Phương pháp Demosaicing để tạo ra ảnh màu

Lê Phạm Công, 07; Đoàn Thế Lên, 16.

Trường Đại học Bách Khoa- Đại học Đà Nẵng

1. Giới thiệu bài toán

Hiện nay, hầu hết các máy ảnh kỹ thuật số dành cho người tiêu dùng đều sử dụng một cảm biến CCD đơn để giảm chi phí và kích thước của máy ảnh. Để giữ lại thông tin màu sắc, một bộ lọc màu được đặt trước CCD. Kết quả là, chỉ có một độ nhạy màu (đỏ, xanh lá cây hoặc xanh dương) có sẵn tại mỗi vị trí không gian.

Demosaicing là quá trình tái tạo hình ảnh màu sắc từ một hình ảnh được chụp bằng cảm biến chỉ có một độ nhạy màu tại mỗi điểm ảnh. Demosaicing sử dụng các thuật toán để ước tính các giá trị màu sắc bị thiếu và tạo ra một hình ảnh màu sắc hoàn chỉnh.

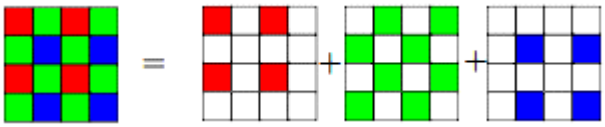
Demosaicing là quá trình quan trọng trong xử lý ảnh kỹ thuật số. Nó được sử dụng để tái tạo hình ảnh màu sắc hoàn chỉnh từ một hình ảnh đơn màu tại mỗi điểm ảnh. Điều này rất quan trọng trong việc lưu trữ, xử lý và truyền tải hình ảnh kỹ thuật số, đặc biệt là trong các ứng dụng liên quan đến máy ảnh số và camera điện thoại. Các phương pháp demosaicing có thể được sử dụng để tăng cường chất lượng hình ảnh và cải thiện trải nghiệm người dùng.

2. Phương pháp

2.1. Phương pháp nội suy tuyến tính (Bilinear Interpolation)

Việc tách màu là vấn đề giải quyết việc khôi phục ảnh đã được chụp với CFA, sao cho với mỗi điểm ảnh CCD, chúng ta có thể liên kết một giá trị RGB đầy đủ. Cách tiếp cận đơn giản nhất cho việc tách màu là sử dụng phương pháp nội suy tuyến tính đa biến (Bilinear Interpolation), trong đó ba mức màu được nội suy độc lập bằng cách sử dụng phương pháp nội suy tuyến tính đối xứng từ các hàng xóm gần nhất của cùng một màu.

Mô hình sắp xếp không gian phổ biến nhất của bộ lọc màu được gọi là Bayer CFA, được đặt theo tên của nhà phát minh của nó [1] (xem hình 1).



Hình 1. Bộ lọc màu Bayer CFA và các lưới màu đỏ, xanh lục và xanh lam.

Tạo một hình ảnh ba màu trên mỗi pixel từ một hình ảnh một màu trên mỗi pixel có thể được coi như một vấn đề nội suy. Các thuật toán Demosaicing phải tái tạo lại các màu bị thiếu. Cách đơn giản và hiệu quả nhất để demosaicing là phương pháp nội suy tuyến tính (bilinear interpolation). Các bộ lọc tích chập sau đây dùng để tính toán phép nội suy một cách hiệu quả.

$$F_{R,B} = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} /4; F_G = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 0 \end{bmatrix} /4 \quad (1)$$

$F_{R,B}$ và F_G là hai bộ lọc nội suy tuyến tính cho các lưới màu đỏ và xanh dương, và lưới màu xanh lục, tương ứng. Bộ lọc nội suy tuyến tính cho lưới màu đỏ và xanh dương giống nhau là do sự tương đồng trong sự phân bố giá trị màu sắc tại các pixel của 2 màu này. Vì G mang

nhiều thông tin về độ sáng cho mắt người, tần số lấy mẫu của nó gấp đôi tần số lấy mẫu của R và B.

Hình ảnh được nội suy cho thấy hai hiện tượng bất lợi thường gặp trong demosaicing: làm mờ hình ảnh và tạo ra màu giả, còn được gọi là hiện tượng dịch màu [2].

2.2. Phương pháp Alleysson [2]

Trong các máy ảnh CCD mà chỉ sử dụng một chip cảm biến màu, chỉ có một độ nhạy màu duy nhất tại mỗi vị trí không gian. Với loại cảm biến này, thông tin không gian và màu sắc được kết hợp với nhau.

Spatial-chromatic sampling[2] được đề xuất bởi David Alleysson, là một phương pháp tiếp cận để tách luminance và chrominance từ hình ảnh màu bằng cách sử dụng tính chất của các cảm biến màu.

Luminance và Chrominance là hai thành phần của màu sắc được sử dụng để mô tả màu sắc trong hệ thống màu sắc. Luminance là thành phần độ sáng của màu sắc, trong khi chrominance là thành phần độ bão hòa màu sắc. Luminance và chrominance được sử dụng để xác định màu sắc của một điểm ảnh trong hệ thống màu sắc, như RGB (Red, Green, Blue) hoặc YUV (Luminance, Chrominance Blue, Chrominance Red). Việc tách luminance và chrominance rất quan trọng trong xử lý ảnh và video để thực hiện các bước tiền xử lý như đồng bộ hóa màu sắc, cân bằng màu sắc, nâng cao chất lượng ảnh và video.

Trong hình ảnh màu, mỗi điểm ảnh có chứa thông tin về ba thành phần màu sắc (R, G, B) tại mỗi vị trí không gian. Tuy nhiên, các thông tin này bị trộn lẫn vào nhau và khó phân tách, làm cho việc xử lý hình ảnh màu trở nên phức tạp.

Phương pháp này giải quyết vấn đề này bằng cách sử dụng hai bộ lọc khác nhau để tách riêng luminance và chrominance từ ảnh gốc. Các bước thực hiện của phương pháp là:

2.2.1. Tách luminance:

Đầu vào để xử lý tách thành phần luminance là ảnh mosaic, với mô hình sắp xếp không gian của bộ lọc màu được gọi là Bayer CFA (xem hình 1).

Để tách luminance theo phương pháp Spatial-chromatic sampling với ảnh theo mô hình Bayer CFA, ta cần làm theo các bước sau:

- Đọc ảnh màu theo mô hình Bayer CFA và tách nó thành các channel màu tương ứng. Trong mô hình Bayer CFA, mỗi pixel chỉ có thông tin về một trong ba màu đỏ, xanh hoặc lục.
- Áp dụng bộ lọc trung bình trên ảnh màu để tính toán độ sáng trung bình của từng pixel [2]. Bộ lọc trung bình được mô tả như phương trình (2) Kết quả sẽ tạo ra một ảnh độ sáng trung bình (luminance) với kích thước giống với ảnh gốc.

$$F_L = \begin{bmatrix} -2 & 3 & -6 & 3 & -2 \\ 3 & 4 & 2 & 4 & 3 \\ -6 & 2 & 48 & 2 & -6 \\ 3 & 4 & 2 & 4 & 3 \\ -2 & 3 & -6 & 3 & -2 \end{bmatrix} /64 \quad (2)$$

2.2.2. Tách chrominance:

Để tách chrominance theo phương pháp Spatial-chromatic sampling với ảnh theo mô hình Bayer CFA, ta cần làm theo các bước sau:

- Đọc ảnh màu theo mô hình Bayer CFA và tách nó thành các channel màu tương ứng. Trong mô hình Bayer CFA, mỗi pixel chỉ có thông tin về một trong ba màu đỏ, xanh hoặc lục.
- Thành phần chrominance được tính bằng cách lấy ảnh màu theo mô hình Bayer CFA trừ đi thành thần Luminance (L) đã tính được ở bước 2.2.1:

$$Chrominance = \{R - L;\ G - L;\ B - L\} \quad (3)$$

2.2.3. Nội suy tái tạo lớp chrominance:

Sau khi tính toán được các giá trị chrominance, ta cần khôi phục lại các giá trị chrominance còn thiếu trong ảnh theo mô hình Bayer CFA. Quá trình này được thực hiện bằng cách sử dụng một thuật toán tối ưu hóa để dự đoán các giá trị màu bị thiếu dựa trên các giá trị màu có sẵn của các pixel láng giềng.

Thuật toán này tương tự phương pháp nội suy tuyến tính, sử dụng các bộ lọc tích chập ở phương trình (1) để khôi phục các giá trị chrominance theo mô hình CFA.

2.2.4. Kết hợp luminance và chrominance sau khi tái tạo để tạo ra ảnh màu hoàn chỉnh

Các giá trị Chrominance sau khi khôi phục sẽ bố trí theo mô hình Bayer CFA, để tái tạo lại ảnh màu hoàn hình, cần kết hợp thành phần luminance và thành phần chrominance sau khi khôi phục.

2.3. Phương pháp nội suy tuyến tính chất lượng cao (High- quality Linear Interpolation for Demosaicing of Bayer-Patterned color images):

Phương pháp nội suy tuyến tính chất lượng cao là một phương pháp nâng cao chất lượng của phương pháp nội suy tuyến tính (bilinear interpolation), được sử dụng để tái tạo (reconstruct) dữ liệu số bị mất mát (missing data) hoặc bị nhiễu (noisy data) trong các ứng dụng xử lý tín hiệu số.

Trong nội suy tuyến tính chất lượng cao, một mô hình tuyến tính được sử dụng để xác định các giá trị tại các điểm dữ liệu bị mất mát hoặc bị nhiễu. Tuy nhiên, để nâng cao chất lượng của phương pháp tuyến tính truyền thống, nội suy tuyến tính chất lượng cao áp dụng các kỹ thuật điều chỉnh (adjustment techniques) để tối ưu hóa các tham số của mô hình tuyến tính và giảm thiểu sai số dự đoán.

Phương pháp này được mô tả như sau:

- Các cạnh có độ chói mạnh hơn nhiều so với các thành phần màu sắc. Do đó, khi xem xét phép nội suy của giá trị màu lục ở vị trí pixel màu đỏ chẳng hạn, không loại bỏ giá trị màu đỏ ở vị trí đó – đó là thông tin có giá trị. Thay vào đó, so sánh giá trị màu đỏ đó với ước tính của nó cho phép nội suy song tuyến tính cho các mẫu màu đỏ gần nhất. Nếu nó khác với ước tính đó, điều đó có thể có nghĩa là có sự thay đổi độ chói mạnh ở pixel. Giá trị xanh được nội suy song tuyến tính bằng cách thêm vào một phần của độ sáng ước tính này.[5]
- Sử dụng công thức để nội suy các giá trị G tại một vị trí R [5]

$$\hat{g}(i,j) = \widehat{g_B}(i,j) + \alpha\Delta_R(i,j) \quad (4)$$

- Trong đó $\Delta_R(i,j)$ là gradient của R tại vị trí đó[5]

$$\Delta_R(i,j) \triangleq r(i,j) - \frac{1}{4}\sum r(i+m, j+n) \quad (5)$$

(m,n)={(0,-2), (0,2), (-2,0), (0,-2)}

- Tương tự áp dụng tại vị trí R và B[5]

+ Đối với vị trí R:

$$\hat{g}(i,j) = \widehat{g_B}(i,j) + \beta\Delta_G(i,j) \quad (6)$$

+ Đối với vị trí B:

$$\hat{g}(i,j) = \widehat{g_B}(i,j) + \gamma\Delta_B(i,j) \quad (7)$$

Để xác định các giá trị phù hợp cho các tham số khuếch đại { }, chúng tôi đã sử dụng phương pháp Wiener; nghĩa là, chúng tôi đã tính toán các giá trị dẫn đến phép nội suy sai số bình phương trung bình tối thiểu, dựa trên số liệu thống kê bậc hai được tính toán từ một bộ dữ liệu tốt (bộ ảnh Kodak được sử dụng). Sau đó, chúng tôi tính gần đúng hệ số Wiener tối ưu bằng bội số nguyên của lũy thừa nhỏ $\frac{1}{2}$, với kết quả cuối cùng $\alpha = \frac{1}{2}, \beta = \frac{5}{8}$ và $\gamma = \frac{3}{4}$. Từ các giá trị {α, β, γ}, chúng ta có thể tính toán các hệ số bộ lọc FIR tuyến tính tương ứng cho mỗi trường hợp nội suy. [5]

3. Thực nghiệm

3.1. Dữ liệu:

Các phương pháp được đánh giá bằng bộ dữ liệu Kodak [4].

3.2. Tiêu chí đánh giá:

3.2.1. Chỉ số PSNR (Peak Signal-to-Noise Ratio):

Tỷ lệ tín hiệu trên nhiễu cao nhất (PSNR) là một thuật ngữ kỹ thuật cho tỷ lệ giữa công suất tối đa có thể có của tín hiệu và công suất của nhiễu làm hỏng ảnh hưởng đến độ trung thực của biểu diễn. Do nhiều tín hiệu có dải động rất rộng nên PSNR thường được biểu thị dưới dạng đại lượng logarit sử dụng thang decibel .

PSNR thường được sử dụng để định lượng chất lượng tái tạo đối với hình ảnh và video chịu nén mất dữ liệu .

PSNR được tính bằng công thức sau[3]:

$$PSNR_{(f,g)} = 10\log_{10}\frac{255^2}{MSE_{(f.g)}} \quad (8)$$

$$MSE_{(f,g)} = \frac{1}{m.n.3}\sum_{i=1}^{m}\sum_{j=1}^{n}\left[f_{ij} - g_{ij}\right]^2 \quad (9)$$

Trong đó:

f_{ij} là ma trận ảnh gốc

g_{ij} là ma trận hình ảnh xuống cấp

m,n lần lượt là số hàng,cột pixel

i,j lần lượt là số hàng và cột của hình ảnh

Giá trị PSNR tiến tới vô cùng khi MSE tiến tới không; điều này cho thấy giá trị PSNR cao cung cấp chất lượng hình ảnh cao hơn. Ngược lại, giá trị nhỏ của PSNR cho thấy sự khác biệt cao giữa các hình ảnh[3].

3.2.2. Chỉ số SSIM (Structural Similarity Index):

Chỉ số SSIM được sử dụng để đo mức độ giống nhau giữa hình ảnh đầu vào và ảnh sinh ra. Công thức SSIM dựa trên ba thông số để so sánh: độ chói (luminance), tương phản (contrast) và cấu trúc (structure). Một ảnh sinh ra là tốt nếu:

+ Những điểm ảnh có mức độ sáng tối khác nhau, và càng có nhiều mức độ sáng tối càng có nhiều chi tiết ảnh => Ảnh chất lượng tốt

+ Một bức ảnh không phải độ tương phản càng cao thì càng tốt mà nên có sự hài hòa cân đối giữa sáng và tối. => Độ da dạng

Chỉ số SSIM được tính bởi công thức sau[4]:

$$SSIM(x,y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2+\mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)} \quad (10)$$

Trong đó:

μ_x, μ_y là là trung bình của cửa sổ X, Y.

σ_x, σ_y là phương sai của cửa sổ X, Y

σ_{xy} là hiệp phương sai giữa X và Y

c_1, c_2 là hằng số nhỏ để tránh mẫu bằng không.

3.3. Kết quả:

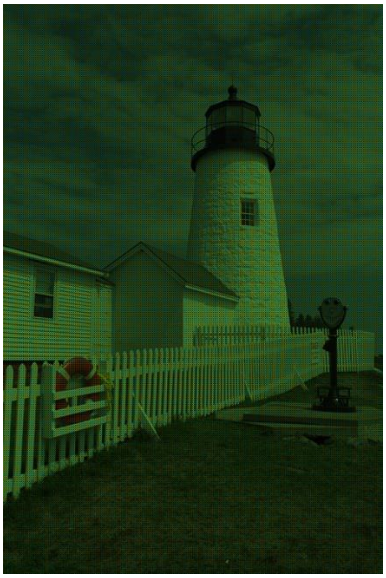
3.3.1. Bảng kết quả thử nghiệm:

	Nội suy tuyến tính		Phương pháp Alleysson		Nội suy tuyến tính chất lượng cao	
	PSNR	SSIM	PSNR	SSIM	PSNR	SSIM
ảnh 1	32.4425	0.8064	35.2267	0.9623	32.2016	0.7931
ảnh 2	36.5468	0.8956	38.5063	0.9579	36.2864	0.8893
ảnh 3	37.7431	0.9281	41.1017	0.9790	37.3641	0.9214
ảnh 4	36.4127	0.9102	39.5748	0.9736	36.1662	0.9054
ảnh 5	33.284	0.8751	36.7736	0.9778	32.9406	0.8619
ảnh 6	33.7678	0.8446	36.3811	0.9627	33.4530	0.8332
ảnh 7	37.3573	0.9496	40.5371	0.9835	36.7929	0.9452
ảnh 8	32.3983	0.8256	34.2027	0.9576	31.9652	0.8122
ảnh 9	36.9149	0.9186	40.1374	0.9776	36.5499	0.9156
ảnh 10	36.9662	0.9181	40.7535	0.9788	36.6442	0.9144
ảnh 11	34.6600	0.8679	37.5040	0.9676	34.3853	0.8595
ảnh 12	36.8338	0.9063	39.7842	0.9733	36.5076	0.9008
ảnh 13	31.7196	0.7748	34.7135	0.9668	31.6227	0.7637
ảnh 14	34.0244	0.8680	37.4313	0.9701	33.7364	0.8582
ảnh 15	36.6551	0.9090	38.8904	0.9688	36.3736	0.9053
ảnh 16	35.3713	0.8744	38.4386	0.9665	35.0616	0.8662
ảnh 17	36.8359	0.9251	41.2782	0.9849	36.5277	0.9214
ảnh 18	33.9935	0.8663	37.4922	0.9733	33.8048	0.8581
ảnh 19	34.6399	0.8724	37.7681	0.9719	34.3127	0.8659
ảnh 20	36.6903	0.9121	39.6569	0.9690	36.4849	0.9085
ảnh 21	34.6509	0.8866	37.6263	0.9719	34.4569	0.8802
ảnh 22	35.0521	0.8826	38.0167	0.9655	34.8295	0.8753
ảnh 23	39.2555	0.9530	41.5730	0.9785	38.7034	0.9422
ảnh 24	34.2046	0.8741	37.3042	0.9751	33.9464	0.8651

Bảng 1 . Bảng kết quả thực nghiệm các phương pháp bằng bộ dữ liệu Kodak

3.3.2. Hình ảnh minh họa:

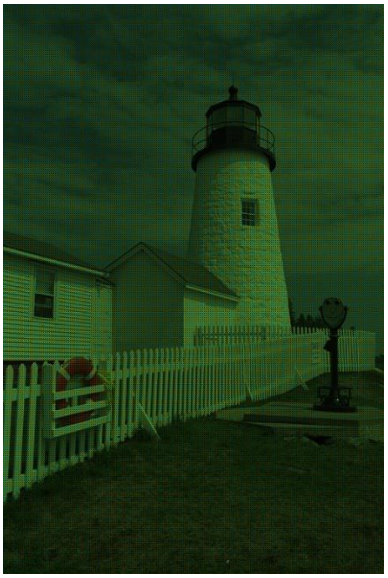

Hình a1


Hình b1


Hình c1

Hình 2 . Ví dụ minh họa về phương pháp nội suy tuyến tính: Hình a1) Ảnh mẫu; Hình b1) ảnh mosaic dựa trên mô hình Bayer CFA; Hình c1) ảnh đã khôi phục bằng phương pháp nội suy tuyến tính.

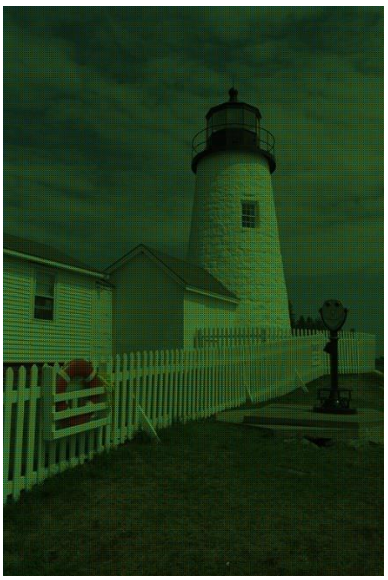

Hình a2


Hình b2


Hình c2

Hình 3 . Ví dụ minh họa về phương pháp nội suy tuyến tính: Hình a2) Ảnh mẫu; Hình b2) ảnh mosaic dựa trên mô hình Bayer CFA; Hình c2) ảnh đã khôi phục bằng phương pháp Alleysson.

Hình a3



Hình b3

Hình c3

Hình 4 . Ví dụ minh họa về phương pháp nội suy tuyến tính: Hình a3) Ảnh mẫu; Hình b3) ảnh mosaic dựa trên mô hình Bayer CFA; Hình c3) ảnh đã khôi phục bằng phương pháp nội suy tuyến tính chất lượng cao.

3.3.3. So sánh đánh giá:

3.3.3.1. So sánh:

	Nội suy tuyến tính	Phương pháp Alleysson	Nội suy tuyến tính chất lượng cao
Ưu điểm	-Đơn giản, dễ hiểu và dễ triển khai. -Phù hợp cho các bộ dữ liệu có dạng đơn giản, tuyến tính và các giá trị cách nhau đều.	-Phù hợp với các bộ dữ liệu phức tạp, không phải dạng tuyến tính. -Cho phép ước lượng đường cong thông qua các điểm dữ liệu gần nhất, giúp cải thiện độ chính xác của ước lượng. -Độ chính xác cao hơn so với nội suy tuyến tính chất lượng cao, nội suy tuyến tính với cùng số lượng điểm dữ liệu.	-Thuật toán đơn giản, dễ hiểu và thực hiện. -Phù hợp với các bài toán có số lượng điểm dữ liệu lớn và đơn giản. -Không cần tối ưu hóa đặc biệt, giảm thời gian tính toán.
Nhược điểm	-Không phù hợp với các bộ dữ liệu có dạng phức tạp và không phải dạng tuyến tính. -Không hiệu quả nếu các điểm dữ liệu cách nhau quá xa.	-Có thể bị overfitting nếu số điểm dữ liệu không đủ lớn hoặc không đều. -Phức tạp hơn và cần thực hiện các phép tính toán phức tạp hơn so với phương pháp nội suy tuyến tính.	-Yêu cầu nhiều điểm dữ liệu hơn so với Alleyson để đạt được độ chính xác tương đương. -Không phù hợp với các bài toán cần độ chính xác cao và số lượng điểm dữ liệu ít.

Bảng 2. Bảng so sánh các phương pháp demosaicing

3.3.3.2. Đánh giá:

Phương pháp Allison là một phương pháp đơn giản và nhanh chóng để ước lượng giá trị bị khuyết trong các bộ dữ liệu lớn. Nó thường được sử dụng trong các trường hợp mà mức độ khuyết dữ liệu không quá cao và không có quan hệ phức tạp giữa các biến. Tuy nhiên, phương pháp này có thể dẫn đến ước lượng sai lệch nếu mức độ khuyết dữ liệu quá cao hoặc nếu các biến trong bộ dữ liệu có mối tương quan cao với nhau.

Phương pháp nội suy tuyến tính là một phương pháp phổ biến để ước lượng giá trị bị khuyết trong các bộ dữ liệu. Phương pháp này dựa trên giả định rằng các giá trị bị khuyết có thể được dự đoán bằng một phép nội suy tuyến tính từ các giá trị còn lại trong bộ dữ liệu. Phương pháp này cho kết quả khá chính xác khi mức độ khuyết dữ liệu không quá cao và các biến trong bộ dữ liệu không có mối tương quan quá cao với nhau.

Phương pháp nội suy tuyến tính chất lượng cao là một phương pháp tốt để ước lượng giá trị bị khuyết trong các bộ dữ liệu lớn và có mức độ khuyết dữ liệu cao. Phương pháp này sử dụng một mô hình tuyến tính để dự đoán các giá trị bị khuyết dựa trên các giá

trị còn lại trong bộ dữ liệu. Phương pháp này cho kết quả rất chính xác khi mức độ khuyết dữ liệu cao và các biến trong bộ dữ liệu có mối tương quan cao với nhau.

4. Kết luận

Bài viết đã giới thiệu về 3 phương pháp Demosaicing để tạo ảnh màu cũng như cách thực hiện các phương pháp và sự đánh giá tổng quan về hiệu năng và hiệu quả của các phương pháp mang lại. Nhìn chung, nếu bộ dữ liệu có dạng đơn giản và các giá trị cách nhau đều thì phương pháp nội suy tuyến tính là lựa chọn phù hợp. Nếu các bài toán có số lượng điểm dữ liệu lớn và đơn giản thì phương pháp nội suy tuyến tính chất lượng cao là lựa chọn. Trong khi đó, nếu bộ dữ liệu phức tạp và các điểm dữ liệu không đều nhau, phương pháp Alleysson có thể cải thiện độ chính xác của ước lượng

Tài liệu tham khảo

[1] B.E. Bayer, 1976, Color imaging array, US Patent 3,971,065

[2] David Alleysson, Sabine Süsstrunk, Jeanny Hérault, Color demosaicing by estimating luminance and opponent chromatic signals in the Fourier domain, Proc. IS&T/SID 10th Color Imaging Conference, 331-336, 2002.

[3] A. Horé and D. Ziou, "Image Quality Metrics: PSNR vs. SSIM," 2010 20th International Conference on Pattern Recognition, Istanbul, Turkey, 2010, pp. 2366-2369, doi: 10.1109/ICPR.2010.579.

[4] Bui Duc Tho, Ho Phuoc Tien, Nguyen Tan Khoi, Demosaicing Algorithm- How to evaluate it?, Kỷ yếu Hội nghị KHCN Quốc gia lần thứ XI về Nghiên cứu cơ bản và ứng dụng Công nghệ thông tin (FAIR); Hà Nội, ngày 09-10/8/2018 DOI: 10.15625/vap.2018.00014

[5] H. S. Malvar, Li-wei He and R. Cutler, "High-quality linear interpolation for demosaicing of Bayer-patterned color images," 2004 IEEE International Conference on Acoustics, Speech, and Signal Processing, Montreal, QC, Canada, 2004, pp. iii-485, doi: 10.1109/ICASSP.2004.1326587.

[6] Bộ dữ liệu Kodak: https://github.com/MohamedBakrAli/Kodak-Lossless-True-Color-Image-Suite.git

Phụ lục

a. Phương pháp nội suy tuyến tính:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
import os
# nội suy tuyến tính
def bilinear_interpolation(img):
   (r,c,_) = img.shape
   mr = np.zeros((r, c))
```

```
    mg = np.zeros((r, c))
    mb = np.zeros((r, c))
    mr[::2, ::2] = 1
    mb[1::2, 1::2] = 1
    mg = 1 - mb - mr
    I_red = img[:, :, 2].astype(np.float64)
    I_green = img[:, :, 1].astype(np.float64)
    I_blue = img[:, :, 0].astype(np.float64)
    red = mr * I_red
    green = mg * I_green
    blue = mb * I_blue
    demos_picture = np.zeros((r, c, 3), dtype=np.uint8)
    demos_picture[:, :, 0] = blue
    demos_picture[:, :, 1] = green
    demos_picture[:, :, 2] = red
    w_R = np.array([[1, 2, 1], [2, 4, 2], [1, 2, 1]])/4
    w_G = np.array([[0, 1, 0], [1, 4, 1], [0, 1, 0]])/4
    w_B = w_R
    blue = cv2.filter2D(blue, -1, w_B)
    green = cv2.filter2D(green, -1, w_G)
    red = cv2.filter2D(red, -1, w_R)
    picture_restored = np.zeros((r, c, 3), dtype=np.uint8)
    picture_restored[:, :, 0] = blue
    picture_restored[:, :, 1] = green
    picture_restored[:, :, 2] = red
    return picture_restored
# end of bilinear_interpolation
# Đánh giá chất lượng ảnh
# PSNR
def cv_psnr(img1, img2):
    # xác thực đầu vào
    assert img1.dtype == np.uint8 and img2.dtype == np.uint8, 'images must be 8-bit'
```

```
    assert img1.shape == img2.shape, 'images must have the same size'
    # tính MSE
    mse = np.sum((img1-img2)**2)/(img1.shape[0]*img1.shape[1]*3)
    # tính PSNR
    psnr = 10*np.log10(255**2/mse)
    return psnr
# end of cv_psnr
# SSIM
def cv_ssim(img1, img2):
    # xác thực đầu vào
    assert img1.dtype == np.uint8 and img2.dtype == np.uint8, 'images must be 8-bit'
    assert img1.shape == img2.shape, 'images must have the same size'
    k1 = 0.01
    k2 = 0.03
    L = 2**8 - 1  # 8-bit dynamic range of pixel-values
    C1 = (k1 * L)**2
    C2 = (k2 * L)**2
    opts = {'ksize': (11, 11), 'sigmaX': 1.5, 'sigmaY': 1.5}
    I1 = np.float64(img1)
    I2 = np.float64(img2)
    # mean
    mu1 = cv2.GaussianBlur(I1, **opts)
    mu2 = cv2.GaussianBlur(I2, **opts)
    # variance
    sigma1_2 = cv2.GaussianBlur(I1**2, **opts) - mu1**2
    sigma2_2 = cv2.GaussianBlur(I2**2, **opts) - mu2**2
    # covariance
    sigma12 = cv2.GaussianBlur(I1*I2, **opts) - mu1*mu2
    # SSIM index
    map_ = ((2 * mu1*mu2 + C1) * (2 * sigma12 + C2)) / \
          ((mu1**2 + mu2**2 + C1) * (sigma1_2 + sigma2_2 + C2))
    val = np.mean(map_)
```

```
            return val
        # end of cv_ssim
        img_dir = "Image_processing/image/Kodak/"
        img_list = os.listdir(img_dir)
        for img_name in img_list:
            img = cv2.imread(img_dir+img_name)
            img_bilinear = bilinear_interpolation(img)
            psnr = cv_psnr(img, img_bilinear)
            ssim = cv_ssim(img, img_bilinear)
            print(img_name)
        print("PSNR: ", psnr,"    ","SSIM: ", ssim)
```

b. Phương pháp Alleysson:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
import os
def alleysson(img):
    (row, col, rgb) = img.shape
    mr = np.zeros((row, col))
    mg = np.zeros((row, col))
    mb = np.zeros((row, col))
    mr[::2, ::2] = 1
    mb[1::2, 1::2] = 1
    mg = 1 - mr - mb
    red = img[:, :, 0]
    green = img[:, :, 1]
    blue = img[:, :, 2]
    red = np.multiply(red.astype(float), mr)
    green = np.multiply(green.astype(float), mg)
    blue = np.multiply(blue.astype(float), mb)
    multi_img = np.zeros((row, col, 3), dtype=np.uint8)
    multi_img[:, :, 0] = blue.astype(np.uint8)
```

```
multi_img[:, :, 1] = green.astype(np.uint8)
multi_img[:, :, 2] = red.astype(np.uint8)
F_L = np.array([[-2, 3, -6, 3, -2],
                [3, 4,  2, 4,  3],
                [-6, 2, 48, 2, -6],
                [3, 4,  2, 4,  3],
                [-2, 3, -6, 3, -2]]) * (1/64)
out = red + green + blue
lum = cv2.filter2D(out, -1, F_L)
multi_chr = out - lum
redd = np.zeros((row, col))
greenn = np.zeros((row, col))
bluee = np.zeros((row, col))
redd[::2, ::2] = multi_chr[::2, ::2]
bluee[1::2, 1::2] = multi_chr[1::2, 1::2]
greenn = multi_chr - redd - bluee
smp_chr = np.zeros((row, col, 3), dtype=np.float64)
smp_chr[:, :, 0] = bluee
smp_chr[:, :, 1] = greenn
smp_chr[:, :, 2] = redd
wrb = np.array([[1, 2, 1],
                [2, 4, 2],
                [1, 2, 1]])/4

wg = np.array([[0, 1, 0],
               [1, 4, 1],
               [0, 1, 0]])/4

redd = cv2.filter2D(redd, -1, wrb)
greenn = cv2.filter2D(greenn, -1, wg)
bluee = cv2.filter2D(bluee, -1, wrb)
chr = np.zeros((row, col, 3), dtype=np.float64)
```

```
    chr[:, :, 0] = bluee
    chr[:, :, 1] = greenn
    chr[:, :, 2] = redd
    picture_res = np.zeros((row, col, 3), dtype=np.uint8)
    picture_res[:, :, 0] = np.clip(chr[:, :, 0] + lum, 0, 255)
    picture_res[:, :, 1] = np.clip(chr[:, :, 1] + lum, 0, 255)
    picture_res[:, :, 2] = np.clip(chr[:, :, 2] + lum, 0, 255)
    return picture_res
def cv_psnr(img1, img2):
    # xác thực đầu vào
    assert img1.dtype == np.uint8 and img2.dtype == np.uint8, 'images must be 8-bit'
    assert img1.shape == img2.shape, 'images must have the same size'
    # tính MSE
    mse = np.sum((img1-img2)**2)/(img1.shape[0]*img1.shape[1]*3)
    # tính PSNR
    psnr = 10*np.log10(255**2/mse)
    return psnr
def cv_ssim(img1, img2):
    # xác thực đầu vào
    assert img1.dtype == np.uint8 and img2.dtype == np.uint8, 'images must be 8-bit'
    assert img1.shape == img2.shape, 'images must have the same size'
    k1 = 0.01
    k2 = 0.03
    L = 2**8 - 1  # 8-bit dynamic range of pixel-values
    C1 = (k1 * L)**2
    C2 = (k2 * L)**2
    opts = {'ksize': (11, 11), 'sigmaX': 1.5, 'sigmaY': 1.5}
    I1 = np.float64(img1)
    I2 = np.float64(img2)
    # mean
    mu1 = cv2.GaussianBlur(I1, **opts)
    mu2 = cv2.GaussianBlur(I2, **opts)
```

```
    # variance
    sigma1_2 = cv2.GaussianBlur(I1**2, **opts) - mu1**2
    sigma2_2 = cv2.GaussianBlur(I2**2, **opts) - mu2**2
    # covariance
    sigma12 = cv2.GaussianBlur(I1*I2, **opts) - mu1*mu2
    # SSIM index
    map_ = ((2 * mu1*mu2 + C1) * (2 * sigma12 + C2)) / \
        ((mu1**2 + mu2**2 + C1) * (sigma1_2 + sigma2_2 + C2))
    val = np.mean(map_)
    return val
img_dir = "Image_processing/image/Kodak/"
img_list = os.listdir(img_dir)
for img_name in img_list:
    img = cv2.imread(img_dir+img_name)
    img_bilinear = alleysson(img)
    psnr = cv_psnr(img, img_bilinear)
    ssim = cv_ssim(img, img_bilinear)
    print(img_name)
    print("PSNR: ", psnr, "    ", "SSIM: ", ssim)
```

c. Phương pháp nội suy tuyến tính chất lượng cao:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
import os
# def hight_quality_Bilinear_interpolation(img):
img = cv2.imread("Image_processing/image/Kodak/19.png")
(row, col, rgb) = img.shape
mr = np.zeros((row, col))
mg = np.zeros((row, col))
mb = np.zeros((row, col))
mr[::2, ::2] = 1
mb[1::2, 1::2] = 1
```

```
mg = 1 - mr - mb
red = img[:, :, 2]
green = img[:, :, 1]
blue = img[:, :, 0]
red = np.multiply(red.astype(float), mr)
green = np.multiply(green.astype(float), mg)
blue = np.multiply(blue.astype(float), mb)
multi_img = np.zeros((row, col, 3), dtype=np.uint8)
multi_img[:, :, 0] = blue.astype(np.uint8)
multi_img[:, :, 1] = green.astype(np.uint8)
multi_img[:, :, 2] = red.astype(np.uint8)
w_G = 1/4 * np.array([[0, 1, 0], [1, 0, 1], [0, 1, 0]])
w_R = 1/4 * np.array([[1, 0, 1], [0, 0, 0], [1, 0, 1]])
w_B = 1/4 * np.array([[1, 0, 1], [0, 0, 0], [1, 0, 1]])
r = np.zeros((row, col))
g = np.zeros((row, col))
b = np.zeros((row, col))
r[0::2, 0::2] += red[0::2, 0::2]
g[0::2, 1::2] += green[0::2, 1::2]
g[1::2, 0::2] += green[1::2, 0::2]
b[1::2, 1::2] += blue[1::2, 1::2]
g = g+cv2.filter2D(g, -1, w_G)
r1 = cv2.filter2D(r, -1, w_R)
r2 = cv2.filter2D(r+r1, -1, w_G)
r = r+r1+r2
b1 = cv2.filter2D(b, -1, w_B)
b2 = cv2.filter2D(b+b1, -1, w_G)
b = b+b1+b2
rows = g.shape[0]
cols = g.shape[1]
GatRB = np.array([
    [0,   0,  -1,   0,   0],
```

```
    [0,   0,   2,   0,   0],
    [-1,   2,   4,   2,  -1],
    [0,   0,   2,   0,   0],
    [0,   0,  -1,   0,   0]
])
out_g = np.zeros(g.shape)
out_g[:] = g
for i in range(rows - 5):
    for j in range(cols - 5):
        if g[i+2, j+2] == 0:
            gx = g[i:i+5, j:j+5]
            if r[i+2, j+2] != 0:
                # G at R location
                rx = r[i:i+5, j:j+5]
                out_g[i+2, j+2] = np.average(gx * GatRB + rx * GatRB)
            elif b[i+2, j+2] != 0:
                # G at B location
                bx = b[i:i+5, j:j+5]
                out_g[i+2, j+2] = np.average(gx * GatRB + bx * GatRB)
# red interpolation
RatGrow = np.array([
    [0,   0, 1/2,   0,   0],
    [0,  -1,   0,  -1,   0],
    [-1,   4,   5,   4,  -1],
    [0,  -1,   0,  -1,   0],
    [0,   0, 1/2,   0,   0]
])
RatGcol = np.array([
    [0,   0,  -1,   0,   0],
    [0,  -1,   4,  -1,   0],
    [1/2,   0,   5,   0, 1/2],
    [0,  -1,   4,  -1,   0],
```

```
    [0,   0,  -1,   0,   0]
])
RatB = np.array([
    [0,   0, -3/2,   0,   0],
    [0,   2,   0,   2,   0],
    [-3/2,   0,   6,   0, -3/2],
    [0,   2,   0,   2,   0],
    [0,   0, -3/2,   0,   0]
])
out_r = np.zeros(g.shape)
out_r[:] = r
for i in range(rows - 5):
    for j in range(cols - 5):
        if r[i+2, j+2] == 0:
            rx = r[i:i+5, j:j+5]
            if g[i+2, j+2] != 0 and (r[i+2, j+1] != 0 and r[i+2, j+3] != 0):
                # R at G, R row
                gx = g[i:i+5, j:j+5]
                out_r[i+2, j+2] = np.average(gx * RatGrow + rx * RatGrow)
            elif g[i+2, j+2] != 0 and (r[i+1, j+2] != 0 and r[i+3, j+2] != 0):
                # R at G, R col
                gx = g[i:i+5, j:j+5]
                out_r[i+2, j+2] = np.average(gx * RatGcol + rx * RatGcol)
            elif b[i+2, j+2] != 0:
                # R at B
                bx = b[i:i+5, j:j+5]
                out_r[i+2, j+2] = np.average(bx * RatB + rx * RatB)
BatGrow = RatGrow
BatGcol = RatGcol
BatR = RatB
out_b = np.zeros(g.shape)
out_b[:] = b
```

```
for i in range(rows - 5):
    for j in range(cols - 5):
        if b[i+2, j+2] == 0:
            bx = b[i:i+5, j:j+5]
            if g[i+2, j+2] != 0 and (b[i+2, j+1] != 0 and b[i+2, j+3] != 0):
                # R at G, R row
                gx = g[i:i+5, j:j+5]
                out_b[i+2, j+2] = np.average(gx * BatGrow + bx * BatGrow)
            elif g[i+2, j+2] != 0 and (b[i+1, j+2] != 0 and b[i+3, j+2] != 0):
                # R at G, R col
                gx = g[i:i+5, j:j+5]
                out_b[i+2, j+2] = np.average(gx * BatGcol + bx * BatGcol)
            elif r[i+2, j+2] != 0:
                # B at R
                rx = r[i:i+5, j:j+5]
                out_b[i+2, j+2] = np.average(bx * BatR + rx * BatR)
out = np.zeros((rows, cols, 3), dtype=np.uint8)
out[:, :, 0] = out_b
out[:, :, 1] = out_g
out[:, :, 2] = out_r
# return out
def cv_psnr(img1, img2):
    # xác thực đầu vào
    assert img1.dtype == np.uint8 and img2.dtype == np.uint8, 'images must be 8-bit'
    assert img1.shape == img2.shape, 'images must have the same size'
    # tính MSE
    mse = np.sum((img1-img2)**2)/(img1.shape[0]*img1.shape[1]*3)
    # tính PSNR
    psnr = 10*np.log10(255**2/mse)
    return psnr
def cv_ssim(img1, img2):
    # xác thực đầu vào
```

```
    assert img1.dtype == np.uint8 and img2.dtype == np.uint8, 'images must be 8-bit'
    assert img1.shape == img2.shape, 'images must have the same size'
    k1 = 0.01
    k2 = 0.03
    L = 2**8 - 1  # 8-bit dynamic range of pixel-values
    C1 = (k1 * L)**2
    C2 = (k2 * L)**2
    opts = {'ksize': (11, 11), 'sigmaX': 1.5, 'sigmaY': 1.5}
    I1 = np.float64(img1)
    I2 = np.float64(img2)
    # mean
    mu1 = cv2.GaussianBlur(I1, **opts)
    mu2 = cv2.GaussianBlur(I2, **opts)
    # variance
    sigma1_2 = cv2.GaussianBlur(I1**2, **opts) - mu1**2
    sigma2_2 = cv2.GaussianBlur(I2**2, **opts) - mu2**2
    # covariance
    sigma12 = cv2.GaussianBlur(I1*I2, **opts) - mu1*mu2
    # SSIM index
    map_ = ((2 * mu1*mu2 + C1) * (2 * sigma12 + C2)) / \
           ((mu1**2 + mu2**2 + C1) * (sigma1_2 + sigma2_2 + C2))
    val = np.mean(map_)
    return val
cv2.imwrite(
    "E:\SOURCE_CODE\Image_processing\image\edited\Alleysson\multi_img.png",
multi_img)
cv2.imwrite(
    "E:\SOURCE_CODE\Image_processing\image\edited\Alleysson\out.png", out)
cv2.imshow("img", img)
cv2.imshow("ảnh mosaic", multi_img)
cv2.imshow("ảnh đã khôi phục", out)
cv2.waitKey(0)
```