

# BIF7101

## Travail pratique 1

Wanlin Li

Pour chacune des questions, des descriptions des étapes sont à coté du code en police verte (Ils commencent par #). Des explications des résultat sont en police bleue.

**Collaborations :** Il y a des défis à surmonter sur ce travail. Nous avons amélioré nos capacités pendant la discussion. J'ai discuté avec monsieur Reinhartz et mes camarades (Douja Meftah, Hamza Maroub, Illiass Amaich, Zineb Mohssine). Ils m'aident beaucoup.

### Problématique

#### Questions

- [65pts] Dans cette question, nous allons créer un petit réacteur évolutif artificiel. Cet exercice va être divisé en plusieurs sous-questions. Le but va être d'observer comme un ensemble de séquences d'ADN évolue lorsque leur probabilité de se développer dépend de sa capacité à encoder une protéine.

Quelques constantes:

- i L'alphabet sera toujours composé de A, C, G et T.
- ii La longueur d'une séquence est toujours 50
- iii Une génération a toujours 100 individus.

- [5pts] Faites une fonction  
jc69(<séquence>, <taux\_mutation>)  
qui va retourner la séquence mutée selon le modèle de Jukes Cantor.  
Avec une probabilité de mutation  $u$  la matrice de transition est:

	A	C	G	T
A	1 - u	u/3	u/3	u/3
C	u/3	1-u	u/3	u/3
G	u/3	u/3	1-u	u/3
T	u/3	u/3	u/3	1-u

```
> jc69('ACGT', 0)
'ACGT'
>jc69('ACGT', 1)
'CATA'
>jc69('ACGT', 0.5)
AATT
```

```
[ 8 ] import random
import numpy as np
import pandas as pd
```

```
[38] def jc69(sequence, taux_mutation):
    u = taux_mutation
    sequence_mutee = ""
    DNAList = ["A", "C", "G", "T"]
    # la probabilité de mutation u la matrice de transition
    df = pd.DataFrame({"A": [1-u,u/3,u/3,u/3],
                       "C": [u/3,1-u,u/3,u/3],
                       "G": [u/3,u/3,1-u,u/3],
                       "T": [u/3,u/3,u/3,1-u]}, index = DNAList)
    for base in sequence:
        # Choisir des éléments avec des probabilités différentes
        base_change = np.random.choice(DNAList, 1, p=[df[base]["A"], df[base]["C"], df[base]["G"], df[base]["T"]])[0]
        sequence_mutee += base_change

    return sequence_mutee
```

▶ jc69("ACGT", 0)

'ACGT'

(b) [5pts] Faites une fonction

k2p(<séquence>, <taux\_transition>, <taux\_transversion>)

qui va retourner la séquence mutée selon le modèle de Kimura deux paramètres. Avec un taux de transition  $\alpha$  et de transversion  $\beta$  la matrice de transition est:

	A	C	G	T
A	$1 - (\alpha + 2\beta)$	$\beta$	$\alpha$	$\beta$
C	$\beta$	$1 - (\alpha + 2\beta)$	$\beta$	$\alpha$
G	$\alpha$	$\beta$	$1 - (\alpha + 2\beta)$	$\beta$
T	$\beta$	$\alpha$	$\beta$	$1 - (\alpha + 2\beta)$

>k2p('ACGT', 0, 0)

ACGT

```
[48] def k2p(sequence,taux_transition,taux_transversion):
    a = taux_transition
    b = taux_transversion
    sequence_mutee = ""
    DNAList = ["A","C","G","T"]
    # la probabilité de mutation u la matrice de transition
    df = pd.DataFrame({"A": [1-(a+2*b),b,a,b],
                       "C": [b,1-(a+2*b),b,a],
                       "G": [a,b,1-(a+2*b),b],
                       "T": [b,a,b,1-(a+2*b)]}, index = DNAList)
    for base in sequence:
        # Choisir des éléments avec des probabilités différentes
        base_change = np.random.choice(DNAList, 1, p=[df[base]["A"], df[base]["C"], df[base]["G"], df[base]["T"]])[0]
        sequence_mutee += base_change

    return sequence_mutee
```

▶ k2p('ACGT', 0, 0)

ACGT

(c) [5pts] Faites une fonction

`selection(<liste_séquences>, <liste_scores>)`  
ou `<liste_séquences>` et `<liste_scores>` sont deux listes de mêmes longueurs. Le score à la position  $i$  correspond à la séquence à la position  $i$ . Si une séquence a comme score  $s$  alors la probabilité de la choisir est de :

$$\frac{s}{\sum_i <\text{liste_scores}>[i]}$$

Par exemple, si nous avons les séquences `['ACGT', 'CCGT', 'CGAT']` avec les scores `[1, 5, 4]` alors la fonction devrait retourner ACGT 10% du temps, CCGT 50% du temps et CGAT 40% du temps.

```
>selection(['ACGT', 'CCGT', 'CGAT'], [1,5, 4])
CCGT
>selection(['ACGT', 'CCGT', 'CGAT'], [1,5, 4])
CGAT
>selection(['ACGT', 'CCGT', 'CGAT'], [1,5, 4])
CCGT
```

```
▶ def selection(list_sequences, list_scores):
    p_choisir = {}
    # mettre le score à la position i correspond à la séquence à la position i en employant .
    zipped = list(zip(list_sequences, list_scores))
    # calculer le score pour chaque séquence dans la liste de séquences.
    for item in zipped:
        seq = item[0]
        score = float(item[1])
        p_choisir[seq] = score/sum(list_scores) # Stocker dans le dictionnaire pour une recherche facile

    for sequence in list_sequences:
        # Choisir des éléments avec des probabilités différentes
        seq_choice = np.random.choice(list(p_choisir.keys()), 1, p= list(p_choisir.values()))[0]

    return seq_choice
```

```
[81] selection(['ACGT', 'CCGT', 'CGAT'], [1,5, 4])
'CCGT'
```

(d) [10pts] Faites une fonction

```
fitness(<séquence>, <protéine_cible>)
qui va
(1) calculer les 3 protéines possibles encodées dans la séquence Indice:
Biopython https://biopython.org/wiki/Seq (enlever les *! Indice:
replace('*', ''))
```

(2) aligner chacune avec la protéine cible, **avec pénalité de -2 pour commencer un nouveau gap et de -1 pour le prolonger**, en utilisant la matrice de score BLOSUM Indice: Biopython <https://biopython.org/docs/1.75/api/Bio.pairwise2.html>

(3) retourne le meilleur score d'alignement **+ 10**.

Cette fonction est déterministe, elle doit toujours retourner la même chose

```
>fitness('ACGACGACGAT', 'QN')
9
>fitness('GACTAGTCGATTCATGCTAC', 'IIII')
10
```

```
[85] !pip install biopython
Collecting biopython
  Downloading https://files.pythonhosted.org/packages/3a/cd/0098eaff841850c01da928c7f509b72fd3e1f51d77b772e24de9e2312471/biopython-1.78-cp37-cp37m-ma
    |████████████████████████████████| 2.3MB 5.6MB/s
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (from biopython) (1.19.5)
Installing collected packages: biopython
Successfully installed biopython-1.78
```

```
from Bio.SubsMat import MatrixInfo as matlist
from Bio import pairwise2
```

```

def fitness(sequence, prot_cible):
    #(1) calculer les 3 protéines possibles encodées dans la séquence Indice
    # changer les index. Le coding_dna va commencer à la première, deuxième et troisième pos
    proteins_possible = []
    for i in range(3):
        coding_dna = Seq(sequence[i:])
        protein = str(coding_dna.translate()).replace('*', '')
        proteins_possible.append(protein)
    #print(proteins_possible)

    #(2) aligner chacune avec la protéine cible

    matrix = matlist.blosum62 # en utilisant la matrice de score BLOSUM I

    # valeur initiale : Définir un score suffisamment grand et impossible
    score = 99999999999999

    for aa in proteins_possible:
        # aligner chacune avec la protéine cible,
        # avec pénalité de -2 pour commencer un nouveau gap et de -1 pour le prolonger
        alignments = pairwise2.align.globalds(aa, prot_cible, matrix, -2, -1)
        for align in list(alignments):
            if score == 99999999999999:
                score = align[2]           #le premier score
            else:
                if align[2] > score:
                    score = align[2]
                    # Comparez avec le score précédent, afficher toujours le score le plus élevé

    return int(score) + 20 #retourne le meilleur score d'alignement + 20.

```

[ 35 ] fitness('GACTAGTCGATTCTATGCTAC', 'IIII')

```

/usr/local/lib/python3.7/dist-packages/Bio/Seq.py:1250: BiopythonWarning:
20

```



fitness('ACGACGACGAT', 'QN')

```

/usr/local/lib/python3.7/dist-packages/Bio/Seq.py:1250: BiopythonWarning:
19

```

19

(e) [15pts] Faites une fonction

```
evolution(<generation>, <fonction_mutation>, <parametres_mutations>,
```

```
<proteine_cible>)
```

avec <generation> une liste de séquence, <fonction\_mutation> sera soit jc69 ou k2p et <parametres\_mutations> sera une liste avec les paramètres associés (e.g. evolution(['ACGTAG', 'AGGAGA'], jc69, [0.001]), 'WY'). Cette fonction va :

- Trouver le score avec fitness de chaque séquence
- Échantillonner avec la fonction selection autant de séquences qu'il y en a dans générations, en utilisant les scores précédents
- Muter chaque séquence échantillonnée avec la fonction et paramètre fournis
- Retourner la liste de nouvelles séquences

```
>evolution(['ACGTAG', 'AGGAGA'], jc69, [0.001], 'QN')
['AGGAGA', 'ACGTAG']
```

```
▶ def evolution(génération,fonction_mutation,parametres_mutations,proteine_cible):
    # (1) Trouver le score avec fitness de chaque séquence
    scores_génération = {}
    for seq in génération:
        scores_génération[seq] = fitness(seq, protéine_cible)
    scoresList_génération = list(scores_génération.values())
    print("le score avec fitness de chaque séquence:", scoresList_génération)

    # (2)Échantillonner avec la fonction selection autant de séquences
    # qu'il y en a dans générations, en utilisant les scores précédents
    seqListgénération = list(scores_génération.keys())
    séquences_selectionnées = []
    for i in range(len(seqListgénération)):
        seq_selectionnée = selection(seqListgénération,scoresList_génération)
        séquences_selectionnées.append(seq_selectionnée)
    print("séquence échantillonnée:", séquences_selectionnées)

    # (3)Muter chaque séquence échantillonnée avec la fonction et paramètre fournis
    new_séquences = []
    if fonction_mutation == 'jc69':
        for seq in séquences_selectionnées:
            new_séquences.append(jc69(seq,parametres_mutations[0]))
    elif fonction_mutation == "k2p":
        for seq in séquences_selectionnées:
            new_séquences.append(k2p(seq,parametres_mutations[0],parametres_mutations[1]))
    else:
        print("Cette fonction_mutation ne existe pas")
    print("nouvelles séquences", new_séquences)

    return new_séquences
```

```

▶ evolution(['ACGTAG', 'AGGAGA'], "jc69", [0.001], 'QN')

⇒ le score avec fitness de chaque séquence: [9, 11]
séquence échantillonnée: ['ACGTAG', 'AGGAGA']
nouvelles séquences ['ACGTAG', 'AGGAGA']
/usr/local/lib/python3.7/dist-packages/Bio/Seq.py:2338: BiopythonWarning: Partial codon, len(sequence) not a multiple of three. Explicitly trim the sequence
    BiopythonWarning,
['ACGTAG', 'AGGAGA']

```

(f) [25pts] Pour les deux protéines suivantes: **QEN** et **LAD** répéter 10x la procédure suivante:

- Générer une séquence d'ADN au hasard de longueur **50**, la copier 100 fois dans une liste. Ce sera la génération 0.
- En utilisant la fonction **evolution** calculer 500 générations. À chaque génération, calculer la fraction des séquences qui encodent la protéine cible **Enlever les \* de la protéine après translation** (voir Q1d).
- Faire une figure avec sur l'axe des X les générations, et l'axe des Y le nombre la fraction des séquences qui encodent la protéine cible avec 10 séquences de départ différentes

Faites le trois fois avec jc69 en utilisant comme taux de mutations **1/5, 1/50, et 1/500**.

Faites le trois fois avec k2p en utilisant comme taux ( $\alpha, \beta$ ) : **[(1/50, 1/10), (1/5, 1/100), (1/50, 1/100)]**.

Vous devriez avoir 12 images. 6 par protéines cibles. Chaque image devra avoir 10 courbes, pour les 10 séquences aléatoires de départ. Discutez des résultats en maximum 1 page + figures.

```

▶ # (1) Générer une séquence d'ADN au hasard de longueur 50, la copier 100 fois dans une liste.
# Ce sera la génération 0.
def DNARandomCreator(length):
    DNA_random = ''
    s = list(np.random.choice(["A", "C", "G", "T"], length, p=(0.25,0.25,0.25,0.25)))

    DNA_random = DNA_random.join(s)
    #len(DNA_random)
    return DNA_random

def generation0(number_seq,length_seq):
    generation0 = []
    for i in range(number_seq):
        generation0.append(DNARandomCreator(length_seq))
    #print(len(generation0))
    return generation0

g0 = generation0(100,50)
g0[1:6]

⇒ ['ACGGCTACCCAACGTTACATGGAAATAGAACGCATAAACGGTGGAGTTA',
    'AGCGGAAAAAAAAAGCACCCCTGTCCATGAACAGCTCCTCATGCACAA',
    'ACATATGATCGATCGACATTATTCACTGTAACCTCCCCGTTATCACA',
    'AGGTGTCAAGAGATAATTACATCTAGAATGCTGACGATGACTCCGAT',
    'GTCCGGTGCAGAAAGACGTTAGTAGCCCGGAGGAACGTGTACGCAA']

```

```

# (2) calculer la fraction des séquences qui encodent la protéine cible
def fractionProtCibleEncodee(sequence_list,protein_cible):
    n_sequences_cible = 0
    for seq in sequence_list:
        proteins_possible = []
        # calculer les 3 protéines possibles encodées dans la séquence
        for i in range(3):
            coding_dna = Seq(seq[i:])
            # Enlever les * de la protéine après translation
            protein = str(coding_dna.translate()).replace('*', '')
            proteins_possible.append(protein)
    #print("proteins_possible",proteins_possible)

    #Le nombre des séquences qui encodent la protéine cible sur les 3 protéines possibles
    n_cible = 0
    for prot in proteins_possible:
        #print(prot)
        if protein_cible in prot:
            n_cible += 1
            #print("n_cible",n_cible)

    #Des trois possibilités, tant qu'une réussisse , alors on compte la séquence
    if n_cible > 0:
        n_sequences_cible += 1
    #print("n_sequences_cible",n_sequences_cible)

    return n_sequences_cible/len(sequence_list)

```

► # (3) En utilisant la fonction evolution calculer 500 générations.  
 # À chaque génération, calculer la fraction des séquences qui encodent la protéine cible

```

def fractionCiblePerGeneration(protein_cible,fonction_mutation,parametres_mutations,n_total_generation):

    fractionCiblePerGeneration = []

    g0 = generation0(100,50)

    fraction_g0 = fractionProtCibleEncodee(g0,protein_cible)

    fractionCiblePerGeneration.append(fraction_g0)

    last_generation = g0

    n_generation = 1

    while n_generation <=500:
        new_generation = evolution(last_generation,fonction_mutation,parametres_mutations,protein_cible)

        #fraction_new_generation = fractionProtCibleEncodee(new_generation,protein_cible)

        fractionCiblePerGeneration.append(fractionProtCibleEncodee(new_generation,protein_cible))

        last_generation = new_generation
        n_generation +=1
        #print(f'generation {n_generation} : fraction {fraction_new_generation:.2f}')

    return fractionCiblePerGeneration

```

Toutes les étapes sont mises sur une fonction (`fractionCiblePerGeneration (protein_cible, fonction_mutation, parametres_mutations, n_total_generation)`). Pour chaque situation, nous pouvons réutiliser cette fonction 10 fois avec des paramètres correspondants.

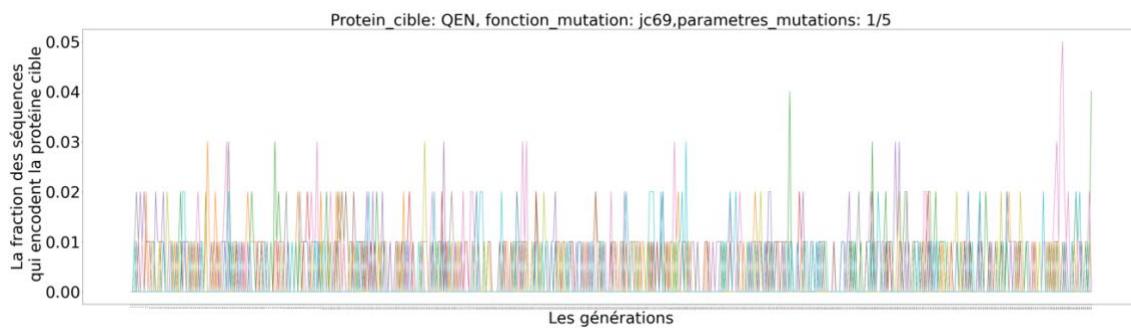
```
[ ] # (4) Faire une figure avec sur l'axe des X les générations  
# l'axe des Y le nombre la fraction des séquences qui encodent la protéine cible  
# avec 10 séquences de départ différentes
```

```
#4.1 Stocker les données sur DataFrame
```

```
▶ df_jc69_gen_5 = pd.DataFrame({"individu1":fraction_jc69_5_1,  
                                "individu2":fraction_jc69_5_2,  
                                "individu3":fraction_jc69_5_3,  
                                "individu4":fraction_jc69_5_4,  
                                "individu5":fraction_jc69_5_5,  
                                "individu6":fraction_jc69_5_6,  
                                "individu7":fraction_jc69_5_7,  
                                "individu8":fraction_jc69_5_8,  
                                "individu9":fraction_jc69_5_9,  
                                "individu10":fraction_jc69_5_10,  
                                "generation":range(501)})  
  
df_jc69_gen_5.head()
```

	individu1	individu2	individu3	individu4	individu5	individu6	individu7	individu8	individu9	individu10	generation
0	0.0	0.00	0.00	0.0	0.00	0.0	0.00	0.00	0.00	0.00	0
1	0.0	0.00	0.00	0.0	0.00	0.0	0.00	0.00	0.00	0.00	1
2	0.0	0.00	0.00	0.0	0.00	0.0	0.01	0.01	0.01	0.01	2
3	0.0	0.00	0.01	0.0	0.00	0.0	0.01	0.02	0.00	0.00	3
4	0.0	0.01	0.00	0.0	0.01	0.0	0.00	0.01	0.00	0.00	4

```
▶ # Faire la figure  
import seaborn as sns  
import matplotlib.pyplot as plt  
from matplotlib.pyplot import figure  
  
plt.figure()  
  
figure(num=None, figsize=(60, 16), dpi=80, facecolor='w', edgecolor='k')  
  
plt.xticks(range(501), df_jc69_gen_5['generation'])  
plt.plot(df_jc69_gen_5['individu1'].astype(float))  
plt.plot(df_jc69_gen_5['individu2'].astype(float))  
plt.plot(df_jc69_gen_5['individu3'].astype(float))  
plt.plot(df_jc69_gen_5['individu4'].astype(float))  
plt.plot(df_jc69_gen_5['individu5'].astype(float))  
plt.plot(df_jc69_gen_5['individu6'].astype(float))  
plt.plot(df_jc69_gen_5['individu7'].astype(float))  
plt.plot(df_jc69_gen_5['individu8'].astype(float))  
plt.plot(df_jc69_gen_5['individu9'].astype(float))  
plt.plot(df_jc69_gen_5['individu10'].astype(float))  
  
plt.xlabel("Les générations", fontsize = 50)  
plt.ylabel("La fraction des séquences\nqui encodent la protéine cible", fontsize=50)  
plt.title('Protein_cible: QEN, fonction_mutation: jc69,parametres_mutations: 1/5', fontsize=50)  
plt.yticks(fontsize=50)  
plt.xticks(fontsize=5)  
  
for item in plt.gca().xaxis.get_ticklabels():  
    item.set_rotation(90)  
plt.savefig('df_jc69_gen_5')  
  
plt.show()
```



```
protein_cible = 'QEN'  
fonction_mutation = 'jc69'  
parametres_mutations = [1/50]
```

```
[121] #Faites le trois fois avec jc69 en utilisant comme taux de mutations 1/50
protein_cible = 'QEN'
fonction_mutation = 'jc69'
parametres_mutations = [1/50]
n_total_generation = 500

#répéter 10x cette procédure
fraction_jc69_50_1 = fractionCiblePerGeneration(protein_cible,fonction_mutation,parametres_mutations,n_total_generation)
print(fraction_jc69_50_1)

/usr/local/lib/python3.7/dist-packages/Bio/Seq.py:2338: BiopythonWarning: Partial codon, len(sequence) not a multiple of three
  BiopythonWarning,
```

```
[129] fraction_jc69_50_8 = fractionCiblePerGeneration(protein_cible,fonction_mutation,parametres_mutations,n_total_generation)
      print(fraction_jc69_50_8)

/usr/local/lib/python3.7/dist-packages/Bio/Seq.py:2338: BiopythonWarning: Partial codon, len(sequence) not a multiple of three
BiopythonWarning,
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.015625, 0.03333333333333333, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
```

```

▶ df_jc69_gen_50 = pd.DataFrame({"individu1":fraction_jc69_50_1,
                                 "individu2":fraction_jc69_50_2,
                                 "individu3":fraction_jc69_50_3,
                                 "individu4":fraction_jc69_50_4,
                                 "individu5":fraction_jc69_50_5,
                                 "individu6":fraction_jc69_50_6,
                                 "individu7":fraction_jc69_50_7,
                                 "individu8":fraction_jc69_50_8,
                                 "individu9":fraction_jc69_50_9,
                                 "individu10":fraction_jc69_50_10,
                                 "Generation": range(501)})
df_jc69_gen_50.tail()

```

	individu1	individu2	individu3	individu4	individu5	individu6	individu7	individu8	individu9	individu10	Generation
496	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	496
497	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	497
498	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	498
499	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	499
500	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	500

```

import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure

plt.figure()

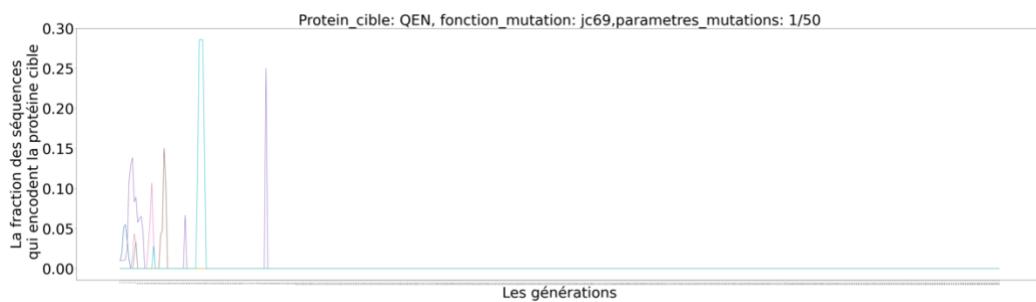
figure(num=None, figsize=(60, 16), dpi=80, facecolor='w', edgecolor='k')

plt.xticks(range(501), df_jc69_gen_50['Generation'])
plt.plot(df_jc69_gen_50['individu1'].astype(float))
plt.plot(df_jc69_gen_50['individu2'].astype(float))
plt.plot(df_jc69_gen_50['individu3'].astype(float))
plt.plot(df_jc69_gen_50['individu4'].astype(float))
plt.plot(df_jc69_gen_50['individu5'].astype(float))
plt.plot(df_jc69_gen_50['individu6'].astype(float))
plt.plot(df_jc69_gen_50['individu7'].astype(float))
plt.plot(df_jc69_gen_50['individu8'].astype(float))
plt.plot(df_jc69_gen_50['individu9'].astype(float))
plt.plot(df_jc69_gen_50['individu10'].astype(float))

plt.xlabel("Les générations", fontsize = 50)
plt.ylabel("La fraction des séquences\n qui encodent la protéine cible", fontsize=50)
plt.title('Protein_cible: QEN, fonction_mutation: jc69,parametres_mutations: 1/50', fontsize=50)
plt.yticks(fontsize=50)
plt.xticks(fontsize=5)

for item in plt.gca().xaxis.get_ticklabels():
    item.set_rotation(90)
plt.savefig('df_jc69_gen_50')
plt.show()

```



```
protein_cible = 'QEN'  
fonction_mutation = 'jc69'  
parametres_mutations = [1/500]
```

```

plt.figure()

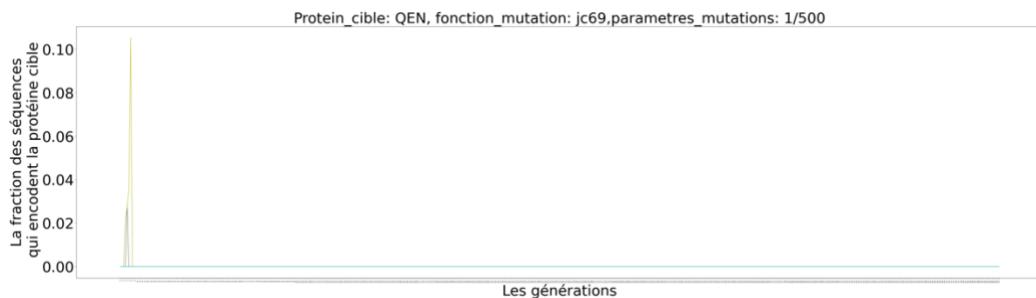
figure(num=None, figsize=(60, 16), dpi=80, facecolor='w', edgecolor='k')

plt.xticks(range(501), df_jc69_gen_500['Generation'])
plt.plot(df_jc69_gen_500['individu1'].astype(float))
plt.plot(df_jc69_gen_500['individu2'].astype(float))
plt.plot(df_jc69_gen_500['individu3'].astype(float))
plt.plot(df_jc69_gen_500['individu4'].astype(float))
plt.plot(df_jc69_gen_500['individu5'].astype(float))
plt.plot(df_jc69_gen_500['individu6'].astype(float))
plt.plot(df_jc69_gen_500['individu7'].astype(float))
plt.plot(df_jc69_gen_500['individu8'].astype(float))
plt.plot(df_jc69_gen_500['individu9'].astype(float))
plt.plot(df_jc69_gen_500['individu10'].astype(float))

plt.xlabel("Les générations", fontsize = 50)
plt.ylabel("La fraction des séquences\nqui encodent la protéine cible", fontsize=50)
plt.title('Protein_cible: QEN, fonction_mutation: jc69,parametres_mutations: 1/500', fontsize=50)
plt.yticks(fontsize=50)
plt.xticks(fontsize=5)

for item in plt.gca().xaxis.get_ticklabels():
    item.set_rotation(90)
plt.savefig('df_jc69_gen_500')
plt.show()

```



Pour toutes les autres images, les mêmes étapes sont employées avec des paramètres différents.

```

protein_cible = 'LAD'
fonction_mutation = 'jc69'
parametres_mutations = 1/5

[147] #Faites le trois fois avec jc69 en utilisant comme taux de mutations 1/5
protein_cible = 'LAD'
fonction_mutation = 'jc69'
parametres_mutations = [0.2]
n_total_generation = 500

#répéter 10x cette procédure
fraction_jc69_lad_5_1 = fractionCiblePerGeneration(protein_cible,fonction_mutation,parametres_mutations,n_total_generation)
print(fraction_jc69_lad_5_1)

/usr/local/lib/python3.7/dist-packages/Bio/Seq.py:2338: BiopythonWarning: Partial codon, len(sequence) not a multiple of three
BiopythonWarning,
[0.0, 0.02, 0.02, 0.01, 0.02, 0.02, 0.02, 0.01, 0.01, 0.0, 0.01, 0.0, 0.02, 0.0, 0.03, 0.0, 0.01, 0.02, 0.01, 0.04, 0.01, 0.0,

```

```
protein_cible = 'LAD'  
fonction_mutation = 'jc69'  
parametres_mutations = 1/50
```

```
#Faites le trois fois avec jc69 en utilisant comme taux de mutations 1/50
protein_cible = 'LAD'
fonction_mutation = 'jc69'
parametres_mutations = [1/50]
n_total_generation = 500

#répéter 10x cette procédure
fraction_jc69_lad_50_1 = fractionCiblePerGeneration(protein_cible,fonction_mutation,parametres_mutations,n_total_generation)
print(fraction_jc69_lad_50_1)

/usr/local/lib/python3.7/dist-packages/Bio/Seq.py:2338: BiopythonWarning: Partial codon, len(sequence) not a multiple of three
BiopythonWarning,
[0.01, 0.04, 0.0625, 0.0833333333333333, 0.1375, 0.25, 0.14492753623188406, 0.1076923076923077, 0.09836065573770492, 0.090909
```

```
protein_cible = 'LAD'  
fonction_mutation = 'jc69'  
parametres_mutations = 1/500
```

```
#Faites le trois fois avec jc69 en utilisant comme taux de mutations 1/500
protein_cible = 'LAD'
fonction_mutation = 'jc69'
parametres_mutations = [1/500]
n_total_generation = 500

#répéter 10x cette procédure
fraction_jc69_lad_500_1 = fractionCiblePerGeneration(protein_cible,fonction_mutation,parametres_mutations,n_total_generation)
print(fraction_jc69_lad_500_1)

> /usr/local/lib/python3.7/dist-packages/Bio/Seq.py:2338: BiopythonWarning: Partial codon, len(sequence) not a multiple of three
  BiopythonWarning,
[0.01, 0.04, 0.06666666666666667, 0.07692307692307693, 0.04166666666666664, 0.05555555555555555, 0.06666666666666667, 0.0, 0.
```

```
protein_cible = 'QEN'  
fonction_mutation = "k2p"  
parametres_mutations = (1/50,1/10)
```

```
#Faites le trois fois avec k2p en utilisant comme taux de mutations [1/50,1/10]
protein_cible = 'QEN'
fonction_mutation = 'k2p'
parametres_mutations = [1/50,1/10]
n_total_generation = 500

#répéter 10x cette procédure
fraction_k2p_gen_50_10_1 = fractionCiblePerGeneration(protein_cible,fonction_mutation,parametres_mutations,n_total_generation)
print(fraction_k2p_gen_50_10_1)

/usr/local/lib/python3.7/dist-packages/Bio/Seq.py:2338: BiopythonWarning: Partial codon, len(sequence) not a multiple of three
BiopythonWarning,
```

```
protein_cible = 'QEN'  
fonction_mutation = "k2p"  
parametres_mutations = ((1/5, 1/100),
```

```
[1]: #Faites le trois fois avec k2p en utilisant comme taux de mutations [1/5,1/100]
protein_cible = 'QEN'
fonction_mutation = 'k2p'
parametres_mutations = [1/5,1/100]
n_total_generation = 500

#répéter 10x cette procédure
fraction_k2p_gen_5_100_1 = fractionCiblePerGeneration(protein_cible,fonction_mutation,parametres_mutations,n_total_gen)
print(fraction_k2p_gen_5_100_1)
```

```
protein_cible = 'QEN'  
fonction_mutation = "k2p"  
parametres_mutations = (1/50, 1/100)
```

```
#Faites le trois fois avec k2p en utilisant comme taux de mutations [1/50,1/100]
protein_cible = 'QEN'
fonction_mutation = 'k2p'
parametres_mutations = [1/50,1/100]
n_total_generation = 500

#répéter 10x cette procédure
fraction_k2p_gen_50_100_1 = fractionCiblePerGeneration(protein_cible,fonction_mutation,parametres_mutations,n_total_ge
print(fraction_k2p_gen_50_100_1)
```

```
protein_cible = 'LAD'  
fonction_mutation = "k2p"  
parametres_mutations = (1/50,1/10)
```

```
[39] #Faites le trois fois avec k2p en utilisant comme taux de mutations [1/50,1/10]
protein_cible = 'LAD'
fonction_mutation = 'k2p'
parametres_mutations = [1/50,1/10]
n_total_generation = 500

#répéter 10x cette procédure
fraction_k2p_lad_50_10_1 = fractionCiblePerGeneration(protein_cible,fonction_mutation,parametres_mutations,n_total_gen
print(fraction_k2p_lad_50_10_1)

/usr/local/lib/python3.7/dist-packages/Bio/Seq.py:2338: BiopythonWarning: Partial codon, len(sequence) not a multiple o
BiopythonWarning,
[0.01, 0.02, 0.01, 0.0, 0.0, 0.01, 0.0, 0.0, 0.0, 0.0, 0.01, 0.03, 0.0, 0.0, 0.0, 0.0, 0.01, 0.02, 0.02, 0.01, 0.02, 0.
```

```
protein_cible = 'LAD'  
fonction_mutation = "k2p"  
parametres_mutations = ((1/5, 1/100)
```

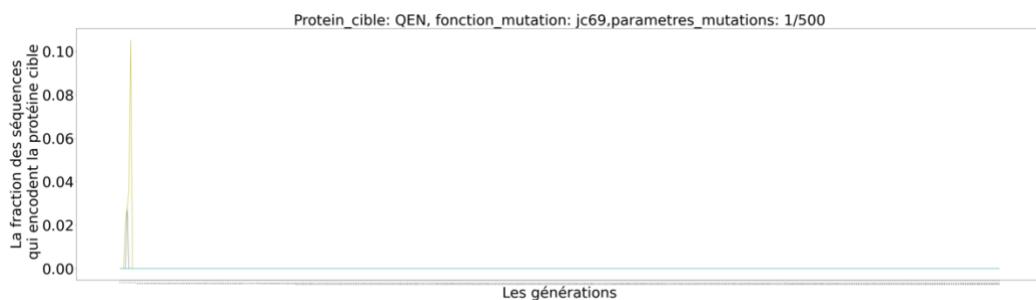
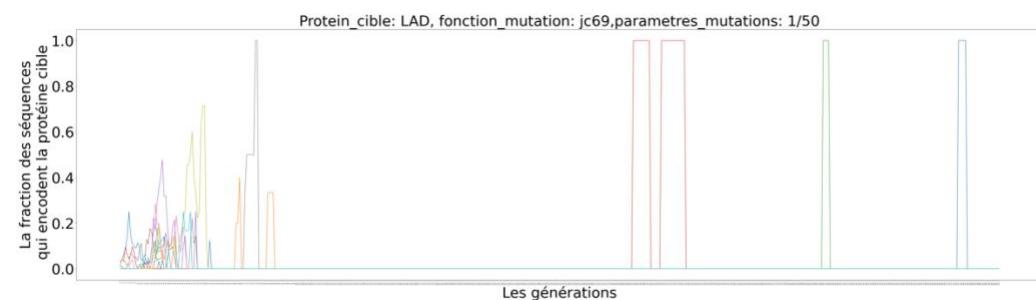
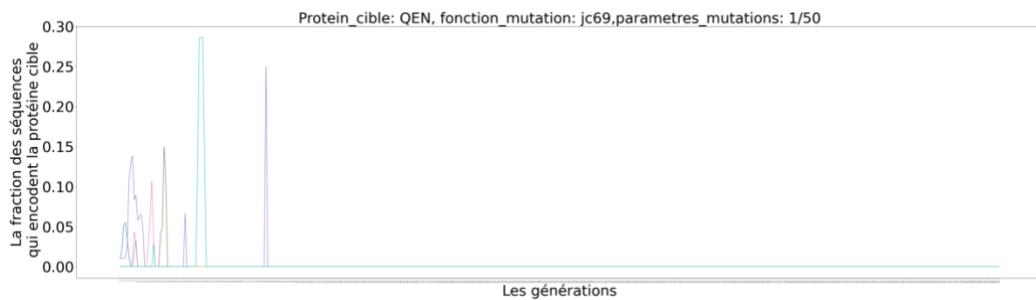
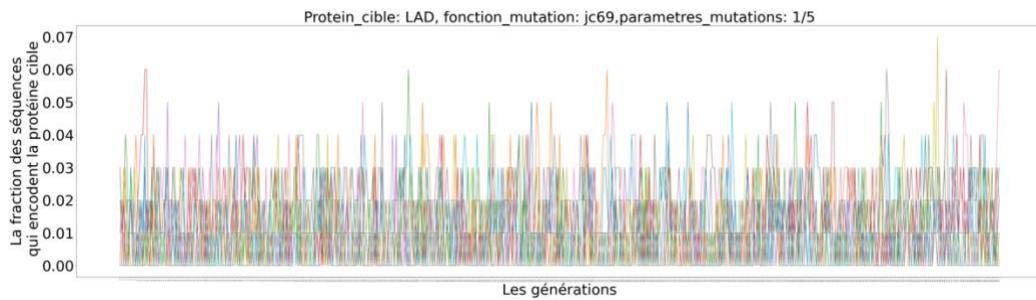
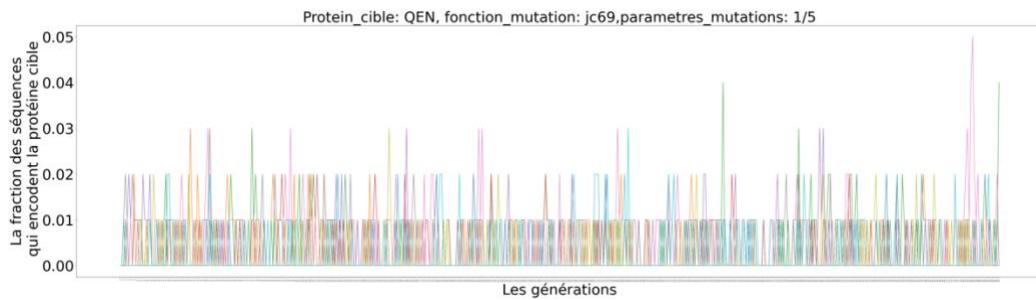
```
[53] #Faites le trois fois avec k2p en utilisant comme taux de mutations [1/5,1/100]
protein_cible = 'LAD'
fonction_mutation = 'k2p'
parametres_mutations = [1/5,1/100]
n_total_generation = 500

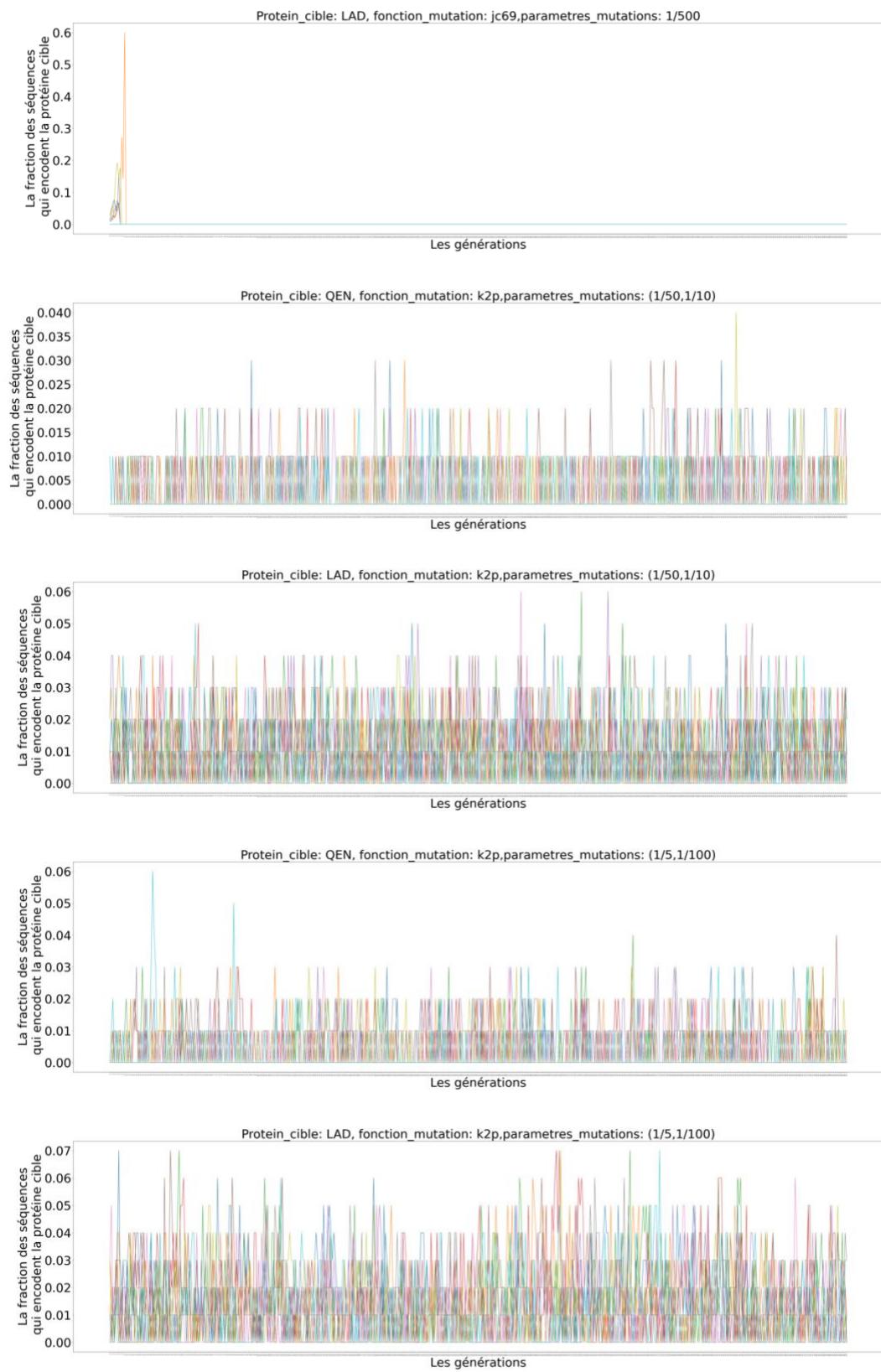
#répéter 10x cette procédure
fraction_k2p_lad_5_100_1 = fractionCiblePerGeneration(protein_cible,fonction_mutation,parametres_mutations,n_total_gen
print(fraction_k2p_lad_5_100_1)
```

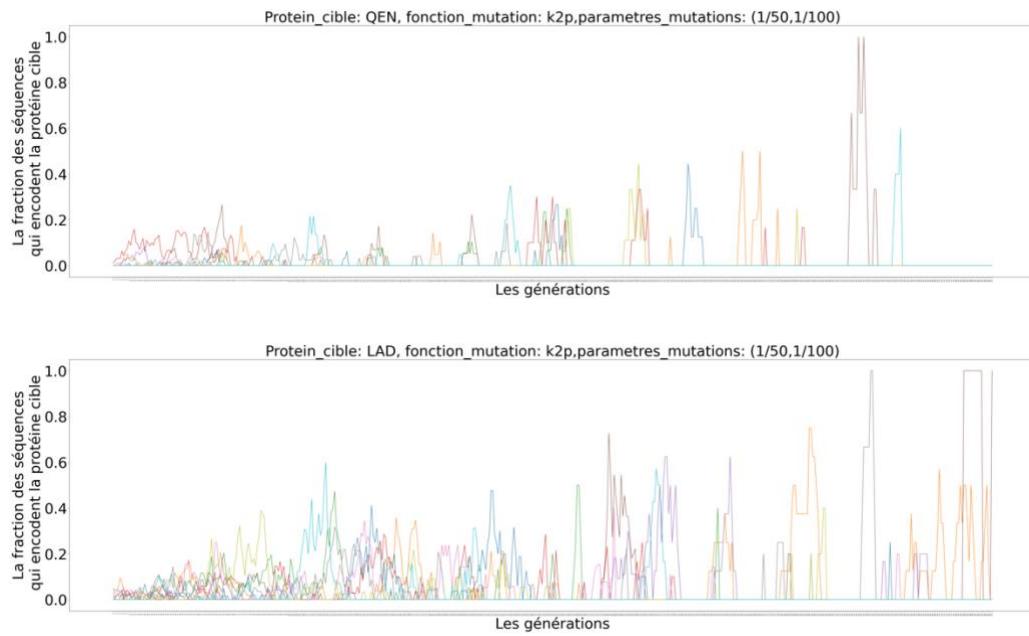
```
protein_cible = 'LAD'  
fonction_mutation = "k2p"  
parametres_mutations = (1/50, 1/100)
```

```
[65] #Faites le trois fois avec k2p en utilisant comme taux de mutations [1/50,1/100]
protein_cible = 'LAD'
fonction_mutation = 'k2p'
parametres_mutations = [1/50,1/100]
n_total_generation = 500

#répéter 10x cette procédure
fraction_k2p_lad_50_100_1 = fractionCiblePerGeneration(protein_cible,fonction_mutation,parametres_mutations,n_total_ge
print(fraction_k2p_lad_50_100_1)
```







- 1) Pour ces deux modèles d'évolution des séquences (Juke-Cantors et Kimura), quand la probabilité de mutation est plus petite, après des générations, la capacité à encoder certaine protéine est limitée. Ça veut dire leur probabilité de se développer diminue.
- 2) Comparer jc69 en taux de mutation 1/50 et k2p en taux (1/50, 1/100), avec le modèle Juke-Cantors, la majorité d'individus n'encode pas la protéine cible après environ 60 générations. Même si le modèle Kimura en taux de transversion plus petit, il y a plus d'individu encodant la protéine cible après 60 génération. Nous pouvons trouver plus de diversité dans le modèle Kimura.
- 3) Dans tous les 12 cas, il y a des événements occasionnels. Ça veut dire, dans une génération, un individu a une fraction des séquences qui encodent la protéine cible beaucoup plus élevé que l'autre individu et que l'autre générations. Cela simule l'apparition de mutations importante au cours de l'évolution. Cette situation est plus évidente lorsque la probabilité de mutation est faible.
- 4) Selon *Inverse DNA codon table*, pour Gln(Q), Glu(E), Asn(N) et Asp(D), chacun de ces quatre acides aminés ont deux codons. Cependant, Leu(L) a six codons. Ala(A) a quatre codons. Donc, il existe huit combinaisons pour traduire QEN ( $2*2*2 = 8$ ), mais les combinaisons pour traduire LAD sont 48 ( $6*4*2 = 48$ ). C'est la raison pour laquelle la diversité et la moyenne des fractions sont plus élevées en LAD, lorsque les autres paramètres sont identiques.

2. [35pts] Dans cette question nous allons évaluer la qualité de différentes méthodes de reconstruction phylogénétique sur des données de RFAM. Utiliser l'alignement **FASTA (gaped)** de la famille RF0001 : <https://rfam.xfam.org/family/RF0001#tabview=tab2> ainsi que d'une autre famille au choix. Attention, toutes les familles ne fonctionnent pas car il manque de l'information.

(a) [5pts] Faire une fonction

`lire_fasta(<fichier_ali>)` qui prend en entrée le chemin d'un fichier d'alignement FASTA et retourne un alignement de type AlignIO de Biopython (<https://biopython.org/wiki/AlignIO>). Le nom des séquences doit être modifié. Il est dans le format X/Y et il ne faut garder que le X. S'il se retrouve plusieurs fois, en garder une au hasard. Indice: attribut `.id`. L'alignement de la fin doit avoir au plus 25 séquences, les choisir au hasard.

- Télécharger le fichier

**Family: 5S\_rRNA (RF00001)**  
Description: 5S ribosomal RNA

```
▶ def lire_fasta(fichier_ali):
    alignment = AlignIO.read(open(fichier_ali), "fasta")
    ali_Dict = {} # S'il se retrouve plusieurs fois, en garder une au hasard.
                  # Donc, employer le dictionnaire
    while len(ali_Dict) < 25:
        i = randint(0,len(alignment)-1) # choisir l'index des séquences au hasard.
        ali_ID = re.sub(r"\d*-\d*", "", alignment[i].id)
        #Il est dans le format X/Y et il ne faut garder que le X.
        ali_Dict[ali_ID] = str(alignment[i].seq)
        # dictionary ----> MultipleSeqAlignment object
    seqRecord_list = []
    for key, value in ali_Dict.items():
        record = SeqRecord(Seq(value), id=key)
        seqRecord_list.append(record)
    #retourner un alignement de type AlignIO de Biopython
    align = MultipleSeqAlignment(seqRecord_list,
                                 annotations={"tool": "demo"})
    return align
```

```

data = lire_fasta('RF00001.afa.txt')
print(data)

⇒ Alignment with 25 rows and 230 columns
--GGAUGC-GA-U-C-AU-ACC----AG-C-A-C-U---AA----...UAC AY034776.1
--GCGUAC-GG-C-C-AU-ACU----AC-C-G-G-G--AA----...GCU X01484.1
--UCCGGU-GA-C-C-AU-ACC----CA-A-A-C-G---GA----...GCC M35170.1
--GCCUAC-GG-C-C-AU-ACU----AG-C-C-U-G---AA----...GCU DQ020568.1
--CUGGU-GG-C-C-UG-AGC----GG-U-G-U-G---CC----...GGC X03902.1
--CCUGGU-GG-C-G-AU-GGC----GA-G-A-A-G---GU----...GGC Z11816.1
--AGGUGC-GA-U-C-AU-ACC----AG-C-A-C-U---AA----...CUC X15199.1
--CCUGGU-GA-U-U-AU-GGA----GA-G-A-A-G---GC----...GGU D90255.1
--AUUAUC-GG-C-U-AU-AGA----UG-G-U-A-G---AA----...AUU M11398.1
--UACGGC-GG-C-C-AU-AGC----GG-A-G-G-G---GA----...AAC X15126.1
--GUCAAC-GG-C-C-AU-AGC----AC-G-C-U-G---UA----...ACA X83208.1
--CCUGGC-GA-C-C-AU-AGC----GU-U-U-U-G---GA----...GGC X02239.1
--GUUGGU-GG-U-U-AU-UGU----GU-C-G-G-G---GG----...ACC X55255.1
--AUCUGC-GG-C-C-AU-ACC----GU-G-A-U-G---AA----...GU- M35571.1
--AUCUGG-GG-C-C-AU-ACC----AC-A-G-C-C---GA----...GU- X00067.1
--GAUGCC-GA-U-C-AU-ACC----AG-C-A-C-U---AA----...AGA AF516401.1
--CCUGGU-GG-U-U-AA-AGA----AA-A-G-A-G---GA----...GG- X57791.1
--GUCGGU-GG-C-G-AU-GGC----GG-G-G-A-G---GG----...ACA Z50069.1
...
--UGCGGU-GG-C-A-AU-GGC----AA-G-A-A-G---GA----...CGC X15245.1

```

(b) [5pts] Faire une fonction

matrice\_distance(<ali>) qui prend un alignement tel que produit par la fonction précédente et retourne une matrice de distance.

Pour ce faire, utiliser DistanceCalculator de Biopython avec le modèle identity (voir  
<https://biopython.org/docs/1.75/api/Bio.Phylo.TreeConstruction.html>).

```
[7] from Bio.Phylo.TreeConstruction import DistanceCalculator
      from Bio import AlignIO
```

```

def matrice_distance(ali):
    calculator = DistanceCalculator('identity')
    dm = calculator.get_distance(ali)
    return dm

```

```

data = lire_fasta('RF00001.afa.txt')
md = matrice_distance(data)
print(md)

Z11821.1      0
X16225.1      0.27391304347826084
AF416765.1    0.19130434782608696
J01012.1      0.30000000000000004
X02249.1      0.17391304347826086
M36309.1      0.30000000000000004
M34768.1      0.17826086956521736
X02026.1      0.09999999999999998
X65710.1      0.28695652173913044
M33889.1      0.12608695652173918
X00574.1      0.30434782608695654
M29167.1      0.21739130434782605
X56490.1      0.27391304347826084
X16688.1      0.27391304347826084
X13035.1      0.27391304347826084
X03903.1      0.16521739130434787
K02362.1      0.28260869565217395
AJ131599.1    0.19999999999999996
AF033641.1    0.23478260869565215
X02713.1      0.13478260869565217
Z50065.1      0.18695652173913047
X02706.1      0.27391304347826084
U89919.1      0.27826086956521734
AF416766.1    0.16521739130434787
M33888.1      0.17826086956521736
Z11821.1      X16225.1
X16225.1      AF416765.1
AF416765.1    J01012.1
J01012.1      X02249.1
X02249.1      M36309.1
M36309.1      M34768.1

```

(c) [10pts] Faire une fonction

`arbre_nj(<mat_dist>)` qui à partir d'une matrice de distance retourne l'arbre NJ dans une string en format NEWICK voir <https://biopython.org/wiki/Phylo>.

```

>data = read_afa('RF00001.afa.txt')
>md = matrice_distance(data)
>arbre = arbre_nj(md)
>print(arbre)
((X01556.1:0.18370,X55260.1:0.01630)Inner1:0.00326,M16174.1:0.01413,X55267.1:0.02500)I

```

```
[ 37] from Bio.Phylo.TreeConstruction import DistanceTreeConstructor
      from Bio import Phylo
```

```

[195] def arbre_nj(mat_dist):
        #Constructeur pour les arbres
        constructor = DistanceTreeConstructor()
        tree_nj = constructor.nj(mat_dist)
        # write the information of tree in the "newick"format
        tree_nj_newick = tree_nj.format("newick")

        return tree_nj_newick

```

```

data = lire_fasta('RF00001.afa.txt')
md = matrice_distance(data)
arbre = arbre_nj(md)
print(arbre)

(z11816.1:0.07680,((M33892.1:0.11227,((X55254.1:0.01828,X15126.1:0.01650)Inner2:0.02310,D13616.1:0.03560)

```

(d) [5pts] Faire une fonction

`melange_newick(<ali>)` qui retourne un arbre en format newick tel que fait précédemment, mais avec les noms mélangés aléatoirement.

```

>data = read_afa('RF00001.afa.txt')
>md = matrice_distance(data)
>arbre = arbre_nj(md)
>print(arbre)
((X01556.1:0.18370,X55260.1:0.01630)Inner1:0.00326,M16174.1:0.01413,X55267.1:0.02500)Ir
>arbre2 = melange_newick(data)
>print(arbre2)
((X55260.1:0.18370,X01556.1:0.01630)Inner1:0.00326,X55267.1:0.01413,M16174.1:0.02500)Ir

```

```

def melange_newick(data):
    # mélanger les séquences aléatoirement.
    data_original = list(data)
    random.shuffle(data_original)
    data_shuffle = MultipleSeqAlignment(data_original)

    #print(data_shuffle)

    from Bio.Phylo.TreeConstruction import DistanceTreeConstructor
    calculator = DistanceCalculator('identity')
    constructor = DistanceTreeConstructor(calculator, 'nj')
    tree2 = constructor.build_tree(data_shuffle)
    #print(tree2)

    # write the information of tree in the "newick" format
    tree_nj2_newick = tree2.format("newick")
    return tree_nj2_newick

```

```

data = lire_fasta('RF00001.afa.txt')
md = matrice_distance(data)
arbre = arbre_nj(md)
print(arbre)
arbre2 = melange_newick(data)
print(arbre2)

(((X02025.1:0.05027,(X13662.1:0.04499,Z11820.1:0.07240)Inner13:0.01495)Inner15:0.01111,(Z35330.1:0.01255,X06575.1:0.0178
(((Z35330.1:0.01255,X06575.1:0.01789)Inner2:0.06008,(X02025.1:0.05027,(Z11820.1:0.07240,X13662.1:0.04499)Inner13:0.01495

```

(e) [10pts]

Faire une fonction

`eval_ete(<fichier_ali>)` qui prend en entrée le chemin d'un fichier d'alignement FASTA et va faire 10 fois:

- Créer un arbre newick avec 25 séquences, ainsi que celui avec les noms mélangés.
- Créer le vrai arbre de NCBI pour ces 25 séquences Indice: [http://etetoolkit.org/docs/latest/tutorial/tutorial\\_ncbitaxonomy.html](http://etetoolkit.org/docs/latest/tutorial/tutorial_ncbitaxonomy.html)
- Calculer le ratio entre la distance Robinson-Foulds de l'arbre NJ et le vrai arbre NCBI divisé par la distance Robinson-Foulds de l'arbre mélangé et le vrai arbre NCBI.

Faire 2 figures, une par famille RFAM. Chaque figure représente les ratios des 25 simulations. Discuter en maximum une page.

```
▶ !pip install ete3

[67] # (0) lire un fichier d'alignement FASTA, choisir 25 séquences au hasard.
myData = lire_fasta('RF00001.afa.txt')

▶ #(1) Créer le vrai arbre de NCBI pour ces 25 séquences

# 1.1) obtenir les ID de NCBI, mettre des ID_alignement et ID_NCBI_tax dans un dictionnaire.

def getTaxIDs(data):
    from Bio import Entrez
    Entrez.email = 'wanlinli2018@gmail.com'
    tax_ids_dict = {}
    for align in data:
        #print(align.id)
        tax_ids_dict[align.id] = Entrez.read(Entrez.elink(db='taxonomy',
                                                       dbfrom='nuccore',
                                                       id=align.id))[0]['LinkSetDb'][0]['Link'][0]['Id']

    return tax_ids_dict

[69] ids_dict = getTaxIDs(myData)
print("ids_dict",ids_dict)

ids_ncbi_tax = list(ids_dict.values())
print("ids_ncbi_tax",ids_ncbi_tax)

ids_dict {'X05535.1': '3018', 'Z11816.1': '1471', 'X67494.1': '5221', 'Z50069.1': '1837', 'M11535.1': '934',
ids_ncbi_tax ['3018', '1471', '5221', '1837', '934', '1359', '40318', '6230', '1530', '512', '55710', '134',
```

```
[43] # La première fois il faut télécharger la DB

from ete3 import NCBITaxa
ncbi = NCBITaxa()

[70] # 1.2) Arbre des taxonomies ETE3
from ete3 import Tree
from ete3 import NCBITaxa

def creeNCBIArbre(ids_dict):
    ncbi = NCBITaxa()
    # obtenir les ID
    tax_ids_ncbi = list(ids_dict.values())
    tree_ncbi = ncbi.get_topology(tax_ids_ncbi)
    #print(tree_ncbi)
    tree_ncbi_nwk = tree_ncbi.write()
    return tree_ncbi_nwk

[71] tree_ncbi_nwk = creeNCBIArbre(ids_dict)
tree_ncbi_nwk

'((((134:1,56359:1):1:1,934:1,512:1):1:1,(((1471:1,1359:1):1:1,1530:1):1:1,((28444:1,2014:1):1:1,(1837:1,33910:1):1:1,(1954:1,40318:1):1:1,1704:1):1:1):1:1,(((6230:1,55710:1):1:1,6063:1):1:1,((5489:1,101028:1):1:1,5221:1):1:1):1:1,(3163:1,3371:1):1:1,3018:1):1:1,(2187:1,145262:1):1:1);'

[72] # (3.1) Créer un arbre newick avec ces 25 séquences, changer ID_alignement à ID_NCBI_tax
#ainsi que celui avec les noms mélangés
def changerID(data):
    # change the id
    data2 = data
    for item in data2:
        item.id = str(ids_dict[item.id])      # Très très important, assurer le type d'ID,str()
    return data2

dataWithIDtax = changerID(myData)
dataWithIDtax

<<class 'Bio.Align.MultipleSeqAlignment'> instance (25 records of length 230) at 7f19122bcd0>

▶ print(dataWithIDtax)

Alignment with 25 rows and 230 columns
--AGAAC-GG-C-C-AU-ACC----AC-G-U-C-G---AU----...CU- 3018
--CCUGGU-GG-C-G-AU-GGC----GA-G-A-A-G---GU----...GGC 1471
--AUCCUC-GG-C-C-AU-AGA----AU-G-A-C-G---AA----...UU- 5221
--GUCGGU-GG-C-G-AU-GGC----GG-G-G-A-G---GG----...ACA 1837
--CUUGC-GA-C-C-AU-AGC----GG-A-A-U-G---GA----...AGC 934
--UUUGGU-CA-U-C-AU-UGC----GA-U-G-G-A---GA----...AGU 1359
--UUCGGU-GG-U-C-AU-AGC----AU-G-A-G-G---GA----...AAC 40318
--GGAAC-GA-C-C-AU-ACC----AU-G-C-U-G---AA----...UCA 6230
--CCUAGU-GA-U-G-AU-GGC----GU-A-G-A-G---GA----...GGU 1530
--CCUGAC-GA-C-C-AU-AGC----AA-G-G-U-G---GU----...GGC 512
--GUCUAC-GG-C-C-AU-AUC----AC-G-U-U-G---AA----... 55710
--CUUUGGU-GA-C-C-AU-AGC----GA-G-C-G-U---GA----...AGC 134
--UGUGGU-GG-U-U-AU-UGC----UG-G-A-G-G---GU----...CGG 2014
--GCCUAC-GG-C-C-AU-ACC----AC-G-U-U-G---AA----...GCU 6063
GCCUUGGU-CA-C-A-UG-UGC----CC-C-U-G-G---UA----...AGG 56359
--UGCUAC-GU-U-C-AU-ACC----AC-U-C-A-G---AA----...UGU 3163
```

```
[74] calculator = DistanceCalculator('identity')
    mat_dist = calculator.get_distance(dataWithIDtax)
    #Constructeur pour les arbres
    constructor = DistanceTreeConstructor()
    tree_nj = constructor.nj(mat_dist)
    #print(tree_nj)
    # write the information of tree in the "newick"format
    tree_nj_nwk = tree_nj.format("newick")
    print(tree_nj_nwk)

((((56359:0.12675,(512:0.10310,(134:0.06733,934:0.05875)Inner11:0.00777)Inner12:0.01238)Inner20:0.01000,(135
```

```
[75] #Créer le arbre newick qui a les noms mélangés.
tree_nj_nomMelange_nwk = melange_newick(dataWithIDtax)
print(tree_nj_nomMelange_nwk)

(((56359:0.12675,(512:0.10310,(934:0.05875,134:0.06733)Inner11:0.00777)Inner12:0.01238)Inner20:0.01000,(1359

# (3.1) Calculer la distance Robinson-Foulds de l'arbre NJ et le vrai arbre NCBI
from ete3 import Tree

arbre_nj = Tree(tree_nj_nwk,format=1)
arbre_nj_nomMelange = Tree(tree_nj_nomMelange_nwk,format=1)
arbre_ncbi = Tree(tree_ncbi_nwk,format=1)

rf_nj_ncbi, maxrf, commonleaves, partst1, partst2, _ , _ = arbre_nj.robinson_foulds(arbre_ncbi,unrooted_trees=True)

print("RF distance is %s over a total of %s" %(rf_nj_ncbi, maxrf))
print("Partitions in arbre_ncbi that were not found in arbre_nj:", partst1 - partst2)
print("Partitions in arbre_nj that were not found in arbre_ncbi:", partst2 - partst1)


```

```
[77] # (3.2) Calculer la distance Robinson-Foulds de l'arbre mélange et le vrai arbre NCBI
rf_nomMelange_nj, maxrf, commonleaves, partst1, partst2, _, _ = arbre_nj_nomMelange.robinson_foulds(arbre_ncbi,unrooted_trees=True)

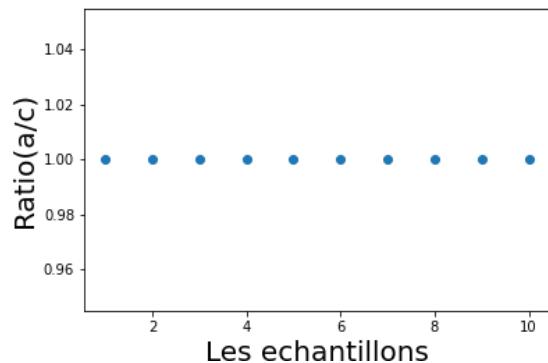
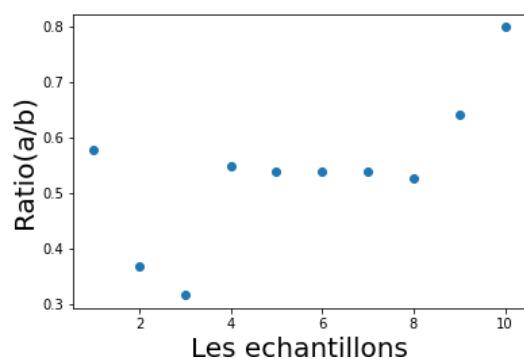
print("RF distance is %s over a total of %s" %(rf_nomMelange_nj, maxrf))
print("Partitions in arbre_ncbi that were not found in arbre_nj_nomMelange:", partst1 - partst2)
print("Partitions in arbre_nj_nomMelange that were not found in arbre_ncbi:", partst2 - partst1)
```

---

RF distance is 26 over a total of 40  
Partitions in arbre\_ncbi that were not found in arbre\_nj\_nomMelange: {('101028', '134', '1359', '145262', '1471', '1530', '1837', '5489')  
Partitions in arbre\_nj\_nomMelange that were not found in arbre\_ncbi: {('101028', '134', '1359', '145262', '1471', '1530', '1837', '5489')}

```
[79] #Calculer le ratio entre la distance Robinson-Foulds de l'arbre NJ  
#et le vrai arbre NCBI divisé par la distance Robinson-Foulds de  
#l'arbre mélangé et le vrai arbre NCBI.  
  
ratio_RF = rf_nj_ncbi/rf_nomMelange_nj  
ratio_RF
```

Répétez 10 fois et enregistrez les résultats



## Conclusion :

- 1) Les noms mélangés aléatoirement ne change pas le résultat d'arbre.
  - 2) La distance Robinson-Foulds de l'arbre NJ et le vrai arbre NCBI est égale la distance Robinson-Foulds de l'arbre mélangé et le vrai arbre NCBI
  - 3) Pour échantillons différents dans une famille RFAM, la distance Robinson-Foulds varie
    - Télécharger le fichier d'alignement d'une autre famille.
    - Utiliser l'alignement FASTA (gaped) de la famille RF00002  
<https://rfam.xfam.org/family/RF00002#tabview=tab2>
    -