Ce problème s'intéresse à différents aspects relatifs à la sécurité aérienne et plus précisément au risque de collision entre deux appareils. Dans la première partie nous abordons l'enregistrement des plans de vol des différentes compagnies aériennes. La deuxième partie explique la manière d'attribuer à chaque vol un couloir aérien répondant au mieux aux souhaits des compagnies aériennes. Enfin, la troisième partie s'intéresse à la procédure de détection d'un risque de collision entre deux avions se croisant à des altitudes proches.

Une liste d'opérations et de fonctions qui peuvent être utilisées dans les fonctions Python figure en fin d'énoncé. Les candidats peuvent coder toute fonction complémentaire qui leur semble utile. Ils devront dans ce cas préciser le rôle de cette fonction, la signification de ses paramètres et la nature de la valeur renvoyée.

I Plan de vol

Afin d'éviter les collisions entre avions, les altitudes de vol en croisière sont normalisées. Dans la majorité des pays, les avions volent à une altitude multiple de 1000 pieds (un pied vaut 30,48 cm) au-dessus de la surface isobare à 1013,25 hPa. L'espace aérien est ainsi découpé en tranches horizontales appelées niveaux de vol et désignées par les lettres « FL » (*flight level*) suivies de l'altitude en centaines de pieds : « FL310 » désigne une altitude de croisière de 31000 pieds au-dessus de la surface isobare de référence.

Eurocontrol est l'organisation européenne chargée de la navigation aérienne, elle gère plusieurs dizaines de milliers de vol par jour. Toute compagnie qui souhaite faire traverser le ciel européen à un de ses avions doit soumettre à cet organisme un plan de vol comprenant un certain nombre d'informations : trajet, heure de départ, niveau de vol souhaité, etc. Muni de ces informations, Eurocontrol peut prévoir les secteurs aériens qui vont être surchargés et prendre des mesures en conséquence pour les désengorger : retard au décollage, modification de la route à suivre, etc.

Nous modélisons (de manière très simplifiée) les plans de vol gérés par Eurocontrol sous la forme d'une base de données comportant deux tables :

- la table `vol` qui répertorie les plans de vol déposés par les compagnies aériennes ; elle contient les colonnes
  - `id_vol` : numéro du vol (chaine de caractères) ;
  - `depart` : code de l'aéroport de départ (chaine de caractères) ;
  - `arrivee` : code de l'aéroport d'arrivée (chaine de caractères) ;
  - `jour` : jour du vol (de type `date`, affiché au format `aaaa-mm-jj`) ;
  - `heure` : heure de décollage souhaitée (de type `time`, affiché au format `hh:mi`) ;
  - `niveau` : niveau de vol souhaité (entier).

`id_vol`	`depart`	`arrivee`	`jour`	`heure`	`niveau`
AF1204	CDG	FCO	2016-05-02	07:35	300
AF1205	FCO	CDG	2016-05-02	10:25	300
AF1504	CDG	FCO	2016-05-02	10:05	310
AF1505	FCO	CDG	2016-05-02	13:00	310

Figure 1 Extrait de la table `vol` : vols de la compagnie Air France entre les aéroports Charles-de-Gaule (Paris) et Léonard-de-Vinci à Fiumicino (Rome)

- la table `aeroport` qui répertorie les aéroports européens ; elle contient les colonnes
  - `id_aero` : code de l'aéroport (chaine de caractères) ;
  - `ville` : principale ville desservie (chaine de caractères) ;
  - `pays` : pays dans lequel se situe l'aéroport (chaine de caractères).

`id_aero`	`ville`	`pays`
CDG	Paris	France
ORY	Paris	France
MRS	Marseille	France
FCO	Rome	Italie

Figure 2 Extrait de la table `aeroport`

Les types SQL `date` et `time` permettent de mémoriser respectivement un jour du calendrier grégorien et une heure du jour. Deux valeurs de type `date` ou de type `time` peuvent être comparées avec les opérateurs habituels (`=`, `<`, `<=`, etc.). La comparaison s'effectue suivant l'ordre chronologique. Ces valeurs peuvent également être comparées à une chaine de caractères correspondant à leur représentation externe (`'aaaa-mm-jj'` ou `'hh:mi'`).

I.A – Écrire une requête SQL qui fournit le nombre de vols qui doivent décoller dans la journée du 2 mai 2016 avant midi.

I.B – Écrire une requête SQL qui fournit la liste des numéros de vols au départ d'un aéroport desservant Paris le 2 mai 2016.

I.C – Que fait la requête suivante ?

```
SELECT id_vol
FROM vol
     JOIN aeroport AS d ON d.id_aero = depart
     JOIN aeroport AS a ON a.id_aero = arrivee
WHERE
     d.pays = 'France' AND
     a.pays = 'France' AND
     jour = '2016-05-02'
```

I.D – Certains vols peuvent engendrer des conflits potentiels : c'est par exemple le cas lorsque deux avions suivent un même trajet, en sens inverse, le même jour et à un même niveau. Écrire une requête SQL qui fournit la liste des couples $(\mathrm{Id}_1, \mathrm{Id}_2)$ des identifiants des vols dans cette situation.

II Allocation des niveaux de vol

Lors du dépôt d'un plan de vol, la compagnie aérienne doit préciser à quel niveau de vol elle souhaite faire évoluer son avion lors de la phase de croisière. Ce niveau de vol souhaité, le RFL pour *requested flight level*, correspond le plus souvent à l'altitude à laquelle la consommation de carburant sera minimale. Cette altitude dépend du type d'avion, de sa charge, de la distance à parcourir, des conditions météorologiques, etc.

Cependant, du fait des similitudes entre les différents avions qui équipent les compagnies aériennes, certains niveaux de vols sont très demandés ce qui engendre des conflits potentiels, deux avions risquant de se croiser à des altitudes proches. Les contrôleurs aériens de la région concernée par un conflit doivent alors gérer le croisement de ces deux avions.

Pour alléger le travail des contrôleurs et diminuer les risques, le système de régulation s'autorise à faire voler un avion à un niveau différent de son RFL. Cependant, cela engendre généralement une augmentation de la consommation de carburant. C'est pourquoi on limite le choix aux niveaux immédiatement supérieur et inférieur au RFL.

Ce problème de régulation est modélisé par un graphe dans lequel chaque vol est représenté par trois sommets. Le sommet 0 correspond à l'attribution du RFL, le sommet + au niveau supérieur et le sommet − au niveau inférieur. Chaque conflit potentiel entre deux vols sera représenté par une arête reliant les deux sommets concernés. Le cout d'un conflit potentiel (plus ou moins important en fonction de sa durée, de la distance minimale entre les avions, etc.) sera représenté par une valuation sur l'arête correspondante.

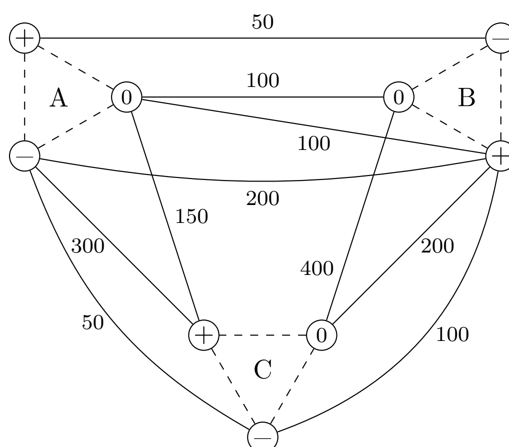

Figure 3 Exemple de conflits potentiels entre trois vols

Dans l'exemple de la figure 3, faire voler les trois avions à leur RFL engendre un cout de régulation entre A et B de 100 et un cout de régulation entre B et C de 400, soit un cout total de la régulation de 500 (il n'y a pas de

conflit entre A et C). Faire voler l'avion A à son RFL et les avions B et C au-dessus de leur RFL engendre un conflit potentiel de cout 100 entre A et B et 150 entre A et C, soit un cout total de 250 (il n'y a plus de conflit entre B et C).

On peut observer que cet exemple possède des solutions de cout nul, par exemple faire voler A et C à leur RFL et B au-dessous de son RFL. Mais en général le nombre d'avions en vol est tel que des conflits potentiels sont inévitables. Le but de la régulation est d'imposer des plans de vol qui réduisent le plus possible le cout total de la résolution des conflits.

II.A – Implantation du problème

Chaque vol étant représenté par trois sommets, le graphe des conflits associé à n vols $v_0, v_1, ..., v_{n-1}$ possède $3n$ sommets que nous numéroterons de 0 à $3n-1$. Nous conviendrons que pour $0 \leqslant k < n$:
- le sommet $3k$ représente le vol v_k à son RFL ;
- le sommet $3k+1$ représente le vol v_k au-dessus de son RFL ;
- le sommet $3k+2$ représente le vol v_k au-dessous de son RFL ;

Le cout de chaque conflit potentiel est stocké dans une liste de $3n$ listes de $3n$ entiers (tableau $3n \times 3n$) accessible grâce à la variable globale `conflit` : si i et j désignent deux sommets du graphe, alors `conflit[i][j]` est égal au cout du conflit potentiel (s'il existe) entre les plans de vol représentés par les sommets i et j. S'il n'y a pas de conflit entre ces deux sommets, `conflit[i][j]` vaut 0. On convient que `conflit[i][j]` vaut 0 si les sommets i et j correspondent au même vol (figure 4).

On notera que pour tout couple de sommets (i, j), `conflit[i][j]` et `conflit[j][i]`, représentent un seul et même conflit et donc `conflit[i][j] == conflit[j][i]`.

```
conflit = [ [   0,   0,   0, 100, 100,   0,   0, 150,   0],
            [   0,   0,   0,   0,   0,  50,   0,   0,   0],
            [   0,   0,   0,   0, 200,   0,   0, 300,  50],
            [ 100,   0,   0,   0,   0,   0, 400,   0,   0],
            [ 100,   0, 200,   0,   0,   0, 200,   0, 100],
            [   0,  50,   0,   0,   0,   0,   0,   0,   0],
            [   0,   0,   0, 400, 200,   0,   0,   0,   0],
            [ 150,   0, 300,   0,   0,   0,   0,   0,   0],
            [   0,   0,  50,   0, 100,   0,   0,   0,   0] ]
```

Figure 4 Tableau des couts des conflits associé au graphe représenté figure 3

II.A.1) Écrire en Python une fonction `nb_conflits()` sans paramètre qui renvoie le nombre de conflits potentiels, c'est-à-dire le nombre d'arêtes de valuation non nulle du graphe.

II.A.2) Exprimer en fonction de n la complexité de cette fonction.

II.B – Régulation

Pour un vol v_k on appelle *niveau relatif* l'entier r_k valant 0, 1 ou 2 tel que :
- $r_k = 0$ représente le vol v_k à son RFL ;
- $r_k = 1$ représente le vol v_k au-dessus de son RFL ;
- $r_k = 2$ représente le vol v_k au-dessous son RFL.

On appelle *régulation* la liste $(r_0, r_1, ..., r_{n-1})$. Par exemple, la régulation $(0, 0, ..., 0)$ représente la situation dans laquelle chaque avion se voit attribuer son RFL. Une régulation sera implantée en Python par une liste d'entiers.

Il pourra être utile d'observer que les sommets du graphe des conflits choisis par la régulation r portent les numéros $3k + r_k$ pour $0 \leqslant k < n$. On remarque également qu'au sommet s du graphe correspond le niveau relatif $r_k = s \bmod 3$ et le vol v_k tel que $k = \lfloor s/3 \rfloor$.

II.B.1) Écrire en Python une fonction `nb_vol_par_niveau_relatif(regulation)` qui prend en paramètre une régulation (liste de n entiers) et qui renvoie une liste de 3 entiers `[a, b, c]` dans laquelle a est le nombre de vols à leurs niveaux RFL, b le nombre de vols au-dessus de leurs niveaux RFL et c le nombre de vols au-dessous de leurs niveaux RFL.

II.B.2) Cout d'une régulation

On appelle *cout d'une régulation* la somme des couts des conflits potentiels que cette régulation engendre.

a) Écrire en Python une fonction `cout_regulation(regulation)` qui prend en paramètre une liste représentant une régulation et qui renvoie le cout de celle-ci.

b) Évaluer en fonction de n, la complexité de cette fonction.

c) Déduire de la question a) une fonction `cout_RFL()` qui renvoie le cout de la régulation pour laquelle chaque avion vole à son RFL.

II.B.3) Combien existe-t-il de régulations possibles pour n vols ?

Est-il envisageable de calculer les couts de toutes les régulations possibles pour trouver celle de cout minimal ?

II.C – L'algorithme Minimal

On définit le *cout d'un sommet* comme la somme des couts des conflits potentiels dans lesquels ce sommet intervient. Par exemple, le cout du sommet correspondant au niveau RFL de l'avion A dans le graphe de la figure 3 est égal à $100 + 100 + 150 = 350$.

L'algorithme *Minimal* consiste à sélectionner le sommet du graphe de cout minimal ; une fois ce dernier trouvé, les deux autres niveaux possibles de ce vol sont supprimés du graphe et on recommence avec ce nouveau graphe jusqu'à avoir attribué un niveau à chaque vol.

Dans la pratique, plutôt que de supprimer effectivement des sommets du graphe, on utilise une liste `etat_sommet` de $3n$ entiers tels que :
- `etat_sommet[s]` vaut 0 lorsque le sommet s a été supprimé du graphe ;
- `etat_sommet[s]` vaut 1 lorsque le sommet s a été choisi dans la régulation ;
- `etat_sommet[s]` vaut 2 dans les autres cas.

II.C.1)

a) Écrire en Python une fonction `cout_du_sommet(s, etat_sommet)` qui prend en paramètres un numéro de sommet s (n'ayant pas été supprimé) ainsi que la liste `etat_sommet` et qui renvoie le cout du sommet s dans le graphe défini par la variable globale `conflit` et le paramètre `etat_sommet`.

b) Exprimer en fonction de n la complexité de la fonction `cout_du_sommet`.

II.C.2)

a) Écrire en Python une fonction `sommet_de_cout_min(etat_sommet)` qui, parmi les sommets qui n'ont pas encore été choisis ou supprimés, renvoie le numéro du sommet de cout minimal.

b) Exprimer en fonction de n la complexité de la fonction `sommet_de_cout_min`.

II.C.3)

a) En déduire une fonction `minimal()` qui renvoie la régulation résultant de l'application de l'algorithme Minimal.

b) Quelle est sa complexité ? Commenter.

II.D – Recuit simulé

L'algorithme de *recuit simulé* part d'une régulation initiale quelconque (par exemple la régulation pour laquelle chacun des avions vole à son RFL) et d'une valeur positive T choisie empiriquement. Il réalise un nombre fini d'étapes se déroulant ainsi :
- un vol v_k est tiré au hasard ;
- on modifie r_k en tirant au hasard parmi les deux autres valeurs possibles ;
  - si cette modification diminue le cout de la régulation, cette modification est conservée ;
  - sinon, cette modification n'est conservée qu'avec une probabilité $p = \exp(-\Delta c/T)$, où Δc est l'augmentation de cout liée à la modification de la régulation ;
- le paramètre T est diminué d'une certaine quantité.

Écrire en Python une fonction `recuit(regulation)` qui modifie la liste `regulation` passée en paramètre en appliquant l'algorithme du recuit simulé. On fera débuter l'algorithme avec la valeur $T = 1000$ et à chaque étape la valeur de T sera diminuée de 1%. L'algorithme se terminera lorsque $T < 1$.

Remarque. Dans la pratique, l'algorithme de recuit simulé est appliqué plusieurs fois de suite en partant à chaque fois de la régulation obtenue à l'étape précédente, jusqu'à ne plus trouver d'amélioration notable.

III Système d'alerte de trafic et d'évitement de collision

L'optimisation globale des niveaux de vol d'un système d'avions étudiée précédemment est complétée par une surveillance locale dans l'objectif de maintenir à tout instant une séparation suffisante avec tout autre avion. La règlementation actuelle impose aux avions de ligne d'être équipé d'un système embarqué d'évitement de collision en vol, ou TCAS pour *traffic collision avoidance system*.

Nous nous intéressons au fonctionnement du TCAS vu d'un avion particulier que nous appelons *avion propre*. Les avions qui volent à proximité de l'avion propre sont qualifiés d'*intrus*.

Le système TCAS surveille l'environnement autour de l'avion propre (typiquement dans un rayon d'un soixantaine de kilomètres) pour identifier les intrus et déterminer s'ils présentent un risque de collision. Pour cela, le TCAS évalue, pour chaque intrus, le point où sa distance avec l'avion propre sera minimale. Ce point est appelé CPA, pour *closest point of approach*. Le fonctionnement global (simplifié) du système TCAS est décrit par la fonction `TCAS` donnée figure 5.

Cette fonction maintient une liste des CPA des avions situés dans son périmètre de surveillance (variable `CPAs`). Cette liste est limitée à un nombre restreint d'intrus (`intrus_max`) dont l'instant du CPA est proche (dans moins de `suivi_max`) de façon à garantir la détection et le traitement d'un éventuel danger dans un temps suffisamment court.

```python
def TCAS():
    """Fonction principale du système TCAS."""
    intrus_max = 30 # nombre maximum d'avions suivis
    suivi_max = 100 # délai maximum pour le CPA (en secondes)
    CPAs = []  # liste des CPA des avions suivis
    while (TCAS_actif()):
        intrus = acquerir_intrus()
        enregistrer_CPA(intrus, CPAs, intrus_max, suivi_max)
        traiter_CPAs(CPAs)
```

Figure 5

La fonction `TCAS_actif` renvoie la position de l'interrupteur principal du système.

La fonction `acquerir_intrus` détermine la position et la vitesse d'un intrus particulier. À chaque appel, elle s'intéresse à un avion différent dans le périmètre de surveillance du TCAS. Lorsque tous les intrus ont été examinés, l'appel suivant revient au premier intrus qui est toujours dans le périmètre.

La fonction `enregistrer_CPA` intègre dans la liste CPA de nouvelles informations telles que fournies par la fonction `acquerir_intrus`.

Enfin la fonction `traiter_CPAs` examine les CPA des intrus les plus dangereux pour décider si l'un d'eux présente un risque de collision. Si c'est le cas, cette fonction détermine la manœuvre à effectuer (monter ou descendre) en coordination avec le système TCAS de l'intrus concerné et génère une alarme et une consigne à destination du pilote.

III.A – Acquisition et stockage des données

Chaque avion est équipé d'un émetteur radio spécialisé, appelé *transpondeur*, qui fournit automatiquement, en réponse à l'interrogation d'une station au sol ou d'un autre avion, des informations sur l'avion dans lequel il est installé. La fonction `acquerir_intrus` utilise les données du système de navigation de l'avion propre, les données fournies par le transpondeur de l'intrus, le relèvement de son émission et les informations fournies par le système de contrôle aérien au sol.

III.A.1) Les transpondeurs utilisent tous la même fréquence radio. Afin d'éviter la saturation de cette fréquence, en particulier dans les zones à fort trafic, chaque émission ne doit pas durer plus de 128 µs. Le débit binaire utilisé est de 10^6 bits par seconde ; chaque message commence par une marque de début de 6 bits et se termine par 4 bits de contrôle et une marque de fin de 6 bits.

$$\underbrace{d\ d\ d\ d\ d\ d}_{\text{début}}\ \underbrace{x\ x\ x \ldots x\ x\ x}_{\text{données}}\ \underbrace{e\ e\ e\ e}_{\text{contrôle}}\ \underbrace{f\ f\ f\ f\ f\ f}_{\text{fin}}$$

Déterminer le nombre maximum de bits de données dans une émission de transpondeur.

III.A.2) Le système TCAS souhaite récupérer l'altitude et la vitesse ascensionnelle de chaque intrus en interrogeant son transpondeur. La réponse du transpondeur contient systématiquement un numéro d'identification de l'avion sur 24 bits. Les autres informations sont des entiers codés en binaire qui peuvent varier dans les intervalles suivants :

- altitude de 2000 à 66 000 pieds ;
- vitesse ascensionnelle de −5000 à 5000 pieds par minute.

La taille d'un message de transpondeur est-elle suffisante pour obtenir ces informations en une seule fois ?

III.A.3) Après avoir récupéré les informations nécessaires, la fonction `acquerir_intrus` calcule la position et la vitesse de l'intrus par rapport à l'avion propre et renvoie une liste de huit nombres :

[`id, x, y, z, vx, vy, vz, t0`]

où

- `id` est le numéro d'identification de l'intrus ;
- `x, y, z` les coordonnées (en mètres) de l'intrus dans un repère orthonormé $\mathcal{R}_0$ lié à l'avion propre ;
- `vx, vy, vz` la vitesse (en mètres par seconde) de l'intrus dans ce même repère ;
- `t0` le moment de la mesure (en secondes depuis un instant de référence).

À des fins d'analyse une fois l'avion revenu au sol, la fonction `acquerir_intrus` conserve chaque résultat obtenu. Chaque nombre est stocké sur 4 octets. En supposant que cette fonction est appelé au maximum 100 fois par seconde, quel est le volume de mémoire nécessaire pour conserver les données de 100 heures de fonctionnement du TCAS ?

Ce volume de stockage représente-t-il une contrainte technique forte ?

III.B – Estimation du CPA

Le but de cette sous-partie III.B est d'estimer le CPA d'un avion intrus à partir des informations fournies par la fonction `acquerir_intrus`. Pour cela on suppose que l'avion propre et l'avion intrus poursuivent leur route sans changer de direction ni de vitesse. Vu les distances entre les deux avions, on néglige la courbure de la Terre, on considère donc qu'ils suivent un mouvement rectiligne uniforme.

III.B.1) On se place dans le repère orthonormé $\mathcal{R}_0$ utilisé par la fonction `acquerir_intrus`. On note O l'origine de ce repère (qui correspond à la position de l'avion propre), $G(t)$ la position de l'intrus à l'instant t et $\vec{V}$ sa vitesse dans $\mathcal{R}_0$ (supposée constante). La fonction `acquerir_intrus` fournit ainsi les coordonnées dans $\mathcal{R}_0$ des vecteurs $\overrightarrow{OG}(t_0)$ et $\vec{V}$ pour l'intrus `id`.

Exprimer le vecteur $\overrightarrow{OG}(t)$ en fonction des informations fournies par la fonction `acquerir_intrus`.

III.B.2) Déterminer l'expression du temps t_c qui correspond à l'instant où les deux avions sont le plus proches (instant du CPA).

III.B.3) Montrer qu'il n'y a pas de risque de collision si le produit scalaire $\overrightarrow{OG}(t_0) \cdot \vec{V}$ est positif.

III.B.4) Écrire en Python une fonction `calculer_CPA(intrus)` qui prend en paramètre une liste de la forme de celle produite par la fonction `acquerir_intrus` et qui renvoie :
- `None` s'il n'y pas de risque de collision ;
- ou une liste de 3 nombre `[tCPA, dCPA, zCPA]` où
  - `tCPA` est l'instant prévu pour le CPA (t_c) ;
  - `dCPA` la distance en mètres entre les deux avions au moment du CPA ;
  - `zCPA` la différence d'altitude en pieds entre les deux avions au moment du CPA (négative si l'intrus est plus bas que l'avion propre).

Pour répondre à cette question, on considère que l'avion propre vole horizontalement (vol de croisière) et que l'axe $\vec{z}$ du repère $\mathcal{R}_0$ est orienté verticalement vers le haut.

III.C – Mise à jour de la liste des CPA

Chaque élément de la liste `CPAs` (figure 5) est une liste de 4 nombres `[id, tCPA, dCPA, zCPA]` où `id` est l'identifiant de l'intrus et les autres éléments ont la signification précisée à la question III.B.4. Un exemple de liste `CPAs` est donné figure 6. Ainsi, `CPAs[2][3]` indique la séparation verticale (800 pieds) au CPA pour le troisième avion suivi, dont l'identifiant est 32675398 et dont le CPA aura lieu à l'instant 1462190455 (nombre de secondes depuis l'instant de référence).

```
CPAs = [ [ 11305708,  1462190400,  2450, -1000],
         [ 12416823,  1462190412,  2500,   500],
         [ 32675398,  1462190455,  2000,   800],
         [ 18743283,  1462190463,  2100,  -200] ]
```

Figure 6 Exemple de liste de CPA

La fonction `traiter_CPA` attend une liste triée par ordre chronologique. La fonction `enregistrer_CPA` s'assurera donc que la liste `CPAs` qu'elle produit est triée par ordre croissant des `tCPA` (`CPAs[i-1][1] <= CPAs[i][1]` pour tout `i` compris entre 1 et `len(CPAs)-1`). La fonction `traiter_CPAs` ne modifie pas l'ordre de la liste `CPAs`, `enregistrer_CPA` peut donc supposer que la liste qu'elle reçoit en paramètre est déjà triée.

III.C.1) Écrire en Python une fonction `mettre_a_jour_CPAs(CPAs, id, nv_CPA, intrus_max, suivi_max)` qui prend en paramètre
- `CPAs` : la liste actuelle des CPA au format décrit ci-dessus ;
- `id` : l'identifiant d'un intrus ;
- `nv_CPA` : les caractéristiques du CPA de l'intrus telles que renvoyées par la fonction `calcul_CPA` (peut éventuellement valoir `None`) ;
- `intrus_max` : un entier indiquant le nombre maximum d'intrus à suivre ;
- `suivi_max` : un entier donnant le délai maximum (en secondes) avant un CPA pour s'intéresser à l'intrus correspondant.

Cette fonction modifie la liste `CPAs` en appliquant les règles suivantes :
- si l'avion `id` est déjà suivi et ne présente plus de risque de collision, ou si son CPA est prévu dans plus de `suivi_max` secondes, il est supprimé de la liste ;
- si l'avion `id` est déjà suivi et que son CPA est prévu dans moins de `suivi_max` secondes, la ligne correspondante est mise à jour avec les nouvelles informations sur le CPA ;
- si l'avion `id` n'est pas suivi et que son CPA est prévu dans moins de `suivi_max` secondes alors,
  - s'il reste de la place dans la liste il est ajouté à la fin,
  - si la liste est pleine (elle contient déjà `intrus_max` intrus) et si le `tCPA` du nouvel intrus est inférieur au `tCPA` du dernier de la liste alors le nouvel intrus remplace le dernier intrus de la liste.

La fonction `mettre_a_jour_CPAs` renvoie :
- `None` si l'avion `id` a été supprimé ou n'a pas été ajouté ;
- ou un entier indiquant l'indice de la ligne de `CPAs` qui a été modifiée ou ajoutée.

III.C.2) Suite à un appel à la fonction `mettre_a_jour_CPAs`, il se peut que la liste `CPAs` ne soit plus triée par ordre croissant des `tCPA`.

Écrire en Python une fonction `replacer(ligne, CPAs)` qui modifie la liste `CPAs` de façon à ce qu'elle soit ordonnée par `tCPA` croissant, en sachant que la seule ligne qui n'est éventuellement pas à sa place est celle d'indice `ligne`.

Autrement dit, la fonction `replacer(ligne, CPAs)` doit remettre la ligne `CPAs[ligne]` à sa place dans l'ordre des `tCPA` croissants.

III.C.3) Écrire en Python la fonction `enregistrer_CPA` utilisée par la fonction `TCAS` (figure 5).

III.D – Évaluation des paramètres généraux du système TCAS

III.D.1) Le système TCAS est capable de détecter un intrus à une distance d'environ 60 km. Un avion de ligne vole typiquement à 900 km·h^{-1}. Quel temps va-t-il s'écouler entre la première détection d'un intrus et son CPA si celui-ci se situe à proximité de l'avion propre ?

Comparer ce résultat à la valeur standard du paramètre `suivi_max` (100 s) et commenter.

III.D.2) Afin de garantir le confort des passagers, les pilotes limitent généralement la vitesse ascensionnelle de leur avion à 1500 pieds par minute.

Si deux avions volent l'un vers l'autre à la même altitude, combien de temps avant leur CPA doivent-ils commencer à manœuvrer afin de se croiser avec une différence d'altitude d'au moins 500 pieds ?

III.D.3) Le système TCAS émet une alarme dans le cockpit demandant au pilote de monter ou descendre s'il détecte un intrus avec un `zCPA` inférieur à 500 pieds et un `tCPA` dans moins de 25 secondes. Commenter cette valeur.

La consigne de monter ou descendre est élaborée après concertation entre les systèmes TCAS des deux avions concernés afin d'éviter de conseiller la même manœuvre aux deux pilotes.

III.D.4) Les spécifications du système TCAS prévoient que la position de chaque intrus doit être vérifiée au moins une fois par seconde. Elles limitent d'autre part le nombre d'intrus suivis à 30. En déduire le temps maximum dont dispose la fonction `TCAS` pour exécuter une fois sa boucle.

III.D.5) Quel est le facteur limitant la vitesse d'exécution de la fonction `TCAS` ? En déduire un ordre de grandeur du temps minimum d'exécution d'une boucle. Est-il compatible avec les spécifications du système TCAS ?

Opérations et fonctions Python disponibles

Fonctions
- `exp(x)` calcule l'exponentielle du nombre x
- `randint(n)` (`n` entier) renvoie un entier aléatoire compris entre 0 et `n-1` inclus
- `random()` renvoie un nombre flottant tiré aléatoirement dans $[0, 1[$ suivant une distribution uniforme
- `sqrt(x)` calcule la racine carrée du nombre x
- `time()` renvoie l'heure sous la forme d'un nombre de secondes depuis un instant de référence

Opérations sur les listes
- `u + v` construit une liste constituée de la concaténation des listes `u` et `v` :
  `[1, 2] + [3, 4, 5]` → `[1, 2, 3, 4, 5]`
- `n * u` construit une liste constitué de la liste `u` concaténée `n` fois avec elle-même :
  `3 * [1, 2]` → `[1, 2, 1, 2, 1, 2]`
- `u.append(e)` ajoute l'élément `e` à la fin de la liste `u` (équivalent à `u = u + [e]`)
- `del(u[i])` supprime de la liste `u` son élément d'indice `i`
- `u.insert(i, e)` insère l'élément `e` à la position d'indice `i` dans la liste `u` (en décalant les éléments suivants) ; si `i >= len(u)`, `e` est ajouté en fin de liste
- `len(u)` donne le nombre d'éléments de la liste `u` :
  `len([1, 2, 3])` → 3, `len([[1,2], [3,4]])` → 2
- `u[i], u[j] = u[j], u[i]` permute les éléments d'indice `i` et `j` dans la liste `u`

• • • FIN • • •