Q1.

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

Q2. On réduit au même dénominateur $C(p) = K_p \frac{T_i p+1+T_d T_i p^2}{T_i p} = \frac{K_p+K_p T_i p+K_p T_d T_i p^2}{T_i p}$

```
def correcteur(Kp,Ti,Td):
    num=[Kp*Td*Ti,Kp*Ti,Kp]
    den=[Ti,0]
    return(num,den)
numC,denC=corrrecteur(Kp,Ti,Td)
```

Q3. Avant la première boucle for la liste Q1 vaut: Q1=[0,0,0,0]

Pour k=2 et i=1 on a : R=[a*r,a*s+b*r,b*s,0]

A la fin de la fonction : R=[a*r,a*s+b*r,c*r+b*s,c*s]

Cette fonction permet de calculer les coefficients du produit de 2 polynômes et les stocke dans la variable R. Si les 2 polynômes d'entrées P et Q sont donnés par ordre décroissants de puissances, la sortie est une liste des coefficients par ordre décroissants des puissances.

Q4. Fonction inverse :

```
def inverse(liste):
    taille=len(liste)
    liste_r=[]
    for k in range(taille):
        liste_r.append(liste[taille-1-k])
    return liste_r
```

Q5. Fonction multi_FT

```
def multi_FT(num1,den1,num2,den2):
    num=multi_listes(num1,num2)
    den=multi_listes(den1,den2)
    return(num,den)
```

Contrairement à ce qui est dit dans l'énoncé il est inutile d'utiliser la fonction inverse.

Q6. Somme de deux polynômes :

```
def somme_poly(P,Q):
    deg_somme=max([len(P)-1,len(Q)-1])
    somme=zeros(deg_somme+1)
    Pinv=inverse(P)
    Qinv=inverse(Q)
    for k in range(len(Pinv)):
        somme[k]=Pinv[k]
    for k in range(len(Qinv)):
        somme[k]+=Qinv[k]
    somme=inverse(somme)
    return somme
```

Q7. Stabilité

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

Q8. **s_fin** est la valeur finale de la réponse. L'indice **j** parcourt la liste **s** de la dernière valeur vers la première. On remonte donc le temps et on s'arrête (**break** stoppe la boucle inconditionnelle) dès que l'on sort de la bande +/- 5% :

- soit par le bas : s[j]>s_fin∗0.95 **and** s[j−1]<s_fin∗0.95
- soit par le haut : s[j]<s_fin∗1.05 **and** s[j−1]>s_fin∗1.05

Q9. Temps de réponse

```
def temps_reponse(s,t):
    T5=0
    s_fin=s[-1]
    j=len(s)-1
    while j>0 and (s[j]>s_fin*0.95 and s[j]<s_fin*1.05):
            T5=t[j]
            j=j-1
    return T5
```

Q10. Dépassement

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

Q11. "méthodes des rectangles à droite" (critere_IAE_rect) et "méthode des trapezes" (critere_IAE_trap)

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

```
def critere_IAE_trap(s,t):
    IAE =0
    for k in range(1,len(t)):
        IAE +=(abs(1-s[k])+abs(1-s[k-1]))*(t[k]-t[k-1])/2
    return IAE
```

Q12. La fonction qui pondère le poids de chaque critère renvoie 1 pour les valeurs T_{50}, D_{10} et IAE_0. Pour qu'une configuration soit meilleure que la solution de référence, il faut que la fonction renvoie une valeur inférieure à 1.

Q13. Coefficients du correcteur

```
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*Id/(2**16-1)
    return(Kp,Ti,Td)
```

Le nombre de combinaisons possibles pour le triplet est: $nb = 2^{16} \cdot 2^{16} \cdot 2^{16} = 2^{48}$

Q14. Calcul du cout d'une combinaison

```
def calcul_cout(Ip,Ii,Id):
    Kp,Ti,Td=calcul_coef_correcteur(Ip,Ii,Id)
    numC,denC=correcteur(Kp,Ti,Td)
    num_BO,den_BO=multi_FT(numG,denG,numC,denC)
    numBF,den_BF=FTBF(num_BO,den_BO)
    stable=stabilite(den_BF)
    if stable:
        temps_BF,sol_BF=rep_Temp(Kp,Ti,Td)
        Tr=temps_reponse(sol_BF,temps_BF)
        D=depassement(sol_BF,temps_BF)
        IAE=critere_IAE_trap(sol_BF,temps_BF)
        cout=ponderation_cout(Tr,D,IAE)
    else:
        cout=100
    return cout
```

Q15. Tester toutes les combinaisons possibles n'est pas envisageable.

Il faudrait $(2^{16})^3 \cdot 0{,}0103 \approx 2{,}89 \cdot 10^{12} s \approx 92 \cdot 10^3 ans$

Sans calculatrice : $2^{48} = 2^8 \cdot 2^{40} \approx 256 \cdot 10^{12}$

Q16. La fonction **genererGene(n)** permet de générer une possibilité de gène sous la forme d'une chaîne de n caractères aléatoires composée de 0 et de 1. On initialise donc n à 16.

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

Q17. Initialisation d'une population initiale

```
def generer_liste_initiale(n):
    CandidatS=[]
    for i in range(100):
        Candidat=[genererGene(n),genererGene(n),genererGene(n)]
        CandidatS.append(Candidat)
    return(CandidatS)
```

Q18. $(1001001000000000)_2 = (1 + 5 + 64)_{10} = (73)_{10}$ **ATTENTION** : nombre binaire inversé (bit de poids faible en 1er) sinon on obtiendrait $(37276)_{10}$

Q19. Passage binaire decimal

```
def decodage(b2):
    b10=0
    for k in range(len(b2)):
        b10+=int(b2[k])*2**k
    return(b10)
```

Q20. La fonction Tri utilise le tri par insertion dont la complexité est en O(n) au meilleur des cas et en O(n²) au pire des cas avec n=len(L).

Q21. Croisement

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

Q22. Pour le candidat ['1111111111111111','0000000000000000','1111111100000000'] la fonction mutation(Candidat,1,12) renvoie le candidat ['1111111111111111','0000000000001000','1111111100000000']

Q23. Nouvelle génération

```
def nouvGeneration(L):
    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):
        E1,E2=croisement(L[int(random()*19+1)],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)))
    return(L_new)
```

Q24. Gestion des doublons :

```
def doublons(L): # sur le même principe que le tri par insertion
    for i in range(1,len(L)): # on va comparer l'élément d'indice i avec les précédents
        k=i-1 # on commence par le précédent
            while k >= 0:
                if L[i] == L[k]: # on a trouvé un doublon
                    L[i] = [genererGene(n), genererGene(n), genererGene(n)] # tirage aléatoire candidat
                    k = i-1 # mais il faut recommencer nos comparaisons pour ne pas introduire un doublon
                else :
                    k = k-1 # pas de doublon, on remonte la liste pour comparer
    return(L)
```

Q25. Cette requête permet de déterminer la valeur minimale de score : 0.618248479198 (Id 65).

Q26. Longévité

```
SELECT min(score),disparition-apparition AS longevite
FROM Historique
```

Cette requête renvoie 49-21=28. On peut arrêter l'algorithme avant la 50^{e} itération car le second plus ancien a une longévité de 26.

Q27. Cette requête détermine les identifiants des candidats présents à l'itération 15. Elle renvoie 46 et 50 d'après l'extrait de la base de données de l'énoncé.

Q28. Valeur moyenne des gains du PID :

```
SELECT AVG(gene_Kp),AVG(gene_Ti),AVG(gene_Td)
FROM Historique
WHERE apparition<20 AND disparition>20
```

Q29. Dans les deux cas, l'erreur statique est nulle (ce qui est garanti pas la présence de l'action intégrale).
On constate que l'algorithme génétique permet d'obtenir un réglage du correcteur meilleur que la méthode de compensation de pôles. En effet le temps de réponse est plus faible (environ 0,1 s contre 0,14 s), le premier dépassement est plus faible (2% contre 5%). Il permet donc d'obtenir un système aussi précis mais plus stable et plus rapide.