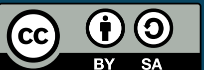
TIT_TOINOU



zeste de savoir

Les espaces de couleurs RVB et TSV

12 août 2019

Table des matières

I. Tutoriel 5

1. Prérequis 6
- 1.1. Lumière . . . 6
  - 1.1.1. Qu'est-ce que la lumière ? . . . 6
  - 1.1.2. Spectre d'un rayon lumineux . . . 6
  - 1.1.3. Absorption et émission . . . 7
  - 1.1.4. Spectres . . . 7
- 1.2. Espaces de couleurs . . . 7
  - 1.2.1. Espaces de couleurs . . . 7
  - 1.2.2. L'approximation RVB pour l'émission . . . 8
  - 1.2.3. L'espace de couleurs CMJ . . . 8
- 1.3. Les transformations mathématiques . . . 9
  - 1.3.1. L'exemple du robinet d'eau chaude . . . 10
- Contenu masqué . . . 10

2. Mélangeur et mitigeur 11
  - 2.0.1. L'exemple du segment . . . 12
- 2.1. Stockage des variables . . . 13
  - 2.1.1. Représentation des variables continues en mémoire . . . 13
  - 2.1.2. Stockage de RVB . . . 14
  - 2.1.3. Pourquoi l'intervalle $[0; 1]$? . . . 14
- 2.2. Zeste de Processing . . . 14
  - 2.2.1. Qu'est-ce que Processing et pourquoi ? . . . 14
  - 2.2.2. A savoir sur Processing . . . 15
  - 2.2.3. Exemple de code . . . 15
- 2.3. Exo : il n'y a plus de cyan ! . . . 16
  - 2.3.1. Cahier des charges . . . 16
  - 2.3.2. Indices . . . 16
  - 2.3.3. Une solution . . . 16
- Contenu masqué . . . 17

3. Filtres RVB 23
- 3.1. Traiter les canaux séparément . . . 23
  - 3.1.1. Fonction `RentrerDans01` . . . 23
  - 3.1.2. Fonction affine . . . 24
  - 3.1.3. Fonction puissance (ou gamma) . . . 24
  - 3.1.4. Postérisation . . . 25
  - 3.1.5. Valeurs différentes selon les canaux . . . 26
- 3.2. Opérations sur tous les canaux . . . 27
  - 3.2.1. N&B : Moyenne, min et max . . . 27

3.2.2. Échange de canaux 28
3.2.3. Combinaisons linéaires 28
3.3. Toutes ces opérations en même temps ! 28
3.4. Exo : contraste automatique 29
3.4.1. Enoncé 29
3.4.2. Une solution 29
Contenu masqué 30

4. De TSV à RVB **37**
4.1. Teintes ”pures” 37
4.2. Recréer l’arc-en-ciel 39
4.2.1. Barycentre entre deux couleurs 39
4.2.2. Coder une fonction d’arc-en-ciel 39
4.3. Exo : générer une roue des couleurs 40
4.3.1. Ce qui est demandé 40
4.3.2. Une solution pour la roue des couleurs en Processing 41
4.3.3. La roue des couleurs postérisée ! 41
4.4. Saturation et valeur 42
4.4.1. Qu’est-ce que la Valeur ? 42
4.4.2. Qu’est-ce que la Saturation ? 42
4.4.3. Exemples avec des images ! 43
4.4.4. Encore des barycentres ! 44
4.4.5. Vérifications mathématiques de la formule finale 44
4.5. TSV vers RVB 45
4.5.1. Au travail ! 45
4.5.2. Vérifications avec deux roues des couleurs 45
4.6. Exo : carrés de couleurs aléatoires 45
4.6.1. Ce qui est demandé 45
4.6.2. Une solution 46
4.6.3. Quelques rendus 46
Contenu masqué 46

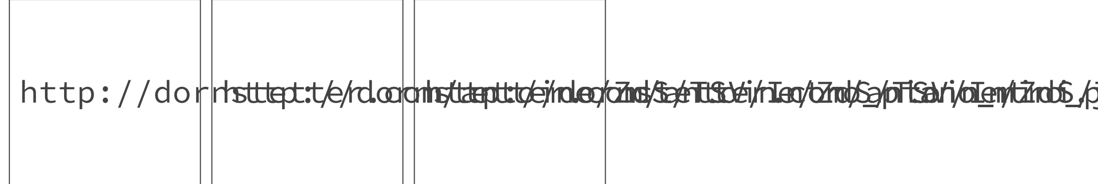
5. Filtres TSV **50**
5.1. RVB vers TSV 50
5.1.1. Renverser les étapes de l’algorithme 50
5.1.2. Vérification élémentaire 51
5.2. Précautions à prendre 52
5.2.1. Teinte 52
5.2.2. Saturation et Valeur 52
5.2.3. Bijection entre TSV et RVB 52
5.3. Canaux Saturation et Valeur 53
5.3.1. Une autre façon de voir les fonctions affines 53
5.3.2. Refaire un exemple vu 53
5.3.3. Gamma 53
5.3.4. Forcer les canaux 54
5.4. Canal Teinte 54
5.4.1. Rotation 54
5.4.2. Homothétie et symétrie 55

5.4.3. Forcer la teinte 55
Contenu masqué 56

Les espaces de couleurs permettent de **représenter, de stocker et de transmettre des ensembles de couleurs**. Ils sont utilisés dans les caméras, appareils photos, écrans de télévision et d'ordinateurs, ondes et antennes hertziennes, lumières, en traitement d'images et de vidéos, etc..

Ce tutoriel a pour but de vous introduire au **traitement d'images par ordinateur** à l'aide de Processing, une variante de Java, en **manipulant les espaces de couleurs RVB et TSV**, présentés dans ce tutoriel.

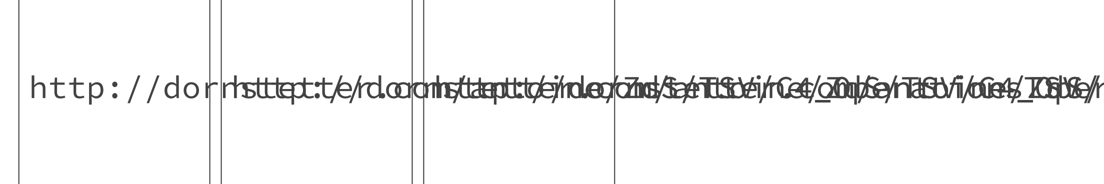
Après un gros chapitre obligatoire de prérequis dans les domaines de la physique, des maths et de la programmation, nous allons étudier les opérations sur les couleurs dans le modèle le plus courant en informatique, le RVB. Voici par exemple les transformations d'images (l'originale est le plus à gauche) que vous serez capable de produire après le second chapitre :



Puis nous nous intéresserons **aux paramètres de la lumières qui sont perceptibles** par les humains en étudiant séparément la Teinte, la Valeur et enfin la Saturation.

Vous devrez ensuite **comprendre par vous-même** comment décrire l'espace de couleurs TSV et faire la liaison avec le RVB.

Enfin viendra le plus intéressant : les filtres d'images dans l'espace TSV ! Voici quelques filtres que vous serez capables de programmer vous-même après le quatrième chapitre :

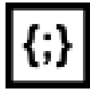


Voici ce dont vous avez besoin pour ce tutoriel :

1. En mathématiques vous devez connaître les opérations de base, les réels, les entiers, la notion de fonction, les intervalles de réels, les ensembles et leur notation..
2. En programmation vous devez déjà connaître un minimum : variables, boucles, fonctions, tableaux, objets..
3. Et enfin vous devez vous munir d'un papier, d'un crayon et de beaucoup de patience pour arriver à trouver les formules de maths !

En principe le tutoriel vous donne tout pour comprendre ce que vous codez ; les algorithmes vus en Processing seront donc faciles à transcrire dans un autre langage de programmation.

Vous pouvez télécharger l'archive contenant tous les exemples du tutoriel :



Télécharger l'archive

Bonne lecture !

Première partie

Tutoriel

1. Prérequis

Certaines connaissance en physique et en mathématiques sont nécessaires pour aborder la suite du tutoriel.

Nous allons aussi introduire l'utilisation de Processing pour le traitement d'images.

1.1. Lumière

Pour décrire les couleurs, nous devons tout d'abord nous pencher sur la lumière et ses propriétés physiques.

1.1.1. Qu'est-ce que la lumière ?

En première approximation, la lumière est composée de rayons lumineux se propageant en ligne droite. Un rayon lumineux est composé par des photons. Les photons sont les particules élémentaires de la lumière et ont une principale caractéristique à retenir : la longueur d'onde, qui est étroitement lié avec l'énergie emmagasinée par ce dernier.

L'œil est capable de voir certaines longueurs d'onde : celles qui correspondent entre le rouge et le violet. L'œil humain n'est **pas capable de voir au dessus** (infrarouge) **ou au dessous** (ultraviolet).

1.1.2. Spectre d'un rayon lumineux

A chaque fois que vous voyez du rouge à un endroit, il y a des milliards de milliards (!) de photons de longueur d'onde rouge qui sont partis de cet endroit, qui ont traversé l'espace entre cet endroit et vos yeux puis qui sont arrivés dans leurs rétines respectives et enfin qui ont "imprimé" des cellules sensibles au rouge dans vos rétines.

Il y a trois types de ces cellules appelées "cônes" : celles sensibles aux grandes longueurs d'ondes, celles sensibles aux moyennes et enfin celles sensibles aux courtes.

La couleur que l'on perçoit est reconstruite par le cerveau en réalisant des combinaisons linéaires de la quantité de lumière perçue par chacun de ces cônes (pour chaque direction de votre vision !).

> **!** En réalité ce que l'on voit est une *impression de rouge* (cf. explication après sur comment tromper l'œil).

1.1.3. Absorption et émission

Ce n'est pas parce que vous apercevez un objet que ce dernier **émet** de la lumière.

Il y a des corps qui émettent de la lumière comme le soleil par exemple. Tous les autres objets se contentent :

— soit d'absorber la lumière reçue : le photon n'est **pas réémis** par le corps et son énergie est emmagasinée ;
— soit de la refléter : le photon continue sa route dans une autre direction avec **la même longueur d'onde**.

Ce comportement est **entièrement déterminé** par la longueur d'onde du photon. Un faisceau de lumière qui contient du rouge et du bleu peut atterrir sur un corps qui absorbe le photon rouge et reflète le photon bleu. La couleur perçue de l'objet sera... bleue !

Autre exemple, un objet qui apparaît noir à l'humain absorbe tous les photons *visibles* (sous-entendu par l'humain). Il emmagasine plus d'énergie, c'est une des raisons qui font qu'il est plus chaud que d'autres objets.

Ainsi la couleur d'un objet qui n'émet pas de lumière que l'on perçoit dépend complètement de ses propriétés d'absorption !

> C'est la raison pour laquelle une lumière presque blanche comme celle du Soleil illumine des objets que l'on perçoit après rouge, violet, bleu, vert, etc...

1.1.4. Spectres

La réalité est un poil plus complexe : pour une longueur d'onde donnée, la proportion de photons absorbés par l'objet (par rapport à tous les photons reçus) est fixe. On parle de taux d'absorption pour une longueur d'onde.

On peut alors tracer ce taux d'absorption quand la longueur d'onde varie. Ce graphique s'appelle **un spectre d'absorption**.

On peut aussi tracer le spectre d'émission d'une source de lumière.

1.2. Espaces de couleurs

1.2.1. Espaces de couleurs

Deux faisceaux lumineux avec des spectres différents peuvent être perçus de la même couleur. Il n'y a donc pas d'équivalence entre "voir une couleur" et "déterminer le spectre de la couleur reçu".

> Pour déterminer le spectre d'un objet, d'une matière, d'un atome, d'une molécule ou d'étoiles dans le ciel on est donc obligés de faire des expériences ; on ne peut pas se fier à l'impression que nos yeux et notre cerveau nous donnent.

Il est ainsi possible de "tromper" l'œil et le cerveau humain. C'est grâce à ça que vous pouvez lire ce cours en ce moment-même.

Le but d'un modèle de couleurs est à la fois d'être plus ou moins bon pour **tromper l'humain** (afin de lui faire percevoir la couleur que l'on veut) et de pouvoir **stocker et transmettre** ces couleurs :

1. physiquement : les mélanges de peinture depuis des siècles, les pellicules pour la photographie et le cinéma, l'impression...
2. analogiquement : les anciennes télévisions, les écrans cathodiques...
3. numériquement : les écrans de nos appareils électroniques, les appareils photos numériques...

1.2.2. L'approximation RVB pour l'émission

Le modèle de couleurs le plus répandu en informatique est le RVB (pour Rouge Vert Bleu), en anglais RGB (pour Red Green Blue).

Ce modèle consiste à faire semblant que toutes les couleurs que l'on perçoit sont un mélange des trois longueurs d'ondes dites *primaires* : celles respectivement du rouge, du vert et du bleu. Ces trois couleurs ne sont pas prises aux hasard : elles correspondent aux maximums d'absorption des trois types de cônes dans les rétines !

Ainsi on a trois coefficients entre 0 et 1 qui indiquent la proportion de chacune des couleurs *primaires* dans le résultat final.

1.2.3. L'espace de couleurs CMJ

Cyan Magenta Jaune est un espace de couleurs dans lequel on décrit les quantités de Cyan, de Magenta et de Jaune qui doivent être absorbées (et non émis comme pour le RVB) pour obtenir la couleur finale à partir d'une couleur blanche.

- Si les trois quantités sont nulles, rien n'est absorbé et on obtient la couleur blanche.
- Si toutes les quantités sont maximales, tout est absorbé et on obtient la couleur noire.
- L'absorption de Cyan et de Magenta donne du Bleu.
- L'absorption de Cyan et de Jaune donne du Vert.
- L'absorption de Magenta et de Jaune donne du Rouge.

Cet espace de couleurs est beaucoup utilisé en impression couleur d'imprimantes grand public (chaque canal représentant la proportion d'encre à mélanger pour chaque endroit de l'image à imprimer).

1.2.3.1. CMJN : ajout de noir en absorption

Une idée assez simple permet d'économiser de l'encre : on ajoute une quatrième encre noire (moins chère que les autres). Étant donné que le Noir est l'ajout de Cyan, de Magenta et de Jaune, tout mélange de CMJ contient une certaine "dose" de Noir.

On peut alors tout simplement rajouter la quantité d'encre noire nécessaire pour créer la couleur, puis soustraire cette quantité de Noir des trois quantités de CMJ initiales.

> Comment calculer la quantité de Noir à partir de CMJ ? Solution :
>
> Contenu masqué n°1

> Remarquez que si les trois quantités de CMJ sont égales (pour imprimer un gris), il n'y a plus besoin de rajouter de l'encre Cyan, Magenta ou Jaune (les quantités deviennent égales à 0 après soustraction de la quantité de Noir).

1.2.3.2. RGBW : ajout de blanc en émission

On peut faire le même raisonnement "en émission" avec le modèle RGB pour créer le modèle *RGBW* (Red Green Blue White).

> Cette idée est utilisée dans certaines télévision : les LED bleues ont une durée de vie plus courte que les autres (rouge et verte). On ajoute alors une quatrième LED de couleur blanche afin de minimiser l'utilisation de la LED bleue.

1.3. Les transformations mathématiques

Pour suivre le reste du tutoriel vous avez besoin de comprendre comment on peut exprimer de différentes façons le même jeu de variables. Par exemple les trois quantités de CMJ contiennent *la même information* que les trois quantités de RVB ou les quatre quantités de CMJN, *mais exprimées différemment.*

Chaque modèle a son utilité : il faut alors savoir écrire les formules pour passer d'un modèle à un autre.

1.3.1. L'exemple du robinet d'eau chaude

Contenu masqué

Contenu masqué n°1

Mathématiquement, la quantité de Noir est tout simplement le minimum des trois autres quantités. Retourner au texte.

2. Mélangeur et mitigeur

Les robinets ont accès à deux sources d'eau : un tuyau d'eau froide à une température T_f et un tuyau d'eau chaude à température T_c. La fonction du robinet est de servir de l'eau plus ou moins chaude en débit variable.

Il y a deux types de robinet d'eau chaude. Lorsque vous avez à faire à un mélangeur, vous pouvez choisir la quantité d'eau froide ainsi que la quantité d'eau chaude. Ce système est proche de ce qui se passe réellement dans la tuyauterie (mélanger les deux sources d'eau). Sauf que ce système apporte un problème du point de vue de l'utilisateur : ce dernier est sensible aux paramètres "température de l'eau" ainsi que "débit de l'eau". En effet, lorsque vous trempez votre main dans la sortie du robinet, vous ressentez plus ou moins de chaleur au niveau de la main ainsi qu'une pression plus ou moins forte. Mais jamais vous ne vous direz "tiens il y a 60 % d'eau froide à 10°C ainsi que 40% d'eau chaude à 25°C" .

Lorsque vous avez à faire à un mitigeur, par contre, vous pouvez directement décider de la température (horizontalement) ainsi que du débit (verticalement).

2.0.0.1. Modélisation mathématique

Nous avons donc deux systèmes pour définir nos variables. Il y a le système "interne" avec deux variables f et c représentant respectivement la quantité d'eau froide ainsi que la quantité d'eau chaude. Puis il y a le système "utilisateur" avec deux variables T et Q représentant respectivement la température T ainsi que Q la quantité d'eau en sortie.

Maintenant nous allons devoir trouver les formules pour passer d'un système à un autre. Les formules pour passer du système interne au système utilisateur (c'est à dire calculer T et Q en fonction de f et c) proviennent de calculs physiques simple. Puisque l'eau se mélange de façon homogène, la température est simplement un barycentre entre les deux température T_f et T_c avec des coefficients respectifs f et c : $\frac{fT_f + cT_c}{f + c}$. Encore par un raisonnement physique, les quantités d'eau s'ajoutent entre elles : $Q = f + c$. Du coup on peut même écrire $T = \frac{fT_f + cT_c}{Q}$!

> **!** Il est important de comprendre cette notion de barycentre pour la suite du tutoriel.

> **i** A chaque fois que vous établissez une formule, il est important de vérifier qu'elle est cohérente avec quelques cas pratiques. Ici on vérifie bien que :
>
> — Si il y a autant d'eau chaude que d'eau froide ($f = c$) alors $T = \frac{T_f + T_c}{2}$

— Si il n'y a pas d'eau froide ($f = 0$) alors $T = T_c$ peu importe la valeur de c !
— Et inversement si il n'y a pas d'eau chaude ($c = 0$) alors $T = T_f$ peu importe la valeur de f !
— La température n'est pas définie quand il n'y a pas de débit d'eau ($Q = 0$).

2.0.0.2. Transformation inverse

Maintenant pour vous entraîner votre tâche va être de trouver la transformation inverse, c'est-à-dire comment passer de (T, Q) à (f, c) !

2.0.0.3. Bijection entre les deux systèmes

A tout couple de variables $(f, c) \in (\mathbb{R}^{+2})^*$ (c'est à dire que f et c sont des réels positifs non tous les deux nuls en même temps) correspond un couple de variables $(Q, T) \in \mathbb{R}^{+*} \times [T_f; T_c]$ et réciproquement.

On dit que les ensembles $(\mathbb{R}^{+2})^*$ et $\mathbb{R}^{+*} \times [T_f; T_c]$ sont en *bijections* ! On démontre avec les formules pour passer d'un système à un autre qu'ils sont équivalents.

On est obligé d'enlever $Q = 0$ (et du coup $f = c = 0$) puisque la température n'est pas définie dans ce cas !

2.0.0.4. Intérêt

L'intérêt de disposer d'un autre système pour exprimer nos variables est de pouvoir appliquer une opération dans cet autre système. Par exemple si on a trouvé la bonne température, on veut pouvoir la fixer puis augmenter le débit sans se pré-occuper des valeurs de f et c.

L'algorithme est en trois étapes :

1. Transformer nos variables f, c en T, Q ;
2. Modifier T, Q en T', Q'. Par exemple $Q' = 2Q$ pour avoir un débit deux fois plus élevé et $T' = T$ garder la même température.
3. Transformer les nouvelles variables T', Q' en f', c' .

2.0.1. L'exemple du segment

Un autre exemple simple pour bien comprendre cette notion. Il y a deux façons de définir un segment fermé en une dimension. Soit vous spécifiez la valeur minimale et la valeur maximale, soit vous spécifiez le centre du segment ainsi que le "rayon" du segment (un segment est un cercle en une dimension).

Ainsi on a les formules suivantes :

$$\begin{cases} \text{centre} = \dfrac{\min + \max}{2} \\ \text{rayon} = \dfrac{|\max - \min|}{2} \end{cases}$$

Nouvel exercice : trouver la transformation inverse !

Contenu masqué n°2

2.0.1.1. Intérêts

Encore une fois ici il peut être pratique de vouloir changer le centre du segment ou son rayon sans se soucier des valeurs min et max. Si on veut réaliser une translation du segment, il suffit d'ajouter une valeur au centre ! Si on veut que le segment soit deux fois plus large, il suffit du multiplier le rayon par deux. Ce modèle de segment est aussi plus pratique pour réaliser des homothéties. L'algorithme en 3 étapes décrit pour le robinet est valable ici avec nos deux nouveaux espaces de variables.

2.1. Stockage des variables

2.1.1. Représentation des variables continues en mémoire

Il faut bien stocker des valeurs en mémoire pour les variables que l'on va manipuler.

Les formules que nous écrivons font appels à des réels mathématiques, qui peuvent être aussi précis que l'on veut. Elles sont indépendantes de la précision des représentations en mémoire de ces variables.

> Cette représentation est importante, puisque c'est elle qui décide **combien** de valeurs différentes la variable peut prendre ainsi que quand deux variables sont **identiques** (elles sont trop proches pour être différentes en mémoire).

Pourtant la façon dont sont codées ces variables intervient forcément quand on les manipule !

Pour éviter de faire intervenir tout le temps ces paramètres, on utilise encore l'algorithme en trois étapes décrit ci-dessus où les deux espaces colorimétriques sont `Couleur_Signal` (qui est proche de la façon dont est stocké la couleur) et `Couleur_Information` (qui est apte à être manipulé par nos formules de maths).

> En Processing `Couleur_Signal` sera la classe `color` et dans la suite du tutoriel `Couleur_Information` correspondra à notre classe `RVB`.

2.1.2. Stockage de RVB

Les couleurs RVB par exemple sont généralement codées de sorte que chaque canal (Rouge, Vert ou Bleu) soit stocké sur 8 bits (= 256 valeurs). Comme chaque canal est un nombre entre 0 et 255 inclus, il est courant de diviser chaque canal par 255 en calcul flottant (normalisation vers $[0, 1]$), d'appliquer nos formules mathématiques sur la couleur et enfin de remultiplier chaque canal par 255 (puis converti en entier) afin de pouvoir stocker la couleur.

2.1.3. Pourquoi l'intervalle $[0; 1]$?

Je rappelle que les valeurs de 0 et 1 signifient respectivement le minimum et le maximum des plages des quantités de chaque canal. Ils sont un peu arbitraires dans le sens où on aurait pu dire que chaque canal "vit" dans $[4; 8.5]$ par exemple (au lieu de $[0; 1]$).

Mais 0 et 1 sont pratiques puisque :

1. 0 est l'élément absorbant de l'addition ;
2. 0 est l'élément nul de la multiplication ;
3. 1 est l'élément absorbant de la multiplication.

On a les conséquences suivantes :

1. Le noir est l'unique couleur telle que tout ajout (composante par composante) avec une autre couleur ne change pas cette dernière ;
2. Le noir est l'unique couleur telle que toute multiplication (composante par composante) avec une autre couleur la rend noir ;
3. Le blanc est l'unique couleur telle que toute multiplication avec une autre couleur ne change pas cette dernière.

Nous verrons par la suite que c'est pratique pour les formules de maths utilisées.

2.2. Zeste de Processing

2.2.1. Qu'est-ce que Processing et pourquoi ?

Ce tutoriel est accompagné d'exemples pour pouvoir appliquer les formules vues dans des cas concrets. Processing est un langage de programmation proche du Java accompagné d'un IDE libre disponible sur beaucoup de plateformes. Processing est en partie pensé pour réaliser du traitement d'images.

Processing est donc très pratique pour ce tutoriel : sur n'importe quel OS vous pouvez coder avec l'IDE et suivre le tutoriel.

2.2.2. A savoir sur Processing

1. Lors du lancement de l'application, la fonction `setup()` est appelée ;
2. L'application dispose d'une zone graphique où l'on peut modifier les pixels et afficher ce que l'on veut ;
3. On peut changer les dimensions de cette zone avec un appel à `size( LargeurEnPixels , HauteurEnPixels )` ;
4. Pour modifier les pixels de cette zone graphique, on modifie le tableau unidimensionnel `pixels` où les pixels sont rangés d'abord par ligne (axe x), puis par colonnes (axe y) ;
5. Avant de modifier `pixels`, on les charge en faisant `loadPixels()`
6. Après avoir modifié `pixels`, on les met à jour en faisant `updatePixels()`
7. `save( adresse )` permet de sauvegarder cette zone dans une image dont l'adresse est l'argument ;
8. Les objets `PImage` permettent de charger des images externes, les modifier, et les sauvegarder (cf. exemple)

2.2.3. Exemple de code

Voici un exemple de code Processing, qui effectue simplement une symétrie horizontale d'une image externe :

Contenu masqué n°3

http://dor

! Certaines images dans ce tutoriel sont **stockés dans un format destructifs** (par exemple certaines images en JPG ou les animations en GIF) dans le sens où l'image stockée et l'image en sortie du programme Processing (ce qu'il y a dans le tableau `pixels`) sont légèrement différentes, de sorte à pouvoir réduire la taille du fichier. Si on veut pouvoir réellement étudier l'image de sortie une bonne pratique à prendre en traitement d'images est de **ne pas comprimer l'image de sortie** du programme.

i Les images en PNG de ce tutoriel par contre sont fidèles à la sortie du programme.

2.3. Exo : il n'y a plus de cyan !

2.3.1. Cahier des charges

Cet exercice va mettre en pratique tout ce que l'on a vu dans cette partie. Vous allez devoir écrire des formules de maths et les transcrire en Processing. Voici le cahier des charges du programme Processing :

1. Le programme prend en paramètre l'adresse d'une image "source".
2. Cette image sera modifiée et stockée dans une adresse dite "cible"
3. Chaque pixel de l'image cible est de couleur du même pixel de l'image source mais dont la composante Cyan (du modèle de couleur CMJN) est nulle.
4. Bonus : le programme calcule les proportions nécessaire de chaque encre pour imprimer l'image avant modification (la quantité d'encre divisée par la surface de l'image, en supposant que l'ajout d'encre est proportionnel à la quantité du canal).

Comme vous l'avez compris, ce programme permet de simuler le fait que vous soyez à court d'encre Cyan lorsque vous imprimez l'image source.

2.3.2. Indices

Voici quelques indices si vous n'y arrivez pas :

Indice n°1 :

Contenu masqué n°4

Indice n°2 :

Contenu masqué n°5

2.3.3. Une solution

Voici une solution possible :

Contenu masqué n°6

http://dor

```
proportion d'encre Cyan          : 13.722335 %
proportion d'encre Magenta          : 8.488182 %
proportion d'encre Jaune          : 0.018533962 %
proportion d'encre Noir          : 41.635376 %
```

Contenu masqué

Contenu masqué n°2

$$\min = \text{centre} - \text{rayon} \max = \text{centre} + \text{rayon}$$

Il y a bijection entre $(\min, \max)$ dans $\mathbb{R}^2$ sachant que $\min <= \max$ et $(\text{centre}, \text{rayon}) \in \mathbb{R} \times \mathbb{R}^+$. Retourner au texte.

Contenu masqué n°3

```
// Parametres utilisateurs
String AdresseImageEntree = "/piano.jpg";
String AdresseImageSortie = "/sortie.jpg";

// Variables globales
PImage Image;
int    Largeur;
int    Hauteur;

// Fonction principale
void setup()
{
  // Chargement de l'image en entrée
  Image = loadImage( AdresseImageEntree );
  Largeur = Image.width;
  Hauteur = Image.height;

  // Paramètres de l'application
  size( Largeur , Hauteur );

  // Debut de modification de pixels[]
  loadPixels();

  // Parcours de pixels en Hauteur
```

```
25    for( int y = 0 ; y < Hauteur ; y++ )
26    {
27      // Parcours de pixels en Largeur
28      for( int x = 0 ; x < Largeur ; x++ )
29      {
30        // Operation sur le pixel
31        pixels[ y*Largeur + x ] = Image.pixels[ (Hauteur - 1 -
              y)*Largeur + x ];
32      }
33    }
34
35    // Fin de modification de pixels[]
36    updatePixels();
37
38    // Sauvegarde de l'image modifiée en sortie
39    save( AdresseImageSortie );
40  }
```

Retourner au texte.

Contenu masqué n°4

Vous pouvez créer une classe `CMJN` et une méthode pour obtenir une couleur en `CMJN` depuis une couleur en `RVB`. Retourner au texte.

Contenu masqué n°5

Pour la conversion `RVB` vers `CMJN`, pré-occupez-vous d'abord de convertir en `CMJ` puis occupez-vous enfin de la composante Noir. Retourner au texte.

Contenu masqué n°6

```
 1  // Paramètres utilisateurs
 2  String AdresseImageEntree = "/piano.jpg";
 3  String AdresseImageSortie = "/sortie.jpg";
 4
 5  // Variables globales
 6  PImage Image;
 7  int    Largeur;
 8  int    Hauteur;
 9
10  // Fonction principale
11  void setup()
```

```
12 {
13   // Chargement de l'image en entrée
14   Image = loadImage( AdresseImageEntree );
15   Largeur = Image.width;
16   Hauteur = Image.height;
17 
18   // Paramètres de l'application
19   size( Largeur , Hauteur );
20 
21   // Debut de modification de pixels[]
22   loadPixels();
23 
24   RVB C_RVB = new RVB();
25   CMJN C_CMJN = new CMJN();
26 
27   float proportionCyan = 0.0;
28   float proportionMagenta = 0.0;
29   float proportionJaune = 0.0;
30   float proportionNoir = 0.0;
31   float Surface = Hauteur * Largeur;
32 
33   // parcours de pixels en Hauteur
34   for( int y = 0 ; y < Hauteur ; y++ )
35   {
36     // parcours de pixels en Largeur
37     for( int x = 0 ; x < Largeur ; x++ )
38     {
39       C_RVB.DefinirDepuisColor( Image.pixels[ y*Largeur + x ] );
40       // Operation sur la couleur en RVB du pixel
41 
42       C_CMJN.DefinirDepuisRVB( C_RVB ); // Etape 1 : on convertit
             la couleur en CMJN
43 
44       proportionCyan    += C_CMJN.c;
45       proportionMagenta += C_CMJN.m;
46       proportionJaune   += C_CMJN.j;
47       proportionNoir    += C_CMJN.n;
48 
49       C_CMJN.c = 0.0;                  // Etape 2 : on enlève le
             canal cyan
50       C_RVB.DefinirDepuisCMJN( C_CMJN );// Etape 3 : on reconvertit
             la couleur en RVB
51 
52       // Fin de l'opération
53       pixels[ y*Largeur + x ] = C_RVB.ConvertirColor();
54     }
55   }
56 
57   // Fin de modification de pixels[]
58   updatePixels();
```

```
59
60    // Sauvegarde de l'image modifiée en sortie
61    save( AdresseImageSortie );
62
63    proportionCyan    /= Surface;
64    proportionMagenta /= Surface;
65    proportionJaune   /= Surface;
66    proportionNoir    /= Surface;
67
68    println( "proportion d'encre Cyan \t: "    + proportionCyan    *
        100.0 + " %" );
69    println( "proportion d'encre Magenta \t: " + proportionMagenta *
        100.0 + " %" );
70    println( "proportion d'encre Jaune \t: "   + proportionJaune   *
        100.0 + " %" );
71    println( "proportion d'encre Noir \t: "    + proportionNoir    *
        100.0 + " %" );
72  }
73
74  // Classes
75
76
77  // Classe pour gérer les couleurs
78  // en Rouge Vert Bleu
79  class RVB
80  {
81    float r;
82    float v;
83    float b;
84
85    RVB()
86    {
87      r = 0;
88      v = 0;
89      b = 0;
90    }
91
92    // Conversion color -> RVB
93    RVB DefinirDepuisColor( color C )
94    {
95      r = red  ( C ) / 255.0;
96      v = green( C ) / 255.0;
97      b = blue ( C ) / 255.0;
98
99      return this;
100   }
101
102   // Conversion RVB -> color
103   color ConvertirColor()
104   {
```

```
105     return color(
106         r * 255.0
107       , v * 255.0
108       , b * 255.0
109     );
110   }
111
112
113   // Conversion CMJN -> RVB
114   RVB DefinirDepuisCMJN( CMJN C )
115   {
116     r = ( C.c - C.m - C.j - C.n + 1.0 )/2.0;
117     v = ( C.m - C.c - C.j - C.n + 1.0 )/2.0;
118     b = ( C.j - C.m - C.c - C.n + 1.0 )/2.0;
119
120     return this;
121   }
122 }
123
124 // Classe pour gérer les couleurs
125 // en Cyan Magenta Jaune Noire
126 class CMJN
127 {
128   float c;
129   float m;
130   float j;
131   float n;
132
133   CMJN()
134   {
135     c = 0;
136     m = 0;
137     j = 0;
138     n = 0;
139   }
140
141   // Conversion RVB -> CMJN
142   CMJN DefinirDepuisRVB( RVB C )
143   {
144     c = 1.0 - ( C.v + C.b );
145     m = 1.0 - ( C.r + C.b );
146     j = 1.0 - ( C.r + C.v );
147
148     n = min( c , min( m , j ) );
149
150     c -= n;
151     m -= n;
152     j -= n;
153
154     return this;
```

```
155     }
156 }
```

Retourner au texte.

3. Filtres RVB

Nous allons voir ici comment créer des filtres d'images dans l'espace de couleurs RVB.

On cherche donc des fonctions mathématiques de $[0,1]^3$ dans $[0,1]^3$ (de RVB dans RVB) à insérer dans ce bout de code adapté de l'exo précédent :

Contenu masqué n°7

3.1. Traiter les canaux séparément

Pour commencer on peut chercher des fonctions $f : [0,1] \to [0,1]$ à appliquer sur les canaux séparément : l'opération sur le pixel sera alors $\begin{pmatrix} R \\ V \\ B \end{pmatrix} \mapsto \begin{pmatrix} f(R) \\ f(V) \\ f(B) \end{pmatrix}$.

3.1.1. Fonction `RentrerDans01`

La première préoccupation consiste à toujours ramener les valeurs dans $[0,1]$ (au cas où on a fait n'importe quoi) grâce à la fonction dont voici le graphe :

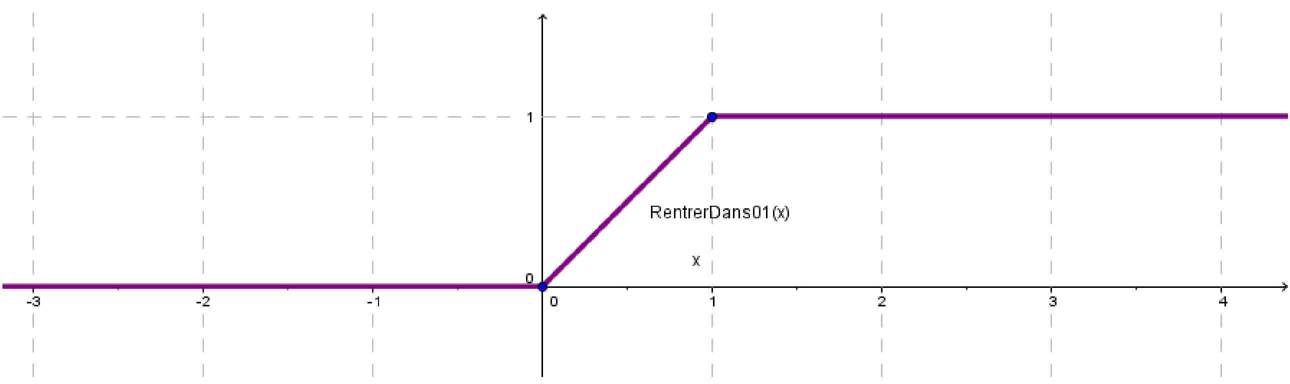


Figure 3.1. – Graphique de la fonction 'RentrerDans01'

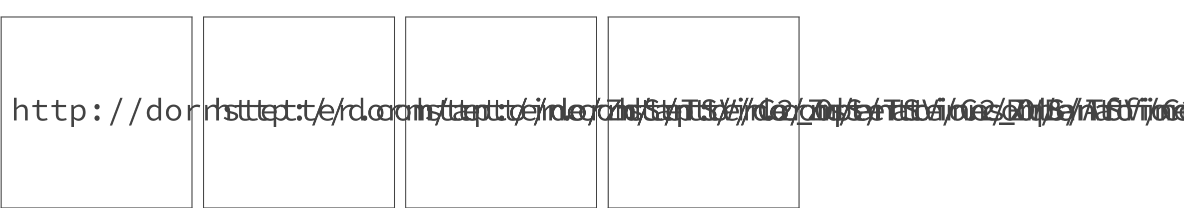
Je vous laisse coder cette fonction en trois parties ! Vous devrez bien évidemment insérer cette fonction dans le code dans la partie "Fonctions utiles".

3.1.2. Fonction affine

On peut par exemple appliquer une fonction affine à chaque canal : $f(x) = ax + b$ avec $(a, b) \in \mathbb{R}^2$. a est le coefficient directeur de la droite et b sa valeur à l'origine.

En particulier si on applique $f(x) = 1 - x$ sur chaque canal on réalise le négatif de l'image.

Contenu masqué n°8



Pour un a positif :

- $a > 1$: éclaircit l'image et plus a augmente, plus l'image sera clair ;
- $a < 1$: assombrit l'image et plus a diminue, plus l'image sera sombre ;
- $b > 0$: éclaircit l'image puisque le noir deviendra gris ;
- $b < 0$: assombrit l'image puisqu'un certain seuil de gris (qui varie selon a) deviendra noir ;

Cette fonction peut provoquer **la perte d'informations** dans l'image i.e. certains pixels de différentes couleurs deviennent identiques et ne peuvent donc plus être différenciés.

3.1.3. Fonction puissance (ou gamma)

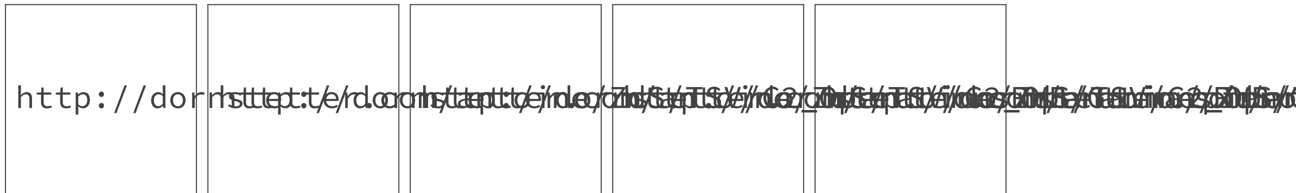
Cette famille de fonctions permet de donner plus d'importance au "1" (c'est-à-dire que les valeurs proches de 0 sont plus éloignées de 0 qu'avant, tandis que les valeurs proches de 1 se rapprochent plus et se confondent), ou au contraire de donner plus d'importance au 0, selon un paramètre g.

FIGURE 3.2. – Graphique animé de la famille de fonctions puissance

Elle s'exprime comme cela : $f_g(x) = x^g$ (lire x à la puissance g), avec $x \in [0;1]$ et $g \in \mathbb{R}^{+*}$ En Processing avec la fonction `pow` : `pow(x,g)`.

Cette fonction est une généralisation des fonctions $x \mapsto x$ ($g = 1$), $x \mapsto x^2$ ($g = 2$), $x \mapsto x^3$ ($g = 3$), $x \mapsto \sqrt{x}$ ($g = 1/2$) à un paramètre continu g.

Contenu masqué n°9



- $g < 1$ éclaircit l'image, plus g diminue, plus l'image sera clair ;
- $g > 1$ obscurcit l'image, plus g augmente, plus l'image sera sombre ;
- La particularité de ces fonctions sont que $f(0) = 0$ et $f(1) = 1$, contrairement aux fonctions affines vues précédemment. Ainsi le blanc reste blanc et le noir reste noir une fois la fonction appliquée.

3.1.4. Postérisation

On peut aussi utiliser une fonction en escalier, de paramètre n, qui *discrétise* les valeurs que peut prendre un canal : les valeurs du canal peuvent prendre seulement n valeurs différentes équitablement espacées.

Voici le graphe d'une fonction en escalier à 8 "marches" ($n = 8$) :

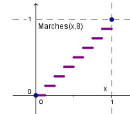


Figure 3.3. – Graphique de la fonction en escalier à 8 marches

Encore une fois à vous de comprendre comment coder cette fonction !

Indice n°1 :

Contenu masqué n°10

Indice n°2 :

Contenu masqué n°11

> **!** Il est fortement déconseillé de lire les indications secrètes. Vous êtes sensé trouver les formules et les implémenter correctement. Le tutoriel a pour but de vous guider à trouver tout le code vous-même. Comprendre ce qu'on code est le plus important !

/opt/zds/data/contents-public/les-espaces-de-c

Figure 3.4. – Graphique animé de la famille des fonctions en escalier en utilisant 'round' pour que $f(1)=1$

Contenu masqué n°12

3.1.5. Valeurs différentes selon les canaux

Enfin toutes les opérations que l'on a vu là peuvent être faites sur les canaux pris séparément : ainsi on peut attribuer des valeurs différentes aux paramètres (les nombres a, b, g, n) par canal. On aura donc par exemple les paramètres a_R, a_V, a_B pour respectivement le paramètre a du canal Rouge, le paramètre a du canal Vert et le paramètre a du canal Bleu.

Par exemple avec une fonction gamma par canal :

Contenu masqué n°13

De façon encore plus générale on peut appliquer différentes fonctions aux canaux. Si on a les fonctions $f, g, h : [0,1] \to [0,1]$ l'opération sur le pixel sera alors : $\begin{pmatrix} R \\ V \\ B \end{pmatrix} \mapsto \begin{pmatrix} f(R) \\ g(V) \\ h(B) \end{pmatrix}$

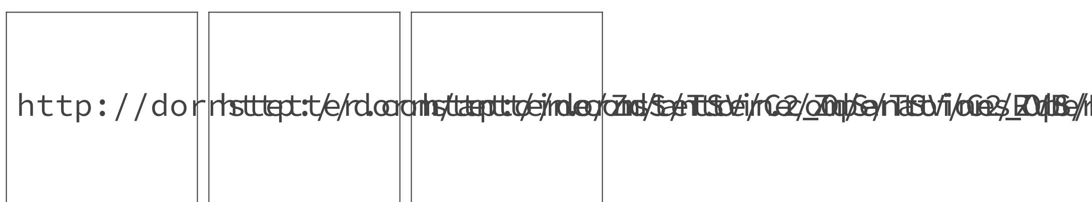
3.2. Opérations sur tous les canaux

Nous allons voir ici des opérations sur les pixels qui s'écrivent comme ceci : $\begin{pmatrix} R \\ V \\ B \end{pmatrix} \mapsto \begin{pmatrix} f(R,V,B) \\ g(R,V,B) \\ h(R,V,B) \end{pmatrix}$.

3.2.1. N&B : Moyenne, min et max

Ici on va chercher à rendre une image monochromatique i.e. toutes les couleurs ont des canaux égaux entre eux (ce qui donne un niveau de gris). Puisqu'on a trois nombre (R,V,B), on peut choisir la moyenne des trois, le minimum des trois ou bien le maximum des trois. Voici les résultats :

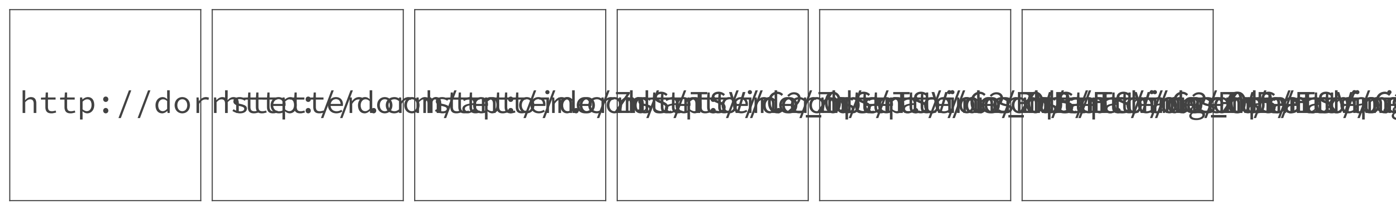
Contenu masqué n°14



Et on peut aussi prendre la moitié de la couleur d'avant et la moitié de la couleur moyenne pour rendre l'image un peu plus N&B sans l'être totalement. Je vous laisse imaginer d'autres opérations .

3.2.2. Échange de canaux

Une idée toute simple consiste à échanger les valeurs des canaux. Par exemple transformer (R, V, B) en (V, B, R) !



L'opération $(R, V, B) \rightarrow (V, B, R)$ par exemple transforme le rouge en bleu, le vert en rouge et le bleu en rouge. Répétée trois fois elle revient à la couleur d'origine.

3.2.3. Combinaisons linéaires

Pour généraliser les fonctions affines, on exprime le canal Rouge par une combinaison linéaire des trois canaux et on ajoute une certaine quantité : $f(R, V, B) = aR + bV + cB + d$. On fait de même pour les deux autres canaux avec des valeurs différents, ce qui nous donne 12 coefficients à déterminer pour appliquer le filtre.

Contenu masqué n°15

Cette formule permet de généraliser les fonctions affines vues précédemment, la moyenne N&B, l'échange de canaux et même la conversion RVB-CMJ. Je vous laisse en exercice trouver les bons coefficients pour retomber sur ces filtres !

3.3. Toutes ces opérations en même temps !

Et enfin vous pouvez vous amuser à appliquer toutes ces opérations à la suite :

Contenu masqué n°16

3.4. Exo : contraste automatique

3.4.1. Enoncé

L'exercice de ce sous-chapitre sera simple. Vous devez remettre tous les canaux de toutes les couleur dans l'intervalle $[0; 1]$. Par exemple si tous les canaux Rouge des couleurs de l'image sont entre 0.2 et 0.9, l'image de sortie aura tous les canaux Rouge entre 0 et 1 : ceux qui étaient égaux à 0.2 seront égaux à 0 et ceux qui étaient égaux à 0.9 seront égaux à 1 après l'application du filtre.

Ceci vaut pour les deux autres canaux indépendamment. Au travail !

3.4.2. Une solution

Contenu masqué n°17

```
R : 0.36078432 , 1.0
V : 0.3529412 , 1.0
B : 0.32941177 , 1.0
```

Quand vous codez des fonctions ou des mini programmes, il est utile de faire des vérifications simples pour s'assurer de leur bon fonctionnement. Par exemple ici la vérification à faire est : si les canaux sont déjà entre 0 et 1, l'image de sortie est **la même** que celle en entrée.

Contenu masqué

Contenu masqué n°7

```
// Parametres utilisateurs
String AdresseImageEntree = "/piano.jpg";
String AdresseImageSortie = "/sortie2.jpg";

// Variables globales
PImage Image;
int    Largeur;
int    Hauteur;

// Fonction principale
void setup()
{
  // Chargement de l'image en entrée
  Image = loadImage( AdresseImageEntree );
  Largeur = Image.width;
  Hauteur = Image.height;

  // Paramètres de l'application
  size( Largeur , Hauteur );

  // Debut de modification de pixels[]
  loadPixels();

  RVB C_RVB = new RVB();
  CMJN C_CMJN = new CMJN();

  // Parcours de pixels en Hauteur
  for( int y = 0 ; y < Hauteur ; y++ )
  {
    // Parcours de pixels en Largeur
    for( int x = 0 ; x < Largeur ; x++ )
    {
      C_RVB.DefinirDepuisColor( Image.pixels[ y*Largeur + x ] );

      // Réaliser ici l'opération sur la couleur en RVB du pixel

      pixels[ y*Largeur + x ] = C_RVB.ConvertirColor();
    }
  }

  // Fin de modification de pixels[]
  updatePixels();

  // Sauvegarde de l'image modifiée en sortie
```

```
45     save( AdresseImageSortie );
46   }
47 
48   // Fonctions utiles
49 
50   // Classes
51 
52   // Classe pour gérer les couleurs
53   // en Rouge Vert Bleu
54   class RVB
55   {
56     float r;
57     float v;
58     float b;
59 
60     RVB()
61     {
62       r = 0;
63       v = 0;
64       b = 0;
65     }
66 
67     // Conversion color -> RVB
68     RVB DefinirDepuisColor( color C )
69     {
70       r = red   ( C ) / 255.0;
71       v = green( C ) / 255.0;
72       b = blue ( C ) / 255.0;
73 
74       return this;
75     }
76 
77     // Conversion RVB -> color
78     color ConvertirColor()
79     {
80       return color(
81           r * 255.0
82         , v * 255.0
83         , b * 255.0
84       );
85     }
86   }
```

Retourner au texte.

Contenu masqué n°8

```
// Operation sur la couleur en RVB du pixel
C_RVB.r = RentrerDans01( a * C_RVB.r + b );
C_RVB.v = RentrerDans01( a * C_RVB.v + b );
C_RVB.b = RentrerDans01( a * C_RVB.b + b );
```

Retourner au texte.

Contenu masqué n°9

```
// Operation sur la couleur en RVB du pixel
C_RVB.r = pow( C_RVB.r, g );
C_RVB.v = pow( C_RVB.v, g );
C_RVB.b = pow( C_RVB.b, g );
```

Retourner au texte.

Contenu masqué n°10

En Processing `int(x)` permet de récupérer la partie entière de `x`, c'est-à-dire le plus grand entier inférieur ou égale à `x`. Retourner au texte.

Contenu masqué n°11

Vous pouvez appliquer "l'algorithme en 3 étapes" vu au sous-chapitre précédent à ce problème. A vous de trouver les espaces en question.. Retourner au texte.

Contenu masqué n°12

```
// Operation sur la couleur en RVB du pixel
C_RVB.r = Marches( C_RVB.r , n );
C_RVB.v = Marches( C_RVB.v, n );
C_RVB.b = Marches( C_RVB.b, n );
...
// Fonctions utiles
float Marches( float x , int n )
{
  return round( x * (n - 1.0) ) / (n - 1.0);
```

```
10 }
```

Retourner au texte.

Contenu masqué n°13

```
1 float ga = 1.5;
2 float gb = 0.8;
3 float gc = 0.4;
4 ...
5 // Operation sur la couleur en RVB du pixel
6 C_RVB.r = pow( C_RVB.r , ga );
7 C_RVB.v = pow( C_RVB.v , gv );
8 C_RVB.b = pow( C_RVB.b , gb );
```

Retourner au texte.

Contenu masqué n°14

```
1  // Operation sur la couleur en RVB du pixel
2
3  if( action == 0 )
4  {
5    z = ( C_RVB.r + C_RVB.v + C_RVB.b ) / 3.0;
6  }
7  else if( action == 1 )
8  {
9    z = min( C_RVB.r , min( C_RVB.v , C_RVB.b ) );
10 }
11 else
12 {
13   z = max( C_RVB.r , max( C_RVB.v , C_RVB.b ) );
14 }
15
16 C_RVB.Definir( z , z , z );
```

Retourner au texte.

Contenu masqué n°15

```
float[][] M = new float[][]{
    new float[]{ 1.0 , 0.2 , 1.4 }
  , new float[]{ 0.2 , -0.8 , 0.2 }
  , new float[]{ 1.1 , 1.0 , -1.3 }
};

RVB CouleurNoir = new RVB( 0.3 , 0.8 , 1.6 );
...
// Operation sur la couleur en RVB du pixel
r = C_RVB.r;
v = C_RVB.v;
b = C_RVB.b;

C_RVB.r = RentrerDans01( M[0][0]*r + M[0][1]*v + M[0][2]*b +
    CouleurNoir.r );
C_RVB.v = RentrerDans01( M[1][0]*r + M[1][1]*v + M[1][2]*b +
    CouleurNoir.v );
C_RVB.b = RentrerDans01( M[2][0]*r + M[2][1]*v + M[2][2]*b +
    CouleurNoir.b );
```

Retourner au texte.

Contenu masqué n°16

```
// Operation sur la couleur en RVB du pixel

C_RVB.r = Marches( C_RVB.r , 4 );
C_RVB.v = Marches( C_RVB.v , 5 );
C_RVB.b = Marches( C_RVB.b , 6 );

C_RVB.v += C_RVB.r;
C_RVB.v /= 2.0;

C_RVB.r = pow( C_RVB.r , 1.6 );
C_RVB.v = pow( C_RVB.v , 0.8 );
C_RVB.b = pow( C_RVB.b , 1.9 );

C_RVB.r = RentrerDans01( 3.0*C_RVB.r + 0.1 );
C_RVB.v = RentrerDans01( C_RVB.v );
C_RVB.b = RentrerDans01( 1.5*C_RVB.b );
```

Retourner au texte.

Contenu masqué n°17

```
// Parametres utilisateurs
String AdresseImageEntree = "/piano2.jpg";
String AdresseImageSortie = "/piano2_contraste_automatique.jpg";

float min = 0.0;
float max = 1.0;

// Variables globales
PImage Image;
int    Largeur;
int    Hauteur;

// Fonction principale
void setup()
{
  // Chargement de l'image en entrÃ©e
  Image = loadImage( AdresseImageEntree );
  Largeur = Image.width;
  Hauteur = Image.height;

  // ParamÃ¨tres de l'application
  size( Largeur , Hauteur );

  // Debut de modification de pixels[]
  loadPixels();

  RVB C_RVB = new RVB();

  float minR = 1.0;
  float maxR = 0.0;

  float minV = 1.0;
  float maxV = 0.0;

  float minB = 1.0;
  float maxB = 0.0;

  // Premier parcours des pixels
  for( int y = 0 ; y < Hauteur ; y++ )
  {
    for( int x = 0 ; x < Largeur ; x++ )
    {
      C_RVB.DefinirDepuisColor( Image.pixels[ y*Largeur + x ] );

      minR = min( C_RVB.r , minR );
      maxR = max( C_RVB.r , maxR );

```

```
48        minV = min( C_RVB.v , minV );
49        maxV = max( C_RVB.v , maxV );
50
51        minB = min( C_RVB.b , minB );
52        maxB = max( C_RVB.b , maxB );
53
54      }
55    }
56
57    println( "R : " + minR + " , " + maxR );
58    println( "V : " + minV + " , " + maxV );
59    println( "B : " + minB + " , " + maxB );
60
61    // Second parcours des pixels
62    for( int y = 0 ; y < Hauteur ; y++ )
63    {
64      for( int x = 0 ; x < Largeur ; x++ )
65      {
66        C_RVB.DefinirDepuisColor( Image.pixels[ y*Largeur + x ] );
67        C_RVB.r = IntervalABdansCD( C_RVB.r , minR , maxR , min , max
              );
68        C_RVB.v = IntervalABdansCD( C_RVB.v , minV , maxV , min , max
              );
69        C_RVB.b = IntervalABdansCD( C_RVB.b , minB , maxB , min , max
              );
70        pixels[ y*Largeur + x ] = C_RVB.ConvertirColor();
71      }
72    }
73
74    // Fin de modification de pixels[]
75    updatePixels();
76
77    // Sauvegarde de l'image modifiÃ©e en sortie
78    save( AdresseImageSortie );
79  }
```

Retourner au texte.

4. De TSV à RVB

Le modèle de couleurs RVB est assez limité, avec les filtres que l'on a créé on avait du mal à gérer la teinte des images. En échangeant les canaux par exemple on pouvait transformer une image à dominante rouge en une image à dominante bleue. Mais les modifications de teintes en RVB sont assez limitées : on ne pouvait pas la rendre un peu moins rouge et un peu plus orange pour continuer l'exemple. De plus, la gestion du noir & blanc était un peu empirique.

On va donc s'intéresser dans les chapitres qui suivent au modèle TSV qui va nous permettre de créer de meilleurs filtres plus simplement.

Dans ce chapitre le but va être de programmer une fonction qui **à partir d'une couleur en TSV, la transforme en couleur exprimée dans l'espace de couleurs RVB**. Nous allons commencer par créer une simple roue des couleurs ; ensuite nous verrons les notions de Saturation et de Valeur.

4.1. Teintes "pures"

On cherche à générer un dégradé de toutes les couleurs *pures*. On appelle couleur pure la couleur que l'on voit lorsque la lumière est monochromatique i.e. il n'y a qu'un seul type de photon avec une seule longueur d'onde présent dans ses faisceaux. La couleur perçue n'est pas mélangée avec d'autres couleurs.

On cherche donc à recréer.. l'arc en ciel ! Ou encore ce que l'on voit lors du passage de la lumière blanche à travers un prisme :

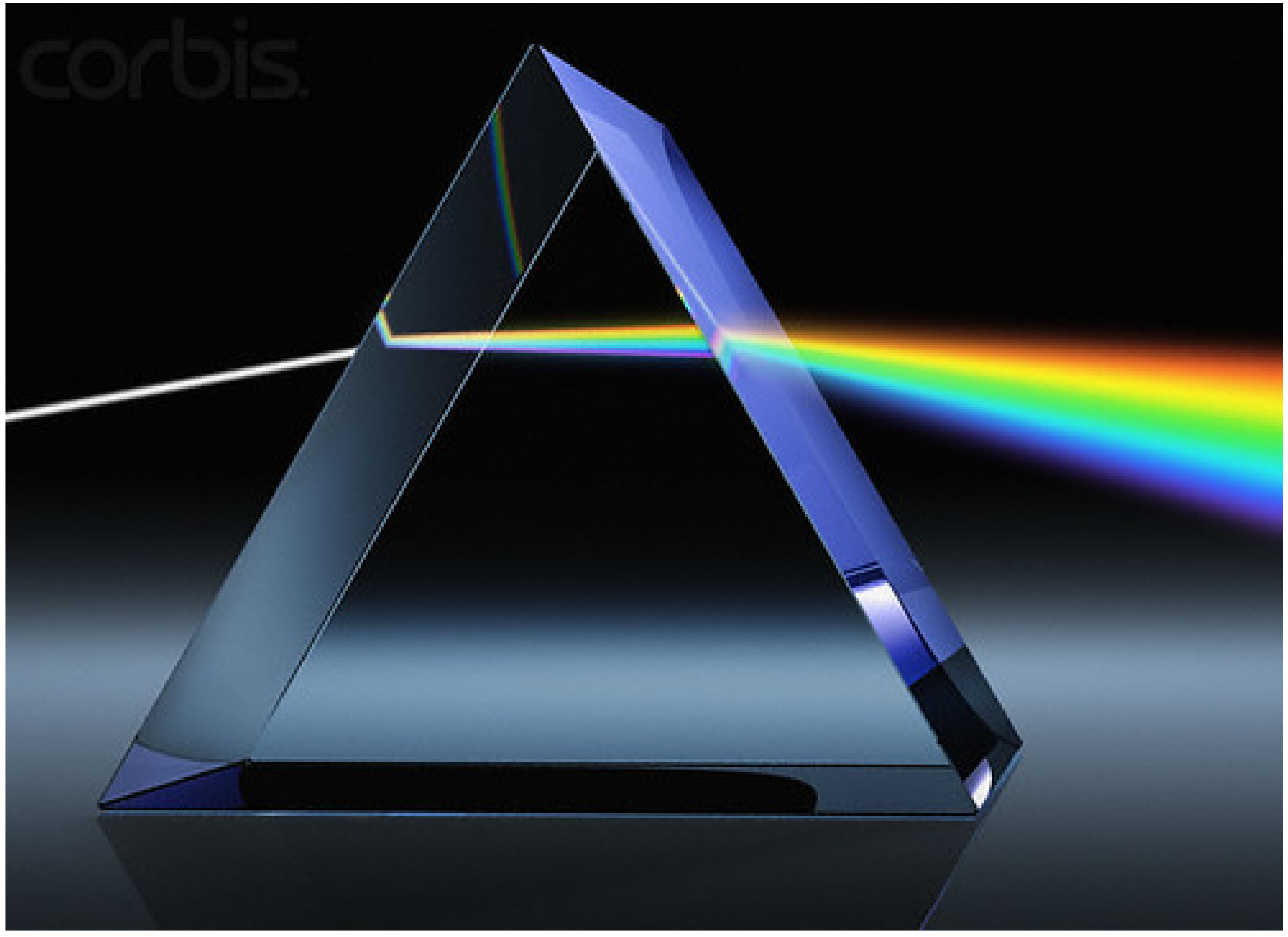


Figure 4.1. – Diffraction de la lumière par un prisme

Vous pouvez apercevoir sur cette photo que l'arc-en-ciel est un dégradé entre les trois couleurs primaires et secondaires alternées du modèle RVB.

Nous allons essayer de recréer ce dégradé, c'est-à-dire trouver une fonction qui à partir d'un réel nous donne une des couleurs de l'arc-en-ciel **de façon aussi précise que l'on veut**.

> Pour résumer le plus important, un des canaux dans notre nouvel espace de couleurs sera **la Teinte** :
> — qui indique l'emplacement de la couleur sur l'arc-en-ciel et permettra directement de catégoriser la couleur parmi Rouge Jaune Vert Cyan Bleu Magenta,
> — et qui n'est défini que si la couleur **n'est pas noire** (puisqu'à ce moment là il n'y a pas de lumière émise)
> — et que si la couleur **n'est pas blanche** puisque la lumière blanche est composée de toutes les teintes.

4.2. Recréer l'arc-en-ciel

4.2.1. Barycentre entre deux couleurs

Avant de continuer, il faut savoir réaliser une fonction de barycentre entre deux valeurs réelles a et b. Voici les hypothèses que l'on pose pour trouver cette fonction :

1. Cette fonction de la variable x est affine en fonction de x
2. Quand x vaut 0 la fonction vaut a
3. Quand x vaut 1 la fonction vaut b

Voici les transcriptions mathématiques de ces hypothèses :

Contenu masqué n°18

Voici une solution au problème si vous n'avez toujours pas trouvé :

Contenu masqué n°19

4.2.2. Coder une fonction d'arc-en-ciel

Pour recréer l'arc-en-ciel on va donc tout simplement réaliser 6 barycentres entre les couleurs primaires et secondaires (de façon alternative). Par exemple entre le Rouge et le Vert se trouve le Jaune. L'arc-en-ciel débute donc par un dégradé (comprendre mathématiquement un barycentre) entre le Rouge et le Jaune, puis continue avec un dégradé entre le Jaune et le Vert.

La même logique continue :

Contenu masqué n°20

!

Ne pas oublier le dernier dégradé qui permet de revenir à la couleur initiale. Je rappelle que la teinte est une information circulaire, comme un angle qui peut s'exprimer de façon unique entre 0 et 360° non inclus. Le fait que le rouge débute l'arc-en-ciel est **arbitraire**, vous faites ce que vous voulez de ce point de vue-là. Dans la suite du tutoriel, par convention, le Rouge correspondra à une teinte de 0.

4.2.2.1. Intervalle de Teinte

Il ne vous reste plus qu'à décider des bornes dans lesquels l'information de Teinte va se situer. Cette borne est arbitraire mais il est important de savoir qu'elle change la précision de l'information de teinte.

Par exemple la teinte peut-être entre 0 et 1 (pour faire comme les autres variables), entre 0 et 6 (puisqu'il y a RJVCBM), entre 0 et 2π ou 360 (si on fait l'analogie avec un angle exprimé en radians ou en degrés)...

4.2.2.2. Algorithme final

L'algorithme final consiste simplement à repérer entre quelles couleurs il faut réaliser un dégradé. Pour cela, on divise notre intervalle en 6 intervalles, on **sélectionne** les deux couleurs avec lesquelles il faut faire un dégradé, on **calcule le coefficient** du barycentre. On a plus qu'à appliquer la fonction de barycentre avec le coefficient et les deux couleurs choisies.

Pour sélectionner les deux couleurs avec lesquelles faire le dégradé, je vous conseille de trouver la première en s'inspirant de la fonction en escalier du sous-chapitre précédent. La seconde couleur est naturellement la suivante parmi RJVCBM.

Ne pas oublier le dernier dégradé qui est celui du Magenta au Rouge.

Vous avez maintenant tout ce qu'il vous faut pour produire l'algorithme final qui permet **d'avoir une couleur pure à partir de son information de Teinte**.

4.3. Exo : générer une roue des couleurs

4.3.1. Ce qui est demandé

Maintenant que vous avez programmé votre fonction de Teinte, vous pouvez vous amuser à générer une roue des couleurs : cela permettra en plus de vérifier que vous ne vous êtes pas trompé !

Pour cela, vous avez juste besoin de changer la teinte en fonction de l'angle du pixel par rapport au centre de l'image.

Contenu masqué n°21

Indice : | La fonction en Processing `atan2(y,x)` permet de récupérer l'angle modulo du point (x,y) par rapport au point (0,0).

4.3.2. Une solution pour la roue des couleurs en Processing

Voici une solution possible de la roue des couleurs en Processing si votre fonction de teinte est `RVB RVB_Depuis_Teinte(float teinte){...}` :

Contenu masqué n°22

http://dornstetter.com/antoine/ZdS/TSV/C3_TSVa

FIGURE 4.2. – Roue des couleurs

Si vous ne trouvez pas le même type d'image (si elle est simplement pivoté c'est bon), vous vous êtes peut-être trompé dans l'algorithme de roue des couleurs.

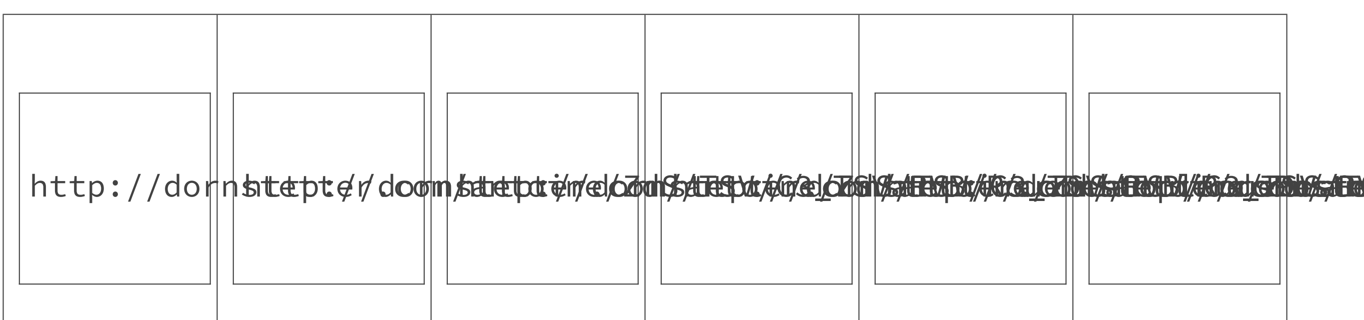
Je vous invite alors à déboguer votre programme et à réfléchir encore une fois aux formules de maths vues depuis le début du tutoriel.

> Je ne vais pas vous donner le code pour `RVB_Depuis_Teinte`. C'est à vous de le coder ! Vous avez toutes les indications dans ce tutoriel pour y arriver .
> Dans la suite du tutoriel, toutes les fonctions que vous devez coder vous même (classe RVB, TSV & méthodes de conversion) se trouveront dans le fichier `RVBTSV.pde` qui doit être placé dans le même dossier que votre application pour que celle-ci fonctionne. Je n'ai pas uploadé ce fichier sur mon site : les exemples que vous téléchargez ne peuvent pas marcher tels quels.

4.3.3. La roue des couleurs postérisée !

Vous pouvez dès maintenant créer une roue des couleurs "discrète" en postérisant la teinte :

1 et 7 marches	2 et 8 marches	3 et 12 marches	4 et 16 marches	5 et 32 marches	6 et 64 marches
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]



Ici les images pivotent puisque la postérisation dépend aussi d'où "commencent" les marches. Je rappelle que la valeur de la teinte à 0 est arbitraire ; ces animations montrent donc pour chaque valeur de la teinte à l'origine ce que la postérisation de la teinte donne.

Sur la roue des couleurs à 2 marches, vous pouvez apercevoir les couleurs opposées. Sur la roue des couleurs à 3 marches celles qui sont à 120° les unes des autres (par exemple les couleurs primaires sont à 120° des couleurs secondaires et vice-versa).

4.4. Saturation et valeur

Pour finir de décrire l'espace TSV nous devons nous pencher sur les notions de Valeur et de Saturation.

4.4.1. Qu'est-ce que la Valeur ?

La Valeur permet d'**assombrir une couleur** : une valeur de 0 (le minimum) rend toute couleur Noire (absence de lumière). Les couleurs qui sont les plus lumineuses ont une valeur de 1 (le maximum), comme les couleurs générées par l'algorithme de roue des couleurs vu dans le chapitre précédent.

La Valeur *ressemble* à la luminosité et est parfois confondue avec celle-ci, puisqu'une valeur plus grande indique une couleur plus lumineuse et vice-versa. Cependant les imperfection du modèle TSV font que ce sont des grandeurs différentes. Le modèle TSV présenté dans ce tutoriel ne doit pas être confondu avec le modèle TSL (Teinte Saturation Luminosité).

4.4.2. Qu'est-ce que la Saturation ?

Comme les deux sortes de robinets qui disposent tous les deux de deux grandeurs indépendantes, nos espaces de couleurs nécessitent *trois degrés de libertés* pour pouvoir exprimer toutes les combinaisons de couleurs possibles.

Le modèle TSV manque d'une troisième quantité pour pouvoir décrire l'ensemble des couleurs prises par le modèle RVB (puisque le TSV *dérive* du RVB).

Remarquez qu'on ne peut pas encore décrire une image en niveau de gris avec notre modèle Teinte-Valeur. Cette troisième quantité est la Saturation et permet de décrire la "distance" de la couleur T-V au niveau de gris pris par la Valeur.

> Le niveau de gris donné par son (unique) indication de Valeur est $(R, V, B) = (Valeur, Valeur, Valeur)$.

4.4.3. Exemples avec des images !

Opération	piano.jpg	normandie.jpg
Saturation moyenne et écart-type	$29\% \pm 30\%$	$28\% \pm 20\%$
Valeur moyenne et écart-type	$32\% \pm 30\%$	$50\% \pm 35\%$
Image d'origine	http://dornst[illegible]	[illegible]
Saturation divisée par 2	http://dornst[illegible]	[illegible]
Saturation multipliée par 2	http://dornst[illegible]	[illegible]

Image d’origine		
Valeur divisée par 2		
Valeur multipliée par 2		

4.4.4. Encore des barycentres !

Pour implémenter la Valeur et la Saturation nous allons encore avoir besoin des barycentres ! Et encore une fois vous allez devoir trouver vous-même !

Je vais vous pré-mâcher une dernière fois le travail :

Contenu masqué n°23

4.4.5. Vérifications mathématiques de la formule finale

Comme d’habitude, voici quelques vérifications que vous devez faire sur la formule que vous venez de trouver (sachant que l’on applique la formule sur une couleur pure) :

1. Saturation $= 0$ donne un gris, l’information de teinte est perdue (peu importe la *Teinte* la couleur est la même) ;
2. Saturation $= 1$ donne toujours une des trois composantes RVB qui est nulle ;
3. Saturation réalise bien une transformation linéaire vers $Valeur$ lorsqu’on fixe $Valeur$;
4. Valeur $= 0$ donne du noir, les informations de teinte et de saturation sont perdues ;

5. Valeur = 1 donne toujours une des trois composantes RVB qui est maximale ;
6. Valeur réalise bien une transformation linéaire vers le noir lorsqu'on fixe *Saturation*.

4.5. TSV vers RVB

4.5.1. Au travail !

Maintenant vous devez simplement mettre toutes vos formules dans une méthode de classe qui permet de calculer une couleur en RVB depuis une couleur en TSV.

4.5.2. Vérifications avec deux roues des couleurs

Voici deux images pour vérifier que nous avons bien codé la même chose :

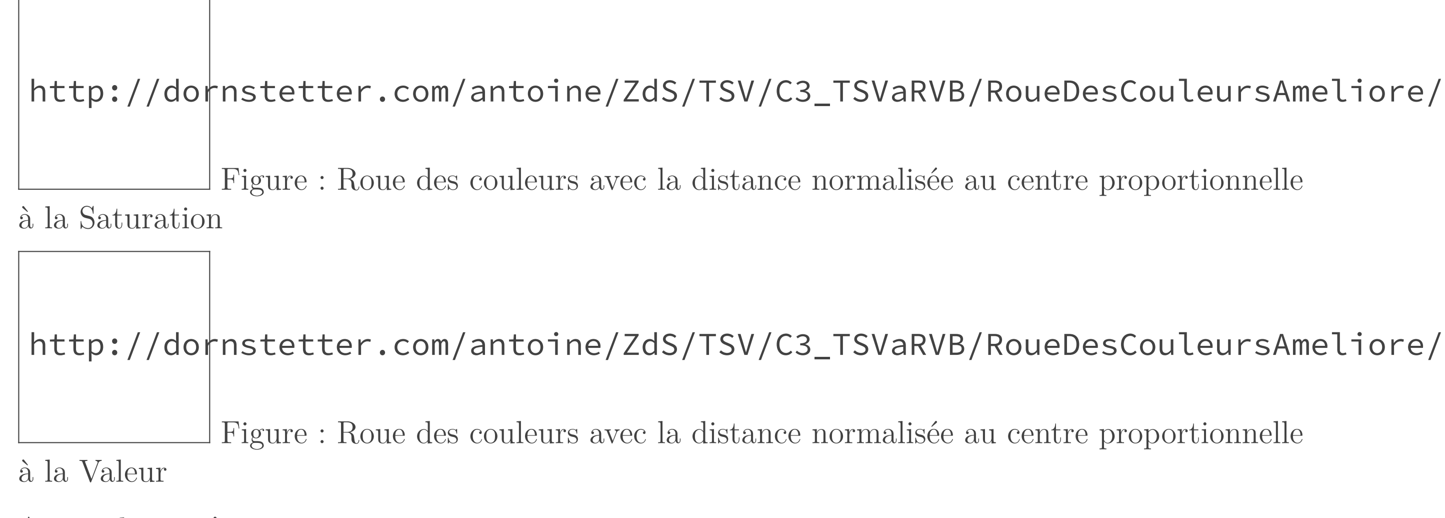


Figure : Roue des couleurs avec la distance normalisée au centre proportionnelle à la Saturation

Figure : Roue des couleurs avec la distance normalisée au centre proportionnelle à la Valeur

A vos claviers !

4.5.2.0.1. Une solution

Contenu masqué n°24

4.6. Exo : carrés de couleurs aléatoires

4.6.1. Ce qui est demandé

Votre tâche est de créer un programme qui génère une image carrée composée de carrés dont la couleur est aléatoire dans l'espace TSV, chaque composante étant tirée au hasard dans un certain intervalle paramétrable. Vous pouvez lire la documentation officielle Processing au cas où il vous manquerait une fonction.

4.6.2. Une solution

Contenu masqué n°25

4.6.3. Quelques rendus

	Rendu 1	Rendu 2	Rendu 3	Rendu 4
Rendu	http://dornst[illegible]	[illegible]	[illegible]	[illegible]
Teinte $\in$	$[0;1]$	$[0;1]$	$[0.05;0.15]$	$[0.4;0.8]$
Saturation $\in$	$[0;1]$	$[0.25;0.75]$	$[0.5;1.0]$	$[0.2;0.7]$
Valeur $\in$	$[0;1]$	$[0.5;1.0]$	$[0.4;0.9]$	$[0.3;0.8]$

Essayez de faire la même chose dans l'espace RVB avec autant de facilité .

Contenu masqué

Contenu masqué n°18

1. il existe deux coefficients c et d tels que $f(x) = cx + d$
2. $f(0) = a$
3. $f(1) = b$

Retourner au texte.

Contenu masqué n°19

Il faut remplacer $f(0)$ et $f(1)$ par son expression affine et les équations nous donnent : $c = b - a$ ainsi que $d = a$.

D'où $f(x) = (1-x)a + xb$ .. **C'est la formule du barycentre entre a et b !**

Retourner au texte.

Contenu masqué n°20

L'arc-en-ciel se poursuit avec un dégradé du Vert vers le Cyan, puis du Cyan vers le Bleu, puis du Bleu vers le Magenta et enfin du Magenta vers le Rouge. Retourner au texte.

Contenu masqué n°21

Retourner au texte.

Contenu masqué n°22

```
pixels[ y*Largeur + x ] =
  RVB_Depuis_Teinte(
    atan2(
        float(y) - Hauteur/2.0
      , float(x) - Largeur/2.0
    ) / TWO_PI
  ).ConvertirColor();
```

Retourner au texte.

Contenu masqué n°23

Il y a plusieurs façons de trouver la formule finale : soit vous commencez par appliquer la Valeur puis la Saturation, ou le contraire :

1. La Saturation est le coefficient d'un barycentre entre la couleur pure et sa Valeur ;
2. La Valeur est le coefficient d'un barycentre entre le Noir et la couleur à laquelle on a déjà appliqué la Saturation.

Retourner au texte.

Contenu masqué n°24

```
dx = float(x)/Largeur - 0.5;
dy = float(y)/Hauteur - 0.5;

C_RVB.DefinirDepuisTSV(
  new TSV(
      atan2( dy , dx ) / TWO_PI
    , sqrt( dx*dx + dy*dy )
```

```
        , 1.0
    )
);
```

Retourner au texte.

Contenu masqué n°25

```
// Parametres utilisateurs
String AdresseImageSortie = "/CouleursAleatoires_4.png";
int NbCarresEnLargeur = 32;
int CoteCarreEnPixels = 6;

float Teinte_min = 0.4;
float Teinte_max = 0.8;

float Saturation_min = 0.2;
float Saturation_max = 0.7;

float Valeur_min = 0.3;
float Valeur_max = 0.8;

// parcours des carrés en Hauteur
for( int y = 0 ; y < NbCarresEnLargeur ; y++ )
{
  // parcours des carrés en Largeur
  for( int x = 0 ; x < NbCarresEnLargeur ; x++ )
  {
    C_TSV.Definir(
        random( Teinte_min     , Teinte_max )
      , random( Saturation_min , Saturation_max )
      , random( Valeur_min     , Valeur_max )
    );

    C_RVB.DefinirDepuisTSV( C_TSV );
    C_color = C_RVB.ConvertirColor();

    // parcours des pixels du carré en Hauteur
    for( int dy = 0 ; dy < CoteCarreEnPixels ; dy++ )
    {
      // parcours des pixels du carré en Largeur
      for( int dx = 0 ; dx < CoteCarreEnPixels ; dx++ )
      {
        pixels[ ( y*CoteCarreEnPixels + dy )*Largeur +
            x*CoteCarreEnPixels + dx ] = C_color;
      }
```

```
38         }
39     }
40 }
```

Retourner au texte.

5. Filtres TSV

Nous cherchons maintenant à créer des filtres en TSV.

Mais pour cela il nous manque une fonction essentielle : celle qui à partir d'une couleur RVB renvoie cette même couleur exprimée dans l'espace TSV.

5.1. RVB vers TSV

> Voici l'algorithme le plus "dur" et le moins assisté du tutoriel ! Une fois ce sous-chapitre passé vous allez pouvoir vous amuser .

5.1.1. Renverser les étapes de l'algorithme

L'algorithme d'avant consistait à sélectionner une couleur sur la roue des couleurs, puis d'appliquer une saturation et une valeur. Nous allons donc aller dans l'autre sens :

5.1.1.1. 1) Étudier la formule de la Valeur et de la Saturation

Vous devez étudier les barycentres que vous avez écrit pour retrouver les variables de Saturation et de Valeur (qui sont désormais inconnues) à partir des autres qui sont maintenant connes.

Vous allez devoir raisonner de la même façon que le passage du mitigeur au mélangeur en essayant de renverser les formules trouvées auparavant.

Les formules que vous trouvez doivent faire ressortir le fait qu'échanger les canaux R,V,B ne modifie pas la Saturation et la Valeur.

Indice :

Contenu masqué n°26

5.1.1.2. 2) Modifier R,V,B

Il faut maintenant modifier les variables R, V et B pour retrouver une couleur pure. On aura alors Saturation = 1 et Valeur = 1

Indice n°1 :

Contenu masqué n°27

Indice n°2 :

Contenu masqué n°28

5.1.1.3. 3) Renverser la fonction de Teinte

Enfin, vous n'avez plus qu'à trouver la Teinte d'une couleur à partir de sa position sur la roue des couleurs pures !

Indice :

Contenu masqué n°29

5.1.2. Vérification élémentaire

Je vous laisse trouver des tests pour vos formules. En attendant, une vérification élémentaire s'impose : convertir une couleur RVB en TSV, puis la re convertir en RVB et vérifier qu'elle est identique. Vous pouvez appliquer cela pour tous les pixels d'une image.

Contenu masqué n°30

http://dornstetter.com/antoine/ZdS/TSV/C4_Oper

L'image de sortie est sensée **être identique** ! Ce test permet de montrer visuellement que les deux espaces de couleurs sont en bijection !

> Vous pouvez aussi faire la différence entre les deux images et vérifier que l'on obtient bien une image complètement noire.

5.2. Précautions à prendre

Vous devez prendre quelques précautions avant de manipuler des couleurs dans l'espace TSV.

5.2.1. Teinte

Je le rappelle encore une fois, que la Teinte est **une grandeur circulaire**. Vous ne pouvez donc pas la traiter comme les autres informations : **toute information de Teinte est à considérer modulo 1** (ou l'amplitude que vous avez choisi comme intervalle de définition de la teinte si ce n'est pas $[0;1]$).

5.2.2. Saturation et Valeur

D'autres précautions à prendre sont relatives à notre définition de la Saturation et de la Valeur : **un niveau de gris ne contient pas d'information de Teinte**. Donc :

1. Faites attention lorsqu'à la sortie d'une transformation TSV la Saturation est devenue nulle (niveau de gris) : **l'information de Teinte n'a plus de sens**.
2. Faites attention lorsqu'à l'entrée d'une transformation TSV la Saturation est nulle (niveau de gris) : **l'information de Teinte n'a pas de sens**.

> Il est préférable dans votre code de prévoir le cas où la Teinte n'est pas définie (par exemple utiliser le `NaN`). Il est aussi pratique de faire en sorte que la couleur noire aie une Saturation égale à 0.

5.2.3. Bijection entre TSV et RVB

En fait on n'a pas vraiment démontrer que TSV et RVB étaient en bijection puisqu'on n'avait pas déterminé les ensembles qui l'étaient.

Le paragraphe précédent justifie le fait que l'on enlève tous les niveaux de gris (blanc et noir inclus!) pour trouver deux sous-espaces du RVB et du TSV qui sont en bijection. Ainsi les ensembles en bijection sont :

— pour le RVB $(r,v,b) \in [0;1]^3 \backslash \{r=v=b\}$: la dernière condition enlevant les gris,
— pour le TSV $(t,s,v) \in [0;1[\times]0;1]\times]0;1]$: $t=1$ représente la même couleur que $t=0$, on enlève les gris (dans la saturation) et le noir (dans la valeur).

5.3. Canaux Saturation et Valeur

Dans ce sous-chapitre nous allons voir qu'est-ce qu'on peut modifier sur les canaux de Saturation et de Valeur. Mais avant, une petite digression mathématique .

5.3.1. Une autre façon de voir les fonctions affines

Au lieu d'appliquer $f(x) = ax+b$, vous pouvez appliquer une homothétie 1D : $f(x) = k(x-c)+c$ où k est le coefficient de dilatation ($k = 1$ donne la fonction identité, $k < 0$ donne une symétrie par rapport à c) et c le centre de l'homothétie (la quantité invariante de notre fonction).

Par exemple $f(x) = 2(x - 0.5) + 0.5$ transforme l'intervalle $[0.25; 0.75]$ en $[0; 1]$ en laissant 0.5 identique. Autre exemple, la fonction pour réaliser le négatif d'un canal, $f(x) = 1 - x = -1(x - 0.5) + 0.5$ est en réalité une symétrie par rapport à 0.5.

Exemple avec une homothétie de rapport 3 sur la Valeur et dont le centre parcours $[0; 1]$:

Ceci vaut bien évidemment aussi pour l'espace RVB.

Vous pouvez coder ces filtres dès maintenant.

5.3.2. Refaire un exemple vu

Ensuite vous pouvez refaire l'exemple du sous-chapitre I.3.4 où on a multiplié ces canaux par un coefficient ainsi que calculer la valeur moyenne de ces canaux !

5.3.3. Gamma

Vous pouvez aussi appliquer une fonction gamma sur ces canaux :

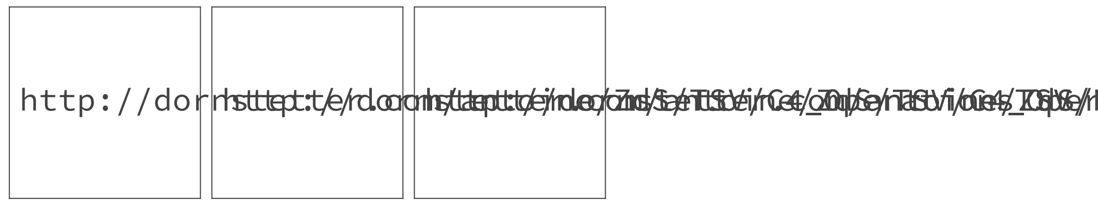
A gauche le gamma parcourt $[0.3; 2.0]$ sur la Saturation à gauche et parcourt $[0.3; 2.0]$ à droite sur la Valeur.

5.3.4. Forcer les canaux

Vous pouvez carrément forcer ces canaux à une certaine quantité.

Ici la Saturation parcourt $[0; 1]$:

Ici la Saturation (à gauche) et la Valeur (à droite) parcourent $[0; 1]$:



5.4. Canal Teinte

Dans ce sous-chapitre nous allons voir ce que l'on peut faire avec le canal Teinte.

5.4.1. Rotation

Pour le canal Teinte vous pouvez effectuer une rotation, ce qui revient simplement à rajouter une quantité.

> Ceci permet de généraliser l'échange de canaux. Rajouter $\pm 1/3$ de votre intervalle de Teinte transforme (R, V, B) en (B, R, V) ou (V, B, R). Rajouter la moitié de l'intervalle transforme toute couleur primaire en couleur secondaire (opposée sur la roue des couleurs) et vice-versa.

5.4.2. Homothétie et symétrie

Voici ce que donne une symétrie ($k = -1$) et dont le centre parcourt la Roue des couleurs :

Voici ce que donne une homothétie avec $k = 4$ dont le centre parcourt la Roue des couleurs :

5.4.3. Forcer la teinte

Vous pouvez redéfinir la teinte de tous les pixels de l'image à une certaine valeur.

C'est un joli effet qui permet à l'image finale d'être le mélange de Noir, de Blanc, ainsi que de la couleur pure associée à la teinte choisie.

Ici dans le temps varie la teinte choisie pour redéfinir tous les pixels.

Le tutoriel n'est pas fini ! D'autres opérations en RVB et TSV, un TP et un chapitre sur comment visualiser l'espace TSV sont à venir.

J'espère que vous avez compris l'importance des espaces de couleurs !

Vous pouvez en étudier d'autres (XYZ, le YUV, le Lab...), comprendre leurs différences, leur histoire, pourquoi ils ont été créés et à quels besoins ils répondent.

Vous pouvez aussi continuer votre apprentissage du traitement d'images et de vidéos, découvrir et créer de nouveaux filtres, trouver des situations d'applications concrètes ou transformer des images seulement pour son but esthétique.. La voie est ouverte .

Contenu masqué

Contenu masqué n°26

Intéressez vous à la valeur minimale ainsi que la valeur maximale des trois canaux. Retourner au texte.

Contenu masqué n°27

Vous devez appliquer aux canaux R,V et B **la fonction inverse** de ce que vous avez fait pour passer d'une couleur pure à une couleur TSV. Retourner au texte.

Contenu masqué n°28

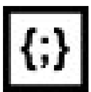
La fonction inverse de $f(x) = ax + b, a! = 0$ est $f^{-1}(x) = (x - b)/a$. Retourner au texte.

Contenu masqué n°29

Pour créer `Teinte_Depuis_RVB` vous allez devoir, comme dans `RVB_Depuis_Teinte`, distinguer 6 cas différents, et retrouver le coefficient d'un des 6 barycentres. Retourner au texte.

Contenu masqué n°30

```
1 // Operation sur la couleur en RVB du pixel
2 C_TSV.DefinirDepuisRVB( C_RVB );
3 C_RVB.DefinirDepuisTSV( C_TSV );
```



Télécharger le code

Retourner au texte.

Liste des abréviations

N&B Noir Blanc. 1, 27, 28

RVB Rouge Vert Bleu. 1–3, 8, 9, 14, 23, 28, 37, 38, 41, 43–46, 50–53, 55

TSV Teinte Saturation Valeur. 2, 3, 37, 41–43, 45, 50–52, 55, 56