# Accélération du calcul de la désorientation cristalline

Juillet 2024

Encadrant : Henry Proudhon. Participants : Esteban DAUDE, Valentin DEUMIER, Sioban NIERADZIK-KOZIC, Nathan RAPIN

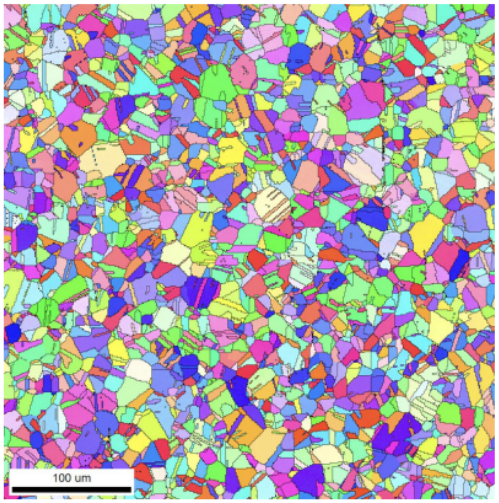
# Sommaire

# Introduction : présentation du sujet

CENTRE DES MATERIAUX
P.M FOURT

PYMICRO

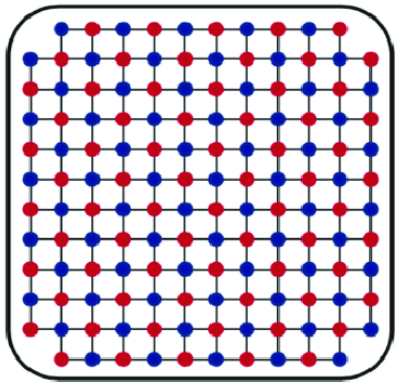
Pymicro : librairie Python



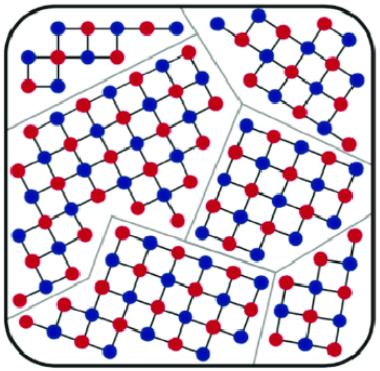

Images par diffraction d'électrons rétrodiffusés d'un échantillon d'acier forgé.

# Sommaire

# Orientation et désorientation cristalline

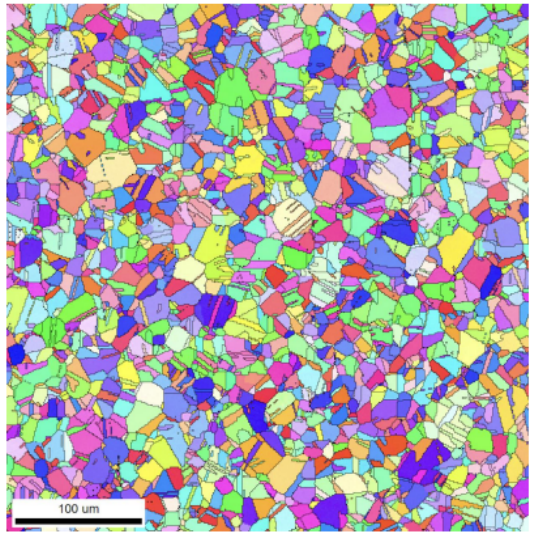


Crystalline



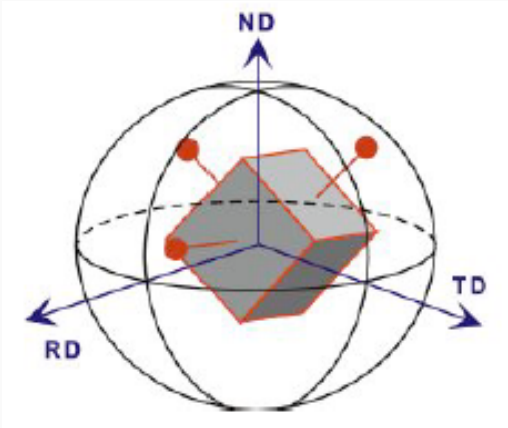
Polycrystalline

Aube de turbine monocristalline



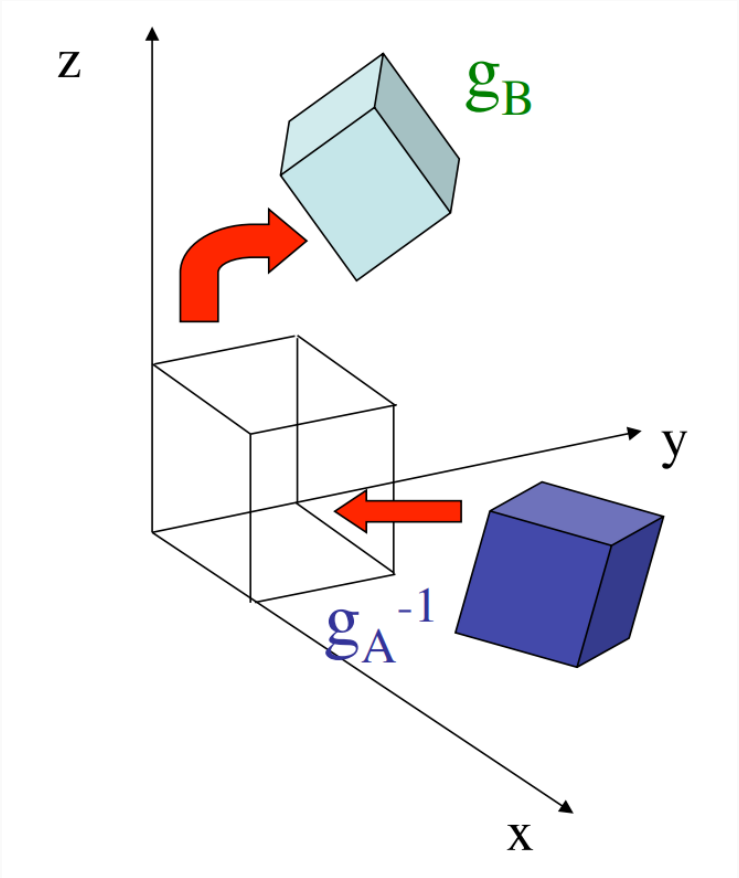

Images par diffraction d'électrons rétrodiffusés d'un échantillon d'acier forgé.

# Orientation et désorientation cristalline




Orientation d'une cristal par rapport au référentiel de l'échantillon

Description de l'orientation de grains




Désorientation cristalline : différence d'orientation entre deux cristaux ou grains adjacents dans un matériau polycristallin

# Importance de l'orientation cristalline

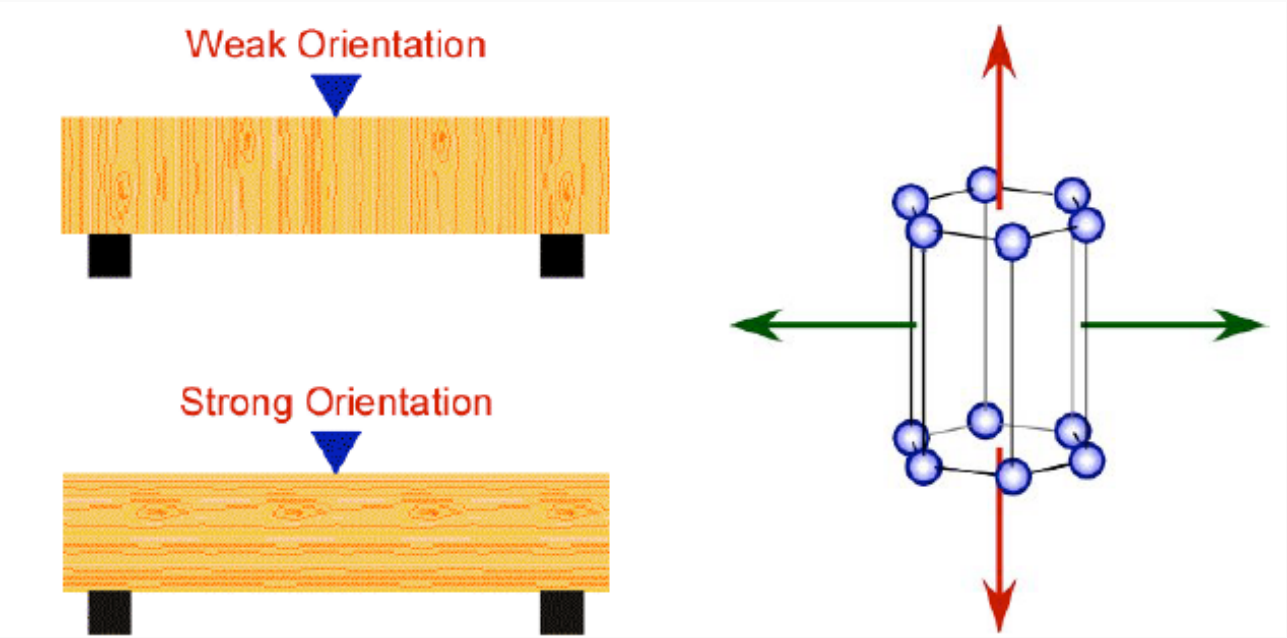
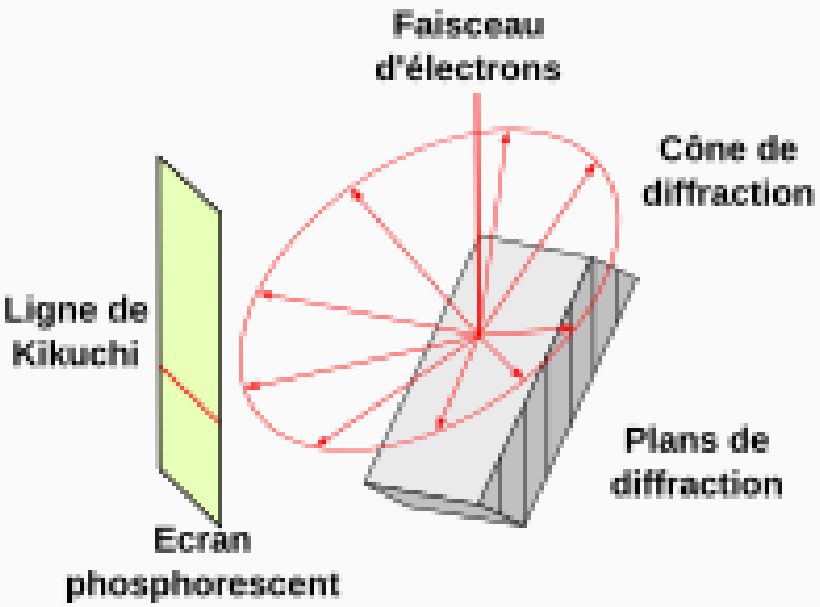
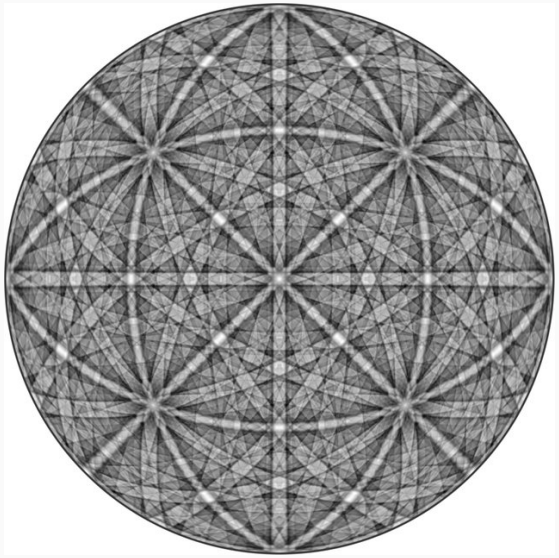



Illustration de l'anisotropie

# EBSD: Mesurer l'orientation

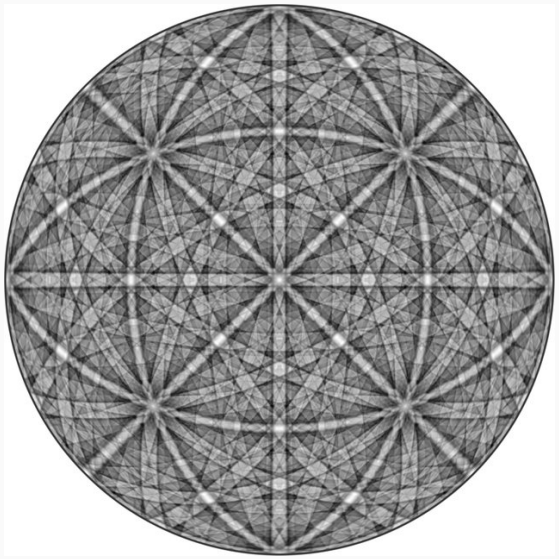



EBSD Setup

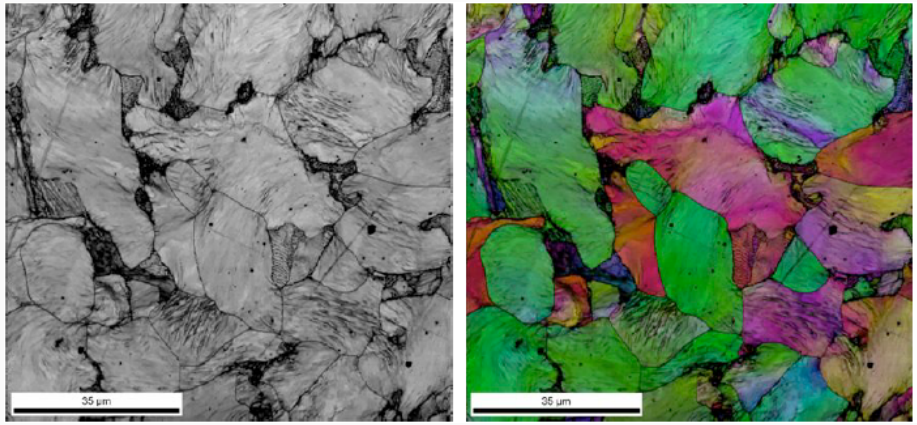


Lignes de Kikuchi

# EBSD: Mesurer l'orientation

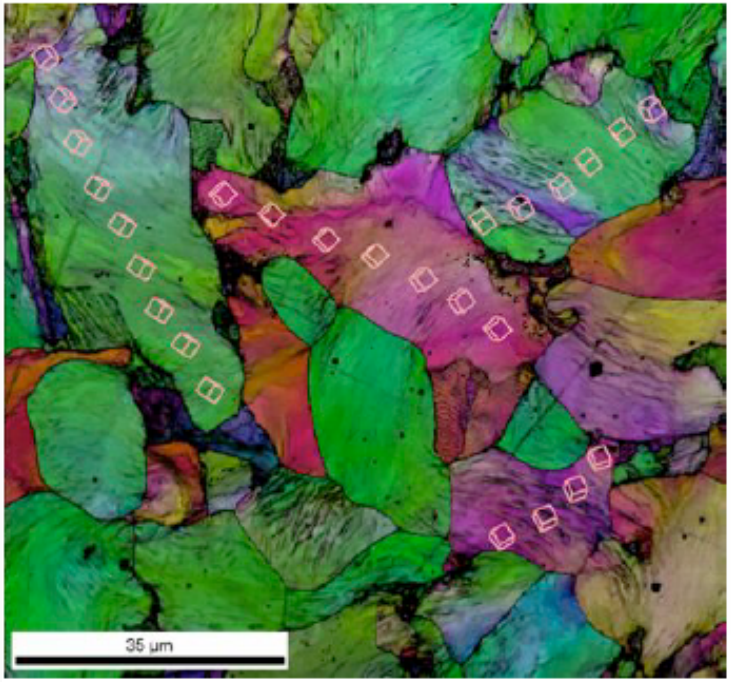


Lignes de Kikuchi

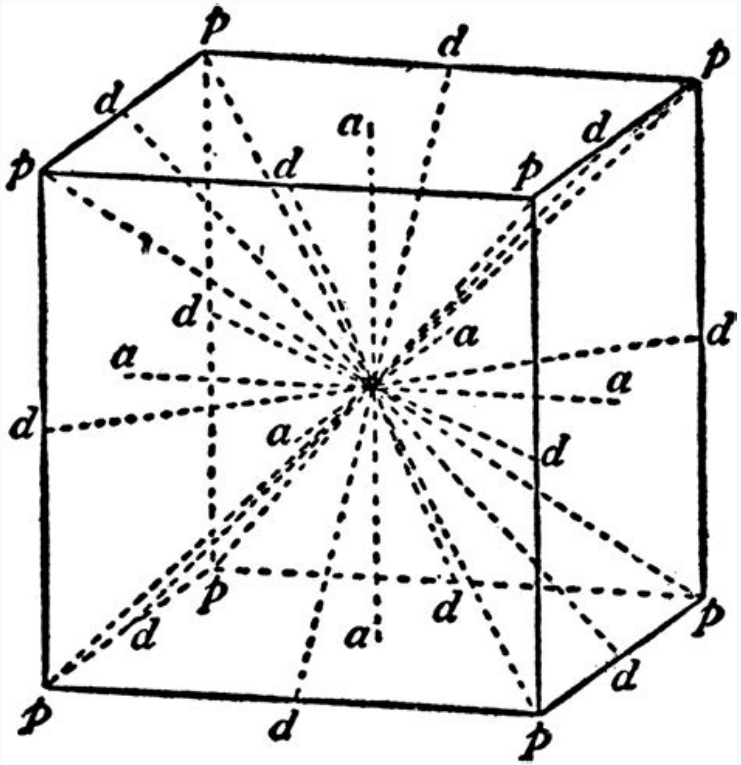



EBSD mapping of misorientation in ferritic steel

# Segmentation des grains

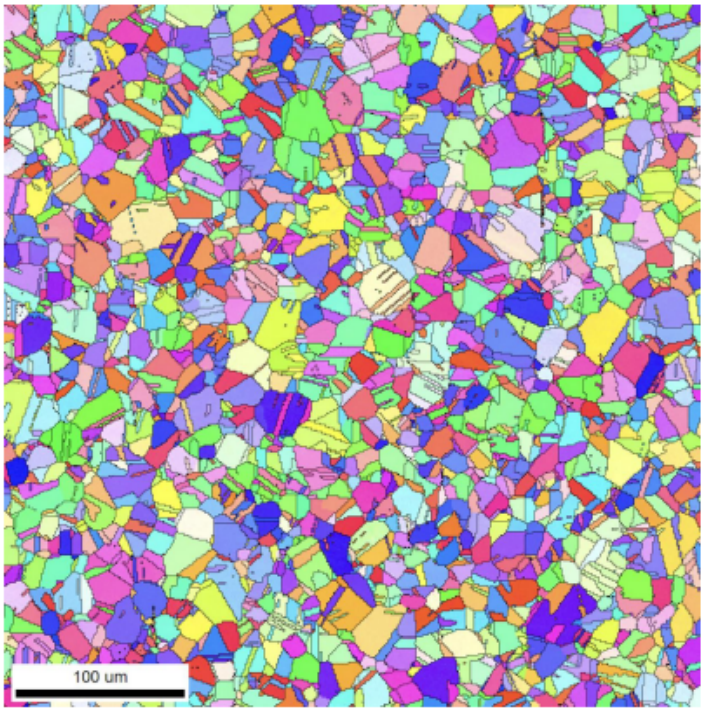



Gradient d'orientation



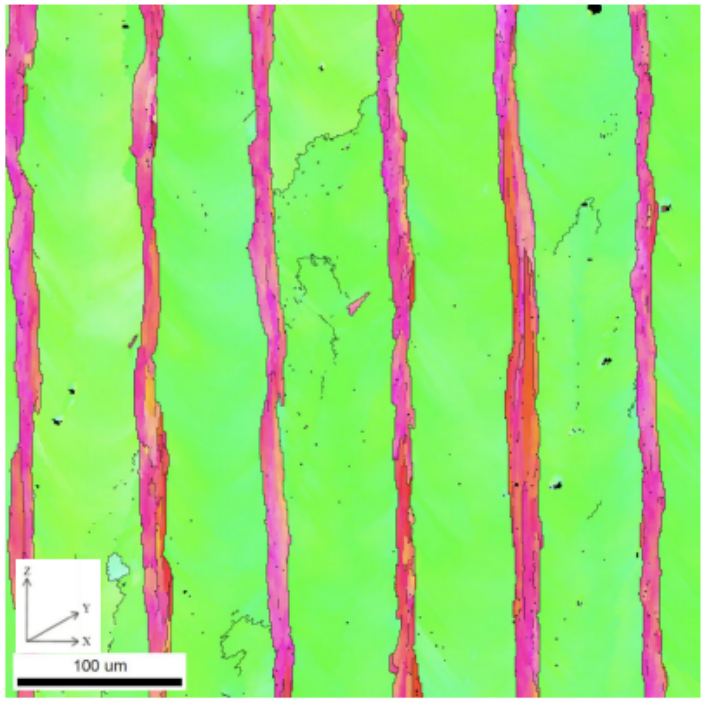

Symétries du cube

# Exemple: Dans l'industrie




Images par diffraction d'électrons rétrodiffusés d'un échantillon d'acier forgé.

EBSD Acier forgé

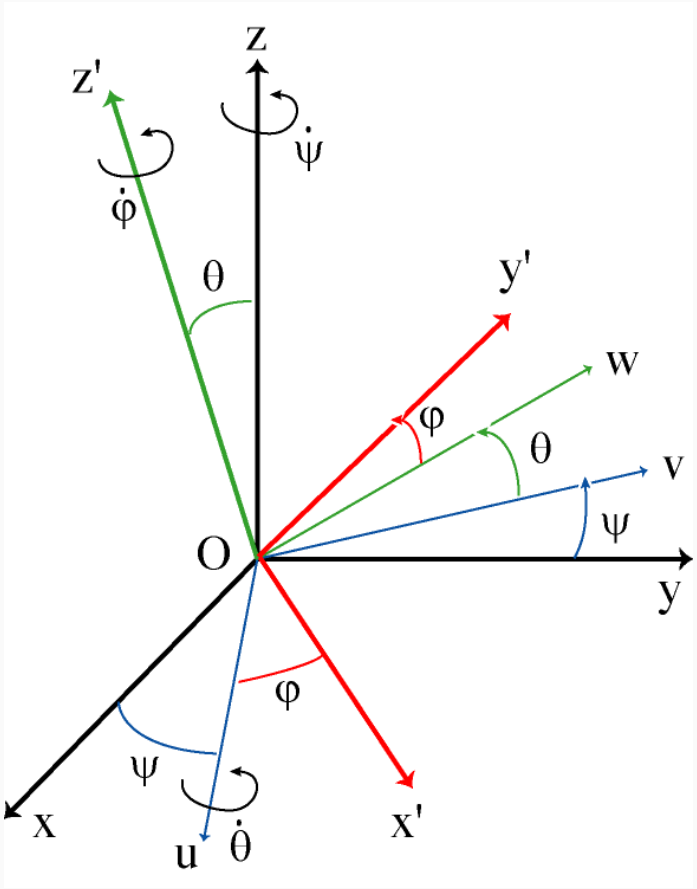



Images par diffraction d'électrons rétrodiffusés d'un échantillon d'acier en fabrication additive par LPB-F.

EBSD Acier fabrication additive LPB-F

# Sommaire

# Les angles d'Euler




Succession de trois rotations :

$$Oxyz \underset{\psi}{\rightarrow} Ouvz$$

$$Ouvz \underset{\theta}{\rightarrow} Ouwz'$$

$$Ouwz' \underset{\phi}{\rightarrow} Ox'y'z'$$

**Remarque**

$(\psi, \theta, \phi) \in [0; 2\pi[\times[0; \pi[\times[0; 2\pi[$.

**Attention**

La rotation est passive par convention, la rotation active correspondante est donc $(-\phi, -\theta, -\psi)$.

# Matrices de rotation

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = A \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} \; et \; \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = A^T \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

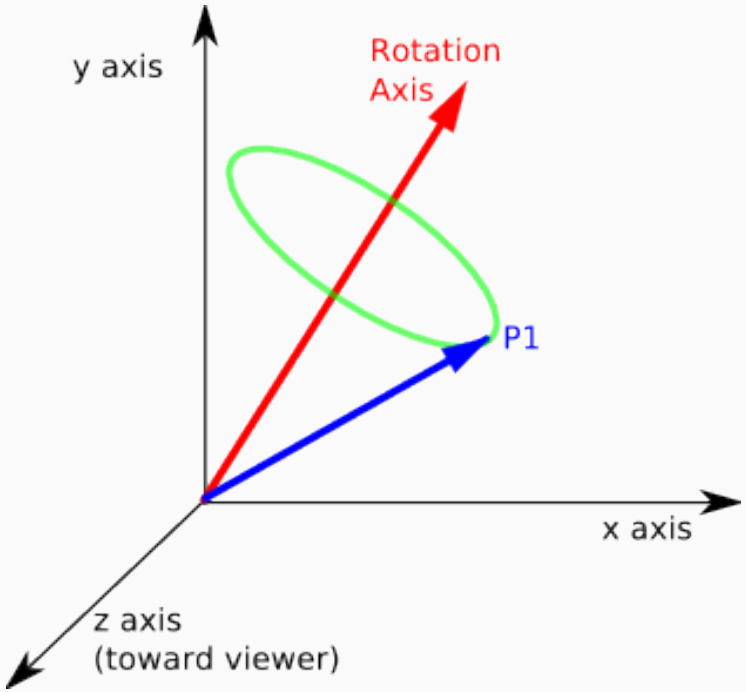
avec

$$A = \begin{pmatrix} \cos(\psi)\cos(\phi) - \sin(\psi)\cos(\theta)\sin(\phi) & -\cos(\psi)\sin(\phi) - \sin(\psi)\cos(\theta)\cos(\phi) & \sin(\psi)\sin(\theta) \\ \sin(\psi)\cos(\phi) + \cos(\psi)\cos(\theta)\sin(\phi) & -\sin(\psi)\sin(\phi) + \cos(\psi)\cos(\theta)\cos(\phi) & -\cos(\psi)\sin(\theta) \\ \sin(\theta)\sin(\phi) & \sin(\theta)\cos(\phi) & \cos(\theta) \end{pmatrix}$$

,

$$A \in \mathcal{SO}_3(\mathbb{R})$$

# Représentation axe-angle




Rotation caractérisée par un couple $(\mathbf{n}, \omega) \in \mathbb{R}^3 \times [0; \pi]$ avec $\mathbf{n}$ unitaire.

**Convention**
Une rotation donnée par $(\mathbf{n}, \omega)$ avec $\omega \in ]\pi; 2\pi[$ est donc représentée en réalité par $(-\mathbf{n}, 2\pi - \omega)$.

# Les quaternions

**Définition**
L'ensemble $\mathbb{H}$ des quaternions est une $\mathbb{R}$-algèbre de dimension 4 contenant $\mathbb{C}$. On note généralement sa base canonique $(1, i, j, k)$. On a comme relations fondamentales $i^2 = j^2 = k^2 = ijk = -1$.

**Remarque**
Le produit n'est pas commutatif sur $\mathbb{H}$, par exemple on a $ij = -ji = k$.

**Défintion**
Le conjugué $q^*$ d'un quaternion $q = q_0 + q_1 i + q_3 j + q_3 k$ est défini par $q^* = q_0 - q_1 i - q_2 j - q_3 k$.

# Utilisation des quaternions

**Définition**
La norme d'un quaternion $q = q_0 + q_1 i + q_2 j + q_3 k$ est donnée par $\|q\| = \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2}$. Si $\|q\| = 1$, on dit que $q$ est unitaire.

**Propriété**
Si $q$ est unitaire, $q = \cos(\frac{\omega}{2}) + \sin(\frac{\omega}{2})(n_1 i + n_2 j + n_3 k)$ où $\mathbf{n} = (n_1, n_2, n_3)$ est un vecteur unitaire. On retrouve la représentation axe-angle.

# Lien entre quaternions et représentation axe-angle

**Remarque**
La convention $\omega \in [0; \pi]$ légitime de prendre la partie réelle positive, $q$ et $-q$ représentant la même rotation. On définit alors $\mathcal{H}_+ = \{q \in \mathbb{H} \mid \|q\| = 1, q_0 \geq 0\}$.

**Propriété**
$(\mathbf{n}, \omega) \in S_1 \times [0; \pi] \mapsto \cos(\frac{\omega}{2}) + \sin(\frac{\omega}{2})(n_1 i + n_2 j + n_3 k) \in \mathcal{H}_+$ est un isomorphisme de groupe.

# Intérêt des quaternions

Loi de composition pour la représentation axe-angle :

$(\mathbf{u}, \omega) \circ (\mathbf{v}, \phi) = (\mathbf{n}, \theta)$ avec :

$$n_1 = \frac{\cos(\frac{\phi}{2})\sin(\frac{\omega}{2})v_1 + \cos(\frac{\omega}{2})\sin(\frac{\phi}{2})u_1 + \sin(\frac{\phi}{2})\sin(\frac{\omega}{2})(\mathbf{u} \times \mathbf{v})_1}{\sqrt{1 - (\cos(\frac{\phi}{2})\cos(\frac{\omega}{2}) - \sin(\frac{\phi}{2})\sin(\frac{\omega}{2})\mathbf{u} \cdot \mathbf{v})^2}}$$

$$n_2 = \frac{\cos(\frac{\phi}{2})\sin(\frac{\omega}{2})v_2 + \cos(\frac{\omega}{2})\sin(\frac{\phi}{2})u_2 + \sin(\frac{\phi}{2})\sin(\frac{\omega}{2})(\mathbf{u} \times \mathbf{v})_2}{\sqrt{1 - (\cos(\frac{\phi}{2})\cos(\frac{\omega}{2}) - \sin(\frac{\phi}{2})\sin(\frac{\omega}{2})\mathbf{u} \cdot \mathbf{v})^2}}$$

$$n_3 = \frac{\cos(\frac{\phi}{2})\sin(\frac{\omega}{2})v_3 + \cos(\frac{\omega}{2})\sin(\frac{\phi}{2})u_3 + \sin(\frac{\phi}{2})\sin(\frac{\omega}{2})(\mathbf{u} \times \mathbf{v})_3}{\sqrt{1 - (\cos(\frac{\phi}{2})\cos(\frac{\omega}{2}) - \sin(\frac{\phi}{2})\sin(\frac{\omega}{2})\mathbf{u} \cdot \mathbf{v})^2}}$$

$$\theta = 2\arccos(\cos(\tfrac{\phi}{2})\cos(\tfrac{\omega}{2}) - \sin(\tfrac{\phi}{2})\sin(\tfrac{\omega}{2})\mathbf{u} \cdot \mathbf{v})$$

# Sommaire

Calculs EBSD : classe **OimScan**

→ **OimScan** utilise la fonction **segment grains**, qui identifie les différents grains cristallographique selon la désorientation entre eux

→ **segment grains** a besoin de calculer les désorientations entre tout les points de la cartographie, tenant compte des symétries : elle appelle la fonction **disorientation** de la classe **Orientation** C'est la fonction que l'on va chercher à optimiser

# Code d'origine

```python
class Orientation:
    def disorientation(self, orientation, crystal_structure=Symmetry.triclinic):
        the_angle = np.pi
        the_axis = np.array([0., 0., 1.])
        the_axis_xyz = np.array([0., 0., 1.])
        symmetries = crystal_structure.symmetry_operators()
        (gA, gB) = (self.orientation_matrix(), orientation.orientation_matrix())  # nicknames
        for (g1, g2) in [(gA, gB), (gB, gA)]:
            for j in range(symmetries.shape[0]):
                sym_j = symmetries[j]
                oj = np.dot(sym_j, g1)  # the crystal symmetry operator is left applied
                for i in range(symmetries.shape[0]):
                    sym_i = symmetries[i]
                    oi = np.dot(sym_i, g2)
                    delta = np.dot(oi, oj.T)
                    mis_angle = Orientation.misorientation_angle_from_delta(delta)
                    if mis_angle < the_angle:
                        # now compute the misorientation axis, should check if it lies in the fundamental zone
                        mis_axis = Orientation.misorientation_axis_from_delta(delta)
                        # here we have np.dot(oi.T, mis_axis) = np.dot(oj.T, mis_axis)
                        # print(mis_axis, mis_angle*180/np.pi, np.dot(oj.T, mis_axis))
                        the_angle = mis_angle
                        the_axis = mis_axis
                        the_axis_xyz = np.dot(oi.T, the_axis)
        return the_angle, the_axis, the_axis_xyz
```

Fonction disorientation avec matrices

# Code d'origine

```python
def symmetry_operators(self, use_miller_bravais=False):
    """Define the equivalent crystal symmetries.

    Those come from Randle & Engler, 2000. For instance in the cubic
    crystal struture, for instance there are 24 equivalent cube orientations.

    :return array: A numpy array of shape (n, 3, 3) where n is the \
    number of symmetries of the given crystal structure.
    """
    if self is Symmetry.cubic:
        sym = np.zeros((24, 3, 3), dtype=float)
        sym[0] = np.array([[1., 0., 0.], [0., 1., 0.], [0., 0., 1.]])
        sym[1] = np.array([[0., 0., -1.], [0., -1., 0.], [-1., 0., 0.]])
        sym[2] = np.array([[0., 0., -1.], [0., 1., 0.], [1., 0., 0.]])
        sym[3] = np.array([[-1., 0., 0.], [0., 1., 0.], [0., 0., -1.]])
        sym[4] = np.array([[0., 0., 1.], [0., 1., 0.], [-1., 0., 0.]])
        sym[5] = np.array([[1., 0., 0.], [0., 0., -1.], [0., 1., 0.]])
        sym[6] = np.array([[1., 0., 0.], [0., -1., 0.], [0., 0., -1.]])
        sym[7] = np.array([[1., 0., 0.], [0., 0., 1.], [0., -1., 0.]])
        sym[8] = np.array([[0., -1., 0.], [1., 0., 0.], [0., 0., 1.]])
        sym[9] = np.array([[-1., 0., 0.], [0., -1., 0.], [0., 0., 1.]])
        sym[10] = np.array([[0., 1., 0.], [-1., 0., 0.], [0., 0., 1.]])
        sym[11] = np.array([[0., 0., 1.], [1., 0., 0.], [0., 1., 0.]])
        sym[12] = np.array([[0., 1., 0.], [0., 0., 1.], [1., 0., 0.]])
        sym[13] = np.array([[0., 0., -1.], [-1., 0., 0.], [0., 1., 0.]])
        sym[14] = np.array([[0., -1., 0.], [0., 0., 1.], [-1., 0., 0.]])
        sym[15] = np.array([[0., 1., 0.], [0., 0., -1.], [-1., 0., 0.]])
        sym[16] = np.array([[0., 0., -1.], [1., 0., 0.], [0., -1., 0.]])
        sym[17] = np.array([[0., 0., 1.], [-1., 0., 0.], [0., -1., 0.]])
        sym[18] = np.array([[0., -1., 0.], [0., 0., -1.], [1., 0., 0.]])
        sym[19] = np.array([[0., 1., 0.], [1., 0., 0.], [0., 0., -1.]])
        sym[20] = np.array([[-1., 0., 0.], [0., 0., 1.], [0., 1., 0.]])
        sym[21] = np.array([[0., 0., 1.], [0., -1., 0.], [1., 0., 0.]])
        sym[22] = np.array([[0., -1., 0.], [-1., 0., 0.], [0., 0., -1.]])
        sym[23] = np.array([[-1., 0., 0.], [0., 0., -1.], [0., -1., 0.]])
```

Exemple des opérateurs de symétrie dans le cas cubique

# Intérêt des quaternions

Le calcul matriciel est lourd informatiquement : le produit matriciel nécessite beaucoup d'opérations

Le passage en quaternion permet de réduire le calcul au produit de quaternion et l'inversion à la conjugaison du quaternion

```
#Product of to 'normal' quaternions
def Q_product(q1, q2) :
        r0 = q1[0]*q2[0] - q1[1]*q2[1] - q1[2]*q2[2] - q1[3]*q2[3]
        r1 = q1[0]*q2[1] + q1[1]*q2[0] + q1[2]*q2[3] - q1[3]*q2[2]
        r2 = q1[0]*q2[2] - q1[1]*q2[3] + q1[2]*q2[0] + q1[3]*q2[1]
        r3 = q1[0]*q2[3] + q1[1]*q2[2] - q1[2]*q2[1] + q1[3]*q2[0]
        return np.array([r0, r1, r2, r3])
```

Fonction de calcul du produit de 2 quaternions

# Conversion entre représentations des désorientations

```python
def qu2ax_angle(q):
    q0 = q[:, 0]
    angle = 2*np.arccos(np.clip(q0, epsilon, 1 - epsilon))
    return angle

def qu2ax_axis(q):
    q0, q1, q2, q3 = q[:, 0], q[:, 1], q[:, 2], q[:, 3]
    axis = np.array([x for x in zip(q1, q2, q3)])

    n = len(q0)
    ax_array = np.zeros((n, 3))

    angle0_ind = np.where(q0 >= 1 - epsilon)
    ax_array[angle0_ind] = np.array([[0, 0, 1]]*len(angle0_ind))

    q00_ind = np.where(q0 <= epsilon)
    ax_array[q00_ind] = axis[q00_ind]

    others_ind = np.where(np.logical_and((q0 < 1 - epsilon), (q0 > epsilon)))
    s = np.sign(q0[others_ind])/np.linalg.norm(axis[others_ind], axis=1)
    ax_array[others_ind] = np.multiply(s[:, np.newaxis],axis[others_ind])

    return ax_array

def qu2ax_vect(q):
    angle = qu2ax_angle(q)
    axis = qu2ax_axis(q)
    return np.column_stack((axis, angle))
```

Fonction de passage de la représentation avec les quaternions (qu) à la représentation axes/angles (ax)

# Passage du code avec les quaternions

Il faut commencer par convertir en quaternions les opérateurs de symétrie :

```
"""
Defining the symetry operators in quaternions
"""
Q_cubic = np.zeros((24, 4), dtype=float)
Q_cubic[0] = om2qu(np.array([[1., 0., 0.], [0., 1., 0.], [0., 0., 1.]])) #ok
Q_cubic[1] = om2qu(np.array([[0., 0., -1.], [0., -1., 0.], [-1., 0., 0.]])) #ok
Q_cubic[2] = om2qu(np.array([[0., 0., -1.], [0., 1., 0.], [1., 0., 0.]])) #ok
Q_cubic[3] = om2qu(np.array([[-1., 0., 0.], [0., 1., 0.], [0., 0., -1.]])) #ok
Q_cubic[4] = om2qu(np.array([[0., 0., 1.], [0., 1., 0.], [-1., 0., 0.]])) #ok
Q_cubic[5] = om2qu(np.array([[1., 0., 0.], [0., 0., -1.], [0., 1., 0.]])) #ok
Q_cubic[6] = om2qu(np.array([[1., 0., 0.], [0., -1., 0.], [0., 0., -1.]])) #ok
Q_cubic[7] = om2qu(np.array([[1., 0., 0.], [0., 0., 1.], [0., -1., 0.]])) #ok
Q_cubic[8] = om2qu(np.array([[0., -1., 0.], [1., 0., 0.], [0., 0., 1.]])) #ok
Q_cubic[9] = om2qu(np.array([[-1., 0., 0.], [0., -1., 0.], [0., 0., 1.]])) #ok
Q_cubic[10] = om2qu(np.array([[0., 1., 0.], [-1., 0., 0.], [0., 0., 1.]])) #ok
Q_cubic[11] = om2qu(np.array([[0., 0., 1.], [1., 0., 0.], [0., 1., 0.]])) #ok
Q_cubic[12] = om2qu(np.array([[0., 1., 0.], [0., 0., 1.], [1., 0., 0.]])) #ok
Q_cubic[13] = om2qu(np.array([[0., 0., -1.], [-1., 0., 0.], [0., 1., 0.]])) #ok
Q_cubic[14] = om2qu(np.array([[0., -1., 0.], [0., 0., 1.], [-1., 0., 0.]])) #ok
Q_cubic[15] = om2qu(np.array([[0., 1., 0.], [0., 0., -1.], [-1., 0., 0.]])) #ok
Q_cubic[16] = om2qu(np.array([[0., 0., -1.], [1., 0., 0.], [0., -1., 0.]])) #ok
Q_cubic[17] = om2qu(np.array([[0., 0., 1.], [-1., 0., 0.], [0., -1., 0.]])) #ok
Q_cubic[18] = om2qu(np.array([[0., -1., 0.], [0., 0., -1.], [1., 0., 0.]])) #ok
Q_cubic[19] = om2qu(np.array([[0., 1., 0.], [1., 0., 0.], [0., 0., -1.]])) #ok
Q_cubic[20] = om2qu(np.array([[-1., 0., 0.], [0., 0., 1.], [0., 1., 0.]])) #ok
Q_cubic[21] = om2qu(np.array([[0., 0., 1.], [0., -1., 0.], [1., 0., 0.]])) #ok
Q_cubic[22] = om2qu(np.array([[0., -1., 0.], [-1., 0., 0.], [0., 0., -1.]])) #ok
Q_cubic[23] = om2qu(np.array([[-1., 0., 0.], [0., 0., -1.], [0., -1., 0.]])) #ok
```

Opérateurs de symétrie en quaternions, exemple cubique

# Passage du code avec les quaternions

On commence par modifier le code existant en remplaçant simplement les opérations matricielles par leur homologue en quaternion

```
586     def Q_disorientation(self, orientation, crystal_structure=Symmetry.triclinic):
587
588         the_angle = np.pi
589         symmetries = crystal_structure.Q_symmetry_operators()
590         (Q_gA, Q_gB) = (self.quaternion(), orientation.quaternion())  # nicknames
591         for (Q_g1, Q_g2) in [(Q_gA, Q_gB), (Q_gB, Q_gA)]:
592             for j in range(symmetries.shape[0]):
593                 sym_j = symmetries[j]
594                 oj = Q_product(Q_g1, sym_j) # the crystal symmetry operator is left applied
595                 for i in range(symmetries.shape[0]):
596                     sym_i = symmetries[i]
597                     oi = Q_product(Q_g2, sym_i)
598                     delta = qu2om(Q_product(Q_conjugate(oj), oi))
599                     mis_angle = Orientation.misorientation_angle_from_delta(delta)
600                     if mis_angle < the_angle:
601                         # now compute the misorientation axis, should check if it lies in the fundamental zone
602                         mis_axis = Orientation.misorientation_axis_from_delta(delta)
603                         # here we have np.dot(oi.T, mis_axis) = np.dot(oj.T, mis_axis)
604                         # print(mis_axis, mis_angle*180/np.pi, np.dot(oj.T, mis_axis))
605                         the_angle = mis_angle
606                         the_axis = mis_axis
607                         the_axis_xyz = np.dot((qu2om(oi)).T, the_axis)
608         return the_angle, the_axis, the_axis_xyz
609
```

Passage du code avec les quaternions

# Problèmes

Plusieurs problèmes sont alors apparus :

→ Le code ne renvoyait plus les bons résultats :

Problèmes dans une fonction de conversion

Problème de signe dans la fonction produit

→ Le code est ralentit ! Sur un même exemple :

Avec les matrices de rotation : 1min 48s

```
scan.segment_grains(tol=5., min_ci=0.0)
✓ 1m 48.6s

grain segmentation for EBSD scan, misorientation tolerance=5.0, minimum confidence index=0.0
segmentation progress: 100.00 %
23 grains were segmented

array([[ 1,  1,  1, ..., 19, 19, 19],
       [ 1,  1,  1, ..., 19, 19, 19],
       [ 1,  1,  1, ..., 19, 19, 19],
       ...,
       [ 3,  3,  3, ..., 22, 22, 22],
       [ 3,  3,  3, ..., 22, 22, 22],
       [ 3,  3,  3, ..., 22, 22, 22]])
```

Avec les quaternions : 4min 21s

```
scan.segment_grains(tol=5., min_ci=0.0)
✓ 4m 21.2s

grain segmentation for EBSD scan, misorientation tolerance=5.0, minimum confidence index=0.0
segmentation progress: 100.00 %
23 grains were segmented

array([[ 1,  1,  1, ..., 19, 19, 19],
       [ 1,  1,  1, ..., 19, 19, 19],
       [ 1,  1,  1, ..., 19, 19, 19],
       ...,
       [ 3,  3,  3, ..., 22, 22, 22],
       [ 3,  3,  3, ..., 22, 22, 22],
       [ 3,  3,  3, ..., 22, 22, 22]])
```

# Vectorisation du code

Pour tenter d'accélérer le code, il faut maintenant tirer partie des quaternions pour vectoriser massivement les calculs de produit.
On commence par vectoriser la fonction de calcul du produit et du conjugué

```python
#Vectorized product of to arrays of quatenions, of the same size
def Q_product_vect(q1,q2) :
    r = np.empty((q1.shape[0], 4))
    r[:,0] = q1[:,0]*q2[:,0] - q1[:,1]*q2[:,1] - q1[:,2]*q2[:,2] - q1[:,3]*q2[:,3]
    r[:,1] = q1[:,0]*q2[:,1] + q1[:,1]*q2[:,0] + q1[:,2]*q2[:,3] - q1[:,3]*q2[:,2]
    r[:,2] = q1[:,0]*q2[:,2] - q1[:,1]*q2[:,3] + q1[:,2]*q2[:,0] + q1[:,3]*q2[:,1]
    r[:,3] = q1[:,0]*q2[:,3] + q1[:,1]*q2[:,2] - q1[:,2]*q2[:,1] + q1[:,3]*q2[:,0]
    return r
```

Calcul vectorisé du produit de quaternions

```python
def Q_conjugate_vect(q):
    r = np.empty((q.shape[0], 4))
    r[:,0] = q[:,0]
    r[:,1] = -q[:,1]
    r[:,2] = -q[:,2]
    r[:,3] = -q[:,3]
    return r
```

Calcul vectorisé de la conjugaison de quaternion

# Vectorisation du code

On peut maintenant vectoriser la fonction disorientation, qui devient beaucoup plus compacte :

```
624     def Q_disorientation_vect_bis(self, orientation, crystal_structure=Symmetry.triclinic):
625         the_angle = np.pi
626         symmetries = crystal_structure.Q_symmetry_operators()
627         (Q_gA, Q_gB) = (self.quaternion(), orientation.quaternion())
628         oj = Q_product_semivect_left(Q_gA, symmetries) #crystal symetry operators are left applied
629         oi = Q_product_semivect_left(Q_gB, symmetries)
630         mis_angles = qu2ax_angle(Q_product_vect(Q_conjugate_vect(oj), oi)) #compute the misorientation angle
631         the_angle = np.min(mis_angles) #take the minimum to have the final misorientation angle
632         the_axis = None #there is no need to compute the_axis and the_axis_xyz in segment_grains
633         the_axis_xyz = None
634         return the_angle, the_axis, the_axis_xyz
635
636
```

Fonction disorientation vectorisée

# Résultats - Code d'origine

```
scan.segment_grains(tol=5., min_ci=0.0)
```

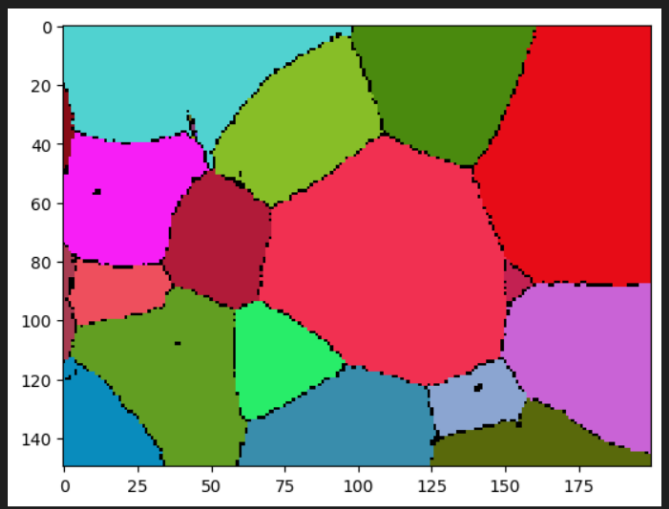
✓ 1m 48.6s

```
grain segmentation for EBSD scan, misorientation tolerance=5.0, minimum confidence index=0.0
segmentation progress: 100.00 %
23 grains were segmented

array([[ 1,  1,  1, ..., 19, 19, 19],
       [ 1,  1,  1, ..., 19, 19, 19],
       [ 1,  1,  1, ..., 19, 19, 19],
       ...,
       [ 3,  3,  3, ..., 22, 22, 22],
       [ 3,  3,  3, ..., 22, 22, 22],
       [ 3,  3,  3, ..., 22, 22, 22]])
```

```
# plot the segmented grain structure using a random color map: each color represents a grain
rand_cmap = Microstructure.rand_cmap(first_is_black=True)
plt.imshow(scan.grain_ids.T, cmap=rand_cmap, interpolation='nearest')
plt.show()
```

✓ 0.1s




Code d'origine

# Résultats - Code vectorisé

```
scan.segment_grains(tol=5., min_ci=0.0)
```

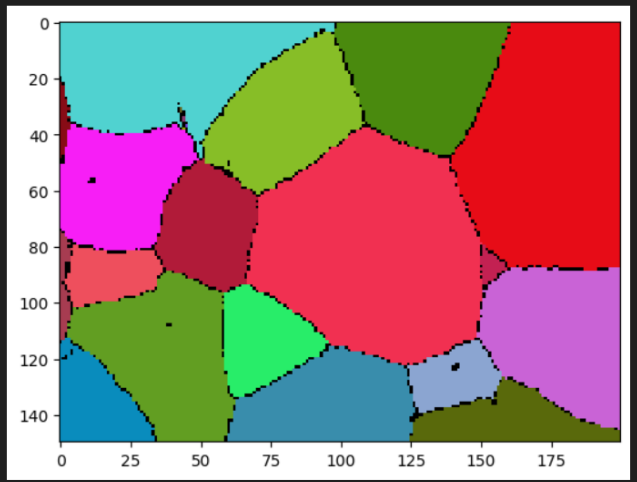
✓ 11.0s

```
grain segmentation for EBSD scan, misorientation tolerance=5.0, minimum confidence index=0.0
segmentation progress: 100.00 %
23 grains were segmented

array([[ 1,  1,  1, ..., 19, 19, 19],
       [ 1,  1,  1, ..., 19, 19, 19],
       [ 1,  1,  1, ..., 19, 19, 19],
       ...,
       [ 3,  3,  3, ..., 22, 22, 22],
       [ 3,  3,  3, ..., 22, 22, 22],
       [ 3,  3,  3, ..., 22, 22, 22]])
```

```
# plot the segmented grain structure using a random color map: each color represents a grain
rand_cmap = Microstructure.rand_cmap(first_is_black=True)
plt.imshow(scan.grain_ids.T, cmap=rand_cmap, interpolation='nearest')
plt.show()
```

✓ 0.1s




Code vectorisé

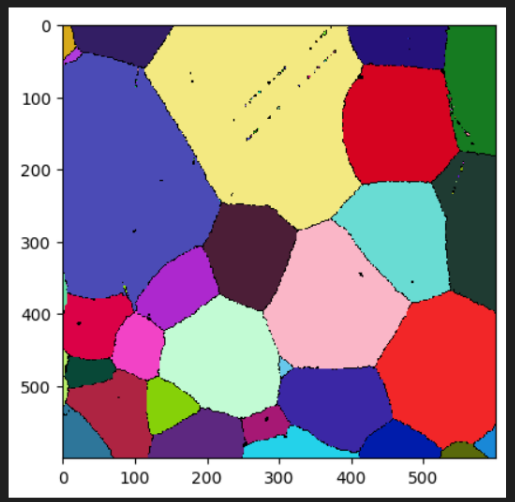
# Résultats - exemple plus grand

```
scan.segment_grains(tol=5., min_ci=0.0)
✓ 23m 26.3s

grain segmentation for EBSD scan, misorientation tolerance=5.0, minimum confidence index=0.0
C:\Users\esteb\Desktop\Projet info S2\pymicro\pymicro\crystal\microstructure.py:500: RuntimeWarning: invalid value en
  n /= np.sqrt((delta[1, 2] - delta[2, 1]) ** 2 +
segmentation progress: 100.00 %
135 grains were segmented

array([[  1,   1,   1, ..., 124, 124, 124],
       [  1,   1,   1, ..., 124, 124, 124],
       [  1,   1,   1, ..., 124, 124, 124],
       ...,
       [  5,   5,   5, ..., 133, 133, 133],
       [  5,   5,   5, ..., 133, 133, 133],
       [  5,   5,   5, ..., 133, 133, 133]])
```

```
# plot the segmented grain structure using a random color map: each color represents a grain
rand_cmap = Microstructure.rand_cmap(first_is_black=True)
plt.imshow(scan.grain_ids.T, cmap=rand_cmap, interpolation='nearest')
plt.show()
✓ 0.1s
```




Test avec un exemple plus grand

# Résultats - exemple plus grand

```
scan.segment_grains(tol=5., min_ci=0.0)
```

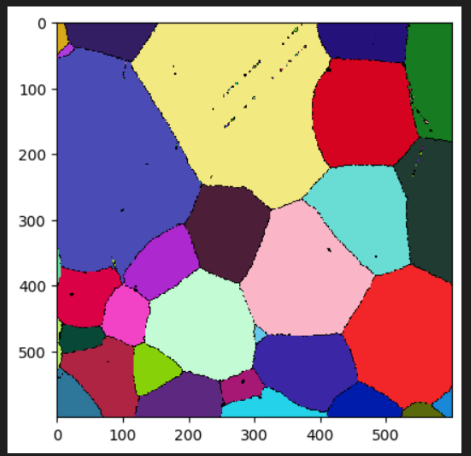
✓ 4m 20.7s

```
grain segmentation for EBSD scan, misorientation tolerance=5.0, minimum confidence index=0.0
segmentation progress: 100.00 %
135 grains were segmented

array([[  1,   1,   1, ..., 124, 124, 124],
       [  1,   1,   1, ..., 124, 124, 124],
       [  1,   1,   1, ..., 124, 124, 124],
       ...,
       [  5,   5,   5, ..., 133, 133, 133],
       [  5,   5,   5, ..., 133, 133, 133],
       [  5,   5,   5, ..., 133, 133, 133]])
```

```
# plot the segmented grain structure using a random color map: each color represents a grain
rand_cmap = Microstructure.rand_cmap(first_is_black=True)
plt.imshow(scan.grain_ids.T, cmap=rand_cmap, interpolation='nearest')
plt.show()
```

✓ 0.1s




Test avec un exemple plus grand

# Conclusion

→ Le passage en quaternions permet de rendre le code :

Plus simple

Plus rapide d'un facteur 6 à 8

→ Cependant, il reste encore quelques zones d'ombre et du travail de débugage et de test.

→ L'utilisation des quaternions est donc une piste très intéressante d'accélération des calculs dans pymicro, qui pourra être généralisée à d'autres fonctions de la librairie pour se passer au maximum des matrices de rotation.