Concours Blanc 1 - Janvier 2016 : Correction

1 Tracé de courbes représentatives de fonctions et calcul d'intégrales

1.1 Tracé de la courbe représentative d'une fonction

Question 1. Créer une fonction `f(x)` qui prend en argument un flottant `x` et renvoie le résultat de :

$$f(x) = 1 - \frac{1+x}{1+x^4}$$

```
def f(x):
      return 1-(1+x)/(1+x**4)
```

Question 2. Construire la liste `X=[0, 0.02, 0.04, 0.06 , ... , 1.98, 2]`. On pourra utiliser la méthode de son choix.

Une construction par compréhension est la plus simple et la plus claire ici :

```
X=[i*0.02 for i in range(101)]
```

On peut également partir d'une liste vide à laquelle on ajoute des éléments dans une boucle :

```
X=[]
for i in range(101):
      X.append(i*0.02)
```

Enfin, il est possible d'utiliser la commande `linspace` du module `numpy`. Attention toutefois à ne surtout pas utiliser des arguments non entiers dans un `range` usuel!

```
import numpy as np
X=np.linspace(0,2,101)
```

Question 3. Construire la liste `Y` des images par f des éléments de la liste `X`

Ici une construction par compréhension avec parcours de `X` par éléments est de loin la solution la plus simple :

```
Y=[f(x) for x in X]
```

Question 4. Quelle(s) commande(s) permettent d'afficher la courbe représentative de f sur $[0\,;2]$?

```
import matplotlib.pyplot as plt
plt.plot(X,Y)
plt.show()
```

1.2 Méthodes des trapèzes

Question 5. Écrire une fonction `Trapeze(a,b,f,n)` qui prend en argument deux flottants `a` et `b` représentant les bornes d'intégration ; une fonction `f` et un entier strictement positif `n` représentant le nombre de pas d'intégration, et qui renvoie la valeur approchée de l'intégrale de f sur $[a;b]$ avec n pas, calculée par la méthode des trapèzes.

La question a été traitée en TP. On prendra garde à ne par compter les bornes plusieurs fois, ou les oublier. Le plus naturel :

```
def Trapeze(a,b,f,n):
      s=0
      pas=(b-a)/n
      for i in range(n-1):
            s=s+pas*(f(a+i*pas)+f(a+(i+1)*pas))/2
      return s
```

Question 6. Calculer avec la méthode des trapèzes, en utilisant $n = 10^4$ points l'intégrale suivante :

$$\int_0^2 1 - \frac{1+x}{1+x^4}\mathrm{d}x$$

On pourra utiliser toutes les fonctions de l'exercice ; y compris celles de la partie 1.1.

Une syntaxe du type `lambda x:` est maladroite ici, f étant déjà définie. Très simplement :

```
print(Trapeze(0,2,f,10**4))
```

2 Autour du codage des flottants

On rappelle que le codage des flottants en double précision, c'est à dire sur 64 bits, d'un réel x utilise l'écriture $x = (-1)^s 2^e(1+m)$ où s est codé sur 1 bit, e est codé sur 11 bits et est décalé de -1023 (ainsi le codage 100 0000 0001 correspond à l'exposant 1025-1023=2) et que e appartient à l'intervalle [-1022 ; +1023]. Enfin m est codé sur 52 bits.

2.1 Deux exemples

Question 7. Coder à la main -19,75 ; on prendra soin de regrouper les bits quatre par quatre.

- Le signe de -19,75 est négatif : s=1
- On a 19,75=16+2+1+0.5+0.25=$2^4 + 2^1 + 2^0 + 2^{-1} + 2^{-2} = 2^4(1 + 2^{-3} + 2^{-4} + 2^{-5} + 2^{-6}$
- L'exposant est donc 4, il est décalé de 1023 donc on code 4+1023=1027 : e=100 0000 0011
- La mantisse se lit dans l'écriture précédente : m=0011 1100 0000 0000

ce qui donne :
1100 0000 0011 0011 1100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

Question 8. Décoder à la main le nombre flottant codé en double précision par :
0100 0000 0100 1010 1110 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

On isole les données pour commencer :
- s=0 : le nombre est positif
- e=100 0000 0100 : code 1028 ; après décalage de 1023 on a un exposant égal à 5
- m=1010 1110 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 ; on a codé $1+2^{-1}+2^{-3}+2^{-5}+2^{-6}+2^{-7}$

Au final, le nombre codé est :
$2^5(1 + 2^{-1} + 2^{-3} + 2^{-5} + 2^{-6} + 2^{-7}) = 2^5 + 2^4 + 2^2 + 2^0 + 2^{-1} + 2^{-2} = 32 + 16 + 4 + 1 + 0.5 + 0.25 = 53.75$

2.2 Intégrité des données binaires

La suite de ce problème est dédiée à l'étude de la conversion de nombres décimaux en nombres flottants et vice-versa.
Nous utiliserons la variable x pour désigner le flottant écrit en base décimale.
Pour nos représentations de données sous formes binaires nous utiliserons trois variables :
- s qui sera un entier, représentant le signe en codage flottant double précision.
- E qui sera une liste de longueur 11, ne contenant que des 0 et des 1 et représentant l'exposant tel que codé en double précision, lue de gauche à droite.
- L qui sera une liste de longueur 52, ne contenant que des 0 et des 1 et représentant la mantisse tel que codée en double précision, lue de gauche à droite.

Question 9. Quelles seraient les valeur de s, E et L pour le nombre flottant 1. ; on prendra gare au décalage de 1023 dans l'exposant.

Le codage de 1 en flottant double précision est : s=1 ; e=1023 ; m=0..

Avec les spécifications de l'énoncé on obtient :
- s=0
- E=[0,1,1,1,1,1,1,1,1,1,1]
- M=[0,...,0] (ou M=[0]*52 ou encore [0 for i in range(52)] si on souhaite rester rigoureux)

Question 10. Écrire une fonction `test01(L)` qui prend en argument une liste `L` et qui retourne le booléen `True` si `L` n'a que des 0 ou des 1 et le booléen `False` sinon.

On parcourt la liste L (par éléments) ; si on rencontre autre chose qu'un 0 ou un 1 ; on renvoie `False` ce qui interrompt la fonction (et le parcours de la boucle) ; sinon on ne fait rien ; on renvoie `True` à la fin du parcours de la boucle :

```
def test01(L):
      for x in L:
            if x!=0 and x!=1:
                  return False
      return True
```

Question 11. Écrire une fonction `test(s,E,M)` d'arguments un entier s, et deux listes E et M, qui vérifient si les variables s, E et M sont susceptibles de représenter un codage de nombre flottant dans notre convention. On prendra garde aux longueurs de E et L et on se servira de la fonction précédente.

On vérifie les types des objets utilisés, leur longueur pour les listes ; puis on réutilise la fonction précédente :

```
def test(s,E,M):
      if type(s)!=int or type(E)!=list or type(M)!=list:
            return False
      if len(E)!=11 or len(M)!=52:
            return False
      if s!=0 and s!=1:
            return False
      return test01(E) and test01(M)
```

2.3 Conversion binaire décimal

Dans cette partie l'objectif est de convertir une représentation s,E,M en flottant x.

Question 12. Écrire une fonction `sommebin(L)` d'argument `L` une liste de longueur n qui calcule la quantité suivante :

$$\sum_{k=0}^{k=n-1} L[k] \times 2^{n-1-k}$$

Un parcours par indice est ici adapté :

```
def sommebin(L):
      s=0
      n=len(L)
      for k in range(n):
            s=s+L[k]*2**(n-1-k)
      return s
```

On pourra se servir de cette fonction même si on n'a pas su la coder dans la suite, et elle sera supposée juste par le correcteur pour les questions suivantes. Reprogrammer de manière inexacte une variante de cette fonction sera par contre pénalisé.

Question 13. Écrire une fonction `exposant(E)` qui prend en argument une liste `E` valide et représentant l'écriture binaire de l'exposant d'un nombre codé en double précision et qui renvoie l'entier correspondant à e dans l'écriture $x = (-1)^s 2^e (1+m)$

Il suffit de se servir de la question précédente et de décaler la valeur obtenue de 1023 :

```
def exposant(E):
      return sommebin(E)-1023
```

Question 14. Écrire une fonction `mantisse(M)` qui prend en argument une liste `M` valide et représentant l'écriture binaire de la mantisse d'un nombre codé en double précision et qui renvoie le flottant m dans l'écriture $x = (-1)^s 2^e (1+m)$

Sur le même principe, mais on prendra garde à multiplier le résultat par 2^{-52} :

```
def mantisse(M):
      return sommebin(M)*2**-52
```

Question 15. Écrire une fonction `sEMtofloat(s,E,M)` qui prend en arguments les variables s,E et M, définies plus haut, supposées valides, et qui renvoie le flottant x qu'elles codent.

En reprenant les questions précédentes et la définition de x :

```
def sEMtofloat(s,E,M):
      return (-1)**s * 2**exposant(E) * (1+mantisse(M))
```

2.4 Conversion décimal binaire sans problème d'arrondi

Cette sous-partie s'intéresse au codage d'un décimal x non nul écrit en base 10 comme un flottant double précision, codé à l'aide des variables `s`, `E` et `M`. On ne s'intéresse pas pour le moment au problème d'arrondi de la mantisse.

1. La première étape consiste à chercher la valeur de la variable `s`. Pour cela on regarde le signe de x. On travaille à partir de là sur y la valeur absolue de x, qui peut être obtenue en Python à l'aide de `abs(x)`.

2. La seconde étape consiste à chercher la valeur de l'exposant e, puis à la transformer en liste `E`. Pour cela, on pourra utiliser la formule suivante :

   $$e = 1023 + \lfloor \log_2(y) \rfloor$$

   La fonction $\log_2$ désigne le logarithme en base 2, elle peut être obtenue en Python avec la fonction `log` de la bibliothèque `numpy` puis en utilisant `log(y,2)` (on prendra garde à la syntaxe utilisée pour importer des fonctions depuis un module).
   $\lfloor a \rfloor$ désigne la partie entière du nombre a.
   L'entier e ainsi obtenu est converti en une liste représentant ses bits codés dans une liste de 11 éléments et correspondant à son écriture en binaire non signé sur 11 bits.

3. Dans la dernière étape on travaille sur le décimal $z = \dfrac{y}{2^{\lfloor \log_2(y) \rfloor}}$. On a nécessairement $1 \leqslant z < 2$.
   L'algorithme suivant permet ensuite d'obtenir les éléments de `M` :
   - affecter z-1 à z
   - créer une liste `M` vide
   - pour `i` allant de 1 à 52 faire :
   - multiplier z par 2
   - si $z \geqslant 1$ ajouter 1 à la fin de `M` et affecter z-1 à z ; sinon ajouter 0 à la fin de `M`
   - fin du pour
   - renvoyer M

Question 16. Écrire une fonction `variable_s(x)` d'argument un décimal x qui renvoie la valeur de s, comme décrit dans la première étape de cet algorithme.

x est supposé non nul ... une structure `if..else..` est possible ici :

```
def variable_s(x):
      if x>0:
            return 1
      else:
            return 0
```

On peut bien sûr être plus malin :

```
def variable_s(x):
      return x//abs(x)
```

Question 17. Écrire une fonction `exposant(x)` d'argument un décimal x de signe quelconque, qui calcule $y = |x|$ et renvoie la valeur de e calculée à l'aide de la formule donnée dans la deuxième partie de cet algorithme.

Il suffit d'importer une fonction logarithme depuis une bibliothèque (par exemple `numpy` ou `math`) et de suivre scrupuleusement l'algorithme décrit. On notera que l'énoncé comporte une erreur subtile : il y a deux fonctions appelées exposant !...

```
import numpy as np

def exposant(x):
      y=abs(x)
      e=1023+int(np.log(y,2))
      return e
```

Question 18. En se servant de la fonction exposant codée à la question précédente, qu'on pourra utiliser même si on a pas su répondre à cette question, écrire une fonction `listeE` qui prend en argument un décimal x, et qui renvoie la liste `E` comme décrite à la seconde étape de l'algorithme.

Nous noterons dans cette question `exposant` la fonction exposant de la question précédente... et non la première fonction codée. Nous utiliserons par ailleurs l'algorithme décrit en début d'année de codage d'un entier non signé en binaire.

```
def listeE(x):
      e=exposant(x)
      L=[]
      for i in range(11):
            L.append(e//2**(10-i))
            e=e%2**(10-i)
      return L
```

Question 19. Écrire une fonction `listeM` d'argument un décimal x qui renvoie la liste `M`, telle que décrite dans la troisième étape de l'algorithme.

```
def listeM(x):
      y=abs(x)
      z=y/(2**int(log(y,2)))
      z=z-1
      M=[]
      for i in range(1,53):
            z=z*2
            if z>=1:
                  M.append(1)
                  z=z-1
            else:
                  M.append(0)
      return M
```

3 Bits de parité et codes de Hamming (extrait banque PT 2015)

Un signal transmis peut comporter des erreurs dans les données transmises ; il est indispensable de détecter ces erreurs, et, dans la mesure du possible, de les corriger sans nécessiter une nouvelle transmission.

3.1 Bits de parité

Une technique très simple pour s'assurer qu'une donnée binaire sera correctement lue par son récepteur est de lui adjoindre un bit de parité, égal, par définition, à :
- 0 si la donnée comporte un nombre pair de 1 (et donc si ses bits sont de somme paire) ;
- 1 si la donnée comporte un nombre impair de 1 (et donc si ses bits sont de somme impaire)

Après réception de la donnée, le récepteur recalcule le bit de parité, et le compare à celui que l'émetteur lui a adressé. Si la donnée n'a pas été altérée lors de la transmission, alors les deux bits de parité sont forcément identiques.

Question 20. Donner les bits de parité associés aux représentations binaires des entiers 5, 16 et 37.

Il suffit d'écrire ces nombres en binaire (non signé) et de compter le nombre de bits égaux à 1 :

1. 5 s'écrit 101 en binaire, il y a deux "1" donc le bit de parité de 5 est 0 ;
2. 16 s'écrit 10000 en binaire, il y a un "1" donc le bit de parité de 16 est 1 ;
3. 37 s'écrit 100101 en binaire, il y a trois "1" donc le bit de parité de 37 est 1.

Question 21. Écrire une fonction `parite(bits)` qui prend en argument une liste `bits`, constituée d'entiers valant 0 ou 1, et retournant l'entier 0 ou 1 correspondant à son bit de parité.

En se servant de la fonction `sum` de Python, on peut écrire :

```
def parite(bits):
    return sum(bits)%2
```

À la main, on somme les éléments de `bits` et on renvoie le reste de la division euclidienne de la somme par 2 (plutôt qu'une structure `if ... else ...` très lourde ici) :

```
def parite(bits):
    s=0
    for x in bits:
        s=s+x
    return s%2
```

3.2 Code de Hamming

Le code de Hamming est un exemple d'utilisation de bits de parité pour détecter et corriger des erreurs. Nous nous intéressons ici au code (7,4), ainsi appelé car il consiste à joindre trois bits de parité à quatre bits de données, ce qui donne un message de longueur totale de sept bits. Ces trois bits de parité sont ainsi définis : si la donnée s'écrit (d_1, d_2, d_3, d_4) avec $d_i = 0$ ou 1 alors :

- p_1 est le bit de parité du triplet (d_1, d_2, d_4)
- p_2 est le bit de parité du triplet (d_1, d_3, d_4)
- p_3 est le bit de parité du triplet (d_2, d_3, d_4)

Le message encodé, que l'on transmet, s'écrit $(p_1, p_2, d_1, p_3, d_2, d_3, d_4)$

Question 22. Écrire une fonction `encode_hamming(donnees)` qui prend en argument une liste `donnees` de quatre bits (constitués d'entiers 0 ou 1) et qui retourne une liste de bits contenant le message encodé. On utilisera la fonction précédente.

```
def encode_hamming(donnees):
    d = donnees
    return [parite([d[0],d[1],d[3]]), parite([d[0],d[2],d[3]]), d[0], parite(d[1:])]+d[1:]
```

Le contrôle après réception d'un message ainsi encodé est relativement simple. On pourrait naturellement recalculer les trois bits de parité de la donnée et les comparer aux valeurs transmises, mais la technique proposée par Hamming est de calculer les *trois bits de contrôle* suivants, notés (c_1, c_2, c_3) à partir du *message complet* (données et bits supplémentaires) noté $(m_1, ..., m_7)$:
- c_1 est le bit de parité de l'ensemble (m_4, m_5, m_6, m_7)

– c_2 est le bit de parité de l'ensemble (m_2, m_3, m_6, m_7)
– c_3 est le bit de parité de l'ensemble (m_1, m_3, m_5, m_7)

On montre que si le message a été bien encodé selon les règles précédentes et n'a pas été altéré, alors les trois bits de contrôle doivent être nuls ; si ce n'est pas le cas, il y a eu une erreur. L'intérêt de la technique de Hamming est que, dans le cas particulier où l'erreur est unique, le mot de contrôle donne la représentation binaire de la position de cette erreur *en numérotant à partir de 1.* Par exemple, si $(c_1, c_2, c_3) = (0, 1, 1)$ alors l'erreur porte sur le troisième bit du message. Il suffit ainsi d'inverser ce bit (le mettre à 1 si il est à 0 et inversement) pour corriger l'erreur.

La donnée décodée est alors constituée des quatre bits (d_1, d_2, d_3, d_4) qui se trouvent respectivement en positions 3, 5, 6 et 7 (toujours numérotés à partir de 1) conformément à la description de l'encodage donnée ci-dessus.

Question 23. Écrire une fonction `decode_hamming(message)` qui prend en argument une liste `message` de sept bits (constitués d'entiers 0 ou 1) et retournant une liste de quatre bits contenant la donnée décodée. En cas d'erreur, on affichera à l'écran un avertissement indiquant la position du bit affecté et on effectuera la correction. On supposera dans cette question que s'il y a une erreur, alors elle est unique.

```
def decode_hamming(message):
      m = message
      c = parite([m[0],m[2],m[4],m[6]]) + 2*parite([m[1],m[2],m[5],m[6]]) + 4*parite(m[3:]))
      if c != 0:
            print("le bit d'indice "+str(c)+" a été mal transmis")
            cor = m.pop(c)
            m.insert(c, 1-cor)
      return [m[2]] + m[4:]
```

Question 24. Déterminer le codage de Hamming de la donnée `1011`, puis la donnée décodée par l'algorithme dans l'hypothèse où les deux premiers bits du message codé ont été incorrectement transmis. Quel a été l'effet de la « correction » sur la donnée dans ce cas ?

En tapant dans la console `encode_hamming([1,1,0,1])` on obtient : `[1, 0, 1, 0, 1, 0, 1]`
Pour trouver le message obtenu en décodant un message où les deux premiers bits ont été altérés, on tape dans la console :
`encode_hamming([0, 1, 1, 0, 1, 0, 1])` on obtient :
`le bit d'indice 3 a été mal transmis [1, 1, 0, 1]`

Question 25. Sans coder, proposer un moyen simple de différencier une erreur double d'une erreur unique au moyen d'un bit de parité supplémentaire, et expliquer comment cela permet d'éviter le problème mis en évidence à la question précédente. On s'appuiera sur les méthodes introduites dans cette partie. On ne demande pas d'essayer de corriger la double erreur.

En cas d'une double erreur, le code détecte une anomalie. Cependant le syndrome propose un chiffre binaire correspondant toujours à une troisième colonne différente des deux précédentes. Ainsi une troisième erreur est ajoutée, car le code ne dispose d'aucun moyen pour différencier une simple d'une double erreur. Pour pallier cet état de fait, ainsi que pour obtenir un code de dimension exactement égale à un octet, un huitième bit est souvent ajouté, il correspond à la parité des sept bits précédents. Une deuxième erreur est ainsi détectée, elle ne peut être corrigée, en revanche l'information d'une double erreur est disponible remplaçant une tentative maladroite, par une demande de retransmission.

Le décodage se fait de la manière suivante :
– S'il n'y a aucune erreur le syndrome est nul.
– Si une erreur s'est produite sur les sept premiers bits, les trois premiers éléments du syndrome donnent la position de l'erreur. L'existence d'un nombre d'erreur impair est confirmé par le huitième bit.
– Si deux erreurs se sont produites, les trois premiers éléments du syndrome ne sont pas tous nuls et le huitième bit indique une parité exact, signal d'un nombre pair d'erreurs. Une retransmission est nécessaire.
– Si une erreur s'est produite sur le huitième bit, l'absence d'erreur sur les trois premiers éléments du syndrome permet de localiser l'erreur et le message est validé.