Optimisation, traitement d'image et éclipse de Soleil

Partie I

Introduction

Le 20 mars 2015 a eu lieu en France une éclipse partielle de Soleil qu'il était particulièrement agréable d'observer [1] dans l'Est de la France et plus particulièrement en Alsace. De nombreuses photos ont été prises à cette occasion et dans l'idée de faire un montage, un petit traitement numérique s'impose. Nous disposons en fait de deux jeux de photos distincts :

1. Celles stockées dans le répertoire `telescope/` sont des prises de vue effectuées à Rastatt en Allemagne [2] à partir d'un télescope protégé par un filtre « spécial Soleil » et a donc donné des photos où le Soleil est un cercle quasi-parfait de même que l'empreinte de la Lune. Le but du traitement ultérieur va être de recentrer les photos pour qu'une animation sur les trois photos disponibles puisse montrer l'avancement de la Lune et que l'on puisse facilement calculer le pourcentage d'obscurcissement lors de chaque prise de vue.
2. Au contraire, celles du dossier `kleber/` ont été prises dans la cours du lycée Kléber par une élève [3] avec un téléphone portable lors de la projection de l'image du Soleil sur un mur à l'aide d'une lunette de fortune fabriquée à partir d'un banc d'optique, d'un miroir et de deux lentilles convergentes. Là, le but premier du traitement sera de récupérer une image correcte du disque solaire en l'isolant du fond orange sur lequel il était projeté (et pour lequel le contraste n'est pas optimal) pour finalement ajuster une ellipse et essayer de compenser la déformation due à la projection par une transformation géométrique.

Figure 1 – Images de chaque sous-ensemble disponible : à gauche le répertoire `telescope/`, à droite `kleber/`.

1. NON, les jours d'éclipse ne sont pas plus dangereux que les jours normaux : observer directement le Soleil crâme la rétine tout aussi efficacement un jour normal...
2. Aimablement fournie par Mme Barbier.
3. Laetitia De Nadaï de PCSI1.

Partie II

K-means algorithm

II.1 Principe

Pour sélectionner sur la photo les zones appartenant au Soleil de celles appartenant au fond de l'image, on va utiliser un algorithme en « apprentissage non supervisé » dit de « partitionnement en k-moyennes ». L'idée est de partitionner des points dans un certain espace de données (ici les couleurs des pixels en RGB) en plusieurs groupes de caractéristiques voisines. Prenons l'exemple d'une taverne de la Terre du Milieu [4]. Considérons qu'elle ne puisse contenir que trois races :

— des elfes, très grands mais de largeurs d'épaules plutôt étroites ;

— des humains, un peu moins grands mais larges d'épaules ;

— et des nains, « petits mais costaud [5] ».

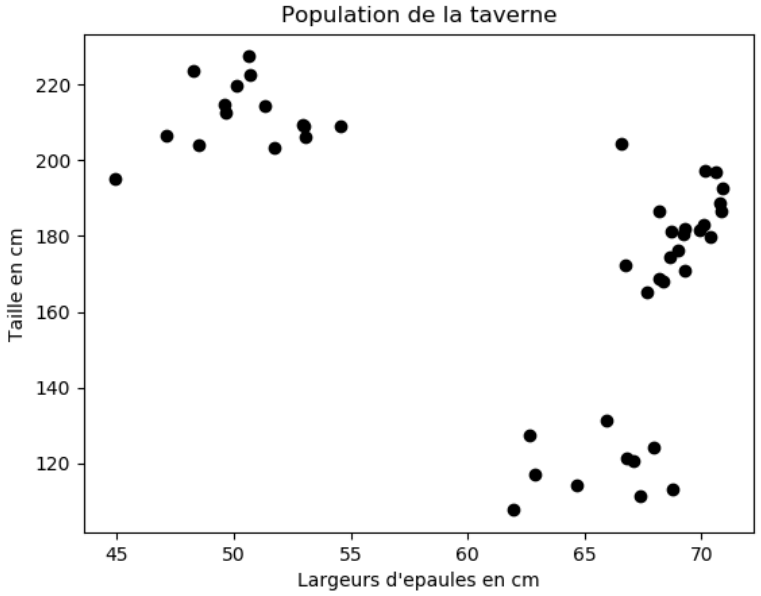

Figure 2 – Statistiques sur une population d'une taverne de la Terre du Milieu

Si l'on représente pour chaque personne du bar la taille en fonction de la largeur d'épaule, il est quasi-certain que l'on puisse identifier « à l'œil » la race de la personne concernée comme c'est le cas sur le graphique de la figure 2. L'idée de l'algorithme des k-moyennes est alors de se donner k (ici trois) points de départ de ce que l'on va nommer des « centroïdes » et itérer une même démarche en espérant que cela partitionne correctement notre groupe. Plus exactement, on itère toujours les deux mêmes étapes :

— En premier lieu rassembler chaque point dans le camp du centroïde qui lui est le plus proche dans l'espace de données correspondant. On peut pour ce faire choisir n'importe quelle norme, comme par exemple la norme euclidienne usuelle. Plus formellement, si le i^{e} point de donnée a pour coordonnées $(x_1^{(i)}, x_2^{(i)})$ dans notre espace (Epaules,Taille) et que le j^{e} centroïde a pour coordonnées $(\mu_1^{(j)}, \mu_2^{(j)})$ dans ce même espace, alors la classe d'appartenance $c^{(i)}$ de ce point i vaut le numéro j (entre 0 et $k-1$ pour Python) du centroïde qui permet de minimiser la quantité $\sqrt{\left(x_1^{(i)} - \mu_1^{(j)}\right)^2 + \left(x_2^{(i)} - \mu_2^{(j)}\right)^2}$.

4. Voir l'œuvremajeure de J. R. R. Tolkien.
5. Comme les petits pimousses

— Une fois les points distribués sur chaque centroïde, on calcule la position moyenne de chaque ensemble et on considère ces positions moyennes comme les nouveaux centroïdes pour l'itération suivante.

Au bout d'un certain nombre d'itérations, les groupes ne devraient plus varier et on stoppe l'algorithme [6]. La figure 3 présente l'évolution de la position des centroïdes au cours de l'algorithme ainsi que le partitionnement finalement trouvé.

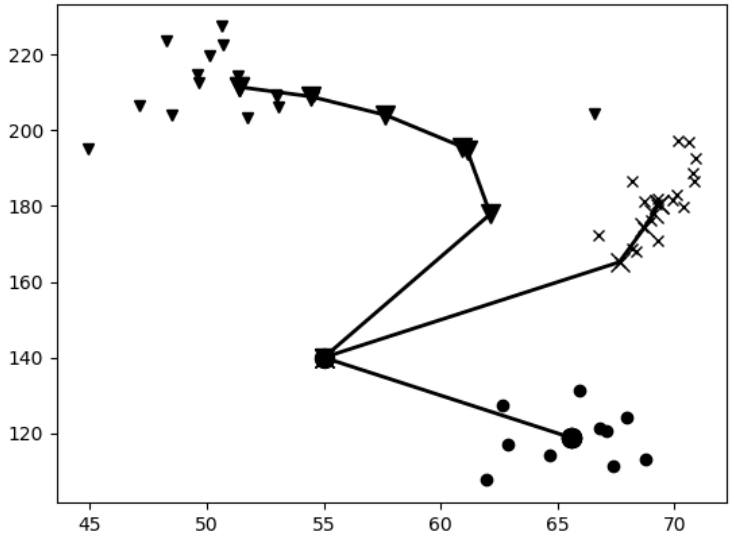

Figure 3 – Suivi de l'avancée de l'algorithme

On remarque sur la figure 3 que l'algorithme fait (presque) ce que l'on veut. Même en partant d'un même point de départ, les trois centroïdes se différencient pour arriver aux alentours du bon résultat même s'il est un humain qui est bizarrement classifié parmi les elfes [7]...

Néanmoins, on n'obtient pas toujours ce que l'on veut comme le montre la figure 4 où l'on a démarré dans une configuration qui détecte le groupe des grandes personnes (humains et elfes) ainsi que deux sous-groupes parmi les nains (les grands nains et les petits nains) du fait d'un mauvais choix des conditions de départ [8]. Il faut donc en général se méfier du résultat et il est conseillé de démarrer l'algorithme avec plusieurs conditions initiales choisies aléatoirement pour les centroïdes pour voir vers quelles positions cela converge « le plus souvent ». Mais pour le problème qui va nous intéresser, cela devrait fonctionner plutôt bien.

6. En pratique, mieux vaut commencer en faisant un nombre fini (par exemple 10) d'itérations pour vérifier que cela converge bien avant de se lancer dans une boucle `while`

7. L'explication est en fait assez claire : les tailles et largeurs d'épaules n'ont pas des domaines de variations de même amplitude, la norme choisie a donc tendance à dilater la zone d'influence suivant l'échelle horizontale, plus petite que l'échelle verticale. Pour utiliser l'algorithme, il faut normalement toujours normaliser les données de sorte à soustraire leur moyenne et diviser par leur écart-type, mais le problème ne se posera pas pour l'analyse d'image car les trois régimes de couleurs auront les mêmes intervalles de variation (entre 0 et 255), contrairement aux données choisies ici.

8. Et en profitant aussi de l'absence de normalisation comme discuté dans la note précédente.

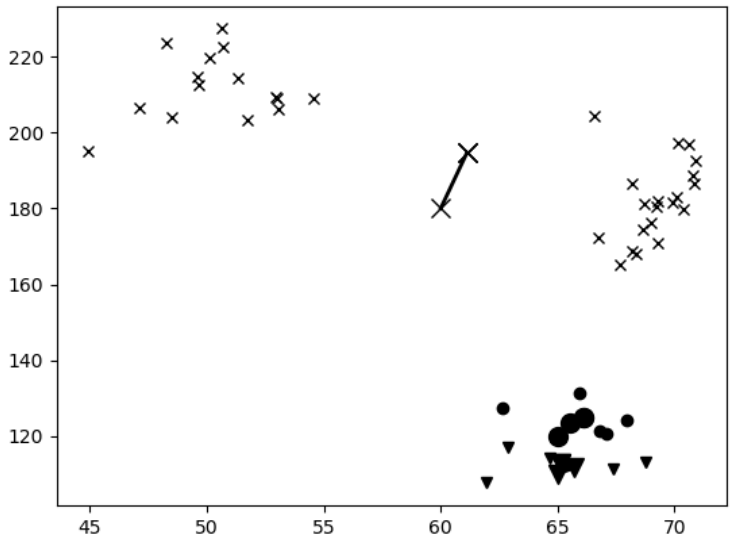

Figure 4 – Suivi de l'avancée de l'algorithme avec un départ biaisé

II.2 Implémentation

Nous allons découper l'algorithme dans l'écriture de trois fonctions.

Distribution

`distribue_selon_centroides(donnees,centroides)` prends deux arguments en paramètres :

- Un tableau Numpy `donnees` de données de dimensions (m, p) à m lignes et p colonnes (chaque ligne représentant les coordonnées d'un point de mesure de l'échantillon)
- Un tableau Numpy `centroides` de dimensions (k, p) contenant k lignes à p éléments (chaque ligne représentant les coordonnées d'un candidat centroïde).

La fonction doit renvoyer un tableau Numpy de dimensions m contenant le numéro (entre 0 et $k-1$) du centroïde le plus proche avec la norme euclidienne usuelle [9].

Calcul des centroïdes

`recalcule_centroides(donnees,classes,centroides)` s'occupe de recalculer la position des nouveaux centroïdes à partir des différents groupes qui ont été définis. Elle doit prendre trois arguments :

- `donnees` est un tableau Numpy de dimensions (m, p) comme avant.
- `classes` est un tableau Numpy de dimension m tel qu'il a été renvoyé par la fonction précédente. Il contient donc le numéro du centroïde auquel « appartient » chaque point des données.
- `centroides` est un tableau Numpy de dimensions (k, p) comme avant contenant les positions des centroïdes précédents.

La fonction doit renvoyer un tableau Numpy de dimensions (k, p) contenant les positions des nouveaux centroïdes calculés en faisant la moyenne des positions de tous les points appartenant à un groupe donné (dans l'ordre des groupes donnés par les numéros du tableau `classes`). Si jamais un centroïde ne possède aucun point associé, il reste par défaut en place (d'où l'intérêt d'avoir le tableau `centroides` en paramètre).

9. Pas besoin de s'embêter avec la racine, la comparaison marche aussi en ne regardant que le carré de la norme.

Attention, il ne faut pas modifier directement le tableau `centroides` donné en paramètre mais s'assurer qu'on en fait une copie [10].

Algorithme complet

`algo_kmeans(donnees,depart_centroides)` va appliquer les deux fonctions précédentes jusqu'à convergence des centroïdes. On lui donne en paramètre le tableau Numpy des données ainsi que des estimations pour les points de départ des centroïdes voulus. Il doit renvoyer les positions finales des centroïdes sous forme d'un tableau Numpy de dimensions (k, p) ainsi qu'un tableau Numpy de dimension m associant à chaque point son numéro de centroïde et permettant donc de partitionner les données en différents groupes.

II.3 Utilisation dans le cas du Soleil

En utilisant l'algorithme développé précédemment, partitionner les pixels des images fournies pour le Soleil en deux groupes : ceux qui appartiennent au Soleil et ceux qui appartiennent au fond de l'image. Pour lire le fichier image, on pourra utiliser la commande `imageio.imread` du module `imageio` (normalement par défaut fourni avec Pyzo) qui renvoie un `np.array` de dimensions `largeur` × `hauteur` × 3 lorsque les pixels sont codés en RGB comme c'est le cas ici. Plus particulièrement, on pourra utiliser comme départ

```
1 import imageio  # Importation du module idoine
2
3 im = imageio.imread('telescope/Eclipse1.jpg') # Lecture de l'image
4
5 # On récupère la largeur, la hauteur et le nombre de couleurs par pixel
6 largeur,hauteur,couleurs = im.shape
```

Écrire une fonction `enveloppe(image)` qui prend en argument la chaîne de caractère représentative de l'emplacement d'une image sur le disque [11] et renvoie une liste de doublets repérant les coordonnées des points délimitant la surface du Soleil sur la photo en utilisant l'algorithme défini précédemment. On doit pouvoir afficher une image du type de la figure 5 à partir de la liste précédente.

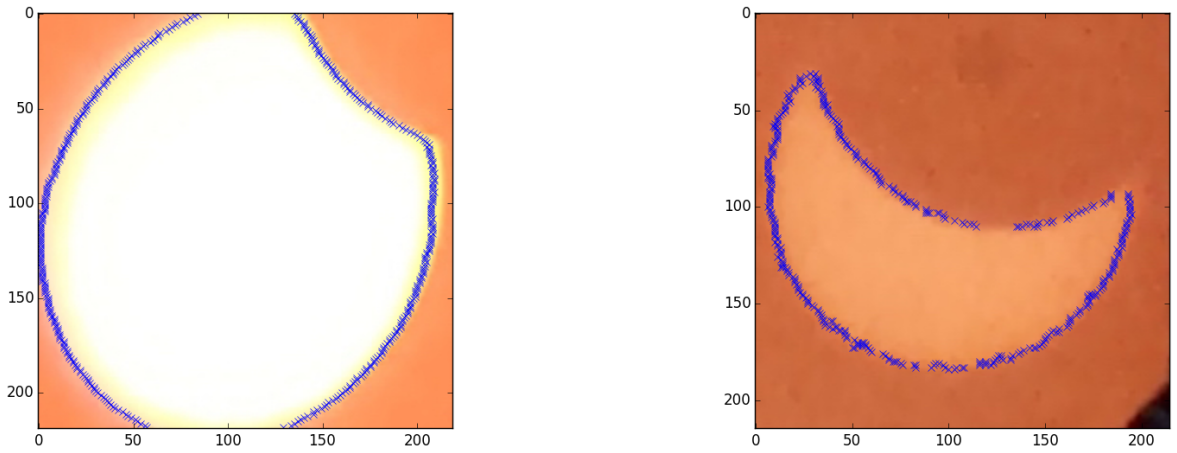

Figure 5 – Détection de l'enveloppe solaire

10. Avec `nouveau = np.array(centroides)` par exemple.
11. Par exemple `'kleber/eclipse_08.jpg'`.

Partie III

Régression « linéaire »

III.1 Principe

Supposons que l'on dispose d'un ensemble de m couples de valeurs $(x_{(i)}, y_{(i)})$ et que l'on veuille ajuster un polynôme du type $y = \theta_0 + \theta_1 x + \theta_2 x^2 = f_\Theta(x)$ à ces données en notant Θ le vecteur colonne ${}^t(\theta_0, \theta_1, \theta_2)$. Il va donc falloir ajuster les valeurs des θ_j de sorte que l'écart entre la prédiction $f_\Theta(x)$ et la valeur effective y soit la plus faible possible. Une manière de procéder est de tenter de minimiser la fonction de coût suivante :

$$J(\Theta) = \sum_{i=1}^{m} (y_{(i)} - f_\Theta(x))^2$$

Cela peut se faire en calculant le gradient de la fonction précédente et en suivant la direction de plus grande pente jusqu'à arriver à un minimum, mais c'est relativement long et compliqué, sans compter qu'il va falloir choisir le pas de déplacement pour converger rapidement tout en étant suffisamment précis... Il existe néanmoins une autre méthode [12]. Notons X la matrice à m lignes et trois colonnes (pour notre exemple)

$$X = \begin{pmatrix} 1 & x_{(1)} & x_{(1)}^2 \\ \vdots & \vdots & \vdots \\ 1 & x_{(i)} & x_{(i)}^2 \\ \vdots & \vdots & \vdots \\ 1 & x_{(m)} & x_{(m)}^2 \end{pmatrix} \qquad \text{alors} \qquad X\Theta = \begin{pmatrix} \theta_0 + \theta_1 x_{(1)} + \theta_2 x_{(1)}^2 \\ \vdots \\ \theta_0 + \theta_1 x_{(i)} + \theta_2 x_{(i)}^2 \\ \vdots \\ \theta_0 + \theta_1 x_{(m)} + \theta_2 x_{(m)}^2 \end{pmatrix} = \begin{pmatrix} f_\Theta(x_{(1)}) \\ \vdots \\ f_\Theta(x_{(i)}) \\ \vdots \\ f_\Theta(x_{(m)}) \end{pmatrix}$$

c'est-à-dire le vecteur colonne des prédictions des résultats $f_\Theta(x_{(i)})$ pour chaque point de données. De même si l'on note Y le vecteur colonne contenant les m valeurs $y_{(i)}$, alors la fonction de coût s'écrit sous forme vectorielle

$$J(\Theta) = \|Y - X\Theta\|^2$$

On peut alors montrer [13] que la fonction suivante est minimisée lorsque l'on vérifie l'équation normale

$${}^tX\,X\,\Theta = {}^tX\,Y \qquad \text{soit} \qquad \Theta = \left({}^tX\,X\right)^{-1}\,{}^tX\,Y$$

à condition bien sûr que la matrice ${}^tX\,X$ soit bien inversible, ce qui devrait être le cas dans la plupart des cas pratiques où le nombre de points est suffisant.

III.2 Implémentation

Écrire une fonction `ajustement(X,Y)` qui prend en argument une matrice `X` de m lignes et autant de colonnes (disons p) que l'on aura de paramètres d'ajustement ainsi qu'un vecteur colonne (sous la forme d'une `np.matrix` à m lignes et une seule colonne). La fonction doit renvoyer une matrice à p lignes et 1 seule colonne contenant les paramètres d'ajustement optimaux recherchés.

III.3 Utilisation dans le cas du Soleil

Régression « circulaire »

On va vouloir ajuster l'équation d'un cercle à tous les points trouvés sur la circonférence du Soleil. L'équation en question s'écrit

$$(x-a)^2 + (y-b)^2 = R^2 \qquad \text{soit} \qquad x^2 + y^2 = 2ax + 2by + (R^2 - a^2 - b^2)$$

12. Algèbre linéaire à la rescousse !

13. Voir `http://en.wikipedia.org/wiki/Linear_least_squares_(mathematics)#Derivation_of_the_normal_equations`

La matrice X va donc devoir contenir une colonne avec les valeurs des x, une colonne avec les valeurs des y et une colonne avec seulement des 1 (pour la partie constante). Le vecteur colonne Y quant à lui doit contenir les valeurs de x^2+y^2 pour chaque point. Écrire une fonction `regression_circulaire(couples)` qui prend en entrée une liste de couples (x,y) sur lesquels appliquer la régression (sous la forme d'un `np.array` de dimension $(m,2)$) et renvoie les valeurs de a, b et R obtenues.

Recentrage des images

À partir des images du dossier `telescope/`, mettre au point une procédure qui permette de transformer les images de sorte que le Soleil soit toujours exactement au même endroit sur les trois images de sorte que la projection successive des trois images permette de visualiser l'avancée de la trace de la Lune « comme un film ». Hint : il faudra itérer [14] lors de la régression circulaire pour réussir à supprimer des bords les points correspondant à la frontière Soleil-Lune.

Pour vérifier que la régression se fait correctement, on pourra utiliser des courbes de niveau en surimposant le lieu des points où $(x-a)^2+(y-b)^2-R^2$ vaut 0 sur l'image comme le montre la figure 6 en utilisant le code suivant

```
xl = np.arange(largeur)
yl = np.arange(hauteur)
xgrid,ygrid = np.meshgrid(xl,yl)
zgrid = (xgrid-a)**2 + (ygrid-b)**2 - R**2
plt.contour(xgrid,ygrid,zgrid,levels=[0],colors=['white'],linewidth=4)
```

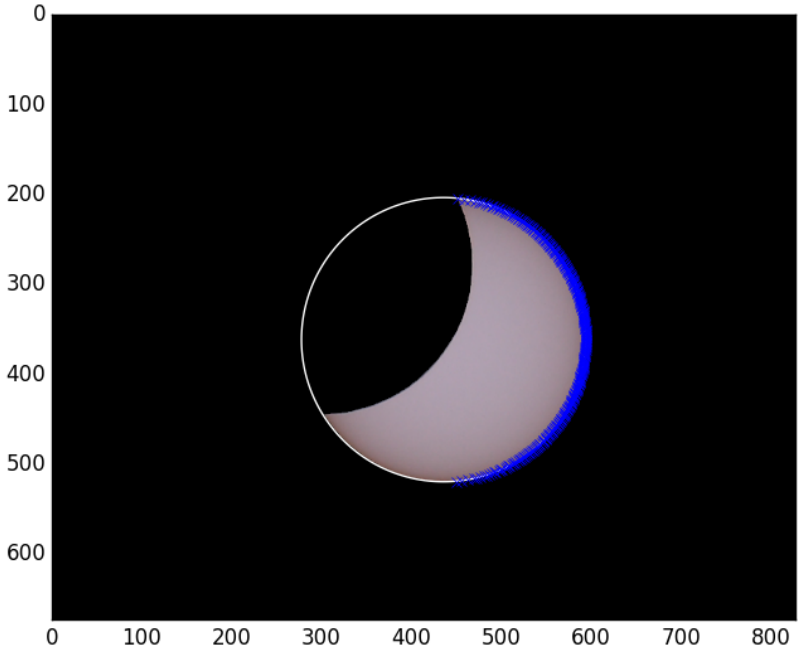

Figure 6 – Ajustement du cercle solaire

14. Une autre possibilité est d'utiliser l'algorithme de détection d'enveloppe convexe décrit dans le sujet d'info MP-PC X 2015.

Régression « elliptique »

Les images du dossier `kleber/` sont bien moins bonnes (et c'est normal) que celles du dossier `telescope/` : la projection est de travers, la prise de vue est de travers, bref, tout est de travers ☺. De ce fait, le beau cercle solaire est déformé en ellipse qui faut donc apprendre à ajuster. L'équation la plus générale d'une telle ellipse s'écrit

$$x^2 + Axy + By^2 + Cx + Dy + E = 0$$

La matrice X va cette fois contenir des lignes du type $(xy, y^2, x, y, 1)$ pour chaque position (x, y) d'un point candidat pour se placer sur l'ellipse. À l'inverse, le vecteur colonne Y devra contenir toutes les valeurs de x^2 pour chaque point. Écrire une fonction `regression_elliptique(couples)` qui prend en entrée une liste de couples (x, y) sur lesquels appliquer la régression (sous la forme d'un `np.array` de dimension $(m, 2)$) et renvoie les valeurs des coefficients A à E obtenus.

Recircularisation des images

Utiliser la fonction précédente sur les images du répertoire `kleber` pour « redresser » le Soleil. Comme pour la régression circulaire, on aura besoin de trouver un moyen de différencier les points de la frontière qui sont vraiment sur le pourtour solaire de ceux qui sont sur le pourtour lunaire. On aura aussi besoin de la manière de transformer une ellipse en cercle à partir de la donnée de ses paramètres A à E. Une manière de procéder pourrait être la suivante :

- Posons $f(x, y) = x^2 + Axy + By^2 + Cx + Dy + E$
- Le centre Ω de l'ellipse définie par $f(x, y) = 0$ est tel que $\dfrac{\partial f}{\partial x} = \dfrac{\partial f}{\partial y} = 0$, soit
$$x_\Omega = \frac{DA - 2BC}{4B - A^2} \qquad \text{et} \qquad y_\Omega = \frac{CA - 2D}{4B - A^2}$$
- En se plaçant dans le repère centré sur Ω, on peut effectuer un changement de base orthonormée vers la b.o.n. correspondant aux grand et petit axes de l'ellipse. Pour ce faire, on peut utiliser la routine `np.linalg.eig` du module `linalg` de NumPy (à importer donc) qui permet de récupérer à la fois les valeurs propres et les vecteurs propres en l'appliquant sur la matrice
$$M = \begin{pmatrix} 1 & A/2 \\ A/2 & B \end{pmatrix}$$
- Dans cette base, si l'on note λ_1 et λ_2 (avec $\lambda_1 > \lambda_2$) les deux valeurs propres trouvées, il suffit de multiplier la coordonnée selon le vecteur propre correspondant à λ_1 par $\sqrt{\lambda_2/\lambda_1}$ pour se retrouver avec un cercle qu'il suffit de replacer dans la base initiale par le changement de base inverse. L'ensemble des trois opérations précédentes peut se réaliser en multipliant trois matrices (deux de passage et une diagonale) à stocker dans un coin pour savoir, pour chaque pixel, où envoyer sa couleur [15]. Un exemple de résultat est donné en figure 7.

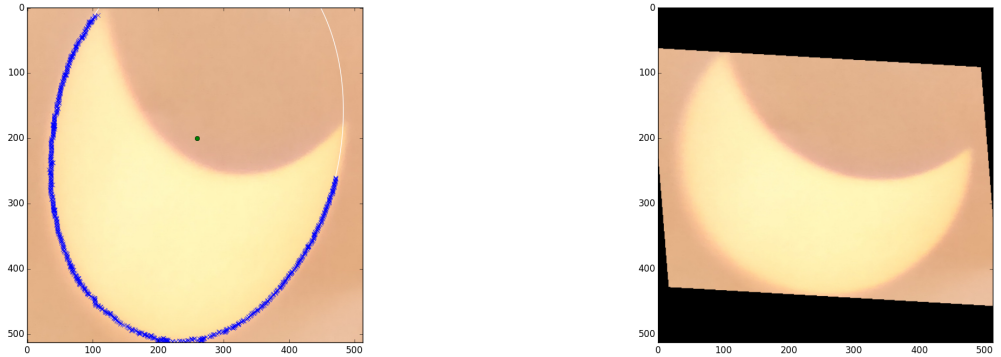

Figure 7 – Exemple d'application : à gauche l'image d'origine, à droite l'image « redressée ».

15. Ne pas oublier de rajouter les coordonnées du centre Ω qui est le point fixe de la transformation.