In [1]:

```
from IPython.core.display import display, HTML
display(HTML("<style>.rendered_html { font-size: 12px; }</style>"))
```

Simulation des données démographiques et génomiques de deux sous-populations structurées en âge et interconnectées par des flux de gènes

Projet POPSIZE (IRD UMR Marbec, CRPMEM de La Réunion, projet FEAMP Mesure 28)

Delord, C. - Juin 2023

Objectif :

Dans le cadre du projet POPSIZE, nous nous intéressons à des populations de grands pélagiques marins. Nous nous plaçons ici dans un cadre relativement simple dans lequel on considère deux sous-populations théoriques structurées en âges, où chaque classe d'âge présente un taux de survie et de fécondité propre et identique entre mâles et femelles. Les deux sous-populations peuvent échanger des flux de gènes par voie de migration plus ou moins importante. Leur abondance, en nombre total d'individus, ainsi que leur structure en âges sont stables au cours du temps et identiques d'une sous-population à l'autre.

Nous souhaitons simuler les données démographiques et génétiques propres à ces deux sous-populations en tant que représentation simplifiée d'une espèce de top-prédateur à forte abondance et capacité de migration, le thon germon (*Thunnus alalunga*).

Ces données simulées, après post-traitement, seront utilisées pour évaluer la capacité de plusieurs méthodes à estimer l'abondance totale (taille census Nc) et la taille efficace (Ne) de telles populations, malgré leur forte abondance et la présence potentielle d'une sous-structuration génétique spatiale faible mais significative. Les méthodes testées sont diverses et peuvent s'appuyer (i) sur les spectres de fréquences alléliques (SFS), tels que les outils implémentés dans le logiciel moments (Jouganous et al. 2018) ou dadi (Gutenkunst et al. 2011]), (ii) sur le déséquilibre de liaison tels que les outils implémentés dans moments-LD (Ragsdale & Gravel 2020), NeEstimator2 (Do et al. 2014) ou GONE (Santiago et al. 2020), ou encore (iii) sur des analyses d'apparentement telle que la méthode de *close-kin mark-recapture* (CKMR) (Bravington et al. 2016)).

Le présent document vise à fournir une description détaillée de la procédure de simulation développée pour obtenir des données démographiques et génétiques sur lesquelles nous pourrons ensuite tester ces différentes méthodes. Il s'attache à intégrer, dès que possible, des informations complémentaires via des liens URL menant vers des publications ou des forums d'échanges scientifiques afin de permettre d'aller plus loin. Pour plus d'informations ou en cas de question, ne pas hésiter à envoyer un e-mail à l'adresse suivante: chrys.delord@gmail.com.

Caractéristiques des sous-populations simulées :

Avant d'entrer dans les concepts théoriques et les aspects techniques de programmation, commençons tout d'abord par spécifier les caractéristiques démographiques et génétiques des populations que nous allons simuler.

Nos deux sous-populations présentent les caractéristiques suivantes:

- Les 2 sous-populations se composent chacune de ~17710 individus au total (paramètre ***K***), dont 5000 nouveaux-nés (paramètre ***final_cohort_size***) générés à chaque événement de reproduction. Ces effectifs sont stables

dans le temps, donc d'un événement de reproduction au suivant. La structure en âge est fixée en début de simulation (paramètre vecteur ***W***) et chaque classe d'âge se caractérise par une valeur de mortalité (paramètre vecteur ***L***) et de fécondité relative (paramètre vecteur ***B***). L'âge maximal est fixé à 15 ans (tout individu dépassant l'âge de 15 ans décède) et l'âge à maturité est fixé à 4 ans (tout individu de 4 ou plus peut se reproduire).

- Les 2 sous-populations échangent une proportion continue et symétrique d'individus à chaque pas de temps, fixée à 5% (paramètre ***m***). On considère donc des flux de gènes stables et homogènes (les effectifs des 2 sous-populations étant identiques) d'une sous-population à l'autre.
- L'information génétique des individus est portée par 5 chromosomes indépendants de 4e08 paire de bases (400 Mbp) chacun, avec un taux de recombinaison classique fixé à 1e-08 par meiose et par paire de bases. Cela correspond à un génome d'une taille totale de 2.0 gigabases. Chacun des 5 chromosomes présente une longueur de 400 centimorgans si l'on raisonne en distance génétique. Ainsi, deux locus pris au hasard seront totalement indépendants s'ils proviennent de deux chromosomes distincts mais seront au contraire partiellement liés s'ils proviennent d'un même chromosome, comme cela est le cas pour un jeu de données réel à haute densité de marqueurs.

Choix de la procédure de simulation :

Les sous-populations simulées sont structurées en âge afin de se rapprocher de la réalité biologique de la plupart des espèces de grands pélagiques, qui présentent en effet des générations chevauchantes. Pour simuler cette réalité biologique, les outils de simulation individu-centrés sont nécessaires (par opposition à des outils de simulation échantillon- ou population-centrés tels que ceux qui s'appuient sur le coalescent). Nous avons fait le choix d'utiliser le logiciel de simulation individu-centré SLiM. Au moment d'initialiser la simulation, nous spécifierons que les populations simulées ne s'appuiront pas sur une approximation du modèle de population de Wright-Fisher (contrairement à ce que proposent les outils de simulation basés sur le coalescent), en activant la fonctionnalité ***nonWF***. Les outils de simulation individu-centrés ont l'avantage de permettre la simulation de populations biologiquement réalistes, mais présentent l'inconvénient de nécessiter un temps de calcul très long et d'utiliser beaucoup de mémoire vive et de mémoire de stockage, puisque chaque individu est modélisé distinctement des autres.

Le logiciel SLiM permet d'extraire, de chaque simulation, des informations démographiques et généalogiques diverses sur tout ou partie des individus simulés. On peut ainsi échantillonner une partie de ces individus selon des critères que l'on choisit (par exemple, on échantillonne au hasard 50 individus dans chaque classe d'âge et dans chaque sous-population), et consigner par exemple leur âge et leur sexe mais aussi leurs relations d'apparentements avec d'autres individus issus de la même simulation, par exemple. En outre, l'activation de la fonctionnalité de **tree-sequence recording**, un élément-clé dont nous reparlerons plus tard, permet également de consigner les généalogies de gènes au sein de l'échantillon.

Au delà d'une information démographique réaliste, nous souhaitons simuler l'information génomique relative aux individus présents dans nos sous-populations (notamment sous forme de génotypes pour un certain nombre de loci dont les positions relatives sur le génome sont connues). SLiM peut générer tout ou partie de cette information en simulant des chromosomes porteurs ou non de variabilité génétique, néanmoins, cette procédure augmente très considérablement le temps de calcul. La fonctionnalité de **tree-sequence recording** entre alors en jeu. Dans le cas où seule la variabilité génétique neutre nous intéresse (et dépend donc uniquement de forces évolutives neutres et non pas, a priori, de la fitness des individus simulés), il est possible de s'affranchir de la nécessité de simuler cette variabilité génétique neutre avec SLiM, et donc de gagner du temps. Seule une généalogie de gènes, occasionnée par les différents événements de reproduction et de recombinaison au cours du temps, sera exportée à l'issue de la simulation, sous-forme de **tree sequence**. Ce **tree sequence** peut alors être utilisé comme point de départ à une nouvelle étape de simulation plus classique, basée sur le coalescent, qui se chargera d'une part de compléter les généalogies si besoin en reconstituant leur trajectoire évolutive passée, d'autre part d'y intégrer de la variabilité génétique neutre en fonction de leur topologie (pour plus de détails, consulter le rapport scientifique final du projet POPSIZE, section I.3.2).

Les librairies Python *tskit* et *pyslim* permettent de faire le lien entre les informations portées par un fichier **tree sequence** issu du logiciel SLiM d'une part, et la librairie Python *msprime* d'autre part. Cette dernière permet la réalisation de simulations basées sur plusieurs modèles de coalescence et donc, de simuler rapidement les trajectoires évolutives passées et présentes de populations de Wright-Fisher ainsi que leur diversité génétique associée. La combinaison de SLiM et *msprime* permet d'associer les avantages respectifs des approches individu-centrées et de coalescence : le réalisme biologique d'une part et la reconstitution de trajectoires évolutives passées d'autre part, le tout en une durée optimisée (qui peut néanmoins rester très longue dans le cadre de la simulation d'information génomique pour de grandes populations).

Le présent document contiendra ainsi des sections de code rédigées via le langage Eidos spécifique à SLiM, mais également des sections rédigées en langage Python pour ce qui concerne l'utilisation des librairies *pyslim* et *msprime*.

Pré-requis :

Afin de ré-effectuer ces simulations, il est indispensable d'installer les librairies et logiciels suivants (si possible en utilisant Anaconda) :

- SLiM version 3.7
- Python3 avec les librairies *demesdraw*, *math*, *matplotlib.pyplot*, *msprime*, *numpy*, *pandas*, *pyslim* et *tskit*.
- R version >= 4.0.5 avec les libraries *CKMRpop*, *dartR*, *psych* and *tidyverse* et toutes leurs dépendances.

Disclaimer :

Les sections de code ci-dessous, bien que revues par les développeurs des logiciels SLiM et *msprime*, n'ont pas vocation à constituer une manière unique de simuler les données attendues. Elles peuvent également se révéler sous-optimales en termes de vitesse et de gestion de la mémoire. En outre, les fonctionnalités des logiciels SLiM, *pyslim* et *msprime* évoluant très rapidement, il est possible que la syntaxe de certaines lignes devienne obsolète avec le temps. Enfin, l'ensemble des sections de code présentées ci-dessous s'inspire d'exemples trouvés en divers lieux et places dans la littérature scientifique et technique. Nous nous sommes attachés à citer autant de références et de sources que possible et espérons n'en avoir oublié aucune. Nous remercions chaleureusement Jérôme Kelleher, Peter Ralph et Ben Haller pour leur travail de maintenance et d'animation des groupes d'entraide liés à SLiM et *pyslim/msprime*.

Etape n°1.

Simulation de l'histoire contemporaine des sous-populations à l'aide d'une approche individu-centrée :

SLiM ver. 3.7

Nous commençons par exécuter le script Eidos dans notre console Anaconda à l'aide de la commande `slim POPSIZE_SLiM_Cohort5000_m005.txt > output_POPSIZE_SLiM_Cohort5000_m005.txt`.

Le script `POPSIZE_SLiM_Cohort5000_m005.txt` s'écrit comme suit. Nous allons découper ce fichier en plusieurs parties afin d'en faciliter la lecture et la compréhension.

Notre script débute par quelques lignes commentées (débutant par un double-slash //) fournissant des informations et des références. Puis, nous devons définir certaines des fonctionnalités de la simulation. Enfin, nous initialisons des fichiers texte de sortie dans lequels plusieurs informations démographiques seront stockées à chaque pas de temps simulé.

Script `POPSIZE_SLiM_Cohort5000_m005.txt` **(Partie 1/4)**

```
// Ce script Eidos vise à simuler 2 sous-populations de type "nonWF" (non Wright
-Fisher), structurées en âges et échangeant des flux de gènes durant 100 pas de
 temps (événements de reproduction) en utilisant SLiM37 avec les fonctions 'pedi
gree tracking' et 'tree sequence recording'.
// Il vise égaement à générer des fichiers de sortie contenant des informations
 diverses sur les individus simulés. Ces fichiers sont construits sur le même mo
dèle que ceux générés par le simulateur Spip et la librairie R CKMRpop (Anderson
2022, https://doi.org/10.1111/1755-0998.13513).
// Ce script effectue un échantillonnage en série ('serial sampling') avec la fo
nction 'treeSeqRememberIndividuals()'.
// Les lignes de codes rédigées s'inspirent essentiellement des sources suivante
s :
//     - la vignette CKMRpop : https://eriqande.github.io/CKMRpop/articles/using
-other-simulation-programs.html associés à la publication d'Anderson (2022) (htt
ps://doi.org/10.1111/1755-0998.13513).
//     - la 'SLiM recipe' 16.2 (Manuel d'utilisation SLiM màj 13 février 2022)
//     - la 'SLiM recipe' 16.5 (Manuel d'utilisation SLiM màj 13 février 2022)
//     - la section 4.1.6 du Manuel d'utilisation SLiM màj 13 février 2022)
//     - la discussion: https://groups.google.com/g/slim-discuss/c/PaCARghZy9Q
//     - la vignette msprime: https://tskit.dev/msprime/docs/stable/ancestry.htm
l?highlight=multiple#multiple-chromosomes

initialize() {

    setSeed(getSeed());
    initializeSLiMModelType("nonWF"); // la simulation ne s'appuie pas sur les h
ypothèses d'une population de Wright-Fisher. Nous simulerons, notamment, des gén
érations chevauchantes et non des générations discrètes.
    initializeSLiMOptions(keepPedigrees = T); // on active l'enregistrement du p
edigree des individus.
    initializeTreeSeq(); // on active la fonctionnalité de tree-sequence recordi
ng.
    initializeSex("A"); // sexes séparés: les mâles et les femelles sont distinc
ts.

    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0); // un type unique et homogène
 d'élément génomique et de mutation.
    initializeGenomicElement(g1, 0, 1999999999); // un "chromosome" d'une taille
de 2.0 gigabases.
    initializeMutationRate(0.0); // on ne simule aucune variabilité génétique à
 ce stade. Des mutations neutres seront incorporées lors de la phase de simulati
on pyslim/msprime.
    initializeRecombinationRate(c(1.0e-08, 0.5, 1.0e-08, 0.5, 1.0e-08, 0.5, 1.0e
-08, 0.5, 1.0e-08), c(400000000, 400000001, 800000000, 800000001, 1200000000, 12
00000001, 1600000000, 1600000001, 1999999999));
    // on découpe notre chromosome en 5 sections égales librement recombinantes
 (c.f 'SLiM recipe' 6.1.4)
    // le taux de recombinaison au sein de chaque section est de 1.0e-08.
    m1.convertToSubstitution = T;
```

```
    defineConstant("K", 17710); // abondance totale de chaque sous-population dès la génération 1 (incluant les nouveaux-nés), toutes classes d'âge confondues - cet effectif restera constant (à quelques individus près) au cours du temps.
    // on définit les taux de mortalité L et de fécondité relative B des classes d'âge de 0 à 15. Tous les individus d'âge zéro (nouveaux-nés) survivent jusqu'à l'âge 1:
    defineConstant("L", c(0.00, 0.46, 0.38, 0.34, 0.31, 0.29, 0.31, 0.34, 0.38, 0.44, 0.55, 0.55, 0.55, 0.55, 0.60, 1.00));
    defineConstant("B", c(0.00, 0.00, 0.00, 0.00, 0.10, 0.20, 0.30, 0.40, 0.50, 0.60, 0.70, 0.80, 0.90, 1.00, 1.10, 1.20));
    defineConstant("W", c(25000.00, 25000.00, 13500.00, 8370.00, 5524.00, 3811.70, 2706.31, 1867.35, 1232.45, 764.12, 427.91, 192.56, 86.65, 38.99, 17.55, 7.02)); // on définit les proportions relatives W des différentes classes d'âge (les valeurs brutes importent peu, seules leurs proportions relatives sont importantes: elles définissent la structure en âge de chaque sous-population).
    defineConstant("m", 0.05); // taux de migration m, qui sera constant et symétrique entre sous-populations.

    // initialisation des fichiers de sortie pour le stockage des informations démographiques des individus simulés:
    // ces fichiers sont analogues aux fichiers générés par le logiciel Spip utilisé par la librairie R CKMRpop.
    writeFile("SLiM_prekill_census.tsv", "year\tpop\tage\tmale\tfemale");
    writeFile("SLiM_postkill_census.tsv", "year\tpop\tage\tmale\tfemale");
    writeFile("SLiM_demo_table.tsv", "generation\tgen_time\tkbar\tvk\tne_demo");
    writeFile("SLiM_samples.tsv", "ID\tsyears_pre\tpop_pre\tsyears_post\tpop_post\tsyears_dur\tpop_dur");
    writeFile("SLiM_ancestries.tsv", "ID\tancestors");

    // initialisation d'un objet de type dictionnaire, vide pour le moment, qui servira à stocker des informations nécessaires au calcul du temps de génération et de la variance de succès reproducteur ("variance in lifetime reproductive output") entre individus, que nous effectuerons sur plusieurs cohortes successives. Ces informations permettront d'obtenir la taille efficace démographique, par génération, des deux sous-populations simulées.
    defineGlobal("n_offspring", Dictionary());
}
```

Après cette phase d'initialisation, il va nous falloir définir les modalités des événements de reproduction (qui assurent par ailleurs le passage d'un pas de temps au suivant). Ceci s'effectue dans une section ("callback") *reproduction()*. Un géniteur et une génitrice sont échantillonnés avec une probabilité qui dépend de leur âge et du paramètre vecteur ***B***. Le couple génère un nombre non-nul de descendants suivant une loi de Poisson de paramètre 2,3. Le processus de tirage géniteur-génitrice se poursuit jusqu'à que le nombre de 5000 nouveaux-nés soit atteint. A ce stade, l'effectif stable des sous-populations n'est donc pas une propriété émergente du modèle. Bien qu'il serait plus élégant qu'il le soit, cela n'impacte cependant pas la validité des informations démographiques et génomiques générées.

Dans ce même "callback" *reproduction()*, le dictionnaire "n_offspring" stocke peu à peu des informations sur le nombre total de descendants générés, au cours de leur existance, par 11 cohortes successives. Une fonction permet d'en déduire la taille efficace démographique par génération ainsi que le temps de génération moyen. Ces informations sont exportées au fur et à mesure dans le fichier `SLiM_demo_table.tsv`.

Script `POPSIZE_SLiM_Cohort5000_m005.txt` **(Partie 2/4)**

```
reproduction() {

    for (subpop0 in sim.subpopulations) {
        fecundity = B[subpop0.individuals.age];
        subpop0.individuals.tagF = fecundity;
        potential_dads = subpop0.individuals[subpop0.individuals.age > 0 & subpo
p0.individuals.sex == "M"];
        potential_moms = subpop0.individuals[subpop0.individuals.age > 0 & subpo
p0.individuals.sex == "F"];
        current_cohort_size = c(0);
        final_cohort_size = 5000; // nombre final de nouveaux-nés souhaité en fi
n de reproduction.

        while (current_cohort_size < final_cohort_size) {
            dad = sample(potential_dads, size = 1, replace = F, weights = potent
ial_dads.tagF);
            mom = sample(potential_moms, size = 1, replace = F, weights = potent
ial_moms.tagF);
            litterSize = 0;
            do {
                litterSize = rpois(1, 2.3);
            }
            while (litterSize == 0);

            if (sim.generation < 28) {
                vec_mom = n_offspring.getValue(format('%d', mom.pedigreeID));
                vec_mom[1] = asFloat(((vec_mom[0] * vec_mom[1]) + (asFloat(litte
rSize) * asFloat(mom.age))) / (vec_mom[0] + asFloat(litterSize)));
                vec_mom[0] = asFloat(vec_mom[0] + asFloat(litterSize));
                n_offspring.setValue(format('%d', mom.pedigreeID), vec_mom);
                vec_dad = n_offspring.getValue(format('%d', dad.pedigreeID));
                vec_dad[1] = asFloat(((vec_dad[0] * vec_dad[1]) + (asFloat(litte
rSize) * asFloat(dad.age))) / (vec_dad[0] + asFloat(litterSize)));
                vec_dad[0] = asFloat(vec_dad[0] + asFloat(litterSize));
                n_offspring.setValue(format('%d', dad.pedigreeID), vec_dad);
            }

            for (j in seqLen(litterSize)) {
                offspring = subpop0.addCrossed(mom, dad);
                offspring.tag = subpop0.id;
                if (sim.generation < 28) {
                    n_offspring.setValue(format('%d', offspring.pedigreeID), c
(0.0, 0.0, asFloat(sim.generation)));
                }
            }
            current_cohort_size = current_cohort_size + litterSize;
        }
    }

    // Calcul de la taille efficace démographique et du temps de génération pour
la cohorte née 15 ans plus tôt (toutes sous-populations confondues):
```

```
    if (sim.generation > 16 & sim.generation < 28) { // ce calcul est effectué s
ur 11 cohortes successives, nées du pas de temps 2 au pas de temps 12, et unique
ment lorsque l'ensemble des individus d'une même cohorte sont décédés et ne peuv
ent assurément plus se reproduire (c'est-à-dire 15 pas de temps après leur naiss
ance).
        test0 = c(0.0, 0.0, 0.0);
        for (i in unique(n_offspring.allKeys)) {
            if (n_offspring.getValue(i)[2] == asFloat(sim.generation - 15)) {
                test0 = rbind(test0, n_offspring.getValue(i));
                n_offspring.setValue(i, NULL);
            }
        }
        lifetime_kbar = mean(test0[1:nrow(test0)-1, 0]);
        lifetime_vk = var(c(test0[1:nrow(test0)-1, 0]));
        gen_time = sum(test0[, 0]*test0[, 1])/sum(test0[, 0]); // temps de génér
ation
        ne_demo = (4*nrow(test0)*gen_time)/(lifetime_vk+2); // taille efficace d
émographique
        line = paste(c(sim.generation - 15, gen_time, lifetime_kbar, lifetime_v
k, ne_demo), sep = "\t");
        writeFile("SLiM_demo_table.tsv", line, append=T);
    }

    if (sim.generation == 28) { // désactivation du dictionnaire 'n_offspring' a
u-delà du pas de temps 28.
        // Ceci permet d'accélérer la vitesse de simulation à partir de maintena
nt.
        rm(variableNames="n_offspring");
    }

    self.active = 0; // le callback reproduction() se désactive de lui-même lors
que le nombre de nouveaux-nés atteint 5000.
}
```

Dans la section ("callback") *early()* du pas de temps 1, on initialise directement les deux sous-populations à l'abondance totale ***K*** en "remplissant" chacune des classes d'âges en fonction de la structure en âges fixée par le paramètre vecteur ***W***. D'une manière générale, les sections *early()* permettent ici de faire intervenir la migration symétrique entre sous-populations et l'enregistrement des effectifs précis par classe d'âge avant survenue de la mortalité.

Script `POPSIZE_SLiM_Cohort5000_m005.txt` **(Partie 3/4)**

```
1 early() { // pour le premier pas de temps (début de la simulation).

    sim.addSubpop("p1", K);
    sim.addSubpop("p2", K);
    p1.individuals.age = sample(seq(0, 15), size = K, replace = T, weights = W);
// remplissage de la sous-population p1.
    p2.individuals.age = sample(seq(0, 15), size = K, replace = T, weights = W);
// remplissage de la sous-population p2.
    p1.individuals.tag = 1;
    p2.individuals.tag = 2;
```

```
    for (i in c(p1.individuals.pedigreeID, p2.individuals.pedigreeID)) {
        n_offspring.setValue(format('%d', i), c(0.0,0.0,1.0));
    }
}

early() { // pour chaque pas de temps, avant survenue de la mortalité.

    // enclenche le processus de migration entre sous-populations :
    nIndividuals = sum(sim.subpopulations.individualCount);
    nMigrants = rpois(1, nIndividuals * m);
    migrants = sample(sim.subpopulations.individuals, nMigrants);

    for (migrant in migrants) {
        do dest = sample(sim.subpopulations, 1);
        while (dest == migrant.subpopulation);
        dest.takeMigrants(migrant);
    }

    // enregistre l'effectif de chaque classe d'âge, avant mortalité, dans le fi
chier de sortie "SLiM_prekill_census.tsv":
    for (subpop in sim.subpopulations) {
        print(tabulate(subpop.individuals.age, maxbin=15));
        inds = subpop.individuals;
        ages = inds.age;
        age_bins = 0:15;  // age categories, 0 to 15
        male_ages = ages[inds.sex == "M"];
        female_ages = ages[inds.sex == "F"];
        m_census = tabulate(male_ages, maxbin = 15);
        f_census = tabulate(female_ages, maxbin = 15);

        for(a in age_bins) {
            line = paste(sim.generation, subpop.id, a, m_census[a], f_census[a],
sep = "\t");
            writeFile("SLiM_prekill_census.tsv", line, append=T); // effectifs p
ar classe d'âge avant mortalité.
        }

        mortality = L[ages];
        survival = 1 - mortality;
        inds.fitnessScaling = survival;
    }
}
```

D'une manière générale, les sections *late()* permettent ici l'enregistrement des effectifs précis par classe d'âge après survenue de la mortalité.

Plus important, elles nous permettent également ici d'effectuer un "échantillonnage" en série (c'est-à-dire sur plusieurs pas de temps successifs) d'individus, plus précisément au cours des 11 derniers pas de temps (90 à 100) de la simulation. Il s'agit de stocker l'information génétique et démographique d'une partie des individus au sein des sous-populations simulées. Ici, on "échantillonne" une proportion de 10% de chaque classe d'âge de 1 à 15 ans (`ns = rbinom(1, num 1 to 15, 0.10);`) (ce qui, notons-le, serait beaucoup plus laborieux pour des effectifs de

populations plus élevés que 17710, pour lesquels il faudrait alors sans doute diminuer cette proportion). Il faut également noter que cet échantillonnage se fait avec remise: un individu échantillonné ne disparaît pas de la population, et peut potentiellement être ré-échantillonné lors d'un pas de temps suivant du simple fait du hasard, comme dans le cas d'un protocole de capture-marquage recapture. Il est possible, si nécessaire, de tenir compte de cet état de fait au cours du post-traitement des données simulées puisque chaque individu possède un identifiant unique, que l'on conserve lors de l'export des informations démographiques.

Chaque individu échantillonné verra ses informations démographiques (e.g., pas de temps de naissance, âge(s)/lieu(x)/pas de temps de capture) sauvegardées au sein du fichier `SLiM_samples.tsv`. Grâce à l'activation de l'enregistrement du pedigree, nous pouvons également sauvegarder les identifiants uniques des parents et des grands-parents de chaque individu échantillonné, au sein du fichier `SLiM_ancestries.tsv`. Par ailleurs, les généalogies de gènes correspondant aux individus échantillonnés sont conservées grâce à l'activation de la fonction `sim.treeSeqRememberIndividuals();`. C'est ce qui permettra d'attribuer à ces individus une information génétique lors de la prochaine phase de simulation basée sur le coalescent. Sans cette option, seuls les individus présents au tout dernier pas de temps simulé sont assurés d'être exportés avec une généalogie de gènes correcte. Dans le cadre d'un échantillonnage en série (*serial sampling*) comme effectué ici, la fonction `sim.treeSeqRememberIndividuals();` est donc très importante.

Script `POPSIZE_SLiM_Cohort5000_m005.txt` **(Partie 4/4)**

```
late() {
    // enregistre l'effectif de chaque classe d'âge, après mortalité, dans le fi
chier de sortie "SLiM_postkill_census.tsv":
    for (subpop in sim.subpopulations) {
        inds = subpop.individuals;
        ages = inds.age;
        age_bins = 0:15;  // age categories, 0 to 15
        male_ages = ages[inds.sex == "M"];
        female_ages = ages[inds.sex == "F"];
        m_census = tabulate(male_ages, maxbin = 15);
        f_census = tabulate(female_ages, maxbin = 15);

        for(a in age_bins) {
            line = paste(sim.generation, subpop.id, a, m_census[a], f_census[a],
sep = "\t");
            writeFile("SLiM_postkill_census.tsv", line, append=T);  // effectifs
par classe d'âge après mortalité.
        }
    }
}

90:100 late() {
    for (subpop in sim.subpopulations) {
        num_1_to_15 = sum(tabulate(subpop.individuals.age, maxbin = 15)[1:15]);

        ns = rbinom(1, num_1_to_15, 0.10); // on collecte l'information démograp
hique et généalogique d'une partie des individus simulés ("échantillonnage").
        samps = subpop.sampleIndividuals(ns, minAge = 1, maxAge = 15);
        sim.treeSeqRememberIndividuals(samps); // SLiM va conserver l'informatio
n généalogique des individus échantillonnés.

        for(s in samps) {
```

```
            s_name = paste0(s.sex, sim.generation - s.age, "_", s.tag, "_", s.pe
digreeID);
            line = paste(s_name, "", "", sim.generation, subpop.id, "", "", sep
 = "\t");
            writeFile("SLiM_samples.tsv", line, append = T);
            s_anc = c(s.pedigreeID, s.pedigreeParentIDs, s.pedigreeGrandparentID
s);
            s_anc = s_anc[c(0, 2, 1, 6, 5, 4, 3)];
            s_anc_commas = paste(s_anc, sep = ",");
            line = paste(s_name, s_anc_commas, sep = "\t");
            writeFile("SLiM_ancestries.tsv", line, append = T); // exporte les i
nformations de pedigree des individus échantillonnés (parents et grands-parent
s).
        }
    }
}

100 late() { // enfin, lors du tout dernier pas de temps simulé, on exporte le
 "tree sequence" contenant la généalogie des gènes de tous les individus du temp
s présent, mais aussi ceux dont l'information a été conservée par la fonction 's
im.treeSeqRememberIndividuals();'.
    sim.treeSeqOutput("POPSIZE_SLiM_Cohort5000_m005.trees");
}

// Fin du script Eidos.
```

Le fichier final `POPSIZE_SLiM_Cohort5000_m005.trees` est notre principale sortie pour le moment. Avec le fichier `SLiM_demo_table.tsv`, il va nous permettre de passer à l'étape n°2, à savoir la reconstitution de la trajectoire évolutive passée (c'est-à-dire antérieure aux 100 pas de temps simulés à l'aide du logiciel SLiM) de nos deux sous-populations. Cette reconstitution s'effectue sur la base de la topologie du **tree sequence** exporté via SLiM et à l'aide d'une approche par coalescence implémentée dans les librairies Python *pyslim* et *msprime*, un procédé appelé **récapitation**. Nous allons donc à présent quitter le langage de programmation Eidos et utiliser la console Python.

Etape n°2.

Simulation de la trajectoire évolutive passée des sous-populations à l'aide d'une approche par coalescence :

pyslim ver. 0.700 et msprime ver. 1.2.0

Nous exécutons à présent le script Python dans notre console Anaconda à l'aide de la commande `python POPSIZE_pyslim_Cohort5000_m005.py > output_POPSIZE_pyslim_Cohort5000_m005.txt`.

Le script `POPSIZE_pyslim_Cohort5000_m005.py` s'écrit comme suit. Nous allons là encore découper ce fichier en plusieurs parties afin d'en faciliter la lecture et la compréhension.

Script `POPSIZE_pyslim_Cohort5000_m005.py` **(Partie 1/7)**

Il s'agit dans un premier d'importer l'ensemble des outils nécessaires dans notre environnement de travail. Puis, nous importons notre principal fichier de travail: le fichier **tree sequence** issu de l'étape n°1. La fonction `print(ts)` nous permet de visualiser quelques informations sur ce **tree sequence**. On retrouve environ 2.6e06 généalogies distinctes

('trees') ainsi que la taille de 2.0 Gb du génome simulé. Le **tree sequence** contient 36 521 individus, comprenant la totalité des individus survivants du tout dernier pas de temps simulé dans SLiM, mais également des individus échantillonnés auparavant et décédés mais dont l'information a été conservée par la fonction 'sim.treeSeqRememberIndividuals();'.

In [1]:

```
# Importation des librairies Python nécessaires:
import datetime, demesdraw, functools, itertools, math, msprime, os, pysli
m, tskit
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import statistics as s

# Paramètres de la simulation en cours:
str1 = "5000"
str2 = "m005"
migrate = 0.05

print('Hi! You are currently using pyslim ver ', pyslim.__version__, ', ts
kit ver ', tskit.__version__, ' and msprime ver ', msprime.__version__, '.
See you soon.', sep='')
current_day = datetime.datetime.now().strftime("%Y%m%d")

# Importation du fichier "tree sequence" issu de l'étape n°1:
# os.getcwd()
# os.chdir('C:\\Users\\Sidon\\OneDrive\\Documents\\SLiM_5000_m005_runfile
s')
ts = pyslim.load("POPSIZE_SLiM_Cohort%s_%s.trees" % (str1, str2))
print(ts)
```

```
Hi! You are currently using pyslim ver 0.700, tskit ver 0.4.1
and msprime ver 1.2.0. See you soon.
```

TreeSequence	
Trees	2562579
Sequence Length	2000000000
Time Units	ticks
Sample Nodes	73042
Total Size	209.4 MiB

Table	Rows	Size	Has Metadata
Edges	5248573	160.2 MiB	No
Individuals	36521	3.5 MiB	Yes
Migrations	0	8 Bytes	No
Mutations	0	1.2 KiB	No
Nodes	157113	5.7 MiB	Yes

Populations	3	2.3 KiB	Yes
Provenances	1	12.3 KiB	No
Sites	0	16 Bytes	No

Script `POPSIZE_pyslim_Cohort5000_m005.py` **(Partie 2/7)**

On importe le fichier `SLiM_demo_table.tsv` qui nous permettra d'extraire la taille efficace démographique calculée par génération au cours de l'étape n°1 pour chacune des sous-populations. On vérifie que le nombre d'individus toujours en vie lors du dernier pas de temps simulé dans SLiM (s'élevant à 25435 d'après le fichier `SLiM_postkill_census.tsv` pour le pas de temps 100) se retrouve bien dans notre **tree sequence** au pas de temps 0 (`ts.individuals_alive_at(0)`), qui nous rapporte effectivement un total de 12692+12743 = 25435 individus.

Enfin, nous visualisons la distribution des temps de coalescence le long du génome simulé. En seulement 100 pas de temps simulé avec SLiM, il est très peu probable que le TMRCA ("*time to the most recent common antestor*", la racine de la généalogie de gènes) ait pu être atteint où que ce soit le long du génome, et nous nous attendons à ce que la taille des généalogies, en nombre de pas de temps, soit homogène et corresponde simplement au nombre de pas de temps simulés.

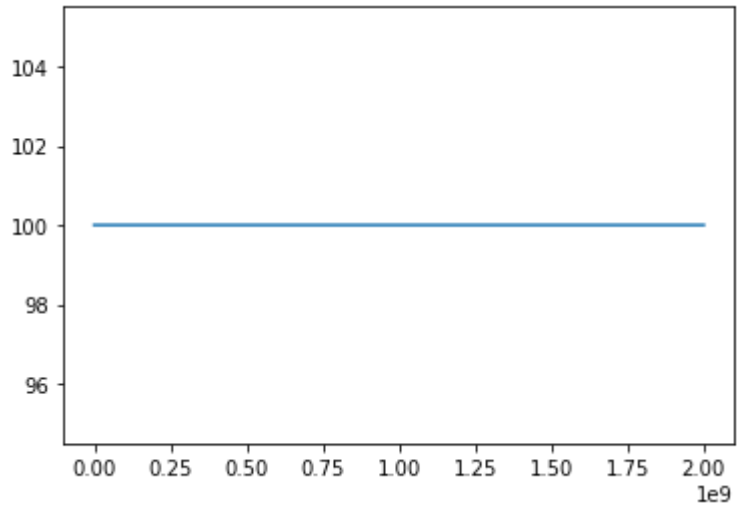
In [2]:

```
# Importation du fichier "SLiM_demo_table.tsv" issu de l'étape n°1:
SLiM_demo = pd.read_table("SLiM_demo_table.tsv", sep ='\t', header = 0, in
dex_col=False)

# Visualisation des populations au temps présent (0 générations en arrièr
e, correspond au dernier pas de temps simulé dans SLiM):
alive = ts.individuals_alive_at(0)
num_alive = [0 for _ in range(ts.num_populations)]
for i in alive:
  ind = ts.individual(i)
  num_alive[ind.population] += 1

for pop, num in enumerate(num_alive):
  print(f"Nombre d'individus au temps présent dans la population {pop}: {n
um}") # Nombre d'individus vivants au dernier pas de temps avant reproduct
ion.

# Calcul de la taille (en générations) des généalogies/arbres de coalescen
ce (= "tree heights"):
def tree_heights(ts):
    heights = np.zeros(ts.num_trees + 1)
    for tree in ts.trees():
        if tree.num_roots > 1: # MRCA non atteint
            heights[tree.index] = ts.slim_generation
        else:
            children = tree.children(tree.root)
            real_root = tree.root if len(children) > 1 else children[0]
            heights[tree.index] = tree.time(real_root)
    heights[-1] = heights[-2]
    return heights

# Visualisation de la taille (en générations) des arbres de coalescences a
```

```
vant récapitation (elle devrait être identique pour tous les loci: y=nombr
e de pas de temps simulés avec SLiM):
breakpoints = list(ts.breakpoints())
heights = tree_heights(ts)
plt.step(breakpoints, heights, where='post')
plt.show()
plt.savefig('pyslim_tree_height0_Cohort%s_%s.png' % (str1, str2))
```

```
Nombre d'individus au temps présent dans la population 0: 0
Nombre d'individus au temps présent dans la population 1: 126
92
Nombre d'individus au temps présent dans la population 2: 127
43
```

```
/home2/datawork/cdelord/conda-env/simupop-slim/lib/python3.7/
site-packages/pyslim/slim_tree_sequence.py:36: FutureWarning:
The SlimTreeSequence class is being phased out, as most impor
tant functionality is provided by tskit. Please see the `docu
mentation <https://tskit.dev/pyslim/latest/previous_versions.
html>`_. Please access ts.metadata['SLiM']['generation'] inst
ead.
  FutureWarning
```




```
<Figure size 432x288 with 0 Axes>
```

Script `POPSIZE_pyslim_Cohort5000_m005.py` **(Partie 3/7)**

La phase de récapitation va permettre de compléter les généalogies de gènes présentes dans notre **tree sequence**, c'est-à-dire à faire en sorte que le TMRCA soit atteint sur l'ensemble du génome simulé. Pour cela, il nous faut fournir à *pyslim* des informations sur la démographie présente et passée des sous-populations simulées. C'est ce que nous ferons à l'aide du module `msprime.Demography.from_tree_sequence(ts)`.

Afin de garder une taille efficace constante tout au long de la chronologie (ce qui n'est pas obligatoire et ne sera pas toujours le cas, mais s'avère plus simple pour le présent exemple), nous allons choisir comme point de départ, au temps présent, la taille efficace démographique que nous avons calculée sur plusieurs cohortes successives au cours de l'étape n°1. La trajectoire évolutive simulée ici pour le passé de nos deux sous-populations sera très simple: on considèrera que les deux sous-populations ont toujours conservé une taille efficace constante depuis leur divergence. Elles constituaient auparavant une seule et même population ancestrale, dont la taille efficace correspondait à la somme de leurs tailles efficaces locales. Tout ceci peut être visualisé à l'aide des représentations obtenues via le module `demography.debug()`.

Un simple calcul de moyenne harmonique depuis `SLiM_demo_table.tsv` indique que la taille efficace démographique s'élève à 2772 par génération et par sous-population, en moyenne, pour cette simulation SLiM. Le temps de génération est estimé, quant à lui, à 6,989.

Avant d'aller plus loin, il nous faut évoquer un point **crucial** concernant la combinaison de simulations de type **nonWF** avec SLiM, et de type Wright-Fisher (coalescent) avec *pyslim*/*msprime*. Ce point de précaution indispensable est souligné par l'avertissement qui s'affiche en rouge en sortie de cette portion de code. En effet, l'unité de temps considérée par SLiM et *pyslim*/*msprime* n'est pas la même: elle correspond à un pas de temps/cycle de reproduction/"tick" dans le premier cas, et à une génération de Wright-Fisher dans le second cas. Il nous faut donc procéder à un ré-échelonnage de certains paramètres (a minima la taille efficace, les taux de mutation et de recombinaison) afin d'en tenir compte et d'assurer un bon ajustement des simulations individu-centrées avec celles basées sur le coalescent. Une discussion détaillée de cela est disponible dans la section Time Units du manuel en ligne *pyslim*. D'autres informations utiles sont consultables par cet échange direct avec les développeurs de la librarie : Seeking for best practices with hybrid "backward-forward" simulation.

Nous pouvons désormais poursuivre:

- **Précisions sur la taille efficace *Ne* à définir dans `pop.initial_size` :**

Comme mentionné plus tôt, la valeur de 2772 correspond à la taille efficace démographique, par génération et par sous-population, calculée à partir des données simulées dans SLiM. Elle tient compte du temps de génération et est différente de la taille efficace calculée par cycle de reproduction, qui correspondrait à une valeur *Nb* (effective number of breeders). Dans la mesure où les deux populations simulées présentent une taille efficace égale et constante dans le temps, et dans la mesure où ces deux sous-populations échangent des flux de gènes constants et symétriques, nous devrions pouvoir considérer que la taille efficace globale correspond à la somme des tailles efficaces locales (e.g, Wang & Caballero, 1999). C'est pourquoi ici, nous obtenons `Ne_demo` en divisant la valeur de taille efficace globale issue `SLiM_demo_table.tsv` par 2. Afin de ré-échelonner la taille efficace pour tenir compte de la différence d'unité de temps entre SLiM et *pyslim*/*msprime*, il nous faut multiplier `Ne_demo` par le temps de génération `gen_time`. Nous obtenons ainsi la valeur `Ne_pyslim` à renseigner dans `pop.initial_size`.

En outre, notre taille efficace démographique `Ne_demo` (qui est un type de taille efficace de variance) est donc ici utilisée comme un proxy de taille efficace de valeur propre (qui est une valeur asymptotique de taille efficace vers laquelle tailles efficace de variance et de consanguinité convergent à l'équilibre à l'échelle d'un groupe de populations). A moins que des événements démographiques critiques n'interviennent au cours de la trajectoire évolutive des sous-populations, nous devrions pouvoir considérer que cette taille efficace démographique / de valeur propre soit également équivalente à la taille efficace de coalescence (Ryman et al. 2019) et donc utilisable dans le modèle démographique utilisé ici dans *pyslim* une fois ré-échelonnée (e.g, `Ne_pyslim`). Cette approximation est permise par la très grande simplicité de la trajectoire évolutive simulée ici.

Cependant, si l'on souhaitait simuler des événements démographiques contemporains (e.g., bottleneck ou déclin d'abondance lié à la surpêche), ou passés (e.g., croissance de la population ancestrale au cours du Pléistocène) ou encore, si l'on souhaitait modéliser des dynamiques distinctes entre les deux sous-populations (e.g., déclin de l'abondance d'une des sous-populations tandis que la seconde garde un effectif stable), alors nous devrions reconsidérer tout cela. La taille efficace démographique devrait vraisemblablement être calculée d'une autre manière lors de l'étape n°1 de SLiM (cf., rapport scientifique final du projet POPSIZE section I.b). Cette taille efficace ne serait plus nécessairement équivalente à la taille efficace de coalescence, et une réflexion serait à mener pour établir la meilleure manière de modéliser la trajectoire évolutive passée des sous-populations dans le module `msprime.Demography.from_tree_sequence(ts)` de *pyslim* en intégrant à la fois cette taille efficace contemporaine, ses fluctuations en remontant le temps, et sa transition éventuelle vers la taille efficace de coalescence.

Voir aussi: Questions about treeseq good practices

- **Précisions sur le flux de migration *migrate* à définir dans `demography.set_symmetric_migration_rate` :**

Dans notre cas, pour lequel les flux de migration sont symétriques entre deux sous-populations de taille efficace et d'abondance identiques, nous pouvons garder la même valeur du paramètre de migration *m* et *migrate* entre l'étape n°1 SLiM et l'étape n°2 *pyslim*. Cela, bien que la définition du paramètre de migration ne soit pas strictement identique entre les deux logiciels (pour SLiM, il s'agit de la proportion moyenne d'individus d'une sous-population qui migrent vers une autre, alors que pour *pyslim*/*msprime*, il s'agit de la proportion d'individus haploides d'une sous-population ayant un parent issu d'une autre sous-population). Là encore, si l'on souhaitait simuler des sous-populations d'effectifs différents et/ou avec des flux de migration asymétriques, nous devrions fixer les flux de migration de manière différente entre les étapes respectives SLiM et *pyslim*.

Voir aussi: Coalescence trajectory for non symmetric migration Voir aussi: msprime doc > Demographic Models > Model > Definitions

- **Précisions sur les taux de recombinaison à définir dans `pyslim.recapitate()` :**

Tout comme la taille efficace et le taux de mutation (cf., Partie 6/7), les taux de recombinaison doivent être ré-échelonnés pour tenir compte de la différence d'unité de temps considérée par SLiM et *pyslim*. On obtient le taux de recombinaison par génération de Wright-Fisher en divisant le taux de recombinaison fixé dans SLiM (exprimé par pas de temps/cycle de reproduction/méiose), par le temps de génération. Ici, on considère donc: `r_chrom` = 1e-08 / 6.98 = 1.435e-9. Le même calcul s'appliquera au taux de mutation (cf., Partie 6/7).

In [3]:

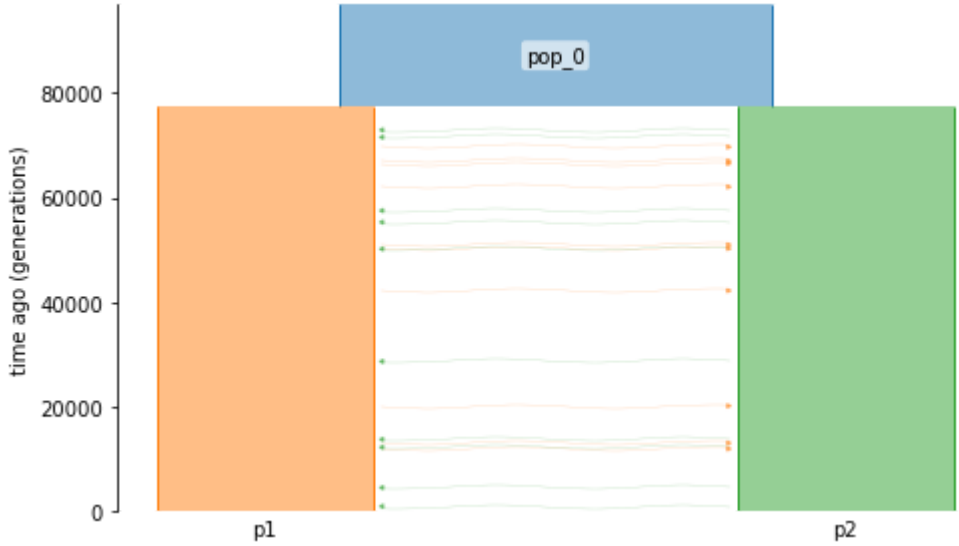
```
# ------------------------------------------------------------------------
------------------
# RECAPITATION (ici, nous devons spécifier chaque événement démographique
 survenu durant l'histoire passée des sous-populations,
# ainsi que leur taille efficace, les flux de migrations et les taux de re
combinaison):
# ------------------------------------------------------------------------
------------------

Ne_demo = (s.harmonic_mean(SLiM_demo['ne_demo']))/2 # Taille efficace par
 sous-population.
# La taille efficace démographique Ne_demo sera ici utilisée comme proxy d
e la taille efficace de coalescence.
gen_time = (s.harmonic_mean(SLiM_demo['gen_time']))
print("Mean, per-generation effective size obtained from SLiM reproductive
outputs is ", Ne_demo, ". Generation time is ", gen_time, ".")

Ne_pyslim = Ne_demo*gen_time # Re-échelonnage de la taille efficace par gé
nération de Wright-Fisher pour chaque sous-population.
t_split = 4*Ne_pyslim # On fixe le temps de divergence entre sous-populati
ons de manière à favoriser l'équilibre migration-dérive.

# Initialisation des taux de recombinaison par section de chromosome:
r_chrom = 1.435e-9 # Re-échelonnage du taux de recombinaison par génératio
n de Wright-Fisher.
r_break = math.log(2)
chrom_positions = [0, 400000000, 800000000, 1200000000, 1600000000, 200000
0000]
map_positions = [chrom_positions[0], chrom_positions[1], chrom_positions[1
] + 1, chrom_positions[2], chrom_positions[2] + 1, chrom_positions[3], chr
om_positions[3] + 1, chrom_positions[4], chrom_positions[4] + 1, chrom_pos
itions[5]]
rates = [r_chrom, r_break, r_chrom, r_break, r_chrom, r_break, r_chrom, r
```

```
break, r_chrom]
rate_map = msprime.RateMap(position=map_positions, rate=rates)

# Initialisation des informations démographiques et définition de la traje
ctoire évolutive passée:
demography = msprime.Demography.from_tree_sequence(ts)
for pop in demography.populations:
    if pop.name in ('p1', 'p2'):
        pop.initial_size = Ne_pyslim # Taille efficace locale de chaque so
us-population.
    else:
        pop.initial_size = 2*Ne_pyslim # Taille efficace de la population
 ancestrale "pop_0", avant divergence.
# demography.add_population(name="pop_anc", initial_size= int(Ne_pyslim*
2))
# La ligne de code ci-dessus générait un bug décrit ici: https://groups.go
ogle.com/g/slim-discuss/c/WA-c1hUWgIY.
# En réalité, pour insérer une nouvelle population non-existante au tree s
equence, nous devons initialiser ses métadonnées manuellement comme expliq
ué par P.Ralph dans l'URL ci-dessus.
# Ici, on tire donc finalement parti de la population vide "pop_0" automat
iquement présente dans le tree sequence. Elle deviendra notre population a
ncestrale.
demography.add_population_split(time=t_split, derived=["p1", "p2"], ancest
ral="pop_0") # Divergence entre les sous-populations.
demography.set_symmetric_migration_rate(populations=["p1", "p2"], rate=mig
rate) # Définition des flux de migration entre les sous-populations.

debug = demography.debug() # Ce module nous permet de visualiser la trajec
toire évolutive modélisée précédemment pour vérification.
print(debug)
mod = demography.to_demes()
ax = demesdraw.tubes(mod)

# ON PEUT ALORS LANCER LA RECAPITATION:
rts = pyslim.recapitate(ts, demography=demography, recombination_rate=rate
_map, random_seed=1)
```

```
Mean, per-generation effective size obtained from SLiM reprod
uctive outputs is  2772.0601826540315 . Generation time is
6.988647213350303 .
DemographyDebugger
╠═══════════════════════════════════════╗
║ Epoch[0]: [0, 7.75e+04) generations ║
╠═══════════════════════════════════════╝
╟    Populations (total=3 active=2)
║    ┌──────────────────────────────────────────────────────┐
║    │    │     start│       end│growth_rate  │  p1  │  p2  │
║    ├──────────────────────────────────────────────────────┤
║    │  p1│   19373.0│   19373.0│ 0           │  0   │ 0.05 │
║    │  p2│   19373.0│   19373.0│ 0           │ 0.05 │  0   │
║    └──────────────────────────────────────────────────────┘
╟    Events @ generation 7.75e+04
║    ┌──────────────────────────────────────────────────────────────────
──────────────────────────┐
║    │       time│type         │parameters          │effect
│
╟    ├──────────────────────────────────────────────────────────────────
──────────────────────────┤
║    │  7.749e+04│Population   │derived=[p1, p2]    │Moves all
```

```
╟    │  7.749e+04│Population  │derived=[p1, p2],  │Moves all
lineages from derived          │
╟    │           │Split       │ancestral=pop_0    │population
s 'p1' and 'p2' to the         │
╟    │           │            │                   │ancestral
'pop_0' population. Also set   │
╟    │           │            │                   │the derive
d populations to inactive,     │
╟    │           │            │                   │and all mi
gration rates to and from      │
╟    │           │            │                   │the derive
d populations to zero.         │
╟    └──────────────────────────────────────────────────────┘
╟──────────────────────────────────┘
╟╔═══════════════════════════════════════╗
╟║ Epoch[1]: [7.75e+04, inf) generations ║
╟╚═══════════════════════════════════════╝
╟    Populations (total=3 active=1)
╟    ┌──────────────────────────────────────────────┐
╟    │       │     start│       end│growth_rate   │
╟    ├──────────────────────────────────────────────┤
╟    │  pop_0│   38745.9│   38745.9│ 0            │
╟    └──────────────────────────────────────────────┘
```

```
/home2/datawork/cdelord/conda-env/simupop-slim/lib/python3.7/
site-packages/msprime/ancestry.py:831: TimeUnitsMismatchWarni
ng: The initial_state has time_units=ticks but time is measur
ed in generations in msprime. This may lead to significant di
screpancies between the timescales. If you wish to suppress t
his warning, you can use, e.g., warnings.simplefilter('ignor
e', msprime.TimeUnitsMismatchWarning)
  warnings.warn(message, TimeUnitsMismatchWarning)
```



Script `POPSIZE_pyslim_Cohort5000_m005.py` **(Partie 4/7)**

Suite à notre phase de récapitation, il s'agit de vérifier que les généalogies de gènes de notre **tree sequence** ont, cette fois-ci, trouvé leur TMRCA, et donc que la coalescence est complète tout le long du génome. Nous pouvons visualiser de nouveau la distribution des temps de coalescence de long du génome et constater que "le temps moyen de coalescence, divisé par 4, donne 39747.89704852677" ce qui correspond à peu près à la valeur de taille efficace de la population ancestrale (et à la somme des tailles efficaces des sous-populations) telles que fixées dans *pyslim*.

La fonction `print(rts)` nous permet de visualiser quelques informations sur ce **tree sequence** après récapitation. Par rapport à sa version précédente, il contient des arbres plus "denses" (valeur `Edges` et `Nodes`) et des généalogies plus nombreuses.

In [4]:

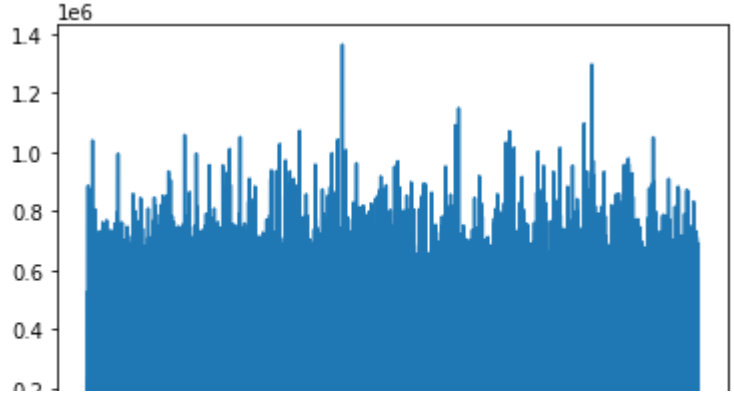
```
# Affichage des informations principales du tree sequence, après récapitat
ion.
print(rts)
assert(max([t.num_roots for t in rts.trees()]) == 1)

# Visualisation de la taille (en générations) des arbres de coalescences a
près récapitation.
# A présent, elle devrait varier entre loci car tous n'auront pas le même
 temps de coalescence (TMRCA).
breakpoints = list(rts.breakpoints())
heights = tree_heights(rts)
print('Le temps moyen de coalescence, divisé par 4, donne ', np.mean(heigh
ts)/4)
plt.step(breakpoints, heights, where='post')
plt.show()
plt.savefig('pyslim_tree_height1_Cohort%s_%s.png' % (str1, str2))

# On vérifier que suite à la récapitation, tous les arbres convergent vers
une même racine.
orig_max_roots = max(t.num_roots for t in ts.trees())
recap_max_roots = max(t.num_roots for t in rts.trees())
print(f"Nombre de racines distinctes avant récapitation: {orig_max_roots}
\n"
      f"Nombre de racines distinctes après récapitation: {recap_max_roots}
")
rts.dump("POPSIZE_pyslim_Cohort%s_%s_rts.trees" % (str1, str2))

# Visualisation de l'évolution des taux de coalescence au cours du temps:
def count_tree_coalescence_pairs(ts, epochs):
    nepochs = len(epochs)
    times = dict()
    for tr in ts.trees():
        for node in tr.nodes():
            if tr.is_leaf(node):
                continue
            nlineages = np.array([tr.num_samples(child) for child in tr.ch
ildren(node)])
            ns = functools.reduce(lambda s, t: s + t, (x*y for x, y in ite
rtools.combinations(nlineages, 2)))
            t = tr.time(node)
            times[t] = ns + times.get(t, 0)
    counts = np.fromiter(times.values(), dtype=int)
    times = np.fromiter(times.keys(), dtype=float)
    indices = np.digitize(times, epochs)
    coalesced = np.zeros(shape=nepochs)
    np.add.at(coalesced, indices, counts)
    total = np.cumsum(coalesced[::-1])[::-1]
    #counts = count_tree_coalescence_pairs(mts, bins)
    plt.figure(1)
    plt.plot(epochs[:-1], coalesced[1:])
    plt.xlabel("Nombre de générations avant présent (en bins de 100 généra
tions)")
    plt.ylabel("Nombre d'événements de coalescence")
```

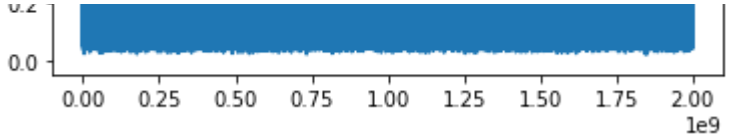
```
    plt.savefig("Fig_noCoalescedPairsCohort%s_%s.png" % (str1, str2), bbox
_inches="tight")
    plt.show()
    plt.figure(2)
    plt.plot(epochs[:-1], total[1:]-coalesced[1:])
    plt.xlabel("Nombre de générations avant présent (en bins de 100 généra
tions)")
    plt.ylabel("Proportion de lignées ancestrales restantes")
    plt.savefig("Fig_propAncestralLineages_Cohort%s_%s_simR_%s.png" % (str
1, str2, rep), bbox_inches="tight")
    plt.show()
    return coalesced[1:], total[1:], epochs
```

TreeSequence	
Trees	5787202
Sequence Length	2000000000
Time Units	ticks
Sample Nodes	73042
Total Size	753.3 MiB

Table	Rows	Size	Has Metadata
Edges	18083512	551.9 MiB	No
Individuals	36521	3.5 MiB	Yes
Migrations	0	8 Bytes	No
Mutations	0	1.2 KiB	No
Nodes	2188851	59.9 MiB	Yes
Populations	3	2.3 KiB	Yes
Provenances	2	14.4 KiB	No
Sites	0	16 Bytes	No

```
Le temps moyen de coalescence, divisé par 4, donne  39747.897
04852677
```






```
Nombre de racines distinctes avant récapitation: 1636
Nombre de racines distinctes après récapitation: 1

<Figure size 432x288 with 0 Axes>
```

Script `POPSIZE_pyslim_Cohort5000_m005.py` **(Partie 5/7)**

Nous allons maintenant procéder à la phase dite de "simplification". La simplification va nous permettre de "débroussailler" notre **tree sequence**. En effet, suite à sa récapitation, celui-ci est très lourd et très dense mais, surtout, nous n'avons pas besoin de connaître l'information génétique de tous les individus vivants au temps présent (qui, à ce stade, sont toujours intégrés au **tree sequence** puisqu'exportés par défaut à la fin de l'étape n°1). En effet, seule l'information génétique des individus échantillonnés dans SLiM entre les pas de temps 90 à 100 nous intéresse. Le processus de simplification va nous permettre de ne conserver, dans notre **tree sequence**, que les généalogies relatives à ces individus et dont les identifiants uniques, conservés dans notre fichier `SLiM_samples.tsv`, peuvent également être retrouvés dans le **tree sequence**.

La fonction `print(sts)` nous permet de visualiser quelques informations sur ce **tree sequence** après simplification. Par rapport à sa version précédente, il contient des arbres moins "denses" (valeur `Edges` et `Nodes`) et des généalogies moins nombreuses. Surtout, il contient un nombre bien moindre d'individus, plus précisément 14978. Cette valeur est très proche du nombre de 14856 individus échantillonnés au cours de l'étape n°1. La légère différence provient du fait que certains individus présents dans le **tree sequence** ne faisaient pas partie de notre échantillon mais ont été conservés (de manière automatique) par *pyslim*, car impliqués dans les généalogies de notre échantillon.

In [2]:

```
# ------------------------------------------------------------------------
------------------
# SIMPLIFICATION (ici, nous devons spécifier l'identité des "noeuds" assoc
iés aux individus qui ont été mémorisés
# au cours de l'échantillonnage à l'étape n°1 grâce à la fonction "sim.tre
eSeqRememberIndividuals();" de SLiM.).
# ------------------------------------------------------------------------
------------------

# Importation des identifiants de tous les individus échantillonnés au cou
rs de l'étape n°1, stockés dans le fichier "SLiM_samples.tsv".
# Nous allons utiliser cette information pour simplifier notre "tree seque
nce" suite à sa récapitation.
samples0 = pd.read_csv("SLiM_samples.tsv", delimiter="\t", header=0)
samples = samples0.drop_duplicates(subset='ID', keep='first', inplace=Fals
e, ignore_index=True) # Nous n'avons besoin que des identifiants uniques,
 même si certains individus ont pu être échantillonnés plusieurs fois.
samples['pid'] = samples.ID.str.split("_").str[2]
samp_ped = samples.pid.astype(int).tolist()

# Création d'une liste vide pour stocker les identifiants pyslim de chaque
individu correspondant aux identifiants SLiM (pedigree_id) des échantillon
s de samp_ped.
keep_indivs = []
for i in rts.individuals():
    pid = i.metadata["pedigree_id"]
```

```
    if pid in samp_ped:
        keep_indivs.append(i.id)

# Le nombre d'individus présents dans samp_ped et dans keep_indiv doit êtr
e identique:
print('Notre fichier SLiM_samples.tsv contient ', len(samp_ped), ' échanti
llons avec un identifiant SLiM pedigree_IDs, et notre liste keep_indivs co
ntient ', len(keep_indivs), ' échantillons avec un identifiant pyslim indi
vidual.id.', sep='')

# Création d'une liste vide pour stocker les identifiants des noeuds (pysl
im) associés aux échantillons de keep_indivs.
keep_nodes = []
for i in keep_indivs:
    keep_nodes.extend(rts.individual(i).nodes)

# ON PEUT ALORS LANCER LA SIMPLIFICATION:
# en conservant uniquement, dans notre "tree sequence", les individus dont
l'identifiant correspondent aux individus échantillonnés au cours de l'éta
pe n°1 (keep_nodes).
# Nous éliminons les autres individus qui étaient vivants au temps présent
(i.e., dans la dernière génération simulée dans SLiM).
sts = rts.simplify(keep_nodes)
print(sts)
sts.dump("POPSIZE_pyslim_Cohort%s_%s_sts.trees" % (str1, str2))
```

```
/home2/datawork/cdelord/conda-env/simupop-slim/lib/python3.7/
site-packages/ipykernel/__main__.py:10: SettingWithCopyWarnin
g:
A value is trying to be set on a copy of a slice from a DataF
rame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.o
rg/pandas-docs/stable/user_guide/indexing.html#returning-a-vi
ew-versus-a-copy
```

```
Notre fichier SLiM_samples.tsv contient 14856 échantillons av
ec un identifiant SLiM pedigree_IDs, et notre liste keep_indi
vs contient 14856 échantillons avec un identifiant pyslim ind
ividual.id.
```

TreeSequence	
Trees	4827996
Sequence Length	2000000000
Time Units	ticks
Sample Nodes	29712
Total Size	694.0 MiB

Table	Rows	Size	Has Metadata
Edges	16636023	507.7 MiB	No
Individuals	14978	1.4 MiB	Yes

Migrations	0	8 Bytes	No
Mutations	0	1.2 KiB	No
Nodes	2132052	57.9 MiB	Yes
Populations	3	2.3 KiB	Yes
Provenances	3	14.9 KiB	No
Sites	0	16 Bytes	No

Script `POPSIZE_pyslim_Cohort5000_m005.py` **(Partie 6/7)**

Il nous reste à présent à ajouter de la variabilité génétique à notre **tree sequence**. Nous pouvons utiliser un modèle de mutation classique (ici `msprime.JC69()`, le modèle de Jukes et Cantor, 1969).

La fonction `print(mts)` nous permet de visualiser quelques informations sur ce **tree sequence** après ajout de mutations. Le nombre de `Sites` et de `Mutations`, toujours égaux à zéro dans les versions précédentes, sont maintenant passés à 5.098e06 et 5.105e06, respectivement (certains sites variables portent donc plus d'une mutation).

In [2]:

```
# ------------------------------------------------------------------------
------------------
# ADDITION DE VARIABILITE GENETIQUE NEUTRE (OVERLAY/MUTATE)
# (La diversité génétique neutre est répartie, sous forme de l'ajout de va
riants, le long du "tree sequence").
# Par la suite, nous pourrons ainsi exporter des génotypes pour chaque ind
ividu échantillonné dans SLiM.
# ------------------------------------------------------------------------
------------------

mts = pyslim.SlimTreeSequence(msprime.sim_mutations(sts, rate=1.435e-09, m
odel=msprime.JC69(), keep=True))
# Re-échelonnage du taux de mutation par génération de Wright-Fisher: 'rat
e=1.435e-09'.
print(f"Notre tree sequence comporte maintenant {mts.num_sites} sites vari
ables, {mts.num_mutations} mutations,\n"
      f"et la diversité nucléotidique moyenne est de {mts.diversity():0.3
e}.")
print(mts)
mts.dump("POPSIZE_pyslim_Cohort%s_%s_mts.trees" % (str1, str2))
```

```
Notre tree sequence comporte maintenant 5104971 sites variabl
es, 5111560 mutations,
et la diversité nucléotidique moyenne est de 2.221e-04.
```

TreeSequence	
Trees	4827996
Sequence Length	2000000000
Time Units	ticks

Sample Nodes	29712
Total Size	996.0 MiB

Table	Rows	Size	Has Metadata
Edges	16636023	507.7 MiB	No
Individuals	14978	1.4 MiB	Yes
Migrations	0	8 Bytes	No
Mutations	5111560	180.4 MiB	No
Nodes	2132052	57.9 MiB	Yes
Populations	3	2.3 KiB	Yes
Provenances	4	15.6 KiB	No
Sites	5104971	121.7 MiB	No

Script `POPSIZE_pyslim_Cohort5000_m005.py` **(Partie 7/7)**

Nous ne souhaitons pas exporter l'information génétique de tous les sites variables. Nous allons sélectionner un petit nombre de loci dont les caractéristiques nous conviennent. Nous n'exporterons que des sites bialléliques par exemple, et dont les fréquences alléliques sont suffisamment élevées pour être informatives. Ici, nous exportons un total de 30000 loci. Notre table de génotypes finale, exportée sous forme d'un fichier *"variant call format"* (.vcf) ; contiendra donc l'information génétique des 14856 individus échantillonnés au cours de l'étape n°1 avec leurs identifiants uniques (qui sont par ailleurs toujours conservés dans notre fichier `SLiM_samples.tsv` également) sur ces 30000 loci.

Ce fichier .vcf constitue donc la principale sortie de notre script *pyslim* et servira de base de référence à d'éventuels sous-échantillonnages d'individus (par exemple, pour ne travailler qu'avec les génotypes des individus d'un certain âge, capturés à un certain pas de temps, etc.) ou de loci (par exemple, en effectuant une sous-sélection de 1000 loci parmi les 30000 disponibles, ou en sélectionnant spécifiquement des loci issus d'une même section de chromosome). Ce type de post-traitement du fichier .vcf a été développé à l'aide du script R dédié `POPSIZE_vcf_output_processing.R`, qui s'appuie entre autres sur le fichier `SLiM_samples.tsv` et mobilise notamment les librairies *CKMRpop* et *dartR* pour manipuler les individus échantillonnés en fonction de différents critères, et générer des sous-fichiers .vcf (ou autres formats, e.g., PLINK) correspondant à des sous-sélections d'individus/de loci.

In [2]:

```
# ------------------------------------------------------------------------
------------------
# POST-TRAITEMENT DE NOTRE TREE-SEQUENCE ET EXTRACTION DE GENOTYPES.
# A présent que nous avons généré de la diversité génétique, nous pouvons
 exporter des tables de génotypes pour tout ou partie des individus échant
illonnés dans SLiM:
# ------------------------------------------------------------------------
------------------
```

```
# Diverses fonctions nous permettent de sélectionner les sites variables s
usceptibles de nous intéresser.

# Fonction pour le retrait des sites variables à plus de 2 allèles (on ne
 garde que les loci de type SNP biallélique).
def removeMultiAllelic_global(mts):
    sites_to_discard = []
    for site in mts.sites():
        if len(site.mutations) != 1:
            sites_to_discard.append(site.id)
    mts_new = mts.delete_sites(sites_to_discard)
    print(mts_new.num_sites, " sites restants après le retrait des loci à
 plus de 2 allèles.")
    return mts_new

# Fonction pour la sélection des variants en fonction de leur fréquence al
lélique minoritaire (MAF) sur l'ensemble des individus échantillonnés.
# (Attention cependant, cette fonction n'est pas encore optimale car elle
 considère tous les échantillons tous pas de temps confondus.)
# (Il faudrait pouvoir considérer les périodes d'échantillonnages indépend
amment les unes des autres pour bien réfléchir aux loci à sélectionner.)
def removeRareVariants_global(mts, minor_frequency):
    sites_to_discard = []
    for v in mts.variants():
        if np.sum(v.genotypes[mts.samples()]) / len(mts.samples()) < minor
_frequency:
            sites_to_discard.append(v.site.id)
    mts_new = mts.delete_sites(sites_to_discard)
    print(mts_new.num_sites, " sites restants après le retrait des loci de
MAF < ", minor_frequency, ".")
    return mts_new

# Fonction pour échantillonner un nombre 'num_loc' de variants le long du
 génome afin d'exporter ensuite leurs génotypes.
def randomSitesSampler(mts, num_loc, span):
    sites_to_discard = []
    if span > chromosomeSize:
        span = chromosomeSize
        print("Warning: la valeur span a été fixée égale à chromosomeSize,
car elle excédait initialement cette valeur.")
    for site in mts.sites():
        if site.position > span-1:
            sites_to_discard.append(site.id)
    mts_new = mts.delete_sites(sites_to_discard)
    sites_to_discard = []
    if num_loc > mts_new.num_sites:
        num_loc =  mts_new.num_sites
        print("Warning: la valeur num_loc value a été fixée égale à mts.nu
m_sites, car elle excédait initialement le nombre de variants disponibles
 sur la longueur span.")
        print("Le nombre de sites variables disponibles est de: ", mts_new
.num_sites, ".")
    sample_loc = np.random.choice(mts_new.sites(), size=(mts_new.num_sites
-num_loc), replace=False)
    for v in sample_loc:
        sites_to_discard.append(v.id)
    mts_new = mts_new.delete_sites(sites_to_discard)
    print("-- Nombre de variants conservés: ", mts_new.num_sites)
    return mts_new
```

```
chromosomeSize = 2e09 # Rappel de la taille de génome simulé, 2.0 Gb.
mts1 = removeMultiAllelic_global(mts) # Retrait des sites variables multi-
alléliques.
mts1 = removeRareVariants_global(mts1, 0.005) # Retrait des sites très peu
variables. A user avec précautions.
mts1 = randomSitesSampler(mts1, 30000, chromosomeSize) # Sélection aléatoi
re de 30000 loci le long du génome.
print(mts1)

mts1.dump("POPSIZE_pyslim_Cohort%s_%s_mts1.trees" % (str1, str2))

indivlist = []
indivnames = []
for i in mts1.individuals():
    if mts1.node(i.nodes[0]).is_sample():
        indivlist.append(i.id)
        assert mts1.node(i.nodes[1]).is_sample()
        pid0 = i.metadata['pedigree_id']
        indivnames.append(samples.ID[samples.pid == str(pid0)].item())
with open("POPSIZE_pyslim_output_Cohort%s_%s.vcf" % (str1, str2), "w") as
vcffile:
    mts1.write_vcf(vcffile, individuals=indivlist, individual_names=indivn
ames)

# Fin du script Python.
```

```
5098387  sites restants après le retrait des loci à plus de 2
allèles.
2348675  sites restants après le retrait des loci de MAF <
0.005 .
-- Nombre de variants conservés:  30000
```

TreeSequence	
Trees	4827996
Sequence Length	2000000000
Time Units	ticks
Sample Nodes	29712
Total Size	695.7 MiB

Table	Rows	Size	Has Metadata
Edges	16636023	507.7 MiB	No
Individuals	14978	1.4 MiB	Yes
Migrations	0	8 Bytes	No
Mutations	30000	1.1 MiB	No
Nodes	2132052	57.9 MiB	Yes
Populations	3	2.3 KiB	Yes
Provenances	8	17.5 KiB	No

Sites	30000	732.4 KiB	No

Notre procédure de simulation est à présent terminée. Pour consulter les procédures de post-traitement des données, consulter le document R Markdown `POPSIZE_Script_PostProcessing_exempleFR` fourni en Annexe du rapport scientifique final du projet POPSIZE, ainsi que les sections I.3.3, OP.I.4 et OP.I.5 du même rapport.

Fin de document Jupyter Notebook.