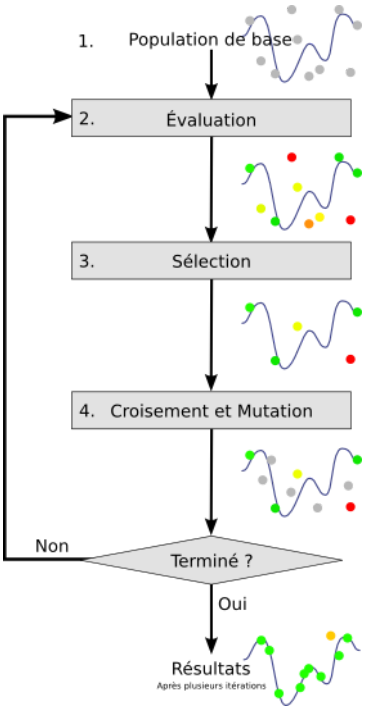
# Séance du 4 mai




Objectif : établir un cahier des charges avec le client.

## Cahier des charges vu avec le client

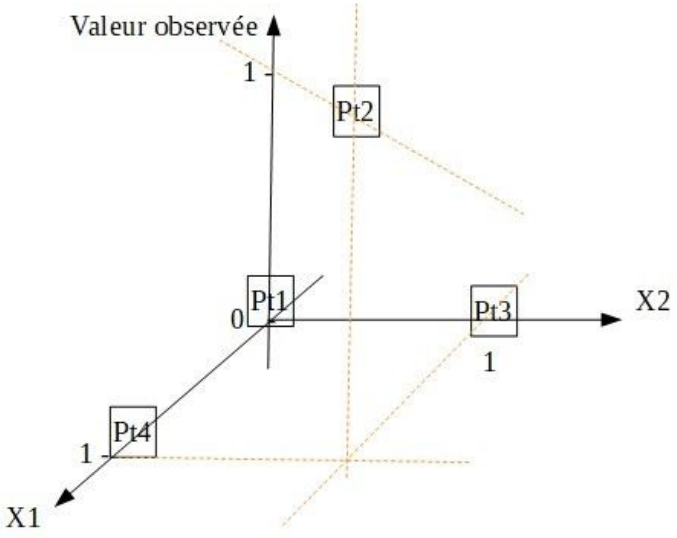
**Fonction du délivrable :** Établir une formule optimale correspondant à la meilleure combinaisons de gènes pour retrouver les résultats fournis par l'utilisateur.

**But :** prédire la n-ème coordonnée d'un point à partir des n-1 premières.
Démarche : trouver une fonction qui minimise la somme des écarts entre valeurs prédites et valeurs réelles (de la n-ème coordonnée).

**Input :** fichier csv avec m lignes correspondant aux expériences et n colonnes correspondant aux gènes.
Les valeurs prises ne peuvent être que 0 ou 1 selon l'absence ou présence d'expression du gène concerné dans l'expérience en question.

L'utilisateur peut entrer différents paramètres :

- nombre d'itérations (il faudra définir un max d'itérations et une valeur par défaut)
- nombre d'enfants par génération




Règle candidate :
[Les points dont décrits par (X1, X2, Y)]

| X1 | and | X2 | : | Y |
|---|---|---|---|---|
| 0 | | 0 | | 0 |
| 1 | | 1 | | 1 |
| 0 | | 1 | | 0 |
| 1 | | 0 | | 0 |

**Ouput :**
-Meilleure fonction basée sur la SSE le plus faible OU sur un nombre de fonctions avec les SCE les plus faibles avec des scores éventuels

On peut établir le score de la manière suivante :
fitness = -nombre d'erreurs  = - SSE= - \sum_{i=1}^m (f(X_{i,1:n-1}) - X_{i,n})^2

erreur = f(X_{i,1:n-1}) != X_{i,n}

Pour la SSE, qui est la somme des carrés des écarts, on veut pour chaque ligne de notre tableau des résultats observés où le gène d'intérêt est exprimé (dernière colonne == 1), la somme des différences avec la valeur prédite (contenue dans la formule) pour chaque colonne.

Plus elle est élevée, moins c'est bien.
fitness = -SSE, donc plus la fitness est élevée, mieux c'est.

- Graphe sous Python (librairie Panda ?) -> Allez voir 'python.auxiliary.zip' (fonction pour tracer un arbre), il y a déjà un fichier python pour faire un graphe à partir d'une liste de points générés au fur et à mesure des générations.

On peut imaginer un point vert si la prédiction est bonne et un point rouge sinon (ça serait en 1D juste une succession de points)

- structure du meilleur individu
- meilleure formule

**Entre input et output :** création de la fitness function
Au début du code, il faut des fonctions randoms (on ne connaît pas encore le lien entre 1ère, n-1-ième, …. et n-ième coordonnée)
Une fonction prend en argument les coordonnées 1 à n-1, et renvoie la coordonnée n.

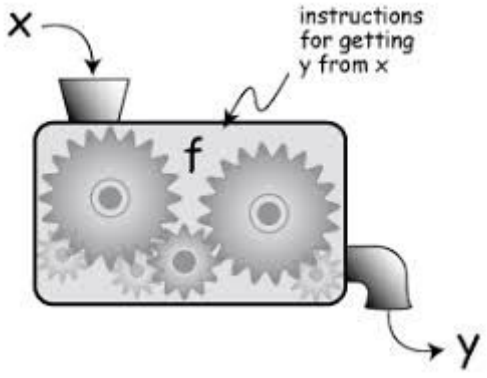



**Figure 1 : illustration d'une fonction; légende : x  variables en entrée, y est le résultat prédit**

Le nombre de fonctions randoms obtenues est défini par le client.
Si on prend le nombre de noeuds : il faut créer plusieurs branches avec pour chaque noeud une opération
Comment les choisir ? -> randomisé :

- pour les noeuds internes: 3 options possibles (and, or , not)
- pour les feuilles (points terminaux d'un arbre): x1….x_{n-1} ou 0 ou 1

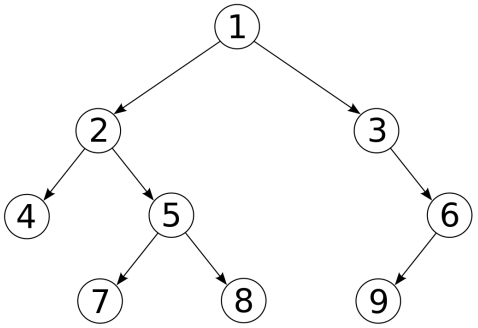



**Figure 2 : illustration d'un arbre, chaque noeud correspond à une mutation**

A chaque itération, induire des mutations correspondant à des modifications de la fitness function. Les mutations ont une probabilité p donnée pour chaque noeud
à chaque itération, le score est calculé : c'est la SSE : somme des carrés des écarts entre valeurs prédites et valeurs réelles fournies par l'utilisateur.
Est-ce qu'on décide de garder quelques fonctions avec faible score SSE ou seulement la meilleure fonction ? → On ne garde que la meilleure en fonction de ce score, cette nouvelle fonction est gardée ou non pour la prochaine génération.
Il faudrait fixer un seuil pour ce score : si ce seuil est atteint alors on renvoie le output, sinon on continue les itérations.
Ce seuil doit pouvoir être entré par l'utilisateur lorsqu'il utilise le programme. Mais il doit être initialisé à une valeur par défaut si l'utilisateur ne rentre pas de valeur pour ce paramètre.

Date des meetings : 11/05 ; 18/05 ; 02/06 ; 08/06 ; 10/06 ; 13/06 ; 14/06

Date de rendu du délivrable : 15/06

Forme du rendu du délivrable : le logiciel fonctionnel permettant de retourner une formule optimale à partir des paramètres entrés par l'utilisateur soit le nombre de générations voulues, le nombre d'enfants par génération et un tableau .CSV donné en entrée.

Il sera accompagné de la documentation l'explicitant.

Idées dans la classe à produire : méthode pour extraire les données ; méthode pour chaque cas de calcul (AND, OR et AND) [fonction récursive] ; méthode pour obtenir un nombre aléatoire pour les randomisations à utiliser ; méthode pour return un fichier avec les résultats.
Ensuite fichier python qui prend l'output et met en forme les résultats (formule, graphe)-> on peut s'inspirer du code donné 'python_auxiliary.zip' (prend une liste de liens en entrée) fourni

# Séance du 11 mai

## Objectif : analyser le tableau de données en entrée et établir une structure pour le code, début de pseudocode

**Dataset** : 'binary_gene_expression_ACE2_tfs.csv'
**Présentation dataset** de Maître Sergio :
Each column represents a gene (the first n-1 columns code for transcription factors, i.e. genes that will control the expression of other genes, and column n, represents the gene that codes for the ACE2 enzyme, which is the entry point of SARS-CoV-2 in the cells)
Each row corresponds to an experimental condition (patient sick, not sick, under treatment etc...). If X_{ij} = 1 then it means that the expression of gene j in condition i is higher than the average expression of gene j (otherwise X_{ij} = 0).

The goal is to determine which logical association between the transcription factors allows us to model the expression of the gene that codes for ACE2.

But final : Déterminer quel(s) facteur(s) de transcription permet(tent) une expression optimale du gène qui code pour ACE2 ? Soit quelle association de 0/1 assurerait le mieux l'expression d'ACE2.

Conditions expérimentales

| | gene1 | gene2 | ... | gene n-1 | gene n (codant pour l'ACE) |
|---|---|---|---|---|---|
| exp1 | X11 | X12 | | X1n-1 | X1n |
| exp2 | X21 | | | | |
| ... | | | | | |
| expm | Xm1 | | | | Xmn |

gènes codant pour des facteurs de transcription (contrôle de l'expression d'autres gènes)

| | A | B | C | D | E |
|---|---|---|---|---|---|
| 1 | | ENSG00000184677.16 | ENSG00000162419.12 | ENSG00000117000.8 | ENSG00000162367.11 |
| 2 | GTEX-WYBS-1126-SM-3NMAM | 0 | 0 | 1 | 0 |
| 3 | GTEX-WH7G-0726-SM-3NMBM | 1 | 0 | 0 | 0 |
| 4 | GTEX-OOBJ-0526-SM-48TDK | 0 | 1 | 0 | 1 |
| 5 | GTEX-UJMC-0726-SM-3GADX | 1 | 1 | 0 | 1 |
| 6 | GTEX-13O3O-0726-SM-5J1N7 | 0 | 0 | 1 | 1 |
| 7 | GTEX-1B8KZ-0526-SM-73KW2 | 1 | 1 | 0 | 1 |
| 8 | GTEX-1HCU7-0926-SM-A96TK | 1 | 1 | 0 | 0 |
| 9 | GTEX-17GQL-0726-SM-731BL | 1 | 1 | 0 | 1 |
| 10 | GTEX-18D9B-1026-SM-CNPO5 | 0 | 0 | 0 | 0 |
| 11 | GTEX-1H1DG-1026-SM-9MOJS | 0 | 0 | 1 | 0 |

**Liste des méthodes** :

- ouvrir fichier csv/importer les données -> (m tableau de bool?)

1ère colonne : nom des conditions expérimentales
1ère ligne : nom des gènes

On peut rentrer les données sous forme de matrice. En C++, ce serait un tableau 2D en parcourant avec une boucle le fichier csv. Problème : c'est peut-être pas adapté car ici nos lignes font toujours n colonnes. On pourrait alors faire m tableaux 1D qu'on appellerait 1 à m → ce serait + rapide à traiter) En tous les cas, ça serait des tableaux de booléens donc il nous faut une méthode qui détermine le nombre de lignes **m** et qui crée un tableau de booléen par ligne.

- un opérateur AND
- un opérateur OR
- un opérateur NOT
- générer lambda combinaisons aléatoires (commencer avec individus petits, pas bcp de noeuds)
- calcul le nombre de noeuds? (nb d'opérations à faire pour chaque combinaison)
- Remonter un arbre :
  On commence par calculer le résultat qu'on a quand on applique le même opérateur à l'ensemble des Xi [ex : X11 && X12 && X13 &&... = X1n]
  On regarde les meilleurs fitness (SSE) pour voir les meilleurs opérateurs.
  Stocker les liens.
  NB : La SSE s'applique sur la somme pour toutes les m expressions (y_prédit - y_donné)². Elle teste la qualité d'une fonction.

Est-ce qu'on pourrait pas faire une matrice avec (n-1 -1) valeurs choisies au hasard au début. Elle serait remplie par des 1 si on veut appliquer AND, des 2 si on veut appliquer OR et des 3 si on veut appliquer des NOT.

| Indice | Opérateurs | Matrice d'opérateurs |
|---|---|---|
| 1 | X11 and X12 | 1 |
| 2 | X12 or X13 | 1 |
| 3 | X13 or X14 | 2 |
| ... | ... | ... |
| n-2 | not X1n-1 | 3 |

=> (((X11 AND X12) AND X13) OR X14) ….NOT X1n-1

Si on a un NOT il nous faut un autre opérateur pour le lier aux autres

[ex : X11 AND (NOT X12) ]

Penser à faire plusieurs classes pour le code.
On a lambda formules f par génération. Lambda fonctions candidates définies aléatoirement au début. Après, on garde la meilleure et on travaille dessus. A la génération 2, on peut imaginer avoir 1 fonction f parent + (lambda-1) fonctions f filles.

## File handling (2): Read from a text file

| Python 3 | C | C++ |
|---|---|---|
| `file = open("aaa.txt", "r")`<br>`words = file.read().split()`<br>`name = words[0]`<br>`age = words[1]`<br>`print(name + " " + str(age))`<br>`file.close()` | `#include <stdio.h>`<br><br>`int main() {`<br>`  char login[100];`<br>`  int age;`<br>`  FILE * myfile;`<br>`  myfile = fopen("aaa.txt", "r");`<br>`  fscanf(myfile, "%99s", login);`<br>`  fscanf(myfile, "%d", &age);`<br>`  fclose(myfile);`<br>`  printf("%s %d\n", login, age);`<br>`  return 0;`<br>`}` | `#include <fstream>`<br>`#include <iostream>`<br>`using namespace std;`<br><br>`int main() {`<br>`  char login[100];`<br>`  int age;`<br>`  fstream myfile;`<br>`  myfile.open("aaa.txt", ios::in);`<br>`  myfile >> login;`<br>`  myfile >> age;`<br>`  myfile.close();`<br>`  cout << name << " " << age << endl;`<br>`  return 0;`<br>`}` |
| **File opening modes**<br>`"r" : read` | **File opening modes**<br>`"r" : read` | **File opening modes**<br>`std::ios::in : read` |

**PSEUDOCODE** :

- ❖ Classe qui regroupe tout : nb de noeuds, consignes
- ❖ méthode import du fichier csv

header = true
csv :
colonne 1 : nom des expériences
n-2 premières colonnes sont les variables explicatives (on ne compte pas la colonne “titre”)
colonne n c’est variable réelle à expliquer

*Déclaration: nom : (type)*
nb_lignes (int) = 0
Ouvrir le fichier csv, parcourir le fichier et pour chaque '\0' rencontré, nb_lignes +=1

nb_colonnes (int) = 0
Ouvrir le fichier csv, tant que '\0' n'est pas atteint, parcourir le fichier et pour chaque ',' rencontrée, ,b_colonnes+=1
m = 1
n = 1

Tableau lignes[nb lignes][nb_colonnes] : (booléens) //crée le tableau

*Programme:*

```
     Pour m à nb_ligne-1 :
          Pour n à nb_colonnes-1 : // on est sur la ligne, on
cherche la valeur de chaque colonne pour la ligne courante
               // Allouer la valeur précédent la virgule en
position n
               Après la n-ième ',' faire : // boucle While
sous entendue avec ','
               tab_final[m][n] -> valeur;
```

csvStr.split("\n").map(function(row){return row.split(",");})

```
function csvToArray (csv)

     { rows = csv.split("\n"); return rows.map(function (row) {

          return row.split(",");

     });

};
```

## ❖ Classe Operateur

```
#include <string>

public :

Constructeur : Operateur(string operation, bool eltA, bool eltB)
               Operateur(string operation, bool eltA)
//Le second constructeur existe surtout pour l'opération NOT qui ne
s'applique que sur 1 élément
Fonctions membres :
```

```
string id_operation();//Les getters
bool fonction_operateur(); //retourne le résultat après application de
l'opérateur

Attributs (protected/private) :
string id_operation_ ; //AND ; OR ; NOT
bool elementA_;
bool elementB_;
```

| P | Q | R | Q ou R | P et Q | P et R | (P et Q) ou (P et R) | P et (Q ou R) |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Fonction membre :

```
bool fonction_operateur() {
      bool res;
      if (id_operation_ == "AND") {
            res = elementA_ && elementB_;
      } else if (id_operation_ == "OR") {
            res = elementA_ || elementB_;
      } else if(id_operation_ == "NOT") {
            res = !elementA;
      }
      return res;
}
```

## ❖ Classe Fonction

public :

Constructeur : Formule (Objet opérateur principal) // tableau comme Ian a dit ?

Fonctions membres :

initialisation_random(); // Ian a fait une tentative plus bas

mutation(); // à préciser dans le pseudocode !

ajout_de_noeud() ; // appelé avec une probabilité dans mutation -> en gros c'est appel de constructeur 'opérateur'

Attributs (protected/private) :

entier nb_noeuds;

- Génération d'une formule random

Notre formule aura la forme :

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| not/yes | A | and/or | not/yes | B | and/or | not/yes | c | and/or | not/yes | D |

Etc.

```
n = nombre de gènes

tabgene[] = tableau contenant les gènes


ny[] = {'','!'}

ao[] = {'&&', '||'}


rankNY[] = string [n]

sizeNY = n

rankAO[] = string [n-1]

sizeAO = n-1

rankGene[] = string [n]

sizeGene = n


used[]

sizeUsed = 0
```

```
for each gene in tabgene //attribue un ordre pour les genes dans la formule
	free = true
	do
		rank = random int [0, n-1]
		for j in [0, sizeUsed[
			if used[j] == rank
				free = false
				break

	while free == false
	used[sizeUsed] = rank
	sizeUsed ++
	rankGene[rank] = gene

for j in [0,sizeNY[ //remplis le tableau attribuant soit NOT soit rien avant
chaque gene
	r = random {0,1}
	rankNY[j] = ny[r]

for j in [0,sizeAO[ //remplis le tableau mettant soit AND soit OR entre
chaque gene
	r = random {0,1}
	rankAO[j] = ao[r]



// execution de la formule

data[] //tableau contenant les données

bool result = data[rankGene[j]]
```

```
if rankYN[0] == "!"
      result = !result

for j in [0,n-1[

      if rankAO[j] == "&&"
            if rankYN[j] == "!"
                  result = result && ! data[rankGene[j+1]]
            else
                  result = result && data[rankGene[j+1]]
      else
            if rankYN[j] == "!"
                  result = result || ! data[rankGene[j+1]]
            else
                  result = result || data[rankGene[j+1]]
```

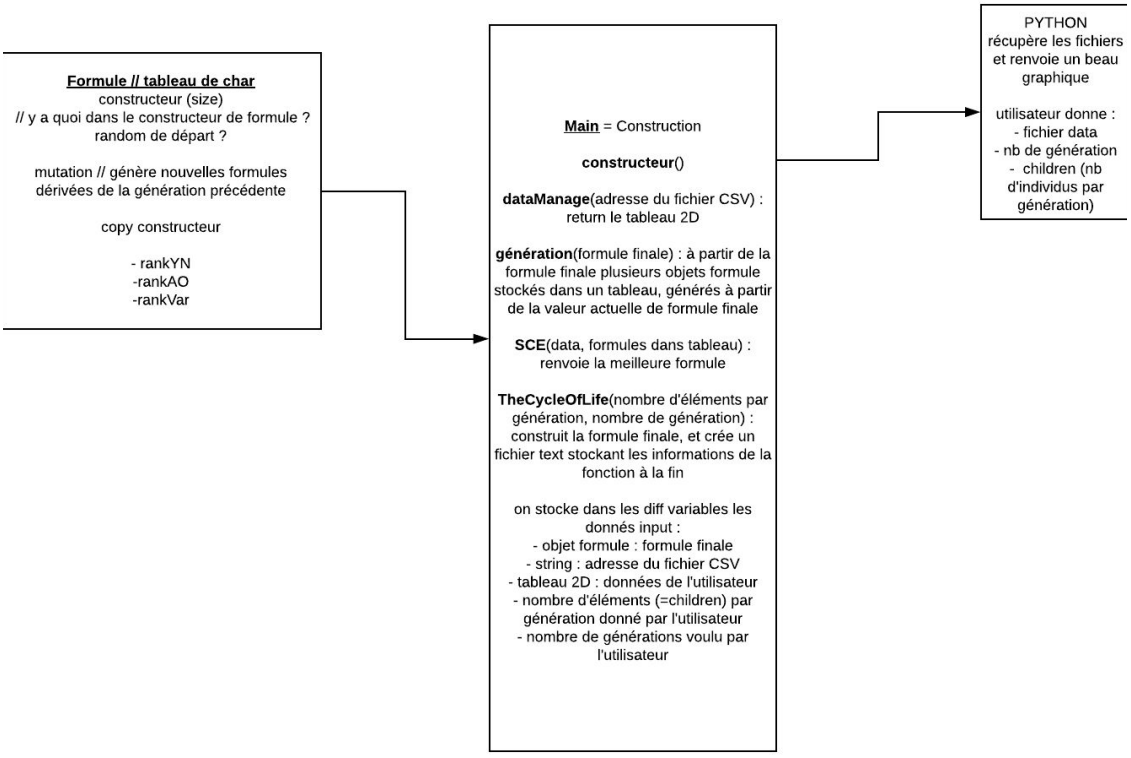
*TO DO :* **SSE, implémenter le code en C++ et python**

# Séance du 18 mai

## Objectif : continuer à structurer le code en classes, réalisation d'un diagramme en UML pour le représenter, début du code

3eme classe : qui fasse l'avancement de la construction de la fonction

Main = Construction




constructeur()

dataManage(adresse du fichier CSV) : return le tableau 2D

génération(formule finale) : à partir de la formule finale plusieurs objets formule stockés dans un tableau, générés à partir de la valeur actuelle de formule finale

SCE(data, formules dans tableau) : renvoie la meilleure

TheCycleOfLife(nombre d'éléments par génération, nombre de génération) : construit la formule finale, et crée un fichier text stockant les informations de la fonction à la fin

- objet formule : formule finale
- string : adresse du fichier CSV

- tableau 2D : données de l'utilisateur
- nombre d'éléments (=children) par génération donné par l'utilisateur
- nombre de générations voulu par l'utilisateur

*Ce qu'est sensé faire le programme :*

Dans le fichier Python, l'utilisateur note le nombre de générations voulu, le nombre d'individus par génération, l'adresse du fichier .CSV.
Il fournit ces arguments au fichier C++ construction.cpp qui doit alors créer la classe construction à partir de son constructeur.

La classe a dans ses attributs de quoi stocker les informations fournies par l'utilisateur (adresse, nbr générations, nbr individus par génération). Elle a aussi les méthodes generation (capable de générer un tableau de formules mutées à partir de la formule initiale et de les classer pour n'en garder que la meilleure qui est alors stockée dans l'attribut formule de la classe construction, la mutation est en parallèle stockée dans historique → dans la classe formule, il a 3 attributs qui stockent la position, le type et le rang de l'échange (-1 si pas d'échange). Et l'attribut historique de la classe construction stocke les 3 valeurs à l'issue de chaque génération dans un tableau de données (3 à chaque fois)) et SSE (capable de prendre un tableau de formules et de return celle avec le meilleur score, utilisée par generation)

Par son constructeur, construction stocke d'abord ces données là où il faut (aux bons attributs).
Ensuite la méthode dataManage permet de prendre l'adresse du fichier .CSV, d'extraire les données qui y sont et de les stocker dans l'attribut tableau sous forme de tableau 2D. Il y a ensuite initialisation de myFormule, objet de type formule.
La classe formule est capable de générer une formule aléatoire à partir d'un jeu de données (ou juste au hasard ?). Elle a aussi une méthode permettant de créer une mutation sur l'objet en cours.

La méthode theCycleOfLife est alors lancée : pour le nombre de générations spécifié, elle prend la myFormule actuelle, y applique generation qui détermine le meilleure individu avec SSE et le stocke en tant que nouveau myFormule et stocke la mutation réalisée dans l'historique.
La fin de l'action du constructeur est de créer un fichier contenant les informations concernant myFormule (permettant de la reformer ailleurs).

On revient alors au fichier Python dont la ligne suivante est de récupérer ce fichier nouvellement formé et de représenter la formule qui s'y trouve.

**TO DO : continuer l'implémentation en C++ et python selon la répartition établie dans l'équipe**

# Séance du 02 juin

## Objectif : coder les différentes méthodes et les tester, commencer la documentation

Notre code limite les possibilité de fonctions reliant les points à :
X1 opération X2 opération X3 …

On doit faire en sorte de palier à ça, en :

- Faisant en sorte qu'une colonne (= un gène) puisse apparaître plusieurs fois modifiant la formule pour qu'elle ait une taille variable (et non être de même taille que le nombre de gènes : elle peut être plus petite ou plus grande) [fait]
- Faisant en sorte qu'une colonne puisse ou pas ne pas apparaître (modifier mutation pour qu'il y ait des insertions ou délétions de gènes dans la formule)

Il faut coder une méthode prédiction() dans Fonction avec le code qui lit la prédiction faite par la fonction (prendre le code qui est dans SSE pour l'instant).
SSE doit juste prendre la prédiction et la comparer au tableau 2D puis renvoyer un score pour chaque ligne, et pour chaque formule la SSE est la somme de toutes les lignes. BEST SSE correspond à la plus basse !

En sortie : on peut récupérer la fonction sous format string et utiliser la librairie python sympy pour la simplifier.

Pour la documentation : utiliser Doxygen
=>https://www.doxygen.nl/manual/docblocks.html

Présentation : 10 minutes de démo (mini tuto de ce qui peut être fait) et  minutes de questions

**Récap de ce qu'il faudrait avoir à la fin :**

- dans fonction.cpp, fonction doit être de taille variable (on peut passer en argument la taille souhaitée) et génère une formule aléatoire
  -> Attention un gène PEUT apparaître plusieurs fois dans cette formule aléatoire (il peut garder le même identifiant)
- dans fonction.cpp modifier mutations pour qu'on ait :
  - des insertions : un gène peut s'ajouter dans formule (attention en fonction de sa position dans la formule il faut mettre un YN et/ou AO)
  - des délétions : un gène est retiré de formule (donc son YN et son AO sont retirés aussi sauf si le gène est en dernière position de la formule)
- dans fonction.cpp mettre une méthode prédiction qui renvoie un tableau de int (même structure que formule actuellement c-a-d YN Var AO)

- dans construction.cpp : génération prend en argument une formule. Génération construit un tableau avec plusieurs enfants qui sont des copies de formule, les enfants sont mutés, puis on donne ce tableau à prédiction (qui renvoie un tableau avec les prédictions pour chaque ligne de chaque enfant) le tableau de prédiction est donné à SSE (qui renvoie LE meilleur enfant, c-a-d SSE le plus bas), génération renvoie best_formule.
- dans construction.cpp : the Cycle of Life fait appel à génération et lui donne la formule de départ, génération fait son taff, renvoie best_formule, best_formule est donné à génération etc. Quand le cycle est fini on a LA best_formule résultant de toutes les générations.

*TO DO* **: continuer l'implémentation en C++ et python en travaillant à plusieurs sur les parties restantes, trouver une solution aux problèmes de gestion de mémoire**

# Séance du 08 juin

Objectif : coder les différentes méthodes et les tester, poursuivre la documentation, préparer une présentation au client

**UPDATE :**

- dans fonction.cpp : formule_ est désormais un attribut de la classe fonction (contient le tableau avec la formule YN Var AO)
- prediction est dans construction.cpp (sinon je voyais pas comment faire, y a plein d'attributs qu'on a besoin qui sont dans construction, par ex nb_ligtab2D etc)
- nouveaux attributs dans construction.cpp : storage_ et predict_
- **prediction** nous fait les memes caprices que SSE avant, donc pour l'instant **ça ne marche pas comme prévu**
- la formule est maintenant générée avec une taille (sizef_) aléatoire, n_ est seulement le nombre de variables existantes dans lesquelles on "pioche" pour créer notre formule

## TO-DO LIST :

- Nombre d'itérations (il faudra définir un max d'itérations et une valeur par défaut). [Done]
- Forme du rendu du délivrable : le logiciel fonctionnel permettant de retourner une formule optimale à partir des paramètres entrés par l'utilisateur soit le nombre de générations voulues, le nombre d'enfants par génération et un tableau .CSV donné en entrée. [Done]
- Si on stocke les valeurs de SSE obtenues au fur et à mesure, on peut tracer un graphe avec la vitesse d'obtention de la meilleure formule, comme ça l'utilisateur sait s'il n'a pas donné assez de génération en entrée. [Done]