Utilisation de Python en cours d'informatique - BCPST 2

Marc Manceau

Avril 2014

Sommaire

Chapitre 1

Utilisation de Python en informatique "scientifique"

Contents

1.1 Console iPython et modules scientifiques

1.1.1 Console iPython

Avantages :
— multi-plateforme
— lancement d'un fichier facilement avec "%run", copie avec "%cpaste"
— complétion des commandes
— compréhension des commandes unix précédées d'un " !" (exemple : objet = !ls)

1.1.2 Module Numpy

Pour tout ce qui est gestion des matrices (ndarray). Toutes les opérations arithmétiques se font élément par élément. Donc pour faire le produit matriciel, ne pas faire $A * B$. Utiliser la fonction dot(A, B).

Il y a des méthodes pratiques :
— ndarray.shape()
— ndarray.sum()
— ndarray.min()
— ndarray.max()
— ndarray.transpose()
— ndarray.resize((2,6))

Les fonctions de la vie de tous les jours :
— arange(début, fin, pas)
— linspace(début, fin, nbre_de_valeurs)
Quelques fonctions pour l'algèbre linéaire classique :
— a.transpose()
— inv(a)
— le produit matriciel dot(a,b)
— eye(2) fait une matrice carrée I_2.
— trace(a)
— eig(a) retourne les valeurs et vecteurs propres

Module Scipy

Un gros package qui comprend notamment, parmi les fonctions qu'on utilise le plus :

scipy.integrate avec notamment les fonctions quad(), odeint(), ode().

scipy.optimize avec minimize(), leastsq(), curve_fit().

scipy.stats avec plein de distributions de v.a. et autres outils pour les utiliser.

scipy.fftpack pour les transformations de Fourier.

Module Matplotlib

Pour tout ce qui est tracés graphiques scientifiques. Il y a toujours une fonction pour ce qu'on souhaite faire.

Et Pylab ?

Pylab, c'est l'ensemble : scipy+numpy+matplotlib. Tout est directement importé, pour faire une environnement comparable à matlab ou scilab.

1.1.3 Compétences à acquérir

Utiliser une bibliothèque

Plusieurs syntaxes sont possibles :

```
1 from scipy import integrate
2 import numpy
3 from pylab import *
```

Utiliser l'aide

Lorsqu'on ne connaît pas du tout une librairie, la première aide consiste à aller voir la documentation sur internet. Par exemple, scipy, numpy et matplotlib disposent de nombreux exemples, des listes de fonctions, d'une aide pour la prise en main.

Ensuite, lorsque la syntaxe d'une fonction est mal connue, le terminal apporte une aide précieuse :

```
1 ?plot
2 help(plot)
```

Pour sortir de l'aide, utiliser les touches "q" ou "echap".

Syntaxe

Maths +, -, *, ** (puissance), % (modulo), exp, log (naturel attention), cos, sin, sqrt, abs, ...

Booléen &, |, not(), ==

Float/int Attention à l'erreur $3/2 = 1$ en python, mais $3.0/2.0 = 1.5$. Ce n'est plus le cas sous python 3.

Indentation Elle est essentielle.

Fonction Avec la syntaxe suivante :

```
1 def moyenne(a,b):
2         moyenne = (a+b)/2.0
3         return moyenne
```

Commentaires avec le caractère # devant une ligne de commentaire.

Condition Syntaxe suivante :

```
1 if booleen1:
2         execution1
3 elif booleen2:
4         execution2
5 else:
6         execution3
```

Boucle Syntaxe suivante :

```
1 for i in liste:
2         execution
3 while booleen:
4         execution
```

Problèmes typiques de syntaxe

Une liste (à agrandir au fur et à mesure) des points qui peuvent poser problème régulièrement :
— la division entière par défaut (ce n'est plus le cas à partir de python 3).
— la copie d'une liste, pour ne pas faire un alias : utiliser *b = list(a)*.

1.2 Algorithmes classiques à connaître

1.2.1 Parcourir une liste

Recherche dans une liste

On fait ça avec nos deux syntaxes de boucles disponibles.

```
1  liste = [1,2,5,9,10,3,8,6,11]
2  objet = 5
3
4  trouvaille = False
5  for element in liste:
6          if element == objet:
7                  trouvaille = True
8
9  print trouvaille
10
11 trouvaille2 = False
12 i = 0
13 while trouvaille2==False & i<len(liste):
14         if liste[i] == objet:
15                 trouvaille2 = True
16         i += 1
17
18 print trouvaille2
```

Ceci retourne, dans notre cas, True et True. Ouf.

Recherche du maximum d'une liste de nombres

La version qui donne le maximum. On peut aussi renvoyer l'argmax facilement.

```
1  liste = [1,2,5,9,10,3,8,6,11,2]
2
3  max = liste[0]
4  for element in liste:
5          if element > max:
6                  max = element
7
8  print max
```

Calcul de la moyenne

La principale source de bug possible est la division entière. Pour l'éviter et avoir la vraie valeur de la moyenne, on transforme notre somme en type 'float'.

```
1  liste = [1,2,5,9,10,3,8,6,11,2]
2
3  Somme = 0
4  for element in liste:
5          Somme += element
6
7  print float(Somme)/len(liste)
```

Recherche d'un mot dans une chaîne de caractères

Dans Python, une chaîne de caractère est une liste. On peut donc la parcourir de la même façon, et rechercher un mot dans une phrase de cette façon.

```
phrase = 'Hier je suis alle manger sur le bord de la Seine'
mot = 'sur'

j = 0
i = 0
while (i < len(phrase)) & (j < len(mot)):
        if phrase[i] == mot[j]:
                j += 1
        else:
                j = 0
        i += 1

if j == len(mot):
        print 'Trouve !'
else:
        print 'Mot absent de la phrase'
```

1.2.2 Trier une liste

Le but de cette partie est d'imaginer ou de comprendre des algorithmes, et leurs différences en terme de temps de calcul. Dans un code utilisant de gros tableaux, on pourra utiliser la fonction Python *sort()*.

Tri par sélection

Sans doute le plus immédiat quand on cherche à résoudre l'exercice. L'algorithme consiste à trouver le min de la liste de 1 à n, à l'échanger avec la première position. Puis le min de la liste de 2 à n, qu'on place en deuxième position, puis ...

```
liste = [11,2,5,9,10,3,8,6,12,11,2,0.5]
n = len(liste)

for j in arange(0, len(liste)-1):
        # On trouve le minimum et sa position
        min = liste[j]
        argmin = j
        for i in arange(j+1,n):
                if liste[i] < min:
                        min = liste[i]
                        argmin = i

        # On place le minimum en position j et l'element en position j a la place du minimum
        liste[argmin] = liste[j]
        liste[j] = min
```

Tri par insertion

Le mécanisme réalisé par un joueur de carte qui reçoit une nouvelle carte et la place dans sa main. Le jeu correspond au début à la première carte, et on place la deuxième carte en fonction de la première. Puis on place la troisième carte en fonction des deux autres. Puis ...

```
liste = [11,2,5,9,10,3,8,6,12,11,2,0.5]
n = len(liste)

i = 1;
while i < n :
        carte = liste[i]
        j = i-1
        # Tant que la carte recue est inferieure, on decale le contenu de toutes les cellules vers la
        while (j>=0) & (carte < liste[j]):
```

```
10                  liste[j+1] = liste[j]
11                  j -= 1
12
13          # On pose la carte le ou on s'est arrete
14          liste[j+1] = carte
15          i += 1
```

Tri à bulle

Cet algorithme consiste à comparer deux cellules adjacentes, et à les ranger entre elles. En faisant glisser la bulle le long du tableau, la plus grande valeur remonte à la surface (à l'extrémité droite du tableau). Puis on recommence sur un tableau plus petit. Lorsque le glissement de la fenêtre n'entraîne plus aucun mouvement, le tableau est rangé.

```
1  liste = [11,2,5,9,10,3,8,6,12,11,2,0.5]
2  n = len(liste)
3
4  j = n-1
5  bool = True              # Ce booleen nous dit s'il faut continuer a faire remonter des bulles
6
7  # Tant que les bulles modifient le tableau
8  while bool:
9          bool = False
10         i = 0
11         while i<j:
12                 # la comparaison 2 a 2 au sein de la bulle
13                 if liste[i]>liste[i+1]:
14                         buffer = liste[i+1]
15                         liste[i+1] = liste[i]
16                         liste[i] = buffer
17                         bool = True
18
19                 # la bulle avance
20                 i += 1
21         # la taille de la liste a trier diminue
22         j -= 1
```

Calcul de la médiane

Une fois notre liste triée, il est facile de prendre la valeur centrale. Pour optimiser un peu plus, on ne pourrait trier notre tableau que jusqu'à la moitié de sa taille, pas besoin de plus.

```
1  # Un fonction triant une liste selon l'algorithme du tri par insertion
2  def Tri(liste):
3          i = 1;
4          while i < n :
5                  carte = liste[i]
6                  j = i-1
7                  while (j>=0) & (carte < liste[j]):
8                          liste[j+1] = liste[j]
9                          j -= 1
10                 liste[j+1] = carte
11                 i += 1
12         return liste
13
14 # Ma fonction mediane
15 def Mediane(liste):
16         liste = Tri(liste)
17         n = len(liste)
18         if n%2==0:
```

```
19                  mediane = (liste[n/2-1]+liste[n/2])/2
20          else:
21                  mediane = liste[n/2]
22          return mediane
23
24  # Un exemple
25  liste = [11,2,5,9,10,3,8,6,12,11,2,0.5]
26  print Mediane(liste)
```

Tri rapide quicksort

Partition de la liste La première étape de l'algorithme consiste à chercher un endroit où couper notre tableau, de façon à ce que tous les éléments de gauches soient inférieurs à tous les éléments de droite. On s'autorise à permuter éventuellement des termes du tableau.

Si on se donne une valeur pivot $T(p)$, avec $p \in \{1, ..., \}$), on cherche à obtenir la propriété :

$$\forall i \in \{0, ...p\}, \quad \forall j \in \{p+1, ...n\}, \quad T(i) \leq T(p) \leq T(j)$$

Pour réaliser ça, on prend vraiment un indice i qui parcourt le tableau de gauche à droite, et un indice j qui le parcourt de droite à gauche. Lorsque $T(i) > T(p)$, i s'arrête. De l'autre côté, j s'arrête lorsque $T(j) < T(p)$. Lorsque les deux indices sont arrêtés, on échange les valeurs dans le tableau.

Et on poursuit ça jusqu'à ce que les deux indices se rencontrent. L'endroit où ils se rencontrent est notre point de pivot.

```
1  def Partition(liste):
2          n = len(liste)
3          # initialisation avec le pivot et nos deux indices
4          pivot = liste[0]
5          i = 0
6          j = n-1
7          while (j>0) & (liste[j] > pivot):
8                  j -= 1
9          while i<j :
10                 buffer = liste[i]
11                 liste[i] = liste[j]
12                 liste[j] = buffer
13                 j -= 1
14                 i += 1
15                 while (j>0) & (liste[j] > pivot):
16                         j-=1
17                 while (i<n) & (liste[i] < pivot):
18                         i+=1
19         return liste[0:j+1], liste[j+1:n]
```

Le tri Il se fait simplement grâce à un appel récursif et au travail réalisé dans la fonction "Partition". L'ensemble revient donc à faire des partitions de plus en plus petites, puis à tout rejoindre bout à bout. On voit ici comment concaténer deux listes : liste1 + liste2.

```
1  def Tri_quicksort(tableau):
2          t1, t2 = Partition(tableau)
3          if len(t1)>1:
4                  t1 = Tri_quicksort(t1)
5          if len(t2)>1:
6                  t2 = Tri_quicksort(t2)
7          return t1+t2
8
9  liste_ex = [0,11,2,5,9,10,3,8,6,12,11,2,0.5]
10 Tri_quicksort( liste_ex )
```

1.2.3 Simulation de variables aléatoires

A partir du générateur aléatoire fourni par le langage python.

On utilisera par exemple le module : *from random import **, puis on utilisera la fonction *random()* à chaque fois que l'on souhaite simuler une variable aléatoire sur $[0, 1]$.

V.a. prenant un nombre fini de valeurs

Une loi de Bernouilli :

```
1 def Bernouilli(p):
2         if random()<p:
3                 retour=1
4         else:
5                 retour=0
6         return retour
```

Une loi binomiale :

```
1 def Binomiale(n,p):
2         """Loi de la somme de n variable de Bernouilli independantes"""
3         somme=0
4         for i in range(0,n):
5                 somme+=Bernouilli(p)
6         return somme
```

Une loi uniforme sur un ensemble discret fini :

```
1 def Uniforme(E):
2         """Chaque element de la liste E a la meme probabilite de tomber"""
3         r = random()
4         i = 0
5         while r > (i+1)/float(len(E)):
6                 i+=1
7         return E[i]
```

Une loi géométrique sur $\mathbb{N}^*$:

```
1 def Geometrique(p):
2         """Loi d'attente du premier succes d'une suite d'experiences independantes de Bernouilli"""
3         t = 1
4         while Bernouilli(p) == 0:
5                 t+=1
6         return t
```

V.a. à densité

Loi uniforme sur un segment $[a, b]$:

```
1 def Uniforme(a,b):
2         return a+random()*(b-a)
```

Par inversion de la fonction de répartition, avec l'exemple de la loi exponentielle.

On a, lorsqu'on sait inverser la fonction de répartition de X :

$$\begin{aligned} F_{F_X^{-1}(U)}(y) &= P(F_X^{-1}(U) \leq y) \\ &= P(U \leq F_X(y)) \\ &= F_X(y) \end{aligned}$$

Dans le cas où X suit une loi exponentielle de paramètre λ, on obtient :

$$F_X(x) = 1 - e^{-\lambda x}$$
$$F_X^{-1}(u) = \frac{-1}{\lambda}\ln(1-u)$$

D'où le code suivant :

```
1 def Exponentielle(lamb):
2         return -1/float(lamb)*log(1-random())
```

1.2.4 Autres algorithmes au programme

Algorithme de Dijkstra

Recherche de plus court chemin dans un graphe pondéré à poids positifs. Le graphe peut être représenté par une matrice d'adjacence ou par une liste.

Algorithmes simples sur une image en niveaux de gris

Les algorithmes présentés sont du type « à balayage » et restent très simples : éclaircissement, accentuation du contraste, flou, accentuation de contours. On met en évidence l'importance de tels algorithmes en les appliquant sur des images issues de la biologie et des géosciences.

Chapitre 2

Projet - Evolution des fréquences alléliques dans une population

Contents

2.1 Introduction

On s'intéresse à l'évolution, au sein d'une population d'individus de taille finie N, de la diversité allélique. On définit pour la suite les termes de génétique des populations suivants :

Gène c'est un fragment de génome transcrit.

Relation d'homologie deux gènes sont dits homologues s'ils descendent d'un même gène ancestral. C'est une relation d'équivalence (réflexive, symétrique, transitive).

Locus classe d'équivalence de la relation d'homologie.

Relation d'isoaction deux gènes homologues déterminant le même phénotype sont isoactifs. C'est une relation d'équivalence.

Allèle classe d'équivalence de la relation d'isoaction.

On s'affranchit ainsi de la question "Qu'est-ce qu'un gène ?", et on va pouvoir modéliser l'évolution de la représentation de nos allèles dans la population.

Notons pour simplifier que le locus se résume à une position physique sur la séquence d'ADN. Et pour un phénotype d'intérêt qui serait moléculaire (la séquence du gène), l'allèle se résume à une version des gènes du locus.

L'objet de la modélisation est de comprendre l'évolution des fréquences alléliques dans la population. On s'intéresse à un locus particulier, et on introduit les notations suivantes :

$$\begin{aligned} 2N &:= \text{nombre de gènes du locus dans une population diploïde de } N \text{ individus.} \\ A_t &:= \text{v.a. égale au nombre de gènes appartenant à l'allèle A à la génération t.} \\ f_A &:= \frac{A_t}{2N} \end{aligned}$$

Classiquement, 4 "forces évolutives" sont responsables de l'évolution des fréquences alléliques :

sélection selon qu'un allèle apporte un avantage ou désavantage sélectif à son porteur.

migration flux d'un groupe de gènes appartenant à des allèles différents.

dérive fluctuation stochastique des fréquences dans une population finie.

mutation transformation un gène pour le faire passer d'un allèle à un autre, potentiellement nouveau.

Nous nous intéresserons dans ce sujet aux deux dernières forces uniquement.

2.2 Effet de la dérive sur l'évolution d'une fréquence allélique

Les générations sont supposées non-chevauchantes et la taille de la population est constante. A la génération $t+1$, chaque gène tire son parent uniformément avec remise, parmi les gènes de la génération précédente. Tout se passe comme sur la figure 2.1 : les gènes sont tirées dans une urne gamétique infinie.

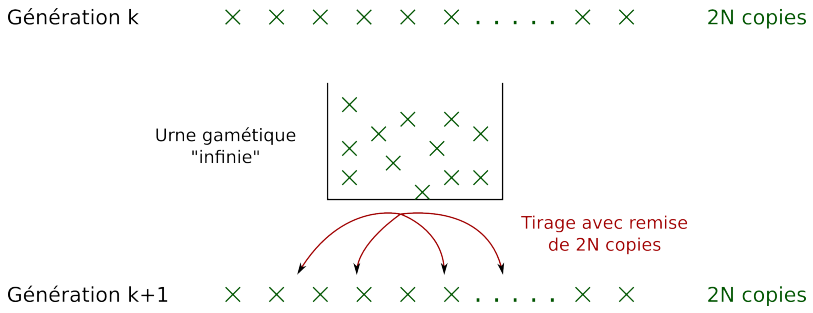

FIGURE 2.1 – Tirage des gènes d'une génération à l'autre.

Soit $a_0 \in \mathbb{R}$. On commence à la génération 0 avec $A_0 = a_0$.

Q 1 *Déterminer la loi de A_1. Donner son espérance et sa variance.*

Q 2 *Soient $(U_1, ...U_{2N})$ des variables aléatoires indépendantes de loi uniforme sur $[0,1]$. Donner un algorithme permettant d'obtenir une v.a. de même loi que A_1 à partir de celles-ci.*
A partir du générateur de nombre aléatoire rand() *de la bibliothèque* Numpy*, implémentez l'algorithme permettant de simuler la variable aléatoire A_1.*

Q 3 *Calculer, $\forall t \in \mathbb{N}^*$, $\forall (i,j) \in \{0, ...2N\}^2$, $\mathbb{P}(A_{t+1} = j | A_t = i)$. En déduire un algorithme permettant de simuler de proche en proche le nombre de gènes appartenant à l'allèle A sur un nombre t_{max} fixé de générations, et représenter quelques exemples de ces trajectoires aléatoires sur le même graphe.*

2.3 Derive et temps avant homogénéisation de la population

On part d'un système à deux allèles A et B, avec $A_0 = a_0$ donné. On appelle τ la variable aléatoire égale au temps de fixation d'un allèle dans la population. Celui-ci est défini de la façon suivante :

$$\tau \quad := \quad \min(t \in \mathbb{N}, \quad A_t = 0 \quad \text{ou } A_t = 2N)$$

On admet que $\tau < \infty$ presque sûrement.

Q 4 *Imaginer et implémenter un algorithme permettant de simuler des réalisations indépendantes de la variable aléatoire τ.*

Q 5 *A partir de cet algorithme, représenter un histogramme empirique de la loi de τ et en calculer la moyenne empirique.*

Q 6 *Intuitivement, pour quelle valeur de départ a_0 le temps moyen de fixation sera-t-il le plus fort ?Le plus faible ? Vérifier empiriquement sur quelques valeurs.*

2.4 Probabilité de fixation d'un allèle sous l'influence de la dérive génétique

On s'intéresse à présent à la probabilité de fixation de l'allèle A. On appelle F l'événement de fixation de l'allèle, et p_i la probabilité de fixation de l'allèle A partant de $A_0 = i$. Plus précisément :

$$\begin{aligned} F &:= \{A_\tau = 2N\} \\ p_i &:= \mathbb{P}(F | A_0 = i) \end{aligned}$$

Q 7 *Simuler une suite de réalisations indépendantes de la v.a. indicatrice $\mathbf{1}_F$. Calculer $\mathbb{E}(\mathbf{1}_F)$.*

Q 8 *En déduire un moyen de calculer numériquement une valeur approchée de p_i pour différentes valeurs de i. Les représenter sur un graphe.*

Q 9 *Faire une conjecture sur la fonction $p_i = f(i)$.*

2.5 Dérive et évolution de l'hétérozygotie

En biologie, un individu diploïde est dit hétérozygote lorsque les deux gènes homologues qu'il possède appartiennent à deux allèles différents. On introduit ici une nouvelle variable aléatoire H_k représentant l'hétérozygotie à la génération k :

$$H_k \quad := \quad \frac{\binom{X_k}{1}\binom{2N - X_k}{1}}{\binom{2N}{2}} = \frac{2X_k(2N - X_k)}{2N(2N-1)}$$

Intuitivement, ça correspond à tirer des paires de gènes sans remise, et à calculer la probabilité d'avoir obtenu deux allèles (un individu hétérozygote).

Q 10 *Simuler quelques trajectoires de $(H_t)_{0 \leq t \leq t_{max}}$.*

Q 11 *Au regard de ces premières questions, conclure quant à l'effet de la dérive sur la diversité allélique de la population.*

2.6 Ajout de mutations

On suppose à présent qu'à chaque génération, chaque gène a une probabilité μ de muter. Lorsqu'un gène mute, il change d'allèle, pour devenir un allèle jamais découvert auparavant. On parle alors de modèle à infinité d'allèle.

Q 12 *Introduire une variable aléatoire égale au nombre de nouveaux allèles découverts par génération. Donner sa loi, son espérance, sa variance.*

Q 13 *Proposer et implémenter une fonction* DeriveAvecMutation(N, μ, T_{max}) *permettant de simuler le nombre d'allèles dans la population entre 0 et T_{max} générations.*

Q 14 *Représenter des trajectoires pour des valeurs de μ et N différentes. Conclure quant à l'effet de la dérive et des mutations sur la diversité génétique de la population.*

Chapitre 3

Projet - Etude de séquences biologiques

Contents

L'essor des techniques de séquençage de l'ADN, dans les années 1970, a permis la multiplication de données génétiques, provenant de tous les domaines de la vie. Ces séquences sont analysées en routine à l'aide d'outils de bio-informatique permettant d'en rechercher les séquences potentiellement codantes, ou d'aligner différentes séquences entre elles de manière optimale. Ces alignements sont ensuite utilisés par exemple pour chercher des séquences homologues chez d'autres espèces, ou encore reconstruire la phylogénie de plusieurs espèces.

Remarquons que le terme de "séquences biologiques" ne se limite pas aux seules données génétiques. Les mêmes algorithmes sont employés pour des séquences de nucléotides ou d'acides aminés. On se limite dans ce sujet aux données génétiques pour la facilité de l'alphabet le composant (A,T,G,C).

Ce sujet présente en première partie deux méthodes de recherche de séquences codantes, qui sont plutôt rapides à implémenter. La seconde partie consiste à aligner deux séquences entre elles de manière optimale, et permet de coder un algorithme plus intéressant.

3.1 Recherche de séquences codantes

3.1.1 Rechercher d'ORF

On appelle ici *ORF* une "phase ouverte de lecture" ("Open Reading Frame" en anglais) une séquence d'ADN commençant par un codon d'initiation (ATG, GTG, TTG), se terminant par un codon STOP (TAA, TAG, TGA), et ne contenant aucun codon STOP en son sein. Notons qu'il y a trois phases de lecture décalées d'un nucléotide sur une même séquence.

On se limite pour plus de facilité à des séquences procaryotes sans intron.

On représente la séquence comme une chaîne de caractères dont chaque case est un nucléotide 'a', 't', 'g', ou 'c'.

Q 1 *Ecrire une ou plusieurs fonctions répertoriant toutes les ORFs de la séquence étudiée.*

FIGURE 3.1 – Les trois phases de lecture contiennent dans cet exemple 2 ORFs.

Q 2 *Améliorer cette fonction en ne renvoyant que les ORFs dont la longueur est supérieure à une valeur donnée, choisie selon vos à-priori de biologistes.*

Q 3 *Appliquer le code à une "vraie" séquence génétique.*

3.1.2 Méthode des biais

Cette deuxième méthode repose sur l'hypothèse que la distribution des nucléotides ne se fait pas "au hasard" dans une région codante, et est d'autant plus différente d'une distribution à l'équilibre que la séquence est soumise à sélection.

Q 4 *Proposer une fonction calculant la fréquence de chaque nucléotide dans la séquence.*

Q 5 *Calculer les fréquences des 4 nucléotides dans une fenêtre glissante de taille w autour du nucléotide i, et représenter ces fréquences en fonction de i.*

Q 6 *Rechercher graphiquement des régions potentiellement codantes dans la même séquence que précédemment. Se superposent-elles avec certaines ORFs ?*

3.2 Comparaison de 2 séquences

3.2.1 Tracé graphique "dotplot"

On appelle matrice de dotplot entre deux séquences de longueur m et n, une matrice de dimension m, n. Chaque position (i, j) de cette matrice prend une valeur booléenne *TRUE* si le nucléotide numéro i de la première séquence est le même que le nucléotide de numéro j de la seconde séquence. Dans le cas contraire, sa valeur est *FALSE*.

Q 7 *Réaliser le code permettant cette représentation graphique.*

Q 8 *Tester en représentant les dotplots de :*
- *une séquence contre elle-même.*
- *une séquence présentant un palyndrome avec elle-même.*
- *une séquence présentant des répétitions avec elle-même.*
- *deux séquences homologues.*

3.2.2 Alignement de 2 séquences suivant l'algorithme de Needleman & Wunsch

On souhaite ici mettre en correspondance de façon optimale les résidus de deux séquences, en autorisant des gaps (trous) dans les séquences alignées. L'alignement commence avec le premier résidu de chacune des séquences et se termine avec le dernier.

Le coût d'un alignement se calcule ainsi :

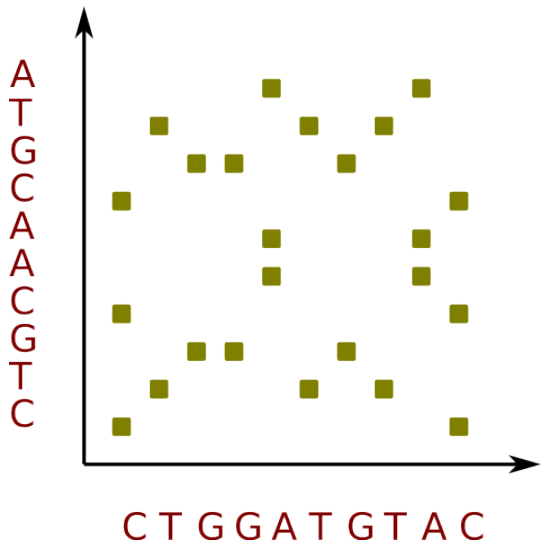

FIGURE 3.2 – Exemple de petit dotplot entre deux séquences.

— chaque mise en correspondance de deux nucléotides différents a un coût d (pour "différent").
— chaque gap introduit a un coût g.
— chaque nucléotide aligné avec le même nucléotide ne coûte rien.

Par exemple, l'alignement suivant :

AATGCGCCAT
AAC - CTC - AT

aurait un coût total de : $2d + 2g$.

On souhaite trouver l'alignement dont le coût est minimal.

La première idée consiste à réaliser tous les alignements possibles (exhaustif) et à ne conserver que le meilleur. Mais le nombre d'alignements possible augmente très vite avec la taille des séquences.

Le principe de la programmation dynamique est de décomposer le problème considéré en problèmes plus simples. Ici, pour aligner nos deux séquences de façon optimale jusqu'à leurs indices respectifs (i, j), on peut :

— les aligner de façon optimale jusqu'à $(i, j-1)$ et introduire un gap final dans la première séquence.
— les aligner de façon optimale jusqu'à $(i-1, j)$ et introduire un gap final dans la seconde séquence.
— aligner de façon optimale jusqu'à $(i-1, j-1)$ et mettre le résidu i en correspondance avec le résidu j.

On choisit alors l'action qui permet d'obtenir le coût minimal.

On appelle $P(i, j)$ le coût minimal de l'alignement jusqu'aux positions i de la première séquence et j de la seconde séquence. On initialise avec :

$$P(i, 0) = g * i$$
$$P(0, j) = g * j$$

Q 9 *Ecrire la fonction permettant de calculer la matrice P.*

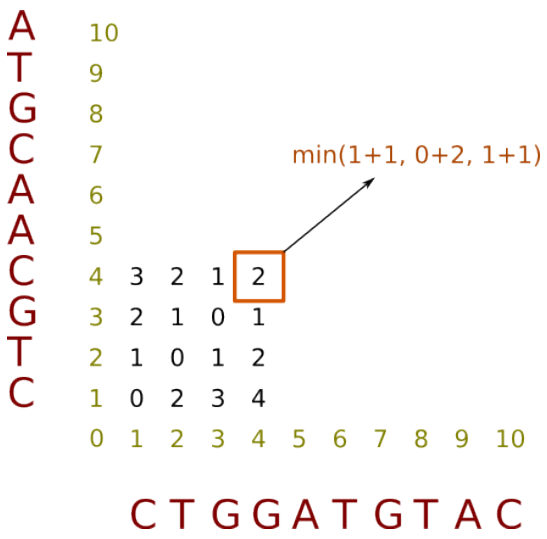

FIGURE 3.3 – Exemple de construction de la matrice des couts d'alignements optimaux, pour $(g, d) = (1, 2)$.

Q 10 *Utiliser cette matrice pour trouver un alignement "de bout en bout" optimal. Est-il nécessairement unique ? Ecrire une fonction récupérant tous les alignements optimaux.*

Q 11 *Choisir des coûts de mésappariements ou de gaps en fonction de vos connaissances biologiques et de ceux que vous voulez favoriser. Peut-être peut-on considérer moins coûteux un mésappariements entre deux bases présentant un taux de substitution élevé ?*

3.2.3 Idée pour aller plus loin

Pour ceux qui auraient fini, on peut envisager d'aligner plusieurs séquences ensemble (à l'aide d'un autre logiciel, sans détailler l'algorithme), puis d'analyser cet alignement multiple sous Python. Essayer par exemple d'obtenir une séquence consensus sur une famille de gènes...

Si vous avez vraiment beaucoup de temps, on peut également essayer de complexifier avec un coût différent pour les gaps : une pénalité d'ouverture forte, puis une pénalité d'extension plus faible. C'est plus réaliste biologiquement, mais l'algorithme devient un peu plus complexe...

Chapitre 4

Projet - Dynamique d'un système de proie-prédateur

Contents

4.1 Introduction

On utilise ce type de modélisation pour prédire l'évolution d'un système simple de particules appartenant à deux types, dont la première (la proie) se multiplie sans aide, tandis que la seconde (le prédateur) se nourrit de la proie pour se multiplier.

Nous emprunterons dans la suite le vocabulaire de l'épidémiologie :

S Susceptibles : la quantité de proies, qui sont susceptibles de se faire manger/infecter.

I Infectés : la quantité de prédateurs, qui sont susceptibles de manger/infecter les susceptibles.

Nous allons voir deux façons de modéliser un tel système. Dans un premier temps, nous le modéliserons de manière déterministe (les quantités de I et de S seront gourvernées par des équations différentielles). Puis nous le modéliserons de manière probabiliste (les individus pourront se reproduire, se manger ou mourir, avec des probabilités différentes). Et bien sûr, on pourra comparer les deux façons de faire !

4.2 Modélisation déterministe

4.2.1 Système d'équations différentielles

Les équations gouvernant la taille des réservoirs sont les suivantes :

$$\begin{aligned} \frac{dS}{dt} &= \alpha S - \beta SI \\ \frac{dI}{dt} &= \beta SI - \gamma I \end{aligned}$$

On appelle α le taux de naissance des individus susceptibles, β le taux d'infections, et γ le taux de décès des individus infectés.

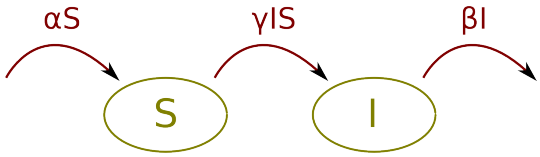

FIGURE 4.1 – Représentation schématique des flux entre les compartiments I et S sur un modèle déterministe.

Dans ces équations, le taux de naissance est multiplié par la quantité d'individus susceptibles, reflétant le fait que plus on a d'individus, plus on a de naissances. De même, le taux d'infection est multiplié par le nombre de proies et de prédateurs, ce qui reflète le nombre de "rencontres" entre nos deux types.

4.2.2 Comportement simple du système

Q 1 *Que se passe-t-il pour $S = 0$? $I = 0$?*

Q 2 *Expliquer ce qu'il se passe pour $\beta = 0$, et résoudre le système d'équations différentielles.*

Q 3 *On appelle état d'équilibre un état pour lequel $(\dot{S}, \dot{I}) = (0, 0)$. Trouver les états d'équilibre du systèmes.*

4.2.3 Résolution numérique des équations différentielles

Q 4 *Résoudre numériquement ce système d'équations différentielles à l'aide de la méthode d'Euler, avec une condition initiale $y0$ donnée, un pas de temps et une fenêtre temporelle données.*

Q 5 *Utiliser ensuite la fonction* odeint *du package* scipy.integrate *pour résoudre numériquement le système.*

Q 6 *Représenter des trajectoires $I = f(t)$ et $S = f(t)$ sur le même graphique. Quel est le comportement du système ?*

4.2.4 Visualisation de l'espace des phases

Q 7 *Représenter les trajectoires $I = f(S)$ pour différentes conditions initiales.*

On appelle "espace des phases" le champ de vecteurs de coordonnees $(\frac{dS}{dt}, \frac{dI}{dt})$ en fonction de S et I.

Q 8 *Représenter l'espace des phases à l'aide de la fonction* quiver*, et y superposer le tracé de quelques trajectoires $I = f(S)$.*

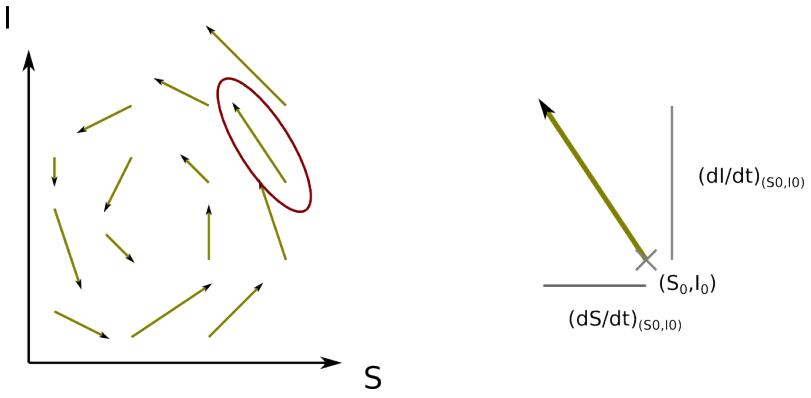

FIGURE 4.2 – Espace des phases d'un système (S, I). En chaque point, on représente le vecteur des vitesses $(\dot{S}, \dot{I})$. Cette représentation permet de bien comprendre le comportement du système autour d'un point d'équilibre.

4.3 Modélisation probabiliste

4.3.1 Enoncé du modèle

Dans cette partie, nous manipulerons directement les individus et les nombres d'individus (entiers naturels) de chaque type. Nous noterons N_S le nombre d'individus susceptibles et N_I le nombre d'individus infectés.

Les règles sont les suivantes :

— chaque individu S donne naissance, indépendamment de tous les autres individus, à un nouvel individu S à taux α.
i.e. le prochain temps auquel un individu donne naissance est indépendant de tous les autres, et suit une loi exponentielle de paramètre α.

— chaque individu I meurt, indépendamment de tous les autres individus, à taux γ.
i.e. le prochain temps auquel un individu meurt est indépendant de tous les autres, et suit une loi exponentielle de paramètre γ.

— chaque couple d'individus I-S est transformé en I-I par infections à taux β.
i.e. le prochain temps auquel il y a infection dans un couple est indépendant de tous les autres et suit une loi exponentielle de paramètre β.

4.3.2 Simulation du modèle

Q 9 *Pour une population de N_S individus, montrer que la loi du prochain temps de naissance dans l'ensemble de la population est une loi exponentielle de paramètre αN_S.*

De même, le prochain temps de mort suit une loi exponentielle de paramètre γN_I et le prochain temps d'infection suit une loi exponentielle de paramètre $\beta N_S N_I$.

Q 10 *A partir du générateur de nombre aléatoire* rand(), *écrire une fonction permettant de simuler une variable aléatoire de loi exponentielle de λ.*

Q 11 *Etant donné un état (N_S^0, N_I^0), écrire une fonction permettant de simuler le prochain temps auquel un événement de naissance, de mort ou d'infection a lieu.*

On admettra que l'on peut se reposer sur la propriété d'absence de mémoire de l'exponentielle pour simuler notre processus. Ainsi, après chaque événement ayant lieu à un instant t, on oublie ce qui s'est passé, et on simule le prochain événement ayant lieu sur notre nouvelle population (N_S^t, N_I^t).

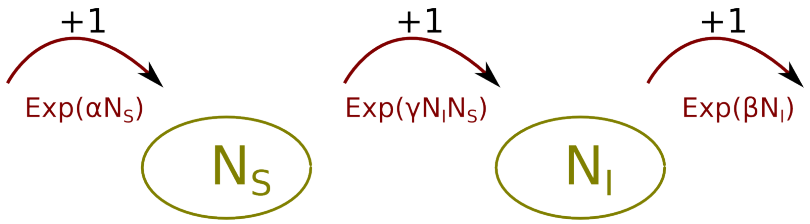

FIGURE 4.3 – Représentation schématique des flux entre les compartiments I et S sur un modèle stochastique.

Q 12 *Ecrire une fonction s'appuyant sur la précédente pour simuler l'évolution de* (N_S, N_I) *sur une durée* t. *Prenez garde à ce qui se passe lorsque* $N_S = 0$ *ou* $N_I = 0$.

Q 13 *Représenter les trajectoires aléatoires* $N_S = f(t)$ *et* $N_I = f(t)$ *superposées sur le même graphe. Jouer avec les paramètres pour observer des cycles d'infection.*

4.4 Comparaison des deux types de modélisation

Q 14 *Représenter des trajectoires temporelles aléatoires et déterministes sur le même graphe.*

Q 15 *Quels sont les comportements différents des deux types de modèles ?*

Chapitre 5

Projet - Recherche dans une image

Contents

5.1 Organisation du programme

Nous souhaitons réaliser un programme permettant de délimiter et compter des objets d'une certaine couleur donnée sur une image. Pour cela, il nous faudra :

— transformer une image en couleur en noir et blanc.
— délimiter des tâches de noir sur une image en noir et blanc.
— s'intéresser aux caractéristiques de surface et de nombre des tâches.

Notez que vous pouvez vous intéresser à chacune des parties de ce "programme" de façon séparée si vous le souhaitez. Soyez seulement cohérents sur les objets de sortie que vous attendez de chaque fonction. Le coeur du travail consistera à délimiter les objets sur l'image en noir et blanc.

5.2 Import d'une image en couleur

On utilisera pour importer une image le module *PIL*, puis le module *numpy* pour travailler sur des tableaux.

Q 1 *Essayer les quelques lignes de code ci-dessous pour se familiariser avec le format de nos données d'images sous python. C'est ce format matriciel qu'on utilisera par la suite.*

```
1 img = Image.open("levures.jpg")
2 arr = array(img)
3 r,g,b = img.split()
4 img2 = Image.fromarray(arr)
```

Q 2 *Transformer une image en couleur en une image en noir et blanc (matrice de 0 ou 1). Sur l'exemple "levure.jpg", on transformera en pixels de valeur 1 tous les pixels ayant une couleur rouge proche d'une couleur choisie de référence.*

5.3 Délimitation des objets

5.3.1 Première étape de marquage

La première étape de notre algorithme est dite "de marquage". Elle consiste à parcourir la matrice et à :

— identifier chaque nouveau patch par un nouveau numéro.
— identifier les taches de numéros différents qui ne sont qu'une (on dira que les taches sont voisines).

La figure 5.1 permet d'illustrer ce qui peut se passer lors du parcours du tableau.

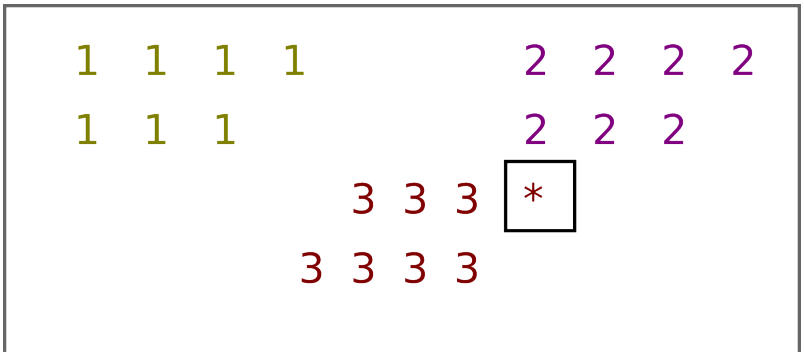

FIGURE 5.1 – Marquage des taches lors du parcours de notre tableau. L'arrivée sur le "3" entouré est l'instant où l'on se rend compte que nos deux taches ne font qu'une. On dira qu'elles sont voisines dans la suite.

Q 3 *Réaliser une fonction* Marquage*, prenant en argument une image en noir et blanc, et renvoyant :*
— *une image de même taille contenant les numéros des taches*
— *un vecteur contenant les relations de voisinage entre les taches*

Le vecteur V listant les relations de voisinage sera le suivant. A chaque numéro de tache i créé, on pose $V[i] = i$. Puis, lorsqu'un élément i est voisin de j, on posera $V[i] = j$.

Ainsi, l'exemple en figure sera noté : $V = (0, 1, 2, 3)$ avant d'arriver sur le "3" entouré. A cet instant, on se rend compte que 2 et 3 sont voisins, et on a alors $V = (0, 1, 2, 2)$.

5.3.2 Seconde étape de diffusion au voisinage

Lorsqu'une tache a est voisine d'une tache b, on notera $a \sim b$. On considère que :
— $a \sim a$.
— si $a \sim b$, alors $b \sim a$.
— si $a \sim b$ et $b \sim c$, alors $a \sim c$.

Ces règles définissent des cercles de voisinages distincts. Par exemple, avec les relations :

$$1 \sim 2 \quad , \quad 3 \sim 5 \quad , \quad 4 \sim 6 \quad , \quad 6 \sim 7$$

on obtient les groupes : $\{1, 2\}$, $\{3, 5\}$, $\{4, 6, 7\}$. On souhaite ne conserver qu'un numéro représentant chaque cercle de voisinage.

Q 4 *Ecrire une telle fonction, prenant un vecteur de relation de voisinage, et renvoyant un vecteur de relations de voisinage simplifié, où chaque cercle n'a qu'un unique représentant.*

Notons que dans l'exemple ci-dessus, la fonction doit nous transformer, par exemple, le vecteur $(1, 1, 5, 4, 5, 7, 4)$ en $(1, 1, 5, 4, 5, 4, 4)$.

Q 5 *Utiliser ce vecteur des voisinages simplifiés pour écrire une fonction permettant de ne conserver qu'un numéro pour chaque groupe de taches voisines, sur l'image. On appelera cette fonction* Diffusion.

On pourra utiliser, pour tester ce script, une matrice du style de la figure 5.2.

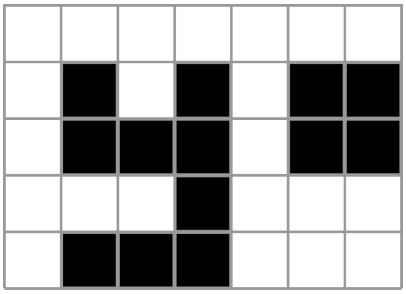

FIGURE 5.2 – Une idée de configuration de matrice pour tester vos scripts.

5.4 Caractéristiques des objets

5.4.1 Elimination du bruit

On se donne un pourcentage fixe p. Par exemple, 30%. On peut alors éliminer un bruit indésirable en supprimant tous les objets de taille inférieure à 30% fois la taille de l'objet le plus gros.

Q 6 *Ecrire une telle fonction* Elimination_bruit.

5.4.2 Nombre et surface de nos objets

Q 7 *Selon vos envies de biologistes en herbe, se servir des fonctions précédemment écrites pour calculer, par exemple :*

- *le nombre de colonies de bactéries sur une boite de pétri.*
- *la plus grosse cellule sur une microscopie.*
- *la surface totale de patchs déforestés avec une image satellite.*
- *...*

Chapitre 6

Projet - Phénomène de percolation

Contents

6.1 Introduction

On habille ce sujet avec de l'épidémiologie, mais ce pourrait également être un feux de forêt, ou un écoulement de fluide en milieu poreux, des migrations d'espèces vivantes dans un milieu, ... Un peu tout ce qui se propage de proche en proche.

Dans notre cadre d'épidémiologie, nous utiliserons dans la suite les lettres :

S pour les individus susceptibles (susceptibles d'être infectés, ce sont donc des individus sains).

M pour des individus immunisés à la naissance, qui sont sains et ne peuvent pas développer la maladie.

I pour les individus infectés.

R pour les individus ayant développé la maladie (ces individus là sont sains aussi, et ne changent plus d'état). Le "R" vient de "recovery". Notons qu'on pourrait aussi mettre des morts, si on est moins optimiste.

La population sera représentée par une matrice carrée de côté L.

6.2 Etablissement de la configuration initiale

On débute avec une matrice remplie d'individus S et M. On souhaite réaliser le pavage de notre matrice aléatoirement, de façon à ce qu'une proportion p choisie appartienne au type S.

Enfin, un premier individu S tiré au sort développe la maladie et devient I.

Q 1 *Ecrire une fonction créant une situation initiale aléatoire selon vos paramètres de choix.*

6.3 Propagation de la maladie

La maladie se propage de proche en proche dans la matrice. A chaque étape de propagation on parcourt toute la matrice et :

— tous les individus S voisins d'un individu I deviennent de type I.
— les individus qui étaient de type I au tour précédent deviennent R.

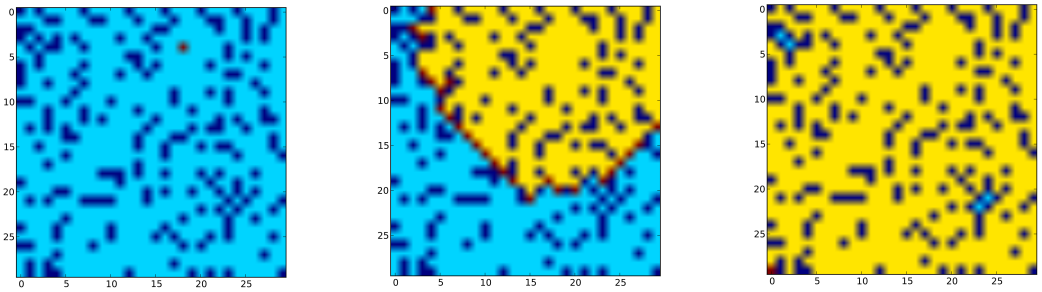

FIGURE 6.1 – Exemple de propagation de l'epidemie dans la population. En rouge : les infectés, en jaune : les guéris, en bleu clair : les suceptibles, et en bleu foncé les individus immunisés de naissance.

Q 2 *Ecrire une fonction de propagation permettant de réaliser une étape de propagation de la maladie.*

Q 3 *Ecrire une fonction permettant de tester si la maladie est en train de se propager dans une matrice.*

Q 4 *Mettre au point une fonction d'affichage d'une population (matrice). On pourra utiliser la fonction* imshow().

6.4 Utilisation du programme

Q 5 *Ecrire une fonction permettant de récupérer le nombre d'individus appartenant à chaque type (et, ce qui sera particulièrement intéressant, de type R) dans une matrice de population.*

Q 6 *Agencer les différentes fonctions pour s'intéresser, selon vos envies, à différentes questions :*
- *comment varie la durée moyenne d'une épidémie en fonction de p, de L ?*
- *comment varie la proportion moyenne d'individus touchés par une épidémie, en fonction de p, de L ?*
- *combien de départ d'épidémies sont nécessaires, en moyenne, pour que toute la population "susceptible" ait contracté la maladie ?*

Q 7 *Représenter les résultats à ces questions sous forme de graphiques clairs. On parlera de "seuil de percolation" p_c pour désigner la proportion d'individus S à partir de laquelle la propagation d'une maladie est "rapide"/"facile".*

Chapitre 7

Projet abandonné - Modèle d'épidémiologie déterministe

Contents

7.1 Présentation des modèles

On considère un modèle d'épidémiologie, permettant de représenter l'évolution de la maladie dans la population au cours du temps. On parle de modèle déterministe, car les flux d'individus entre les compartiments sont gouvernés par des équations différentielles.

On introduit tout d'abord le vocabulaire suivant pour les différents réservoirs de la population :

S pour "Susceptible" : l'ensemble des individus n'ayant jamais contracté la maladie appartient à ce réservoir.

I pour "Infectés" : l'ensemble des individus ayant contracté la maladie, et n'ayant pas encore guéri.

R pour "Recovered" : l'ensemble des individus ayant été infecté, mais ayant guéri et développé une immunité à la maladie.

On va étudier dans ce projets deux modèles différents, pouvant s'appliquer à des maladies différentes.

1. Dans le premier modèle, on considère une maladie sévère, dont on ne guérit pas. On peut penser par exemple à Ebola. Pour que le comportement soit intéressant, on prend donc en compte des naissances dans le compartiment S, et des morts dans le compartiment I.
2. Dans le second modèle, on s'intéresse à une maladie de type "grippe". On suppose la taille totale des réservoirs constants (pas de naissance, ni de morts à l'échelle de la propagation de la maladie). Et les individus du réservoir I peuvent passer dans le réservoir R.

7.2 Modélisation d'un système $S - I$ avec naissances et morts

7.2.1 Modèle

Les équations gouvernant la taille des réservoirs sont les suivantes :

$$\begin{aligned} \frac{dS}{dt} &= \alpha S - \beta SI \\ \frac{dI}{dt} &= \beta SI - \gamma I \end{aligned}$$

7.2.2 Comportement simple du système

Q 1 *Que se passe-t-il pour $S = 0$? $I = 0$?*

Q 2 *Expliquer ce qu'il se passe pour $\beta = 0$, et résoudre le système d'équations différentielles.*

7.2.3 Résolution numérique des équations différentielles

Q 3 *Résoudre numériquement ce système d'équations différentielles à l'aide de la méthode d'Euler, avec une condition initiale $y0$ donnée, un pas de temps et une fenêtre temporelle données.*

Q 4 *Utiliser ensuite la fonction* odeint *du package* scipy.integrate *pour résoudre numériquement le système.*

Q 5 *Représenter des trajectoires $I = f(t)$ et $S = f(t)$ sur le même graphique. Quel est le comportement du système ?*

Q 6 *Représenter les trajectoires $I = f(S)$ pour différentes conditions initiales.*

On appelle "espace des phases" le champ de vecteurs de coordonnees $(\frac{dS}{dt}, \frac{dI}{dt})$ en fonction de S et I.

Q 7 *Représenter l'espace des phases à l'aide de la fonction* quiver*, et y superposer le tracé de quelques trajectoires $I = f(S)$.*

7.2.4 Interprétation biologique

On s'intéresse au temps écoulé entre deux pics d'épidémie.

Q 8 *Etudier l'influence des conditions initiales, ainsi que des paramètres (α, β, γ) sur le temps entre deux épidémies.*

Q 9 *Interpréter ces informations en terme de politique publique de gestion des épidémies, ou d'évolution des systèmes proies/prédateurs d'une façon générale dans l'histoire de la vie.*

7.3 Modélisation d'un système $S-I-R$ à taille de population constante

7.3.1 Modèle

Les équations gouvernant la taille des réservoirs sont les suivantes :

$$\begin{aligned} \frac{dS}{dt} &= \beta SI \\ \frac{dI}{dt} &= \beta SI - \gamma I \\ \frac{dR}{dt} &= \gamma I \end{aligned}$$

Q 10 *Expliquer la signification des différents paramètres et le comportement attendu du modèle.*

7.3.2 Etude numérique

Q 11 *Résoudre numériquement ce système d'EDO. Représenter de façon superposée les solutions $S(t)$, $I(t)$ et $R(t)$.*

Q 12 *Etudier des caractéristiques de l'épidémie en fonction des paramètres. Par exemple : le nombre maximum d'infectés, le temps pendant lequel une certaine proportion de la population est infectée, ...*

Q 13 *Interpréter biologiquement les résultats obtenus.*

7.4 Elements de correction, type de graphes obtenus

Pour les équations de Lotka-Volterra, les types de graphes obtenus sont présentés en figures 7.1 et 7.2.

La seconde partie est plus simple à interpréter, et ne repose que sur le type de graphe présenté en figure 7.3.

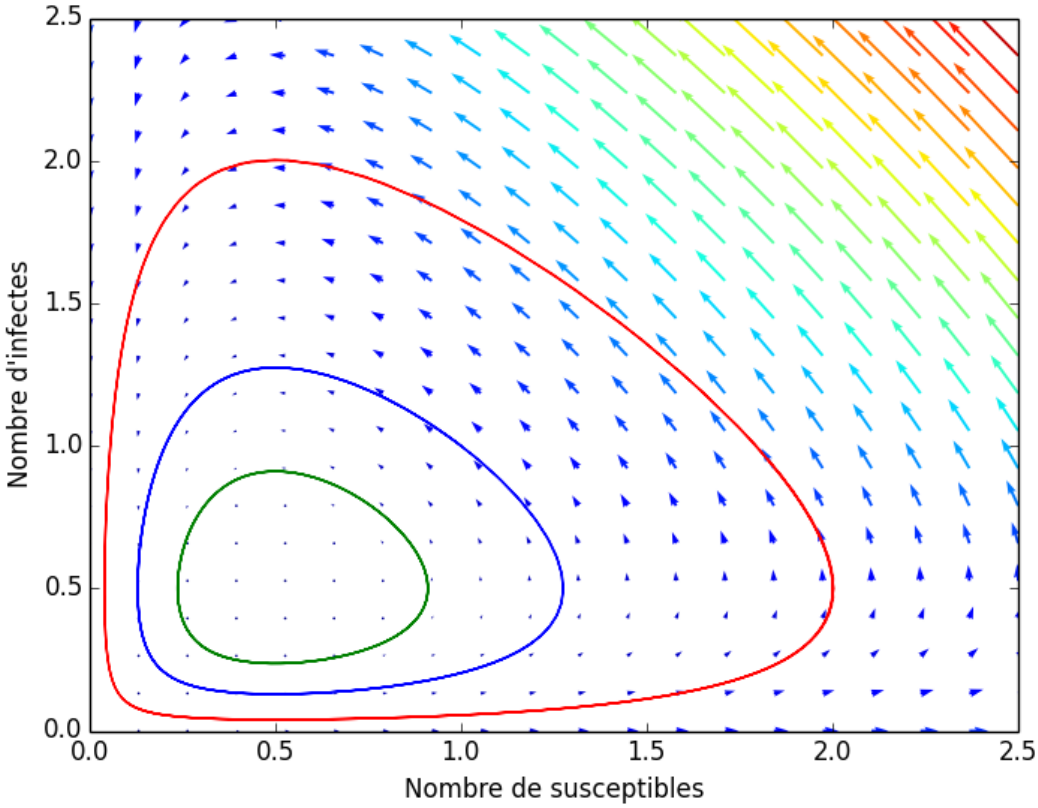

FIGURE 7.1 – Trajectoires dans l'espace des phases.

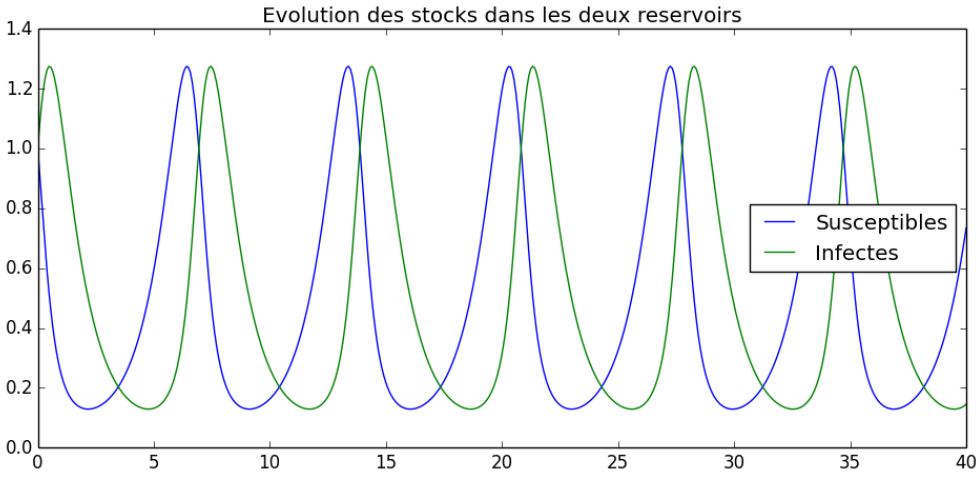

FIGURE 7.2 – Trajectoires cycliques en fonction du temps.

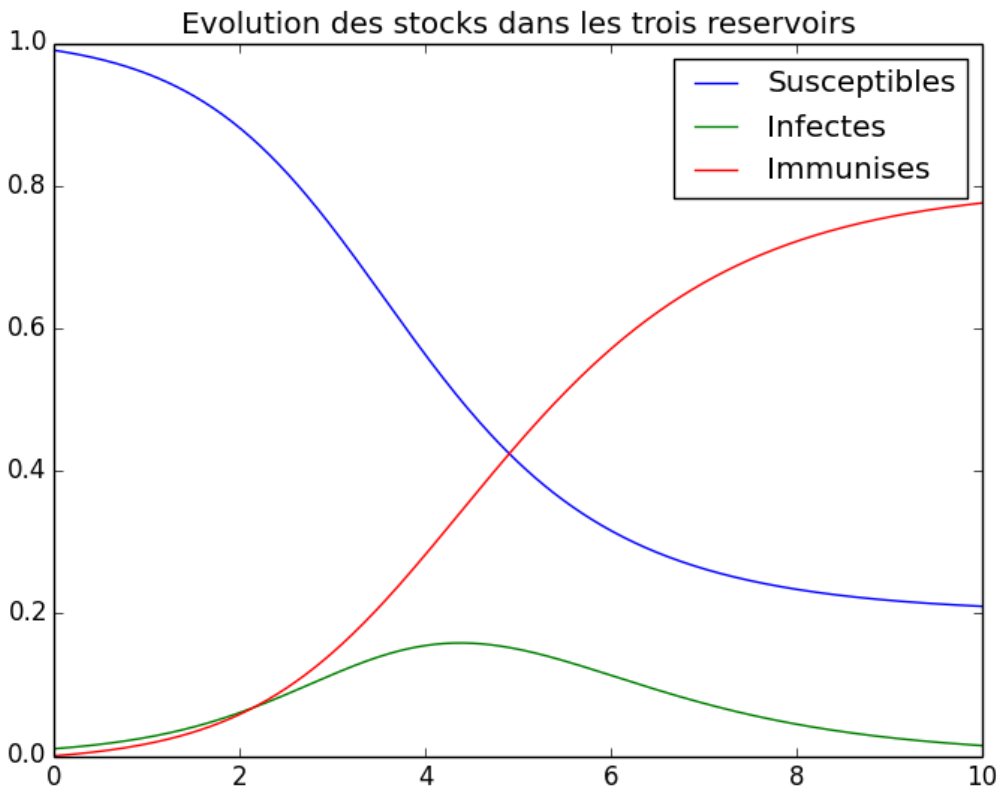

FIGURE 7.3 – Trajectoires du modèle SIR en fonction du temps.

Chapitre 8

Projet abandonné - Etude des relations de parenté sur le modèle de Wright-Fisher

Contents

Dans cette partie, on s'intéresse à retracer l'histoire généalogique de quelques individus choisis de notre population. Le temps est toujours compté en nombre de générations, le temps actuel est encore $t = 0$, mais le temps s'écoule cette fois-ci vers le passé, de sorte qu'on remonte dans le temps lorsque t croît.

8.1 Coalescence de deux lignées

Soient deux gènes de la population au temps 0. Appelons :

$$\begin{aligned} E_2^i &:= \text{l'événement "les deux gènes ont le même parent à la génération i"} \\ t_2 &:= \min(i \in \mathbb{N}^*, \quad E_2^i \text{ a lieu}) \end{aligned}$$

Q 1 *Calculer $\mathbb{P}(E_2^i)$, en déduire, $\forall k \in \mathbb{N}^*$, l'expression de la probabilité $\mathbb{P}(t_2 \geq k)$. Quelle est la loi de t_2 ? Calculer son espérance et sa variance.*

Q 2 *Proposer un algorithme permettant de simuler des réalisations de la variable aléatoire t_2 à partir d'un générateur de nombre aléatoire uniforme. L'implémenter.*

8.2 Arbre généalogique de n individus

On souhaite à présent connaître l'arbre généalogique d'un échantillon de n gènes au lieu de 2. Lorsque deux gènes rencontrent un ancêtre commun, on dit que les deux lignages coalescent en un seul. Un exemple est montré en figure 8.1. On pose :

$$t_n := \text{temps pendant lequel on a } n \text{ lignages}$$

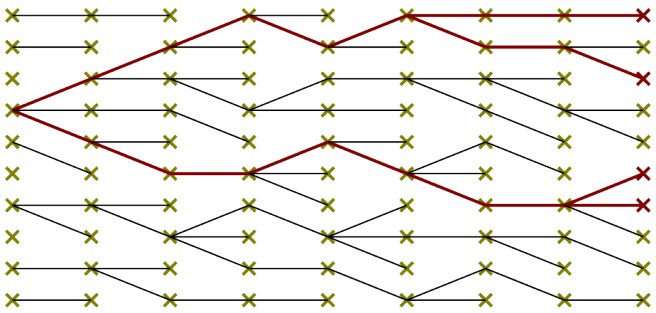

FIGURE 8.1 – La coalescence dans le modèle de Wright-Fisher.

Q 3 *Calculer la probabilité que deux lignages coalescent à la génération précédente. Calculer la probabilité que trois lignages coalescent en un seul à la génération précédente. On négligera dans la suite cette probabilité.*

Q 4 *Déterminer la loi de t_n.*

Un arbre peut être codé numériquement comme une chaîne de caractère, avec des parenthèses. Un formalisme particulier a été créé : celui de Newick que nous allons utiliser ici. Ce formalisme est présenté en figure 8.2.

Q 5 *Réaliser et implémenter l'algorithme permettant de simuler l'arbre généalogique de nos n individus.*

Q 6 *Utiliser la bibliothèque python* Bio.phylo *et notamment les fonctions* Phylo.read *et* Phylo.draw*, pour représenter l'arbre généalogique obtenu.*

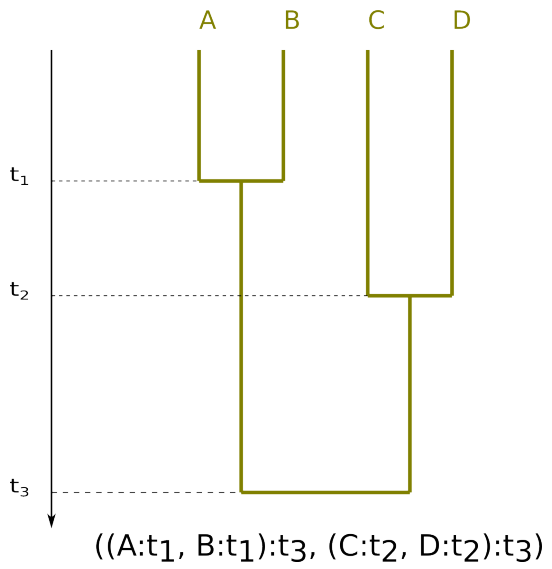

FIGURE 8.2 – Formalisme *Newick* de représentation d'arbre.

8.3 Temps écoulé avant de trouver le plus récent ancêtre commun, et longueur totale de l'arbre

On appelle T^n_{MRCA} le temps passé avant de trouver le plus récent ancêtre commun à nos n gènes.

Q 7 *Exprimer T^n_{MRCA} en fonction des $(t_j)_{2\leq j\leq n}$. En déduire une expression analytique de $\mathbb{E}(T^n_{MRCA})$.*

Q 8 *Simuler numériquement la distribution de T^n_{MRCA}, en représenter l'histogramme, et calculer la moyenne empirique.*

On appelle L^n_{tot} la longueur de branche totale de notre arbre généalogique.

Q 9 *Exprimer L^n_{tot} en fonction des $(t_j)_{2\leq j\leq n}$. En déduire une expression analytique de $\mathbb{E}(L^n_{tot})$.*

Q 10 *Simuler numériquement la distribution de L^n_{tot}, en représenter l'histogramme, et calculer la moyenne empirique.*

8.4 Ajout de mutations sur le coalescent : nombre de sites ségrégeant

On suppose un modèle de mutation à nombre de site infini. A chaque génération, chaque gène a une probabilité μ de muter. Lorsqu'il mute, un nouveau site (=nucléotide) de la séquence, jamais touché auparavant, est transformé.

Notons qu'un modèle à infinité de site implique un modèle à infinité d'allèle.

Q 11 *En supposant une longueur de locus finie L de nucléotides, et un gène ayant subi dans son histoire évolutive k mutations, calculer la probabilité qu'aucun nucléotide ne soit touché plus d'une fois. Choisir des valeurs de L et k crédibles pour critiquer le modèle.*

Malgré tout, on continue avec le modèle à infinité de site. On appelle nombre de site ségrégeant, et on le note S_n, le nombre de nucléotides présentant des différences au sein des n gènes de notre échantillon.

Q 12 *Calculer, pour les valeurs de L et i pour lesquelles ça a un sens, $\mathbb{P}(S^n = i|L^n_{tot} = L)$. En déduire une manière de simuler numériquement le nombre de site ségrégeant.*

Q 13 *Calculer numériquement une approximation de $\mathbb{E}(S^n)$.*

8.5 Ajout de mutations sur le coalescent : nombre d'allèles

On s'intéresse à présent à K^n, le nombre d'allèles au sein de notre échantillon de n gènes. Pour cela, on introduit la variable aléatoire ξ^n égale au temps d'attente avant de rencontrer une mutation dans l'histoire évolutive de n gènes.

Q 14 *Donner la probabilité qu'un gène mute à la génération précédente. Que deux gènes mutent ? Trois ? Dans la suite, on supposera μ très faible, et on négligera les événements de mutation multiple sur la même génération.*

Q 15 *A partir de l'approximation précédente, donner la loi de ξ^n. Quelle est son espérance, sa variance ? Comment la simuler ?*

On s'intéresse à l'hétérozygotie dans la population. Pour cela, on tire $n = 2$ individus dans la population. Si les deux gènes appartiennent au même allèle, l'individu créé est hétérozygote. Dans le cas contraire, il est homozygote.

Q 16 *Calculer $\mathbb{P}(\xi^2 \leq t_2)$. En déduire une valeur analytique de l'hétérozygotie dans la population. Comparer la valeur théorique avec celle obtenue par simulation.*

Q 17 *Concevoir et implémenter un algorithme permettant de simuler numériquement des réalisations de la variable aléatoire K^n.*

Q 18 *Approcher numériquement $\mathbb{E}(K^n)$ pour différentes valeurs de N et μ. Qu'en déduisez-vous sur l'effet de la dérive et des mutations sur la diversité génétique ?*

8.6 Elements de correction

8.6.1 Idées d'ouverture si les élèves avancent trop bien

— Faire intervenir des recombinaisons entre individus, dans un Wright-Fisher avec reproduction sexué.
— Faire varier la taille de la population au cours du temps, et voir le changement de forme de l'arbre.

8.6.2 Type de graphes à obtenir

Pour des tracés d'histogrammes empiriques, et pour se donner une idée de la loi, on peut utiliser :

```
1 hist(ma_distribution, bins=100, norm=True)
```

Un exemple est présenté en figure 8.3.

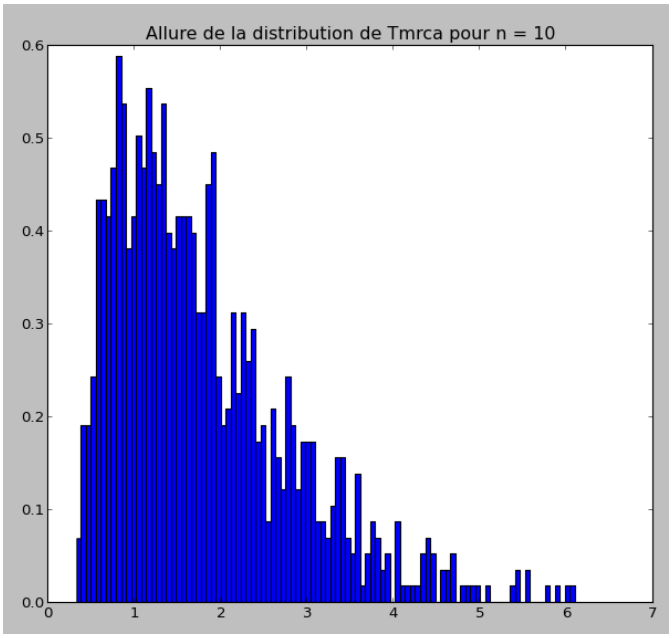

FIGURE 8.3 – Exemples de représentation d'une distribution empirique.

Les résultats de tracés graphiques d'arbres généalogiques doivent ressembler, eux, à ce qui est montré en figure 8.4.

8.6.3 Remarques diverses

La première remarque concerne les différents temps d'attente. Dans ce type de modèle, on a plutôt l'habitude d'approcher ça, en supposant $N \to \infty$, par des temps d'attente exponentiels. Peut-être qu'on peut le faire ici aussi avec les élèves de BCPST. Ca peut être l'occasion de rajouter une ou deux questions, et de travailler la simulation d'une variable aléatoire continue par inversion de la fonction de répartition.

La réalisation de l'algorithme pour créer un arbre, et ensuite créer des arbres séparés à cause des rencontres avec des mutations, peut poser problème. On peut travailler des schémas qui resemblent à ce que j'ai en figure 8.5 ou 8.6 pour comprendre ce qui se passe.

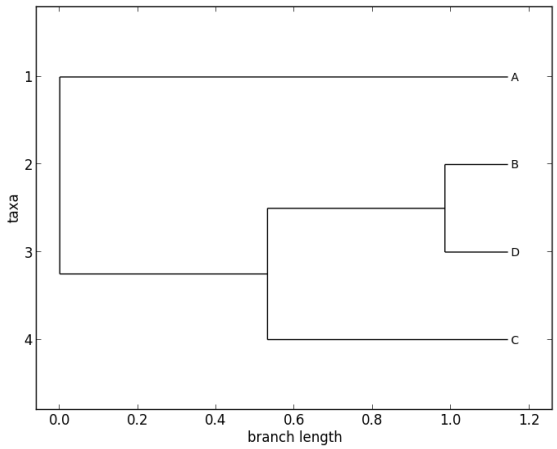

FIGURE 8.4 – Exemple de représentation d'arbre.

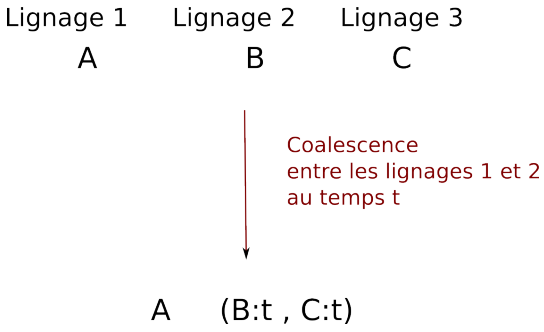

FIGURE 8.5 – Algorithme de réalisation d'un arbre.

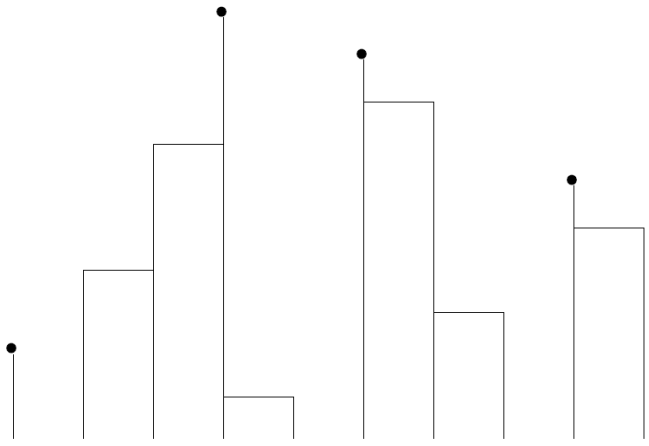

FIGURE 8.6 – Algorithme pour trouver la partition allélique sur un coalescent. Chaque première mutation rencontré détermine l'allèle. Figure tirée du livre *Probability models for DNA sequence evolution - Rick Durrett - 2008*