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

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

Question 1

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

Question 2

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

Question 3

`Q1=[r,s,0,0]` avant la boucle `for k ...` car il y a plusieurs boucles for, le sujet n'est pas parfaitement clair.
à la fin de l'iteration `k=2` et `i=1` : `R=[a*r,a*s+b*r,b*s]`
à la fin `R=[a*r,a*s+b*r,b*s+c*r,c*s]`
La fonction `multi_listes(P,Q)` renvoie la liste des coefficients du polynôme produit des polynômes représentés par les listes P et Q.
Pour moi, si les coefficients des polynômes P et Q sont rangés par ordre des puissances décroissantes de x, les coefficients du produit seront rangés également par ordre de puissances décroissantes de x.
Si les coefficients des polynômes P et Q sont rangés par ordre des puissances croissantes, les coefficients du produit seront rangés également par ordre de puissances croissantes.

Question 4

```
def inverse(liste):
    n=len(liste)
    liste_r=zeros(n)
    for i in range(n):
        liste_r[i]=liste[n-1-i]
    return liste_r
```

Question 5

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

Selon moi il n'y a pas besoin de la fonction inverse ici.

Question 6

```
def somme_poly(P,Q):  #pour des coefficients par ordre decroissant des puissances de x
    P1=inverse(P)
    Q1=inverse(Q)
    len_res = max([len(P),len(Q)] )
    R1=zeros(len_res )
    for i in range(len(P)):
        R1[i]=R1[i]+P1[i]
    for j in range(len(Q)):
        R1[j]=R1[j]+Q1[j]
    somme=inverse(R1)
    return somme
```

Question 7

```
def stabilite(P):  #pour des coefficients par ordre decroissant des puissances de x
    parties_reelles_racines = real(roots(inverse(P)))
    for valeur in parties_reelles_racines:
        if valeur >=0:
            return False
    return True
```

Question 8

On part du dernier instant, la valeur est donc bien dans l'intervalle `[0.95s_fin; 1.05 s_fin]`. On examine chaque instant en partant du dernier : dés que la réponse sort de l'intervalle à l'instant précédent, on mémorise l'instant dans T5 et on interrompt la boucle. T5 est donc une valeur dans l'intervalle à 5%, la première telle que après on reste toujours dans cet intervalle. C'est donc une valeur approchée majorant le temps de réponse à 5% .

Question 9

```
def Temps_reponse(s,t):
    s_fin=s[-1]
    j=len(T)-1
    while 0.95*s_fin < s[j-1] < 1.05*s_fin:
        j=j-1
    T5=t[j]
    return T5
```

Question 10

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

Question 11

```
def critere_IAE(s,t):
    IAE=0
    for i in range(len(t)-1):
        IAE = IAE+abs(s[i]-1)*(t[i+1]-t[i])  #methode des rectangles
    return IAE
```

Question 12

`ponderation_cout(T5,D1,IAE)` renverra une valeur dans [0,1[si la configuration (T5,D1,IAE) est meilleure que celle de reference (T50,D10,IAE0)

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
```

cela fait $(2^{16}) * (2^{16}) * (2^{16}) = 2^{48} \sim 3.10^{14}$ possibilités.

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 :
        T5, D1, IAE = Temps_reponse(s,t), depassement(s,t), critere_IAE(s,t)
        t,s = rep_Temp(Kp, Ti, Td)
        cout = ponderation_cout(T5, D1, IAE)
    else :
        cout = 100
    return cout
```

Question 15

La durée serait de $3.10^{14} * 10.3 * 10^{-3} \sim 3.10^{12}$ secondes environ 10^9 heures, trop long.