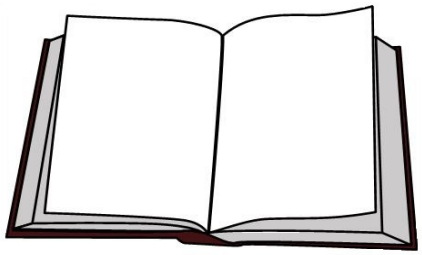
Manuel d'utilisation du programme[1] d'assemblage de l'ADN



Ce manuel a été réalisé par : Julie **PRATX**
Dans le cadre d' : un projet en Omiques et Bioinformatique
Le : le 8 décembre 2019

1 Programme réalisé en Python 3.7.3 (data_ADN_part1.py & data_ADN_part2.py)

Partie I

La partie I de ce programme a pour but principal de générer les données aléatoirement pour être utilisées dans la partie II du programme.

I. Les variables et les import

```
24 import random
25 #                  Les listes et dictionnaires
26 nucleic = ["A","T","G","C"]
27 adaptateurs = []
28 #                   Pour le brin sens 5'-> 3'
29 readBS = []
30 valeur_q = []
31 valeur_qualité = []
32 dico_readsS = {}
33 #                  Pour le brin non sens 3' <- 5'
34 readBNS =[]
35 valeur_qC = []
36 valeur_qualitéC = []
37 dico_readsNS = {}
```

import random est un module qui permet de générer les données aléatoirement sans devoir ré-écrire toute la fonction.

nucleic est une liste ([]) qui contient les 4 nucléotides :

- A pour Adénine
- T pour Thymine
- C pour Cytosine
- G pour Guanine

adaptateurs est une liste ([]) qui sert à stocker les adaptateurs générés par le programme.

1. Brin sens (BS) (5’ → 3’)

readBS est une liste ([]) qui stocke les reads générés à partir de la séquence d’ADN.

valeur_q est une liste ([]) qui stocke les valeurs de qualité générées dans la procédure *def valeurQBS(read,nb_read)* pour chaque base contenue dans un read.

valeur_qualité voir valeur_q

dico_readsS est un dictionnaire ({}) qui stocke l’index des reads (0, 1, 2, 3, etc.…), les reads et leur valeur de qualité associée.
Il est organisé sous cette forme : {index du read : [[Read], longueur du read, [valeur de qualité]]}

2. Brin non sens (BNS) (3’ ← 5’)

readBNS est une liste ([]) qui stocke les reads générés à partir de la séquence d’ADN.

valeur_qC est une liste ([]) qui stocke les valeurs de qualité générées dans la procédure *def valeurQBNS(read,nb_read)* pour chaque base contenue dans un read.

valeur_qualitéC voir valeur_qC

dico_readsNS est un dictionnaire ({}) qui stocke l'index des reads (0, 1, 2, 3, etc.…), les reads et leur valeur de qualité associée.
Il est organisé sous cette forme : {index du read : [[Read], longueur du read, [valeur de qualité]]}

II. La fonction random

```
"""
Cette procédure prend pour argument num (nombre de caractère que l'on veut générer)
et alphabet (type de caractère que l'on veut générer).
Son but est de nous créer une séquence de nucléotides aléatoirement.
"""
print();print("1 - Séquence initiale d'ADN générée aléatoirement", "\n")
def randseq(num,alphabet):
    seq = ''
    while len(seq) < num:
        seq += nucleic[random.randint(0, len(nucleic) - 1)]
    return seq
# test
seq = randseq(80, nucleic)
print("Séquence initiale ADN :")
print("5'-", seq, "-3'", "\n")
```

Pour modifier la taille de la séquence, on change le nombre qui se trouve entre parenthèse.
Par exemple, pour une longueur de 200 nucléotides, on va écrire **randseq(200, nucleic)**.

III. Générer les séquences complémentaires

```
##############################################################################
#                          Brin complémentaire                               #
##############################################################################
"""
Cette procédure prend pour argument sequence (qui correspond à notre séquence
généré aléatoirement à la procédure précédente).
Son but est de générer le brin d'ADN complémentaire qui sera utilisé dans la
partie 2 du programme.
"""
def brincomplementaire(sequence):
    seqC = ''
    for i in sequence:
        if i == "A":
            seqC += "T"
        if i == "T":
            seqC += "A"
        if i == "G":
            seqC += "C"
        if i == "C":
            seqC += "G"
    return seqC

seqC = brincomplementaire(seq)
print("Séquence initiale complémentaire ADN :")
print("3'-", seqC, "-5'", "\n")
```

Cette procédure génère automatiquement le brin non sens (reverse) de la séquence que l'on a généré précédemment (si cette séquence est passée en paramètre).
Ici, i parcourt la séquence et à chaque fois qu'il rencontre un « A » il écrit un « T » , pour un « G » il écrit un « C » et inversement. La boucle continue à tourner tant qu'il y a de caractères à traiter.

A la fin, la chaîne de caractères stockées dans la variable seqC est retournée pour pouvoir être utilisée pour exécuter la procédure.

IV. Création des reads pour le brin sens

```
81 ##############################################################################
82 #                            Reads brin sens                                 #
83 ##############################################################################
84 """
85 Cette procédure (pour le brin sens) prend pour argument seq (la séqeunce a
86 transformer en reads) et nb_read (le nombre de read que l'on veut).
87 Dans cette version, les reads sont de mêmes tailles et le décalage est de 1
88 nucléotide vers la droite.
89 """
90 #print();print("2 - Reads", "\n")
91 def readsBS(seq,nb_read):
92     for i in range (0,nb_read):
93         s = seq[i:i+20]
94         readBS.append(s)
95     return readBS
96
97 readsBS(seq, 10)
```

Pour cette version du programme, les reads créés sont de taille identique. Dans cet exemple, la taille est fixée à 20. On peut modifier cette longueur en changeant le nombre dans la procédure.
Par exemple, pour générer des reads de 100 nucléotides, on va écrire **seq[i:i+100]**.
On peut également modifier le nombre de décalage pour la segmentation du read. Ici, on décale d'un nucléotide vers la droite. Pour décaler de 2 nucléotides vers la droite, on peut écrire **seq[i+1:i+20]**.

V. Génération des valeurs de qualité pour le brin sens

```
 98 ##############################################################################
 99 #                Valeur de qualité des reads brin sens                       #
100 ##############################################################################
101 """
102 Cette procédure (pour le brin sens) prend pour argument read (la séquence
103 générée précédemment) et nb_read (définit dans la procédure précédente).
104 """
105 #print();print("3 - Valeur de qualité", "\n")
106 def valeurQBS(read,nb_read):
107     for i in range (nb_read):
108         for j in range (nb_read):
109             valQ = random.randint(0,100)
110             valeur_qualité.append(valQ/100)
111     for k in range(nb_read):
112         valeur_q.append(valeur_qualité[20*k:20*k+20])
113     return valeur_q
114
115 valeurQBS(readsBS, 20)
116 print("Liste des valeurs de qualité :", valeur_qualité, "\n")
```

La première boucle de la procédure sert à générer des valeurs de qualité ayant 2 chiffres après la virgule. Pour se faire, on choisit un intervalle entre 0 et 100 pour la fonction random.randint et ensuite on divise ce résultat par 100. Si on veut 3 chiffres après la virgule, on pose un intervalle de 0 à 1000 et on divise par 1000.
La deuxième boucle sert attribuer une valeur de qualité pour chaque nucléotide de chaque read. Ici, la taille du read est fixée à 20 nucléotides. Si on veut des reads de 40 nucléotides, on doit remplacer

les 20 par des 40 comme ceci : **valeur_qualité[40*k:40*k+40]** et dans l'appel de la procédure **valeurQBS(readsBS, 40)**.

VI. Création des reads pour le brin non sens

```
118 ##########################################################################################
119 #                              Reads brin non sens                                       #
120 ##########################################################################################
121 """
122 Cette procédure (pour le brin non sens) prend pour argument seq (la séqeunce a
123 transformer en reads) et nb_read (le nombre de read que l'on veut).
124 Dans cette version, les reads sont de mêmes tailles et le décalage est de 1
125 nucléotide vers la droite.
126 """
127 #print();print("2 - Reads", "\n")
128 def readsBNS(seq,nb_read):
129     for i in range (0,nb_read):
130         s = seq[i:i+20]
131         readBNS.append(s)
132     return readBNS
133 #test
134 #readsBNS(seqC, 10)
```

Voir *IV. Création des reads pour le brin sens*

VII. Génération des valeurs de qualité pour le brin non sens

```
135 ##########################################################################################
136 #              Valeur de qualité des reads brin non sens                                 #
137 ##########################################################################################
138 """
139 Cette procédure (pour le brin non sens) prend pour argument read (la séquence
140 générée précédemment) et nb_read (définit dans la procédure précédente).
141 """
142 #print();print("3 - Valeur de qualité", "\n")
143 def valeurQBNS(read,nb_read):
144     for i in range (nb_read):
145         for j in range (nb_read):
146             valQ = random.randint(0,100)
147             valeur_qualitéC.append(valQ/100)
148     for k in range(nb_read):
149         valeur_qC.append(valeur_qualitéC[20*k:20*k+20])
150     return valeur_qC
151
152 #test
153 #valeurQBNS(readBNS, 20)
154 #print("Liste des valeurs de qualité :", valeur_qualitéC, "\n")
```

Voir *V. Génération des valeurs de qualité pour le brin sens*

VIII. Création du dictionnaire pour les index, les reads et les valeurs de qualité pour le brin sens

```
155 ##############################################################################
156 #                    Dico brin sens 5' -> 3'                                 #
157 ##############################################################################
158 """
159 Cette procédure pour le brin sens prend pour argument seq (brin sens) et
160 nb_read (le nombre de clé du dictionnaire).
161 Son but est de stocker les index, les reads et leur valeur de qualité dans un
162 dictionnaire.
163 """
164 print();print("2 - Dico brin sens 5' -> 3'", "\n")
165 def dicoReadsS(seq,nb_read):
166     r1 = readsBS (seq,nb_read)
167     v1 = valeurQBS (r1,nb_read)
168     id = []
169     for i in range(nb_read):
170         id = i
171         dico_readsS[id] = [0,0,0]
172     for id, i in zip(dico_readsS.keys(),range(nb_read)):
173         dico_readsS[id][0] = [r1[i]]
174         dico_readsS[id][1] = len(r1[i])
175         dico_readsS[id][2] = v1[i]
176     print("Dico brin sens 5'-> 3' :", dico_readsS, "\n") # affiche le dico des
177 #    reads du brin sens
178
179 dicoReadsS(seq, 65)
```

Cette procédure permet de créer un dictionnaire pour stocker les index des reads, les reads et leurs valeurs de qualité.
Rappelons qu'un dictionnaire est composé de « clés » et de « valeurs » et ayant cette nomenclature : {clé : valeur}.
Les index sont, en réalité, les clés du dictionnaire. La clé « 0 » est l'index « 0 » du read et de ces valeurs de qualité. La première boucle de la procédure sert à mettre en forme le dictionnaire. En effet, dico_readsS[2][id] sera les clés des différentes entrées du dictionnaire et [0,0,0] sera la disposition des différentes valeurs du dictionnaire.
La deuxième boucle sert à définir comment seront rangées les différentes valeurs que l'on doit stocker dans le dictionnaire.

- En position 0 : dico_readsS[id][0] : le read
- En position 1 : dico_readsS[id][1] : la longueur du read
- En position 2 : dico_readsS[id][2] : les valeurs de qualité

A l'affichage, le dictionnaire ressemble à : {index:[[Read], longueur du read, [valeur de qualité]], …. }

2 dico_readsS : dictionnaire pour les reads du brin sens (S)

IX. Création du dictionnaire pour les index, les reads et les valeurs de qualité pour le brin non sens

```
180 ##############################################################################
181 #                   Dico brin non sens 3' -> 5'                              #
182 ##############################################################################
183 """
184 Cette procédure pour le brin non sens prend pour argument seq (brin non sens) et
185 nb_read (le nombre de clé du dictionnaire).
186 Son but est de stocker les index, les reads et leur valeur de qualité dans un
187 dictionnaire.
188 """
189 print();print("2' - Dico brin non sens 3' -> 5'", "\n")
190 def dicoReadsNS(seq,nb_read):
191     r1 = readsBNS (seqC,nb_read)
192     v1 = valeurQBNS (r1,nb_read)
193     id = []
194     for i in range(nb_read):
195         id = i
196         dico_readsNS[id] = [0,0,0]
197     for id, i in zip(dico_readsNS.keys(),range(nb_read)):
198         dico_readsNS[id][0] = [r1[i]]
199         dico_readsNS[id][1] = len(r1[i])
200         dico_readsNS[id][2] = v1[i]
201     print("Dico brin non sens 3' -> 5' :", dico_readsNS) # affiche le dico des
202 #    reads du brin non sens
203
204 dicoReadsNS(seqC, 65) # seqC = séquence complémentaire
```

Voir *VIII. Création du dictionnaire pour les index, les reads et les valeurs de qualité pour le brin sens*

X. Génération de séquence « adaptateur »

```
205 ##############################################################################
206 #                              Adaptateur                                    #
207 ##############################################################################
208 """
209 Cette procédure prend pour argument nb (nombre d'adaptateur).
210 Son but est de générer une liste d'adaptateur qui sera utilisée dans la partie
211 2 du programme.
212 """
213 def adaptateursF(nb):
214     for i in range (1, nb+1):
215         ad = randseq(4, nucleic)
216         adaptateurs.append(ad)
217     return adaptateurs
218
219 #adaptateursF(10)
```

Cette procédure permet de générer une séquence nucléique stockée dans une liste qui va servir d'adaptateur dans la partie 2 du programme.
Ici, la taille de la séquence est fixée à 4 nucléotides. Pour avoir un séquence de taille supérieure, par exemple, 30 nucléotides, on doit écrire **randseq(30, nucleic)**. Pour choisir le nombre d'adaptateur que l'on veut créer, en paramètre de la fonction, on change le nombre écrit par défaut (ici 10). Par exemple, pour avoir 25 adaptateurs, on écrit **adaptateursF(25)** et ces 25 adaptateurs seront ajoutés à la liste adaptateurs.

XI. Création des fichiers de sauvegardes des données

```
220 ##################################################################################
221 #                        Sauvegarde dans un fichier                              #
222 ##################################################################################
223 """
224 Ces 5 procédures ont pour même arguments file (nom du fichier de sauvegarde des
225 données).
226 Leur but est de créer des fichiers texte pour stocker les dictionnaires / listes
227 générés dans la partie 1 du programme.
228 """
229 def saveDicoBS(file):
230     NomDuFichier = (file)
231     Fichier = open(NomDuFichier,'w') # Crée un fichier *.txt en écriture
232     Fichier.write(str(dico_readsS)) # Ecrit le dictionnaire dans le fichier
233     Fichier.close()
234 
235 saveDicoBS("Dictionnaire_BS_OBI.txt")
236 
237 def saveDicoBNS(file):
238     NomDuFichier = (file)
239     Fichier = open(NomDuFichier,'w') # Crée un fichier *.txt en écriture
240     Fichier.write(str(dico_readsNS)) # Ecrit le dictionnaire dans le fichier
241     Fichier.close()
242 
243 saveDicoBNS("Dictionnaire_BNS_OBI.txt")
244 
245 
246 def saveSeqIniBS(file):
247     NomDuFichier = (file)
248     Fichier = open(NomDuFichier,'w')
249     Fichier.write(seq)
250     Fichier.close()
251 
252 saveSeqIniBS("Séquence_ini_BS_OBI.txt")
253 
254 def saveSeqIniBNS(file):
255     NomDuFichier = (file)
256     Fichier = open(NomDuFichier,'w')
257     Fichier.write(seqC)
258     Fichier.close()
259 
260 saveSeqIniBNS("Séquence_ini_BNS_OBI.txt")
261 
262 def saveAdaptateurs(file):
263     NomDuFichier = (file)
264     Fichier = open(NomDuFichier,'w')
265     Fichier.write(str(adaptateurs))
266     Fichier.close()
267 
268 saveAdaptateurs("Adaptateurs_OBI.txt")
```

Toutes ces procédures sont identiques dans leur manière de fonctionner. Pour sauvegarder et conserver les données générées par le programme, on crée des fichiers (ici des fichiers texte). Toutes les données stockées doivent être sous forme de string[3]. Les dictionnaires et les listes doivent donc être converties pour être stockées dans les fichiers. Pour cela, on doit écrire *str* devant l'argument. Comme, par exemple, **Fichier.write(*str*(dico_readsS))**.

3 Chaîne de caractères

Partie II

La partie II de ce programme a pour but principal de traiter les données générées aléatoirement dans la partie I du programme.

I. Les variables et les imports

```
21 #                        Les imports
22 import ast
23 import numpy as np
24 #                Listes et dico des brins sens et non sens
25 seq_ini = [] # liste temporaire
26 adaptateurs = [] # liste des adaptateurs
27 #                   Pour le brin sens 5'-> 3'
28 BS = [] # Brin Sens (BS) liste temporaire
29 dico_BS = {} # dictionnaire des reads brin sens
30 seq = [] # séquence initiale d'ADN
31 #                   Pour le brin non sens 3' <- 5'
32 BNS = [] #Brin Non Sens (BNS) liste temporaire
33 seqC = [] # sequence initiale Complémentaire
34 dico_BNS = {} # dictionnaire des reads brin non sens
```

import ast sert à convertir les strings en listes ou dictionnaires.

import numpy as np permet d'effectuer des calculs numériques avec Python.

seq_ini est une liste ([]) qui stocke temporairement la séquence initiale d'ADN.

adaptateurs est une liste ([]) qui stocke les adaptateurs générés dans la partie I du programme.

BS (pour Brin Sens) est une liste ([]) temporaire utilisée dans des procédures.

dico_BS est un dictionnaire ({}) qui stocke les index, les reads et les valeurs de qualité du brin sens.

seq est une liste ([]) qui stocke en permanence la séquence initiale d'ADN pour le brin sens.

BNS (pour Brin Non Sens) est une liste ([]) temporaire utilisée dans des procédures.

seqC est une liste ([]) qui stocke en permanence la séquence initiale d'ADN pour le brin non sens.

dico_BNS est un dictionnaire ({}) qui stocke les index, les reads et les valeurs de qualité du brin non sens.

II . Chargement des sauvegardes

```
#print("               Chargement de la sauvegarde :", "\n")
# Les 2 dictionnaires contenant les reads et les val qualité des 2 brins d'ADN
"""
Ici, on charge tous les fichiers textes qui contiennent les données générées
dans la partie 1 du programme.
Ces données vont être traitées à l'aide de différents filtres.
"""
Fichier = open('Dictionnaire_BS_OBI.txt','r')
for sortie1 in Fichier:
    BS = sortie1
Fichier.close()
Fichier = open('Dictionnaire_BNS_OBI.txt','r')
for sortie1 in Fichier:
    BNS = sortie1
Fichier.close()

# Le double brin d'ADN
Fichier = open('Séquence_ini_BS_OBI.txt','r')
for sortie2 in Fichier:
    seq_ini = sortie2
    seq.append(seq_ini)
#    print("Séquence initiale brin sens :", "\n")
#    print("5'-", seq, "-3'", "\n")
Fichier.close()
Fichier = open('Séquence_ini_BNS_OBI.txt','r')
for sortie2 in Fichier:
    seq_ini = sortie2
    seqC.append(seq_ini)
#    print("Séquence initiale brin non sens :", "\n")
#    print("3'-", seqC, "-5'", "\n")
Fichier.close()

# Adaptateurs
Fichier = open('Adaptateurs_OBI.txt','r')
for sortie3 in Fichier:
    ADP = sortie3
Fichier.close()
```

Toutes ces procédures sont semblables dans leur fonctionnement. Elles sont toutes pour but de charger les données générées dans la partie I du programme. Toutes les sorties de fichier sont de type « string »

III. Mise en forme

```
"""
Dans cette partie, on transforme le type de sortie (string) en (dict) ou en
(list) pour pouvoir les traiter dans les procédures suivantes.
"""
# Retour dans le dico
dico_BS = ast.literal_eval(BS)
#print("Dictionnaire brin sens :", dico_BS, "\n")
dico_BNS = ast.literal_eval(BNS)
#print("Dictionnaire brin non sens :", dico_BNS, "\n")
# Retour dans une liste
adaptateurs = ast.literal_eval(ADP)
#print("Liste d'adaptateurs :", adaptateurs, "\n")
```

Ces quelques lignes de code servent à rendre le type d'origine aux sorties de la procédure précédente. En effet, **ast.literal_eval(BS)** va transformer le type « str[4] » en type « dict[5] » . De même, **ast.literal_eval(ADP)** va transformer le type « str » en type « list[6] ».

IV. Affichage du contenu du dictionnaire

```
"""
Cette procédure sert à afficher les différentes clés et valeurs contenus dans
les dictionnaires.
On a donc :
    - le dico_BS qui contient les reads et valeurs de qualité des Brins Sens
    (5' -> 3')
    - le dico_BNS qui contient les reads et valeurs de qualité des Brins Non
    Sens (3' <- 5')
"""
def LoadReadSet(file):
    for cle, valeur in dico_BS.items():
      #print("Reads + longueur + val qualité brin sens :", cle, valeur, "\n")
      return dico_BS
    for cle, valeur in dico_BNS.items():
      #print("Reads + longueur + val qualité brin non sens :", cle, valeur, "\n
      return dico_BNS

LoadReadSet(dico_BS) # pour les brins sens
print()
LoadReadSet(dico_BNS) # pour les brins non sens
print()
```

Cette procédure a pour but d'afficher ce que contient un dictionnaire. On passe en argument le dictionnaire à traiter. Les « print » servent juste à vérifier que tout s'affiche correctement.

4 string : chaîne de caractères
5 dictionnaire
6 liste

V. Filtrage par adaptateurs

```
"""
Cette procédure prends pour arguement ReadSet (à remplacer par le dictionnaire
à traiter) et Adaptateurs (à remplacer par une séquence de nucléotides).

Elle sert à trouver des nucléotides complémentaires au différents reads que
compose le dictionnaire sélectionné en argument et à les éliminer.
"""
def FiltreAdapateurs(ReadSet, Adaptateurs):
    for cle in ReadSet.keys():
        if Adaptateurs == ReadSet[cle][0][0][0:4]:
            # retourne le read emputé de ces 4 premiers nucleotides
            ReadSet[cle][0][0] = ReadSet[cle][0][0][4:]
            #  longueur du read après le passage du filtre
            ReadSet[cle][1] = ReadSet[cle][1] - 4
            # nouvelles valeurs de qualité
            ReadSet[cle][2] = ReadSet[cle][2][4:]
        if Adaptateurs == ReadSet[cle][0][0][16:20]:
            # retourne le read emputé de ces 4 derniers nucléotides
            ReadSet[cle][0][0] = ReadSet[cle][0][0][:-4]
            #  longueur du read après le passage du filtre
            ReadSet[cle][1] = ReadSet[cle][1] - 4
            # nouvelles valeurs de qualité
            ReadSet[cle][2] = ReadSet[cle][2][:-4]
    return ReadSet

print("Dico BS :", FiltreAdapateurs(dico_BS, 'ATGA'), "\n")
print("Dico BNS :", FiltreAdapateurs(dico_BNS, 'TACT'), "\n")
```

Cette procédure sert à réaliser un filtrage à partir d'un adaptateur que l'on aura mis en argument de la fonction. On choisit le dictionnaire à traiter et l'adaptateur à appliquer. Le résultat sera affiché à l'aide du « print ».

VI. Filtre par seuil de qualité

```
"""
Cette procédure prend pour argument ReadSet (vu dans la procédure précédente)
et SeuilQualité (élimine les nucléotides et leur valeur de qualité associée
si le seuil de qualité n'est pas atteint).
"""
def FiltreExtremites(ReadSet, SeuilQualite):
    for cle in ReadSet.keys():
        # retourne le read entier
        R = ReadSet[cle][0][0]
        valsuppr = []; new = []
        for i in range (len(R)):
            # toutes les valeurs qualité
            if ReadSet[cle][2][i] < SeuilQualite:
                valsuppr.append(i)
            else:
                new.append(ReadSet[cle][2][i])
        # nouvelle liste de nucléotides
        ReadSet[cle][0][0] = ''.join(R[j] for j in range(len(R)) if j not in valsuppr)
         # longeur de la chaine de nucleotide
        ReadSet[cle][1] = len(ReadSet[cle][0][0])
        # nouvelle liste de valeur qualité
        ReadSet[cle][2] = new
    return ReadSet

print("Filtre sur valuers de qualités minimales :"
      , FiltreExtremites(dico_BS, 0.3), "\n")
```

Cette procédure sert à éliminer tous les nucléotides et leur valeur de qualité associée si le seuil, fixé par l'utilisateur, n'a pas été atteint. Ici, le seuil est fixé à 0.3. Si on veut modifier ce seuil, il suffit de modifier le chiffre dans l'appel de la fonction. Par exemple, pour un seuil à **0.25**, on écrit **FiltreExtremites(dico_BS, 0.25)**. De même, on peut modifier le dictionnaire à traiter pour travailler sur celui du brin non sens. A ce moment là, on écrira **FiltreExtremites(dico_BNS, 0.25)**.

VII. Filtre de qualité en fonction de la moyenne des valeurs de qualité

```
"""
Cette procédure prend pour argument ReadSet (vu dans les procédures précédentes
et SeuilQualité. Cette fois-ci, le seuil de qualité rentré en argument va
éliminer les reads entiers ainsi que leur valeur de qualité si la moyenne de
ces valeurs (nldr de qualité) ont une moyenne strictement inférieure au seuil
définit par l'utilisateur.
"""
def Filtre(ReadSet, SeuilQualite):
    faible = []
    for cle in ReadSet.keys():
        moyenne = np.average(ReadSet[cle][2])
        if moyenne < SeuilQualite:
            faible.append(cle)
    for i in faible:
        ReadSet.pop(i)
    return ReadSet

print("Filtre sur les qualités moyennes :", Filtre(dico_BS, 0.6), "\n")
```

Cette procédure sert à éliminer tous les reads dont la moyenne de la somme des valeurs de qualité n'atteint pas la moyenne fixée par l'utilisateur. Ici, la moyenne est de 0.6 et le dictionnaire traité est le dico_BS. On peut changer ces 2 paramètres. Si on veut travailler sur le dictionnaire du brin non sens avec un seuil de qualité fixé à 0.8, on écrira **Filtre(dico_BNS, 0.8)**.

VIII. Le meilleur chevauchement

```
"""
Cette procédure prend pour argument Read (séquence de nucléotide à tester) et
ReadSet (vu dans les procédures précédentes).
Son but est de nous indiquer l'ID du Read qui se chevauche le mieux avec le
Read que l'on a entré en paramètre.
"""
def BestReadChevauchant(Read, ReadSet):
    maximum = max_chev = 0; best_id = None
    # contrôle de chaque Read du dictionnaire
    for cle in ReadSet.keys():
        # longueur du Read le plus court avec le chevauchement maximal
        c = lmin = min(len(Read),len(ReadSet[cle][0][0]))
        # test à partir du chevauchement maximal
        while c > 0 and c <= lmin:
            if ReadSet[cle][0][0][:c] == Read[-c:] or ReadSet[cle][0][0][-c:] == Read[0:c]:
                max_chev = c
                break
            else:
                c -= 1
        if maximum < max_chev:
            maximum = max_chev
            best_id = cle
    return best_id

Read = 'GGTCATACATAGTC'
print("L'ID du read qui correspond le mieux avec la séquence suiavnte", Read,
      "est le read n°" , BestReadChevauchant (Read, dico_BS), "\n")
```

Cette procédure sert à définir l'ID[7] du read contenu dans le dictionnaire choisi par l'utilisateur, qui se chevauche le mieux avec le read passé en argument par l'utilisateur. Par exemple, ici on travaille dans le dictionnaire du brin sens (dico_BS) et notre read est défini dans la variable (Read =). Tous les paramètres de la fonction peuvent être changés à la guise de l'utilisateur.

IV. Assemblage

```
"""
Cette procédure permet re reassembler les reads après les différentes étapes de
filtrage et de recomposer au mieux la séquence de départ.
"""
```

Cette procédure n'est pas disponible pour le moment. Veuillez-nous en excuser.

7 index