

Projet 2 : à la recherche de la petite bête

Quelle est cette pathologie mystérieuse ? Nous sommes le 1er janvier 2020, et une équipe vient de séquencer des échantillons pulmonaires de personnes présentant une maladie respiratoire atypique. Cette question hante le laboratoire de recherche dans lequel vous vous trouvez.

Les données sont accessibles [à cette adresse](#) (et les données brutes de séquençage peuvent directement être [récupérées ici](#)).

Votre but sera d'identifier à quel virus ressemble le pathogène présent dans cet échantillon pulmonaire. Mais attention, l'échantillon récupéré contient principalement des cellules humaines, on s'attend à ce que seule une petite fraction corresponde à des séquences du pathogène.

Parmi les pathogènes envisagés, nous allons chercher si on trouve des séquences semblables à ces virus : grippe, rhinovirus, HIV et coronavirus.

Voici les adresses auxquelles vous pouvez récupérer les séquences des génomes correspondants :

- [Grippe A](#)
- [HIV-1](#)
- [Rhinovirus](#)
- [Alpha-coronavirus](#)

Dans tous les cas, téléchargez le fichier dont l'extension est `.fna.gz` (c'est un fichier FASTA compressé) et décompressez-le (par exemple avec la commande `gunzip`).

Le but est donc d'aligner les reads sur chacun de ces génomes et de voir sur quel génome, le plus de reads sont alignés. L'alignement que vous développerez consiste en une stratégie *seed and extend*.

seed On extrait des courtes séquences (des k -mers) du read et on essaie de les trouver, sans tolérer d'erreur, dans le génome d'intérêt.

extend Une fois une graine trouvée, on essaie d'aligner le read en entier sur cette région du génome.

Comment aligner ?

Il existe de nombreuses façons de réaliser cet alignement. Néanmoins, un préalable est de pouvoir rechercher des k -mers dans un génome et de connaître leur position dans ce génome. Pour cela, le génome devra être indexé, nous allons choisir une table des suffixes.

Ensuite, pour la phase d'extension, vous pourrez utiliser des bibliothèques existantes qui permettent de réaliser de l'alignement semi-global de séquences. Par exemple la bibliothèque `parasail` est efficace et disponible aussi bien en Python qu'en C/C++, Rust ou Java.

Passage à l'échelle

Dans un premier temps vous alignerez vos reads sur de petits génomes (des génomes de virus), néanmoins votre programme doit idéalement pouvoir passer à l'échelle sur de plus grands génomes... par exemple de chauve-souris. Attention donc à la mémoire dont vous aurez besoin.

D'autre part, vous avez un grand nombre de reads à traiter, le traitement ne doit donc pas être trop long (en Python, **il ne faut pas s'attendre à traiter beaucoup plus de 1 000 reads par seconde**). Pour optimiser le traitement, penser à ce qui vous permettra de gagner du temps :

- il n'y a pas forcément besoin d'essayer tous les k -mers d'un read : entre un k -mer et le suivant dans un read, $k - 1$ nucléotides sont communs. Il y a donc peu de nouvelle information en passant d'un k -mer au suivant. On pourrait donc envisager de sauter des k -mers sans perte d'information et ce qui permettrait de gagner du temps
- de même, si on souhaite gagner de la place, il n'est peut-être pas indispensable d'indexer tous les suffixes mais on pourrait n'en indexer qu'une fraction (à l'inverse, il faudrait dans ce cas probablement interroger tous les k -mers pour compenser la perte d'information dans l'index).

*Combien de temps pour
faire tourner votre
programme ?*

De manière générale, avant de foncer vers une solution, il faut toujours se poser la question de son passage à l'échelle — à la fois en temps de calcul et en espace nécessaire — sur de grands génomes (en milliards de nucléotides) ou sur de grands jeux de données, en centaines de millions de reads.

Calcul de la table des suffixes

Pour calculer la table des suffixes, il ne faut pas stocker l'ensemble des suffixes du génome (à aucun moment). Nous avons vu en cours qu'un tel stockage demande un espace quadratique ($\Theta(n^2)$), ce qui n'est pas envisageable sur des génomes.

Pour trier une table des suffixes, sans énumérer les suffixes, il suffit de commencer par une table d'entiers de 0 à $n-1$ (par exemple `[i for i in range(len(genome))]`). Ensuite il faut trier ce tableau. Mais ce qu'on souhaite ce ne sont pas que les entiers eux-mêmes soient triés mais les suffixes qui commencent aux positions désignées par ces entiers. Ainsi, si on a deux entiers i et j dans ce tableau, on ne souhaite évidemment pas comparer i et j pour les trier mais on souhaite comparer `genome[i:]` avec `genome[j:]`. Pour cela, en Python, vous pouvez passer une fonction dans le paramètre `key` de la fonction `sorted` (voir [la documentation Python](#)).

Exemple (je vous laisse comprendre ce que fait ce code) :

```
1 def compare_my_tuples(tuple):
2     return tuple[1]
3
4 l = [(2.5, 'A'), (4.1, 'D'), (1.1, 'C'), (3.7, 'F'), (2.8, 'B')]
5 sorted(l, key=compare_my_tuples)
```

Lire des fichiers FASTQ compressés

En ligne de commande, vous pouvez regarder le contenu d'un fichier FASTQ compressé sans le décompresser, par exemple avec la commande `zless` :

```
1 zless fichier.fastq.gz
```

Il faut ensuite appuyer sur `q` pour quitter le programme.

Vous pouvez également directement lire des fichiers FASTQ compressés via Python, avec le module `BioPython` (à installer via `Thonny`, si vous utilisez cet éditeur, ou en ligne de commande via `pip : pip install biopython`). Par exemple pour afficher toutes les séquences du fichiers FASTQ :

```
1 with gzip.open('fichier.fastq.gz', 'rt') as fastq:
2     for record in SeqIO.parse(fastq, 'fastq'):
3         print(record.seq)
```

Attention, à noter que `record.seq` n'est pas de type `str`, il faut donc convertir cet objet si vous souhaitez l'utiliser comme une chaîne de caractères.

Le type renvoyé par `SeqIO.parse` est un objet `SeqRecord`, dont [la documentation se trouve ici](#).

Connaître l'avancée de son programme

Dans ce type de programme, le temps d'exécution est généralement proportionnel au nombre de reads. Mesurer le temps de calcul de votre programme sur 10 000 reads suffit à avoir une idée du temps nécessaire pour traiter 1 read, et donc à en déduire le temps qui sera nécessaire sur le jeu de données complets.

Par ailleurs, en Python, il est assez facile de connaître la vitesse d'exécution d'une boucle pendant l'exécution à l'aide du package `tqdm`.

Vous pouvez l'essayer avec ce bout de code. Il suffit de passer par la fonction `tqdm.tqdm` pour générer une sortie qui donne le nombre d'itérations par seconde :

```
1 import tqdm
2 j=0
3 for i in tqdm.tqdm(range(int(10e8))):
4     j=j+i
```

Une autre utilisation de `tqdm` :

```

1 import tqdm
2 import time
3
4 progress = tqdm(total = 100)
5 for i in range(100):
6     time.sleep(.1) # Juste pour ralentir le programme pour voir l'effet de tqdm
7     progress.update(1)
8 progress.close()

```

Utiliser parasail

La bibliothèque parasail fournit de nombreuses fonctions (voir la [documentation](#)), selon le type d'alignement souhaité, ainsi que selon les informations désirées sur l'alignement.

```

1 import parasail
2 al = parasail.sg_dx_scan_sat("TGTTGCATCTTCAGCTAGTCTGGAGCAA",
3                             "GGACAGGGCTTTGAGTGGATGTGATGGATCATCACCTACCTGGAACCCAACGTATACC",
4                             5, 1, parasail.dnafull)
5 print(al.score)

```

Cette fonction permet uniquement d'accéder au score de l'alignement mais pas à l'alignement lui-même (ce qui demande de calculer la traceback, ce qui est plus coûteux). Si jamais une telle traceback est nécessaire, il existe d'autres fonctions pour y accéder. Par exemple :

```

1 al = parasail.sg_dx_trace_scan_sat("TGTTGCATCTTCAGCTAGTCTGGAGCAA",
2                                   "GGACAGGGCTTTGAGTGGATGTGATGGATCATCACCTACCTGGAACCCAACGTATACC",
3                                   5, 1, parasail.dnafull)
4 traceback = al.get_traceback()
5 print(traceback.ref+"\n"+traceback.query)

```

Multi-threading

Un ordinateur possède parfois plusieurs processeurs et même plusieurs cœurs sur chaque processeur. Chaque cœur peut traiter une tâche indépendante. Par défaut, un programme ne s'exécute que sur un seul cœur. Un certain type de programmation (appelé SIMD pour *Single Input Multiple Data*) permet d'exploiter au maximum les capacités d'un processeur, c'est ce que fait la bibliothèque parasail, mais la programmation SIMD est complexe.

On peut également faire un sorte qu'un programme lance ces calculs sur plusieurs processeurs. En Python, la bibliothèque multiprocessing permet cela.

```

1 from multiprocessing.pool import ThreadPool
2 from time import sleep
3 from random import random
4
5 array = [("AACG", 1), ("AGGA", 2), ("TGCA", 3), ("GGAA", 4), ("ATTT", 5), ("CCAC", 6),
6         ("GCAC", 7), ("TAGG", 8)]
7
8 def process(data1, data2):
9     print(data1, data2)
10    sleep(1)
11
12 pool = ThreadPool(processes=2)
13 for data in array:
14     pool.apply_async(process, data)
15 pool.close()
16 pool.join()

```

Le code précédent lance la fonction process sur deux cœurs. Il y a donc deux appels à la fonction qui peuvent être faits en parallèle sur deux cœurs. Si vous lancez l'exécution de ce code, vous verrez que les affichages se feront en général deux par deux. Notez qu'avec un tel code tqdm n'est pas utile, car la boucle s'exécute en une seule traite, mais les processus sont ensuite mise en attente. L'exécution apparaîtrait donc immédiatement comme complétée, alors qu'elle ne l'est pas réellement.

Réflexion sur la qualité des alignements

Un algorithme d'alignement produira toujours un alignement. Le fait d'obtenir un alignement n'est donc pas une garantie de qualité de celui-ci. Il faudra donc vous poser la question

de la pertinence des alignements obtenus et des critères, les plus flexibles possibles, afin de déterminer si un alignement est satisfaisant. Par exemple, dire qu'un score de 100 est un seuil minimum pour considérer un alignement comme de qualité n'est pas très satisfaisant car un tel score peut varier en fonction de la longueur de l'alignement, de la longueur des reads et du jeu de score employé. Il faut donc trouver un critère qui soit à la fois pertinent et qui s'adapte au jeu de score employé ou à la longueur des séquences à aligner.

Commencer petit

Le fichier que vous avez récupéré contient beaucoup de séquences. Afin de tester votre programme, il serait raisonnable de ne commencer par le lancer que sur des petits jeux de données afin d'avoir un retour rapide et de ne pas attendre plusieurs minutes, voire plusieurs heures, avant de savoir si le résultat de votre programme vous convient.

Afin d'évaluer la pertinence de vos choix, vous pouvez concevoir des jeux de données synthétiques, dont vous connaissez le contenu, afin de vérifier que les résultats de votre programme sont cohérents avec vos attentes.

Des programmes permettent de simuler (grossièrement) le comportement des séquenceurs et peuvent générer des fichiers de reads. L'un d'eux est `art`, que vous pouvez installer en utilisant `conda`.

1. Installer `miniconda`

2. Puis installer `art` avec la commande suivante : `conda install -c bioconda art`

Ensuite, une utilisation classique est :

```
1 art_illumina -ss MSv3 -c 10000 -i ref.fa -l 250 -o reads
```

Cela permettra de générer un fichier `reads.fq` contenant les reads simulés, ces reads auront une longueur de 250 (-l 250) et seront issus des séquences présentes dans le fichier `ref.fa`, 10 000 reads de chaque séquence sera généré (-c 10000).

Afin de générer des reads provenant de l'espèce humaine, qui ne devraient pas s'aligner avec des espèces de virus, on aura besoin de récupérer un fragment du génome humain. La commande ci-dessous permet de récupérer une partie du chromosome 1 humain (accession NC_000001.11) entre les positions 15 400 000 et 15 900 000 et stocke la séquence dans le fichier `chr1_human.fa`.

```
1 wget -O chr1_human.fa
2 'https://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.efetch.fcgi?db=nucleotide&id=NC_000001.11&rettype=fasta&retmode=text&seq_start=15400000&seq_stop=15900000'
```

Il vous faut ensuite constituer un fichier qui contiendra les séquences de référence sur lesquelles construire votre jeu de référence. Par exemple `cat chr1_human.fa rhinovirus.fa > ref.fa` pour mettre les séquences présentes dans les fichiers `chr1_human.fa` et `rhinovirus.fa` dans un même fichier `ref.fa`. On peut ensuite simuler des reads à partir de cette référence. Une fois simulés, la provenance des reads pourra être obtenue grâce au nom des reads qui contiennent soit le numéro d'accèsion du chromosome 1 humain, soit celui du rhinovirus.

À titre d'exemple, [on vous fournit](#) un fichier de séquençage simulé contenant 10 000 reads humains et 10 000 reads de rhinovirus. En utilisant `art` vous pourrez composer vos propres fichiers de tests.

Entrée et sortie

Votre programme prendra en entrée un génome au format FASTA, ou multi-FASTA ainsi qu'un jeu de reads au format FASTQ. En sortie, votre programme écrira un fichier contenant uniquement les reads alignés sur le génome dans un format tabulé, détaillé après.

Votre programme acceptera donc les paramètres suivants **obligatoires** :

- -genome : chemin vers un fichier FASTA contenant le génome de référence
- -reads : chemin vers un ou plusieurs fichiers FASTQ.gz contenant des reads à aligner sur le génome de référence
- -out : fichier de sortie

Il acceptera également des paramètres optionnels, dont :

- -k : la taille des *k*-mers utilisés

Format de sortie

Le fichier de sortie produira une ligne par read aligné sur le génome de référence. Chaque ligne sera séparée en différentes colonnes, séparées par des tabulations.

Voici la liste des colonnes :

- Nom du read dans le fichier FASTQ
- Position (approximative) du début de l'alignement sur le génome de référence
- Score de l'alignement
- Séquence du read
- (optionnel) [CIGAR de l'alignement](#) (fourni par `parasail`)

Nouveau jeu de données

Après la réussite de la première étape, et de l'identification du virus, on vous donne accès à un nouveau jeu de données qui permettrait de remonter à l'origine de cette pandémie. Des échantillons prélevés sur des chauves-souris dans un endroit encore tenu secret ont été expédiés à quelques laboratoires autour du monde pour analyse.

Votre laboratoire, reconnu pour la qualité de ses analyses, figure parmi les heureux élus : vous avez réceptionné un échantillon il y a quelques jours. L'équipe du *wet lab* s'est immédiatement mise au travail pour mettre en place le séquençage de ces échantillons provenant de chauve-souris.

Le séquenceur vient de terminer son travail et [vous récupérez un fichier FASTQ](#) (ou [un extrait comportant un million de reads](#)) contenant les reads produits.

À l'aide de votre outil, retirez tous les reads correspondant au SARS-CoV-2 et à la chauve-souris il doit rester un certain nombre de reads qui, pour diverses raisons, n'ont pas été filtrés.

Il est normal que des reads ne soient pas filtrés. Cela peut être à cause de reads comportant trop d'erreurs de séquençage, à cause de contamination du séquençage par un autre séquençage, des morceaux d'adaptateurs qui ont été séquencés, ...ou une autre espèce qui était réellement présente dans l'échantillon.

Pour savoir ce qu'il en est, déterminez quels sont les k -mers les plus fréquents parmi les reads non filtrés¹ et déterminez s'ils appartiennent à une espèce donnée. Si c'est le cas vous filtrerez également les reads appartenant à cette espèce.

C'est très bien de faire cette partie également, mais il ne faut pas forcément avoir pour objectif de tout faire.

Rendu

Le rendu de votre travail est attendu pour le **lundi 25 mars 12h**.

Ce rendu devra comporter les éléments suivants :

- Votre code source permettant de réaliser les différentes tâches demandées
- Un fichier README expliquant la manière d'utiliser (voire compiler s'il y a lieu) vos programmes
- Des tests (par exemple des doctests en Python) permettant de vérifier le bon fonctionnement de vos fonctions, mais aussi des tests fonctionnels qui vérifient, sur de tous petits exemples, que votre programme produit le fichier de sortie attendue,
- Un petit rapport (au format Markdown, par exemple) expliquant votre travail
 - les structures de données utilisées,
 - l'implantation choisie pour ces structures de données,
 - la raison de ces choix,
 - les manières de favoriser le passage à l'échelle de vos programmes,
 - la façon dont vous avez choisi les paramètres pour vos programmes, ainsi que les évaluations réalisées afin de déterminer ces paramètres,
 - les résultats obtenus (nombre de reads alignés pour chaque espèce sur les différents jeux de données (y compris le jeu de données simulé fourni ainsi que les jeux de données simulés que vous aurez créé (détailler leur contenu), temps écoulé et mémoire consommée).

1. Pour cela, vous pouvez utiliser le programme `kmc` qui permet de compter les k -mers, une fois lancé sur votre jeu de données, vous utiliserez `kmc_dump` pour visualiser les k -mers et leurs nombre d'occurrences