Optimisation d'un correcteur PID par un algorithme génétique

MP - Mercredi 20 novembre 2019 - 1h30

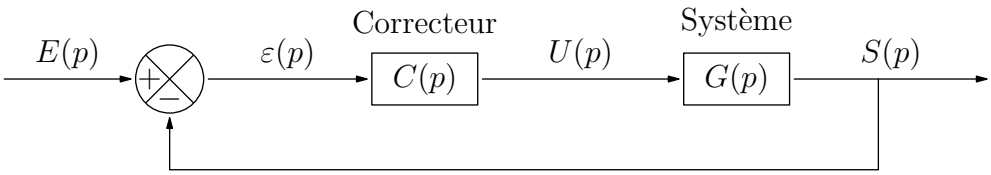
I

Position du problème

1. La correction d'un système asservi linéaire n'est jamais aisée et souvent réalisée au moyen de méthodes plus ou moins empiriques.
Un **correcteur Proportionnel-Intégral-Dérivateur (PID)** est fréquemment employé pour améliorer le comportement d'un système asservi. On se propose ici d'utiliser un algorithme génétique pour choisir les caractéristiques d'un tel correcteur.

I.1 Fonctions de transfert

2. On considère le système asservi à retour unitaire décrit par le schéma-bloc ci-dessous.



Les notations sont usuelles :
- La variable temporelle est notée t et la variable de Laplace, p.
- Les grandeurs sont notées en minuscules dans le domaine temporel (par exemple : $e(t)$, $s(t)$) et en majuscules dans le domaine de Laplace (par exemple : $E(p)$, $S(p)$). On rappelle que E est la transformée de Laplace de e au sens où
$$E(p) = \int_0^{+\infty} \exp(-pt) e(t)\, \mathrm{d}t.$$
- On note $C(p)$, la fonction de transfert du correcteur PID, qu'on suppose parfait.
- Le système linéaire à réguler est connu par sa fonction de transfert $G(p)$. Cette fonction de transfert est une fonction rationnelle :
$$G(p) = \frac{N_G(p)}{D_G(p)}$$
où les deux polynômes N_G et D_G sont premiers entre eux et vérifient $\deg N_G = k < \ell = \deg D_G$.

3. Pour déterminer la réponse temporelle du système corrigé, il faut définir la **fonction de transfert en boucle ouverte** $BO(p)$ et la **fonction de transfert en boucle fermée** $BF(p)$.
3.1 On *admettra* que, pour un système asservi à retour unitaire, on a

$$BO(p) = C(p) \cdot G(p) = \frac{N_O(p)}{D_O(p)} \tag{1}$$

ainsi que

$$BF(p) = \frac{BO(p)}{1 + BO(p)} = \frac{N_F(p)}{D_F(p)} = \frac{N_O(p)}{N_O(p) + D_O(p)} \tag{2}$$

où N_O et D_O sont deux polynômes.

3.2 On a alors

$$S(p) = BF(p) \cdot E(p)$$

et la réponse temporelle cherchée $s(t)$ s'obtient avec la transformée de Laplace inverse de $S(p)$.

3.3 Nous nous intéresserons ici à la **réponse indicielle**, c'est-à-dire à la réponse $s(t)$ à un échelon unitaire :

$$\forall\, t > 0, \quad e(t) = 1 \qquad\qquad \forall\, p > 0, \quad E(p) = \frac{1}{p}.$$

Pour un système asservi à retour unitaire, l'**erreur** $\varepsilon(t) = e(t) - s(t)$ est la différence entre la **consigne** $e(t)$ et la **réponse** $s(t)$, de telle sorte que le signal d'entrée $e(t)$ est aussi le signal attendu en sortie.

3.4 Outils informatiques

Pour réaliser cette étude, nous allons recourir au module `scipy.signal` qui permet, connaissant la fonction de transfert d'un système, de calculer et tracer l'allure de la réponse indicielle.

On trouvera plus loin [32] un exemple d'utilisation et des extraits de la documentation relative à ce module.

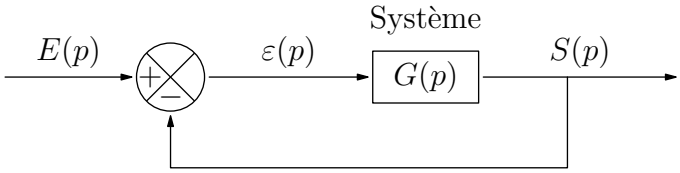
Dans tout le sujet, on se conformera à la documentation du module `scipy.signal` pour représenter les polynômes.

I.2 Système à réguler

4. Le système que nous allons étudier est décrit par la fonction de transfert suivante.

(3)
$$G(p) = \frac{8}{9 + 1,13p + 0,273p^2 + 0,0072p^3}$$

Afin d'améliorer la réponse temporelle de ce système, on réalise un asservissement, décrit par le schéma-bloc ci-dessous.



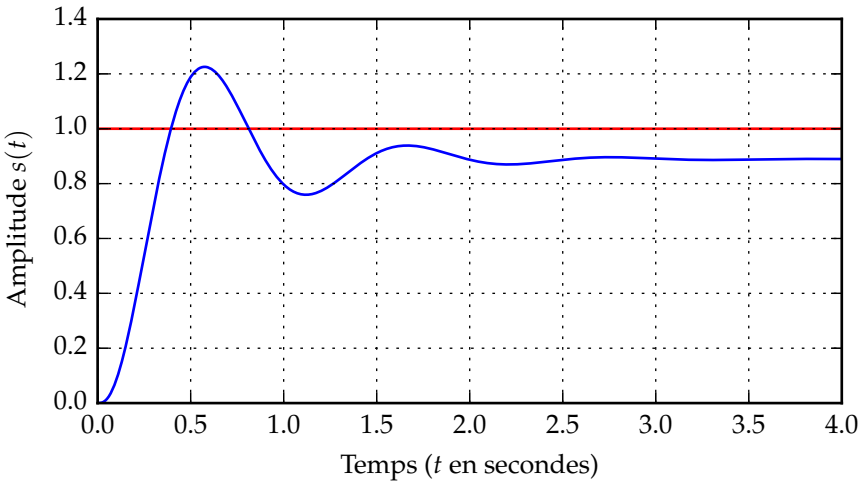
Q 1. *On souhaite tracer l'allure de la réponse temporelle $s(t)$ de ce système à un échelon unitaire $e(t)$ au moyen du module* `scipy.signal`.

```
import numpy as np
from scipy.signal import lti, step

Tmax = 4
T = np.linspace(0, Tmax, 1000)
numG =            # à compléter
denG =            # à compléter
G = lti(numG, denG)
t, s = step(G, T)
```

À l'aide de la documentation sur `scipy.signal` *et de l'expression* (3)*, donner les valeurs des variables* `numG` *et* `denG`.

5. La réponse temporelle est représentée sur la figure ci-dessous.



Réponse indicielle sans correction

On constate sur cette figure que la réponse temporelle du système asservi présente plusieurs défauts :
– Défaut de précision : la valeur de consigne n'est pas atteinte ;
– Temps de réponse trop long : la réponse met près de 2 s pour approcher de la valeur finale ;
– Dépassement important : le pic d'amplitude de la réponse dépasse la valeur de consigne d'environ 20% ;
– Oscillations sensibles.
Il paraît donc nécessaire de choisir un correcteur pour améliorer la réponse de ce système.

II

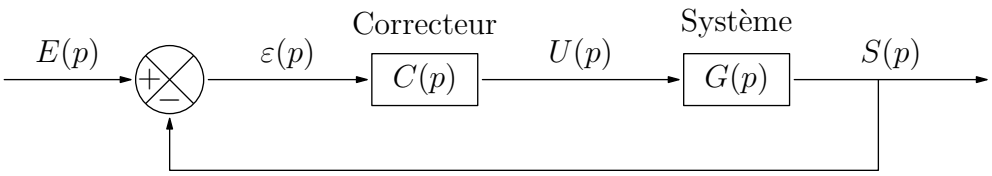
Correction du système

II.1 Correcteur PID

6. On installe dans le système un correcteur PID, de fonction de transfert :

$$C(p) = K_p \cdot \left(1 + \frac{1}{T_i \cdot p} + T_d \cdot p\right).$$

Le système est décrit par le schéma-bloc suivant.



Q 2.a *Définir deux polynômes $N_C(p)$ et $D_C(p)$ tels que la fonction de transfert du correcteur puisse s'écrire de la manière suivante.*

$$C(p) = \frac{N_C(p)}{D_C(p)}.$$

Ces polynômes seront représentés par les listes `numC` *et* `denC` *dans la suite.*

Q 2.b *Écrire une fonction* `correcteur(Kp, Ti, Td)` *qui retourne un couple* `(numC, denC)` *associé aux caractéristiques K_p, T_i et T_d du correcteur PID.*

II.2 Calcul des fonctions de transfert

7. Pour calculer la fonction de transfert en boucle ouverte $BO(p)$ et la fonction de transfert en boucle fermée $BF(p)$ à l'aide des relations (1) et (2), il faut pouvoir calculer le produit et la somme de deux polynômes.

Q 3. Produit de deux polynômes

On considère la fonction `multi_listes(P, Q)` *dont le code est le suivant.*

```
def multi_listes(P, Q):
  n = len(P)+len(Q)-1
  P1, Q1 = np.zeros(n), np.zeros(n)
  P1[:len(P)] = P
  Q1[:len(Q)] = Q
  R = np.zeros(n)
  for k in range(n):
    for i in range(k+1):
      R[k] += P1[i]*Q1[k-i]
  return R
```

Q 3.a *On exécute la fonction* `multi_listes(P, Q)` *avec* $P = [1, 1, 1]$ *et* $Q = [1, -1]$.

1. *Identifier les polynômes P et Q.*
2. *Donner les valeurs de n et des listes $P1$ et $Q1$.*

Q 3.b *On revient au cas général.*

3. *Expliquer pourquoi l'exécution de la fonction* `multi_listes` *termine.*
4. *Démontrer que la liste retournée représente bien le produit des polynômes représentés par les listes P et Q.*

Q 4. *On suppose connues deux fonctions de transfert*

$$F_1(p) = \frac{N_1(p)}{D_1(p)} \quad et \quad F_2(p) = \frac{N_2(p)}{D_2(p)}$$

et on cherche à calculer le produit $F(p) = F_1(p) \cdot F_2(p)$.
Écrire une fonction `multi_FT(num1, den1, num2, den2)` *dont les arguments sont respectivement les listes qui représentent les polynômes $N_1(p)$, $D_1(p)$, $N_2(p)$ et $D_2(p)$ et qui retourne un couple* `(num, den)` *qui représente la fonction de transfert $F(p)$.*

Q 5. Somme de deux polynômes

Écrire une fonction `somme_poly(P, Q)` *dont les arguments sont deux listes représentant des polynômes et qui retourne la liste qui représente la somme de ces deux polynômes.*
NB : on ne supposera pas *que les deux listes* `P` *et* `Q` *ont même longueur.* On pourra *utiliser la fonction* `max` *de Python (qui retourne le plus grand élément d'une liste).*

II.3 Tracé de la réponse indicielle

8. Pour tracer la réponse indicielle du système en fonction du temps, il reste à calculer la fonction de transfert en boucle fermée et à déduire la réponse indicielle de cette fonction de transfert.

8.1 La fonction de transfert en boucle fermée se déduit facilement de la fonction de transfert en boucle ouverte [**3.1**].

```
def FTBF(num, den):
  num_bf = num
  den_bf = somme_poly(num, den)
  return (num_bf, den_bf)
```

8.2 On peut ensuite utiliser le module `scipy.signal` pour en déduire la réponse indicielle.

```
def rep_Temp(Kp, Ti, Td, T_max=1.5):
  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)
  BF = lti(num_BF, den_BF)
  temps, signal = step(BF, T=linspace(0, T_max, 200*T_max))
  return temps, signal
```

On admet ici que, pour la plupart des corrections apportées, la valeur de consigne (arbitrairement choisie égale à 1) est atteinte en moins de $1,5$ seconde.

8.3 Les deux listes produites par la fonction `rep_Temp` seront utilisées dans tout le reste du sujet. Il importe donc de bien comprendre que la liste `temps` contient les instants d'échantillonnage :

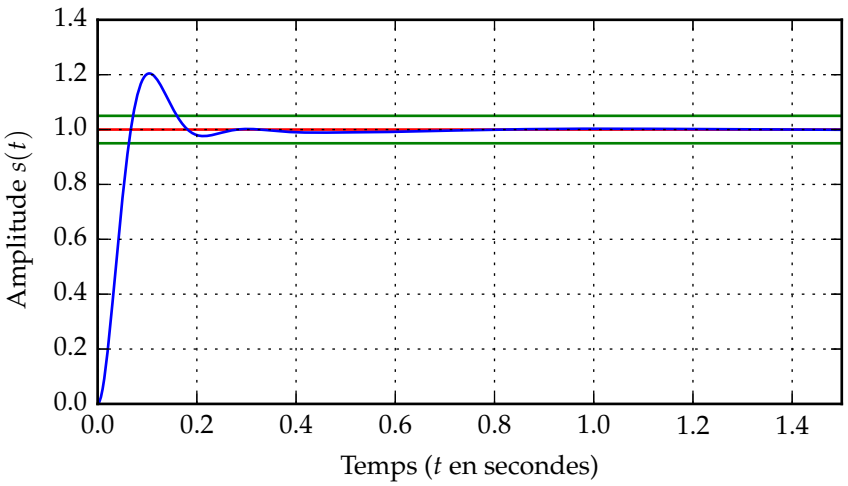
$$\texttt{temps} = [t_0 = 0,\ t_1,\ t_2,\ \ldots,\ t_{N-1} = \texttt{T\_max}]$$

et que la liste `signal` contient un échantillon de valeurs du signal de sortie $s(t)$:

$$\texttt{signal} = [s(t_0),\ s(t_1),\ s(t_2),\ \ldots,\ s(t_{N-1})].$$

En particulier, ces deux listes ont même longueur.

9. Avec $K_p = 5$, $T_i = 0,12\,\text{s}$ et $T_p = 0,218\,\text{s}$, on obtient la réponse indicielle suivante.



Réponse indicielle avec correction

III

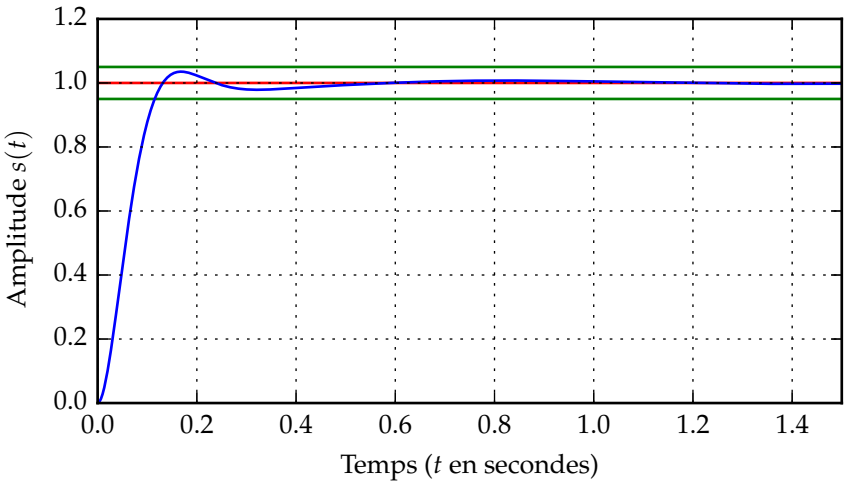
Critères d'optimisation

III.1 Réglage pratique

10. Une méthode pratique de réglage (par exemple, la méthode de **compensation des pôles**) a permis de déterminer les valeurs suivantes pour les coefficients du correcteur PID.

$$K_{p0} = 1,9 \qquad T_{i0} = 0,1\,\text{s} \qquad T_{d0} = 0,3\,\text{s}$$

La réponse indicielle, tracée ci-dessous, est sensiblement améliorée.



Première amélioration de la correction

11. L'étude suivante a pour objectif d'améliorer encore le réglage du correcteur PID.

III.2 Stabilité

12. On s'intéresse ici à la **stabilité** du système.

12.1 On rappelle que, pour les systèmes étudiés ici, une fonction de transfert est une fonction rationnelle et on *admet* que, dans nos calculs, toutes les fonctions rationnelles sont représentées sous forme irréductible. Dans ces conditions, les **pôles** d'une fonction de transfert sont les racines du dénominateur.

12.2 On rappelle aussi que le système étudié est **stable** si, et seulement si, la partie réelle de chaque pôle est strictement négative.

13. La stabilité du système présente un double intérêt.

13.1 Tout d'abord, si le système est stable, la réponse temporelle est convergente, ce qui permet de définir la **valeur finale** (= valeur du signal en régime permanent).

13.2 D'autre part, la présence d'une intégration dans le correcteur PID nous garantit que, si le système est stable, la valeur finale sera bien égale à la valeur de consigne. On dit dans ce cas que l'**erreur indicielle** est nulle.

Q 6. *On note $D(p)$, le dénominateur d'une fonction de transfert $F(p)$, représenté par une liste* `D`*. Écrire une fonction* `stabilite(D)` *à valeurs booléennes, qui retourne* `True` *si, et seulement si, la fonction de transfert $F(p)$ est stable. On utilisera les fonctions* `roots` *et* `real` *du module* `numpy`*. (La documentation est donnée en annexe* **[34]**.*)*

III.3 Critères usuels d'optimisation

14. On l'a rappelé [**13.1**] : lorsque le système est stable, la réponse indicielle converge vers une valeur limite appelée *valeur finale* et le correcteur PID nous assure alors que l'erreur indicielle [**13.2**] est nulle.

15. On décrit alors la qualité de la réponse indicielle avec les notions suivantes.

15.1 Le **temps de réponse** est la durée au bout de laquelle l'amplitude de la réponse reste égale à la valeur finale à $\pm 5\%$ près.

$$\forall\, t \geqslant T_5, \qquad \frac{|s(t) - s_\infty|}{s_\infty} \leqslant 5\%$$

15.2 Le **dépassement relatif** est défini par

$$D_1 = \frac{s_{\max} - s_\infty}{s_\infty}$$

où $s_{\max}$ est l'amplitude maximale de la réponse et s_∞ est la valeur finale.

15.3 Typiquement, on cherche un temps de réponse aussi bref que possible et un dépassement relatif inférieur à 5%. Dans certains cas (machines d'usinage), tout dépassement est proscrit.

Q 7. *Comparer les réponses indicielles tracées au* [**9**] *et au* [**10**] *en termes d'erreur indicielle, de temps de réponse et de dépassement relatif.*

Q 8.a *Les arguments de la fonction* `temps_reponse` *sont la liste* `s` *des amplitudes de la réponse indicielle et la liste* `t` *des instants d'échantillonnage. (En pratique, ces listes sont les valeurs retournées par la fonction* `rep_Temp` [**8.3**]*.)*

```
def temps_reponse(s, t):
  T = 0
  s_fin = s[-1]
  for tt in range(len(s)):
    j = len(t)-tt-1
    dep_i = (s[j]>s_fin*0.95 and s[j-1]<s_fin*0.95)
    dep_s = (s[j]<s_fin*1.05 and s[j-1]>s_fin*1.05)
    if (dep_i or dep_s):
      T = t[j]
      break   # sortie (anticipée) de la boucle for
  return T
```

Démontrer que la valeur retournée par la fonction `temps_reponse` *majore le temps de réponse défini ci-dessus. (On donnera en particulier le sens des variables* `s_fin`, `dep_i` *et* `dep_s`*.)*

Q 8.b *Proposer une réécriture simplifiée de la fonction* `temps_reponse(s, t)` *à l'aide d'une boucle* `while`*.*

Q 9. *Proposer un code pour la fonction* `depassement(s, t)` *qui retourne le dépassement relatif* D_1*. (On pourra utiliser la fonction* `np.max` *qui calcule le plus grand élément d'une liste.)*

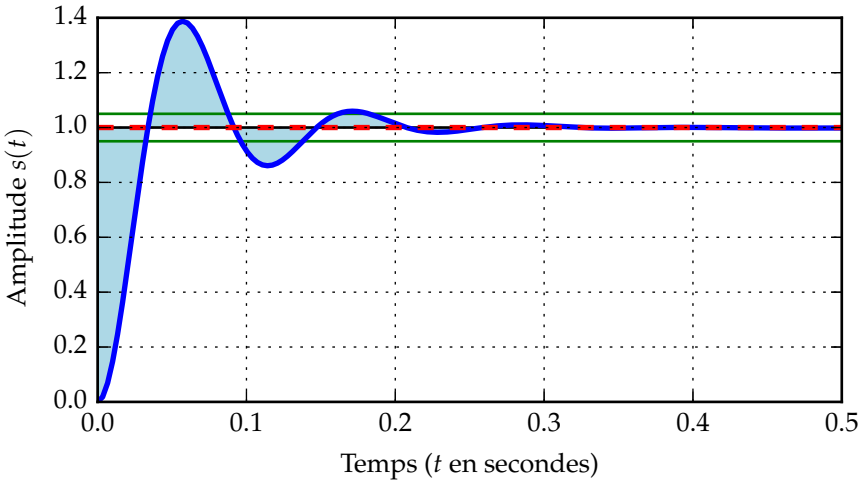
III.4 Critère intégral

16. En notant $e(t) = 1$, la consigne d'entrée et $s(t)$, la réponse du système, on définit l'erreur

$$\varepsilon(t) = e(t) - s(t).$$

Aux critères précédents, on peut ajouter un nouveau critère de qualité qui va estimer l'amplitude des oscillations lors du régime transitoire : l'**erreur intégrale**

$$EI = \int_0^{+\infty} |\varepsilon(t)| \, \mathrm{d}t.$$



Q 10. *Proposer une fonction* `critere_EI(s, t)` *qui prend pour arguments l'échantillon* `s` *des valeurs de la réponse* $s(t)$ *aux instants d'échantillonnage regroupés dans la liste* `t` *et qui retourne une valeur approchée de* EI.
Comme pour la question **[Q8.a]**, *on pourra supposer que ces deux listes ont été retournées par la fonction* `rep_Temp`.

III.5 Fonction de coût

17. Valeurs de référence
Pour les paramètres K_p, T_i et T_d choisis au **[10]**, on obtient des valeurs de référence pour le temps de réponse, le dépassement relatif et l'erreur intégrale :

$$T_0 = 0,123\,\mathrm{s} \qquad D_0 = 3,8\,\% = 0,038 \qquad EI_0 = 0,068\,\mathrm{USI}.$$

18. Fonction de coût
Nous cherchons des valeurs de K_p, T_i et T_d qui permettent d'améliorer le système pour ces trois critères. À cet effet, nous définissons une fonction `ponderation_cout(T5, D1, EI)` qui retourne le coût global associé à ces trois critères.

```
T0, D0, EI0 = 0.123, 0.038, 0.068  # Constantes de référence

def ponderation_cout(T5, D1, EI):
  k1, k2, k3 = 1, 1, 1
  cout = (k1*T5/T0 + k2*D1/D0 + k3*EI/EI0)/(k1+k2+k3)
  return cout
```

On considérera que le réglage du correcteur PID est optimal lorsque cette fonction de coût global est minimale.

Q 11.a *Pourquoi est-il nécessaire d'introduire une fonction de coût ?*
Q 11.b *Expliquer les calculs effectués par la fonction* `ponderation_cout`.

Q 11.c *Que signifierait un autre choix des constantes* `k1`, `k2`, `k3` *?*

19. Plus précisément, nous cherchons les valeurs de K_p, T_i et T_d dans les intervalles suivants.

$$K_p \in [0,01\,;100] \qquad T_i \in [0,01\,\text{s}\,;100\,\text{s}] \qquad T_d \in [0\,\text{s}\,;50\,\text{s}]$$

Nous considérerons que chacune de ces trois variables ne peut prendre que 2^{16} valeurs réparties uniformément dans chacun de ces intervalles. Il existe donc trois entiers I_p, I_i et I_d tels que

$$K_p = K_{p,\min} + \frac{(K_{p,\max} - K_{p,\min}) \cdot I_p}{2^{16}-1} \quad \text{et} \quad 0 \leqslant I_p < 2^{16} \tag{4}$$

$$T_i = T_{i,\min} + \frac{(T_{i,\max} - T_{i,\min}) \cdot I_i}{2^{16}-1} \quad \text{et} \quad 0 \leqslant I_i < 2^{16} \tag{5}$$

$$T_d = T_{d,\min} + \frac{(T_{d,\max} - T_{d,\min}) \cdot I_d}{2^{16}-1} \quad \text{et} \quad 0 \leqslant I_d < 2^{16} \tag{6}$$

Q 12.a *Expliquer pourquoi la démarche précédente est nécessaire.*

Q 12.b *Cette démarche a-t-elle des conséquences négatives sur la précision des résultats obtenus ?*

Q 12.c *Combien y a-t-il de combinaisons possibles pour régler le correcteur PID ?*

Q 13. *Proposer une fonction* `calcul_coeff_correcteur(Ip, Ii, Id)` *qui retourne les valeurs des coefficients K_p, T_i et T_d calculées en fonction des trois entiers I_p, I_i et I_d selon les formules* (4), (5) *et* (6).

20. Il s'agit maintenant de calculer le coût de chaque combinaison (I_p, I_i, I_d) possible. Il faut pour cela

- déterminer les coefficients K_p, T_i et T_d du correcteur PID ;
- déterminer la fonction de transfert en boucle ouverte (numérateur et dénominateur) ;
- déterminer la fonction de transfert en boucle fermée (numérateur et dénominateur) ;
- si la fonction de transfert en boucle fermée est stable, il faut calculer le temps de réponse T_5, le dépassement D_1 et l'erreur intégrale EI avant de retourner la valeur de `ponderation_cout(T5, D1, EI)` ;
- si la fonction de transfert en boucle fermée n'est pas stable, on renvoie un coût infini.

Q 14. *On veut disposer d'une fonction* `calcul_cout(Ip, Ii, Id)` *qui mette en œuvre l'algorithme décrit au* **[20]**. *Compléter le code suivant à l'aide des fonctions définies précédemment.*

```
def calcul_cout(Ip, Ii, Id):
  # Fonction de transfert C(p) du correcteur PID
  Kp, Ti, Td = calcul_coeff_correcteur(Ip, Ii, Id)
  numC, denC = correcteur(Kp, Ti, Td)
  # Fonction de transfert en boucle ouverte
  num_BO, den_BO = multi_FT(numG, denG, numC, denC)
  # Fonction de transfert en boucle fermée
  num_BF, den_BF = FTBF(num_BO, den_BO)
  # début de la partie à compléter
  stable =
  if stable:

  else:
    cout = np.inf    # coût infini
  # fin de la partie à compléter
  return cout
```

NB : Les variables `numG` *et* `denG` *sont des variables globales, qui ont été définies à la question* **[Q1.]**.

Q 15. *Chaque exécution de la fonction* `calcul_cout` *dure environ* 10 ms. *Est-il envisageable de calculer le minimum de cette fonction en testant toutes les combinaisons possibles pour I_p, I_i et I_d ?*

IV

Résolution par un algorithme génétique

IV.1 Principe

21. Les algorithmes génétiques sont des processus d'optimisation itératifs. Ils permettent d'approcher la solution d'un problème d'optimisation (pour nous, la recherche d'un minimum pour la fonction `calcul_cout` définie en [**20**]) en mimant les mécanismes de l'évolution conduite par le génie génétique.

22. La structure générale du programme que nous allons décrire est donnée en annexe [**35**].

22.1 La phase initiale consiste à choisir une population quelconque de n_{pop} individus. Ici, $n_{\text{pop}} = 100$ et chaque individu sera identifié à son **génotype**, c'est-à-dire à un triplet d'entiers (I_p, I_i, I_d).

22.2 Pour passer de la k-ième à la $(k+1)$-ième génération d'individus, on évalue chacun des n_{pop} génotypes de la k-ième génération et on ne retient que les 20 meilleurs, c'est-à-dire ceux dont le coût est le moins élevé. On définit ensuite 80 nouveaux individus par croisements et mutations des 20 meilleurs génotypes.
De la sorte, génération après génération, le nombre d'individus reste constant et les caractéristiques de la population s'améliorent (au sens où les valeurs de la fonction `calcul_cout` tendent à diminuer).

22.3 On répète le cycle sélection-croisement-mutation un certain nombre de fois. De manière un peu arbitraire, nous effectuerons 50 cycles — sans chercher un meilleur critère d'arrêt.

22.4 Le meilleur génotype de la dernière génération est alors utilisé pour fixer les valeurs de I_p, I_i et I_d.

IV.2 Population initiale

23. Chaque individu possède trois gènes : les paramètres entiers I_p, I_i et I_d qui peuvent prendre 2^{16} valeurs distinctes [**19**].
Pour faciliter le génie génétique (croisements et mutations), on représente ces trois gènes par des listes de 16 booléens.
Chaque individu est donc représenté par une liste de 3 listes de 16 booléens.

Q 16. *On considère les fonctions suivantes. On rappelle* [**34**] *que la commande* `np.randint(0,1)` *permet de simuler des variables aléatoires indépendantes qui suivent la loi de Bernoulli de paramètre* $1/2$.

```
def rand_b():
  if np.randint(0,1):
    return True
  else:
    return False

def genererGene(lgr):
  gène = [ rand_b() for i in range(lgr) ]
  return gène
```

Expliquer brièvement le fonctionnement de ces deux fonctions. Quelle valeur de `lgr` *doit-on utiliser ?*

Q 17. *Compléter le code suivant pour que l'exécution de la fonction crée une liste initiale de* n_{pop} *individus.*

```
def generer_liste_initiale(n_pop):
  individus = []
  for i in          # à compléter
    c =             # à compléter
    individus.append(c)
  return individus
```

24. Les arguments I_p, I_i et I_d de la fonction `calcul_cout` sont des entiers **[Q14.]**. La fonction `decodage(g)` a pour but de convertir le gène `g` (une liste de 16 booléens, c'est-à-dire de 16 bits) en un nombre entier `n_g` compris entre 0 et $2^{16} - 1$.

Q 18. *On convient que le gène* `g` *représente le codage binaire de l'entier* `n_g`*, avec la convention que le booléen* `g[0]` *donne le bit de poids le plus faible (et donc que le booléen* `g[15]` *donne le bit de poids le plus fort).*
Proposer un code pour la fonction `decodage(g)`.

IV.3 Sélection des individus

25. Pour chaque génération, il faut **[22.2]** retenir les 20 meilleurs génotypes pour créer la génération suivante. Parmi ces génotypes, les 5 meilleurs seront enregistrés dans une base de données. →**[30]**

Q 19. *On propose le code suivant.*

```
def perf(Li):
  Ip, Ii, Id = decodage(Li[0]), decodage(Li[1]), decodage(Li[2])
  return calcul_cout(Ip, Ii, Id)

def insertion(c, p, Top, Perf):
  n, i = len(Perf), 0
  while (i<n) and (p>Perf[i]):
    i += 1
  Top = Top[:i]+[c]+Top[i:-1]
  Perf = Perf[:i]+[p]+Perf[i:-1]
  return Top, Perf

def selection(L, n):    # On admet que n<len(L)
  T, P = [[] for i in range(n)], [np.inf for i in range(n)] # np.inf = +∞
  for individu in L[:n]:
    perf_i = perf(individu)
    T, P = insertion(individu, perf_i, T, P)
  for individu in  L[n:]:
    perf_i = perf(individu)
    if perf_i<P[-1]:
      T, P = insertion(individu, perf_i, T, P)
  return T, P
```

Q 19.a *On* admet *que les arguments* `Top` *et* `Perf` *de la fonction* `insertion` *sont deux listes de même longueur et que la liste* `Perf` *est une liste croissante de réels.*
On admet *également que la valeur de l'argument* `p` *est strictement inférieure à la valeur de* `Perf[-1]`.
Expliquer le fonctionnement de `insertion`.

Q 19.b *Expliquer ce que réalise la fonction* `selection`. *Préciser en particulier la nature et le contenu des variables* `T` *et* `P` *retournées par cette fonction.*

Q 19.c *Dans quelle mesure ce code fournit-il le résultat attendu de manière efficace ?*

IV.4 Génie génétique

26. Croisements
Avec les gènes de deux individus `P1` et `P2`, la fonction `croisement(P1, P2)` crée les gènes de deux descendants `E1` et `E2`.

26.1 Dans un premier temps, les trois gènes de `P1` et `P2` sont découpés en trois morceaux.

$$g = [g_0, \ldots, g_{15}] \longmapsto [g_0, \ldots, g_3]/[g_4, \ldots, g_{11}]/[g_{12}, \ldots, g_{15}]$$
$$h = [h_0, \ldots, h_{15}] \longmapsto [h_0, \ldots, h_3]/[h_4, \ldots, h_{11}]/[h_{12}, \ldots, h_{15}]$$

26.2 Dans un second temps a lieu le croisement proprement dit : les parties centrales sont échangées.

$$\begin{array}{ccc} [g_0, \ldots, g_3] & [h_4, \ldots, h_{11}] & [g_{12}, \ldots, g_{15}] \\ [h_0, \ldots, h_3] & [g_4, \ldots, g_{11}] & [h_{12}, \ldots, h_{15}] \end{array}$$

26.3 Dans un troisième temps, on recombine les morceaux pour créer les nouveaux gènes.

$$g' = [g_0, \ldots, g_3, h_4, \ldots, h_{11}, g_{12}, \ldots, g_{15}]$$
$$h' = [h_0, \ldots, h_3, g_4, \ldots, g_{11}, h_{12}, \ldots, h_{15}]$$

Q 20. *Compléter le code suivant pour qu'il effectue les opérations décrites ci-dessus.*

```
def DCR(g, h, i, j):
  return g[:i]+h[i:j]+g[j:]

def croisement(P1, P2):
  E1 = [DCR(          ) for (g,h) in zip(P1, P2)] # à compléter
  E2 = [DCR(          ) for (g,h) in zip(P1, P2)] # à compléter
  return E1, E2
```

27. Mutations

Afin de limiter le risque d'apparition de clones dans la $(k+1)$-ième génération, chaque génotype obtenu par croisement de deux génotypes de la k-ième génération subit une mutation sur l'un de ses gènes.

Q 21. *Cette mutation est codée par la fonction suivante.*

```
def mutation(c):
  i = np.randint(0, len(c)-1)
  j = np.randint(0, len(c[i])-1)
  c[i][j] = not(c[i][j])
  return None
```

Expliquer le fonctionnement de cette fonction.

28. Nouvelle génération

On a choisi de conserver les 20 meilleurs génotypes de la k-ième génération pour constituer la $(k+1)$-ième génération. Ces génotypes sont regroupés dans une liste `L`.

$$L = [c_0, \ldots, c_{19}]$$

Les 80 autres génotypes de la $(k+1)$-ième génération sont créés en deux temps.

28.1 Vingt génotypes de la $(k+1)$-ième génération sont issus de croisements du meilleur génotype de la k-ième génération (c'est-à-dire c_0) avec 10 génotypes choisis aléatoirement parmi $c_1, \ldots, c_{19}$ (première boucle `for` de la fonction `nouvelle_generation(L)` ci-dessous).

28.2 Soixante autres génotypes de la $(k+1)$-ième génération sont issus par croisements de 30 couples formés à partir des génotypes c_1, ..., c_{19}.

Q 22. *Compléter le code ci-dessous en respectant le cahier des charges donné en* [**28.2**].

```
def croisement_mutation(P1, P2, L):
  E1, E2 = croisement(P1, P2)
  mutation(E1)
  mutation(E2)
  L.append(E1)
  L.append(E2)
  return None

def nouvelle_generation(L):
  premier = L[0]
  suivants = np.sample(L[1:], 10)
  for c in suivants:
    croisement_mutation(premier, c, L)
  # boucle à compléter
  for i in

  # fin de la boucle à compléter
  return L
```

29. Doublons

Au fil des générations, les patrimoines génétiques des génotypes tendent à se ressembler. Il se peut que les génotypes de la $(k+1)$-ième génération ne soient pas tous différents.

Q 23. *On note* `L`*, une liste de n génotypes. L'exécution de la commande* `doublons(L)` *doit repérer si un génotype apparaît plus d'une fois et remplacer chaque clone par un nouveau génotype choisi au hasard parmi l'ensemble de tous les génotypes possibles.*

Q 23.a *Proposer un code pour la fonction* `doublons`.

Q 23.b *Dans quelle mesure peut-il y avoir encore des doublons dans la liste* `L` *après exécution de la fonction* `doublons` *?*

V

Historique des résultats

30. Pour chaque génération, on enregistre les cinq meilleurs génotypes dans une base de données.

30.1 On constitue ensuite une table `Historique` dont les attributs sont les suivants.

- `Id` : clé primaire, identifiant du génotype
- `gene_Kp` : flottant, paramètre K_p du génotype
- `gene_Ti` : flottant, paramètre T_i du génotype
- `gene_Td` : flottant, paramètre T_d du génotype
- `score` : flottant, valeur retournée par la fonction `calcul_cout` pour le génotype
- `apparition` : entier, numéro de la première génération pour laquelle le génotype a figuré parmi les cinq meilleurs génotypes
- `disparition` : entier, numéro de la dernière génération pour laquelle le génotype a figuré parmi les cinq meilleurs génotypes

30.2 La **longévité** d'un génotype dans cette table est le nombre de générations pour lesquelles il a figuré parmi les cinq meilleurs génotypes.

Q 24.a *Que calcule la requête suivante ?*

```
SELECT MIN(score) FROM Historique
```

Q 24.b *Que calcule la requête suivante ?*

```
SELECT Id FROM Historique
WHERE 15>apparition AND 15<disparition
```

Q 24.c *Écrire une requête qui calcule la valeur moyenne des paramètres K_p, T_i et T_d pour les cinq meilleurs génotypes de la 20-ième génération.*

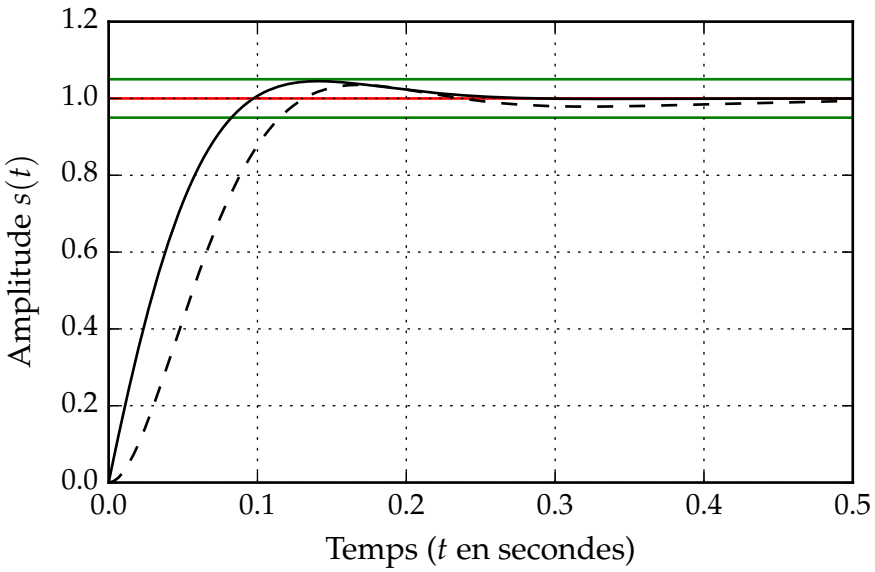
Q 24.d *Écrire une requête qui calcule la longévité du meilleur génotype.*

VI

Conclusion

31. Sur la figure suivante, on a représenté en tirets la réponse indicielle avec les paramètres choisis au **[10]** et en trait continu la réponse avec les paramètres fournis lors d'une exécution de l'algorithme génétique.

$$K_p = 97,3269 \qquad T_i = 0,01\,\text{s} \qquad T_d = 32,9862\,\text{s}$$



Q 25. *Comparer les deux graphes.*

Annexes

VII

Librairie scipy.signal

32. Le code ci-dessous illustre l'utilisation de la librairie `scipy.signal` pour tracer la réponse indicielle d'un système du second ordre caractérisé par sa fonction de transfert

$$G(p) = \frac{K}{1 + 2\dfrac{z \cdot p}{\omega_n} + \dfrac{p^2}{\omega_n^2}}.$$

```
import numpy as np
from scipy.signal import lti, step

K, z, wn = 1.3, 0.3, 10
d0, d1, d2 = 1, 2*z/wn, 1/wn**2
# Numérateur
num = [ K ]
# Dénominateur
den = [ d2, d1, d0 ]
# Fonction de transfert
G = lti(num, den)
# Réponse à un échelon
t, signal = step(G)
t, signal = step(G, T=np.linspace(min(t), t[-1], 1000))
```

32.1 Représentation des polynômes

Un polynôme est représenté par la liste de ses coefficients, rangés par ordre *décroissant* de degré. Ainsi, si le numérateur et le dénominateur de la fonction de transfert sont les polynômes

$$N(p) = n_0 + n_1 p + \cdots + n_k p^k \quad \text{et} \quad D(p) = d_0 + d_1 p + \cdots + d_\ell p^\ell,$$

ils seront représentés par les listes suivantes :

$$\texttt{num} = [n_k, n_{k-1}, \ldots, n_1, n_0] \quad \text{et} \quad \texttt{den} = [d_\ell, d_{\ell-1}, \ldots, d_1, d_0].$$

32.2 Fonction de transfert

La fonction `lti(num, den)` permet de définir la fonction de transfert $G(p)$ d'un système linéaire autonome (`lti` pour *Linear Time Invariant system*). Les arguments `num` et `den` sont les listes qui représentent le numérateur et le dénominateur de $G(p)$.

32.3 Réponse indicielle

La fonction de transfert `G` d'un système linéaire autonome étant définie (au moyen de `lti`), l'exécution de `step(G)` retourne un couple `(t, s)`. La liste `t` est un échantillon de N instants :

$$\texttt{t} = [t_0, t_1, \ldots, t_{N-1}] \qquad \text{où } t_0 = 0 \text{ et } t_{N-1} = T_{\max}$$

et la liste `s` est l'échantillon

$$\texttt{s} = [s(t_0), s(t_1), \ldots, s(t_{N-1})]$$

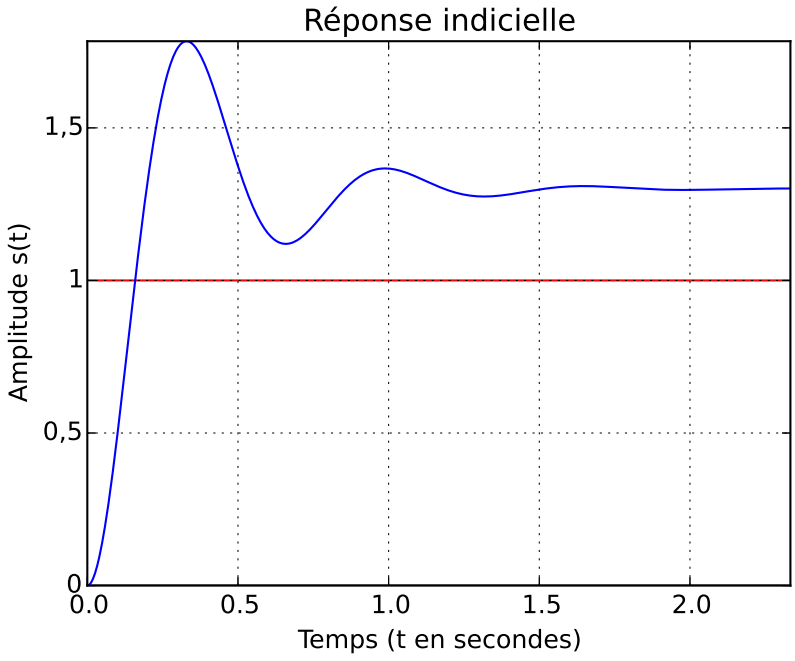
où $[t \mapsto s(t)]$ est la réponse indicielle du système, c'est-à-dire la réponse temporelle à un échelon (*step*) unitaire.
Sans précision de l'utilisateur, la fonction `step` détermine automatiquement la durée $T_{\max}$ de l'échantillonnage et le nombre N de points.
L'usage veut qu'on exécute deux fois la fonction `step` : la première fois en la laissant déterminer de manière autonome la durée $T_{\max}$, la seconde fois en récupérant cette durée ($T_{\max}$ = `t[-1]`) et en imposant le nombre N de points (utilisation de `linspace` en deuxième argument).

33. On peut alors tracer le graphe de la réponse indicielle.

```
from matplotlib import pyplot as plt

# Consigne (entrée)
plt.hlines(1, min(t), max(t), colors='r')
# Réponse à l'échelon (sortie)
plt.plot(t, signal)
# Légende
plt.title('Réponse indicielle')
plt.xlabel('Temps (t en secondes)')
plt.ylabel('Amplitude s(t)')

plt.hlines(0, min(t), max(t))
plt.xlim(xmax = max(t))
plt.grid()
```



VIII

Librairie numpy

34.1 On rappelle que `np.zeros(n)` retourne un tableau `numpy` de n termes nuls.

34.2 Racines d'un polynôme

Le polynôme $3x^3 - 3x^2 - 2x + 1$ est représenté par la liste `P` $= [3, -3, -2, 1]$. L'exécution de `np.roots(P)` retourne une liste donnant une valeur approchée des racines complexes du polynôme.
Si `Q` est une liste de nombres complexes, la commande `np.real(Q)` retourne la liste de leurs parties réelles.

34.3 Nombres aléatoires

Si a et b sont deux entiers, la commande `np.randint(a, b)` retourne un entier compris *au sens large* entre a et b. Des appels itérés à cette commande simulent correctement un échantillon de variables aléatoires indépendantes qui suivent la loi uniforme sur l'intervalle $\{a, a+1, \ldots, b-1, b\} \subset \mathbb{Z}$.

34.4 Tirages sans remise

Si `L` est une liste de longueur N et n, un entier inférieur à N, la commande `np.sample(L, n)` retourne une n-liste choisie au hasard parmi les $N!/n!$ listes possibles. Cela revient à effectuer n tirages successifs *sans remise* parmi les N éléments de la liste `L`.

IX

Structure du programme d'optimisation génétique

35.1 Paramètres globaux

Il ne devrait jamais y avoir de paramètres *numériques* dans un code, seulement des paramètres *littéraux*.
On définit donc toutes les constantes au début du code, ce qui permet de changer facilement leur valeur si le besoin s'en fait sentir.

```
n_pop = 100 # Taille de la population
lgr_g = 16  # longueur d'un gène
n_sel = 20  # Nombre d'individus avec lesquels on construit la génération suivante
cycle = 50  # Nombre de générations
```

35.2 Programme principal

Selon le paradigme *top-down*, on commence par écrire le programme principal en suivant l'algorithme choisi.

```
# Première génération (individus choisis aléatoirement)
L = generer_liste_initiale(n_pop)
# On met en route l'algorithme eugénique
for génération in range(cycle):
  # Sélection des meilleurs génotypes d'une génération
  top_liste, top_perf = selection(L, n_sel)
  # Création de la génération suivante (croisements et mutations)
  L = nouvelle_generation(top_liste)
  # Élimination des éventuels clones (remplacés par des individus aléatoirement choisis)
  doublons(L)
# On retient le meilleur génotype de la dernière génération
genes_choisis = L[0]
# On calcule les coefficients du correcteur PID associés à ce génotype
v = decodage(genes_choisis[0]), decodage(genes_choisis[1]), decodage(genes_choisis[2])
Kp1, Ti1, Td1 = calcul_coeff_correcteur(v[0], v[1], v[2])
```

35.3 On s'occupe ensuite d'écrire les diverses fonctions utilisées dans le programme principal (ainsi que les inévitables fonctions auxiliaires qui ont la noble tâche de simplifier la lecture du code final mais n'ont aucune raison d'apparaître dans le programme principal).

Fonction	Réalisation
`generer_liste_initiale(n_pop)`	**[Q17.]**
`selection(L, n_sel)`	**[Q19.]**
`nouvelle_generation(L)`	**[Q22.]**
`doublons(L)`	**[Q23.]**

35.4 Sur le programme principal, rien n'explique pourquoi la fonction `selection` doit retourner la liste des meilleurs génotypes ainsi que la liste de leurs performances (et non pas seulement la liste des meilleurs génotypes).
La raison est simple : pour constituer la base de données étudiée au **[30]**, il faut connaître la performance des meilleurs génotypes. On s'est donc donné le moyen de disposer de ces valeurs sans avoir à refaire les calculs (qui sont ici, et de loin, les plus coûteux en temps). Il suffit alors d'ajouter une ligne au programme principal pour constituer la base de données.

Corrigé

I Position du problème

R 1. Comme l'exige la documentation de `scipy.signal`, on représente un polynôme par la liste de ses coefficients, en commençant par le coefficient de plus haut degré.

```
numG = [ 8 ]
denG = [ 0.0072, 0.273, 1.13, 9 ]
```

II Correction du système

R 2.a D'après la définition du correcteur,

$$C(p) = \frac{K_p \cdot (1 + T_i \cdot p + T_i T_d \cdot p^2)}{T_i \cdot p}.$$

On peut donc poser $N_C(p) = K_p T_i T_d \cdot p^2 + K_p T_i \cdot p + K_p$ et $D_C(p) = T_i \cdot p + 0$.

R 2.b Traduction immédiate.

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

Produit de deux polynômes

R 3.a D'après les coefficients donnés, $P = X^2 + X + 1$ et $Q = X - 1$. (Pour Q, il faut se souvenir que le premier coefficient de la liste est le coefficient dominant du polynôme.)

R 3.a D'après le code, $n = 3 + 2 - 1 = 4$ et les listes $P1$ et $Q1$ sont obtenues en complétant les listes P et Q par des zéros. Les valeurs de $P1$ et $Q1$ sont donc respectivement $[1, 1, 1, 0]$ et $[1, -1, 0, 0]$.

R 3.b Une boucle `for` termine toujours...

R 3.b Notons a et b, les degrés respectifs de P et Q.
Par définition, la longueur de P est égale à $a + 1$. La longueur de R, égale par définition à n, est donc égale à

$$[a + 1] + [b + 1] - 1 = a + b + 1$$

donc $\deg R = \deg P + \deg Q = \deg(PQ)$.
Considérons maintenant les coefficients des trois polynômes :

$$P = p_0 X^a + p_1 X^{a-1} + \cdots + p_a \qquad P1 = [p_0^1 = p_0, p_1^1 = p_1, \ldots, p_a^1 = p_a, p_{a+1}^1 = 0, \ldots, p_{a+b}^1 = 0]$$
$$Q = q_0 X^b + q_1 X^{b-1} + \cdots + q_a \qquad Q1 = [q_0^1 = q_0, q_1^1 = q_1, \ldots, q_b^1 = q_b, q_{b+1}^1 = 0, \ldots, q_{a+b}^1 = 0]$$
$$R = r_0 X^{a+b} + r_1 X^{a+b-1} + \cdots + r_{a+b}.$$

D'après le code,

$$\forall\, 0 \leqslant k < a+b+1, \quad r_k = \sum_{0 \leqslant i < k+1} p_i^1 q_{k-i}^1 \qquad \text{soit} \qquad \forall\, 0 \leqslant k \leqslant a+b, \quad r_k = \sum_{0 \leqslant i \leqslant k} p_i^1 q_{k-i}^1$$

(puisque tous les indices sont des entiers) tandis que, d'après le cours sur les polynômes,

$$\forall\, 0 \leqslant k \leqslant a+b, \qquad r_k = \sum_{\substack{0 \leqslant i \leqslant a \\ 0 \leqslant j \leqslant b \\ i+j=(a+b)-k}} p_{a-i} q_{b-j} = \sum_{\substack{0 \leqslant a-i \leqslant a \\ 0 \leqslant b-j \leqslant b \\ (a-i)+(b-j)=k}} p_{a-i} q_{b-j} = \sum_{\substack{0 \leqslant u \leqslant a \\ 0 \leqslant v \leqslant b \\ u+v=k}} p_u q_v.$$

Si $k-i > b$, alors $q_{k-i}^1 = 0$ (par définition de $Q1$) et si $i > a$, alors $p_i^1 = 0$ (par définition de $P1$). Dans tous les autres cas, on a $0 \leqslant i \leqslant a$, $0 \leqslant k-i \leqslant b$ et $i+(k-i) = k$, donc on a bien

$$\forall\, 0 \leqslant k \leqslant a+b, \qquad \sum_{\substack{0 \leqslant u \leqslant a \\ 0 \leqslant v \leqslant b \\ u+v=k}} p_u q_v = \sum_{0 \leqslant i \leqslant k} p_i^1 q_{k-i}^1$$

ce qui prouve que la liste R contient exactement les coefficients du produit PQ (en suivant la convention en vigueur).

R 4. Le code traduit simplement l'égalité suivante.

$$F(p) = \frac{N_1(p)N_2(p)}{D_1(p)D_2(p)}$$

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

R 5. Somme de deux polynômes

Les explications suivent le code.

```
def somme_poly(P, Q):
  deg_max = max([len(P), len(Q)])
  S = np.zeros(deg_max)
  S[-len(P):] += P
  S[-len(Q):] += Q
  return S
```

En général, $\deg(P+Q) \leqslant \max\{\deg P, \deg Q\}$ mais pour nous, qui ne manipulons ici que des polynômes à coefficients *positifs*,

$$\deg(P+Q) = \max\{\deg P, \deg Q\}.$$

On commence donc par créer un tableau de zéros de la bonne taille.
Comme les coefficients sont rangés par ordre de degré décroissant, il faut sommer les coefficients de P et de Q en les calant à droite.

	X^b	$\cdots$	X^{a+1}	X^a	$\cdots$	X^0
Initialisation de S	0	$\cdots$	0	0	$\cdots$	0
P	0	$\cdots$	0	p_0	$\cdots$	p_a
Q	q_0	$\cdots$	q_{b-a-1}	q_{b-a}	$\cdots$	q_b

(Pour s'y retrouver à coup sûr, remarquer que la somme de l'indice et de l'exposant est *constante* pour chaque polynôme, égale au degré de ce polynôme.)

III Critères d'optimisation

R 6. On calcule la liste des racines du polynôme P. On parcourt ensuite cette liste pour vérifier que chaque partie réelle est strictement négative.
Il se peut que le polynôme P possède une racine imaginaire pure. Comme la fonction `np.roots` calcule des valeurs approchées des racines de P, elle pourrait alors retourner une partie réelle très petite mais strictement négative ! On **convient** ici que la partie réelle d'une racine est strictement négative lorsque la partie réelle de sa valeur approchée calculée par `np.roots` est inférieure à -10^{-10}.

```
def stabilite(P):
  racines = np.roots(P)
  stable = True
  for r in racines:
    stable = stable and (np.real(r)<-1e-10)    # Re(r) < 0
  return stable
```

Variante : la même chose en version compacte !

```
def stabilite(P):
    return (np.real(np.roots(P))<-1e-10).all()
```

Variante 2 : on peut aussi s'arrêter dès qu'on tombe sur une racine dont la partie réelle est positive. (Si on parvient au terme de la boucle `for`, c'est que toutes les parties réelles sont strictement négatives.)

```
def stabilite(P):
  for r in np.roots(P):
    if (np.real(r)>-1e-10): # convention numérique pour Re(r) >= 0
      return False          # sortie anticipée de la boucle for et de la fonction
  return True               # sortie normale
```

(Personnellement, je n'aime pas qu'une fonction fasse apparaître plus d'une instruction `return`.)

R 7. Dans les deux cas, l'erreur indicielle est nulle : la valeur finale est égale à la valeur de consigne.
Dans le second cas, le temps de réponse est un peu plus court (on passe de $0,15\,\text{s}$ à $0,12\,\text{s}$ environ) et le dépassement relatif a considérablement diminué (on passe de 20% à moins de 5% de dépassement).

R 8.a La variable `s_fin` est égale au dernier élément de la liste `s`, c'est donc la valeur finale (en admettant que la durée d'observation soit assez longue pour que le régime permanent soit atteint : c'est bien ce que l'on constate sur les différentes figures).
Les variables `dep_i` et `dep_s` sont booléennes. La variable `dep_i` (pour *dépassement inférieur*) prend la valeur `True` lorsque l'amplitude de la réponse est au moins égale à 95% de la valeur finale à l'instant `t[j]` mais inférieure à 95% de la valeur finale à l'instant précédent `t[j-1]`.
De manière analogue, la variable `dep_s` (pour *dépassement supérieur*) est égale à `True` lorsque l'amplitude de la réponse est au plus égale à 105% de la valeur finale à l'instant `t[j]` mais supérieure à 105% de la valeur finale à l'instant précédent.
L'un de ces deux booléens est donc égal à `True` lorsqu'on sort de l'intervalle $[s_\infty - 5\%, s_\infty + 5\%]$ entre l'instant `t[j]` et l'instant précédent `t[j-1]`.
L'indice j *décroissant* dans la boucle depuis sa valeur maximale (égale à `len(t)` -1), le premier indice `j` pour lequel un dépassement se produit est donc *le plus grand* indice j_0 pour lequel

$$t_{j_0-1} \notin [s_\infty - 5\%, s_\infty + 5\%] \qquad \text{et} \qquad \forall\, j \geqslant j_0, \quad t_j \in [s_\infty - 5\%, s_\infty + 5\%].$$

Le temps de réponse à 5% vérifie donc $t_{j_0-1} < T_5 \leqslant t_{j_0}$ et la valeur t_{j_0} retournée par la fonction `temps_reponse` est un majorant de T_5.

R 8.b On suit le même principe : on parcourt la liste des amplitudes depuis la fin (= lorsque la valeur finale est atteinte) et dès qu'on trouve un indice j tel que `s[j]` n'est plus dans l'intervalle $[s_\infty - 5\%, s_\infty + 5\%]$, on retourne l'instant `t[j+1]`.

```
def temps_reponse(s, t):
  s_fin, T5 = s[-1], t[-1]
  seuil_min, seuil_max = 0.95*s_fin, 1.05*s_fin
  j = len(s)-1
  while (seuil_min<s[j] and s[j]<seuil_max):
  # Tant qu'on reste dans l'intervalle [s_∞ − 5%, s_∞ + 5%], on remonte le temps
    j = j-1
  return t[j+1]
```

R 9. Le dernier élément d'une liste (ou d'un tableau unidimensionnel) est l'élément d'indice -1.

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

REMARQUE.— L'expression de D_1 ne fait pas apparaître les instants d'échantillonnage. Par conséquent, la fonction `depassement` n'utilise pas le tableau `t`, bien qu'il fasse partie de la liste des arguments de cette fonction.

R 10. On compare la sortie $s(t)$ à la consigne $e(t) = 1$ par un calcul d'intégrale. Si la durée d'observation $T_{\max}$ est assez longue, on peut convenir que

$$\int_0^{+\infty} \big|s(t) - e(t)\big| \, \mathrm{d}t \approx \int_0^{T_{\max}} \big|s(t) - e(t)\big| \, \mathrm{d}t,$$

ce qui est manifestement le cas pour chacun des graphes qui figure dans l'énoncé. Cela permet d'estimer la valeur de l'intégrale par la méthode des rectangles (cette méthode ne peut s'appliquer que lorsque l'intervalle d'intégration est *borné*) : en notant N, la longueur des échantillons `s` et `t`,

$$\int_0^{T} \big|s(t) - e(t)\big| \, \mathrm{d}t \approx \sum_{k=1}^{N-1} |s_k - 1| \cdot (t_k - t_{k-1}).$$

On n'a pas vraiment besoin de l'indice k dans ce calcul, il suffit de parcourir simultanément les trois listes $(s_k)_{1\leqslant k<N}$ = `s[1:]`, $(t_k)_{1\leqslant k<N}$ = `t[1:]` et $(t_{k-1})_{1\leqslant k<N} = (t_k)_{0\leqslant k<N-1}$ = `t[:-1]` (qui ont toutes même longueur).

```
def critere_EI(s, t):
  EI = 0
  for sk, td, tg in zip(s[1:], t[1:], t[:-1]):
    EI += abs(1-sk)*(td-tg)
  return EI
```

R 11.a Il s'agit de comparer des triplets, c'est-à-dire des vecteurs de $\mathbb{R}^3$. Il n'y a pas de relation d'ordre total naturelle sur $\mathbb{R}^3$, alors qu'on dispose d'une relation d'ordre total sur $\mathbb{R}$ (avec $\leqslant$).
Une fonction de coût $f : \mathbb{R}^3 \to \mathbb{R}$ permet alors de comparer deux triplets : on **convient** que le triplet (x_1, y_1, z_1) est meilleur que le triplet (x_2, y_2, z_2) lorsque $f(x_1, y_1, z_1) \leqslant f(x_2, y_2, z_2)$ (puisqu'on cherche ici à minimiser f).

REMARQUE.— On pourrait procéder autrement en convenant que le triplet (x_1, y_1, z_1) est meilleur que le triplet (x_2, y_2, z_2) lorsqu'on a simultanément $x_1 \leqslant x_2$, $y_1 \leqslant y_2$ et $z_1 \leqslant z_2$. Mais cette relation d'ordre n'est pas totale sur $\mathbb{R}^3$ et certains triplets ne peuvent donc pas être comparés au triplet de référence (ce qui est gênant pour l'exécution d'un algorithme).
Pour des raisons techniques, on préfère donc une *amélioration globale de l'ensemble des critères* (via la fonction de coût) à une *amélioration sur chaque critère*.

R 11.b On rapporte chaque critère à une valeur de référence (en calculant les trois quotients) et la fonction `ponderation_cout` retourne l'isobarycentre des trois proportions.
Un triplet (T, D, I) est donc meilleur que le triplet de référence (T_0, D_0, I_0) lorsque le rapport moyen

$$\frac{1}{3} \cdot \left(\frac{T}{T_0} + \frac{D}{D_0} + \frac{I}{I_0} \right)$$

est inférieur à 1. Bien entendu, ce rapport moyen peut être inférieur à 1 même si un des trois rapports est supérieur à 1... On l'a déjà dit, on retient ici une solution d'amélioration *globale* de l'ensemble des critères.

R 11.c Si les constantes k_1, k_2 et k_3 ne sont pas toutes égales, on définit une moyenne globale qui n'attribue pas la même importance à chacun des trois critères (le critère privilégié est celui qui porte la constante k_i la plus grande).

R 12.a En informatique, on ne peut manipuler que des quantités **discrètes** (= qui prennent un nombre *fini* de valeurs différentes). Les variables **continues** (= qui peuvent prendre un nombre *infini* de valeurs différentes) n'existent qu'en mathématiques...

R 12.b La différence entre deux valeurs de K_p est un multiple entier de $(K_{p,\max} - K_{p,\min})/(2^{16} - 1) \approx 1,5.10^{-3}$.
La valeur optimale de K_p sera donc connue à 10^{-3} près, on ne pourra en aucun cas espérer un résultat plus précis. Cela dit, une telle précision est largement suffisante en pratique !
(Idem pour les deux autres grandeurs, bien sûr !)

R 12.c Il y a 2^{16} valeurs possibles pour chacune des trois grandeurs, soit $(2^{16})^3 = 2^{48} \approx 10^{15}$ triplets. C'est énorme !

REMARQUE.— On *doit savoir* que $2^{10} = 1024 \approx 10^3$. Par conséquent, $2^{48} = (2^{10})^5 \cdot 2^{-2} \approx 1/4 \cdot 10^5$.

R 13. Puisqu'il s'agit d'effectuer trois calculs analogues, on définit une fonction auxiliaire, ce qui vaut mieux que d'écrire trois fois de suite le même code.
Autre bonne pratique : pas de constante numérique, uniquement des constantes littérales ! (Et en particulier `2**lgr_g` au lieu du `2**16` suggéré par l'énoncé.)

```
# Constantes de référence
Kp_min, Kp_max = 0.01, 100.
Ti_min, Ti_max = 0.01, 100.
Td_min, Td_max = 0., 50.

# Fonction auxiliaire
def c_c_c_aux(i, v_min, v_max):
    return (v_max-v_min)*i/(2**lgr_g-1)+v_min

def calcul_coeff_correcteur(Ip, Ii, Id):
    Kp = c_c_c_aux(Ip, Kp_min, Kp_max)
    Ti = c_c_c_aux(Ii, Ti_min, Ti_max)
    Td = c_c_c_aux(Id, Td_min, Td_max)
    return Kp, Ti, Td
```

R 14. Compte-tenu des fonctions déjà définies, il suffit de suivre pas à pas le cahier des charges...

```
def calcul_cout(Ip, Ii, Id):
  Kp, Ti, Td = calcul_coeff_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_BF) # stabilité de la FTBF
  if stable:
    # calcul de la réponse indicielle pour un système stable
    t, s = rep_Temp(Kp, Ti, Td)
    # calcul des trois critères
    T5 = temps_reponse(s, t) # temps de réponse à 5%
    D1 = depassement(s, t)   # dépassement
    EI = critere_EI(s, t)    # erreur intégrale
    # calcul du coût de ces trois critères
    cout = ponderation_cout(T5, D1, EI)
  else:
    # coût par défaut d'un système instable
    cout = np.inf
  return cout
```

REMARQUE.— Ne surtout pas hésiter à utiliser une fonction qu'on n'aurait pas écrite ! Ce n'est pas une difficulté, c'est le principe de la programmation *top-down* (même si la question présente suit le principe *bottom-up* : synthèse de fonctions simples déjà programmées pour élaborer une fonction complexe).

R 15. On a compté environ 10^{15} triplets à tester. Si chaque test dure environ 10^{-2} secondes, la durée totale du test est de l'ordre de 10^{13}s. Seulement voilà, il n'y a que $3 \cdot 10^7$ secondes par an, on en a pour quelques dizaines de milliers d'années...

IV Résolution par un algorithme génétique

R 16. La fonction `randint(0,1)` retourne un entier compris au sens large entre 0 et 1, c'est-à-dire égal à 0 ou à 1. L'utilisation répétée de cette commande simule une suite de variables aléatoires indépendantes qui suivent toutes la loi de Bernoulli $\mathscr{B}(1/2)$.

La fonction `rand_b` traduit la suite aléatoire de 0 et de 1 en suite aléatoire de `False` et de `True`.

Le gène retourné par la fonction `genererGene(lgr)` est donc une liste de `lgr` booléens. Pour suivre l'énoncé, il faut prendre `lgr` = `lgr_g` = 16.

Comme les booléens sont choisis indépendamment et avec équiprobabilité, la création d'un gène revient à choisir un gène au hasard (avec équiprobabilité) parmi les 2^{16} gènes possibles.

R 17. Puisqu'il faut créer n_{pop} individus, l'indice i de la boucle `for` doit prendre n_{pop} valeurs différentes. Par ailleurs, chaque génotype est une liste de 3 gènes.

```
def generer_liste_initiale(n_pop):
  individus = []
  for i in range(n_pop):
    c = [ genererGene(lgr_g), genererGene(lgr_g), genererGene(lgr_g) ]
    individus.append(c)
  return individus
```

REMARQUE.— L'utilisation de la constante globale `lgr_g` (au lieu de la valeur numérique 16) est une *bonne pratique*. Qu'on se le dise !

R 18. D'après la convention donnée par l'énoncé, l'entier codé par le gène `g` $= [g_0, \dots, g_{\ell-1}]$ est égal à

$$\sum_{0 \leqslant i < \ell} 2^i g_i = \sum_{\substack{0 \leqslant i < \ell \\ g_i = \texttt{True}}} 2^i$$

(en suivant l'usage habituel : `True` vaut 1 et `False` vaut 0).
On va donc calculer de proche en proche les termes d'une suite géométrique ($2^k = 2 \cdot 2^{k-1}$) et en déduire progressivement la valeur de l'entier `n_g`.

```
def decodage(g):
  n_g, v = 0, 1  # v_0 = 2^0
  for gi in g:
    if gi:       # Si g_i vaut True,
      n_g += v   # alors on ajoute v_i = 2^i à n_g. (Sinon, on ne modifie pas la valeur de n_g.)
    v *= 2       # On passe de v_i à v_{i+1} = 2v_i = 2^{i+1}.
  return n_g
```

R 19.a Si la longueur de `Perf` est égale à n et si $p <$`Perf[-1]`, la fonction `insertion` commence par calculer l'unique indice $0 \leqslant i < n$ tel que

$$p_0 \leqslant p_1 \leqslant \cdots \leqslant p_{i-1} < p \leqslant p_i \leqslant \cdots \leqslant p_{n-1} = \texttt{Perf[-1]}.$$

On scinde alors les deux listes `Top` et `Perf` en deux morceaux, en éliminant le dernier élément de chacune des deux listes.

$$[T_0, \dots, T_{n-1}] \mapsto [T_0, \dots, T_{i-1}] / [T_i, \dots, T_{n-2}] \qquad [P_0, \dots, P_{n-1}] \mapsto [P_0, \dots, P_{i-1}] / [P_i, \dots, P_{n-2}]$$

et on insère `c` et `p` dans ces deux listes, de telle sorte que la nouvelle liste `Perf` soit encore une liste croissante de n réels.

$$[T_0, \dots, T_{i-1}] / [T_i, \dots, T_{n-2}] \mapsto [T_0, \dots, T_{i-1}, c, T_i, \dots, T_{n-2}]$$
$$[P_0, \dots, P_{i-1}] / [P_i, \dots, P_{n-2}] \mapsto [P_0, \dots, P_{i-1}, p, P_i, \dots, P_{n-2}]$$

R 19.b Initialement, `T` est une liste de n individus fictifs (des listes vides) et `P` est une liste de n performances infinies.
Les n premiers individus de la liste `L` sont insérés dans la liste `T` par ordre croissant de performance (mécanisme de la fonction `insertion`). Pour chacun de ces individus, la performance réalisée est nécessairement meilleure (inférieure) que le plus grand élément de `P` (qui est $+\infty$).
Pour les individus suivant, on calcule la performance et, seulement si on a trouvé une performance strictement inférieure au plus grand élément de la liste `P`, on insère le individu dans la liste `T` et sa performance dans la liste `P`.
On obtient ainsi la liste `T` des n meilleurs génotypes de `L`, avec la liste `P` de leur performances respectives.

R 19.c De toutes les opérations effectuées, c'est l'exécution de `calcul_cout` qui est la plus coûteuse en temps (et de très loin). Or cette fonction est calculée une fois, et une seule, pour chaque individu et comme on ne peut pas faire moins, le code n'est pas loin d'être optimal.

REMARQUE.— On pourrait améliorer le code de la fonction `insertion` (on connaît un algorithme d'insertion dans une liste triée de complexité sous-linéaire), mais ce serait une amélioration marginale du point de vue du temps total de calcul (on va insérer des éléments parmi les 20 meilleurs...) et cela compliquerait la lecture du code...

REMARQUE.— De même, on connaît des algorithmes de tri bien plus performants que le tri par insertion. Mais nous ne cherchons qu'à trier les 20 meilleurs éléments et non pas à trier toute la liste `L` !

R 20. La fonction `DCR` découpe après l'indice i et après l'indice j, croise et recombine les morceaux.
Pour suivre l'énoncé, on prend $i = 4$ et $j = 12$. Lorsque p parcourt `P1` et que q parcourt `P2`, on calcule le descendant `E1` en travaillant sur le couple (g,h) et le descendant `E2` avec le couple symétrique (h,g).

```
def croisement(P1, P2):
  E1 = [DCR(g, h, 4, 12) for (g,h) in zip(P1, P2)]
  E2 = [DCR(h, g, 4, 12) for (g,h) in zip(P1, P2)]
  return E1, E2
```

REMARQUE.— Pour *bien faire*, il faudrait définir des constantes globales au lieu d'utiliser les valeurs numériques 4 et 12 (*air connu*).

R 21. L'argument `c` est un génotype, c'est-à-dire une liste de trois gènes. On choisit au hasard un entier $0 \leqslant i \leqslant 2$. Le gène `c[i]` est une liste de 16 booléens et on choisit au hasard un entier $0 \leqslant j \leqslant 15$.
En python, les listes sont des objets *mutables* : on peut donc modifier le j-ème booléen du gène $c[i]$ (l'opérateur `not` change un booléen en son contraire). Il n'est pas nécessaire de retourner la liste modifiée.

R 22. Pour le moment, on a choisi 10 éléments parmi les 19 éléments de la liste `L[1:]` (les vingt meilleurs génotypes à l'exception du meilleur) et on a créé 20 nouveaux individus en croisant le meilleur génotype avec ces 10 génotypes (on a créé 2 descendants à chaque croisement). Il y a donc pour le moment $20 + 20 = 40$ individus dans la liste `L`.
Il reste à choisir 30 couples dans la liste `L[1:20]` pour créer $2 \times 30 = 60$ nouveaux individus par croisement (de manière à obtenir une nouvelle liste de $40 + 60 = 100$ individus). Le code manquant est le suivant.

```
for i in range(30):
  ech = sample(L[1:20], 2)  # deux éléments distincts choisis au hasard parmi L[1],...,L[19]
  croisement_mutation(ech[0], ech[1], L)
```

REMARQUE.— Lors de la définition de la liste `suivants`, la longueur de `L` est égale à 20. Mais chaque appel à la fonction `croisement_mutation` ajoute deux éléments à la liste `L`. Par la suite, il faut donc bien préciser qu'on choisit les couples dans `L[1:20]` et non dans `L[1:]`.

R 23.a Il y a un doublon lorsqu'il existe deux indices $0 \leqslant j < i$ tels que $L_i = L_j$ (l'individu L_i est alors un **clone** de L_j). Chaque individu $c = L_i$ pris dans $(L_1, \ldots, L_{n-1})$ doit donc être comparé avec les individus L_0, …, L_{i-1} qui le précèdent et, en cas d'égalité des génotypes, il doit être remplacé.

```
def doublons(L):
  for i in range(1, len(L)):
    c = L[i]
    if c in L[:i]:  # Si le génotype c figure dans le début de la liste L,
                    # on le remplace par un génotype pris au hasard.
      L[i] = [ genererGene(lgr_g), genererGene(lgr_g), genererGene(lgr_g) ]
  return None
```

Ici encore, il est inutile de retourner une valeur puisque la fonction `doublons` modifie la liste `L` (type mutable, on l'a rappelé).

REMARQUE.— La complexité de la fonction `doublons` ne peut être moins que quadratique : il ne suffit pas de parcourir la liste des génotypes, il faut comparer chaque génotype à tous les autres ! Heureusement pour nous, chaque comparaison est peu coûteuse en temps (beaucoup moins que le calcul de la performance de chaque génotype, qui est le vrai facteur limitant de notre programme) et l'exécution de la fonction `doublons` ne ralentira pas particulièrement le programme global.

R 23.b Dès qu'un individu faisant doublon est détecté, il est remplacé par un individu pris au hasard de façon équiprobable parmi les $2^{48} \approx 10^{15}$ individus possibles. La probabilité de choisir l'un des 100 individus de la liste est environ $10^2/10^{15} = 10^{-13}$... Autrement dit, il n'y a *aucune chance* de créer un doublon en remplaçant un clone par un individu pris au hasard !

V Historique des résultats

R 24.a La requête retourne la plus petite valeur de l'attribut `score` dans la table `Historique`, c'est-à-dire le coût du meilleur génotype calculé par notre programme.

R 24.b On retourne l'identifiant des génotypes qui sont apparus avant la quinzième génération et qui ont disparus après la quinzième génération. Ils font donc partie des cinq meilleurs génotypes de la quinzième génération.

REMARQUE.— Il se peut que certains des meilleurs génotypes de la quinzième génération disparaissent avec la quinzième génération : ces génotypes ne figureront pas dans le résultat de la requête.

R 24.c Les cinq meilleurs génotypes de la vingtième génération sont ceux qui ont apparu au plus tard à la vingtième génération et disparu au plus tôt dès la vingtième génération. On applique la fonction **AVG** (pour *average*) aux attributs `gene_Kp`, `gene_Ti` et `gene_Td` pour ces cinq enregistrements.

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

R 24.d *Admettons* qu'il n'y ait qu'un seul meilleur génotype ! On identifie ce génotype par son score au moyen d'une sous-requête. Il ne reste plus qu'à récupérer les générations de son apparition et de sa disparition.

```
SELECT disparition-apparition AS longevite
FROM Historique
WHERE score = (SELECT MIN(score)
               FROM Historique)
```

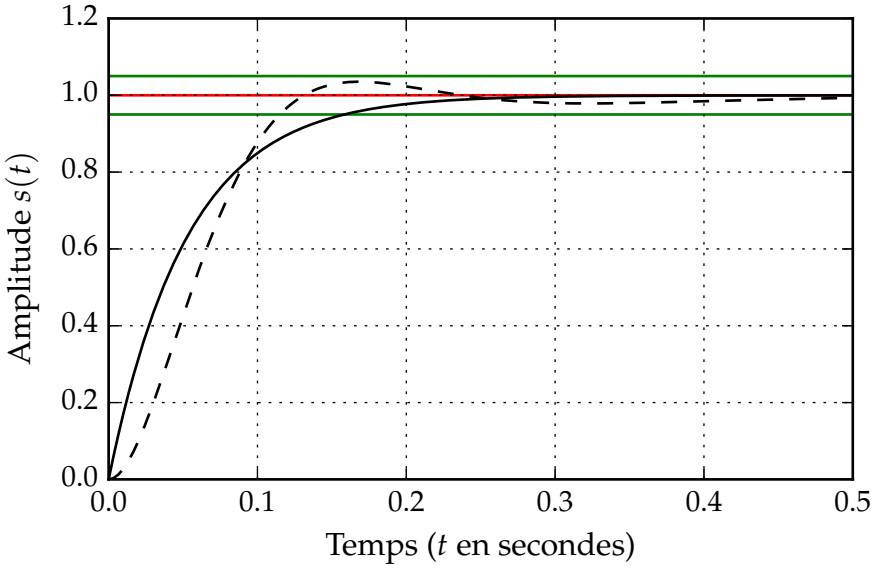
VI Conclusion

R 25. On a maintenant un meilleur correcteur du point de vue du temps de réaction (on passe de $0,12$s à $0,08$s environ) et du critère intégral (les oscillations sont moindres et amorties plus vite).
En revanche, le dépassement paraît un peu plus important que pour le correcteur précédent. Toutefois, le dépassement relatif reste inférieur à 5%, ce qu'on a jugé acceptable pour cette étude [**15.3**].

- En exécutant à nouveau le programme (à partir d'une autre population initiale), on arrive à

$$K_p = 88,786 \qquad T_i = 0,430\,\text{s} \qquad T_d = 35,548\,\text{s}$$

et à la figure suivante.

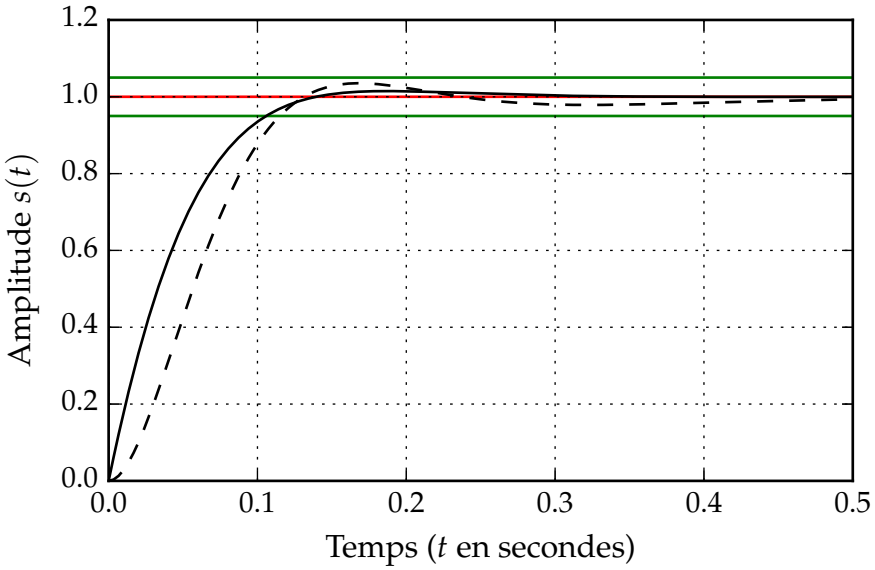


Cette fois, le dépassement a quasiment disparu et le critère intégral semble s'être également amélioré, mais au prix d'un temps de réponse allongé ($0,15\,\text{s}$ environ).

- Partant de ces données, on arrive par tâtonnements aux valeurs suivantes :

$$K_p = 93,25 \qquad T_i = 0,2\,\text{s} \qquad T_d = 34,25\,\text{s}$$

pour lesquelles on observe la figure suivante.



Il semble cette fois que les critères aient été améliorés !