Projet “Parazite”

Auteurs : Dariush Mollet, Léonard Jequier , Bastien Vallat

Cours Programmation pour biologistes
Dr. Alessandro Villa
Semestre de printemps 2017

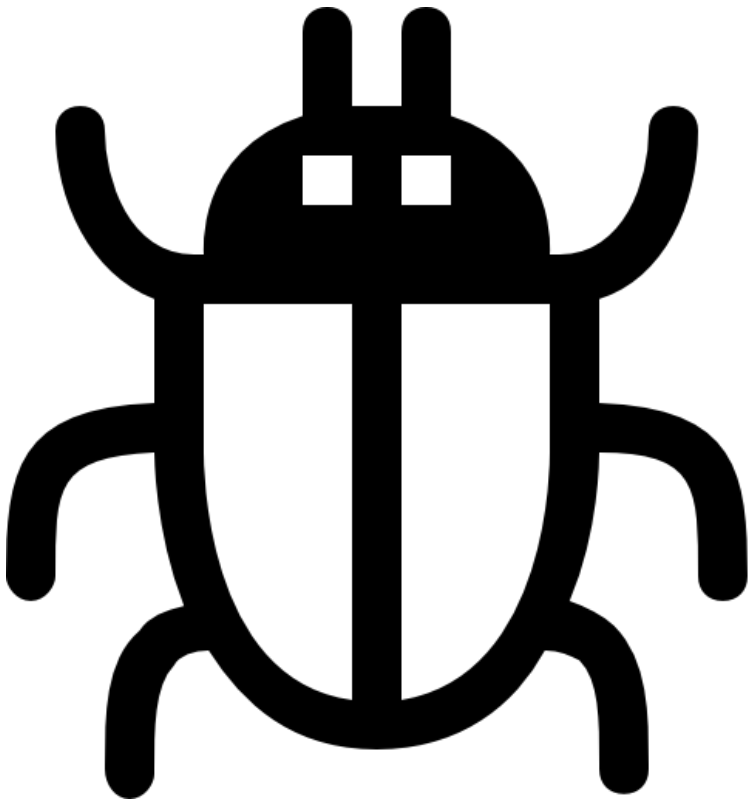


Table des Matières

Introduction

L'idée de départ du projet était de simuler de manière intuitive et dynamique une population animale ayant des interactions et dans laquelle se propagent des parasites.

Le parasitisme est une relation biologique ou l'un des deux protagonistes tire profit de l'autre pour accomplir son cycle de vie. Un aspect important de cette relation est l'évolution du parasite. Celui ci a intérêt à se transmettre d'hôtes en hôtes pour accomplir son cycle de vie. Pour cela, il devra maximiser son nombre d'infection secondaires. Dans les relations les plus simples, celui-ci peut être représenté mathématiquement comme ceci:

$$R0 = \frac{\beta \cdot N}{\mu \cdot \alpha \cdot r}$$

Pour le parasite, le numérateur représente la probabilité de se transmettre par unité de temps, et se décompose en N, le nombre d'occasion par unité de temps et β la probabilité de transmission à chaque occasion.
Le dénominateur représente la durée de l'infection et est composé de μ, la probabilité de mort naturelle de l'hôte, α, la probabilité de mort de l'hôte due au parasite, aussi appelée virulence et r, le taux de guérison. Chacune de ces probabilités est exprimée par unité de temps.

Pour maximiser le nombre d'infection secondaire, le parasite pourra, dans notre modèle, agir sur:

- sa virulence
- sa probabilité de transmission.
- le taux de guérison de l'hôte.

Seulement, ces trois facteurs sont liés par un autre facteur, la charge parasitaire. En effet, un parasite qui se multipliera beaucoup dans l'hôte augmentera son taux de transmission et laissera peu de chance de guérison à l'hôte, mais sera également très virulent car il lui prendra beaucoup d'énergie. Cela réduira la durée de l'infection et donc le temps qu'il a a disposition pour se transmettre. Inversement, si la charge parasitaire est trop faible, le taux de guérison élevé réduit la durée de l'infection et par extension le nombre d'infection secondaires.
Il s'agira alors pour le parasite d'optimiser son nombre d'infection secondaire en faisant des compromis entre ces facteurs. Ce compromis sera modélisé de deux manières différentes dans notre programme(voir au chapitre Parasite Set_new_vir, Set_new_transmrate, Set_new_RecovProb et au chapitre theory_tester2) .
Nos objectifs sont de créer un modèle qui nous permet de retrouver les résultats attendus en théorie, à savoir un nombre d'infection secondaires qui suit la formule ci-dessus et de montrer comment les parasites vont évoluer afin d'optimiser ces paramètres qui influencent leur nombre d'infections secondaires.

Méthodes

Résumé /fonctionnement général

Ce programme permet de lancer 3 modes de simulation :

Mode : “theory tester”

But : Ce mode permet de tester si le nombre d’infections secondaires dans notre modèle (R0 dans notre formule) suit bien ce qui est attendu d’après la théorie.

Particularités : Une seule souche de parasite est présente au départ et il n’y pas de création de nouvelles souches ni de mutations, les individus sains ne peuvent pas être résistants à ce parasite. On effectue plusieurs simulations en faisant à chaque fois varier “l’effect” qui représente la charge parasitaire. C’est un nombre compris entre 1 et 10 qui permet de calculer un compromis entre la virulence du parasite, le taux de transmission et la probabilité de guérison du parasite.

Résultats obtenus : Les résultats de chaque simulation sont inscrits sous forme de csv (un par simulation) dans le dossier data_effect_tester. Pour chaque simulation, on enregistre le nombre d’infections secondaires en fonction du temps.

Mode : “war”

But : Ce mode permet d’observer comment évolue la transmission dans un système plus complexe avec des phénomènes de mutations et de résistances. Ce mode est un peu trop complexe pour pour tester une théorie précise mais cela reste intéressant à observer.

Particularités : Dans ce mode on lance un certain nombre d'individus sains ainsi qu’un certain nombre de parasites de la même souche au début du programme, cependant les parasites peuvent muter et on verra donc l’apparition de nouvelles souches dont les attributs vont varier de manière aléatoire (virulence, transmission, guérison). Les individus sains peuvent également développer une résistance contre une souche si celui-ci a réussi à guérir après une infection.

Résultats obtenus : On obtient à la fin plusieurs fichiers. Le premier, data.csv, contient les données générales sur la simulation (nombre de parasites, virulence moyenne, taux de transmission moyen, nombre de parasites et individus sains en fonction du temps). Dans le dossier data un fichier csv est écrit pour chaque ayant existé et qui contient son historique.

Mode “all night long”

But: Permettre de faire tourner une simulation semblable au mode “War” indéfiniment

Particularité : Ce mode est similaire au mode “war” à la différence que dès que le nombre de parasite tombe à zéro, de nouveaux sont rajouté de façon à ce que la simulation continue. De plus lorsque la population devient trop grande, la mortalité augmente. Inversement, si la taille de la population descend sous un certain seuil, le taux de reproduction augmente. Cela permet de rester dans des taille des populations contrôlées.
Résultats obtenus : Idem que pour mode “war” simple

Librairies importées.

matplotlib – numpy

Matplotlib (et sa dépendance numpy) permet de créer des graphiques en 2D sous python.

Plus d’infos : https://matplotlib.org/

kivy

Kivy permet la création d’interface graphique assez poussées et multi plateformes. (Windows, linux, Android, MacOs, ...). Notre programme utilise donc cette interface.
Plus d’infos : https://kivy.org/#home

datetime/time

Permet de manipuler facilement les dates et le temps dans un programme
Plus d’infos : https://docs.python.org/2/library/datetime.html

csv

Cette librairie permet de créer, lire et écrire des fichiers csv sous python. Elle nous est ici utile pour enregistrer les valeurs obtenues pendant la simulation de manière générale (nombre d’individu infectés/sains en fonction du temps par exemple) et d’avoir un historique des souches de parasites qui ont existé.

os

Elle est utilisé dans notre programme pour vérifier l’existence d’un fichier csv laissé par une simulation précédente et pour créer un dossier data destiné à contenir les fichiers csv sur chaque souche.

Random

Le module random permet la génération de chiffres aléatoires. Il est utilisé dans notre programme pour attribuer les position des boules, pour leur donner une vitesse et une direction. Il est aussi utilisé pour générer des chiffre au entre 0 et 1 qui sont comparés au probabilités d’infection, de guérison, de mutation, etc.

Contenu des différents fichiers

main.py

Ce fichier contient quasiment l'ensemble du programme, il importe et regroupe les autres fichier du dossier (les constantes, classe des individus, sains , malades,...) lance l'interface graphique et exécute le programme. On peut le structurer en plusieurs parties

1. Importation des librairies, création des fichiers de données et des dictionnaires utilisés plus loin

On démarre le programme par l'importation de toutes les librairies et des autres fichiers python du répertoire qui contiennent les différentes fonctions appelées dans main.
On appelle tout d'abord la fonction seed afin d'avoir un résultat reproductible d'une fois à l'autre. Ensuite on crée plusieurs dictionnaires :

- dico_id = {}

format : {id:objet (pour tous les individus vivants)}

- balls_dictionnary = {} :

format : {id:[widget_ball,individual, position]}
Contient tous les objets « ball » vivants à un moment donné de la simulation. Chaque clé correspond à l'id de chaque individu, les caractéristiques de l'objet « ball », les caractéristiques de l'individu et finalement sa position.

- strain_dictionary = {}

format : {souche:[vir,transmission, guérison][liste des infectés]}
Contient toutes les souches qui existent ou qui ont existé. La clé correspond à l'ID du premier individu infecté (ex. Souche:1374,). La première entrée associée à cette clé est une liste qui va contenir les valeurs de virulence, taux de transmission et chance de guérison associé à cette souche. La seconde entrée contient la liste des ID tous les individus qui ont été infectés par cette souche.

- dico_of_strains_for_csv = {}

{souche:variable.writerow()}
Permet d'entrer pour chaque souche, un fichier csv qui sera mis à jour à chaque tour de la simulation et qui contient des informations sur l'évolution de la souche dans le temps.

- List_of_healthies

liste des individus sains vivants

- List_of_parazites

liste des parasites vivants

On veut ensuite créer un dossier data qui va contenir l'historique de chaque souche sous forme de csv. Pour cela on vérifie d'abord si le dossier data existe déjà (auquel cas il est supprimé pour effacer les valeurs de la simulation précédente). Cela s'effectue grâce au fichier del_useless_files.py.
Il faut ensuite créer le fichier data.csv dans lequel va être inscrit l'évolution de la simulation au cours du temps. A nouveau on vérifie si un fichier du même nom existe déjà et on propose à l'utilisateur de confirmer la suppression du fichier si c'est le cas.

2. Création/suppression des individus sains, ou parasites

def create_id()

Cette fonction est un compteur qui s'incrémente et qui permet de créer un nouvel ID qui pourra être donné à la création d'un nouvel individu.

def add_one_healthy() :

Cette fonction ajoute un individu sain.
Un id est créé en appelant la fonction create_id et est écrit dans la variable temp. On crée un nouvel individu de classe Healthy avec comme paramètre l'ID temporaire créé et on l'ajoute à la liste des individus vivants(list_of_healthies.append(Healthy(temp)). La fonction retourne ensuite l'individu créé.

Un try/except permet de s'assurer qu'une erreur dans la création d'un individu ne fait pas planter tout le programme. Si l'id qui vient d'être créé pour l'individu existe déjà ou si un autre problème survient, le programme n'est pas interrompu mais la création de l'individu n'a pas lieu et la fonction retourne un message d'erreur.

def add_one parazite (p = None, effect = None) :

Cette fonction ajoute un parasite à la simulation.
Comme pour la création d'un individu sain on commence par créer un ID unique. Ensuite, selon le type de trade_off choisi dans CONSTANTES.py, la fonction va créer des attributs virulence, taux de transmission et taux de guérison. Ensuite elle va utiliser le constructeur "Parazite" pour créer un parasite avec ces paramètres et va l'ajouter au différentes listes et dictionnaires décrits plus haut .

def kill(root,p):

Cette fonction élimine un individu indiqué en argument (p). D'abord on s'assure que l'individu existe, ensuite on vérifie à quelle liste il appartient (liste des infectés en vie ou liste des individus sains en vie). Finalement on l'enlève du dictionnaire des id et des balles, on le retire de l'interface et on supprime définitivement l'objet avec la fonction del.

def reproduce(root,p):

Cette fonction duplique un individu « p » donné en argument.
On crée une balle, on lui attribue une position aléatoire puis on l'ajoute à l'interface pour qu'elle soit affichée. Si l'individu "p" était parasité, son descendant sera immédiatement infecté par la souche de de "p", pour autant qu'il n'y ait pas de mutation lors de la reproduction.

def reproduce(p)

On reproduit l'individu p (parent). Pour positionner le nouvel individu on prend d'abord la position de p à laquelle on ajoute une valeur aléatoire en x et en y. On ajoute ensuite l'individu au BallContainer. Si le parent est parasité, le nouvel individu est aussi parasité par la même souche que le parent, on lance également la fonction de mutation aléatoire random_mutation_on avec comme argument "reproduction". Si le parent est sain, un nouvel individu de classe "Healthy" est ajouté et il acquiert les résistances de son parent.

3. Fonctions faisant appel au hasard et probabilités(mutations, risque de mourir, de guérir…)

def guerison(p):

Prend un parasite p en argument. On vérifie que p est bien un parasite, on tire une valeur au hasard et on la compare à la probabilité de transmettre la résistance, si cette valeur tirée au hasard est inférieure au seuil qu'on a fixé avec la constante TRANSMISSION_OF_RESISTANCE_PROB on ajoute la résistance à cette souche comme attribut à l'individu qui guérit. On retire l'individu guérit de la liste des parasites, on l'ajoute à la liste des individus sains et on change sa couleur en conséquence. SI p n'est pas un parasite on imprime « pas un parazite » dans la console.

def cure_the_lucky_ones(dt) :

On parcourt la liste de tous les parasites toutes les secondes. A chaque itération on tire un nombre au hasard entre 0 et 1 et on le compare à la constante BASE_CHANCE_OF_HEALING * la probabilité de guérison spécifique au parasite en question. Si le nombre tiré est inférieur on transforme le parasite en individu sain en appelant la fonction guérison.

def mutate_those_who_wish(dt) :

On parcourt la liste des parasites. Si la probabilité de mutation spontanée) de base (CHANCE_OF_MUTATION_ON_NOTHING est plus grande qu'un nombre au hasard entre 0 et 1, on utilise la fonction random_mutation_on pour faire muter un parasite, avec l'argument what = living

def kill_those_who_have_to_die(root,dt) :

On parcourt la liste des parasites et des individus sains, on compare une valeur tirée au hasard avec la constante DYING_PROB qui représente leur risque de mourir à chaque instant, si la valeur tirée est inférieure l'individu meurt. Les parasites ont plus de risque de mourir qui dépend de leur virulence car la valeur DYING_PROB est multiplié par la valeur de virulence du parasite et qui dépend de chaque souche.

def reproduce_those_who_have_to(root,dt) :

On parcourt la liste de tous les individus sains, puis celle des parasites. Pour chaque individu on tire une valeur au hasard et on compare avec la constante REPRODUCTION_PROB, si la valeur tirée au hasard est inférieure à la constante, on appelle la fonction reproduce et l'individu se reproduit.

def random_mutation_on(para_i, what) :

Cette fonction lance un test de probabilité pour savoir si une mutation aléatoire va avoir lieu et créer une nouvelle souche.
Un simple test détermine quel type de mutation on veut effectuer :

- En cas de nouvelle infection :
  - CHANCE_OF_MUTATION_ON_INFECTION
  - MAX_FITNESS_CHANGE_ON_INFECTION
- En cas de reproduction :
  - CHANCE_OF_MUTATION_ON_REPRODUCTION
  - MAX_FITNESS_CHANGE_ON_REPRODUCTION
- Tout au long de leur vie :
  - CHANCE_OF_MUTATION_ON_NOTHING
  - MAX_FITNESS_CHANGE_ON_NOTHING

Si le trade_off est définit comme 'dada' dans CONSTANTES.py:
Un dictionnaire de 3 fonctions est créé, contenant les set_New_Vir, set_New_Trans et set_New_Recov. On appelle aléatoirement une des trois fonctions afin d'incrémenter (ou décrémenter) l'attribut correspondant d'un ratio aléatoire maximisé par le MAX_FITNESS_CHANGE correspondant.

Si le trade_off est définit comme 'leo' dans CONSTANTES.py
On utilise la fonction trade_off(voir chapitre trade_off.py) afin de créer les nouveaux paramètres du parasite.

Dans les deux cas:
A nouveau, on tire une valeur au hasard entre 0 et 1, on la compare à la variable qui dépend de la condition dans lequel se trouve le parasite.
S'il y a mutation alors une nouvelle souche apparaît et elle correspond à l'id du parasite duquel provient cette nouvelle souche.Par exemple, le parasite avant l'ID102 créera la Souche-102 correspondante. On crée donc l'ID de la souche, on inscrit cette ID comme clé dans le dictionnaire des souches et on lui attribue une liste de nouvelles valeurs : [nouvelle valeur de virulence, taux de transmission, taux de guérison] ainsi qu'une deuxième liste qui va contenir la liste des individus infectés par la souche et qui ne contient à ce moment-là que l'individu qui lui a donné naissance.

4. infections, collisions, rencontres entre parasites

def infect_him(para_i,heal_i, parazites_reproducing=False) :

Cette fonction est appelée dès qu'un collision entre un parasite et un individu sain peut entraîner une infection. On prend d'abord la souche du parasite et on regarde si elle apparait dans la liste des résistances de l'individu sain. Si l'individu sain est résistant, la fonction ne fait rien de plus et l'infection n'a pas lieu. Si l'individu n'est pas résistant à cette souche il est infecté, il est donc ajouté à la liste des parasites en reprenant tous les attributs du parasite qui l'a infecté à l'excepetion de son id. Il est ensutie retiré de la liste des inidividus sains. Comme il y a infection on appelle la fonction random_mutation_on avec comme attribut « infection » pour savoir si il y a mutation et apparition d'une nouvelle souche.

def parazite_against_parazite(p1,p2)

S'il y a collision entre deux parasites, le plus fort a une probabilité d'éliminer l'autre et prendre sa place. La force relative des parasites entre eux est représentée par l'attribut leur valeur de virulence, le parasite le plus virulent peut potentiellement éliminer l'autre . On effectue un test en tirant une valeur aléatoire et on la compare avec la chance d'infection (INFECTION_CHANCE * Transmission rate propre au parasite) du parasite le plus fort. Si on tire une valeur inférieure, le parasite le plus faible est éliminé. L'individu le plus faible prend donc toutes les valeurs du parasite qui a gagné y compris sa souche et l'id de l'individu infecté par le parasite gagnant est mis à jour dans le strain_dictionary.

def actions_when_collision(p1,p2):

On compare les classes des individus qui sont entrés en collision. Pour cela on a un tuple qui contient les valeurs [Healthy, Parazite, Parazite], on regarde la classe de p1 est on la retire du tuple. On a alors 2 possibilités :

- P1 est de type Healthy, le tuple contient alors désormais [Parazite,Parazite]
  On teste si p2 fait partie d'une classe qui reste dans le tuple.
  - SI p2 est de classe Healthy on voit qu'il ne fait pas partie du tuple et donc la condition n'est pas vraie, il ne se passe rien.
  - Si p2 est de classe Parazite, il est dans le tuple et la condition est vraie. On est alors dans le cas d'une collision entre un parasite et un individu sain et on va donc tiré une valeur au hasard et tester si il y a infection ou pas en fonction de la constante INFECTION_CHANCE et du taux de transmission du parasite

- P1 est de classe Parazite , le tuple contient alors désormais les valeurs [Healthy, Parazite]
  On teste si p2 fait partie d'une classe qui reste dans le tuple. Ce qui sera forcément le cas.
  - Si p2 est un Parazite on est d'une collision parasite-parasite, on effectue alors un test pour savoir si ils vont s'affronter dont la probabilité dépend de la constante PARAZITE_FIGHT_CHANCE
  - Si p2 est de classe Healthy, on est dans le même cas de collision Healthy-parasite décrit ci-dessus.

5. Boucle principale

class mainApp(App) :

Cette classe représente le programme en lui-même.

def build(self) :

root = BallsContainer() est la fenêtre du programme, il s'agit donc d'un objet de type BallContainer qui sera décrit ci-dessous.
Les commandes précédées de Clock.schedule_one ne vont être effectuée qu'une fois au lancement du programme.

C'est le cas de Clock.schedule_once (root.update_files,1.1) qui va lancer la fenêtre, de Clock.schedule_once(root.start_balls,1) qui va générer dans l'interface et afficher les balles d'après les différentes dictionnaires et qui va les mettre initialement en mouvement lorsqu'oj lance le programme ou encore de Clock.schedule_once(root.update_life_and_death,1.1) qui va tuer/générer des nouveaux individus pour la première fois.

A l'inverse, le code précédé de Clock.schedule_interval va être effectuée régulièrement à un rythme donné par la constante DELTA_TIME. C'est le cas des fonction update et udpate_life_and_death qui sera bordée en détail dans le chapitre suivant.
Window.bind(on_key_down = root.Keyboard) permet de lancer des fonctions directement en appuyant sur de touches de son clavier pendant que le programme s'éxécute.

6. BallsContainer(Widget)

Classe du widget qui contient les boules, c'est quelque chose de propre à kivy. Pause = False signifie que le programme n'est pas en pause (si si, sérieux). Il faut préciser que num_healthies, num_parasites, et toutes ces variables sont de type NumericProperty (plus d'info ici : https://kivy.org/docs/api-kivy.properties.html)

def start_balls(sefl,dt)

Cette fonction est appelée tout au début, elle crée les objets (widgets et individus) et les lie. Elle est divisée en deux parties (très similaires). On veut créer un certain nombre de Healthies et de Parazite. On passe en argument le nombre de ces individus et lance une boucle :

- on appelle le constructeur du widget Ball()
- et on lui donne des attributs (position, vitesse) et on le rajoute comme widget enfant de BallsContainer
- on appelle le constructeur de Healthy
- et on lie ces deux objets au travers du dictionnaire balls_dictionnary

La boucle appelant les parasites est très similaire, excepté que si des arguments sont passé à main.py dans lors du lancement (et qu'ils sont au nombre de 2 - autrement dit on a appelé le script via theory-tester2.py), il passe le second argument à add_one_parazite afin de créer un parasite sur mesure.

La dernière ligne, de la même manière, fait en sortes que si un seul individu a été créé (via theory-tester2.py), il se reproduise instantanément 10x (afin d'éviter qu'il ne meure dans les premières secondes, par manque de chance).

def update(self, dt)

[le fonctionnement du quadtree (et son appel) sont expliqués sous la section 'Quadtree'.]

Update est appelé par l'application principale 60x/seconde.Elle gère donc principalement la mise à jour des positions et multiples collisions des objets.

- Dans la 1e boucle itérant à travers les objets, elle insère les objets ([[position], idd]) dans le quadtree et en profite pour mettre à jour la position dans le dictionnaire balls_dictionnary selon les positions des widgets balls (rappel, le fait de lier les positions comme valeurs directs des id nous simplifie le code lorsque l'on veut accéder aux positions : on n'a pas besoin d'accéder aux widgets)
- Dans la 2e boucle itérant à travers les objets, on récupère les id des objets pouvant potentiellement entrer en collision avec le premier objet. On crée un 2e dictionnaire (other_balls) ayant la même structure que le dictionnaire principal mais contenant uniquement les clés des balles susmentionnées.
    - On teste alors s'il y a collision avec la fonction physical_collisions2 (de collision.py - pour information, c'est une copie de physical_collision(1) qui reçoit les idd des balles en argument supplémentaire)
        - s'il y a collision, on appelle la fonction qui gère les multiples actions entre deux objets (selon leurs types, etc.)

        - on profite de cette 2e boucle pour mettre à jour les collisions avec les murs et la position des objets à t+1 (grâce à update(dt))

def update_life_and_death(self, dt)

Appelée tous les dt (donc normalement toute les secondes) dans le build. Cette fonction met régulièrement à jour les morts, guérisons et reproductions. En mode theory_tester, elle appelle la fonction shall_we_kill_the_simulation qui ferme la simulation (détaillée un peu plus bas) et en mode all_night_long, la fonction all_nighter qui maintient la population de parasite à une taille raisonnable pour l'ordinateur.

def update(self, dt, filename = None)

Appelle les fonctions qui mettent régulièrement à jour l'écriture dans les fichiers à savoir update_numbers et update_data_files (détaillées légèrement plus bas), elle est appelé tous les dt (soit normalement toutes les secondes) comme update_life_and_death.

def all_nighter(self)

Cette fonction permet de maintenir la population a une taille raisonable en mode all_night_long. D'abord on va chercher les variable REPRODUCTION_PROB et DYING_PROB en dehors de la fonction avec global. On effectue ensuite plusieurs tests avec if, chacun de ces tests nécessite que le mode all_night_long soit activé (mode == "all_night_long") ensuite qu'une condition correspondante :

- si il n'y a plus de parasites (c'est à dire que len(list_of_parasites) <1):
  On lance une boucle qui va créer un nouveau parasite, cette boucle se répète un nombre de fois qui est défini par la variable NB_PARASITES.

- si il y a trop de parasites (len(list_of_parasites) > 300)
  La probabilité de mourir de base DYING_PROB est augmentée car elle est remplacée par la valeur de ROOF_DYING_PROB qui va baisser la durée de vie de tous les individus pour diminuer leur nombre

- si il y a trop d'individus sains (len(list_of_healthies) > 300)
  Même chose que si trop de parasites, DYING_PROB prend la valeur deROOF_DYING_PROB

- si la somme des individus et des parasites est inférieure ou égale à 250
  DYING_PROB reprend sa valeur normale STOCK_DYING_PROB

- si la somme des individus et des parasites est inférieure à 50
  Dans ce cas la population doit pouvoir croître plus rapidement, son taux de reproduction de base REPRODUCTION_PROB est augmenté en prenant la valeur de BOTTOM_DYING_PROB

- si la somme des individus et des parasites est supérieure à 50
  Le taux de reproduction est restauré à son état de base STOCK_DYING_PROB

def update_numbers(self, filename = None)

Cette fonction met à jour les chiffres affichés dans la fenêtre de l'interface.
L'attribut nombre de parasites ou de healthies vivants (respectivement self.num_parazites et self.num_healthies) et calculé d'après la longueur de la liste des individus sains ou des parasites vivants (len(list_of_parazites ou list_of_healthies)).
Les variables sumvir, sumrecov et sumtrans sont les sommes des valeurs des parasites (utilisées pour calculer les valeurs moyennes de la population).
Tempdic est un dictionnaire temporaire des parasites (utilisé pour mesurer le nombre de parasites issus d'une certaine souche)et la liste top_idds contient la liste des trois souches les plus fréquentes (afin de les afficher sur le côté de la GUI)

- la 1e boucle passe dans la liste des individus et additionne les valeurs moyennes dans sumvir/trans/recov. Elle append à tempdic les souches {souche : [nombre d'ind, idd]}
- la 2e boucle cherche dans le dictionnaire les idd avec le nombre d'individu le plus élevé (en utilisant la fonction idd_max 3x de suite) et les place dans la liste top_idds (avec leurs attributs/nombre d'individus). Cette liste est ensuite utilisée par main.kv afin d'afficher des informations agréables
- on profite de la fonction afin de gérer les temps de pause (on soustrait le temps perdu)
- et enfin, on écrit quelques données dans des .csv, selon le type de mode
    - si le nombre d'arguments est >2 (theory_tester2) : on rajoute le nombre d'infections secondaires (donc le nombre d'individu de l'unique souche)

    - s'il est > 1 (theory_tester) : on rajoute les valeurs moyennes et les nombres de Parazites/Healthy

- les dernières lignes sont des 'magouilles' afin de régler des problèmes d'affichage graphique. En effet, Kivy lance les widget dans un ordre qui n'est pas toujours le même. Comme on veut s'assurer que les panneaux latéraux (nombre de parasites/attributs des top_idds et autres) soient toujours au premier plan (et qu'on ne voulait pas perdre de temps à comprendre comment mieux utiliser cette interface graphique), on utilise une méthode simple :
- on cherche tous les widgets enfants de BallsContainer (dans l'ordre que l'on souhaite)
- on les supprime et les rajoute
- on en profite aussi pour rajouter 3 boules immobiles (et non indexées) qui affichent les couleurs des top_idds, pour plus de lisibilité

Le try/except permet d'éviter de faire planter le programme si list_of_parazites est vide car on risque une division par zéro.

def shall_we_kill_the_simulation(self, dt)

En mode 'theory_tester', il faut arrêter la simulation lorsque le temps imparti est écoulé ou que les parasites ont disparus, afin de lancer la simulation suivante. C'est le rôle de cette fonction. Elle est appelée dans update_life_and_death toutes les secondes(voir ci dessus).

def update_data_files(self, dt)

Cette fonction prend dt en argument, elle est appelée dans update_files, toutes les secondes.
En mode 'war' et 'all_night_long', on crée des fichiers csv qui contiennent, en colones, le temps, le nombre d'individu dans la population, le nombre de parasites, le nombre d'individus, le pourcentage de parasites, la virulance moyenne, la transmission moyenne et le taux de guérison moyen.

def on_pause(self)

Cette fonction découple les fonctions update, update_life_and_death, et update_files de l'horloge de Kivy. Event est défini dans build et contient:
Clock.schedule_interval(partial(root.update_files, filename = args), 60*DELTA_TIME), où root = BallsContainer(). La fonction update_files doit être découplée ainsi car elle prend un argument.

def on_resume(self)

Cette fonction permet de coupler à nouveau les fonction en pause à l'horloge de Kivy.

def on_stop(self)

Cette fonction ferme l'application. Elle est appelée dans shall_we_kill_the_simulation et dans Keyboard (voir ci-dessous).

def on_touch_down(self, touch)

C’est une fonction par défaut de Kivy, quand on clique sur la fenètre et que la simulation est sur pause, elle affiche un graphique du nombre d’infection secondaire de chaque souche en fonction de sa virulence.

def Keyboard(self, window, keycode, *args)

Cette fonction est en partie gérée par Kivy. Elle est paramétrée pour lancer la fonction on_pause lorsqu’on appuie espace Pour reprendre, il faut appuyer à nouveau sur espace et la fonction on_resume sera lancée.
La flèche gauche permet de ralentir les balles et la flèche droite de les accélérer.
Pour finir, la touche S arrête le programme.

def idd_max(self,dico)

Cette fonction est appelée dans update numbers. Elle permet de parcourir le dictionnaire donné en argument. Elle stocke le nombre d’individu vivants infectés par cette souche dans la liste a et stocke le souche correspondantes dans la liste b. Elle retourne finalement les trois souches qui ont infectés le plus d'hôtes au moment ou la fonction est appelée.

CONSTANTES.py

À l'intérieur du dossier parasites, le fichier CONSTANTES.py contient tous les paramètres qui sont utilisés comme constantes pendant la simulation. On peut donc y entrer les valeurs souhaitées avant de lancer le programme.

Mode
3 valeurs possibles : "war" "all_night_long", "theory_tester"

Mode theory tester		
SIMULATION_TIME	300	Durée maximal d'une simulation (en secondes)

Mode all night long		
STOCK_DYING_PROB	0.13	Probabilité de base de mourir pour chaque individu
ROOF_DYING_PROB	0.2	Probabilité de mourir lorsque la population a atteint son maximum, elle doit être élevée afin que la pop. diminue au lieu d'augmenter
STOCK_REPRODUCTION_PROB	0.13	Probabilité de reproduction de base des individus
BOTTOM_REPRODUCTION	0.15	Probabilité de reproduction lorsque la population a atteint le seuil minimum, doit être élevée pour que la pop ne s'éteigne pas

GUI		
DELTA_TIME	1/60.0	SI la valeur est 1, le mode all_night_long est activé
MAX_BALL_SPEED	100	Vitesse maximale des individus
BASE_COLOR	[0,0,1]	Couleur de base des individus (ne se verra que chez les Healthy)

Main : paramètres généraux		
TRADE_OFF	« leo »	Type de trade-off choisi pour le mode war et all night long

NB_SAINS	100	Définit le nombre d'individus sains initial
NB_PARASITES	1	Définit le nombre d'individus parasités initial
MAX_VELOCITY	1	Vitesse max des individus parasités
MAX_VIRULENCE	1	Virulence maximale des parasites
NB_OF_FILES_TO_KEEP	3	Nombre de fichiers csv de souches que l'on souhaite garder (les n plus grands)

Individual : relatif à tous les individus		
DYING_PROB	0.13	Probabilité de mourir des individus
REPRODUCTION_PROB	0.13	Probabilité de reproduction (duplication) d'un individu

Healthy : relatif aux individus sains		
TRANSMISSION_RESISTANCE	1	Prob. d'un individu sain de transmettre sa résistance

Parazite : relatif aux parasites		
BASE_FITNESS	0.8	Valeur de base du trade-off (si trade_off = dada)
MAX_FITNESS	1.8	Valeur max du trade-off (si trade_off = dada)
INFECTION_CHANCE	0.4	Probabilité d'infection en cas de contact
BASE_CHANCE_OF_HEALING	0.1	Chance de base de guérison d'un individu parasité
CHANCE_OF_MUTATION_ON_INFECTION	0.1	Probabilité de mutation à chaque infection
MAX_FITNESS_CHANGE_ON_INFECTION	0.2	Changement de fitness max si mutation par infection
CHANCE_OF_MUTATION_ON_REPRODUCTION	0.2	Probabilité de mutation à chaque reproduction

MAX_FITNESS_CHANGE_ON_REPRODUCTION	0.2	Changement de fitness max si mutation par reproduction
CHANCE_OF_MUTATION_ON_NOTHING	0.05	Probabilité de mutation à chaque instant
MAX_FITNESS_CHANGE_ON_NOTHING	0.05	Changement de fitness max si mutation aléatoire
PARAZITES_FIGHT_CHANCE	0.5	Prob. de combat quand 2 parasites se rencontrent

Le dossier nice_const_values

ce dossier contient un certain nombre de presets (fichiers CONSTANTES.py et CHANG_CONST.py) prédéfinis pour tester le code.

Le preset WAR propose une expérimentation de base où les parasites ont un haut taux de devoir se mesurer aux autres parasites. Ceci conduit généralement à une sélection pour une haute virulence qui finalement mène à l'extinction de la population. Le trade-off utilisé dans ce cas est

Le preset Stable propose des valeurs des probabilités de compétition entre parasites plus bas et repeuple automatiquement le programme lorsque le nombre de Parazites tombe à 0.

Les presets theory-tester et theory-tester2 sont à utiliser lorsque l'on veut faire tourner un grand nombre de simulations pour acquérir des données.

Afin d'utiliser correctement ce dossier, il suffit de copier coller les deux fichiers de constantes dans le dossier src courant. Bien sûr, on peut par la suite modifier des paramètres individuels.

collision.py

Les individus dans notre simulation étant représentés par des balles rebondissantes, ce fichier décrit leur comportement physique dans l'interface graphique. On a donc une classe « Ball » avec ses attributs et différentes fonctions qui vont gérer les différents types de collision qui peuvent arriver.
IL faut donc importer un certain nombre de modules lié à kivy (permettant notamment d'avoir des vecteur, la différence de temps,...), le module math pour avoir accès notamment aux différentes fonctions trigonométriques math.sin, math, sqrt, math.cos,etc... ainsi qu'aux paramètre de la simulation du fichier CONSTANTES.py.

class Ball(Widget)

Cet objet existe déjà dans la librairie kivy (en tant que Widget) et nous n'avons donc pas besoin de créer un constructeur.

L'objet Ball est défini par sa vitesse dans l'axe x et y ainsi qu'un angle dans lequel il va et une couleur. La vitesse en X et Y est regroupé sous une seule variable appelée velocity qui sera notamment utilisée dans la fonction self.update.
Il faut préciser le sens des axes : lorsqu'une balle se déplace de gauche à droite, sa position en x augmente avec le temps et donc sa vélocité sera positive, si à l'inverse elle va de droite à gauche, sa vélocité sera négative. Pour l'axe Y, si la balle se dirige vers le haut elle aura une vélocité en Y positive et si elle se déplace vers le bas elle sera négative.

def update(self)

Met à jour la position de la balle après un déplacement. La nouvelle position self.pos = au vecteur (sens et orientation) dans lequel la balle se déplace multiplié par sa vitesse (velocity, en x et y) multiplié par la différence de temps et ce depuis sa position précédente.
def get_col(self), def set_col(self, a)
la fonction set_col donne une couleur à la balle selon le paramètre a, la fonction get_cot renvoie la couleur de la balle.

def physical_collision2 (ball, other ball)

Comportement de deux balles lorsqu'elles entrent en collision.

def physical_wall_collisions

Comportement d'une balle qui rebondit contre une paroi. Un « mur » invisible se situe à l'intérieur de la fenêtre de l'interface à exactement la distance du rayon d'une balle afin que lorsque le centre de la balle atteint ce mur, le rayon extérieur de la balle va heurter la fenêtre du programme et on aura alors une collision propre.
Par exemple on voit dans la première condition que si le centre de la balle (balle.x) se situe plus à gauche que le mur (wall.x) alors qu'elle se déplace vers la gauche (velocity < 0), sa direction s'inverse et la balle rebondit dans le sens inverse à la même vitesse (collision élastique). Idem à droite, si le centre de balle se situe plus a droite que le paroi droite du mur et qu'elle se dirige vers la droite (velocity > 0), alors elle rebondit et pars à gauche avec la même vitesse. Ce mécanisme est identique pour l'axe Y.

individual.py

Ce fichier contient la classe "individual » ainsi que des fonctions permettant d'accéder aux caractéristiques d'un objet qui serait de ce type. On a d'abord l'importation de la librairie random (qui sert dans les classes qui en héritent : Healthy et Parazite) et des choix des paramètres de la simulation qui sont inscrit dans le fichier CONSTANTES.py

class Individual

def _init_(self)

Le constructeur de la classe. Un objet de type individu est définit par sa couleur, son id et sa résistance

def __ str__(self)

Permet d'afficher plus proprement les attributs de l'objet quand on le print directement.

def __del__(self)

Permet de supprimer l'objet
def getSpeed, def getIdd, def getResistance, def getPosition
Renvoie respectivement la vitesse de l'objet, son ID, sa résistance ou sa position.

def addResistance

Les résistance sont stockées sous forme d'une liste qui contient les Souche desquels ont guéri l'individu si RESISTANCE_AFTER_RECOVERY en défiit comme 1 dans CONSTANTES.py.Il possèdera aussi les résistance de ses parents, si TRANSMISSION_RESISTANCE est définit comme 1. L'individu ne pourra alors plus être infecté par un parasite de cette souche.

healthy.py

Ce fichier contient la class Healthy qui est donc celle des individus sains. Elle hérite de tous les attributs de la classe "Individual". Il faut alors importer au début individual.py.

class Healthy(Individual)

def _init_(self)

Hérité de "individual" (au-dessus)

def __ str__(self)

Hérité de "individual" (au-dessus)

parazites1.py

Contient la classe parasite et les fonctions qui lui sont liées. Comme Healthy cette classe hérite également de Individual on importe donc en début de programme individual.py ainsi que deux autres libraires : __future__ et numpy.

classe Parazite(Individual)

def _init__(self, vir, rate, rec, idd, par=[ ], strain = [ ])
On hérite des attributs de individual et on en ajoute d'autres spécifiques aux parasites à savoir :

- La virulence

- Le taux de transmission
- La probabilité qu'un individu atteint guérisse
- La parenté
- La souche

def_str_

Pareil que pour la classe « Individual »

def set_New_vir(self,r)

def set_New_Transmrate (self,r)

def set_New_RecovProb, (self,r)

Ces trois fonctions permettent de définir de nouvelles valeurs de virulence/taux de transmission/taux de guérison et s'assurant que la fitness totale de l'individu ne dépasse pas un certain seuil - et que chaque attribut ne peut pas dépasser son propre seuil maximum, défini comme 1 (ou passer sous le seuil minimal : 0). C'est la première façon de modéliser les compromis des parasites décrits dans l'introduction.
Nous allons va utiliser set_New_vir comme exemple pour montrer leur fonctionnement en gardant à l'esprit que les deux autres fonctions sont similaires.

- pour commencer, il est important de comprendre que la fitness est définie comme la somme (linéaire) des trois valeurs. Ainsi, il est considéré qu'une virulence de 1 (autrement dit, une mortalité augmentée de 100%) est équivalente à un taux de transmission de 1 (augmenté de 100%) ou d'une recovery de 1 (probabilité de guérison = taux de base. Une recovery de 0 signifie que la probabilité de guérison est inchangée (par rapport au taux de base, une recovery de 1 signifie que le taux de guérison est réduit à 0)
- on commence par calculer la différence de fitness engendrée par le changement
- on stock le signe (et le signe inverse) de cette différence (pour des raisons pratiques)
- on essaie par 5 de trouver des valeurs nouvelles de transmission et de recovery qui ne nous fassent pas dépasser le seuil de fitness maximum
    - pour cela il faut savoir qu'une virulance qui augmente (autrement dit une charge parasitaire qui augmente) signifie généralement que le taux de transmission augmente également et que le taux de guérison diminue
    - et donc on définit aléatoirement, et dans la fourchette de valeurs disponibles (si la transmission doit monter alors qu'elle vaut 0.9, elle ne peut pas augmenter de plus de 1 - 0.9 = 0.1)
    - et comme mesure de sécurité, on attribue tout de même nos nouvelles valeurs à l'intérieur de min et de max, afin de s'assurer que celles-ci ne dépassent pas les seuils maximaux

Les fonctions set_New_Transmrate et set_New_RecovProb sont similaires, aux détails prêts qu'une augmentation en taux de transmission signifie que la virulance augmente et que la probabilité de guérision diminue - et qu'une probabilité de guérision qui augmente signifie que la virulence et le taux de transmission diminuent.

def getPar(self), getvir(self), getTransmrate(self), getRecovProb(Self), getStrain(Self)

Renvoie la parenté, la virulence, le taux de transmission, la probabilité de guérison et la souche de l'objet parasite en question.

def setStrain,

Définit la souche du parasite.

def getTotalFitness

Retourne la valeur de fitness totale, définie par la virulence, le taux de transmission et le taux de guérison pour le parasite.

del_useless_files.py

def delete_useless_files(nb_of_files_to_keep)

En lançant la simulation en mode war et all night long, on se retrouve rapidement avec des milliers de fichiers csv qui contiennent l'historique de chaque souche qui a existé dans le dossier "data". Comme la plupart de ces souches n'ont pas survécu très longtemps ces données sont sans intérêts. Cette fonction qui est destinée à être lancée manuellement après une simulation permet d'effacer les plus petits fichiers en de n'en garder que les n fichiers les plus gros qu'on souhaite. On choisit le nombre de fichiers que l'on souhaite garder en modifiant la valeur de NB_FILES_TO_KEEP dans le fichier CONSTANTES.py. Par exemple si on veut garder les 3 plus gros fichiers on va mettre comme la valeur de NB_FILES_TO_KEEP à 3.

Cette fonction parcourt le dossier data et itère chaque fichier qui s'y trouve pour récupérer sa taille. Toutes les tailles de fichiers sont contenues dans une liste. On met cette liste dans l'ordre et on sélectionne la valeur de la [-nb_of_files_to_keep] valeur de la liste que l'on souhaite garder. On donne cette valeur à une variable appelée NB_OF_FILES_TO_KEEP. On itère de nouveau la liste des fichiers et on récupère leur taille qu'on compare avec NB_OF_FILES_TO_KEEP, si elle est inférieure le fichier est supprimé. On ne se retrouve donc qu'avec les n plus gros fichiers du dossier data.

quadtree.py

Cet algorithme nous permet d'optimiser les collisions. Une solution naïve afin de générer des collisions aurait été d'itérer sur chaque objet et de tester s'il est suffisamment proche des autres objets :

```
for i in boules :
        for j in boules restantes :
                collision ?
```

néanmoins avec une complexité d'O(n^2) on s'est très vite rendu compte que c'était un facteur limitant à nos simulations (un exemple de vieux code traîne - gui1_old.py - afin de comparer).

Le quadtree est un algorithme de division de l'espace (par 4 en 2D, on utilise un octtree en 3D) afin de limiter le nombre de tests de collisions. Une méthodologie était proposée par Steven Lambert[1] en 3 étapes :

- split (crée des zones)
- insert (place les objets dans ces zones)
- fetch (retourne les objets dans la même zone que X)

Le programme de base avait été crée de sortes à ce qu'il reçoive des positions (liste de 4 coordonnées) en argument. Il a par la suite été modifié afin qu'il reçoive une liste de ces positions et l'IDD de l'objet en question ([[pos], idd]) afin que le main.py puisse retrouver quel objet correspond à quel widget (kivy). Pour l'explication de cet algorithme, on va ignorer les idds - il ne faut simplement pas oublier que le fichier en soi les reçoit et les retourne également.

Dans cette classe, tous les objets sont representés par une liste de 4 éléments : [x, x+delta_x, y, y_delta_y]. Autrement dit, un carré (ou un losange).
la classe Quadtree a 4 attributs :
objects qui contient les objets, box qui contient les quadtree enfants, actual_level qui correspond au niveau (de profondeur, le quadtree parent est le niveau 0, les premiers enfants sont au niveau 1 etc.) et space qui délimite les coordonnés (selon la liste définie au-dessus) du Quadtree.
Il a deux attributs globaux qui sont le nombre d'objets maximum par quadtree et le niveau maximal du quadtree. Après quelques tests, il nous a semblé que des valeurs de 4 et 5 tenaient très bien la route pour 100<x<1200 individus

Pour la suite, il est important de comprendre que cette classe compte beaucoup sur des fonctions récursives, car chaque tree peut contenir 4 autres trees (dans box).

Les fonctions __init__, add_objet et remove_object sont assez évidentes (constructeur, rajoute et enlève un objet (une liste de positions) au niveau actuel).

def split(self) :

cette fonction divise l'espace actuel en 4 (en les milieux) et crée des Quadtree enfants qu'il rajoute à son attribut box (dans l'ordre en bas à gauche, en bas à droite, en haut à gauche, en haut à droite)

def index(self,obj) :

cette fonction renvoie le quadtree dans lequel il peut être inseré. Concrètement, il vérifie juste que le carré entre en entier dans l'une des 4 divisions de l'espace. S'il ne peut pas être inséré entièrement (il traverse une des deux médianes) alors il reste dans le quadtree actuel. Cette fonction renvoie un int entre -1 (parent) et 0, 1, 2 ou 3 (les 4 cases)

[1] https://gamedevelopment.tutsplus.com/tutorials/quick-tip-use-quadtrees-to-detect-likely-collisions-in-2d-space--gamedev-374

def insert(self,obj,idd):

cette fonction est celle qui insère l'objet dans le quadtree correspondant. C'est une fonction récursive qui fait les actions suivantes, dans cet ordre :

- récupère l'index de l'objet (dans quelle case est-ce que je fit ?)
- si l'index vaut -1, je suis inséré dans le quadtree actuel (car je ne fit pas plus bas)
- si l'index vaut autre chose, c'est que je fit plus bas. Dans ce cas :
    - je split le quadtree actuel
    - je prends tous les objets que je contiens et pour chaque objet :
        - je récupère leur index
        - et si cet index est != -1, je les insère plus bas (et les enlève de mes objets)

def fetch(self, obj, idd) :

c'est la fonction qui retourne tous les objets qui sont dans le même quadtree, ou au-dessus de l'objet reçu en argument - autrement dit, tous les objets avec lesquels l'objet pourrait collisioner.

- récupère l'index de l'objet
- si l'index est != -1, se rend dans le quadtree correspondant et fetch l'objet en récursif
- sinon il append tous les objets présents, et tous ceux des quadtree parents en resortant de la récursive
- il retourne cette liste d'objets

le quadtree est ensuite utilisé de cette manière (dans main.py) :

- le quad parent est construit avec arguments (niveau 0, sa taille qui est la taille du widget BallContainer et donc de la fenêtre)
- pour tous les individus, on les insère dans le quadtree (O(n))
- pour tous les individus, on fetch ceux qui collisionnent (potentiellement) (O(n))
    - on teste s'ils collisionnent effectivement, et si oui : collision (O(?) difficile à calculer mais certainement moindre que n)

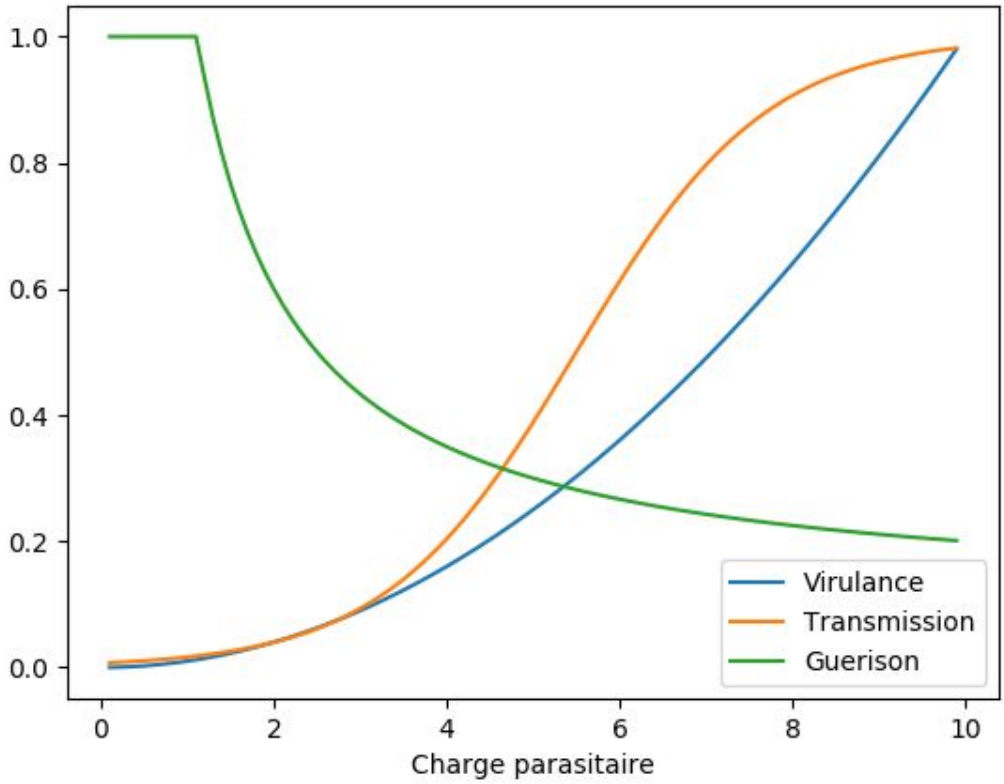
trade_off.py

def trade_off(para_i = None, effect_arg = None)

Cette fonction retourne une liste de trois valeurs: une virulence α, un taux de transmission β et un taux de guérison **r**. Ces trois valeurs sont calculées à partir d'une valeur "effect" entre 0 et 10 qui représente la charge parasitaire.Les paramètre se calculent ainsi avec effect = x:

$$\alpha = \frac{X^2}{100} \qquad \beta = \frac{1}{1 + e^{(-(x/1.1 - 5))}} \qquad r = 0.1 + \frac{1}{x}$$

On peut les représenter graphiquement:

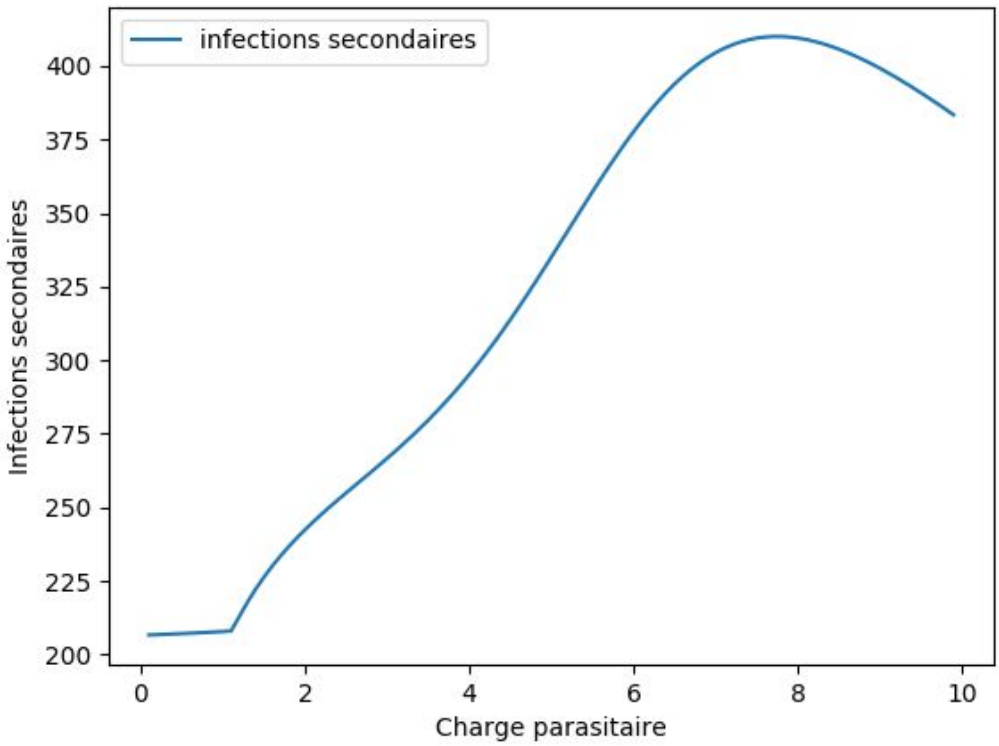


Ces formules ont été choisies arbitrairement, mais correspondent à ce qu'on peut observer dans la nature: la virulance et la transmission augmentent avec la charge parasitaire et le taux de guérison diminue quand la la charge parasitaire augmente.

Cette manière de calculer les trade_off permet de calculer plus facilement les paramètre qui maximise le nombre d'infection secondaire théorique. Pour rappel celui ci est de :

$$R0 = \frac{\beta \cdot N}{\mu \cdot \alpha \cdot r}$$

Représenté graphiquement, avec N = 100:



On s'attend donc à ce que les souches qui ont une charge parasitaire entre 7.5 et 8.5 forment plus d'infections secondaires.

Si on ne spécifie pas d'argument effect_arg, la charge parasitaire est choisie au hasard, sinon elle vaudra la valeur d'effect_arg. C'est utilisé notamment dans le mode theory_tester, pour lancer la simulation avec des valeurs d'effect croissantes, afin de vérifier si le nombre d'infection secondaires maximum correspond à celui attendu selon la théorie.

Enfin, si on lui donne un parasite en argument, elle calcule la charge parasitaire à partir de la virulence, en utilisant l'inverse de la fonction décrite ci-dessus et elle lui ajoute un nombre au hasard entre -2 et 2 afin de permettre une évolution des paramètre lors des évènement de mutations. C'est utilisé dans le mode 'war' ou 'all_night_long' si les trade_off sont définis comme 'leo'.

theory-tester.py and theory-tester2.py

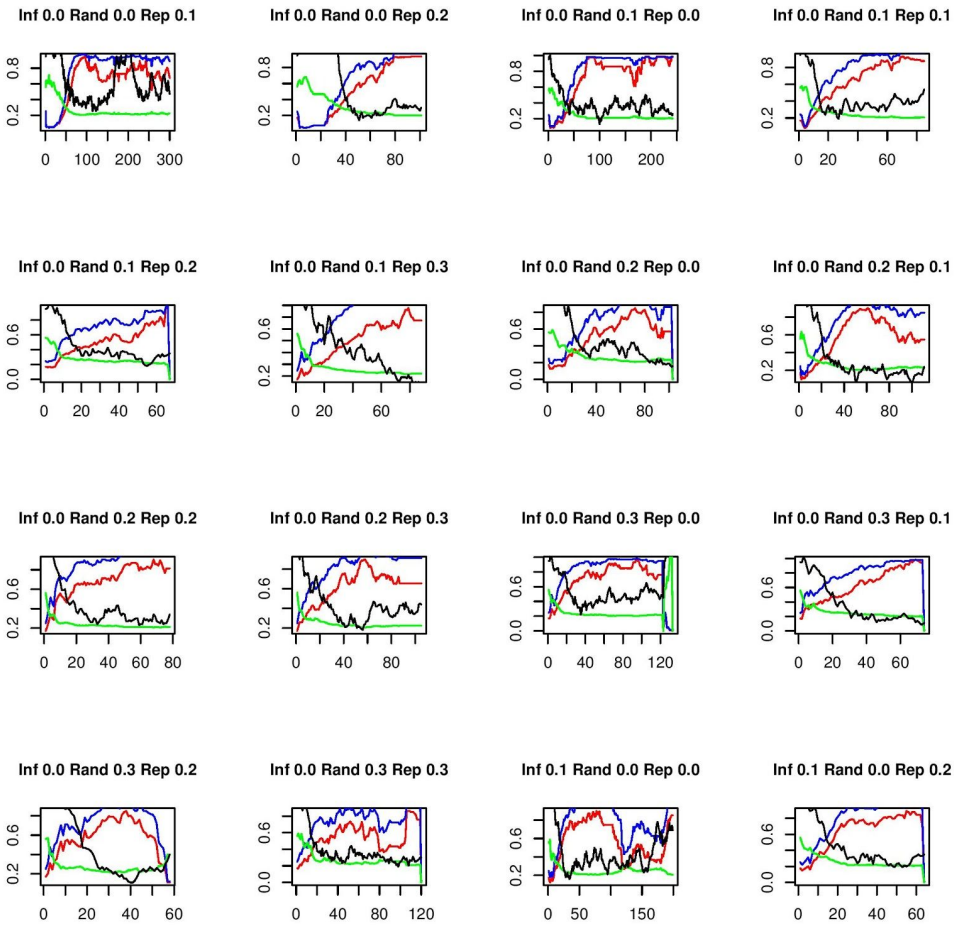
ces deux fonctions lancent le programme en boucle en variant certains paramètres, afin de pouvoir récolter des données

theorytest.py :

le tester reçoit une liste de paramètres à modifier et essaie toutes les combinaisons possibles, à l'aide de la fonction increment_difficulty.

il lance alors launch() qui modifie certaines variables de CHANGING_CONST.py avant de lancer l'application principale. Cette dernière reçoit un nom de fichier csv en argument, dépendant des paramètres entrés. Malheureusement, nous avons dû passer par une méthode 'pas très propre' afin de lancer le programme : os.system("python main.py "+str(filename)) car des conflits de variables gardées en mémoire ne nous permettaient pas de (facilement) lancer deux instances de kivy dans le même programme.

Le but de theory-tester était principalement de proposer un script simple permettant de tester plusieurs paramètres. Des tests simples ont été fait (pour montrer que ça marche) en étudiant différentes valeurs de mutations. On a pu ainsi représenter graphiquement l'évolution des valeurs de virulance/transmission/recovery moyennes (et le nombre de parasites) à l'aide d'un simple script R (ici : en rouge la virulence moyenne, en bleu la transmission moyenne, en vert la recovery moyenne et en noir le nombre de parasites. La simulation est arrêtée à 300 secondes où dès que la population de parasites tombe à 0).



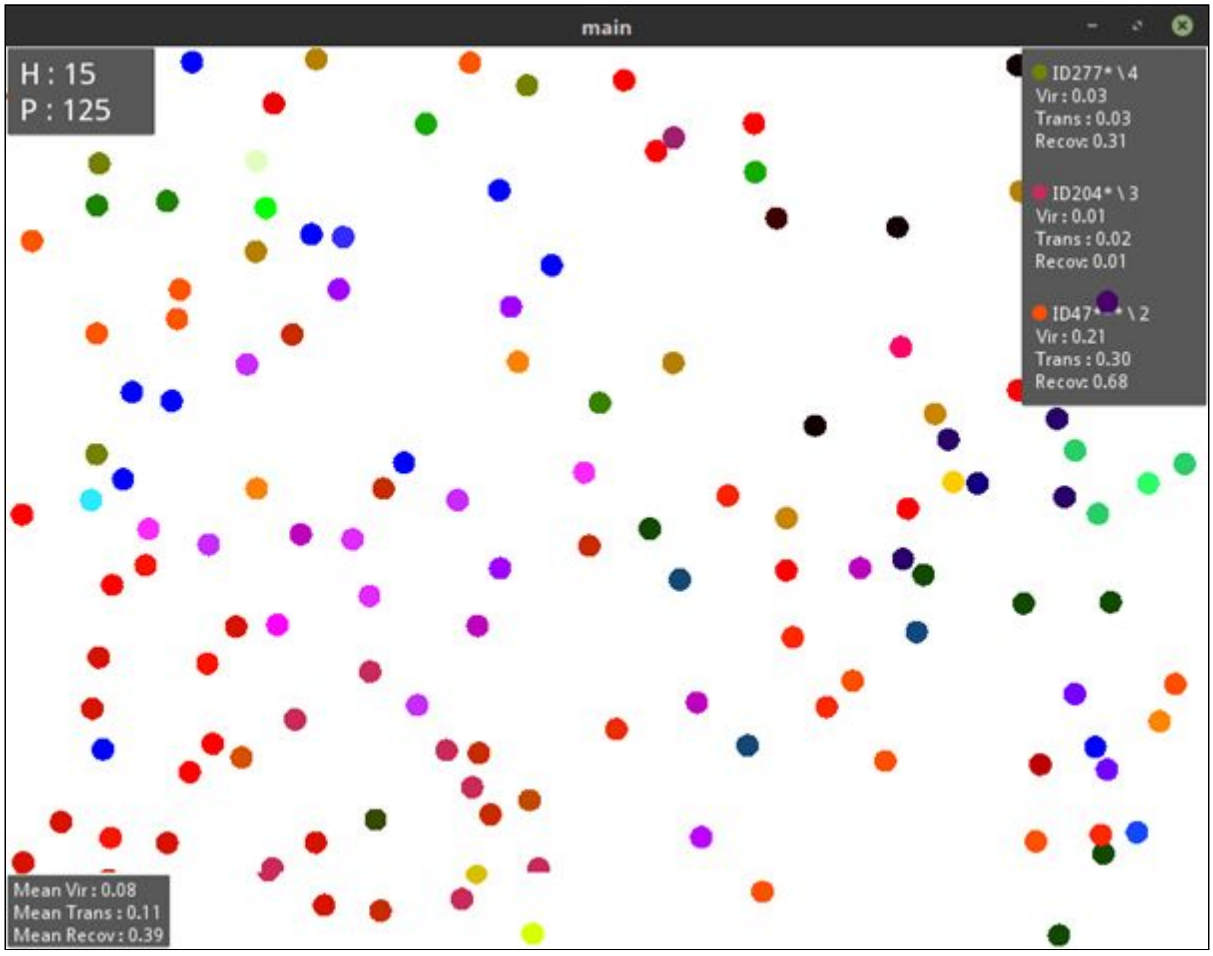
Theory_tester2.py

Theory-tester2 avait comme but de tester non pas des paramètres 'environnementaux' mais plutôt différentes souches de parasites dans un même "environnement". Le principe est le même, on lance le programme en boucle mais cette fois en lui passant en argument une variable définissant la

charge parasitaire d'une souche de départ. On utilise donc les trades_off == 'leo', pour lesquels on peut spécifier une valeur de charge parasitaire en argument (effect_arg). L'objectif étant de mesurer l'efficacité d'une souche dans les conditions idéales, on met une seule souche de parasites dans chaque simulation et on observe le nombre d'infections secondaire au cours du temps.Pour simplifier la programmation on ajoute un seul individu et pour éviter qu'il disparaisse immédiatement, on le fait se reproduire dix fois. Là-aussi, il nous est facile de représenter graphiquement les résultats, c'est-à-dire le nombre de parasites après un temps donné pour chaque simulation.

Résultats

Fenêtre de l'application:



Légende :
en haut à gauche : le nombre d'individus sains (H) et de parasites (P)

en haut à droite : les 3 souches de parasites avec le plus d'individus en vie (mode war et all night long) avec leur valeur de virulence(Vir), transmission(Trans) et probabilité de guérison(Recov)

en bas à gauche : valeurs moyennes de virulence(Mean Vir), transmission (Mean Trans) et de guérison (Mean Recov)

De plus il est possible de mettre en pause la simulation et de cliquer sur la fenêtre pour obtenir un graphique du nombre d'infection secondaires en fonction de la virulence.

Fichiers csv obtenus :

mode theory tester :

Les fichiers des simulations sont dans le dossier data_effect_tester. La fonction crée un fichier par simulation.

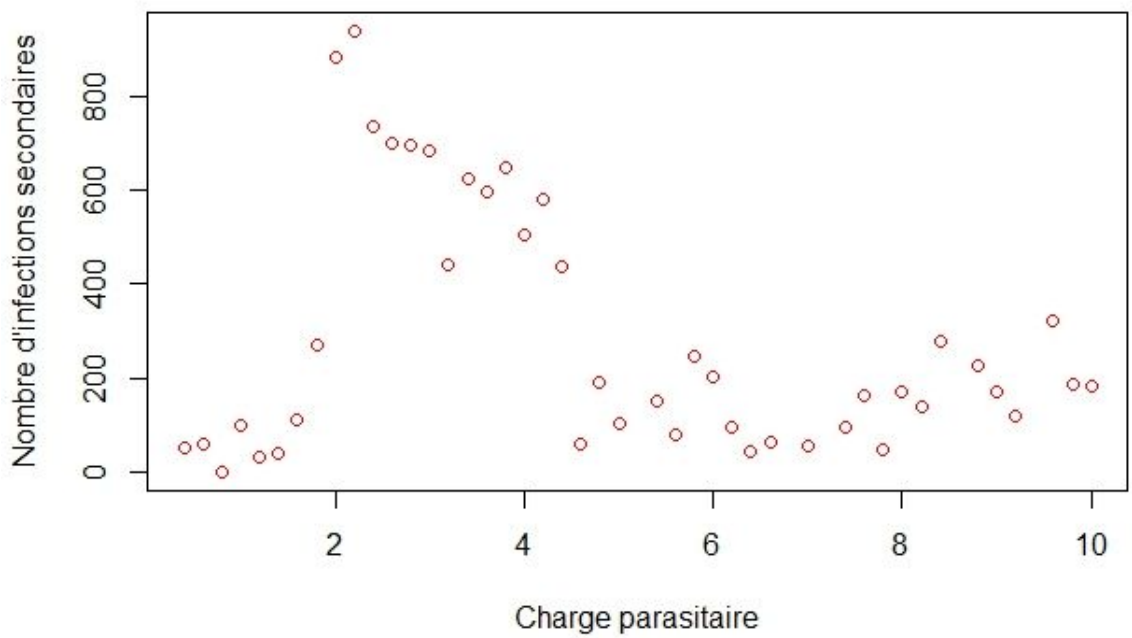
mode war et all night long :

- nb_total_sains_infectes.csv
- data/souches-###.csv

Interprétation des résultats :

Mode "theory_tester"

theory_tester2:
Comme décrit dans le chapitre theory tester ci-dessus, ce mode permet de voir si certaine valeurs de charge parasitaire mène à un plus grand nombre d'infections secondaires et si notre modèle obtient les résultats attendus en théorie dans les conditions les plus simples,avec un parasite à la fois.
Voici les résultats obtenus, représentés graphiquement. On peut voir que les paramètres calculés à partir d'une charge parasitaire entre 2 et 3 ont tendance à avoir un nombre d'infection secondaires plus élevées.



Cependant, ce n'est pas ce que nous attendions en théorie. Pour rappel, les niveaux de charge parasitaires qui maximisent le nombre de transmission secondaire théorique se situaient entre 7 et 8.
Les raisons qui conduisent à ces résultats ne sont pas clairs. Nous avons cependant remarqué que la taille de et surtout la densité de la population ont une influence majeure. En effet, on arrive très rapidement à des population presque entièrement constituées de parasites. Le nombre de nouvelles infections secondaires est alors limité par le nombre de guérisons, ce qui pourrait déplacer l'optimal vers la gauche, là où les valeurs de taux de guérison sont plus élevées.

Mode "war"

Nous n'avions pas vraiment de théorie à tester avec le mode war. Nous avons simplement observé un grand nombre de simulation en faisant varier quelques paramètres.

Devant le trop grande nombre de combinaisons variations possible, nous allons seulement détailler l'influence des attribut des parasites, et ce dans plusieurs conditions : à savoir la virulence, le taux de transmission et le taux de guérison des parasites. Nous allons brièvement énumérer comment chacun des paramètre peut changer des aspects des simulations, en essayant d'expliquer pourquoi certaines valeurs sont finalement préférées à d'autres :

- la taille de la fenêtre : ce paramètre est modifié dans les premières lignes de main.py. Une plus grande fenêtre mène généralement à un nombre de collisions réduits et donc à des équilibres avec des parasites peu virulents car vulnérables. Lorsque la fenêtre est très petite, les contacts sont au contraire très nombreux et

donc une forte sélection sur la virulence prend place. Nous avons préféré une taille de ~1000x1000 pixels, où une virulence supérieure à 0.7 est rare. Il faut bien comprendre que ce paramètre est étroitement corrélé à la taille des boules. En effet, on pourrait compenser une taille de fenêtre plus grande en augmentant la taille des individus (définie dans main.kv).

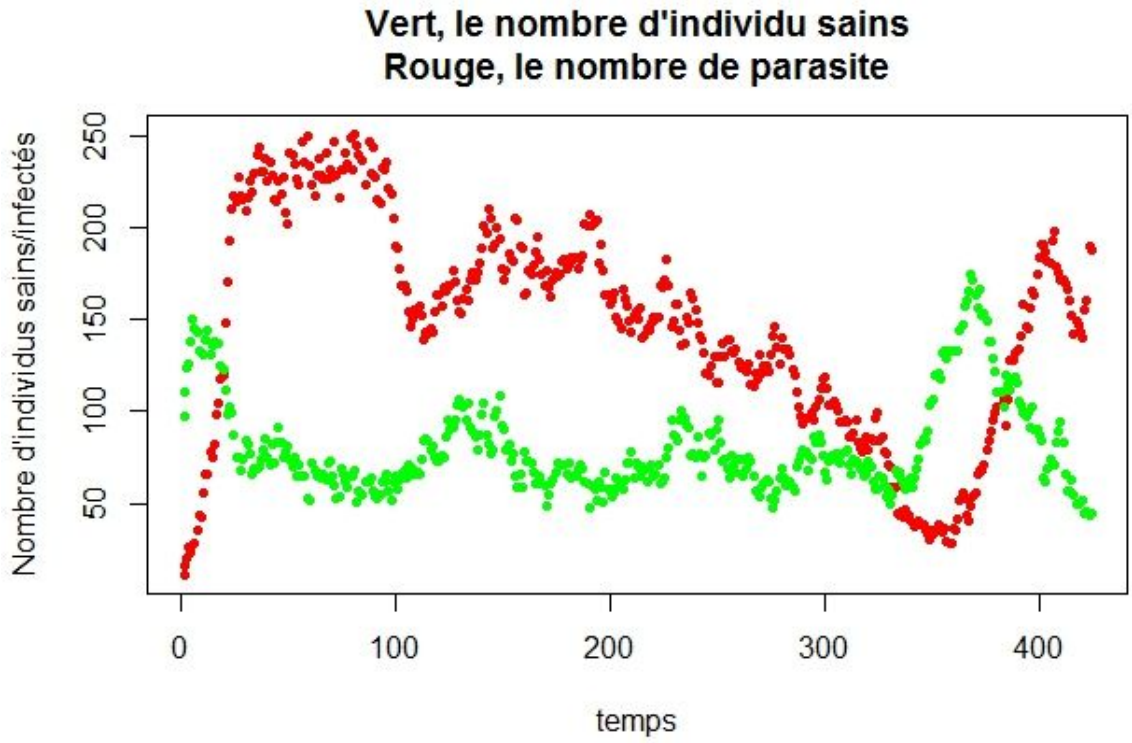
- La probabilité de se reproduire et de mourir (de base) sont étroitement corrélées avec la vitesse de base (tout en gardant un rapport entre ces deux attributs de ~1). En effet, comme les fonctions qui mettent à jour la santé des individus sont appelées chaque seconde, il faut s'assurer que les boules soient suffisamment rapides sinon nous risquons de ne voir aucune interaction (collision). Les valeurs que nous avons essayées nous ont menées à un peu plus de 10% de chance (par seconde), en ayant une probabilité de reproduction légèrement supérieure à la probabilité de décès - afin de compenser pour la virulence des parasites.
- la transmission des résistances est un facteur important permettant la différentiation entre des modèles avec compétition entre souches et les modèles à souche unique. Une résistance élevée instaure une pression sélective (négative) élevée en forçant les parasites à muter souvent afin de contrer ces résistances. Ceci s'observe effectivement, lorsqu'une souche ne reste pas longtemps dans le top 3. De façon similaire, la résistance après recovery est aussi déterminée comme 100%, afin de s'assurer qu'un individu guéri crée effectivement une résistance à son ancien parasite.
- le taux d'infection est un paramètre déterminant dans les simulations. Il ne peut dépasser 0.5 (cela signifierait qu'un parasite peut dépasser 100% de chance de transmission) et le mettre trop bas force les parasites à être sélectionnés pour leur taux de transmission - ce que nous observons. Un taux de base de 0.4 signifie une amélioration possible de 40% (en absolu) et s'avère être un bon compromis.
- les probabilités de mutations sont la source de bruit la plus importante. Il nous a semblé indispensable de mettre une chance de mutation spontanée (on nothing) très basse, car non seulement elle est rarement observée (cela signifie concrètement qu'un parasite mute à l'intérieur d'un hôte et que la souche mutée finit par dominer l'hôte), mais en plus de ça elle confère généralement un désavantage au parasite - car aucun individu nouveau n'a été contaminé (alors que la compétition inter-parasites augmente). Les deux autres valeurs de mutations peuvent être légèrement plus élevées afin d'observer des évolutions progressives des souches. Des valeurs de probabilité de mutation élevées amènent généralement beaucoup de variation dans les virulences/transmissions/recovery moyennes et de ce fait produisent des systèmes plus instables.
- La chance de guérison de base influence étroitement la sélection sur la recovery. Afin de rester dans des taux de guérison raisonnables, nous avons considéré qu'un parasite a environ autant de chance de disparaître par la mort de son hôte que par sa guérison.
- Enfin, parazite_fight_chance influence grandement la virulence moyenne de la population. Nous avons observé qu'une valeur de 1 sélectionne fortement les parasites sur leur virulence, amenant des systèmes instables (car des populations fortement parasitées disparaissent subitement) alors que des valeurs basses permettent aux autres attributs de se développer. Il nous également semblé cohérent

- comme en réalité un hôte pourrait être contaminé par plusieurs parasites différents
- de garder cette valeur en dessous de 50%. Une valeur de parazite_fight_chance basse a un effet de sélection positif, où un très grand nombre de souches différentes peut être observé tout au long de la simulation. Là aussi, il nous a semblé peu cohérent d'observer un grand nombre de souches qui ne puissent pas faire de contaminations multiples.

De manière générale, le mode 'War' nous a permis de mieux définir les paramètres qui nous permettaient de lancer des simulations de longue durée et de comparer ainsi nos données avec les modèles théoriques de parasitologie. Pour cela, nous avons introduit le mode 'all_night_long' qui rajoute des parasites lorsque ceux-ci disparaissent (afin de contrebalancer notre modèle limité à quelques centaines d'individus) et pose également des seuils de populations, pour imiter les limitations de l'environnement.

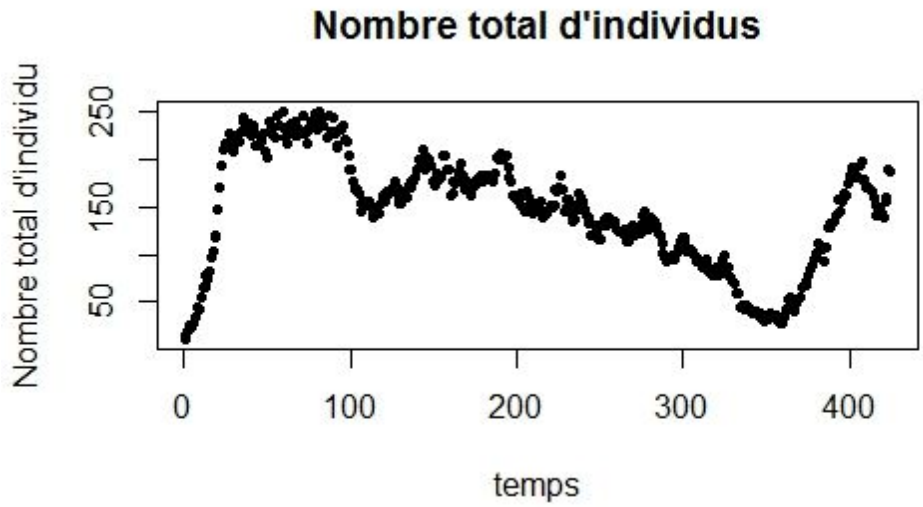
Mode "all_night_long"

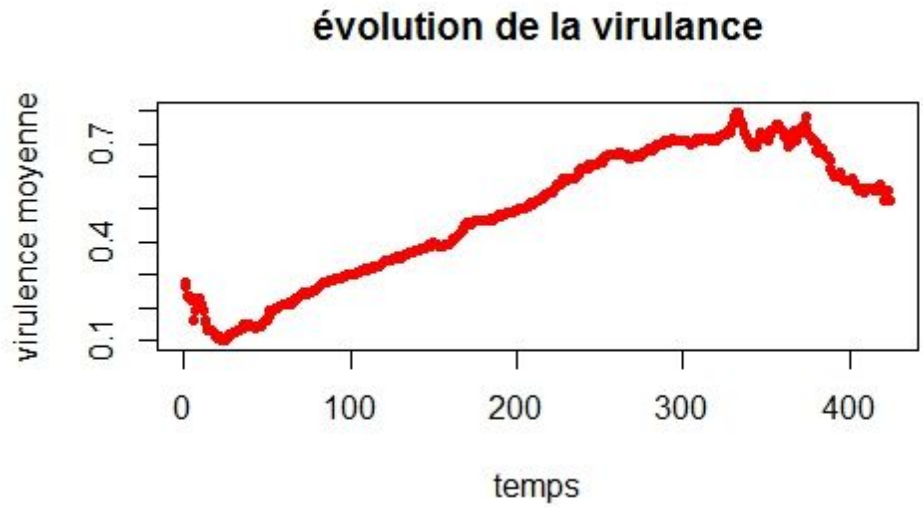
Vous trouverez dans le dossier nice_const_value les fichiers de constantes pour la simulation décrite dans ce chapitre sous le nom all-night-long, nous l'avons laissé tourner pendant 430 secondes. Voici représentée graphiquement l'évolution des tailles de populations saines et infectées.



Ce qu'on observe en premier lieu c'est que le nombre d'individu sain augmente dans un premier temps, puis que les parasites se propagent très rapidement. On voit ensuite le nombre de parasite diminuer jusqu'à redevenir plus petit que le nombre de Healthy. À ce moment les healthy peuvent à nouveau proliférer et à nouveau, lorsqu'ils atteignent un nombre d'environ 150, les parasites prolifèrent.

On peut à présent s'intéresser à l'évolution de la virulence et de la taille totale de la population au cours du temps.





On voit d'abord que la courbe représentant l'évolution de la taille de la population totale ressemble énormément à celle de l'évolution de la taille de la population de parasites.

Ensuite, on observe que population augmente énormément au début puis elle diminue au fur et à mesure alors que la virulence moyenne des parasites augmente. On voit que lorsque la virulence atteint 0.7, la population a diminué de deux tiers par rapport à son niveau maximum.

La corrélation inverse entre taille de la population et virulence moyenne est forte. Notre hypothèse est que lorsque la population est grande, il est plus avantageux pour les parasites d'avoir une forte virulence et un forte taux de transmission, car les contacts occasions de transmission sont plus nombreuses. De plus, la constantes parasite_fight_chance est ici

définie comme 0.3. Les parasites plus virulents ont donc plus d'hôtes potentiels et remplacent donc les parasites les moins virulents.

Discussion

Points fort :

Si on se réfère à l'introduction : "L'idée de départ du projet était de simuler de manière intuitive et dynamique une population animale ayant des interactions et dans laquelle se propagent des parasites." L'objectif est clairement rempli et nous avons réussi à réaliser le projet que nous voulions.

Même si cela ne faisait pas partie de la matière vue en cours, ce projet nous a permis de découvrir la programmation orientée objet ainsi que la création d'une interface potentiellement agréable à utiliser pour l'utilisateur.

Il est difficile de travailler à plusieurs sur le même programme mais nous avons pu surmonter ce problème en nous familiarisant avec le système git qui permettait de synchroniser nos travaux régulièrement.

Points faibles :

Jusque là les graphiques et les données récoltées en mode "theory tester" ne sont pas totalement conformes à ce que prévoit la théorie du cours de parasitologie.

Les données issues des mode war et all_night_long ne donnent pas forcément des données utilisables "scientifiquement", dans le sens ou on ne teste pas une hypothèse précise. Cependant, elles nous permettent d'observer des tendances que nous pourrions par la suite essayer de tester de en respectant la démarche scientifique. De plus, peut-être qu'avec des connaissances supplémentaire en parasitologie nous pourrions mieux analyser et tirer quelque chose d'utile des données apportées l'historique de chaque souche et de l'évolution de la population avec la complexité des mutations et des résistances.

Un autre point important posant problème pour une utilisation plus scientifique du modèle, malgré le seed appelé au début de la fonction nous n'obtenons pas de résultats reproductibles. Nous avons une hypothèse à ce sujet qui est développée dans la conclusion.

Il a été difficile au début de sélectionner la bonne librairie pour l'interface graphique. 2 candidats potentiels étaient retenus : pygame et pivy. En nous répartissant le travail, des essais avec l'interface pygame et pivy ont été faits simultanément. Pygame a donné de bons résultats pour un nombre de boules pas trop importants, mais lorsque l'algorithme du fichier quadtree a été développé l'interface Kivy s'est trouvée nettement plus efficace pour gérer un grand nombre d'individus à la fois, c'est donc celle ci qui a été retenue et une partie du travail fait pour pygame a dû être adapté à pivy. Nous avons donc perdu du temps à ce moment car des fonctions ont été codées à double.

Leçons retenues en vue d'un prochain projet :

Ce type de projet, surtout en groupe, prend énormément de temps et nous avons passé plusieurs nuit à travailler dessus. Il est difficile de se coordonner même si git nous a beaucoup simplifié la vie par la suite. Il aurait probablement fallu définir mieux notre objectif au début. Une fois l'interface en place nous étions vraiment enthousiastes d'avoir obtenu quelque chose nous avons essayé plein de fonctions et de conditions différentes et nous nous sommes peut-être un peu trop dispersés.

Conclusion

Travail réalisé :

Nous avons été peut-être un peu trop ambitieux au niveau du développement du programme. Nous sommes réellement fiers de l'outil que nous avons pu créer et nous avons eu beaucoup d'intérêt à le développer et à l'améliorer régulièrement du mieux que nous pouvions mais nous avons sous-estimé le temps nécessaire au traitement des données et à la rédaction du rapport ce qui fait que nous n'avons pas vraiment pu profiter du potentiel que nous offre ce programme. Nous aurions peut-être dû faire quelque chose de plus simple et passer plus de temps à traiter et interpréter les données que nous aurions obtenus.

Développements futurs / améliorations :

Comme expliqué précédemment dans les points faibles, nous avons un problème de reproductibilité des données alors qu'avec le seed nous devrions avoir toujours la même trajectoire pour chaque boule à chaque simulation. Nous pensons que de par la complexité du système ce phénomène peut s'expliquer par une application de la théorie du chaos : un changement infime peut mener à de grands bouleversements. Dans notre cas, un très grand nombre de collisions est généré (le système est très complexe). Les positions de toutes les boules sont calculées 60x/seconde mais un changement infime dans la vitesse du processeur peut décaler ces delta t et donc les collisions peuvent être (très légèrement) différentes à chaque simulation. Ces petits détails (angles) mènent à des systèmes complètement différents quelques milliers de collisions plus tard. Une façon de vérifier cette hypothèse est de lancer la simulation avec très peu d'individus (3 à 5) et on se rend compte que le système est absolument reproductible jusqu'à intervalle de temps plus grand. Il aurait été intéressant de vérifier notre hypothèse en regardant si un petit nombre d'individus a effectivement toujours la même trajectoire dans chaque simulation (en dessinant sa trajectoire par exemple), et si c'est le cas, à partir de quel produit de nombre d'individus*intervalle de simulation on commence à voir apparaître ce phénomène de non-reproductibilité des données. Malheureusement nous n'avons pas eu le temps de le faire.

Le programme a été terminé proche du délai et nous n'avons pas pu faire beaucoup de simulations. Il aurait notamment été intéressant, en mode theory_tester, d'avoir plusieurs mesures pour chaque valeur de charge parasitaire et de pouvoir comparer des moyennes plutôt que des résultats uniques. De manière générale, pour aller plus loin dans l'analyse des résultats il faudrait continuer à faire plus de simulations avec d'autres paramètres.

Les simulations prennent beaucoup de puissance de calcul. Il aurait pu être intéressant d'ajouter des librairies nous permettant de pouvoir faire du calcul partagé entre nos ordinateurs. Comme par exemple celles qu'on peut trouver ici : https://wiki.python.org/moin/ParallelProcessing

On aurait pu ajouter une fonction pour que deux parasites puissent muter ensemble pour former un "super-parasite" qui aurait par exemple 2 fois la fitness max des parasites.

En modifiant la forme et la taille de la fenêtre on obtient des résultats différent, il aurait pu être préférable de partir sur une fenêtre ronde car on remarque que les boules ont tendance à s'agglutiner dans le coins ce qui fait que la rencontre des individus n'est pas vraiment aléatoire puisqu'elle est différente à chaque endroit de la fenêtre.

Références

Jacob Koella, documents du cours intitulé *Parasitologie générale*, semestre de printemps 2016, Université de Neuchâtel.

Steven Labert, Use Quadtrees to detect collisions in 2D space, https://gamedevelopment.tutsplus.com/tutorials/quick-tip-use-quadtrees-to-detect-likely-collisions-in-2d-space--gamedev-374