



IMT Atlantique

Bretagne-Pays de la Loire
École Mines-Télécom



Documentation du projet de stage :

« Étude de la faisabilité d'un système de mesure et d'analyse de résistivité à l'aide du micro-contrôleur XIAO Seeed Studio nRF52840 »

**Sous la supervision de Mr. Lahuec
Département d'électronique et d'optique
IMT Atlantique Brest**

Boulefred-Derrar Inès

Juillet-Août 2024

Sommaire :

I. Introduction

- A. Objectif du document

II. Description du système

- A. Fonctionnement du SAADC

- B. Description générale du système

III. Explication des codes

- A. Code de traitement des résistances

- 1. Logigramme

- 2. Explication des choix faits

- a) Les paramètres d'initialisation

- b) La fonction set-up

- c) Calcul d'une tension de référence

- d) La fréquence d'échantillonnage et le nombre total d'échantillons

- e) Le sur-échantillonnage

- f) L'étalonnage

B. Code random forest

- 1. Logigramme

- 2. Explication des choix faits

- a) Les hyper-paramètres de l'algorithme random Forest

- b) La pondération des types de noeuds

- c) Condition finale de détermination de la présence de DMMP

- d) Entraînement

IV. Tests et validation

- A. Données de résistance

- 1. Méthode de tests

- 2. Résultats et analyse

- B. Fréquence d'échantillonnage

- 1. Méthode de tests

- C. Allocation de la mémoire et durées de calcul

- 1. Méthode de tests

- 2. Résultats et analyse

V. Annexe

I. Introduction :

A. Objectif du document

L'objectif de cette documentation technique est de délivrer les informations nécessaires à la manipulation du dispositif de lecture et de traitement des données de résistivité. Ces informations comprennent également les explications du fonctionnement du dispositif, ainsi que les réflexions faites lors de la création qui seraient pertinentes à la manipulation ou modification. Le dispositif est composé de deux blocs principaux : un circuit à ponts diviseurs de tension ainsi que le micro-contrôleur Seeed Studio XIAO nRF52840.

L'objectif du projet est le suivant : Tester la possibilité de réaliser un dispositif de détection de changement de résistivité dans un circuit fait à base de ponts diviseurs, à l'aide du micro contrôleur XIAO nRF52840, ainsi qu'en étudier les avantages et les limitations.

II. Description du système

A. Fonctionnement du SAADC :

Le convertisseur analogique numérique du micro-contrôleur nRF52840 est un convertisseur à approximation successive. Il fonctionne comme suit :

1. Les signaux analogiques de tension provenant des entrées sont filtrés avec un filtre anti-repliement, afin qu'ils correspondent au critère de Nyquist.
2. La tension est ensuite figée à l'aide d'un figeur de tension, le temps de la conversion, c'est à dire le temps de stocker la valeur de 16 bits qui correspond à la tension en entrée.
3. Le signal de tension traverse un comparateur, qui prend 2 tensions en entrée : le signal analogique à convertir, et un signal de tension intégré au SAADC, qui provient d'un troisième composant, le convertisseur numérique analogique intégré. Le comparateur renvoie une sortie haute (signal binaire 1) si la tension d'entrée est plus grande que la tension de référence, et une sortie basse (0) dans le cas contraire.
4. La sortie du comparateur est ensuite transmise au SAR, le registre à approximation successive.

5. Jusqu'à ce que la valeur de tension soit estimée au niveau de précision que le MSB permet, la sortie du registre à approximation successive est transmise au CAN (convertisseur analogique numérique) puis au comparateur, dans une boucle.

B. Description générale du système :

Ci-dessous deux photos et un schéma du circuit électrique avec lequel les mesures de résistance ont été effectuées.

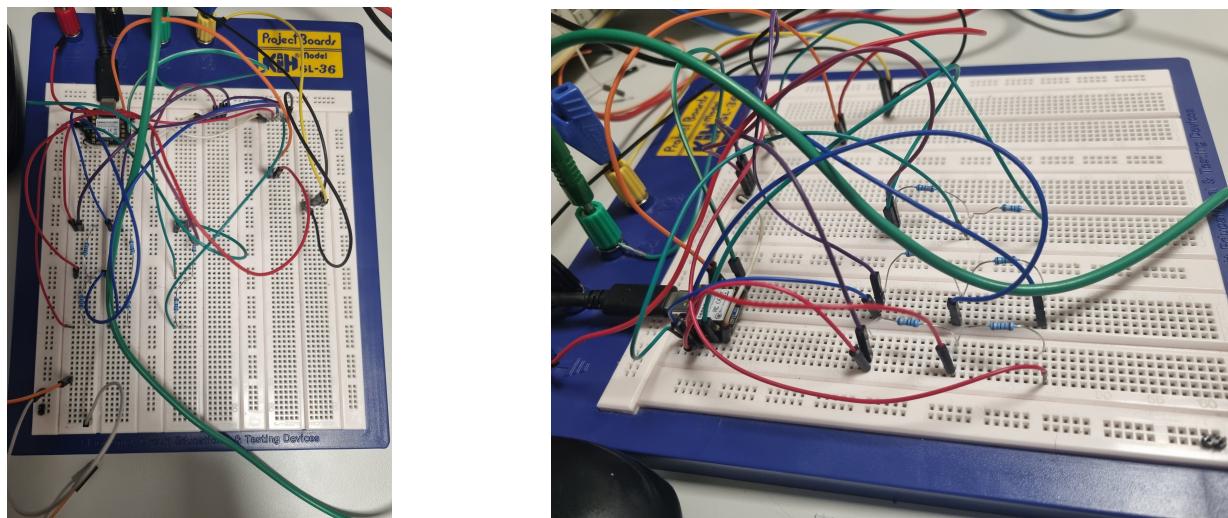


Figure 1 : Photos du pont diviseur de tension à 3 résistances

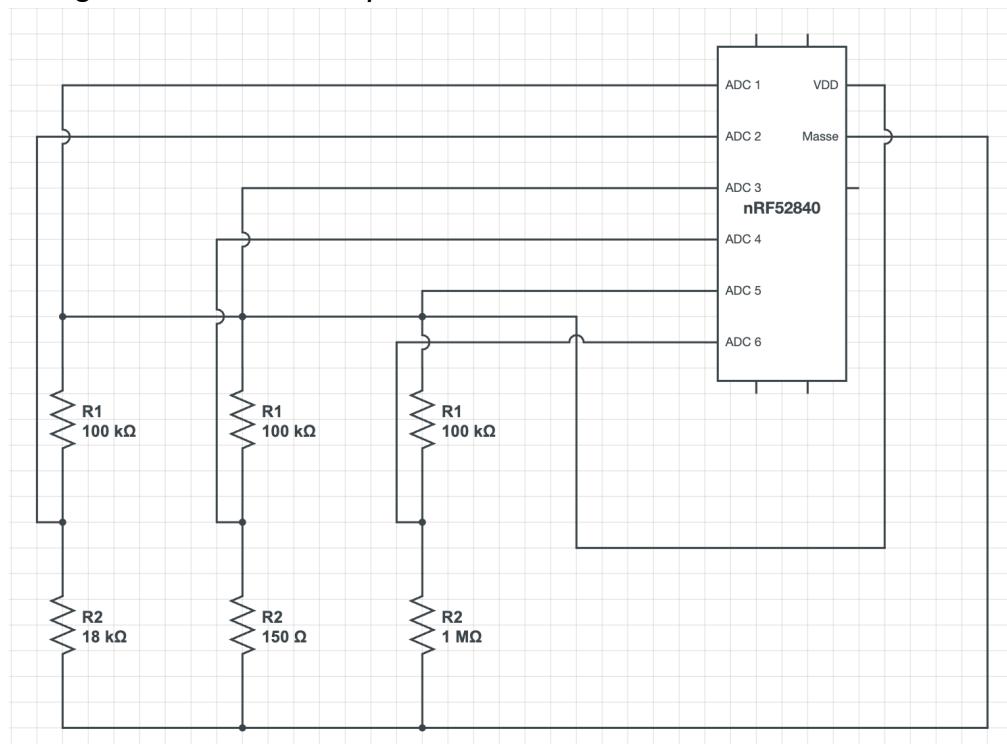


Figure 2 : Schéma du pont diviseur

En reliant à l'aide de fils les GPIO de l'ADC (Convertisseur analogique-numérique) du micro-contrôleur et les bornes de la résistance R1 connue, on peut mesurer la tension U2 et U pour chaque pont de la manière suivante : Le SAADC stocke une valeur de tension sous forme d'un chiffre à 16 bits, allant de 0 à 65535. 65535 correspond à la tension maximale mesurable, soit dans ce cas précis VDD = 3.3V. Il suffit ensuite de faire une conversion pour obtenir une valeur de tension allant de 0 à 3.3V à partir du nombre à 16 bits.

La tension U2 est celle aux bornes des résistances R2, et la tension U celle aux bornes de chaque pont. Cela nous permet de calculer la tension U1 aux bornes des résistances R1 connues avec $U1 = U - U2$.

Sachant que les résistances R1 et R2 sont en série, le courant I qui les traverse est identique. En appliquant la loi d'Ohm, on obtient $I = \frac{U1}{R1} = \frac{U2}{R2}$.

A partir de cette équation, on peut isoler la résistance inconnue R2 pour obtenir l'équation suivante : $R2 = \frac{U2 \times R1}{U1}$.

La précision de cette méthode est évaluée dans la partie **IV. Tests et validation**.

Le circuit ne peut comporter au maximum que 3 ponts diviseurs de tension, du fait du nombre limité de GPIO reliés à l'ADC. Effectivement, l'ADC de la carte nRF58240 n'a que 6 entrées/sorties, ce qui ne permet que la mesure de 3 couples de tension U/U2.

III. Explication des codes

A. Code de traitement des résistances

1. Logigramme :

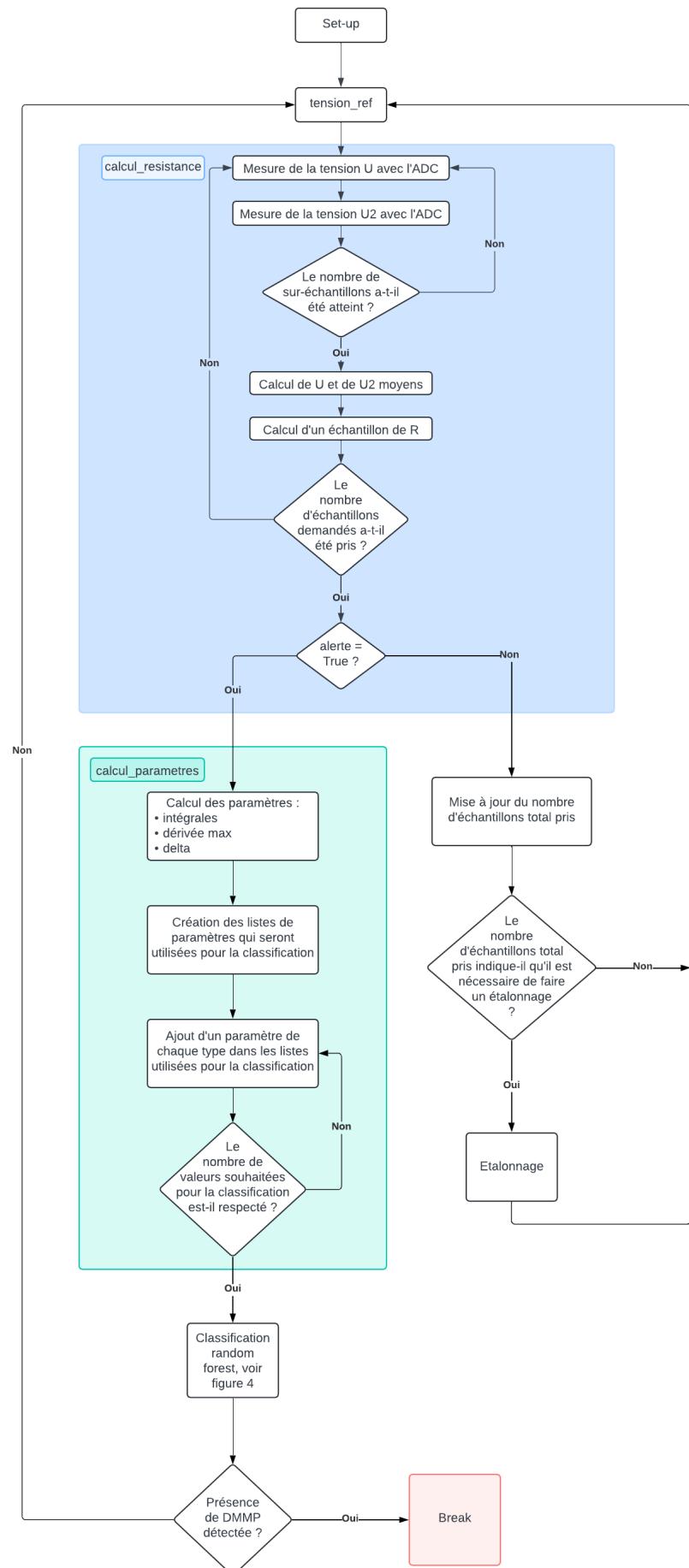


Figure 3 : Logigramme du code de traitement des résistances

2. Explication des choix faits :

a) Les paramètres d'initialisation :

Les paramètres d'initialisation du programme sont les suivants :

1. Le nombre de résistances du circuit
2. Le nombre de sur-échantillons à effectuer (voir ci-dessous)
3. Le nombre d'échantillons par montée ainsi que la durée d'une montée :
Ce paramètre permet de calculer la fréquence d'échantillonnage nécessaire afin de pouvoir repérer le DMMP. Comme la fréquence d'échantillonnage est la même pour tous les ponts diviseurs, quelque soit la résistance, il est préférable de paramétriser le programme avec les valeurs correspondantes à la résistance nécessitant la fréquence d'échantillonnage la plus élevée.
4. Le pourcentage de déviation maximal acceptable pour les données de résistance : pourcentage_danger. Ce paramètre est également commun à toutes les résistances.
5. La résolution souhaitée de l'ADC, qui influence sa précision.
6. Le paramètre d'étalonnage, qui permet de définir la fréquence d'étalonnage en échantillons.
7. Les valeurs des résistances connues

PIN_U : Pins de la carte connectés au points du circuit permettant de récupérer la tension U pour chaque résistance, respectivement U2.

PIN_alim : Pin de la carte par lequel la tension VDD sera délivrée au circuit.

PIN_test et PIN_ref : Pins mis en place pour envoyer un signal à l'oscilloscope à chaque mesure de tension et ainsi mesurer les fréquences d'échantillonnage.

b) La fonction set-up :

La fonction set-up sert à initialiser la résolution de l'ADC, l'alimentation du circuit avec la tension VDD, ainsi que le calcul du délai nécessaire à mettre en place pour respecter la fréquence d'échantillonnage demandée via les paramètres d'initialisation. Elle effectue également un premier étalonnage.

c) Calcul d'une tension de référence :

Il a été jugé nécessaire de calculer une tension dite de référence avant de mesurer chaque lot d'échantillons de résistances. Effectivement, il a été

observé au multimètre que la tension délivrée par la carte fluctuait autour de 3.3V, descendant parfois jusqu'à 2.7V.

Ainsi, la tension VDD délivrée réellement par la carte est calculée à chaque itération de la boucle principale, et les calculs de résistance sont basés sur sa valeur. Cela permet accessoirement de réduire la nécessité d'un régulateur de tension dans le circuit, tout en augmentant la précision des mesures.

d) La fréquence d'échantillonnage et le nombre total d'échantillons :

Le réglage de la fréquence d'échantillonnage de l'ADC n'a pas pu être effectué directement à partir des registres qui y sont reliés. Deux méthodes ont été testées pour modifier la fréquence d'échantillonnage à ce niveau :

1. Modification de la fréquence d'échantillonnage de l'ADC directement via le registre SAMPLE RATE de l'ADC.
2. Mise en place d'un canal PPI (Programmable Peripheral Interconnect) entre la tâche SAMPLE de l'ADC et un timer du périphérique TIMER. Cette solution faisait en sorte que la tâche SAMPLE de l'ADC soit déclenchée tous les n ticks du timer mis en place (le code correspondant est dans le dossier des codes, sous le nom *3R PPI timer RTC*)

Une explication probable sur l'échec de la mise en place de ces deux méthodes serait que la fréquence d'échantillonnage de l'ADC ait effectivement bien été modifiée, mais que cette modification ne se répercute pas sur les résultats de temps observés entre chaque échantillon pris, car ce temps serait d'avantage conditionné par le délai imposé par l'exécution des lignes de code.

La fonction délai sert à mettre en place un délai entre la prise des échantillons, afin de respecter la fréquence d'échantillonnage souhaitée. La fréquence d'échantillonnage maximale (sans délai aucun) a été mesurée à l'oscilloscope, et est de 246.31 Hz. Toute fréquence d'échantillonnage supérieure à 246.31 Hz ne pourra pas être respectée.

La fonction calcul_resistance calcule n (avec n fixé à 100 pendant les tests) échantillons de résistance pour chaque pont diviseur. Cette mesure a été mise en place pour deux raisons :

1. Lorsque les valeurs de résistance ne respectent pas le pourcentage de déviation de sécurité, une fois les n = 100 échantillons pris, le calcul des paramètres utiles à la classification sont lancés. Cependant, les calculs nécessitent qu'un certain nombre d'échantillons aient été effectués au préalable.
2. La fréquence d'échantillonnage n'est pas constante si un minimum d'échantillons ne sont pas pris à la suite. Effectivement, la fréquence

d'échantillonnage est difficilement respectable si le temps que toutes les lignes de code venant avant la prise des échantillons dans la boucle main soient effectuées varie d'une itération à une autre. Ce temps est aussi non négligeable, et ferait trop baisser la fréquence d'échantillonnage maximale.

C'est pour cela que 100 échantillons sont pris à la suite. Il faut environ 50 échantillons au minimum pris à la suite afin que la fréquence d'échantillonnage soit homogénéisée. Changer le nombre d'échantillons pris à la suite permet de décharger la RAM de la carte, au prix de réduire le nombre de données pour le calcul des paramètres (intégrale, delta, dérivée_max) et donc la durée d'évolution de la résistivité analysée lors de la classification.

e) Le sur-échantillonnage :

Le sur-échantillonnage implique que n échantillons soient effectués pour chaque mesure de tension afin de calculer une valeur de résistance moyenne à partir de ces tensions.

Il permet de rendre la précision des valeurs plus robustes et moins dépendantes de petits changements sur une échelle de temps courte. Afin de ne pas trop augmenter le temps de calcul des résistances, le nombre de sur-échantillons lors des tests était de 3.

f) L'étalonnage :

L'étalonnage de la carte nRF52840 est à faire en cas de changements de température de l'environnement de la carte. La fréquence d'étalonnage a été réglée à 100000 échantillons préalablement, car l'effet de la fréquence d'étalonnage sur les mesures n'a pas pu être étudié durant l'étude.

B. Code random forest

1. Logigramme

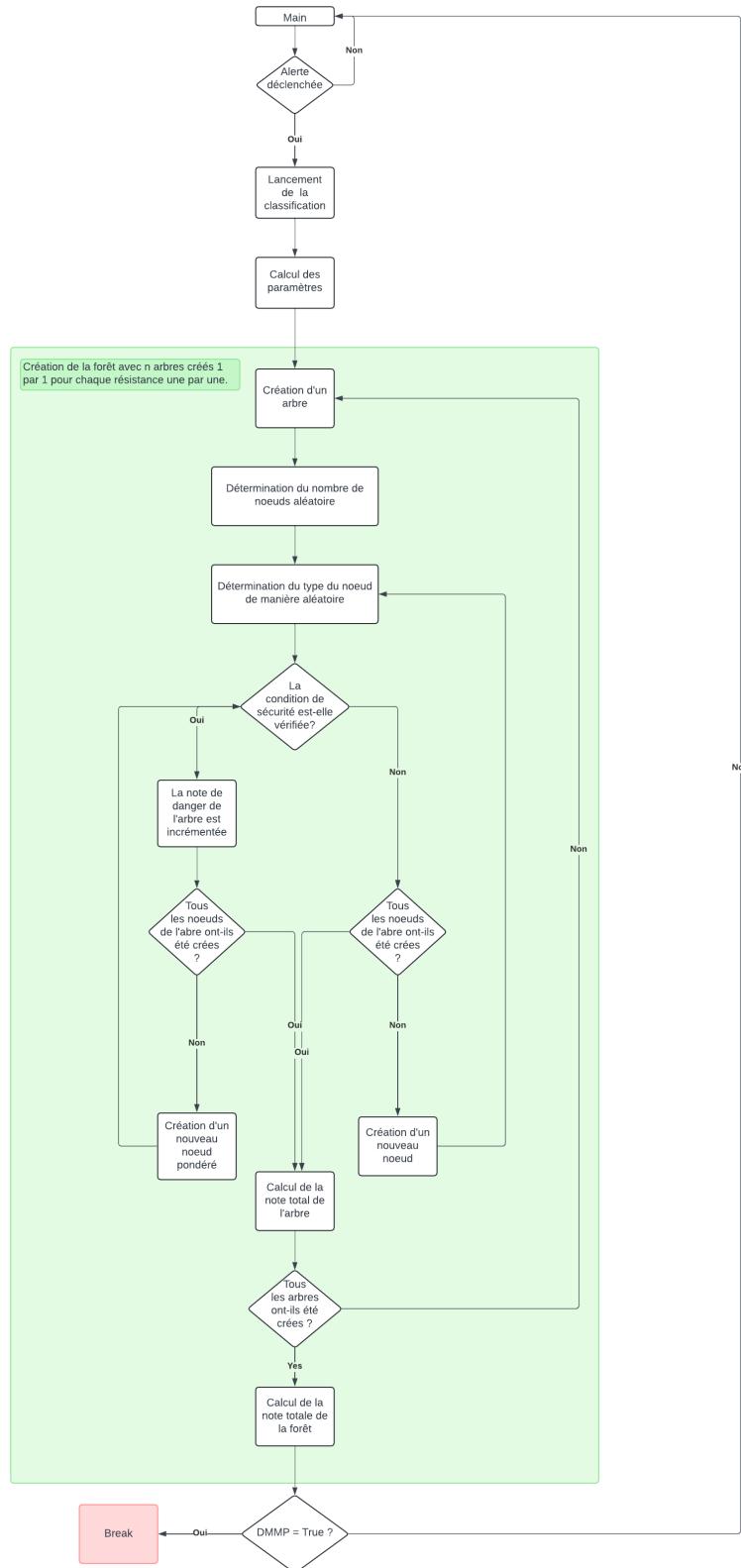


Figure 4 : Logigramme de l'algorithme random forest

2. Explication des choix faits :

L'algorithme random forest a été conçu intégralement selon le logigramme ci-dessus, car il n'existe pas encore de bibliothèque de Deep Learning facilitant la mise en place de cet algorithme sous micro-python.

a) Les hyper-paramètres de l'algorithme random forest :

Les hyper-paramètres de l'algorithme sont les suivants :

1. nb_donnees_pour_classif indique la longueur des listes de paramètres que l'on souhaite avoir en tant que jeu de données pour la classification. Chaque liste de paramètres (integralef, merivee_maxf, deltaf, sommef) utilisée pour la classification comportera ainsi 3 (le nombre de résistances) listes de longueur n avec n = nb_donnees_pour_classif.
2. Les seuils sont les intervalles acceptable pour chaque valeur des paramètres (integrale, dérivée_max, delta, et somme). Si la donnée choisie au hasard parmi le jeu de donnée ne rentre pas à l'intérieur de l'intervalle défini par le seuil, le noeud correspondant votera pour la présence de DMMP.
3. Le nombre d'arbre, ainsi que le nombre de noeuds minimal et maximal acceptable, afin que chaque arbre ait un nombre de noeuds aléatoire.
4. Les paramètres pris en compte pour la détermination de la présence de DMMP. L'étude a eu pour paramètres : delta, integrale, derivee_max et la somme des trois premiers paramètres, conformément à l'étude effectuée sur la présence d'ammoniaque de manière non invasive dans l'haleine.

b) Pondération du poids des types de noeuds :

Lorsque le noeud d'un arbre détermine avoir constaté la présence de DMMP, l'arbre est construit de telle façon à ce que le noeud suivant choisisse son paramètre d'étude au hasard mais selon une pondération.

Effectivement, selon l'étude effectuée sur l'analyse de présence d'ammoniaque dans l'haleine, la précision du diagnostic varie selon les paramètres de la classification et selon l'algorithme. Ci-dessous, un tableau synthétisant les résultats expérimentaux de l'effet du choix de paramètre et de l'algorithme de classification sur la précision du diagnostic:

Table 3
Diagnostic accuracy (in %) obtained by different classification algorithms and different set of features.

Tested algorithm	Features of the resistive curve used to test the algorithm			
	ΔR	∂R	$\int R$	$\Delta R + \partial R + \int R$
LDA	64	64	67	80
RF	77	77	50	80
SVM	72	77	75	85
MLP	75	78	63	85

Figure 5 : Effet du choix de paramètre et d'algorithme sur la précision des résultats de la classification

Le poids de chaque paramètre dans le tirage au sort pour déterminer le type du noeud suivant respecte les données du tableau ci-dessus.

c) Condition finale de détermination de la présence de DMMP :

Afin qu'un arbre détermine qu'il y a présence de DMMP, il faut que plus de la moitié de ses noeuds votent ainsi.

Afin que la forêt dans son ensemble et donc la classification finale détermine qu'il y a présence de DMMP, il faut que plus de la moitié de ses arbres votent ainsi.

Si le DMMP est détecté à la fin de la classification, et donc que le booléen DMMP prend la valeur True, la boucle principale prend automatiquement fin.

d) Entraînement :

Un code d'entraînement dans le dossier des codes est également fourni afin de déterminer les hyper-paramètres les plus efficace à la détection de DMMP. Il est sous le nom *entraînement random forest*.

L'entraînement n'a pas pu être mené à bien dans la durée impartie de l'étude, ainsi le code n'a pas été testé. Cependant il ne devrait théoriquement être effectué qu'une seule fois pour chaque carte fonctionnelle.

IV. Tests et validation

A. Données de résistance

1. Méthode de test

La précision des données de résistance mesurées grâce au montage et au code micro python a été évaluée selon la méthode suivante :

Plusieurs valeurs de résistances ont été mesurées avec deux résistances de référence différentes, afin de déterminer laquelle d'entre elles permettrait d'obtenir une meilleure précision. Les résistances de références étaient de : 10,03 kΩ et 100 kΩ. Le critère de précision employé est le pourcentage d'erreur, qui s'obtient selon la formule suivante : % d'erreur = (valeur mesurée - valeur théorique)/valeur théorique.

Les résistances mesurées à l'aide de ces résistances de référence variaient de 1000 Ω à 10 MΩ, avec un facteur x10 entre chaque palier de résistance à mesurer. En premier lieu, le test a été fait pour les résistances à mesurer suivantes : 1kΩ, 10kΩ, 100kΩ.

Les résistances ont été mesurées via le code, à partir d'un pont diviseur de tension unique, toutes les secondes pendant une durée de 120 secondes.

Les résultats se trouvent dans les figures 6 et 7.

Suite à cela, le code final avec la mise au point de la fonction set-up et l'architecture complète pour la mesure de plusieurs résistances a été testé avec des valeurs de résistances faisant partie des résistances mesurées avec haute précision au test précédent. Les données de résistances obtenues par ce code ont été analysées de la même manière, soit dans les conditions suivantes :

- Pont diviseur de tension à 3 résistances
- Prise de mesure toutes les secondes pendant 12 secondes
- Résistance de référence de 100 000 Ω
- Les résistances mesurées étaient de 17 960 Ω , 149900 Ω , et 0,99 M Ω

On observe les résultats ci-dessous dans la figure 9.

2. Résultats et analyse

Ci-dessous les résultats de précision en fonction de la résistance de référence R1 :

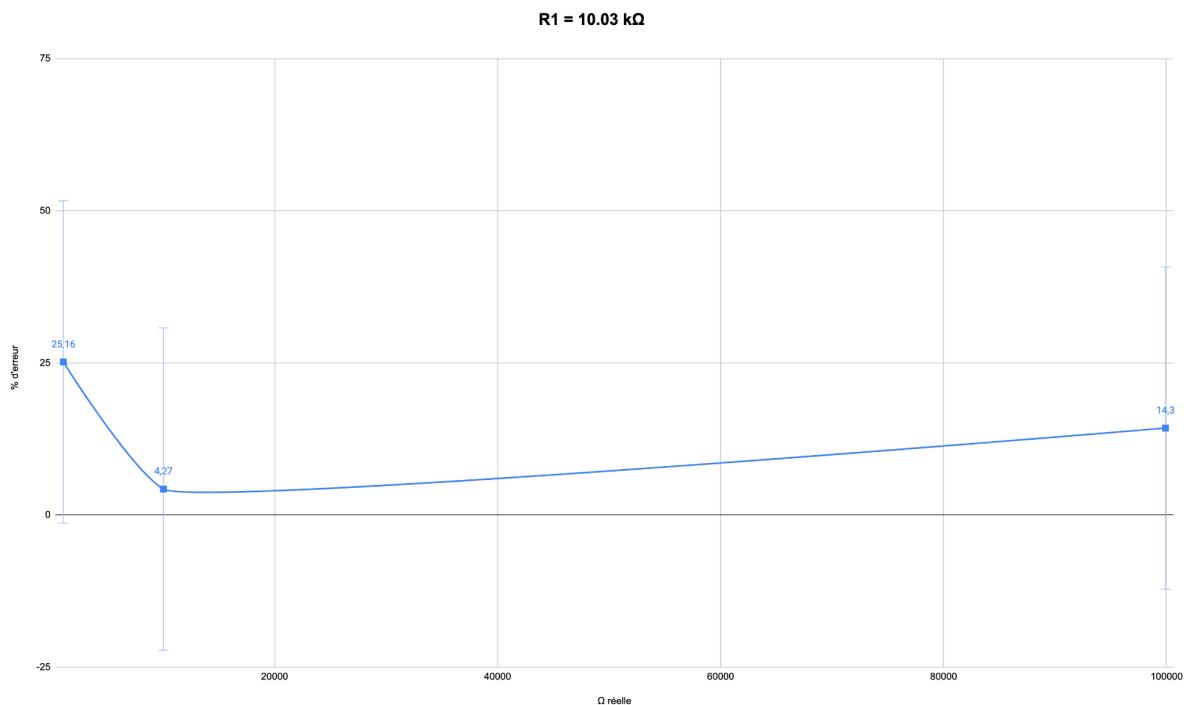


Figure 6 : Pourcentage d'erreur et barres d'écart type des mesures de résistance pour la résistance de référence $R1 = 10,03 \text{ k}\Omega$

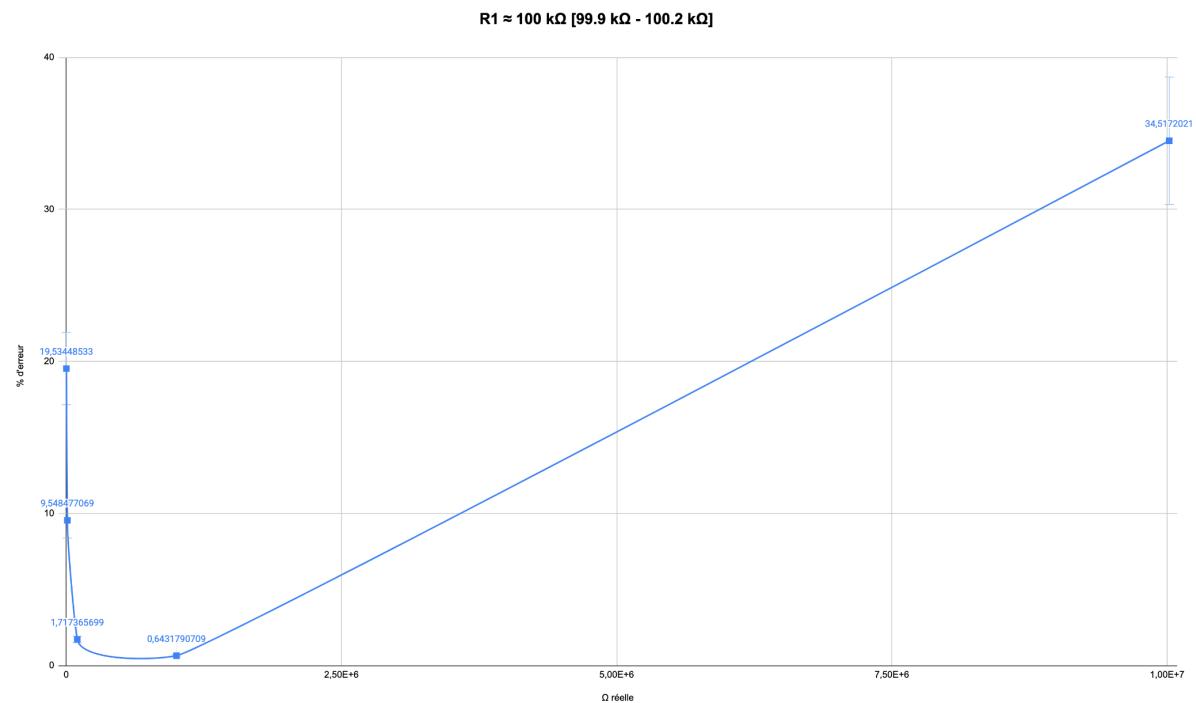


Figure 7 : Pourcentage d'erreur et barres d'écart type des mesures de résistance pour la résistance de référence $R1 = 100 \text{ k}\Omega$

La moyenne des pourcentages d'erreur pour les résistances mesurées de 1kΩ, 10kΩ et 100kΩ avec une résistance de référence de 10.03kΩ est de : 14,58%, avec un écart type allant de 0,69 à 26,4%.

La moyenne des pourcentages d'erreur pour les résistances de 1kΩ, 10kΩ et 100kΩ avec la résistance de référence de 100kΩ est de : 2,76% avec un écart type allant de 2,49 à 5,18%.

Ainsi, la résistance de référence de 100kΩ a été retenue pour les tests qui s'en sont suivis, étant donné qu'elle permet la mesure de résistance avec une plus grande précision pour le circuit, que cela soit en terme de pourcentage d'erreur moyen ou d'écart type constaté.

Suite à cette comparaison, des tests de précision ont été également fait pour voir jusqu'à quelle résistance mesurée la précision de la résistance de référence de 100kΩ était satisfaisante. Le test a été fait pour les résistances mesurées de 1MΩ et 10MΩ, voir figure 8 ci-dessous. On observe conséquemment que le seuil de précision de la résistance de référence de 100kΩ reste satisfaisant jusqu'à une résistance mesurée de 1MΩ.

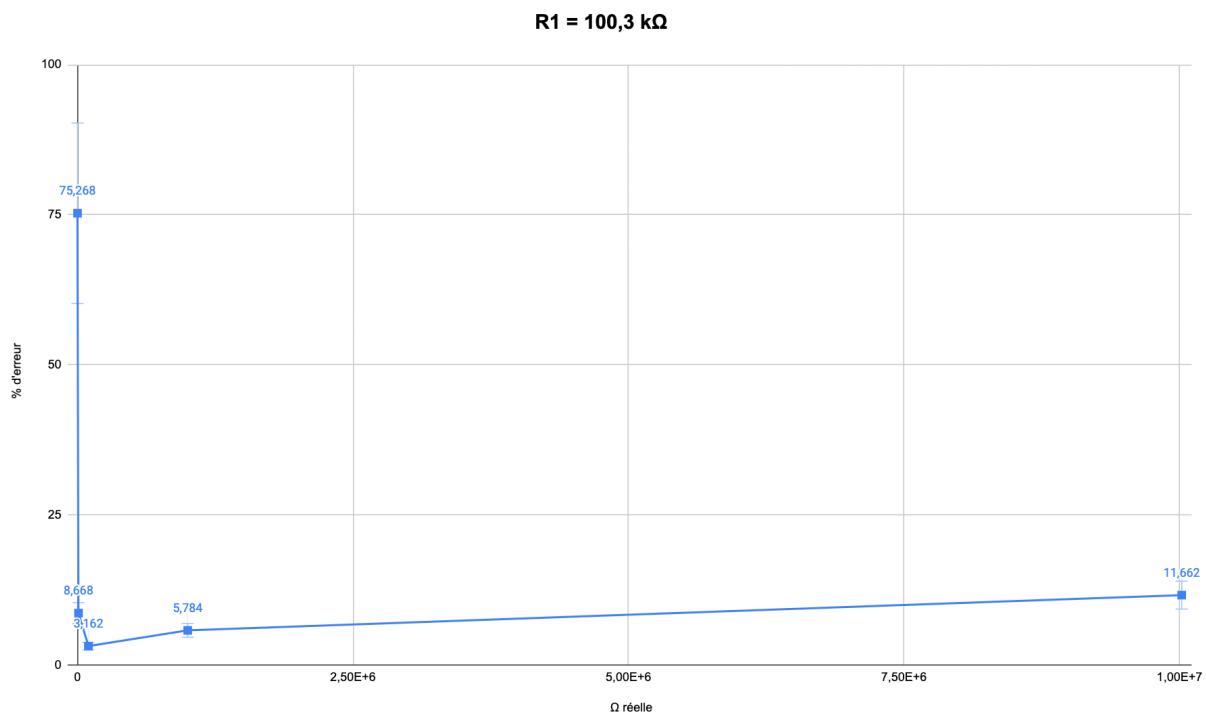


Figure 8 : Pourcentage d'erreur et barres d'écart type des mesures de résistances sur 1 pont diviseur, avec une résistance de référence de 100 300 Ohm.

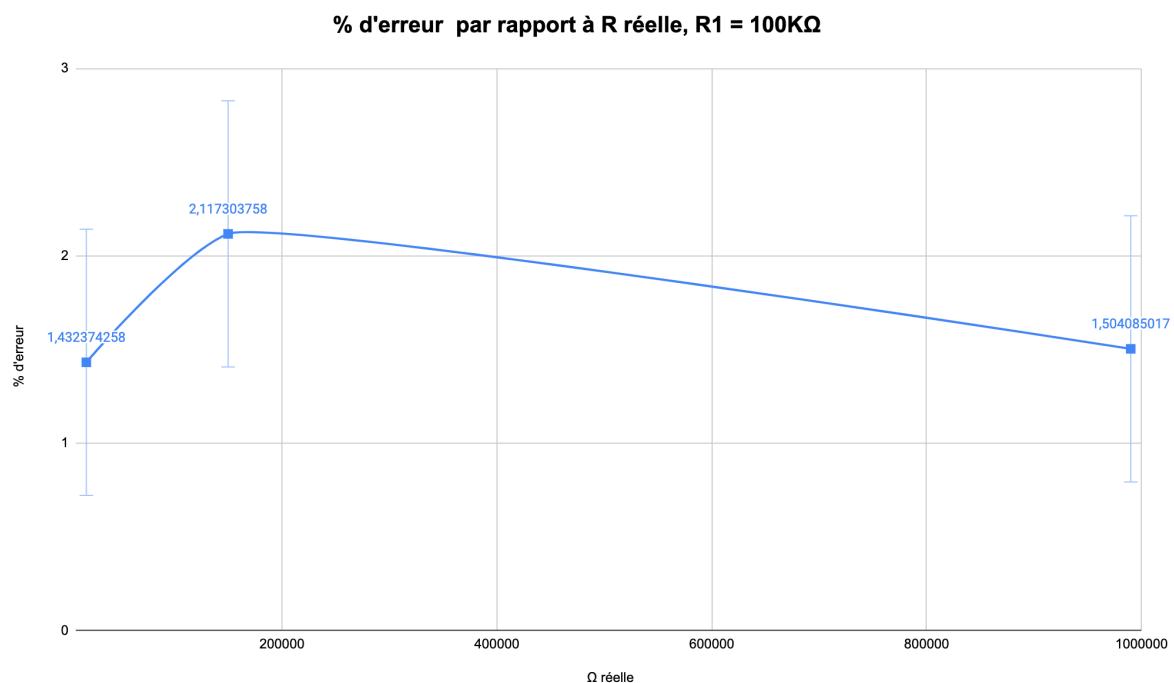


Figure 9 : Pourcentage d'erreur et barre d'écart type des mesures de résistance sur 3 ponts diviseurs, à l'aide du code final et d'une résistance de référence de 100kΩ.

B. Fréquence d'échantillonnage

1. Méthode de test

La fréquence d'échantillonnage a été mesurée à l'aide d'un oscilloscope.

Avant et après chaque lecture de l'ADC dans le code (ligne `tension_U_analog[i].read_u16()` ou `tension_U2_analog[i].read_u16()`) un signal est envoyé d'un GPIO de la carte jusqu'à l'oscilloscope, ce qui permet de visualiser clairement le début et la fin de chaque lecture de l'ADC.

En récupérant les données temporelles exactes à l'aide de l'interface de l'oscilloscope (voir figure 11 ci-dessous), les différentes fréquences intéressantes ont pu être calculées. Parmi elles, on compte notamment la fréquence d'échantillonnage pour un échantillon au complet (c'est à dire avec la prise de tous les sur-échantillons pour chaque résistance), la fréquence de sur-échantillonnage (prise de tous les sur-échantillons pour une résistance) et la fréquence d'échantillonnage simple (prise d'un seul échantillon, sans sur-échantillonnage).

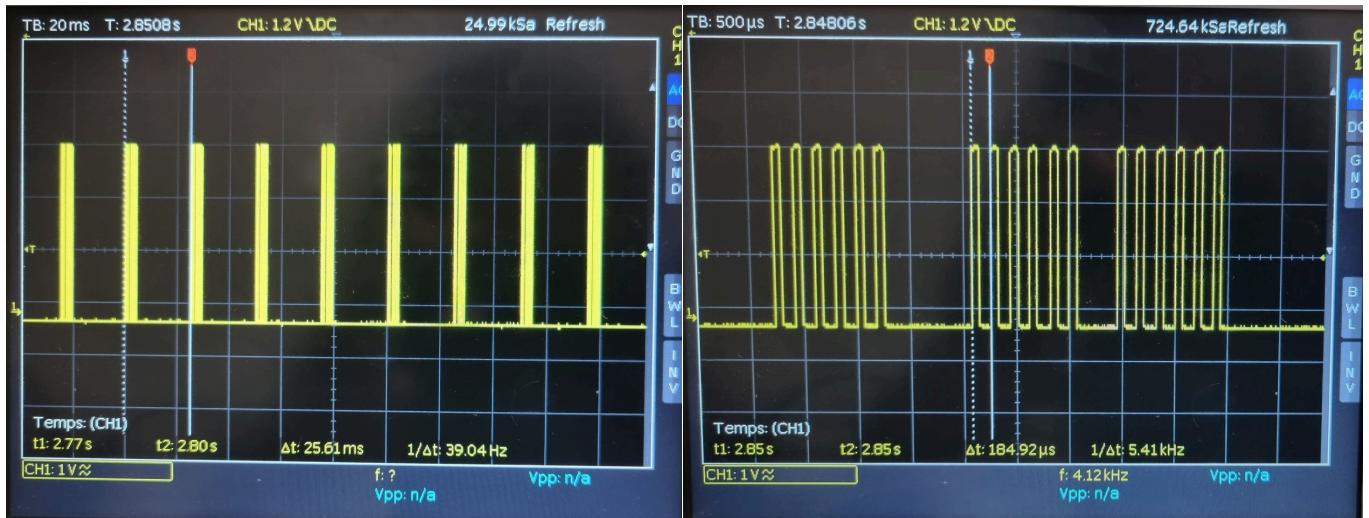


Figure 11 : Exemples de différentes mesures de fréquence d'échantillonnage via l'oscilloscope

C. Allocation de la mémoire et durées de calcul

1. Méthode de test

Afin de d'optimiser la mémoire au mieux, les listes stockant les données nécessaires aux calculs sont supprimées entièrement dès qu'elles ne sont plus nécessaires. Elles sont recréées au début des fonctions pertinentes. Si non, elles sont vidées dès que possible.

La commande `gc.collect` est également utilisée à chaque itération de la boucle principale (calcul des résistances, vérification de l'état de l'alerte, lancement éventuel de la classification, étalonnage éventuel) afin de libérer de la mémoire en supprimant tous les objets du code non référencés le plus souvent possible, dans la limite de l'utile.

Le code *tuple PPI 3R* est un équivalent antérieur du code final, où les listes ont été remplacées par des tuples. Malheureusement, bien qu'en théorie moins couteux en mémoire, cette version du code était finalement moins optimale en consommation de mémoire.

La fonction `ticks_us` du module `time` a permis de récupérer les données de temps au début et à la fin de l'exécution de chaque fonction. La fonction `mem_free` du module `gc` permet d'obtenir la mémoire libre avant et après l'exécution d'une fonction, et ainsi la mémoire utilisée.

Les ressources nécessaires à l'exécution de ces fonctions, bien que minimes, peuvent cependant altérer légèrement les résultats obtenus.

2. Résultats et analyse

Ci-dessous l'évolution de la mémoire à chaque itération de la boucle principale :

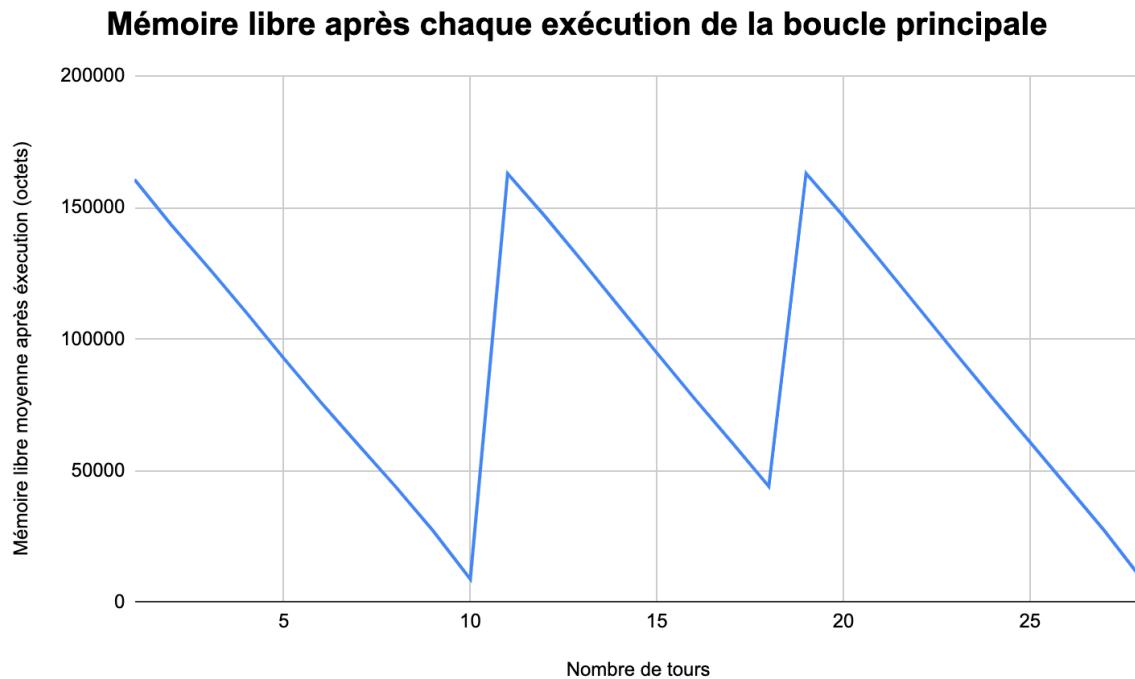


Figure 12 : Évolution de la mémoire libre du micro-contrôleur

La boucle principale a pu être itérée 28 fois avant qu'il n'y ait plus de mémoire disponible.

Théoriquement, la carte dispose d'1MB de mémoire flash, et 256 kB de mémoire RAM. Avant l'exécution de la première fonction, elle dispose encore de 173 968 octets soit environ 174 kB de RAM.

Ci-dessous la consommation de mémoire selon les fonctions du code :

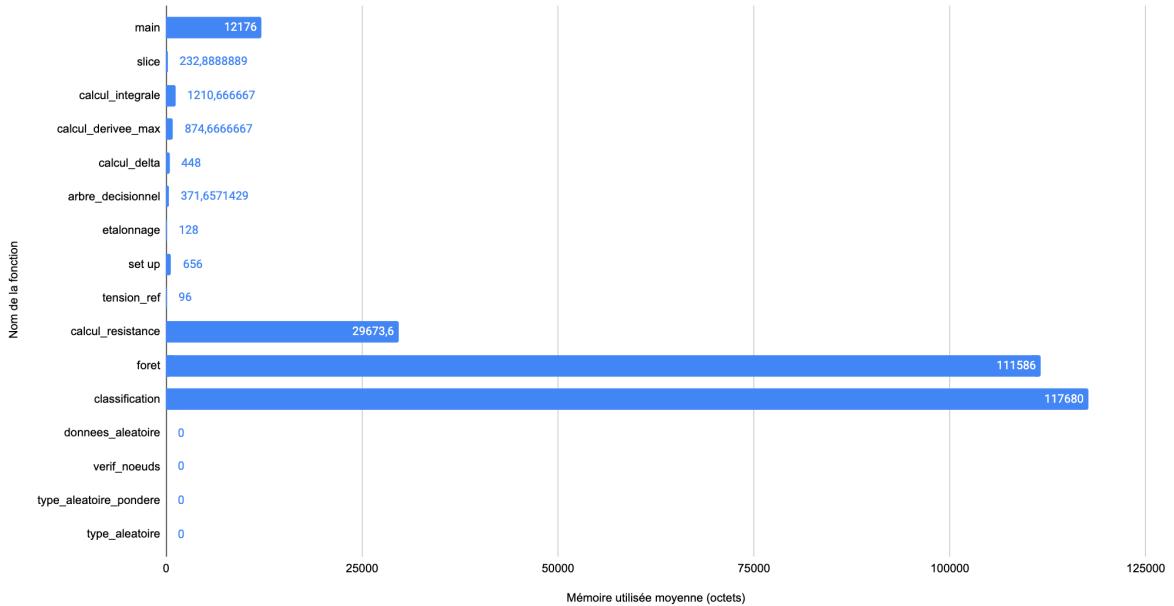


Figure 13 : Mémoire moyenne utilisée pour chaque fonction (octets)

On constate ainsi que la fonction classification, qui inclut la fonction forêt, nécessite à minima la moitié de la mémoire totale de la carte, et ce pour une classification à 10 arbres, avec un nombre de noeuds variant entre 0 et 5 aléatoirement. Si on compare son besoin de mémoire à la mémoire disponible sur la carte au lancement du code, la fonction classification utilise 67,64% des ressources en mémoire disponibles au lancement.

A chaque itération, la boucle principale (main) qui englobe toutes les fonctions du code, consomme 12 176 octets, soit 7% de la mémoire disponible au lancement du code.

La gestion de la mémoire ne permet au code complet d'être lancé plus de quelques minutes, voir même quelques secondes.

Il est difficile de calculer précisément la mémoire totale nécessaire à un fonctionnement de la carte qui durera plusieurs heures, car la libération périodique de mémoire au fur et à mesure des itérations de la boucle principale semble être aléatoire. La fonction gc.collect() ne semble pas libérer la même quantité de mémoire à chaque itération de la boucle principale, comme l'illustre la figure 12.

Cependant, il est possible d'aboutir à une valeur précise en faisant l'hypothèse que l'on pourrait estimer la consommation de mémoire de la boucle principale en prenant sa moyenne sur les 28 itérations.

Soit la mémoire consommée pour une itération : $173\ 968/28 = 6213,14$ octets.

Si nous estimons que la boucle principale est itérée une fois toutes les 10 secondes, soit 3600 fois par heure, il faudrait théoriquement $6213,14 \times 360 \times 10 = 223673140,19$ octets, soit 21,34 mégaoctets pour une consommation de 10 heures.

Quant à la vitesse d'exécution, la figure 14 ci-dessous donne un aperçu global du temps pris par chaque fonction :

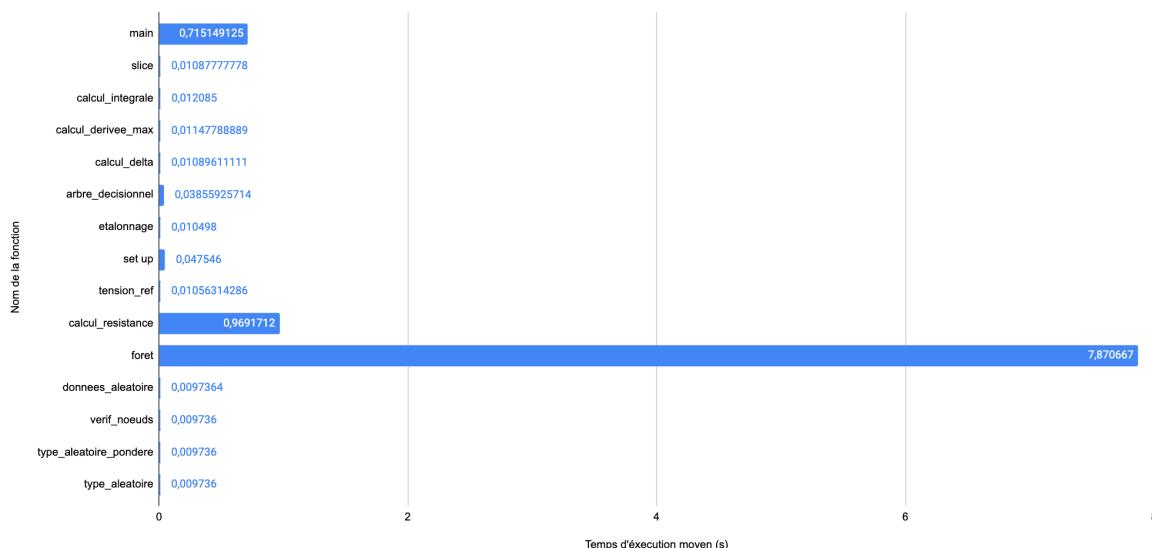


Figure 14 : Moyenne du temps d'exécution de chaque fonction

Il devient alors évident que la classification Random Forest est la partie du code à la plus haute durée d'exécution, avec presque 8 secondes d'exécution en moyenne.

Bien que la durée d'exécution de la boucle principale main mesurée et retranscrite dans ce diagramme est alors erronée, car la classification RF fait partie de la boucle principale main.

V. Conclusion :

Cette étude a permis de mettre en lumière plusieurs conclusions vis-à-vis de la faisabilité du projet.

Effectivement, la conclusion principale est que la quantité de mémoire disponible sur la carte est insuffisante à l'implémentation de plusieurs mesures de résistances, avec sur-échantillonnage, stockage de données pour le calcul de dérivées et d'intégrales, et avec une classification de type Random Forest.

Cette conclusion est suffisante pour affirmer que la carte ne saurait suffire à tourner plusieurs heures par elle-même.

Parmi les inconvénients dits secondaires, on retrouve le nombre de GPIO de l'ADC. Bien que la carte dispose de 11 GPIO au total (voir **Annexe : Ports de la carte**), seuls 6 d'entre eux permettent d'interagir avec l'ADC. Sachant qu'il faut prendre la mesure de 2 tensions différentes pour chaque pont, il n'est pas possible de mesurer plus de 3 résistances.

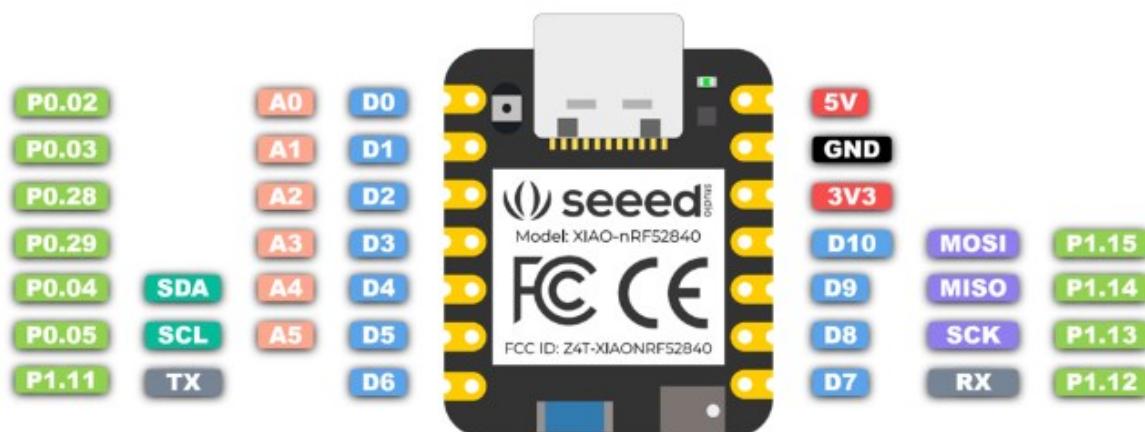
On notera également qu'au niveau des durées d'exécution, certaines limites s'imposent. Effectivement, lorsque la classification est lancée, les mesures de résistances ne peuvent être faites à nouveau qu'après dix secondes au minimum, selon les hyper-paramètres. Ce temps correspond à une forêt de 10 arbres et 0-5 noeuds. De plus, il n'est pas possible d'avoir une fréquence d'échantillonnage supérieure à 246.3 Hz.

Cependant, l'étude démontre que la précision des mesures entre $10\text{k}\Omega$ et $1\text{M}\Omega$ pour une résistance de référence de $100\text{k}\Omega$ est très bonne. Et théoriquement, avec un aménagement de mémoire, le projet pourrait autrement être mené à bien car il n'y a pas eu d'autres problèmes aussi limitants que la gestion de la mémoire.

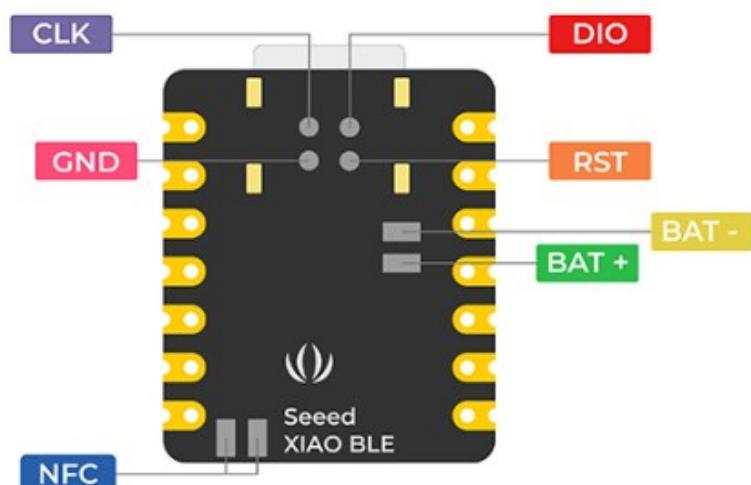
La consommation d'énergie n'a pas été mesurée, ainsi aucune conclusion ne peut être tirée à ce niveau.

VI. Annexe

1. Ports de la carte :



Digital Analog Pin No. IIC UART SPI GND Power



2. Documentation officielle de la carte et étude faite sur la présence d'ammoniaque dans l'haleine :

La documentation officielle de la carte est accessible dans le répertoire GitHub contenant tous les documents pertinents et les codes.

3. Code source principal commenté :

```

78
79
80 #ALIMENTATION DU CIRCUIT AVEC VDD
81 GPIO = Pin(PIN_alim, Pin.OUT)
82 GPIO.value(1)
83
84 #PARAMETRES DE FREQUENCE D'ECHANTILLONNAGE
85 frequence_echantillonnage = nb_echantillons_par_montee / duree_montee_s
86 print("Fréquence d'échantillonage demandée :", frequence_echantillonnage, "Hz")
87 periode_echantillonnage = 1 / frequence_echantillonnage
88 periode_min = (periode_min_3S_3R * nb_resistances * nb_sur_echantillons) / 9
89 if periode_echantillonnage > periode_min:
90     attente_us = (periode_echantillonnage - periode_min) * 1e6
91     print("Fréquence d'échantillonage ajustée")
92 else:
93     print("La fréquence d'échantillonage ne peut pas dépasser 246.31 Hz.")
94     attente_us = 0
95
96 #PREMIER ETALONNAGE
97 nb_etaлоннages = etalonnage(nb_etaлоннages)
98 print("Set-up terminé.")
99
100 return nb_etaлоннages, attente_us
101
102 def etalonnage(nb_etaлоннages):
103     nb_etaлоннages += 1
104     mem32[BASE_ADDR_ADC + 0x500] = 1 #activation du SAADC via le registre correspondant (obligatoire pour que l'étaлоннage se fasse)
105     ENABLE = mem32[BASE_ADDR_ADC + 0x500]
106     mem32[BASE_ADDR_ADC + 0x00C] = #envoi de la commande d'étaлоннage au registre correspondant
107     check_ETLN = mem32[BASE_ADDR_ADC + 0x110]
108     #vérification
109     while check_ETLN < 1 :
110         check_ETLN = mem32[BASE_ADDR_ADC + 0x110]
111     if check_ETLN == 1 :
112         print("Eталоннage effectué")
113     return nb_etaлоннages
114
115 def tension_ref(PIN_U):
116     for j in range(nb_resistances):
117         import gc
118         import utime
119         import random
120
121 #VARIABLES INITIALISATION
122 nb_resistances = 3
123 nb_sur_echantillons = 3
124 nb_echantillons_par_montee = 50000
125 duree_montee_s = 150
126 pourcentage_danger = 0.01 #pourcentage de déviation des valeurs de résistance au dessus duquel déclencher la classification
127 resolution_ADC = 0x3 #0 : 8 bits, 1 : 10 bits, 2 : 12 bits, 3 : 14 bits
128 parametre_etaлоннage = 1000000000000 #nombre d'échantillons à effectuer avant d'étaлоннer à nouveau
129 R1 = (100000.0, 100000.0, 100000.0) #inconnues : 18k / 150k / 1M
130
131 #Définition des pins :
132 PIN_U = (28, 3, 5) #voir schéma intégré au rapport pour numéros des pins
133 PIN_U2 = (29, 4, 2)
134 PIN_alim = 43
135 PIN_test = 47
136 PIN_ref = 44
137 PIN_LED = 13
138
139 #HYPER-PARAMETRES RANDOM FOREST :
140 nb_donnees_pour_classif = 100
141 seuil_delta = [0.5, 0.7] #intervalle dans lequel une donnée de delta doit être comprise pour indiquer qu'il n'y a pas de danger
142 seuil_derivee_max = [0.5, 0.7] #intervalle dans lequel une donnée de dérivée doit être comprise pour indiquer qu'il n'y a pas de danger
143 seuil_integrale = [0.5, 0.7] #intervalle dans lequel une donnée d'intégrale doit être comprise pour indiquer qu'il n'y a pas de danger
144 seuil_somme = [1.5, 2.1] #intervalle dans lequel une donnée de somme des paramètres (intégrale, dérivée, delta) doit être comprise pour indiquer qu'il n'y a pas de danger
145 nb_arbres = 10
146 parametres = ("delta", "integrale", "derivee_max", "somme")
147 nb_noeuds_min = 0
148 nb_noeuds_max = 5
149
150 #VARIABLES INITIALISATION - à ne pas modifier
151 nb_parametres = len(parametres)
152 vote_arbre = 0
153 vote_grave = v
154 vote_foret = 0
155 alerte = False
156 DMMP = False
157 r = 0
158 j = 0
159 nb_etaлоннages = 0
160 BASE_ADDR_ADC = 0x40007000
161 total_sample = 0
162 R2 = [0] * nb_resistances
163 ref = [0] * nb_resistances
164 tension_U2_analog = [0] * nb_sur_echantillons
165 tensionU2 = [0] * nb_sur_echantillons
166 tension_U_analog = [0] * nb_sur_echantillons
167 tensionU = [0] * nb_sur_echantillons
168 debut = [0] * nb_sur_echantillons
169 periode_min_3S_3R = 1/246.31
170 TST = Pin(PIN_test, Pin.OUT)
171 REF = Pin(PIN_ref, Pin.OUT)
172 somme = [0] * nb_resistances
173 integrale = [0] * nb_resistances
174 derivee_max = [0] * nb_resistances
175 delta = [0] * nb_resistances
176 nb_echantillons = 100
177
178 #FONCTIONS
179
180 def delai(attente_us):
181     debut_delai = utime.ticks_us()
182     while utime.ticks_diff(utime.ticks_us(), debut_delai) < attente_us:
183         pass
184 #=> cette fonction sert à mettre en place un délai de attente_us micro secondes
185
186 def setup(resolution_ADC, nb_etaлоннages, PIN_alim, nb_echantillons_par_montee, duree_montee_s):
187     #RESOLUTION SAADC
188     mem32[BASE_ADDR_ADC + 0x5F0] = resolution_ADC
189     check_resolution = mem32[BASE_ADDR_ADC + 0x5F0]
190     print("Résolution SAADC =", check_resolution)
191     print("0 : 8 bits, 1 : 10 bits, 2 : 12 bits, 3 : 14 bits")
192     print(" ")

```

```

115 def tension_ref(pin_u):
116     for i in range(nb_resistances):
117         tension_ref_analog = ADC(Pin(PIN_U[i], Pin.IN)) #définition du pin
118         ##REF!.value(1) #insert uniquement à envoyer un signal pour signaler la lecture de l'ADC (utilisé pour mesurer la fréquence d'échantillonnage à l'oscilloscope)
119         tension_ref = tension_ref_analog.read_u16()
120         ##REF!.value(0)
121         ref[i] = tension_ref * (3.3 / 65535)
122     return ref
123 #=> cette fonction serv à récupérer la valeur de tension VDD de manière préliminaire afin de l'utiliser comme référence dans la mesure des
124 #tensions dans la fonction calcul_referen
125 #On utilise la valeur maximale de VDD (3.3V) pour la mesurer, valeur qui a fourni les résultats les plus correspondants à la mesure au voltmètre.
126
127 def calcul_resistance(ref, R1, PIN_U, PIN_U2, tension_U2_analog, tensionU2, tension_U_analog, tensionU, alerte, attente_us, nb_resistances):
128     for z in range(nb_resistances, nb_resistances + nb_échantillons): #nb_échantillons = 100 valeurs échantillonées de résistance sont retournées par la fonction, voir
129         for j in range(nb_resistances):
130             U = [0] * nb_sur_échantillons #création de la liste qui contiendra les n valeurs de la tension U, avec n le nombre de sur échantillons
131             U2 = [0] * nb_sur_échantillons #création de la liste qui contiendra les n valeurs de la tension U2, avec n le nombre de sur échantillons
132             for i in range(nb_sur_échantillons):
133                 tension_U_analog[i] = ADC(Pin(PIN_U2[i], Pin.IN)) #initialisation du GPIO de mesure de U2
134                 ##REF!.value(1) #utilise uniquement à la mesure de la fréquence d'échantillonnage avec oscilloscope
135                 tensionU2[i] = tensionU2[i] * (ref[j] / 65535) #conversion de la valeur numérique de 16 bits récupérée par l'ADC en volts
136                 tension_U_analog[i] = ADC(Pin(PIN_U[i], Pin.IN)) #même processus que pour U2
137                 ##REF!.value(0)
138                 tensionU[i] = tension_U_analog[i].read_u16()
139                 ##REF!.value(1)
140                 tensionU[i] = tension_U_analog[i].read_u16()
141                 ##REF!.value(0)
142                 U[i] = tensionU[i] * (ref[j] / 65535)
143                 U_moy = sum(U) / nb_resistances #moyenne des sur-échantillons pour avoir les valeurs finales de U et U2 avec lesquelles calculer la valeur de résistance échar
144                 U2_moy = sum(U2) / nb_resistances
145                 del U #Libération de RAM
146                 del U2
147                 R2i = (U2_moy * R1[j]) / (U_moy - U2_moy) #calcul intermédiaire
148                 if len(R2) < nb_resistances*nb_sur_échantillons: #les n = nb_resistances*nb_échantillons premières valeurs sont automatiquement ajoutées à la li
149                     R2.append(R2i)
150                     debut.append(utime.ticks_us()*1e-6) #prise du temps de mesure de la résistance pour obtenir les intervalles de temps nécessaires au calcul des paramètres
151                 else:
152                     if (R2i > R2[len(R2)-(nb_resistances)]*(1-pourcentage_danger)) and (R2i < R2[len(R2)-(nb_resistances)]*(1+pourcentage_danger)): #comparaison au pourcent
153                         #print("Les échantillons respectent un écart de : ", pourcentage_danger*100, "%")
154                         D2.append(D2i)

```

```

153             #print("Les échantillons respectent un écart de : ", pourcentage_danger*100, "%")
154             R2.append(R2i)
155             debut.append(utime.ticks_us()*1e-6)
156         else:
157             #print("Alerte!")
158             alerte = True #déclenchement de l'alerte si les valeurs sortent en dehors de l'intervalle de sécurité défini par le pourcentage de danger
159         if alerte:
160             print("Alerte !")
161         return R2, alerte, debut
162
163 def slice(R2, nb_resistances):
164     R_slice = [[] for _ in range(nb_resistances)]
165     for i in range(nb_resistances):
166         for j in range(i+nb_resistances, len(R2), nb_resistances):
167             R_slice[i].append(R2[j])
168     return R_slice
169 #=> la liste R_slice est une liste contenant n listes, chacune contenant toutes les valeurs récupérées pour chaque résistance. (avec n le nombre de résistances)
170 #La liste originelle R2 contient n listes dans lesquelles il y a les valeurs de toutes les résistances récupérées pour chaque échantillon.
171
172
173 def calcul_integrale(nb_resistances, debut, R2, k):
174     integrales = [] #liste qui stocke temporairement les valeurs d'intégrales pour chaque échantillon avant de les ajouter à la somme totale des intervalles sur tous les
175     for i in range(0, nb_resistances):
176         integrales.append([0, 0])
177         integrale.append([0])
178     for i in range(0, len(R2)-(2*nb_resistances)-1, nb_resistances): #on parcourt la liste des résistances de 0 aux avant-dernières valeurs de résistance, (toutes les
179         if i == 0:
180             debut_precedent = 0
181         else:
182             debut_precedent = debut[i-1]
183         for j in range(0, nb_resistances):
184             integrales[j].append(((R2[i+j]+R2[i+nb_resistances+j])/2)*(debut[i]-debut_precedent))) #calcul d'intégrale d'un échantillon avec la méthode des trapèzes
185             integrale[j][0] = integrale[j][0] + integrales[j][len(integrales[j])-1] #ajout de la dernière intégrale calculée à la somme des intégrales de tous les échanti
186     del integrales
187     return integrale
188
189 def calcul_delta(R_slice, nb_resistances):
190     R_steady = []
191     R_initial = []
192     R_initial = []
193     for i in range(0, nb_resistances):
194         R_steady.append(0)
195         R_initial.append(0)
196         delta.append(0)
197         R_steady[i] = max(R_slice[i], key=abs)
198         R_initial[i] = min(R_slice[i], key=abs)
199         delta[i] = [(R_steady[i]-R_initial[i])/R_initial[i]]
200     del R_steady
201     del R_initial
202     del R_slice
203     return delta
204 #=> cette fonction calcule le paramètre delta selon la formule données à la p. 619 du document sur l'analyse de présence d'ammoniaque
205
206 def calcul_derivee_max(R_slice, debut, nb_resistances):
207     derivee = [[] for _ in range(nb_resistances)]
208     for i in range(0, len(R_slice[0])-1):
209         if i == 0:
210             debut_precedent = 0
211         else:
212             debut_precedent = debut[i-1]
213         for j in range(0, nb_resistances):
214             derivee[j].append((R_slice[j][i]-R_slice[j][i-1])/(debut[i]-debut_precedent))
215     derivee_max.append(max(derivee[j], key=abs))
216     del derivee
217     return derivee_max
218
219
220 def calcul_parametres(R2, alerte, nb_resistances, ref, R1, PIN_U, PIN_U2, tension_U2_analog, tensionU2, tension_U_analog, tensionU, attente_us, total_sample, nb_donnees_p
221     #creation des listes de stockage des paramètres
222     integralef = [[0] for _ in range(nb_resistances)]
223     derivee_maxf = [[0] for _ in range(nb_resistances)]
224     deltaf = [[0] for _ in range(nb_resistances)]
225     sommef = [[0] for _ in range(nb_resistances)]
226
227     #le calcul des paramètres sera lancé n fois avec n = nb_donnees_pour_classif:
228     for k in range(nb_donnees_pour_classif):
229         #calcul des paramètres :
230         integrale = calcul_integrale(nb_resistances, debut, R2, k)
231         derivee_maxf = calcul_derivee_max(R_slice, debut, nb_resistances)
232         deltaf = calcul_delta(R_slice, nb_resistances)
233
234         for i in range(nb_resistances):
235             integralef[i].append(integrale[i][0]) #ajoute la valeur d'intégrale calculée pour chaque résistance à la liste qui stockera toutes les données de paramètre
236             derivee_maxf[i].append(derivee_max[i][0])
237             deltaf[i].append(delta[i][0])
238             sommef[i].append(integralef[i][k] + abs(deltaf[i][k]) + abs(derivee_maxf[i][k]))
239
240     #print("Paramètres acquis pour la, k, ième fois")
241
242     #libération de l'espace mémoire :
243     integrale.clear()
244     deltaf.clear()
245     derivee_maxf.clear()
246     debut.clear()
247     R_slice.clear()
248     R2.clear()
249     R2, alerte, total_sample = main(ref, R1, PIN_U, PIN_U2, tension_U2_analog, tensionU2, U2, tension_U_analog, tensionU, U, alerte, attente_us, nb_resistances, total
250
251     return integralef, derivee_maxf, deltaf, sommef
252
253
254 def classification(R2, alerte, nb_resistances, ref, R1, PIN_U, PIN_U2, tension_U2_analog, tensionU2, tension_U_analog, tensionU, attente_us, total_sample, nb_donnees_pou
255     print("Classification lancée")
256     #obtention des paramètres :
257     integralef, derivee_maxf, deltaf, sommef = calcul_parametres(R2, alerte, nb_resistances, ref, R1, PIN_U, PIN_U2, tension_U2_analog, tensionU2, tension_U_analog, tens
258     #print("commencement for")
259     #lancement de la fonction random forest :
260     DMMF = foret(integralef, derivee_maxf, deltaf, sommef) #DMMF est un booléen, true si la forêt le détecte, false sinon
261     #print("deltaf =", deltaf)
262     #print("integralef =", integralef)
263     #print("derivee_maxf =", derivee_maxf)
264     del deltaf
265     del derivee_maxf
266     del integralef
267     print("Classification terminée.")
268     return DMMF

```

```

271
272 def type_aleatoire() :
273     #On choisit le paramètre que le noeud va évaluer avec parametres = ("delta", "integrale", "derivee_max", "somme")
274     return random.choice(parametres)
275
276 def type_aleatoire_pondere() :
277     #le choix du paramètre que le noeud va évaluer dépend des valeurs de précision obtenue en fonction des paramètres utilisés avec l'algorithme random forest
278     #ces valeurs sont expérimentales et proviennent du tableau 3 dans le document sur l'analyse de présence d'ammoniaque p. 622
279     ponderations = [0.77, 0.50, 0.77, 0.80]
280     poids_total = sum(ponderations)
281     rand = random.uniform(0, poids_total) #choix d'un nombre aléatoire parmis l'intervalle [0, poids total des pondérations]
282     cumul = 0
283     #dans la boucle suivante on additionne les poids de chaque paramètre jusqu'à ce que le cumul des poids soit supérieur au nombre aléatoire choisi
284     #Le paramètre qui sera dépasser le cumul est sélectionné
285     for random.choice(parametre), poids in zip(parametres, ponderations):
286         cumul += poids
287         if rand <= cumul:
288             return parametre
289
290
291 def donnees_aleatoires(type_noeud, n, integralef, derivee_maxf, deltaf, sommef) :
292     #n représente la résistance choisie dans la boucle de la fonction foret
293     #une fois que au hasard, les paramètres calculés est sélectionnée, en fonction du type du noeud
294     if type_noeud == 'delta' :
295         donnee = random.choice(deltaf[n])
296     if type_noeud == 'derivee_max' :
297         donnee = random.choice(derivee_maxf[n])
298     if type_noeud == 'integrale' :
299         donnee = random.choice(integralef[n])
300     if type_noeud == 'somme' :
301         donnee = random.choice(sommef[n])
302     return donnee
303
304
305 def verif_noeud(type_noeud, donnee) :
306     #définition de la condition de détermination de la présence de DMMP pour le noeud, en fonction de son type
307     #les valeurs de seuils sont des hyper paramètres à déterminer avec l'entraînement
308     #voir si il est mieux de choisir le seuil par random.choice ou de comparer la valeur aléatoire au min et max du seuil
309
310
311
312
313
314
315
316
317
318
319
320
321 def arbre_decisionnel(n, integralef, derivee_maxf, deltaf, sommef) :
322     vote_noeuds = 0 #initialisation des votes des noeuds
323     vote_arbre = 0 #initialisation des vote globaux des arbres
324     nb_noeuds = random.randint(nb_noeuds_min, nb_noeuds_max) #le nombre de noeud de chaque arbre est choisi de manière aléatoire
325     type_noeud = type_aleatoire() #le type du premier noeud est choisi au hasard
326     for i in range(0, nb_noeuds) :
327         #selection du paramètres à évaluer de manière aléatoire en fonction du type du noeud :
328         donnee = donnees_aleatoires(type_noeud, n, integralef, derivee_maxf, deltaf, sommef)
329         #vérification du respect de la condition de présence de DMMP :
330         condition = verif_noeud(type_noeud, donnee)
331         if condition : #si il y a présence de DMMP :
332             vote_noeuds += 1 #le noeud vote pour dmmp
333             i += 1
334         type_noeud = type_aleatoire_pondere() #le type du noeud suivant sera déterminé avec le choix aléatoire pondéré pour améliorer la précision
335     else :
336         # "création" d'un nouveau noeud avec un type choisi de manière totalement aléatoire :
337         type_noeud = type_aleatoire()
338     if vote_noeuds >= (nb_noeuds/2) :
339         vote_arbre += 1
340         # -> l'arbre vote pour la présence de DMMP si plus de la moitié de ses noeuds votent ainsi
341     return vote_arbre
342
343 def foret(integralef, derivee_maxf, deltaf, sommef) :
344     vote_foret = 0
345     for n in range (0, nb_arbres) :
346         for n in range(0, nb_resistances) :
347             #création d'un arbre pour chaque résistance avec n = le nombre d'arbre décidé en tant qu'hyper paramètre
348
349             #voir si il est mieux de choisir le seuil par random.choice ou de comparer la valeur aléatoire au min et max du seuil
350             if type_noeud == 'delta' :
351                 condition = donnee > random.choice(seuil_delta) or donnee < random.choice(seuil_delta)
352             if type_noeud == 'derivee_max' :
353                 condition = donnee > random.choice(seuil_derivee_max) or donnee < random.choice(seuil_delta)
354             if type_noeud == 'integrale' :
355                 condition = donnee > random.choice(seuil_integrale) or donnee < random.choice(seuil_delta)
356             if type_noeud == 'somme' :
357                 condition = donnee > random.choice(seuil_somme) or donnee < random.choice(seuil_delta)
358             return condition
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990

```