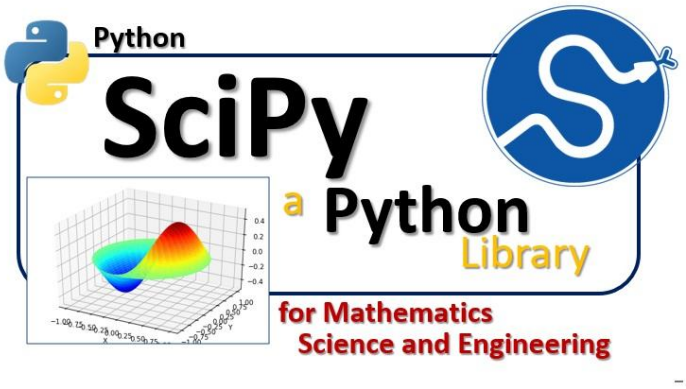
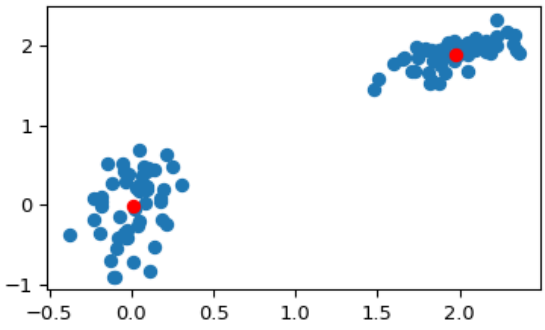
Méthodes pratiques sur SCIPY





- Méthodes pratiques sur SCIPY -

1. Introduction

SciPy est un projet visant à unifier et fédérer un ensemble de **bibliothèques Python à usage scientifique**. Scipy utilise les tableaux et matrices du module **NumPy**.

Il contient par exemple des modules pour l'optimisation, l'algèbre linéaire, les statistiques, le traitement du signal ou encore le traitement d'images.

Il offre également des possibilités avancées de visualisation grâce au module matplotlib.

Comme nous avons déjà parlé des méthodes d'algèbre linéaire dans Numpy, nous allons voir d'autres méthodes ici propre à SCIPY.

2. Module *interpolate*

Qu'est ce que le module scipy.interpolate() ?

Petite illustration avec un exemple :

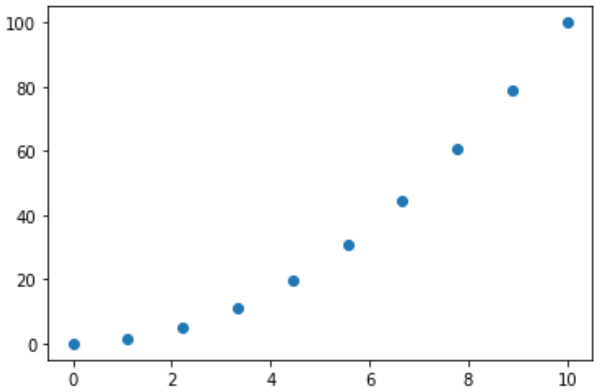
Supposons que l'on soit en train de faire de l'acquisition avec 2 capteurs dont un a un problème et ne renvoit pas toutes les valeurs prélevées. On voit des valeurs Nan apparaitre…

Capteur 1	Capteur 2
0.02	25
0.24	Nan
0.60	Nan
0.93	27
1.23	Nan
1.45	Nan
1.8	28.5

Ces valeurs ne sont pas gâchées, on peut tenter une interpolation

Supposons maintenant que notre capteur nous donne les points suivants à partir d'une fonction x^2 :

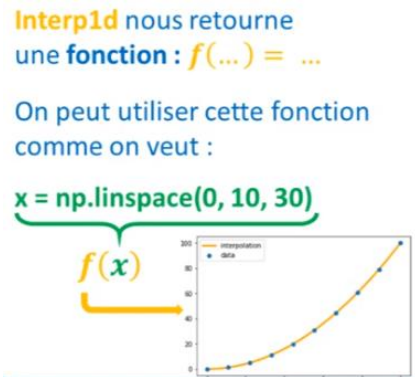
```
import numpy as np
import matplotlib.pyplot as plt
x=np.linspace(0,10,10)
y=x**2
plt.scatter(x,y)
```



L'objectif est **d'interpoler cette courbe**, c'est-à-dire **obtenir plus de points** entre nos points.

- Méthodes pratiques sur SCIPY -

Pour faire cette interpolation, nous allons utiliser la fonction *interp1d* présente dans le module *interpolate* de Scpiy. Ceci va nous permettre de générer une autre fonction f dite d'interpolation (ici elle sera forcée à être linéaire).



Taper le code suivant :

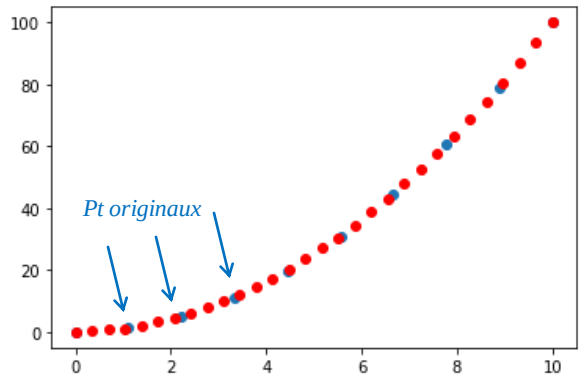
```
from scipy.interpolate import interp1d
f=interp1d(x,y,kind="linear")
```

Maintenant que nous avons créé cette fonction f, on peut en faire ce que l'on veut. Par exemple : on va pouvoir passer dans f un nouvel axe x sur lequel on veut avoir 30 points. On enregistre ce passage dans une nouvelle variable result qui est le tableau Numpy qui va contenir les différents points interpolés.

Voilà ce que l'on peut coder maintenant :

```
new_x=np.linspace(0,10,30)
result=f(new_x)

# on trace l'ancien relevé et le nouveau interpolé
plt.scatter(x,y)
plt.scatter(new_x,result,c="r")
```



Attention : lorsque l'on fait une interpolation, il faut veiller à ce qu'elle ne cache pas ce qu'il se passe dans la réalité !!

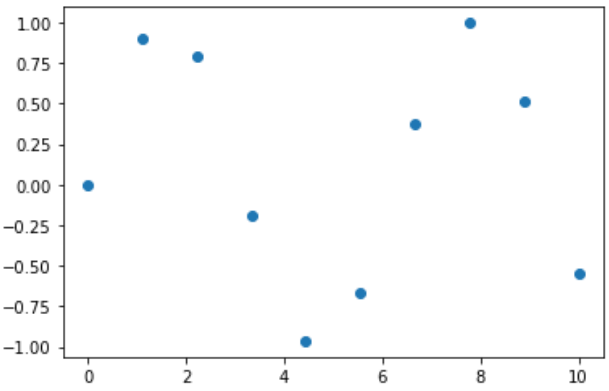
Choisir judicieusement :
- **Pas d'interpolation**
- **Fréquence**

- Méthodes pratiques sur SCIPY -

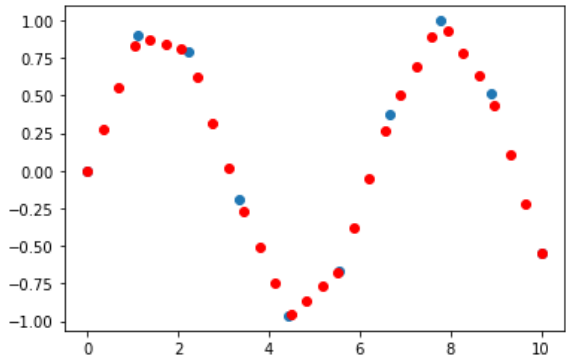
Autres types d'interpolations :

Si nous avons un ensemble de valeur comme suit (loi sinus), nous n'allons pas faire d'interpolation linéaire.

```
x=np.linspace(0,10,10)
y=np.sin(x)
plt.scatter(x,y)
```



Si on reprend notre interprétation linéaire, voilà ce que l'on obtiendrait :

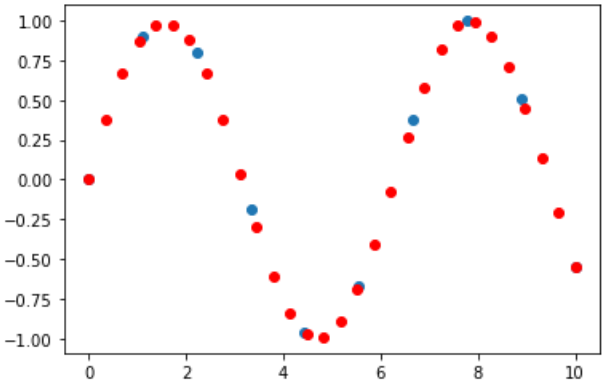


Dans ce cas, il vaut mieux prendre une interpolation *cubic*.

```
f=interp1d(x,y,kind="cubic")

new_x=np.linspace(0,10,30)
result=f(new_x)

# on trace l'ancien relevé et le nouveau interpolé
plt.scatter(x,y)
plt.scatter(new_x,result,c="r")
```



Il existe beaucoup de type d'interpolation dans le module interpolate de Scipy…cf documentation si besoin…

3. Module *optimize*

Quand on parle d'**optimisation**, on fait de plus en plus référence à des **problèmes de minimisation.**

Qu'est ce que le module scipy.optimize() ?

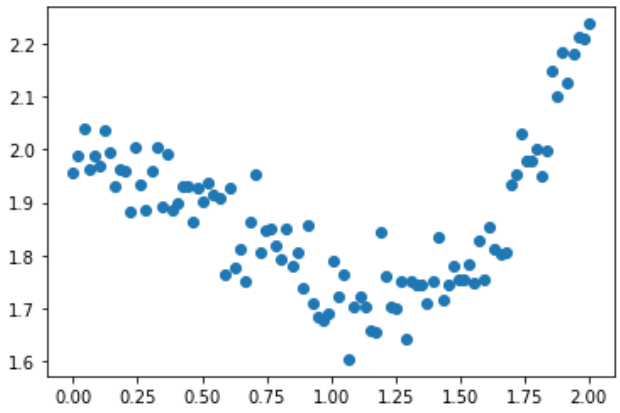
Dans ce module, on retrouve une fonction minimise avec pleins d'algorithmes différents, mais également une fonction *curve_fit* qui nous permet d'optimiser le placement d'une courbe dans un nuage de points. On trouve aussi de quoi faire de la programmation linéaire qui nous permet de résoudre un problème d'optimisation tout en respectant certaines contraintes.

3.1. La fonction curve_fit

Pour illustrer cette fonction, on va créer un nuage de points comme suit, qui est un polynôme de degré 3 sur lequel j'ai rajouté un peu de bruit (avec random).

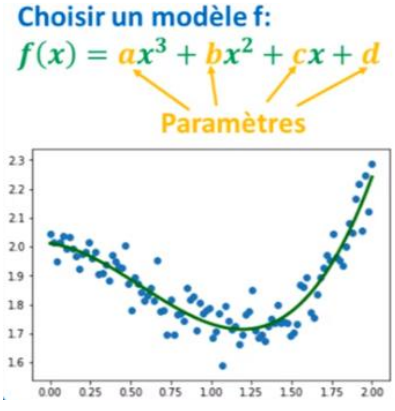
Voici le code que vous pouvez taper :

```
x=np.linspace(0,2,100)
y=1/3*x**3-3/5*x**2+2+np.random.randn(x.shape[0])/20
plt.scatter(x,y)
```



Ce que je souhaiterai faire c'est de créer un **modèle statistique qui rentre parfaitement bien dans mon nuage de points**. Pour cela, on peut utiliser la fonction *curve_fit* qui se sert de la méthode des moindres carrés pour trouver les meilleurs paramètres a,b,c …d'un modèle que l'on aura fournit à notre fonction.

Pour utiliser cette fonction curve.fit, il faut définir avant un modèle. Je propose une fonction f de degré 3 comme suit.



Créer une fonction f :

```
def f(x,a,b,c,d):
    return a*x**3+b*x**2+c*x+d
```

Maintenant ce modèle nous allons le donner à curve_fit

```
from scipy import optimize
optimize.curve_fit(f,x,y)
```

En exécutant le code avec un print, cela nous donne **2 tableaux Numpy**. Le premier donne les **différents paramètres du modèle** (ici a,b,c,d) et le deuxième est **la matrice de covariance** de notre modèle (relation entre les paramètres).

```
[ 0.36619516, -0.71348163,  0.11059977,  1.97408013]),
[[ 0.00104738, -0.00314214,  0.0025011 , -0.00040634],
 [-0.00314214,  0.00970096, -0.00805239,  0.00140021],
 [ 0.0025011 , -0.00805239,  0.00714539, -0.00140736],
 [-0.00040634,  0.00140021, -0.00140736,  0.00037727]]
```

paramètres a,b,c,d

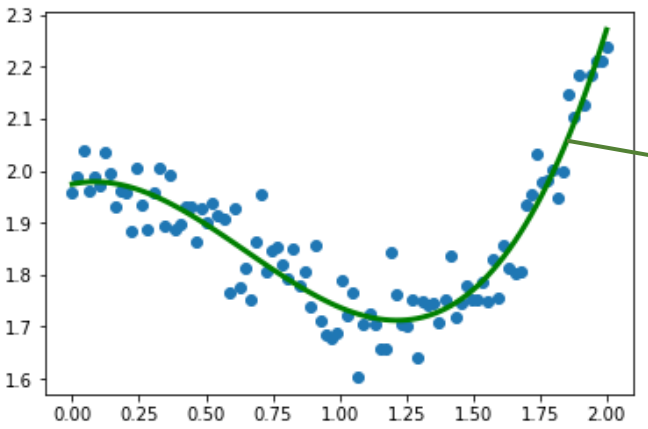
matrice de covariance

Nota : pour notre exemple d'optimisation, seuls les paramètres optimisés sont importants, oubliez la matrice de covariance

On va affecter ces 2 tableaux dans 2 variables et tracer dans matplotlib notre modèle en fonction de x en récupérant les 4 paramètres.

Voici le code :

```
from scipy import optimize
params, params_cov=optimize.curve_fit(f,x,y)
plt.scatter(x,y)
plt.plot(x,f(x,params[0],params[1], params[2], params[3]),c="g", lw=3)
```



Voilà notre modèle d'ordre 3 qui approche au mieux le nuage de points …

4. Module *minimize*

Ce module permet de minimiser n'importe quelle fonction mathématique.

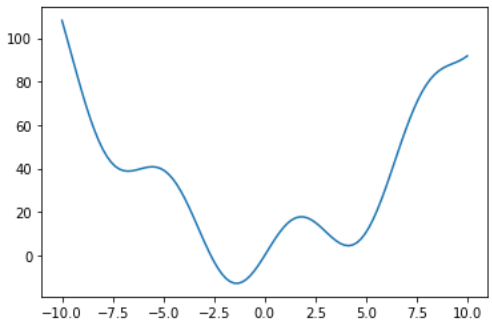
Prenons un exemple :

Créer une fonction f (composée d'un x^2 avec un sinus) qui donne une courbe présentant plusieurs minimums avec le code suivant :

```
def f(x):
    return x**2+15*np.sin(x)

x=np.linspace(-10,10,100)
plt.plot(x,f(x))
```

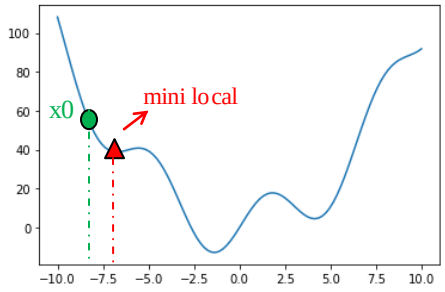
```
[<matplotlib.lines.Line2D at 0xa41b1b5190>]
```



- Méthodes pratiques sur SCIPY -

Appeler dans optimize la fonction minimize en entrant f et un point de départ x0, ici nous prendrons x0=-8

optimize.minimize execute un algorithme de minimisation, selon un point de depart **x0**
-> minimum LOCAL



Dans la fonction minimize, **l'algorithme (de type Newton, Lagrange)** va converger petit à petit vers un minimum depuis le point de départ.

Maintenant, tapons notre code puis exécutons pour lancer l'optimisation :

```
optimize.minimize(f,x0=-8)
```

```
      fun: 38.81120617595851
 hess_inv: array([[0.11637184]])
      jac: array([4.76837158e-07])
  message: 'Optimization terminated successfully.'
     nfev: 14
      nit: 6
     njev: 7
   status: 0
  success: True
        x: array([-6.73789948])
```

→ *dans toutes ces infos obtenues, seule le tableau x nous intéresse.*

x est la **coordonnée pour lequel l'algorithme a trouvé le minimum local.**

Comment faire pour avoir le mini global de la courbe ?

On peut choisir un autre point de départ. Ici avec x0=-5 on trouve x=-1.38 qui est bien le mioni global.

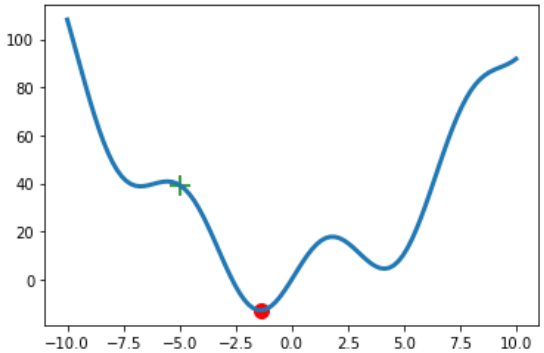
Comment récupérer la valeur de x pour afficher dans matplotlib les différents points sur la courbe ?

On affecte la ligne de code précédente à une variable result (par ex) et en forçant avec .x à la fin.

Puis avec quelques lignes de codes on va pouvoir afficher les points x0 et minimum local. Essayer le code ci-contre :

```
x0=-5
result = optimize.minimize(f,x0=x0).x

plt.plot(x,f(x),lw=3)
plt.scatter(result, f(result),s=100,c="r")
plt.scatter(x0, f(x0),s=200,marker="+",c="g")
plt.show()
```



5. Modules pour le traitement du signal

Dans Scipy nous avons plusieurs modules à disposition pour faire du traitement du signal. Les 2 plus intéressants sont : signal processing, discrete_fourier transformation, etc…

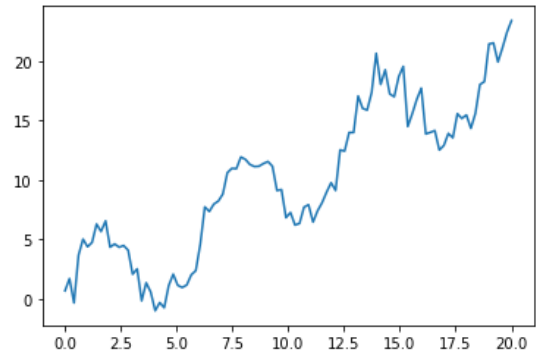
5.1. Module signal processing

Nous allons voir comment ce module permet **d'éliminer toute tendance linéaire** que l'on aurait **dans un signal**.

Entrer le code suivant permettant de tracer un signal quelconque composé de x et sin(x) avec du bruit :

```
import numpy as np
import matplotlib.pyplot as plt

x=np.linspace(0,20,100)
y=x+4*np.sin(x)+np.random.randn(x.shape[0])
plt.plot(x,y)
```



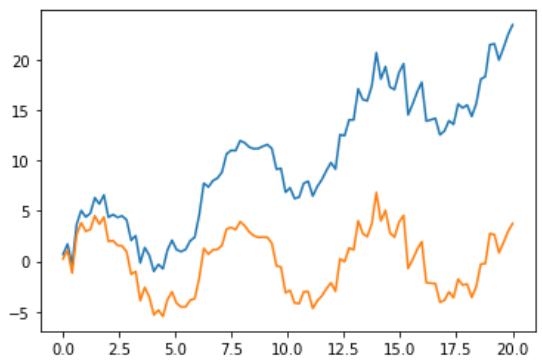
Cherchons à éliminer la tendance linéaire de notre signal.

Pour cela nous allons utiliser une fonction intéressante du module signal qui est la fonction ***detrend()***
Vous pouvez continuer ainsi votre code :

```
from scipy import signal
new_y = signal.detrend(y)

plt.plot(x,y)
plt.plot(x,new_y)
```

```
[<matplotlib.lines.Line2D at 0xa41d49cb20>]
```



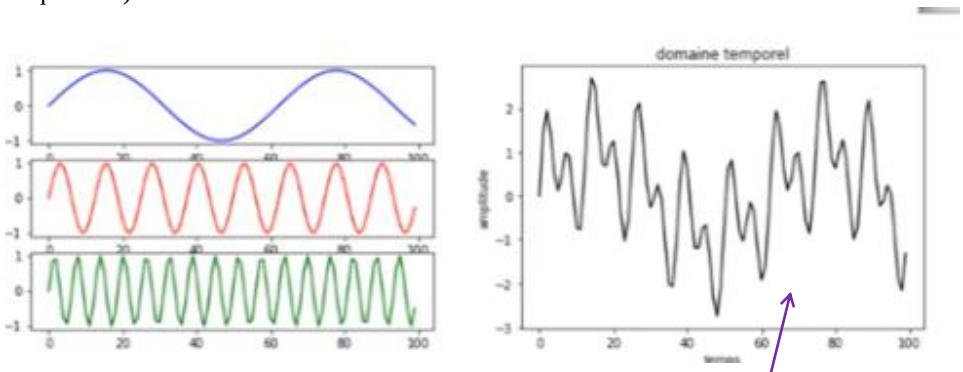
→ *On a bien éliminé la tendance linéaire*

- Méthodes pratiques sur SCIPY -

5.2. Module pour la transformation de Fourier

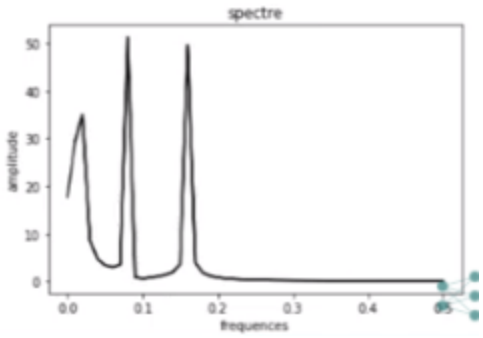
La **transformation de Fourier** est une technique mathématique (cf TP matplotlib fait début d'année), qui permet **d'extraire et d'analyser les fréquences qui sont présentes dans un signal périodique.**

Par exemple, ci-dessous, nous avons, à gauche, 3 signaux périodiques, chacun avec sa propre fréquence. Ces 3 signaux, si on les combine ensemble, ils vont nous donner le signal du graphique du milieu (cf sons, ondes…=combinaisons de fréquences).



La transformation de Fourier permet de prendre un signal comme le nôtre, et d'en **extraire les différentes fréquences qui le compose.** Le résultat donne un graphique appelé **spectre** dans lequel **en abscisse il y a les fréquences** et en **ordonnée les amplitudes des fréquences.**

Voilà ce que cela donne pour notre exemple :



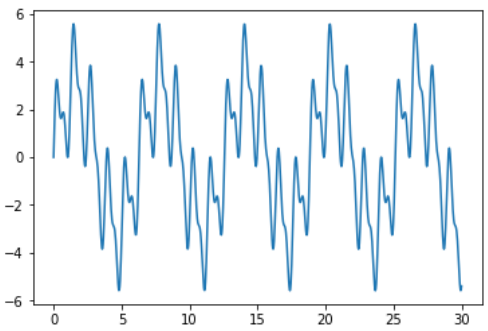
Comment générer un spectre de ce type avec Scipy ?

On va charger le **module fftpark** et utiliser 2 fonctions : *fft()* et *fftfreq()*

```
power = fftpack.fft(signal)
freq = fftpack.fftfreq(signal.size)
```

Commencer par générer le signal de départ avec matplotlib :

```
x=np.linspace(0,30,1000)
y=3*np.sin(x)+2*np.sin(5*x)+np.sin(10*x)
plt.plot(x,y)
```



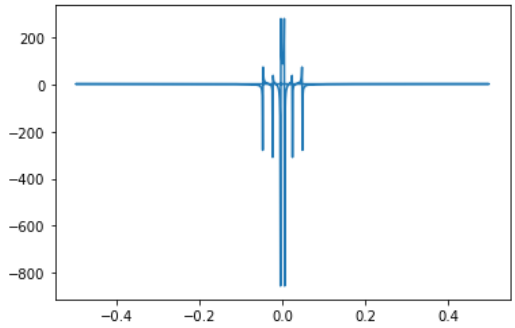
- Méthodes pratiques sur SCIPY -

On va d'abord importer le module fftpack puis créer une variable : fourier, de notre signal y et une autre : frequence, qui fera passer le nombre de données de notre signal y.

Puis on trace le spectre dans matplotlib et on s'apercoit qu'il y a des fréquences et amplitudes négatives…on va les effacer…par des **FILTRES.**

```
from scipy import fftpack
fourier=fftpack.fft(y)
frequence=fftpack.fftfreq(y.size)

plt.plot(frequence,fourier)
```

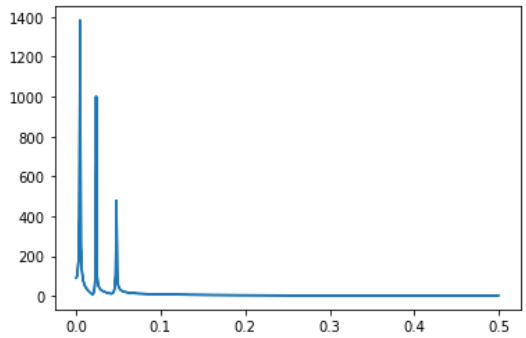


Comment filtrer ?

On va utiliser tout simplement la **valeur absolue** de nos variables…

```
from scipy import fftpack
fourier=fftpack.fft(y)
power=np.abs(fourier)
frequence=fftpack.fftfreq(y.size)

plt.plot(np.abs(frequence),power)
```



Mais que fait-on de ces spectres ?

- *reconnaissance vocale*
- *filtrage images et sons*

… etc

Un **exemple d'application à l'astrophysique…**

Comment faire pour connaître la composition chimique d'un astre lointain ? Chaque atome possède un rayonnement qui lui est particulier : les photons émis n'ont que certaines fréquences très particulières. On peut donc observer le spectre du signal lumineux de l'étoile et reconnaître les marques particulières de certains éléments chimiques, voir figure

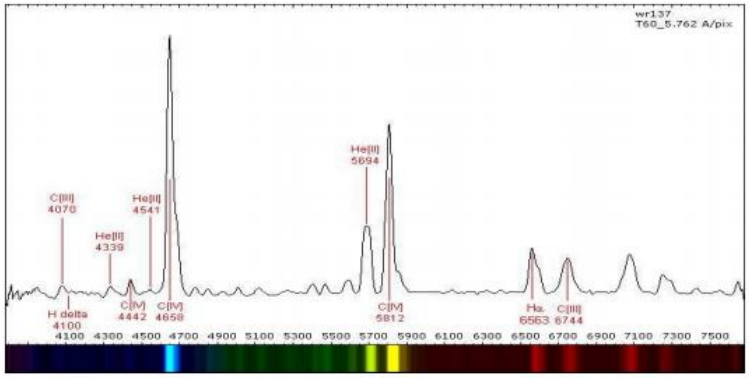
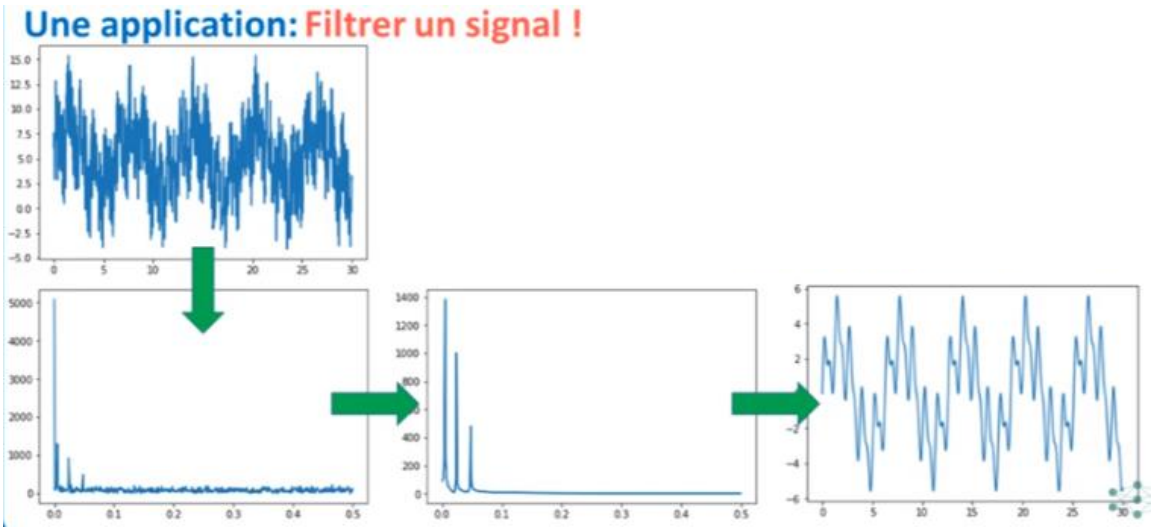


Fig. 1: le spectre de l'étoile Wolf-Rayet Wr137 réalisé au T-60 du Pic du Midi.

- Méthodes pratiques sur SCIPY -

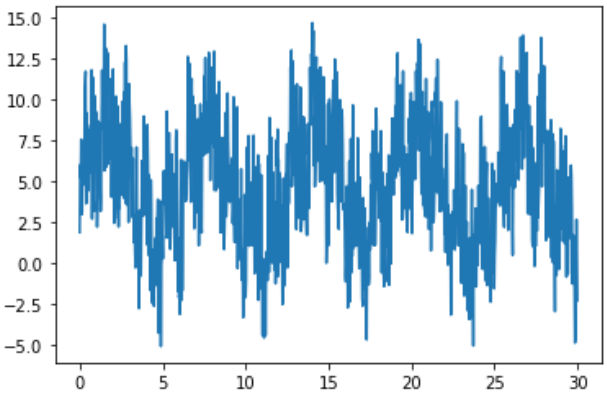
Nous allons ici voir comment avec scipy et fourier on peut filtrer un signal en 3 étapes.

Etape 1 : faire la transformation de fourier pour obtenir le spectre de notre signal
Etape 2 : filtrer toute les valeurs inférieures à un certain seuil via du Booléan Indexing (cf cours Numpy)
Etape 3 : sur le spectre nettoyé, appliquer la transformation inverse de Fourier.

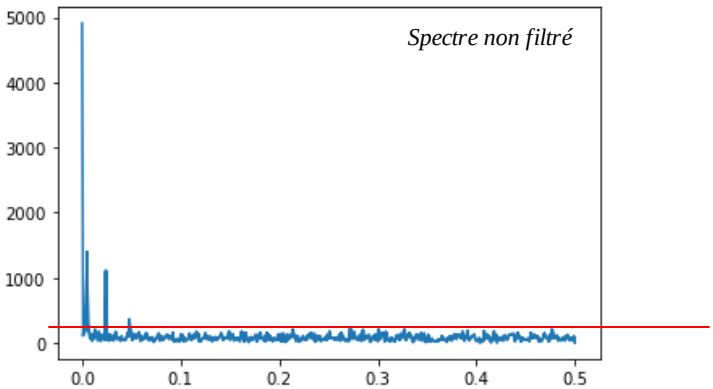


Allons-y, commencer par créer le signal bruité de départ sous python :

```
x=np.linspace(0,30,1000)
y=3*np.sin(x)+2*np.sin(5*x)+np.sin(10*x)+np.random.random(x.shape[0])*10
plt.plot(x,y)
```



On passe comme page précédente par Fourier pour avoir le spectre :



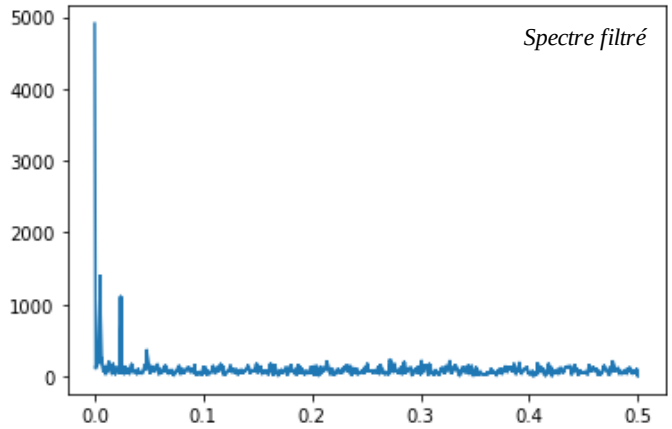

```
fourier=fftpack.fft(y)
power=np.abs(fourier)
frequence=fftpack.fftfreq(y.size)

plt.plot(np.abs(frequence),power)
```

- Méthodes pratiques sur SCIPY -

On va maintenant filtrer toutes les valeurs inférieures à un seuil choisi, ici on peut prendre 400 par une technique de booléan indexing comme suit :

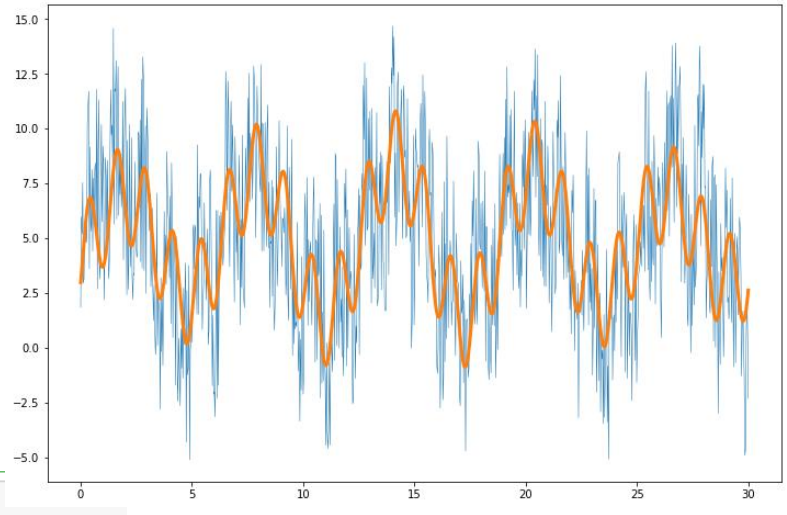
```
fourier[power<400]=0
plt.plot(np.abs(frequence),power)
```



Il nous reste à appliquer la transformée de Fourier inverse pour obtenir notre signal filtré. Pour cela dans le module fftpack, il y a la fonction *ifft()* pour faire l'inversion et avec un peu de matplotlib on va obtenir notre signal filtré…

Voici le code :

```
filtered_signal=fftpack.ifft(fourier)
plt.figure(figsize=(12,8))
plt.plot(x,y,lw=0.5)
plt.plot(x,filtered_signal,lw=3)
plt.show()
```



Bien sûr, la **variable filtered_signal** est **un tableau Numpy que l'on peut récupérer…**

```
print(filtered_signal)
```

```
[ 2.97653577+0.00000000e+00j  3.34831547-2.27373675e-16j
  3.73317041+0.00000000e+00j  4.12355545-2.27373675e-16j
  4.51182537+0.00000000e+00j  4.89040695+1.13686838e-16j
  5.25196961+3.41060513e-16j  5.58959029-2.84217094e-16j
  5.89690926+1.70530257e-16j  6.16827299+5.68434189e-17j
  6.39886099-5.68434189e-17j  6.58479381-2.27373675e-16j
  6.72321953+2.27373675e-16j  6.81237704+1.70530257e-16j
  6.85163445+4.54747351e-16j  6.84150192+0.00000000e+00j
  6.78361854-3.41060513e-16j  6.68071355-4.54747351e-16j
  6.53654288+2.27373675e-16j  6.35580227-2.27373675e-16j
```

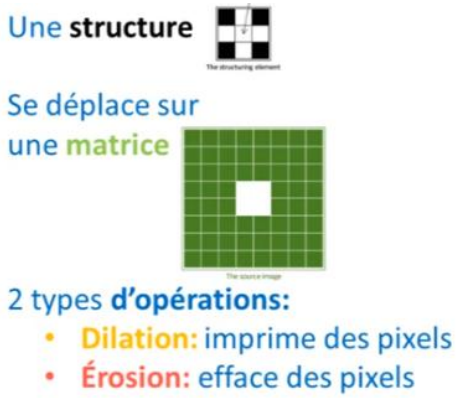
6. Modules pour le traitement d'images

Dans Scipy, un module très intéressant permet de faire du traitement d'image : *scipy.ndimage*.Dans ce module, il existe beaucoup de fonctions pour traiter des images, nous allons voir les plus intéressants.

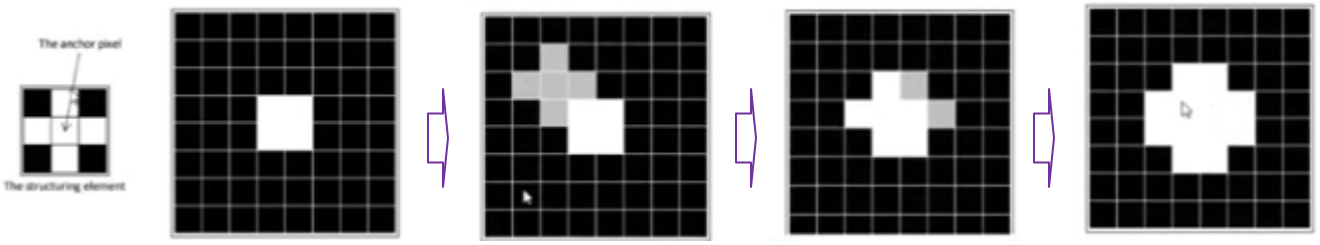
6.1. Scipy – Morphologie

C'est une technique mathématique qui permet de **transformer une matrice et donc des images**. Le principe est simple, on défini une structure (qui ressemble le plus souvent à une croix) et qui va se déplacer de pixel en pixel sur toute notre image.

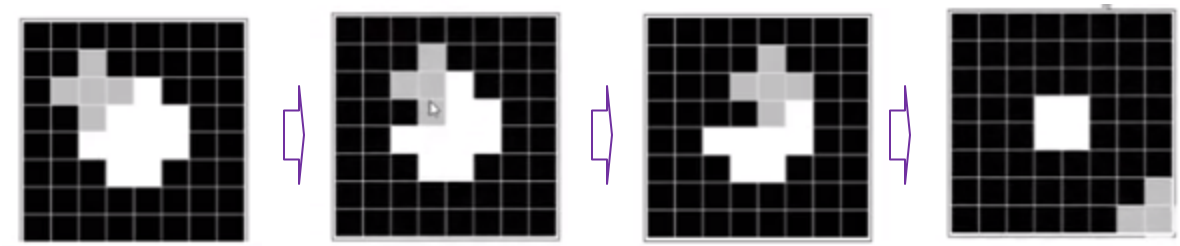
Lorsqu'elle va rencontrer un pixel blanc, cette structure va effectuer une opération : elle va soit imprimer tout autour d'elle des pixels blancs selon son motif (croix, L, U…) = **DILATION**, soit effacer des pixels = **EROSION**



Exemple de dilation :



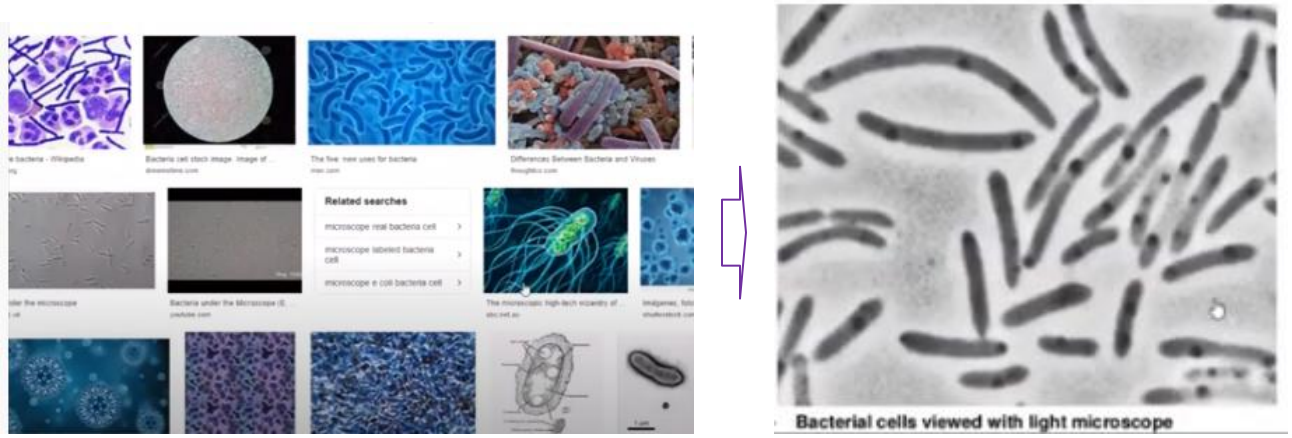
Exemple d'érosion :



- Méthodes pratiques sur SCIPY -

Utilisation pour nettoyer les arctéfacts (petits défauts dans une image) :

Avec cette technique, nous allons montrer comment enlever les impuretés sur une image prise dans Google images. On choisi cete image de bactéries.

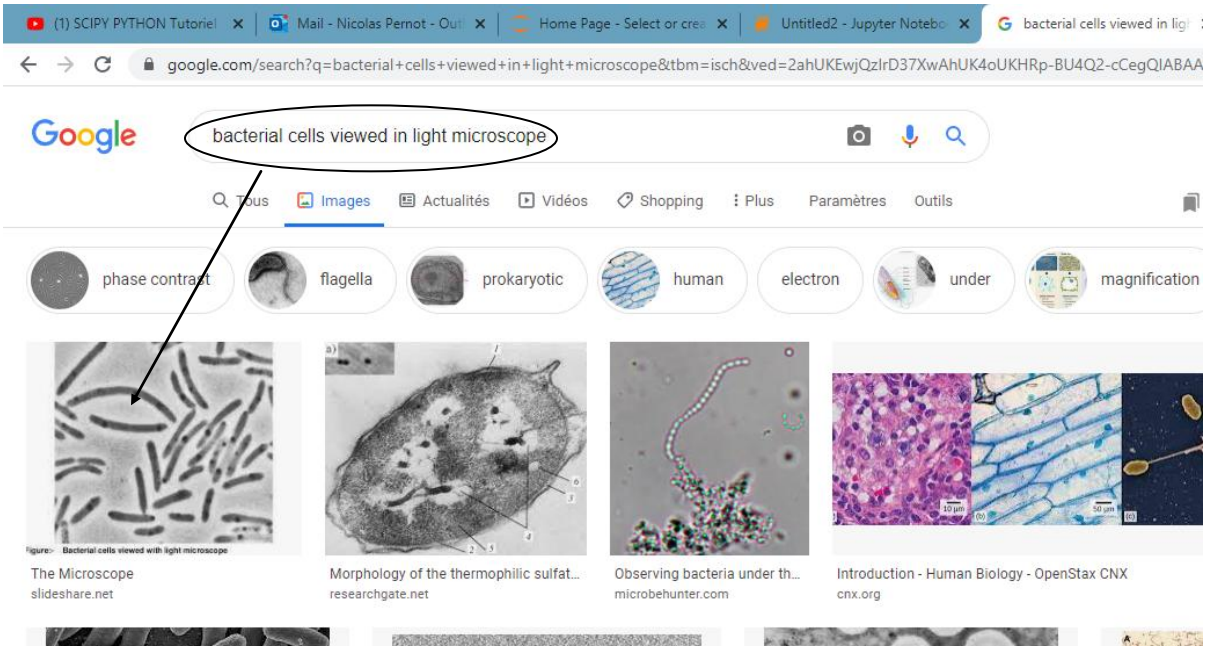


On va la télécharger et on va essayer de faire 2 choses :

1) *Extraire les bactéries de l'arrière plan de la photo*
2) *Segmenter l'image pour mettre une étiquette sur chaque bactérie, puis mesurer la taille de chacune d'elles…*

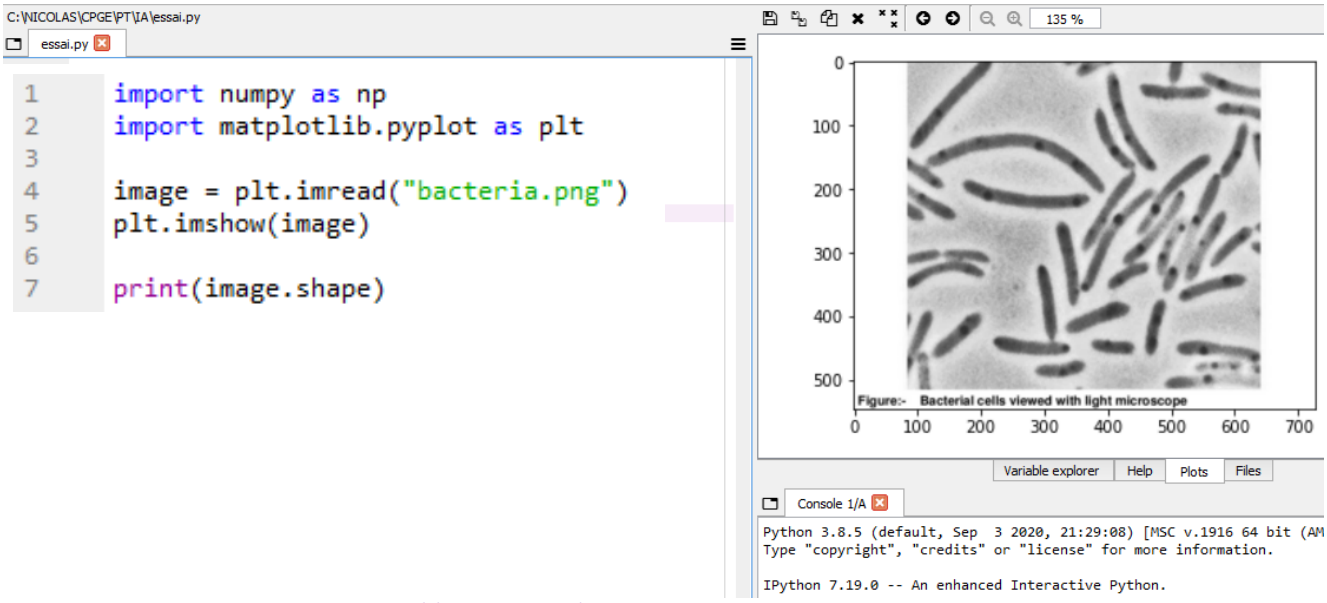
On présentera tous ces résultats dans un graphique matplotlib.

Avant tout chose, télécharger cette photo de google dans votre répertoire de travail, renommer la « bacteria » puis l'enregistrer en format png puis on va l'importer dans Jupiter ou idle.



- Méthodes pratiques sur SCIPY -

Voici le code en utilisant Spyder pour changer ☺



*L'image est un tableau Numpy de 4 dimensions avec 546*728 pixels*

Avec un petit subsetting, on va réduire l'image à 2 dimensions en niveaux de gris

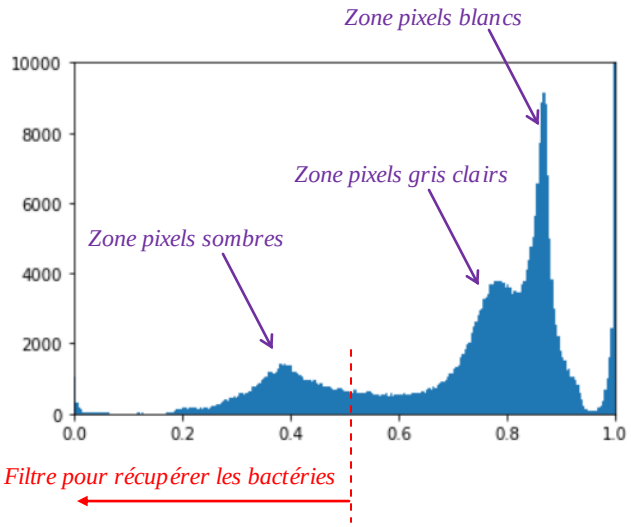
```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  image = plt.imread("bacteria.png")
5  image = image[:,:,0]
6  plt.imshow(image,cmap="gray")
7
8  print(image.shape)
```

Etape 1 : on va extraire les bactéries de l'arrière plan

On va simplement utiliser du **booléan indexing.** On va faire une copie de l'image puis applatir le tableau 2d de l'image en un tablea 1d avec la fonction souvenez vous : *ravel()* pour en faire un histogramme à 255 catégories (bins).

Tester le code suivant :

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  image = plt.imread("bacteria.png")
5  image = image[:,:,0]
6
7  image_copy = np.copy(image)
8  plt.hist(image_copy.ravel(),bins=255)
9  plt.axis([0,1,0,10000])
10 plt.show()
```



On retrouve bien sur la photo les 3 zones de couleurs.

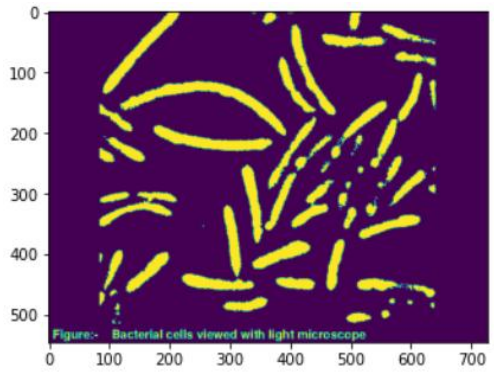
- Méthodes pratiques sur SCIPY -

Nous allons extraire les pixels les plus sombres pour récupérer les bactéries.
Nous allons encore utiliser du booléan indexing pour filtrer toutes les valeurs inférieures à environ 0.5 sur l'histogramme.

Voici le code que vous pouvez tester :

```
11
12    image = image <0.5
13    plt.imshow(image)
```

Et voilà l'image obtenue.



Etape 2 : segmentation de l'image pour étiquetter les bactéries

Dans le module ndimage, nous allons utiliser la fonction *label()*. Elle va nous retourner 2 variables : label_image qui sera **l'image étiquettée** et n_labels qui est le **nombre d'étiquettes** placées sur cette image.

Tester le code suivant :

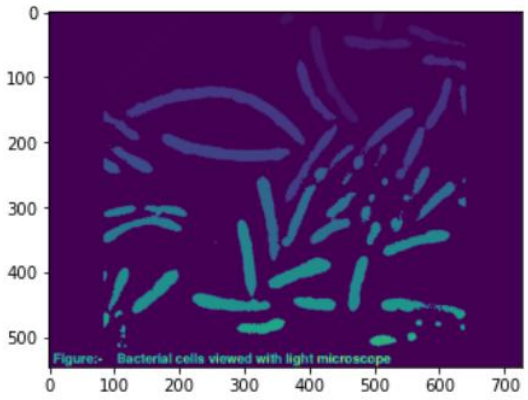
```
16    from scipy import ndimage
17
18    label_image, n_labels = ndimage.label(image)
19    print(n_labels)
```

```
In [5]: runfile(
156
```

*Dans notre image il y a **156 petits groupes créés***

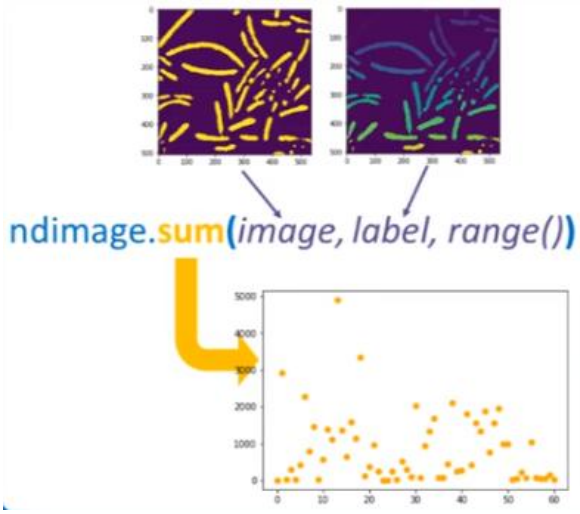
Et pour visualiser ces groupes, on peut taper :

```
21
22    plt.imshow(label_image)
```



- Méthodes pratiques sur SCIPY -

Enfin, on va s'amuser à mesurer la taille (en pixel) de chacun des groupes créés. On va utiliser la fonction *sum()* de ndimage.



Tester le code suivant :

```
24  sizes = ndimage.sum(image,label_image, range(n_labels))
25  print(sizes)
```

On obtient un tableau Numpy qui contient 153 éléments avec leur taille en pixel respectives…

```
In [7]: runfile('C:/NICOLAS/CPGE/PT/IA/essai.py', wdir='C:/NICOLAS/CPGE/P
156
[0.000e+00 2.953e+03 2.900e+01 1.000e+00 1.000e+00 3.300e+01 4.470e+02
 3.040e+02 1.000e+00 1.000e+00 2.284e+03 7.890e+02 1.480e+03 6.300e+01
 3.000e+00 2.000e+00 2.000e+00 1.000e+00 2.100e+01 5.000e+00 5.460e+02
 1.720e+02 1.386e+03 2.000e+00 1.116e+03 3.070e+02 4.890e+03 1.367e+03
 7.550e+02 1.594e+03 1.132e+03 3.349e+03 3.000e+00 1.420e+02 3.750e+02
 3.000e+00 4.000e+00 9.540e+02 6.000e+00 5.000e+00 2.000e+00 2.540e+02
 3.750e+02 8.000e+00 1.000e+00 1.000e+00 1.300e+01 5.170e+02 3.030e+02
 3.000e+00 1.030e+02 3.000e+00 2.026e+03 8.200e+01 2.300e+01 4.000e+00
 9.570e+02 1.341e+03 1.000e+00 1.675e+03 8.000e+01 4.000e+00 7.600e+01
 1.180e+02 5.090e+02 2.229e+03 2.570e+02 7.000e+00 2.000e+00 2.730e+02
 1.000e+00 1.797e+03 4.190e+02 2.000e+00 1.553e+03 5.000e+00 1.324e+03
 1.893e+03 7.710e+02 1.559e+03 5.700e+01 1.946e+03 1.000e+03 9.910e+02
 6.000e+00 2.000e+00 1.000e+00 1.500e+01 4.160e+02 3.000e+00 5.000e+00
 7.200e+01 1.460e+02 1.044e+03 6.600e+01 4.800e+01 4.500e+01 6.100e+01
 1.400e+02 2.200e+01 4.800e+02 2.000e+00 2.300e+01 1.000e+01 9.300e+01
 9.000e+00 1.250e+02 8.000e+00 4.500e+01 4.500e+01 4.500e+01 8.000e+00
 8.900e+01 8.000e+00 8.500e+01 4.400e+01 7.000e+00 8.400e+01 8.000e+00
 4.900e+01 4.900e+01 4.900e+01 3.300e+01 1.020e+02 7.300e+01 4.200e+01
 6.700e+01 8.000e+00 7.100e+01 5.900e+01 6.700e+01 4.300e+01 3.300e+01
```

Voilà un tableau intéressant, on peut imaginer maintenant pouvoir faire des statistiques, calculer une moyenne, écart type…etc…

Aller, il y a tant d'autres fonctions intéressantes, n'hésitez pas à consulter la doc Scipy si besoin …

- Méthodes pratiques sur SCIPY -