Informatique

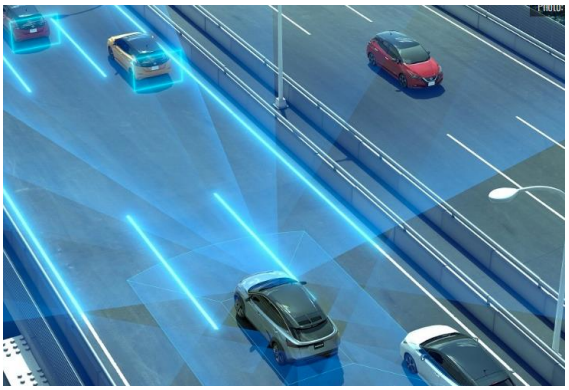
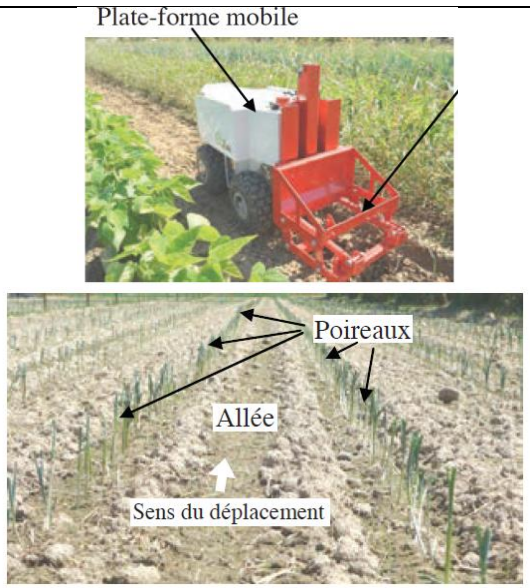
Matrices de pixels et images

Détection automatique de lignes blanches

A.I. Contexte

Ce sujet aborde la détection automatique de lignes blanches sur la route, une problématique clé pour les véhicules ou robots autonomes. Il se déroule en plusieurs étapes progressives.
On commence par découvrir comment Python représente les images sous forme de tableaux numériques, avant d'ouvrir et afficher l'image de travail. L'image couleur est ensuite convertie en niveaux de gris puis en noir et blanc, afin d'isoler les pixels les plus lumineux, ceux qui correspondent aux lignes blanches.
Cette binarisation laisse cependant apparaître de nombreux pixels parasites : une étape de convolution permet de les supprimer et de ne conserver que les zones significatives. On extrait alors la liste des coordonnées des pixels blancs restants, qui constitue le jeu de données sur lequel s'appuie la suite du traitement.
Pour trouver la droite qui modélise au mieux la ligne blanche, on met en place les outils statistiques nécessaires (moyenne, variance et covariance) puis on applique la méthode des moindres carrés, qui fournit l'équation de la droite la plus proche de l'ensemble des points blancs.
Le résultat est finalement superposé à l'image d'origine sous forme d'un tracé rouge, rendant la détection visuellement interprétable.
Une dernière partie étend la méthode au cas de plusieurs lignes, en introduisant un algorithme d'exploration par zones pour identifier, dénombrer et traiter chaque ligne blanche indépendamment.

Voici deux exemples d'application :

Véhicules autonomes	Robots autonomes agricoles (extrait CCP MP 2016 SI)
	

Nous nous limiterons dans ce sujet à des lignes blanches au centre de la route, aucune ligne n'étant présente sur les côtés.

Nous avons dans le dossier de travail l'image ci-dessous nommée « Image.bmp » et travaillerons avec des chemins relatifs.

Cette image contient 392 lignes et 1960 colonnes et ses pixels sont de type RGB.

Vous avez en annexe à la fin de ce sujet un extrait de l'aide de Python à propos des modules matplotlib et numpy.

Consignes spécifiques :

- L'usage des calculatrices est interdit
- Les modules seront importés avec leurs notations usuelles (np, plt…)

Les images sous python

On ouvre les images avec la fonction imread du module matplotlib.

Question 1: Préciser le type des images ainsi ouvertes

Question 2: Préciser le type de codage des nombres dans ces images

Question 3: En détaillant le calcul, préciser le nombre d'octets que prend un pixel dans la mémoire de l'ordinateur

Question 4: Préciser enfin la taille de l'image fournie en Mo arrondie au dixième

Ouverture de l'image

L'image bmp est placée dans le répertoire où notre fichier Python est stocké.

Question 5: Créer une fonction Affiche(fig,im) qui permet d'afficher l'image im (format issu de imread) sur la figure fig

Question 6: Ecrire les lignes de code créant l'image « Image » et l'affichant sur la figure 1

Traitement noir et blanc

Question 7: Créer une fonction NG(im,kR,kG,kB) qui renvoie une nouvelle image (format array de numpy) qui sera la transformation de l'image im en nuances de gris avec kR, kG et kB les facteurs sélectionnant les couleurs R, G et B de l'image (im non modifiée)

On prendra $kR = kG = kB = \frac{1}{3}$.

Question 8: Ecrire les lignes de codes créant l'image « Image_NG » et affichant l'image en nuances de gris sur la figure 2

Nous allons transformer l'image obtenue en noir et blanc tel que : $\begin{cases} im[l,c,0] > k \Rightarrow Blanc \\ im[l,c,0] \leq k \Rightarrow Noir \end{cases}$

Question 9: Créer une fonction NB(im,k) qui renvoie une nouvelle image en noir et blanc comme souhaité à partir de l'image im obtenue par la fonction NG (im non modifiée)

Question 10: Ecrire les lignes de codes créant l'image « Image_NB » avec le critère 240 et l'affichant sur la figure 3

Remarque : k devra être adapté aux conditions (heure, lieu…) mais ce n'est pas l'objet du sujet.

Question 11: Préciser la complexité en temps des fonctions NG et NB

Voici le résultat obtenu :

Cette image présente une multitude de points blancs qui nuisent grandement à la détermination de la droite passant au mieux par les pixels blancs aux moindres carrés abordée par la suite.

Suppression des points blancs

On donne le code suivant :

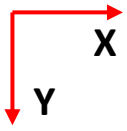
```
def Convolution(im,K):
    Nl,Nc = im.shape[0:2]
    Dim_K = K.shape[0]
    k = Dim_K//2
    Im_Conv = np.copy(im)
    for c in range(k,Nc-k):
        for l in range(k,Nl-k):
            Im_Loc = im[l-k:l+k+1,c-k:c+k+1]
            R,G,B = 0,0,0
            n = K.shape[0]
            for ll in range(n):
                for cc in range(n):
                    tk = K[ll,cc]
                    RM,GM,BM = Im_Loc[ll,cc]
                    R += tk*RM
                    G += tk*GM
                    B += tk*BM
            R,G,B = int(R),int(G),int(B)
            R,G,B = max(min(R,255),0),max(min(G,255),0),max(min(B,255),0)
            Im_Conv[l,c] = R,G,B
    return Im_Conv
```

Question 12: Proposer un traitement à l'image « Image_NB » afin d'obtenir le résultat ci-dessous qui sera affiché sur la figure 4

Remarque : l'image Image_NB sera écrasée et remplacée par le résultat du traitement

Création des listes des pixels blancs

Pour la suite, on appelle X et Y les coordonnées d'un pixel comme le couple $[c, l]$.



On souhaite obtenir deux listes LX et LY contenant respectivement les colonnes et les lignes des pixels blancs de l'image. Ainsi, pour un pixel blanc d'indice i, ses lignes et colonnes seront : $LX[i], LY[i] = X, Y = c, l$

Question 13: Créer une fonction Extraction(im) renvoyant ces listes

Question 14: Créer les lignes de codes créant LX et LY

Moyenne - Variance - Covariance

Si L est une liste de flottants composée de N termes on rappelle les formules donnant sa moyenne *m* et sa variance *V* :

$$\mathrm{m} = \frac{1}{N}\sum_{i=0}^{N-1} L[i] \quad ; \quad V = \frac{1}{N}\sum_{i=0}^{N-1} (L[i]-m)^2$$

Si L_1 et L_2 sont deux listes de flottants composées chacune de N termes, on appelle covariance de L_1 et L_2 le nombre :

$$cov(L_1, L_1) = \left(\frac{1}{N}\sum_{i=0}^{N-1} L_1[i]L_2[i]\right) - m_x m_y$$

où m_x et m_y sont les moyennes respectives de L_1 et L_2.

Question 15: Ecrire une fonction Moy(L) qui, étant donnée une liste de nombres L, renvoie sa moyenne

Question 16: Ecrire une fonction Var(L) qui, étant donnée une liste de nombres L, renvoie sa variance. On veillera à obtenir la meilleure complexité possible

Question 17: Ecrire une fonction Cov(L1,L2) qui, étant données deux listes L1 et L2 contenant le même nombre d'éléments, renvoie la covariance de L1 et L2.

Droite aux moindres carrés

Nous allons maintenant déterminer l'équation de la droite qui passe au mieux par les pixels blancs de l'image aux moindres carrés.

Dans un premier temps, un peu de cours sur les approximations de droites aux moindres carrés. On souhaite déterminer l'équation de la droite qui passe au plus près, au sens des moindres carrés, d'un ensemble de points.

Soit la droite d'équation : $y = ax + b$

On souhaite minimiser la somme des carrés des distances entre N points connus - dont les abscisses et les ordonnées sont stockées dans deux listes X et Y - et leurs projetés sur la droite parallèlement à l'axe (Oy). Si on note S cette somme, on a :

$$S(a,b) = \sum_{i=0}^{N-1} \big(y_i - (ax_i + b)\big)^2 = \sum_{i=0}^{N-1} (y_i - ax_i - b)^2$$

Le minimum de S sera atteint lorsque les deux dérivées partielles de S sont nulles. Il faut donc résoudre le système ci-dessous :

$$\begin{cases} \dfrac{\partial S(a,b)}{\partial a} = 0 \\ \dfrac{\partial S(a,b)}{\partial b} = 0 \end{cases} \Leftrightarrow \begin{cases} -2\displaystyle\sum_{i=0}^{N-1} x_i(y_i - ax_i - b) = 0 \\ -2\displaystyle\sum_{i=0}^{N-1} (y_i - ax_i - b) = 0 \end{cases}$$

En notant m_x et m_y les valeurs moyennes des listes X et Y, les solutions a et b du système sont :

$$\begin{cases} a = \dfrac{\left[\frac{1}{N}\sum_{i=0}^{N-1}(x_i y_i)\right] - m_x m_y}{\left[\frac{1}{N}\sum_{i=0}^{N-1}({x_i}^2)\right] - {m_x}^2} = \dfrac{cov(X,Y)}{V(X)} \\ b = m_y - a m_x \end{cases}$$

Remarque : on peut montrer que $\left[\frac{1}{N}\sum_{i=0}^{N-1}({x_i}^2)\right] - {m_x}^2 = \left[\frac{1}{N}\sum_{i=0}^{N-1}(x_i - m_x)^2\right]$

Question 18: Ecrire une fonction Droite(LX,LY) qui renvoie le couple (a,b) des coefficients de la droite qui passe au mieux au sens des moindres carrés par les points dont les abscisses sont dans la liste LX et les ordonnées dans la liste LY

Remarque : on supposera que la droite verticale X=k ne peut exister (c'est très improbable).

Question 19: Ecrire les lignes de codes permettant de calculer a et b sur l'image étudiée

Affichage du résultat

On souhaite finalement ajouter une droite rouge à la photo initiale représentant la droite obtenue par la méthode des moindres carrés.

Pour rappel, a et b correspondent à la droite D telle que :

$$l = a * c + b$$

Si une droite possède l'équation $Ax + By + C = 0$, alors la distance d'un point $P(X, Y)$ à la droite s'écrit :

$$d = \left| \frac{AX + BY + C}{\sqrt{A^2 + B^2}} \right|$$

Soit P un point stocké sous forme de liste [X,Y] de ses coordonnées [c,l].
Soit LD la liste [A,B,C] des trois réels représentant l'équation de la droite D.

Question 20: Créer la fonction Distance(P,LD) renvoyant la distance du point P à la droite D
Question 21: Créer une fonction Ajoute_Droite(im,LD,dst) qui copie im, lui ajoute des pixels rouges si leur distance à la droite D est inférieure à dst et la renvoie
Question 22: Créer les lignes de code créant l'image « Image_Res » contenant les pixels rouges sur l'image de départ avec un critère de distance de 3 pixels et l'affichant sur la figure 5

Voici le résultat obtenu :

Si vous utilisez cet algorithme pour la résolution d'un problème de détection de ligne, voici quelques remarques :

- Avoir une image de qualité (assez de pixels) pour éviter des calculs trop approximatifs
- Bien choisir le critère sur l'image N&B pour qu'après traitement, il n'y ait AUCUN pixel blanc isolé qui change fortement la droite obtenue à la fin
- Être conscient que du fait de la perspective, le bas de la ligne à plus de poids (plus de pixels) dans le calcul aux moindres carrés sur le haut. Si besoin, utiliser un algorithme de recadrage

Cas de plusieurs lignes

On s'intéresse maintenant à l'image suivante disponible dans le dossier de travail sous le nom « Image 2.bmp » :

Question 23: Ecrire le code permettant de réaliser tout le traitement en noir et blanc avec convolution afin d'obtenir le résultat ci-dessous stocké dans l'image « Image_NB » et affiché sur la figure 6 (utiliser Crit_NB=160)

L'image obtenue ci-dessus présente une grande « zone » noire et un grand nombre de « zones » blanches. Nous allons programmer une méthode permettant d'associer un numéro à chacune de ces zones afin de pouvoir les dénombrer et de les étudier.

Pour chaque pixel de l'image désigné par une liste [l,c] de sa ligne l et sa colonne c, on souhaite déterminer chacun de ses 4 voisins (gauche, droite, bas, haut) dans cet ordre sous la forme d'une liste de 4 listes [li,ci]. Un voisin qui sortirait de l'image sera défini comme égal au pixel [l,c].

Question 24: Proposer une fonction Liste_voisins(l,c,Nl,Nc) prenant en argument la ligne l et colonne c du pixel étudié, les nombres de lignes Nl et colonnes Nc de l'image, et renvoyant la liste de ses 4 voisins comme précisé ci-dessus

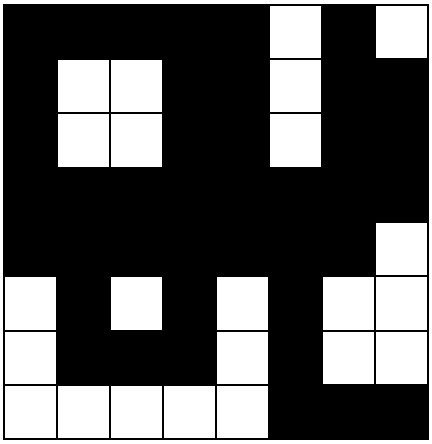
Exemple de résultats obtenus :

```
>>> Liste_voisins(0,0,10,10)
[(0, 0), (0, 1), (1, 0), (0, 0)]

>>> Liste_voisins(1,1,10,10)
[(1, 0), (1, 2), (2, 1), (0, 1)]

>>> Liste_voisins(10,10,10,10)
[(10, 9), (10, 9), (9, 10), (9, 10)]
```

On suppose avoir à disposition une table T sous forme d'array à deux dimensions, de mêmes dimensions que l'image étudiée, contenant des entiers codés sur 64 bits. Elle est initialisée à des valeurs de -1 partout. On souhaite que cette table contienne par la suite des entiers positifs croissants tels que toutes les cases contenant l'entier k représentent une « zone » numéro k de l'image noir et blanc.

Ainsi, comme le premier pixel en haut à gauche est noir, la première zone de numéro k=0 sera l'intégralité du noir de l'image ne formant qu'une zone. La zone 1 sera une première zone blanche, la zone deux une seconde, etc.

Voici un exemple de table T associée à l'image de 8x8 pixels ci-contre en réalisant un parcours colonne par colonne (pour chaque colonne, puis pour chaque ligne) de la procédure que nous allons mettre en place :



0	0	0	0	0	4	0	6
0	2	2	0	0	4	0	0
0	2	2	0	0	4	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	5
1	0	3	0	1	0	5	5
1	0	0	0	1	0	5	5
1	1	1	1	1	0	0	0

Ce qui donnera, en affichant la table T avec notre fonction Affiche (lorsque ce sont des entiers, et non des triplets, on obtient un dégradé de couleurs de bleu à jaune automatiquement) :

Dans la suite, on dira qu'une case T[l,c] a été **marquée** si sa valeur T[l,c] est différente de -1.
Par ailleurs, on dira par abus de langage « **pixel [l,c]** » pour désigner le pixel à la ligne l et la colonne c et « **case [l,c]** » la valeur de la case à la ligne l et la colonne c de la table T.
Enfin, comme nous travaillerons sur l'image en noir et blanc, on parlera de « **valeur** » 0 ou 255 pour identifier les valeurs R=G=B de tous les pixels.

On souhaite créer une procédure qui, à partir d'un pixel initial [l,c] quelconque de l'image associé à une case T[l,c] initialisée à un entier k dans T, explore tous ses voisins non encore marqués ayant la même valeur, afin de les marquer du même entier k, et procède ainsi pour tous les voisins des voisins (etc.), tant qu'il y en a. Cette procédure va donc permettre de remplir la table T avec le même entier k pour chaque zone de l'image.

Principe de la fonction d'exploration à partir du pixel initial [l,c] :

- Initialiser une pile (liste) avec le couple [l,c] du pixel initial
- Tant que la pile n'est pas vide :
    - Dépiler un pixel [l,c]
    - Affecter l'entier k à la case [l,c]
    - Déterminer tous les voisins du pixel [l,c]
    - Pour chaque voisin non marqué de même valeur :
        - Ajouter ce voisin à la pile

  Attention : Tout n'est volontairement pas dit

Exemples : L'appel de la fonction en

- [0,0] va créer toute la zone de 0 de la table T
- [5,0] va créer toute la zone de 1 dans la table T

Question 25: Proposer une fonction Explorer(imnb,l,c,T,k) prenant en argument la ligne l et la colonne c du pixel initial et l'entier k, et réalisant la procédure attendue

Il reste maintenant à appeler cette procédure sur l'intégralité des pixels non marqués de la table T sur l'image en noir et blanc à partir du pixel [0,0].

Principe de la fonction de détermination des zones :

- Initialisation d'une table T d'entiers à -1 (`dtype='int64'`)
- Initialisation de l'entier k à 0
- Parcours de tous les pixels colonne par colonne
- Exploration du pixel s'il n'est pas marqué en lui attribuant l'entier k (création de la zone k)
- Incrémentation de l'entier k

  Attention : Tout n'est volontairement pas dit

Question 26: Créer la fonction Zones(imnb) réalisant cette procédure et renvoyant la table T attendue

Question 27: Ecrire les lignes de code créant la table « Table » associée à l'image, l'affichant sur la figure 7, et créant et affichant dans la console le nombre de zones trouvées « Nb_Zones »

Remarque : On pourra utiliser np.amax(A) pour obtenir le maximum d'un array A.

Dans la suite, les arguments n et T lors de la définition des fonctions seront le nombre de zones « Nb_Zones » et la table « Table ».

Vous devriez obtenir ceci :

On souhaite mettre en place une fonction qui, en un seul parcours de la table T, renvoie une liste LD des données D de chaque zone LD[k]=D=[Taille,Ml,Mc,Col,l_min,c_min,l_max,c_max] avec :

- Taille : nombre de pixels de la zone k (entier)
- Ml : ligne moyenne de la zone k (flottant)
- Mc : colonne moyenne de la zone k (flottant)
- Col : valeur R du triplet RBG de la zone k (0 : Noir, 255 : Blanc)
- l_min,c_min,l_max,c_max sont les limites de la zone k (incluses)

Remarque : le point [Ml,Mc] est le centre de la zone k sur l'image. Pour rappel, on obtient le centre (L,C) d'un ensemble de n pixels (Li,Ci) ainsi :

$$\begin{pmatrix} M_l \\ M_c \end{pmatrix} = \begin{pmatrix} \frac{1}{n}\sum_{i=0}^{n-1} L_i \\ \frac{1}{n}\sum_{i=0}^{n-1} C_i \end{pmatrix}$$

On propose le code suivant :

```
def Donnees(T,n,imnb):
    Nl,Nc = T.shape
    Taille,Ml,Mc,Col,l_min,c_min,l_max,c_max = 0,0,0,-1,# ZONE 1 #
    LD = [[Taille,Ml,Mc,Col,l_min,c_min,l_max,c_max] for k in range(n)]
    for c in range(Nc):
        for l in range(Nl):
            k = # ZONE 2 #
            Taille,Ml,Mc,_,l_min,c_min,l_max,c_max = LD[k]
            Ml = # ZONE 3 #
            Mc = # ZONE 4 #
            Taille += 1
            Col = # ZONE 5 #
            if l < l_min:
                # ZONE 6 #
            if l > l_max:
                # ZONE 7 #
            if c < c_min:
                # ZONE 8 #
            if c > c_max:
                # ZONE 9 #
            LD[k] = [Taille,Ml,Mc,Col,l_min,c_min,l_max,c_max]
    return LD
```

Question 28: Compléter les différentes zones du code proposé et proposer les lignes de code permettant de créer la liste « Donnees_Zones » contenant les données issues de la fonction Donnees

Dans la suite, l'argument LD lors de la définition des fonctions sera cette liste « Donnees_Zones ».
On souhaite pouvoir extraire chacune des zones de l'image en noir et blanc afin d'enregistrer une nouvelle image de fond noir et contenant la seule zone k.

Question 29: Proposer une fonction Extraction(imnb,T,k) renvoyant une nouvelle image (format array) de fond noir de mêmes dimensions que imnb contenant la seule zone numéro k dans sa couleur (blanc ou noir, le noir étant évidemment inutile)

On souhaite créer une fonction Extraction_Blanc permettant de parcourir toutes les zones dans LD et affichant puis enregistrant une nouvelle image dans le répertoire de travail pour chaque zone blanche de l'image en noir et blanc si elle possède plus de Tmin=50 pixels avec pour nom son indice k suivi de l'extension « .png ».

Question 30: Proposer une fonction Extraction_Blanc(imnb,T,n,LD,Tmin) réalisant le travail souhaité
Question 31: Ecrire les lignes de code permettant de créer les images attendues

Il ne restera plus qu'à faire tourner l'algorithme précédent sur chacune de ces images pour identifier toutes les lignes blanches présentes dans le champ de vision de la caméra.

ANNEXE

Extraits de l'aide numpy	
shape	shape(a) Return the shape of an array. Parameters ---------- a : array_like Input array. Returns ------- shape : tuple of ints The elements of the shape tuple give the lengths of the corresponding array dimensions.
copy	copy(a, order='K', subok=False) Return an array copy of the given object. Parameters ---------- a : array_like Input data. Returns ------- arr : ndarray Array interpretation of `a`.

ones	ones(shape, dtype=None, order='C', *, like=None)
	Return a new array of given shape and type, filled with ones.
	Parameters
	---------- 	
	shape : int or sequence of ints
	Shape of the new array, e.g., ``(2, 3)`` or ``2``.
	dtype : data-type, optional
	The desired data-type for the array, e.g., `numpy.int8`. Default is `numpy.float64`.
	order : {'C', 'F'}, optional, default: C
	Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory.
	like : array_like
	Reference object to allow the creation of arrays which are not NumPy arrays. If an array-like passed in as ``like`` supports the ``__array_function__`` protocol, the result will be defined by it. In this case, it ensures the creation of an array object compatible with that passed in via this argument.
	.. note::
	The ``like`` keyword is an experimental feature pending on acceptance of :ref:`NEP 35 <NEP35>`.
	.. versionadded:: 1.20.0
	Returns
	------- 	
	out : ndarray
	Array of ones with the given shape, dtype, and order.
	See Also
	-------- 	
	ones_like : Return an array of ones with shape and type of input.
	empty : Return a new uninitialized array.
	zeros : Return a new array setting values to zero.
	full : Return a new array of given shape filled with value.
	Examples
	-------- 	
	>>> np.ones(5)
	array([1., 1., 1., 1., 1.])
	>>> np.ones((5,), dtype=int)
	array([1, 1, 1, 1, 1])
	>>> np.ones((2, 1))
	array([[1.],[1.]])
	>>> s = (2,2)
	>>> np.ones(s)
	array([[1., 1.],[1., 1.]])

Extraits de l'aide matplotlib.pyplot	
imread	imread(fname, format=None) Read an image from a file into an array. Parameters ---------- fname : str or file-like The image file to read: a filename, a URL or a file-like object opened in read-binary mode. format : str, optional The image file format assumed for reading the data. If not given, the format is deduced from the filename. If nothing can be deduced, PNG is tried. Returns ------- `numpy.array` The image data. The returned array has shape - (M, N) for grayscale images. - (M, N, 3) for RGB images. - (M, N, 4) for RGBA images.
imshow	imshow(X, cmap=None, norm=None, aspect=None, interpolation=None, alpha=None, vmin=None, vmax=None, origin=None, extent=None, *, filternorm=True, filterrad=4.0, resample=None, url=None, data=None, **kwargs) Display data as an image, i.e., on a 2D regular raster. The input may either be actual RGB(A) data, or 2D scalar data, which will be rendered as a pseudocolor image. For displaying a grayscale image set up the color mapping using the parameters ``cmap='gray', vmin=0, vmax=255``. The number of pixels used to render an image is set by the axes size and the *dpi* of the figure. This can lead to aliasing artifacts when the image is resampled because the displayed image size will usually not match the size of *X* (see :doc:`/gallery/images_contours_and_fields/image_antialiasing`). The resampling can be controlled via the *interpolation* parameter and/or :rc:`image.interpolation`. Parameters ---------- X : array-like or PIL image The image data. Supported array shapes are: - (M, N): an image with scalar data. The values are mapped to colors using normalization and a colormap. See parameters *norm*, *cmap*, *vmin*, *vmax*. - (M, N, 3): an image with RGB values (0-1 float or 0-255 int). - (M, N, 4): an image with RGBA values (0-1 float or 0-255 int), i.e. including transparency. The first two dimensions (M, N) define the rows and columns of the image. Out-of-range RGB(A) values are clipped.
figure	figure(num=None, figsize=None, dpi=None, facecolor=None, edgecolor=None, frameon=True, FigureClass=<class 'matplotlib.figure.Figure'>, clear=False, **kwargs) Create a new figure, or activate an existing figure. Parameters ---------- num : int or str, optional A unique identifier for the figure.
axis	axis(*args, emit=True, **kwargs) Convenience method to get or set some axis properties. option : bool or str If a bool, turns axis lines and labels on or off. If a string, possible values are: ======== == Value Description ======== == 'on' Turn on axis lines and labels. Same as ``True``. 'off' Turn off axis lines and labels. Same as ``False``.
savefig	savefig(*args, **kwargs) Save the current figure. Call signature:: savefig(fname, *, dpi='figure', format=None, metadata=None, bbox_inches=None, pad_inches=0.1, facecolor='auto', dgecolor='auto', backend=None, **kwargs) The available output formats depend on the backend being used. Parameters ---------- fname : str or path-like or binary file-like A path, or a Python file-like object, or possibly some backend-dependent object such as `matplotlib.backends.backend_pdf.PdfPages`. If *format* is set, it determines the output format, and the file is saved as *fname*. Note that *fname* is used verbatim, and there is no attempt to make the extension, if any, of *fname* match *format*, and no extension is appended. If *format* is not set, then the format is inferred from the extension of *fname*, if there is one. If *format* is not set and *fname* has no extension, then the file is saved with :rc:`savefig.format` and the appropriate extension is appended to *fname*.