**Compte-rendu projet LO21**

I. Objectif du projet :

Le projet a pour objectif la définition et la manipulation des types abstraits de données « Individu » et « Population ». Un individu est représenté par une suite de bits et une population est une suite d’Individus. Il s’agit d’une version simplifiée d’algorithmes génétiques.

II. Choix de conception et d’implémentation des structures de données et démarche :

Tout d’abord, notre projet contient 4 nouveaux types abstraits qui sont les suivants :  
 - Bit  
 - individu  
 - population  
 - param\_qualite

Les types « Bit », « individu » et « population » sont des listes chaînées, dont la structure est imposée par le projet. « Bit » désigne un charactère qui peut prendre la valeur ‘0’ ou ‘1’, « individu » est une liste de bit, qui pointe vers une structure « bit », et « population » est une liste d’individu, qui pointe vers une structure « personne ». Ces types ont donc la structure suivante :

unsigned char Bit

bit :  
 Bit valeur  
 individu suivant  
bit\* individu

personne :  
 individu valeur  
 population suivant  
personne\* population

Le type param\_qualite est un type abstrait que nous avons introduit. Il permet de regrouper la valeur de tous les paramètres dont dépend le calcul de la qualité d’un individu. Ce sont des paramètres qui sont demandés à l’utilisateur dès le lancement du programme. Nous l’avons introduit afin de réduire le nombre d’arguments qui doivent être passés aux différents sous-programme ayant besoin de ces paramètres. La structure de ce type est le suivante :

param\_qualite :  
 char type\_fonction  
 double A  
 double B

type\_fonction est un caractère qui permet de décider la fonction mathématique utilisé pour calculer la qualité d’un individu. Il peut prendre les valeurs suivantes :  
 - ‘s’ : fonction carré  
 - ‘l’ : fonction logarithme népérien  
 - ‘c’ : fonction cosinus  
A et B sont deux réels permettant de paramétrer le résultat.  
La qualité est donnée par : f((x/2^longIndiv)\*(B - A) + A), avec f la fonction utilisée, et longIndiv le nombre de bits constituées par l’individu en question.

Afin d’organiser le projet au mieux, nous avons réunis les structures de données dans un dossier unique, appelé « structures », contenant les quatre nouveaux types dans trois fichiers distincts (le type « Bit » est dans le même fichier que le type « individu »). Ces fichiers sont des header, ce qui nous permet de les inclure facilement dans des fonctions de notre code. Ceci a aussi été fait un souci de réutilisation de ces types dans des projets ultérieurs.

III. Algorithmes des sous-programmes :

Nous avons écrit les algorithmes des sous-programmes permettant de réaliser toutes les opérations nécessaires afin de simuler la fluctuation de la population au fil des générations. Afin de distinguer les variables et les types, nous mettrons les types en majuscule et les variables en minuscule. Ces fonctions sont les suivantes :

1) Type abstrait « Individu »

Fonction ENTIER longueur\_indiv

Compte le nombre de bits que contient un individu et renvoie le résultat.

Lexique :

- suc() : élément suivant de la liste

- est\_vide() : teste si la liste est vide

Donnée :

- INDIVIDU personne

Début :

ENTIER nombre = 0

Tant que non est\_vide(personne) faire

nombre = nombre + 1

personne = suc(personne)

Fin tant que

longueur\_indiv <- nombre

Fin

Fonction INDIVIDU ajouter\_indiv\_q

Ajoute un bit à un individu en fin de liste et renvoie le nouvel individu.

Lexique :

- suc() : élément suivant de la liste

- longueur\_indiv() : longueur de la liste en paramètre

- val() : valeur du bit pointé dans une liste de bits (individu)  
 - creer\_liste() : alloue de l’espace dans la mémoire pour stocker une liste de bits

Données :

- INDIVIDU personne

- BIT valeur

Début :

INDIVIDU nouv\_bit = creer\_liste()

val(nouv\_bit) = valeur

suc(nouv\_bit) = indéfini

Si longueur\_indiv(personne) = 0 alors

personne = nouv\_bit

Sinon

INDIVIDU temp = personne  
 ENTIER i

Pour i de longueur\_indiv(personne) à 2 par -1 faire

temp = suc(temp)

Fin Pour

suc(temp) = nouv\_bit

Fin Si

ajouter\_indiv\_q <- personne

Fin

Fonction INDIVIDU croiser\_indiv

Créé un nouvel individu obtenu en croisant deux individus donnés selon une probabilité pCroise (0 <= pCroise <= 1) et renvoie le nouvel individu.

Lexique :

- suc() : élément suivant de la liste

- longueur\_indiv() : longueur de la liste en paramètre

- val() : valeur du bit pointé dans une liste de bits (individu)

- ajouter\_indiv\_q() : ajoute un bit en queue de la liste de bits d’un individu

- creer\_liste() : créer une liste vide

- rand() : génère un nombre réel aléatoire compris entre 0 et 1

Données :

- INDIVIDU personne1

- INDIVIDU personne2

- REEL pCroise

Début :

INDIVIDU nouvel\_indiv = creer\_liste()

ENTIER i

Pour i de longueur\_indiv(personne1) à 1 par -1 faire

Si rand() <= pCroise alors

nouvel\_indiv = ajouter\_indiv\_q(nouvel\_indiv, val(personne1))

Sinon

nouvel\_indiv = ajouter\_indiv\_q(nouvel\_indiv, val(personne2))

Fin Si

personne1 = suc(personne1)

personne2 = suc(personne2)

Fin pour

croiser\_indiv <- nouvel\_indiv

Fin

Fonction ENTIER decoder\_indiv

Calcule la valeur décimale de la liste de bits d’une individu et renvoie le résultat.

Lexique :

- suc() : élément suivant de la liste

- longueur\_indiv() : longueur de la liste en paramètre

- val() : valeur du bit pointé dans une liste de bits (individu)

Donnée :

- INDIVIDU personne

Début :

ENTIER somme = 0

ENTIER i

Pour i de (longueur\_indiv(personne) - 1) à 0 par -1 faire

somme = somme + (val(personne)) \* 2^i

personne = suc(personne)

Fin Pour

decoder\_indiv <- somme

Fin

Fonction INDIVIDU initialiser\_indiv\_iter

Initialise de manière itérative un individu comportant une nombre de bits donné, qui sont générés aléatoirement.

Lexique :

- ajouter\_indiv\_q() : ajoute un bit en queue de la liste de bits d’un individu

- rand\_bit() : génère un nombre aléatoire, qui peut être soit 0, soit 1

- creer\_liste() : créer une liste vide

Donnée :

- ENTIER longIndiv : correspond au nombre de bit que contiendra l’individu

Début :

INDIVIDU personne = creer\_liste()

ENTIER i

Pour i de 1 à longIndiv par 1 faire

personne = ajouter\_indiv\_q(personne, rand\_bit())

Fin Pour

initialiser\_indiv\_iter <- personne

Fin

Fonction INDIVIDU initialiser\_indiv\_recur

Initialise de manière récursive un individu comportant une nombre de bits donné, qui sont générés aléatoirement.

Lexique :

- ajouter\_indiv\_q() : ajoute un bit en queue de la liste de bits d’un individu

- rand\_bit() : génère un nombre aléatoire, qui peut être soit 0, soit 1

- creer\_liste() : créer une liste vide

Données :

- ENTIER longIndiv : correspond au nombre de bit que contiendra l’individu

Début :

Si longIndiv = 1 alors

initialiser\_indiv\_recur <- ajouter\_indiv\_q(creer\_liste(), rand\_bit())

Sinon

initialiser\_indiv\_recur <- ajouter\_indiv\_q(initialiser\_indiv\_recur(longIndiv - 1), rand\_bit())

Fin Si

Fin

Fonction REEL qualite\_indiv

Calcule la qualité d'un individu selon sa valeur décimale et renvoie le résultat.

Lexique :

- longueur\_indiv() : longueur de la liste en paramètre

- decoder\_indiv() : donne la valeur décimale de la liste de bit d'un individu

- val\_param() : la valeur du paramètres portant le nom donné

Données :

- INDIVIDU personne

- PARAM\_QUALITE parametres

Début :

REEL qualite

REEL X = (decoder\_indiv(personne) / 2^longueur\_indiv(personne)) \* (val\_param(B) – val\_param(A)) + val\_param(A)

Choisir val\_param(type\_fonction) parmi

's' : qualite = -X^2

'l' : qualite = -log(X)

'c' : qualite = -cos(X)

Sinon rien

Fin Choisir

qualite\_indiv <- qualite

Fin

2) Type abstrait « Population »

Fonction ENTIER longueur\_pop

Compte le nombre d’individus que contient une population donnée et renvoie le résultat.

Lexique :

- est\_vide() : teste si la liste est vide

- suc() : élément suivant de la liste

Données :

- POPULATION personnes

Début :

ENTIER nombre = 0

Tant que non est\_vide(personnes) faire

nombre = nombre +1

personnes = suc(personnes)

Fin Tant que

longueur\_pop <- nombre

Fin

Fonction POPULATION ajouter\_pop\_q

Ajoute un individu à une population

Lexique :

- suc() : élément suivant de la liste

- longueur\_pop() : longueur de la population (nombre d’individus)

- val() : individu de la population

- creer\_liste() : alloue de l’espace dans la mémoire pour stocker une liste d’individus

Données :

- INDIVIDU valeur

- POPULATION personnes

Début :

INDIVIDU nouv\_personne = creer\_liste()

val(nouv\_personne) = valeur

suc(nouv\_personne) = indéfini

Si (longueur\_pop(personnes) = 0) alors

personnes = nouv\_personne

Sinon

POPULATION temp = personnes

ENTIER i

Pour i de longueur\_pop(personnes) à 2 par -1 faire

temp = suc(temp)

Fin Pour

suc(temp) = nouv\_personne

Fin Si

ajouter\_pop\_q <- personnes

Fin

Fonction POPULATION initialiser\_pop\_iter

Initialisation de manière aléatoire d’une liste d’individu de façon itérative

Lexique :

- initialiser\_indiv\_iter() : initialisation d'un individu de manière itérative

- ajouter\_pop\_q() : ajoute un individu à une population en queue de liste

Données :

- ENTIER TaillePop

- ENTIER longIndiv

Début :

POPULATION personnes = indéfini

ENTIER i

Pour i de 1 à TaillePop par 1 faire

personnes = ajouter\_pop\_q(personnes, initialiser\_indiv\_iter(longIndiv))

Fin Pour

initialiser\_pop\_iter <- personnes

Fin

Fonction POPULATION initialiser\_pop\_recur

Initialisation de manière aléatoire d’une liste d’individu de façon récursive

Lexique :

- initialiser\_indiv\_recur() : initialisation d'un individu de manière récursive

- ajouter\_pop\_q() : ajoute un individu à une population en queue de liste

Données :

- ENTIER TaillePop

- ENTIER longIndiv

Début :

Si TaillePop = 1 alors

initialiser\_pop\_recur <- ajouter\_pop\_q(indéfini, initialiser\_indiv\_recur(longIndiv))

Sinon

initialiser\_pop\_recur <- ajouter\_pop\_q(initialiser\_pop\_recur(TaillePop - 1, longIndiv), initialiser\_indiv\_recur(longIndiv))

Fin Si

Fin

Fonction POPULATION habitant\_num

Renvoie l’habitant placé à une position donnée. Attention : la fonction ne vérifie pas au préalable que la position est valide.

Lexique :

- suc() : élément suivant de la liste

Données :

- ENTIER n

- POPULATION personnes

Début :

ENTIER i

Pour i de 1 à n - 1 par 1 faire

personnes = suc(personnes)

Fin Pour

habitant\_num <- personnes

Fin

Fonction POPULATION queue\_pop

Renvoie le dernier habitant d’une population.

Lexique :

- longueur\_pop() : longueur de la population (nombre d’individus)

- habitant\_num() : l'habitant d'une population situé à un certain rang

Données :

- POPULATION personnes

Début :

queue\_pop <- habitant\_num(personnes, longueur\_pop(personnes))

Fin

Fonction POPULATION enchainer\_pop

Enchaîne deux populations données, c’est-à-dire, place la deuxième population en queue de la première.

Lexique :

- suc() : élément suivant de la liste

- longueur\_pop() : longueur de la population (nombre d’individus)

- queue\_pop() : dernier élément de la liste

Données :

- POPULATION p1

- POPULATION p2

Début :

Si (longueur\_pop(p1) = 0) alors

enchainer\_pop <- p2

Sinon

suc(queue\_pop(p1)) = p2

enchainer\_pop <- p1

Fin Si

Fin

Fonction POPULATION trier\_pop

La fonction trie les individus en fonction de leur qualité au moyen de Quicksort (tri rapide composé de deux étapes : Diviser et Régner)

Lexique :

- longueur\_pop() : longueur de la population (nombre d’individus)

- suc() : élément suivant de la liste

- ajouter\_pop\_q() : ajoute un individu à une population en queue de liste

- val() : individu de la population

- qualite\_indiv() : qualité d'un individu

- enchainer\_pop() : enchaîne la première population donnée avec la deuxième

Données :

- POPULATION personnes

- PARAM\_QUALITE parametres

Début :

Si (longueur\_pop(personnes) = 0) alors

trier\_pop = indéfini

Sinon

REEL pivot <- qualite\_indiv(val(personnes), parametres)

POPULATION habitant = suc(personnes)

POPULATION inf\_pivot = indéfini

POPULATION sup\_pivot = indéfini

ENTIER i

Pour i de longueur\_pop(personnes) à 2 par -1 faire

Si (qualite\_indiv(val(habitant), parametres) < pivot)

inf\_pivot = ajouter\_pop\_q(inf\_pivot, val(habitant))

Sinon

sup\_pivot = ajouter\_pop\_q(sup\_pivot, val(habitant))

Fin Si

habitant = suc(habitant)

Fin Pour

inf\_pivot = trier\_pop(inf\_pivot, parametres)

sup\_pivot = trier\_pop(sup\_pivot, parametres)

trier\_pop <- enchainer\_pop(enchainer\_pop(sup\_pivot, ajouter\_pop\_q(indéfini, val(personnes))), inf\_pivot)

Fin Si

Fin

Fonction POPULATION selectionner\_pop

Sélectionne les meilleurs individus d’une population en tronquant la liste et en la complétant par recopie des tselect premiers éléments

Lexique :

- longueur\_pop() : longueur de la population (nombre d’individus)

- suc() : élément suivant de la liste

- ajouter\_pop\_q() : ajoute un individu à une population en queue de liste

- val() : individu de la popualtion

- habitant\_num() : l'habitant d'une population situé à un certain rang

Données :

- POPULATION personnes

- ENTIER tselect

Début :

POPULATION nouv\_pop = indéfini

Si tselect = 0 faire

selectionner\_pop <- nouv\_pop

Sinon

ENTIER nb\_habitant = longueur\_pop(personnes)

ENTIER i

Pour i de 0 à nb\_habitant - 1 par 1 faire

Si i/tselect > 0 alors

nouv\_pop = ajouter\_pop\_q(nouv\_pop, val(habitant\_num(nouv\_pop, i % tselect + 1)))

Sinon

nouv\_pop = ajouter\_pop\_q(nouv\_pop, val(personnes))

Fin Si

personnes = suc(personnes)

Fin Pour

selectionner\_pop <- nouv\_pop

Fin Si

Fin

Fonction POPULATION enlever\_hab

Enlève un habitant de la population, sans le supprimer

Lexique :

- suc() : élément suivant de la liste

- habitant\_num() : l'habitant d'une population situé à un certain rang

Données :

- ENTIER num\_habitant

- POPULATION personnes

Début :

Si num\_habitant = 1 alors

personnes = suc(personnes)

Sinon

suc(habitant\_num(personnes, num\_habitant - 1)) = habitant\_num(personnes, num\_habitant + 1)

Fin Si

enlever\_hab <- personnes

Fin

Fonction POPULATION croiser\_pop

Croise une populations pour donner une nouvelle génération.

Lexique :

- enlever\_hab() : enlève un individu d'une population

- ajouter\_pop\_q() : ajoute un individu à une population en queue de liste

- longueur\_pop() : longueur de la population (nombre d’individus)

- val() : individu de la popualtion

- croiser\_indiv() : naissance d'un enfant avec deux individu

- habitant\_num() : l'habitant d'une population situé à un certain rang

- rand() : génère un nombre entier aléatoire compris entre le premier argument et le deuxième

Données :

- POPULATION personnes

- DOUBLE pCroise

- ENTIER nbCroise

Début :

POPULATION nouvelle\_pop = indéfini

ENTIER i

Pour i de longueur\_pop(personnes)/2 à 1 par -1 faire

INDIVIDU personne1

INDIVIDU personne2

ENTIER num\_hab1 = rand(1, longueur\_pop(personnes))

personne1 = val(habitant\_num(personnes, num\_hab1))

personnes = enlever\_hab(personnes, num\_hab1)

ENTIER num\_hab2 = rand(1, longueur\_pop(personnes))

personne2 = val(habitant\_num(personnes, num\_hab2))

personnes = enlever\_hab(personnes, num\_hab2)

ENTIER j

Pour j de 1 à nbCroise par 1 faire

nouvelle\_pop = ajouter\_pop\_q(nouvelle\_pop, croiser\_indiv(personne1, personne2, pCroise))

Fin Pour

Fin Pour

croiser\_pop <- nouvelle\_pop

Fin

Fonction VIDE simuler\_pop

Débute la simulation de la population en fonction des paramètres donnés et affiche le résultat.

Lexique :

- decoder\_indiv() : donne la valeur décimale de la liste de bit d'un individu

- qualite\_indiv() : qualité d'un individu

- initialiser\_pop() : initialise une population constitué d’un nombre donné d’individus, eux-mêmes constitués d’un nombre donné de bits

- longueur\_pop() : longueur de la population (nombre d’individus)

- croiser\_pop() : Croise les individus de la population pour donner une nouvelle génération

- trier\_pop() : Trie les individus de la population selon leur qualité

- sélectionner\_pop() : Recopie les meilleurs individus dans une population triée

- val() : individu de la popualtion

Données :

- ENTIER longIndiv

- REEL pCroise

- ENTIER TaillePop

- ENTIER tSelect : 0 <= tSelect <= 100

- ENTIER nGen

- ENTIER nbCroise

- PARAM\_QUALITE parametres

Début :

POPULATION pop = initialiser\_pop(TaillePop, longIndiv)

ENTIER i

Pour i de 1 à nGen par 1 faire

pop = selectionner\_pop(trier\_pop(croiser\_pop(pop, pCroise, nbCroise), parametres), tSelect\*longueur\_pop(pop)/100)

Écrire(« Génération », i, « traitée. Individus restants : », longueur\_pop(pop))

Fin Pour

Si longueur\_pop(pop) = 0 alors

Écrire(« La population a été éteinte. »)

Sinon

Écrire(« Il reste », longueur\_pop(pop), « individus dans la population et le meilleur individu a pour valeur », decoder\_indiv(val(pop)), « et a pour qualité », qualite\_indiv(val(pop), parametres))

Fin Si

Fin

IV. Jeux d’essais et commentaires sur les résultats

Une fois notre programme aboutit, nous le testons avec divers valeurs des paramètres, qui sont demandés au lancement du programme afin de voir leur influence sur les résultats, et aussi la cohérence avec le modèle de la vie réelle. Les paramètres demandés sont les suivants :

longIndiv : le nombre de bits constitués par les individus de la population  
pCroise (0 <= pCroise <= 1) : la probabilité qu’un bit soit hérité du premier individu lors du croisement de deux individus  
TaillePop : le nombre d’habitants dans la population  
tSelect (0 <= tSelect <= 100) : le taux de sélection après croisement de la population  
nGen : le nombre de générations qui seront traitées lors de la simulation  
nbCroise : le nombre de fois où deux individus se croisent (et forment d’individus)  
type\_fonction : le caractère définissant la fonction mathématique à utiliser pour calculer la qualité d’un individu  
A : la valeur du paramètre A pour le calcul de la qualité  
B : la valeur du paramètre B pour le calcul de la qualité

Nous réalisons les tests suivants donnant les résultats ci-dessous :

longIndiv = 8  
pCroise = 0.5  
TaillePop = 50  
tSelect = 50  
nGen = 30  
nbCroise = 2  
type\_fonction = ‘s’  
A = -1  
B = 1

Résultat : Il reste 50 individus dans la population et le meilleur individu a pour valeur 127 et a pour qualité -0.0001.

Commentaire : Dans ce cas, nous constatons immédiatement que la qualité est presque maximale. En effet, la qualité est la fonction -X², qui est négative, mais la formule ne permet d’obtenir la valeur maximale que pour la valeur de l’individu correspondant à 256/2, qui est 128 ici, sur 8 bits.

longIndiv = 8  
pCroise = 0.5  
TaillePop = 20  
tSelect = 90  
nGen = 5  
nbCroise = 2  
type\_fonction = ‘s’  
A = -1  
B = 1

Résultat : Il reste 20 individus dans la population et le meilleur individu a pour valeur 115 et a pour qualité -0.0103.

Commentaire : Ici, étant donné le nombre faible de générations traitées, du nombre élevé de tSelect, et de la taille très petite de la population, nous nous attendons à une faible fluctuation de la qualité au fil des générations, et donc à une qualité ayant une faible probabilité d’atteindre la valeur maximale, ce qui est le cas ici.

longIndiv = 8  
pCroise = 0.5  
TaillePop = 20  
tSelect = 15  
nGen = 5  
nbCroise = 2  
type\_fonction = ‘s’  
A = -1  
B = 1

Résultat : Il reste 20 individus dans la population et le meilleur individu a pour valeur 127 et a pour qualité -0.0001.

Commentaire : Ici, nous avons uniquement changé le tSelect, qui est de 15% à présent. Ceci a un fort impact sur le résultat, qui est identique au premier test, qui a été fait avec une population plus grande, et sur bien plus de générations.

longIndiv = 16  
pCroise = 0.5  
TaillePop = 50  
tSelect = 50  
nGen = 30  
nbCroise = 2  
type\_fonction = ‘l’  
A = 0.1  
B = 5

Résultat : Il reste 50 individus dans la population et le meilleur individu a pour valeur 4 et a pour qualité 2.2996.

Commentaire : Dans ce cas, nous avons repris les même paramètres que dans le premier test sur la population, mais nous avons changé les paramètres permettant de calculer la qualité : nous avons remplacé la fonction carré (‘s’), par la fonction logarithme népérien (‘l’), nous avons pris des individus représentés par 16 bits au lieu de 8, et les paramètres A et B sont différents. Nous constatons que nous sommes plus éloigné de la valeur maximale de la qualité, qui est de 0 ici (donnant une qualité de 2.3026).

longIndiv = 16  
pCroise = 0.5  
TaillePop = 200  
tSelect = 50  
nGen = 30  
nbCroise = 2  
type\_fonction = ‘l’  
A = 0.1  
B = 5

Résultat : Il reste 50 individus dans la population et le meilleur individu a pour valeur 0 et a pour qualité 2.3026.

Commentaire : Dans ce cas, nous avons simplement fait le test sur une population quatre fois plus grande que précédemment. Il y a ainsi plus de chance que l’individu possédant la meilleur qualité soit présent dans la population, et nous obtenons comme attendu, une meilleur qualité, qui est maximale ici.

longIndiv = 32  
pCroise = 0.5  
TaillePop = 100  
tSelect = 60  
nGen = 50  
nbCroise = 2  
type\_fonction = ‘c’  
A = -3.14159265359  
B = 3.14159265359

Résultat : Il reste 100 individus dans la population et le meilleur individu a pour valeur 416 et a pour qualité 1.0000.

Commentaire : Dans ce cas, nous prenons la fonction cosinus avec un nombre assez grand de générations. Nous constatons que la valeur semble, à première vue, maximale, mais ceci est en réalité du aux erreur d’arrondissement. En effet, la valeur maximale n’est normalement atteinte que lorsque la valeur de l’individu vaut 0 ou 2^32. Cependant, 416 est négligeable par rapport à 2^32, donc nous obtenons le même résultat que si la valeur était de 0. Nous constatons que nous sommes tout de même très loin de la valeur donnant la qualité maximale malgré un nombre important de générations et ceci est dû au nombre important de bits qui réduit les chances d’obtention de la meilleur combinaison.

longIndiv = 32  
pCroise = 0.5  
TaillePop = 50  
tSelect = 75  
nGen = 25  
nbCroise = 2  
type\_fonction = ‘c’  
A = -3.14159265359  
B = 3.14159265359

Résultat : Il reste 100 individus dans la population et le meilleur individu a pour valeur 32885 et a pour qualité 1.0000.

Commentaire : Dans ce cas, nous constatons que malgré les paramètres favorisant une faible fluctuation, nous atteignons une valeur s’apparentant à celle qui est maximale.

longIndiv = 32  
pCroise = 0.5  
TaillePop = 10  
tSelect = 75  
nGen = 15  
nbCroise = 3  
type\_fonction = ‘c’  
A = -3.14159265359  
B = 3.14159265359

Résultat : Il reste 3702 individus dans la population et le meilleur individu a pour valeur 606464 et a pour qualité 1.0000.

Commentaire : Dans ce cas, nous testons notre programme avec une population en croissance, c’est-à-dire, une moyenne supérieur à 2 enfants par couple. Nous le testons avec 3 enfants par couple. Nous constatons que malgré le nombre élevé d’habitants à la fin, il résulte un nombre très éloigné de 0, donnant la qualité maximale, mais toujours négligeable par rapport à 2^32. Mais ceci est en réalité causé par le faible nombre d’habitants au début qui défavorise la diversité par la suite. Par ailleurs, le programme a pris environ 1 minute et 20 secondes pour donner le résultat.

Conclusion des essais : Nous pouvons ainsi tirer plusieurs conclusions :

- Plus la longueur d’un individu est élevé, moins il y a de chances d’obtenir la meilleure qualité  
- Plus la population est grande au départ, plus il y a de chances d’obtenir la meilleure qualité  
- Plus la sélection est large (tselect est grand), moins il y a de chances d’obtenir la meilleur qualité  
- Plus le nombre de génération est grande, plus il y a de chances d’obtenir la meilleur qualité  
- La nature de la fonction utilisé joue un rôle en combinaison avec la longueur de la population

V. Conclusion

Pour conclure, le modèle que nous donnons avec notre programme de l’évolution des gènes portés par les individus dans une population est globalement satisfaisante. Nous pouvons voir que l’influence des paramètres dans l’algorithme sont cohérentes avec la vie réelle. Nous pouvons tout de même pointer une grande limite, qui est l’absence du genre des individus pour faire des croisements compatibles.