**Groupe A : Projet 1**

**BRUNEL Terry**

**THELIER Rémi**

**Rapport Projet 1 : Algorithmique et Programmation**

**Retour d’expérience sur GitHub**

Nous nous sommes tous les deux créer un compte GitHub. Nous avons eu des difficultés à l’installer et le lancer. En effet, nous n’avions pas bien compris toutes les étapes et lignes de commande à écrire sur le tableau de commande noir après le téléchargement et installation de Git. Une fois cette étape franchie grâce au visionnage de nombreux tutoriels, nous avons réussi à nous créer un compte utilisable. GitHub nous a été utile lorsque nous travaillions séparément, nous téléchargions le fichier python sur les comptes de l’un et de l’autre pour voir comment l’autre avait avancé le projet et pouvoir continuer à la suite. Cependant, lorsque nous nous sommes fait des séances programmation en vision pendant les horaires de « Algorithmique et Programmation », nous travaillions directement sur Spyder.

**Retour d’expérience sur Python**

Terry : je n’avais jamais fait de Python, ce projet a donc été ma première expérience sur Python.

Rémi : j’ai pratiqué Python pendant 2 ans en CPGE, cela m’a uniquement permis de faire les tris pendant ce projet. Cependant, je n’avais aucune connaissance en matière de fonctions python et d’importation de modules. Ce projet a été une découverte sur beaucoup de fonctionnalités dans Python.

**Choix des algorithmes**

Nous avons tout d’abord importé plusieurs bibliothèques, à commencer par Panda car il nous permet d’importer les données du documents Excel :

data = pd.read\_csv(r"C:\Users\remit/ProjetAlgo/EIVP\_KM.csv",sep=';')

Nous avons aussi importé les bibliothèques numpy pour apporter des fonctionnalités mathématiques et matplotlib pour tracer des courbes.

Après l’import des données Excel, nous avons transformé ces tableaux de données en liste de données grâce à la fonctionnalité **tolist()**. Voici 2exemples :

**noise=data["noise"].tolist()**

**temperature=data["temp"].tolist()**

**Projet Partie 1**

Tout d’abord, nous avons procédé à un tri de la liste date avec la fonction **Tri\_date()** prenant la liste sentat=data["sent\_at"].tolist() en argument. On utilise pour cela 1 boucle for contenant une boucle for qui elle contient une condition if. Cette condition compare les dates pour trier les dates et les autres listes de données dans l’ordre croissant et chronologique.

La fonction **Tri\_date(sentat)** est appelé date : date=Tri\_date(sentat)

Le programme de la Partie 1 se compose de 2 « parties » dû à la condition if.

L’utilisateur devra choisir s’il veut entrer « une demande d’un intervalle de temps ou non ».

Alors la 1ere partie est celle fonctionnant avec un intervalle de temps demandé, on l’appellera par la suite **« programme secondaire »** étant donné qu’il s’agissait d’une option dans le sujet. La 2ème partie est celle dont toutes les dates sont parcourues, que nous appellerons **« programme principal »** puisque ce programme était demandé dans le sujet.

Pour le « programme principal », on a trié la liste date par capteur que l’on nomme time1, ….,time 6, puis on attribue les éléments de la liste température selon la liste de temps par capteurs aux listes cap1\_temp, …., cap1\_co2 (pour capteur 1)

time1=[]

time2=[]

time3=[]

time4=[]

time5=[]

time6=[]

for i in range(len(date)):

if idcapteur[i]==1: -> si l’id du capteur vaut 1 alors on ajoute la date à time 1

time1.append(date[i])

elif idcapteur[i]==2: : -> si l’id du capteur vaut 2 alors on ajoute la date à time 2

time2.append(date[i])

elif idcapteur[i]==3: :

time3.append(date[i])

elif idcapteur[i]==4: :

time4.append(date[i])

elif idcapteur[i]==5: :

time5.append(date[i])

elif idcapteur[i]==6: : -> si l’id du capteur vaut 6 alors on ajoute la date à time 6

time6.append(date[i])

cap1\_temp=[]

cap1\_hum=[]

cap1\_lum=[]

cap1\_noi=[]

cap1\_co2=[]

for i in range(len(time1)):

cap1\_temp.append(temperature[i])

cap1\_hum.append(humidite[i])

cap1\_lum.append(luminosite[i])

cap1\_noi.append(noise[i])

cap1\_co2.append(co2[i])

**-Afficher des courbes montrant l’évolution d’une variable en fonction du temps**

Nous avons utilisé les fonctionnalités de matplotlib pour afficher les courbes : **plt.plot( , )** et **plt.show()**.

Voici toutes les fonctions de matplotlib que l’on a utilisées :

**plt.plot( , )** : permet le traçage de la courbe, label=’…’ permet de donner un nom à la courbe, nous avons afficher les courbes en bleu et en rouge pour l’indice de corrélation avec respectivement la notation ‘b—‘ (blue) et ‘r—‘ (red).

**plt.title()**: permet d’écrire un titre

**plt.annotate()**: permet d’annoter, d’écrire du texte sur un graphique. Il faut donner les coordonnées x et y pour bien placer son texte.

plt.annotate("maximum:",(1.0,0.8),xycoords="axes fraction")

plt.annotate(maxi(luminosite),(1.20,0.8),xycoords="axes fraction") -> on fait appel aux fonctions statistiques pour afficher les valeurs sur le graphique.

**plt.xlabel()** et **plt.ylabel()** : permettent de préciser respectivement le nom des données en abscisse et en ordonnées.

**plt.legend()** permet d’afficher la légende, les annotations, le titre, les abscisses et ordonnées.

**plt.show()** : affiche les courbes sur la console

Lorsque l’utilisateur demande les graphes d’une donnée, les abscisses sont toujours les dates, rangées par ordre chronologique.

Le 1er graphe à s’afficher est celui de la donnée demandée enregistrée par **tous** les capteurs en fonction du temps (abscisse : **date**) par appel de la fonction date. L’affichage de ce graphe est fonctionnel dans le « programme principal » et le « programme secondaire »

Puis, dans le « programme principal », les 6 graphes suivant correspondent aux données de chaque capteur en fonction de date. Chaque graphe affichant par capteur nous permet de voir l’enregistrement propre au capteur correspondant.

L’affichage de ces 6 courbes n’est pas disponible dans le « programme secondaire » à cause d’un problème développé dans la sous-partie **BONUS : Programme pour créer une liste selon un intervalle de temps compris entre la start\_date et la end\_date demandée par l’utilisateur**.

Nous aurions pu créer une fonction affichant des graphes prenant comme argument des variables (a,b) pour ensuite faire appel à cette fonction. Cependant, bien que plus rapide, cette méthode ne permet pas d’afficher un titre nommant exactement le nom de la donnée étudiée et d’afficher son unité à côté des valeurs ou des ordonnées.

**-Afficher les valeurs statistiques sur la courbe**

Nous avons décidé de programmer les fonctions de statistiques même si nous aurions pu les utiliser directement via la bibliothèque.

* **maximum(Y**) : cette fonction permet de retourner le maximum d’une liste de données. Elle prend comme argument une liste. On crée une variable ici nommée « maxi » à laquelle on affecte la valeur du premier élément de la liste. Ensuite, on compare tous les éléments de la liste à la variable maxi grâce à une boucle for (permet de parcourir tous les éléments de la liste) et un if. Cet if apporte une condition, en l’occurrence il permet de comparer un élément de la liste à maxi, et d’être affecté à maxi s’il est supérieur ou égal à maxi. Enfin, une fois tous les éléments de la liste comparés et parcourus, on retourne maxi.
* **minimum(Y) :** cette fonction permet de retourner le minimum d’une liste de données. Elle prend comme argument une liste. On crée une variable ici nommée « mini » à laquelle on affecte la valeur du premier élément de la liste. Ensuite, on compare tous les éléments de la liste à la variable mini grâce à une boucle for (permet de parcourir tous les éléments de la liste) et un if. Cet if apporte une condition, en l’occurrence il permet de comparer un élément de la liste à mini, et d’être affecté à mini s’il est inférieur ou égal à mini. Enfin, une fois tous les éléments de la liste comparés et parcourus, on retourne mini.
* **mediane(Y) :** cette fonction permet de calculer la médiane c’est-à-dire la valeur centrale de la liste. Nous avons utilisé la fonction **sorted(Y)** permettant de trier dans l’ordre croissant les éléments de la liste. Il y a différents cas dans le calcul d’une médiane : cas 1 pour lequel la liste est impaire, cas 2 pour lequel la liste est paire, cas 3 si la liste est vide et auquel cas il n’y a pas de médiane. Nous avons créer alors une condition avec if pour déterminer si la liste est impaire ou paire : si le résultat de la division euclidienne de la liste par 2 est nulle alors le nombre d’éléments de la liste est pair puis on retourne l’élément milieu de la liste, si le résultat de la division euclidienne par 2 est 1 alors le nombre d’éléments de la liste est impair et on retourne la moyenne des 2 éléments centraux (on utilise la fonction **sum()** pour sommer ces 2 valeurs et ensuite faire la moyenne en divisant par 2.
* **moyenne(Y) :** cette fonction permet de calculer la moyenne d’une liste. Elle prend une lliste comme argument. Pour initialiser, on crée une variable ici nommée « mo ». A l’aide d’une boucle for, on calcule la somme des éléments de la liste. Ensuite, on divise cette somme par la longueur de la liste (utilisation de len(liste), soit par le nombre d’éléments de la liste. Puis, on impose 2 chiffres après la virgule avec la fonction moy=**round(mo,2)** Enfin, on retourne moy.
* **variance(Y) :** cette fonction permet de calculer la variance d’une liste. La fonction prend une liste comme argument. Nous avons affecté la fonction **moyenne(Y)** à une variable nommée **moy**. Nous avons également initialisé la variable va en lui attribuant la valeur 0. Avec une boucle for, on parcourt la liste de longueur **len(Y)** et on calcule la somme des écarts à la moyenne (valeur de l’élément – moyenne de la liste), le tout au carré. Une fois la boucle for terminé, on divise va par la longueur de la liste. Puis, on impose 2 chiffres après la virgule avec la fonction var=**round(va,2)**On retourne la variance.
* **ecart\_type(Y) :** cette fonction permet de calculer l’écart-type d’une liste. On prend une liste comme argument. Par définition, l’écart-type est la racinée carrée de la variance. On fait appel à la fonction **variance(Y)** et on la met à la puissance 0.5 (racine carrée). Puis, on impose 2 chiffres après la virgule avec la fonction ecart=**round(ecar,2).**On retourne ainsi l’écart-type.

**-Calculer l’indice « humidex »**

Nous avons utilisé une formule publiée sur Wikipédia pour calculer l’indice « humidex ».

 \text{Humidex} = t_{air} + 0,5555 \times (6,11 \times e^{5417,7530 \times \left(\frac{1}{273.16} - \frac{1}{{T_d}}\right)} - 10)

{\displaystyle {\text{Humidex}}=T\_{\text{air}}+0.5555\left[6.11\times e^{5417.7530\left({\frac {1}{273.16}}-{\frac {1}{273.15+T\_{\text{rosee}}}}\right)}-10\right]}

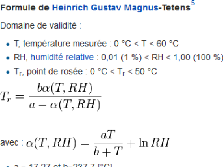
tair est la température de l'air ([degré Celsius](https://fr.wikipedia.org/wiki/Degr%C3%A9_Celsius))

[e = 2,71828](https://fr.wikipedia.org/wiki/E_(nombre))

Td est le [point de rosée](https://fr.wikipedia.org/wiki/Point_de_ros%C3%A9e) (degré Celsius)

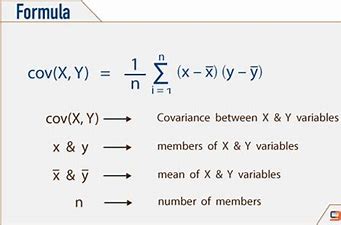
Avant d’utiliser la formule de l’indice humidex, il nous a fallu trouver la température de rosée. Nous avons calculé la température de rosée (ici nommée Tr) à l’aide la formule suivante trouvée sur Wikipédia :

L’humidité relative nous étant donnée sur Excel, nous avons pu calculer la température de rosée sans oublier de diviser l’humidité relative par 100 pour obtenir une valeur entre 0 et 1 dans le log comme indiquer sur la formule.

Aussi, pour simplifier les calculs et faciliter la lisibilité, nous avons décidé de créer une fonction **T\_ros** calculant la température de rosée. Elle retourne une liste T\_ros. Et nous avons fait appel à cette fonction lors du calcul de l’indice humidex dans la fonction **humidex()** en mettant la liste de la température de rosée en argument. Cette fonction retourne la liste H

**-Calculer l’indice de corrélation entre deux variables**

Pour calculer l’indice de corrélation entre deux variables, il a fallu tout d’abord calculer la covariance de ces 2 variables. La covariance est la moyenne des produits des écarts des valeurs à la moyenne de chaque série. Nous avons donc utilisé la formule suivante :

La fonction **cov(X,Y)** calculant la covariance de 2 listes de données prend en argument 2 listes choisies par l’utilisateur.

On initialise 2 listes. Grâce à une boucle for, on ajoute à l’une des 2 listes initialisées la moyenne des produits des écarts des éléments à la moyenne de leurs listes de données respectives. Enfin, la 2e liste initialisée divisent tous les éléments de la 1ère liste par le nombre d’éléments d’une liste de données.

**-BONUS : Programme pour créer une liste selon un intervalle de temps compris entre la start\_date et la end\_date demandée par l’utilisateur :**

Difficulté avec la création d'une start\_date et end\_date : difficulté dans le passage de la chaine de caractère à une liste pouvant être utilisée dans le programme de tri. En effet gênés par le format date ( **-** et  **:**), nous avons dû passer par une étape intermédiaire, celle de créer et séparer la chaine de caractère puis de la transformer en integer avec **int()**. Nous avons ensuite simplifié en ne prenant que les indices correspondant aux jours car les données sont de la même année et du même mois, cela permet d’alléger le programme.

La fonction **Newdate(Y**) permet de transformer les dates au bon format de liste d’integer (exemple : 2019-08-15 15 :20 :26 devient par **Newdate** 15152026. Nous avons eu des difficultés après avoir réalisé la fonction **Newdate(Y)** car nous avions enlevé la précision des heures. De ce fait, lors de l'apparition des courbes, les valeurs des données (ordonnées) d'un jour identique étaient entassées sur le même abscisse. Nous avons dû augmenter la précision en rajoutant 000000 et 235959 pour trier correctement.

Ensuite, nous avons créé un programme **indicestart(start\_date)** qui retourne l’index de la première date supérieure à la start\_date entrée par l’utilisateur. Pour cela, nous avons employé une boucle for qui parcoure tous les éléments de la liste avec un if pour comparer les éléments de la liste à la start\_date. Nous avons créé un programme **indeceend(end\_date)** presque identique pour renvoyer l’index de la dernière date inférieure à la end\_date entrée par l’utilisateur. Nous avons également utilisé une boucle for et une condition if.

Enfin, nous avons créé les listes de données à partir de ces indices (exemple : **SEnewdate=newdate[indicestart(start\_date):indiceend(end\_date)]** et **SEco2=co2[indicestart(start\_date):indiceend(end\_date)]** )

Nous avons ensuite fait face à un problème. En effet, avec les listes **SEnewdate,… ,SEco2,** nous avons voulu redistribuer le temps et les différentes données à chaque capteur respectif de la même manière que pour le programme principal (cette manière fonctionne pour le programme principal c’est-à-dire celui sans intervalle de temps demandée). Pourtant, cette fois-ci cela ne fonctionnait pas exactement. Par exemple si on demandait des dates du 15 août au 19 août, la fonction **SEtime1()** ne retournait que les valeurs du 15 août au 17 août. Nous n’avons pas réussi à résoudre ce problème et espérons connaître la raison et la solution pour le contourner.

**Exemple programme secondaire (avec demande d’intervalle) :**

SEtime1=[]

SEtime2=[]

SEtime3=[]

SEtime4=[]

SEtime5=[]

SEtime6=[]

for i in range(len(SEnewdate)):

if idcapteur[i]==1:

SEtime1.append(SEnewdate[i])

elif idcapteur[i]==2:

SEtime2.append(SEnewdate[i])

elif idcapteur[i]==3:

SEtime3.append(SEnewdate[i])

elif idcapteur[i]==4:

SEtime4.append(SEnewdate[i])

elif idcapteur[i]==5:

SEtime5.append(SEnewdate[i])

elif idcapteur[i]==6:

SEtime6.append(SEnewdate[i])

**Exemple programme principal :**

time1=[]

time2=[]

time3=[]

time4=[]

time5=[]

time6=[]

for i in range(len(date)):

if idcapteur[i]==1:

time1.append(date[i])

elif idcapteur[i]==2:

time2.append(date[i])

elif idcapteur[i]==3:

time3.append(date[i])

elif idcapteur[i]==4:

time4.append(date[i])

elif idcapteur[i]==5:

time5.append(date[i])

elif idcapteur[i]==6:

time6.append(date[i])

**Projet Partie 2 : Sujet 1- Les anomalies**

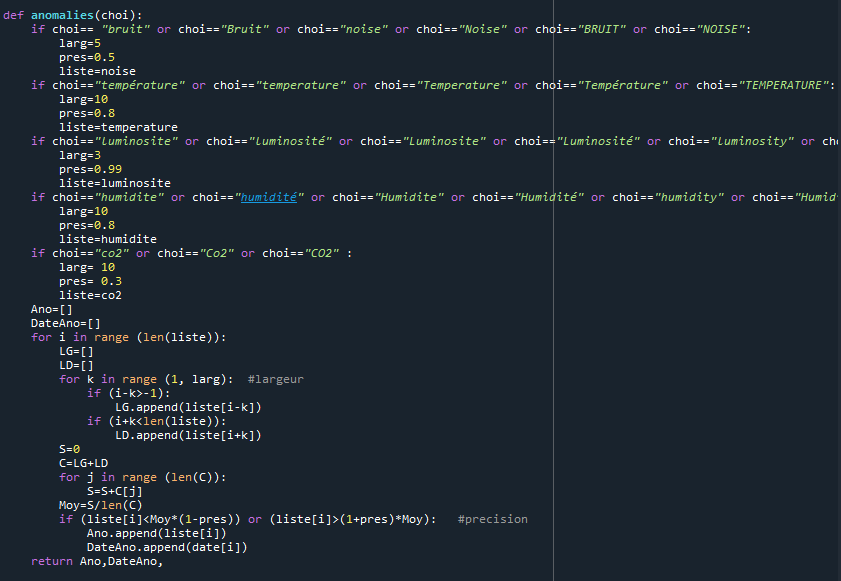
En parcourant le fichier Excel à travers les listes créées sur Python, nous avons pu constater certaines incohérences au sein des valeurs. Ne connaissant pas les modalités des anomalies nous avons décidé de traiter les anomalies de valeurs et de temps.

**Anomalies 1 : type valeurs**

nous avons pu voir que des écarts importants apparaissaient d’une donnée à l’autre au sein des listes de données. Ainsi nous avons défini la fonction « anomalies » permettant de détecter les valeurs ayant un écart à la moyenne trop important. Le but étant de repérer les anomalies de type :

21-23-30-22-23.

Voici ci-dessous la fonction :



Les boucles IF en entrée de la fonction permettent de faire varier les paramètres de précision et largeur de données selon le choix de données fait par l’utilisateur. Le choix fait par l’utilisateur se rapporte directement à une des listes créées au départ du programme.

En voici un exemple :

if choi=="co2" or choi=="Co2" or choi=="CO2" :

larg= 11 🡪 10 valeurs avant et après la valeurs i sont récoltées.

pres= 0.3 🡪 Précision de 30% d’écart à la moyenne.

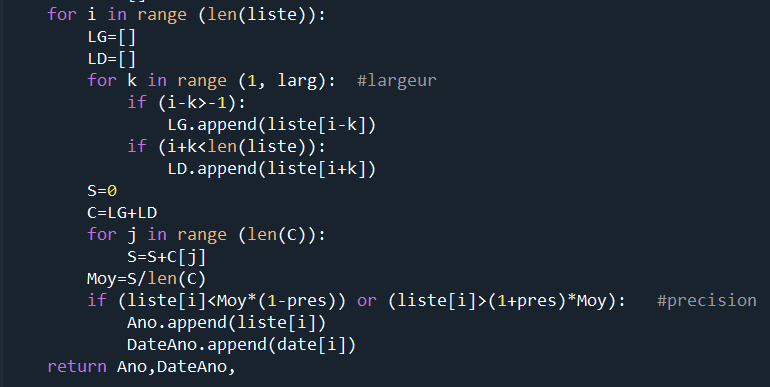
liste=co2

Nous avons décidé de ne pas faire un écart à la moyenne fixe car nous pensions que cela serait trop réducteur et que trop de mesures seraient comptabilisées comme anomalies.

Ainsi le programme s’effectue de la sorte que le paramètre « larg » désigne le nombre de valeurs prises avant et après la valeur étudiée pour créer une moyenne arithmétique locale plus précise. En terme de précision nous avons choisi arbitrairement les précisions d’intervalle de confiance selon le type de données.

Ensuite deux liste vide sont créées Ano (pour anomalies) et DateAno (pour les dates correspondantes aux anomalies).

Puis, la boucle for est mise en place :



Dans la première boucle for nous initialisons deux listes LG et LD pour les données avant (i-k) et après (i+k) la donnée étudiée pour ensuite dans la seconde boucle for faire la moyenne de ces valeurs.

Pour finir la condition if permet de définir si la valeur est au-dessus ou en dessous de l’intervalle de confiance choisi dans les paramètres.

La fonction une fois appelée affiche la liste des anomalies et la liste de leurs dates correspondantes.

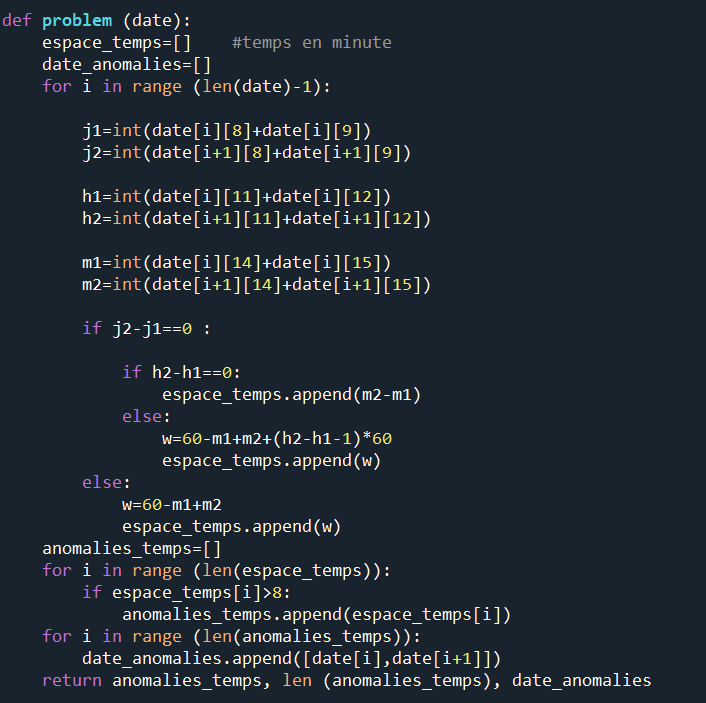
Résultat :

La fonction est efficace, retourne de nombreuse anomalie pour le bruit, le Co2 et l’humidité. Cependant les grandes variations de lumières allant de 0 à 712 sont plus compliquées à gérer puisqu’elles peuvent être compter comme des anomalies si les paramètres choisis ne sont pas assez précis.

**Anomalies 2 : type temps/ intervalle**

Nous avons aussi souhaité analyser les intervalles de temps entre les récoltes de données des capteurs. De manière générale, une valeur est récoltée toute les minutes cependant parfois des écarts de 10 minutes ou plus se font entre les récoltes de capteur. Nous avons donc conclus qu’un intervalle de temps trop grand pouvait engendrer des anomalies et être à l’origine des écarts de valeurs.

Ainsi nous avons codé une fonction permettant de détecter les intervalles de temps incohérents et d’afficher les dates de ces derniers.



La fonction définie commence par la création de deux listes vides pour y ajouter les intervalles de temps de toutes les données et une seconde liste pour les dates correspondantes aux anomalies de temps trouvés.

La première boucle « for » est mise en place pour comparer les dates de deux données successives (D1 et D2) d’une même liste. La comparaison se fait d’abord avec le jour ensuite avec l’heure puis avec les minutes. Nous ne nous sommes pas concentrés sur les secondes car nous avons déterminé qu’un écart de moins d’une minute n’était pas une anomalie.

La condition « IF » qui suit permet de faire la différence des dates de récoltes d’anomalies. Si nous sommes dans un cas ou les valeurs ont été prises entre deux heures différentes.

Si les valeurs sont récoltées dans les mêmes horaires :

Il suffit donc de faire la différence entre les minutes de la date de D2 et de la date de D1.

Or si les valeurs sont prises à des intervalles d’une heure ou plus la formule du dessus ne marche plus.

Ainsi la formule W permet de palier à cela.

Exemple 🡪 pour 22h58 et 23h10 🡪 nous devons trouver 12 min d’intervalle :

W = 60 – 58 + 10 +( 23 – 22 - 1 ) \* 60 = 12 min

Pour les intervalles entre 23h et 00h nous avons admis que l’intervalle ne serait pas de plus d’une heure donc nous avons appliqué cette formule :

W = 60 – 30+ 40 = 20 min

La seconde boucle « for » range dans la liste anomalies\_temps tous les intervalles plus grands que 8 minutes (valeur arbitraire).

La dernière boucle « for » implémente les dates aux bornes de l’intervalle de temps dans la liste date\_anomalies.

Pour terminer la fonction retourne les listes des anomalies, des dates correspondantes et le nombre d’anomalies.

Les anomalies sont directement affichés après les programmes à l’aide de 3 **print().**

**-BONUS : Programme permettant de trouver les périodes horaires d’occupation des bureaux**

Pour trouver les périodes horaires d’occupation des bureaux, nous devons d’abord trouver les dates d’occupation des bureaux.

Les bureaux ne sont pas occupés tous les jours, par exemple le week-end ou les jours fériés.

De plus, en étudiant les données, on se rend compte que le mois d’août a connu des périodes caniculaires étant donné que la température ne baissait pas la nuit.

Pour trouver les dates d’occupation des bureaux, il faut trouver les listes de données directement liées à la présence humaine :

-La température et l’humidité ne peuvent pas être utilisées même si elles augmentent s’il y a des présences humaines dans une pièce. Elles auraient pu être un indicateur, encore plus la nuit SAUF en cas de canicules. A cause des canicules, ces 2 données ne sont pas fiables.

-La quantité de CO2 aurait pu être indicateur or lors de temps caniculaire, le CO2 ne diminue que trop peu pour pouvoir être indicateur de présence humaine.

-La luminosité est un très bon indicateur de présence humaine d’autant plus en août lorsque les journées sont longues, qu’il fait nuit après 20h puisque les gens travaillent rarement au-delà de cette heure. Nous l’avons donc pris comme indicateur de présence humaine.

-Le bruit est le meilleur indicateur de présence humaine. En effet, en bureau, l’augmentation du bruit ne peut se faire que par des hommes. S’il y a du bruit, c’est qu’il s’agit d’une anomalie, et il y en a, c’est le seul point faible de cet indicateur.

La prise de la luminosité en indicateur et donc en facteur de sélectivité (en plus du bruit) pour la condition if nous révèlera que les employés ont oublié d’éteindre les lumières plusieurs nuits comme la nuit du 25 août 2019.

Le dernier indicateur pour trouver les périodes horaires est de relever les anomalies.

Après étude de l’évolution du bruit et de la luminosité le jour et la nuit, nous en avons observé grâce aux courbes et informations statistiques précédentes, que la nuit, le niveau sonore est de 27dBA sauf anomalies. Nous avons regardé sur internet les valeurs moyennes volume sonore dans un bureau : 40dBA

Nous avons créé une fonction **occup\_bureau** prenant la liste noise en argument. On initialise une liste occ\_bureau. On utilise une boucle for pour parcourir les éléments de la liste noise et on a ajouté une condition dans la boucle : **if noise[i]>40 and luminosite[i]>200 :**

Nous avons porté le seuil à 200 Lux en supposant qu’à 200 Lux il était déjà soit très tôt soit très tard pour un jour d’été, de plus l »éclairage moyen d’un bureau en Lux est de 500 Lux. Nous avons également pris en compte les anomalies avec la condition suivante : **or dateanom[i]!=date[i],** cette condition n’ajoute pas la date à la liste occ\_bureau si la date est une date d’anomalies.

On ajoute un élément à occ\_bureau dès que la condition est respectée.

En parallèle, nous avons programmée une fonction **nonoccup\_bureau** avec une condition inverse afin de vérifier si la somme des longueurs des 2 listes correspondait à la longueur totale des listes de données excel.

**ocbu=occup\_bureau(noise)**

La fonction **chiffrage** transforme la liste ocbu en une nouvelle liste de string grâce à **str()**

**c=chiffrage(ocbu)**

La fonction **chiff** modifie la liste c, en effet on ne garde que les chiffres et on oublie les notations « : ». Ainsi on a une liste d’entiers appelé **ch** dont on pourra trouver le minimum et le maximum.

**debhor=min(ch)**

**finhor=max(ch)**

**debhora=str(debhor) ->** on retransforme les listes en chaîne de caractère.

**finhora=str(finhor)**

**debuthoraire=debhora[0:2]+":"+debhora[2:4]+":"+debhora[4:6]** -> on ajoute à nouveau les « : » aux chaînes pour retrouver un format d’heure.

**finhoraire=finhora[0:2]+":"+finhora[2:4]+":"+finhora[4:6]**

**print("la période horaire d'occupation des bureaux est de",debuthoraire, "à",finhoraire) ----->**on affiche le résultat.

Nous avons trouvé des périodes horaires allant de 06:15:50 à 19:05:15

Cela semble excessif, nous n’avons sûrement pas été assez restrictif dans les paramètres et nous n’avons pas non plus pris en compte les anomalies ce qui explique l’horaire très matinal.