

Rapport de stage : détection de falsifications d'images JPEG

Côme Eupherte, Mickael Assaraf

Encadrants : Tina Nikoukhah, Rafael Grompone von Gioi

Lors de la compression JPEG, certaines traces sont laissées sur l'image. Falsifier une image va localement faire modifier ces traces et créer des inconsistances qui permettent donc de détecter ces falsifications. Nous proposons 3 algorithmes de détection de falsification. Ils utilisent en partie 3 méthodes existantes (une déjà mise en algorithme : partie 4.1 et deux dont la mise en algorithme nous a nécessité un travail d'adaptation théorie-algorithme et une compréhension en profondeur de la méthode : partie 4.2 et 4.3). Ces méthodes ont finalement été utilisées dans le but de détecter des falsifications via une démarche personnelle (partie 5). On finira par comparer ces méthodes à celles déjà existantes (partie 6).

1 Contexte

Ces dernières années, l'évolution de l'imagerie numérique a créé la nécessité de trouver de nouvelles méthodes de vérification pour pallier les problèmes de sécurité. Ces derniers sont de plus en plus présents, notamment dans le domaine de la falsification d'image : fraudes à l'assurance, fake news... De nombreuses méthodes existent et ont déjà fait leurs preuves, mais, avec l'évolution de ces méthodes, les falsificateurs sont aussi devenus plus performants et réalisent des trucages toujours plus difficiles à détecter, d'où le besoin permanent d'amélioration des anciennes méthodes et la création de nouvelles.

Nous nous intéressons dans ce stage aux méthodes concernant des images JPEG ; en effet ce format est le format d'image le plus répandu. Lors de la compression JPEG d'images, l'algorithme de compression laisse des traces identifiables sur l'image, traces sur lesquelles sont basées les méthodes de vérification étudiées.

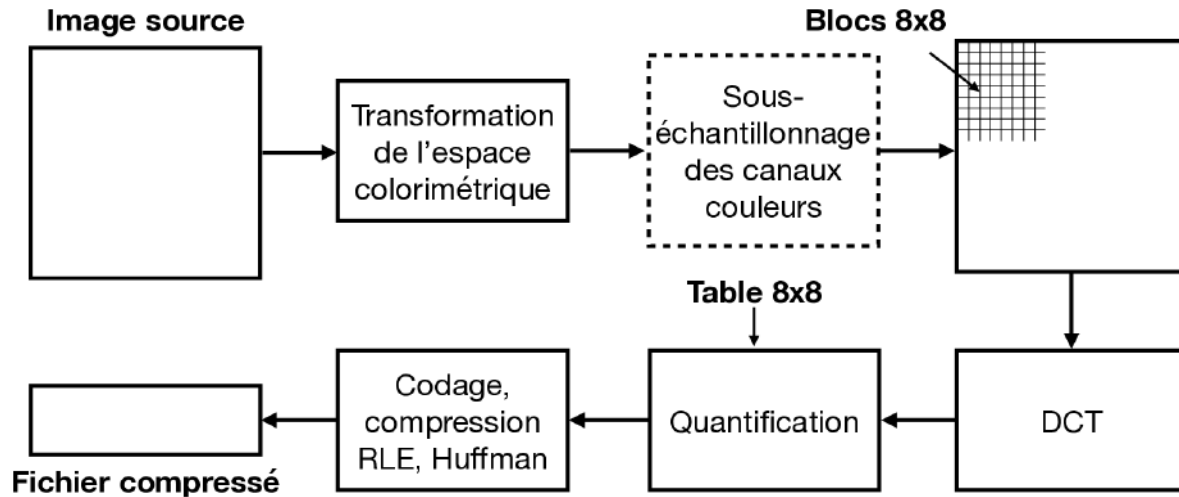
Les falsifications pour lesquelles nos algorithmes ont été testés sont : le clonage (prise d'une partie d'une image pour la dupliquer sur l'image que l'on veut falsifier), le recadrage (l'image originelle a été recadrée pour garder uniquement certaines informations), le gommage (une certaine partie de l'image a été gommée ou floutée) et l'insertion (incrustation d'une partie d'une autre image dans celle que l'on veut falsifier). Toutes ces falsifications étant appliquées à l'image après compression JPEG. En revanche avant que l'image subisse cette compression, elle a subi des modifications causées par l'appareil photo utilisé, lors du transfert de l'appareil vers un ordinateur, etc. Si la falsification a lieu durant ces étapes, ou sur une image n'ayant pas subi de compression JPEG, nos méthodes ne fonctionneront pas (car elles utilisent les propriétés JPEG).

A noter que l'étude de falsifications sur les images JPEG reste très efficace (malgré les cas mentionnés ci-dessus) car JPEG est le format d'image le plus répandu et pour des non-spécialistes, falsifier l'image avant que celle-ci n'ait été compressée est très difficile (car avoir accès à cette image est rare, ça veut dire qu'on a pris nous même la photo et on l'a sauvegardée au format brute).

2 Algorithme JPEG

La première étape pour réaliser ces algorithmes est donc de comprendre l'effet que l'algorithme de compression JPEG a sur une image, et les traces qu'il y laisse.

2.1 Description de l'algorithme JPEG

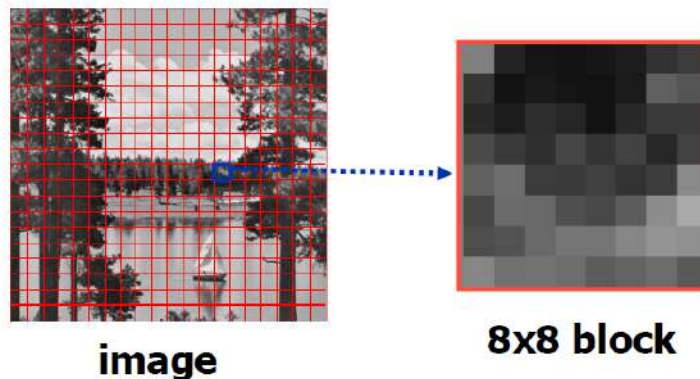


- L'image originelle est une image où chaque pixel est représenté par un triplet d'une matrice correspondant aux valeurs RGB (Red Green Blue) du pixel, elle subit une **transformation de l'espace colorimétrique**. La manière de conserver les valeurs des coefficients est changée de RGB en un système $Y Cr Cb$ (Luminance Chrominance-Rouge Chrominance-Bleue) dans lequel :

$$Y = 0.299 R + 0.587 G + 0.114 B, C_r = \frac{R - Y}{1.402} \quad \text{et} \quad C_b = \frac{B - Y}{1.772}.$$

La luminance correspond à ce que voit principalement l'œil humain. Ainsi lors de la compression JPEG, par souci d'optimisation d'espace de stockage, il y aura plus de pertes d'information dans les canaux Cr et Cb que dans Y (d'où l'intérêt de cette transformation RGB→YCrCb). Puisque Y est le paramètre le plus important de l'image nous nous concentrerons uniquement sur ce canal (bien que les méthodes utilisées puissent être généralisées aux 3 canaux Y , Cr et Cb , cela demanderait un coût algorithmique bien plus grand : exécuter sur chaque canal l'algorithme existant, alors que la précision obtenue en travaillant uniquement sur Y est suffisante).

- Ensuite chaque canal subit un **sous échantillonnage** (lequel dépend de l'algorithme de compression JPEG utilisé). En pratique, souvent, dans les canaux Cr et Cb seule une valeur sur deux est considérée et le canal Y est conservé tel quel.
- Ensuite l'image est séparée en **blocks 8x8** et sur chaque bloc est appliquée une transformation en cosinus discrète (DCT).



- La DCT utilisée ici est la **DCTII**, c'est une transformation similaire à la transformation de Fourier qui va transformer une matrice 8×8 : u de valeurs de pixels (d'un canal) en une matrice 8×8 : U de coefficients (appelés par la suite coefficients DCT). La formule est la suivante :

$$U[n, l] = \sum_{m=0}^7 \sum_{k=0}^7 u[m, k] \cdot \cos\left(\frac{(2m+1)n\pi}{16}\right) \cdot \cos\left(\frac{(2k+1)l\pi}{16}\right), \quad \forall l, n \in \llbracket 0, 7 \rrbracket$$

Cette opération est inversible et ne comporte pas de perte d'information (donc pas de traces laissées sur l'image). L'intérêt de cette DCT2 est que dans la matrice U , les coefficients correspondant à de hautes fréquences du bloc 8x8 (les coefficients en bas à droite de U) impactent assez peu l'image 8x8 originale u . Ainsi on peut perdre un peu d'information sur les valeurs de ces coefficients (dans le but de minimiser l'espace de stockage de l'image JPEG) sans trop perdre les formes de l'image originelle, c'est ce qui est fait dans l'étape suivante.

- Chaque matrice U de coefficients DCT2 passe ensuite par une étape de **quantification** pour obtenir la matrice C stockée dans le fichier JPEG. La matrice C est obtenue par la formule :

$$C = [U/Q],$$

où Q est une matrice de quantification (qui définit le taux de compression JPEG). En voici deux exemples pour une compression de 50% et de 90% (plus ce facteur de qualité est élevé, plus la compression est faible et inversement) :

$$Q_{50} = \begin{pmatrix} 16. & 11. & 10. & 16. & 24. & 40. & 51. & 61. \\ 12. & 12. & 14. & 19. & 26. & 58. & 60. & 55. \\ 14. & 13. & 16. & 24. & 40. & 57. & 69. & 56. \\ 14. & 17. & 22. & 29. & 51. & 87. & 80. & 62. \\ 18. & 22. & 37. & 56. & 68. & 109. & 103. & 77. \\ 24. & 35. & 55. & 64. & 81. & 104. & 113. & 92. \\ 49. & 64. & 78. & 87. & 103. & 121. & 120. & 101. \\ 72. & 92. & 95. & 98. & 112. & 100. & 103. & 99 \end{pmatrix} \quad Q_{90} = \begin{pmatrix} 3. & 2. & 2. & 3. & 5. & 8. & 10. & 12. \\ 2. & 2. & 3. & 4. & 5. & 12. & 12. & 11. \\ 3. & 3. & 3. & 5. & 8. & 11. & 14. & 11. \\ 3. & 3. & 4. & 6. & 10. & 17. & 16. & 12. \\ 4. & 4. & 7. & 11. & 14. & 22. & 21. & 15. \\ 5. & 7. & 11. & 13. & 16. & 12. & 23. & 18. \\ 10. & 13. & 16. & 17. & 21. & 24. & 24. & 21. \\ 14. & 18. & 19. & 20. & 22. & 20. & 20. & 20. \end{pmatrix}$$

La division se réalise terme à terme entre U et Q et $[.]$ correspond à l'arrondi à l'entier le plus proche. Après un encodage (selon un algorithme RLE et Huffman) sans perte d'information (et donc ne laissant pas de traces), seules les matrices C et Q sont stockées dans le fichier JPEG. On peut noter que d'autres informations liées aux autres canaux de couleur sont aussi stockées dans le fichier JPEG (matrices de quantification différentes, information liée au sous échantillonnage ...). Ces informations sur Q peuvent disparaître si l'image JPEG subi d'autres conversions.

La déquantification (le procédé inverse de la quantification) se fait en multipliant terme à terme les coefficients de C et de Q . Ainsi plus les valeurs q_{ij} , $i, j \in \llbracket 1, 8 \rrbracket$ ((i, j) est la fréquence à laquelle correspond q_{ij}) dans Q sont grandes plus il y aura une approximation grossière lorsque l'on quantifie puis déquantifie.

On remarque de plus que dans Q_{50} et Q_{90} les valeurs de q pour les hautes fréquences sont plus élevées que pour les basses fréquences, ce qui découle du fait que les hautes fréquences influent peu sur l'image (visuellement) et sont donc moins importantes à conserver fidèlement.

Pour obtenir une image à partir du fichier JPEG, il faut réaliser tout ce processus dans le sens inverse (on rappelle que Q est stockée dans le fichier JPEG et que toutes les autres étapes sont des opérations inversibles communes à toutes les compressions JPEG).

2.2 Traces sur l'image JPEG

Lors de cette compression, la seule étape avec de la perte d'information est la quantification (la perte étant d'autant plus grande que le taux de compression est faible), c'est donc cette étape qui va laisser des traces sur l'image que l'on utilisera pour nos algorithmes de détection.

$$\text{Image} \xrightarrow{\text{Bloc 8x8 et DCT II}} U \xrightarrow{\text{quantification par } Q_1} C_1$$

On va s'intéresser à la fonction de masse des coefficients C_1 pour lesquels on considère que leur répartition est aléatoire pour une image quelconque (non compressée). On l'appelle \mathbb{P}_{C_1} .

Il est important de connaître d'abord la répartition des coefficients dans U , appelons f_U sa densité.

L'estimation de ces fonctions n'est réalisable qu'à fréquence donnée (en ne considérant que les coefficients correspondant à une fréquence). On travaille donc dans la suite pour une fréquence (i, j) , avec $q = q_{ij}$ son coefficient de quantification.

Le travail dans [3] nous donne f_U :

$$f_U(x) = \sqrt{\frac{2}{\pi}} \frac{\left(|x| \sqrt{\frac{\beta}{2}}\right)^{\alpha - \frac{1}{2}}}{\beta^\alpha \Gamma(\alpha)} \cdot K_{\alpha - \frac{1}{2}} \left(|x| \sqrt{\frac{2}{\beta}}\right)$$

où (α, β) sont des paramètres à déterminer (qui dépendent de la fréquence et du type d'image notamment) et K_θ (la fonction de Bessel modifiée d'ordre θ) et Γ (la fonction gamma) sont des fonctions facilement calculables par algorithme.

On peut en déduire facilement \mathbb{P}_{C_1} (probabilité discrète car C_1 a été quantifié) car la quantification est la composition de deux opérations simples (division et arrondi). Pour une fréquence q_1 donnée :

$$\mathbb{P}_{C_1}(k) = \begin{cases} G(|k|) - G(|k| - 1) & \text{si } k \in \mathbb{Z}^* \\ 2G(0) & \text{si } k = 0 \end{cases}$$

où l'on a posé :

$$G(k) = \frac{1}{2} g(k) \left[K_{\alpha - \frac{1}{2}}(g(k)) L_{\alpha - \frac{3}{2}}(g(k)) + K_{\alpha - \frac{3}{2}}(g(k)) L_{\alpha - \frac{1}{2}}(g(k)) \right]$$

ainsi que $g(k) = q(k + \frac{1}{2}) \sqrt{\frac{2}{\beta}}$ et L est la fonction de Struve modifiée, ces deux fonctions étant facilement calculables par algorithme. Finalement on a donc accès à la répartition des coefficients de C_1 en fonction de q_1 et de deux paramètres (α, β) (qui dépendent de l'image utilisée).

Malheureusement il arrive souvent que l'on n'ait pas accès directement aux coefficients C_1 : dans les cas où on n'a pas accès au fichier JPEG mais seulement à l'image finale (par exemple si on essaie d'analyser une image dans un journal papier, ou si l'image JPEG a été transformée en PNG avant d'être publiée). Dans ce cas pour obtenir les coefficients C_1 il suffit en théorie de ré-appliquer la séparation en blocs 8x8, la DCTII et la quantification à l'image JPEG. Le problème est que lorsque l'image JPEG est affichée (en pixels de couleurs), la valeur des couleurs des pixels ne peut être que entière, l'algorithme de décompression JPEG applique donc un arrondi sur ces valeurs avant d'afficher l'image. Il y a donc une erreur ε qui apparaît entre les coefficients DCT estimés après compression et les vrais coefficients DCT avant compression.

$$C_1 \cdot Q_1 \xrightarrow{DCT^{-1} \text{ puis arrondi}} \text{image JPEG} \xrightarrow{DCT} C_1 \cdot Q_1 + \varepsilon$$

$$\varepsilon = C_1 \cdot Q_1 - DCTII([DCTII^{-1}(C_1 \cdot Q_1)])$$

En vertu du théorème centrale limite, ε est supposé de distribution Gaussienne : $\varepsilon \sim \mathcal{N}(0, \frac{1}{12})$ (les paramètres étant déterminés dans [2]).

Ainsi quand on réapplique $DCTII$ à l'image pour obtenir les coefficients $C_1 \cdot Q_1$ (qui sont censés être quantifiés) on obtient des erreurs Gaussiennes autour de chaque valeur de quantification :

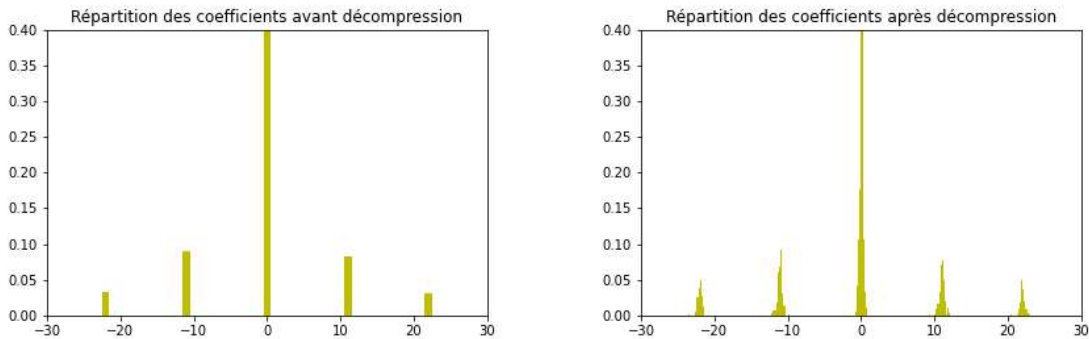


figure 1 : Histogrammes des répartitions des valeurs des coefficients de $C_1 \cdot Q_1$ réelles et estimées après décompression pour une quantification $q_1 = 11$

3 Etat de l'art

3.1 Méthodes Diverses

Il y a plusieurs méthodes existantes qui utilisent les traces laissées par l'algorithme JPEG pour détecter les falsifications d'images. Les méthodes que nous avons étudiées sont de deux types : celles qui repèrent les grilles de quantification JPEG et celles qui analysent la répartition des coefficients DCT.

- **Méthodes de repérage de la grille des blocks 8x8** : on détecte une grille JPEG et si celle-ci n'est pas alignée avec le pixel (0,0) (comme c'est le cas dans l'algorithme JPEG) ou si plusieurs (ou aucune) grilles différentes apparaissent sur l'image alors il y a eu falsification.

La méthode [7] applique une transformation DCT aux 64 grilles JPEG possibles afin de détecter l'origine de la grille (la "bonne" grille étant celle qui donne la taille de fichier la plus faible, car *a priori* si la grille test et la vraie grille JPEG ne sont pas alignées, alors on perd l'optimisation d'espace liée à la quantification).

La méthode [8] repère la grille en recherchant des maxima locaux dans les valeurs des couleurs des pixels (en effet la compression JPEG fait apparaître des discontinuités faibles des couleurs entre les blocks 8x8 et donc des maxima des valeurs YC_rC_b le long de la grille JPEG).

La méthode [6], appelée ZERO, repère la grille globale en cherchant pour chaque pixel quel block 8x8 parmi les 64 blocks contenant ce pixel possède le plus de zéro après re-transformation DCT, en effet la compression JPEG réduit à zéro beaucoup de coefficients DCT (et *a priori* si le block 8x8 n'est pas aligné avec la grille JPEG il n'y a pas de raison que des zéros apparaissent).

Ces méthodes ont l'avantage d'être rapides et de plutôt bien marcher la plupart du temps mais ne peuvent pas repérer de falsification d'image pour un collage d'une image JPEG sur l'image à falsifier si les grilles JPEG sont alignées (1 chance sur 64 si la falsification est faite sans faire attention à ce point).

- **Méthodes d'analyse des coefficients DCT** : elles sont utilisées en partie pour pallier ce défaut des méthodes précédentes.

La méthode GHOST [1] re-compresse l'image pour différents facteurs de qualité Q_F . Ainsi pour le bon facteur de re-compression (celui correspondant à l'image originelle), on est censé très peu modifier l'image originelle. Si la partie falsifiée n'est pas compressée avec le même facteur que l'image originelle ou si les grilles JPEG ne sont pas alignées en re-compressant avec le Q_F original, on verra que la seule partie de l'image à avoir été modifiée par la re-compression est la partie falsifiée. Cependant celui qui souhaite détecter une falsification doit analyser par lui-même différentes re-compressions (pour différents Q_F) pour pouvoir donner un résultat.

D'autres méthodes s'intéressent aux traces laissées par la quantification : DQ [5] repère la périodicité observable des valeurs des coefficients DCT après une double compression et BAM [12] estime une matrice Q (qui correspond à la matrice utilisée lors de la compression JPEG de l'image) et regarde les erreurs entre les coefficients DCT et les multiples de cette matrice Q estimée.

3.2 Estimation de la matrice de quantification Q

La méthode que nous avons mise au point s'inspire de la méthode BAM mais utilise une estimation différente de la matrice de quantification Q . Nous nous sommes donc intéressés à différentes méthodes d'estimation de cette matrice.

Dans l'algorithme BAM, au vu de la répartition périodique des coefficients DCT après décompression (figure), on fait subir une transformée de Fourier à l'histogramme puis on dérive le résultat obtenu deux fois, le nombre de minima locaux est alors $q - 1$, ce qui, appliqué à chaque fréquence, donne Q .

Une autre méthode [11] consiste à étudier pour chaque valeur de quantification q testée l'erreur entre les valeurs des coefficients DCT et le multiple de q le plus proche, puis grâce à une validation statistique dite "*a contrario*" on choisit la bonne valeur. Cette méthode présente l'avantage d'avoir une validation statistique ce qui permet d'éviter des valeurs aberrantes dans la matrice de quantification estimée. Cependant pour la validation elle fait l'hypothèse que la répartition des erreurs des coefficients DCT dans le cas non quantifié est uniforme ce qui est une approximation valable mais reste une approximation (voir partie 4.1).

D'autres méthodes calculent explicitement la densité des coefficients DCT avant quantification [10] et après quantification [9]. Ainsi après un choix de candidats de coefficients de quantification possibles, on peut estimer le coefficient le plus probable en utilisant la méthode de maximum likelihood [10] (voir partie 4.2).

Une autre méthode [9] concernant les images qui ont été doublement compressées consiste à utiliser la distribution des coefficients DCT après une quantification pour en déduire celle après une seconde quantification. Ainsi connaissant la seconde quantification (soit donnée par le fichier JPEG soit en réalisant la deuxième compression nous même) on peut calculer la répartition des coefficients DCT après la deuxième quantification en fonction de la matrice de quantification de la première. Après un choix

de candidats de coefficients de quantification on peut déterminer cette matrice en regardant le minimum (sur les q testés) de la distance entre la densité estimée des coefficients DCT et l'histogramme de ces derniers (voir partie 4.3).

4 Calcul de Q

4.1 Méthode avec fausses détections contrôlées (FDC)

Nous avons commencé pour prendre en main la manipulation d'images JPEG par coder l'algorithme avec fausses détections contrôlées décrit dans [11].

- En entrée de l'algorithme on donne une image, donc la première étape est de réaliser la DCTII (sur l'image déjà compressée par JPEG):

$$\text{Image JPEG} \xrightarrow{\text{Bloc } 8 \times 8 \text{ et DCT II}} U$$

```

11 # implement 2D DCT
12 def dct2(a):
13     return dct(dct(a.T, norm='ortho').T, norm='ortho')
14
15 # implement 2D IDCT
16 def idct2(a):
17     return idct(idct(a.T, norm='ortho').T, norm='ortho')
18
19 def dctblock(image):
20     X,Y,dim=image.shape
21     lumi=0.3*image[:, :, 0]+0.6*image[:, :, 1]+0.1*image[:, :, 2]
22     lesblock=[]
23     for i in range(0,X,8):
24         for j in range(0,Y,8):
25             if j+8<Y and i+8<X:
26                 lesblock.append(dct2(lumi[i:i+8,j:j+8] ))
27
28     return(lesblock)

```

Ce code pour effectuer les DCTII sera utilisé dans les autres méthodes tel quel, il rend une liste unidimensionnelle de tous les blocks 8x8 de coefficients DCT.

On va travailler par la suite pour une fréquence (l'algorithme réalise les étapes suivantes pour chaque fréquence pour déterminer la matrice Q qui a servi à faire la compression JPEG de l'image), les coefficients et les erreurs considérés sont donc seulement ceux d'une fréquence.

- On va ensuite déterminer la valeur q contenue dans Q en comparant les erreurs entre les coefficients de U et le multiple de q le plus proche (erreurs qui suivent des lois gaussiennes, voir figure 1). Si on normalise ces erreurs (pour qu'elles soient contenues dans $[0, 1]$) alors l'erreur est censée être minimale pour $q_{\text{testé}} = q_{\text{JPEG}}$. En pratique on va donc sommer pour tous les coefficients u d'une fréquence l'erreur :

$$e = 2 \left| \frac{u}{q_{\text{testé}}} - \left\lceil \frac{u}{q_{\text{testé}}} \right\rceil \right|.$$

On exclut le cas $\left\lceil \frac{u}{q_{\text{testé}}} \right\rceil = 0$ car même pour une image non JPEG la répartition des coefficients autour de 0 suit une loi qui ressemble à une Gaussienne (f_U dans la partie 2.2). Donc les erreurs sont semblables dans un cas JPEG comme dans un cas non JPEG et ne permettent pas de différencier les deux. Ce qui nous donne l'algorithme suivant :

```

37 for i in range(0,8):
38     for j in range(0,8):
39         if i!=0 or j!=0:
40             for q in range(1,256):
41                 s=0
42                 n=0
43                 for block in lesblock:

```

```

44     v=block[i,j]
45     V=round(v/q)
46     if V!=0:
47         e=2*np.abs((v/q)-V)
48         s=s+e
49         n=n+1

```

- Plutôt que de simplement considérer que le bon q correspond à celui pour lequel la somme des erreurs s est minimale, on réalise ensuite une validation statistique basée sur la méthode *a contrario*. Celle-ci suppose que si notre faible erreur, s , n'est pas apparue grâce à une compression, alors elle est apparue au hasard. Donc on calcule la probabilité que s apparaisse de manière aléatoire et si cette probabilité est suffisamment faible alors il y a eu compression JPEG. En pratique, on calcule le nombre de fausses alarmes (NFA) qui correspond au nombre de fois où sur une image non-JPEG quelconque l'erreur s serait apparue en moyenne :

$$NFA(s) = N_{\text{test}} \cdot \mathbb{P}(S \leq s)$$

où N_{test} est le nombre de fois où un calcul de s est réalisé dans notre algorithme (63 fréquences et à chaque fréquence 255 q différents sont testés) : $N_{\text{test}} = 63 \times 255$. Et S est la variable aléatoire donnant l'erreur s pour une image non-JPEG. Pour avoir une idée de la répartition de S on trace les erreurs pour une image non-JPEG :

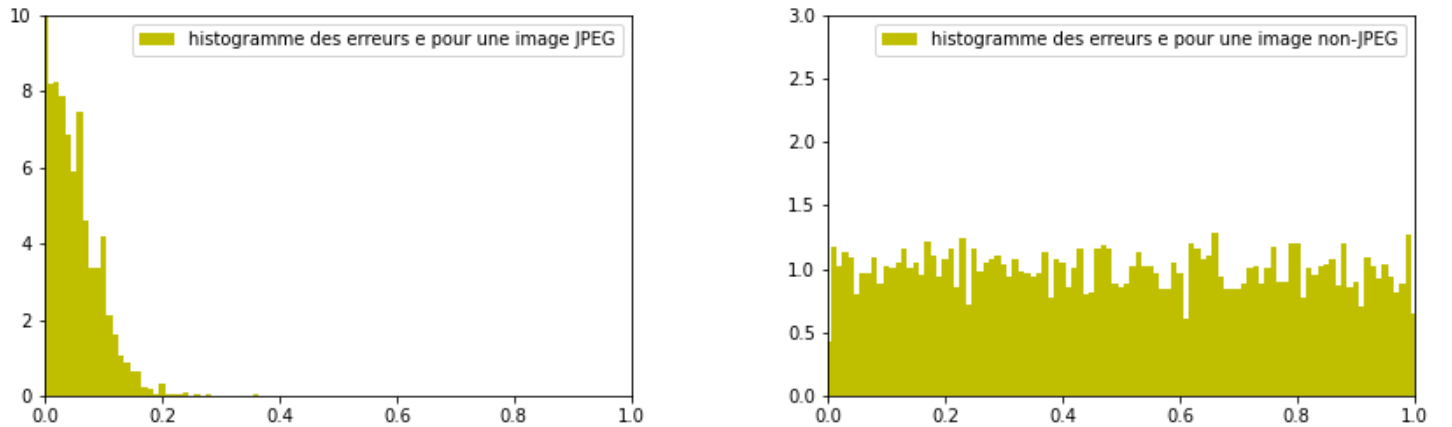


figure 2 : Histogrammes des erreurs e calculées pour $q_{\text{testé}} = 8$ (qui correspond au q de la compression JPEG)

On vérifie ainsi que l'erreur pour une image JPEG suit une loi Gaussienne et on approche l'erreur e pour une image non-JPEG par une variable aléatoire uniforme à valeur dans $[0, 1]$. En réalité cette répartition n'est pas exactement uniforme mais la nature de f_U (voir figure 2) assure que cette approximation est valable.

A noter que si l'on n'enlève pas les coefficients proches de zéro (ce qui correspond à la ligne 46 de l'algorithme), on obtient une répartition des erreurs comme ci-dessous et on ne peut pas donner de modèle satisfaisant pour S .

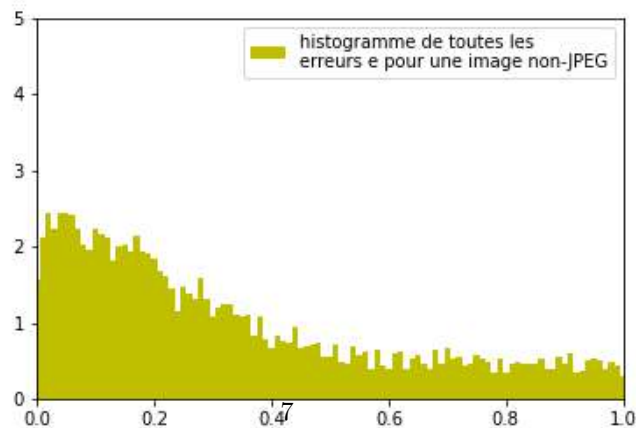


figure 3 : Histogrammes de toutes les erreurs e d'une image non compressée pour $q_{\text{testé}} = 8$

Ainsi $S = \sum_{i=1}^n e_i$ (n est le nombre de coefficients pour lesquels e est calculé) suit une distribution de Irwin-Hall et alors :

$$\begin{aligned}\mathbb{P}(S \leq s) &= \frac{1}{n!} \sum_{k=0}^{\lfloor s \rfloor} (-1)^k \binom{n}{k} (s-k)^n \\ &\leq \frac{s^n}{n!}.\end{aligned}$$

Ici on veut réduire le nombre de fausses alarmes pour qu'en moyenne il n'y en ait pas, on imposera donc $NFA(s) \leq 1$ comme test permettant de considérer qu'une valeur d'erreur s est suffisamment faible pour que le $q_{\text{testé}}$ soit significatif.

On utilisera alors $NFA(s) = 255 \times 63 \times \frac{s^n}{n!}$.

```

51 |         nfa=np.log10(63*255)+n*np.log10(s)-n*np.log10(n)+n*np.log10(np.exp(1))-(1/2)*np.
52 |         log10(2*n*np.pi)
53 |         if nfa<NFA[i,j] and nfa<=0:
54 |             Q[i,j]=q
55 |             NFA[i,j]=nfa

```

Algorithmiquement le NFA est plus simplement calculé avec des log, la factorielle est calculée avec l'approximation de Stirling et le q finalement choisi est celui avec le NFA le plus faible (si celui-ci est inférieur à 1).

• Résultats :

On a testé cette méthode sur des images JPEG dont on connaissait les matrices de quantification, voici un résultat typique :

$$Q_{\text{calculé}} = \begin{pmatrix} -1. & 9. & 8. & 13. & 19. & 32. & 41. & 48. \\ 10. & 10. & 11. & 15. & 21. & 45. & 47. & 44. \\ 11. & 10. & 13. & 19. & 32. & 45. & 54. & 45. \\ 11. & 14. & 18. & 23. & 40. & 69. & 63. & 49. \\ 14. & 18. & 30. & 44. & 53. & 86. & -1. & -1. \\ 19. & 28. & 43. & 50. & 64. & 82. & -1. & -1. \\ 39. & 50. & 61. & 69. & -1. & -1. & -1. & -1. \\ 57. & 74. & -1. & -1. & -1. & -1. & -1. & -1. \end{pmatrix} \quad Q_{\text{réel}} = \begin{pmatrix} 13. & 9. & 8. & 13. & 19. & 32. & 41. & 49. \\ 10. & 10. & 11. & 15. & 21. & 46. & 48. & 44. \\ 11. & 10. & 13. & 19. & 32. & 46. & 55. & 45. \\ 11. & 14. & 18. & 23. & 41. & 70. & 64. & 50. \\ 14. & 18. & 30. & 45. & 54. & 87. & 82. & 62. \\ 19. & 28. & 44. & 51. & 65. & 83. & 90. & 74. \\ 39. & 51. & 62. & 70. & 82. & 97. & 96. & 81. \\ 58. & 74. & 76. & 78. & 90. & 80. & 82. & 79. \end{pmatrix}$$

Un des avantages de cette méthode est qu'elle permet de s'assurer que les coefficients donnés sont très proches des vrais coefficients grâce à la validation par méthode *a contrario* : lorsque qu'aucune erreur n'est significative ($NFA(s) > 1$) alors on affiche -1 .

On remarque que le premier coefficient qui correspond à la composante continue des blocks 8x8 n'est pas calculé, cela est du au fait qu'il répond à d'autres propriétés que les coefficients correspondants aux composantes alternatives (notamment f_U est faux pour la fréquence 0). C'est une remarque qui restera valable dans les méthodes suivantes. Les coefficients de Q correspondant à de grandes fréquences (HF) ont aussi du mal à être calculés car lors de la quantification les coefficients DCT HF sont quantifiés avec un plus grand q qu'en basses fréquences. Ainsi il y a plus de chances que les coefficients HF soient approchés à 0 et donc l'algorithme a moins de coefficients non proches de 0 sur lesquels travailler, ce qui donne une précision en HF plus faible.

4.2 Méthode ML

Ensuite nous avons étudié deux méthodes théoriques utilisant la répartition des coefficients DCT [10] pour lesquelles les algorithmes de détection n'ont jamais été fournis. D'après l'étude 2.2, il reste à estimer les valeurs de α et de β . Pour cela on utilise une méthode de maximum likelihood. Celle-ci se base sur le principe que si on a un certain nombre d'échantillons (x_i) d'une variable aléatoire X (ici ce sont les coefficients DCT à une fréquence donnée qui suivent une loi f_U) et que l'on connaît un certain nombre de probabilités approchant $\mathbb{P}_i(X)$, alors la probabilité approchant le mieux $\mathbb{P}(X)$ est \mathbb{P}_j avec j vérifiant

$$j = \operatorname{argmax}_i \sum_k \log(P_i(x_k)).$$

En appliquant ça aux échantillons de coefficients (u_i) (coefficients de U pour une fréquence donnée), et aux fonctions f_U pour différents (α, β) on peut en déduire le (α, β) correspondant à l'image,

$$(\alpha_{ML}, \beta_{ML}) = \operatorname{argmax}_{(\alpha, \beta)} \sum_{i=1}^N \log(f_U(u_i)).$$

Grâce à la méthode de maximisation de Nelder-Mead on peut résoudre cela numériquement en donnant comme valeur initiale

$$(\alpha_{MM}, \beta_{MM}) = \left(\frac{3}{\frac{m_4}{m_2^2} - 3}, \frac{m_2}{\alpha_{MM}} \right)$$

où m_i est le i -ème moment de U .

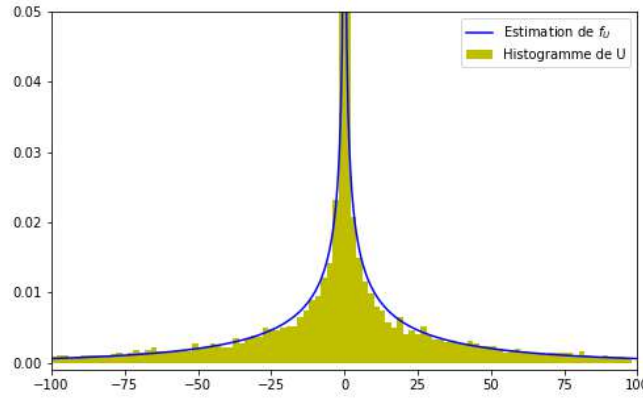


figure 4 : Estimation de f_U (à partir de U) par rapport à l'histogramme des coefficients de U pour une fréquence donnée

```

22 def f(x,alpha,beta):
23     a=kv(alpha-1/2,np.abs(x)*np.sqrt(2/beta))
24     b=np.sqrt(2/np.pi)*((np.abs(x)*np.sqrt(beta/2))**(alpha-1/2))/(gamma(alpha)*beta**alpha)
25     return(a*b)
26
27 def sumf(alpha,beta,Vk):
28     A=-np.sum(np.log(f(Vk,alpha,beta)))
29     return A
30
31 def estimationML(Vk):
32     #Vk est le vecteur des coefficients U pour une frequence donnee
33     M4=moment(Vk,moment=4)
34     M2=moment(Vk,moment=2)
35     alphaM=3/(M4/(M2**2)-3)
36     betaM=M2/alphaM
37     x0=np.array([alphaM,betaM])
38     def minimf(x):
39         return sumf(x[0],x[1],Vk)
40     res=minimize(minimf,x0,method='Nelder-Mead')
41     [alphaML,betaML]=res.x
42     return alphaML,betaML

```

Cependant on n'a en pratique pas accès aux coefficients U , en revanche on a les coefficients C_1 (obtenus après quantification de U). La régularité de f_U assure que si on applique la méthode ML aux coefficients C_1 on obtiendra un résultat proche de celui

qu'on aurait en réalisant ML sur U et donc une bonne approximation de (α, β) .

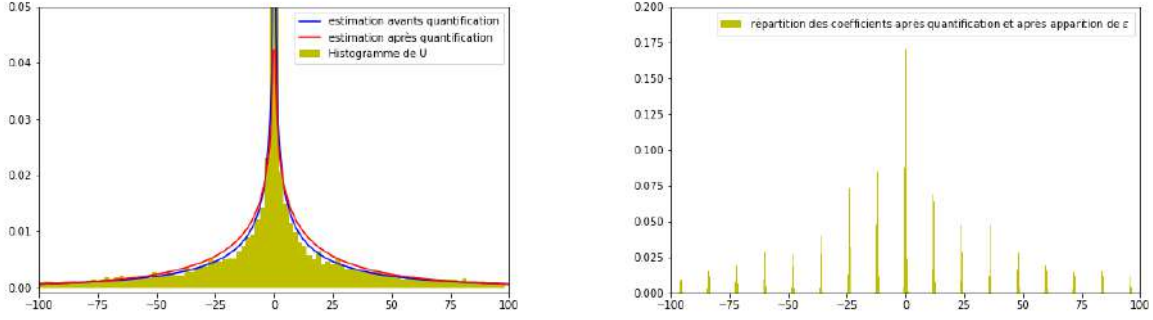


figure 5 : Estimations de f_U (une à partir de U et une à partir des coefficients obtenus après quantification et après apparition de ε) et à droite l'histogramme des coefficients servant à estimer la deuxième estimation de f_U

On va ensuite choisir les candidats les plus probables pour la valeur de la quantification q pour chaque fréquence. Lorsqu'on applique une DCT à une image JPEG on obtient les coefficients $\tilde{u} = cq + \varepsilon$ (voir 2.2). On peut noter alors $\tilde{c} = \frac{[\tilde{u}]}{\tilde{q}}$ où \tilde{q} est le coefficient test. En notant $\tilde{N} = \sum \mathbb{1}[\tilde{u}] = cq$ (le nombre de coefficients après deuxième DCT qui seront égaux à ceux de la première quantification à un arrondi près) et $n_{\tilde{q}} = \sum \mathbb{1}[\tilde{c} \in \mathbb{Z}]$ (on simule une deuxième quantification et $n_{\tilde{q}}$ est le nombre de coefficients \tilde{u} qui seraient inchangés par cette deuxième quantification) on peut distinguer 3 cas :

- $\tilde{q} = 1$: on a alors $n_{\tilde{q}} = N$ c'est à dire $\rho_{\tilde{q}} = \frac{n_{\tilde{q}}}{N} = 1$;
- $\tilde{q} > 1$ et $\tilde{q}|q$: on a alors $n_{\tilde{q}} = \tilde{N}$ c'est à dire $\rho_{\tilde{q}} = 0.916$. Ceci est calculé grâce à la loi normale de ε vue précédemment car on a : $\tilde{c} = \frac{[\tilde{u}]}{\tilde{q}} = \frac{[cq + \varepsilon]}{\tilde{q}} = c \frac{q}{\tilde{q}} + \frac{[\varepsilon]}{\tilde{q}}$ donc $\tilde{c} \in \mathbb{Z}$ ne dépend que de ε (en ignorant les cas $|\varepsilon| > 1.5$ car très improbables).
Alors $\frac{\tilde{N}}{N} \approx \mathbb{P}([\varepsilon] = 0) = 0.916$;
- $\tilde{q} > 1$ et $\tilde{q} \nmid q$: on a alors $n_{\tilde{q}} < \tilde{N}$ c'est à dire $\rho_{\tilde{q}} < 0.916$ (en supposant encore une fois $\mathbb{P}(|\varepsilon| > 1.5) = 0$).

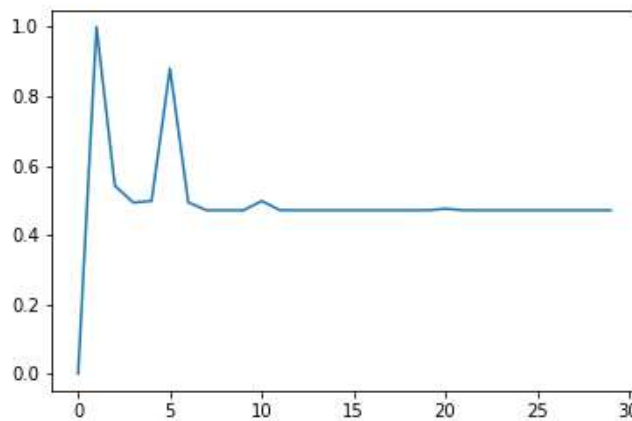


figure 6 : Evaluations de $\rho_{q_{\text{testé}}}$ en fonction de $q_{\text{testé}}$ pour un vrai facteur de quantification JPEG : $q = 5$

On peut alors choisir un ensemble S de coefficients possibles :

$$S = \left\{ \tilde{q}, \tilde{q}|\tilde{Q} \text{ et } \rho_{\tilde{q}} \geq t \right\}$$

avec \tilde{Q} le bin de l'histogramme contenant le plus de coefficients non nuls et t un seuil empiriquement choisi. Ainsi, en appliquant la méthode du maximum likelihood, on peut estimer q^* (le q le plus probable) :

$$q^* = \operatorname{argmax}_{\tilde{q} \in S} \sum_{i=1}^{n_{\tilde{q}}} \log P_{C_1}(c_i | \alpha_{ML}, \beta_{ML}, \tilde{q})$$

où $c_i = \frac{u_i}{\tilde{q}}$.

Le problème est que $P_{C_1}(c_i)$ est seulement défini quand c_i est entier. Donc le nombre de coefficients pour lesquels la probabilité d'apparition peut être évaluée est limité (et vaut $n_{\tilde{q}}$). Donc dans cette méthode de maximum likelihood les échantillons de la variable aléatoire (dont on cherche à connaître la loi) changent en fonction de \tilde{q} et donc de la probabilité testée. Cela rend donc en théorie cette estimation de q^* fautive (mauvaise application d'une méthode ML).

C'est pour cela qu'on a fait un choix préalable de S qui va assurer que pour les \tilde{q} testés on aura des $n_{\tilde{q}}$ proches les uns des autres, ce qui justifie alors l'utilisation de la méthode ML.

Malheureusement cela ne résout pas totalement ce problème et on a pas réussi à faire marcher la méthode ML algorithmiquement, il est donc nécessaire de considérer une autre méthode, avec une probabilité qui prend en compte l'apparition de ε .

4.3 Methode DKL

Ainsi nous avons étudié une troisième méthode [9], se basant aussi sur la formule de la densité de probabilité de la répartition des coefficients DCT. Elle utilise une seconde quantification et on traitera les cas suivants : soit l'image est une seule fois quantifiée, on cherche sa matrice de quantification pour cela on re-quantifie alors l'image avec un coefficient q_2 uniforme sur toute l'image, soit l'image est deux fois quantifiée et sous format JPEG on lit alors l'entête de l'image pour obtenir la matrice de la seconde quantification Q_2 afin de déterminer la matrice de la première quantification Q_1 .

$$c_1 \xrightarrow{\text{Déquantification}} c_1 \cdot q_1 \xrightarrow[\text{puis DCT II}]{\text{IDCT II puis arrondi}} c_1 \cdot q_1 + \varepsilon \xrightarrow{\text{Quantification } q_2} c_2$$

Détaillons les étapes de la seconde déquantification : pour chaque fréquence, pour un coefficient de quantification q_2 on obtient :

$$c_2 = \left\lfloor \frac{c_1 \cdot q_1 + \varepsilon}{q_2} \right\rfloor$$

où c_2 est le coefficient DCT quantifié une seconde fois et ε l'erreur liée à l'arrondi lors de la première déquantification (voir 2.2). L'objectif est de retrouver q_1 on va alors donner un intervalle dans lequel doit se trouver $c_1 \cdot q_1$ pour que le coefficient c_2 puisse être obtenu (toujours en considérant $\varepsilon \in [-1.5, 1.5]$).

Théorème 4.1. *Pour un coefficient de seconde quantification q_2 , la valeur de $c_1 q_1$ se trouve dans \mathcal{R}_{q_2} définie par :*

$$\mathcal{R}_{q_2}(c_2) = \left[c_2 q_2 - \left\lfloor \frac{q_2}{2} + \varepsilon \right\rfloor, c_2 q_2 + \left\lfloor \frac{q_2}{2} - \varepsilon \right\rfloor - 1 \right]$$

- Cas 1 : $q_2 = 2m, m \in \mathbb{Z}$, comme $\varepsilon \sim \mathcal{N}(0, \frac{1}{12})$ on peut dire que la probabilité que ε soit en dehors de $[-\frac{3}{2}, \frac{3}{2}]$ est négligeable. Ainsi on obtient les cas suivants :

$$\begin{cases} \mathcal{R}_{q_2}^{(1)}(c_2) = [c_2 q_2 - m + 2, c_2 q_2 + m + 1] & \text{si } \varepsilon \in I^{(1)} \\ \mathcal{R}_{q_2}^{(2)}(c_2) = [c_2 q_2 - m + 1, c_2 q_2 + m] & \text{si } \varepsilon \in I^{(2)} \\ \mathcal{R}_{q_2}^{(3)}(c_2) = [c_2 q_2 - m, c_2 q_2 + m - 1] & \text{si } \varepsilon \in I^{(3)} \\ \mathcal{R}_{q_2}^{(4)}(c_2) = [c_2 q_2 - m - 1, c_2 q_2 + m - 2] & \text{si } \varepsilon \in I^{(4)} \end{cases}$$

où $I^{(1)} = [-\frac{3}{2}, -1[$, $I^{(2)} = [-1, 0[$, $I^{(3)} = [0, 1[$, $I^{(4)} = [1, \frac{3}{2}]$.

- Cas 2 : $q_2 = 2m + 1, m \in \mathbb{Z}$.

Dans ce cas, $\mathcal{R}_{q_2}(c_2)$ devient :

$$\begin{cases} \mathcal{R}_{q_2}^{(1)}(c_2) = [c_2 q_2 - m + 1, c_2 q_2 + m + 1] & \text{si } \varepsilon \in I^{(1)} \\ \mathcal{R}_{q_2}^{(2)}(c_2) = [c_2 q_2 - m, c_2 q_2 + m] & \text{si } \varepsilon \in I^{(2)} \\ \mathcal{R}_{q_2}^{(3)}(c_2) = [c_2 q_2 - m - 1, c_2 q_2 + m - 1] & \text{si } \varepsilon \in I^{(3)} \end{cases}$$

où $I^{(1)} = \left[-\frac{3}{2}, -\frac{1}{2}\right]$, $I^{(2)} = \left[-\frac{1}{2}, \frac{1}{2}\right]$, $I^{(3)} = \left[\frac{1}{2}, \frac{3}{2}\right]$.

Corrolaire 4.2. Notons p_j , $1 \leq j \leq n$ la probabilité que l'erreur ε soit dans l'intervalle $I^{(j)}$ et $\bar{p}(k)$ la probabilité que la valeur $k \in \mathcal{R}_{q_2}(c_2)$ (k correspond à une valeur de $c_1.q_1$) soit celle engendrant c_2 . On peut exprimer $\bar{p}(k)$ comme :

$$\bar{p}(k) = \sum_{j=1}^n \mathbf{1} \left[k \in \mathcal{R}_{q_2}^{(j)}(c_2) \right] \cdot p_j,$$

où $\mathbf{1}[A]$ est la fonction indicatrice de l'évènement A .

Proof. Ce corollaire découle de la formule des probabilités totales appliquée à la disjonction de cas précédente. \square

On peut en déduire :

$$\begin{aligned} P_{C_2}(c_2) &= \mathbb{P}[C_2 = c_2] \\ &= \sum_{k \in \mathcal{R}_{q_2}(c_2), k=q_1 l} \mathbb{P}[k | C_2 = c_2] \cdot \mathbb{P}[C_1 q_1 = k] \\ &= \sum_{k \in \mathcal{R}_{q_2}(c_2), k=q_1 l} \mathbb{P}[k | C_2 = c_2] \cdot \mathbb{P}[C_1 = l] \end{aligned}$$

Ainsi on déduit la densité de probabilité pour les coefficients DCT quantifiés deux fois :

$$P_{C_2}(c_2) = \sum_{k \in \mathcal{R}_{q_2}(c_2), k=q_1 l} \bar{p}(k) \cdot P_{C_1}(l)$$

laquelle dépend uniquement de q_2 (qui est connue) et de q_1 (qui est à déterminer).

Maintenant, on va déduire de cette formule le pas de quantification q_1 , tout d'abord on va effectuer un filtrage des candidats q_1 les plus probables. Soit $\kappa_{c_2}(q)$ (la fonction indicatrice de l'évènement "il existe une chance que q ait engendré c_2 ") :

$$\kappa_{c_2}(q) = \mathbf{1} [\exists k \in \mathbb{Z} \mid kq \in \mathcal{R}_{q_2}(c_2)].$$

On remarque que pour c_2 fixé, si $q = q_1$ alors $\kappa_{c_2}(q) = 1$, d'où on obtient \mathcal{S} , l'ensemble des coefficients possibles :

$$\mathcal{S} = \left\{ q \in \mathbb{Z}_* \mid \frac{1}{N} \sum_{c_2 \in \mathcal{C}_2} \kappa_{c_2}(q) = 1 \right\}.$$

En pratique $\frac{1}{N} \sum_{c_2 \in \mathcal{C}_2} \kappa_{c_2}(q)$ peut avoir une valeur inférieure à 1 à cause de l'erreur ε ($|\varepsilon| > 1.5$ est improbable mais pas impossible) et le modèle qui n'est parfait lors de la deuxième quantification. On prend donc :

$$\mathcal{S} = \left\{ q \in \mathbb{Z}_* \mid \frac{1}{N} \sum_{c_2 \in \mathcal{C}_2} \kappa_{c_2}(q) \geq 0.9 \right\}.$$

Finalement, comparons la densité de probabilité de C_2 (dépendant de q_1) obtenue théoriquement et l'histogramme normalisé de C_2 :

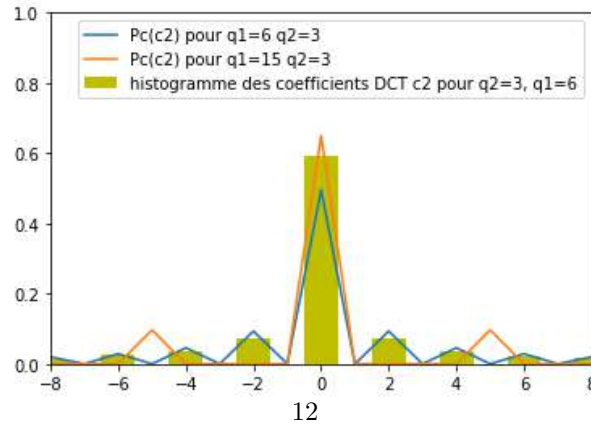


figure 7: Comparaisons de P_{C_2} calculées pour $q_1 = 6$ et $q_1 = 15$ et de l'histogramme des coefficients C_2 obtenue pour une quantification avec $q_1 = 6$

On remarque que, comme attendu, la fonction théorique correspond mieux à l'histogramme pour le bon coefficient $q_1 = 6$ (pour des *bins* différents de 0).

Ainsi on va introduire une distance pour faire la comparaison entre les deux, l'article propose l'utilisation de la distance DKL :

$$\bar{D}_{KL}(q) = \sum_{|i| \geq 2} H(i) \log \frac{H(i)}{P_{C_2}(i | \theta)} + \sum_{|i| \geq 2} P_{C_2}(i | \theta) \log \frac{P_{C_2}(i | \theta)}{H(i)}.$$

Cependant comme on peut le voir figure 7, les valeur $H(i)$ de l'histogramme et P_{C_2} peuvent valoir zéro ce qui donne des valeurs indéfinies ou infinies. Pour pallier ce problème, nous avons choisi la distance L_1 définie par :

$$D_{L_1}(q) = \sum_{|i| \geq 2} |H(i) - P_{C_2}(i | \theta)|.$$

La raison de ce choix est purement empirique, la distance L_1 donne les meilleurs résultats. On va alors pour obtenir le pas de quantification q_1 considérer l'arg min de cette distance :

$$q^* = \arg \min_{q \in \mathcal{S}} D_{L_1}(q).$$

Finalement le choix de filtrer les candidats avec \mathcal{S} n'est pas bénéfique au temps de calcul, on a donc préféré prendre un jeu de valeurs pour \mathcal{S} entre 1 et 255.

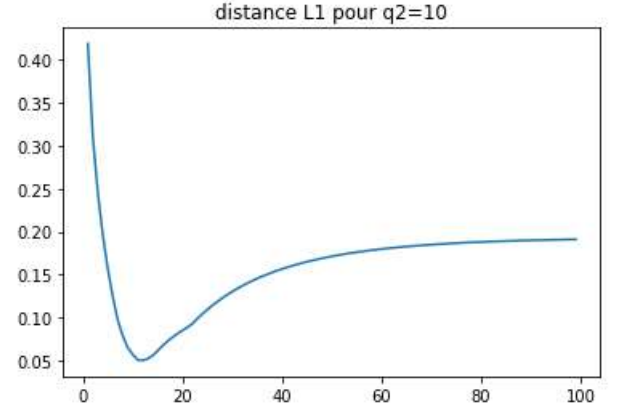
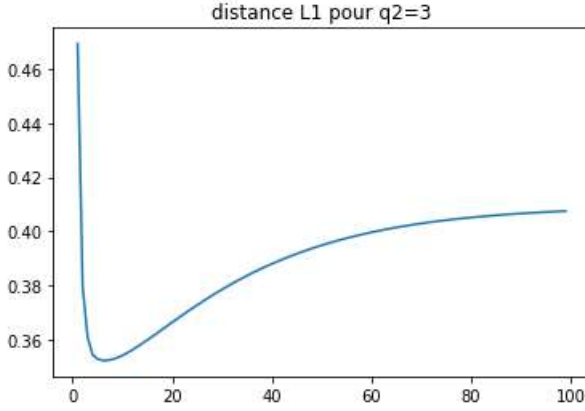


figure 8 : Distance L_1 entre histogramme et P_{C_2} en fonction de q_1 pour et $q_2 = 3$ (pour une image quantifié avec $q_1 = 6$)

figure 9 : Distance L_1 entre histogramme et P_{C_2} en fonction de q_1 pour et $q_2 = 10$ (pour une image quantifié avec $q_1 = 6$)

Cependant comme sur les figures ci-dessus, on obtient un minimum correspondant au q_1 de l'image seulement si $q_2 < q_1$. On en déduit un premier algorithme pour calculer la distance L_1 :

```
1 def sumPc(V,q2,q,a2,b2):
2     somme=0
3     V=np.array(V)
4     A=histo(V/q2,B) #on regarde l'histogramme quantifi avec q2
5     for i in range(0,len(bins)):
6         if np.abs(bins[i])>=2: ## on ne prend pas le pic central
7             a=A[i]
8             b=Pc(bins[i],q,q2,a2,b2)
9             somme=somme+np.abs(b-a)
10    return somme
```

On peut alors coder l'algorithme en simulant une deuxième quantification avec un unique q_2 pour toute l'image et donc l'utiliser pour obtenir la matrice de quantification:

```

3 def leQ(image, q2 = 3):
4     image2=np.array(Image.open(image))
5     lesq=np.arange(1,100)
6     Q=np.zeros((8,8))
7     lesBlock=np.array(dctblock(image2)) ##on fait la DCT
8     for i in range(0,8):
9         for j in range(0,8):
10             V=np.round(lesBlock[:,i,j])
11             a2,b2=estimationML(V)
12             lesdistance=np.array([sumPc(V,q2,q,a2,b2) for q in lesq])
13             q=lesq[np.argmin(lesdistance)]
14             Q[i,j]=q
15     return(Q)

```

Ou on peut aussi détecter la matrice de quantification Q_2 de l'entête et déterminer la première matrice Q_1 :

```

4 def Q1Q2(image): #donne la matrice de la seconde quantification et de la premiere
5     quantification
6     Q2=donner_Q(Image.open(image))
7     image2=np.array(Image.open(image))
8     lesB=np.array(dctblock(image2))
9     lesq=np.arange(2,100)
10    Q1=np.zeros((8,8))
11    for i in range(0,8):
12        for j in range(0,8):
13            V=np.round(lesB[:,i,j])
14            q2=Q2[i,j]
15            a2,b2=estimationML(V) #on estime les coefficient avec la m thode ML
16            lesdistance=np.array([sumPc(V,q2,q,a2,b2) for q in lesq])
17            q=lesq[np.argmin(lesdistance)]
18            Q1[i,j]=q
19    return(Q1)

```

- **Résultats :** On a testé le premier algorithme avec différents q_2 on obtient les meilleurs résultats pour $q_2 = 2$ ou 3 (pour s'assurer que $q_2 < q_1$ voici un résultat typique sur une images JPEG dont on connaissait les matrices de quantification:

$$Q_{\text{calculé}} = \begin{pmatrix} 1. & 9. & 8. & 13. & 19. & 32. & 43. & 49. \\ 10. & 10. & 11. & 15. & 21. & 46. & 49. & 46. \\ 11. & 10. & 13. & 19. & 32. & 46. & 54. & 46. \\ 11. & 14. & 18. & 23. & 40. & 70. & 64. & 99. \\ 14. & 18. & 30. & 44. & 53. & 86. & 81. & 1. \\ 19. & 28. & 43. & 50. & 64. & 82. & 3. & 2. \\ 40. & 52. & 61. & 70. & 80. & 1. & 3. & 3. \\ 56. & 76. & 7. & 7. & 3. & 2. & 1. & 4. \end{pmatrix} \quad Q_{\text{réel}} = \begin{pmatrix} 13. & 9. & 8. & 13. & 19. & 32. & 41. & 49. \\ 10. & 10. & 11. & 15. & 21. & 46. & 48. & 44. \\ 11. & 10. & 13. & 19. & 32. & 46. & 55. & 45. \\ 11. & 14. & 18. & 23. & 41. & 70. & 64. & 50. \\ 14. & 18. & 30. & 45. & 54. & 87. & 82. & 62. \\ 19. & 28. & 44. & 51. & 65. & 83. & 90. & 74. \\ 39. & 51. & 62. & 70. & 82. & 97. & 96. & 81. \\ 58. & 74. & 76. & 78. & 90. & 80. & 82. & 79. \end{pmatrix}$$

figure 10 : résultats de la méthode L_1 pour $q_2 = 3$

Cette méthode est plus rapide que la précédente cependant elle ne comporte pas de validation statistique il n'est donc pas possible de savoir si le coefficient trouvé est le bon ce qui est un inconvénient. Elle permet surtout de déterminer la matrice d'une première quantification pour une image doublement quantifiée dans le cas où $Q_2 < Q_1$ (c'est à dire $q_2 < q_1$ pour environ toute l'image en terme de pas de quantification):

$Q_{\text{calculé}}$	$Q_{\text{réel}}$	$Q_{\text{entête}}$
$\begin{pmatrix} 2. & 9. & 8. & 13. & 19. & 33. & 42. & 53. \\ 10. & 10. & 11. & 15. & 21. & 47. & 46. & 48. \\ 11. & 10. & 13. & 19. & 33. & 43. & 57. & 48. \\ 11. & 14. & 18. & 23. & 42. & 64. & 64. & 53. \\ 14. & 18. & 31. & 46. & 56. & 85. & 93. & 2. \\ 19. & 29. & 45. & 50. & 60. & 93. & 2. & 2. \\ 40. & 49. & 62. & 69. & 93. & 2. & 2. & 2. \\ 57. & 80. & 84. & 89. & 2. & 2. & 2. & 2. \end{pmatrix}$	$\begin{pmatrix} 13. & 9. & 8. & 13. & 19. & 32. & 41. & 49. \\ 10. & 10. & 11. & 15. & 21. & 46. & 48. & 44. \\ 11. & 10. & 13. & 19. & 32. & 46. & 55. & 45. \\ 11. & 14. & 18. & 23. & 41. & 70. & 64. & 50. \\ 14. & 18. & 30. & 45. & 54. & 87. & 82. & 62. \\ 19. & 28. & 44. & 51. & 65. & 83. & 90. & 74. \\ 39. & 51. & 62. & 70. & 82. & 97. & 96. & 81. \\ 58. & 74. & 76. & 78. & 90. & 80. & 82. & 79. \end{pmatrix}$	$\begin{pmatrix} 3. & 2. & 2. & 3. & 5. & 8. & 10. & 12. \\ 2. & 2. & 3. & 4. & 5. & 12. & 12. & 11. \\ 3. & 3. & 3. & 5. & 8. & 11. & 14. & 11. \\ 3. & 3. & 4. & 6. & 10. & 17. & 16. & 12. \\ 4. & 4. & 7. & 11. & 14. & 22. & 21. & 15. \\ 5. & 7. & 11. & 13. & 16. & 21. & 23. & 18. \\ 10. & 13. & 16. & 17. & 21. & 24. & 24. & 20. \\ 14. & 18. & 19. & 20. & 22. & 20. & 21. & 20. \end{pmatrix}$

Cet aspect de cette méthode est vraiment intéressant lors d'un envoi d'une image par le biais d'application comme Messenger, Whatsapp ... En effet l'image est souvent requantifiée par l'application et cela devient compliqué à analyser via d'autres méthodes.

5 Algorithme de détection de falsifications à partir de Q

5.1 Principe et algorithme

En entrée de l'algorithme on fournit une image (falsifiée ou non) et on suppose que l'on connaît une matrice de quantification Q qui correspond à la compression JPEG de l'image originelle (cette matrice peut être calculée en appliquant les méthodes précédentes à l'image falsifiée). Les coefficients DCT de l'image qui correspondent à l'image originelle devraient donc être quantifiés autour de valeurs de Q , en revanche dans la partie falsifiée il n'y a *a priori* pas de raison que ce soit le cas : ce n'est pas le cas si la partie falsifiée provient d'une image PNG ou d'une image JPEG avec une matrice de quantification différente (ou lors de la falsification si les grilles JPEG de l'image originelle et de la partie falsifiée n'ont pas été alignées). Si la partie falsifiée ne provient pas d'une image (si il y a eu gommage, floutage ...) il n'y a encore une fois aucune raison que les coefficients DCT soient quantifiés autour de Q .

Ainsi on va regarder sur chaque block 8x8 (U) de coefficients DCT de l'image, l'erreur :

$$E = \frac{1}{64} \sum_{1 \leq i, j \leq 8} 2 \left| \frac{U_{ij}}{Q_{ij}} - \left\lfloor \frac{U_{ij}}{Q_{ij}} \right\rfloor \right|.$$

Cette erreur pour un bloc DCT de l'image JPEG originelle vaut en moyenne $\mathbb{E}(E_{\text{JPEG}}) = \mathbb{E}(e_{\text{JPEG}})$ et comme on sait que $e_{\text{JPEG}} = |\varepsilon|$ et $\varepsilon \sim \mathcal{N}(0, \frac{1}{12})$, alors on a $\mathbb{E}(E_{\text{JPEG}}) = 0.230$.

De même pour une image PNG quelconque $\mathbb{E}(E_{\text{PNG}}) = \mathbb{E}(e_{\text{PNG}}) = 0.5$.

Si on transforme la matrice des erreurs (pour toute l'image falsifiée) en une image de couleur (les couleurs allant de la valeur 0.230 à 0.5), alors on observera des grandes erreurs (couleur jaune) là où l'image a été falsifiée et de faibles erreurs là où l'image originelle n'a pas été changée (couleur bleue). On obtient (l'image originelle est celle du chien) :

```

8 | for i in range(I//8):
9 |     for j in range(J//8):
10 |         a=dct2(V[i,j]) #V est la matrice des blocs 8x8 des coefficients DCT
11 |         erreurs=np.abs(a/Q-np.round(a/Q))
12 |         B[i,j]=np.mean(np.resize(erreurs,64)[1:])

```



figure 11 : Image falsifiée, sa carte des erreurs et l'échelle couleur de cette carte, le chien est compressé à 65% et le lion n'est pas compressé

En réalité la formule de E utilisée est : $E = \text{moyenne}_{i,j} \left(2 \left| \frac{U_{ij}}{Q_{ij}} - \left\lfloor \frac{U_{ij}}{Q_{ij}} \right\rfloor \right| \right)$ où les éléments considérés dans la moyenne sont ceux tels que $(i,j) \neq (1,1)$ pour enlever la valeur correspondant à la fréquence nulle et $\left\lfloor \frac{U_{ij}}{Q_{ij}} \right\rfloor \neq 0$ pour éviter le problème mentionné en figure 3 (problème qui réduit l'erreur E dans la partie falsifiée).

Le problème est qu'il va alors apparaître des blocks 8x8 sur lesquels E ne sera pas défini (la condition $\left\lfloor \frac{U_{ij}}{Q_{ij}} \right\rfloor \neq 0$ n'est jamais vérifiée), cela arrive particulièrement sur les zones de l'image de couleur uniforme (donc les coefficients DCT U_{ij} correspondant à des oscillations sont tous nuls) ou plus généralement des zones floues sur l'image, avec peu d'oscillations des couleurs. On introduit alors une nouvelle couleur dans notre carte des erreurs : le vert. Elle correspond à des zones où la nature de l'image fait que l'algorithme ne peut rien détecter et donc on ne peut décider si il y a falsification ou non.



figure 12 : Image falsifiée et sa nouvelle carte des erreurs, le chien est compressé à 65% et le lion n'est pas compressé

On remarque que des erreurs élevées (points jaunes) apparaissent sur l'image originelle et des erreurs faibles sur la partie falsifiée, cela se justifie simplement par le fait que $\mathbb{P}(E_{\text{JPEG}} \geq 0.5) \approx 0.0015\% \neq 0$ (calculé algorithmiquement en supposant que 10 coefficients par blocks 8x8 vérifient les 2 conditions imposées sur E_{JPEG}) et $\mathbb{P}(E_{\text{PNG}} \leq 0.230) \approx 0.11\% \neq 0$ (calculé avec la loi d'Irwin-Hall sous les mêmes hypothèses que précédemment). Ces probabilités sont tout de même suffisamment faibles pour justifier l'utilisation de cette échelle de carte des erreurs.

5.2 Ajout des méthodes de calcul de Q

Cependant le calcul de la matrice de quantification n'est pas exact et donc les erreurs dépendent aussi de ce calcul. Il est donc impossible d'obtenir une carte des erreurs aussi claire qu'en figure 12. Voyons comment adapter cet affichage des erreurs selon les méthodes de calcul de Q .

Les tests dans cette partie sont faits sur l'image suivante (image originale JPEG sur laquelle a été collée une partie d'une image JPEG en faisant attention à aligner les grilles de blocks 8x8) :



- **Méthode L_1 avec une quantification :**

Dans le cas où l'image originale n'a subi qu'une seule compression, on peut déterminer Q avec la méthode L_1 en faisant nous-même la deuxième compression. On a vu précédemment que cette méthode risque sur les hautes fréquences de donner des valeurs aberrantes de Q , ce qui peut poser problème dans l'algorithme d'affichage de la carte des erreurs. La méthode que nous utilisons est expérimentale (à cause de la nature de cette méthode L_1 , il est très difficile de faire une validation statistique de la justesse des coefficients dans Q) et consiste simplement à ne pas prendre en compte les hautes fréquences.

Après de nombreux tests, il semble que retirer la dernière ligne et la dernière colonne lors du calcul des erreurs est un bon compromis entre 'garder suffisamment de coefficients pour éviter que trop de cas indécis (cas des pixels verts) apparaissent' et 'supprimer les hautes fréquences pour lesquelles il y a de grandes chances que les erreurs calculées soient trop élevées à cause du calcul de Q '. A noter que ce problème ne concerne que la partie JPEG de l'image originelle ; la partie falsifiée n'est pas affectée par un choix imprécis de Q car ce Q ne correspondra pas (généralement) à une compression de cette partie falsifiée.

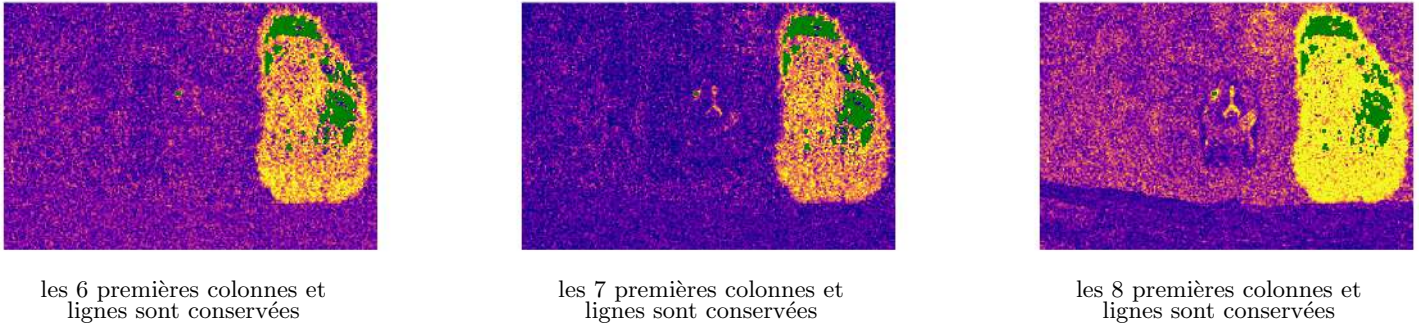


figure 13 : Cartes des erreurs en ne prenant en compte que certaines fréquences

- **Méthode avec fausses détections contrôlées (FDC):**

Cette méthode et plus particulièrement la validation statistique est très adaptée à l'affichage de la carte des erreurs car les coefficients dans Q différents de -1 sont très proches des coefficients originaux du Q_{JPEG} et donc les erreurs calculées sont quasiment les erreurs mentionnées en 5.1.

Il suffit alors de ne pas considérer les fréquences pour lesquelles le coefficient correspondant dans Q est -1 et on obtient le résultat suivant :

```

9  for i in range(I//8):
10     for j in range(J//8):
11         a=dct2(V[i,j]) #V est la matrice des blocs 8x8 des coefficients DCT
12         erreurs=np.abs(a/Q-np.round(a/Q))
13         moyenne_erreur_block88=0
14         compteur=0
15         for i in range(8):
16             for j in range(8):
17                 if np.round(a[i,j]/Q[i,j])!=0 and Q[i,j]!=-1 and i+j!=0:
18                     moyenne_erreur_block88 += erreurs[i,j]
19                     compteur+=1
20         moyenne_erreur_block88=moyenne_erreur_block88/compteur

```

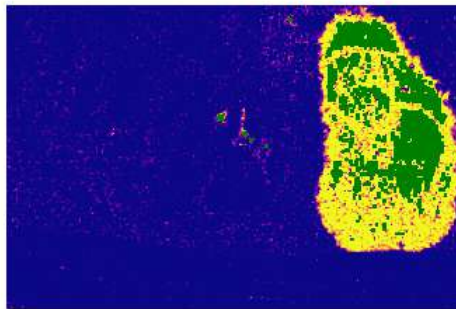


figure 14 : Cartes des erreurs pour la méthode avec fausses détections contrôlées

On remarque qu'il y a plus de vert qui apparaît, c'est dû au fait qu'on rajoute encore une condition sur les coefficients qui sont considérés dans le calcul de E .

- **Méthode L_1 avec double quantification :**

Le cas de la double quantification est un peu spécial : en effet les coefficients ont été re-quantifiés et on cherche ici à afficher l'erreur de ces coefficients re-quantifiés par rapport à la première quantification.

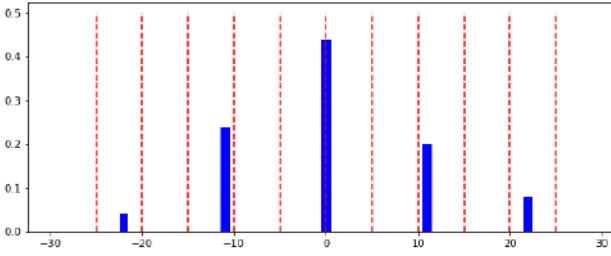


figure 15 : Histogramme coefficients Q1 C1 (bleu)
et les multiples de Q2 (rouge) pour q1=11 q2=5

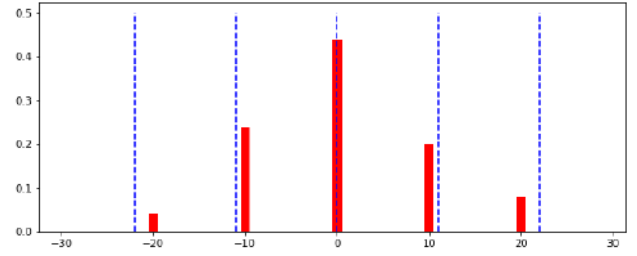


figure 16 : Histogramme coefficients Q2 C2 (rouge)
et les multiples de Q1 (bleu) pour q1=11 q2=5

L'erreur $\left(2 \left| \frac{U_{ij}}{Q_{ij}} - \left\lfloor \frac{U_{ij}}{Q_{ij}} \right\rfloor \right| \right)$ pour un coefficient $C_2 Q_2$ après deuxième quantification est $2 \cdot \min_{k \in \mathbb{Z}} \left(\left| \frac{c_2 q_2 - k q_1}{q_1} \right| \right)$ à un $\frac{\varepsilon}{q_1}$ près. Bien que cette erreur reste plus faible en moyenne que l'erreur de la partie falsifiée qui n'a été compressée qu'une fois avec Q_2 , distinguer les deux parties de l'image est moins simple que dans le cas simple compression.

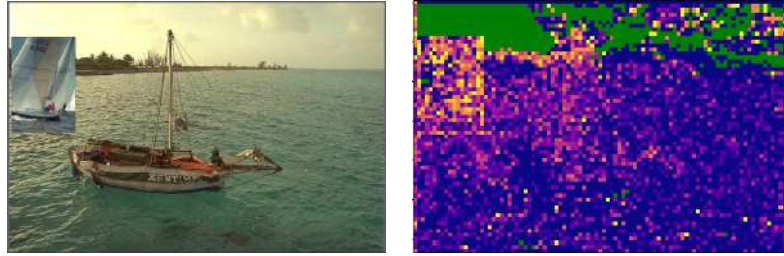
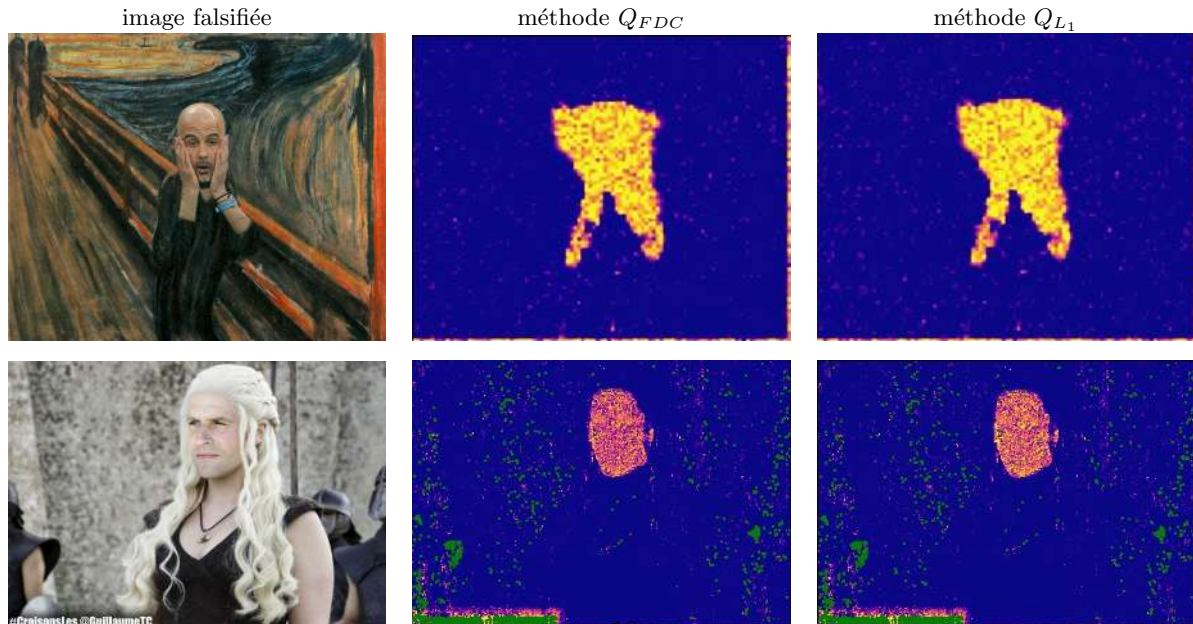


figure 17 : Carte des erreurs pour une image falsifiée qui a subi une deuxième compression Q_2

Globalement ça reste la méthode la plus efficace lors d'une double compression (les autres méthodes calculent très mal Q_1) dans les cas où la deuxième compression est plus faible que la première ($Q_2 \leq Q_1$).

Voici plusieurs tests de ces algorithmes sur plusieurs images 'bien' falsifiées (Q_{FDC} est la méthode avec fausses détections contrôlées, Q_{L_1} la méthode L_1 avec simple compression et L_1 avec double compression n'est pas testé ici car il correspond à des cas plus particuliers) :



5.3 Limites

Malheureusement bien que cette méthode (un calcul de Q puis l’affichage d’une carte des erreurs) permet de détecter de nombreux cas posant problèmes à d’autres méthodes, elle n’est pas parfaite.

Elle est dépendante du calcul de Q qui est réalisé sur toute l’image, ainsi si la falsification prend trop de place dans l’image (d’après nos test, tant que la falsification prends moins de ??? de l’image ce problème n’est pas présent) alors le calcul de Q sera faussé (car alors ??? de l’image n’est pas quantifié sur ces coefficients).

De la même manière la précision lors du calcul de Q est dépendante de la taille de l’image ; il est impossible de calculer correctement les coefficients q pour de hautes fréquences (ou q élevé) pour une petite image. Cela a peu d’effet sur le résultat global (car les erreurs E sont principalement influencées par les fréquences correspondant à un q faible) mais cela influe quand-même sur la facilité de distinction entre la partie falsifiée et la partie originelle.

Comme on a pu le voir dans la partie précédente si la première quantification est faible et fait apparaître des 1 dans Q , il y a de grandes chances que la plupart de la partie falsifiée soit affichée en vert dans la carte des erreurs. Comme le vert correspond à un résultat *a priori* indécis, dans ce cas il y a nécessité que l’utilisateur de l’algorithme vérifie que cette partie verte corresponde à une zone floue ou uniforme. Si ce n’est pas le cas et que du vert apparaît c’est sûrement dû à une falsification.








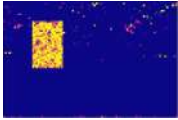
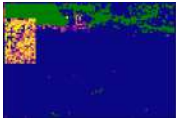
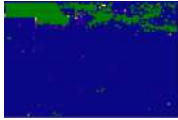
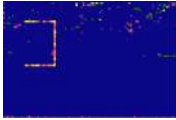
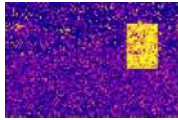

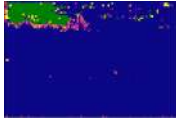
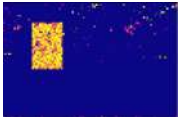
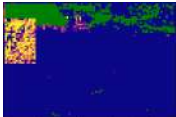
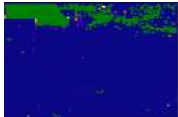
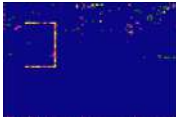
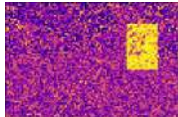
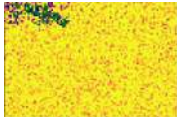
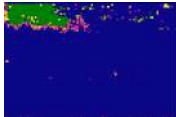
De manière opposée si une partie de l’image originelle est de couleur uniforme ou floue, elle risque d’apparaître verte, cela nécessite encore une fois un peu de réflexion de la part de l’utilisateur.

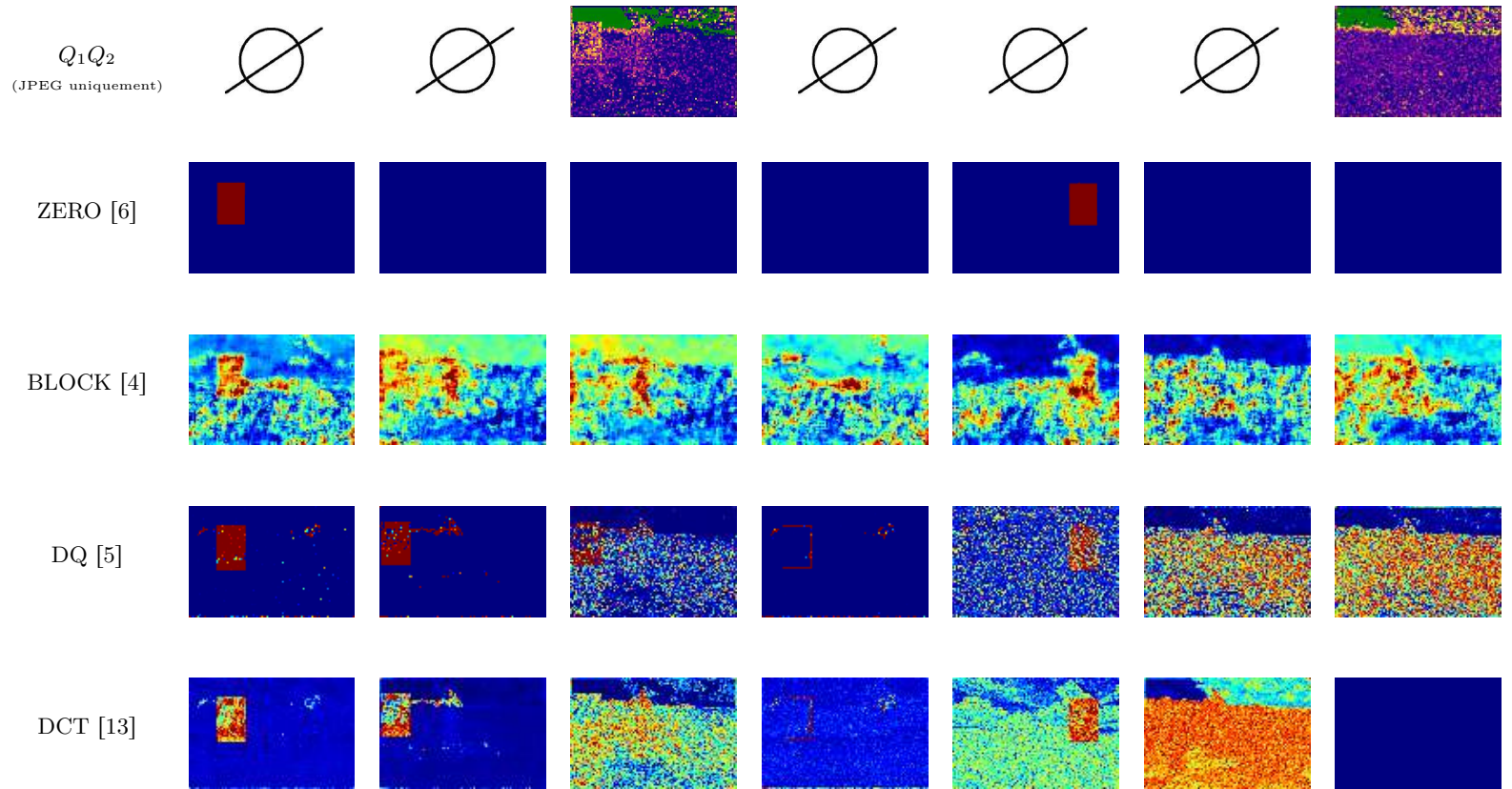
Finalement le dernier problème notable est la durée d’exécution de cet algorithme total. De par la nature de la méthode qui est complexe, c’est normal qu’il prenne plus de temps à exécuter que les algorithmes mentionnés en 3.1 mais on sait que ces algorithmes pourraient être beaucoup plus rapides que la manière dont on les a codés (par exemple la méthode en 4.1 a déjà été codée par d’autres chercheurs et nos algorithmes sont beaucoup plus lents que ceux qu’ils ont créés). Cela vient du fait que l’on a travaillé sur Python quand du code C ou Matlab serait plus rapide et du fait que nos connaissances en optimisation d’algorithmes étaient nulles au début de ce stage (et bien que des tentatives d’amélioration de rapidité ont été faites, elle n’ont pas été suffisamment concluantes). Finalement l’algorithme complet (calcul de Q puis affichage de carte) prend plusieurs minutes pour une image de taille classique quand les autres algorithmes de détection de falsification s’exécutent en une dizaine de secondes sur la même image.

Remarque : le calcul de E fait que E est maximum pour $q_1 = 1$ (E dépend de $\frac{\varepsilon}{q_1}$), c’est pour cela qu’avec le calcul de Q utilisant la méthode L_1 la partie originelle de l’image présente des grandes erreurs (couleur violette) en comparaison avec la méthode avec fausses détections contrôlées. En effet dans cette dernière méthode la vérification $NFA \leq 1$ fait que $q = 1$ ne sera jamais donné comme résultat. Ce désavantage est contrebalancé par l’apparition de plus de vert dans cette dernière méthode. Bien sûr cette remarque est non-pertinente pour des compressions Q_1 ne comportant pas de 1.

6 Comparaison des méthodes globales

Voici plusieurs tests des algorithmes mentionnés en 3.1 ainsi que notre algorithme sur plusieurs images falsifiées (Q_{FDC} est la méthode avec fausses détections contrôlées, Q1Q2 la méthode L_1 avec double compression et Q_{L_1} la méthode L_1 avec simple compression) :

image							
(Q_1, Q_1')	(70, 80)	(40, 80)	(40, 80)	(80, 80)	(90, 80)	PNG	(80, /)
Q_2	/	/	70	/	/	/	/
alignement grilles	non	oui	oui	oui	non	/	/
Q_{FDC}							
Q_{L_1}							



Dans ces tests, les falsifications utilisées sont des insertions (car celles ci peuvent simuler toutes les falsification citées en introduction), Q_1 (respectivement Q'_1 et Q_2) correspond au taux de compression de l'image originelle (respectivement de la partie falsifiée et de l'image l'image après falsification). L'alignement des lignes considéré ici est celui entre l'image originelle et la partie falsifiée.

Pour la quatrième image, l'image falsifiée est compressée avec le même Q que l'image originelle et les grilles JPEG sont alignées, on ne détecte donc que les bords de l'image falsifiée (sur les bords, il y a altération des coefficients DCT car sur un même block 8x8 des pixels des deux images sont présent). Les deux dernières images ne sont pas falsifiées l'une est PNG et l'autre JPEG.

Remarque : les programmes créés ont été adaptés pour aussi pouvoir marcher sur des images en nuances de gris (une image de gris suit les mêmes propriétés que la luminance d'une image RGB).








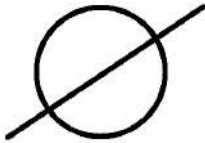
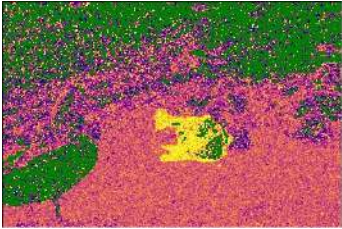
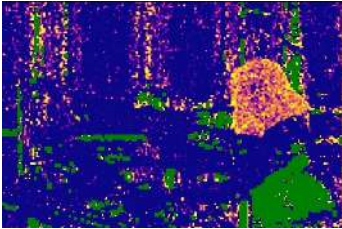
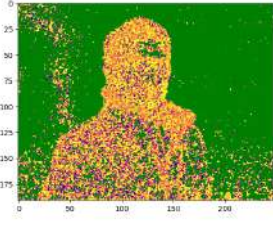

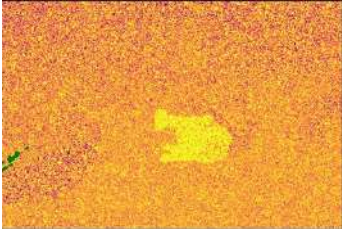
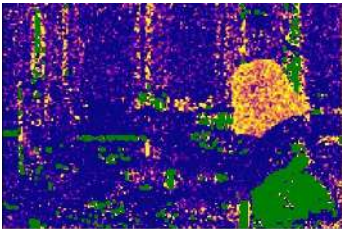
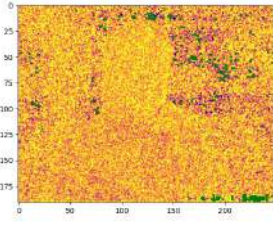
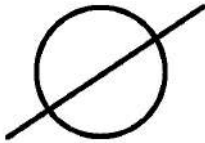
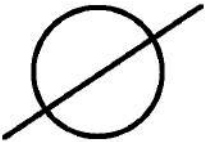
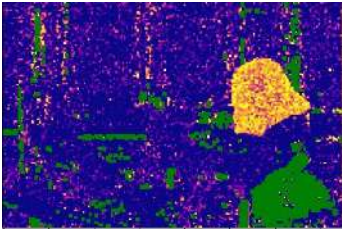
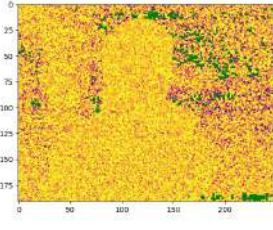


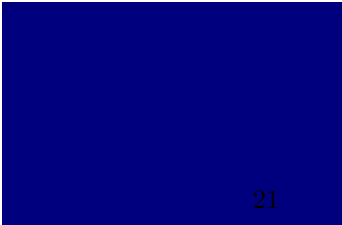

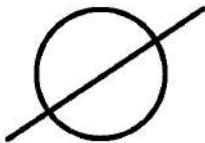
7 Conclusion

Finalement, on constate que les algorithmes que nous avons créé permette belle et bien de détecter les falsifications que nous voulions détecter. Chacune des 3 méthodes donne de meilleurs résultats dans différents cas : $Q_1 Q_2$ quand l'image a été re-compressée après falsification, Q_{FDC} quand l'image originelle n'est pas compressée faiblement (si il y des 1 dans Q_1 , des zones vertes supplémentaires apparaissent) et Q_{L_1} est utile dans les cas ou les deux autres méthodes fonctionnent moins bien. Un problème étant qu'il est très difficile *a priori* de déterminer dans lequel de ces cas on se trouve, il faudrait donc, quand on veut tester une image, appliquer les 3 méthodes et comparer les résultats. De plus ces trois méthodes ne donnent pas de faux positifs sur des images non falsifiées.

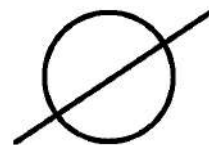
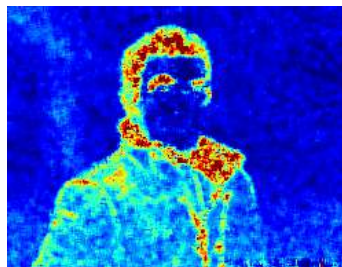
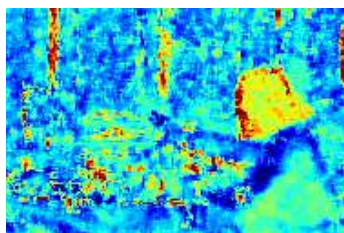
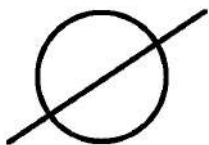
Les algorithmes que nous avons créé permettent bien de contrebalancer les problèmes des méthodes qui n'utilisent que la grille des blocks 8x8, et propose une bonne alternative aux méthodes déjà existantes utilisant la répartition des coefficients DCT. Globalement, le plus gros problème de ces méthodes est la duré d'exécution des programmes qui est bien en dessous de ce qui est proposé par les méthodes déjà existantes.

8 Annexes

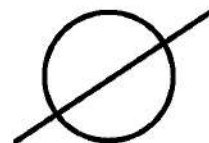
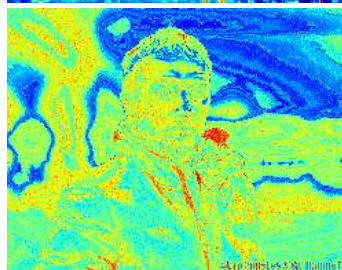
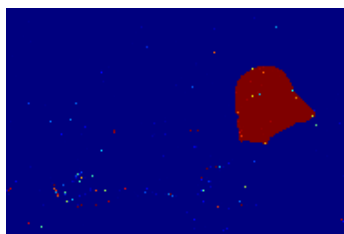
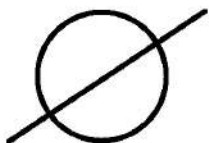
8.1 Quelques cas réel :

image originelle				
image falsifiée				
(Q_1, Q'_1) Q_2 alignement grilles	$Q_1 = Q'_1$? non	$(40, 80)$ 96 oui	$(40, 80)$ 95 oui	$(,)$? oui et non (plusieurs falsifications)
Q_{FCD}				
Q_{L_1}				
$Q_1 Q_2$				
ZERO				

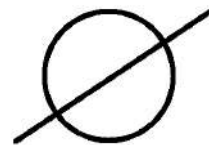
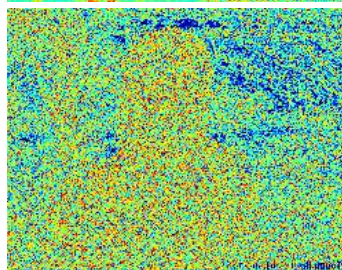
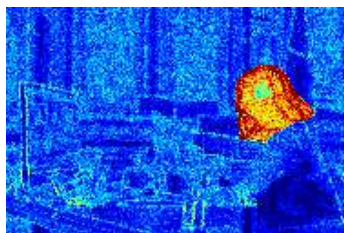
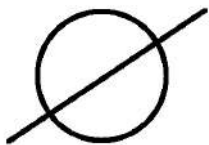
BLOCK



DQ



DCT



La première image présente une falsification qui est un copier-coller interne à l'image, la deuxième et la troisième ont des falsifications classiques et la dernière présente deux copier-coller internes à l'image, un avec les grilles alignés (celui de gauche) et l'autre non.

8.2 Code Python du programme complet :

```
8 from scipy.stats import norm as normal
9 from scipy.special import modstruve
10 import numpy as np
11 from scipy.fftpack import dct, idct
12 import matplotlib.pyplot as plt
13 import matplotlib as mpl
14 from skimage.util import view_as_blocks
15 from PIL import Image
16 from scipy.special import kv
17 from math import gamma
18 from scipy.stats import moment
19 from scipy.optimize import minimize
20 from math import factorial
21 from math import log
22 #import sys
23
24 #-----Blocks-----
25
26 # implement 2D DCT
27 def dct2(a):
28     return dct(dct(a.T, norm='ortho').T, norm='ortho')
29
30 # implement 2D IDCT
31 def idct2(a):
32     return idct(idct(a.T, norm='ortho').T, norm='ortho')
33
```

```

34 def dctblock(lumi):
35     X,Y=lumi.shape
36     lesblock=[]
37     for i in range(0,X,8):
38         for j in range(0,Y,8):
39             if j+8<Y and i+8<X:
40                 lesblock.append(dct2(lumi[i:i+8,j:j+8] ))
41     return(lesblock)
42
43 #-----alpha beta -----
44 def f(x,alpha,beta):
45     a=kv(alpha-1/2,np.abs(x)*np.sqrt(2/beta))
46     if alpha<170 :#si alpha > 170, python ne peut pas calculer Gamma(alpha)
47         b=np.sqrt(2/np.pi)*((np.abs(x)*np.sqrt(beta/2))**(alpha-1/2))/(gamma(alpha)*beta**
48             alpha)
49         return(a*b)
50     else:
51         return(np.exp(np.log(a) +
52             log(np.sqrt(2/np.pi))+
53             np.log(np.abs(x)*np.sqrt(beta/2))*(alpha-1/2)
54             -(log(factorial(np.int(alpha)))+log(beta))*alpha))
55
56 def sumf(alpha,beta,Vk):
57     A=-np.sum(np.log(f(Vk,alpha,beta)))
58     return A
59
60 def estimationML(Vk):
61     M4=moment(Vk,moment=4)
62     M2=moment(Vk,moment=2)
63     alphaM=3/(M4/(M2**2)-3)
64     betaM=M2/alphaM
65     x0=np.array([alphaM,betaM])
66     def minimf(x):
67         return sumf(x[0],x[1],Vk)
68     res=minimize(minimf,x0,method='Nelder-Mead')
69     [alphaML,betaML]=res.x
70
71     return alphaML,betaML
72
73 #-----Calculs Probabilites PC2 -----
74 def P(V,alphaP,betaP,q):
75     X=np.round(V)
76     def g(x):
77         return q*(x+1/2)*np.sqrt(2/betaP)
78     def G(x):
79         gx=g(x)
80         return 1/2*gx*(kv(alphaP-1/2,gx)*modstruve(alphaP-3/2,gx)+kv(alphaP-3/2,gx)*modstruve(
81             alphaP-1/2,gx))
82     if X==0:
83         return 2*G(0)
84     return G(np.abs(X))-G(np.abs(X)-1)
85
86 p1=normal.cdf(-1,loc=0,scale=1/np.sqrt(12))-normal.cdf(-3/2,loc=0,scale=np.sqrt(1/12))
87 p2=normal.cdf(0,loc=0,scale=1/np.sqrt(12))-normal.cdf(-1,loc=0,scale=np.sqrt(1/12))
88 p3=normal.cdf(1,loc=0,scale=1/np.sqrt(12))-normal.cdf(0,loc=0,scale=np.sqrt(1/12))
89 p4=normal.cdf(3/2,loc=0,scale=np.sqrt(1/12))-normal.cdf(1,loc=0,scale=np.sqrt(1/12))
90 p5=normal.cdf(-1/2,loc=0,scale=np.sqrt(1/12))-normal.cdf(-3/2,loc=0,scale=np.sqrt(1/12))
91 p6=normal.cdf(1/2,loc=0,scale=np.sqrt(1/12))-normal.cdf(-1/2,loc=0,scale=np.sqrt(1/12))
92 p7=normal.cdf(3/2,loc=0,scale=np.sqrt(1/12))-normal.cdf(1/2,loc=0,scale=np.sqrt(1/12))

```

```

93
94 def pt(k,q2,c2):
95     somme=0
96     if q2%2==0:
97         m=q2/2
98         somme=0
99         if c2*q2-m+2<=k<=c2*q2+m+1:
100             somme=somme+p1
101         if c2*q2-m+1<=k<=c2*q2+m:
102             somme=somme+p2
103         if c2*q2-m<=k<=c2*q2+m-1:
104             somme=somme+p3
105         if c2*q2-m-1<=k<=c2*q2+m-2:
106             somme=somme+p4
107     else:
108         m=(q2-1)/2
109         if c2*q2-m+1<=k<=c2*q2+m+1:
110             somme=somme+p5
111         if c2*q2-m<=k<=c2*q2+m:
112             somme=somme+p6
113         if c2*q2-m-1<=k<=c2*q2+m-1:
114             somme=somme+p7
115     return(somme)
116
117 def Pc(c2,q,q2,a2,b2):
118     somme=0
119     q1=q
120
121     a=np.int(c2*q2-np.int(q2/2)-1)
122     b=np.int(c2*q2+np.int(q2/2)+1+1)
123
124
125     for k in range(a,b):
126         if k%q1==0:
127
128             l=k/q1
129             somme=somme+pt(k,q2,c2)*P(l,a2,b2,q1)
130     return(somme)
131 def histo(V,bins):
132     A=[]
133
134     for b in bins:
135         a=(np.count_nonzero(V<=b)-np.count_nonzero(V<=b-1))/(len(V))
136         A.append(a)
137     return A
138
139
140
141 bins=np.arange(100)-50
142 B=bins+0.5
143
144
145 def sumPc(V,q2,q,a2,b2):#somme des erreurs L1 entre histogramme et probabilite
146     somme=0
147     V=np.array(V)
148     A=histo(V/q2,B)
149     for i in range(0,len(bins)):
150         if np.abs(bins[i])>=2:
151             a=A[i]
152             b=Pc(bins[i],q,q2,a2,b2)
153             somme=somme+np.abs(b-a)

```



```

154     return somme
155
156 #-----programmes calcul Q-----
157 def leQ(image,lumi,q2 = 2):
158     lesq=np.arange(1,100)
159     Q=np.zeros((8,8))
160     lesBlock=np.array(dctblock(lumi))
161     for i in range(0,8):
162         for j in range(0,8):
163             V=np.round(lesBlock[:,i,j])
164             a2,b2=estimationML(V)
165             lesdistance=np.array([sumPc(V,q2,q,a2,b2) for q in lesq])
166             q=lesq[np.argmin(lesdistance)]
167             Q[i,j]=q
168
169     return(Q)
170
171
172
173
174 def donner_Q(im):#Obtenir le vrai Q (a partir d une image JPEG)
175     R=im.quantization[0]
176     Q=np.zeros((8,8))
177     for k in range(8):
178         if k%2==1:
179             for m in range(k+1):
180                 Q[m][k-m]=R[int((k)*(k+1)/2+m)]
181         else:
182             for m in range(k+1):
183                 Q[k-m][m]=R[int((k)*(k+1)/2+m)]
184     for k in range(7):
185         if k%2==1:
186             for m in range(7-k):
187                 Q[k+1+m][7-m]=R[36+int(k*(14-k+1)/2+m)]
188         else:
189             for m in range(7-k):
190                 Q[7-m][k+1+m]=R[36+int(k*(14-k+1)/2+m)]
191     return(Q)
192
193
194 def Q1Q2(image,lumi): #donne la matrice de la seconde quantification et de la premiere
    quantification
195     Q2=donner_Q(Image.open(image))
196     lesB=np.array(dctblock(lumi))
197     lesq=np.arange(1,100)
198     Q1=np.zeros((8,8))
199     for i in range(0,8):
200         for j in range(0,8):
201             V=np.round(lesB[:,i,j])
202             q2=Q2[i,j]
203             a2,b2=estimationML(V/q2)
204             lesdistance=np.array([sumPc(V,q2,q,a2,b2) for q in lesq])
205             q=lesq[np.argmin(lesdistance)]
206             Q1[i,j]=q
207     return(Q1)
208
209 def Qtable(image,lumi):#algorithme de Tina (pas optimis )
210     X,Y=lumi.shape
211     lesblock=dctblock(lumi)
212     NFA=10*np.ones((8,8))
213     Q=-1*np.ones((8,8))

```

```

214     for i in range(0,8):
215         for j in range(0,8):
216             if i!=0 or j!=0:
217                 for q in range(1,256):
218                     s=0
219                     n=0
220                     for block in lesblock:
221                         v=block[i,j]
222                         V=round(v/q)
223                         if V!=0:
224                             e=2*np.abs((v/q)-V)
225                             s=s+e
226                             n=n+1
227                     if n!=0:
228                         nfa=np.log10(63*255)+n*np.log10(s)-n*np.log10(n)+n*np.log10(np.exp(1))-(1/2)*np.
                             log10(2*n*np.pi)
229                         if nfa<NFA[i,j] and nfa<=0:
230                             Q[i,j]=q
231                             NFA[i,j]=nfa
232     return Q
233
234 def lesmatrices(image,q2): #donne les 3 matrices entete, tina et nous (on a besoin de q2)
    prends 3
235     Q2=donner_Q(Image.open(image))
236     print('Qentete=')
237     print(Q2)
238     image2=np.array(Image.open(image)).astype(np.float)
239     Qarticle=leQ(image2,q2)
240     Qtina=Qtable(image2)
241     print('Qarticle=')
242     print(Qarticle)
243     print('Qtina=')
244     print(Qtina)
245     return(True)
246
247 #-----programme total-----
248
249
250
251 def erreur(image,func): #ecrire "image.jpg" ou "image.png" suivant avec image le nom de l
    image
252     # func prend les valeur Q1Q2, leQ, Qtable suivant la methode choisit
253      #(leQ est la methode QL1 et Qtable celle avec fausses detections controlees)
254     Table=np.array(Image.open(image))
255     if np.shape(Table)[-1]==3 or np.shape(Table)[-1]==4:
256         lumi=0.299*Table[:, :, 0]+0.587*Table[:, :, 1]+0.114*Table[:, :, 2]
257     else:
258         lumi=Table
259     n,m=np.shape(lumi)
260     Jpeg=255*np.zeros((((n-1)//8+1)*8,((m-1)//8+1)*8))
261     Jpeg[:, :m]=lumi
262     Jpegblock=view_as_blocks(Jpeg,(8,8))
263     I,J=np.shape(Jpeg)
264     erreurmap=np.ones((I//8,J//8))
265     Q1=func(image,lumi)
266     for i in range(I//8):
267         for j in range(J//8):
268             DCTblock=dct2(Jpegblock[i,j])
269             erreurs=np.zeros((8,8))
270             erreurs=np.abs(DCTblock/Q1-np.round(DCTblock/Q1))[0:6,0:6]
271             erreursD=np.delete(np.resize(erreurs,64)[1:], np.where(1-(1-np.array(np.abs(np.

```

```

272         resize(DCTblock/Q1,64))
273         [1:]<0.5).astype(int))*(1-(np.array(np.resize(Q1,64))
274         [1:]==-1).astype(int))))
275     if len(erreursD)>0:
276         erreurmap[i,j]=np.mean(erreursD)
277     else:
278         erreurmap[i,j]=-1
279
280     B=np.ma.masked_where(erreurmap==-1,erreurmap)
281     cmap = mpl.cm.plasma
282     cmap.set_bad(color='green')
283     plt.imshow(B, cmap=cmap, vmin=0.1, vmax=0.25) #0==noir
284     plt.savefig('erreur.jpg')
285     return(Q1)
286
287 '''
288 if __name__ == "__main__":
289     image, fonction=sys.argv[1], sys.argv[2]
290     if fonction=='QDKL':
291         func=leQ
292     if fonction=='Qtina':
293         func=Qtable
294     if fonction=='Q1Q2':
295         func=Q1Q2
296     Q1,Qvrai=erreur(image,func)
297     print('Qestime='+str(Q1))
298     print('Qvrai='+str(Qvrai))
299 '''

```

Pour exécuter ce programme il faut lancer erreur('nom image', fonction) où le nom de l'image est juste le titre de l'image PNG/JPG... et fonction $\in \{Q1Q2, leQ, Qtable\}$.

References

- [1] Hany Farid. Exposing digital forgeries from JPEG ghosts. *IEEE transactions on information forensics and security*, 4(1):154–160, 2009.
- [2] G.Quin F.Luo, J.Huang. “*JPEG error analysis and its application to digital image forensics*”. 2010.
- [3] R.Cogranne F.Retraint T.H.Thai. “*Statistical model of natural images*”. 2012.
- [4] Weihai Li, Yuan Yuan, and Nenghai Yu. Passive detection of doctored JPEG image via block artifact grid extraction. *Signal Processing*, 89(9):1821–1829, 2009.
- [5] Zhouchen Lin, Junfeng He, Xiaou Tang, and Chi-Keung Tang. Fast, automatic and fine-grained tampered jpeg image detection via dct coefficient analysis. *Pattern Recognition*, 42(11):2492–2501, 2009.
- [6] T. Nikoukhah, J. Anger, T. Ehret, M. Colom, J.M. Morel, and R. Grompone von Gioi. JPEG grid detection based on the number of dct zeros and its application to automatic and localized forgery detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2019.
- [7] Tina Nikoukhah, Miguel Colom, Jean-Michel Morel, and Rafael Grompone von Gioi. Détection de grille JPEG par compression simulée. *preprint*, 2019.
- [8] Tina Nikoukhah, Miguel Colom, Jean-Michel Morel, and Rafael Grompone von Gioi. Local JPEG Grid Detector via Blocking Artifacts, a Forgery Detection Tool. *Image Processing On Line*, 10:24–42, 2020. <https://doi.org/10.5201/ipol.2020.283>.
- [9] Thanh Hai Thai and Rémi Cogranne. Estimation of primary quantization steps in double-compressed jpeg images using a statistical model of discrete cosine transform. *IEEE Access*, 7:76203–76216, 2019.
- [10] Thanh Hai Thai, Rémi Cogranne, Florent Retraint, et al. Jpeg quantization step estimation and its applications to digital image forensics. *IEEE Transactions on Information Forensics and Security*, 12(1):123–133, 2016.
- [11] Jean-Michel Morel Rafael Grompone von Gioi Tina Nikoukhah, Miguel Colom. “*Jpeg quantization table estimation with controlled false detections*”. preprint.

- [12] Shuiming Ye, Qibin Sun, and Ee-Chien Chang. Detecting digital image forgeries by measuring inconsistencies of blocking artifact. In *2007 IEEE International Conference on Multimedia and Expo*, pages 12–15. Ieee, 2007.
- [13] Shuiming Ye, Qibin Sun, and Ee-Chien Chang. Detecting digital image forgeries by measuring inconsistencies of blocking artifact. In *2007 IEEE International Conference on Multimedia and Expo*, pages 12–15. IEEE, 2007.