démontrer l'existence d'une nouvelle particule en physique subatomique, il faut atteindre un seuil de 5.

Pour effectuer ce calcul il faut connaître les incertitudes types sur chaque grandeur, S d'une part et $E/\sqrt{2}$ d'autre part.

Les incertitudes types sur les mesures de tensions sont données dans la documentation de l'oscilloscope :

> Nous sommes dans une configuration sans offset (ajout d'une tension continue).

DC gain accuracy	offset and position = 0, maximum operating temperature change of ±5 °C after self-alignment	
	input sensitivity > 5 mV/div	±1.5 % of full scale
	input sensitivity ≤ 5 mV/div	±2 % of full scale
Offset accuracy		±(0.5 % × |offset| + 0.1 div × input sensitivity + 1 mV)
DC measurement accuracy	after adequate suppression of measurement noise by using high-resolution sampling mode or waveform averaging	±(DC gain accuracy + offset accuracy)

Pour l'incertitude type sur $E/\sqrt{2}$ on peut utiliser les règles de calcul des incertitudes :

$$u(E/\sqrt{2}) = u(E)/\sqrt{2}$$

> Ces règles de calcul ont été vue en première année, pour rappel elles sont indiquées dans ce notebook (../incertitudes/incertitudes_de_mesure.ipynb).

Calculer les incertitudes types sur S puis sur $E/\sqrt{2}$:

Entrée[ ]:
```
uS=
uEr2=
```

Calculer l'écart normalisé entre les deux valeurs précédentes et conclure.

Entrée[ ]:
```
EN=np.abs(S-E/np.sqrt(2))/np.sqrt(uS**2+uEr2**2)
print(EN)
```

Nous allons réaliser le tracé de la courbe de gain du filtre. Pour cela, il faut récupérer pour différentes fréquences les valeurs des amplitudes du signal d'entrée et de sortie.

> On peut récupérer les amplitudes crête à crête directement puisque le gain est donné par $G = S/E$ mais aussi $G = S_{cc}/E_{cc}$ (le facteur 2 se simplifie).

Réaliser la prise de mesures correspondante (remplacer les valeurs actuelles).

> Le choix des gammes 1-2-5 et 8 c'est-à-dire 10-20-50 et 80 Hz puis 100-200-500 et 800 Hz et ainsi de suite, permet d'obtenir des points régulièrement répartis sur le graphe.

Entrée[ ]:
```
# Fréquences en Hertz
frequences = [10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000]

# Amplitude CC du signal d'entrée en Volts
E_cc = [5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0]

# Amplitude CC du signal de sortie en Volts
S_cc = [4.95, 4.8, 4.5, 3.5, 2.5, 0.5, 0.25, 0.05, 0.025]
```

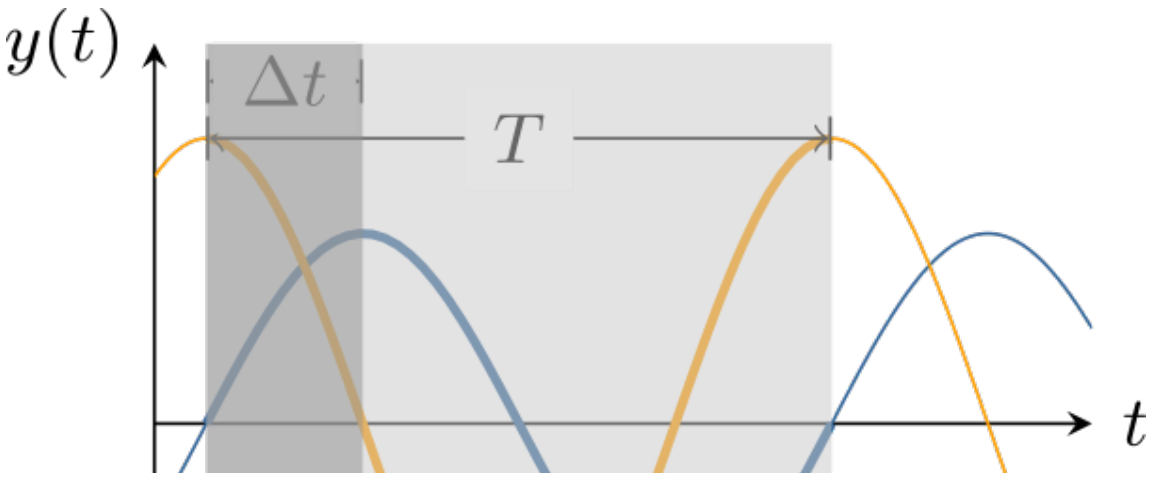
Tracer le diagramme de Bode (courbe de gain) du filtre ainsi élaboré.

Entrée[ ]:
```
# Conversion en tableaux numpy pour faciliter les calculs
f = np.array(frequences)
Ecc = np.array(E_cc)
Scc = np.array(S_cc)

# Calcul du Gain G=Scc/Ecc
G = Scc / Ecc

# Calcul du Gain en décibels (dB) : GdB = 20 * log10(Scc/Ecc)
GdB = 20 * np.log10(G)

# Tracé du diagramme de Bode (en gain)
plt.figure()

# Utilisation de semilogx pour l'échelle logarithmique en abscisse (
plt.semilogx(f, GdB, marker='o', linestyle='-', color='b', label='Mes

# Mise en forme du graphique
plt.title("Diagramme de Bode : Gain en fonction de la fréquence")
plt.xlabel("Fréquence (Hz) [Échelle Log]")
plt.ylabel("Gain (dB)")

# Activation de la grille (principale et secondaire pour bien voir l
plt.grid(True, which="both", ls="-", alpha=0.6)

# (Optionnel) Ajouter une ligne rouge à -3dB pour repérer la fréquenc
plt.axhline(y=-3, color='r', linestyle='--', label='Seuil -3dB')

plt.legend()
plt.tight_layout()

# Affichage
plt.show()
```

Vérifier que la pente de l'asymptote haute fréquence est proche de -20 dB/décade :

Entrée[ ]:
```
# Calcul à adapter, prendre deux valeurs séparées d'une décade
print(GdB[-1]-GdB[-3])
```

Nous allons maintenant tracer le diagramme de Bode en phase de ce filtre. La mesure d'un déphasage s'effectue par une mesure de décalage temporel Δt entre les deux courbes :



On peut alors montrer que :

$$\varphi = \pm\, \omega\, \Delta t = \pm\, 2\pi f\, \Delta t$$

Le signe est donné par l'avance ou le retard de la courbe de $e(t)$ par rapport à celle de $s(t)$ (celle qui est en avance est celle qui passe par un maximum en premier) :

$$\varphi = arg(\underline{H}) = \varphi_s - \varphi_e$$

et donc que $\varphi > 0$ si $\varphi_s > \varphi_e$ c'est à dire si $s(t)$ est en avance sur $e(t)$, et réciproquement.

Réaliser la prise de mesures correspondante (remplacer les valeurs actuelles).

> Le choix des gammes 1-2-5 et 8 c'est-à-dire 10-20-50 et 80 Hz puis 100-200-500 et 800 Hz et ainsi de suite, permet d'obtenir des points régulièrement répartis sur le graphe.

Entrée[19]:
```
# Fréquences en Hertz
frequences = [10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000]

# Décalage temporel du signal de sortie par rapport au signal d'entre
# Attention au signe, à vérifier graphiquement !
Delta_t = [5.0e-3, 5.0e-3, 5.0e-3, 5.0e-3, 5.0e-3, 5.0e-3, 5.0e-3, 5.
```

Tracer le diagramme de Bode (courbe de phase) du filtre ainsi élaboré.

Entrée[ ]:
```
# Conversion en tableaux numpy pour faciliter les calculs
f = np.array(frequences)
Dt = np.array(Delta_t)

# Calcul du déphasage du signal de sortie par rapport au signal d'ent
phi = 2*np.pi*f*Dt

# Tracé du diagramme de Bode (en gain)
plt.figure()

# Utilisation de semilogx pour l'échelle logarithmique en abscisse (f
plt.semilogx(f, GdB, marker='o', linestyle='-', color='b', label='Mes

# Mise en forme du graphique
plt.title("Diagramme de Bode : Déphasage en fonction de la fréquence'
plt.xlabel("Fréquence (Hz) [Échelle Log]")
plt.ylabel("Déphasage (rad)")

# Activation de la grille (principale et secondaire pour bien voir l'
plt.grid(True, which="both", ls="-", alpha=0.6)

plt.legend()
plt.tight_layout()

# Affichage
plt.show()
```

Réalisation et caractérisation d'un filtre passe-bande d'ordre 2

Sur le même principe, nous allons tracer le diagramme de Bode en gain d'un filtre passe-bande d'ordre 2.

Le filtre choisi est le plus simple qui soit, le filtre LCR série (la tension de sortie est prise aux bornes de la résistance).

Déterminer (au brouillon) la fonction de transfert de ce filtre, l'expression de la fréquence propre $f_0 = \omega_0/2\pi$, celle du gain maximal H_0, et celle du facteur de qualité Q en fonction de R, L et C. Sachant que habituellement $R \sim 1$ à $10\,\mathrm{k\Omega}$ (valeur à choisir), et que $Q \sim 5$ est un bon compromis, déterminer les valeur de L et C correspondantes.

Mesurer précisément les valeurs de R et C choisies, prendre la valeur de L indiquée puis en déduire les valeurs de f_0 et Q :

> Mesurer L demande un appareil spécifique, un RLC-mètre, même si nous en avons un au laboratoire on se contentera de la valeur indiquée en tenant compte d'une incertitude plus grande sur sa valeur, cf. plus loin.

Entrée[ ]:
```
R=
L=
C=

f0=1/np.sqrt(L*C)
H0=1
Q=
```

Réaliser le filtre, puis en l'alimentant avec une tension d'entrée sinusoïdale vérifier que :

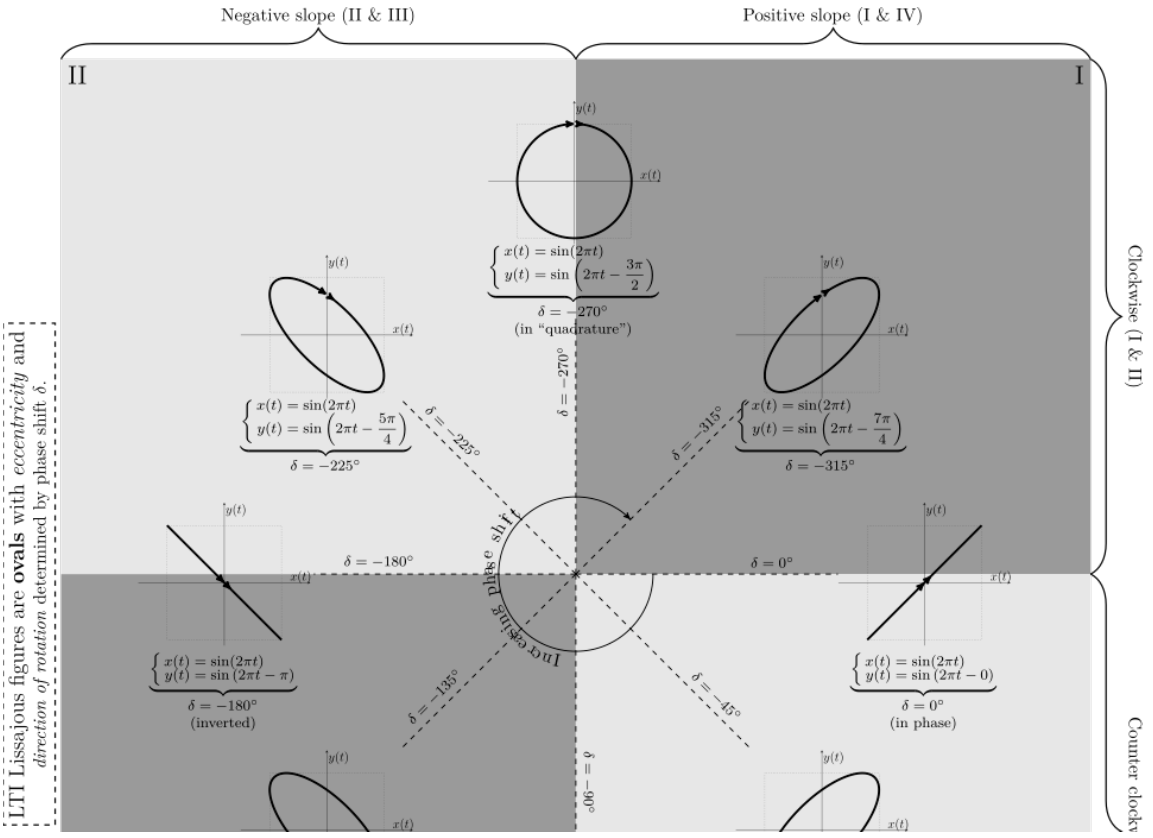
- **l'amplitude du signal de sortie passe par un maximum tel que l'amplitude du signal de sortie est environ égale à l'amplitude du signal d'entrée ($H_0 \simeq 1$) ;**
- **ce maximum est atteint à une fréquence proche de la fréquence propre f_0 :**

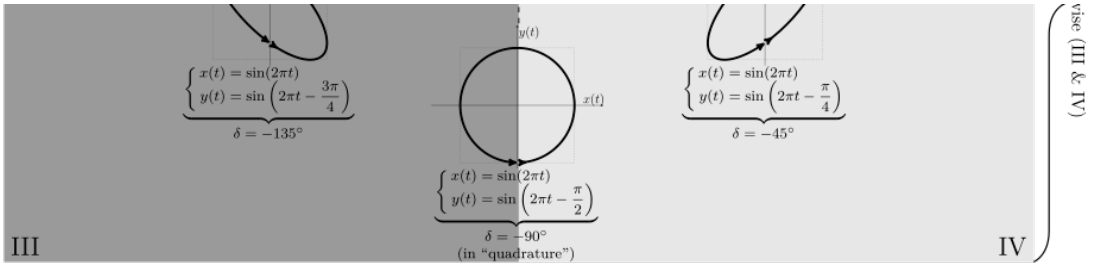
Entrée[ ]:
```
# À f0 :
E=
S=
# Comparaison avec H0=1 :
print(S/E)
```

Pour mieux repérer la fréquence propre, on peut utiliser astucieusement le fait que les signaux d'entrée et de sorie sont en phase à cette fréquence. Il existe en effet une méthode très précise de repérage de la fréquence pour laquelle des signaux sont en phase : l'utilisation du **mode Lissajous**.

Lorsque l'on représente $u_s(t)$ en fonction de $u_e(t)$ ou l'inverse, si les deux signaux sont en phase ils sont proportionnels : la représentation est une droite linéaire.

Ainsi en passant en **mode XY**, puis en faisant varier la fréquence, on peut repérer celle qui correspond à un déphasage nul entre les deux signaux.





Mesurer précisément la fréquence propre f_0 du filtre :

Entrée[ ]:
```
# Mesure
f0m=
# Modèle
print(f0)
```

On va faire le calcul de l'écart normalisé, l'incertitude sur la fréquence propre $\omega_0 = \frac{1}{\sqrt{LC}}$ est donnée selon les formules d'incertitudes par :

$$u(f_0) = f_0 \frac{1}{2} \sqrt{\left(\frac{u(L)}{L}\right)^2 + \left(\frac{u(C)}{C}\right)^2}$$

> Cf. les règles de calcul vue en première année, pour rappel elles sont indiquées dans ce notebook (../incertitudes/incertitudes_de_mesure.ipynb).

Comme C a été mesurée à l'aide d'un multimètre, c'est l'incertitude du multimètre qui prime :

5.6. Capacités

☞ ***Nota Décharger les capacités avant toute mesure***

Gammes	Précision	Courant de mesure	Temps de mesure max.	Protection *	Résolution
50 nF**	1% L + 2 UR	100 nA	0,5 s	600 VRMS	10 pF
500 nF		1 µA			100 pF
5 µF		10 µA			1 nF
50 µF		100 µA			10 nF
500 µF		1 mA	1,5 s		100 nF
5000 µF	2% L + 2 UR		3 s/mF		1 µF
50 mF					10 µF

* protection contre les surcharges, réarmable automatiquement
** l'utilisation de fils très courts et blindés est vivement recommandée pour les mesures effectuées dans cette gamme.

Nombre de points : 5 000
Sélection des gammes : automatique ou manuelle
Tension maximale en circuit ouvert : 7 V

> UR désigne une unité de représentation, donc le nombre d'unité correspondant au dernier chiffre significatif : Pour $1,064\ \mu F$, $2\ UR = 0,002\ \mu F$.

L'incertitude type $u(L)$ sur L sera prise classiquement à 5% de la valeur de L.

Déterminer les incertitudes types sur C et L et en déduire celle sur f_0

Entrée[ ]:
```
uC=
uL=
```

Pour l'incertitude sur la valeur mesurée de f_0, `f0m` , on devra passer par l'évaluation de **l'incertitude de pointé**, c'est à dire que l'on assimile $u(f_{0,m})$ au plus petit intervalle de fréquence tel que lors de la mesure, on peut visuellement considérer les signaux quasiment en phase.

Mesurer $u(f_{0,m})$.

Entrée[ ]:
```
uf0m=
```

En déduire l'écart normalisé entre f_0 (théorique) et $f_{0,m}$ (mesurée). Conclure.

Entrée[ ]:

Tracer le diagramme de Bode (courbe de gain uniquement) du filtre.

> Basez-vous sur le travail de l'activité précédente. Si vous avez le temps, tracez aussi la courbe de phase !

Prise des mesures :

Entrée[ ]:

Tracé :

Entrée[ ]:

Par lecture graphique, déterminer la bande passante de ce filtre, puis en déduire l'expression du facteur de qualité à l'aide de la formule :

$$\Delta\omega = \frac{\omega_0}{Q}$$

Entrée[ ]:
```
# Mesure
Df=
Dw=2*np.pi*Df
# Calcul
Qm=
# Comparaison
print(Q,Qm)
```

On ne calculera pas l'écart normalisé.

> C'est bon là, ça fait déjà 2 fois...

Filtrage d'un signal triangulaire

Choisir un signal triangulaire et l'afficher à l'oscilloscope.

Tracer le spectre de ce signal.

> La solution la plus simple trouvée avec cet oscilloscope est de passer par le menu `FFT`, choisir l'échelle verticale `Vrms`, appuyer sur `Autoset` puis régler les échelles verticale et horizontale pour faire apparaître le spectre.

Utiliser l'un des filtres étudié précédemment pour atténuer fortement les harmoniques du signal triangulaire, de façon à le rendre quasi-sinusoïdal en sortie.

TP analyse spectrale

PE. LEROY

> Capacités exigibles du programme :
>
> **Montages utilisant un ALI**
>
> - Identifier les limitations suivantes : saturation en tension, saturation en courant, vitesse de balayage, bande passante.
> - Mettre en oeuvre divers montages utilisant un ALI.

Liste du matériel :

- Plaquette de montage de composants
- Résistances de $10\ \mathrm{k\Omega}$ ($\times 4$)
- Condensateurs de 100 nF ($\times 2$)
- Boîte de résistance variable
- ALI
- Alimentation $+15/-15$ V
- GBF
- Oscilloscope

On commence par importer les bibliothèques nécessaires :

> La ligne suivante `%matplotlib notebook` est une commande magique qui sert à l'affichage des graphes dans le notebook.

Entrée[ ]:
```
%matplotlib notebook
```

Entrée[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Spectre d'un signal

Déterminer (au brouillon) l'expression de la décomposition en série de Fourier d'un signal créneau impair, symétrique par rapport à l'axe des abscisses.

Faire une représentation graphique de ce signal $e(t)$, en le représentant impair. Comme il est impair, il s'écrit comme une somme de fonctions $\sin()$, si bien que les coefficients a_n sont nuls. Comme il est symétrique par rapport à l'axe des abscisses, il est de valeur moyenne nulle donc $a_0 = 0$. Il est donc de la forme :

$$e(t) = \sum_{n=1}^{\infty} b_n \ \sin(n\omega t)$$

Il reste ainsi à calculer :

$$b_n = \frac{2}{T} \int_0^T e(t) \sin(n\omega t) \mathrm{d}t$$

en décomposant l'intégrale, puis à simplifier l'expression finale avec $\omega = 2\pi/T$.

Visualiser, à l'oscilloscope, le spectre d'un signal créneau.

La solution la plus simple trouvée avec cet oscilloscope est de passer par le menu `FFT` , choisir l'échelle verticale `Vrms` , appuyer sur `Autoset` puis régler les échelles verticale et horizontale pour faire apparaître le spectre.

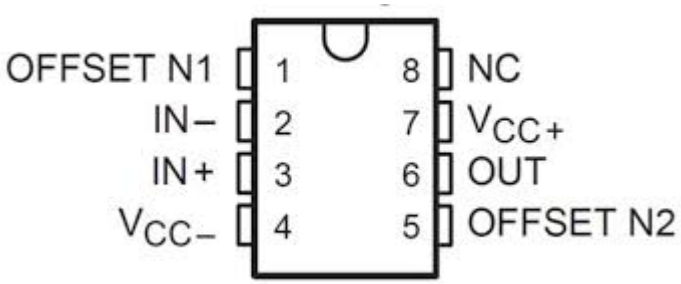
Comparer les résultats expérimentaux aux résultats théoriques. :

Les mesures se feront au curseur.

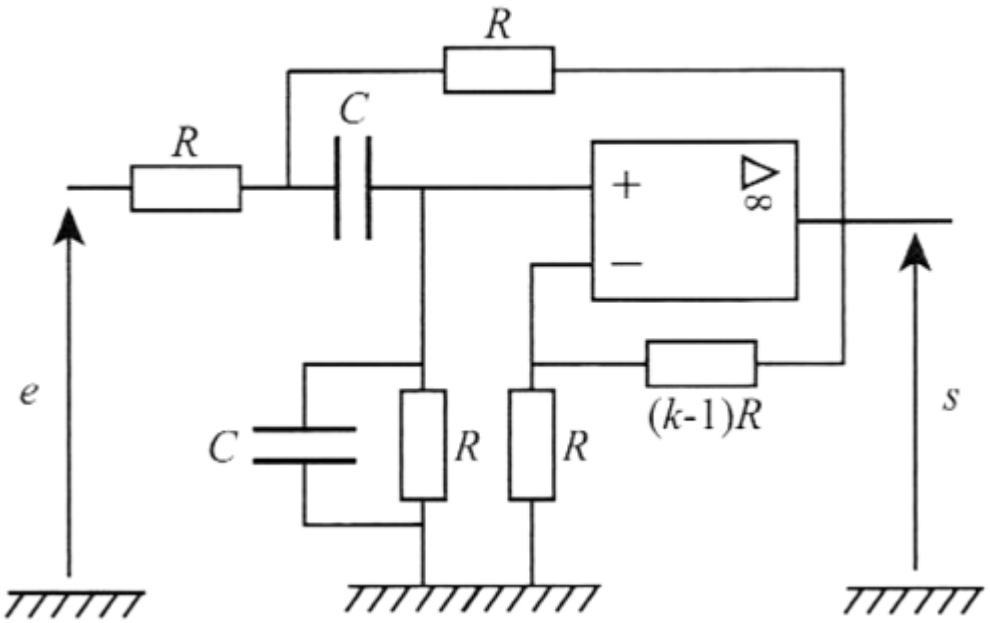
Entrée[ ]:
```
# Mesures
b1=
b2=
b3=
# Valeurs théoriques
E=
print()
print()
print()
```

Élaboration d'un filtre passe-bande très sélectif

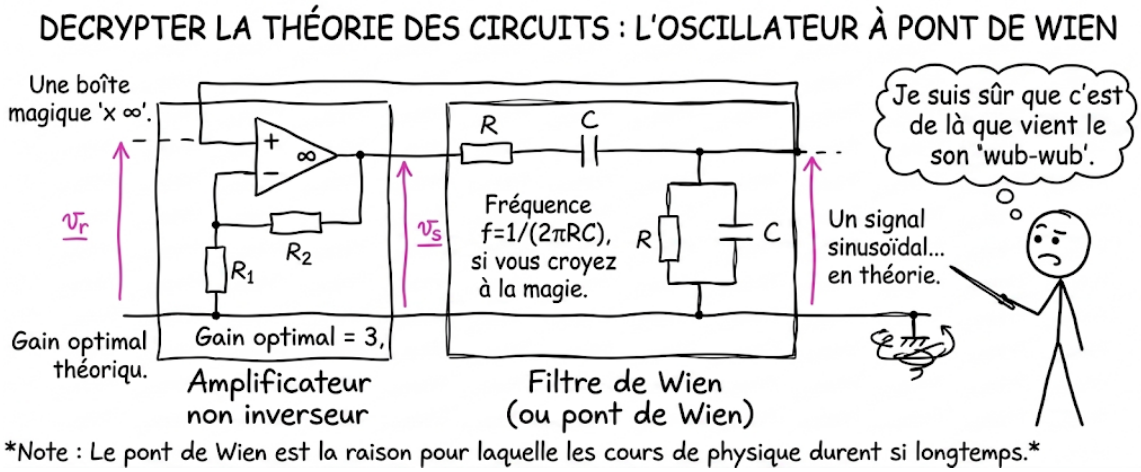
On décide de réaliser un filtre passe-bande très sélectif à l'aide d'un ALI. Pour rappel le branchement d'un ALI s'effectue selon le schéma suivant :



Le montage est alors donné par :



Quelques commentaires, à un branchement près vous pourriez remarquer que c'est **presque** le montage de l'oscillateur à pont de Wien vu en cours :



Bon mais le montage étudié ici est celui d'un filtre passe-bande. Malgré tout, cette ressemblance est intéressante à signaler, car le signal en sortie $s(t)$ aura tendance à osciller même si $e(t) = 0$, ce qui n'est pas voulu ici. C'est le réglage de la valeur de k qui va permettre de ne pas être dans le domaine des oscillations.

Et même ! Il faudra régler la valeur de k de façon à ne pas faire naître d'oscillations, mais **en se plaçant à la limite de leur apparaition**. Plus on est à la limite, plus le filtre est sélectif.

> Pour aller plus loin, ce filtre fait partie de la famille des filtres de Sallen-Key.

Réaliser le montage et régler à vue la valeur de k de façon à être à la limite de l'apparition des oscillations.

Ce montage étant censé être un filtre basse-bande très sélectif, nous allons traver son diagramme de Bode en gain.

> Moi j'aimerais trop que vous déterminiez la fonction de transfert de ce montage, c'est un excellent entrainement, mais nous sommes en TP et les aspects théoriques doivent être succincts, bref, si vois avez le temps, déterminez-là, c'est bon pour ce que vous avez !

Réaliser la prise de mesures (remplacer les valeurs actuelles).

> Le choix des gammes 1-2-5 et 8 c'est-à-dire 10-20-50 et 80 Hz puis 100-200-500 et 800 Hz et ainsi de suite, permet d'obtenir des points régulièrement répartis sur le graphe.

Entrée[ ]:
```
# Fréquences en Hertz
frequences = [10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000]

# Amplitude CC du signal d'entrée en Volts
E_cc = [5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0]

# Amplitude CC du signal de sortie en Volts
S_cc = [4.95, 4.8, 4.5, 3.5, 2.5, 0.5, 0.25, 0.05, 0.025]
```

Et le tracé :

Entrée[ ]:
```
# Conversion en tableaux numpy pour faciliter les calculs
f = np.array(frequences)
Ecc = np.array(E_cc)
Scc = np.array(S_cc)

# Calcul du Gain G=Scc/Ecc
G = Scc / Ecc

# Calcul du Gain en décibels (dB) : GdB = 20 * log10(Scc/Ecc)
GdB = 20 * np.log10(G)

# Tracé du diagramme de Bode (en gain)
plt.figure()

# Utilisation de semilogx pour l'échelle logarithmique en abscisse (
plt.semilogx(f, GdB, marker='o', linestyle='-', color='b', label='Mes

# Mise en forme du graphique
plt.title("Diagramme de Bode : Gain en fonction de la fréquence")
plt.xlabel("Fréquence (Hz) [Échelle Log]")
plt.ylabel("Gain (dB)")

# Activation de la grille (principale et secondaire pour bien voir l
plt.grid(True, which="both", ls="-", alpha=0.6)

# (Optionnel) Ajouter une ligne rouge à -3dB pour repérer la fréquenc
plt.axhline(y=-3, color='r', linestyle='--', label='Seuil -3dB')

plt.legend()
plt.tight_layout()

# Affichage
plt.show()
```

Vérifiez que l'on est bien en présence d'un filtre basse-bande :) Vérifier visuellement le comportement en terme de déphasage.

Mesurer précisément (méthode Lissajous) le fréquence de résonance du filtre.

Entrée[ ]:
```
f0=
```

Transformation d'un signal en signal quasi-sinusoïdal

En envoyant un signal créneau sur ce filtre, il est possible de **sélectionner** des composantes du spectre de ce signal, une à une. C'est à dire que l'on peut faire correspondre la fréquence de résonance du filtre f_0 avec la fréquence f_n d'une harmonique du spectre du signal créneau :

$$f_0 = f_n$$

Envoyer un signal $e(t)$ créneau et régler sa fréquence f de façon à sélectionner le mode fondamental du spectre de ce signal.

> Le mode fondamental est donc de fréquence $f_{n=1}$.

Quelle est la forme du signal de sortie $s(t)$? Puisque l'on sélectionne le mode fondamental on devrait avoir :

$$s(t) \simeq G(\omega_0)\, b_1 \sin(1 \times \omega t + \varphi_e + \varphi(\omega_0))$$

Comparer l'amplitude du signal de sortie avec sa valeur théorique :

Entrée[ ]:
```
# Mesures
S=
E=
# Théoriquement
Sth=
# Comparaison
print(S,Sth)
```

On devrait faire un calcul d'écart normalisé (ou z-score) mais on va passer à la suite :)

Faire de même avec la première harmonique.

> La première harmonique a donc une fréquence $f_{n=2} = 2 \times f_{n=1}$.

Entrée[ ]:
```
# Mesures
S=
E=
# Théoriquement
Sth=
# Comparaison
print(S,Sth)
```

Application à l'analyse spectrale de signaux

Faire de même avec un signal triangulaire.

> La décomposition en série de Fourier pourra être réalisée à l'aide d'une IA :)

Type *Markdown* and LaTeX: α^2

TP oscillateurs électroniques

PE. LEROY

> Capacités exigibles du programme :
>
> **Montages utilisant un ALI**
>
> - Identifier les limitations suivantes : saturation en tension, saturation en courant, vitesse de balayage, bande passante.
> - Mettre en oeuvre divers montages utilisant un ALI.
>
> **Oscillateurs**
>
> - Mettre en œuvre un ALI ou une porte logique pour réaliser un oscillateur.

Liste du matériel :

- Plaquette de montage de composants
- Résistances de $10\ \mathrm{k\Omega}$ ($\times 3$)
- Condensateurs de 10 nF ($\times 2$) et 100 nF
- Boîte de résistance variable
- ALI ($\times 2$)
- Alimentation $+15/-15$ V
- GBF
- Oscilloscope

On commence par importer les bibliothèques nécessaires :

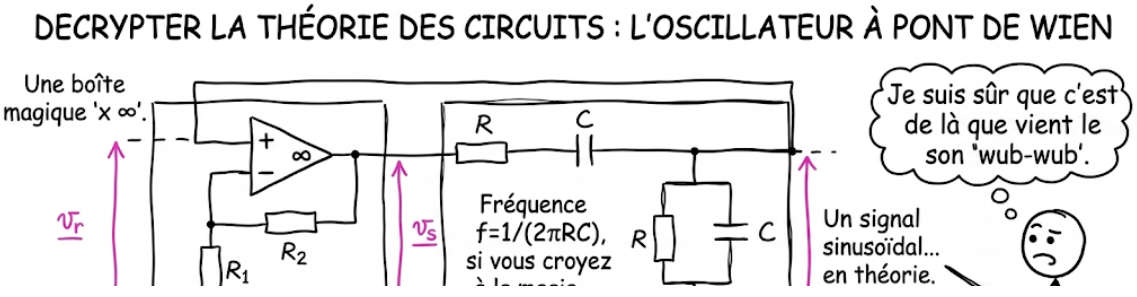
> La ligne suivante `%matplotlib notebook` est une commande magique qui sert à l'affichage des graphes dans le notebook.

Entrée[ ]:
```
%matplotlib notebook
```

Entrée[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Oscillateur quasi-sinusoïdal à pont de Wien

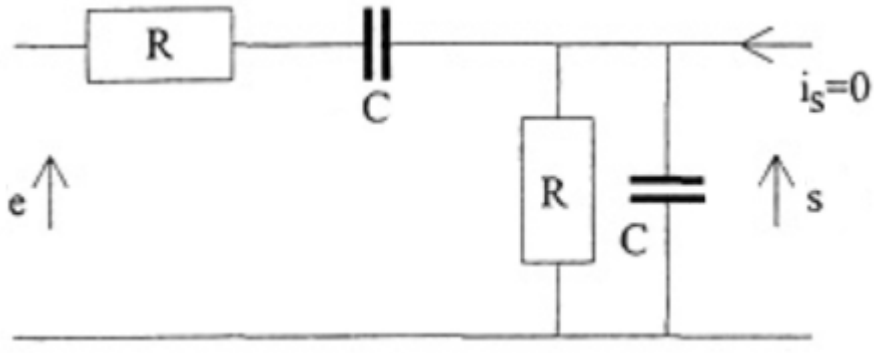
Nous allons étudier expérimentalement l'oscillateur à pont de Wien vu en cours :



Note : Le pont de Wien est la raison pour laquelle les cours de physique durent si longtemps.

L'idée est d'étudier le comportement des deux blocs, avant de les relier entre eux pour former l'oscillateur.

Étude du premier bloc, le filtre de Wien



Vous avez l'habitude maintenant, montage, tracé du diagramme de Bode en gain, détermination de la bande passante et du facteur de qualité.

Déterminer (au brouillon) la fonction de transfert de ce filtre, l'expression de la pulsation propre ω_0, celle du gain maximal H_0, et celle du facteur de qualité Q.

Réaliser le filtre, puis en l'alimentant avec une tension d'entrée sinusoïdale vérifier que :

- **l'amplitude du signal de sortie passe par un maximum tel que l'amplitude du signal de sortie est environ égale au tiers de l'amplitude du signal d'entrée ($H_0 \simeq 1/3$) ;**
- **ce maximum est atteint à une fréquence proche de la fréquence propre f_0 :**

Entrée[ ]:
```
# À f0 :
E=
S=
# Comparaison avec H0=1/3 :
print(S/E)
```

Réaliser la prise de mesures permettant le tracé de la courbe de gain du diagramme de Bode (remplacer les valeurs actuelles).

> Le choix des gammes 1-2-5 et 8 c'est-à-dire 10-20-50 et 80 Hz puis 100-200-500 et 800 Hz et ainsi de suite, permet d'obtenir des points régulièrement répartis sur le graphe.

Entrée[ ]:
```
# Fréquences en Hertz
frequences = [10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000]

# Décalage temporel du signal de sortie par rapport au signal d'entre
# Attention au signe, à vérifier graphiquement !
Delta_t = [5.0e-3, 5.0e-3, 5.0e-3, 5.0e-3, 5.0e-3, 5.0e-3, 5.0e-3, 5.
```

Tracer le diagramme de Bode (courbe de gain) du filtre ainsi élaboré.

Entrée[ ]:
```
# Conversion en tableaux numpy pour faciliter les calculs
f = np.array(frequences)
Ecc = np.array(E_cc)
Scc = np.array(S_cc)

# Calcul du Gain G=Scc/Ecc
G = Scc / Ecc

# Calcul du Gain en décibels (dB) : GdB = 20 * log10(Scc/Ecc)
GdB = 20 * np.log10(G)

# Tracé du diagramme de Bode (en gain)
plt.figure()

# Utilisation de semilogx pour l'échelle logarithmique en abscisse (
plt.semilogx(f, GdB, marker='o', linestyle='-', color='b', label='Mes

# Mise en forme du graphique
plt.title("Diagramme de Bode : Gain en fonction de la fréquence")
plt.xlabel("Fréquence (Hz) [Échelle Log]")
plt.ylabel("Gain (dB)")

# Activation de la grille (principale et secondaire pour bien voir l'
plt.grid(True, which="both", ls="-", alpha=0.6)

# (Optionnel) Ajouter une ligne rouge à -3dB pour repérer la fréquenc
plt.axhline(y=-3, color='r', linestyle='--', label='Seuil -3dB')

plt.legend()
plt.tight_layout()

# Affichage
plt.show()
```

Par lecture graphique, déterminer la bande passante de ce filtre, puis en déduire l'expression du facteur de qualité à l'aide de la formule :

$$\Delta\omega = \frac{\omega_0}{Q}$$

Entrée[ ]:
```
# Mesure
Df=
Dw=2*np.pi*Df
# Calcul
Qm=
# Comparaison
Q=1/3
print(Q,Qm)
```

Étude du second bloc

Le second bloc est un montage amplificatuer non inverseur à ALI.

Sur un brouillon, déterminer la fonction de transfert de ce montage.

Réaliser le montage de façon à ce que le gain soit réglable autour de la valeur 3.

Nous allons en profiter pour mettre en évidence quelques caractéritiques de l'ALI, en le poussant dans ses limites grâce à ce montage.

Evoyer un signal sinusoïdal en entrée, puis augmenter la fréquence jusqu'à ce que le rapport des amplitudes d'entrée et de sortie soit de $S/E = 1/\sqrt{2}$, noter cette fréquence f_c.

> Démarrer à une fréquence classique en TP de $f \sim 1$ kHz.

L'intervalle $[0, f_c]$ est la **bande passante du montage**, due au comportement passe-bas de l'ALI :

$$\frac{\underline{v_s}}{\underline{v_+} - \underline{v_-}} = \underline{\mu} = \frac{\mu_0}{1 + j\omega\tau}$$

Le **temps de réponse du montage** est alors donné par :

$$\tau = \frac{1}{\omega_c} = \frac{1}{2\pi f_c}$$

Entrée[ ]:
```
fc=
tau=
```

Vérifier que la fréquence de résonance du premier montage (filtre) est bien comprise dans la bande passante de ce montage.

Envoyer un signal sinusoïdal en entrée, puis augmenter la fréquence jusqu'à ce que le signal de sortie prenne une forme triangulaire.

On parle de triangularisation du signal, c'est dû à **la limitation de la vitesse de balayage** de l'ALI (<< slew rate >>) :

$$\left|\frac{\mathrm{d}v_s}{\mathrm{d}t}\right| < 10^6 \text{ V. s}^{-1}$$

Dès que e signal se triangularise on peut calculer la valeur de la vitesse limite de balayage, elle correspond à la pente (coefficient directeur) de chaque portion de droite.

Déterminer la valeur de la vitesse limite de balayage de l'ALI.

Entrée[ ]:
```
vlb=
```

Envoyer un signal sinusoïdal en entrée, puis augmenter son amplitude jusqu'à ce que le signal de sortie sature en tension.

Il s'agit de la **saturation de la tension de sortie**, $s(t)$ est tout simplement limité à $\pm V_{sat}$.

Mesurer la valeur de V_{sat}.

Entrée[ ]:
```
Vsat=
```

Brancher une résistance variable en sortie. Envoyer un signal sinusoïdal en entrée, puis augmenter la valeur de la résistance jusqu'à ce que le signal de sortie sature.

Il s'agit ici de la **limitation du courant de sortie** :

$$|i_{s,max}| \sim 20 \text{ mA}$$

Déterminer la valeur du courant limite de sortie de l'ALI :

Entrée[ ]:
```
R=
Usat=
ismax=Usat/R
print(ismax)
```

Mise en oeuvre de l'oscillateur

Réaliser le montage complet de l'oscillateur à pont de Wien.

> On prendra $R_1 = 10 \text{ k}\Omega$ et R_2 très légèrement inférieure à $2R_1$.

En augmentant doucement R_2 vous devriez assister à la naissance des oscillations quasi-sinusoïdales.

N'hésitez pas à régler le balayage pour voir la naissance des oscillations quasi-sinusoïdale avec une augmentation exponentielle de l'amplitude.

Modèle numérique

On peut essayer de faire un modèle numérique du comportement de cet oscillateur, le modèle analytique donne l'équation suivante vue en cours :

$$\frac{\mathrm{d}^2 v_s}{\mathrm{d}t^2} + \frac{\omega_0}{Q}\left(1 - H_0\left(1 + \frac{R_2}{R_1}\right)\right)\frac{\mathrm{d}v_s}{\mathrm{d}t} + \omega_0^2\, v_s = 0$$

avec $H_0 = 1/3$, $Q = 1/3$ et $\omega_0=\dfrac{1}{RC}$

Utilisons la méthode d'Euler pour la résolution. Cette équation différentielle est d'ordre 2, on se propose classiquement de la décomposer en 2 équations d'ordre 1 en introduisant une variable intermédiaire V :

$$\frac{\mathrm{d}v_s}{\mathrm{d}t} = V$$

L'équation différentielle s'écrit alors :

$$\frac{\mathrm{d}V}{\mathrm{d}t} = -\left(\frac{\omega_0}{Q}\left(1 - H_0\left(1 + \frac{R_2}{R_1}\right)\right)V + \omega_0^2\, v_s\right)$$

Écrites sous cette forme, ces deux équations permettent d'exprimer la variation de l'une des 2 grandeurs V ou v_s en fonction des valeurs intantanées des 2 autres, d'où l'on peut tirer le << schéma d'Euler >> de résolution :

$$\frac{v_{s,n+1} - v_{s,n}}{dt} = V_n$$

$$\frac{V_{n+1} - V_n}{dt} = -\left(\frac{\omega_0}{Q}\left(1 - H_0\left(1 + \frac{R_2}{R_1}\right)\right)V_n + \omega_0^2\, v_{s,n}\right)$$

ou :

$$\boxed{v_{s,n+1} = v_{s,n} + dt \cdot V_n}$$

$$\boxed{V_{n+1} = V_n - dt \cdot \left(\frac{\omega_0}{Q}\left(1 - H_0\left(1 + \frac{R_2}{R_1}\right)\right)V_n + \omega_0^2\, v_{s,n}\right)}$$

Il suffit alors de créer une boucle pour calculer les valeurs suivantes (en $n+1$) à partir des valeurs précédentes (en n).

- Si $R_2 < 2R_1$, les oscillations qui naissent du bruit électronique s'amortissent rapidement. Nous allons tester ces conditions initiales :

À $t = 0$, on prendra pour simuler une perturbation électronique due à du bruit $vs(0) = 1$ mV et $V(0) = 0$:

```
vs=1e-3

V=0
```

Compléter le programme suivant pour réaliser la simulation numérique du

comportement de l'oscillateur dans ce cas.

Entrée[ ]:
```
# Définition des constantes
R=10e3 # Ohms
C=10e-9 # nF

R1=10e3 # Ohms
R2=0.9*2*R1

H0=1/3
w0=1/(R*C)
Q=1/3

vs0=0.001 # Valeur initiale de vs (bruit) en V
V0=0 # Valeur initiale de dvs/dt (équations de continuité) en V/s

# Paramètres de la résolution numérique
N=2000 # Nombre de points de la résolution numérique
duree=R*C*2*np.pi*10 # Durée totale (10 périodes)
dt=duree/N # Pas de la résolution numérique

# Définition des listes et de la valeur initiale de la dérivée dv/dt
vs=[vs0]
V=[V0]
t=[0]

# Résolution par la méthode d'Euler (simple discrétisation)
for n in range(N-1):
    vs.append(vs[n]+dt*V[n])

    # À compléter :
    # V.append(...)

    t.append(t[n]+dt)

# Représentation graphique
plt.figure()
plt.plot(t,vs,'r-')
plt.title('Représentation de $v_s(t)$')
plt.xlabel('t (s)')
plt.ylabel('$v_s$ (V)')
plt.legend(['$v_s(t)$'])
#plt.savefig('signal_oscillateur.png')
plt.show()
```

- Si $R_2 > 2R_1$, les oscillations qui naissent du bruit électronique s'amplifient rapidement, et de façon exponentielle. C'est là que l'ALI intervient, puisque sa tension de sortie est limitée à $\pm V_{sat}$.

Pour simuler son comportement il faut donc :

- si `vs[n]` devient soudainement supérieur à `Vsat`, remplacer `vs[n]` par `Vsat` ;
- si `vs[n]` devient soudainement inférieur à `-Vsat`, remplacer `vs[n]` par `-Vsat`.

Compléter le programme suivant pour effectuer cette simulation.

Entrée[ ]:
```
import matplotlib.pyplot as plt
import numpy as np

# Définition des constantes
R=10e3 # Ohms
C=10e-9 # nF

R1=10e3 # Ohms
R2=1.1*2*R1

H0=1/3
w0=1/(R*C)
Q=1/3

Vsat=13.6 # V

vs0=0.001 # Valeur initiale de vs (bruit) en V
V0=0 # Valeur initiale de dvs/dt (équations de continuité) en V/s

# Paramètres de la résolution numérique
N=2000 # Nombre de points de la résolution numérique
duree=R*C*2*np.pi*20 # Durée totale (20 périodes)
dt=duree/N # Pas de la résolution numérique

# Définition des listes et de la valeur initiale de la dérivée dv/dt
vs=[vs0]
V=[V0]
t=[0]

# Résolution par la méthode d'Euler (simple discrétisation)
for n in range(N-1):
    # À compléter :
    if vs[n]>Vsat:
        # vsn=...
    elif vs[n]<-Vsat:
        # vsn=...
    else:
        vsn=vs[n]

    vs.append(vsn+dt*V[n])
    V.append(V[n]-dt*(w0/Q*(1-H0*(1+R2/R1))*V[n]+w0**2*vsn))

    t.append(t[n]+dt)

# Représentation graphique
plt.figure()
plt.plot(t,vs,'r-')
plt.title('Représentation de $v_s(t)$')
plt.xlabel('t (s)')
plt.ylabel('$v_s$ (V)')
plt.legend(['$v_s(t)$'])
#plt.savefig('signal_oscillateur.png')
plt.show()
```

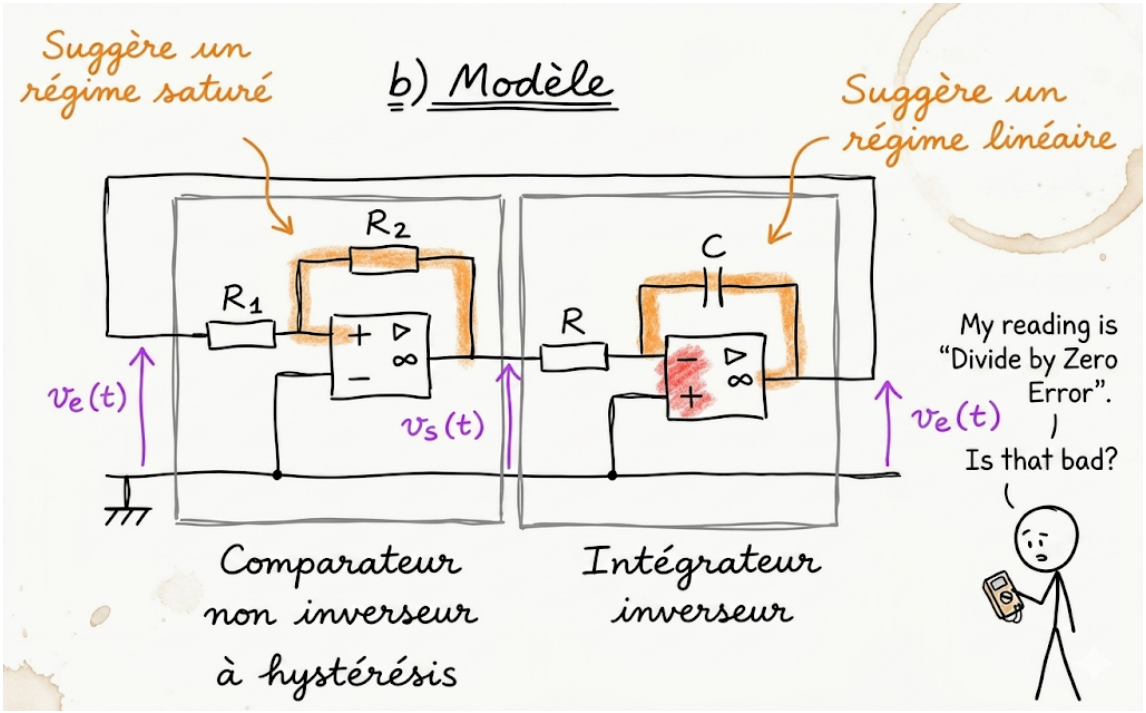
Sur votre montage, augmentez R_2 et comparer le résultat expérimental avec le comportement de la simulation numérique dans les mêmes conditions.

Dans le programme Python n'hésitez pas à augmenter le nombre de points `N` de la résolution par la méthode d'Euler si les résultats divergent. Cette méthode est celle de base, donc la moins stable des méthode de résolution d'équations différentielles.

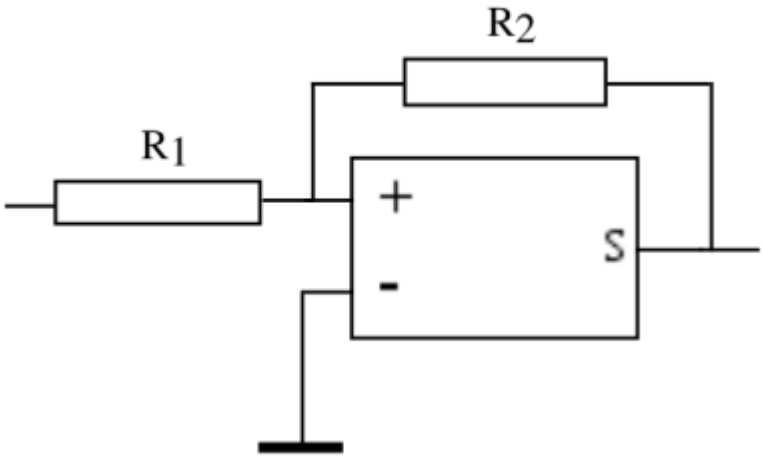
Oscillateur de relaxation

Étude du premier bloc

L'étude de ce montage a été réalisée en cours :



Ce montage comporte 2 blocs, l'un qui est un comparateur à hystérésis, l'autre un intégrateur. On commence par la réalisation du premier bloc :



Au brouillon, réaliser l'étude du comportement de ce comparateur. Déterminer notamment les tensions $v_{e,min}$ et $v_{e,max}$ au delà desquelles v_s bascule à $\pm V_{sat}$.

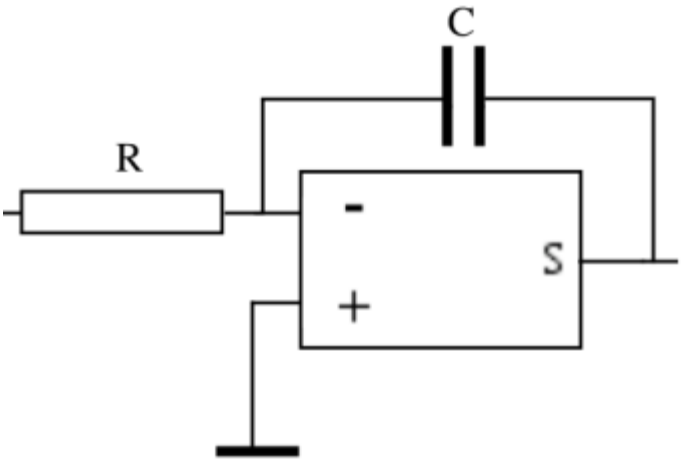
Entrée[ ]:
```
R1=10e3 # Ohm
R2=3*R1
vemax=
vemin=vemax
print(vemin,vemax)
```

Réaliser le montage et afficher le diagramme de fonctionnement $v_s = f(v_e)$ à l'oscilloscope. Comparer les valeurs expérimentales aux valeurs théoriques

> On prendra $R2 \simeq 3R_1$.

Étude du second bloc

Le second bloc est un montage intégrateur inverseur :



On a montré en cours que :

$$v_s(t) = -\frac{1}{RC}\int v_e(t)\mathrm{d}t$$

Réaliser le montage et envoyer successivement en entrée un signal sinusoïdal, créneau et triangulaire. Comparer le signal de sortie avec ce que peut prévoir le modèle.

> On prendra $R = 10\ \mathrm{k\Omega}$ et $C = 100$ nF.

Mise en oeuvre de l'oscillateur

On a montré en cours que l'oscillateur devrait osciller avec la période :

$$T = 4RC\frac{R_1}{R_2}$$

Réaliser le montage complet de l'oscillateur de relaxation, mesurer sa période d'oscillation avec des curseurs et la comparer à celle du modèle :

Entrée[ ]:
```
R1=10e3 # Ohm
R2=3*R1
R=10e3 # Ohm
C=10e-9 # nF

T=4*R*C*R1/R2
Texp=
print(T,Texp)
```

Pour valider la valeur du modèle il faut réaliser un calcul d'écart normalisé.
L'incertitude sur T est donnée par :

$$u(T) = 4\sqrt{\left(\frac{u(R)}{R}\right)^2 + \left(\frac{u(C)}{C}\right)^2 + \left(\frac{u(R_1)}{R_1}\right)^2 + \left(\frac{u(R_2)}{R_2}\right)^2}$$

Calculer l'incertitude type sur T :

> On prendra typiquement 5% d'incertitude sur les valeurs des composants.

Entrée[ ]:
```
uT=
print(uT)
```

L'incertitude sur T_{exp} est donnée par une incertitude de pointé puisque l'on a utilisé des curseurs à l'oscilloscope.

Déterminez l'intervalle de temps dans lequel T_{exp} peut se trouver du fait de votre << pointage >>, et donc l'écart de temps où la mesure peut vous sembler correcte :

Entrée[ ]:
```
uTexp=
```

Calculer l'écart normalisé et valider votre valeur expérimentale :

Entrée[ ]:
```
EN=
print(EN)
```

Type *Markdown* and LaTeX: α^2

TP émetteur radio

PE. LEROY

> Capacités exigibles du programme :
>
> **Modulation et démodulation**
>
> - Élaborer un signal modulé en amplitude à l'aide d'un circuit multiplieur.
> - Réaliser une démodulation synchrone.

Liste du matériel :

- Module multiplieur ($\times 2$)
- Alimentation $+15/-15$ V ($\times 2$)
- GBF ($\times 2$)
- Oscilloscope
- Boîtes de résistance/capacité variables
- RLC-mètre
- Diode (silicium)
- Condensateur de 1 nF
- ALI
- Résistances de 1 et 100 $\mathrm{k\Omega}$
- Bobine à induction (double, avec noyau)

On commence par importer les bibliothèques nécessaires :

> La ligne suivante `%matplotlib notebook` est une commande magique qui sert à l'affichage des graphes dans le notebook.

Entrée[9]:
```
%matplotlib notebook
```

Entrée[3]:
```
import numpy as np
import matplotlib.pyplot as plt
```

L'objectif est de travailler sur la modulation et la démodulation d'amplitude, de façon à fabriquer un émetteur radio.

Modulation d'amplitude

Quelques rappels, la modulation << en amplitude >> consiste donc à faire varier l'amplitude du **signal porteur** (sinusoïdal, haute fréquence) dans le temps :

$$s(t) = a(t)\cos(2\pi f_p t + \varphi_p) \text{ avec } a(t) = A \cdot s_m(t) + B$$

Dans la pratique, on va multiplier un signal porteur de la forme $u_p(t) = U_p \cos(2\pi f_p t + \varphi)$ par un signal constitué :

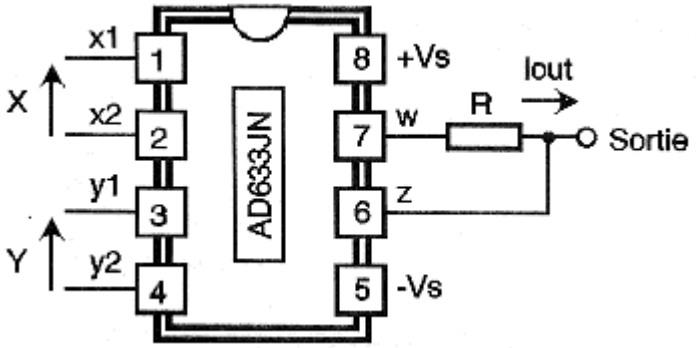
- du **signal modulant** (basse fréquence) de la forme $u_m(t)$;

- d'un signal constant U_0 (<< offset >> sur un GBF).

On obtient alors :

$$\boxed{u(t) = (u_m(t) + U_0)U_p \cos(2\pi f_p t + \varphi_p)}$$

La multiplication s'effectue en pratique à l'aide d'un composant appelé multiplieur :



Ce composant (mis en ouvre dans un boitier) donne comme signal de sortie :

$$s = \frac{(X_2 - X_1) \times (Y_2 - Y_1)}{10} + Z$$

Dans notre cas, **il faudra mettre à la masse** les entrées `X1` , `Y1` et `Z` de facon à obtenir :

$$\boxed{s = \frac{1}{10}(X_2 \times Y_2) = K \cdot (X_2 \times Y_2)}$$

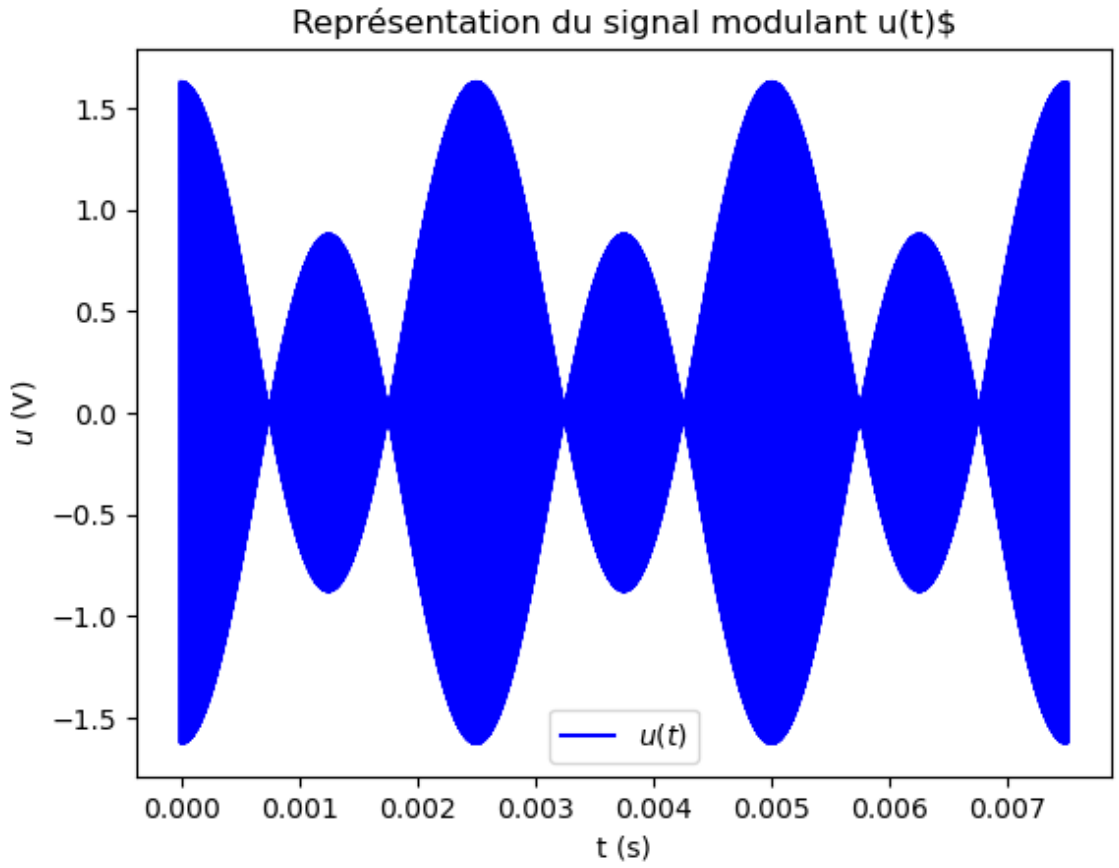
Prenez votre cours ou la carte mentale correspondante pour la suite.

Réaliser un montage permettant de moduler un signal porteur de fréquence $f_p = 240$ kHz et d'amplitude $U_p = 2,5$ V par un signal modulant sinusoïdal de fréquence $f_m = 440$ kHz et d'amplitude $U_m = 5$ V, avec un offset $U_0 = 1,5$ V.

> Attention pour rappel les GBF affichent en général une amplitude crête à crête (ou << pic to pic >>) qui est le double de l'amplitude du signal.

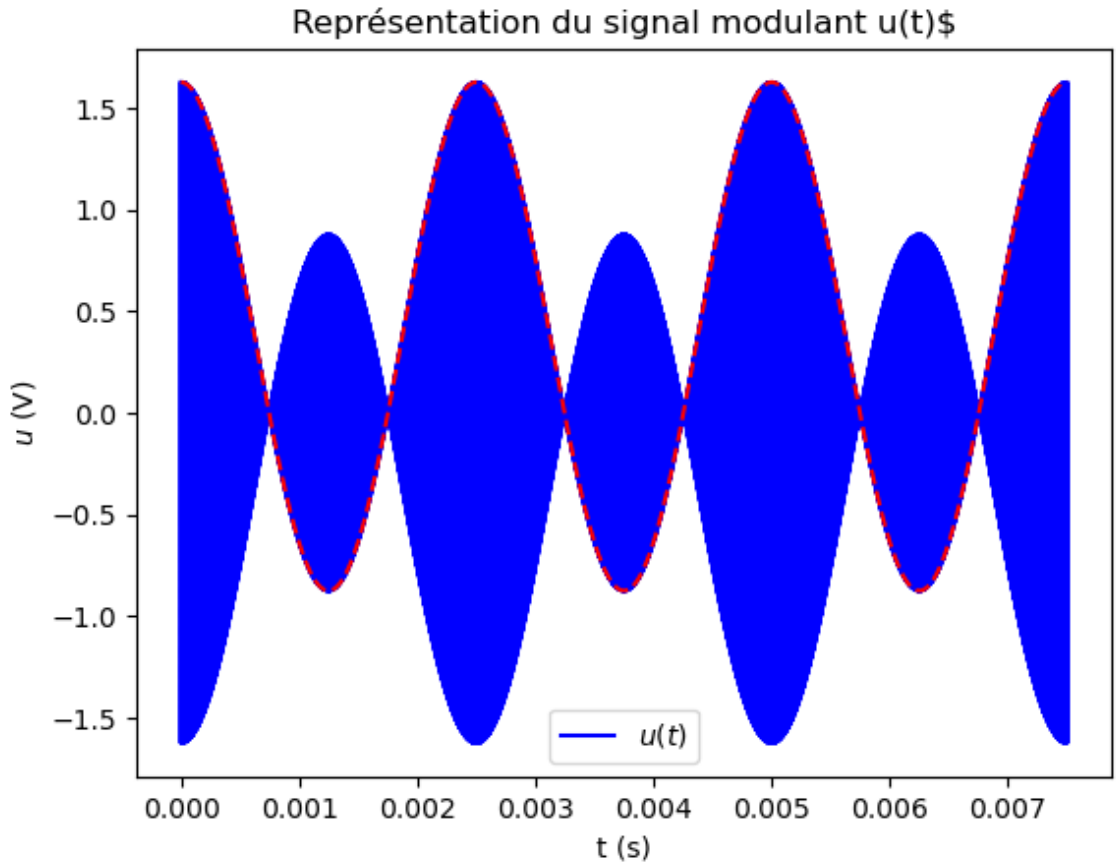
Pour vérifier que vous obtenez les bonnes courbes, on peut modéliser celles-ci :

Entrée[24]:
```
fp=240e3 # Hz
fm=400 # Hz
Up=2.5 # V
Um=5 # V
U0=1.5 # V
phip=0
phim=0

K=1/10

dt=1/fp/200 # Pas temporel
N=int(3/fm/dt) # Affichage de 3 périodes
t=np.array([k*dt for k in range(N)])

u=(Um*np.cos(2*np.pi*fm*t+phim)+U0)*K*Up*np.cos(2*np.pi*fp*t+phip)

plt.figure()
plt.plot(t,u,'b-')
plt.title('Représentation du signal modulant u(t)$')
plt.xlabel('t (s)')
plt.ylabel('$u$ (V)')
plt.legend(['$u(t)$'])
plt.show()
```



Vous pouvez zoomer sur le graphe pour voir le détail du signal. On peut surperposer le signal qui correspond à l'amplitude $a(t)$ dépendant du temps pour voir qu'il suit la forme de l'enveloppe :

Entrée[25]:
```
ue=K*Up*(Um*np.cos(2*np.pi*fm*t+phim)+U0)

plt.figure()
plt.plot(t,u,'b-')
plt.plot(t,ue,'r--')
plt.title('Représentation du signal modulant u(t)$')
plt.xlabel('t (s)')
plt.ylabel('$u$ (V)')
plt.legend(['$u(t)$'])
plt.show()
```



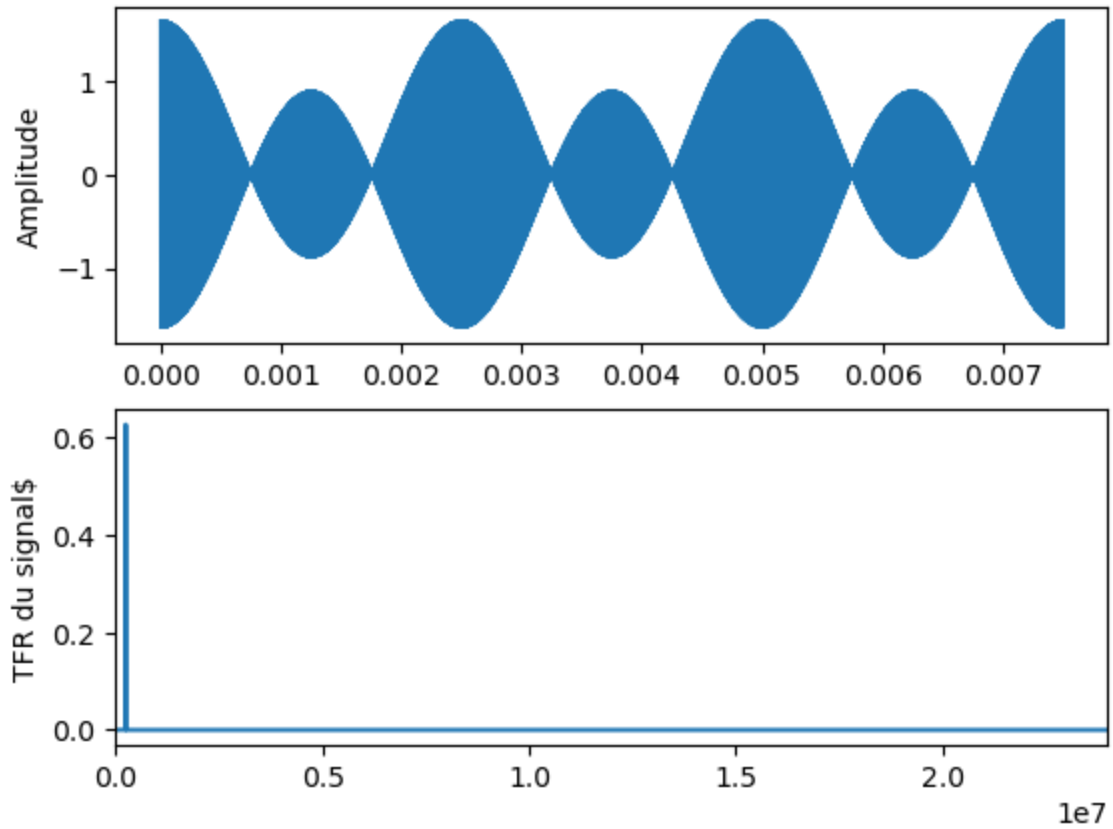
Une fois obtenu ce signal à l'oscilloscope vous pouvez passer à la suite.

Tracer le spectre de ce signal à l'oscilloscope.

> Rappelez-vous, la solution la plus simple trouvée avec cet oscilloscope est de passer par le menu `FFT`, choisir l'échelle verticale `Vrms`, appuyer sur `Autoset` puis régler les échelles verticale et horizontale pour faire apparaître le spectre.

Pour comparaison :

Entrée[33]:
```
import numpy as np
import numpy.fft as npf
import scipy.io.wavfile as spwf
import matplotlib.pyplot as plt

fe=1/dt # Fréquence d'échantillonnage
D=N*dt # Durée de l'enregistrement

liste_freq=np.linspace(0,fe,N)
liste_fft=npf.fft(u) #,norm="forward") # Application de l'algorithme

plt.figure()
plt.subplot(211)
plt.plot(t,u)
plt.ylabel('Amplitude')

plt.subplot(212)
plt.plot(liste_freq,(2/N)*np.absolute(liste_fft))
plt.xlim(0,fe/2)
plt.ylabel('TFR du signal$')
plt.show()
```



Vous pouvez zoomer sur le spectre pour mieux voir celui-ci.

Démodulation d'amplitude

On se propose de tester les 2 types de démodulations vues en cours.

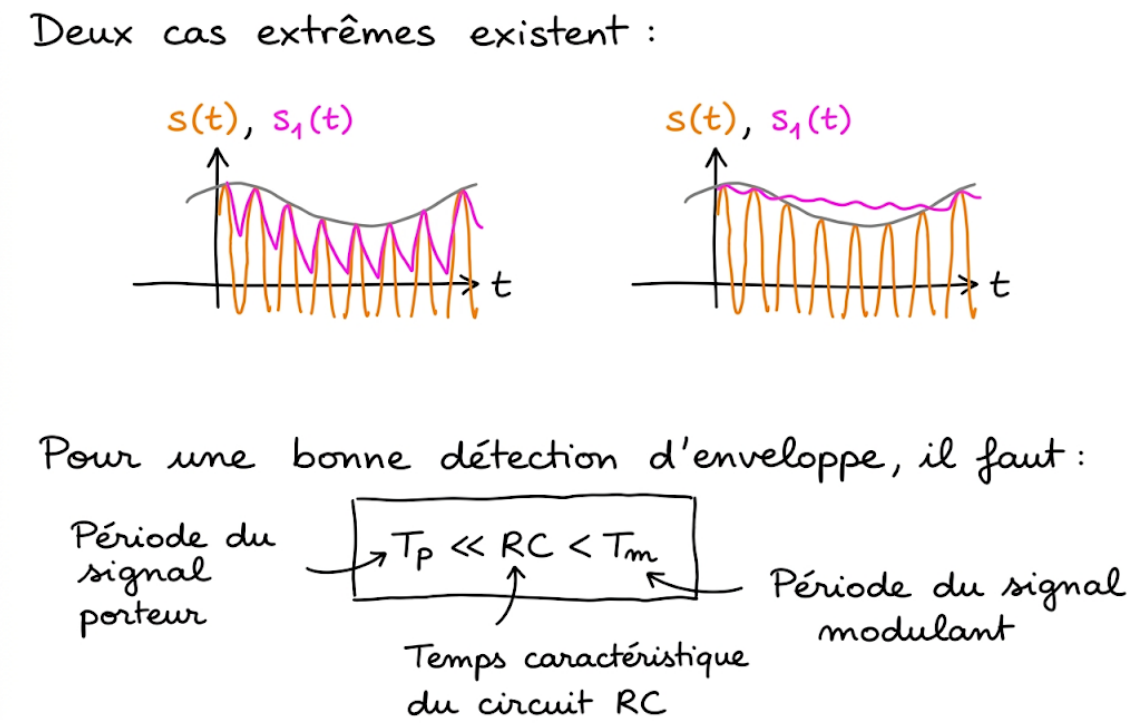
Démodulation par détection d'enveloppe

Réaliser un montage permettant de démoduler le signal modulé $u(t)$ obtenu

précédemment par détection d'enveloppe.

> On prendra R variable et $C = 1$ nF.

En faisant varier R vous devriez voir les cas vus en cours, pour rappel :



Démodulation synchrone (ou démodulation par détection synchrone)

Le principe a été décrit en cours, reprenez celui-ci. Il fait intervenir un filtre passe-bas, le plus simple est le filtre RC, tel que la fréquence de coupure est donnée par :

$$f_c = \frac{1}{2\pi RC}$$

Le filtre doit sélectionner le signal correspondant au signal modulant $u_m(t)$, donc de fréquence f_m.

Sachant que habituellement (pour des raisons d'adaptation d'impédance) $R \sim 1$ à $10\ \mathrm{k\Omega}$, déterminer la valeur de C nécessaire :

Entrée[ ]:
```
R=
C=
fc=
print(fc)
```

Réaliser un montage permettant de démoduler le signal modulé $u(t)$ obtenu précédemment par démodulation synchrone.

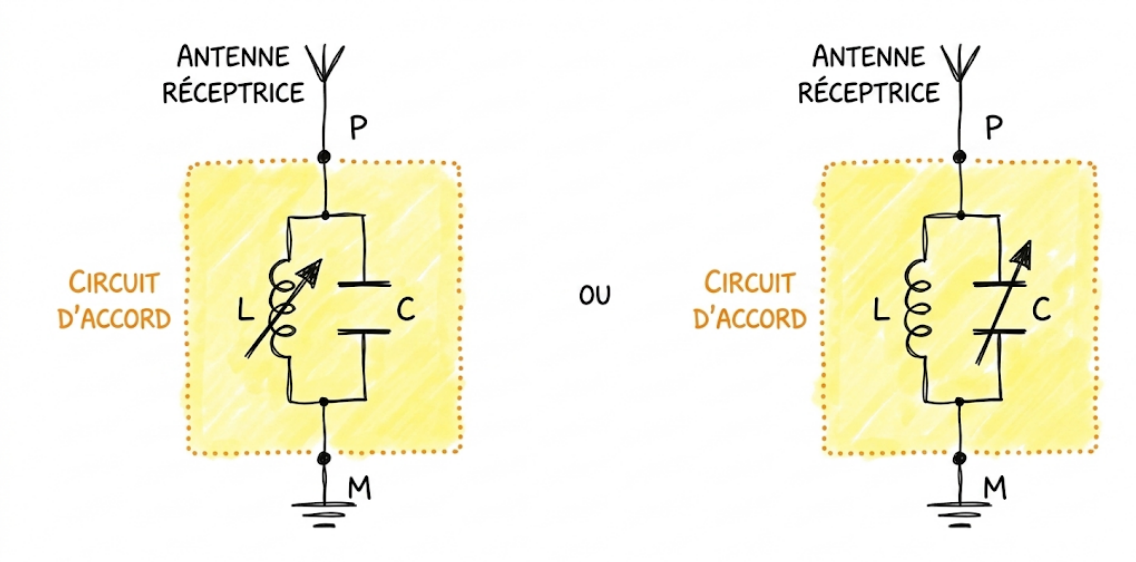
Tracer le spectre de ce signal démodulé à l'oscilloscope.

> Garder ce montage ! Il vous servira dans ce qui suit.

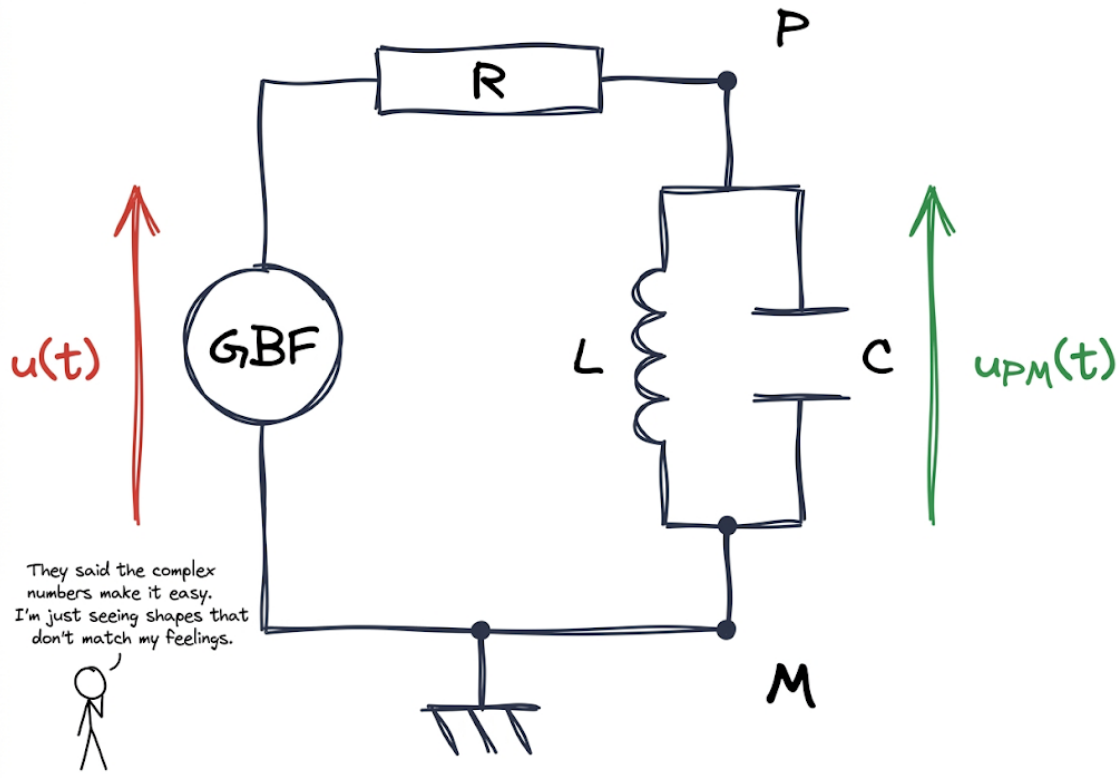
Réception de radios << grandes ondes >>

Comme indiqué en cours, une antenne est un simple fil, qui va récupérer toutes les ondes électromagnétiques présentes. Dans l'environnement, certains signaux sont beaucoup plus intenses que d'autres, notamment ceux issus des câbles électriques (signaux 50 Hz).

Comme on ne souhaite capter que les signaux de fréquences proches de celle du signal porteur f_p (pour les démoduler ensuite), on utilise un pré-filtre au niveau de l'antenne appelé << circuit bouchon >> ou << circuit d'accord >>.



Pour comprendre comment il fonctionne, déterminer (au brouillon) la fonction de transfert du circuit suivant :



Vérifiez que vous pouvez mettre son expression sous la forme canonique d'un filtre

passe-bande, tel que la fréquence de résonance est :

$$f_0 = \frac{1}{2\pi\sqrt{LC}}$$

Vous avez à votre disposition une double bobine avec noyau de fer.

Mesurer l'inductance L de votre bobine à l'aide du RLC-mètre, à la fréquence de mesure de 120 kHz.

Pour capter les signaux voulus, il faut donc que $f0 = f_p$.

Déterminer la valeur de C nécessaire pour capter une radio autour de 240 kHz.

Entrée[ ]:
```
L=
fp=240e3 # Hz
f0=fp
C=
print(C)
```

Réaliser le montage correspondant à cette antenne et son circuit d'accord.

Le signal capté par l'antenne et son circuit d'accord est de très faible amplitude. Il est donc nécessaire de l'amplifier avant de le démoduler (par démodulation synchrone).

Réaliser un montage amplificateur non inverseur à ALI pour amplifier le signal.

> On choisira R_1 et R_2 de façon à ce que le facteur d'amplification (à déterminer au brouillon) soit le plus élevé possible avec les composants à dsposition (peu de chance d'atteindre la saturation en sortie d'ALI).

Réaliser le montage complet :

- **Émetteur radio ;**

> Il s'agit du montage permettant de générer le signal modulé, branchez un fil en sortie du multiplieur qui servira d'antenne émettrice.
>
> > La longueur du fil d'une antenne (émettrice ou réceptrice) idéale est égale à $1/4$ de la longueur d'onde, soit $1/4c/f_p$, calculez-là vous verrez qu'ici c'est compliqué de s'en approcher, mais on retiendra que plus le fil est long meilleure est l'émission (même chose pour la réception).

Entrée[ ]:
```
fp=240e3 # Hz
c=3e8 # m/s
print(1/4*c/fp)
```

- **Récepteur radio constitué de l'antenne et son circuit d'accord, l'amplificateur à ALI et la démodulation synchrone.**

> À l'aide de l'oscilloscope, visualisez et vérifiez successivement et à tout moment les signaux en chaque point du circuit. Notamment vérifiez bien que vous multipliez des signaux d'amplitudes du même ordre de grandeur.

Si le signal de sortie correspond bien au signal modulant, vous pouvez brancher l'écouteur adapté en impédance qui est à votre disposition pour entendre le signal correspondant.

> Une remarque, avec cet écouteur le dernier filtre ne sert à rien : l'écouteur n'est pas sensible aux fréquences à éliminer.

Type *Markdown* and LaTeX: α^2

TP détecteur de métaux

PE. LEROY

> Capacités exigibles du programme :
>
> **Détection synchrone**
>
> - Mesurer une fréquence par une détection synchrone élémentaire à l'aide d'un multiplieur et d'un passe-bas simple adapté à la mesure.

Liste du matériel :

- Plaquette de montage de composants (LAB)
- Résistance de $10\ \mathrm{k\Omega}$ ($\times 2$)
- Boîte de résistance variable
- Boîte de capacité variable
- Bobine de transformateur
- DEL (verte ou jaune)
- Module multiplieur
- ALI
- Alimentation $+15/-15$ V
- GBF
- Oscilloscope
- Multimètre Metrix rouge
- plaques de fer, cuivre et zinc, approximativement de mêmes dimensions
- banc d'optique, avec 2 supports, l'un pour la bobine, l'autre pour une plaque (donc prévoir des supports adaptées)

On commence par importer les bibliothèques nécessaires :

> La ligne suivante `%matplotlib notebook` est une commande magique qui sert à l'affichage des graphes dans le notebook.

Entrée[9]:
```
%matplotlib notebook
```

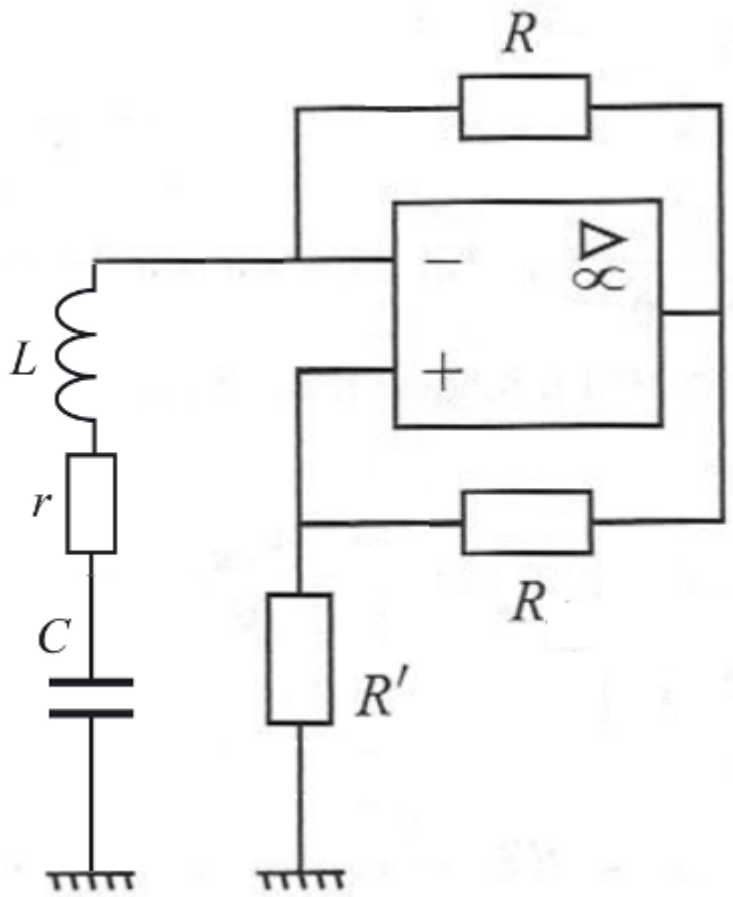
Entrée[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Nous allons fabriquer un détecteur de métaux, sur un principe de base très simple, la détection de la modification de l'inductance propre d'une bobine à l'approche d'un matériau ferromagnétique.

> Cet effet, inductif, sera étudié beaucoup plus tard dans l'année.

Réalisation d'un oscillateur LC à l'aide d'un montage à résistance négative

Comme vu en cours, l'idée est d'utiliser un montage à résistance négative pour compenser la résistance d'un circuit RLC.



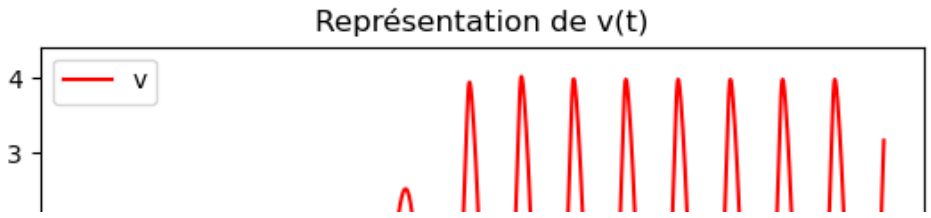
On peut montrer que $v_+ = -R' \cdot I_e$ si bien que R' devra être réglé pour compenser la résistance totale du circuit.

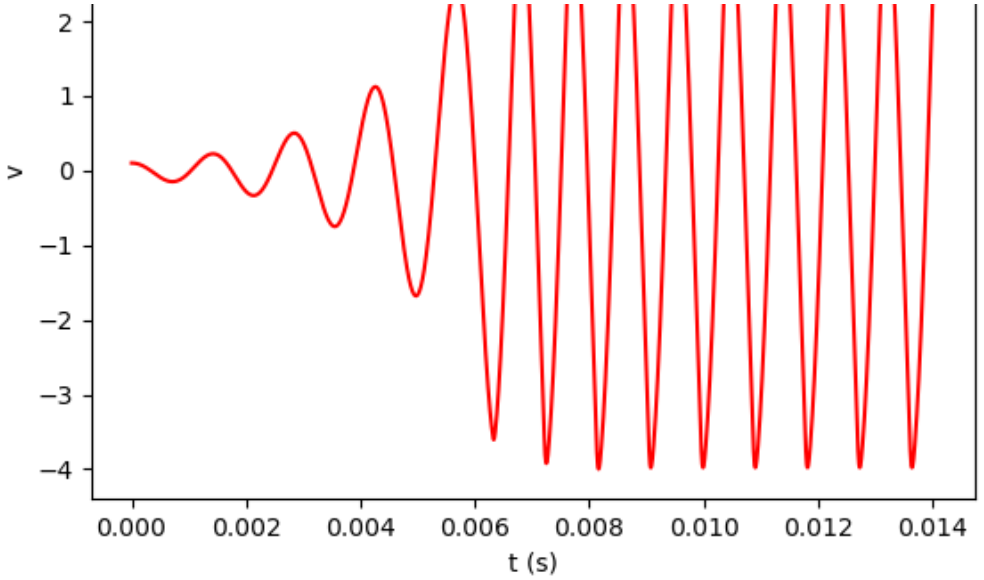
Mais on montrera beaucoup plus tard en cours que la résistance de la bobine répend de la fréquence, si bien que l'on se contentera de régler cette résistance << à la main >> en surveillant la naissance des oscillations dans le circuit.

Réaliser le montage et régler la résstance R' de façon à créer des oscillations quasi-sinusoïdales.

> On prendra $C = 10$ nF et $R = 10\ \text{k}\Omega$. R' est à régler mais devrait être proche de la résistance de la bobine (indiquée dessus, ou à mesurer).

La simulation numérique du comportement de cet oscillateur donne cette courbe pour la naissance des oscillations :





Jouer avec la valeur limite de R' pour visualiser à l'oscilloscope la naissance des oscillations.

> Il faudra utiliser la fonction `Run/Stop` pour figer l'écran de l'oscilloscope.

La fréquence d'oscillation est donnée par (cf. cours) :

$$f0 = \frac{1}{2\pi\sqrt{LC}}$$

Mesurer la fréquence des oscillations à l'oscilloscope, et en déduire la valeur de l'inductance de la bobine à la fréquence d'oscillation.

Entrée[ ]:
```
f0=
C=
L=1/(4*np.pi**2*C*f0**2)
print(L)
```

Élaboration d'un détecteur de métaux

Les premiers détecteurs fonctionnaient selon le principe du battement de fréquence. La technique des très basses fréquences permit d'atteindre une meilleure sensibilité, puis dans les années 1960, l'induction par impulsion fut mise au point et elle est actuellement encore la plus utilisée.

Principe général du détecteur à battement de fréquence

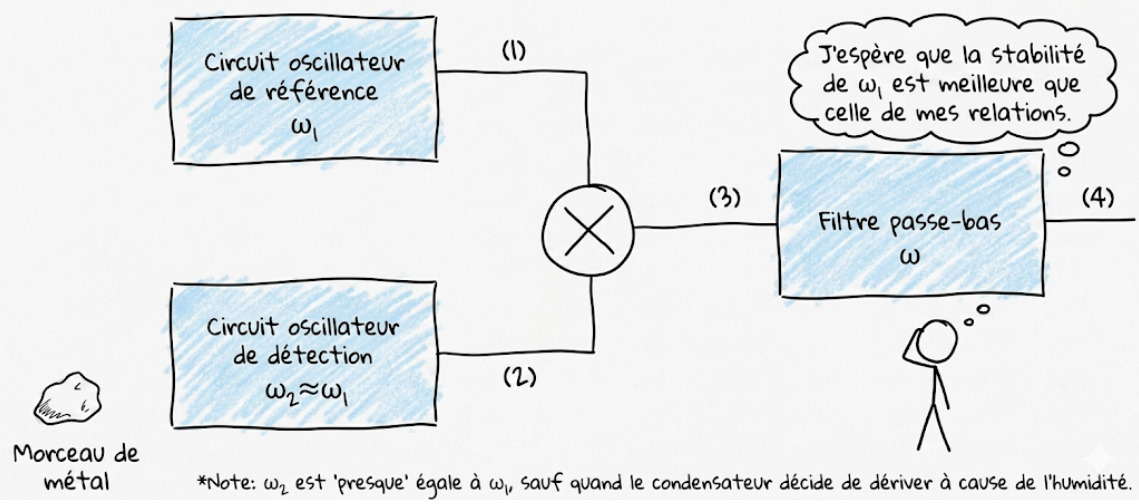
Le principe est d'utiliser deux oscillateurs, l'un fixe, l'autre dont la fréquence d'oscillation dépend de la présence d'un métal.

La présence d'un métal près d'une bobine modifie par mutuelle induction son inductance propre et donc, si un oscillateur est construit avec de cette dernière, celui-ci aura sa fréquence d'oscillation qui varie légèrement lors la présence de métal.

Il suffit ensuite de comparer les fréquences des deux signaux, pour cela on les fait <<

battre >> en les multipliant l'un par l'autre puis en filtrant le signal résultant.

Le montage utilisé suivra le principe décrit sur le diagramme ci-dessous :



Un oscillateur de référence (un GBF pour nous) fournit un signal sinusoïdal (1) de pulsation ω_1 et un oscillateur (notre montage à résistance négative) un signal sinusoïdal (2) de pulsation ω_2.

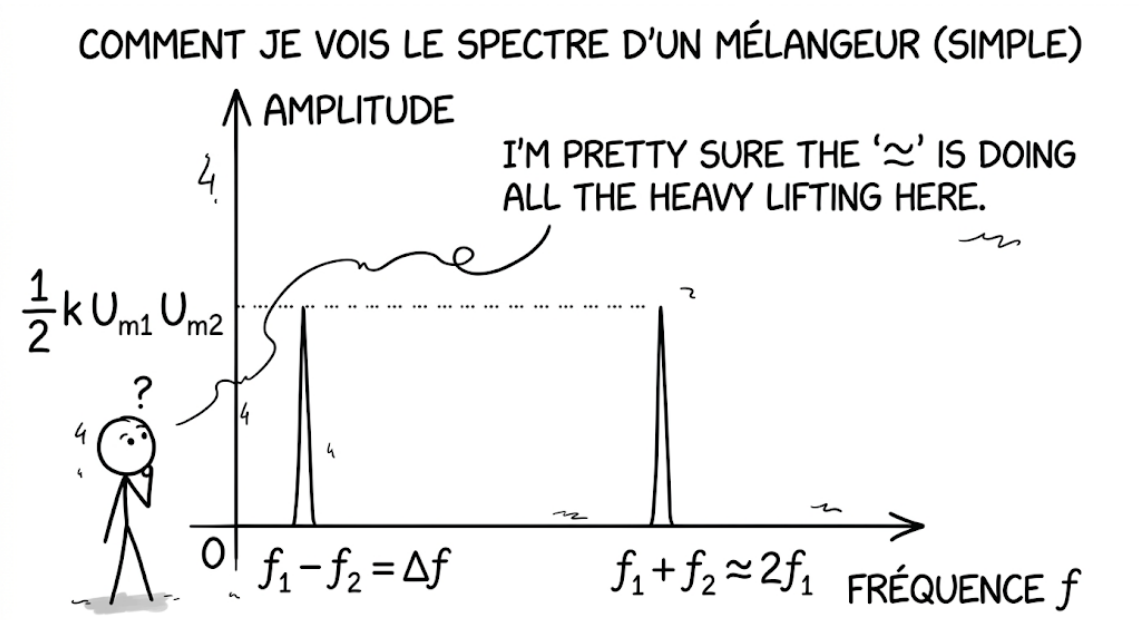
$$u_1 = U_{m1}\cos(\omega_1 t + \varphi_1) \text{ et } u_2 = U_{m2}\cos(\omega_2 t + \varphi_2)$$

Lorsqu'aucun métal n'est présent, on a $\omega_1 = \omega_2$. Lors de la présence d'un métal, la pulsation du signal (2) est légèrement modifiée.

Après multiplication des deux signaux, on obtient en (3) un signal :

$$u_3 = KU_{m1}U_{m2}\cos(\omega_1 t + \varphi_1)\cos(\omega_2 t + \varphi_2) = \frac{KU_{m1}U_{m2}}{2}(\cos((\omega_1 - \omega_2)t + (\varphi_1 \cdot$$

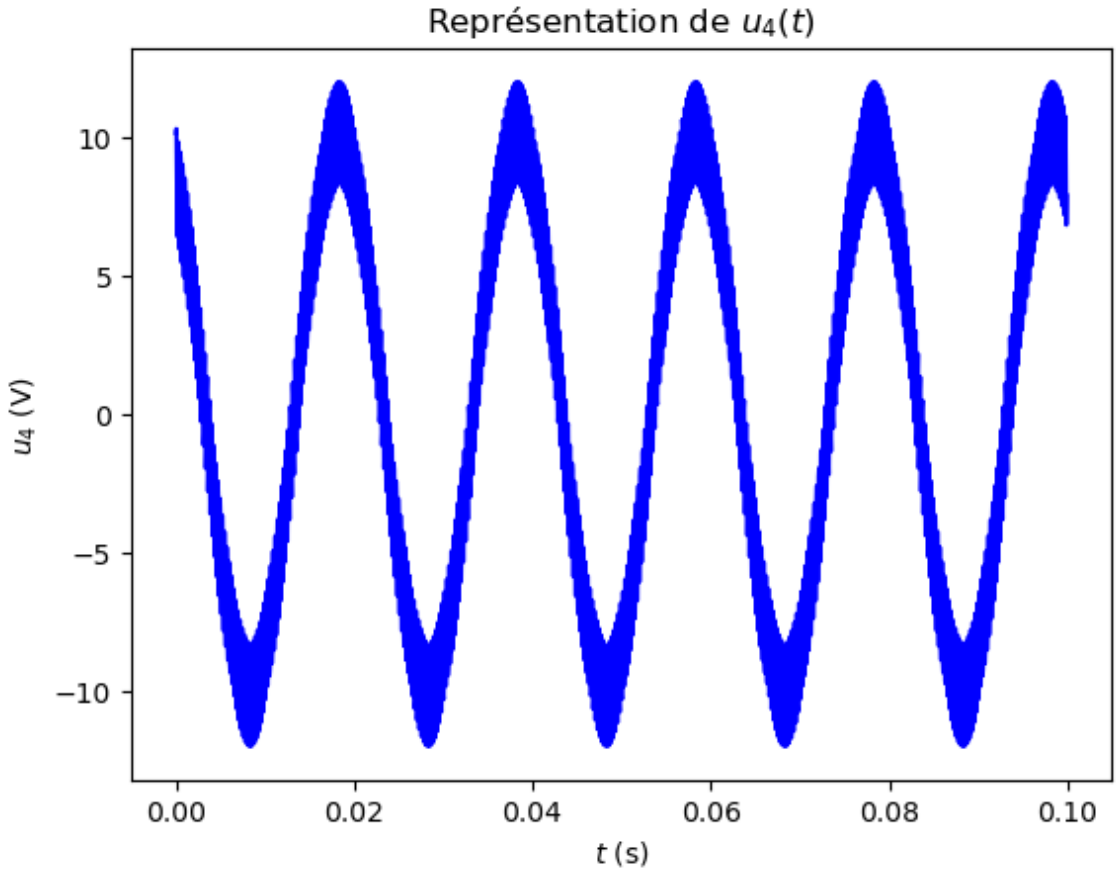
dont le spectre est représenté ci-dessous :



Il ne reste plus qu'à effectuer un filtrage des hautes fréquences pour récupérer un signal (4) dont la fréquence est Δf, que l'on peut envoyer par exemple sur une DEL pour qu'elle clignote si un métal est présent ($f_1 \neq f_2$).

Représentons ce que cela peut donner après filtrage (léger) :

Entrée[22]:
```
Df=50 # Hz

f1=2e3 # Hz
f2=f1+Df # Hz

A=10 # Valeur arbitraire de l'amplitude K.Um1.Um2/2
phi1=0 # Valeur arbitraire de la phase du signal 1
phi2=np.pi/6 # Même chose pour le signal 2

Ghf=0.2 # Valeur arbitraire du gain du filtre à la fréquence f=f1+f2
phihf=-0.8*np.pi/2 # idem pour le déphasage introduit par le filtre
G0=1 # Valeur arbitraire du gain du filtre à la fréquence f1-f2
phi0=0 # idem pour le déphasage introduit par le filtre

duree=5/Df # Affichage de 5 périodes
N=2000
dt=duree/N
t=np.array([k*dt for k in range(N)])

u4=G0*A*np.cos(2*np.pi*(f1-f2)*t+(ph1-phi2)+phi0)+Ghf*A*np.cos(2*np.p

plt.figure()
plt.plot(t,u4,'b-')
plt.title('Représentation de $u_4(t)$')
plt.xlabel('$t$ (s)')
plt.ylabel('$u_4$ (V)')
plt.show()
```



Montage

Il faut maintenant réaliser le montage complet :

- utiliser le montage à résistance négative comme oscillateur de détection (prendre la tension à la sortie de l'ALI) ;
- utiliser un GBF, à régler à la **même fréquence** $f_1 = f_2$ et (c'est mieux) à la même amplitude que le signal de détection ;
- multiplier les 2 signaux ensemble ;
- filtrer le signal à l'aide d'un filtre RC.

La dernière étape (le filtre) peut se faire à l'aide d'un filtre RC, de fréquence de coupure :

$$f_c = \frac{1}{2\pi RC}$$

Déterminer la valeur de C nécessaire sachant que $R \sim 10\ \mathrm{k\Omega}$.

Entrée[ ]:
```
R=10e3 # Ohm
C=
print(C)
```

Réaliser le montage complet et vérifier son bon fonctionnement.

> Comme la résistance de la bobine augmente avec la fréquence, les oscillations peuvent disparaître car R' ne compense plus la résistance totale du circuit dans l'oscillateur à résistance négative. Il faut alors augmenter légèrement R' pour faire redémarrer les oscillations du circuit de détection.

Vous devez constater que lorsqu'aucun morceau de matériau ferromagnétique n'est approché de la bobine, la fréquence du signal $u_4(t)$ est très faible puisque $f_1 \simeq f_2$ (sinon régler la fréquence du GBF f_1 à exactement la même fréquence que l'oscillateur à résistance négative f_2).

Lorsqu'un morceau de matériau ferromagnétique est approché, le fréquence du signal $u_4(t)$ est modifiée de $\Delta f \sim 10$ à 100 Hz. En branchant une DEL en sortie vous pouvez la voir clignoter.

Création d'un banc de test de la sensibilité du détecteur

Nous allons créer un << banc de test >> de ce détecteur. L'idée est de déterminer l'influence de la distance et du type de matériau sur la qualité de la détection.

Monter la bobine et la plaque de fer sur le banc d'optique, puis mesurer la variation de fréquence Δf en fonction de la distance de la plaque.

Entrée[ ]:
```
d=[]
Df_fer=[]

plt.figure()
plt.plot(d,Df_fer,'ro')
plt.title('Représentation de $\Delta f(d)$ pour le fer')
plt.xlabel('$d$ (m)')
plt.ylabel('$\Delta f$ (Hz)')
plt.show()
```

Recommencer avec les mêmes distances pour les plaques de zinc et de cuivre.

Entrée[ ]:
```
Df_zinc=[]
Df_cuivre=[]

plt.figure()
plt.plot(d,Df_fer,'ro')
plt.plot(d,Df_zinc,'bo')
plt.plot(d,Df_cuivre,'go')
plt.title('Représentation de $\Delta f(d)$')
plt.legend(['fer','zinc','cuivre'])
plt.xlabel('$d$ (m)')
plt.ylabel('$\Delta f$ (Hz)')
plt.show()
```

Commentez ces courbes :)

Type *Markdown* and LaTeX: α^2

TP transfert conductoconvectif

PE. LEROY

> Capacités exigibles du programme :
>
> **Bilans d'énergie**
>
> - Mettre en oeuvre une technique de calorimétrie.
> - Déterminer la valeur en eau d'un calorimètre.
> - Estimer les fuites thermiques lors d'expériences réalisées avec un calorimètre.

Liste du matériel :

- Eau chaude (bouilloire)
- Bécher de 500 mL
- Éprouvette graduée de 500 mL
- Balance
- Module Arduino, câcle USB
- Capteur de température étanche Grove DS18B20 ($\times 2$)
- Thermomètre à alcool

On commence par importer les bibliothèques nécessaires :

> La ligne suivante `%matplotlib notebook` est une commande magique qui sert à l'affichage des graphes dans le notebook.

Entrée[ ]:
```
%matplotlib notebook
```

Entrée[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
```

L'objectif est ici de suivre le refroidissement de l'eau chaude versée dans un verre, puis de modéliser finement le phénomène.

> Ce TP devrait servir à beaucoup d'entre vous pour leur TIPE.

Réalisation de l'expérience

Le protocole est le suivant (attention à l'ordre) :

- faire chauffer 500 mL d'eau dans une bouilloire ;
- placer le premier thermomètre relié au module Arduino dans le bécher tout en laissant le second dans l'air près du bécher (mesure de la température ambiante) ;
- verser l'eau dans le bécher (attention aux brulures) ;
- attendre 10 s environ puis lancer l'acquisition des deux températures.

> Attendre 10 s permet aux constituants du thermomètre d'atteindre la température du milieu, mais c'est une réflexion qui demande plus de développements (cf. temps de réponse des thermomètres plus loin).

Comme indiqué dans le protocole le suivi de température se fera à l'aide d'un module Arduino, et nous allons interagir avec le module Arduino directement à partir de ce notebook.

Commencer par brancher les deux thermomètres sur les entrées digitales `D2` et `D3`.

On commence ensuite d'abord par installer une bibliothèque spécifique et faire des tests de communication :

Entrée[ ]:
```
!pip install --user jupyno
```

Entrée[ ]:
```
import jupyno
```

Entrée[ ]:
```
port, FQBN = jupyno.trouver_arduino()
```

Les capteurs de température utilisés ici possèdent un circuit intégré `DS18B20`. Celui-ci nécessite des bibliothèques qui ne sont pas incluses par défaut dans l'IDE Arduino. Il faut donc les importer. Ces bibliothèques ont été récupérées sur le site du constructeur, ce sont les fichiers `OneWire.zip` et `DallasTemperature.zip`. On commence par les importer :

> Répondre `O` (oui) aux questions posées.

Entrée[ ]:
```
jupyno.installer_librairie('OneWire.zip')
jupyno.installer_librairie('DallasTemperature.zip')
```

On commence par créer un fichier vide (<< croquis >>) :

Entrée[ ]:
```
croquis = jupyno.creer_croquis('SuiviTemperature')
```

Nous allons écrire le programme suivant (notez la commande magique au début, qui sert ainsi à écrire le programme dans le << croquis >> directement au sein du notebook), essayez de comprendre sa structure puis exécutez la cellule :

Entrée[ ]:
```
%%ecrire_croquis $croquis

/**
 * Mesures de températures avec deux capteurs DS18B20 GROVE
 * Les mesures sont envoyées sur la liaison série
 */

/* Inclure les bibliothèques */
#include <OneWire.h>
#include <DallasTemperature.h>

/* Création de l'objet OneWire et sensors */
OneWire capteur1(2);   // Création d'un objet One Wire sur la broche
OneWire capteur2(3);   // Création d'un objet One Wire sur la broche

// Indiquer la référence OneWire à la bibliothèque Dallas Temperature
DallasTemperature sensors1(&capteur1);
DallasTemperature sensors2(&capteur2);

/* Pour faire des mesures à intervalle régulier */
unsigned long tempsPrecedent = 0;
unsigned long tempsCourant;
float intervalle = 2000; // Durée entre deux mesures en ms (valeur mi

void setup(){
  Serial.begin(115200);  // Choix de la vitesse

  sensors1.begin();
  sensors2.begin();
}

void loop(){
  tempsCourant = millis();  // Cette variable contient le nombre de m

  if (tempsCourant-tempsPrecedent>=intervalle){
      tempsPrecedent=tempsCourant;

      /* Demande la température aux capteurs */
      sensors1.requestTemperatures();
      float temperature_1 = sensors1.getTempCByIndex(0);  // Obtenir
      sensors2.requestTemperatures();
      float temperature_2 = sensors2.getTempCByIndex(0);  // Obtenir

      /* Envoie les températures sur la liaison série */
      Serial.print(tempsCourant);
      Serial.print(',');
      Serial.print(temperature_1);
      Serial.print(",");
      Serial.println(temperature_2);
  }
}
```

Il reste à le compiler et le téléverser sur le module Arduino :

Entrée[ ]:
```
jupyno.televerser(croquis, port, FQBN)
```

Côté notebook Jupyter nous allons récupérer en direct les données qui sont envoyées

sur le port série par le module Arduino. Pour cela il est nécessaire d'installer une librairie Python pour réaliser la communication sur le port série (décommenter la ligne suivante pour l'installation) :

Entrée[ ]:
```
#!pip install --user pyserial
```

Si l'acquisition est lancée dans le notebook et que vous souhaitez l'annuler, il faut interrompre le noyau et cela a pour conséquence de devoir relancer toute la chaine d'instructions précédente. Pour éviter cela nous allons créer la possibilité d'interrompre la boucle `while` par appui de la touche `q` pour quitter. Même procédé :

> Au moment où j'écris ces lignes je ne suis pas sûr du bon fonctionnement sur les machines du labo, je vous laisse tester en décommentant les lignes correspondantes.

Entrée[ ]:
```
#!pip install --user keyboard
```

On y va ! Essayez de comprendre la structure du programme de récupération des données sur le port série (port de communication) puis exécutez la cellule :

Entrée[ ]:
```
import serial
#import keyboard
#import time

# Acquisition
liste_t=[] # Création de la liste, vide
liste_T1=[] # Création de la liste, vide
liste_T2=[] # Création de la liste, vide
N=20 # Nombre de points

#print('Taper q pour quitter !')

with serial.Serial(port=port,baudrate=115200) as arduino:
    while len(liste_t)<N:
        #if keyboard.is_pressed('q'):
        #    break
        try:
            s=arduino.readline().decode('utf8').split(',')    # Lectu
            liste_t.append(int(s[0])) # Remplissage de la liste des t
            liste_T1.append(float(s[1])) # Remplissage de la liste de
            liste_T2.append(float(s[2]))
            print(int(s[0]),'ms',float(s[1]),'°C',float(s[2]),'°C') #
        except:
            pass
        #time.sleep(0.1) # Pour ne pas surcharger le processeur du fa
```

Cela fonctionne ? Essayez de prendre un capteur au creux de votre main pour voir l'évolution de la température.

Le nombre de points N sera à adapter en fonction de la durée de l'expérience, sachant que l'on peut régler dans le programme Arduino la durée entre deux acquisitions

(`intervalle` en ms).

On peut représenter les deux courbes :

Entrée[ ]:
```
plt.figure()

# Pour régler les paramètres de la représentation graphique (optionne
parameters = {"axes.labelsize": 13, "xtick.labelsize": 13, "ytick.lab
plt.rcParams.update(parameters)

plt.plot(liste_t,liste_T1,'ro')
plt.plot(liste_t,liste_T2,'bo')

plt.xlabel('t (ms)')
plt.ylabel('T (°C)')
plt.legend(['$T_1$','$T_2$'])
plt.title('Courbe d\'évolution de la température')

# Pour sauvegarder l'image de la courbe (optionnel)
plt.savefig('courbe.png')

plt.show()
```

Régler le nombre de points pour réaliser une acquisition sur 10 min. Appliquer le protocole indiqué au début du TP.

> La température T_1 devra être celle de l'eau et T_2 celle de l'air.

On peut aussi sauvegarder les données dans un fichier `CSV` :

Entrée[ ]:
```
# Sauvegarde des données dans un fichier extérieur (optionnel)
nomfichier='donnees.csv'

with open(nomfichier,'w') as fichier:
    fichier.write('t (s)'+';'+'T1 (°C)'+';'+'T2 (°C)'+'\n') # Ecritu
    for i in range(len(liste_T)):
        fichier.write(str(liste_t[i]).replace('.',',')+';'+str(liste_
```

Et l'importation des données depuis le fichier `CSV` se ferait de la facon suivante :

Entrée[ ]:
```
# Importation des données depuis un fichier extérieur (optionnel)
nomfichier='donnees.csv'

liste_t,liste_T1,liste_T2=np.char.replace(np.loadtxt(nomfichier,skipr
```

Analyse des résultats

L'analyse des résultats consiste à évaluer les incertitudes sur chaque mesure de façon à :

- valider le protocole ;
- avoir des critères de validation de la modélisation que l'on fera juste après.

L'incertitude sur la température est liée :

- à **la résolution** du capteur ;
- à **l'incertitude** du capteur (à comparer avec la résolution).

La résolution du capteur de température est de $0,01$°C, c'est à dire qu'il peut donner un résultat avec 2 chiffres après la virgule. Pourtant ce n'est pas son incertitude, qui est beaucoup plus grande.

L'incertitude est donnée dans la documentation du capteur :

PARAMETER	SYMBOL	CONDITIONS		MIN	TYP	MAX	UNITS
Supply Voltage	V_{DD}	Local power (Note 1)		+3.0		+5.5	V
Pullup Supply Voltage	V_{PU}	Parasite power	(Notes 1, 2)	+3.0		+5.5	V
		Local power		+3.0		V_{DD}	
Thermometer Error	t_{ERR}	-10°C to +85°C	(Note 3)			±0.5	°C
		-30°C to +100°C				±1	
		-55°C to +125°C				±2	

On peut lire $0,5$°C en moyenne si l'on opère sur un intervalle entre -10 et 85°C, intervalle sur lequel on opère ici. D'où :

Entrée[ ]:
```
liste_uT=[0.5]*len(liste_T1)
print(liste_uT)
```

L'incertitude sur le temps est d'abord celle de la mesure du temps avec le module Arduino. Elle dépend de deux facteurs :

- la **résolution** de la fonction `millis()` , qui est donc de 1 ms ;
- l'**incertitude** sur la mesure de l'horloge interne ($\sim 0,5$%), ce qui est plutôt mauvais.

La durée entre deux mesures étant de 2000 ms, cela donne 10 ms, supérieure à la résolution, on prend donc juste cette incertitude de 0,5$% :

> Cette incertitude est en %, ce qui signifie qu'elle augmente avec le temps : l'incertitude sur le temps indiqué augmente au cours de l'expérience, on parle de **dérive temporelle**.

Entrée[ ]:
```
liste_t=np.array(liste_t) # On passe du type 'list' au type 'np.array
liste_ut=liste_t*0.5/100
print(liste_ut)
```

Et le **temps de réponse** du capteur ? Le temps de réponse dépend de deux facteurs :

- le temps de calcul (ce capteur est un modèle à circuit intégré) ;
- le temps de mise en température de la partie interne qui est sensible à la température (à travers la paroi de la gaine métallique qui entoure le capteur).

La documentation donne le temps de calcul, qui est au maximum de 750 ms :

PARAMETER	SYMBOL	CONDITIONS		MIN	TYP	MAX	UNITS
Temperature Conversion Time	t_{CONV}	9-bit resolution	(Note 12)			93.75	ms
		10-bit resolution				187.5	
		11-bit resolution				375	

		12-bit resolution		750	

Le temps de diffusion thermique à travers le cylindre qui doit être en aluminium (on prend un ODG de 1 mm d'épaisseur) est donné par :

$$\tau \sim \frac{L^2}{D} \sim \frac{(10^{-3})^2}{10 \times 10^{-5}} \sim 10^{-2}\ \text{s} = 10\ \text{ms}$$

soit est **négligeable devant le temps de calcul**, on prendra donc 1 s de temps de réponse pour être large.

> Ainsi en attendant 10 s de démarrer les mesures on s'assure de la mise en température du capteur avec son milieu.

Par contre, cette incertitude signifie que si la température du milieu varie trop vite, le thermomètre affichera la température jusqu'à 1 s après qu'elle ait été atteinte. Il est donc plus raisonnable de corriger l'incertitude sur le temps en tenant compte de cette incertitude supplémentaire :

> Cf. les règles de calcul vue en première année, pour rappel elles sont indiquées dans ce notebook (../incertitudes/incertitudes_de_mesure.ipynb).
>
> $$u(t) = \sqrt{u_{\text{horloge}}(t)^2 + u_{\text{temps de réponse}}(t)^2}$$

Entrée[ ]:
```
uttr=1 # s
liste_ut=np.sqrt(liste_ut**2+uttr**2)
print(liste_ut)
```

On peut alors représenter les barres d'incertitudes sur le graphe :

> On ne s'intéresse qu'à la température de l'eau pour le moment, soit $T_1(t)$.

Entrée[ ]:
```
# Représentation graphique
plt.figure()

# Pour régler les paramètres de la représentation graphique (optionne
parameters = {"axes.labelsize": 13, "xtick.labelsize": 13, "ytick.lab
plt.rcParams.update(parameters)

# Barres d'incertitudes (attention à l'ordre (incohérent) du passage
plt.errorbar(liste_t,liste_T1,liste_uT,liste_ut,fmt='ro',ecolor='blue

plt.xlabel('t (ms)')
plt.ylabel('T (°C)')
plt.legend(['$T_1$'])
plt.title('Courbe d\'évolution de la température')

# Pour sauvegarder l'image de la courbe (optionnel)
plt.savefig('courbes_temperature.png')

plt.show()
```

Modélisation

On commence **toujours** par un modèle simple, puis éventuellement on l'affine.

> Cela permet de valider les phénomènes physiques mis en jeu que l'on présume lorsque l'on réalise cette démarche de modélisation.

Premier modèle simple

Le système étudié sera l'eau, et on néglige l'influence du bécher.

> Le bécher a bien sûr une influence, mais on en discutera ensuite.

Un bilan d'énergie (premier principe en terme de puissance) donne :

$$\frac{\mathrm{d}U}{\mathrm{d}t} = \phi_e - \phi_s + p_{\text{autres}}$$

On appelle $T(t)$ la température de l'eau et T_{ext} la température de l'air.

Montrer (au brouillon) que l'équation qui régit l'évolution de T est donnée par :

$$\frac{\mathrm{d}T}{\mathrm{d}t} = -\frac{hS}{C}(T - T_{ext})$$

Résoudre cette équation et donner l'expression complète de $T(t)$ en fonction de la température initiale notée T_0.

La solution est classiquement de forme exponentielle, de la forme $a \cdot e^{-b \cdot x} + c$. Nous allons tester ce modèle sur les données expérimentales en réalisant un **ajustement** exponentiel, c'est à dire que l'on va essayer d'ajuster une fonction exponentielle sur les données expérimentales :

Entrée[ ]:
```
import scipy.optimize as spo # Pour la fonction 'curve_fit'

# Ajustement selon une fonction d'ajustement définie
def f_ajust(x,a,b,c):
    return(a*np.exp(-b*x)+c)

# Ajustement sur les données de 'liste_T1'
popt,pcov=spo.curve_fit(f_ajust,liste_t,liste_T1,[70,0.01,20]) # Les

print('Équation a*exp(-b*t)+c : a=',popt[0],'b=',popt[1],'c=',popt[2]

# Création des listes de la fonction ajustée
liste_Tajust=[]
liste_tajust=np.linspace(liste_t[0],liste_t[-1],200) # Liste des vale
for t in liste_tajust:
    liste_Tajust.append(f_ajust(t,*popt)) # '*' indique que 'popt' es

# Représentation graphique
plt.figure()

# Pour régler les paramètres de la représentation graphique (optionne
parameters = {"axes.labelsize": 13, "xtick.labelsize": 13, "ytick.lab
plt.rcParams.update(parameters)

plt.plot(liste_t,liste_T1,'ro')
plt.plot(liste_tajust,liste_Tajust,'b-')

plt.title('Courbe d\'évolution de la température')
plt.xlabel('t (s)')
plt.ylabel('T (°C)')
plt.legend(['$T_1$','$T_{ajust}$'])

# Pour sauvegarder l'image de la courbe (optionnel)
plt.savefig('courbe_ajustement.png')

plt.show()
```

Cela fonctionne-t-il ? Pour vraiment comparer ce modèle aux données expérimentales, il faut :

- tenir compte des incertitudes ;
- comparer les valeurs obtenues lors de l'ajustement avec les valeurs des grandeurs physiques du problème.

On commence par tenir compte des incertitudes :

Entrée[ ]:
```
# Représentation graphique
plt.figure()

# Pour régler les paramètres de la représentation graphique (optionne
parameters = {"axes.labelsize": 13, "xtick.labelsize": 13, "ytick.lab
plt.rcParams.update(parameters)

plt.errorbar(liste_t,liste_T1,liste_uT,liste_ut,fmt='ro',ecolor='blue
plt.plot(liste_tajust,liste_Tajust,'b-')

plt.title('Courbe d\'évolution de la température')
plt.xlabel('t (s)')
plt.ylabel('T (°C)')
plt.legend(['$T_1$','$T_{ajust}$'])

# Pour sauvegarder l'image de la courbe (optionnel)
plt.savefig('courbe_ajustement.png')

plt.show()
```

Analysez le résultat.

Pour ce qui est des valeurs des grandeurs physiques du modèles, montrer que l'on doit avoir :

$$a = T_0 - T_{ext}$$

$$b = \frac{hS}{C}$$

$$c = T_{ext}$$

Il nous faut les valeurs de T_exT, dans ce premier modèle T_exT reste constant (l'air ambiant est modélisé par un thermostat), on peut constater que T_2 varie peu en effet, on va donc prendre la moyenne :

Entrée[ ]:
```
Text=np.mean(liste_T2)
print(Text)
```

T_0 correspond à la température initiale :

Entrée[ ]:
```
T0=liste_T1[0]
print(T0)
```

Le facteur $\frac{hS}{C}$ est plus délicat à estimer car on ne connaît pas h, qui est un facteur << expérimental >>, mais on connaît son ODG : $h \sim 10$ à $100\ \mathrm{W.m^{-2}.K^{-1}}$.

> Pour rappel ce premier modèle correspond à de l'eau << en lévitation >> en interaction uniquement avec l'air extérieur. On peut aussi considérer que l'on a une résistance thermique $R_{th} = \frac{1}{hS}$.

On part donc des valeurs de C et S :

- C est la capacité thermique de l'eau, soit $C = m_{eau} \cdot c_{eau}$;
- S est la surface en contact avec l'air, soit $S = 2\pi RH + \pi R^2$ avec R le rayon du bécher et H sa hauteur.

> On néglige ici l'interaction avec la table au niveau de la base, ce n'est pas si évident .

Entrée[ ]:
```
meau=0.5 # kg
ceau=4180 # J/kg/K

R= # m
H= # m
S=2*np.pi*R*H+np.pi*R**2
print(S)
```

Comparons rapidement :

Entrée[ ]:
```
# Comparaisons pour a et c
print(a,T0-Text)
print(c,Text)

# Valeur de h à comparer à son ODG
print(b*C/S)
```

Sans passer par des calculs d'écarts normalisés, qu'en pensez-vous ?

> Une amélioration rapide du modèle est de prendre en compte le bécher dans le système. En considérant que la température du bécher est toujours en équilibre avec l'eau à tout instant (d'où l'importance d'attendre au moins 10 s avant de démarrer les mesures), il suffit d'ajouter $C_{\text{bécher}} = C_b$ à C. Je vous laisse essayer (la capacité thermique massique du verre PYREX est $c_{\text{PYREX}} = 728\ \text{J.kg}^{-1}\text{.K}^{-1}$).

Second modèle plus élaboré

Dans un second modèle, on peut prendre en compte le sous-système bécher. On considère ainsi deux sous-systèmes auxquels on applique un bilan d'énergie. On appelle $T_b(t)$ la température du bécher, $h_{e/b}$ le coefficient de transfert conductoconvectif entre l'eau et le bécher, $h_{b/a}$ le coefficient de transfert conductoconvectif entre le bécher et l'air extérieur. Pour les surfaces d'interaction, on prendra pour simplifier la même surface $2\pi RH + \pi R^2$. On néglige l'interaction entre l'eau et l'air à la surface de l'eau.

C'est très discutable ça ! Oui, il faudrait plus de temps pour un modèle complet (interaction avec la table, etc.).

La température $T_b(t)$ est difficilement accessible expérimentalement.

$$C\frac{\mathrm{d}T}{\mathrm{d}t} = -h_{e/b}S(T - T_b)$$

$$C_b\frac{\mathrm{d}T_b}{\mathrm{d}t} = -h_{b/a}S(T_b - T_{ext}) - h_{e/b}S(T_b - T)$$

Nous allons réaliser une résolution numérique de ces équations couplées :

Compléter le programme suivant permettant la résolution numérique de ces équations avec la méthode d'Euler :

Entrée[ ]:
```
# Définition des constantes
C=meau*ceau # J/K
mb= # kg
cb=728 # J/kg/K
Cb=mb*cb # J/K

# Paramètres à ajuster
heb=10 # W/m^2/K
hba=10 # W/m^2/K

# Températures limites
T0=liste_T1[0]
Text=np.mean(liste_T2)

# Paramètres de la résolution numérique
N=2000 # Nombre de points de la résolution numérique
duree=liste_t[-1] # Durée totale
dt=duree/N # Pas de la résolution numérique

# Définition des listes
T=[T0]
Tb=[T0]
t=[0]

# Résolution par la méthode d'Euler (simple discrétisation)
for n in range(N-1):
    T.append(T[n]+dt*(-heb*S/C*(T[n]-Text)))
    # À compléter :
    # Tb.append(...)

    t.append(t[n]+dt)

# Représentation graphique
plt.figure()

# Pour régler les paramètres de la représentation graphique (optionne
parameters = {"axes.labelsize": 13, "xtick.labelsize": 13, "ytick.lab
plt.rcParams.update(parameters)

plt.plot(t,vs,'r-')
plt.title('Représentation de $T(t)$')
plt.xlabel('t (s)')
plt.ylabel('$T$ (°C)')

# Pour sauvegarder l'image de la courbe (optionnel)
plt.savefig('modele.png')

plt.show()
```

Cela fonctionne ? On peut superposer la courbe obtenue avec la courbe expérimentale :

Entrée[ ]:
```
# Représentation graphique
plt.figure()

# Pour régler les paramètres de la représentation graphique (optionne
parameters = {"axes.labelsize": 13, "xtick.labelsize": 13, "ytick.lab
plt.rcParams.update(parameters)

plt.errorbar(liste_t,liste_T1,liste_uT,liste_ut,fmt='ro',ecolor='blue
plt.plot(t,T,'r-')

plt.title('Courbe d\'évolution de la température')
plt.xlabel('t (s)')
plt.ylabel('T (°C)')
plt.legend(['$T_1$','$T_{modele}$'])

# Pour sauvegarder l'image de la courbe (optionnel)
plt.savefig('superposition_modele.png')

plt.show()
```

Essayer d'ajuster manuellement les valeurs des paramètres $h_{e/b}$ et $h_{b/a}$ pour avoir une superposition optimale, puis conclure sur la cohérence des valeurs obtenues.

TP détecteur de passager

PE. LEROY

> Capacités exigibles du programme :
>
> - Décrire qualitativement le phénomène d'influence électrostatique.

Liste du matériel :

- Plaquette de montage de composants
- Résistances de $10\ \mathrm{k\Omega}$, $10\mathrm{M\Omega}$
- Condensateur de $100\ \mu\mathrm{F}$
- Condensateur artisanal (2 feuilles d'aluminium plastifiées, chacune connectée à un petit fil)
- Module Arduino, câble USB
- GBF
- Oscilloscope
- Pèse-personne
- Masses marquées assez lourdes (plusieurs jeux pour toute la classe)

On commence par importer les bibliothèques nécessaires :

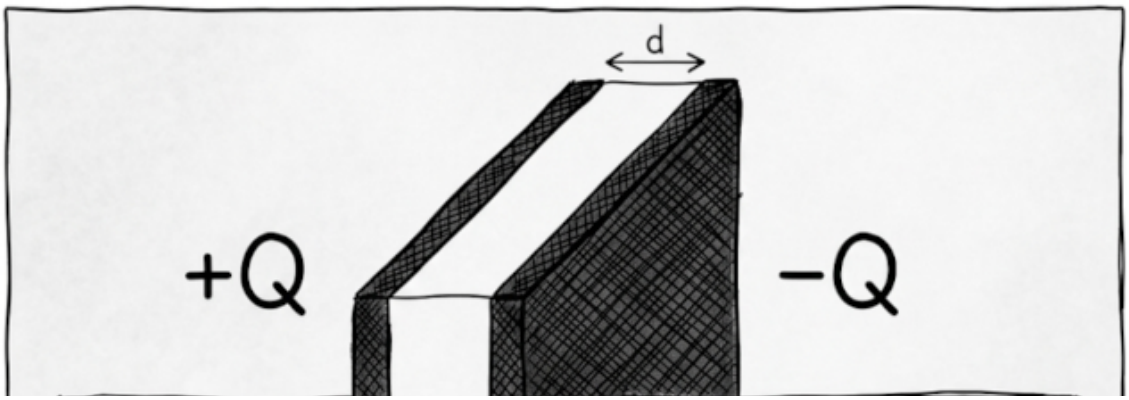
> La ligne suivante `%matplotlib notebook` est une commande magique qui sert à l'affichage des graphes dans le notebook.

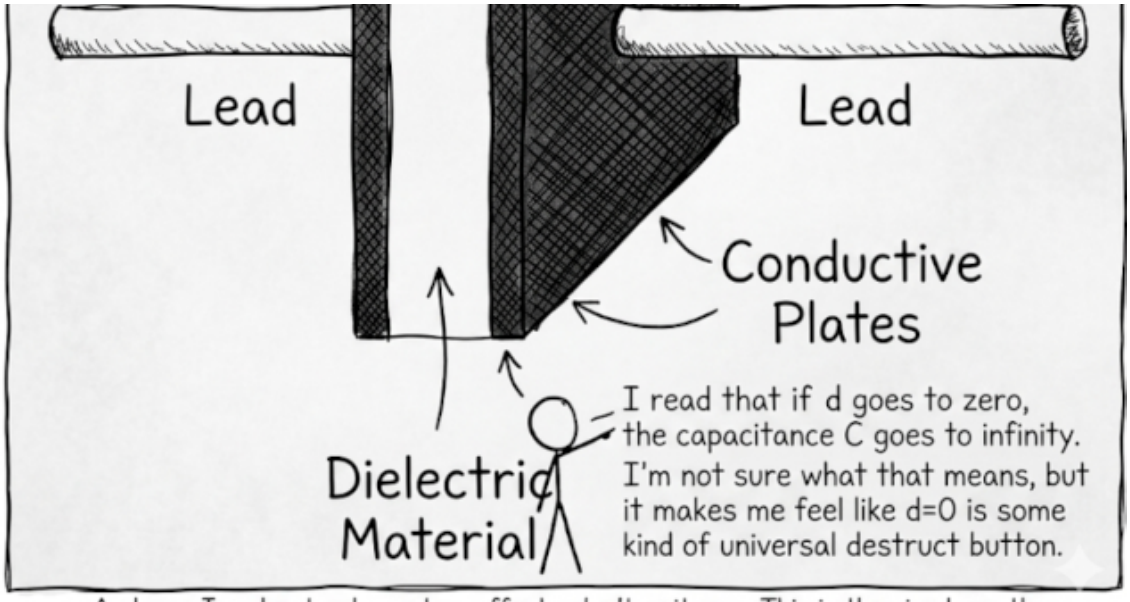
Entrée[9]:
```
%matplotlib notebook
```

Entrée[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Nous allons élaborer un détecteur de passager tel qu'on peut le retrouver dans les véhicules pour détecter si un passager est assis alors que sa ceinture de sécurité n'est pas en place.

Le principe est d'utiliser un condensateur constitué de 2 feuille d'aluminium séparées par un isolant. Lorsqu'il est comprimé, sa capacité C varie, on détecte alors cette variation de capacité.





On commence donc par élaborer une méthode de mesure de la capacité C d'un condensateur, réalisée à l'aide d'un module Arduino.

Le module Arduio

Nous allons utiliser un module Arduino, possédant un microcontrôleur programmable pouvant interagir avec un ensemnle de broches E/S, pour récupérer une tension au cours du temps.

Voici quelques **principes fondamentaux** sur le module Arduino UNO :

- le microcontrôleur peut interagir avec les différentes broches E/S ;
- chaque broche E/S fonctionne entre 0 et 5 V (pas de tension négative) ;
- toutes les tensions sont relatives à la masse GND correspondant à 0 V ;
- certaines broches (analogiques) permettent de mesurer une tension entre 0 et 5 V

(broches A0 à A5) ;
- d'autres broches (numériques) permettent de mesurer ou délivrer une tension de 0 OU 5 V (broches 0 à 13) ;
- le programme doit être écrit en langage C++, qu'il faut alors compiler et téléverser vers le module Arduino.

Mesurer une tension sur une broche analogique

Prenons l'exemple de la mesure d'une tension sur l'entrée analogique `A0`. La fonction à utiliser dans le programme est `analogRead(A0)`. Cette fonction retourne une valeur N entre 0 et 1023, qui correspond à l'échelle 0-5 V.

Ainsi la tension est alors :

$$u = \frac{N}{1023} \times 5$$

Un programme pour mesurer et afficher cette tension pourrait ainsi être :

```
// broches
const int(mesure)=A0;

// grandeurs
float u;

void setup()
{
  // initialisation moniteur serie
  Serial.begin(9600);
}

void loop()
{
  // mesure
  u=analogRead(mesure)*5.0/1023;

  // affichage
  Serial.print("u : ");
  Serial.print(u);
  Serial.println(" V");

  delay(1000); // attente de 1 s
}
```

Plusieurs commentaires s'imposent :

- contrairement au langage Python, l'indentation n'a aucune fonction en C++ (si ce n'est faciliter la lecture), les blocs de code sont définis par des accolades {} ;
- contrairement au langage Python, les grandeurs utilisées doivent être définies avec leur type en début de programme ;
- le code est constitué de deux fonctions (`void`), une première `setup()` ne sera exécutée qu'une seule fois au lancement, une seconde `loop()` qui sera exécutée en boucle (en permanence) ;

- les commentaires sont introduits par `//` .

L'affichage du résultat de la mesure se fait à l'aide des fonctions `Serial.print()` (afficher) et `Serial.println()` (afficher puis aller à la ligne). Les données à afficher sont envoyées sur le port série, où l'on pourra les récupérer à l'aide d'un programme Python.

La vitesse de communication est définie par la fonction `Serial.begin()` et s'exprime en bauds (nombre d'éléments transmis par seconde).

On peut remarquer l'utilisation de la fonction `delay()` pour marquer une pause, ici de 1000 ms.

Exemple

Branchez le module Arduino à l'ordinateur via une prise USB. À l'aide d'un fil, reliez la broche `3.3V` du module à la broche `A0` de celui-ci.

> Ainsi, on met la broche `A0` au potentiel 3,3 V, la mesure donnera donc comme résultat :
>
> $$u = V_{A0} - V_{GND} = 3,3 - 0 = 3,3V$$

Nous allons donc chercher à récupérer dans un programme Python l'état de la broche `A0` .

Nous allons interagir avec le module Arduino directement à partir de ce notebook, c'est à dire écrire et compiler le code puis le transférer au module Arduino. Pour que ce notebook puisse piloter le module Arduino, on commence par installer une bibliothèque spécifique et faire des tests de communication.

Entrée[ ]:
```
!pip install --user jupyno
```

Entrée[ ]:
```
import jupyno
```

Entrée[ ]:
```
port, FQBN = jupyno.trouver_arduino()
```

On commence par créer un fichier vide (<< croquis >>) :

Entrée[ ]:
```
croquis = jupyno.creer_croquis('SuiviTension')
```

Nous allons écrire le programme suivant (notez la commande magique au début, qui sert ainsi à écrire le programme dans le << croquis >> directement au sein du notebook), essayez de comprendre sa structure puis exécutez la cellule :

Entrée[ ]:
```
%%ecrire_croquis $croquis

// broches
const int(mesure)=A0;

// grandeurs
float u;

void setup()
{
  // initialisation moniteur serie
  Serial.begin(9600);
}

void loop()
{
  // mesure
  u=analogRead(mesure)*5.0/1023;

  // affichage
  Serial.println(u);

  delay(1000); // attente de 1 s
}
```

Il reste à le compiler et le téléverser sur le module Arduino :

Entrée[ ]:
```
jupyno.televerser(croquis, port, FQBN)
```

Côté notebook Jupyter nous allons récupérer en direct les données qui sont envoyées sur le port série par le module Arduino. Pour cela il est nécessaire d'installer une librairie Python pour réaliser la communication sur le port série (décommenter la ligne suivante pour l'installation) :

Entrée[ ]:
```
#!pip install --user pyserial
```

On y va ! Essayez de comprendre la structure du programme de récupération des données sur le port série (port de communication) puis exécutez la cellule :

Entrée[ ]:
```
import serial

# # Acquisition
liste_u=[] # Création de la liste, vide
N=20       # Nombre de points

with serial.Serial(port=port,baudrate=9600) as arduino:
    while len(liste_u)<N:
        try:
            s=arduino.readline().decode('utf8').split(',') # Lecture
            liste_u.append(float(s[0])) # Remplissage de la liste
            print(float(s[0]),'V') # Affichage des valeurs (pour véri
        except:
            pass

print(liste_u)
```

Mesurer un temps

Une horloge est nécessairement présente dans le microcontrôleur. On peut par exemple accéder au temps écoulé depuis l'initialisation de la carte (par exemple lors du téléversement du programme), à l'aide de la fonction `millis()` qui retourne ce temps en ms.

Modifier les programmes suivants pour afficher, en plus de la tension u, le temps t écoulé depuis l'initialisation de la carte.

> On récupèrera les valeurs de t dans une liste d'entiers nommée `liste_t` .
>
> On utilisera le type `unsigned long` pour t dans le programme Arduino.

Entrée[ ]:
```
%%ecrire_croquis $croquis

// broches
const int(mesure)=A0;

// grandeurs
float u;

void setup()
{
  // initialisation moniteur serie
  Serial.begin(9600);
}

void loop()
{
  // mesure
  u=analogRead(mesure)*5.0/1023;

  // affichage
  Serial.println(u);

  delay(1000); // attente de 1 s
}
```

Entrée[ ]:
```
jupyno.televerser(croquis, port, FQBN)
```

Entrée[ ]:
```
import serial

# # Acquisition
liste_u=[] # Création de la liste, vide
N=20       # Nombre de points

with serial.Serial(port=port,baudrate=9600) as arduino:
    while len(liste_u)<N:
        try:
            s=arduino.readline().decode('utf8').split(',') # Lecture
            liste_u.append(float(s[0])) # Remplissage de la liste
            print(float(s[0]),'V') # Affichage des valeurs (pour véri
        except:
            pass
```

Imposer une tension sur une broche numérique

On peut vouloir imposer une tension de 0 V ou 5 V sur une broche numérique (on ne peut pas imposer autre chose), cela se fait de la façon suivante :

- on définit la broche numérique dans la fonction `setup()` comme une broche de sortie avec `pinMode(numero_broche,OUTPUT)` ;
- on définit la tension à imposer avec `digitalWrite(numero_broche,HIGH)` pour imposer 5 V ou `digitalWrite(numero_broche,LOW)` pour imposer 0 V ;
- l'affichage du temps et de la tension se fera à l'aide des fonctions `Serial.print(t);` (affichage sans passage à la ligne) et `Serial.println(u);` (affichage avec passage à la ligne), séparées par un `Serial.print(",");` .

Copier et modifier les programmes précédents pour afficher la tension u imposée par la broche numérique 7, basculant alternativement de 0 V à 5 V, toutes les secondes environ, ainsi que le temps t écoulé depuis l'initialisation de la carte (séparer les deux données u, t sur une même ligne par une " , ").

Entrée[ ]:

Entrée[ ]:
```
jupyno.televerser(croquis, port, FQBN)
```

Entrée[ ]:

Représentation d'une tension au cours du temps

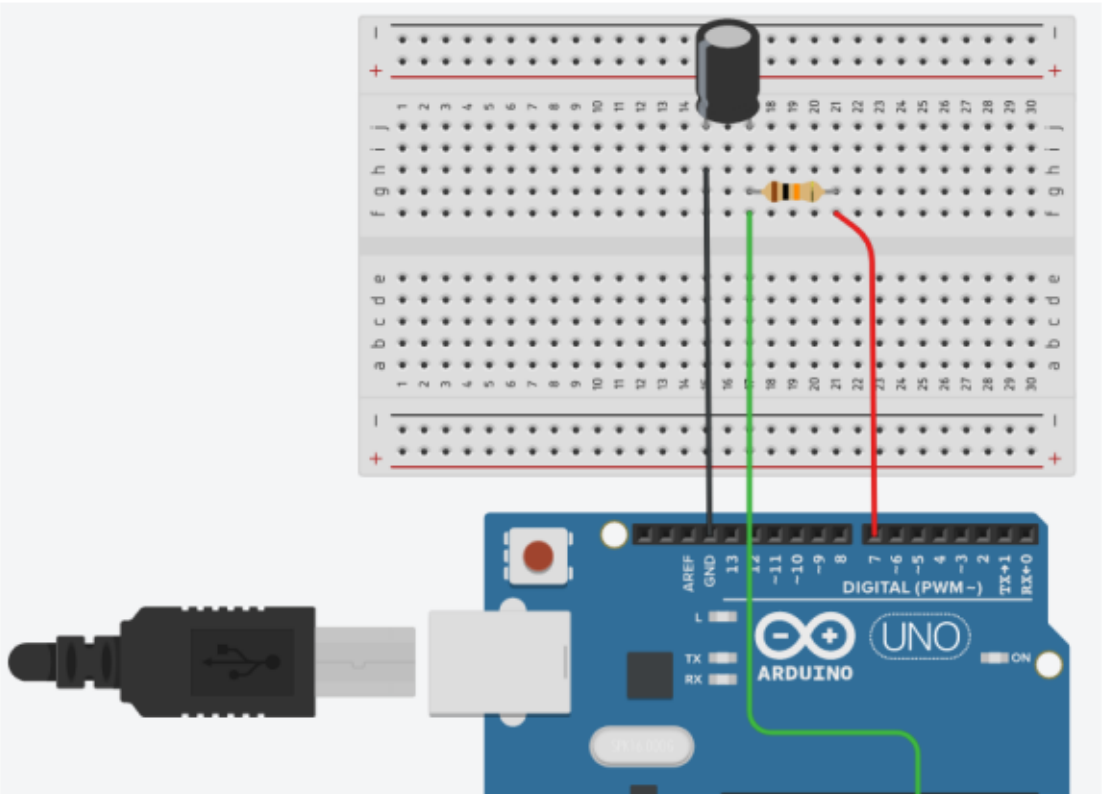
Il suffit alors de réaliser la représentation graphique de $u(t)$:

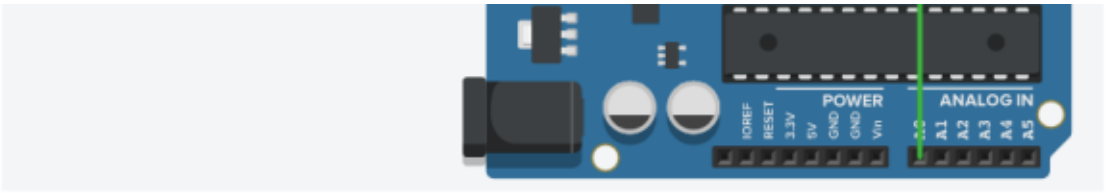
Entrée[ ]:
```
plt.figure()
plt.plot(liste_t,liste_u,'ro')
plt.xlabel('t (ms)')
plt.ylabel('u (V)')
plt.title('Courbe d\'évolution de la tension')
plt.show()
```

Suivi de la charge d'un condensateur avec un module Arduino

Nous sommes prêts ! L'objectif est maintenant de réaliser le suivi de la charge d'un condensateur. Pour cela on réalise un simple circuit RC. Le temps caractéristique d'un tel circuit est donné par :

$$\tau = RC$$





Réaliser le montage ci-dessus. Utiliser les programmes ci-dessous (à adapter éventuellement) pour récupérer l'évolution de la tension aux bornes d'un condensateur en fonction du temps lors de sa charge. Représenter graphiquement le résultat obtenu.

> On prendra $R = 10\ \text{k}\Omega$ et $C = 100\ \mu\text{F}$.

Entrée[ ]:
```
%%ecrire_croquis $croquis

// broches
const int(alim)=7;
const int(mesure)=A0;

// grandeurs
unsigned long t;
float u;

void setup()
{
  // initialisation sortie numérique
  pinMode(alim,OUTPUT);

  // initialisation du moniteur série
  Serial.begin(9600);

  // decharge du condensateur
  digitalWrite(alim,LOW);

  // attente de 3 s
  delay(3000);

  // charge du C
  digitalWrite(alim,HIGH);
}

void loop()
{
  // mesure
  t=millis();
  u=analogRead(mesure)*5.0/1023;

  // affichage
  Serial.print(t);
  Serial.print(",");
  Serial.println(u);
}
```

Entrée[ ]:
```
jupyno.televerser(croquis, port, FQBN)
```

Vous pouvez lancer le programme python ci-dessous puis appuyer sur le bouton RESET du module Arduino pour relancer le programme sur le microprocesseur depuis le début (la fonction `setup()` s'exécutera de nouveau).

Entrée[ ]:
```
import serial

# # Acquisition
liste_t=[] # Création de la liste des temps, vide
liste_u=[] # Création de la liste des tensions, vide
N=20       # Nombre de points

with serial.Serial(port=port,baudrate=9600) as arduino:
    while len(liste_u)<N:
        try:
            s=arduino.readline().decode('utf8').split(',') # Lecture
            liste_t.append(float(s[0])) # Remplissage de la liste
            liste_u.append(float(s[1])) # Remplissage de la liste
            print(float(s[0]),'ms') # Affichage des valeurs (pour vé
            print(float(s[1]),'V') # Affichage des valeurs (pour véri
        except:
            pass
```

Entrée[ ]:
```
plt.figure()
plt.plot(liste_t,liste_u,'ro')
plt.xlabel('t (ms)')
plt.ylabel('u (V)')
plt.title('Courbe d\'évolution de la tension')
plt.show()
```

Mesure de la constante de temps d'un circuit RC

À l'aide des programmes ci-dessous (à adapter éventuellement), mesurer et renvoyer la constante de temps τ du circuit RC précédent et la capacité C correspondante à l'aide du module Arduino, puis afficher la valeur moyenne de la capacité C correspondante sur plusieurs cycles de mesure.

> On prendra $R = 10\ \text{k}\Omega$ et $C = 100\ \mu\text{F}$.

Entrée[ ]:
```
%%ecrire_croquis $croquis

// broches
const int(alim)=7;
const int(mesure)=A0;

// grandeurs
unsigned long debutCharge;
unsigned long tau;
float C;
float R=10e3;

void setup()
{
  // initialisation sortie numérique
  pinMode(alim,OUTPUT);

  // initialisation du moniteur série
  Serial.begin(9600);
  Serial.println("Initialisation...");
}

void loop()
{
  // decharge du condensateur
  digitalWrite(alim,LOW);

  // attente de 3 s
  delay(3000);

  // charge du C
  digitalWrite(alim,HIGH);

  debutCharge=millis(); // mesure du temps initial

  while(analogRead(mesure)<647) // 647=63% de 1023
  {
    // on attend...
  }

  tau=millis()-debutCharge;
  C=tau*1e-3/R*1e9; // tau en ms et C en nF

  // affichage
  Serial.print(tau);
  Serial.print(",");
  Serial.println(C);
}
```

Entrée[ ]:
```
jupyno.televerser(croquis, port, FQBN)
```

Entrée[ ]:
```
import serial
import numpy as np

# # Acquisition
liste_tau=[] # Création de la liste, vide
liste_C=[] # Création de la liste, vide
N=10       # Nombre de points

with serial.Serial(port=port,baudrate=9600) as arduino:
    while len(liste_tau)<N:
        try:
            s=arduino.readline().decode('utf8').split(',') # Lecture
            liste_tau.append(float(s[0])) # Remplissage de la liste
            liste_C.append(float(s[1])) # Remplissage de la liste
            print(float(s[0]),'ms, ',float(s[1]),'nF') # Affichage de
        except:
            pass

#print(liste_tau)
#print(liste_C)

print('Moyenne : C=',np.mean(liste_C),'nF')
```

Élaboration d'un détecteur de passager

L'idée est de remplacer le condensateur précédent par le condensateur constitué des feuilles d'aluminium plastifiées. La capacité de celui-ci étant beaucoup plus faible que le précédent, on va remplacer la résistance par une de valeur $R = 10\ \mathrm{M\Omega}$.

> Attention il faudra modifier la valeur corespondante dans le programme Arduino.

Réaliser la modification du montage. Copier et adapter les programmes précédents pour mesurer et afficher la capacité C du circuit RC ainsi constitué, à chaque cycle de mesure.

Entrée[ ]:

Entrée[ ]:
```
jupyno.televerser(croquis, port, FQBN)
```

Entrée[ ]:

Tester le fait que **comprimer** le condensateur modifie sa capacité. Sachant que l'expression de la capacité d'un condensateur plan est donnée par :

$$C = \frac{\epsilon S}{e}$$

avec S la surface d'une feuille, e l'épaisseur de l'isolant, ϵ la permittivité diélectrique de l'isolant, vérifiez la cohérence de vos observations expérimentales.

L'objectif maintenant est d'étalonner le détecteur.

Mesurer la capacité du condensateur constitué des feuilles d'aluminium en fonction de la masse imposée sur celui-ci, puis représenter son évolution (courbe

d'étalonnage).

> Les mesures seront d'autant meilleurs que la masse est répartie sur toute la feuille, vous pouvez utiliser un carton rigide pour répartir la masse. N'hésitez pas à vous assoir dessus si vous connaissez votre masse. Vous pouvez utiliser divers objets, vous avez une balance à votre disposition pour mesurer leur masse.

Entrée[ ]:
```
m=[]
C=[]

plt.figure()

# Pour régler les paramètres de la représentation graphique (optionne
parameters = {"axes.labelsize": 13, "xtick.labelsize": 13, "ytick.lab
plt.rcParams.update(parameters)

# Tracé
plt.plot(m,C,'ro')
plt.xlabel('m (g)')
plt.ylabel('C (nF)')
plt.title('Courbe d\'évolution de la capacité C en fonction de la mas

# Pour sauvegarder l'image de la courbe (optionnel)
plt.savefig('courbe.png')

plt.show()
```

Modifier le programme Python pour qu'il affiche << Personne présente >> lorsque la masse posée dessus dépasse une certaine valeur.

Entrée[ ]:
```

```

Incertitudes de mesure

Nous allons tracer des barres d'incertitudes sur le graphe précédent, pour cela il faut estimer les incertitudes types sur :

- chaque mesure de C ;
- chaque mesure de m.

Pour la mesure de C, on dispose de la liste `liste_C` comportant les $N = 20$ dernières mesures effectuées par le module Arduino dans les mêmes conditions (on prenait la valeur moyenne ensuite).

On peut donc estimer l'incertitude avec la variabilité observée, en effet, lorsque la variabilité des mesures est accessible, il convient de répéter un grand nombre de fois le processus mesure pour estimer l'incertitude-type sur une unique réalisation de la mesure.

L'écart-type est alors assimilé à l'incertitude-type :

$$\boxed{u(x) = \sigma}$$

On peut calculer l'écart-type d'un ensemble de mesures à l'aide du langage Python :

Entrée[ ]:
```
ecarttype=np.std(liste_C)
print(ecarttype)
```

Par contre, l'incertitude-type sur la moyenne des valeurs $u(\overline{x})$ n'est pas égale à l'incertitude-type sur chaque valeur $u(x)$ (obtenue en calculant l'écart-type sur la distribution de ces valeurs).

Ainsi le résultat s'écrira :

$$\boxed{m = \overline{x} \pm u(\overline{x})}$$

Il existe une formule mathématique permettant d'estimer cet écart-type $u(\overline{x})$ pour N mesures :

$$u(\overline{x}) = \frac{u(x)}{\sqrt{N}} = \frac{\sigma}{\sqrt{N}}$$

> En toute rigueur il faudrait remplacer N par $N - 1$, sinon un biais est introduit.

Ainsi, en une série de mesure, on obtient les points expérimentaux, leur incertitude-type, la moyenne de ces points et grâce à cette formule, l'incertitude-type sur la moyenne. On obtient ainsi une estimation plus précise de la grandeur à mesurer en modérant la variabilité de chaque prise de mesure unique.

Ainsi :

Entrée[ ]:
```
uC=ecarttype/np.sqrt(len(liste_C))
print(uC)
```

Pour l'incertitude sur la masse m que vous avez imposée, elle est donnée par l'incertitude de la balance, soit $0,1$ g :

Entrée[ ]:
```
um=0.1e-3 # kg
```

On peut alors ajouter des barres d'incertitudes-type sur le graphe. Ci-dessous un exemple :

Entrée[ ]:
```
import matplotlib.pyplot as plt

# Listes des valeurs à traiter
liste_T=[92.6,79.9,71.5,65.2,60.2,55.1,51.2,48,45.5,42.8,40.3,38.3,36
liste_t=[0,5,10,15,20,25,30,35,40,45,50,55,60,65,70,90,120,150,180]

# Incertitudes sur chaque mesure
incert_x=0 # Incertitude-type sur l'abscisse
incert_y=2.5 # Incertitude-type sur l'ordonnée

# Représentation graphique
plt.figure()

# Barres d'incertitudes (attention à l'ordre (incohérent) du passage
plt.errorbar(liste_t,liste_T,incert_y,incert_x,fmt='ro',ecolor='blue'

plt.title('Représentation de T(t)')
plt.xlabel('t (s)')
plt.ylabel('T (°C)')
plt.legend(('T'))
plt.show()
```

Copier et adapter le code précédent pour afficher la courbe d'étalonnage $C = f(m)$ avec les barres d'incertitudes correspondantes.

TP filtrage numérique

PE. LEROY

> Capacités exigibles du programme :
>
> **Analyse spectrale**
>
> - Mettre en évidence le phénomène de repliement du spectre provoqué par l'échantillonnage avec un oscilloscope numérique ou une carte d'acquisition.
> - Choisir les paramètres d'une acquisition numérique destinée à une analyse spectrale afin de respecter la condition de Shannon, tout en optimisant la résolution spectrale.
>
> **Électronique numérique**
>
> - Utiliser un convertisseur analogique-numérique et un convertisseur numérique-analogique.

Liste du matériel :

- GBF
- Oscilloscope
- Module Arduino, câble USB
- Plaquette de de montage de composants
- Fils de connexion
- Boîte de résistance variable
- Boîte de capacité variable

On commence par importer les bibliothèques nécessaires :

> La ligne suivante `%matplotlib notebook` est une commande magique qui sert à l'affichage des graphes dans le notebook.

Entrée[9]:
```
%matplotlib notebook
```

Entrée[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import numpy.fft as npf
```

Nous allons travailler sur l'électronique **numérique**. Cela consiste en l'acquisition d'un signal d'entrée par un système numérique, qui réalise alors son traitement numérique, puis génère le signal de sortie.

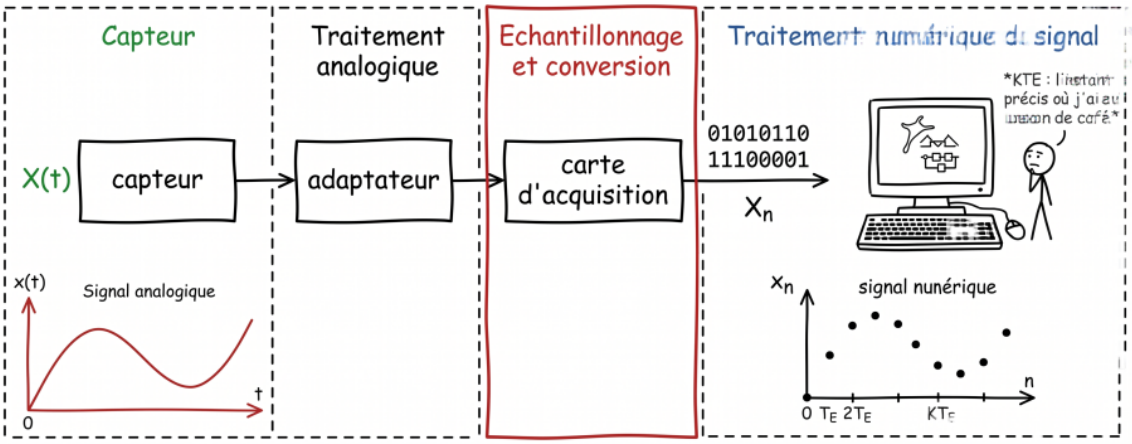
> L'intérêt est que l'on peut programmer n'importe quelle fonction assez simplement, filtres linéaires ou non, verouillage de phase, etc. Modifier la fonction à effectuer revient à mettre à jour le programme qui s'exécute sur le système numérique.

Par exemple dans ce TP nous allons réaliser l'acquisition d'une tension variable par un système Arduino, qui va réaliser un filtrage numérique, puis générer physiquement la tension de sortie que l'on pourra observer à l'oscilloscope.

La mesure d'un signal, qualifié d'**analogique**, se réalise en général à l'aide d'un capteur, auquel il faut souvent ajouter un adaptateur. Par exemple un microphone ne délivrera une tension que de l'ordre du mV, il faudra donc l'amplifier.

Une **conversion analogique-numérique** (CAN) doit ensuite être réalisée. C'est le rôle du système d'acquisition de données (interface d'acquisition, module Arduino, oscilloscope numérique, etc.).

On peut représenter l'ensemble de la chaîne d'acquisition ainsi :



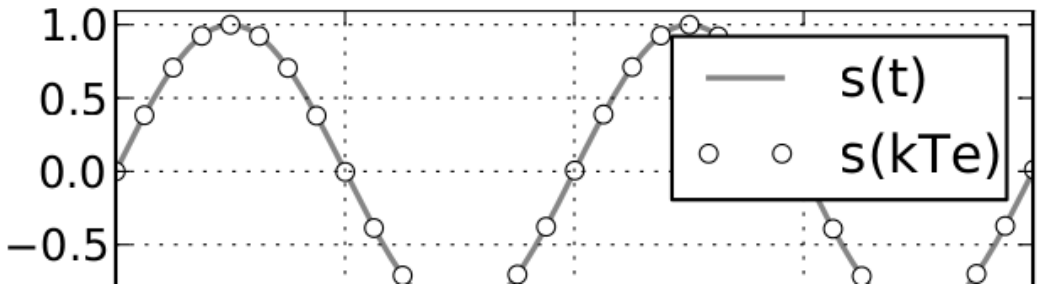
L'échantillonage d'un signal

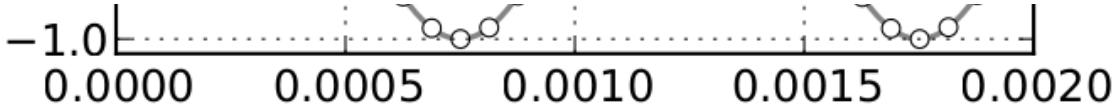
Le critère de Shannon

L'échantillonnage est l'opération qui consiste à **mesurer un signal** en capturant des valeurs **à intervalles réguliers**.

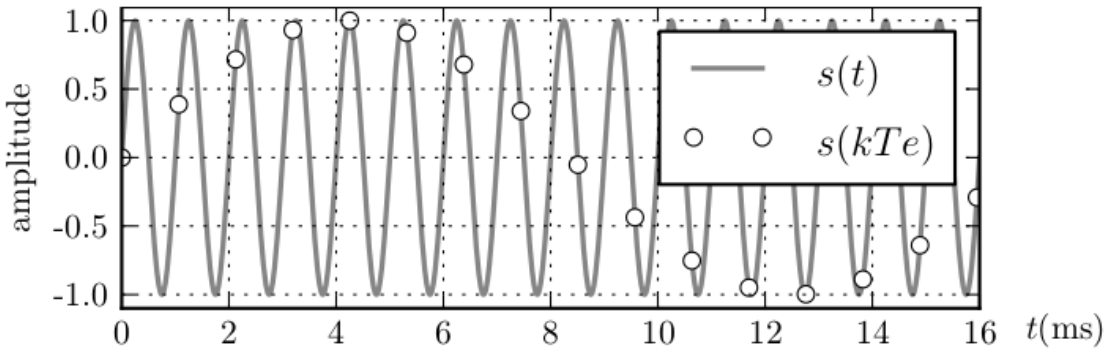
La durée entre deux mesures s'appelle la **période d'échantillonnage**, notée T_e. Se pose alors la question de savoir si les échantillons sont représentatifs du signal initial.

Si la période d'échantillonnage T_e est bien choisie, le signal obtenu est représentatif du signal initial, par exemple pour un signal sinusoïdal :





Si T_e est plus faible, cela ne changera rien. Si par contre elle est plus élevée, il peut se passer le phénomène suivant :

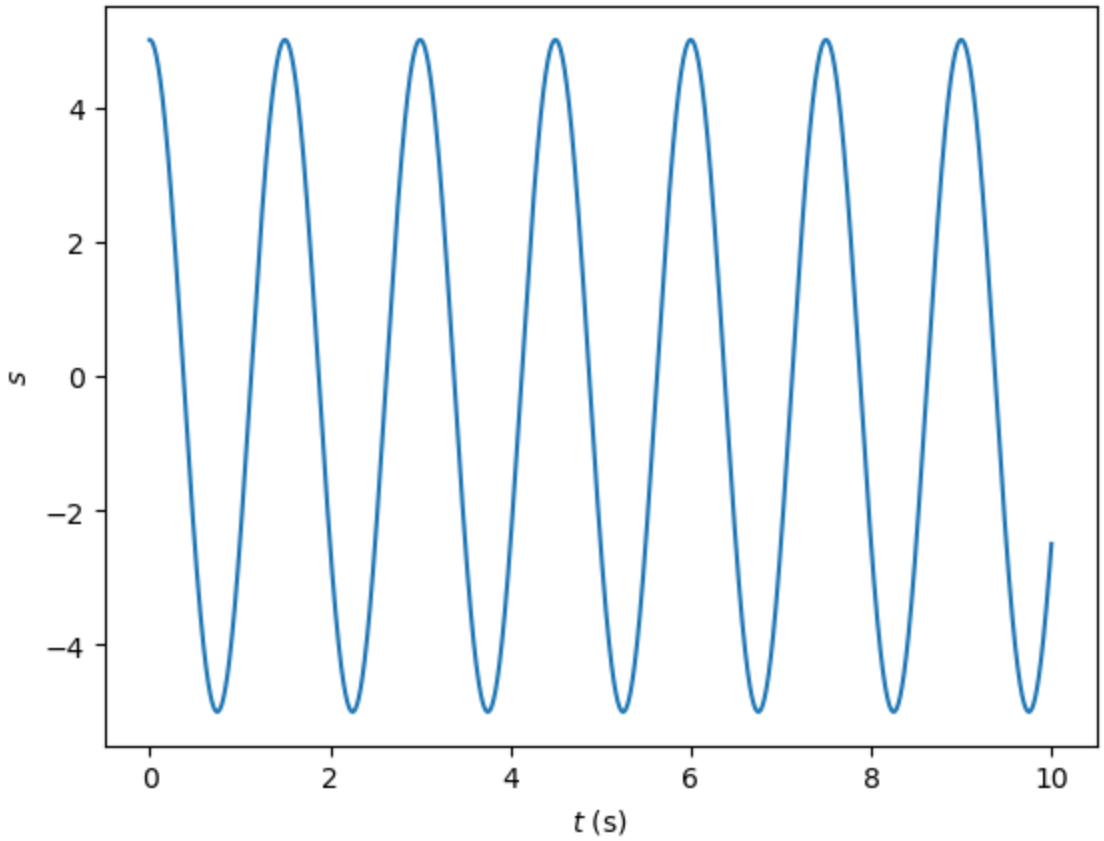


On constate que le signal obtenu n'est pas représentatif du signal initial, et de plus qu'il a une fréquence plus faible : on parle de **repliement de spectre**.

On peut simuler numériquement ce phénomène, soit un signal physique, choisissons-le sinusoïdal pour commencer :

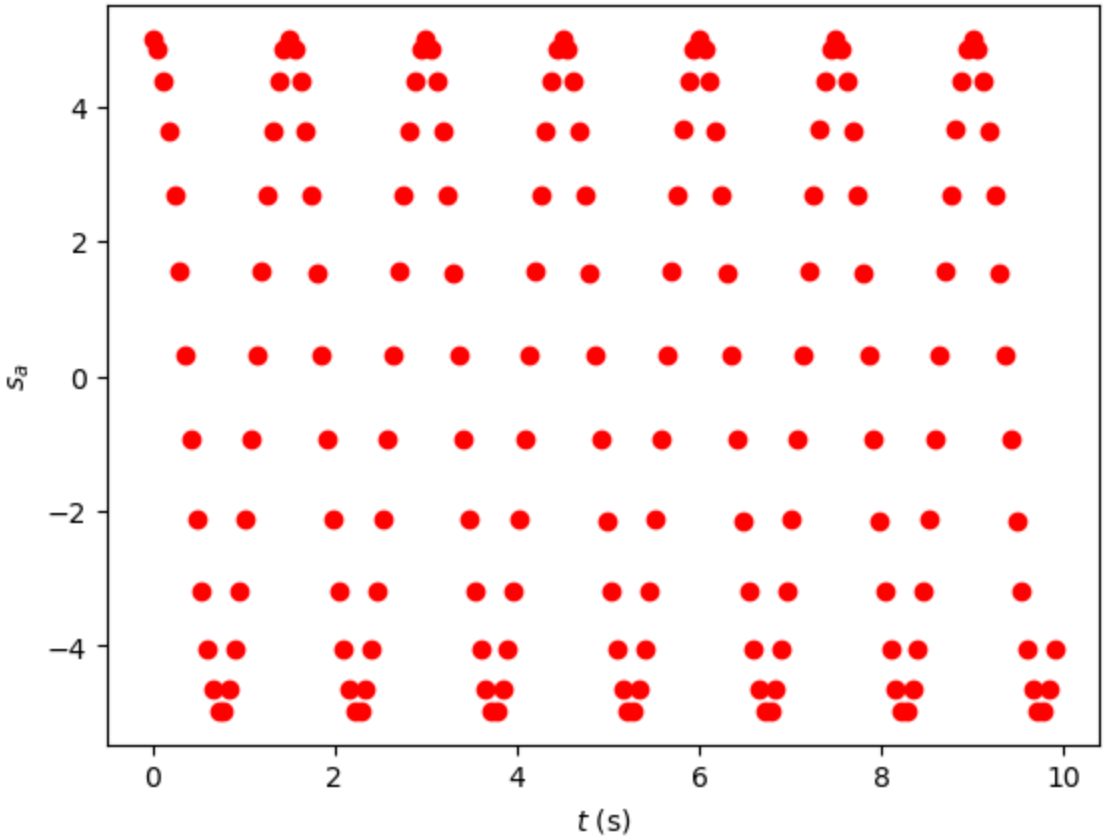
$$s(t) = A\cos\left(\frac{2\pi}{T}t\right)$$

Entrée[10]:
```
duree=10
Np=20000 # Nombre de points pour la représentation du signal physique

t=np.linspace(0,duree,Np)
# Au passage, alternative en compréhension de liste
#t=np.array([k*duree/(Np-1) for k in range(Np)])

A=5
T=1.5 #f=1/T=0,67 Hz
s=A*np.cos(2*np.pi/T*t)

plt.figure()
plt.plot(t,s)
plt.xlabel('$t$ (s)')
plt.ylabel('$s$')
plt.show()
```



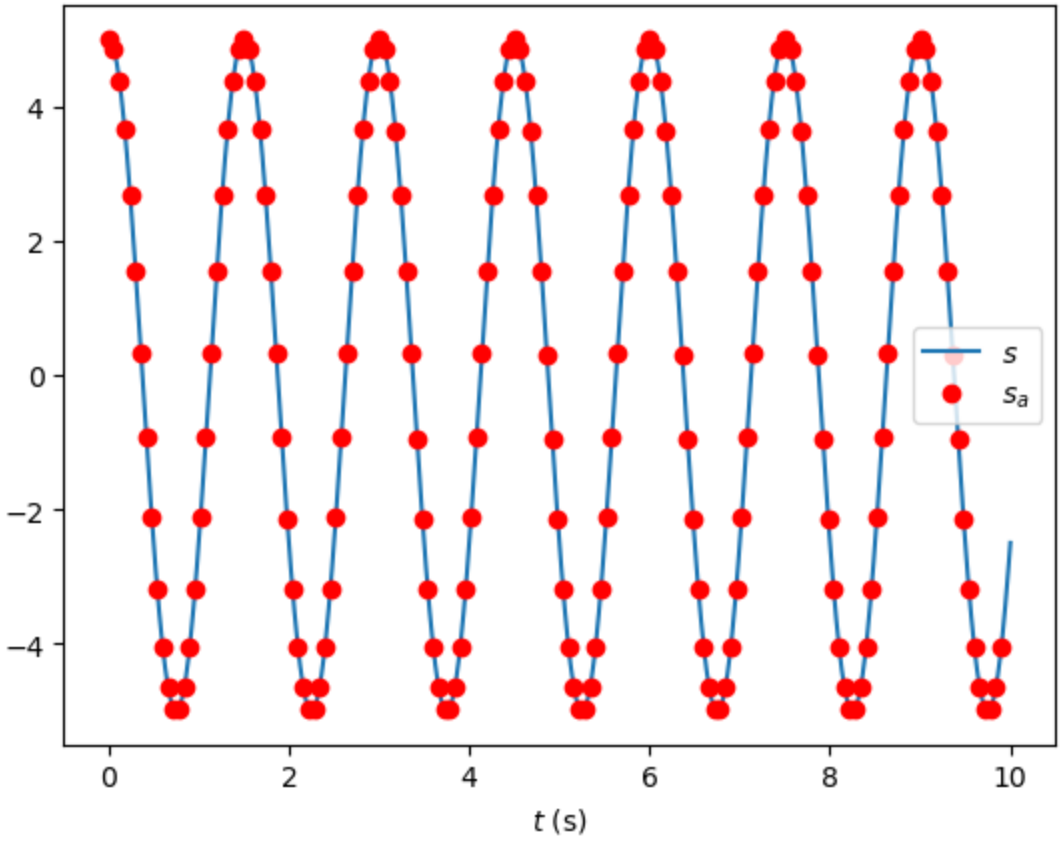
Simulons maintenant la réalisation d'une acquisition de ce signal (autrement dit sa numérisation). Nous allons prendre des valeurs de ce signal à intervalle de temps régulier, cet interval étant la periode d'échantillonnage T_e.

Entrée[11]:
```
Te=T/25
Ne=int(duree/Te) # Nombre de points de mesure
n=int(len(t)/Ne) # Nombre de points entre deux mesures

index_a=[i*n for i in range(Ne)]

ta=[t[i] for i in index_a]
sa=[s[i] for i in index_a]

plt.figure()
plt.plot(ta,sa,'ro')
plt.xlabel('$t$ (s)')
plt.ylabel('$s_a$')
plt.show()
```



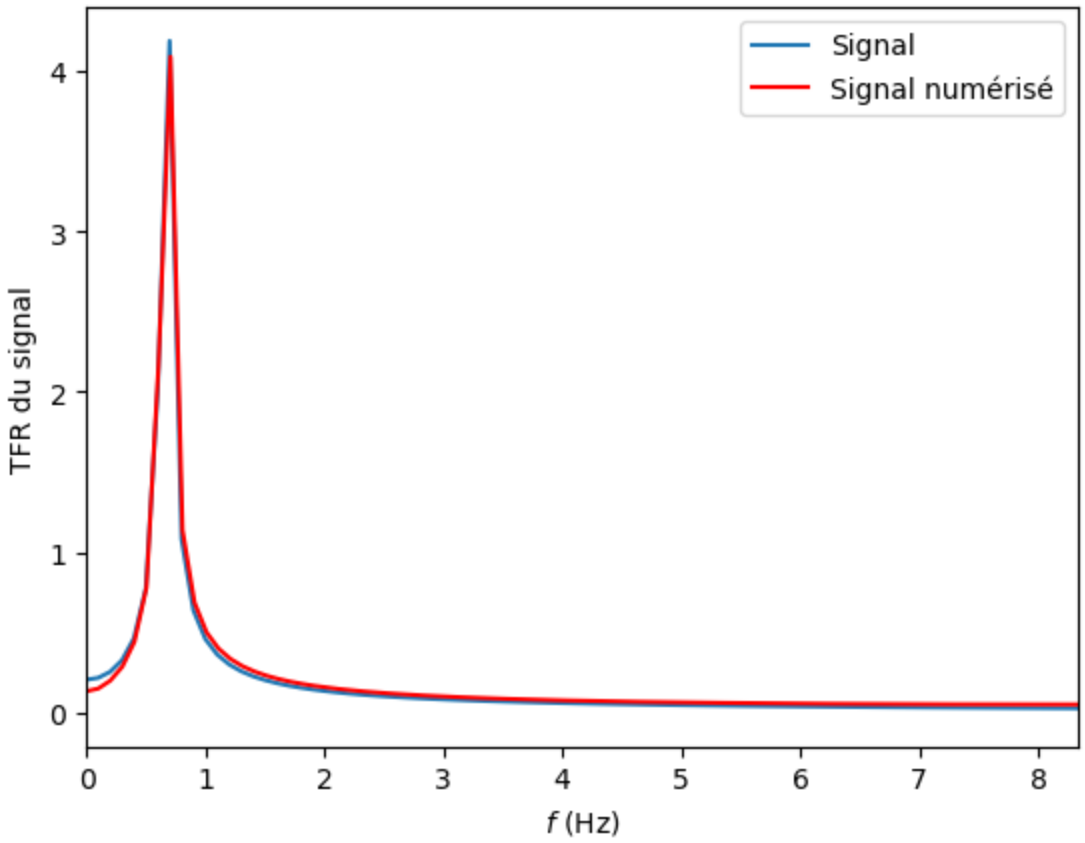
On retrouve bien les variations du signal physique à travers celles du signal numérique.
Superposons les deux signaux :

Entrée[12]:
```
plt.figure()
plt.plot(t,s)
plt.plot(ta,sa,'ro')
plt.xlabel('$t$ (s)')
plt.legend(['$s$','$s_a$'])
plt.show()
```



Traçons les spectres des deux signaux :

Entrée[13]:
```
fep=Np/duree # Fréquence d'échantillonnage du signal physique
fe=1/Te

N=len(t)
Na=len(ta)

freq_s=np.linspace(0,fep,N)
freq_sa=np.linspace(0,fe,Na)

fft_s=npf.fft(s)
fft_sa=npf.fft(sa)

plt.figure()
plt.plot(freq_s,(2/N)*np.absolute(fft_s))
plt.plot(freq_sa,(2/Na)*np.absolute(fft_sa),'r')
plt.xlim(0,fe/2)
plt.xlabel('$f$ (Hz)')
plt.ylabel('TFR du signal')
plt.legend(['Signal','Signal numérisé'])
plt.show()
```



On voit que la courbe n'est pas très lisse, c'est normal, la durée d'acquisition (et donc le nombre de périodes visibles) n'est pas très élevée : **plus grande est la durée d'acquition, meilleur est le résultat de l'algorithme FFT** (meilleure est la résolution spectrale).

> Pour la suite, on va donc augmenter la durée d'acquisition, et on limitera la plage des abscisses dans la représentation graphique à l'aide de `plt.xlim()` de façon à garder quelque-chose de visuel.

Voici ci-dessous l'exemple précédent, avec les améliorations énoncées.

Faire varier les paramètres pour se placer dans les situations suivantes :

- $T_e << T/2$;
- $T_e \simeq T/2$;
- $T_e >> T/2$.

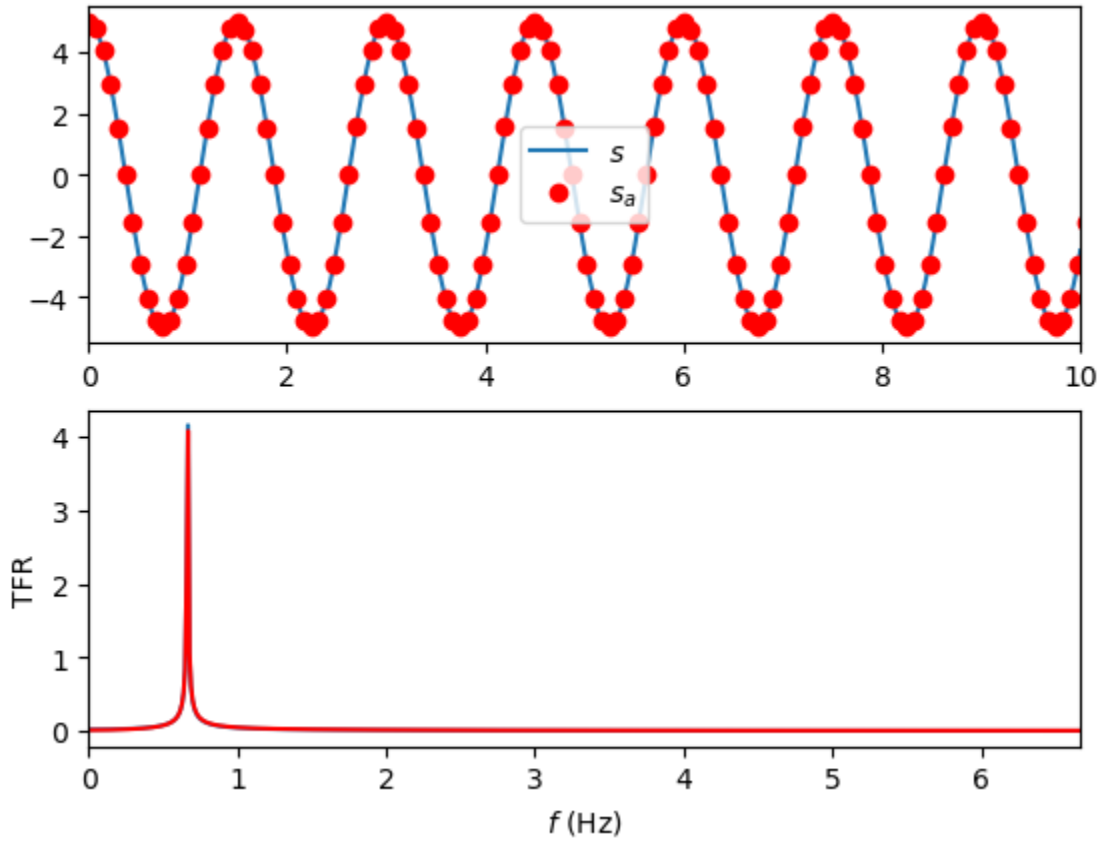
Vous pourrez ainsi visualiser :

- le **repliement de spectre** (la fréquence apparente est plus faible que la fréquence réelle) ;
- le **critère de Shannon** : pour ne pas perdre l'information sur le signal, il faut que :

$$\boxed{f_e > 2\,f}$$

.

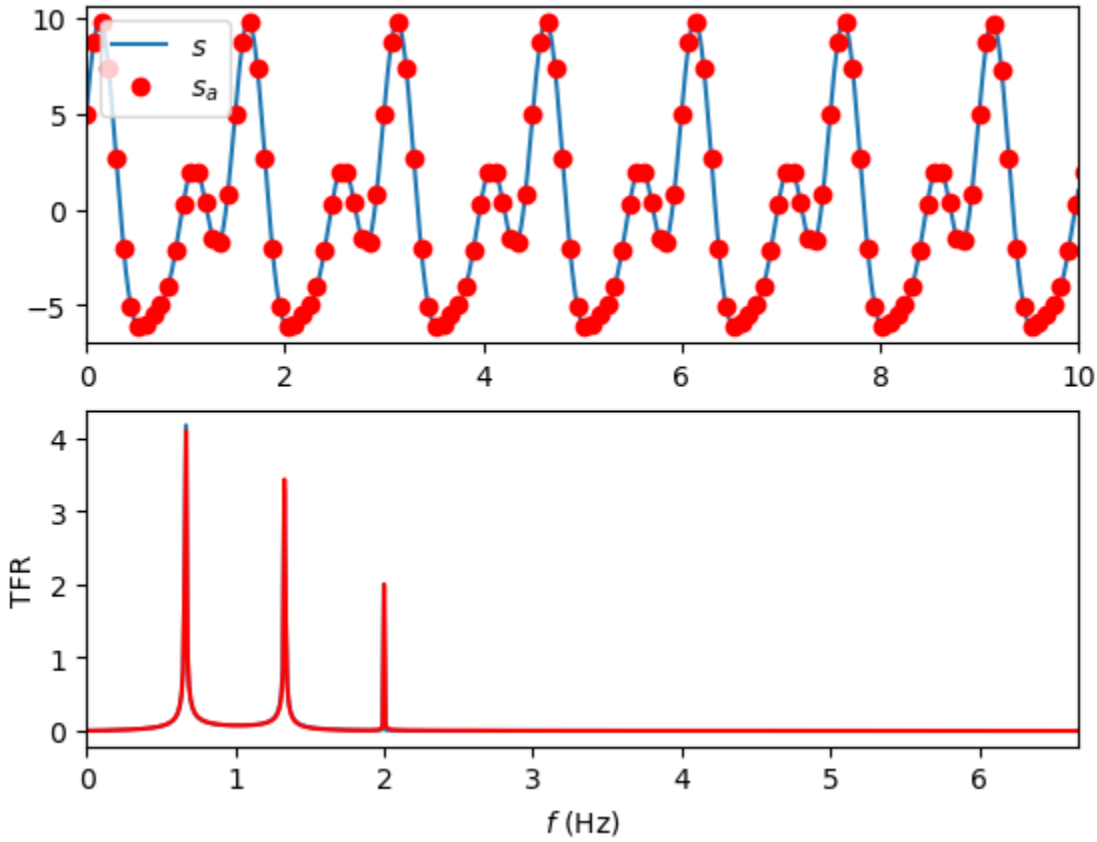
Entrée[14]:
```
duree=100
T=1.5 #f=1/T=0,67 Hz

###
# Valeur à modifier, essayer T/20, T/5 (xlim=duree/10), T/1.9, T/1.1
Te=T/20
xlim=duree/10
###

Np=20000 # Nombre de points pour la représentation du signal physique
t=np.linspace(0,duree,Np)

A=5
s=A*np.cos(2*np.pi/T*t)

Ne=int(duree/Te) # Nombre de points de mesure
n=int(len(t)/Ne) # Nombre de points entre deux mesures

index_a=[i*n for i in range(Ne)]

ta=[t[i] for i in index_a]
sa=[s[i] for i in index_a]

fep=Np/duree # Fréquence d'échantillonnage du signal physique
fe=1/Te

N=len(t)
Na=len(ta)

freq_s=np.linspace(0,fep,N)
freq_sa=np.linspace(0,fe,Na)

fft_s=npf.fft(s)
fft_sa=npf.fft(sa)

plt.figure()
plt.subplot(211)
plt.plot(t,s)
plt.plot(ta,sa,'ro')
plt.xlim(0,xlim)
plt.legend(['$s$','$s_a$'])

plt.subplot(212)
plt.plot(freq_s,(2/N)*np.absolute(fft_s))
plt.plot(freq_sa,(2/Na)*np.absolute(fft_sa),'r')
plt.xlim(0,fe/2)
plt.xlabel('$f$ (Hz)')
plt.ylabel('TFR')
plt.show()
```



Effectuer le même travail avec un signal périodique comprenant plusieurs composantes spectrales :

$$s(t) = 5 \times \cos\left(\frac{2\pi}{1,5}t\right) + 4 \times \sin\left(2 \times \frac{2\pi}{1,5}t\right) + 2 \times \sin\left(3 \times \frac{2\pi}{1,5}t\right)$$

Entrée[16]:
```
duree=100

###
# Valeur à modifier, essayer T/20, T/5 (xlim=duree/10), T/1.9, T/1.1
Te=T/20
xlim=duree/10
###

Np=20000 # Nombre de points pour la représentation du signal physique
t=np.linspace(0,duree,Np)

s=5*np.cos(2*np.pi/1.5*t)+4*np.sin(2*2*np.pi/1.5*t)+2*np.sin(3*2*np.p

Ne=int(duree/Te) # Nombre de points de mesure
n=int(len(t)/Ne) # Nombre de points entre deux mesures

index_a=[i*n for i in range(Ne)]

ta=[t[i] for i in index_a]
sa=[s[i] for i in index_a]

fep=Np/duree # Fréquence d'échantillonnage du signal physique
fe=1/Te

N=len(t)
Na=len(ta)

freq_s=np.linspace(0,fep,N)
freq_sa=np.linspace(0,fe,Na)

fft_s=npf.fft(s)
fft_sa=npf.fft(sa)

plt.figure()
plt.subplot(211)
plt.plot(t,s)
plt.plot(ta,sa,'ro')
plt.xlim(0,xlim)
plt.legend(['$s$','$s_a$'])

plt.subplot(212)
plt.plot(freq_s,(2/N)*np.absolute(fft_s))
plt.plot(freq_sa,(2/Na)*np.absolute(fft_sa),'r')
plt.xlim(0,fe/2)
plt.xlabel('$f$ (Hz)')
plt.ylabel('TFR')
plt.show()
```



Le thérème de Shannon

Ainsi, on peut généraliser le critère de Shannon à un signal périodique quelconque. C'est le **théorème de Shannon** :

Un signal est correctement représenté à partir de ses échantillons, si la fréquence d'échantillonnage f_e est supérieure à deux fois la fréquence maximale de son spectre f_{max}.

Ce théorèlme s'énonce souvent sous la forme du **critère de Shannon** :

$$\boxed{f_e > 2\, f_{max}}$$

> Un exemple d'application de ce théorème est fourni par le format des morceaux musicaux. La fréquence d'échantillonnage en HD est de 44 kHz, supérieure au double de la fréquence maximale audible par une oreille humaine, qui est de l'ordre de 20 kHz.

L'algorithme FFT

Pour tracer les spectres précédent on a utilisé un algorithme numérique appelé FFT, acronyme de l'anglais *Fast Fourier Transform*, qui permet d'avoir une représentation approchée du spectre d'un signal numérique. Cet algorithme, de complexité quasi-linéaire, est implanté par exemple dans les oscilloscopes numériques, les cartes graphiques des ordinateurs, les tableurs, et dans toutes les bibliothèques de calcul numérique.

On retiendra juste que **l'étendue du spectre calculé par l'algorithme FFT s'étale de**

0 **à** $\frac{f_e}{2}$**.**

Ainsi, si le signal dont on réalise l'acquisition possède une composante de fréquence $f > \frac{f_e}{2}$, celle-ci ne vérifie pas le critère de Shannon et elle se retrouve dans l'intervalle $[0, f_e/2[$.

> C'est bien sûr le phénomène de repliement du spectre, qui se retrouve donc dans cet intervalle lors de l'utilisation de l'algorithme FFT.

Les critères de choix pour l'acquisition d'un signal

Expérimentalement, on pourra régler trois paramètres :

- le nombre de points N ;
- la période d'échatillonnage Te ;
- la durée de l'acquisition D.

On utilisera donc la relation :

$$\boxed{T_e = \frac{D}{N}}$$

On prendra soin d'avoir :

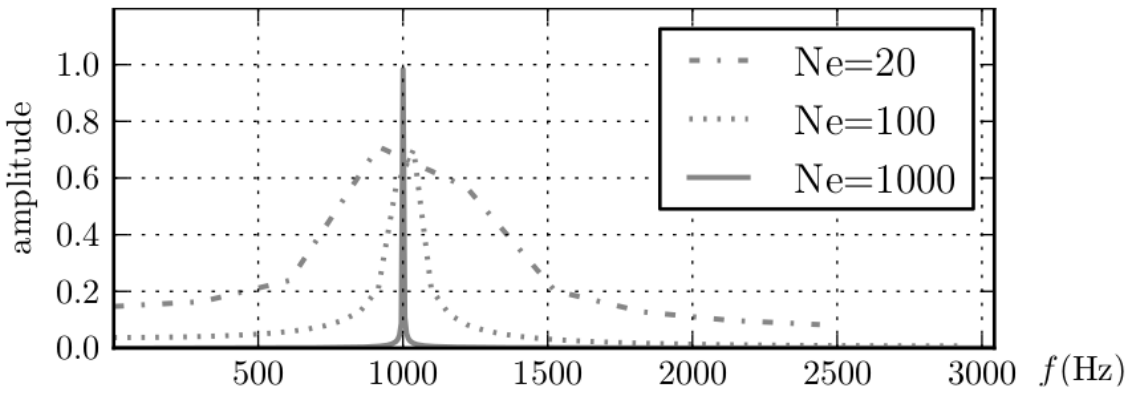
$$\boxed{T_e << T}$$

très inférieur au facteur 2 théorique minimum (condition de Shannon), souvent d'un facteur 20 suffit.

Le choix de la durée d'acquisition se fait souvent avec le nombre n de périodes que l'on souhaite voir à l'écran :

$$\boxed{D \sim nT}$$

> Ces deux paramètres vont jouer sur la **résolution spectrale** Δf, c'est à dire l'écart entre deux points sur le spectre.



On peut montrer que :

$$\Delta f = \frac{1}{D} = \frac{1}{NT_e}$$

La résolution spectrale sera d'autant meilleure que D ou N sont grands.

Le filtrage numérique

Une fois l'acquisition du signal effectuée, il faut réaliser le traitement numérique du signal. Nous allons prendre l'exemple d'un signal audio issu d'une corde de guitare, que vous pouvez écouter ici :

0:00 / 0:00

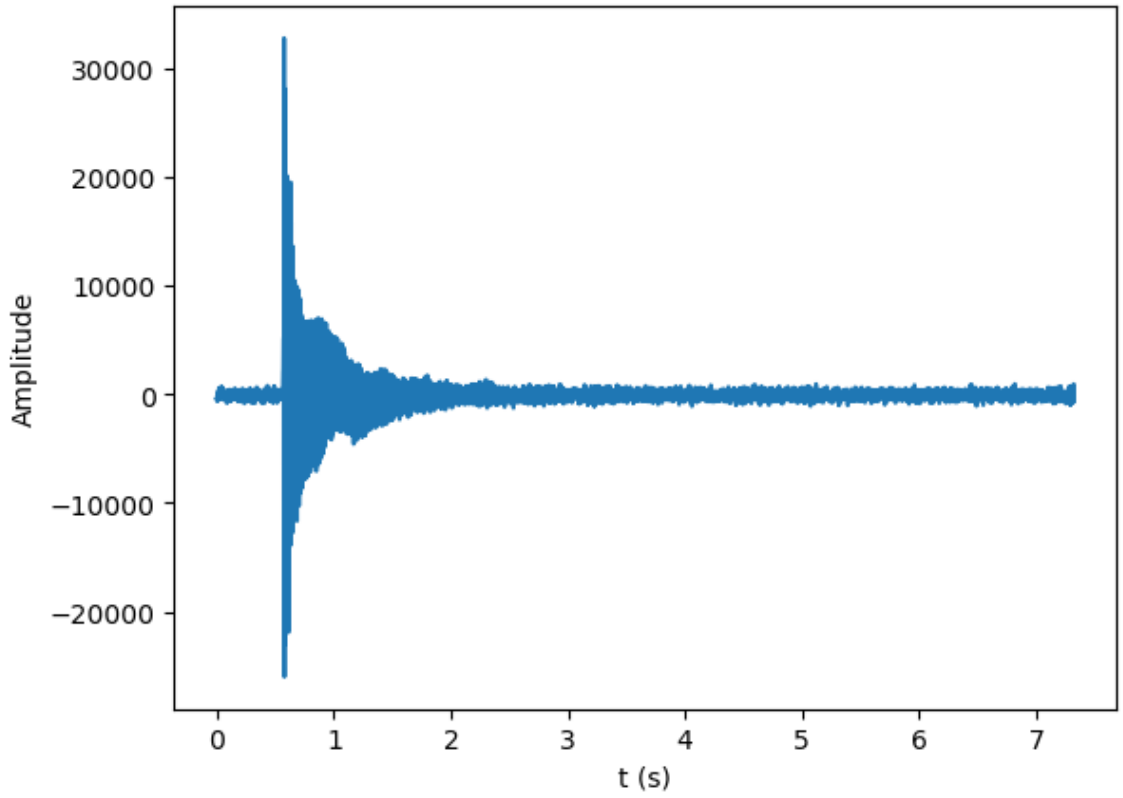
On commence par importer les bibliothèques et expliciter les noms des fichiers source, et du fichier issu du filtrage numérique :

> On utilise ici la bibliothèque `scipy` comme variante de la bibliothèque `numpy`, juste pour changer :)

Entrée[18]:
```
import scipy as sp
import scipy.fftpack as spf
import scipy.io.wavfile as spwf

# Noms des fichiers audio
fichier='corde_guitare.wav'
fichier_filtre='corde_guitare_filtre.wav'
```

Ensuite on extrait les données audio :

Entrée[20]:
```
fe=spwf.read(fichier)[0] # Récupération de la fréquence d'échantillon
N=len(spwf.read(fichier)[1]) # Recuperation du nombre de mesures
D=N/fe # Durée de l'enregistrement

liste_t=np.linspace(0,D,N)
liste_a=spwf.read(fichier)[1] # Récupération des mesures

plt.figure()
plt.plot(liste_t,liste_a)
plt.xlabel('t (s)')
plt.ylabel('Amplitude')
plt.show()
```



On va écrire une fonction permettant de réaliser un filtre passe-bande. Un filtre réalise le traitement d'un signal, souvent caractérisé par une équation différentielle. La fonction de transfert d'un filtre n'est que l'image de cette équation différentielle en notation complexe.

Prenon l'exemple d'un filtre passe-bas du premier ordre. La forme générale de la fonction de transfert est donnée par :

$$\frac{\underline{s}}{\underline{e}} = \frac{1}{1+jx}$$

avec $x = \frac{\omega}{\omega_0} = \frac{\omega}{\omega_c} = \omega\tau$

L'équation différentielle associée est donc :

$$s + \tau \frac{\mathrm{d}s}{\mathrm{d}t} = e$$

On peut << discrétiser >> cette formule de façon à utiliser la méthode d'Euler :

$$\mathrm{d}s = \frac{\mathrm{d}t}{\tau}(e - s)$$

Soit :

$$\boxed{s[n+1] = s[n] + \frac{T_e}{\tau} \times (e[n] - s[n])}$$

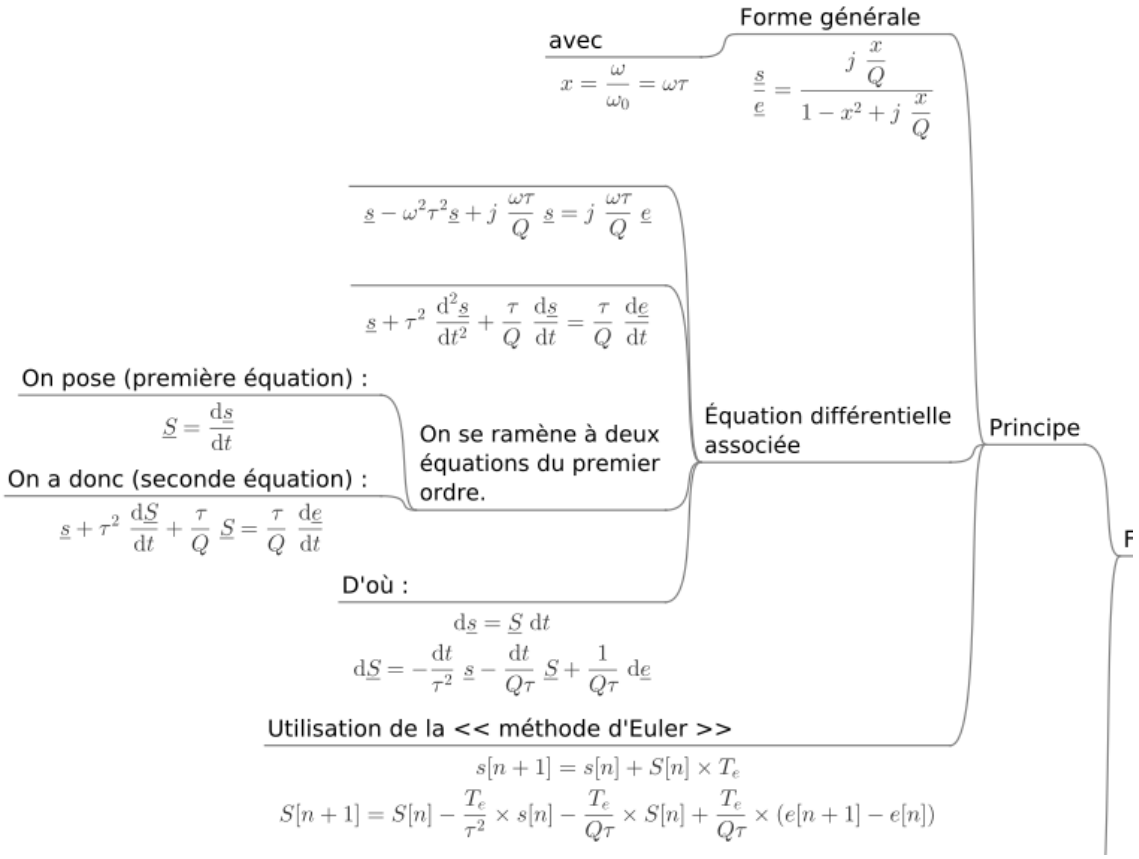
avec T_e la période d'échantillonnage.

Il est alors très simple de programmer un tel filtre en Python :

```
def filtre_passe_bas_1o(e,te,tau):
    s=[0]
    for n in range(len(e)-1):
        s.append(s[n]+te/tau*(e[n]-s[n]))
    return(s)
```

Vous pouvez trouver ici (filtres_numeriques/filtres_numeriques.pdf) une carte mentale qui donne les agorithmes correspondants aux filtres principaux.

Ainsi on trouve pour un filtre passe-bande :



Si bien qu'il peut être codé de la façon suivante :

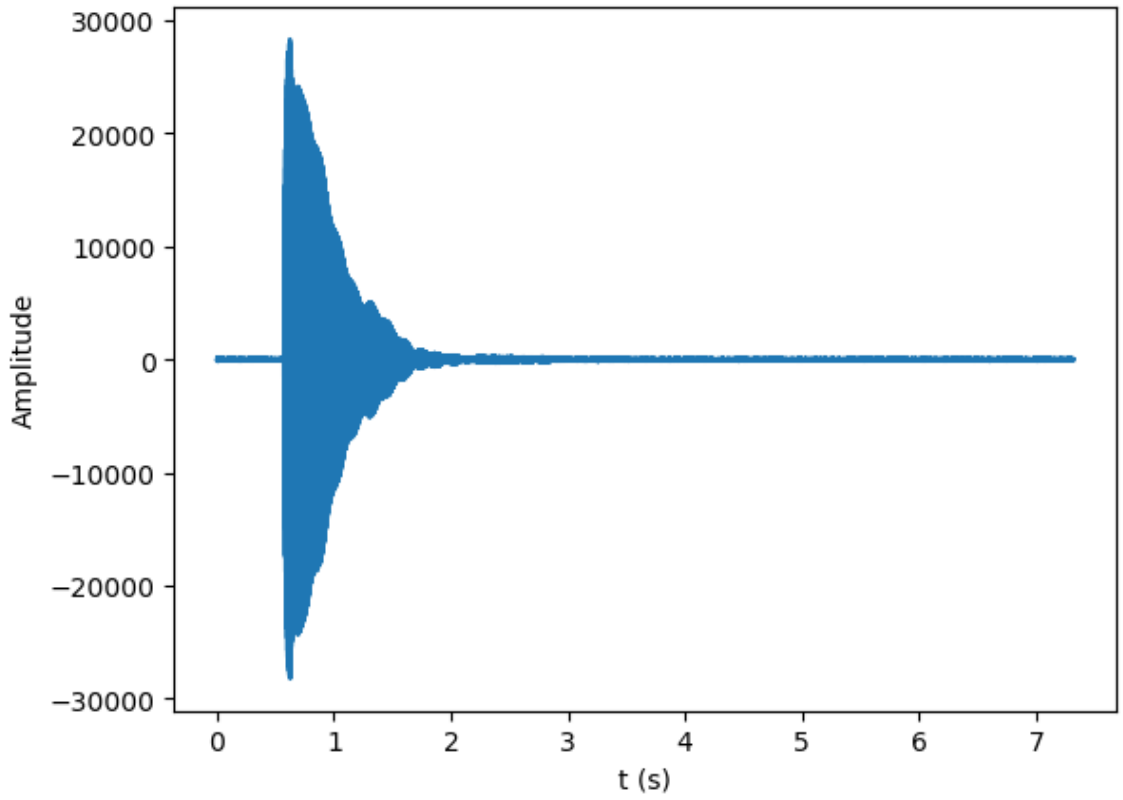
Entrée[21]:
```
def filtre_passe_bande(e,Te,w0,Q,H0):
    s=[0]
    S=[0]
    for n in range(len(e)-1):
        s.append(s[n]+S[n]*Te)
        S.append(S[n]-w0**2*Te*s[n]-w0*Te/Q*S[n]+w0/Q*H0*(e[n+1]-e[n]
    return(np.asarray(s,dtype=np.int16))
```

On applique ce filtre sur le signal, prenons un filtre passe-bande centré sur 1000 Hz :

Entrée[22]:
```
Te=1/fe # Période d'échantillonnage
f0=1000
w0=2*sp.pi*f0
Q=6
H0=1

liste_a_filtre=filtre_passe_bande(liste_a,Te,w0,Q,H0)
```

On peut visualiser le résultat :

Entrée[23]:
```
plt.figure()
plt.plot(liste_t,liste_a_filtre)
plt.xlabel('t (s)')
plt.ylabel('Amplitude')
plt.show()
```



Pour l'écouter, créons l'enregistrement sonore correspondant :

Entrée[24]:
```
spwf.write(fichier_filtre,fe,liste_a_filtre)
```

On peut écouter le résultat :

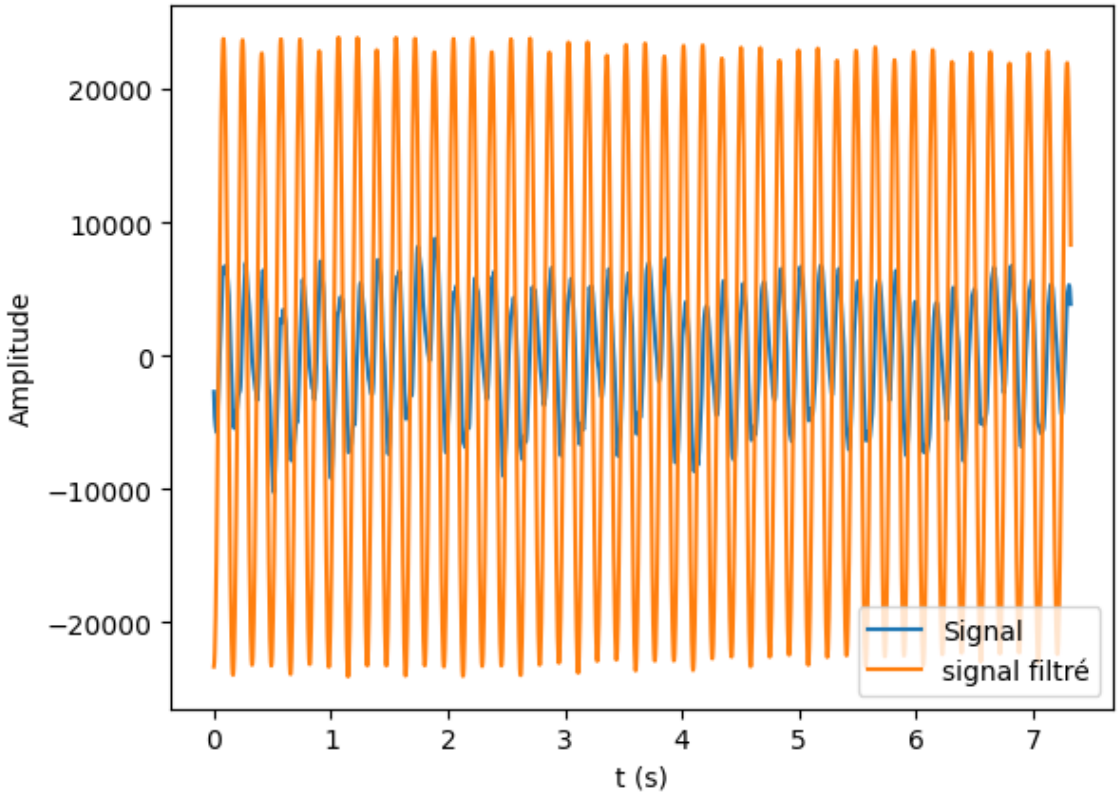
0:00 / 0:00

Pour mieux visualiser les effets, travaillons sur les spectres. Pour tracer le spectre de ces signaux, il faut en isoler une partie. Sur les graphes on peut voir qu'autour de $t_i = 0,7$ s le signal semble d'amplitude correcte :

Entrée[26]:
```
Ni=int(0.7*fe) # Indice correspondant à ti=0,7 s
Nf=Ni+2000 # Extraction de 2000 points autour de ti

liste_ae=liste_a[Ni:Nf]
liste_aef=liste_a_filtre[Ni:Nf]
liste_te=np.linspace(0,D,Nf-Ni)
```

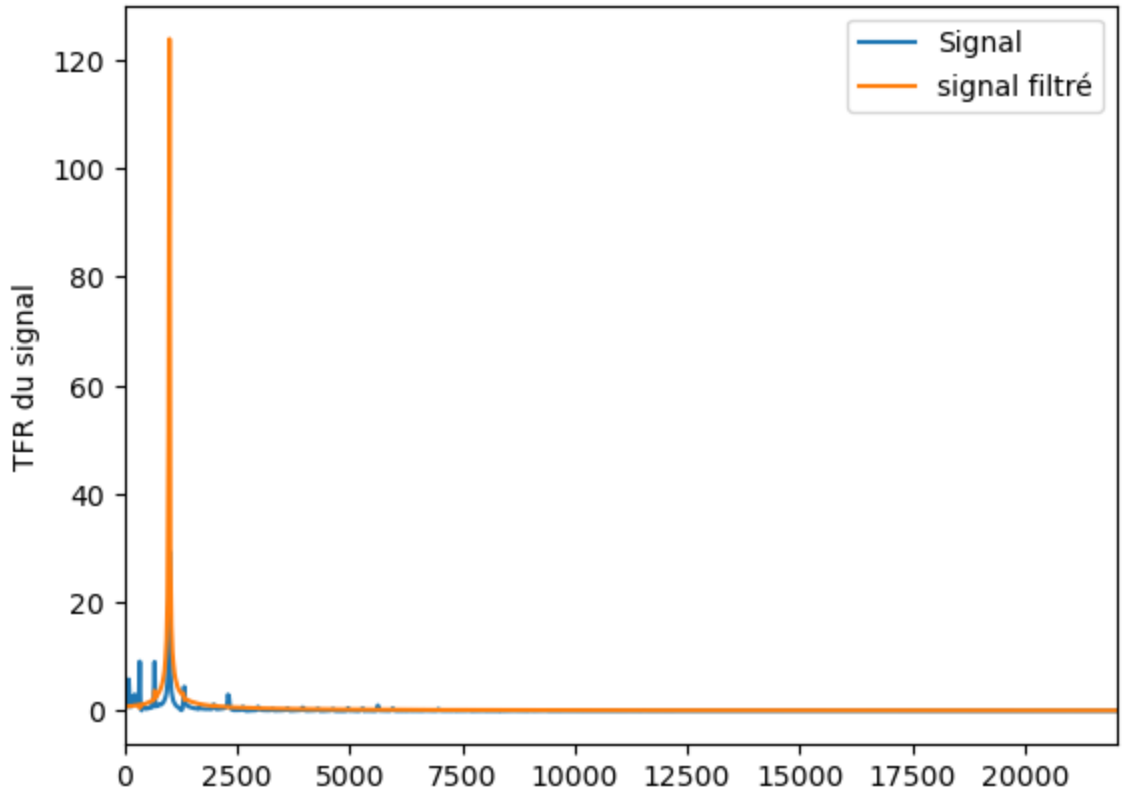
On peut vérifier ce que cela donne :

Entrée[27]:
```
plt.figure()
plt.plot(liste_te,liste_ae)
plt.plot(liste_te,liste_aef)
plt.xlabel('t (s)')
plt.ylabel('Amplitude')
plt.legend(['Signal','signal filtré'])
plt.show()
```



On peut déjà noter que le signal est presque sinusoïdal, ce qui correspondrait bien au résultat de l'application d'un filtre passe-bande. Traçons les spectres de ces signaux :

Entrée[29]:
```
liste_freq=np.linspace(0,fe,Nf-Ni)
liste_fft=spf.fft(liste_ae) # Application de l'algorithme FFT
liste_fft_filtre=spf.fft(liste_aef) # Application de l'algorithme FFT

plt.figure()
plt.plot(liste_freq,(2/N)*np.absolute(liste_fft))
plt.plot(liste_freq,(2/N)*np.absolute(liste_fft_filtre))
plt.xlim(0,fe/2)
plt.ylabel('TFR du signal')
plt.legend(['Signal','signal filtré'])
plt.show()
```



On voit bien l'effet, sélectionner la composante autour de 1000 Hz !

La génération d'un signal de sortie

On va prendre un exemple complet de traitement numérique d'un signal électrique :

- conversion analogique-numérique ou CAN (échantillonnage du signal d'entrée) ;
- traitement numérique ;
- conversion numérique-analogique ou CNA (génération du signal de sortie).

Le traitement numérique du signal est en général confié à des composants dédiés à cette tâche, qui sont souvent des **microcontrôleurs**.

Les deux autres étapes nécessitent deux composants :

- un convertisseur analogique-numérique (CAN) en entrée ;
- un convertisseur numérique-analogique (CNA) en sortie.

Fonctionnement de la carte Arduino

CAN

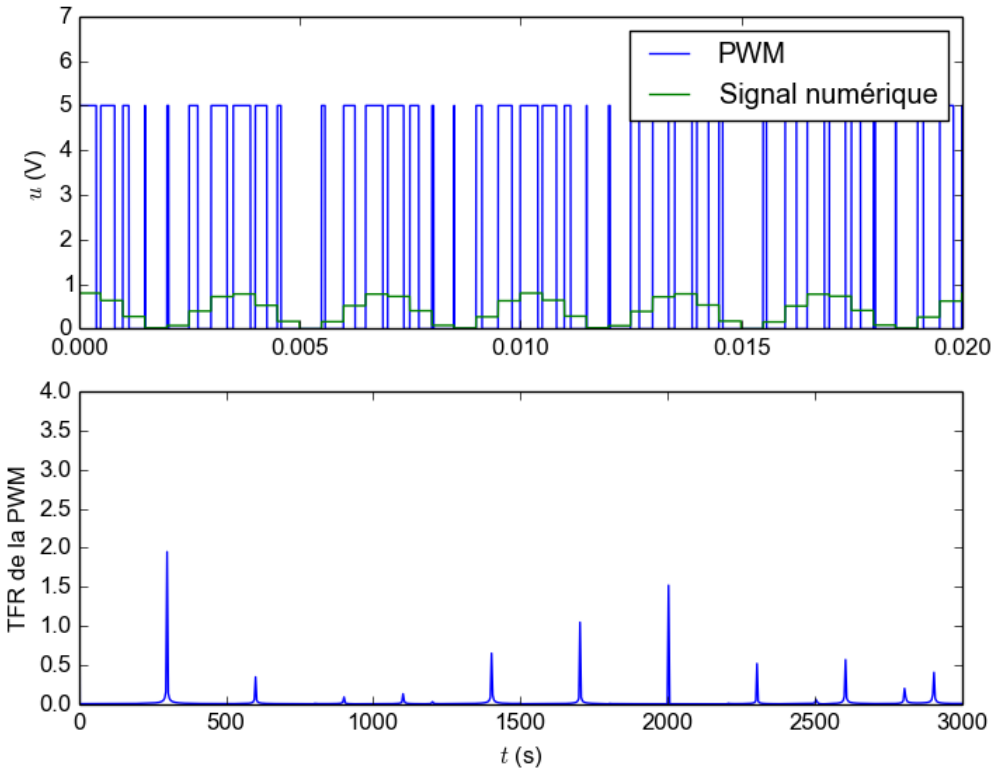
La carte Arduino UNO possède des entrées analogiques qui permettent la conversion analogique-numérique.

Ces entrée analogiques fonctionnent entre 0 et 5 V uniquement : **le signal d'entrée devra toujours être situé dans cet intervalle.**

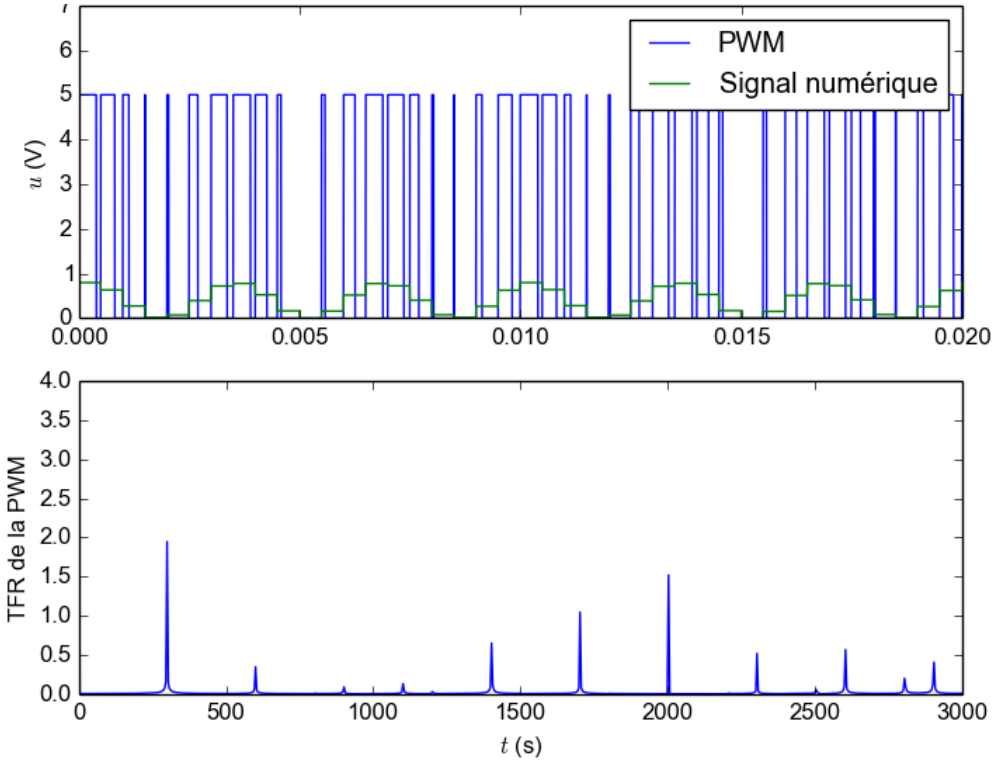
> C'est très important et souvent oublié, vous devrez régler au préalable, **à l'oscilloscope**, le signal d'entrée pour qu'il évolue entre 0 et 5 V, soit :
>
> - régler l'amplitude à moins de 2,5 V, ou l'amplitude crête à crête à moins de 5 V ;
> - introduire une tension de décalage (offset) au moins égale à l'amplitude.

CNA

La carte possède également des sorties qui permettent la conversion numérique-analogique, mais sous une forme un peu particulière. Le signal de sortie se présente sous la forme d'un signal créneau, dont la **largeur de chaque créneau est modulée par le signal**. Ainsi, un signal sinusoïdal sera par exemple représenté par le signal :



On parle de **Modulation de Largeur d'Impulsion** ou **MLI** en français ou PWM en anglais (Pulse Width Modulation). On peut montrer que le spectre de ce signal comporte une composante basse fréquence correspondant au signal à transmettre :



Il est donc nécessaire d'ajouter un filtre passe-bas en sortie de la carte, de fréquence de coupure supérieure à la fréquence du signal et très inférieure à celle du signal créneau de la PWM :

$$\boxed{f_s < f_c << f_{\text{PWM}}}$$

> C'est une limitation de la carte, plus précisément du modèle UNO, les modèle plus élaborés possèdent un véritable CNA. C'est important de comprendre que l'on palie à un manque techniologique avec ce filtre passe-bas en sortie, cela n'a rien à voir avec le traitement numérique que l'on va effectuer à l'aide du microcontrôleur.

L'ajout d'un filtre RC en sortie suffit, les valeurs de R et de C (pour f_c) sont à ajuster :

$$f_c = \frac{1}{2\pi RC}$$

La fréquence de la PWM sera de $f_{\text{PWM}} = 31$ kHz. On peut choisir une fréquence de coupure 10 fois plus faible (une décade), ce qui devrait générer une atténuation d'environ 20 dB (pente de -20 dB/décade).

Entrée[ ]:
```
R=10e3 # Ohm
# Fréquence de la PWM
fPWM=31e3 # Hz
# Fréquence de coupure choisie
fc=fPWM/10
# Calcul de C
C=
```

Traitement numérique à l'aide du microcontrôleur

Première chose, le résultat de la CAN (en entrée) sera donnée par la fonction `analogRead()` qui retourne un nombre entier N compris entre `0` et `255`. Ce

nombre correspond à la fraction de la tension de référence 5 V qui correspond à la tension en entrée, soit :

$$e(t) = \frac{N}{1023} \times 5\ V$$

Ce qui se traduira par le code :

```
e=analogRead(entree)*5.0/1023 ;
```

> Il faut bien indiquer `5.0` pour la valeur de la tension de 5 V, pour forcer la division flottante au lieu de la division entière.

Il en est de même pour la CNA (en sortie) avec la fonction `analogWrite()` :

$$s(t) = \frac{N}{1023} \times 5\ V$$

Soit le code :

```
s_num=s/5.0*1023 ;
```

> Attention le signal de sortie devra lui-même toujours être positif, c'est ce à quoi vous devrez penser si vous détectez des << saturations >> en sortie.

Ensuite il faut programmer le microcontrôleur pour réaliser le traitement numérique. On va commencer par un traitement tout simple, un filtre passe-bas.

Ce filtre de période d'échantillonnage $Te = 0,1$ ms et de fréquence de coupure $f_c = 100$ Hz pourra ainsi être codé de la façon suivante :

```
s=s+0.0628*(e-s) ;
```

En effet ce filtre se code de la façon suivante (cf. carte mentale ici (filtres_numeriques/filtres_numeriques.pdf)) :

Utilisation de la << méthode d'Euler >>

$$s[n+1] = s[n] + \frac{T_e}{\tau} \times (e[n] - s[n])$$

Or :

$$\frac{T_e}{\tau} = Te \cdot \omega_c = Te \cdot 2\pi f_c = 0,0628$$

On y va ? Commencez par le montage :

- **régler à l'aide d'un oscilloscope le signal d'entrée issu du GBF pour qu'il ait une amplitude crête à crête inférieure à 5 V et un offset d'au moins la valeur de l'amplitude, typiquement $2,5$ V d'amplitude et 1 V d'offset. Prendre une fréquence basse comme 10 Hz ;**
- **connecter ce signal d'entrée sur l'entrée `A0` du module Ardiono ;**
- **connecter la sortie `~9` du module Arduino à un filtre passe-bas RC nécessaire pour traiter la PWM en sortie, puis mesurer la tension en sortie du filtre à l'oscilloscope.**

> Je le répète, ce filtre RC compense un manquement au niveau de la conception de la carte Arduino !
>
> > Enfin surtout si on veut que cette carte soit proposée à un prix faible, une alternative à un vrai CNA a été trouvée...

On y va pour la programmation du microcontrôleur, comme dans les TP précédents :

Entrée[ ]:
```
!pip --user install jupyno
```

Entrée[ ]:
```
import jupyno
```

Entrée[ ]:
```
port, FQBN = jupyno.trouver_arduino()
```

On commence par créer un fichier vide (<< croquis >>) :

Entrée[ ]:
```
croquis = jupyno.creer_croquis('FiltrageNumerique')
```

On va utiliser le programme suivant, essayez de comprendre sa structure puis exécutez la cellule :

Entrée[ ]:
```
%%ecrire_croquis $croquis

// broches
int entree = A0; // la broche d'entree
int sortie = 9; // la broche de sortie

// grandeurs
float e; // le signal d'entree
float s=0; // le signal de sortie
int s_num; // le signal numerique de sortie

void setup()
{
  TCCR1B = 1; // initialisation de la PWM a 31  kHz sur les broches 9
}

void loop()
{
  e=analogRead(entree)*5.0/1023;
  s=s+0.0628*(e-s); // le fitre passe bas, fc=100 Hz, Te/tau=Te*wc=0,
  s_num=s/5.0*1023;
  analogWrite(sortie,s_num);
  delay(0.1); // Te=0,1 ms (fe=10 kHz)
}
```

Il reste à le compiler et le téléverser sur le module Arduino :

Entrée[ ]:
```
jupyno.televerser(croquis, port, FQBN)
```

Si vous avez bien manipulé, vous devriez voir le signal de sortie qui correspond au filtrage passe-bas du signal d'entrée.

> Si des effets de << saturation >> sont observés, c'est que les tensions en entrée ou en sortie ne sont pas dans l'intervalle $[0, 5]$ V.

Augmenter doucement la fréquence du signale d'entrée et vérifier le comportement passe-bas du filtre numérique ainsi réalisé.

On est bien d'accord : pas grand intérêt à priori de faire un truc aussi compliqué pour un filtrage passe-bas d'ordre 1 qui se réalise très simplement avec un filtre RC. **Mais l'avantage ici est que la fonction (ici un filtrage) est programmé à l'aide d'un script : on peut programmer n'importe quelle fonction, la faire évoluer de façon dynamique, par exemple en fonction des caractéristiques du signel d'entrée, etc.** Les possibilités sont en fait énormes !

Modifiez le programme pour réaliser maintenant un filtre numérique passe-haut d'ordre 1.

> Et voilà, on change la fonction en changeant la programmation du microconrôleur !

Modifiez le programme pour réaliser maintenant une fonction de votre choix (dérivation, intégration, etc.).

Type *Markdown* and LaTeX: α^2

TP mesure d'une enthalpie de réaction

PE. LEROY

> Capacités exigibles du programme :
>
> **Bilans d'énergie**
>
> - Mettre en oeuvre une technique de calorimétrie.
> - Déterminer la valeur en eau d'un calorimètre.
> - Estimer les fuites thermiques lors d'expériences réalisées avec un calorimètre.

Liste du matériel :

- Calorimètre
- Module Arduino, câcle USB
- Capteur de température étanche Grove DS18B20
- Thermomètre classique (à alcool)
- Éprouvette graduée de 500 mL
- Erlenmeyer de 500 mL
- Balance (précision $\sim 0,1$ g)
- Coupelle de pesée
- Eau chaude (bouilloire)
- Nitrate de potassium (20 g par groupe)

On commence par importer les bibliothèques nécessaires :

> La ligne suivante `%matplotlib notebook` est une commande magique qui sert à l'affichage des graphes dans le notebook.

Entrée[ ]:
```
%matplotlib notebook
```

Entrée[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
```

L'objectif est ici de mesurer l'enthalpie standard de dissolution du nitrate de potassium $\mathrm{KNO_3}$ dans l'eau. Pour cela nous allons suivre l'évolution de la température lors de la dissolution dans un calorimètre.

La réction de dissolution est simplement :

$$\mathrm{KNO_3}(s) = \mathrm{K^+}(aq) + \mathrm{NO_3^-}(aq)$$

Calculer l'enthalpie standard de réaction $\Delta_r H^\circ$ correspondante.

On donne : $\Delta_f H^\circ(\mathrm{KNO_3(s)}) = -492 \text{ kJ. mol}^{-1}$,
$\Delta_f H^\circ(\mathrm{K^+(aq)}) = -251,31 \text{ kJ. mol}^{-1}$, $\Delta_f H^\circ(\mathrm{NO_3^-(aq)}) = -204,45 \text{ kJ. mol}^{-1}$

Entrée[ ]:
```
DrHo=
print(DrHo)
```

C'est donc cette valeur que l'on cherche à mesurer expérimentalement.

Pour cela on réalise la dissolution dans un calorimètre, on est donc dans des conditions qui permettent de modéliser la transformation comme une transformation adiabatique monobare. Pour rappel, on décompose artificiellement la transformation en deux étapes :

- transformation isotherme monobare où l'énergie $\xi_f \Delta_r H^\circ$ est dégagée ;
- utilisation de cette énergie pour faire évoluer la température des constituants qui restent (donc **après réaction**). Si bien que l'on peut écrire :
$$\Delta H_{\text{réaction}} = -\Delta H_{\text{changement de T après réaction}}$$
ou :
$$\boxed{\xi_f \cdot \Delta_r H^\circ = -\sum_i C^\circ_{p,i} \, (T_f - T_0)}$$
s'il n'y a pas de changement d'état pendant l'évolution de la température. T_0 et T_f sont les températures finale et initiale des constituants.

Il y a une subtilité avec l'utilisation d'un calorimètre, en effet les parties internes réagissent thermiquement aussi, c'est à dire qu'une partie de l'énergie mise en jeu dans la réaction va servir à faire varier la température des parties internes du calorimètre.

Il faut donc prendre en compte la capacité thermique des parties internes. Cette donnée est fournie par le constructeur, mais il est préférable de la remesurer.

Mesure de la valeur en eau du calorimètre

Pour cela, on cherche ce que l'on appelle la **masse en eau** μ_{eau} (en kg) du calorimètre. Il s'agit de la masse d'eau qui réagirait thermiquement de la même façon que les parties internes du calorimètre, soit :
$$C^\circ_{p,\text{calorimètre}} = \mu_{eau} \cdot c^\circ_{p,eau}$$

Il s'agit donc de mesurer μ_{eau}. Pour cela voici un protocole :

- mettre une masse d'eau $m = 200$ g à température ambiante T_{amb} dans le calorimètre ;
- ajouter la même masse d'eau $m = 200$ g chaude à la température T_{ch}.
- attendre en homogénéisant doucement et mesurer la température finale T_f à l'aide du thermomètre à alcool.

Ainsi dans cette expérience de calorimétrie on peut appliquer le premier principe appliqué pour une évolution monobare :
$$\Delta H = 0 = m \cdot c^\circ_{p,eau}(T_f - T_{amb}) + m \cdot c^\circ_{p,eau}(T_f - T_{ch}) + \mu_{eau} \cdot c^\circ_{p,eau}(T_f - T_{amb})$$

> On voit l'intérêt de prendre une masse d'eau équivalente : $c_{p,eau}$ se simplifie dans le calcul.

Réaliser le protocole et déterminer la mase en eau du calorimètre.

Entrée[ ]:
```
m=200e-3 # kg
Tamb= # °C
Tc= # °C
Tf= # °C
# Calcul
mu_eau=
print(mu_eau)
```

Nous allons profiter d'avoir des calorimètres identiques pour chaque paillasse de TP, pour effectuer un calcul d'incertitude sur la valeur en eau du calorimètre utilisé.

Indiquer au tableau de la salle votre valeur et recopier la liste des valeurs des différents groupes :

Entrée[ ]:
```
liste_mue=[]
```

On commence par établir une valeur moyenne :

Entrée[ ]:
```
mue=np.mean(liste_mue)
```

On peut donc estimer l'incertitude avec la variabilité observée, en effet, lorsque la variabilité des mesures est accessible, il convient de répéter un grand nombre de fois le processus mesure pour estimer l'incertitude-type sur une unique réalisation de la mesure.

L'écart-type est alors assimilé à l'incertitude-type :

$$\boxed{u(x) = \sigma}$$

On peut calculer l'écart-type d'un ensemble de mesures à l'aide du langage Python :

Entrée[ ]:
```
ecarttype=np.std(liste_mue)
print(ecarttype)
```

Par contre, l'incertitude-type sur la moyenne des valeurs $u(\overline{x})$ n'est pas égale à l'incertitude-type sur chaque valeur $u(x)$ (obtenue en calculant l'écart-type sur la distribution de ces valeurs).

Ainsi le résultat s'écrira :

$$\boxed{m = \overline{x} \pm u(\overline{x})}$$

Il existe une formule mathématique permettant d'estimer cet écart-type $u(\overline{x})$ pour N mesures :

$$u(\overline{x}) = \frac{u(x)}{\sqrt{N}} = \frac{\sigma}{\sqrt{N}}$$

> En toute rigueur il faudrait remplacer N par $N - 1$, sinon un biais est introduit.

Ainsi, en une série de mesure, on obtient les points expérimentaux, leur incertitude-type, la moyenne de ces points et grâce à cette formule, l'incertitude-type sur la

moyenne. On obtient ainsi une estimation plus précise de la grandeur à mesurer en modérant la variabilité de chaque prise de mesure unique.

Ainsi :

Entrée[ ]:
```
umue=ecarttype/np.sqrt(len(liste_mue))
print(umue)
```

Réalisation de l'expérience

Le protocole est le suivant (attention à l'ordre) :

- mettre une masse d'eau $m = 400$ g d'eau à température ambiante dans le calorimètre ;
- placer le thermomètre relié au module Arduino dans le calorimètre ;
- peser 20 g de poudre de nitrate de potassium $\mathrm{KNO_3}$;
- lancer l'acquisition puis attendre 20 s ;

> Cela nous servira pour déterminer précisément la température initiale

- verser le nitrate de potassium dans l'eau et refermer rapidement le calorimètre.

Comme indiqué dans le protocole le suivi de température se fera à l'aide d'un module Arduino, et nous allons interagir avec le module Arduino directement à partir de ce notebook.

On commence par installer une bibliothèque spécifique et faire des tests de communication :

Entrée[ ]:
```
!pip --user install jupyno
```

Entrée[ ]:
```
import jupyno
```

Entrée[ ]:
```
port, FQBN = jupyno.trouver_arduino()
```

Le capteur de température utilisés ici possèdent un circuit intégré `DS18B20` . Celui-ci nécessite des bibliothèques qui ne sont pas incluses par défaut dans l'IDE Arduino. Il faut donc les importer. Ces bibliothèques ont été récupérées sur le site du constructeur, ce sont les fichiers `OneWire.zip` et `DallasTemperature.zip` . On commence par les importer :

> Répondre `O` (oui) aux questions posées.

Entrée[ ]:
```
jupyno.installer_librairie('OneWire.zip')
jupyno.installer_librairie('DallasTemperature.zip')
```

On commence par créer un fichier vide (<< croquis >>) :

Entrée[ ]:
```
croquis = jupyno.creer_croquis('SuiviTemperature')
```

Nous allons écrire le programme suivant (notez la commande magique au début, qui sert ainsi à écrire le programme dans le << croquis >> directement au sein du notebook), essayez de comprendre sa structure puis exécutez la cellule :

Entrée[ ]:
```
%%ecrire_croquis $croquis

/**
 * Mesures de températures avec un seul capteur DS18B20 GROVE
 */

/* Inclure les bibliothèques */
#include <OneWire.h>
#include <DallasTemperature.h>

/* Création de l'objet OneWire et sensors */
OneWire capteur(2);                  // Création d'un objet One Wire
DallasTemperature sensors(&capteur); // Indiquer la référence oneWire

/* Pour faire des mesures à intervalle régulier */
unsigned long tempsPrecedent = 0;
float intervalle = 1000; // Durée entre deux mesures en ms (valeur mi

void setup(){
  Serial.begin(115200);  // Choix de la vitesse
  sensors.begin();
}

void loop(){
  unsigned long tempsCourant = millis();  // Cette variable contient

  if (tempsCourant-tempsPrecedent>=intervalle){
      tempsPrecedent=tempsCourant;

      /* Demande la température aux capteurs */
      sensors.requestTemperatures();
      float temperature=sensors.getTempCByIndex(0);  // Obtenir la te

      /* Envoie le temps et la température sur la liaison série */
      Serial.print(tempsCourant);
      Serial.print(',');
      Serial.println(temperature);

  }
}
```

Il reste à le compiler et le téléverser sur le module Arduino :

Entrée[ ]:
```
jupyno.televerser(croquis, port, FQBN)
```

Côté notebook Jupyter nous allons récupérer en direct les données qui sont envoyées sur le port série par le module Arduino. Pour cela il est nécessaire d'installer une librairie Python pour réaliser la communication sur le port série (décommenter la ligne suivante pour l'installation) :

Entrée[ ]:
```
#!pip install --user pyserial
```

Si l'acquisition est lancée dans le notebook et que vous souhaitez l'annuler, il faut interrompre le noyau et cela a pour conséquence de devoir relancer toute la chaine d'instructions précédente. Pour éviter cela nous allons créer la possibilité d'interrompre la boucle `while` par appui de la touche `q` pour quitter. Même procédé :

> Au moment où j'écris ces lignes je ne suis pas sûr du bon fonctionnement sur les machines du labo, je vous laisse tester en décommentant les lignes correspondantes.

Entrée[ ]:
```
#!pip install --user keyboard
```

On y va ! Essayez de comprendre la structure du programme de récupération des données sur le port série (port de communication) puis exécutez la cellule :

Entrée[ ]:
```
import serial
#import keyboard
#import time

# # Acquisition
liste_t=[] # Création de la liste, vide
liste_T=[] # Création de la liste, vide
N=20       # Nombre de points

#print('Taper q pour quitter !')

with serial.Serial(port=port,baudrate=115200) as arduino:
    while len(liste_T)<N:
        #if keyboard.is_pressed('q'):
        #    break
        try:
            s=arduino.readline().decode('utf8').split(',')    # Lectu
            liste_t.append(int(s[0])) # Remplissage de la liste des t
            liste_T.append(float(s[1])) # Remplissage de la liste des
            print(int(s[0]),'ms',float(s[1]),'°C') # Affichage des va
        except:
            pass
        #time.sleep(0.1) # Pour ne pas surcharger le processeur du fa
```

Cela fonctionne ? Essayez de prendre le capteur au creux de votre main pour voir l'évolution de la température.

Le nombre de points N sera à adapter en fonction de la durée de l'expérience, sachant que l'on peut régler dans le programme Arduino la durée entre deux acquisitions (`intervalle` en ms).

On peut représenter la courbe :

Entrée[ ]:
```
import matplotlib.pyplot as plt

plt.figure()

# Pour régler les paramètres de la représentation graphique (optionne
parameters = {"axes.labelsize": 13, "xtick.labelsize": 13, "ytick.lab
plt.rcParams.update(parameters)

plt.plot(liste_t,liste_T,'ro')

plt.xlabel('t (ms)')
plt.ylabel('T (°C)')
plt.title('Courbe d\'évolution de la température')

# Pour sauvegarder l'image de la courbe (optionnel)
plt.savefig('courbe.png')

plt.show()
```

Régler le nombre de points pour réaliser une acquisition sur 5 min. Appliquer le protocole indiqué précédemment.

On peut aussi sauvegarder les données dans un fichier `CSV` :

Entrée[ ]:
```
# Sauvegarde des données dans un fichier extérieur (optionnel)
nomfichier='donnees.csv'

with open(nomfichier,'w') as fichier:
    fichier.write('t (s)'+';'+'T (°C)'+'\n') # Ecriture de la premièr
    for i in range(len(liste_T)):
        fichier.write(str(liste_t[i]).replace('.',',')+';'+str(liste_
```

Et l'importation des données depuis le fichier `CSV` se ferait de la facon suivante :

Entrée[ ]:
```
# Importation des données depuis un fichier extérieur (optionnel)
nomfichier='donnees.csv'

liste_t,liste_T=np.char.replace(np.loadtxt(nomfichier,skiprows=1,unpa
```

On va pouvoir déterminer les températures initiale et finale du mélange. Pour la température initiale T_0, comme le protocole demandait d'attendre 20 s avant d'ajouter la poudre, on peut faire une moyenne de ces valeurs.

Mesurer la température initiale T_0 du mélange dans le calorimètre :

Entrée[ ]:
```
# À compléter
#T0= np.mean(...) # °C
print(T0)
```

Pour la température finale T_f il s'agit de la température minimale atteinte (vous devez constater que la température augmente doucement au bout d'un certain temps, ce sont les fuites thermiques).

Mesurer la température finale T_f du mélange dans le calorimètre :

Entrée[ ]:
```
# Température finale
Tf= # °C
```

Estimons les incertitudes associées.

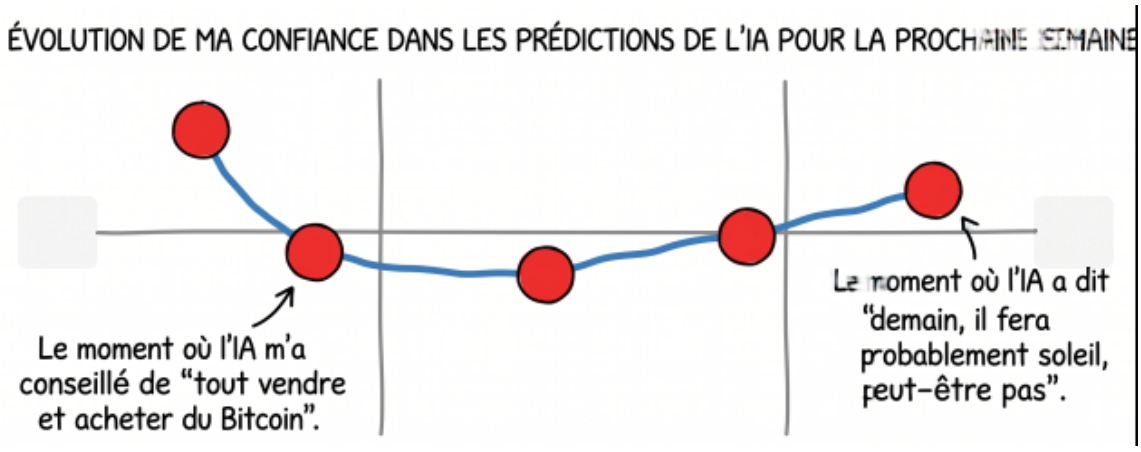
Pour T_0, comme on a accès à un ensemble de valeur, on peut estimer son incertitude avec la variabilité apparente comme précédemment :

Compléter les lignes suivantes avec la partie de la liste des températures avant ajout de la poudre :

Entrée[ ]:
```
# À compléter
#ecarttype=np.std(...)
#uT0=ecarttype/np.sqrt(len(...))  # °C
print(uT0)
```

Pour T_f, il s'agit principalement d'une **incertitude de pointé**, dans le sens où l'on ne connait pas exactement la température minimale puisque les températures sont relevées toutes les 2 s. Il faudra donc estimer cette incertitude en extrapolant la forme de la courbe.

> L'incertitude due au capteur est de l'ordre de $0,5$°C, à voir si nous pouvons la négliger devant l'incertitude de pointé, sinon utiliser la formule de composition des incertitudes, cf. le TP sur l'étude d'un transfert conductoconvectif où nous l'avons utilisée pour l'incertitude sur le temps.



D'après la forme de la courbe (ne pas hésiter pas à zoomer sur le graphe), estimer l'incertitude sur la température finale T_f :

Entrée[ ]:
```
uTf= # °C
print(uTf)
```

Analyse des résultats

La relation vue au début du TP s'écrit donc :

$$\xi_f \cdot \Delta_r H^\circ = -\sum_i C_{p,i}^\circ \, (T_f - T_0)$$

soit :

$$\xi_f \cdot \Delta_r H^\circ = -\left(C^\circ_{p,solution}\,(T_f - T_0) + \mu_{eau} \cdot c^\circ_{p,eau}\,(T_f - T_0)\right)$$

On prendra :

$$C^\circ_{p,solution} \simeq m_{solution} \cdot c^\circ_{p,eau}$$

Un tableau d'avancement donne immédiatement :

$$\xi_f = n = \frac{m}{M}$$

si bien que :

$$\boxed{\Delta_r H^\circ \simeq -\frac{(m_{solution} + \mu_{eau}) \cdot c^\circ_{p,eau}\,(T_f - T_0) \cdot M}{m}}$$

avec m la masse de poudre, M la masse molaire du nitrate de potassium $\mathrm{KNO_3}$, $m_{solution}$ la masse initiale d'eau plus la masse de poudre.

On donne : $M(\mathrm{N}) = 14\ \mathrm{g.mol^{-1}}$, $M(\mathrm{O}) = 16\ \mathrm{g.mol^{-1}}$, $M(\mathrm{K}) = 39\ \mathrm{g.mol^{-1}}$

Déterminer la valeur expérimentale de l'enthalpie standard de dissolution du nitrate de potassium $\Delta_r H^\circ$:

Entrée[ ]:
```
m_solution=0.400+0.030 # kg
m=50 # g
M=39+14+16*3 # g/mol
cp_eau=4180 # J/K/kg

# Valeurs déjà déterminées
print(mue)
print(T0,Tf)

# Calcul de la valeur mesurée
DrHom=

# Comparaison
print(DrHo,DrHom)
```

Pour comparer les deux valeurs correctement, il nous faut faire un calcul d'écart normalisé ou **z-score** : $$ E_N=\dfrac{\left|m_2-m_1\right|}{\sqrt{u(m_1)^2+u(m_2)^2}} $$

Par convention, on qualifie deux résultats de **compatibles** si leur écart normalisé vérifie la propriété : $$ \boxed{E_N \lesssim 2} $$

On pourra considérer que la valeur calculée à l'aide de la loi de Hess aura une incertitude relative égale à la décimale de son plus petit chiffre significatif, soit $0,1 \ \mathrm{kJ.mol^{-1}}$.

Entrée[1]:
```
uDrHo=0.1e3 # J/mol
```

Il nous manque donc juste l'incertitude sur la valeur mesurée. On pourrait profiter du fait que chaque groupe a déterminé une valeur de $\Delta_r H^{°}$ et en étudier la variabilité, mais on va en profiter plutôt pour faire une estimation à partir de l'incertitude sur chaque grandeur qui compose le calcul de $\Delta_r H^{°}$ (incertitudes composées).

On repart de : $$ \Delta_r H^{°} \simeq -\dfrac{(m_{solution}+\mu_{eau}) \cdot c_{p,eau}^° \ (T_f-T_0) \cdot M}{m} $$

Les grandeurs dont l'incertitude a une forte influence (est significative) sont la masse en eau du calorimètre μ_{eau} et les températures finale et initiale T_f et T_0. On considèrera donc que l'incertitude sur $\Delta_r H^{°}$ est essentiellemnt due à celle de ces trois paramètres.

Comme la formule fait intervenir des sommes et des produits, on va partir comme cela est recommandé sur un calcul d'incertitudes composées à l'aide de la méthode de **Monte-Carlo**.

> Pour rappel la méthode est décrite dans ce notebook (../incertitudes/incertitudes_de_mesure.ipynb).

Le principe est de réaliser un tirage aléatoire de valeurs autour de la valeur mesurée, en faisant varier chaque paramètre selon une loi de distribution aléatoire définie. On détermine ensuite l'incertitude-type de la distribution de valeurs obtenues.

Ci-dessous l'exemple présenté dans le notebook sur les incertitudes :

Entrée[ ]:
```
import random as rd

####### Partie à adapter ########
x1=1.10
x2=2.12

ux1=0.01
ux2=0.1

def x(x1,x2): # Définition de la loi permettant de déterminer x
    return 2*x1**3+6*x2

grandeur=x(x1,x2) # Calcul de m

def variations_aleatoires(): # Variation aléatoire de x (méthode Mont
    return x(x1+loi_normale(ux1),
             x2+loi_rectangulaire(ux2))

####### Partie à ne pas modifier ########
def loi_normale(sigma) : # Loi de distribution normale
    return np.random.normal(0,sigma)

def loi_rectangulaire(sigma): # Loi de distribution rectangulaire
    return rd.uniform(-sigma,sigma)

distribution=[] # Distribution des valeurs
N=100000 # Nombre de points utilisés dans la méthode Monte-Carlo

for i in range(N):
    distribution.append(variations_aleatoires())

incertitude_type=np.std(distribution) # Calcul de l'incertitude-type

print('x=',grandeur,'u(x)=',incertitude_type)

plt.figure()
plt.hist(distribution,bins=500)
plt.xlabel('Grandeur $x$')
plt.ylabel('Nombre de tirages')
plt.show()
```

Vérifier que le programme suivant est correctement écrit de façon à déterminer l'incertitude sur la valeur mesurée de $\Delta_r H^{°}$.

> $\Delta_r H^{°}$ sera donc la grandeur x, et μ_{eau}, T_f et T_0 seront les grandeurs x_1, x_2 et x_3.

Entrée[ ]:
```
import random as rd

####### Partie à adapter ########
x1=mue
x2=Tf
x3=T0

ux1=umue
ux2=uTf
ux3=uT0

def x(x1,x2,x3): # Définition de la loi permettant de déterminer x
    res=-(m_solution+x1)*cp_eau*(x2-x3)*M/m
    return res

grandeur=x(x1,x2,x3) # Calcul de m

def variations_aleatoires(): # Variation aléatoire de x (méthode Mont
    return x(x1+loi_normale(ux1),
             x2+loi_normale(ux2),
             x3+loi_normale(ux3))

####### Partie à ne pas modifier ########
def loi_normale(sigma) : # Loi de distribution normale
    return np.random.normal(0,sigma)

def loi_rectangulaire(sigma): # Loi de distribution rectangulaire
    return rd.uniform(-sigma,sigma)

distribution=[] # Distribution des valeurs
N=100000 # Nombre de points utilisés dans la méthode Monte-Carlo

for i in range(N):
    distribution.append(variations_aleatoires())

incertitude_type=np.std(distribution) # Calcul de l'incertitude-type

print('x=',grandeur,'u(x)=',incertitude_type)

plt.figure()
plt.hist(distribution,bins=500)
plt.xlabel('Grandeur $x$')
plt.ylabel('Nombre de tirages')
plt.show()
```

Entrée[ ]:
```
uDrHom=incertitude_type
```

Calculer l'écart normalisé entre les valeurs théorique et mesurée de $\Delta_r H^{°}$, puis conclure.

Entrée[ ]:
```
# Rappel des noms de variables nécessaires
print(DrHo,uDrHo,DrHom,uDrHom)
# Calcul
EN=
print(EN)
```

Type *Markdown* and LaTeX: α^2

TP mesure d'une constante d'équilibre

PE. LEROY

> Capacités exigibles du programme :
>
> **Mesures électriques**
>
> - Mettre en oeuvre des mesures électriques dans un environnement électrochimique.

Liste du matériel :

> en `encadré` uniquement la semaine 1, en **gras** uniquement la semaine 2) :

- Solution saturée d'iodate de baryum $\mathrm{Ba(IO_3)_2}$ préparée par mélange de 500 mL de solution de nitrate de baryum $\mathrm{Ba(NO_3)_2}$ à $1 \times 10^{-2}\ \mathrm{mol.\,L^{-1}}$ et de 500 mL de solution de iodate de potassium $\mathrm{KIO_3}$ à $1 \times 10^{-2}\ \mathrm{mol.\,L^{-1}}$ (agitation 10 à 15 min, puis repos pendant une demi-heure et filtration pendant la séance)
- `Solution d'iodure de potassium` KI à $1 \times 10^{-1}\ \mathrm{mol.\,L^{-1}}$ (50 mL par groupe)
- **Solution de sulfate de sodium** $\mathrm{Na_2SO_4}$ à $2 \times 10^{-2}\ \mathrm{mol.\,L^{-1}}$ (100 mL par groupe)
- `Solution de thiosulfate de sodium` $\mathrm{Na_2S_2O_3}$ à $1 \times 10^{-2}\ \mathrm{mol.\,L^{-1}}$ (100 mL par groupe)
- `Solution d'acide sulfurique` $\mathrm{H_2SO_4}$ à $1 \times 10^{-1}\ \mathrm{mol.\,L^{-1}}$ (50 mL par groupe)
- `Iotect` (ou à défaut empois d'amidon)
- Eau distillée
- Burette graduée de 25 mL
- Éprouvette graduée de 100 mL ($\times 1$)
- Béchers de 250 mL ($\times 2$) et 100 mL ($\times 3$)
- Pipettes jaugées de `20` mL et **50** mL + propipette
- Agitation magnétique
- **Conductimètre (classique, à écran) + notice**

On commence par importer les bibliothèques nécessaires :

> La ligne suivante `%matplotlib notebook` est une commande magique qui sert à l'affichage des graphes dans le notebook.

Entrée[ ]:
```
%matplotlib notebook
```

Entrée[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Principe de la détermination du produit de solubilité K_s de l'iodate de baryum $\mathrm{Ba(IO_3)_2}$

L'objectif de ces deux séances est de mesurer le produit de solubilité de l'iodate de baryum, c'est à dire que si un équilibre existe entre le solide et ses ions en solution, on peut écrire cet équilibre sous la forme :

$$\mathrm{Ba(IO_3)_2}(s) = \mathrm{Ba^{2+}}(aq) + 2\ \mathrm{IO_3^-}(aq)$$

> Par exemple, dans une solution saturée, tout le solide n'arrive pas à se dissoudre : des grains sont toujours présents, en équilibre avec ses ions dissous dans la solution.

La constante d'équilibre s'écrit :

$$K^\circ(T) = \frac{[\mathrm{Ba^{2+}}]_{eq}[\mathrm{IO_3^-}]_{eq}^2}{C^{\circ 3}} = K_s$$

et c'est par définition ce que l'on appelle le **produit de solubilité**, qui ne dépend comme toute constante d'équilibre que de la température.

Pour déterminer K_s nous allons :

- déterminer $[\mathrm{IO_3^-}]_{eq}$ par **titrage indirect** (par déplacement) dans la première séance ;
- déterminer $[\mathrm{Ba^{2+}}]_{eq}$ par **titrage conductimétrique** dans la seconde séance.

> Doser une espèce chimique, c'est déterminer sa concentration. Titrer une espèce c'est la doser, mais à l'aide d'une réaction chimique.
>
> Un réaction de titrage doit être :
> - totale ;
> - rapide ;
> - spécifique de l'espèce à titrer.

Il faudra **filtrer** la solution saturée d'iodate de baryum $\mathrm{Ba(IO_3)_2}$ (retirer le solide présent) pour récupérer une solution qui contient les espèces dissoutes en proportions correspondant à l'équilibre, mais sans que cet équilibre ne puisse opérer. Sans cela, toute disparition d'une espèce lors de son titrage engendrerait une reformation de cet espèces à partir du solide.

> C'est un exemple de déplacement d'équilibre.

Dosage des ions iodate $\mathrm{IO_3^-}$

Nous allons effectuer un titrage indirect par déplacement, voici les étapes :

- ajout en excès d'ions iodures $\mathrm{I^-}$ en milieu acide ;

Montrer (au brouillon) que l'équation de la réaction entre les ions iodate et iodure est la suivante (couples redox mis en jeu $\mathrm{IO_3^-/I_2}$ et $\mathrm{I_2/I^-}$) :

$$\mathrm{IO_3^-}(aq) + 5\mathrm{I^-}(aq) + 6\ \mathrm{H^+}(aq) = 3\ \mathrm{I_2}(aq) + 3\ \mathrm{H_2O}(\ell)$$

- tirage du diiode $\mathrm{I_2}$ formé par les ions thiosulfate $\mathrm{S_2O_3^{2-}}$.

Montrer (au brouillon) que l'équation de la réaction entre le diiode et les ions thiosulfate est la suivante (couples redox mis en jeu $\mathrm{I_2/I^-}$ et $\mathrm{S_4O_6^{2-}/S_2O_3^{2-}}$) :

$$\mathrm{I_2}(aq) + 2\ \mathrm{S_2O_3^{2-}}(aq) = 2\ \mathrm{I^-}(aq) + \mathrm{S_4O_6^{2-}}(aq)$$

La **couleur brune** qui se forme lors du premier mélange est due au diiode $\mathrm{I_2}(aq)$. C'est la disparition de cette couleur qui est utilisée pour repérer l'équivalence lors du titrage de la seconde étape.

> Elle est plus orange que brune.

**Montrer (au brouillon) que ces deux réactions peuvent $etre consid\acute{e}r\acute{e}e stotales. ** > Ondonne$E^{°}(\mathrm{IO_3^{-}/I_2})=1,195V;E^{°}(\mathrm{I_2/I^{-}})=0,62V;E^{°}(\mathrm{S_4O_6^{2-}/S_2O_3^{2-}})=0,08$ V.

Le protocole est le suivant :

- mélanger dans un bécher 20 mL de solution saturée d'iodate de baryum filtrée, 20 mL de solution d'iodure de potassium et 20 mL de solution d'acide sulfurique ;
- titrer le mélange obtenu à l'aide de la solution de thiosulfate de sodium.

> L'équivalence étant difficile à repérer, on utilise une pointe de spatule de thiodène (flacon nommé << Iodex >>) lorsque l'on s'approche de l'équivalence. Il se forme un complexe de coloration bleue très foncée.

Mettre en oeuvre le protocole et déterminer le volume à l'équivalence V_{eq} :

> Bien identifier ce qui doit être prélevé très précisément (pipette jaugée) et moins précisément (éprouvette graduée).

Entrée[ ]:
```
Veq=
print(Veq)
```

Montrer (au brouillon) que la concentration $[\mathrm{IO_3^-}]_{eq}$ est donnée par :

$$[\mathrm{IO_3^-}]_{eq} = \frac{1}{6} C_{\mathrm{S_2O_3^{2-}}} \frac{V_{eq}}{V_{prel}}$$

avec V_{prel} le volution de solution filtrée prélevée initialement (20 mL, à justifier).

Déterminer $[\mathrm{IO_3^-}]_{eq}$.

Entrée[ ]:
```
CIO3eq=
print(CIO3eq)
```

L'incertitude sur cette concentration est à évaluer. Comme son expression ne comprend que des multiplications, on peut utiliser la formule de composition des incertitudes vue dans ce cas.

Voici l'extrait concernant cette règle de calcul vue en première année, pour rappel elles sont toutes indiquées dans ce notebook (../incertitudes/incertitudes_de_mesure.ipynb).

Supposons que l'on calcule $x = ax_1^\alpha x_2^\beta$. L'incertitude-type **relative** de x est alors donnée par :

$$\frac{u(x)}{x} = \sqrt{\left(\alpha\frac{u(x_1)}{x_1}\right)^2 + \left(\beta\frac{u(x_2)}{x_2}\right)^2}$$

Il nous faut les incertitudes types sur $C_{S_2O_3^{2-}}$, V_{eq} et V_{prel}.

Pour V_{prel}, le prélèvement a été effectué à l'aide d'une pipette jaugée de 20 mL, dont on estime classiquement l'incertude type à $0,03$ mL :

Entrée[ ]:
```
uVprel=0.03e-3 # L
```

Pour $C_{S_2O_3^{2-}}$, on va l'estimer à :

Entrée[ ]:
```
uCS2O3=1e-3 # mol/L
print(uCS2O3)
```

Pour V_{eq}, si l'on prend l'incertitude type sur l'indication de la burette graduée, on passe à côté d'une autre incertitude, celle du repérage de l'équivalence dite **incertitude de pointé**. Le repérage n'est pas aisé, et est difficile à la goute près.

> On pourrait faire des statistiques sur les volumes à l'équivaence obtenus par les différents groupes, mais il faudrait que chaque groupe ait manipulé de le même façon, peu probable en Chimie en PSI :)

Estimer le nombre de gouttes qui correspondent à l'incertitude sur votre repérage de l'équivalence. En faisant à nouveau tomber ce nombre de gouttes, mesurer le volume correspondant à l'incertitude sur le volume à l'équivalence.

Entrée[ ]:
```
uVeq=
print(uVeq)
```

Déterminer l'expression (au brouillon) de l'incertitude sur $[IO_3^-]_{eq}$ à l'aide de la formule de composition des incertitudes vue plus haut puis la calculer.

Entrée[ ]:
```
uCIO3eq=
# Pour rappel
print(CIO3eq)
print(uCIO3eq)
```

Dosage des ions baryum Ba^{2+}

Les ions baryum Ba^{2+} seront titrés par conductimétrie. On utilise la réaction suivante, totale :

$$\mathrm{Ba}^{2+}(aq) + \mathrm{SO}_4^{2-}(aq) = \mathrm{BaSO}_4(s)$$

> Elle n'est pas tout à fait instantanée, il faudra attendre quelques secondes de stabilisation à chaque mesure.

C'est un réaction de précipitation, vous pourrez voir la coloration laiteuse qui apparait dès le début du titrage.

L'équivalence se repère par un changement brusque de la conductivité de la solution, en effet :

- avant l'équivalence la réaction ci-dessus s'effectue. Il y a globalement en solution disparition d'ions Ba^{2+} et introduction d'ion Na^{+} (apportés par la solution de sulfate de sodium qui apporte les ions sulfate $mathrmSO_4^{2-}$).

> Le précipité formé ne participe pas à la conduction en solution.

Les conductivité molaires à $25°C$ (en $\mathrm{S.cm^2.mol^{-1}}$) des ions sont :

C'EST UN VRAI SUPER-HÉROS !

Ion	H^+	OH^-	NO_3^-	IO_3^-	SO_4^{2-}	Na^+	Ba^{2+}	K^+
λ^0	350	198	71,4	40,5	160	50,1	127	73,5

TOUS CES IONS... SI CONDUCTEURS !

La conductivité de la solution est donnée par **la loi de Kohlrausch** :

$$\sigma = \sum_i \lambda_i \cdot [X_i] \simeq \sum_i \lambda_i^\circ \cdot [X_i]$$

> σ est notée γ en Physique, c'est la même...

Chaque ion $mathrmSO_4^{2-}$ introduit est accompagné de 2 ions Na^{+} (formule de la solution titrante), si bien que l'on doit comparer :

Entrée[ ]:
```
print(127,2*50,1)
```

Globalement on devrait constater une **baisse** (légère) de la conductivité de la solution.

- après l'équivalence on ajoute simplement des ions Na^{+} et $mathrmSO_4^{2-}$, la conductivité devrait **fortement augmenter**.

Le protocole est le suivant :

- prélever précisément 50 mL de la solution de iodate de baryum filtrée à titrer ;
- ajouter 100 mL d'eau distillée ;
- placer la solution de sulfate de sodium $\mathrm{Na_2SO_4}$ dans la burette graduée ;
- mettre en place le conductimètre (ne pas l'étalonner) ;
- mesurer la conductivité de la solution en fonction du volume versé.

> La réaction n'est pas tout à fait instantanée, il faudra attendre quelques secondes de stabilisation à chaque mesure.

Mettre en oeuvre le protocole et tracer la courbe $\sigma = f(V)$:

> Prendre plus de points **loin** de l'équivalence pour tracer les asymptotes.

Entrée[ ]:
```
V=[] # mL
sigma=[] # ?
```

Entrée[ ]:
```
plt.figure()

# Pour régler les paramètres de la représentation graphique (optionne
parameters = {"axes.labelsize": 13, "xtick.labelsize": 13, "ytick.lab
plt.rcParams.update(parameters)

plt.plot(V,sigma,'ro')

plt.xlabel('$V$ (mL)')
plt.ylabel('$\sigma$ (?)')
plt.title('Titrage conductimétrique')

# Pour sauvegarder l'image de la courbe (optionnel)
plt.savefig('courbe.png')

plt.show()
```

Le volume à l'équivalence se détermine par le tracé des asymptotes. On pourrait utiliser un outil d'ajustement comme la fonction `curve_fit()` mais souvent le résultat est meilleur << à la main >>.

Compléter dans le programme suivant les valeurs de a_1, b_1, a_2 et b_2 pour tracer par tatonnement les asymptotes de la courbe :

Entrée[ ]:
```
plt.figure()

# Pour régler les paramètres de la représentation graphique (optionne
parameters = {"axes.labelsize": 13, "xtick.labelsize": 13, "ytick.lab
plt.rcParams.update(parameters)

# Abscisses des asymptotes
N=200
Va=np.array([k*V/(N-1) for k in range(N)])

# Première asymptote (à compléter)
# a1=...
# b1=...
plt.plot(Va,a1*Va+b1,'b-')

# Seconde asymptote (à compléter)
# a2=...
# b2=...
plt.plot(Va,a2*Va+b2,'b-')

# Tracé des points expérimentaux
plt.plot(V,sigma,'ro')

plt.xlabel('$V$ (mL)')
plt.ylabel('$\sigma$ (?)')
plt.title('Titrage conductimétrique')

# Pour sauvegarder l'image de la courbe (optionnel)
plt.savefig('courbe.png')

plt.show()
```

Déterminer graphiquement (au pointeur) le volume à l'équivalence :

Entrée[ ]:
```
Veq=
print(Veq)
```

Montrer (au brouillon) que la concentration $[\mathrm{Ba}^{2+}]_{eq}$ est donnée par :

$$[Ba^{2+}]_{eq} = C_{\mathrm{SO}_4^{2-}} \frac{V_{eq}}{V_{prel}}$$

avec V_{prel} le volution de solution filtrée prélevée initialement (50 mL, à justifier).

Déterminer $[Ba^{2+}]_{eq}$.

CBaeq= print(CBaeq)

L'incertitude sur cette concentration est à évaluer. Comme son expression ne comprend que des multiplications, on peut utiliser la formule de composition des incertitudes vue dans ce cas.

Voici l'extrait concernant cette règle de calcul vue en première année, pour rappel elles sont toutes indiquées dans ce notebook (../incertitudes/incertitudes_de_mesure.ipynb).

Supposons que l'on calcule $x = ax_1^{\alpha}x_2^{\beta}$. L'incertitude-type **relative** de x est alors

donnée par :

$$\frac{u(x)}{x} = \sqrt{\left(\alpha\frac{u(x_1)}{x_1}\right)^2 + \left(\beta\frac{u(x_2)}{x_2}\right)^2}$$

Il nous faut les incertitudes types sur $C_{\mathrm{SO}_4^{2-}}$, V_{eq} et V_{prel}.

Pour V_{prel}, le prélèvement a été effectué à l'aide d'une pipette jaugée de 50 mL, dont on estime classiquement l'incertude type à $0,05$ mL :

Entrée[ ]:
```
uVprel=0.05e-3 # L
```

Pour $C_{\mathrm{SO}_4^{2-}}$, on va l'estimer à :

Entrée[ ]:
```
uCSO4=1e-3 # mol/L
print(uCSO4)
```

Pour V_{eq}, le repérage de l'équivalence relève clairement d'une **incertitude de pointé**.

> On pourrait faire des statistiques sur les volumes à l'équivaence obtenus par les différents groupes, mais il faudrait que chaque groupe ait manipulé de le même façon, peu probable en Chimie en PSI :)

Estimer graphiquement l'incertitude de pointé sur votre repérage de l'équivalence.

Entrée[ ]:
```
uVeq=
print(uVeq)
```

Déterminer l'expression (au brouillon) de l'incertitude sur $[\mathrm{Ba}^{2+}]_{eq}$ à l'aide de la formule de composition des incertitudes vue plus haut puis la calculer.

Entrée[ ]:
```
uCBaeq=
# Pour rappel
print(CBaeq)
print(uCBaeq)
```

Il ne nous reste plus qu'à calculer K_s et à estimer son incertitude :

$$K_s = \frac{[\mathrm{Ba}^{2+}]_{eq}[\mathrm{IO}_3^-]_{eq}^2}{C^{\circ 3}}$$

Calculer K_s :

Entrée[ ]:
```
Ks=CBaeq*CIO3eq**2/1**3
print(Ks)
```

Son incertitude type se détermine par la formule rappelée plus haut, à partir des incertitudes types sur chaque concentration :

Supposons que l'on calcule $x = ax_1^\alpha x_2^\beta$. L'incertitude-type **relative** de x est alors

donnée par :

$$\frac{u(x)}{x} = \sqrt{\left(\alpha\frac{u(x_1)}{x_1}\right)^2 + \left(\beta\frac{u(x_2)}{x_2}\right)^2}$$

Déterminer l'incertitude type sur K_s.

Entrée[ ]:
```
uKs=
print(uKs)
```

La valeur admise dans les table est de $K_s = 10^{-8,2}$. On considère que l'incertitude type sur cette valeur est négligeable (on la prendra donc nulle).

Calculer l'écart normalisé entre ces valeurs et conclure.

> Petit rappel juste après...

Entrée[ ]:
```
EN=
print(EN)
```

Pour comparer 2 mesures, on utilise **l'écart normalisé** ou **z-score** :

$$E_N = \frac{|m_2 - m_1|}{\sqrt{u(m_1)^2 + u(m_2)^2}}$$

Par convention, on qualifie deux résultats de **compatibles** si leur écart normalisé vérifie la propriété :

$$\boxed{E_N \lesssim 2}$$

> Ce seuil à 2 est d'origine historique. On le retrouve dans de nombreux champs scientifiques, comme la médecine, la pharmacie, la biologie, la psychologie, l'économie, l'écologie, etc. Ce seuil peut différer selon le domaine : par exemple pour démontrer l'existence d'une nouvelle particule en physique subatomique, il faut atteindre un seuil de 5.

Type *Markdown* and LaTeX: α^2

TP étude d'un transformateur

PE. LEROY

> Capacités exigibles du programme :
>
> **Conversion électromagnétique statique de puissance**
>
> - Mettre en \oe uvre un transformateur.
>
> **Puissance électrique**
>
> - Mesurer une puissance moyenne à l'aide d'un wattmètre numérique.

Liste du matériel :

- Alimentation 24 V/50 Hz de puissance
- Résistances de puissance de $1\ \Omega$ et $100\ \mathrm{k}\Omega$
- Condensateur de puissance de $1\ \mu\mathrm{F}$
- Plaque pour composants de puissance
- Circuit magnétique (transformateur)
- Bobines pour circuit magnétique ($\times 2$)
- Oscilloscope
- Multimètres ($\times 2$)
- Wattmètre
- Rhéostat $100\ \Omega$ (<< charge >>)

On commence par importer les bibliothèques nécessaires :

> La ligne suivante `%matplotlib notebook` est une commande magique qui sert à l'affichage des graphes dans le notebook.

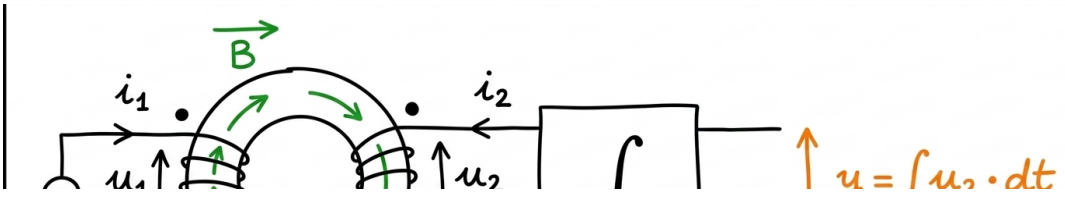
Entrée[ ]:
```
%matplotlib notebook
```

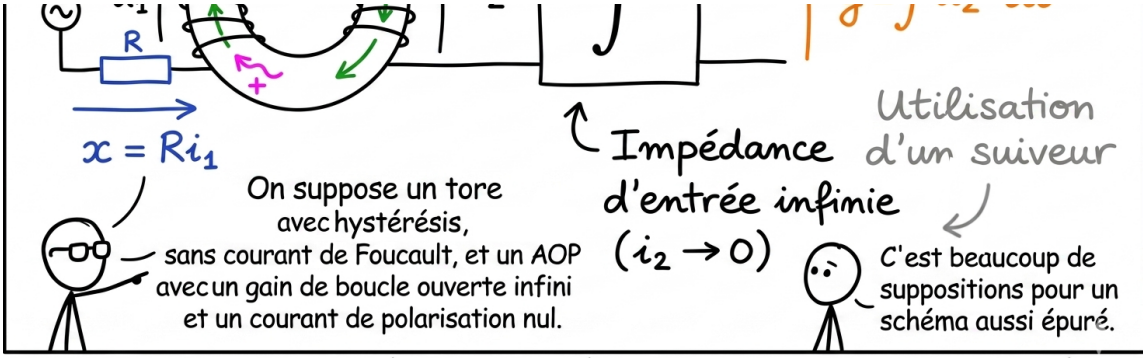
Entrée[2]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Nous allons commencer par tracer le cycle d'hystérésis d'un circuit magnétique qui va nous servir de transformateur dans la suite de la séance.

Tracé du cycle d'hystérésis du matériau

Pour tracer le cycle d'hystérésis nous avons vu en cours le principe de base :





REMARQUE : Le diagramme est dessiné en 'noir et blanc théorique', ce qui est une nuance de gris pour les ingénieurs.

En traçant la tension $y(t)$ en fonction de $x(t)$ on obtient la forme du cycle d'hystérésis $B = f(H)$.

$x(t)$ est la tension aux bornes d'une résistance de $1\ \Omega$.

> Cette résistance est dite << de puissance >> c'est à dire que sa puissance maximale d'utilisation est élevée.

$y(t)$ est la tension en sortie d'un montage intégrateur dont l'impédance d'entrée est élevée. On pourrait faire cela à l'aide d'un ALI si les ALI que l'on avait à notre disposition supportaient une grande puissance, mais ce n'est pas le cas.

> $P_{ALI} \sim 1$ W.

On utilisera donc comme montage intégrateur un simple filtre RC, en effet :

$$\underline{H} = \frac{H_0}{1 + j\omega/\omega_0}$$

avec $H_0 = 1$ et $\omega_0 = \frac{1}{RC}$, si bien que à HF (si $\omega >> \omega_0$) :

$$\underline{H} \sim \frac{H_0}{j\omega/\omega_0}$$

La fréquence utilisée sera de 50 Hz, $R = 100\ \text{k}\Omega$ et $C = 1\ \mu\text{F}$.

Montrer que l'on est dans les conditions permettant d'utiliser le filtre RC comme un intégrateur.

Entrée[4]:
```
R=100e3 # Ohm
C=1e-6 # F
f=50 # Hz
w0=1/(R*C)
w=2*np.pi*f
print(w/w0)
```

```
31.41592653589793
```

Réaliser le montage et visualiser le cycle d'hystérésis du matériau constituant le circuit magnétique du transformateur.

Étude du transformateur en fonctionnement

Étude << en charge >>

On commence par étudier le cas où le transformateur alimente une << charge >> constituée d'un rhéostat de $100\ \Omega$.

> Un rhéostat n'est rien d'autre qu'une résistance variable qui peut superter des puissances élevées car elle peut évacuer la chaleur (regardez la structure << en grille-pain >> de votre rhéostat).

Le montage a la structure suivante :

- l'alimentation de puissance (sortie 24 V/50 Hz) ;
- le transformateur (mettre un nombre de spires différentes en entrée et en sortie) ;
- le rhéostat.

Réaliser le montage.

On commence par vérifier la loi des tensions, pour cela brancher des voltmètres aux bornes :

- du circuit primaire (donc de l'alimentation) ;
- du circuit secondaire (donc du rhéostat).

> Les voltmètres seront réglés en mode `AC` ou `AC+DC`, ici aucune différence.
>
> > C'est quoi la différence ? Demandez à une IA pour voir :)

C'est donc la valeur efficace de chaque tension alternative (sinusoidale ici) que l'on récupère.

Faire varier la valeur de la résistance du rhéostat et mesurer à chaque fois les deux tensions efficaces :

> On utilise directement le type `array` de numpy car il est plus pratique, et cela évite de gérer d'éventuelles conversions à effectuer.

Entrée[ ]:
```
Ueff1=np.array([])
Ueff2=np.array([])
```

Il faut maintenant vérifier la loi des tensions :

$$\frac{U_{eff,2}}{U_{eff,1}} = \frac{N_2}{N_1} = m$$

Représenter `Ueff2` en fonction de `Ueff1` .

Entrée[ ]:

On peut faire un ajustement linéaire pour voir si << visuellement >> les points semblent alignés.

> Contrairement à une idée répandue, le coefficient r^2 n'a aucun intérêt pour valider un modèle physique ou pour estimer des incertitudes-type. Pour vous convaincre que le coefficient r^2 ne permet en rien de juger une régression linéaire, on peut regarder la vidéo suivante (dans laquelle le coefficient `cor` correspond au r) : Vidéo (https://youtu.be/lt4UA75z_KQ)

Le plus simple est d'utiliser la fonction `polyfit()` :

Entrée[ ]:
```
p=np.polyfit(Ueff1,Ueff2,1) # Ajustement de type a*x+b (polynôme de d
a,b=p[0],p[1] # Valeurs de a et b
```

Le coefficient directeur est censé être égal au rapport de transformation $m = N_2/N_1$, comparons :

Entrée[ ]:
```
N1=
N2=
m=N2/N1
print(a,m)
```

Voyons visuellement le résultat :

Entrée[ ]:
```
plt.figure()

# Pour régler les paramètres de la représentation graphique (optionne
parameters = {"axes.labelsize": 13, "xtick.labelsize": 13, "ytick.lab
plt.rcParams.update(parameters)

plt.plot(Ueff1,Ueff2,'ro')
plt.plot(Ueff1,a*Ueff1+b,'b-')
plt.xlabel('$U_{eff,1}$ (V)')
plt.ylabel('$U_{eff,2}$ (V)')
plt.legend(['Expérience','Modèle'])
plt.title('Loi des tensions dans un transformateur')

# Pour sauvegarder l'image de la courbe (optionnel)
plt.savefig('loi_des_tensions.png')

plt.show()
```

Pour correctement valider la modélisation il faut tenir compte des incertitudes sur chaque valeur. Le mode d'emploi du multimètre donne l'incertitude type :

Position commutateur	Gammes	Précision						Impédance D'entrée	Protection	Résolution
		DC*	40Hz**à 200 Hz	200 à 4 kHz	4 à 10 kHz	10 à 30 kHz	30 à 50 kHz			
		5 à 100 % du calibre			3 % typ.	10 % typ. ***	//////////	10MΩ /1GΩ	± 850VPK	

mV	500 mV*			typ.	typ.***		// 100 pF**		100 µV
V_{AC} ou V_{DC}+ SEL/ON	5 V	1 % L + 3 UR	1,5% L + 3 UR	2 % L + 3 UR		3 % L + 3 UR	11MΩ//100pF	± 850VPK	1 mV
	50 V						10 MΩ // 100 pF		10 mV
	500 V								100 mV
	600 V *****		1,5 % L + 3 UR****	//////////////		//////////////			1 V

*AC + DC seulement **20 Hz à 40 Hz = 1,5 % ***à 20 kHz ****jusqu'à 1 kHz
***** tension max. applicable : 600 VRMS CAT IV Fréquence max. : 15 000 [V * kHz] / Input [V]

Nombre de points :	5000
Sélection des gammes :	automatique ou manuelle pour les gammes 5 V, 50 V, 500 V, 600 V*
Réjection de mode commun :	à 50 et à 60 Hz, supérieure à 80 dB

Erreur additionnelle en fonction du facteur crête :
0 % pour un facteur crête inférieur à 1,5
1 % pour un facteur crête de 1,5 à 2
4 % pour un facteur crête de 2 à 3

On peut voir << 1% L + 3 UR >> soir 1% de la valeur lue plus 3 unité de résolution donc 30 mV. Ainsi :

Entrée[ ]:
```
uUeff1=Ueff1*1/100+3*0.01
uUeff2=Ueff2*1/100+3*0.01
```

On peut alors tracer en plus les barres d'incertitudes :

Entrée[ ]:
```
plt.figure()

# Pour régler les paramètres de la représentation graphique (optionne
parameters = {"axes.labelsize": 13, "xtick.labelsize": 13, "ytick.lab
plt.rcParams.update(parameters)

plt.errorbar(Ueff1,Ueff2,uUeff2,uUeff1,,fmt='ro',ecolor='blue',capsiz
plt.plot(Ueff1,a*Ueff1+b,'b-')
plt.xlabel('$U_{eff,1}$ (V)')
plt.ylabel('$U_{eff,2}$ (V)')
plt.legend(['Expérience','Modèle'])
plt.title('Loi des tensions dans un transformateur')

# Pour sauvegarder l'image de la courbe (optionnel)
plt.savefig('loi_des_tensions.png')

plt.show()
```

Conclure.

Profitons-en pour aborder la dernière notion des incertitudes que nous n'avons pas encore utilisée : la détermination de l'incertitude sur une grandeur déterminée par ajustement, ici le rapport de transformation m.

Voici un extrait du notebook qui traite des incertitudes :

Détermination de l'incertitude-type sur les paramètres par méthode Monte-Carlo

On commence par réaliser une régression linéaire sans incertitudes sur les valeurs mesurées. La pente et l'ordonnée à l'origine obtenues sont le résultat numérique de la

mesure.

Pour estimer leurs incertitudes-type, il faut de plus estimer pour chaque point de mesure la valeur de son incertitude-type. Ensuite, à l'aide d'une simulation Monte-Carlo, on utilise cette variabilité individuelle pour générer un grand nombre de distributions de points. Pour chacune de ces distributions, on réalise une régression (ou ajustement) linéaire ce qui conduira au final à un grand nombre de valeurs de a et de b. Il suffit ensuite de réaliser une étude statistique de ces données pour en déduire leur incertitude-type.

> Pour s'assurer qu'une régression linéaire est correcte, on tracera systématiquement sur un graphique les données mesurées ainsi que la droite de la régression linéaire. Le modèle sera validé si, à l'oeil, les points de mesure sont bien alignés et que la droite passe le plus proche de tous les points possibles, en incluant leurs incertitudes-type.

Ci-dessous un exemple **que vous pouvez adapter** en fonction de vos besoins :

Entrée[ ]:
```
import random as rd

####### Partie à adapter ########
x=Ueff1
y=Ueff2

p=np.polyfit(x,y,1) # Ajustement de type a*x+b
a,b=p[0],p[1] # Valeurs de a et b

ux=np.mean(uUeff1)
uy=np.mean(uUeff2)

####### Partie à ne pas modifier ########
def variations_aleatoires(liste_x,liste_y): # Variation aléatoire de
    liste_xa=[]
    liste_ya=[]
    for x,y in zip(liste_x,liste_y):
        liste_xa.append(x+loi_normale(ux))
        liste_ya.append(y+loi_normale(uy))
    return liste_xa,liste_ya

def loi_normale(sigma) : # Loi de distribution normale
    return np.random.normal(0,sigma)

def loi_rectangulaire(sigma): # Loi de distribution rectangulaire
    return rd.uniform(-sigma,sigma)

distribution_a=[] # Distribution des valeurs de a
distribution_b=[] # Distribution des valeurs de b
N=100000 # Nombre de points utilisés dans la méthode Monte-Carlo

for i in range(N):
    xa,ya=variations_aleatoires(x,y)
    p=np.polyfit(xa,ya,1)
    distribution_a.append(p[0])
    distribution_b.append(p[1])

incertitude_type_a=np.std(distribution_a) # Calcul de l'incertitude-t
incertitude_type_b=np.std(distribution_b) # Calcul de l'incertitude-t

print('a=',a,'u(a)=',incertitude_type_a)
print('b=',b,'u(b)=',incertitude_type_b)

yfit=[]
for xfit in x:
    yfit.append(a*xfit+b)

plt.figure()
plt.plot(x,y,'ro')
plt.plot(x,yfit,'g-')
plt.xlabel('$x$')
plt.ylabel('$y$')
plt.show()
```

L'incertitude sur a est bien celle sur m. On ne fera pas le calcul de l'écart normalisé pour comparer a et m.

> Notez que l'on a utilisé les valeurs moyennes des incertitudes sur `Ueff1` et `Ueff2` pour s'adapter au programme, mais en toute rigueur il faudrait tenir compte de chaque incertitude sur chaque valeur.
>
> > C'est bon, c'est pas la NASA ici, on ne fait pas décoller de fusée :)

Et la loi des courants ? Sur le même montage, enlever les voltmètres et les replacer en tant qu'ampèremètres, de façon à mesurer :

- le courant à l'entrée du circuit primaire (donc qui sort de l'alimentation) ;
- le courant dans le circuit secondaire (donc dans le rhéostat).

> ATTENTION un ampèrement se branche en série, il faut bien qu'il soit traversé par le courant pour pouvoir le mesurer !

Même chose on mesurera des courants efficaces, donc en mode `AC` .

La loi des courants est donnée par :

$$\frac{I_{eff,2}}{I_{eff,1}} \simeq \frac{1}{m}$$

> Dans le cours on a bien $\frac{i_2(t)}{i_1(t)} \simeq -\frac{1}{m}$, le signe << - >> n'apparait pas pour les tension efficaces.

Faire la même étude que précédemment pour vérifier la loi des courants.

Entrée[ ]:
```
Ieff1=np.array([])
Ieff2=np.array([])
```

Entrée[ ]:
```

```

Entrée[ ]:
```
p=np.polyfit(Ieff1,Ieff2,1) # Ajustement de type a*x+b (polynôme de d
a,b=p[0],p[1] # Valeurs de a et b
```

Entrée[ ]:
```
N1=
N2=
m=N2/N1
print(a,1/m)
```

Entrée[ ]:
```
plt.figure()

# Pour régler les paramètres de la représentation graphique (optionne
parameters = {"axes.labelsize": 13, "xtick.labelsize": 13, "ytick.lab
plt.rcParams.update(parameters)

plt.plot(Ieff1,Ieff2,'ro')
plt.plot(Ieff1,a*Ieff1+b,'b-')
plt.xlabel('$I_{eff,1}$ (V)')
plt.ylabel('$I_{eff,2}$ (V)')
plt.legend(['Expérience','Modèle'])
plt.title('Loi des courants dans un transformateur')

# Pour sauvegarder l'image de la courbe (optionnel)
plt.savefig('loi_des_courants.png')

plt.show()
```

Voici l'extrait de la notice donnant les renseignements sur les incertitudes pour la mesure de courants :

Gammes	Précision			Protection	Fusibles*	Chute de tension	Résol.	Crête max.
	40Hz à 5kHz	5 à 20kHz	20 à 30kHz					
	5 % à 100 % du calibre							
5 mA	1% L + 3 UR	1,5 % typ.	4 % typ.	600 VRMS	F1 + F2	700 mV	1 µA	10 mA
50 mA	1% L + 3 UR	1,5 % typ.	4 % typ.				10 µA	100 mA
500 mA		1,5 % typ.	4 % typ.			1.5 V	100 µA	1 A
10 A**	1,5% L+3 UR → 2 kHz	1,5 % typ.	######		F2	500 mV	10 mA	30 A

* voir caractéristiques des fusibles paragraphe §. 6.1.1

** surcharge admissible : 20 A pendant 30 s max. avec un temps de pause > 5 min entre 2 tests

Nombre de points : 5 000

Erreur additionnelle en fonction du facteur crête : 0 % pour un facteur crête inférieur à 1,5
1 % pour un facteur crête de 1,5 à 2
4 % pour un facteur crête de 2 à 3

Erreur additionnelle en IAC+DC, pour un courant continu en entrée : 1 %

Entrée[ ]:
```
uIeff1=Ieff1*1.5/100+3*0.01
uIeff2=Ieff2*1.5/100+3*0.01
```

Entrée[ ]:
```
plt.figure()

# Pour régler les paramètres de la représentation graphique (optionne
parameters = {"axes.labelsize": 13, "xtick.labelsize": 13, "ytick.lab
plt.rcParams.update(parameters)

plt.errorbar(Ieff1,Ieff2,uIeff2,uIeff1,,fmt='ro',ecolor='blue',capsiz
plt.plot(Ieff1,a*Ieff1+b,'b-')
plt.xlabel('$I_{eff,1}$ (V)')
plt.ylabel('$I_{eff,2}$ (V)')
plt.legend(['Expérience','Modèle'])
plt.title('Loi des courants dans un transformateur')

# Pour sauvegarder l'image de la courbe (optionnel)
plt.savefig('loi_des_courants.png')

plt.show()
```

Conclure.

Étude << à vide >>

Quand on dit << à vide >>, c'est que le circuit secondaire n'est branché sur rien, il est ouvert.

Sans rien changer au montage précédent, débrancher le rhéostat et mesurer le courant efficace traversant dans le circuit primaire :

Entrée[ ]:
```
Ieffmag=
```

C'est ce que l'on appele le **courant magnétisant**. Même à vide, un courant est nécessaire pour établir le champ magnétique dans le circuit magnétique. Puisque le courant est nul au secondaire, la loi des courants n'est pas vérifiée, mais la loi des tensions si.

> Ainsi, un transformateur laissé branché sur le secteur sans être utilisé consomme de l'énergie électrique.

Mesure de pertes

On a, en tenant compte des pertes << cuivre >> et << fer >>, en notant $< p >\equiv p$:

$$p_1 = p_2 + p_{cuivre} + p_{fer}$$

Par la mesure de p_1 et p_2 on peut évaluer, en fonction de la valeur de la charge, le rendement en puissance :

$$\eta = \frac{p_2}{p_1}$$

A vide la puissance au secondaire est nulle. La puissance mesurée au primaire est principalement due aux pertes << fer >> :

$$p_1 \simeq p_{fer}$$

Ainsi, dans votre montage à vide (le même que pour la mesure du courant

magnétisant), la puissance délivrée par le générateur ne sert quasiment qu'à compenser les pertes << fer >>.

Pour mesurer ces pertes, on ne peut pas juste ajouter un voltmètre car $< p > \neq I_{eff} U_{eff}$, il maque le facteur de puissance $\cos \varphi$. Il faut donc mesurer le déphasage φ entre le courant et la tension. Le watt-mètre réalise tout cela, il mesure le courant, la tension, le déphasage et calcule la puissance moyenne.

Remplacer l'ampèremètre par la partie mesure de l'intensité du wattmètre. Brancher la partie mesure de la tension du watt-mètre aux bornes de l'alimentation. Mesurer les pertes fer dans cette configuration.

Entrée[ ]:
```
pfer= # W
```

Si on met juste un fil au secondaire, on le << court-circuite >>. En court-circuit, la tension au secondaire est quasi-nulle, mais ce n'est évidemment pas le cas de la tension au primaire. Ainsi, la loi des tensions n'est pas vérifiée.

Le courant i_2 est ici le courant de court-circuit, il est donc élevé, et la loi des courants reste bien vérifiée.

> Ainsi, un transformateur court-circuité peut être le siège d'un fort courant au secondaire et être endommagé.

La puissance mesurée au primaire est principalement due aux pertes << cuivre >> :

$$p_1 \simeq p_{cuivre} = R_1 I_{eff,1}^2 + R_2 I_{eff,2}^2$$

puisque la loi des courant est toujours vérifiée.

> En pratique, la tension d'alimentation et le courant i_1 doivent rester suffisamment faibles pour que le courant i_2 n'endommage pas les spires du secondaire.

Court-circuiter le circuit secondaire (remplacer le rhéostat débranché par un fil) puis mesurer les pertes cuivres dans cette configuration.

Entrée[ ]:
```
pcuivre= # W
```

Type *Markdown* and LaTeX: α^2

TP variateur de lumière

PE. LEROY

> Capacités exigibles du programme :
>
> **Puissance électrique**
>
> - Mesurer une puissance moyenne à l'aide d'un wattmètre numérique.

Liste du matériel :

- Alimentation stabilisée continue de puissance (U ou I réglables)
- Ampoule 12 V et son support
- Plaquette de montage de composants (LAB)
- Boîtes de résistance/inductance/capacité variables
- Wattmètre
- GBF
- Oscilloscope
- Transistor MOSFET << Canal n IRF 640~>>
- Diode rapide << BYW80-200 >>
- Platine << onduleur autonome à commande décalée >>
- Miltimètre Metrix (rouge)

On commence par importer les bibliothèques nécessaires :

> La ligne suivante `%matplotlib notebook` est une commande magique qui sert à l'affichage des graphes dans le notebook.

Entrée[ ]:
```
%matplotlib notebook
```

Entrée[2]:
```
import numpy as np
import matplotlib.pyplot as plt
```

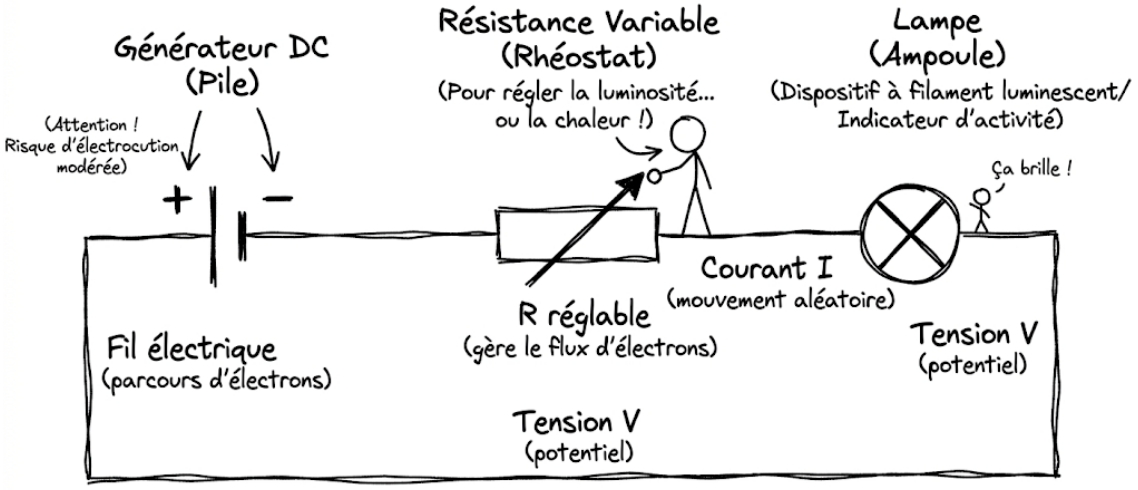
L'ojectif est, à travers l'élaboration d'un variateur de lumière, de montrer l'intérêt immense d'un convertisseur lorsqu'il s'agit de réguler la puissance fournie à un système électrique.

Réalisation d'un variateur << classique >> pour l'éclairage d'une ampoule

On cherche à réguler l'éclairage d'une ampoule, donc à réguler la puissance $p = u_L \cdot i_L$ fournie à cette ampoule par un générateur.

> On peut imaginer u_L et i_L comme constantes, ou même variables dans le temps. Ici nous prendrons une tension constante.

Le montage le plus simple est d'intercaler une résistance variable pour réguler le courant i_L qui traverse la lampe et/ou la tension u_L à ses bornes. La résistance peut être en série, ou en parallèle, prenons-là au plus simple en série, en mode << PDT >> :



NB : L'ampoule brille si V>0 et R est faible. Sauf si elle est grillée. C'est compliqué. 350.5 V, R=50Ω, Lampe=1W

Réaliser le montage et constater la bonne régulation de la luminosité de la lampe. Utiliser un wattmètre pour mesurer la puissance délivrée par le générateur lorsque la tension d'alimentation est de 12 V et que la tension aux bornes de la lampe est de 10 V (à mesurer avec un voltmètre).

> Vérifier que vous avez bien une lampe << 12 V >>.

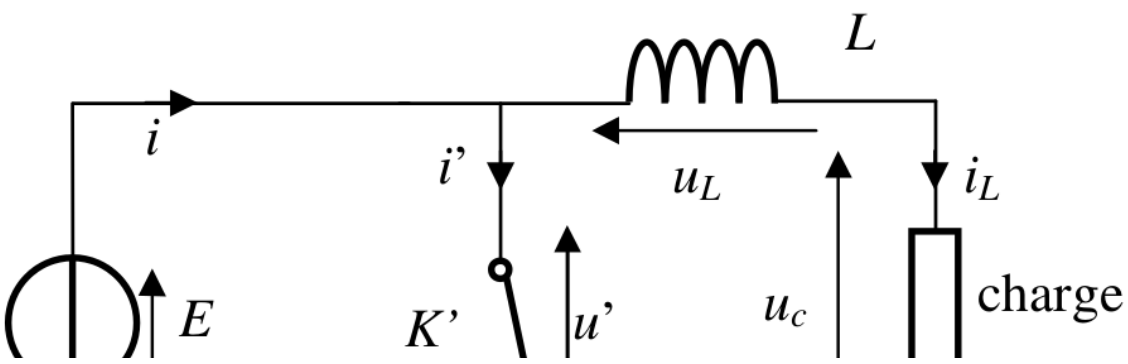
Entrée[ ]:
```
pPDT= # W
```

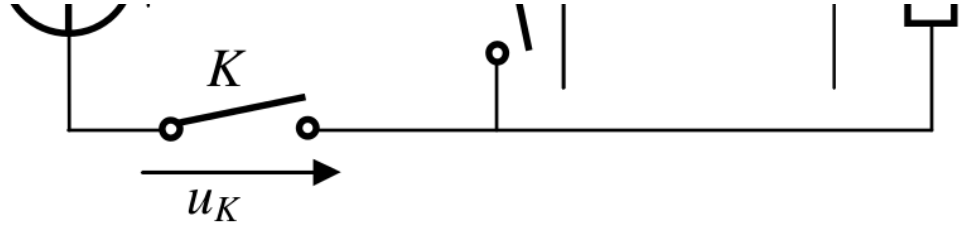
Dans ce genre de montage, il faut comprendre que l'énergie délivrée par le générateur qui n'est pas consommée par la lampe est **dissipée par effet Joule**. On se sert du fait q'une résistance chauffe pour réguler la puissance dans la lampe !

Réalisation d'un variateur plus élaboré pour l'éclairage d'une ampoule

On va utiliser un convertisseur plutôt qu'une résistance pour réguler la puissance fournie à la lampe.

C'est un hacheur série, dont le principe est représenté sur ce schéma avec des interrupteurs :





Périodiquement, on ouvre/ferme chaque interrupteur, sachant que si l'un est fermé l'autre est systématiquement ouvert :

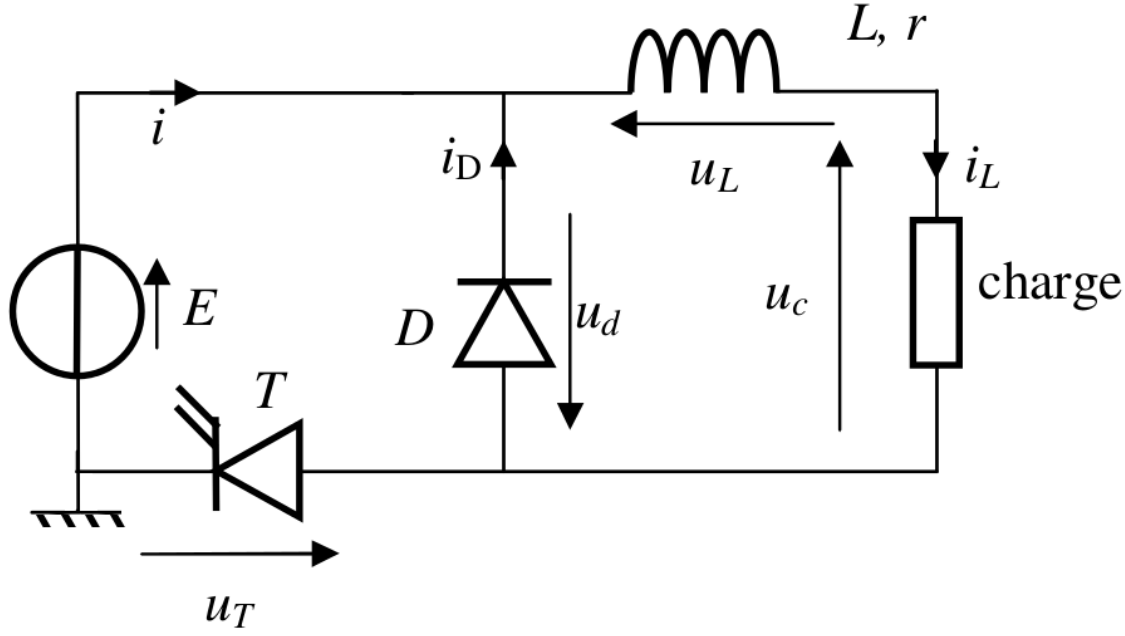
- entre 0 et αT, K est ouvert et K' est fermé ;
- entre αT et T c'est l'inverse.

On appelle **rapport cyclique** la grandeur α.

On a montré en cours que cela s'effectue avec :

- un transistor pour K (commutation commandée, à piloter en ouverture/fermeture par un système extérieur comme un GBF) ;
- une diode pour K' (commutation spontanée, aucun pilotage).

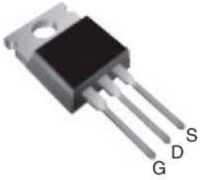
Cela donne :

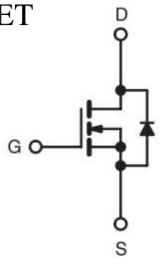


Le transistor et la diode doivent supporter un grand courant (environ 10 A), et pour la diode une tension inverse élevée (environ 100 V). Par ailleurs, ils doivent être assez rapides (surtout la diode, car les diodes les plus courantes ne commutent pas correctement à haute fréquence).

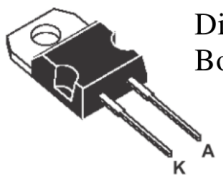
On utilise ainsi un transistor de type MOSFET canal n IRF640 et une diode rapide de puissance BYW80-uT 200 :

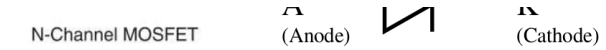
Transistor MOSFET
Canal n IRF 640





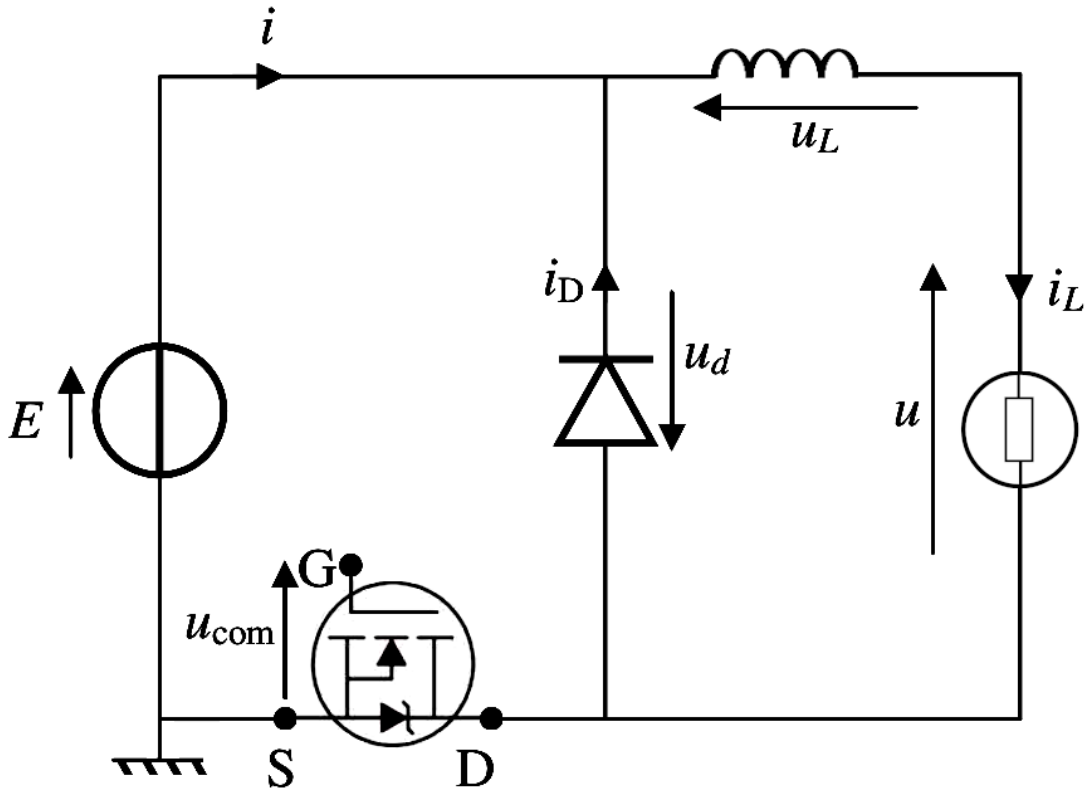
Diode rapide BYW80-200
Boîtier TO-220 AC





C'est la tension u_{com} entre la grille G et la source S qui commande le transistor. Lorsque cette tension est nulle, le courant i entre le drain D et la source S est nul (interrupteur K ouvert), et lorsque la tension a une valeur positive suffisante (10 V ici), le courant i circule librement et u_T est presque nul (interrupteur K fermé).

Le montage final utilisé sera :



La tension de commande u_{com} sera délivrée par le GBF, avec les caractéristiques suivantes (**à régler au préalable à l'oscilloscope**) :

- signal créneau variant entre 0 V et 10 V (**pas de valeurs négatives !**) ;
- fréquence $f = 8,0$ kHz et rapport cyclique initial $\alpha \simeq 65\%$;

La self de lissage est une boîte d'inductance variable (on peut faire varier sa valeur), régler par défaut sur 10 mH.

> La lampe à incandescence est représentée ici par son symbole normalisé.

Réaliser le montage et vérifier qu'en réglant le rapport cyclique α la luminosité de la lampe varie.

> Attention aux bornes du transistor et de la diode, rien ne peut être fait << au hasard >> !

On souhaite désormais visualiser à l'oscilloscope la tension aux bornes de la lampe.

Expliquer à votre camarade pourquoi on ne peut pas simplement brancher l'oscilloscope aux bornes de la lampe.

> Penser à la masse...

Trouver une astuce pour visualiser la tension aux bornes de la lampe avec les deux voies de l'oscilloscope. Observer l'influence de la valeur de l'inductance L

> Penser au mode `Math` où vous pouvez faire par exemple une différence entre deux tensions.

Afficher la tension moyenne aux bornes de la lampe à l'aide du menu `Meas` . Vérifier l'influence de α sur cette valeur. Utiliser un wattmètre pour mesurer la puissance délivrée par le générateur lorsque la tension d'alimentation est de 12 V et que la tension moyenne aux bornes de la lampe est de 10 V (à mesurer avec un voltmètre).

Entrée[ ]:
```
pConv= # W
# Pour comparer
print(pPDT,pConv)
```

Qu'en pensez-vous ?

On a réalisé un montage << hacheur dévolteur >>, c'est à dire l'équivalent d'un transformateur, **sans pertes fer, sans pertes par hystérésis, sans pertes cuivre**.

> Ce n'est pas rien !

On peut montrer que l'on a théoriquement :

$$< u >= \alpha E$$

C'est une sorte de loi des tensions !

$$\frac{< u >}{E} = \frac{< u_s >}{< u_e >} = \alpha = m$$

Ainsi le rapport cyclique α s'aparente au rapport de transformation m.

Mesurer la tension moyenne aux bornes de la lampe et la tension aux bornes du générateur (environ 12 V, à mesurer au voltmètre) en fonction du rapport cyclique α :

Entrée[ ]:
```
um=np.array([])
E=np.array([])
alpha=np.array([])
```

Représenter le rapport $< u > /E$ en fonction de alpha :

Entrée[ ]:

Le coefficient directeur devrait être proche de 1.

Réaliser un ajustement linéaire et déterminer le coefficient directeur de la droite passant au mieux par tous les points.

> En fait un ajustement affine, avec la fonction `polyfit()` .

Entrée[ ]:

Plutôt que de chercher à étudier les incertitudes, profitons-en pour voir un << tips >> pour vérifier que l'ajustement linéaire ne cache par une autre forme de variation. On peut tracer ce que l'on appel **les résidus** r_i en fonction de x_i :

$$r_i = y(x_i) - \overline{y}(x_i)$$

Tracer les résidus de cette représentation et conclure :

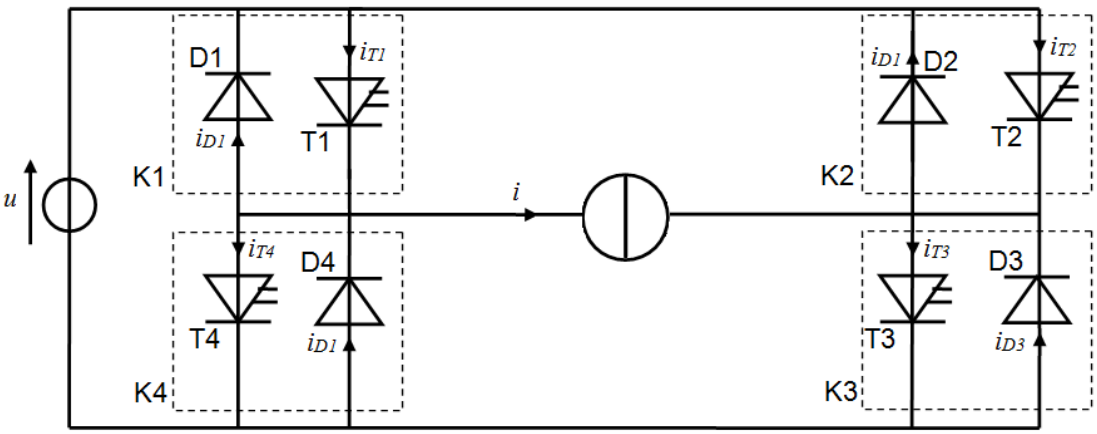
Entrée[ ]:
```
r=um/E-np.mean(um/E)

plt.figure()
plt.plot(alpha,r,'ro')
plt.xlabel('$\alpha$')
plt.ylabel('r')
plt.title('Tracé des résidus')
plt.show()
```

Utilisation d'un onduleur

On a un onduleur tout fait au labo, autant s'en servir :)

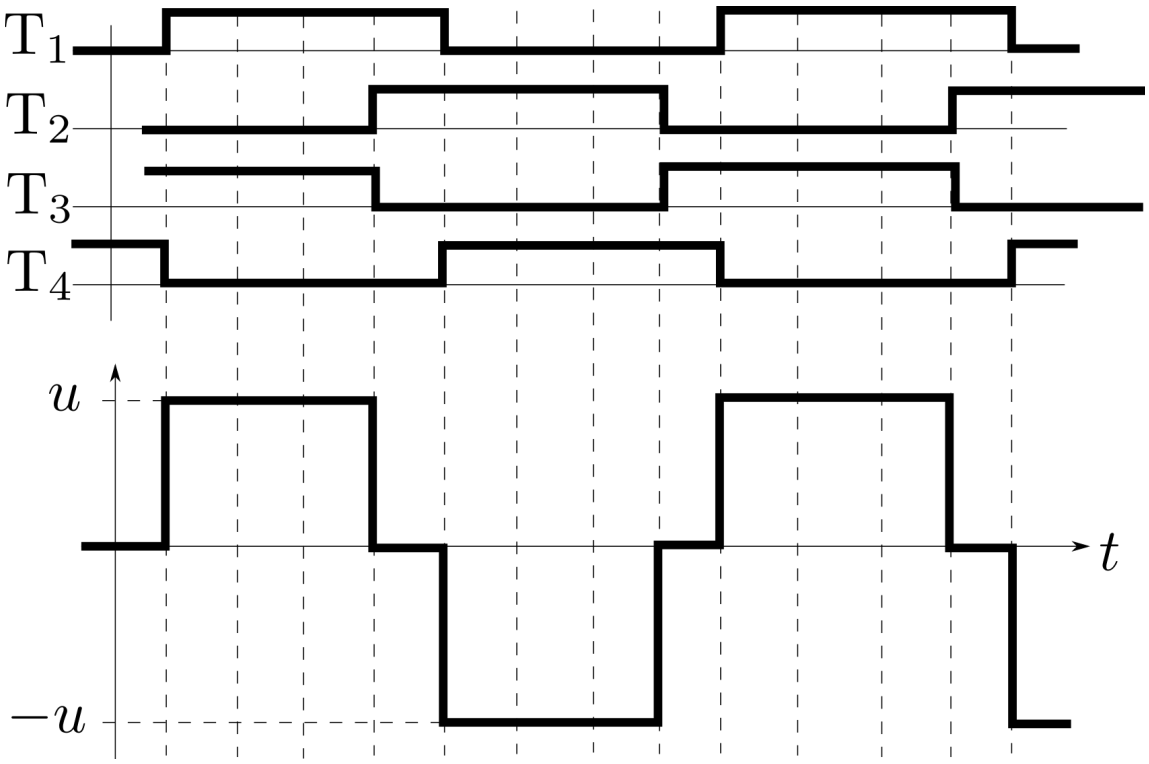
> Il permet de transformer une tension continue en tension alternative.

Le montage de base est appelé hacheur << à 4 quadrants >> :



Les transistors T_1, T_2 T_3 et T_4 sont ouverts ou fermés en fonction du type de signal souhaité.

Nous allons voir dans le paragraphe suivant un exemple de << commande décalée >>. La platine à disposition fonctionne en << commande décalée, c'est à dire qu'il existe des moments où la << charge >> (représentée sur la figure précédente par un dipôle type source de courant) est court-circuitée (c'est le cas lorsque les signaux de commande sur T_1 et T_2 sont identiques, et c'est aussi le cas lorsque ceux sur T_3 et T_4 sont identiques). L'allure des chronogrammes est alors la suivante :



Le signal obtenu a ainsi une fréquence qui dépend du raport cyclique α.

Connecter l'alimentation en 12 V au boitier et visualiser le résultat à l'oscilloscope. Remplacer l'oscolloscope par la lampe et conclure.

Type *Markdown* and LaTeX: α^2

TP véhicule autonome

PE. LEROY

> Capacités exigibles du programme :
>
> **Conversion électromécanique de puissance**
>
> - Mettre en oeuvre une machine à courant continu.

Liste du matériel :

- Alimentation stabilisée (~ 5 V)
- Plaquette de montage de composants (LAB)
- Oscilloscope
- Résistances de $10\ \mathrm{k\Omega}$ et $100\ \Omega$ ($\times 4$)
- Condensateur de $100\ \mathrm{nF}$
- Diodes silicium ($\times 5$)
- Transistor MOSFET << Canal n IRF 640 >>
- Transistors \textbf{NPN} BDX33C ($\times 2$)
- Transistors PNP BDX34C ($\times 2$)
- Petit moteur à courant continu (avec hélice)
- Module Arduino (avec logiciel installé)
- Véhicule (jouet) avec connexions aux moteurs déportée

On commence par importer les bibliothèques nécessaires :

> La ligne suivante `%matplotlib notebook` est une commande magique qui sert à l'affichage des graphes dans le notebook.

Entrée[ ]:
```
%matplotlib notebook
```

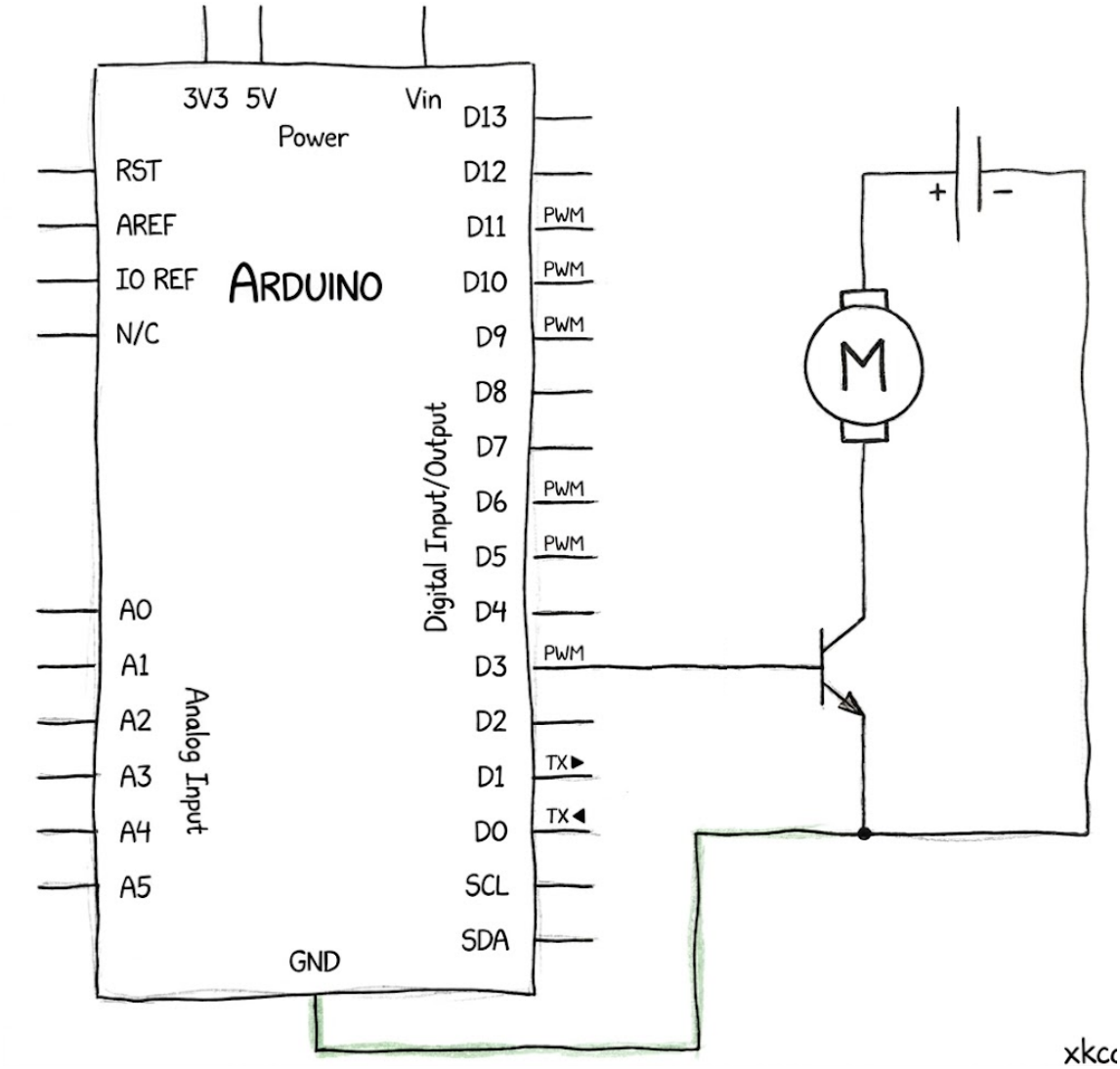
Entrée[2]:
```
import numpy as np
import matplotlib.pyplot as plt
```

L'objectif est de programmer un véhicule (jouet) pour qu'il se gare seul (comme un véhicule autonome).

Pilotage d'un moteur à courant continu

Le pilotage d'un MCC à l'aide d'un module Arduino pourrait sembler simple : relier le moteur à l'une des sorties et pioter cette sortie. Malheureusement les sortie d'une module Arduino sont limitées en puissance, plus exactement chaque sortie ne peut délivrer plus de 20 mA, ce qui est bien insuffisant pour alimenter un moteur à courant continu.

L'idée est alors d'utiliser le module Arduino pour piloter un transistor, qui fera office d'interrupteur dans un circuit comprenant un générateur et le moteur :



Il reste à améliorer ce montage par différents ajouts :

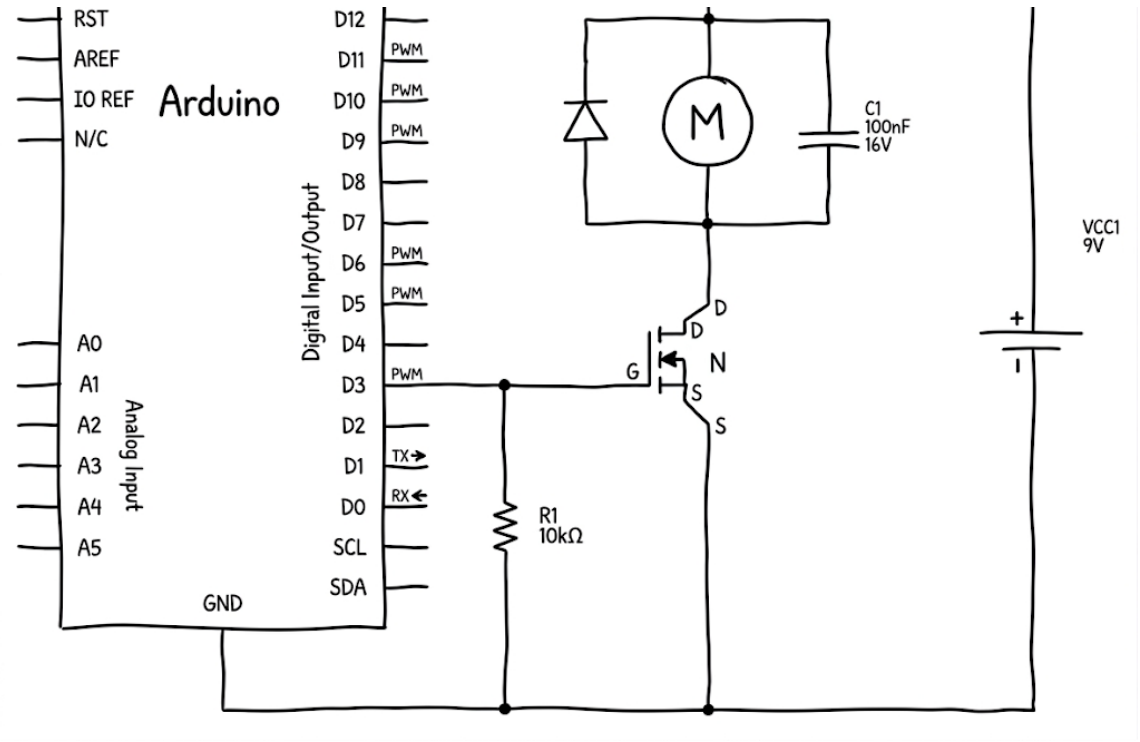
- on ajoute une résistance R_1 entre la broche de sortie du module Arduino et la masse, ainsi si le signal n'est pas défini sur la broche, le transistor sera par défaut bloqué et le moteur ne tournera pas ;

> On parle de << résistance de pull-down >>.

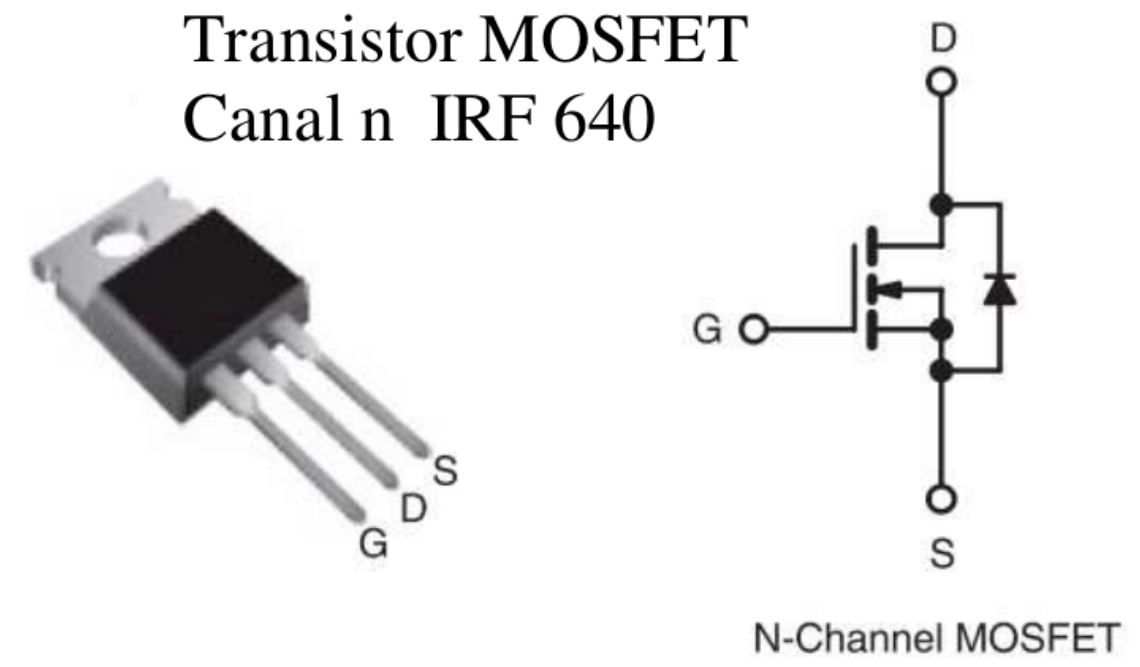
- on ajoute une diode en dérivation aux bornes du moteur, de façon à ce que si une surtension apparait aux bornes du moteur (qui est constitué de bobinages) du fait de l'ouverture du circuit par le transistor, le courant résultatnt circulera dans la diode sans endomager le transistor ;
- on ajoute un condensateur C_1 en dérivation aux bornes du moteur pour protéger le transistor et la diode des parasites électriques engendrés par le fonctionnement du moteur.

> On parle de << condensateur de déparasitage >>.

Cela donne finalement :



On prendra un transistor MOSFET << Canal n IRF 640 >> :



La résistance utilisée sera de valeur $R_1 = 10\ \text{k}\Omega$ et le condensateur de capacité $C_1 = 100$ nF.

Réaliser le montage.

On va désormais pouvoir commander le transistor à l'aide du module Arduino. C'est l'instruction `digitalWrite(3,HIGH)` qui nous permettra de mettre la broche de sortie 3 en niveau haut (5 V), et `digitalWrite(3,LOW)` qui nous permettra de mettre la broche de sortie 3 en niveau bas (0 V), et donc de piloter le caractère passant ou bloquant du transistor.

On y va ! Le début est classique :

Entrée[ ]:
```
!pip install jupyno
```

Entrée[ ]:
```
import jupyno
```

Entrée[ ]:
```
port, FQBN = jupyno.trouver_arduino()
```

Entrée[ ]:
```
croquis = jupyno.creer_croquis('PilotageMoteur')
```

Entrée[ ]:
```
%%ecrire_croquis $croquis

const int com_m=3;

// the setup routine runs once when you press reset
void setup()
{
// initialize the digital pin as an output
pinMode(com_m,OUTPUT);
}

// the loop routine runs over and over again forever
void loop()
{
digitalWrite(com_m,HIGH);
delay(1000);
digitalWrite(com_m,LOW);
delay(1000);
}
```

Entrée[ ]:
```
jupyno.televerser(croquis, port, FQBN)
```

Vous devriez voir votre moteur tourner pendant 1 s, s'arrêter 1 s, et ainsi de suite.

Pilotage en vitesse

Le pilotage en vitesse s'effectue en remplaçant simplement `digitalWrite(com_m,HIGH)` par `analogWrite(com_m,155)` par exemple. En effet, la sortie numérique du module Arduino peut fonctionner de deux façons :

- en mode << haut >> et << bas >> (`HIGHT` et `LOW`) ;
- en mode << créneau >> dont le rapport cyclique α est réglable par une valeur entre `0` et `255` :

$$\alpha = \frac{N}{255}$$

Ainsi entre 0 et αT on aura un niveau haut, puis entre αT et T un niveau bas, et ainsi de suite.

On parle de **Modulation de Largeur d'Impulsion** ou **MLI** en français ou PWM en anglais (Pulse Width Modulation). Typiquement on a sur le modèle UNO :

- un signal PWM de $976,56$ Hz sur les broches de sortie D5 ou D6
- un signal PWM de $490,20$ Hz sur les broches de sortie D3, D9, D10, ou D11

Modifier le programme précédent pour faire tourner le moteur à vitesse

moyenne au lieu de la pleine vitesse pendant 1 s :

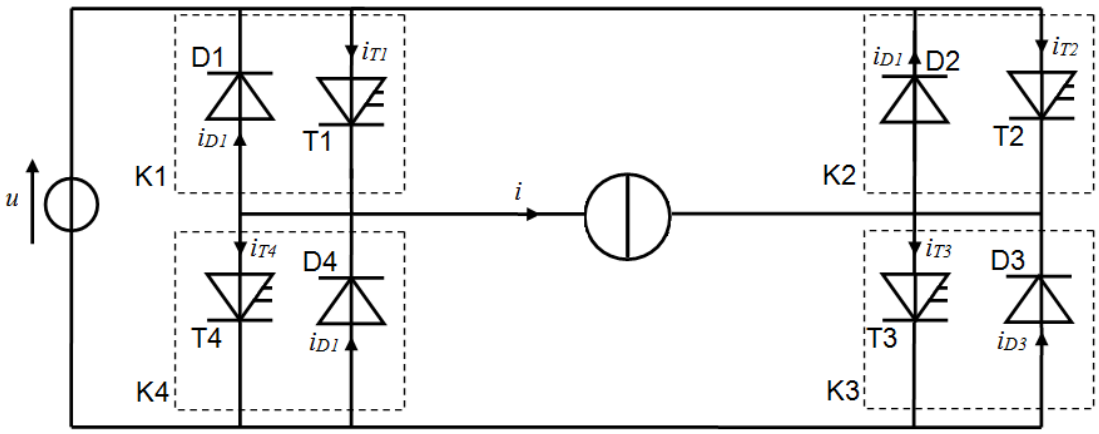
Entrée[ ]:
```
%%ecrire_croquis $croquis

const int com_m=3;

// the setup routine runs once when you press reset
void setup()
{
// initialize the digital pin as an output
pinMode(com_m,OUTPUT);
}

// the loop routine runs over and over again forever
void loop()
{
digitalWrite(com_m,HIGH);
delay(1000);
digitalWrite(com_m,LOW);
delay(1000);
}
```

Entrée[ ]:
```
jupyno.televerser(croquis, port, FQBN)
```

Gardez votre montage, il vous servira dans la dernière partie.

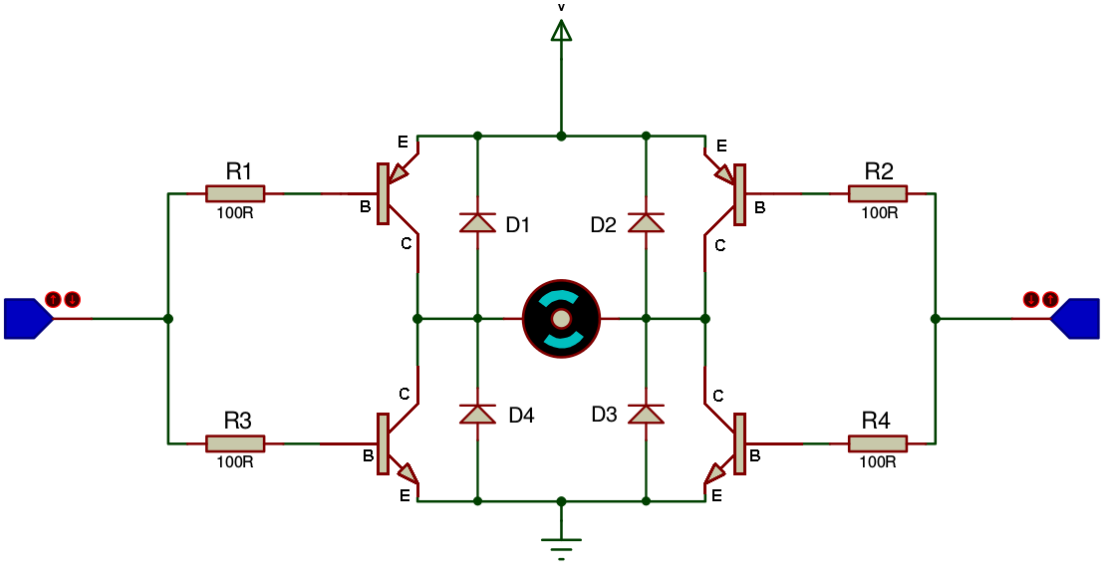
Pilotage en sens

Le montage de base est le même que pour l'onduleur, ce montage est appelé hacheur << à 4 quadrants >> :



Les transistors T_1 et T_3 d'une part, T_2 et T_4 d'autre part sont ouverts ou fermés en même temps (ils sont synchronisés). Ces deux groupes de transistors sont alternativement ouverts et fermés.

Le pilotage des quatre transistors peut s'effectuer à l'aide de signaux logiques issus d'un module Arduino. Pour une mise en oeuvre simplifiée, on peut utiliser des transistors MOS ou bipolaires, **de type PNP pour les interrupteurs du haut (1 et 2) et NPN pour les deux interrupteurs du bas (3 et 4)**. On considèrera que leurs comportements sont simplement inversés face à un même signal de commande.



La commande des différents interrupteurs à l'aide d'un module Arduino s'effectue à l'aide de deux sorties analogiques. Pour exemple, le programme suivant permet de changer le sens de rotation du moteur toutes les 3 s :

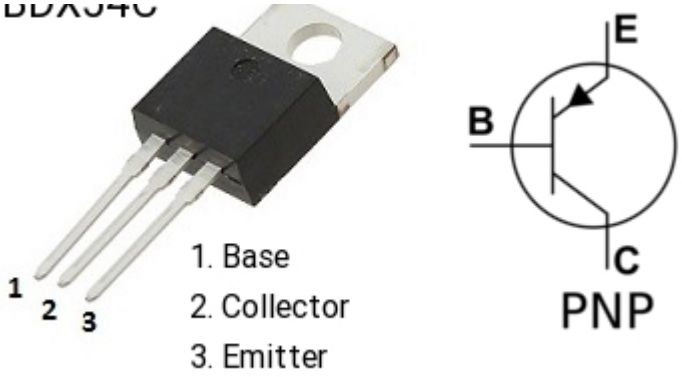
> Notez les numéros des broches pour réaliser correctement les branchements.

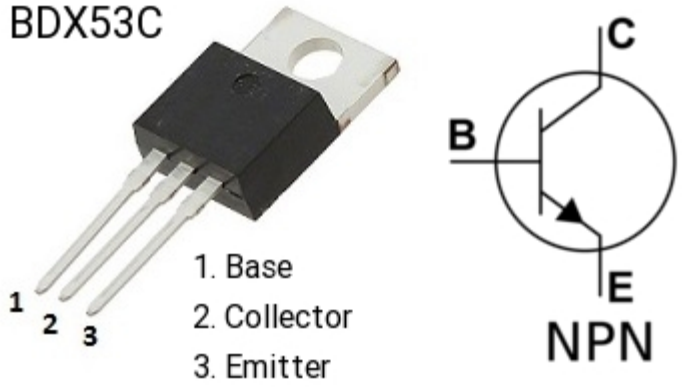
Entrée[ ]:
```
%%ecrire_croquis $croquis

const int com_m1=3;
const int com_m2=5;

// the setup routine runs once when you press reset
void setup()
{
// initialize the digital pin as an output
pinMode(com_m1,OUTPUT);
pinMode(com_m2,OUTPUT);
}

// the loop routine runs over and over again forever
void loop()
{
digitalWrite(com_m2,LOW);
digitalWrite(com_m1,HIGH);
delay(3000);
digitalWrite(com_m1,LOW);
digitalWrite(com_m2,HIGH);
delay(3000);
}
```

Entrée[ ]:
```
jupyno.televerser(croquis, port, FQBN)
```

Il reste à faire le montage ! Pour connecter les transistors utilisés (PNP et NPN), **de type PNP pour les interrupteurs du haut (1 et 2) et NPN pour les deux interrupteurs du bas (3 et 4)**, voici leurs caractéritiques :

BDX54C





Réaliser le montage.

> Vous devriez voir le moteur tourner puis changer de sens toutes les 3 s.

Un véhicule autonome qui fait un créneau

Deux petits véhicules (jouets) sont à votre disposition. Le moteur avant permet de tourner les roues à gauche ou à droite : il devra être piloté en sens. Le moteur arrière permet de faire avancer ou reculer le véhicule : il devra être piloté en vitesse.

Réaliser le pilotage du petit véhicule (jouet) à disposition pour lui faire effectuer un créneau depuis le programme d'un module Arduino.

Type *Markdown* and LaTeX: α^2

TP courbes intensité - potentiel

PE. LEROY

> Capacités exigibles du programme :
>
> **Électrochimie**
>
> - Mettre en \oe uvre un dispositif à trois électrodes pour tracer des courbes courant-potentiel.

Liste du matériel :

- Solution de sulfate de fer (II) à 1×10^{-1} mol. L^{-1} (200 mL par groupe)
- Solution de sulfate de fer (III) à 1×10^{-1} mol. L^{-1} (200 mL par groupe)
- Solution d'acide sulfurique \chm{H_2SO_4} à 1 mol. L^{-1} (50~mL par groupe)
- Eau distillée
- Éprouvettes graduées de 100 mL et 10 mL
- Béchers de 250 mL et 100 mL ($\times 2$)
- Électrodes de platine ($\times 2$)
- Électrode de référence au calomel saturé
- Interface d'acquisition
- Fils
- Résistance de $\sim 50\ \Omega$
- Électrolyseur
- Générateur de tension/courant variables
- Tube à essai (en plastique si possible) ($\times 2$)
- Allumettes

On commence par importer les bibliothèques nécessaires :

> La ligne suivante `%matplotlib notebook` est une commande magique qui sert à l'affichage des graphes dans le notebook.

Entrée[ ]:
```
%matplotlib notebook
```

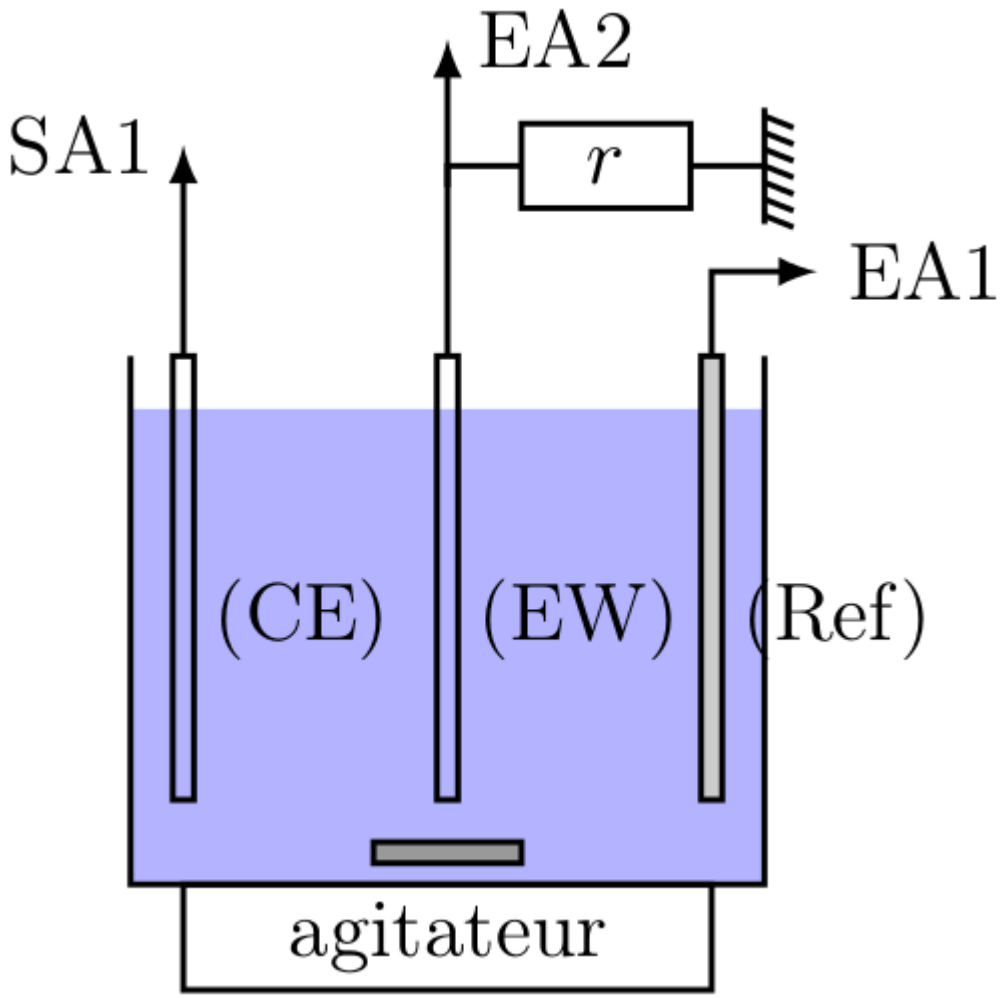
Entrée[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Nous allons tracer des courbes i-E pour une électrode de platine plongeant dans différentes solutions, que l'on va faire fonctionner en anode et en cathode, de façon à tracer les courbes i-E complètes.

Montage à trois électrodes

On utilisera ici (malheureusement car très chère et propriétaire) une interface d'acquisition, de marque Eurosmart, qui est la SYSAM SP5.

Le montage est le suivant :



Le système SYSAM peut émettre, entre la sortie `SA1` et la masse une tension sous la forme d'une rampe. On peut utiliser cette rampe pour générer une tension entre la contre-électrode et l'électrode de travail. L'interface d'acquisition va nous permettre de mesurer le courant traversant l'électrode de travail ainsi que son potentiel.

La voie `SA1` branchée sur la contre-électrode (CE) permet d'imposer la rampe de tension. La voie `EA1` mesure la différence de potentiel entre l'électrode de référence (Ref) et la masse, et la voie `EA2` la différence de potentiel entre l'électrode de travail (EW) et la masse, soit aussi aux bornes de la résistance r.

On a donc :

$$E_{(\mathrm{EW})} - E_{(\mathrm{Ref})} = u_{(\mathrm{EA2})} - u_{(\mathrm{EA1})}$$

D'où :

$$E_{(\mathrm{EW})} = u_{(\mathrm{EA2})} - u_{(\mathrm{EA1})} + E_{(\mathrm{Ref})}$$

De même :

$$i = -\frac{u_{(\mathrm{EA2})}}{r}$$

La voie `EA1` permet donc de mesurer le potentiel de l'électrode de travail (EW). La voie `EA2` permet donc de mesurer le courant passant dans la résistance r et donc dans l'électrode de travail. On choisira pour $r \simeq 50\ \Omega$.

Réaliser le montage.

> Le bécher est pour le moment vide...

Voici les réglages à effectuer pour l'acquisition :

Lancer le logiciel Latis Pro, puis cliquer sur l'icone 
pour sélectionner les voies utiles et accéder aux paramètres de l'acquisition. Choisir un nombre de points égale à 400 pour une durée totale d'acquisition de 40 s.

Cliquez ensuite sur l'icone 
. Décocher `GBF` et cocher `Sortie active` . Choisir la forme `Rampe` . On fera varier la tension entre -5 et 5 volts. On ne fera l'expérience que sur une seule période. Lorsque l'on souhaitera lancer l'acquisition, on appuiera sur `F10` .

Réaliser les réglages.

Mise en évidence du mur du solvant

Pour mettre en évidence le mur du solvant, on utilisera une solution acidifiée pour que la concentration en ions H^+ soit significative et les courbes bien visibles.

Mélanger 100 mL d'eau distillée et de 10 mL de la solution d'acide sulfurique. Réaliser l'acquisition.

Pour visualiser les courbes i-E nous allons exporter les données puis les traiter dans ce notebook. Pour cela aller dans le menu `Fichier` et choisir `Exporter` . On enregistrera les variables `EA1` et `EA2` uniquement (dans la fenêtre `Courbes disponibles` choisir une courbe puis glisser-déplacer cette courbe). Choisir le point `.` comme séparateur décimal et le point virgule `;` comme séparateur dans le fichier au format `CSV` , appelé `donnees.csv` .

On peut alors importer les données :

Entrée[ ]:
```
# Nom du fichier (à placer dans le même répertoire que ce notebook)
nomfichier='donnees.csv'

# Importation
EA1,EA2=np.char.replace(np.loadtxt(nomfichier,skiprows=1,unpack=True,
```

Il faut maintenant créer les nouvelles grandeur i et E :

$$i = -\frac{u_{(\mathrm{EA2})}}{r}$$

et :

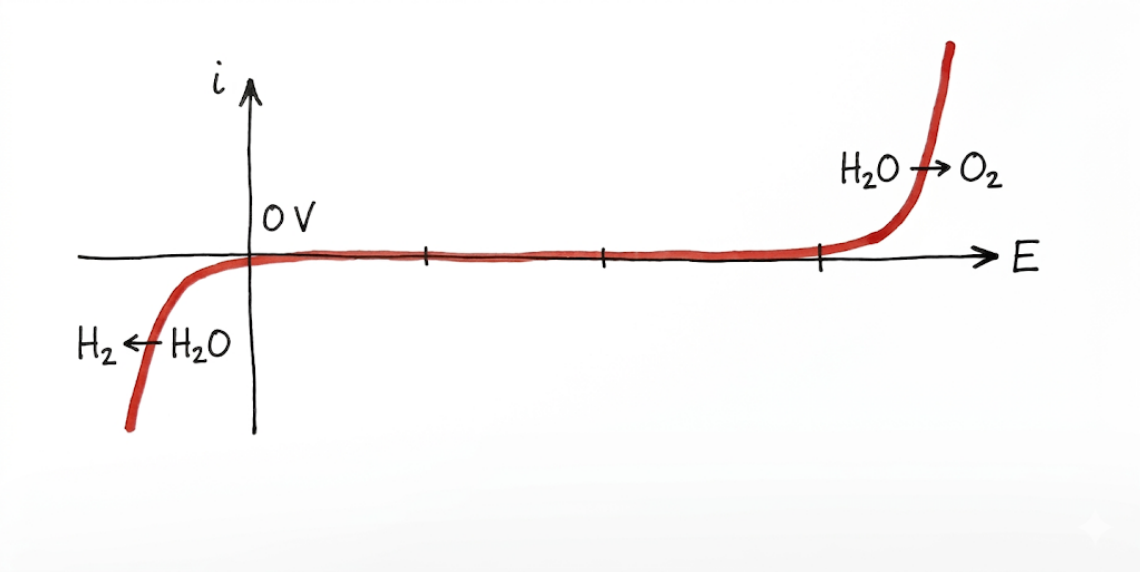
$$E = u_{(\mathrm{EA2})} - u_{(\mathrm{EA1})} + E_{(\mathrm{Ref})}$$

Entrée[ ]:
```
# Nécessaire ?
EA1=np.array(EA1)
EA2=np.array(EA2)

# Paramètres
r=47 # Ohm
Eref=0.248 # V

# Nouvelles grandeurs
i=-EA2/r
E=EA2-EA1+Eref
```

On peut tracer la courbe i-E :

Entrée[ ]:
```
plt.figure()

# Réglages
parameters = {"axes.labelsize": 13, "xtick.labelsize": 13, "ytick.lab
plt.rcParams.update(parameters)

plt.plot(E,i,'b-')

plt.xlabel('E (V)')
plt.ylabel('i (A)')
plt.title('Courbe i-E')

# Sauvegarde de la courbe
plt.savefig('courbe_i-E.png')

plt.show()
```

Vous devriez obtenir ce genre de courbe :



On observe bien la courbe théorique à laquelle on peut s'attendre. Noter les valeurs des potentiels pour lesquels le courant apparait à l'oxydation et à la réduction, y a-t-il des surtensions anodique et cathodique ?

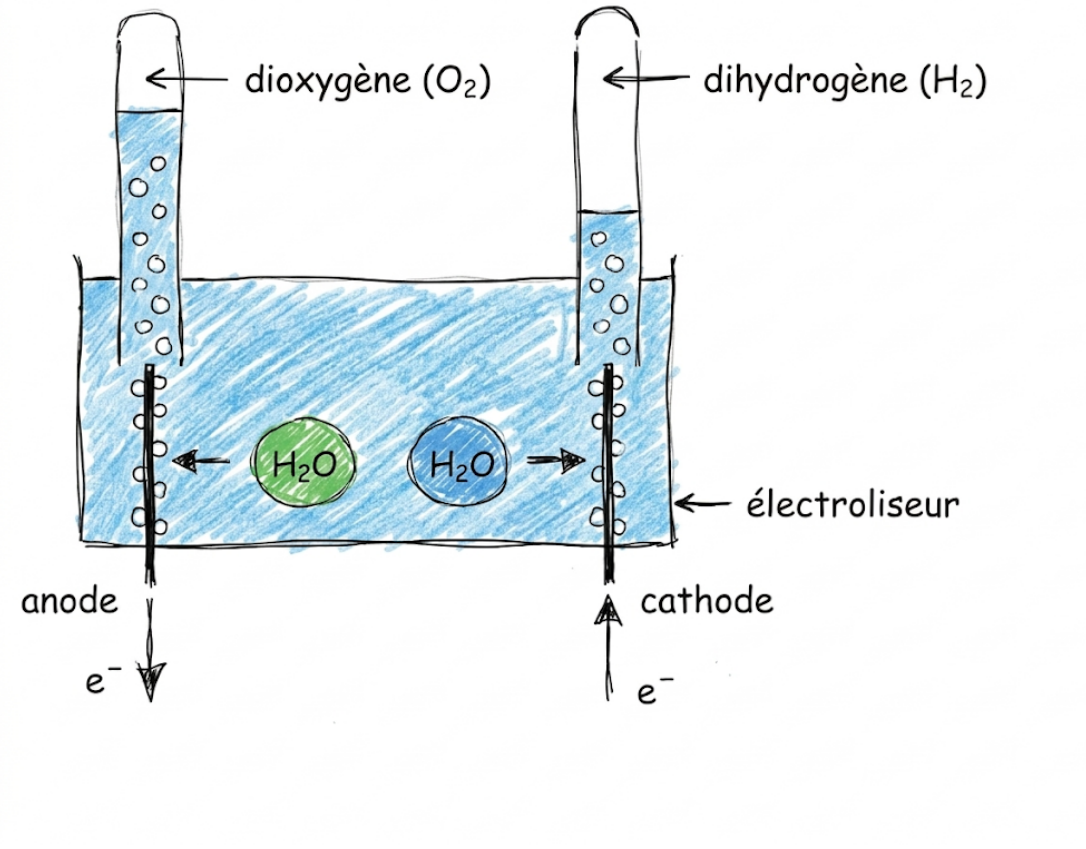
> On donne : $E^\circ(O_2/H_2O) = 1,23\,V$; $E^\circ(H^+/H_2) = 0,00\,V$.

Déterminer les surtensions anodique et cathodique (si elle existent) :

Entrée[ ]:
```
eta_a= # V
eta_c= # V
```

Électrolyse de l'eau

Profitons-en pour effectuer l'électrolyse de l'eau acidifiée :



La tension minimale à imposer se détermine d'après la courbe i-E précédente :

Entrée[ ]:
```
umin= # V
```

La réaction qui se produit est :

$$\mathrm{H_2O}(\ell) = \frac{1}{2}\mathrm{O_2}(g) + \mathrm{H_2}(g)$$

si bien que 2 fois plus de dihydrogène se forme que de dioxygène.

> ATTENTION un mélange $\mathrm{O_2}(g)$ et $\mathrm{H_2}(g)$ dans ces proportions est très EXPLOSIF, ne jamais faire cette électrolyse dans le même tube à essai.

Réaliser le montage et effectuer l'électrolyse. On caractérisera les gaz formés.

> $\mathrm{O_2}(g)$ rallume une allumette qui vient d'être éteinte et $\mathrm{H_2}(g)$ brule avec un bruit caractéristique << plop >>.

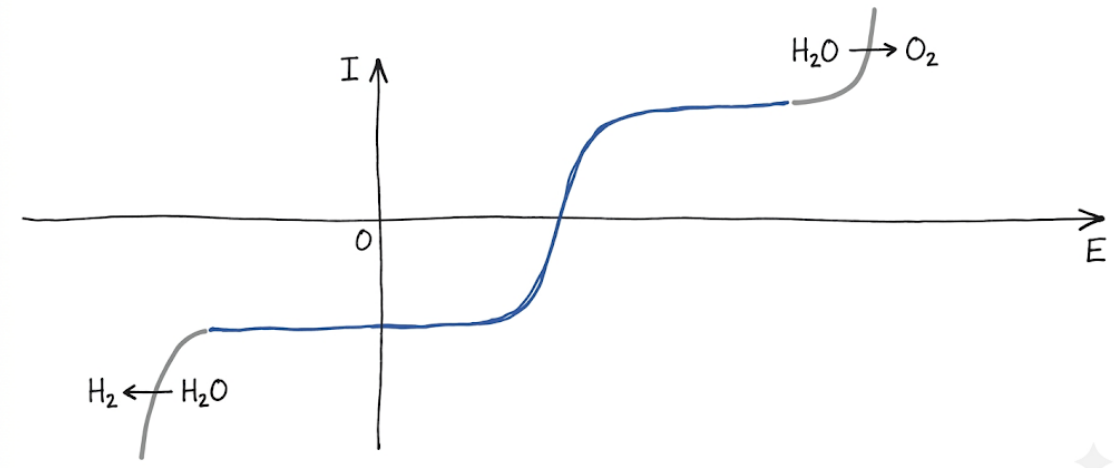
Tracé de la courbe $i - E$ du couple $\mathrm{Fe^{3+}/Fe^{2+}}$

On commence par tracer la courbe du couple $\mathrm{Fe^{3+}/Fe^{2+}}$, pour cela on prendra 50 mL de chaque solution contenant l'ion en question, le tout additionné de 10 mL de la solution d'acide sulfurique.

Réaliser l'expérience et reprendre l'analyse précédente :

Entrée[ ]:
```
plt.figure()

# Réglages
parameters = {"axes.labelsize": 13, "xtick.labelsize": 13, "ytick.lab
plt.rcParams.update(parameters)

plt.plot(E,i,'b-')

plt.xlabel('E (V)')
plt.ylabel('i (A)')
plt.title('Courbe i-E')

# Sauvegarde de la courbe
plt.savefig('courbe_i-E.png')

plt.show()
```

Vous devriez obtenir ce genre de courbe :



On observe que le couple est bien rapide à l'oxydation et à la réduction.

> Vérifiez alors que le potentiel de courant nul est bien le potentiel standard du couple $E^\circ(\mathrm{Fe^{3+}/Fe^{2+}}) = 0,77$ V.

On observe aussi deux paliers de diffusion.

Expliquer à votre binôme pourquoi :). Déterminer les courants anodique et cathodique limites :

Entrée[ ]:
```
ia_lim= # A
ic_lim= # A
```

Influence de divers paramètres

Influence de la convection

Garder le mélange précédent et y placer un agitateur magnétique :

Entrée[ ]:
```
plt.figure()

# Réglages
parameters = {"axes.labelsize": 13, "xtick.labelsize": 13, "ytick.lab
plt.rcParams.update(parameters)

plt.plot(E,i,'b-')

plt.xlabel('E (V)')
plt.ylabel('i (A)')
plt.title('Courbe i-E')

# Sauvegarde de la courbe
plt.savefig('courbe_i-E.png')

plt.show()
```

Déterminer les nouveaux courants anodique et cathodique limites :

Entrée[ ]:
```
ia_lim2= # A
ic_lim2= # A
```

Commenter l'influence de l'agitation sur les courants limites.

Influence de la concentration

Prendre un volume de chaque solution de façon à obtenir le même volume que précédemment (100 mL), le tout additionné de 10 mL de la solution d'acide sulfurique et recommencer l'acquisition :

Entrée[ ]:
```
plt.figure()

# Réglages
parameters = {"axes.labelsize": 13, "xtick.labelsize": 13, "ytick.lab
plt.rcParams.update(parameters)

plt.plot(E,i,'b-')

plt.xlabel('E (V)')
plt.ylabel('i (A)')
plt.title('Courbe i-E')

# Sauvegarde de la courbe
plt.savefig('courbe_i-E.png')

plt.show()
```

Déterminer les nouveaux courants anodique et cathodique limites :

Entrée[ ]:
```
ia_lim3= # A
ic_lim3= # A
```

Commenter l'influence de la concentration sur les courants limites.

Type *Markdown* and LaTeX: α^2

TP phénomènes de corrosion humide

PE. LEROY

> Capacités exigibles du programme :
>
> **Électrochimie**
>
> - Mettre en \oe uvre des piles et des électrolyseurs.
>
> **Mesures de grandeurs électriques : conductance-conductivité, tension électrique, intensité du courant**
>
> - Mettre en oeuvre des mesures électriques dans un environnement chimique et électrochimique.

Liste du matériel :

- Solution de chlorure de sodium \mathrm{Na Cl} à $1\ \mathrm{mol.L^{-1}}$ (100~mL par groupe)
- Solution de sulfate de fer (II) à $1\ \times\ 10^{-1}\ \mathrm{mol.L^{-1}}$ (100 mL par groupe)
- Solution de sulfate de fer (II) à $1\ \times\ 10^{-2}\ \mathrm{mol.L^{-1}}$ (100 mL par groupe)
- Solution électrolytique préparée en dissolvant dans de l'eau distillée, pour 200 mL de solution, $8,2$ g de sulfate de zinc, $5,0$ g de chlorure de potassium et $2,0$ g d'acide borique. Le pH doit être compris entre 4,5 et 6 (200 mL par groupe)
- Solution de ferricyanure de potassium à $0,1\ \mathrm{mol.L^{-1}}$ en flacon compte-gouttes
- Solution de phénolphtaléine en flacon compte-gouttes
- Pont salin
- Eau distillée
- Pipette plastique
- Béchers de 250 mL ($\times 4$)
- Plaques de cuivre ($\times 1$), fer ($\times 2$) et zinc ($\times 1$)
- Supports pour plaques métalliques ($\times 2$)
- Multimètre Metrix (rouge)
- Pinces << crocodile >>
- Fils
- Générateur de tension variable (électrolyse)
- Balance de précision
- Papier de verre (ponçage des plaques)

On commence par importer les bibliothèques nécessaires :

> La ligne suivante `%matplotlib notebook` est une commande magique qui sert à l'affichage des graphes dans le notebook.

Entrée[ ]:
```
%matplotlib notebook
```

Entrée[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
```

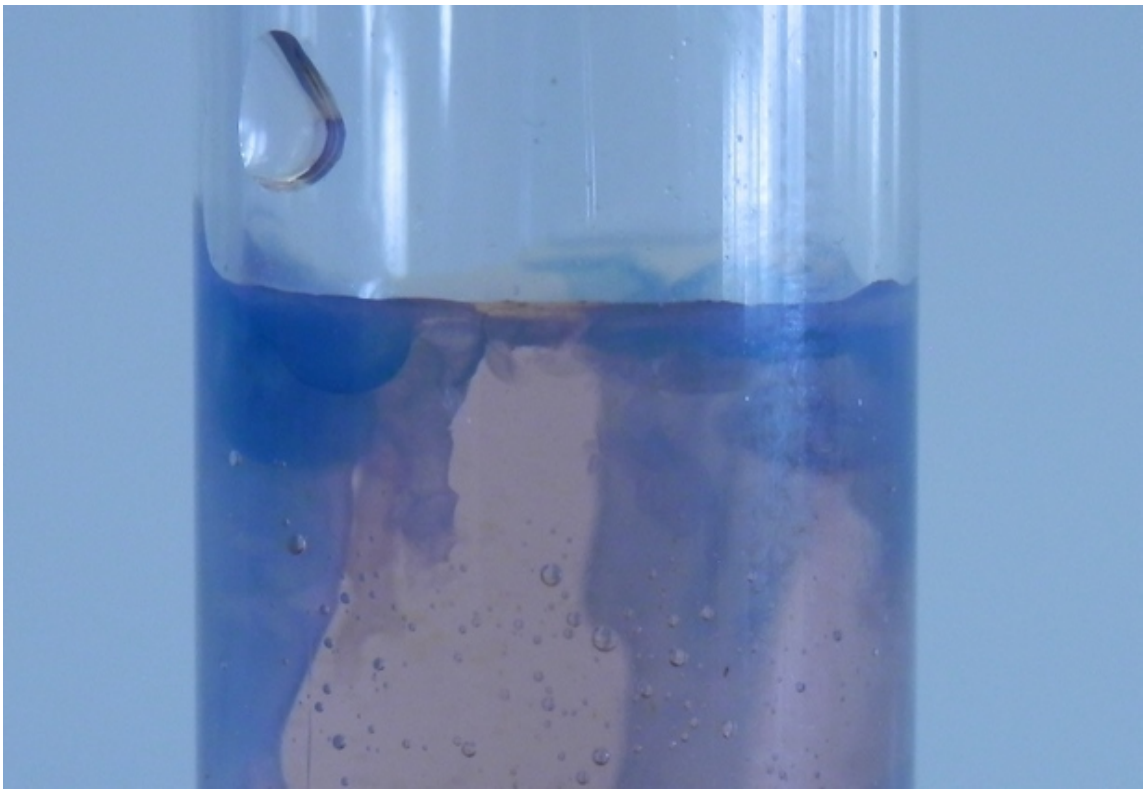
Nous allons créer différentes piles, plus particulièrement des **piles de corrosion** de façon à faire le lien avec les phénomènes de corrosion en milieu humide.

On donne :

- $E^\circ(O_2/H_2O) = 1,23$ V
- $E^\circ(Cu^{2+}/Cu) = 0,34$ V
- $E^\circ(H^+/H_2) = 0,00$ V
- $E^\circ(Fe^{2+}/Fe) = -0,44$ V
- $E^\circ(Zn^{2+}/Zn) = -0,76$ V

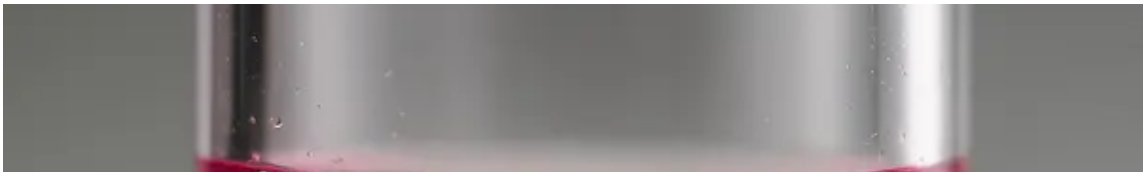
L'hexacyanoferrate (III) de potassium (ou ferricyanure de potassium) $[Fe(CN)_6]^{3-}(aq) + 3\ K^+(aq)$ peut être utilisé pour mettre en évidence la **présence d'ions** Fe^{2+} en donnant un précipité bleu dit << bleu de Prusse >> :

$$2[Fe(CN)_6]^{3-}(aq) + 3Fe^{2+}(aq) = Fe_3[Fe(CN)_6]_2(s).$$



La phénolphtaléine vire au rose **en milieu basique**.

> Un milieu basique est caractérisé par une concentration moindre en ions H^+ et donc plus grande en ions HO^- puisque $K_e = [H^+][HO^-]$ est une constante à T fixée.



Piles de corrosion

Contacts de deux métaux différents (corrosion galvanique)

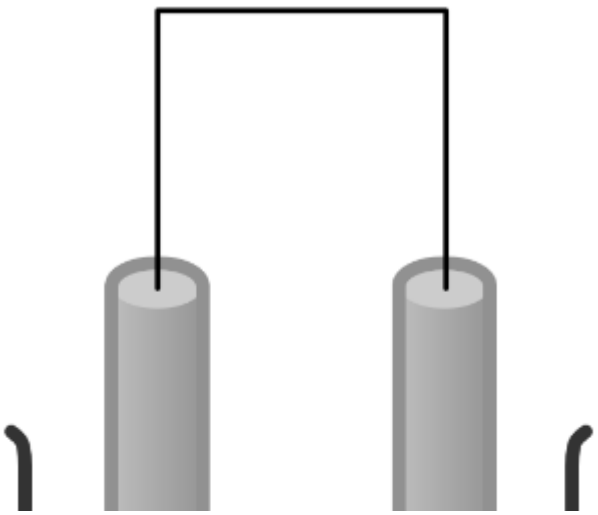
Réaliser le protocole suivant :

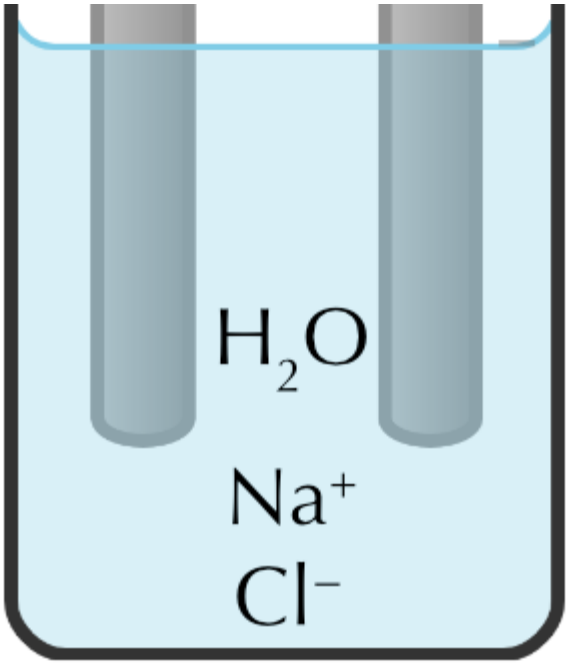
- prendre une plaque de cuivre et une plaque de fer et les installer sur le support au dessus du bécher ;
- remplir le bécher avec la solution de chlorure de sodium de façon à ce que les électrodes trempent dedans ;
- mesurer au voltmètre la différence de potentiels, et identifier le pôle + ;
- enlever le voltmètre et le remplacer par un simple fil (court-circuiter la pile) ;
- faire les tests de présence d'ions Fe^{2+} et de basicité **au niveau de la surface des électrodes**.

> ATTENTION quelques gouttes suffisent !

Reproduire (au brouillon) le schéma suivant et le compléter avec :

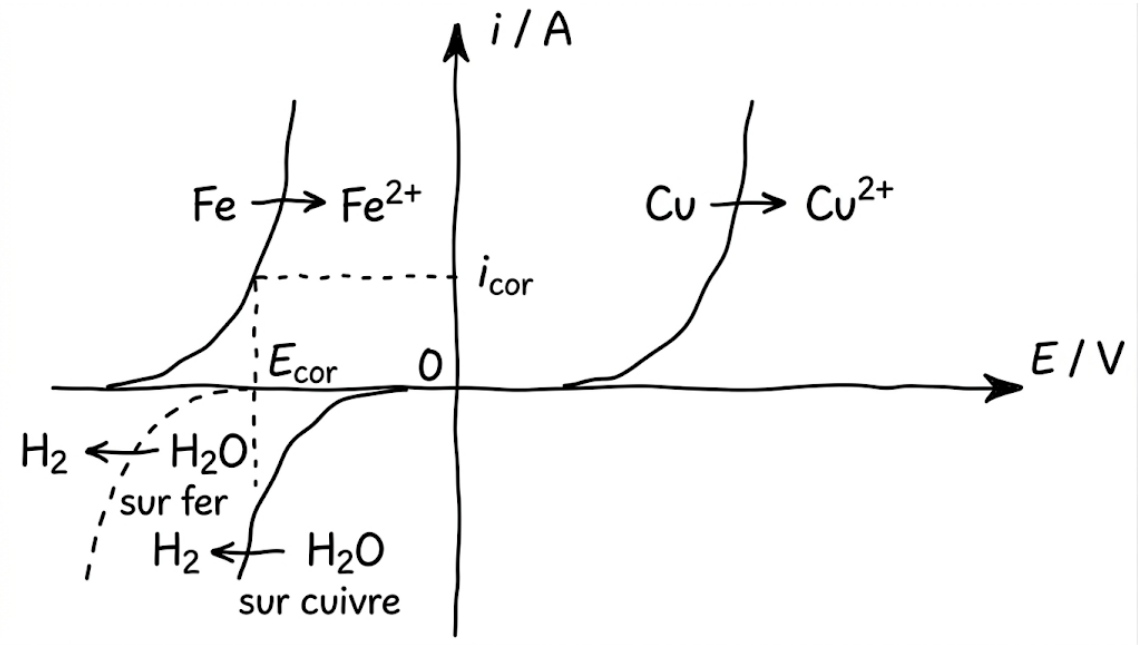
- **le pôle + et le pole -** ;
- **le sens du courant et le circulation des électrons** ;
- **la cathode et l'anode** ;
- **les réactions qui se déroulent à priori aux électrodes**.





Reproduire un diagramme de potentiels standards et expliquer à votre binôme comment justifier une partie des observations expérimentales, notamment la tension à vide observée.

Compléter vos explications avec le schéma suivant :

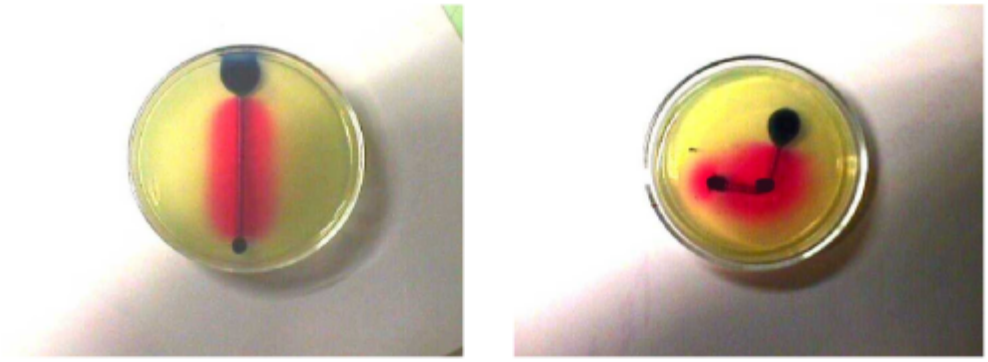


Entrée[ ]:
```
# Tension expérimentale
u=
# Potentiels standards des couples mis en jeu
Eo1=
Eo2=
# Comparaison
print(u,Eo2-Eo1)
```

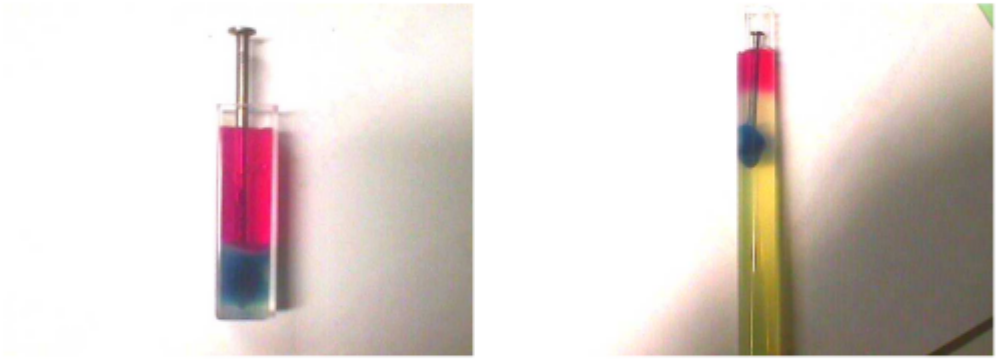
Pièce métallique inhomogène

Observer et analyser les résultats des expériences de corrosion différentielle de

clous possédant des zones ayant été soumises à des contraintes (zones << d'écrouissage >>) :



Corrosion d'un clou possédant des zones « d'écrouissage ».



Corrosion d'un clou possédant une zone à l'air libre.

Corrosion d'un clou possédant des zones protégées par un fil de zinc puis une galvanisation (zinc).

> La solution est un gel d'agar-agar contenant du \chm{NaCl}, du ferricyanure de potassium et de la phénolphtaléïne.

Piles de concentration

Réaliser une pile à l'aide d'un même métal (prendre le fer) plongeant dans deux solutions du cation métallique correspondant mais de concentrations différentes.

> On prendra comme pont salin une bandelette imbibée d'une solution de nitrate de potassium.

Réaliser (au brouillon) l'étude du système sur le même principe que précédemment. On pourra notamment prévoir la veleur de la tension à vide d'après les expressions des potentiels aux électrodes (loi de Nernst).

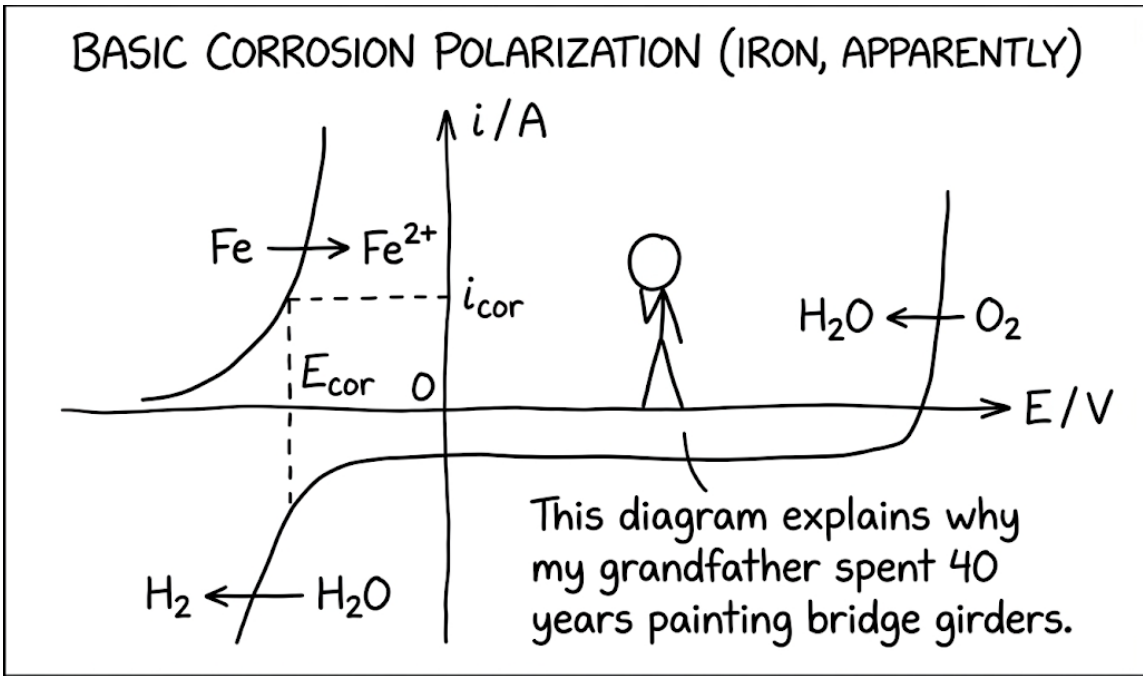
Pile d'Evans dite << d'aération différentielle >>

Réaliser une pile à l'aide d'un même métal (prendre le fer) plongeant dans deux solutions conductrices << d'aérations >> différentes.

> On prendra comme solution conductrice de l'eau salée. On aèrera l'une des solutions à l'aide d'une pipette plstique remplie d'air. On utilisera deux tests pour déterminer les réactions qui se déroulent à chaque électrode.

Réaliser (au brouillon) l'étude du système sur le même principe que précédemment.

> On pourra comparer avec l'expérience présentée sur l'un des photos précédents où un clou en fer est immergé aux $3/4$ dans un gel d'agar-agar contenant du NaCl, du ferricyanure de potassium et de la phénolphtaléïne.
>
> On pourra s'aider de la courbe suivante :



Protection contre la corrosion

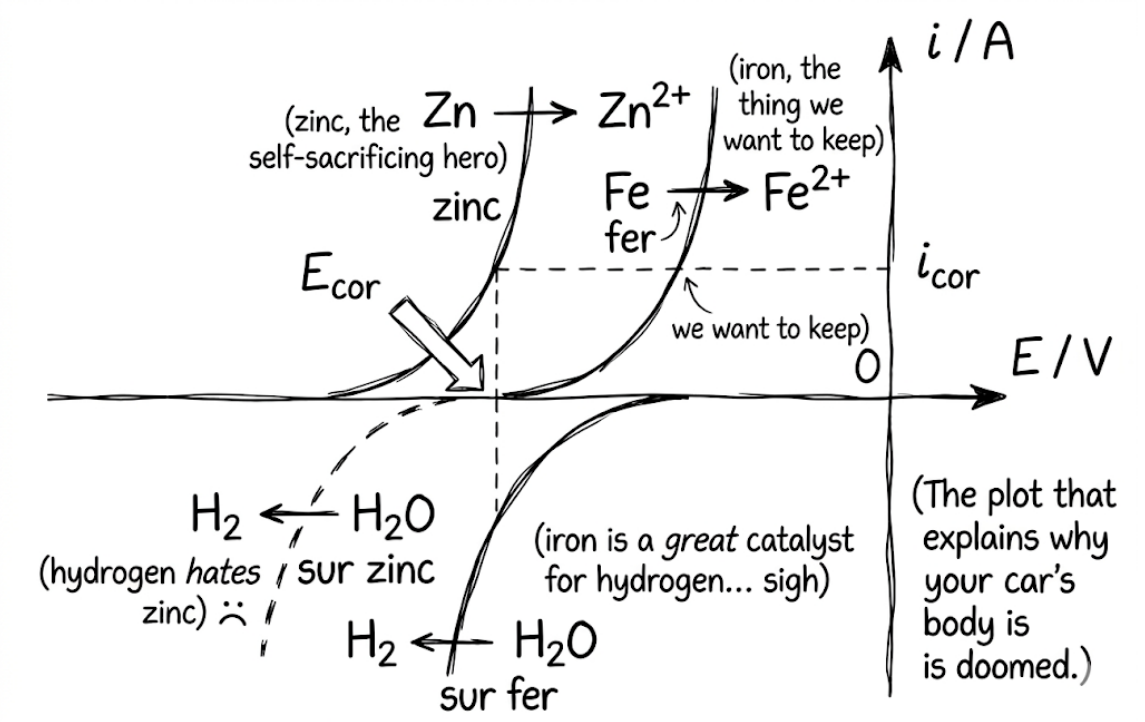
Protection cathodique par << anode sacrificielle >>

Réaliser une expérience qui réalise une protection cathodique par << anode sacrificielle >> d'une lame de fer.

Réaliser (au brouillon) l'étude du système sur le même principe que

précédemment.

> On pourra comparer avec l'expérience de la corrosion du clou en fer protégé par un fil de zinc.
>
> On pourra s'aider de la courbe suivante :



Protection par << revêtement >>

Réaliser une électrolyse permettant de réaliser une protection par revêtement d'une lame de fer par une couche de zinc (électrozingage).

> On utilisera une intensité de ~ 1 A pendant 15 min. On mesurera la masse de zinc déposée expérimentalement.
>
> > Bien peser la lame de fer au début !

Déterminer (au brouillon) le rendement faradique de l'électrolyse.

> Montrer que :
>
> $$> \frac{n_{e^-,utiles}}{2} = \frac{n_{Zn}}{1} = \frac{m_{Zn}}{M_{Zn}} >$$
>
> soit :
>
> $$> I = \frac{\Delta Q}{\Delta t} = \frac{\frac{n_{e^-,utiles}}{R_q} F}{\Delta t} >$$
>
> avec R_q le rendement faradique que l'on peut ainsi déterminer. On donne : $M(\mathrm{Zn}) = 65,4\ \mathrm{g.mol^{-1}}$ et $F = 96500\ \mathrm{C.mol^{-1}}$.

Entrée[ ]:
```
# À mesurer
mZn= # g

# Données
I=1 # A
Dt=15*60 # s
MZn=65,4 # g/mol
F=96500 # C/mol

# Calcul
Rq=
```

Type *Markdown* and LaTeX: α^2

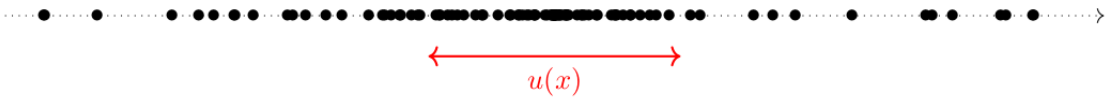
Incertitudes de mesure

PE. LEROY

Quelques définitons

La quantification de la **variabilité** d'une mesure x d'une grandeur est appelée **incertitude-type** et notée $u(x)$. Par définition, l'incertitude-type correspond à **l'écart-type** de la distribution des données issues d'une répétition de la mesure *dans les mêmes conditions*.

> L'incertitude-type permet de quantifier la variabilité d'une mesure. Ainsi, deux mesures x_1 et x_2 issues du même processus sont séparées en moyenne de quelques u(x) par construction de l'incertitude-type en tant qu'écart-type.



On notera un résultat sous la forme :

$$\boxed{m = x \pm u(x)}$$

On gardera systématiquement 2 chiffres significatifs pour l'affichage de l'incertitude-type $u(x)$, et on affichera le résultat m avec un nombre de chiffres significatifs compatible.

> On sort du paradigme << *nombre de chiffres significatifs* = *incertitude* >>. Le nombre de chiffres significatifs ne sert plus qu'à améliorer la lisibilité du résultat.
>
> Ainsi, on pourra écrire : $m = 1,23 \pm 0,28$ cm ou $m = 0,0123 \pm 0,0028$ m, etc.

Estimation du résultat d'une mesure et de l'incertitude-type

Expériences sans variabilité observée (incertitudes de type B)

Certaines expériences n'ont pas de variabilité observée. Cela signifie qu'en reproduisant la mesure, on retrouve systématiquement le même résultat. C'est par exemple le cas lorsque l'on mesure naïvement la taille d'un objet avec la même règle graduée.

Cette absence de variabilité observée n'implique pas une absence de variabilité. Cela signifie juste qu'à l'échelle de cette expérience, avec l'appareil de mesure choisi, la

variabilité est plus faible que la précision de la mesure.

> Ce phénomène n'est pas uniquement lié à l'appareil de mesure. En effet, selon les conditions expérimentales, il n'est parfois pas matériellement possible (ou souhaité) de reproduire le processus de mesure. Dans ce cas, une seule valeur est accessible et il faut tout de même estimer son incertitude-type.

Lors d'une **mesure sans variabilité observée** :

- On estime la valeur mesurée x avec la **graduation la plus proche** (ou tout simplement la valeur affichée par l'appareil) ;
- On estime la **demi-largeur** $\Delta_{1/2}$ de la plus petite plage dans laquelle on est certain de trouver la valeur recherchée, et on estime **l'incertitude-type** $u(x)$ par :

$$\boxed{u(x) = \frac{\Delta_{1/2}}{\sqrt{3}}}$$

> Ce facteur $1/\sqrt{3}$ correspond à l'écart-type d'une distribution unitaire uniforme.

Ci-dessous un exemple :

Entrée[23]:
```
import numpy as np

demi_largeur=0.28

print('u(x)=',demi_largeur/np.sqrt(3))
```

```
u(x)= 0.16165807537309523
```

Le résultat de la mesure est bien sûr toujours :

$$m = x \pm u(x)$$

> L'intervalle $\Delta_{1/2}$ doit être pris le plus faible possible selon les critères personnels de l'expérimentateur et selon les conditions de l'expérience. Il ne doit pas y avoir de règle générale. Par exemple avec une règle graduée au millimètre, si la valeur tombe directement sur une graduation, il est naturel de prendre $\Delta_{1/2} = 0,25$ mm, tandis que si la valeur est entre deux graduations, on prendra plus logiquement $\Delta_{1/2} = 0,5$ mm. Et enfin, un étudiant peu sûr de lui peut choisir de prendre dans le même cas $\Delta_{1/2} = 1$ mm.
>
> Pour les appareils de mesure numérique, il est nécessaire de consulter la notice de l'appareil. Toutefois, bien souvent, les notices ne précisent pas clairement la nature de la valeur de la précision fournie (est-ce une incertitude-type ? Un intervalle ? Un écart-type d'une distribution gaussienne ?). Dans ce cas, on suppose que l'incertitude affichée sur la notice est un intervalle $\Delta_{1/2}$ de certitude de trouver la mesure.

Expériences avec variabilité observée (incertitudes de type A)

Lorsque la variabilité des mesures est accessible, il convient de répéter un grand nombre de fois le processus mesure pour estimer l'incertitude-type sur une unique réalisation de la mesure.

L'écart-type est alors assimilé à l'incertitude-type :

$$\boxed{u(x) = \sigma}$$

On peut calculer l'écart-type d'un ensemble de mesures à l'aide du langage Python. Ci-dessous un exemple :

Entrée[24]:
```
import numpy as np

valeurs=[
1,2,3,4
]

ecarttype=np.std(valeurs)

print('sigma=',ecarttype)
```

```
sigma= 1.118033988749895
```

Pour gagner en précision, nous pouvons utiliser les différents points de mesures effectués pour aller plus loin qu'une simple estimation de l'incertitude-type sur une mesure unique : **utiliser la moyenne des mesures** $\overline{x}$ comme résultat de mesure.

Par contre, l'incertitude-type sur la moyenne des valeurs $u(\overline{x})$ n'est pas égale à l'incertitude-type sur chaque valeur $u(x)$ (obtenue en calculant l'écart-type sur la distribution de ces valeurs).

Ainsi le résultat s'écrira :

$$\boxed{m = \overline{x} \pm u(\overline{x})}$$

Il existe une formule mathématique permettant d'estimer cet écart-type $u(\overline{x})$ pour N mesures :

$$u(\overline{x}) = \frac{u(x)}{\sqrt{N}} = \frac{\sigma}{\sqrt{N}}$$

> En toute rigueur il faudrait remplacer N par $N - 1$, sinon un biais est introduit.

Ainsi, en une série de mesure, on obtient les points expérimentaux, leur incertitude-type, la moyenne de ces points et grâce à cette formule, l'incertitude-type sur la moyenne. On obtient ainsi une estimation plus précise de la grandeur à mesurer en modérant la variabilité de chaque prise de mesure unique.

Ci-dessous un exemple :

Entrée[25]:
```
import numpy as np

valeurs=[
1,2,3,4
]

moyenne=np.average(valeurs)
ecarttype=np.std(valeurs)

print('x=',moyenne,'u(x)=',ecarttype/np.sqrt(len(valeurs)))
```

```
x= 2.5 u(x)= 0.5590169943749475
```

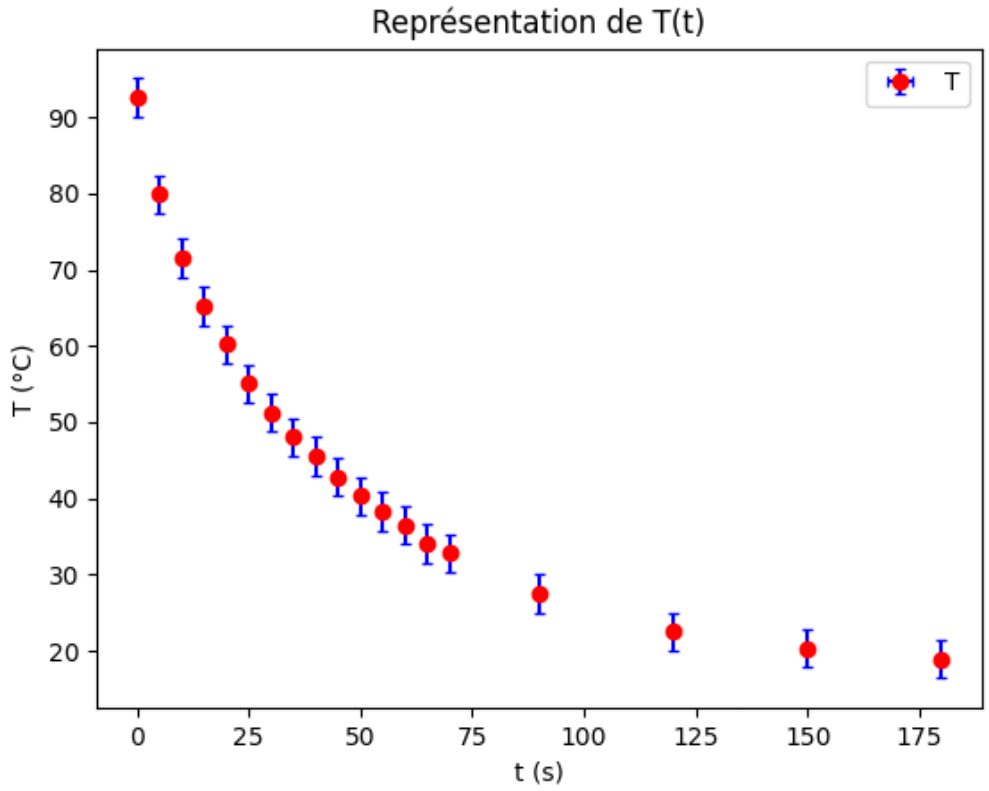
Ajouter des barres d'incertitudes-type sur un graphe

Ci-dessous un exemple **que vous pouvez adapter** en fonction de vos besoins :

> La ligne suivante sert à l'affichage des graphes dans le notebook.

Entrée[26]:
```
#%matplotlib notebook
```

Entrée[27]:
```
import matplotlib.pyplot as plt

# Listes des valeurs à traiter
liste_T=[92.6,79.9,71.5,65.2,60.2,55.1,51.2,48,45.5,42.8,40.3,38.3,36
liste_t=[0,5,10,15,20,25,30,35,40,45,50,55,60,65,70,90,120,150,180]

# Incertitudes sur chaque mesure
incert_x=0 # Incertitude-type sur l'abscisse
incert_y=2.5 # Incertitude-type sur l'ordonnée

# Représentation graphique
plt.figure()
plt.errorbar(liste_t,liste_T,incert_y,incert_x,fmt='ro',ecolor='blue'

plt.title('Représentation de T(t)')
plt.xlabel('t (s)')
plt.ylabel('T (°C)')
plt.legend(('T'))
plt.savefig('courbe_barres_incertitudes.png')
plt.show()
```



Les incertitudes-type composées

Très souvent, la mesure expérimentale n'est pas le résultat recherché de l'expérience. Il faut souvent combiner des mesures entre elles pour obtenir le résultat souhaité.

Incertitude-type composée de type somme

Supposons que l'on calcule $x = \alpha x_1 + \beta x_2$. L'incertitude-type de x est alors donnée par :

$$\boxed{u(x) = \sqrt{(\alpha\, u(x_1))^2 + (\beta\, u(x_2))^2}}$$

Ci-dessous un exemple :

Entrée[28]:
```
import numpy as np

ux1=1.2
ux2=0.6

alpha=1
beta=1

ux=np.sqrt((alpha*ux1)**2+(beta*ux2)**2)

print('u(x)=',ux)
```

```
u(x)= 1.3416407864998738
```

Incertitudes-type composées de type produit

Supposons que l'on calcule $x = ax_1^{\alpha}x_2^{\beta}$. L'incertitude-type **relative** de x est alors donnée par :

$$\frac{u(x)}{x} = \sqrt{\left(\alpha\frac{u(x_1)}{x_1}\right)^2 + \left(\beta\frac{u(x_2)}{x_2}\right)^2}$$

Ci-dessous un exemple :

Entrée[29]:
```
import numpy as np

x1=20.3
x2=10.0

ux1=1.2
ux2=0.6

a=1
alpha=1
beta=1

x=a*x1**alpha*x2**beta
ux=x*np.sqrt((alpha*ux1/x1)**2+(beta*ux2/x2)**2)

print('x=',x,'u(x)=',ux)
```

```
x= 203.0 u(x)= 17.098315706525014
```

Incertitudes composées quelconques

Seules les deux formules précédentes sont à connaître, pour tous les autres cas, nous allons, à l'aide d'une simulation informatique utilisant une part d'aléatoire, calculer l'incertitude-type.

> On parle de **méthode Monte-Carlo**.

Le principe est de réaliser un tirage aléatoire de valeurs autour de la valeur mesurée, en faisant varier chaque paramètre selon une loi de distribution aléatoire définie. On détermine ensuite l'incertitude-type de la distribution de valeurs obtenues.

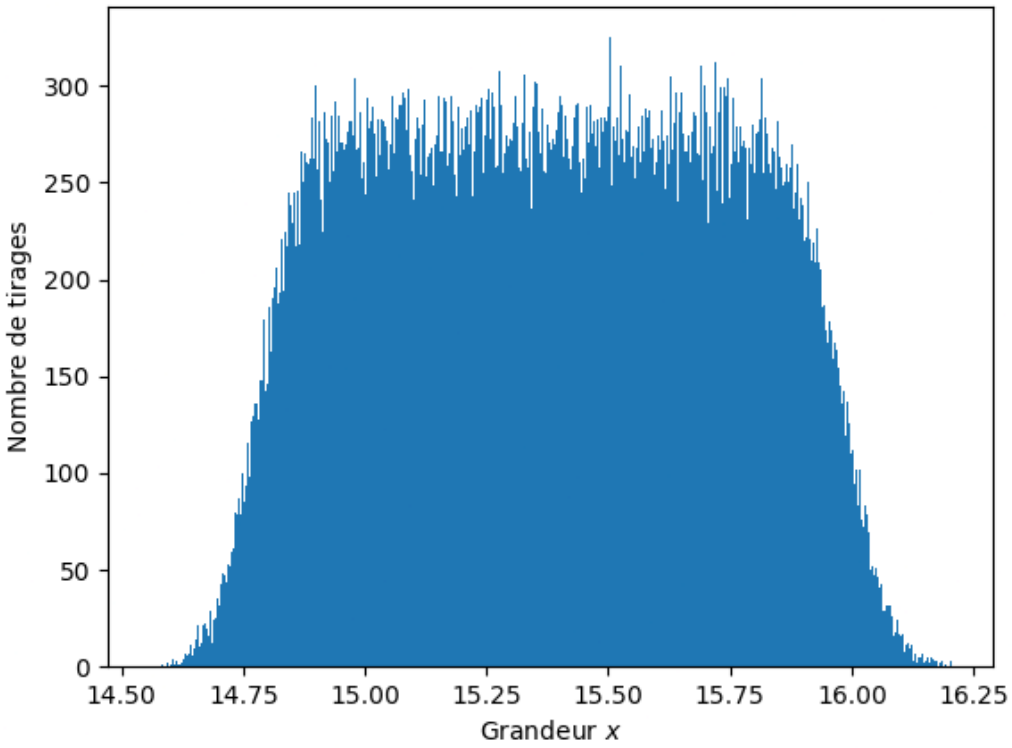
Ci-dessous un exemple **que vous pouvez adapter** en fonction de vos besoins :

> La ligne suivante sert à l'affichage des graphes dans le notebook.

Entrée[30]:
```
#%matplotlib notebook
```

Entrée[31]:
```
import numpy as np
import matplotlib.pyplot as plt
import random as rd

####### Partie à adapter ########
x1=1.10
x2=2.12

ux1=0.01
ux2=0.1

def x(x1,x2): # Définition de la loi permettant de déterminer x
    return 2*x1**3+6*x2

grandeur=x(x1,x2) # Calcul de m

def variations_aleatoires(): # Variation aléatoire de x (méthode Mont
    return x(x1+loi_normale(ux1),
             x2+loi_rectangulaire(ux2))

####### Partie à ne pas modifier ########
def loi_normale(sigma) : # Loi de distribution normale
    return np.random.normal(0,sigma)

def loi_rectangulaire(sigma): # Loi de distribution rectangulaire
    return rd.uniform(-sigma,sigma)

distribution=[] # Distribution des valeurs
N=100000 # Nombre de points utilisés dans la méthode Monte-Carlo

for i in range(N):
    distribution.append(variations_aleatoires())

incertitude_type=np.std(distribution) # Calcul de l'incertitude-type

print('x=',grandeur,'u(x)=',incertitude_type)

plt.figure()
plt.hist(distribution,bins=500)
plt.xlabel('Grandeur $x$')
plt.ylabel('Nombre de tirages')
plt.show()
```

```
x= 15.382000000000001 u(x)= 0.35444262202341204
```



La régression (ou ajustement) linéaire

Principe

Prenons des listes de mesures x et y avec leurs incertitudes. La régression (ou ajustement) linéaire est une opération mathématique qui consiste à trouver les meilleurs coefficients a et b tels que $ax_i + b$ soient les plus proches en moyenne des points de mesures y_i.

> En toute rigueur, il faudrait parler de régression affine et non pas linéaire.
>
> > Bah oui :/

Une régression (ou ajustement) linéaire permet de trouver la *meilleure droite* modélisant le mieux le comportement de ces points. Mathématiquement, on peut optimiser par un calcul dit **des moindres carrés** ce procédé.

> Contrairement à une idée répandue, le coefficient r^2 n'a aucun intérêt pour valider un modèle physique ou pour estimer des incertitudes-type. Pour vous convaincre que le coefficient r^2 ne permet en rien de juger une régression linéaire, on peut regarder la vidéo suivante (dans laquelle le coefficient `cor` correspond au r) : Vidéo (https://youtu.be/It4UA75z_KQ)

Détermination de l'incertitude-type sur les paramètres par méthode Monte-Carlo

On commence par réaliser une régression linéaire sans incertitudes sur les valeurs mesurées. La pente et l'ordonnée à l'origine obtenues sont le résultat numérique de la mesure.

Pour estimer leurs incertitudes-type, il faut de plus estimer pour chaque point de mesure la valeur de son incertitude-type. Ensuite, à l'aide d'une simulation Monte-Carlo, on utilise cette variabilité individuelle pour générer un grand nombre de distributions de points. Pour chacune de ces distributions, on réalise une régression (ou ajustement) linéaire ce qui conduira au final à un grand nombre de valeurs de a et de b. Il suffit ensuite de réaliser une étude statistique de ces données pour en déduire leur incertitude-type.

> Pour s'assurer qu'une régression linéaire est correcte, on tracera systématiquement sur un graphique les données mesurées ainsi que la droite de la régression linéaire. Le modèle sera validé si, à l'oeil, les points de mesure sont bien alignés et que la droite passe le plus proche de tous les points possibles, en incluant leurs incertitudes-type.

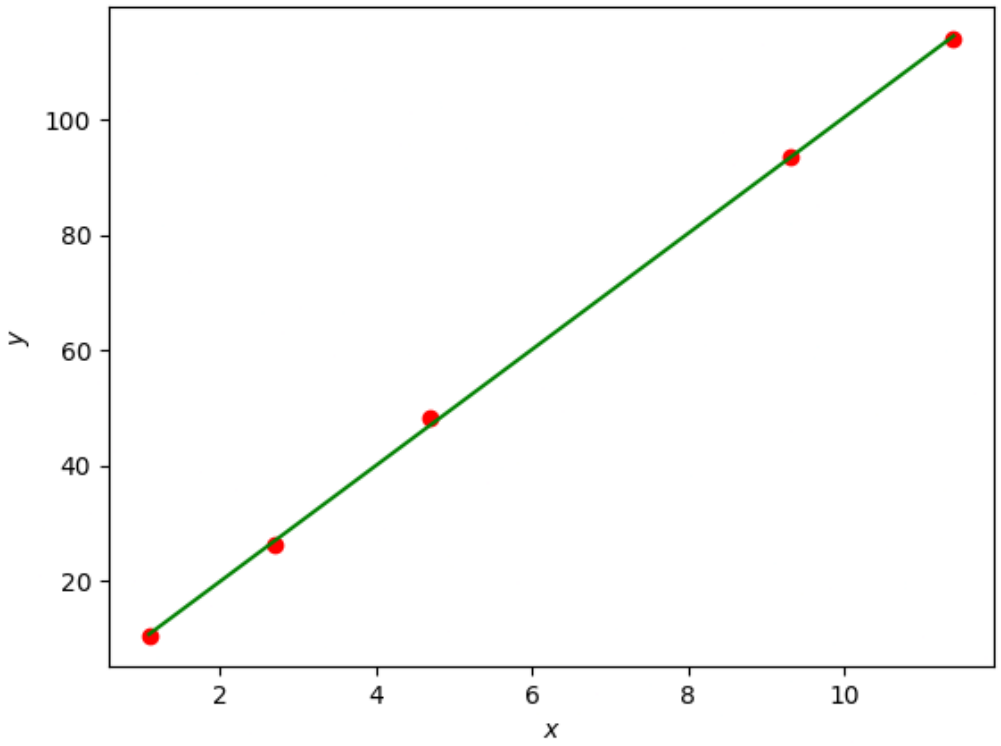
Ci-dessous un exemple **que vous pouvez adapter** en fonction de vos besoins :

> La ligne suivante sert à l'affichage des graphes dans le notebook.

Entrée[32]:
```
#%matplotlib notebook
```

Entrée[33]:
```
import numpy as np
import matplotlib.pyplot as plt
import random as rd

####### Partie à adapter ########
x=[1.1,2.7,4.7,9.3,11.4]
y=[10.4,26.4,48.3,93.5,114.1]

p=np.polyfit(x,y,1) # Ajustement de type a*x+b
a,b=p[0],p[1] # Valeurs de a et b

ux=0.01
uy=0.1

####### Partie à ne pas modifier ########
def variations_aleatoires(liste_x,liste_y): # Variation aléatoire de
    liste_xa=[]
    liste_ya=[]
    for x,y in zip(liste_x,liste_y):
        liste_xa.append(x+loi_normale(ux))
        liste_ya.append(y+loi_normale(uy))
    return liste_xa,liste_ya

def loi_normale(sigma) : # Loi de distribution normale
    return np.random.normal(0,sigma)

def loi_rectangulaire(sigma): # Loi de distribution rectangulaire
    return rd.uniform(-sigma,sigma)

distribution_a=[] # Distribution des valeurs de a
distribution_b=[] # Distribution des valeurs de b
N=100000 # Nombre de points utilisés dans la méthode Monte-Carlo

for i in range(N):
    xa,ya=variations_aleatoires(x,y)
    p=np.polyfit(xa,ya,1)
    distribution_a.append(p[0])
    distribution_b.append(p[1])

incertitude_type_a=np.std(distribution_a) # Calcul de l'incertitude-t
incertitude_type_b=np.std(distribution_b) # Calcul de l'incertitude-t

print('a=',a,'u(a)=',incertitude_type_a)
print('b=',b,'u(b)=',incertitude_type_b)

yfit=[]
for xfit in x:
    yfit.append(a*xfit+b)

plt.figure()
plt.plot(x,y,'ro')
plt.plot(x,yfit,'g-')
plt.xlabel('$x$')
plt.ylabel('$y$')
plt.show()
```

```
a= 10.072302383939773 u(a)= 0.016234701580670648
b= -0.28224592220830036 u(b)= 0.1139534479136905
```



Comparaison de deux mesures : l'écart normalisé

Pour comparer 2 mesures, on utilise **l'écart normalisé** ou **z-score** :

$$E_N = \frac{|m_2 - m_1|}{\sqrt{u(m_1)^2 + u(m_2)^2}}$$

Par convention, on qualifie deux résultats de **compatibles** si leur écart normalisé vérifie la propriété :

$$\boxed{E_N \lesssim 2}$$

> Ce seuil à 2 est d'origine historique. On le retrouve dans de nombreux champs scientifiques, comme la médecine, la pharmacie, la biologie, la psychologie, l'économie, l'écologie, etc. Ce seuil peut différer selon le domaine : par exemple pour démontrer l'existence d'une nouvelle particule en physique subatomique, il faut atteindre un seuil de 5.

Bibiographie :

- Séminaire *Mesure et incertitude* de Julien Browaeys et Nicolas Décamp
- Document en ligne *Mesures et incertitudes* de Maxime Champion