Documentaion GenoWALT

Meije Gawinowski

27 août 2018

Table des matières

1 Fonctionnement général de GenoWALT

La version opérationnelle de GenoWALT (V8 sur GitHub) est bi-allélique mais il existe une version plus basique qui gère le multi-allélisme (Basic_Multiallelic sur GitHub). La gestion du multiallélisme sera détaillée dans une partie dédiée.

Le module GenoWALT est constitué de 5 classes dirigées par un fichier *main* pour l'exécution. D'autres scripts ont été élaborés, notamment pour automatiser la création de fichiers d'entrée. GenoWALT adopte la logique suivante :

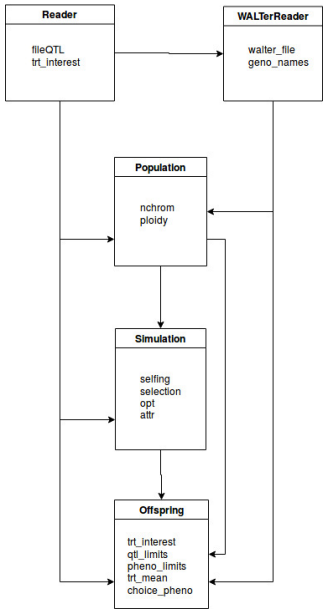


Figure 1 – Diagramme de classes

La logique du modèle est celle-ci :

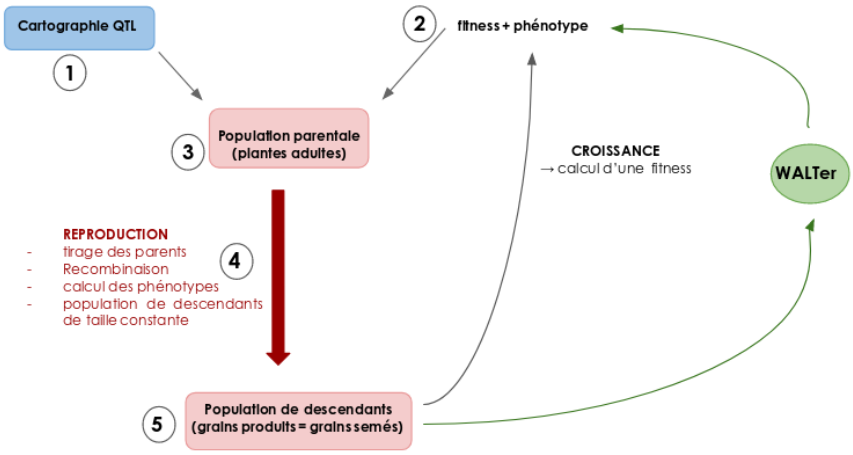


Figure 2 – Schéma du fonctionnement général de GenoWALT

2 Classe Reader

Cette classe permet de lire les informations génétiques des QTL directement depuis un fichier.

2.1 Format des informations QTL

Les données QTL sont organisées dans un fichier .csv sous cette forme pour la version biallélique :

n° QTL	type	chromosome	position	valeur d'effet	nom	dominance
0	['PH']	3	12.6	2.4	qPh1	complete
1	['RS']	5	35.9	-0.1	qRs1	partial
2	['LS']	5	35.9	0.2	QLs1	recessive

Figure 3 – Table qui regroupe les données QTL

Dans cet exemple on a 3 QTL, qui associés aux trait de la hauteur (PH = Plant Height), la surface racinaire (RS = Root Surface) et la surface foliaire (LS = Leaf Surface). Pour chaque QTL on a l'information du chromosome, de la position, la valeur d'effet (dans la même unité que celle du trait), son nom et le régime de dominance. On ne prend en compte cette information de dominance que dans les cas de QTL bi-alléliques (ou l'information est pour l'allèle 1).

2.2 Attributs

Ces attributs sont spécifiés par l'utilisateur :

- *Reader.fileQTL* est le nom du fichier qui contient les informations génétiques des QTL dans le répertoire courant
- *Reader.list_trt* est la liste des traits d'intérêt étudiés (ceux dont on étudie la variabilité génétique)

2.3 Reader.all_indices(value, qlist)

Cette méthode permet de récupérer tous les indices de la valeur *value* dans la liste *qlist*.

2.4 Reader.dictGenerator(data)

L'argument *data* est la table de données QTL. Cette fonction renvoie une liste dictionnaire dont les clés sont les différents traits associés aux QTL de la table et les valeurs sont des listes contenant les chromosomes, les positions, les valeurs et les noms des QTL.

```
dictQTL = { 'PH' : [ chrom = [ ], pos = [ ], val = [ ], names = [ ] ],
            'RS' : [ chrom = [ ], pos = [ ], val = [ ], names = [ ] ],
            'LS' : [ chrom = [ ], pos = [ ], val = [ ], names = [ ] ] }
```

2.5 Reader.limits_definer(data)

L'argument *data* est la table de données QTL. Cette fonction calcule pour chaque trait d'intérêt les limites supérieure et inférieure de la somme des effets des QTL associés à ce trait. La structure de données de sortie est une liste dictionnaire qui a comme clé le nom du trait et comme valeur la liste avec les deux bornes :

qtl_limits = { **'PH'** : [0, 2.4], **'RS'** : [-0.1, 0], **'LS'** : [0,0.2] }

2.6 Reader.pleiotropyManagement(data)

L'argument *data* est la table de données QTL. Cette fonction permet de gérer les cas de pléïotropie, c'est-à-dire lorsque l'on a des QTL multi-traits. En effet il peut arriver qu'un même QTL (même chromosome et même position) soit associé à deux traits différents. Ici on part du principe que pour un même trait on ne peut pas avoir deux QTL sur la même position d'un chromosome (avec un fichier c'est à l'utilisateur de s'assurer sa validité sur ce point et avec une génération aléatoire on considère qu'il y a très peu de chances que ça arrive). Pour vérifier si on a un ou plusieurs cas de pléïotropie on vérifie donc si pour plusieurs traits différents on a des QTL sur un même chromosome et si c'est le cas on regarde s'ils sont situés sur la même position. S'il y a pléïotropie on prend la première occurrence chromosome/position, on change son nom pour fusionner les noms des QTLs concernés et dans la colonne type on ajoute les traits concernés à la liste. On fait également le même changement de nom dans la liste dictionnaire et on supprime les occurrences suivante dans la table QTL. L'information des valeurs de QTL n'est pas changée dans la table des QTL, on y accède avec la liste dictionnaire. Cette fonction renvoie alors la table et la liste dictionnaire des QTL actualisées.

Exemple de gestion de la pléïotropie :

dictQTL = { **'PH'** : **[** chrom = [3], pos = [12.6], val = [2.4], names = ['qPh1'] **]**,
'RS' : **[** chrom = [3,5], pos = [12.6,35.9], val = [-0.1,0.2], names = ['qRs1','qRs2'] **]** }

n° QTL	type	chromosome	position	valeur d'effet	nom	dominance
0	'PH'	3	12.6	2.4	qPh1	complete
1	'RS'	3	12.6	-0.1	qRs1	partial
2	'RS'	5	35.9	0.2	qRs2	recessive

Chromosome et position identiques

Gestion de la pléïotropie

n° QTL	type	chrom	pos	valeur d'effet	nom	dominance
0	['PH','RS']	3	12.6	[2.4,-0.1]	qPh1,qRs1	[complete,partial]
1	['RS']	5	35.9	[0.2]	qRs2	[recessive]

dictQTL = { **'PH'** : **[** chrom = [3], pos = [12.6], val = [2.4], names = ['qPh1qRs1'] **]**,
'RS' : **[** chrom = [3,5], pos = [12.6,35.9], val = [-0.1,0.2], names = ['qPh1qRs1','qRs2'] **]** }

2.7 Reader.dataQTL()

Cette fonction permet de générer une table valide avec toutes les informations sur les QTL. Il s'agit de lire le fichier de données QTL indiqué en attribut *Reader.fileQTL* puis d'appliquer à cette table la gestion de la pléïotropie pour obtenir une table de QTL opérationnelle.

3 Classe WALTerReader

Cette classe permet de lire les informations de population de type WALTer.

3.1 Format des données WALTer

GenoWALT prend deux types de données issues de WALTer. Il prend d'abord un fichier *population_data.csv* qui donne les informations de fitness, de génotype et des traits d'intérêt des différents individus :

Individu	fitness	Genotype	PH	RS
0	0.5	"Geno_1"	140	50
1	0.5	"Geno_2"	80	25

Figure 4 – Table regroupant les données de population

WALTer gère les génotypes sous forme de noms mais c'est à l'utilisateur de définir à quoi correspond un nom de génotype. On a alors un fichier de génotype *geno_conversion.csv* associé pour faire le lien entre les noms de génotypes gérés par WALTer et les véritables données génotypiques dont a besoin GenoWALT :

"Geno_1"	{ qtl0 : [1,1], qtl1 : [0,1] }
"Geno_2"	{qtl0 : [1,0], qtl1 : [0,0] }

Figure 5 – Table de correspondance entre les noms de génotypes et les codages alléliques

Le format final des données génotypiques est une liste dictionnaire comme ceci :

dictGeno = { **'ind 0'** : { 'qtl 0' : [0,1], 'qtl 1':[1,1] } , **'Ind 1'** : { 'qtl0' : [0,0], 'qtl 1' : [1,0] } }

Dans cet exemple on a deux individus (numérotés 0 et 1) et leurs valeurs de fitness regroupés dans le fichier de sortie de WALTer. Il est possible d'intégrer dans ce fichiers toutes les valeurs de traits physiologiques que l'on veut pour ensuite calculer la fitness.

3.2 Attributs

Ces attributs sont définis par l'utilisateur :

- *WALterReader.walter_file* est le fichier avec la fitness et les noms des génotypes de la population
- *WALterReader.geno_names* est le fichier qui fait correspondre les noms des génotypes à des doublets d'allèles

3.3 WALTerReader.WALTer2GenoWALT

Cette fonction permet de traiter le fichier issu de WALTer et le fichier de conversion des noms de génotypes pour obtenir une liste dictionnaire globale des génotypes de tous les individus. Pour chaque individu on initialise une liste dictionnaire (qui contiendra son génotype) puis on regarde le nom du génotype et on récupère la liste dictionnaire associée à ce nom et on met à jour la liste dictionnaire du génotype de l'individu. On obtient finalement une liste dictionnaire de ce type :

dictGeno = { **'ind 0'** : { 'qtl 0' : [1,1], 'qtl 1':[1,0] } , **'ind 1'** : { 'qtl 0' : [1,0], 'qtl 1' : [0,0] } }

Cette fonction renvoie également le vecteur de fitness des individus.

3.4 WALTerReader.setConversion(data)

Cette fonction permet d'établir une table de conversion entre les différentes valeurs de QTL et les combinaisons d'allèles. Quand un QTL a plus de deux allèles on ne prend pas en compte l'information de dominance. Comme on considère une ploïdie égale à 2 on a des combinaisons de 2 allèles. On considère deux allèles 0 (le QTL n'a pas d'effet) et 1 (le QTL a un effet de valeur v). On considère trois régime de dominance :

- "complete"
  $(0,0) \to 0$
  $(0,1) \to v$
  $(1,0) \to v$
  $(1,1) \to v$

- "partial"
  $(0,0) \to 0$
  $(0,1) \to \frac{v}{2}$
  $(1,0) \to \frac{v}{2}$
  $(1,1) \to v$

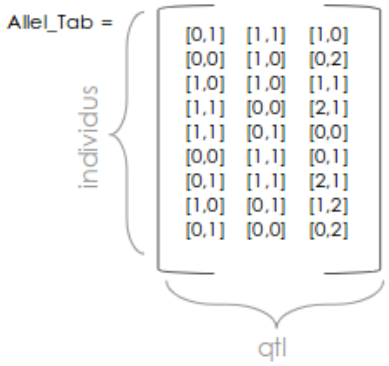
- "recessive"
  $(0,0) \to 0$
  $(0,1) \to 0$

$(1,0) \rightarrow 0$
$(1,1) \rightarrow v$

3.5 WALTerReader.getAllelTab(listGeno)

L'argument *listGeno* correspond à la liste dictionnaire contenant le génotype de la population. Cette fonction manipule la liste et la ré-organise en une table dans laquelle chaque ligne correspond à un individu et chaque colonne à un qtl. Un élément $[i, j]$ de cette table correspond donc à la combinaison d'allèles (al, al_2) de l'individu i au qtl j.

Exemple :



3.6 WALTerReader.dataWALTer(data)

L'argument *data* est la table de QTL sortie par la classe Reader. Cette fonction permet d'organiser les différents résultats de la classe en un seul objet. Cette fonction retourne donc la liste dictionnaire du génotype de la population, la liste dictionnaire de conversion, le tableau avec les valeurs de QTL pour chaque individu et le vecteur de fitness des individus.

4 Classe Population

A partir des informations sur les QTL, le génotype et le phénotype cette classe crée un objet simuPOP Population().

4.1 Attributs

Les attributs sont définis par l'utilisateur :

- *Population.nchrom* est le nombre de chromosome, par défaut fixé à 21
- *Population.ploidy* est la ploïdie, par défaut fixée à 2
- *Population.trt_interest* est la liste de traits d'intérêt, par défaut ne contient que la hauteur. Ce champ est hérité de la classe *Reader*.

4.2 Population.InfoPop(pop,header,datPop)

L'argument *pop* est la population initialisée (les champs d'informations InfoFields ne sont pas encore remplis), *header* est la liste de traits de la table *data_population* et *datPop* est la table *data_population*. Cette fonction permet de rentrer toutes les valeurs des traits physiologiques du fichier de phénotype dans les champs d'informations InfoFields de l'objet Population. On suppose que l'information de fitness est déjà présente dans le fichier *data_population*.

4.3 Population.Creation(tab_qtl,genoPop,fitness_list,header,datPop)

L'argument *tab_qtl* est la table des QTL, *genoPop* est le tableau d'allèles contenant le génotype de la population, *list_fitness* est la liste des valeurs de fitness des individus, *header* est la liste de traits de la table *data_population* et *datPop* est la table *data_population* .

On initialise la fonction avec la fonction Population() du package simuPOP avec le nombre d'individus, le nombre de chromosomes, la ploïdie et les champs d'informations InfoFields. Les champs d'information sont une fonctionnalité très utile de simuPOP, il s'agit de caractéristiques comme le sexe, l'âge ou un trait physiologique.

On récupère également les données QTL (positions, chromosomes, noms des QTL) pour les implémenter dans le génome de la population avec la fonction *addLoci* de simuPOP. On remplit ensuite les champs d'information InfoFields avec les données de *data_population* pour les différents traits physiologiques puis on ajoute le vecteur de fitness dans ces infoFieldss. Avec la fonction lociByNames on peut ensuite récupérer les indices globaux des QTL dans le génome de l'objet Population.

On initialise ensuite le génotype de la population grâce à la table d'allèles créée avec la fonction *WALTerReader.getAllelTab*. Pour chaque individu *ind* et chaque qtl *q* on récupère les deux allèles correspondants dans la table d'allèle, c'est l'objet *allel*, une liste avec deux valeurs dedans et on utilise la fonction *ind.setAllele(val_allel,locus,ch)* avec *val_allel* la valeur de l'allèle, *locus* l'indice global du QTL et *ch* la copie homologue du chromosome (0 ou 1).

L'objet Population se présente alors sous cette forme (exemple avec 10 individus, 8 QTL pour un trait (PH) tous bi-alléliques) :

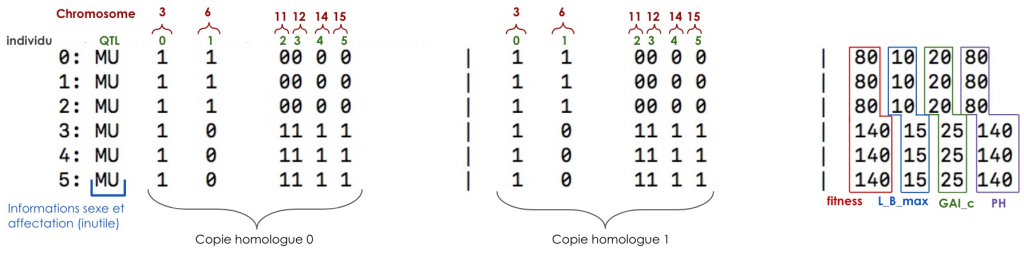


Figure 6 – Capture d'écran annotée de l'objet population parentale

5 Classe Simulation

Cette classe consiste à faire une étape de reproduction pour passer de la population de parents à une population de descendants. Deux méthodes de reproduction on été implémentées, une qui se base sur des valeurs de fitness pour échantillonner les parents pour la reproduction et une autre qui "force" le nombre de descendants par parent.

5.1 Attributs

- *Simulation.selfing* est le taux d'auto-fécondation.
- *Simulation.selection* est le trait sur lequel on fait la sélection pendant la reproduction (donc normalement la fitness).
- *Simulation.opt* est l'option de calcul des taux de recombinaison, selon Haldane ou Kosambi.
- *Simulation.attr* est la façon dont on calcule le nombre de descendants par parent à partir des proportions, soit par tirage, soit par simple multiplication par l'effectif total de la population de descendants.

5.2 Fonctions Ops

Les fonctions initOps, preOps, postOps et finalOps servent à préciser des étapes à appliquer entre les différentes générations. C'est déjà assez peu utile pour nous car on ne fait qu'un seul saut de génération mais en plus on ne considère pas de mutation ni de migration donc on ne se sert pas de ces options.

5.3 Simulation.create_Mating(parpop)

Cette fonction permet de créer un objet de reproduction Mating simuPOP. Pour cela on utilise un *HeteroMating* car on a à la fois de l'allogamie et de l'hétérogamie dans une population de blé. On utilise donc les objets *SelfMating* et *HermaphroditicMating*. Pour chacun de ces objet on précise dans l'argument *ops* les taggers, l'objet de recombinaison, le poids (proportion de chaque régime de reproduction dans la population) et le trait sur lequel on fait la sélection (la fitness). Les taggers permettent de garder certaines informations : ici on utilise pour récupérer les indices du ou des parents de chaque parent (ParentsTagger) mais aussi pour garder l'information du mode de reproduction. Pour ça on utilise un PyTagger défini à partir de alloTagger pour un régime d'allogamie et autoTagger pour un régime d'autogamie. L'objet de recombinaison *Recombinator* prend comme arguments les taux de recombinaisons et les loci sur lesquels il doit les appliquer. Les taux sont calculés avec la fonction *Simulation.calcRecombinaison*.

5.4 Simulation.alloTagger(mode)

Cette fonction va attribuer au champs d'information *mode* la valeur 1.

5.5 Simulation.autoTagger(mode)

Cette fonction va attribuer au champs d'information *mode* la valeur 0.

5.6 calcRecombination(parpop,list_loci)

Cette fonction calcule les taux de recombinaison pour chaque position de loci. On prend les locus i et on regarde le locus suivant $i+1$, s'ils sont sur le même chromosome le taux de recombinaison r est égal à :

- D'après Haldane $r_i = \frac{1}{2}\ \left(1 - e^{-2\ \Delta_i}\right)$, avec Δ_i la distance entre les loci $i+1$ et i
- D'après Kosambi $r_i = tanh(2\ \Delta_i)$, avec Δ_i la distance entre les loci $i+1$ et i

Si les deux loci sont sur des chromosomes différents, alors le taux r_i est égal à 0.5.

5.7 Simulation.create_FitnessSimulation

Cette fonction permet de faire une étape de reproduction à partir de valeurs des fitness.

Avant la reproduction on ajoute les champs d'information du mode de reproduction et de l'identité des parents pour pouvoir utiliser les taggers pendant la reproduction. On utilise ensuite la fonction *evolve* de simuPOP qui s'applique sur l'objet population de parents. Cette fonction prend comme arguments les options Ops (inutiles pour nous), l'objet de reproduction créé avec *Simulation.create_Mating* et le nombre de générations. A l'issue de cette fonction l'objet courant de population n'est plus celle des parent mais celle des descendants avec des champs d'informations initialisés à 0 pour ceux qui n'ont pas été suivis avec des taggers.

Les trois champs de départ PH, Offspring et fitness ont été réinitialisés à 0, le champ *mode* qui indique le régime de reproduction est à 0 pour tous les individus car on a choisi un taux d'autogamie de 1.0. On observe donc des -1 dans le champ *MotherID* pour tous les individus descendants car comme ils sont issus d'une auto-fécondation ils n'ont qu'un seul parent, par défaut considéré comme le père. On retrouve dans le champ *FatherID* les identifiants des parents (pères du coup) de tous les descendants.

5.8 Simulation.create_controlledSimulation

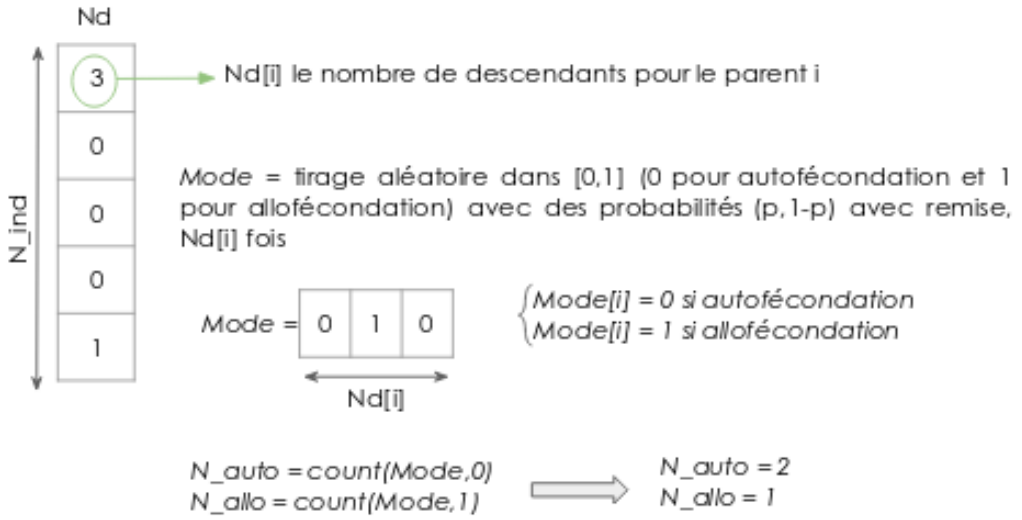
Avec les objets *Mating* de simuPOP utilisés dans la fonction *evolve* on prend en compte un champ de sélection, il s'agit d'un trait contenu dans les InfoFields qui (s'il ne l'est pas déjà) est ramené entre 0 et 1 (probabilité d'être tiré comme parent). On choisit en général la fitness comme champ de sélection. On réalise un tirage aléatoire pondéré par la fitness des parents, ce qui fait qu'on ne contrôle pas exactement le nombre de descendants. Or l'un des usages de GenoWALT sera peut-être de faire de la SSD (Single Seed Descent) c'est-à-dire d'avoir un descendant pour chaque individu parent. Or avec un tirage aléatoire, même pondéré, nous ne sommes pas sûrs d'obtenir ces résultats exacts à moins de "forcer" un nombre précis de descendants à partir de la fitness. Or ce n'est pas une fonctionnalité de simuPOP.

Le modèle WALTer donne en sortie soit un nombre de grains (et donc de descendants) soit directement une fitness pour chaque individu. Si on a un nombre de grain on calcule simplement la fitness d'un individu comme sa proportion dans la population.

On a ensuite deux choix pour évaluer le nombre de descendants par individus :

- Par tirage : soit N_D le nombre de descendants, on effectue un tirage aléatoire pondéré par la fitness de N_D parents puis on compte le nombre de descendants par parent.
- Par multiplication : on multiplie simplement la fitness d'un individu par la taille de la nouvelle population pour avoir le nombre de descendants.

On considère une population parentale avec N_{ind} individus numérotés de 0 à $N_{ind} - 1$ et un vecteur de descendants correspondant N_d. On considère également la probabilité p d'autofécondation rentrée en attribut dans la classe (attribut *selfing*). On considère chaque parent un à un et son nombre de descendants. Pour chaque parent i avec $N_d[i]$ descendants on réalise un tirage avec remise dans un vecteur $[0, 1]$ (0 pour autofécondation et 1 pour allofécondation) $N_d[i]$ fois. On compte ensuite les nombres d'autofécondation et allofécondation N_{allo} et N_{auto} :



On initialise la population de descendants avec la ploïdie, le nombre de chromosome et les champs d'information. On gère ensuite les cas d'autogamie et allogamie séparément.

Gestion de l'autogamie :
Pour un parent i on a tiré un nombre d'autogamie N_{auto}. On extrait alors le parent i de la population parentale pour former une population parentale intermédiaire avec ce seul parent. On effectue une reproduction en appliquant la fonction *evolve* sur cette population avec schéma de reproduction *SelfMating*. On forme alors une population intermédiaire avec N_auto descendants et on les ajoute à la population de descendants finale.

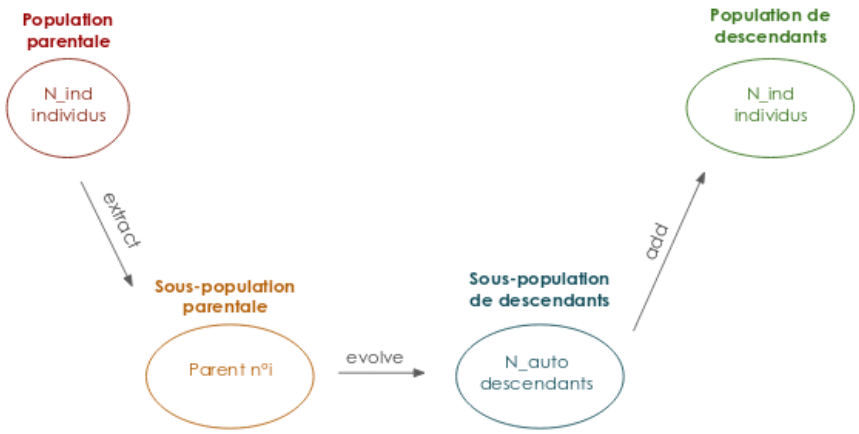


Schéma d'autogamie

Gestion de l'allogamie :

Pour un parent i on a tiré un nombre d'allogamie N_{allo}. Tant que ce nombre n'est pas nul :

1. On sélectionne un deuxième parent (père) avec un tirage pondéré par la fitness d'un individu parmi tous les individus de la population sauf le parent i, c'est l'individu Father
2. On forme une population parentale intermédiaire avec le parent i et le Father
3. Sur cette population on fait une étape de reproduction avec la fonction *evolve* avec un schéma de reproduction *HermaphroditicMating* (avec l'argument *alloSelfing=False*) cette population en une population de descendants intermédiaire avec un individu
4. On ajoute cet individu à la population de descendants finale
5. On décrémente N_{allo} d'une unité.

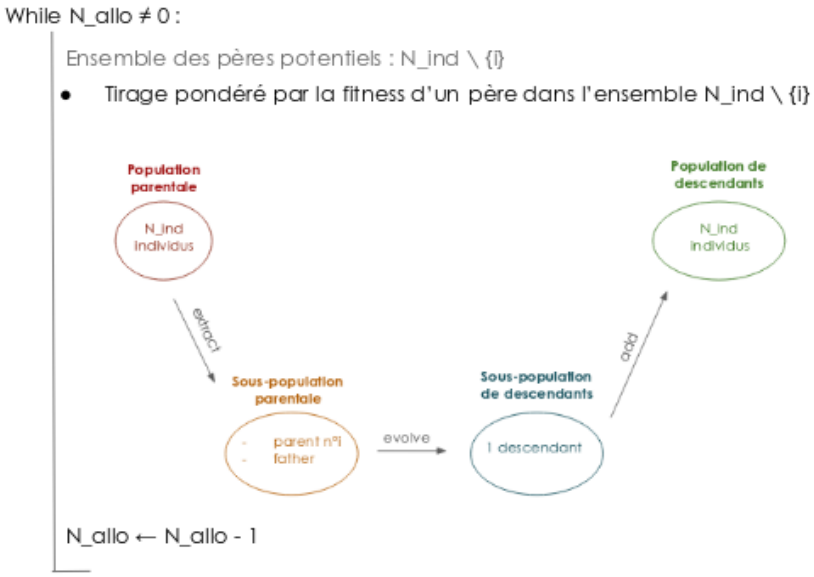


Schéma d'allogamie

Après avoir appliqué l'allogamie et l'autogamie, on obtient alors une population de descendants finale de cette forme :

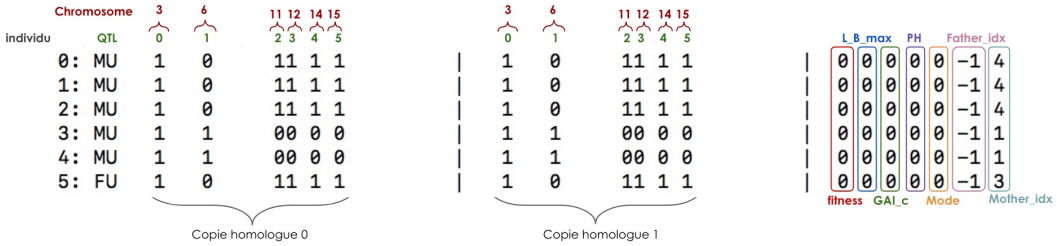


Figure 7 – Capture d'écran annotée de l'objet population de descendants

6 Classe Offspring

Cette classe permet de passer de l'objet Population de la population finale de descendants à des fichiers de phénotype et de génotype que WALTer pourra reprendre.

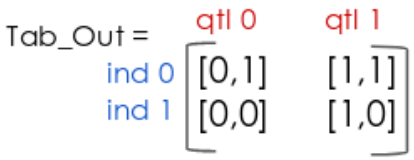
6.1 Attributs

- *Offspring.trt_interest* est la liste de traits d'intérêt (qu'on récupère de l'attribut *Reader.list_trt*),
- *Offspring.qtl_limits* est la liste dictionnaire qui donne les bornes des sommes de QTL pour chaque trait (calculée dans la classe *Reader*),
- *Offspring.pheno_limits* est la liste dictionnaires qui donne les limites phénotypiques réelles pour chaque trait.

6.2 Offspring.OffspringGeno(offpop,conv,tab_qtl)

L'argument *offpop* est l'objet Population() de la population finale de descendants. Pour chaque individu et chaque QTL on regarde les allèles sur les deux versions homologues pour reformer des doublets $[al_1, al_2]$ sous forme d'un tableau.

Exemple : On considère une population finale de descendants avec 2 individus et 2 QTL :



6.3 Offspring.dictGenoOffspring(genotab)

Cette fonction prend comme argument un tableau *genotab* de génotype des descendants pour le convertir en liste dictionnaire.

Exemple : En reprenant l'exemple précédent on a :

```
dictGeno = { 'ind 0' : { 'qtl 0' : [0,1], 'qtl 1':[1,1] } , 'Ind 1' : { 'qtl0' : [0,0], 'qtl 1' : [1,0] } }
```

6.4 Offspring.offspringQTL(dictGeno,conv)

Cette fonction prend en argument le génotype de la population finale de descendants *dictGeno* et la table de conversion allèle/valeur de QTL *conv*. Cette fonction permet d'obtenir un tableau qui contient les valeurs de QTL pour chaque individu et chaque QTL. Les valeurs sont dans des listes, afin de gérer les cas de pléïotropie où on a alors plusieurs valeurs dans une liste.

Exemple : On reprend la précédente liste dictionnaire de génotype et la table de conversion établie avec la classe *WALTerReader*.

```
dictConv = { 'qtl 0' : { '[0,0]' : 0, '[0,1]' : 0.5, '[1,0]' : 0.5, '[1,1]' : 0.5 } ,
 'qtl 1' : {'[0,0]' : 0, '[0,1]' : 1, '[0,2]' : 2, '[1,1]' : 2, '[1,0]' : 1, '[1,2]' : 3, '[2,2]' : 4, '[2,0]' : 2, '[2,1]' : 3 } }
```

```
QTL_Tab = [ [0.5,1.0]   [2.0]
            [0.0,0.0]   [1.0] ]
```

6.5 Offspring.trt_indices(tab_qtl,trt)

Cette fonction parcourt la colonne "type" du tableau de données QTL pour chaque trait et pour chaque QTL on regarde l'indice du trait dans la liste "type" du QTL, et si le trait n'est pas dans cette liste la fonction renvoie l'indice -1. Cela permet de gérer les cas de pléïotropie (et de récupérer la bonne valeur de QTL quand il y en a plusieurs).

6.6 Offspring.rescaling(val,real_min,real_max,obs_min,obs_max)

Cette fonction permet de remettre à l'échelle une valeur initiale de QTL *val* (qui peut être comprise entre un minimum obs_min et un obs_max) entre un minimum réel *real_min* et un maximum réel *real_max*. On transforme d'abord éventuellement la valeur *val* de façon à ce qu'elle soit positive (en lui ajoutant la valeur absolue de obs_min). Puis on applique à

val la fonction $f : x \mapsto \frac{real\_max - obs\_max}{real\_min - obs\_min} \cdot x + real\_min$. Lorsque l'on a des valeurs de QTL qui ne sont pas réalistes, on peut alors passer de la valeur de somme des QTL à une valeur de trait correspondante réaliste.

Exemple :

Data_QTL =

n° QTL	type	chrom	pos	valeur d'effet	nom	dominance
0	['PH','RS']	3	12.6	[2.4,-0.1]	qPh1,qRs1	[complete,partial]
1	['RS']	5	35.9	[0.2]	qRs2	[recessive]

trt_indices(Tab,'PH') = [0,-1] → pour le type PH, il s'agit de la valeur n°0 pour le QTL n°0 et il n'y a pas de valeur pour le QTL n°1

trt_indices(Tab,'RS') = [1,0] → pour le type RS, il s'agit de la valeur n°1 pour le QTL n°0 et de la valeur n°0 pour le QTL n°1

6.7 Offspring.GenoWALT2WALTer(offpop,tab_qtl,conv,write_data)

Cette fonction prend comme arguments la population de descendants *offpop*, les données QTL *tab_qtl*, la liste dictionnaire de conversion *conv* et l'argument *write_data* qui permet de savoir si le fichier *population_data* doit être écrit ou pas. Cette fonction permet de passer d'un objet Population simuPOP à une liste dictionnaire de génotype globale sur la population puis à un fichier *population_data* et un fichier de conversion pour les noms de génotypes *geno_conversion*.

Dans un premier temps on veut créer une liste dictionnaire qui contient les génotypes uniques de la population et leur effectif. On obtient une liste dictionnaire *Dict_Unique_Offsp_Geno*. Par exemple, considérons une population de descendants avec 4 individus et 2 génotypes tels que :

"Geno0"	{ qtl0 : [0,1], qtl1 : [1,1] }
"Geno1"	{ qtl0 : [0,0], qtl1 : [1,0] }

dictGenoOffsp = { **'Ind 0'** : { 'qtl 0' : [0,1], 'qtl 1':[1,1] } , **'Ind 1'** : { 'qtl 0' : [0,0], 'qtl 1' : [1,0]},
'Ind 2' : { 'qtl 0' : [0,1], 'qtl 1':[1,1] } , **'Ind3'** : { 'qtl 0' : [0,0], 'qtl 1' : [1,0]} }

Dans un second temps on s'occupe de créer le fichier de conversion de génotype *geno_conversion.csv*, pour cela on récupère seulement les noms des génotypes et leurs listes dictionnaires associées dans *Dict_Unique_Offsp_Geno* qui prend donc cette forme :

```
dict_Unique_offsp_geno = { 'Geno0' : {'eff' : 2, 'dico': {'qtl 0':[0,1], 'qtl 1':[1,1]} } ,
                           'Geno1' : {'eff' : 2, 'dico' : {'qtl 0':[0,0], 'qtl 1':[1,0]} } }
```

On récupère alors les informations de noms de génotypes et de génotypes comme listes de QTL pour écrire le fichier *geno_conversion.csv*.

Pour l'exemple, on se place dans la situation suivante :

n° QTL	type	chrom	pos	valeur d'effet	nom	dominance
0	['PH','RS']	3	12.6	[2.4,-0.1]	qPh1,qRs1	[complete,partial]
1	['RS']	5	35.9	[0.2]	qRs2	[recessive]

```
qtl_limits = { 'PH' : [0, 2.4], 'RS' : [-0.1, 0.2] }

pheno_limits = {'PH' : [57.9, 121.6], 'RS' : [63.9, 136.5] }
```

```
dictQTL = { 'PH' : [ chrom = [3], pos = [12.6], val = [2.4], names = ['qPh1qRs1'] ],
            'RS' : [ chrom = [3,5], pos = [12.6,35.9], val = [-0.1,0.2], names = ['qPh1qRs1','qRs2'] ] }

dictConv = { 'qtl 0' : { '[0,0]' : [0,0], '[0,1]' : [2.4,-0.05], '[1,0]' : [2.4,-0.05], '[1,1]' : [2.4,-0.1] } ,
             'qtl 1' : {'[0,0]' : [0], '[0,1]' : [0], '[1,0]' : [0], '[1,1]' : [0.2] } }
```

Dans la même structure de dictionnaire des génotypes uniques, on veut stocker les informations de traits et de fitness. Pour chaque génotype, puis pour chaque trait on calcule donc la valeur de trait associé, on crée une clé pour ce trait dans la liste dictionnaire *Dict_Unique_Offsp_Geno*. Pour calculer les valeurs de trait pour un génotype, on récupère les indices de QTL associés à ce trait avec la fonction *Offspring.trt_indices(tab_qtl,trt)*. On obtient alors une liste avec des valeurs de QTL à laquelle on applique la fonction *Offspring.rescaling* pour obtenir la valeur de trait.

- genotype = "Geno0"

  geno_dict = {'qtl 0':[0,1], 'qtl 1':[1,1]}
  global_list_QTL = [[2.4,-0.05], [0.2]] ↷ offspringQTL(geno_dict)

  - trt = PH
    list_PH = [0,-1]
    QTL_val= [2.4]
    sum_QTL = 2.4

    offspringPheno(QTL_val) :
    val_PH0 = rescaling(sum_QTL, real_lims['PH'], obs_lims['PH'])
    val_PH0 = rescaling(2.4,[57.9, 121.6],[0, 2.4])
    val_PH0 = 121.6

  - trt = RS
    list_RS = [1,0]
    QTL_val= [-0.05,0.2]
    sum_QTL = 0.15

    offspringPheno(QTL_val) :
    val_RS0 = rescaling(sum_QTL, real_lims['RS'], obs_lims['RS'])
    val_RS0 = rescaling(0.15,[63.9, 136.5],[-0.1, 0.2])
    val_RS0 = 124.4

- genotype = "Geno1"

  geno_dict = {'qtl 0':[0,0], 'qtl 1':[1,0]}
  global_list_QTL = [[0,0], [0]] ↷ offspringQTL(geno_dict)

  - trt = PH
    list_PH = [0,-1]
    QTL_val= [0]
    sum_QTL = 0

    offspringPheno(QTL_val) :
    val_PH1 = rescaling(sum_QTL, real_lims['PH'], obs_lims['PH'])
    val_PH1 = rescaling(0,[57.9, 121.6],[0, 2.4])
    val_PH1 = 57.9

  - trt = RS
    list_RS = [1,0]
    QTL_val= [0,0]
    sum_QTL = 0

    offspringPheno(QTL_val) :
    val_RS1 = rescaling(sum_QTL, real_lims['RS'], obs_lims['RS'])
    val_RS1 = rescaling(0,[63.9, 136.5],[-0.1, 0.2])
    val_RS1 = 88.1

On obtient alors une liste dictionnaire du type :

```
dict_Unique_offsp_geno = { 'Geno0' : {'eff' : 2, 'dico': {'qtl 0':[0,1], 'qtl 1':[1,1]} , 'PH' : 121.6, 'RS' : 124.4 } ,
                           'Geno1' : {'eff' : 2, 'dico' : {'qtl 0':[0,0], 'qtl 1':[1,0]} , 'PH' : 57.9, 'RS' : 88.1} }
```

Finalement à partir de cette liste dictionnaire on va remplir la table *population_data*. Cette table est constituée d'autant de lignes que d'individus +1 (pour l'en-tête) et autant de colonnes que de traits d'intérêt +3 (individu, fitness, génotype). Pour chaque génotype unique la fitness est calculée, c'est donc à ce niveau qu'une fonction de fitness peut-être appelée (pour l'instant il est seulement indiqué que la valeur de fitness est la valeur de hauteur). Une ligne est créée avec les informations de fitness, génotype et valeurs phénotypiques. Cette ligne est alors répliquée autant de fois qu'il y a d'individus pour ce génotype. A l'issu de cette étape on obtient une table *data_population*.

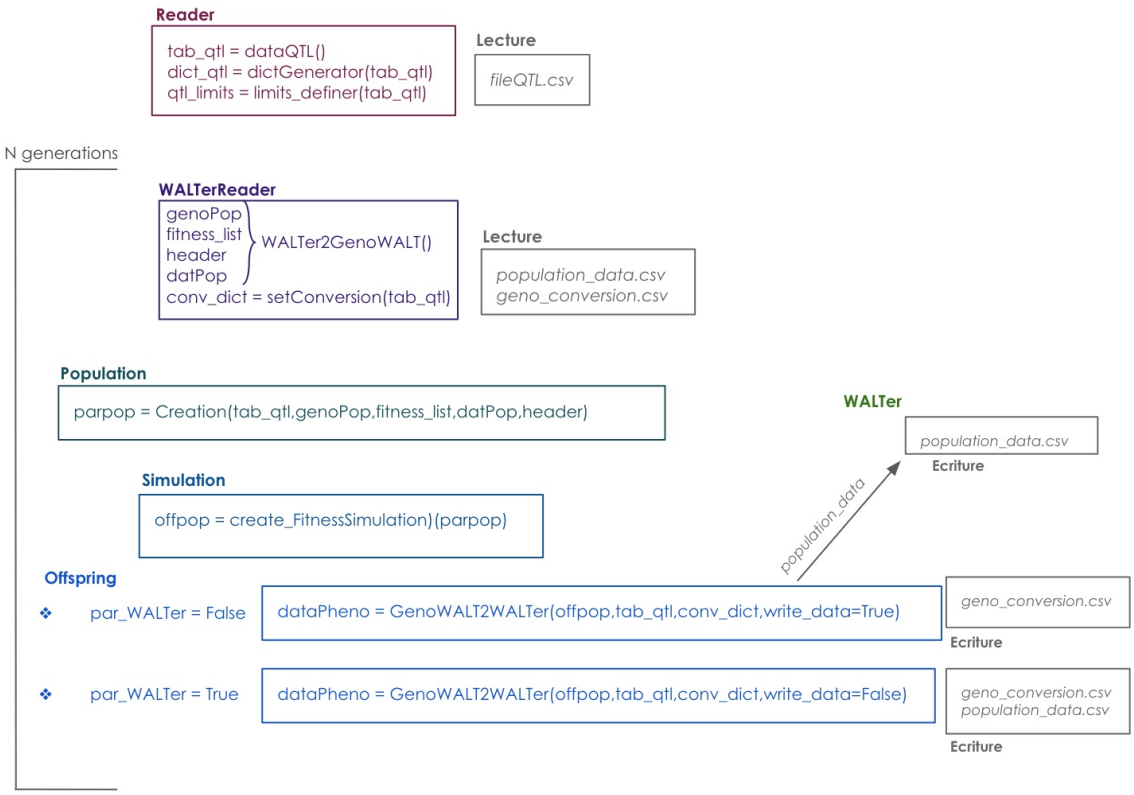
La fonction *Offspring.GenoWALT2WALTer* renvoie cette table comme argument. Si on est dans un cas où GenoWALT est utilisé en Solo, alors cette table est écrite dans un fichier csv car elle sera réutilisée telle quelle pour la génération suivante, l'argument *write_data* est alors égal à *True*. Si on est dans un cas de couplage avec WALTer alors le fichier n'est pas écrit, la table est transmise à un script extérieur à GenoWALT, *pheno2sim.py* qui crée le fichier *variable_geno_params.csv* dont WALTer a besoin. Une fois que WALTer a tourné, le fichier *population_data.csv* est alors écrit à partir des données simulées.

6.8 Offspring.offspringPheno(choice,list_qtl,trt)

Cette fonction permet de passer du génotype au phénotype pour un génotype et un trait donné. Avec l'option *choice_pheno="rescaling"* on calcule la somme des valeurs de QTL sur laquelle on applique la fonction *Offspring.rescaling* pour avoir la valeur du trait.

7 Utilisation des différentes classes dans le *main*

Les différentes classes dans le fonctionnement est expliqué précédemment sont appelées une à une dans le fichier *main.py* qui permet de lancer l'exécution du module.



Un paramètre *par_walter* permet de signaler si le module est utilisé en couplage à WALTer car si c'est le cas il faut faire une conversion de fichier entre les sorties de GenoWALT et les entrées de WALTer (explications dans la section suivante).

8 Création de *variable_geno_params* à partir de GenoWALT

Le fichier *variable_geno_params.csv* regroupe les paramètres génotypiques variables (donc les paramètres écophysiologiques génétiquement déterminés) dont la valeur va être déterminée par GenoWALT. Ce fichier se présente sous la forme suivante :

genotypes	genotype_numbers	PH_Geno0	RS_Geno0	PH_Geno1	RS_Geno1
Geno0,Geno1	2,2	121.6	124.4	57.9	88.1

Pour obtenir des données sous cette forme il faut donc transformer la structure de données gérées par GenoWALT. Pour cela le script *pheno_sim.py* a été conçu. Pour passer de la table *data_population* au format voulu il s'agit de faire d'abord une recherche des génotypes uniques. Pour chaque génotype on récupère l'effectif et les valeurs génétiques des différents traits d'intérêt pour ce génotype. Cette étape n'est effectuée que lors des couplages avec WALTer.

9 Couplage WALTer/GenoWALT

Le couplage comprend 4 scripts python : le script WALTer, le script de conversion du fichier de sortie WALTer au fichier d'entréee de GenoWALT, le script de GenoWALT et le script d'assemblage des fichiers de paramètres pour créer le fichier d'entrée *sim_scheme.csv* de WALTer. Il y a 3 fichiers de paramètres pour créer *sim_scheme* :

- *fixed_expe_cond_params* : les paramètres expérimentaux fixes. Il s'agit des paramètres comme les dimensions de la parcelle, le nombre de plantes, le nombre de rangs, la densité de plantes, le dGAIp ou encore des paramètres pour le module de rayonnement CARIBU.
- *variable_geno_params* : les paramètres génotypiques variables. Il s'agit de la valeur du trait d'intérêt pour chaque génotype ainsi que de l'information *genotypes* avec les noms des différents génotypes et leurs effectifs dans l'information *genotype_numbers*
- *fixed_geno_params* : les paramètres génotypiques fixes. De la même façon que pour les paramètres génotypiques variables, on veut la valeur de tous les traits (qui ne sont pas d'intérêt alors toujours égaux à la même valeur) pour chaque génotype. On garde par exemple les informations sur le nombre de feuilles, le Δ_{prot}, la quantité de lumière reçue (Part) ou le GAI, même si on considère qu'il n'y a pas de variabilité sur ces traits, et donc qu'ils garderont toujours la même valeur au cours des générations. On a donc une structure "trt_geno" avec "trt" le nom du trait et "geno" le nom du génotype. Comme on change les noms des génotypes à chaque génération, on stocke l'information sur ces paramètres avec seulement le nom du trait et la valeur associée. On met à jour ces informations avec les noms des génotypes dans le script *create_simscheme.py*.

L'ensemble des scripts python sont lancés depuis un script bash de la façon suivante :

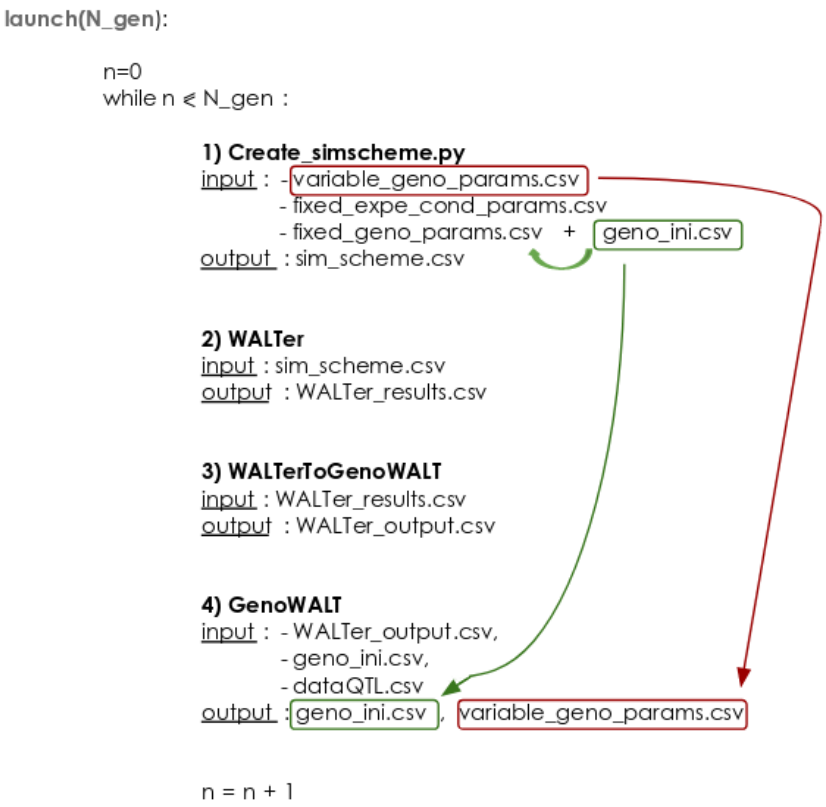


Figure 8 – Schéma du fonctionnement du couplage GenoWALT/WALTer

Lorsqu'on lance une simulation couplée sur plusieurs générations, les fichiers de paramètres doivent être rentrés (initialisés par l'utilisateur) pour créer le fichier *sim_scheme.csv et lancer WALTer*, ainsi que les fichiers *geno_ini.csv* pour créer le simscheme et pour initialiser les génotypes dans GenoWALT, et *dataQTL.csv* pour GenoWALT. Parmi les fichiers de paramètres, seulement *variable_geno_params* va être mis à jour à chaque génération, en parallèle avec le fichier *geno_ini.csv*. Les fichiers de paramètres *fixed* restent les mêmes au cours des générations.

10 Génération automatique des fichiers de départ

La création des fichiers de départ peut être longue et fastidieuse. En effet, pour un génotype mono-trait ou multi-trait il est intéressant pour l'utilisateur de choisir les valeurs des traits d'intérêt. Par exemple, dans un cas où l'on étudie deux trait, la hauteur PH et la surface racinaire RS, on voudrait avoir un génotype (donc des allèles au niveau des différents QTL) qui correspond à une hauteur de 40 cm et une surface racinaire de $100cm^2$. Pour cela la démarche suivante est mise en place : parmi toutes les combinaisons d'allèles possibles sur l'ensemble des QTL (car l'ensemble de toutes les combinaisons est parfois trop élevé), pour chaque génotype échantillonné on calcule une erreur relative par rapport aux valeurs rentrées par l'utilisateur et le génotype retenu est celui avec l'erreur relative la plus faible.

Le script *ini.py* permet de générer ces fichiers en demandant à l'utilisateur les informations nécessaires, à savoir :

- le nombre de génotypes à générer
- le nombre de traits à étudier
- quels sont ces traits
- pour chaque génotype, les valeurs voulues pour chaque trait
- le nombre d'individu par génotype
- pour chaque génotype, si il est purement homozygote, hétérozygote ou mixte
- combien de génotypes à échantillonner dans la démarche pour trouver le génotype le plus proche

A partir de toutes ces informations, le script écrit les fichiers d'entrée pour GenoWALT. L'utilisateur doit toutefois être prudent, pour que les informations qu'il donne sur les traits d'intérêt soient en cohérence avec les données QTL.

11 Gestion du multi-allélisme

La version basique qui gère le multi-allélisme ne permet pas de gérer le multi-trait, le couplage avec WALTer, ni la création automatique des fichiers d'entrée. Dans cette version les fichiers de données QTL sont légèrement différents puisque pour chaque QTL, l'information du nombre d'allèles est ajoutée.

Pour calculer l'effet pour allèle deux distributions ont été implémentées : affine ou exponentielle. Il y a donc un attribut *distrib* pour la classe *WALTerReader*. Deux fonctions ont donc été implémentées dans cette classe :

- WALTerReader.affine(x,N,v)
  Cette fonction attribue une valeur d'effet à chaque allèle d'un QTL, où x correspond à l'allèle (un chiffre entre 0 et $N-1$), v à la valeur optimale du QTL et N au nombre d'allèles pour le QTL concerné. On a ainsi $affine(x, N, v) = \frac{x}{N-1}\ v$. Cette fonction est utilisée quand un QTL a plus de deux allèles.

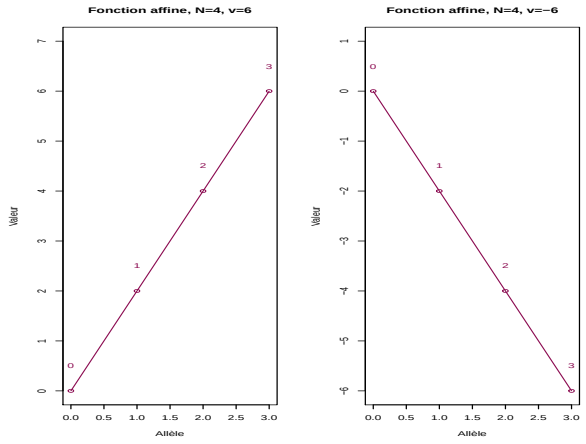
  Exemple : On considère un QTL avec $N = 4$ allèles $\{0, 1, 2, 3\}$ et une valeur $v = 6$ ou $v = -6$. On obtient alors les valeurs suivantes pour les allèles :

Allèle	Valeur
0	0
1	2
2	4
3	6

v = 6

Allèle	Valeur
0	0
1	-2
2	-4
3	-6

v = - 6



– **WALTerReader.expo(x,N,v)**
Cette fonction attribue une valeur d'effet à chaque allèle d'un QTL, où x correspond à l'allèle (entre 0 et $N-1$), v à la valeur optimale du QTL et N au nombre d'allèles pour le QTL concerné. On a ainsi $expo(x, N, v) = \frac{v}{e^{N-1}}\ (e^x - 1)$. Cette fonction est utilisée quand un QTL a plus de deux allèles.

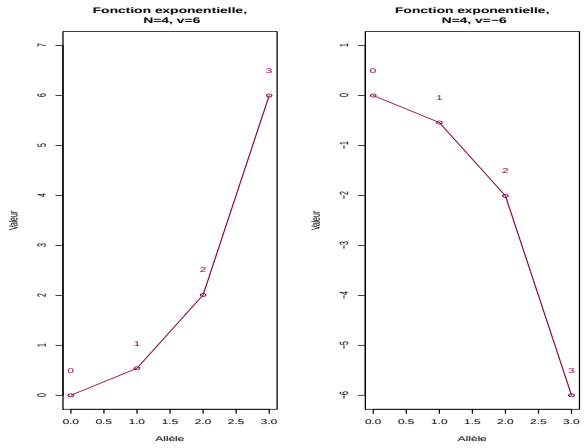
Exemple : On considère un QTL avec $N = 4$ allèles $\{0, 1, 2, 3\}$ et une valeur $v = 6$ ou $v = -6$. On obtient alors les valeurs suivantes pour les allèles :

Allèle	Valeur
0	0
1	0.54
2	2.01
3	6

v = 6

Allèle	Valeur
0	0
1	-0.54
2	-2.01
3	-6

v = - 6



Le cas bi-allélique est beaucoup plus simple : on attribue la valeur 0 à l'allèle 0 et la valeur v à l'allèle 1.

Pour calculer l'effet de la combinaison de deux allèles on fait simplement la moyenne des valeurs d'effets des deux allèles.

La fonction *WALTerReader.setConversion* qui permet de faire la conversion entre les doublets d'allèles et les valeurs génétiques associées doit pouvoir gérer les cas multi-alléliques :

On calcule les valeurs associées à chaque allèle avec la fonction *affine* ou *expo*. Avec la fonction *itertools.product* on calcule les N^2 arrangements d'allèles pris 2 à 2. On regarde les deux allèles pris dans chaque chaque arrangement et on prend comme valeur d'effet pour cette arrangement la moyenne entre les deux allèles.

Exemple : On considère un QTL avec $N = 3$ allèles et une valeur $v = 4$:

Allèle	Valeur
0	0
1	2
2	4

v([0,0]) = 0 (0+0)/2 v([0,1]) = 1 (0+2)/2 v([0,2]) = 2 (0+4)/2

v([1,1]) = 2 (2+2)/2 v([1,0]) = 1 (2+0)/2 v([1,2]) = 3 (2+4)/2

v([2,2]) = 4 (4+4)/2 v([2,0]) = 2 (4+0)/2 v([2,1]) = 3 (4+2)/2

A l'issue de cette fonction on obtient une liste dictionnaire avec comme clés les QTL et comme valeurs une liste dictionnaire où les clés sont les arrangement avec comme valeurs leurs effets.

Exemple : On considère ici deux QTL avec respectivement 2 (dominance complète) et 3 allèles et des valeurs d'effets de 0.5 et 4.

```
dictConv = { 'qtl 0' : { '[0,0]' : 0, '[0,1]' : 0.5, '[1,0]' : 0.5, '[1,1]' : 0.5 } ,
 'qtl 1' : {'[0,0]' : 0, '[0,1]' : 1, '[0,2]' : 2, '[1,1]' : 2, '[1,0]' : 1, '[1,2]' : 3, '[2,2]' : 4, '[2,0]' : 2, '[2,1]' : 3 } }
```