

*Audebert Marion
& Grisi Manon*

Rapport de projet de Mathématiques pour l'informatique : HDR

Ce projet consistait à récupérer plusieurs photos d'une même scène chacune avec une exposition différente et de reconstruire une image en HDR.

Éléments demandés et codés qui fonctionnent :

Nous avons réussi à obtenir une image en niveau de gris en suivant les instructions du projet.

- Nous avons calculé la courbe inverse de la réponse des photosites.
- Puis nous avons calculé l'image d'irradiance
- Et enfin nous avons fait un tone mapping linéaire. Nous avions commencé à regarder les autres algorithmes, mais cela nous paraissait compliqué et nous manquions de temps. Le résultat étant satisfaisant avec l'interpolation linéaire, nous nous en sommes contentées.

Éléments non demandés (options) et codés qui fonctionnent :

Une fois que nous avions un rendu correct d'une image en noir et blanc, nous avons pu passer à la couleur. Il suffisait alors de faire une matrice Eigen pour chaque couleur et de passer trois fois dans chaque fonction, une fois pour chaque canal. Les calculs étant faits, nous avions trois matrices des valeurs de pixels pour chaque canal. Nous pouvions alors les dessiner dans une image RGB8u mais au lieu de passer un niveau de gris dans chaque canal, nous avons passé chaque couleur dans les canaux de l'image de sortie.

Pour la sélection des pixels, nous avons voulu faire une sélection aléatoire. Comme nous avons vu en cours que les fonctions basiques de c++ tel que rand ou srand n'étaient pas optimisées, nous nous sommes intéressées à l'algorithme de Mersenne Twister et avons trouvé une implémentation de cette méthode sur internet.

Éléments non demandés mais pas codés ou qui ne fonctionnent pas, mais pour lesquels vous avez des choses à dire :

Comme nous vous l'avions expliqué par mail, nous avions pensé éviter une recopie des images dans les matrices Eigen en changeant le prototype des fonctions responseRecovery et computeRadianceMap.

En effet, au lieu de passer des matrices Eigen pour chaque canal de chaque image, nous passions directement une image de type ImageRGB8u et un numéro de canal.

Etant donné qu'il n'y avait pas de calculs à faire sur ces matrices, nous pouvions seulement récupérer les valeurs des pixels dans les fonctions.

Mais comme vous nous l'avez dit par mail, si l'on veut des fonctions plus génériques qui peuvent utiliser n'importe quel type d'image, il faut garder votre prototype en passant des matrices Eigen et en faisant la recopie de l'image pour chaque canal dans le main.

Le projet, étape par étape

Pour ce projet, la récupération des images étant déjà donnée, nous ne reviendrons pas dessus.

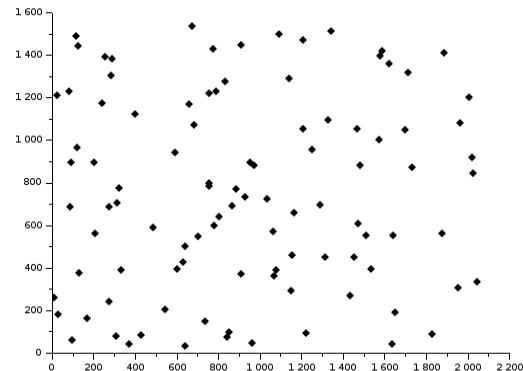
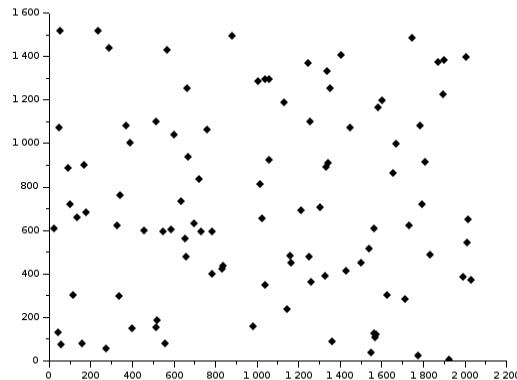
Les images étant récupérées en couleurs (sur 3 canaux), nous voulons les recopier dans des matrices Eigen qui ne contiendrait que le niveau de gris. Pour cela , nous avons suivi une recommandation de l'*Union internationale des télécommunications* :

$$\text{Luminance} = 0,2126 \times \text{Rouge} + 0,7152 \times \text{Vert} + 0,0722 \times \text{Bleu}.$$

Une fois les images récupérées, il faut pouvoir selectionner les pixels que nous prendrons en compte pour calculer la fonction inverse de la réponse des photosites. Comme expliqué dans le projet, pour être cohérent, il faut choisir un nombre de pixels N suivant le nombre d'images P tel que : $N * P > 256$ puisque les valeurs des pixels sont comprises entre 0 et 255.

Choix des pixels :

Pour choisir les pixels qui seront pris en compte sur chaque image, nous avons décidé de le faire aléatoirement. Nous avons vu en cours que les fonctions rand et srand de la librairie standard de c++ n'étaient pas efficaces, nous avons donc trouvé une implémentation de l'algorithme Mersenne Twister. En initialisant la graine avec le temps on obtient ces résultats pour deux tirages de coordonnées aléatoires dans l'image :



Il aurait pu être intéressant de faire une distribution normale pour donner plus de poids aux pixels du centre, mais cela était compliquer pour très peu de changement sur le rendu de l'image, nous ne nous sommes donc pas attardées sur ce point.

Calcul de la fonction inverse de la réponse des photosites :

Une fois les pixels choisis, nous voulons calculer la fonction inverse de la réponse des photosites. Comme expliqué dans le sujet, on peut retrouver cette fonction en minimisant le résidu suivant :

$$O = \underbrace{\sum_{i=1}^N \sum_{j=1}^P \left(g(Z_{i,j}) - \ln E_i - \ln \Delta t_j \right)^2}_{\text{terme de fidélité aux données}} + \lambda \underbrace{\sum_{Z=Z_{min}+1}^{Z_{max}-1} g''(Z)^2}_{\text{terme de lissage}}$$

avec :

$$g''(z) \simeq g(z-1) - 2g(z) + g(z+1).$$

Nous avons mis du temps à comprendre cette formule, mais en réfléchissant très fort, nous avons réussi à observer que l'on pouvait diviser cette formule en deux. En effet, nous voulons faire tendre le terme de lissage et le terme de fidélité aux données vers 0. Étant donné que le terme de lissage ne dépend pas des images, nous pouvons le traiter séparément et l'écrire sous forme de système :

$$\begin{aligned} & \lambda \sum_{Z=Z_{min}+1}^{Z_{max}-1} g''(Z)^2 = 0 \\ \Rightarrow & \lambda \sum_{Z=Z_{min}+1}^{Z_{max}-1} g''(Z) = 0 \\ \Rightarrow & \begin{cases} \lambda g''(1) = 0 \\ \lambda g''(2) = 0 \\ \vdots \\ \lambda g''(254) = 0 \end{cases} \\ \Rightarrow & \begin{cases} \lambda g(0) - 2\lambda g(1) + \lambda g(2) = 0 \\ \lambda g(1) - 2\lambda g(2) + \lambda g(3) = 0 \\ \vdots \\ \lambda g(254) - 2\lambda g(255) + \lambda g(256) = 0 \end{cases} \end{aligned}$$

On fait de même avec le terme de fidélité aux données :

$$\sum_{i=1}^N \sum_{j=1}^P (g(Z_{ij}) - \ln(E_i) - \ln(\Delta t_j))^2 = 0$$

$$\Rightarrow \sum_{i=1}^N \sum_{j=1}^P g(Z_{ij}) - \ln(E_i) - \ln(\Delta t_j) = 0$$

$$\left\{ \begin{array}{l} g(z_{00}) - \ln(E_0) - \ln(\Delta t_0) = 0 \\ g(z_{00}) - \ln(E_1) - \ln(\Delta t_0) = 0 \\ \vdots \\ g(z_{N0}) - \ln(E_N) - \ln(\Delta t_0) = 0 \\ g(z_{01}) - \ln(E_0) - \ln(\Delta t_1) = 0 \\ g(z_{N1}) - \ln(E_N) - \ln(\Delta t_1) = 0 \\ \vdots \\ g(z_{0P}) - \ln(E_0) - \ln(\Delta t_P) = 0 \\ g(z_{NP}) - \ln(E_N) - \ln(\Delta t_P) = 0 \end{array} \right.$$

$$\left\{ \begin{array}{l} g(z_{00}) - \ln(\epsilon_0) = \ln(\Delta t_0) \\ \vdots \\ g(z_{N0}) - \ln(\epsilon_N) = \ln(\Delta t_0) \\ \dots \\ g(z_{0p}) - \ln(\epsilon_0) = \ln(\Delta t_p) \\ \vdots \\ g(z_{Np}) - \ln(\epsilon_N) = \ln(\Delta t_p) \end{array} \right.$$

Ensuite, on voulait obtenir un système matriciel sous la forme $Ax = b$. Il nous a fallu à nouveau du temps pour saisir comment mettre les systèmes précédents sous forme de système matriciel.

x étant l'inconnue, c'est dans ce vecteur que l'on va mettre toutes les valeurs qu'on ne connaît pas, c'est-à-dire les $\ln(E_i)$ et les $g(Z)$. On remplit ensuite A et b pour correspondre aux systèmes précédents.

On obtient alors :

Une fois le système matriciel posé, on le résout au sens des moindres carrés à l'aide de la formule $x = (A^T A)^{-1} A^T b$. On obtient alors toutes les valeurs que peut prendre la fonction g étant donné qu'elle est définie sur [0-255] dans la première partie de x. Les $\ln(E_i)$ contenus dans le reste du vecteur ne seront plus utiles.

Ensuite, nous avons ajouté une ligne pour le facteur d'échelle : $g(128) = 1$; et ajouté la fonction poids décrite dans l'énoncé pour donner moins d'importance aux "noirs" et aux "blancs" surexposés. Ce sont donc les pixels qui sont les plus proches de la valeur médiane (128) qui comptent le plus pour le calcul de la fonction g.

Pour coder ce système, nous avons fait une boucle incrémentant les lignes. Chaque ligne correspondant à un pixel d'une image pour la première partie de la matrice. Puis une valeur de Z allant de 1 à 254 pour la deuxième partie.

Pour la première partie du système (termes de fidélité aux données) :

Pour img allant de 0 à P:

Pour pix allant de 0 à N:

On récupère la valeur du pixel pix de l'image img $\rightarrow Z$

On calcule son poids w

$A(\text{pix} + N * \text{img}, Z) = w;$

$A(\text{pix} + N * \text{img}, 256 + \text{pix}) = -w;$

$b(\text{pix} + N * \text{img}) = w * \log(\text{exposure(img)})$

Pour la deuxième partie du système (termes de lissage) :

col = 0;

Pour i allant de N*P à 255 + N*P:

$b(i) = 0;$

On calcule le poids de $(\text{col}+1) \rightarrow w$

$A(i, \text{col}) = \lambda * w;$

$A(i, \text{col}+1) = -2 * \lambda * w;$

$A(i, \text{col}+2) = \lambda * w;$

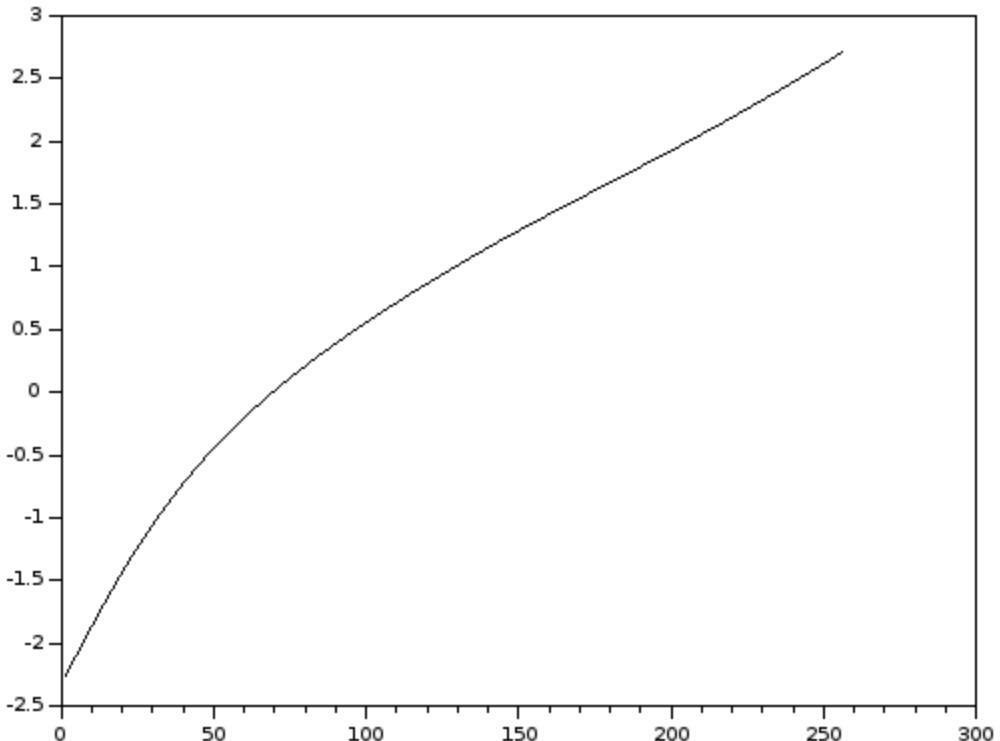
Puis on remplit la dernière ligne pour centrer les données en 1 :

$A(N*P + 255, 128) = 1;$

$b(N*P + 255) = 1;$

Nous avons donc une matrice remplie. Eigen s'occupe de faire les calculs pour la résolution au sens des moindres carrés.

Pour vérifier nos résultats au fur et à mesure et pouvoir débugger cette fonction, nous avons imprimé les résultats dans un fichier .txt pour pouvoir dessiner la courbe de g avec scilab. Pour une image RGB, on obtient :



Maintenant que nous avons obtenu l'inverse de la courbe de réponse des photosites qui nous convient, on peut facilement reconstituer l'image d'irradiance.

Calcul de l'image d'irradiance :

Pour calculer l'image d'irradiance, nous n'avons pas rencontré de problèmes particuliers, nous avons simplement utilisé la fonction donnée dans l'énoncé :

$$\ln(E_i) = \frac{\sum_{j=1}^P w(Z_{ij}) (g(Z_{ij}) - \ln(\Delta t_j))}{\sum_{j=1}^P w(Z_{ij})}$$

Il suffit donc de boucler sur chaque pixel de chaque image pour calculer les poids. Le vecteur exposure nous permet de récupérer les temps d'exposition Δt_j . Et le vecteur renvoyé par la fonction précédente nous permet de retrouver la valeur de g pour le pixel que l'on traite.

Le seul problème que nous avons rencontré, c'est lorsque le dénominateur est égal à 0. On obtient alors un "nan" (Not A Number).

Ce phénomène s'explique lorsque toutes les valeurs de Z sont à 0 ou à 255. On a alors la somme des poids égale à 0. C'est donc pour les "blanc" surexposés et les "noirs".

Nous avons décidé de gérer cela au moment du tone Mapping.

Tone Mapping :

Pour le tone mapping, nous avons seulement fait une interpolation linéaire. Celle-ci donnant un résultat plus que correct, nous ne nous sommes pas intéressées aux autres algorithmes trop compliqués à notre goût.

L'interpolation linéaire consistant à un produit en croix :

$$resultat(i,j) = \frac{(img(i,j) - minImg) * (255 - 0)}{maxImg - minImg}$$

img étant la matrice que nous avons trouvé en calculant l'irradiance.
minImg et maxImg étant les valeurs maximum et minimum de cette matrice.

Il faut donc vérifier avant que le dénominateur n'est pas égal à 0.
Il faut aussi, comme on l'a dit précédemment, gérer le cas où tout les w(Zi,j) valent 0. On a alors img(i,j) qui n'est pas un nombre ("nan"). Lorsque tel est le cas, il s'agit de déterminer s'il est question d'un blanc surexposé ou d'un noir. On regarde donc la valeur de Z pour ce pixel pour n'importe quelle image (la première paraît le plus simple), s'il est proche du max, on le met à 255, s'il est proche du min, on le met à 0.

Nous obtenons alors une matrice avec les valeurs de Ei interpolées entre 0 et 255. Nous pouvons donc dessiner l'image en niveau de gris en mettant la valeur de chaque canal à la valeur trouvée pour ce pixel.

Pour les photos suivantes :



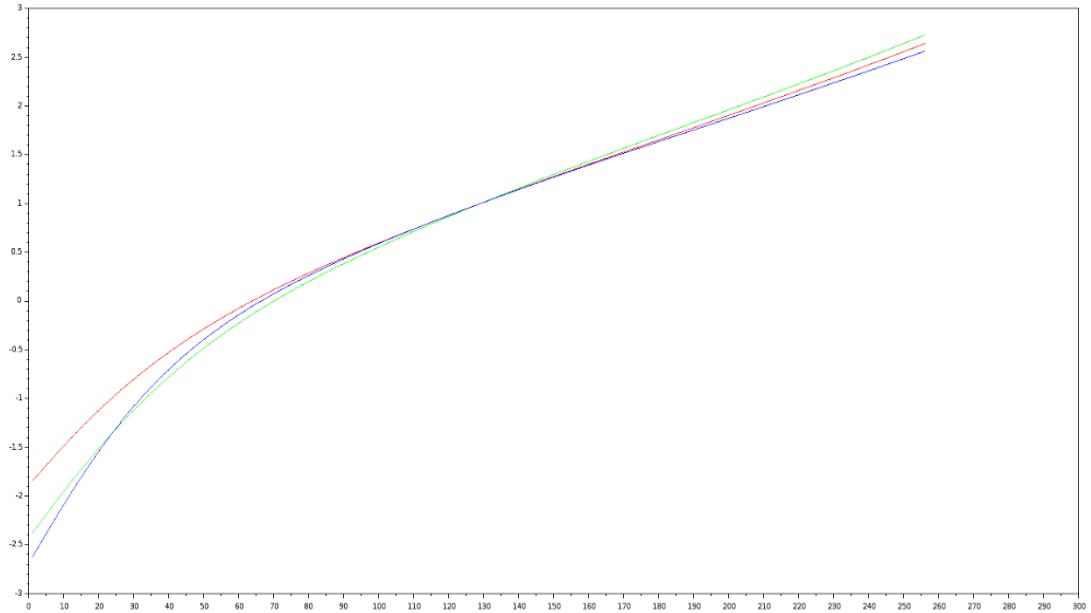


On obtient une image en niveau de gris :



Couleurs :

Pour le passage à la couleur, nous n'avons pas eu de grosses difficultés, nous avons fait la même chose que pour l'image en niveau de gris, mais une fois pour chaque canal. On obtient alors trois courbes de réponse des photosites :



Puis on calcule les trois images d'irradiance et enfin on utilise le tone mapping pour avoir trois images. On peut alors remplir les canaux de l'image de sortie avec ces trois images.

Pour les photos précédentes, on obtient une image rgb :



Conclusion

Sur ce projet, la partie mathématique fût le point le plus hardu à comprendre, il nous a fallut beaucoup de temps pour trouver le système matriciel. Mais en persévrant et en réfléchissant très fort, nous sommes parvenues à résoudre le problème posé.

Une fois ce système trouvé, nous sommes passées à la partie informatique. L'implémentation de l'algorithme était longue mais sans être particulièrement compliquée.

Le debuggage fût le point le plus éprouvant étant donné que nous gérions des matrices de grandes tailles même si nous utilisions des petites images pendant la phase de développement.

Malgré sa difficulté, ce projet reste très intéressant. Il est très plaisant d'arriver à un résultat concret sous forme d'image.