Mini-projet d'algorithmique et programmation — Dégradés

Paul Gaborit

IFIA M1 – Mars 2021 – IMT Mines Albi

1 Consignes

Ce travail s'effectue en binôme. Vous déposerez sur campus une archive (`.zip`, `.rar`, `.7z`, `.tgz`...) **portant les deux noms du binôme** (un seul dépôt par binôme). Contenu attendu :

1. un document (en PDF) contenant au moins :
   a. les noms et prénoms des membres du binôme,
   b. une analyse des problèmes auxquels vous avez été confrontés,
   c. les algorithmes mis en œuvre,
   d. un descriptif de ce que vous avez programmé expliquant les choix que vous avez faits,
   e. des exemples de résultats en détaillant la manière de les produire avec vos programmes.
2. les fichiers sources (les fichiers `.c`) des programmes que vous aurez réalisés,
3. toutes les images (les fichiers `.ppm`) produites.

Il est inutile de mettre vos programmes exécutables (`.exe`) dans votre archive. Nous recompilerons nous-mêmes tous vos programmes lors de l'évaluation de votre travail.

Conseils : n'oubliez pas de commenter vos fichiers sources. Nommez vos fichiers sources, vos programmes et vos images en respectant les consignes données.

2 Objectif

Il existe de nombreuses manières de réaliser des dégradés de couleurs. Ce projet a pour but de comparer deux méthodes différentes d'interpolation des couleurs : dans l'espace des couleurs RVB (*rouge, vert, bleu*) et dans l'espace des couleurs TSV (*teinte, saturation, valeur*).

Dans la suite de ce document ainsi que dans les codes proposés, nous allons utiliser les abréviations anglaises :

— RVB devient donc RGB (pour *red, green, blue*),
— TSV devient donc HSV (pour *hue, saturation, value*).

Dans un premier temps nous allons définir quelques fonctionnalités qui manquent dans la bibliothèque `libimage`.

Ensuite, nous créerons deux programmes permettant de réaliser des dégradés horizontaux en RGB et en HSV.

Puis nous terminerons par l'utilisation de l'interpolation bilinéaire pour réaliser des dégradés entre quatre couleurs en comparant nos deux méthodes.

3 Nouvelles fonctionnalités

3.1 Améliorations de `libimage`

La bibliothèque `libimage` fournit, entre autres, les fonctions `rgb_to_hsv` et `hsv_to_rgb` pour convertir des composantes RGB en HSV et inversement. Mais elle ne propose pas de structures de données pour stocker des couleurs HSV.

Le programme `rgb-hsv.c` (figure 8 page 6) comble cette lacune en définissant le type `couleur_rgb` comme un *alias* du type `couleur` et en définissant le nouveau type `couleur_hsv` qui permet de stocker les 3 composantes d'une couleur HSV. Il définit aussi deux nouvelles fonctions de conversion entre ces deux types de couleurs (`rgb2hsv` et `hsv2rgb`).

La fonction principale (`main`) sert à réaliser des tests unitaires de ces deux nouvelles fonctions et permet aussi de vérifier qu'une conversion RGB vers HSV suivie d'une conversion HSV vers RGB est stable pour toutes les couleurs RGB possibles.

3.2 Création de l'espace XYV

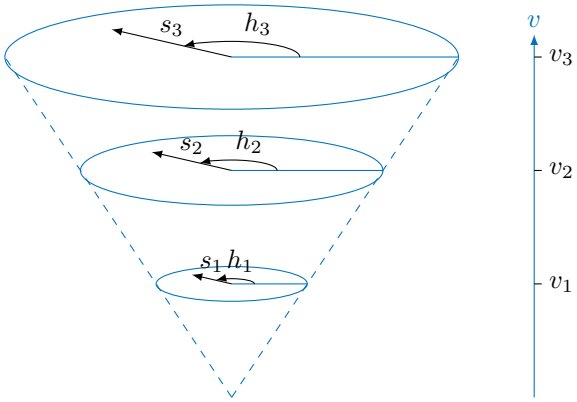
Les 3 composantes HSV définissent la position d'une couleur dans un cône mais les composantes h et s sont des coordonnées polaires assez peu pratiques pour réaliser des interpolations linéraires. Nous allons donc créer un espace de couleurs nommé XYV qui est le même que HSV mais où les deux composantes h et s (des coordonnées polaires) sont remplacées par x et y (des coordonnées cartésiennes) et où v reste inchangé (figure 1 page suivante).

En pratique, vous devez développer un programme nommé `rgb-xyv.c` :

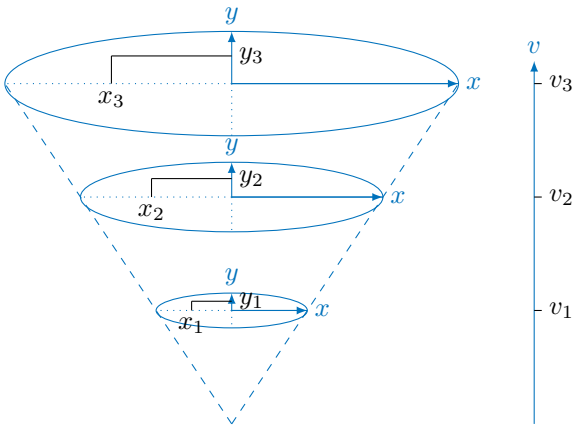
— en reprenant intégralement le code du programme `rgb-hsv.c`
— en y ajoutant la définition du nouveau type `couleur_xyv`
— puis en y ajoutant les deux fonctions de conversions `hsv2xyv` et `xyv2hsv` (vous pouvez même coder les fonctions `rgb2xyv` et `xyv2rgb` en combinant les autres fonctions de conversions).

La fonction principale (`main`) de votre programme `rgb-xyv.c` vérifiera la stabilité d'une conversion RGB vers XYV suivie d'une conversion XYV vers RGB pour toutes les couleurs RGB possibles.

N'oubliez pas que l'espace HSV (qui sert ici d'espace intermédiaire) gère h comme un angle exprimé en degrés et compris entre 0° et 360°.



(a) dans l'espace HSV



(b) dans l'espace XYV

FIGURE 1 – Trois couleurs similaires (seule la composante v_i change) représentées dans l'espace HSV (a) et dans l'espace XYV (b).

4 Dégradés horizontaux

4.1 Dégradés RGB

Le programme `rgb-gradient.c` (en anglais, un *dégradé de couleur* se dit *color gradient*) génère des images contenant des dégradés horizontaux via une interpolation linéaire entre deux couleurs dans l'espace RGB.

Son code reprend les nouveaux types et les nouvelles fonctions du programme `rgb-hsv.c`. Puis il ajoute la fonction `interpolation_rgb` (figure 9 page 7).

La fonction `main` (figure 13 page 8) utilise `argc` et `argv` pour lire les arguments fournis sur la ligne de commande au lancement du programme. Pour appeler ce programme, il faut utiliser une ligne de commande contenant dans l'ordre :

1. le nom du programme lui-même (`argv[0]`),
2. le nom du fichier image à produire (`argv[1]`),
3. la définition de cette image sous la forme *largeur*`x`*hauteur* (`argv[2]`),
4. les 3 composantes de la couleur de gauche sous la forme `r,v,b` (`argv[3]`),
5. les 3 composantes de la couleur de droite toujours sous la forme `r,v,b` (`argv[4]`).

La fonction `argv_vers_couleur_rgb` (figure 12 page 7) permet de factoriser l'extraction d'une couleur depuis un argument de la ligne de commande et la fonction `usage` est appelée à chaque fois que le programme rencontre un souci lors de l'analyse de ces arguments (elle rappelle à l'utilisateur *l'usage* du programme).

4.2 Dégradés RGB et XYV

Vous devez écrire le programme `rgb-xyv-gradient.c`. Pour cela, vous reprendrez l'intégralité du code du programme `rgb-gradient.c` auquel vous ajouterez :

1. la fonction `interpolation_xyv` qui réalisera l'interpolation entre deux couleurs mais dans l'espace XYV (alors que la fonction `interpolation_rgb` le fait dans l'espace RGB).
2. la fonction `gradient_xyv` qui remplira une image par un dégradé horizontal comme le fait la fonction `gradient_rgb` mais en utilisant cette fois une interpolation dans l'espace XYV.
3. un argument supplémentaire sur la ligne de commande (ce sera le dernier argument de la ligne de commande) dont la valeur sera 0 pour choisir un dégradé RGB ou 1 pour choisir un dégradé XYV (les fonctions `main` et `usage` devront être modifiées pour prendre en compte ce nouvel argument).

Les figures 2, 3, 4 et 5 page suivante montrent des exemples de résultats obtenus.

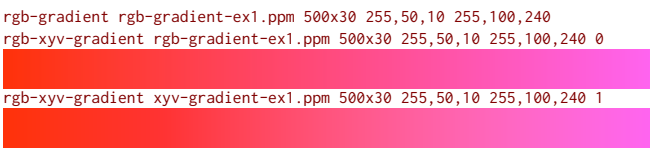


FIGURE 2 – Fichiers images `rgb-gradient-ex1.ppm` et `xyv-gradient-ex1.ppm` (les commandes utilisées sont placées au-dessus des images)

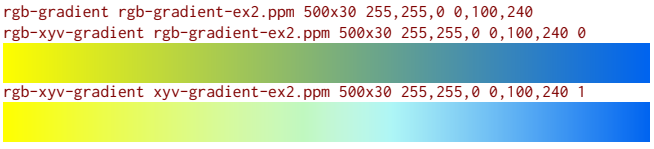


FIGURE 3 – Fichiers images `rgb-gradient-ex2.ppm` et `xyv-gradient-ex2.ppm` (les commandes utilisées sont placées au-dessus des images)

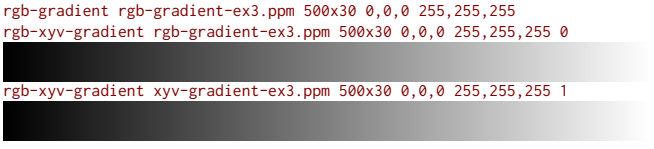
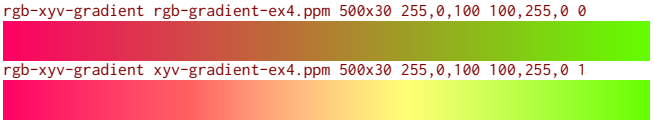


FIGURE 4 – Fichiers images `rgb-gradient-ex3.ppm` et `xyv-gradient-ex3.ppm` (les commandes utilisées sont placées au-dessus des images)

rgb-gradient rgb-gradient-ex4.ppm 500x30 255,0,100 100,255,0
rgb-xyv-gradient rgb-gradient-ex4.ppm 500x30 255,0,100 100,255,0 0



rgb-xyv-gradient xyv-gradient-ex4.ppm 500x30 255,0,100 100,255,0 1

FIGURE 5 – Fichiers images `rgb-gradient-ex4.ppm` et `xyv-gradient-ex4.ppm` (les commandes utilisées sont placées au-dessus des images)

5 Dégradés bilinéaires

Une interpolation bilinéaire permet de réaliser l'interpolation d'une fonction dans un quadrilatère pour lequel la valeur de la fonction est connue en chacun des sommets.

Pour réaliser une interpolation bilinéaire, on procède en deux étapes. Tout d'abord on calcule l'interpolation linéaire le long de deux bords opposés puis ensuite on calcule l'interpolation linéaire entre les deux valeurs obtenues.

Dans l'exemple de la figure 6, on a choisi les bords s_1,s_2 et s_4,s_3 sur lesquels on calcule les valeurs de $f(i_{1,2})$ et $f(i_{3,4})$ par interpolation linéaire. Ensuite pour trouver la valeur d'interpolation en i, on calcule une interpolation linéaire entre $i_{1,2}$ et $i_{3,4}$.

Note : quels que soient les bords opposés choisis, le résultat de l'interpolation bilinéaire reste le même !

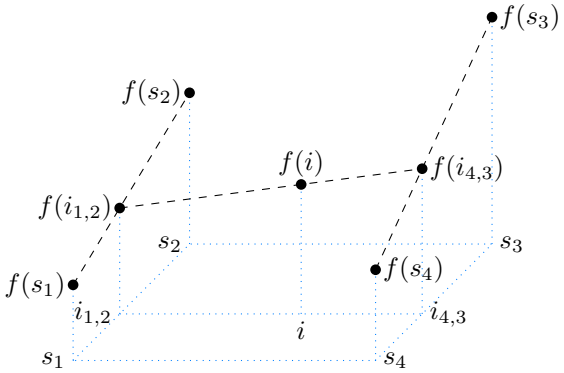


FIGURE 6 – Interpolation bilinéaire au point i : la fonction f est connue uniquement au point s_1, s_2, s_3 et s_4

Nous allons l'utiliser pour réaliser des dégradés de couleurs entre quatre couleurs.

Le programme que vous devez développer s'appellera `rgb-xyv-bilinear.c` et utilisera les arguments suivants :

1. le nom du programme lui-même (`argv[0]`),
2. le nom du fichier image à produire (`argv[1]`),
3. la définition de cette image sous la forme *largeur*`x`*hauteur* (`argv[2]`),
4. les 3 composantes de la couleur du coin en haut à gauche de l'image sous la forme `r,v,b` (`argv[3]`),
5. les 3 composantes de la couleur du coin en haut à droite de l'image sous la forme `r,v,b` (`argv[4]`),
6. les 3 composantes de la couleur du coin en bas à droite de l'image sous la forme `r,v,b` (`argv[5]`),
7. les 3 composantes de la couleur du coin en bas à gauche de l'image sous la forme `r,v,b` (`argv[6]`),
8. une valeur booléenne qui vaudra 0 pour un dégradé bilinéaire dans l'espace RGB ou 1 pour un dégradé bilinéaire dans l'espace XYV (`argv[7]`),

Pour calculer la couleur de chaque pixel de l'image résultat, votre programme devra réaliser une interpolation bilinéaire entre les couleurs des quatre coins de l'image. L'interpolation bilinéaire doit être faire sur chacune des composantes indépendamment les unes des autres (donc sur r, g et b dans l'espace RGB et sur x, y, v dans l'espace XYV).

Pour tester le bon fonctionnement de votre programme, vous pourrez comparer les résultats obtenus aux images d'exemple fournies dans la figure

6 Éléments techniques

6.1 Divisions entières et réelles

En langage C (mais c'est aussi vrai dans d'autres langages informatiques), lorsqu'on effectue une division (via `/`), le type de division diffère selon la nature des opérandes.

Lorsqu'au moins l'un des deux opérandes (le *dividende* ou le *diviseur*) sont des réels (des `double`), le résultat de la division est aussi réel. Ainsi `11.0/4.0` vaut `2.75`.

Lorsque les deux opérandes (le *dividende* et le *diviseur*) sont des entiers (des `int`), la division est une division euclidienne. Ainsi le résultat de `11/4` est exactement `2` (le *quotient*).

Pour effectuer une division réelle entre deux variables entières, il suffit de multiplier le *dividende* par la constante réelle `1.0` (alors que `1` est une constante entière). C'est la méthode utilisée dans la fonction `gradient_rgb` pour calculer correctement la variable `proportion` (figure 10 page 7).

6.2 Les arguments d'un programme

Un programme écrit en C peut utiliser des arguments qu'on lui passe sur la ligne de commandes lors de son appel. Pour cela, il suffit d'utiliser les paramètres `argc` et `argv` de la fonction `main`.

Le paramètre `argc` contient le nombre d'arguments reçus par le programme et le paramètre `argv` est un tableau de chaînes de caractères contenant `argc` éléments. Le premier élément du tableau `argv` contient toujours le nom du programme lui-même. Les autres éléments contiennent les arguments (de `1` à `argc-1`).

Pour lire des valeurs numériques depuis un argument (`argv[i]`), on utilise la fonction `sscanf`. En cas d'erreur, on informe l'utilisateur de l'erreur rencontrée et du bon usage du programme via une fonction `usage`.

Les programmes `rgb-hsv.c` et `rgb-gradient.c` sont des bons exemples d'utilisation des arguments fournis à un

programme par la ligne de commande. Vous devez vous en inspirer pour les programmes à réaliser.

6.3 Fonctions mathématiques utiles

La bibliothèque mathématique fournit des fonctions et une constante utiles pour vos programmes :

- La fonction `sqrt` (que nous avons déjà utilisée) calcule la racine carrée d'un nombre positif.
- Les fonctions `cos` et `sin` calculent respectivement le cosinus et le sinus d'un angle exprimé en radians.
- La fonction `atan2` prend comme paramètres les coordonnées cartésiennes d'un point (dans l'ordre y et x) et calcule l'angle polaire exprimé en radians correspondant au point fourni, en respectant la contrainte suivante :

$$-\pi \leqslant \operatorname{atan2}(y, x) \leqslant +\pi$$

- La constante `M_PI` donne la valeur approchée de π la plus précise possible possible sur votre ordinateur.

Rappel : pour utiliser le bibliothèque mathématique, il vous faut inclure `math.h` au début de votre programme et ajouter l'option `-lm` lors de la compilation (cette option n'est pas indispensable sous Windows).

7 Éléments à rendre

Outre tous vos fichiers sources (les fichiers `rgb-xyv.c`, `rgb-xyv-gradient.c` et `rgb-xyv-bilinear.c`), vous devrez fournir dans votre archive toutes les images produites (les fichiers `.ppm`) et indiquer pour chacune d'entre elles la ligne de commande utilisée pour l'obtenir.

Pour toute fonction que vous aurez écrite vous-même, vous devrez écrire l'algorithme correspondant et expliquer l'idée de son fonctionnement dans votre rapport.

Pour vérifier que vos programmes sont corrects, vous pouvez écrire un programme de comparaison d'images qui comparera deux images : une image que nous fournissons et celle que vous aurez produite avec les mêmes paramètres. Si aucun pixel ne diffère, c'est que votre programme est correct (ou qu'il contient les mêmes erreurs que les nôtres !).

Pour insérer une image dans votre rapport, vous pouvez convertir un fichier `.ppm` en un fichier `.png` via Gimp ou tout autre programme de conversion de format d'images (ne passez pas par le format jpeg qui dégrade les images).

8 Question subsidiaire

L'application « I Love Hue » [1] publiée par *Zut Games Ltd* est un jeu de classement des couleurs utilisant une interpolation bilinéaire. À votre avis, dans quel espace de couleurs (RGB ou HSV/XYV) cette interpolation est-elle réalisée ?

1. http://i-love-hue.com/

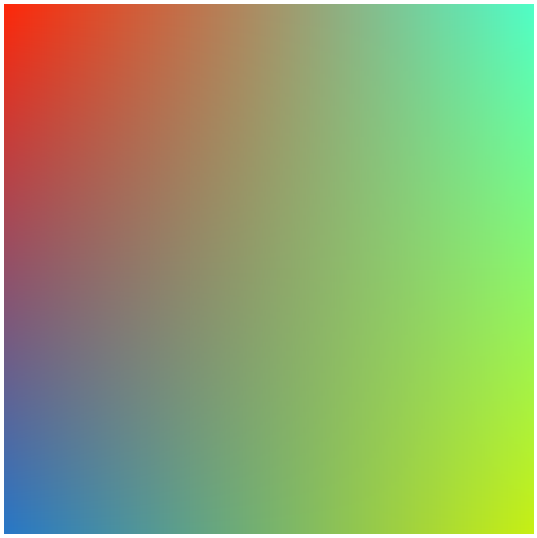
9 Fichiers joints

Voici la liste des fichiers joints à ce PDF :
`rgb-gradient-ex1.ppm`, `rgb-gradient-ex2.ppm`, `rgb-gradient-ex3.ppm`, `rgb-gradient-ex4.ppm`, `xyv-gradient-ex1.ppm`, `xyv-gradient-ex2.ppm`, `xyv-gradient-ex3.ppm`, `xyv-gradient-ex4.ppm`, `rgb-bilinear-ex1.ppm`, `xyv-bilinear-ex1.ppm`, `rgb-bilinear-ex2.ppm`, `xyv-bilinear-ex2.ppm`, `rgb-bilinear-ex3.ppm`, `xyv-bilinear-ex3.ppm`, `rgb-hsv.c`, `rgb-gradient.c`, `libimage.c`, `libimage.h`.

```
rgb-xyv-bilinear rgb-bilinear-ex1.ppm 500x500 \
  255,255,0 0,255,255 255,0,255 255,255,255 0
```

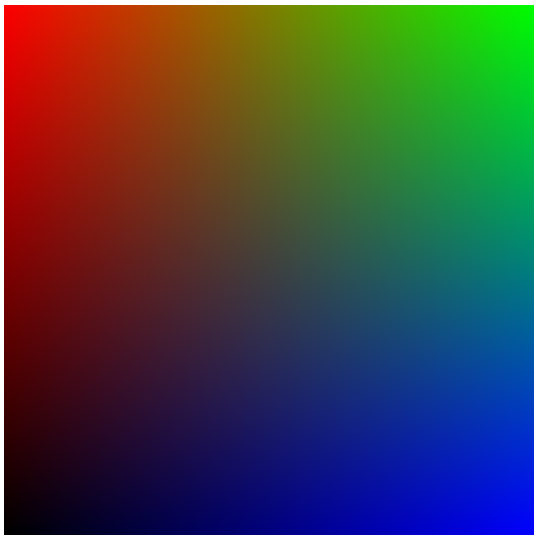
```
rgb-xyv-bilinear xyv-bilinear-ex1.ppm 500x500 \
  255,255,0 0,255,255 255,0,255 255,255,255 1
```

```
rgb-xyv-bilinear rgb-bilinear-ex2.ppm 500x500 \
  255,40,10 80,255,200 200,240,20 40,120,200 0
```



```
rgb-xyv-bilinear xyv-bilinear-ex2.ppm 500x500 \
  255,40,10 80,255,200 200,240,20 40,120,200 1
```

```
rgb-xyv-bilinear rgb-bilinear-ex3.ppm 500x500 \
        255,0,0 0,255,0 0,0,255 0,0,0 0
```



```
rgb-xyv-bilinear xyv-bilinear-ex3.ppm 500x500 \
        255,0,0 0,255,0 0,0,255 0,0,0 1
```

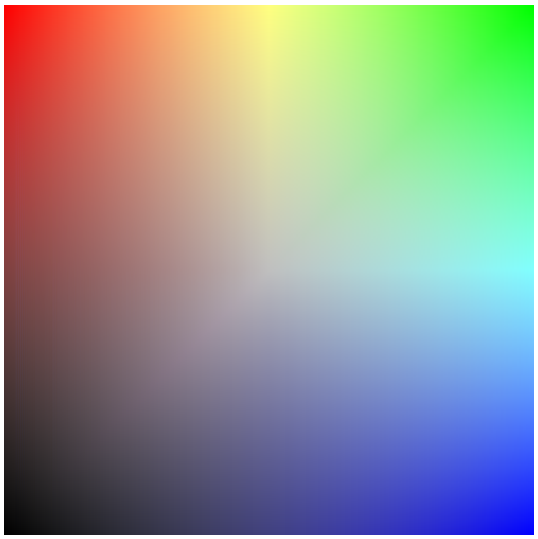


FIGURE 7 – Six exemples de dégradés bilinéaires obtenus via le programme `rgb-xyv-bilinear`. La ligne de commande utilisée est indiquée au-dessus de chaque image (le caractère `\` est utilisé pour couper une ligne trop longue : il ne doit pas être saisi)

```c
#include "libimage.h"

// nouveaux types
typedef couleur couleur_rgb;
typedef struct {
  double h;
  double s;
  double v;
} couleur_hsv;

// prototypes des fonctions
couleur_hsv rgb2hsv(couleur_rgb c_rgb);
couleur_rgb hsv2rgb(couleur_hsv c_hsv);

// nouvelles fonctions de conversions RGB <=> HSV
couleur_hsv rgb2hsv(couleur_rgb c_rgb) {
  couleur_hsv c_hsv;
  rgb_to_hsv(c_rgb.rouge, c_rgb.vert, c_rgb.bleu, &c_hsv.h, &c_hsv.s, &c_hsv.v);
  return c_hsv;
}

couleur_rgb hsv2rgb(couleur_hsv c_hsv) {
  couleur_rgb c_rgb;
  hsv_to_rgb(c_hsv.h, c_hsv.s, c_hsv.v, &c_rgb.rouge, &c_rgb.vert, &c_rgb.bleu);
  return c_rgb;
}

// tests unitaires et test de stabilité des conversions
int main() {
  couleur_rgb c_rgb;
  int nb_conversions_ok = 0;

  for (c_rgb.rouge = 0; c_rgb.rouge < 256; c_rgb.rouge++) {
    for (c_rgb.vert = 0; c_rgb.vert < 256; c_rgb.vert++) {
      for (c_rgb.bleu = 0; c_rgb.bleu < 256; c_rgb.bleu++) {
        couleur_hsv c_hsv;
        couleur_rgb c_rgb_verif;

        // conversion aller-retour entre rgb et hsv (pour tester les nouvelles fonctions)
        c_hsv = rgb2hsv(c_rgb);
        c_rgb_verif = hsv2rgb(c_hsv);

        // vérification de la stabilité
        if (c_rgb_verif.rouge != c_rgb.rouge
            || c_rgb_verif.vert != c_rgb.vert
            || c_rgb_verif.bleu != c_rgb.bleu) {
          printf("Conversion instable: (%d,%d,%d) => (%d,%d,%d)\n",
                 c_rgb.rouge, c_rgb.vert, c_rgb.bleu,
                 c_rgb_verif.rouge, c_rgb_verif.vert, c_rgb_verif.bleu);
        } else {
          nb_conversions_ok++;
        }
      }
    }
  }

  printf("Nb de conversions stables: %d\n", nb_conversions_ok);
  if (nb_conversions_ok == 256 * 256 * 256) {
    printf("Conversions stables pour toutes les couleurs RVB\n");
  }
  return 0;
}
```

FIGURE 8 – Source du fichier rgb-hsv.c

```c
// fonction d'interpolation entre deux couleurs dans l'espace RGB
couleur_rgb interpolation_rgb(double poids, couleur_rgb c1, couleur_rgb c2) {
  couleur res;

  res.rouge = (1 - poids) * c1.rouge + poids * c2.rouge;
  res.vert =  (1 - poids) * c1.vert + poids * c2.vert;
  res.bleu =  (1 - poids) * c1.bleu + poids * c2.bleu;

  return res;
}
```

FIGURE 9 – Code de la fonction `interpolation_rgb` (extrait de `rgb-gradient.c`)

```c
// remplissage d'une image par un dégradé RGB horizontal
void gradient_rgb(image im, couleur_rgb c_left, couleur_rgb c_right) {
  int x, y;
  for (x = 0; x < im.largeur; x++) {
    couleur_rgb c;

    double proportion = x*1.0/(im.largeur-1);
    c = interpolation_rgb(proportion, c_left, c_right);

    for (y = 0; y < im.hauteur; y++) {
      change_couleur(im, x, y, c);
    }
  }
}
```

FIGURE 10 – Code de la fonction `gradient_rgb` (extrait de `rgb-gradient.c`)

```c
// description de l'usage du programme
void usage(char * nom_programme, char * message) {
  fprintf(stderr, "Erreur: %s\n", message);
  fprintf(stderr, "\n");
  fprintf(stderr, "Usage: %s result.ppm LxH r1,v1,b1 r2,v2,b2\n", nom_programme);
  fprintf(stderr, "\n");
  fprintf(stderr, "   result.ppm: fichier image a produire\n");
  fprintf(stderr, "   L, H: largeur et hauteur de l'image (entiers positifs)\n");
  fprintf(stderr, "   r,v,b: composantes rouge, verte et bleue (entiers 0-255)\n");
  exit(1);
}
```

FIGURE 11 – Code de la fonction `usage` (extrait de `rgb-gradient.c`)

```c
// fonction utilitaire d'extraction d'une couleur RGB d'un argument
couleur_rgb argv_vers_couleur_rgb(char * nom_programme, char * argument) {
  couleur_rgb c;

  int res_lecture = sscanf(argument, "%d,%d,%d", &c.rouge, &c.vert, &c.bleu);
  if (res_lecture != 3) {
    usage(nom_programme, "composantes r,v,b illisibles");
  }
  if ((c.rouge < 0 || c.rouge > 255)
      || (c.vert < 0 || c.vert > 255)
      || (c.bleu < 0 || c.bleu > 255)) {
    fprintf(stderr, "Erreur dans '%s'\n", argument);
    usage(nom_programme, "composantes r,v,b hors intervalle 0-255");
  }
  return c;
}
```

FIGURE 12 – Code de la fonction `argv_vers_couleur_rgb` (extrait de `rgb-gradient.c`)

```c
int main(int argc, char *argv[]) {
  int largeur, hauteur;
  couleur_rgb c_gauche, c_droite;
  int res_lecture;
  image im;

  // vérification du bon nombre d'arguments
  if (argc != 5) {
    usage(argv[0], "nombre d'arguments incorrect");
  }

  // extraction de la largeur et de la hauteur
  res_lecture = sscanf(argv[2], "%dx%d", &largeur, &hauteur);
  if (res_lecture != 2) {
    usage(argv[0], "valeurs entieres LxH illisibes");
  }
  if (largeur < 1 || hauteur < 1) {
    usage(argv[0], "L et H ne sont pas strictement positifs");
  }

  // extraction des deux couleurs
  c_gauche = argv_vers_couleur_rgb(argv[0], argv[3]);
  c_droite = argv_vers_couleur_rgb(argv[0], argv[4]);

  // fabrication de l'image
  im = nouvelle_image(largeur, hauteur);
  gradient_rgb(im, c_gauche, c_droite);

  // écriture de l'image finale
  ecrire_image(im, argv[1]);
  return 0;
}
```

FIGURE 13 – Code de la fonction `main` (extrait de `rgb-gradient.c`)