

Informatique C, Master 2 SISEA

BOUSSOT Valentin

2 décembre 2020

Table des matières

1 TP1	2
1.1 Problème 1	2
1.1.1 Objectif	2
1.1.2 Génération de nombres aléatoires	2
1.1.3 Fonctionnement et contrôle du générateur de nombres pseudo-aléatoires (PRNG)	2
1.1.4 Modification du support de la variable aléatoire générée	3
1.1.5 Programme naïf	4
1.1.6 Saisie d'un nombre par la console	4
1.1.7 Fonctionnement de la fonction scanf	4
1.1.8 Programme final	5
1.1.9 Exécution du programme	6
1.2 Problème 2	7
1.2.1 Objectifs	7
1.2.2 Formalisation des fonctions en langage C	7
1.2.3 Génération d'un tableau de couples ($x, f(x)$)	7
1.2.4 Enregistrement et lecture du tableau de couples ($x, f(x)$)	8
1.2.5 Paramétrage du programme par les arguments de la fonction main	9
1.2.6 Main.c	10
1.2.7 Résultats à l'exécution	11
1.3 Problème 3	12
1.3.1 Objectif	12
1.3.2 Implémentation des algorithmes de recherche d'un zéro	12
1.3.3 Main.c	13
1.3.4 Résultats à l'exécution	14
2 TP2	15
2.1 Objectifs	15
2.2 Lire l'entête d'une image au format BMP	15
2.3 Lecture/Ecriture des pixels	15
2.4 Les fonctions de filtrage	16
2.5 Code de la fonction main	17
2.6 Résultats : Images après traitement	18

1 TP1

1.1 Problème 1

1.1.1 Objectif

Créer un programme qui tire aléatoirement un entier sur un intervalle donné et qui demande à l'utilisateur de le trouver, à chaque échec une indication est donnée.

- + si le nombre à trouver est supérieur au nombre donné par l'utilisateur
- si le nombre à trouver est inférieur au nombre donné par l'utilisateur

1.1.2 Génération de nombres aléatoires

La première étape est de générer un nombre aléatoire.

1.1.3 Fonctionnement et contrôle du générateur de nombres pseudo-aléatoires (PRNG)

La bibliothèque standard de C implémente un algorithme de génération pseudo-aléatoire, qui produit une suite de nombres aléatoires **I.I.D** de loi uniforme sur le support : $[0, \text{RAND_MAX}]$, la qualité de cette génération n'est pas critique. D'après la documentation la valeur de **RAND_MAX** est au minimum égal à 32767.

Cette suite découle de façon déterministe d'une valeur initiale (**seed**). Deux seeds différentes impliquent deux suites différentes de nombres aléatoires.

Chaque appel de la fonction **int rand(void)** dont le prototype est donné dans le fichier **stdlib.h** nous renvoie l'élément suivant de la liste de nombres aléatoires générée par la seed. Par défaut cette seed est fixée à 1.

Dans notre programme la fonction **rand()** est appelée une fois et nous souhaitons générer un nouveau nombre aléatoire à chaque exécution.

Dans l'état **rand()** va nous renvoyer sans cesse le premier élément de cette suite il est donc nécessaire de générer de nouvelles suites, il faut donc changer de seed à chaque exécution.

Pour cela on nous propose d'utiliser le temps de la machine via la fonction

time_t time(time_t *) dont le prototype est donné dans le fichier **time.h**. Chaque appel permet d'obtenir soit par valeur de retour, soit en passant un paramètre par pointeur le temps écoulé depuis le premier janvier 1970 à 00 :00 :00 exprimé en secondes.

La fonction **void srand(unsigned int)** dont le prototype est donné dans le fichier **stdlib.h** va nous permettre de modifier cette seed.

À noter également qu'après la modification de la seed la fonction **rand()** renvoie le premier élément de la liste associé à la seed.

Listing 1 – Vérification des propriétés du PRNG

```

1 int n1, n2;
2 n1 = rand();
3 n2 = rand();
4 assert(n1 != n2);
5 //Vérifie que rand renvoie un élément différent de la suite aléatoire
6 //et qu'il est donc très rare d'avoir l'égalité de deux nombres.
7
8 srand(1);
9 n2 = rand();
10 assert(n1 == n2);
11 //Vérifie que la seed par défaut est 1 et que l'appel de rand après la
12 //modification de la seed renvoie le premier élément de la suite aléatoire
13
14 srand(2);
15 n2 = rand();
16 assert(n1 != n2);
17 //Vérifie qu'une seed différente donne des suites différentes =>
18 //le premier élément est probablement différent.

```

La seule contrainte est de changer de seed à chaque exécution. Utilisé le temps de la machine en seconde pour la seed est une solution simple et pertinente ce qui nous permet potentiellement de générer un nombre pseudo-aléatoire toutes les secondes.

1.1.4 Modification du support de la variable aléatoire générée

On va réduire le domaine de recherche du nombre entre NMIN inclue et NMAX inclue : n est un élément de la suite aléatoire $n \in [0, \text{RAND_MAX}]$.

Solution 1 (Proposée)

Soit b le reste de la division euclidienne : $n \equiv b \pmod{NMAX - NMIN + 1} \Rightarrow b \in [0, NMAX - NMIN]$ en ajoutant $b' = b + NMIN, b' \in [NMIN, NMAX]$

Il est claire que cette mise à l'échelle ajoute un biais quand $NMAX-NMIN+1$ ne divise pas RAND_MAX.

Solution 2

Mettre à l'échelle la variable aléatoire de cette façon :

$$\phi(n) = \frac{n}{\text{RAND_MAX}} \cdot (NMAX - NMIN) + NMIN$$

Cette solution n'introduit pas de biais (Numériquement ?) et concerne la loi de la V.A.

Solution 3

S'il est possible de contraindre dans le programme $(NMAX - NMIN + 1)\%2 == 0$ alors il suffit de prendre $2^b = NMAX - NMIN + 1$, b-1 bits qui codent le nombre n.

Vis-à-vis des intervalles souhaités la solution 2 n'impose pas de contrainte et la solution 1 est plus souple que la solution 3.

Du point de vue des performances, c'est difficile à dire je pense que les deux premières solutions se valent (Dépend du contexte), la 3^e est certainement la plus rapide.

Dans notre programme on va utiliser la 3e solution avec un intervalle $[0, 127]$

Soit le code C suivant :

Listing 2 – Génération d'une variable aléatoire discrète de loi uniforme

```

1 assert(NMAX-NMIN+1 % 2 && NMAX-NMIN+1 < RAND_MAX); //Vérification de la contrainte sur l'intervalle
2 srand(time(NULL));
3 int randomNumber = (rand() & (NMAX-NMIN))+NMIN;

```

NMAX et NMIN sont deux macros du fichier d'entête **exe_1.h**

1.1.5 Programme naif

Listing 3 – Programme naif

```
1 #include "exe_1.h" //Define NMIN NMAX
2 #include <stdio.h> //Define printf
3 #include <stdlib.h> //Define EXIT_SUCCESS rand srand
4 #include <time.h> //Define time
5 //#include<assert.h> Pour les tests
6
7 int main(int argc, char **argv){
8     //Génération d'une variable aléatoire discrète de loi uniforme sur le support [NMIN NMAX]
9     #ifdef TEST
10         assert((NMAX-NMIN+1 % 2 && NMAX-NMIN+1 < RAND_MAX)); //Vérification de la contrainte sur l'intervalle
11     #endif
12     srand(time(NULL));
13     int randomNumber = (rand() & (NMAX-NMIN))+NMIN;
14     int step = 0; //Comptabilise le nombre d'étapes nécessaires pour trouver le nombre
15     int number;
16     // Exécuté tant que le joueur n'a pas trouvé randomNumber
17     while(1){
18         printf("Number [[%d, %d]]?\n", NMIN, NMAX);
19         scanf("%d", &number);
20         step++;
21         if(randomNumber != number)
22             printf("%s\n", number > randomNumber ? "-" : "+"); //Indication
23         else break;
24     }
25     printf("Win !!! Steps n°%d\n", step);
26     return EXIT_SUCCESS;
27 }
```

Note :

Le while(1) et le break permet de tester qu'une seule fois (randomNumber != number)
EXIT_SUCCESS et EXIT_FAILURE deux macro de l'entête **stdlib.h** en valeur de retour de la fonction main permet de rajouté un peu de sémantique au code.

Pour donner les indications à l'utilisateur, une condition ternaire me semble une solution plus élégante.

Ce programme n'est pas correct car l'utilisateur peut mettre le programme dans un état où la boucle s'exécute à indéfiniment.

1.1.6 Saisie d'un nombre par la console

On nous impose d'utiliser la fonction **int scanf(char *, ...)** de la bibliothèque standard présente dans le fichier d'entête **stdio.h**.

Note : L'argument muet ... correspond à une liste variable de paramètres de type pointeur (fonctionnalité de la bibliothèque standard).

Un entier compris entre NMIN et NMAX doit être demandé à l'utilisateur.

1.1.7 Fonctionnement de la fonction scanf

La fonction scanf à deux comportements, elle fonctionne conjointement avec un tableau de caractères (Buffer)

Son premier rôle est de demander à l'entrée standard une chaîne de caractère pour la copier dans son buffer si celui-ci est vide.

Son deuxième rôle est de lire le buffer et d'interpréter les caractères selon le format afin de convertir et d'affecter les valeurs dans les arguments.

Tous les caractères correctement interprétés sont retirés du buffer.

Scarf s'arrête quand elle a fini de parcourir le buffer ou lorsqu'une entrée ne correspond pas aux spécifications de format.

Sécurisé la saisie :

Sur la console l'utilisateur peut taper n'importe quoi. Il faut prendre en compte tous les cas possibles pour sécuriser la saisie et ne pas avoir de surprise.

Le cas problématique :

Quand scanf ne peut pas interpréter l'entrée elle ne supprime pas les caractères problématique et n'affecte pas de valeur, en somme l'exécution de la même instruction scanf ne modifie plus l'état du programme et dans notre cas on va boucler.

Pour prendre en compte ce cas il faut d'abord avoir un moyen de le détecter. Il est prévu que scanf renvoie le nombre de paramètres qu'elle a été capable d'interpréter, on s'attend dans notre cas qu'elle nous renvoie 1 quand tous c'est bien passer et 0 sinon. Pour rectifier ce comportement il faut retirer les caractères problématiques du buffer.

D'après la documentation, la fonction **int getchar(void)** présent dans le fichier d'entête **stdio.h** permet de récupérer un caractère sur l'entrer standard, elle retire du buffer le caractère lu.

Les entiers étant séparés par des espaces il suffit d'itérer getchar jusqu'à trouver un caractère espace ou le caractère '\n'.

Soit la fonction suivante :

Listing 4 – Fonction getSafeNumber(int *, const int, const int)

```
1 void getSafeNumber(int *number, const int NMIN, const int NMAX){  
2     printf("Number [%d, %d]\n", NMIN, NMAX);  
3     do{  
4         while(!scanf("%d", number)){  
5             char c;  
6             while((c = getchar()) != ' ' && c != '\n'); //Supprime les caractères problématiques  
7         }  
8     } while(*number < NMIN || *number > NMAX); //Boucle scanf si le nombre récupéré n'est pas dans l'intervalle  
9 }
```

J'en profite pour ajouté une fonctionnalité : Quand le nombre récupérer n'est pas dans l'intervalle on rappelle scanf.

1.1.8 Programme final

Listing 5 – Programme final

```
1 #include "exe_1.h" //Define NMIN NMAX  
2 #include "Number.h" //Define getSafeNumber  
3 #include <stdio.h> //Define printf  
4 #include <stdlib.h> //Define EXIT_SUCCESS rand srand  
5 #include <time.h> //Define time  
6 //#include<assert.h> Pour les tests  
7  
8 int main(int argc, char **argv){  
9     /*Génération d'une variable aléatoire discrète de loi uniforme sur le support [NMIN NMAX]  
10    #ifdef TEST  
11        assert(NMAX-NMIN+1 % 2 && NMAX-NMIN+1 < RAND_MAX); //Vérification de la contrainte sur l'intervalle  
12    #endif  
13    srand(time(NULL));  
14    int randomNumber = (rand() & (NMAX-NMIN)) + NMIN;  
15    int step = 0; //Comptabilise le nombre d'étapes nécessaires pour trouver le nombre  
16    int number;  
17    //Exécuté tant que le joueur n'a pas trouvé randomNumber  
18    while(1){  
19        getSafeNumber(&number, NMIN, NMAX); //Récupération du nombre  
20        step++;  
21        if(randomNumber != number)  
22            printf("%s\n", number > randomNumber ? "-" : "+"); //Indication  
23        else break;  
24    }  
25    printf("Win !!! Steps %d\n", step);  
26    /*return EXIT_SUCCESS;  
27 }
```

1.1.9 Exécution du programme

```
Number [[0, 127]]? Number [[0, 127]]?
100          100
+
Number [[0, 127]]? Number [[0, 127]]?
120          zed rze
-
Number [[0, 127]]? +
110          Number [[0, 127]]?
-
50 dez 75
Number [[0, 127]]? +
105          Number [[0, 127]]?
+
Number [[0, 127]]? Number [[0, 127]]?
107          60
-
+
Number [[0, 127]]? Number [[0, 127]]?
106          65
Win !!! Steps n°6  Win !!! Steps n°6
```

FIGURE 1 – 2 exemples d'exécution du programme

L'exécution de gauche représente une partie "normal".

L'exécution de droite vérifie que la fonction scanf agit correctement.

1.2 Problème 2

1.2.1 Objectifs

- Générer un tableau qui stocke l'ensemble des couples $(x, f(x))$ pour $x \in [MIN : te : MAX]$.
- Enregistrer ce tableau dans un fichier texte et binaire.
- Lire ce tableau à partir d'un fichier texte et binaire.

Les fonctions au choix sont les suivantes :

$$f(x) = x^2, f(x) = x^3 - 3x^2 - 3x - 35, f(x) = \frac{e^{10x}-1}{e^{10x}+1} - 0.5 \quad f(x) = \cos(x + 3\pi/8)$$

Le choix de la fonction, du paramétrage de MIN, MAX et te sont donnés par l'utilisateur à l'exécution.

1.2.2 Formalisation des fonctions en langage C

Listing 6 – Fonctions

```

1 #include <math.h>
2
3 //Le pointeur de fonction associé aux fonctions R->R
4 typedef double (*Function)(double x);
5
6 double square(double x){
7     return pow(x,2);
8 }
9
10 double polynomial(double x){
11     return pow(x, 3)-3*pow(x, 2)-3*x-35;
12 }
13
14 double exponential(double x){
15     return (exp(10*x)-1)/(exp(10*x)+1)-0.5;
16 }
17
18 double sinusoide(double x){
19     return cos(x+3*M_PI/8);
20 }
```

1.2.3 Génération d'un tableau de couples $(x, f(x))$

La structure de données qui va représenter le couple $(x, f(x))$ est la suivante :

Listing 7 – Structure de donnée

```

1 typedef struct Value Value;
2 struct Value{
3     double x; //Abscisses
4     double y; //f(x)
5 };
```

La structure de données des différents paramètres modifiables par l'utilisateur ;

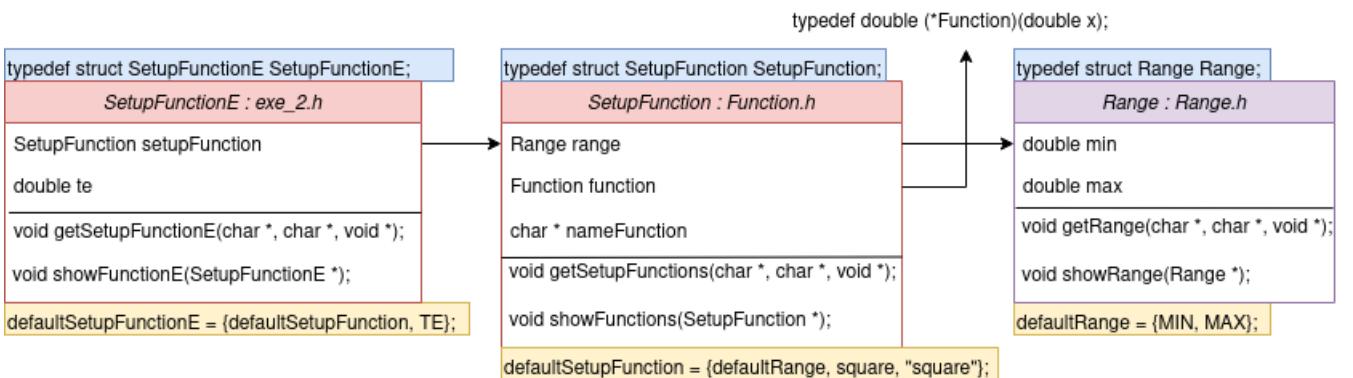


FIGURE 2 – UML de la structure de données des paramètres

Chaque structure est déclarée dans des fichiers d'entête différent, les structures sont initialisées avec des valeurs par défaut. Dans chaque fichier d'entête, deux fonctions sont déclarées, ils vont permettre de modifier/afficher les champs de la structure associée.

Le choix de séparer les champs de données dans des structures différentes est motivé par le fait qu'il n'appartienne pas au même concept, Range modifiera uniquement les bornes, SetupFunction se limitera au choix de la fonction. Spécifiquement dans le problème 2 nous avons besoin du paramètre te qui ne se retrouve pas dans le problème 3, ont pourra alors conservé les deux premières structures et en ajouté d'autres sur le même modèle.

Listing 8 – Définition et remplissage du tableau

```

1 // Nombre de couples (x,y)
2 // Les valeurs des paramètres sont récupérées dans leur structure respective.
3 int number = (setupFunctionE.setupFunction.range.max - setupFunctionE.setupFunction.range.min) / setupFunctionE.te + 1;
4 // Allocation d'un tableau de number éléments de taille sizeof(Value) chacun.
5 Value *values = (Value *) malloc(sizeof(Value)*number);
6
7 int i = 0;
8 for(double x = setupFunctionE.setupFunction.range.min; x <= setupFunctionE.setupFunction.range.max; x+=setupFunctionE.te){
9     Value value = {x, setupFunctionE.setupFunction.function(x)}; // Initialisation d'un nouveau couple
10    values[i++] = value; //Ajout du couple au tableau
11 }
```

Au début du programme la structure setupFunctionE est initialisée.

1.2.4 Enregistrement et lecture du tableau de couples (x,f(x))

L'écriture et la lecture de ce tableau de structure dans les deux formats de fichier ne posent pas de problème particulier.

Listing 9 – Fonctions de lecture/écriture format texte/binaire

```

1 void writeValue(char *nameFile, Value *values, size_t values_size){
2     FILE *file;
3     file = fopen(nameFile, "w"); //Ouverture en écriture en mode texte
4     //Vérification que le fichier peut être créé ou ouvert sur le disque (Permission, Path, ...)
5     if(file != NULL){
6         for(int i = 0; i < values_size; i++){ //Itère tous les éléments du tableau
7             fprintf(file, "%lf\t%lf\n", values[i].x, values[i].y); Écrit dans le fichier avec un formatage
8         }
9         fclose(file); //Libère la ressource
10    } else printf("Error write file");
11 }
12
13 void writeBinaryValue(char *nameFile, Value *values, size_t values_size){
14     FILE *file; //Ouverture en écriture en mode binaire
15     file = fopen(nameFile, "wb");
16     //Vérification que le fichier peut être créé ou ouvert sur le disque (Permission, Path, ...)
17     if(file != NULL){
18         fwrite(values, sizeof(Value), values_size, file); //Ecrit dans le fichier les données du tableau
19         fclose(file); //Libère la ressource
20     } else printf("Error write file");
21 }
22
23
24 void readValue(char *nameFile, Value *values, size_t values_size){
25     FILE *file;
26     file = fopen(nameFile, "r"); //Ouverture en lecture en mode texte
27     //Vérification que le fichier existe et qu'il peut être ouvert sur le disque (Permission, ...)
28     if(file != NULL){
29         for(int i = 0; i < values_size; i++){ //Itère tous les éléments du tableau
30             //Affectation des cellules du tableau avec la ligne formatée
31             fscanf(file, "%lf\t%lf\n", &(values[i].x), &(values[i].y));
32         }
33         fclose(file); //Libère la ressource
34     } else printf("Error read file");
35 }
36
37 void readBinaryValue(char *nameFile, Value *values, size_t values_size){
38     FILE *file;
39     file = fopen(nameFile, "rb"); //Ouverture en lecture en mode binaire
40     //Vérification que le fichier existe et qu'il peut être ouvert sur le disque (Permission, ...)
41     if(file != NULL){
42         //Affectation du tableau avec les données brutes du fichier
43         fread(values, sizeof(Value), values_size, file);
44         fclose(file); //Libère la ressource
45     } else printf("Error read file");
46 }
```

Note d'amélioration : Il y a moyen de factoriser le code au moins pour la gestion de la ressource.

1.2.5 Paramétrage du programme par les arguments de la fonction main

On souhaite laisser l'utilisateur choisir le type de fonction, les bornes et te à l'exécution. Ces informations seront transmises par les arguments de la fonction main.

Il y a deux types d'information à récupérer, les arguments de choix et les arguments pour affecter une valeur, la différence se fera par un double ou simple tiret.

Pour mettre en oeuvre ce système il nous faut tout d'abord parser les arguments en fonction du type d'information.

Listing 10 – Parseur

```

1  /**
2   ---Parseur---
3   char **argv           : Le tableau de chaîne de caractère
4   size_t args_size     : La taille du tableau argv
5   ApplyParam applyParam : Fonction pour appliquer le paramétrage
6   void * setup          : Structure pour stocker le paramétrage
7 */
8   void setParam(char **argv, size_t args_size, ApplyParam applyParam, void * setup){
9       for(int j = 1; j < args_size; j+= 1+*(argv[j]+1) != '-')
10      if(*argv[j] == '-')
11          applyParam(argv[j], j+1 < args_size ? argv[j+1] : 0, setup);
12 }
```

ApplyParam est un pointeur de fonction qui va permettre d'appeler nos fonctions de modification des champs de la structure qui est également en argument.

Pour exemple, la fonction de modification de la structure SetupFunctionE :

Listing 11 – void getSetupFunctionE(char *, char *, void *)

```

1 void getSetupFunctionE(char *key, char *value, void *setupFunctionE){
2     switch(hachage(key)){
3         case HASH_TE: // -te
4             // La valeur vérifie les critères d'un temps d'échantillonnage.
5             if(isNumber(value) && atof(value) > 0) ((SetupFunctionE *)setupFunctionE)->te = atof(value);
6             break;
7         case HASH_HELP:
8             printf("-----HELP-----\nEchantillonage \n\t-te\n");
9             // Pas de break l'help est passé dans les fonctions supérieures.
10            default:
11                getSetupFunctions(key, value, &((SetupFunctionE *)setupFunctionE)->setupFunction));
12                break;
13            }
14        if(hachage(key) == HASH_HELP) exit(EXIT_SUCCESS); // Si la key == '-help' on arrête l'exécution
15    }
```

Si la clé n'est pas dans les cas du switch, la fonction de modification de la structure membre est appelé, jusqu'à remonter à la structure Range.

Un cast explicite sur la structure est nécessaire, c'est le prix de la générnicité en C. (void *).

Listing 12 – long hachage(char *)

```

1 typedef union Hash Hash;
2
3 union Hash{
4     char data[sizeof(long)];
5     long hash;
6 };
7
8 long hachage(char *key){
9     Hash hash;
10    for(int i = 0; i < sizeof(long); i++) hash.data[i] = '\0';
11    for(int i = 0; key[i] != '\0' && i < sizeof(long); i++) hash.data[i] = key[i];
12    return hash.hash;
13 }
```

La fonction hashage appelé avec la clé permet de convertire cette chaîne de caractère dans un type long.

Ce choix technique est motivé premièrement parce qu'il n'est pas possible de comparer des chaînes de caractère dans un switch et deuxièmement, je souhaitais faire une comparaison du type startwith. L'inconvénient c'est qui faut connaître l'équivalent en long de la chaîne à comparé avant compilation ce qui peut être pénible pour ajouté de nouvelle entrée.

1.2.6 Main.c

Listing 13 – Programme final

```

1 #include <stdio.h> //Define printf
2 #include <stdlib.h> //Define EXIT_SUCCESS atof malloc free
3 #include "exe_2.h" //SetupFunctionE getSetupFunctionE showFunctionE showValues
4 #include "Param.h" //hachage setParam
5 #include "Number.h" //isNumber
6
7
8 int main(int argc, char **argv){
9     SetupFunctionE setupFunctionE = defaultSetupFunctionE; //Strucure par défaut
10    //Modification des informations en fonction des arguments de main
11    setParam(argv, argc, getSetupFunctionE, &setupFunctionE);
12    showFunctionE(&setupFunctionE); //Affiche les informations
13
14    //Nombre de couples (x,y)
15    //Les valeurs des paramètres sont récupérées dans leur structure respective.
16    int number = (setupFunctionE.setupFunction.range.max-setupFunctionE.setupFunction.range.min)/setupFunctionE.te+1;
17    Value *values = (Value *) malloc(sizeof(Value)*number);
18
19    int i = 0;
20    for(double x = setupFunctionE.setupFunction.range.min; x <= setupFunctionE.setupFunction.range.max; x+=setupFunctionE.te)
21        Value value = {x, setupFunctionE.setupFunction.function(x)}; //Initialisation d'un nouveau couple
22        values[i++] = value; //Ajout du couple au tableau
23
24    //Ecriture du tableau en texte
25    writeValue("resultsFile/values.dat", values, number);
26    //Ecriture du tableau en binaire
27    writeBinaryValue("resultsFile/values.bin", values, number);
28
29
30    Value *readValues = (Value *) malloc(sizeof(Value)*number);
31    //Lecture du tableau en mode texte
32    readValue("resultsFile/values.dat", readValues, number);
33    //Affiche le résultat de la lecture
34    showValues(readValues, number);
35    Value *readBinaryValues = (Value *) malloc(sizeof(Value)*number);
36    //Lecture du tableau en mode binaire
37    readBinaryValue("resultsFile/values.bin", readBinaryValues, number);
38    //Affiche le résultat de la lecture
39    showValues(readBinaryValues, number);
40    //Evite les fuites de mémoire
41    free(values);
42    free(readValues);
43    free(readBinaryValues);
44    return EXIT_SUCCESS;
45}
46
47//Affichage des couples (x, f(x))
48void showValues(Value *values, size_t values_size){
49    printf("-----Valeurs-----\n");
50    for(int i = 0; i < values_size; i++){
51        printf("%.2lf\t%.2lf\n", values[i].x, values[i].y);
52    }
53}
54
55void getSetupFunctionE(char *key, char *value, void *setupFunctionE){
56    switch(hachage(key)){
57        case HASH_TE: // -te
58            //La valeur vérifie les critères d'un temps d'échantillonnage.
59            if(isNumber(value) && atof(value) > 0) ((SetupFunctionE *)setupFunctionE)->te = atof(value);
60            break;
61        case HASH_HELP:
62            printf("-----HELP-----\nSampling \n\t-te\n");
63            //Pas de break l'help est passé dans les fonctions supérieures.
64        default:
65            getSetupFunctions(key, value, &(((SetupFunctionE *)setupFunctionE)->setupFunction));
66            break;
67    }
68    if(hachage(key) == HASH_HELP) exit(EXIT_SUCCESS); //Si la key == '--help' on arrête l'exécution
69}
70
71
72void showFunctionE(SetupFunctionE *setupFunctionE){
73    printf("-----SETUP-----\n");
74    showFunctions(&(setupFunctionE->setupFunction));
75    printf("Sampling\n\tTe : %lf\n", setupFunctionE->te);
76}

```

1.2.7 Résultats à l'exécution

```

> ./exe_2
-----SETUP-----
Range Min : -10.00 Max : 10.00
Functions Fonction : square
Sampling Te : 1.000000
----Valeurs---
-10.00 100.00
-9.00 81.00
-8.00 64.00
-7.00 49.00
-6.00 36.00
-5.00 25.00
-4.00 16.00
-3.00 9.00
-2.00 4.00
-1.00 1.00
0.00 0.00
1.00 1.00
2.00 4.00
3.00 9.00
4.00 16.00
5.00 25.00
6.00 36.00
7.00 49.00
8.00 64.00
9.00 81.00
10.00 100.00
----Valeurs---
-10.00 100.00
-9.00 81.00
-8.00 64.00
-7.00 49.00
-6.00 36.00
-5.00 25.00
-4.00 16.00
-3.00 9.00
-2.00 4.00
-1.00 1.00
0.00 0.00
1.00 1.00
2.00 4.00
3.00 9.00
4.00 16.00
5.00 25.00
6.00 36.00
7.00 49.00
8.00 64.00
9.00 81.00
10.00 100.00
-----SETUP-----
Range Min : -2.00 Max : 5.00
Functions Fonction : polynomial
Sampling Te : 0.900000
----Valeurs---
-2.00 -49.00
-1.10 -36.66
-0.20 -34.53
0.70 -38.23
1.60 -43.38
2.50 -45.62
3.40 -40.58
4.30 -23.86
----Valeurs---
-2.00 -49.00
-1.10 -36.66
-0.20 -34.53
0.70 -38.23
1.60 -43.38
2.50 -45.62
3.40 -40.58
4.30 -23.86
-----SETUP-----
Range Min : -2.00 Max : -49.00
Functions Fonction : exponential
Sampling Te : 0.900000
----Valeurs---
-2.00 -49.00
-1.10 -36.66
-0.20 -34.53
0.70 -38.23
1.60 -43.38
2.50 -45.62
3.40 -40.58
4.30 -23.86
-----SETUP-----
Range Min : -min -max
Functions --square --polynomial --exponential --sinusoide
Sampling
arch ~/Documents/Etude/M2/Informatique/TP1 master >1 !22 ?37 >
> ./exe_2 --help
-----HELP-----
Sampling
-te
Functions --square --polynomial --exponential --sinusoide
Range --min --max
arch ~/Doc/E/M/Informatique/TP1 master >1 !22 ?37 >
> cat resultsFile/values.dat
2.000000 -49.000000
-1.100000 -36.661000
-0.200000 -34.528000
0.700000 -38.227000
1.600000 -43.384000
2.500000 -45.625000
3.400000 -40.576000
4.300000 -23.863000
arch ~/Doc/E/M/Informatique/TP1 master >1 !22 ?37 >
17:09:38

```

De gauche à droite : Exécution par défaut, affichage du contenu du fichier, Exécution avec modification des paramètres, affichage du contenu du fichier, help du programme.

1.3 Problème 3

1.3.1 Objectif

Trouver les points d'annulation d'une fonction $f(x)$

1.3.2 Implémentation des algorithmes de recherche d'un zéro

La structure suivante permet de stocker le résultat, la complexité, et des variables temporaires. Les champs **a** et **b** correspondent aux bornes de recherche qui à chaque itération sur les 3 algorithmes se réduisent, ils sont initialisés avec le paramètre MIN et MAX de la structure Range.

L'utilité de rassembler ces variables au sein d'une même structure ici est essentiellement de réduire le nombre d'argument des fonctions de recherche de zeros.

Listing 14 – Structure Result

```
1 typedef struct Result Result;
2
3 struct Result{
4     int step;
5     double result;
6     double a,b;
7 };
```

Listing 15 – Implémentation des algorithmes de recherche d'un zéro

```
1 void bisection(Function function , Result *result){
2     double tmp = (result->b-result->a)/2+result->a;
3     if(function(result->a)*function(tmp) < 0) result->b = tmp;
4     else result->a = tmp;
5     result->step++;
6     result->result = tmp;
7 }
8
9 double secante_it(Function function , double x_n, double x_n1){
10    if(function(x_n1)-function(x_n) == 0){
11        printf("Division par zeros\n");
12        exit(EXIT_FAILURE);
13    }
14    return x_n-function(x_n)*(x_n1-x_n)/(function(x_n1)-function(x_n));
15 }
16
17 void secante(Function function , Result *result){
18     double tmp = result->b;
19     result->b = secante_it(function , result->a , result->b);
20     result->a = tmp;
21     result->step++;
22     result->result = tmp;
23 }
24
25 void fausse_position(Function function , Result *result){
26     double tmp = result->b;
27     result->b = secante_it(function , result->a , result->b);
28     if(function(result->b)*function(result->a) < 0) result->a = tmp;
29     result->step++;
30     result->result = tmp;
31 }
```

Ces implémentations résultent des explications du document : AIdeTP1MethodesDeZeroFunction.pdf

Il serait peut-être plus juste de vérifier que $x_n - \frac{f(x_n)*(x_{n1}-x_n)}{(f(x_{n1})-f(x_n))}$ ne dépasse pas une valeur très importante au lieu de vérifié que $f(x_{n1}) - f(x_n) = 0$.

Toujours sur le même principe la structure de données des différents paramètres modifiables par l'utilisateur via les arguments de main :

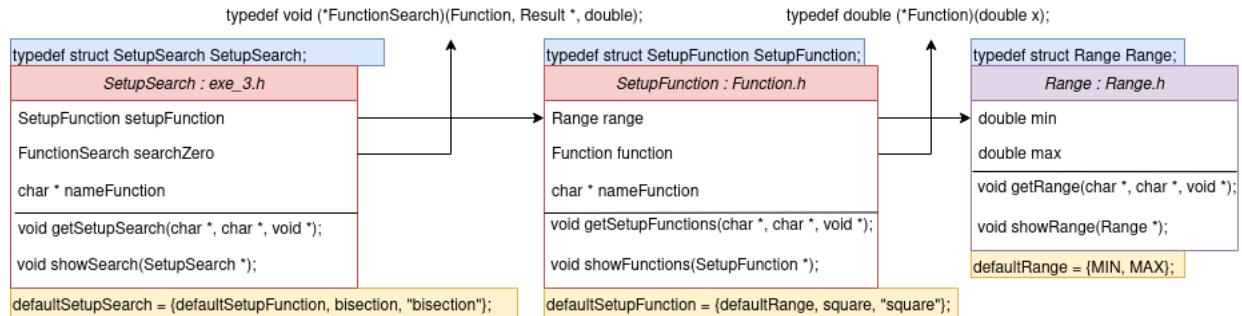


FIGURE 3 – UML de la structure de données des paramètres

En plus du choix de la fonction et des bornes, l'utilisateur peut choisir parmi les 3 algorithmes de recherche du zéro de la fonction.

1.3.3 Main.c

Soit la fonction main suivante :

Listing 16 – Fonction main

```

1 int main(int argc, char **argv){
2
3     SetupSearch setupSearch = defaultSetupSearch; // Structure par défaut
4     // Modification des informations en fonction des arguments de main
5     setParam(argv, argc, getSetupSearch, &setupSearch);
6     showSearch(&setupSearch); // Affiche les informations
7     // Initialisation de la structure Result
8     Result result = {0,0, setupSearch.setupFunction.range.min, setupSearch.setupFunction.range.max};
9
10    // Faire, tant que la |f(x)| > epsilon
11    do{
12        // Modifie la structure result
13        setupSearch.searchZero(setupSearch.setupFunction.function, &result);
14        if(result.step > MAX_STEP){ // En cas de non convergence
15            printf("Pas de résultats\n");
16            exit(EXIT_FAILURE);
17            break;
18        }
19    }
20    while(fabs(setupSearch.setupFunction.function(result.result)) > EPSILON);
21    printf("Résultat : %.10f en %d étapes.\n", result.result, result.step);
22    return EXIT_SUCCESS;
23}

```

Ce programme nous assure que si l'exécution nous renvoie EXIT_SUCCESS alors il a trouvé un point proche du point réel d'annulation.

La validité du résultat ne peut pas être remise en cause. Si on souhaite une meilleure précision il suffit de modifier la macro EPSILON.

1.3.4 Résultats à l'exécution

```
> ./exe_3
-----SETUP-----
Range      Min : -10.00 Max : 10.00
Functions   Fonction : square
Search algorithme
    Algorithme de recherche du zéros de la fonction : bisection
Résultat : 0.000000000 en 1 étapes.
> ./exe_3 --polynomial
-----SETUP-----
Range      Min : -10.00 Max : 10.00
Functions   Fonction : polynomial
Search algorithme
    Algorithme de recherche du zéros de la fonction : bisection
Résultat : 5.000000000 en 2 étapes.
> ./exe_3 --exponential
-----SETUP-----
Range      Min : -10.00 Max : 10.00
Functions   Fonction : exponential
Search algorithme
    Algorithme de recherche du zéros de la fonction : bisection
Résultat : 0.1098614931 en 25 étapes.
> ./exe_3 --sinusoide
-----SETUP-----
Range      Min : -10.00 Max : 10.00
Functions   Fonction : sinusoide
Search algorithme
    Algorithme de recherche du zéros de la fonction : bisection
Résultat : -2.7488934994 en 24 étapes.

> ./exe_3 --secantes
-----SETUP-----
Range      Min : -10.00 Max : 10.00
Functions   Fonction : square
Search algorithme
    Algorithme de recherche du zéros de la fonction : secante
Division par zeros
> ./exe_3 --secantes --polynomial
-----SETUP-----
Range      Min : -10.00 Max : 10.00
Functions   Fonction : polynomial
Search algorithme
    Algorithme de recherche du zéros de la fonction : secante
Résultat : 5.0000000001 en 10 étapes.
> ./exe_3 -min -1 --fausse --square
-----SETUP-----
Range      Min : -1.00 Max : 10.00
Functions   Fonction : square
Search algorithme
    Algorithme de recherche du zéros de la fonction : fausse_pos
Résultat : -0.0009991008 en 1002 étapes.
> ./exe_3 --fausse --polyno
-----SETUP-----
Range      Min : -10.00 Max : 10.00
Functions   Fonction : polynomial
Search algorithme
    Algorithme de recherche du zéros de la fonction : fausse_pos
Résultat : 4.999999769 en 31 étapes.
```

FIGURE 4 –

Les valeurs d'annulation des fonctions :

- Square $x = 0$
- Polynomilal $x = 5$
- Exponential $x = 0.109861228867$
- Sinusoide $x = \frac{\pi}{8} + k\pi, k \in \mathbb{Z}$

Les 4 premières exécutions avec l'algorithme de recherche **bisection** on converge très vite sur les bonnes valeurs.

Par contre les deux autres s'en sortent beaucoup moins bien, voir ne converge pas sur les bornes considérées.

2 TP2

2.1 Objectifs

- Lire une image au format BMP
- Appliquer des transformations au niveau des valeurs des pixels
- Enregistré au format BMP l'image avec les modifications

On va considérer :

- La taille d'une entête de fichier BMP fait 54 octets.
- Seul le nombre de pixels varie d'une image BMP à une autre.
- Les images BMP sont codées en 24 bits.

2.2 Lire l'entête d'une image au format BMP

Premièrement il faut copier dans un tableau de 54 octets de type **unsigned char**, l'entête du fichier.

Tout ce qui nous intéresse c'est le nombre de pixels sur l'image, pour cela l'entête spécifie le nombre de pixels horizontaux/verticaux codés sur 4 octets, respectivement leur premier octet se trouve à l'octet 18 et 22 de l'entête.

Pour convertir les 4 octets, la fonction `getValue` rempli un champ de type **unsigned char** de taille 4 de l'union `Value` et elle renvoie son champ de type **int**.

Listing 17 – Récupération du nombre de pixel horizontal/vertical

```
1 typedef union Value Value;
2
3 union Value{
4     unsigned char data[4];
5     int value;
6 };
7
8 int getValue(unsigned char* header, int offset){
9     Value value;
10    for(int i = 0; i < 4; i++) value.data[i] = header[offset+i];
11    return value.value;
12 }
```

Je l'utilise comme ceci dans le main ;

Listing 18 – Récupération du nombre de pixel horizontal/vertical

```
1 int width = getValue(header, 18);
2 int height = getValue(header, 22);
3 long int nb_pixels = width*height;
```

2.3 Lecture/Ecriture des pixels

Les valeurs des pixels sont données en BGR, avec un octet par canal.

R,G,B sont des tableaux de taille **nb_pixels** de type **unsigned char**.

Listing 19 – Lecture/Ecriture des pixels dans le fichier au format BMP

```
1 void read_RGB(FILE *file, size_t nb_pixels, unsigned char *R, unsigned char *G, unsigned char *B){
2     for(int i = 0; i < nb_pixels; i++){
3         fread(B+i, sizeof(unsigned char), 1, file);
4         fread(G+i, sizeof(unsigned char), 1, file);
5         fread(R+i, sizeof(unsigned char), 1, file);
6     }
7 }
8
9 void write_RGB(FILE *file, size_t nb_pixels, unsigned char *R, unsigned char *G, unsigned char *B){
10    for(int i = 0; i < nb_pixels; i++){
11        fwrite(B+i, sizeof(unsigned char), 1, file);
12        fwrite(G+i, sizeof(unsigned char), 1, file);
13        fwrite(R+i, sizeof(unsigned char), 1, file);
14    }
15 }
```

2.4 Les fonctions de filtrage

Listing 20 – long hachage(char *)

```

1 void RGB_to_BGR(size_t nb_pixels, unsigned char *R, unsigned char *G, unsigned char *B){
2     unsigned char *tmp = malloc(sizeof(unsigned char)*nb_pixels);
3     for(int i = 0; i < nb_pixels; i++) *(tmp+i) = *(B+i);
4     for(int i = 0; i < nb_pixels; i++) *(B+i) = *(R+i);
5     for(int i = 0; i < nb_pixels; i++) *(R+i) = *(tmp+i);
6 }
7
8 void RGB_to_GB(size_t nb_pixels, unsigned char *R, unsigned char *G, unsigned char *B){
9     for(int i = 0; i < nb_pixels; i++) *(R+i) = 0;
10 }
11
12 void invertColor(size_t nb_pixels, unsigned char *R, unsigned char *G, unsigned char *B){
13     for(int i = 0; i < nb_pixels; i++) *(R+i) = 255-*((R+i));
14     for(int i = 0; i < nb_pixels; i++) *(G+i) = 255-*((G+i));
15     for(int i = 0; i < nb_pixels; i++) *(B+i) = 255-*((B+i));
16 }
```

Il est prévu dans le programme, 3 types de transformation de l'image :

- Permutation des canaux **rouge** et **bleu**
- Suppression du canal **rouge**
- Inversion des couleurs

Cette structure de données va permettre de modifier le type de filtrage appliqué et le path de l'image via les arguments de main :

```
typedef void (*Filter)(size_t, unsigned char *, unsigned char *, unsigned char *);
```

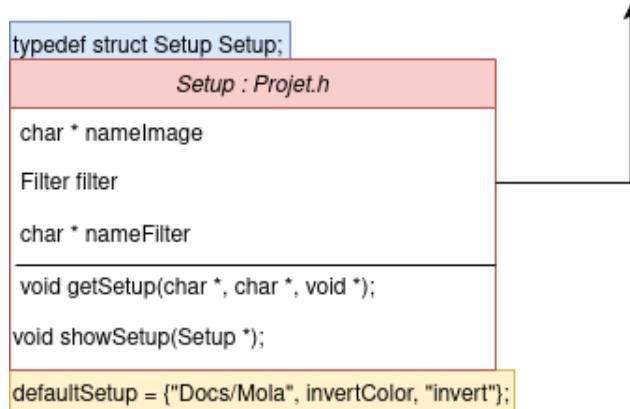


FIGURE 5 –

2.5 Code de la fonction main

Listing 21 – Fonction main

```
1 int main(int argc, char **argv){  
2     Setup setup = defaultSetup;  
3     setParam(argv, argc, getSetup, &setup);  
4     showSetup(&setup);  
5  
6     FILE* fileRead; //Fichier de lecture  
7     FILE* fileWrite; //Fichier de sortie  
8  
9     unsigned char header[LONGUEUR_ENTETE];  
10  
11    char nameReadFile[100];  
12    sprintf(nameReadFile, "%s.bmp", setup.nameImage); //Formatage du path  
13  
14    fileRead=fopen(nameReadFile, "rb");  
15    if(fileRead == NULL){  
16        printf("Error open image !!!\n");  
17        exit(EXIT_FAILURE);  
18    }  
19    //Lecture des 54 octets de l'entête  
20    fread(header, sizeof(char), LONGUEUR_ENTETE, fileRead);  
21  
22    //Taille de l'image  
23    int width = getValue(header, 18);  
24    int height = getValue(header, 22);  
25    long int nb_pixels = width*height;  
26  
27    //Allocation des canaux  
28    unsigned char *R = malloc(sizeof(unsigned char)*nb_pixels);  
29    unsigned char *G = malloc(sizeof(unsigned char)*nb_pixels);  
30    unsigned char *B = malloc(sizeof(unsigned char)*nb_pixels);  
31  
32    //Récupération des valeurs des pixels  
33    read_RGB(fileRead, nb_pixels, R, G, B);  
34  
35    //Application du filtre  
36    setup.filter(nb_pixels, R, G, B);  
37  
38    char nameWriteFile[100];  
39    //Formatage du path de l'image modifié  
40    sprintf(nameWriteFile, "%s_%s.bmp", setup.nameImage, setup.nameFilter);  
41  
42    fileWrite=fopen(nameWriteFile, "wb");  
43    //Recopie de l'entête bmp  
44    fwrite(header, sizeof(unsigned char), LONGUEUR_ENTETE, fileWrite);  
45    //Écriture des valeurs de pixels dans le fichier  
46    write_RGB(fileWrite, nb_pixels, R, G, B);  
47  
48    //Libère les ressources  
49    fclose(fileRead);  
50    fclose(fileWrite);  
51    return EXIT_SUCCESS;  
52 }
```

2.6 Résultats : Images après traitement

