I Banque PT 2015 - IPT partie info

Marc REZZOUK (marc.rezzouk@free.fr)

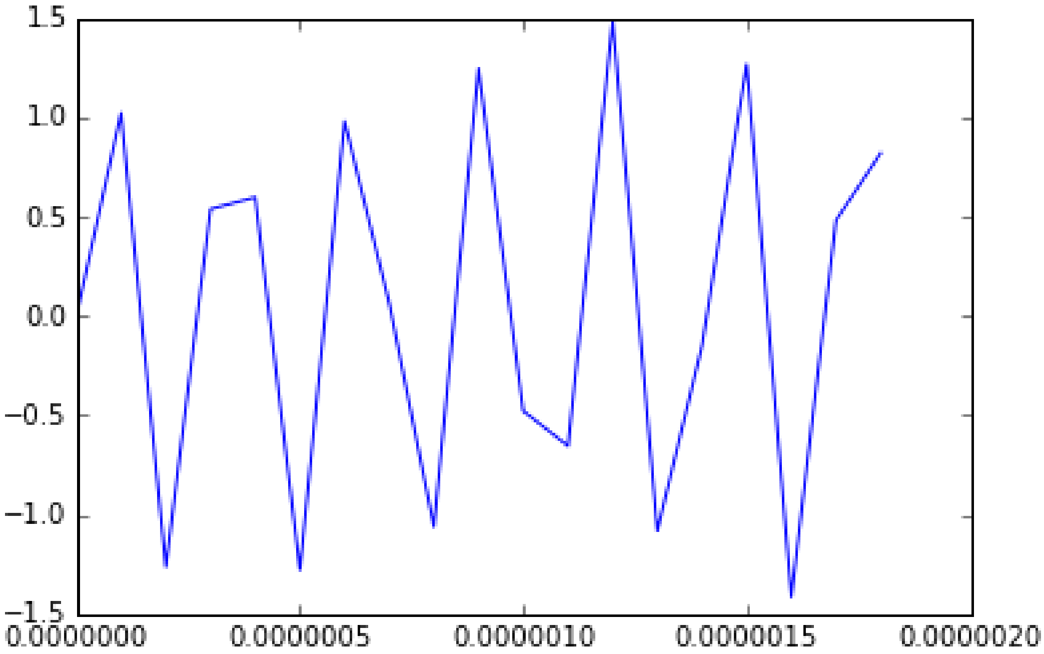
Q1

```
In [1]: from math import ceil

        def init_T(Tmax, dt):
            n = ceil(Tmax / dt)
            T = [i * dt for i in range(n+1)]
            return T

        # version 2
        def init_T2(Tmax, dt):
            T = []
            a = 0.
            while a < Tmax+dt:
                T.append(a)
                a += dt
            return T

In [2]: print(init_T(5.5, 1.))
        print(init_T2(5.5, 1.))

[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0]
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0]
```

Q2

```
In [3]: from math import sin, pi

        EMIN = 1.3
        EMAX = 1.5

        def e(t, f):
            duree = 16 / f
            if t <= duree:
                return EMIN
            elif t <= 2 * duree:
                return EMAX
            elif t <= 3 * duree:
                return EMIN
            else:
                return 0.

        def init_E(T, f):
            return [e(t, f) * sin(2 * pi * f * t) for t in T]

In [4]: %matplotlib inline
        import matplotlib.pyplot as plt

        f = 13.56 * 1e6
        # N = 7
        # dt = 1 / f / N  # découpage en N morceaux sur une période

        dt = 100 * 1e-9    # 100 ns
```

```
T = init_T(1.5 * 16 / f, dt)
E = init_E(T, f)

plt.plot(T, E)
plt.show()
```



Q3

On approxime la dérivée par $\dfrac{\mathrm{d}s}{\mathrm{d}t}(t_i) \approx \dfrac{s(t_{i+1}) - s(t_i)}{(t_{i+1} - t_i)}$

donc $s(t_{i+1}) = \left(1 - \frac{\Delta t}{\tau}\right) s(t_i)$

Q4

Il faut lire entre les lignes… L'approximation "stricte" précédente n'est pas exploitable car on devrait connaître $s(t_{i+2})$. On prend donc une autre approximation avec les données dont on dispose.

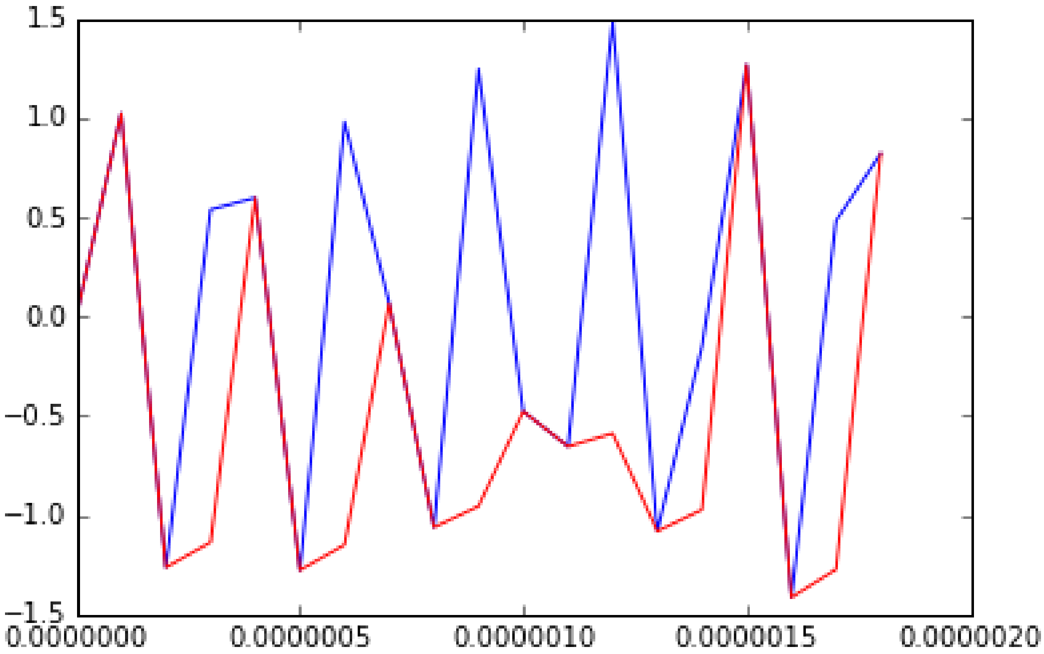
$\dfrac{\mathrm{d}s}{\mathrm{d}t}(t_{i+1}) \approx \dfrac{s(t_{i+1}) - s(t_i)}{(t_{i+1} - t_i)}$

Q5

je ne comprends pas gd chose à leur histoire… C'est vraiment très mal écrit. J'ai dû m'y reprendre à plusieurs fois…

```
In [5]: def solve(T, E, tau):
            S = [E[0]]
            n = len(T)
            diodepass = True
            for i in range(n-1):
                if diodepass:
                    s = E[i+1]
                    b = (s - S[i]) / (T[i+1] - T[i]) + s / tau  # euler arrière
                    if b <= 0:
                        diodepass = False
                else:
                    s = (1 - (T[i+1] - T[i]) / tau) * S[i]  # euler explicite
                    if s <= E[i+1]:
                        diodepass = True

                S.append(s)
            return S
```

```
In [6]: tau = 1e-6     # 1 micro seconde
        S = solve(T, E, tau)

        ss = 30
        plt.plot(T[:ss], E[:ss], 'b')
        plt.plot(T[:ss], S[:ss], 'r')
        plt.show()
```



Q6
En regardant l'échelle des abscisses, il est clair que la figure 3 est celle où la subdivision est la plus grossière, donc de pas de temps 100 ns (ligne polygonale grossière). Sur la figure 4, le tracé reste assez grossier (pour e(t) comme pour s(t)), donc, par élimination, la valeur du pas de temps est 10 ns. Enfin, il reste la figure 5, pas de temps 1 ns, plus précise.
Q7
Si la subdivision est trop grossière, l'approximation dans la formule d'Euler l'est également d'où une simulation assez éloignée de la réalité. En construisant s(t), on cherche à tasser les valeurs autour de Emin et Emax et à repérer ainsi si on a un bit à 1 ou à 0. On voit que seule la figure 5 (1 ns) permet ce répérage en fixant par exemple un seuil à (Emin + Emax)/2.
Q8
J'ai l'impression d'avoir répondu à la question précédente... Il faut un pas de temps très inférieur à la période (1/f) du signal d'entrée.
Q9
Un peu de python (non demandé) en rab...

```
In [7]: def decompbin(n):
            ''' décompose en binaire sous forme de liste '''
            L = []
            while n != 0:
                n, r = n // 2, n % 2
                L.append(r)
            L.reverse()   # ou L = L[::-1]
            return L
```

```
In [8]: print(decompbin(5), decompbin(16), decompbin(37))

[1, 0, 1] [1, 0, 0, 0, 0] [1, 0, 0, 1, 0, 1]
```

Donc le bit de parité est respectivement 0, 1, 1.
Q10

```
In [9]: def parite(bits):
            return sum(bits) % 2

In [10]: for a in [5, 16, 37]:
             print(parite(decompbin(a)))
```

```
0
1
1
```

Q11
Si deux bits sont altérés l'erreur est non détectable. Si seulement un bit est altéré, il n'y a pas moyen de détecter où se trouve le bit altéré (sans rajouter d'autres bits de contrôle).
Q12

```
In [11]: def encode_hamming(donnee):
             d1, d2, d3, d4 = donnee   # pour pas se prendre la tête
             p1 = parite([d1, d2, d4])
             p2 = parite([d1, d3, d4])
             p3 = parite([d2, d3, d4])
             return [p1, p2, d1, p3, d2, d3, d4]
```

Q13

```
In [12]: def decode_hamming(message):
             m1, m2, m3, m4, m5, m6, m7 = message   # idem pour la lisibilité
             c1 = parite([m4, m5, m6, m7])
             c2 = parite([m2, m3, m6, m7])
             c3 = parite([m1, m3, m5, m7])

             if [c1, c2, c3] != [0, 0, 0]:
                 numbit = 4*c1 + 2*c2 + c3
                 print("Attention il y a une erreur dans le message transmis")
                 print("Je corrige le bit n°", numbit)
                 m = message.copy()
                 m[numbit-1] = 1 - m[numbit-1]
                 m1, m2, m3, m4, m5, m6, m7 = m # bourrin mais mettre des [] partout c'est lourd...

             return [m3, m5, m6, m7]
```

Q14

```
In [13]: print("message codé", encode_hamming([1, 0, 1, 1]))
         print("message décodé", decode_hamming(encode_hamming([1, 0, 1, 1])))
```

```
message codé [0, 1, 1, 0, 0, 1, 1]
message décodé [1, 0, 1, 1]
```

Avec les deux premiers bits (poids faibles ? l'énoncé est imprécis)

```
In [14]: print("message décodé", decode_hamming([0, 1, 1, 0, 0, 0, 0]))
```

```
Attention il y a une erreur dans le message transmis
Je corrige le bit n° 1
message décodé [1, 0, 0, 0]
```

si poids fort

```
In [15]: print("message décodé", decode_hamming([1, 0, 1, 0, 0, 1, 1]))
```

```
Attention il y a une erreur dans le message transmis
Je corrige le bit n° 3
message décodé [0, 0, 1, 1]
```

Effet mauvais ? Que répondre ?
Q15
On demande de repérer une double erreur. Cette question me semble trop difficile si on ne connait pas déjà la réponse…
On rajoute un huitième bit de parité sur l'ensemble des 7 bits.
On suppose qu'il n'y a *jamais* plus de deux erreurs.
Si c1, c2, c3 = 0, 0, 0 alors OK, on peut décoder, sinon :

- s'il y a une seule erreur sur les sept premiers bits, on la corrige, le 8e bit de parité est mauvais.
- s'il y a une erreur sur les sept premiers bits et un erreur sur le bit de parité rajouté, on sait qu'il y a une seule erreur sur les sept premiers bits et on corrige tranquillement.
- s'il y a deux erreurs dans les sept premiers bits, **elle est détectée par c1, c2, c3** (information à connaître donc question trop difficile) mais que partiellement corrigée, en revanche le 8e bit de parité est "bon", ce qui permet de détecté cette double erreur. On retransmet alors l'octet (on ne peut pas corriger la double erreur)

En résumé,
si c1, c2, c3 = 0, 0, 0, ok, on décode. Sinon - si le 8e bit est mauvais, on n'a qu'une erreur sur les sept premiers bits. On sait corriger. - si le 8e bit est bon, c'est qu'on a une double erreur. On ne pas peut pas corriger. Il faut retransmettre l'information.
Q16
id_titre est du type int
zones est une liste de deux entiers donc est de type list
date_fin est une liste de trois entiers donc de type list
Q17

```
In [16]: ### pour le test
         lignes = '''\
         49987654
         1, 3, 2015-08-31
         2014-10-29, 08:34:15, 4568
         2014-10-28, 20:21:48, 365
         2014-10-28, 18:47:54, 987
         '''
         lignes = lignes.split('\n')

In [17]: passages = []

         for i in range(2, 5):
             passage = lignes[i].rstrip('\n').split(',')
             date = passage[0].split('-')
             heure = passage[1].split(':')
             ident = passage[2]

             lignepass = [int(date[0]), int(date[1]), int(date[2]),
                          int(heure[0]), int(heure[1]), int(heure[2]), int(ident)]
             passages.append(lignepass)

In [18]: passages

Out[18]: [[2014, 10, 29, 8, 34, 15, 4568],
          [2014, 10, 28, 20, 21, 48, 365],
          [2014, 10, 28, 18, 47, 54, 987]]
```

Q18

```
In [19]: def estAvant(date1, date2):
             annee = date2[0] - date1[0]
             mois = date2[1] - date1[1]
             jour = date2[2] - date1[2]

             return annee > 0 or (annee == 0 and mois > 0) \
                     or (annee == 0 and mois == 0 and jour >= 0)
```

Q19

```
In [20]: def nbSecondesEntre(heure1, heure2):
             return (heure1[0] - heure2[0])*3600 + \
                     (heure1[1] - heure2[1])*60 + (heure1[2] - heure2[2])
```

Q20

```
In [21]: ## on suppose les variables du lecteur globales
         def testPassage(id_titre, zones, date_fin):
             passage = True
             if id_titre in Liste_noire:
                 passage = False

             if Zone < zones[0] or Zone > zones[1]:
                 passage = False

             aujourdhui = Maintenant[:3]
             if estAvant(date_fin, aujourdhui):
                 passage = False

             Heure_lect = Maintenant[3:]
             for i in range(3):
                 if passages[i][6] == Id_Point:  # si même point de passage
                     heure = passages[i][3:6]
                     if nbSecondesEntre(Heure_lect, heure) < 450:
                         passage = False

             return passage
```

Q21
Les comparaisons sur les dates sont à vérifier suivant le SGBD utilisé...

```
In []: SELECT date, heure FROM passages JOIN points on id_point = id
       WHERE ligne = 1 and date >= '2014-07-01' AND date <= '2014-08-31';
```

Q22

```
In []: SELECT COUNT(*) FROM passages JOIN titres AS t ON id_titre = t.id
       JOIN points AS p ON id_point = p.id
       WHERE date = '2014-12-31' AND (t.zone_min > p.zone OR t.zone_max < p.zone);
```