# DOCUMENT RÉPONSE

TSIIN07

Pour l'ensemble du sujet, les bibliothèques usuelles sont chargées avec la syntaxe suivante :

**Librairies**

```
1  from numpy import min, max, zeros, roots, real
2  from scipy import linspace
3  from scipy.signal import lti, step
4  from matplotlib import pyplot as plt
5  from random import random
```

**Question 1**

```
1  numG = [ 8 ]
2
3  denG = [ 0.0072, 0.273, 1.13, 1]
4
5  G = lti(numG, denG)
```

**Question 2**

$C(p) = \dfrac{K_p + K_p T_i p + K_p T_d T_i p^2}{T_i . p}$

Donc $N_C(p) = K_p + K_p T_i p + K_p T_d T_i p^2$ et $D_C(p) = T_i . p$

```
1  def correcteur(Kp,Ti,Td) :
2          num = [Kp*Td*Ti, Kp*Ti, Kp]
3          ...
4          den = [Ti, 0]
5          ...
6
7
8
9
10
11
12
13         return num,den
14 numC,denC=correcteur(Kp,Ti,Td) # appel de la fonction, numC et denC les deux listes de coefficients.
```

## Question 3

```
1  def multi_listes(P,Q) :
2          deg_max=(len(P)-1)+(len(Q)-1)
3          P1=zeros(deg_max+1)
4          Q1=zeros(deg_max+1)
5          #P1 et Q1 deux listes de zéros de dimension deg_max+1
6          for j in range(len(P)) :
7                  P1[j]=P[j]
8          for j in range(len(Q)) :
9                  Q1[j]=Q[j]
10         R=zeros(deg_max+1)
11         for k in range(deg_max+1) :
12                 for i in range(k+1) :
13                         R[k]=R[k]+P1[i]*Q1[k-i]
14         return R
```

Q1= [r, s, 0, 0] avant la boucle "for k …"

(le sujet indique avant "la" boucle for, mais il y en a plusieurs...)

($k = 2$ et $i = 1$) : R= [a*r, a*s + b*r, a*0 + b*s]

(on suppose que c'est le contenu de la liste à la fin de l'itération k = 2 et i=1 qui est demandé)

R= [a*r, a*s + b*r, a*0 + b*s + c*r, a*0 + b*0 + c*s + 0*r] = [a*r, a*s + b*r, b*s + c*r, c*s]

Que fait multi_listes ? elle renvoie la liste des coefficients (rangés par puissance croissante) du produit de deux polynômes représentés par les listes P et Q de leurs coefficients (rangés par puissance croissante)

## Question 4

```
1  def inverse(liste) :
2          liste_r = zeros(len(liste))
3          ...
4          for k in range(len(liste)):
5          ...
6          ...        liste_r[k] = liste[len(liste)-k-1]
7          ...
8          ...
9          ...
10         ...
11         ...
12         ...
13         return liste_r
```

**Question 5**

```
def multi_FT(num1,den1,num2,den2) :
    ...
    prod_num = inverse(multi_listes(inverse(num1), inverse(num2)))
    ...
    prod_den = inverse(multi_listes(inverse(den1), inverse(den2)))
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    return prod_num, prod_den
```

**Question 6**

```
def somme_poly(P,Q) :   # les coefficients sont rangés par puissance croissante dans P et Q
    deg_max = max(len(P)-1, len(Q)-1)
    liste_P, liste_Q = inverse(P), inverse(Q)
    liste_P1, liste_Q1 = zeros(deg_max+1), zeros(deg_max+1)
    for k in range(len(P)) :
        liste_P1[k] = liste_P[k]
    for k in range(len(Q)) :
        liste_Q1[k] = liste_Q[k]
    ...
    liste_somme = zeros(deg_max+1)
    for k in range(deg_max+1) :
        liste_somme[k] = liste_P1[k] + liste_Q1[k]
    ...
    somme = inverse(liste_somme)
    ...
    return somme
```

**Question 7**

```
def stabilite(P) :
    racines = roots(P)
    ...
    parties_réelles = real(roots(P))
    ...
    for k in range(len(parties_réelles)):
        ...
        if parties_réelles[k] >= 0 :
            return False
        ...
        ...
        ...
        ...
    return True
```

## Question 8

```
def Temps_reponse(s,t) :
    T5=0
    s_fin=s[-1]
    for tt in range(len(s)) :
        j=len(t)-tt-1
        if ((s[j]>s_fin*0.95 and s[j-1]<s_fin*0.95)or(s[j]<s_fin*1.05 and s[j-1]>s_fin*1.05)) :
            T5=t[j]
            break
    return T5
```

La fonction examine chaque instant en partant du dernier : dès que la réponse sort de l'intervalle (s_fin*0,95 ; s_fin*1,05), l'instant correspondant est mémorisé (T5) et la boucle est interrompue (break). La valeur s[T5] à l'instant T5 se trouve encore dans l'intervalle de tolérance à 5%, donc T5 est supérieur au temps de réponse à 5%. La valeur approchée T5 majore le temps de réponse à 5%.

## Question 9

```
def Temps_reponse(s,t) :
    ...
    s_fin = s[-1]
    ...
    j = len(t)-1
    ...
    while s[j-1] > 0.95*s_fin and s[j-1] < 1.05*s_fin :
    ...
    ...         j = j-1
    ...
    T5 = t[j]  #valeur approchée majorant le temps de réponse
    ...
    ...
    return T5
```

## Question 10

```
def depassement(s,t) :
    ...
    D1 = (max(s) - s[-1]) / s[-1]
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    return D1
```

## Question 11

```
def critere_IAE(s,t) :
        e = 1
        IAE = 0
        ...
        for k in range(len(s)-1) :
        ...
        ...     epsilon = s[k] - e
        ...
        ...     IAE = IAE + abs(epsilon) *(t[k+1] - t[k])
        ...
        return IAE
```

Le calcul est effectué à l'aide de la méthode des rectangles (à droite). L'intégrale de |ε(t)| sur chaque intervalle (t[k] ; t[k+1]) est approchée par celle du rectangle de hauteur |ε(t[k])|.

## Question 12

```
T50=0.123
D10=0.05
IAE0=0.0733
def ponderation_cout(T5,D1,IAE) :
        k1=1
        k2=1
        k3=1
        cout=1/(k1+k2+k3)*(k1*T5/T50+k2*D1/D10+k3*IAE/IAE0)
        return cout
```

Pour les valeurs (T50, D10,IAE0), ponderation_cout(T5,D1,IAE) renvoie 1. La valeur retournée pour un triplet (T5, D1, IAE) quelconque doit donc être dans l'intervalle [0;1[ pour que la configuration soit considérée comme meilleure que la réponse de référence.

## Question 13

```
def calcul_coef_correcteur(Ip,Ii,Id) :
    ...
    Kp = (100 - 0.01)*Ip/(2**16 - 1) + 0.01
    ...
    Ti = (100 - 0.01)*Ii/(2**16 - 1) + 0.01
    ...
    Td = (50 - 0)*Id/(2**16 - 1) + 0
    ...
    ...
    ...
    ...
    ...
    ...
    return Kp,Ti,Td
```

On dénombre $2^{16}$ valeurs possibles pour chaque paramètre (Ip, Ii, Id. Comme les paramètres sont choisis de manière indépendante, il y a $2^{16} * 2^{16} * 2^{16} = 2^{48}$ combinaisons possibles pour le correcteur, soit $256*(1024)^4 \sim 3.10^{14}$ combinaisons.

## Question 14

```
def calcul_cout(Ip,Ii,Id) :
    Kp,Ti,Td=coef_correcteur(Ip,Ii,Id)
    numC,denC=correcteur(Kp,Ti,Td)
    num_BO, den_BO=multi_FT(numG,denG,numC,denC)
    num_BF,den_BF=FTBF(num_BO,den_BO)
    stable= stabilite(den_BO)
    ...
    if stable == True :
    ...
    ...    t,s = rep_Temp(Kp, Ti, Td)
    ...
    ...    T5, D1, IAE = Temps_reponse(s,t), depassement(s,t), critere_IAE(s,t)
    ...
    ...    cout = ponderation_cout(T5, D1, IAE)
    ...
    else :
    ...
    ...    cout = 100
    ...
    ...
    ...
    ...
    return cout
```

## Question 15

Pour tester chaque combinaison, il faudrait $3.10^{14} * 10,3.10^{-3} \sim 3.10^{12}$ s $\sim 10^9$ h. Cette durée est beaucoup trop importante pour envisager cette stratégie.

## Question 16

```
1  n=16
2  def genererGene(n) :
3          b2=''
4          for i in range(n) :
5                  b2=b2+str(int(random()*2))
6          return b2
```

Le résultat renvoyé est une chaine de caractères (str).

## Question 17

```
1  def generer_liste_initiale(n) :
2          CandidatS=[]
3          for i in range(0,100) :
4                  Candidat=[genererGene(n), genererGene(n), genererGene(n)]
5                  CandidatS.append(Candidat)
6          return CandidatS
```

## Question 18

Le nombre décimal associé vaut $2^0 + 2^3 + 2^6 = 1 + 8 + 64 = 73$

**Question 19**

```
1  def decodage(b2) :
2          b10=0
3          for p in range(0,len(b2)):
4
5              b10 = b2[p]*(2**p)
6
7
8
9
10         return b10
```

**Programme principal**

```
1  cycle=50
2  #-----------------Main-----------------
3  CandidatS=Generer_liste_initiale(n)
4  for i in range(cycle) :
5          CandidatS_top=tri(CandidatS)[0 :20]
6          CandidatS=nouvGeneration(CandidatS_top)
7          CandidatS=doublons(CandidatS)
8  solution=CandidatS[0]
```

**Question 20**

```
1  def perf(Li) : # Li est la liste des attributs d'un candidat
2          lp,li,ld=decodage(Li[0]),decodage(Li[1]),decodage(Li[2])
3          return calcul_cout(lp,li,ld)
4
5  def tri(L) : # L est la liste de tous les candidats
6          for i in range(1,len(L)) :
7                  if perf(L[i])<perf(L[i-1]) :
8                          p=i
9                          while p>0 and perf(L[i])<perf(L[p-1]) :
10                                 p=p-1
11                         X=L.pop(i)
12                         L.insert(p,X)
13         return L
```

Il s'agit d'un tri par insertion *(naïf, il pourrait être amélioré par une recherche dichotomique)*.

Dans le meilleur des cas (liste initiale rangée dans l'ordre croissant), n comparaisons (test "if") et n affectations (i in range ...) sont effectuées, la complexité est d'ordre n (boucle while jamais exécutée).

Dans le pire des cas (liste initiale rangée dans l'ordre décroissant), la boucle while est exécutée $n + (n-1) + (n-2) + \ldots + 1 = n(n+1)/2$ fois, avec une complexité d'ordre 1 (comparaison, pop, insert) pour chaque itération, soit une complexité d'ordre $n^2$. Les opérations hors de la boucle while présentent une complexité d'ordre n, la complexité de la fonction est donc d'ordre $n^2$ dans le pire des cas.

**Question 21**

```
def croisement(P1,P2) :
        E1=[]
        E2=[]
        for k in range(0,3):
        ... E1.append(P1[k][0:4]+P2[k][4:12]+P1[k][12:16])
            E2.append(P2[k][0:4]+P1[k][4:12]+P2[k][12:16])
```

```
    return E1, E2
```

**Question 22**

```
def mutation(E,i,j) :
        if E[i][j]=='1' :
                E[i]=E[i][:j]+'0'+E[i][j+1 :]
        else :
                E[i]=E[i][:j]+'1'+E[i][j+1 :]
        return E
```

La commande renvoie ['1111111111111111', '0000000000001000', '1111111100000000'] .....

Le 13ème bit du deuxième gène a été modifié. .....

.....

.....

.....

**Question 23**

```
def nouvGeneration(L) : # L la liste actuelle des candidats
        L_new=L
        for i in range(10) :
                E1,E2=croisement(L[0],L[int(random()*19)+1])
                L_new.append(mutation(E1,int(random()*3),int(random()*16)))
                L_new.append(mutation(E2,int(random()*3),int(random()*16)))
        for i in range(30) :
                c1,c2 = int(random()*19)+1,  int(random()*19)+1
                while c2 == c1 : #il ne faut pas croiser un candidat avec lui-même !
                ...     c2 = int(random()*19)+1
                E1, E2= croisement(c1,c2)
                L_new.append(mutation(E1,int(random()*3),int(random()*16)))
                L_new.append(mutation(E2,int(random()*3),int(random()*16)))


        return L_new
```

**Question 24**

```
1  def doublons (L) :
2          for i in range(len(L)):
3          ...     for j in range(i+1,len(L)):
4          ...             if L[i]==L[j]:
5          ...                     L[j]=[genererGene(n), genererGene(n), genererGene(n)]
6
7
8
9
10
11
12         ...
```

**Cadre réponse pour les questions 25 à 29**

Préciser le numéro de la question, séparer les réponses.

Q25. Cette requête renvoie la valeur minimale de l'attribut (colonne) score dans la table Historique.

Elle permet d'obtenir le score du meilleur candidat : 0,618248479198 (Id = 65).

Q26. SELECT disparition - apparition FROM Historique

WHERE score = (SELECT min(score) FROM Historique)

Le résultat est 28 (49-21) : cela signifie que les 28 dernières itération n'ont pas apporté d'amélioration dans l'exemple étudié. Il paraît raisonnable de réduire le nombre d'itérations (à 30 par exemple).

Q27. La requête renvoie les identifiants des meilleurs candidats de la 15e itération (ceux qui sont apparus avant et ont disparu après). On obtient les valeurs suivantes : 46, 50.

*Les données fournies semblent incohérentes : de nombreuses lignes présentent la même date d'apparition et de disparition, or la table n'est censée contenir que les candidats qui ont été parmi les 20 meilleurs : ils n'ont donc pas pu apparaître et disparaître lors de la même itération...*

Q28. SELECT AVG(gene_Kp), AVG(gene_Ti), AVG(gene_Td) FROM Historique

WHERE 20 > apparition AND 20 < disparition

Q29. Le résultat fourni par l'algorithme génétique est meilleur à tous points de vue :

- meilleur temps de réponse à 5%
- taux de dépassement inférieur
- IAE plus faible

Sans informations supplémentaires (temps d'exécution…), l'algorithme génétique ne présente que des avantages, il est opportun de l'utiliser.