Université Louis Pasteur de Strasbourg

Master Professionnel 2 ChemoInformatique

RAPPORT DE STAGE DE FIN D'ÉTUDES

Reconstruction et analyse de volumes en microscopie électronique 3D ; application à l'étude morphologique de supports de catalyseurs

---

Présenté par

Rodrigue Roland MAFOUANA

Rapporteurs :	R. SCHURHAMMER
	G. MARCOU
Responsable de stage :	Ovidiu ERSEN
Co Responsable de stage :	Jacques WERCKMANN
Tuteur IFP :	Fanny TIHAY

AVANT PROPOS

Ce travail a été réalisé à l'Institut de Physique et Chimie des Matériaux de Strasbourg (IPCMS), en collaboration avec l'Institut Français du Pétrole (IFP). Il entre dans le cadre de la coopération entre le Groupe Surface et Interface (GSI) de l'IPCMS et la Direction Physique et Analyse de l'IFP.

Je remercie Messieurs Pierre BECCAT et Jacques WERCKMANN, respectivement Responsable de la Direction Physique et Analyse de l'IFP et Responsable du Groupe Microscopie du GSI, de m'avoir accepté pour ce stage qui est le fruit de la coopération entre ces deux départements.

Je remercie mon directeur de stage, monsieur Ovidiu ERSEN, qui a su me conseiller et m'encourager pour la bonne marche de mon stage. Sa tolérance et sa capacité à pouvoir m'aider dans mon travail m'ont énormément marqué. Qu'il trouve ici toute ma reconnaissance.

Je tiens à exprimer ma gratitude Madame Fanny TIHAY, ma tutrice IFP. Elle s'est toujours montrée disponible quand j'ai eu besoin d'elle. Je la remercie pour son énorme contribution.

Mes remerciements vont aussi à tout le personnel du GSI pour son merveilleux accueil et la bonne ambiance qu'il a crée autour de moi. Je remercie particulièrement Christine GOYHENEX, Adèle CARRADO, Marie PILARD Jacques FAERBER, et Isabelle KITZINGER pour tous ces moments que nous avons passé ensemble à discuter autour d'un thé ou une tisane.

Sommaire

I. Introduction

Les nanomatériaux sont présents dans de nombreux domaines de la science dont certains présentent un grand intérêt pour l'industrie chimique et pétrolière. Nous pouvons citer notamment les matériaux utilisés pour le stockage de l'hydrogène, les catalyseurs, les membranes. Ces matériaux ont des propriétés macroscopiques (quantité de stockage, activité, pouvoir séparateur, stabilité etc.) dépendant étroitement de leur structure, morphologie, composition et organisation à l'échelle nanométrique. La compréhension de leurs propriétés nécessite tout d'abord une caractérisation fine à l'échelle du nanomètre. La technique de caractérisation utilisée habituellement à cette échelle est la microscopie électronique à transmission (MET), qui constitue aujourd'hui l'une des techniques d'investigation la plus couramment utilisée dans tous les domaines de la science. Elle permet d'observer et de caractériser la microstructure des matériaux jusqu'à l'échelle atomique afin d'y extraire diverses informations : morphologiques, cristallographiques, chimiques et même magnétiques.

Cependant, utilisée de manière traditionnelle (acquisition d'une image, d'un cliché de diffraction ou d'un spectre), elle reste une technique d'investigation à deux dimensions, car elle fournit des informations provenant de l'observation de l'objet dans un plan perpendiculaire à la direction du faisceau électronique.

La microscopie électronique 3D est une méthode de reconstruction du volume de l'objet à étudier à partir d'une série de projections enregistrées sous différents angles d'observation. Le principe repose sur l'acquisition d'une série d'images en MET classique en faisant tourner l'échantillon dans le microscope. La reconstruction 3D se fait ensuite par ordinateur, en utilisant une méthode de rétroprojection. La partie finale consiste à visualiser cette reconstruction et ensuite l'analyser pour pouvoir extraire des paramètres quantitatifs. Pour ce faire il faut utiliser des procédures de segmentation des données contenues dans le volume reconstruit qui permet de déterminer des paramètres tels que : la surface, la rugosité, la porosité, la densité, la surface spécifique, la distribution en taille des pores, le volume poreux etc. Ces paramètres ont une importance cruciale dans l'industrie pétrolière et intéressent notamment l'Institut Français du Pétrole (IFP) en collaboration avec lequel ce stage a été effectué, et dont la recherche et développement est axée vers l'application et

couvre l'ensemble des domaines techniques liés à l'industrie des hydrocarbures et de leurs dérivés et substituts.

Dans ce contexte, le but essentiel de mon stage était d'utiliser le logiciel *ImageJ*, orienté vers le traitement d'images, pour réaliser la segmentation des données et l'extraction des paramètres quantitatifs. En particulier, j'ai utilisé le langage macro (programme simple constitué d'une série de commandes d'*ImageJ*) qui nous a permis d'automatiser des tâches et de créer des routines qui peuvent être utilisées par tous les utilisateurs du domaine. Notre choix s'est porté sur ce langage pour plusieurs raisons : d'une part, il est implémenté dans un logiciel qui permet d'effectuer les opérations souhaitées ; ensuite, l'écriture d'une macro est relativement simple et facile à comprendre par la suite ; enfin ce langage permet la création des boîtes de dialogue qui lui confère un caractère interactif.

II. Travail préliminaire : Reconstruction volumique à partir d'une série de projection

II. 1 Acquisition de la série

La procédure expérimentale d'acquisition des séries de projections qui font servir à la reconstruction volumique est détaillée dans l'annexe 1.

II. 2 Reconstruction avec le logiciel *IMOD*

La reconstruction est l'étape essentielle dans l'obtention d'un tomogramme, la plus longue et celle qui comporte le plus de difficultés.

Il existe plusieurs logiciels qui peuvent être utilisés pour calculer le volume de l'objet étudié. L'un des plus connus et utilisés, surtout par les microscopistes travaillant dans le domaine de la biologie moléculaire, est le logiciel *IMOD*. C'est le logiciel que nous avons utilisé pour aligner et corriger les images, puis pour générer les reconstructions volumiques qui seront présentées dans ce rapport.

Ce logiciel a été conçu par l'équipe de microscopie électronique 3D des cellules de l'Université de Colorado (The Boulder Laboratory for 3-D Electron Microscopy of Cells, http://bio3d.colorado.edu).

1) Alignement et corrections

Avant de passer au calcul du volume à partir de la série de projections, il faut tout d'abord effectuer deux étapes préliminaires qui sont d'une importance cruciale car elles vont déterminer la qualité de la reconstruction finale (en supposant que les projections ont été correctement acquises). La première étape consiste à **aligner** toutes les projections, c'est-à-dire à les mettre dans un système de coordonnées 3D unique, avec la même origine. La deuxième étape consiste à appliquer à la série de projections certaines **corrections** que nous devons impérativement considérer pour obtenir, à la fin, un volume bien défini.

Ces corrections ont plusieurs origines :

a) les valeurs données pour les *angles de tilt* ne sont pas très précises (imperfections mécaniques du goniomètre) ;

b) les *grandissements* ne sont pas identiques sur toutes les images (nous n'avons pas exactement la même mise au point dans toute la série) ;

c) des *distorsions* sont présentes sur une même image, car l'objet n'est pas nécessairement positionné dans un plan perpendiculaire à la direction d'observation (c'est le cas des objets aplatis aux grands angles de tilt).

Les détails des procédures d'alignement et de correction sont présentés dans l'annexe 2.

2) Calcul du volume : Reconstruction par rétroprojection filtrée

Le principe de calcul est basé sur le fait que, dans l'espace réciproque, toute projection 2D de l'objet correspond à une section centrale dans la Transformée de Fourier (TF) de l'objet orientée perpendiculairement à la direction de projection (*théorème de la coupe centrale*). Donc, en enregistrant une série de projections et en recombinant après les TF correspondantes, on rempli l'espace de Fourier correspondant de l'objet (annexe 3).

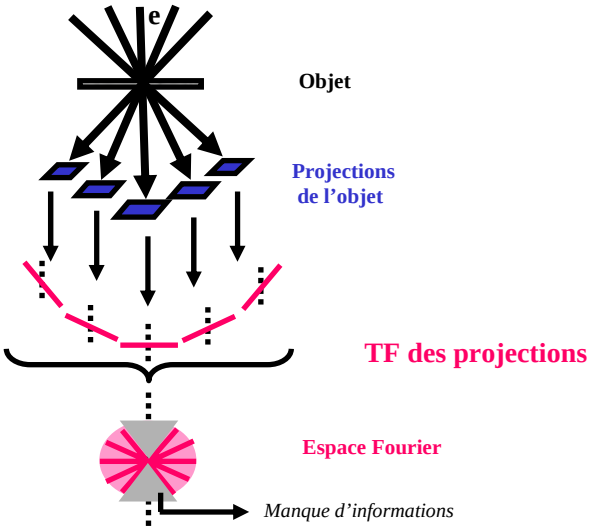


Figure 2 : *Remplissage de l'espace de Fourier de l'objet à partir de la série de projections.*

Du point de vue mathématique, nous pouvons définir la densité de l'objet à étudier comme une fonction *f(x,y,z)* qui, en chaque point de coordonnée *(x,y,z)*, donne la valeur de la densité locale. La reconstruction tomographique consiste à inverser la transformée de Radon (définie comme l'ensemble des projections d'un objet lorsque l'angle de projection varie entre 0 et π, voir figure 3), c'est-à-dire d'estimer la fonction *f(x,y,z)* à partir de l'ensemble de projections.

Différentes techniques de reconstruction (incluant les algorithmes mathématiques utilisés et leur mise en pratique) peuvent être utilisées pour obtenir le volume des objets à partir des séries de projections. D'une part nous avons les méthodes analytiques, comme par exemple les méthodes Fourier et la rétroprojection filtrée ; d'autre part on dispose des méthodes discrètes (ou itératives) travaillant dans l'espace direct qui sont plus difficiles à mettre en œuvre, moins rapides mais présentent l'avantage de pouvoir modéliser les fluctuations statistiques affectant les données brutes. De plus, ces méthodes peuvent être utilisées pour la reconstruction des données acquises suivant une géométrie complexe, ce qui n'est pas le cas de la rétroprojection filtrée.

La méthode la plus connue et la plus utilisée est la rétroprojection filtrée, beaucoup plus rapide que les méthodes itératives. La rétroprojection est le processus inverse de la projection : tandis que la projection produit une image 2D d'un objet 3D, la rétroprojection tente de reconstruire un objet rétroprojeté 3D qui soit en accord avec la projection de départ.

La somme de toutes les rétroprojections individuelles va générer une approximation de l'objet initial. La solution est unique seulement si nous disposons d'une infinité de projections (un ensemble fini de projections est compatible avec plusieurs objets). Plus le nombre de projections est réduit, plus les artefacts d'épandage en étoile sont importants (figure 3).

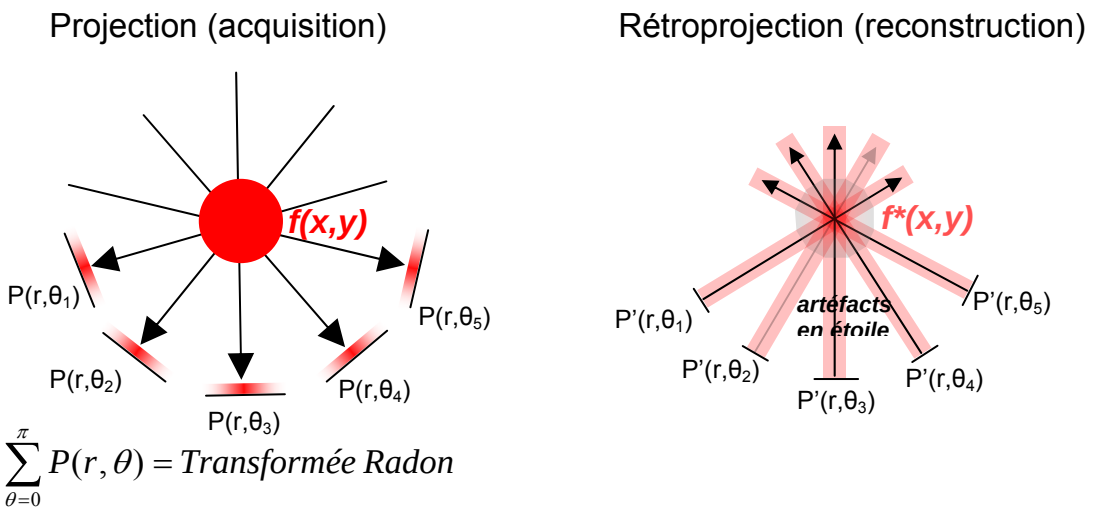


Figure 3 *: Projection et rétroprojection. Objet initial et objet rétroprojété. Artefacts d'épandage en étoile.*

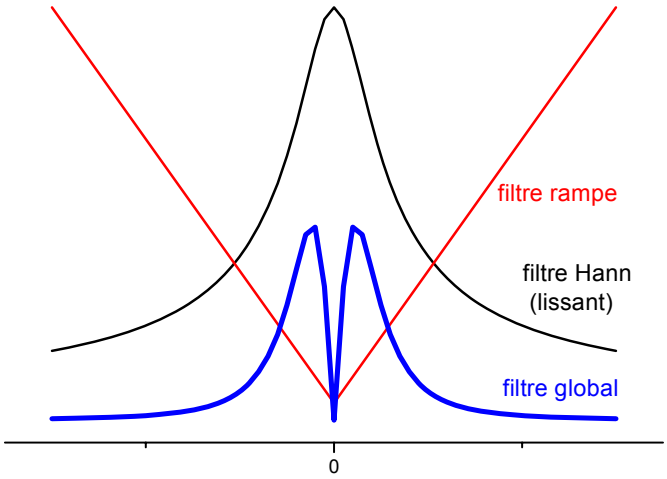


Figure 4 *: Exemple de filtre utilisé dans la rétroprojection filtrée.*

Par rapport à la rétroprojection simple, dans la rétroprojection filtrée les artefacts d'épandage en étoile qui déforment énormément la reconstruction sont diminués. Pour la mettre en pratique, on multiplie tout d'abord les TF de projections par un filtre rampe (voir la figure 3) dans le but de diminuer ces artéfacts (pour plus d'information, voir « *Reconstruction des images tomographiques par rétroprojection filtrée* », F. Dubois, Revue de l'ACOMEN 1998, vol. 4, n° 2). Ensuite, pour réduire les hautes fréquences associées au bruit des images (initialement amplifiées par le filtre rampe) nous pouvons associer au premier filtre un filtre lissant de type « passe-bas » (figure 4), dont la fréquence de coupure sera l'un de paramètres variables dans la procédure de reconstruction.

En résumé, le processus de calcul est le suivant : à partir de la série de projections $P(r,\theta)$ on calcule les transformées de Fourier $FT[P(r,\theta)]$, on applique les deux filtres $|\Gamma|\cdot FT[P(r,\theta)]$, ensuite les TF inverses $P'(r,\theta)=IFT\{|\Gamma|\cdot FT[P(r,\theta)]\}$ et finalement par rétroprojection on obtient une approximation de l'objet 3D.

$$f(x,y,z) \xrightarrow{\text{Projection}} \Sigma\, P(u,v,\theta) \quad (\theta \text{ variable})$$

$$P(u,v,\theta) \xrightarrow{FT} FT[P(u,v,\theta)] \xrightarrow{\text{filtrage}} |\Gamma|\cdot FT[P(r,\theta)] \xrightarrow{IFT} P'(u,v,\theta) \xrightarrow{\text{Rétroprojection}} f^*(x,y,z)$$

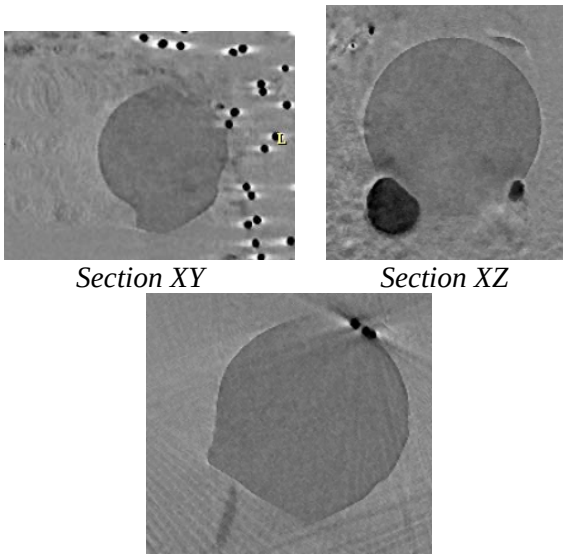
Figure 5 *: Algorithme mathématique utilisé dans la rétroprojection filtrée (la deuxième ligne, la première correspond à l'acquisition).*

3) Extraction de la région d'intérêt

Les toutes dernières étapes de la reconstruction consistent à extraire, du volume obtenu par le calcul, uniquement la région d'intérêt et de combiner les deux tomogrammes si l'on a enregistré deux séries à des angles de tilt différents.

II. 3 Visualisation avec SLICER

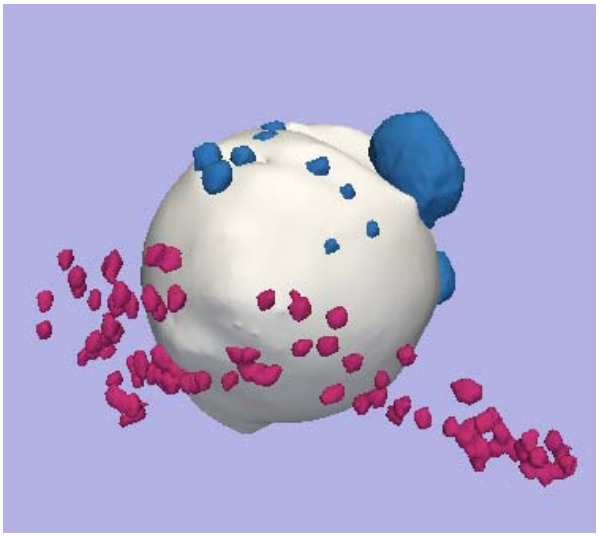
Pour visualiser et analyser les modèles créés, il est possible d'utiliser des logiciels spécifiques à la recherche médicale, comme par exemple le logiciel *SLICER* (http://www.slicer.org) que nous avons parfois utilisé.



Section XY *Section XZ*

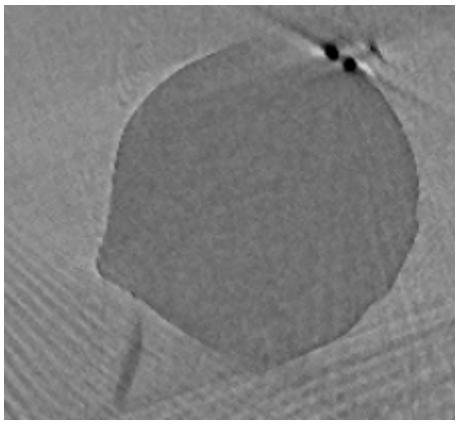
Section YZ

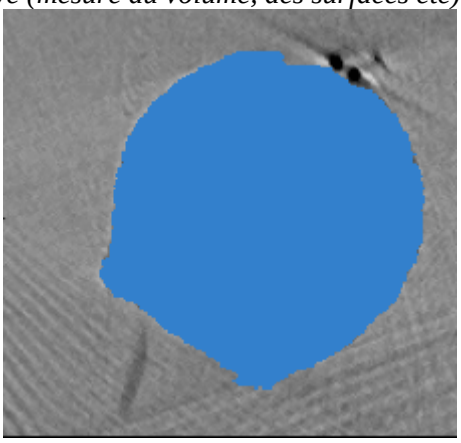
Sections dans le volume reconstruit de l'objet.



Opale en blanc, Part. Ni en bleu, Billes Au en rouge.

Modélisation du volume : visualisation et analyse quantitative (mesure du volume, des surfaces etc).





Exemple de segmentation des données (ici nous avons extrait du volume les pixels bleus correspondant à la bille d'opale).

***Figure 6** : Sections, modèles et segmentation*

On peut procéder de deux manières pour visualiser l'objet reconstruit : soit faire des sections dans l'objet reconstruit, soit, ce qui est plus intéressant mais également plus délicat à

faire, représenter uniquement la surface de l'objet ou le détail que nous étudions en créant ce qu'on appelle un modèle.

Ce modèle peut être défini manuellement, en traçant à la main sur chaque section les contours qui nous intéressent. Sinon, nous devons utiliser des procédés de segmentation sur les données contenues dans le volume reconstruit, ce qui nous permettra de délimiter les objets (ou les différentes parties du même objet qui ne sont pas équivalentes) et ensuite effectuer si possible des mesures quantitatives, au cas où celles-ci s'avéreraient nécessaires. Cette deuxième démarche a fait l'objet de mon stage et sera présentée dans le chapitre suivant.

Le processus le plus simple de segmentation est le seuillage des données en fonction du niveau de gris de chaque pixel. Dans la plupart de cas ce type de segmentation n'est pas suffisant, il faut passer par des algorithmes spécifiques capables de détecter les contours même pour les formes les plus complexes.

III. Utilisation d'un langage de programmation pour l'analyse quantitative des données 3D

Afin de pouvoir exploiter les données volumiques obtenues par reconstruction 3D, nous avons utilisé un langage de programmation spécifique, construit à partir des commandes Java et implémenté dans le logiciel *ImageJ*. Ce logiciel est brièvement décrit dans la première partie de ce chapitre. Sa particularité, qui a d'ailleurs motivée notre choix, est liée à la possibilité de création de plugins et de macros pour nous permettre d'automatiser des opérations à effectuer sur des objets similaires ainsi que pour effectuer un traitement statistique des résultats obtenus.

Dans ce cadre, nous avons crée plusieurs types de macros. Certaines de ces macros ont été construites dans le but de pouvoir travailler et manipuler des volumes et gérer les résultats obtenus, tandis que la plupart a été dédiée à des opérations de segmentation de données afin d'extraire les paramètres quantitatifs d'intérêt.

Les macros que nous avons réalisées au cours de ce stage utilisent soit des plugins *ImageJ* déjà existants, soit des fonctionnalités de base de ce logiciel qui sont utilisées de manière appropriée afin de réaliser les opérations souhaitées dans le bon ordre. Ces deux

types de macros sont l'objet de la deuxième partie de ce chapitre. Certains des résultats obtenus sont présentés dans la dernière partie, ainsi qu'une macro permettant de synthétiser ces résultats et de les présenter dans un format convenable.

III.1 Description du logiciel *ImageJ*

ImageJ est un logiciel libre de traitement d'image basé sur Java et inspiré de NIH (qui est un logiciel libre de traitement et d'analyse d'image pour Macintosh). Il a été crée par Wayne Rasband (wayne@codon.nih.gov), chercheur à l'Institut National de Santé Mentale de Bethesda (Maryland, Etats-Unis). Il est téléchargeable comme application et facile à installer sur n'importe quel ordinateur (il existe des versions pour Windows, Macintosh et Linux) doté d'une machine virtuelle Java 1.1 ou de toute machine virtuelle ultérieure à celle-ci.

La fenêtre d'*ImageJ* contient une barre de menu (au dessus de l'écran). La barre d'outil, la barre de statut, une barre de progression des images, les histogrammes, la ligne profil, etc. s'obtiennent à partir des fenêtres additionnelles.

ImageJ peut ouvrir, éditer, analyser, traiter et imprimer des images en 8, 16 et 32 bits. Les formats des images que l'on peut ouvrir avec ce logiciel sont : le TIFF, le GIF, le JPEG, le BMP, le DICOM et le "RAW". *ImageJ* supporte des "stacks", c'est à dire des séries d'images qui partagent la même fenêtre. On peut y effectuer plusieurs opérations en parallèle et ouvrir plusieurs fenêtres (images) simultanément, leur nombre n'étant limité que par la mémoire disponible.

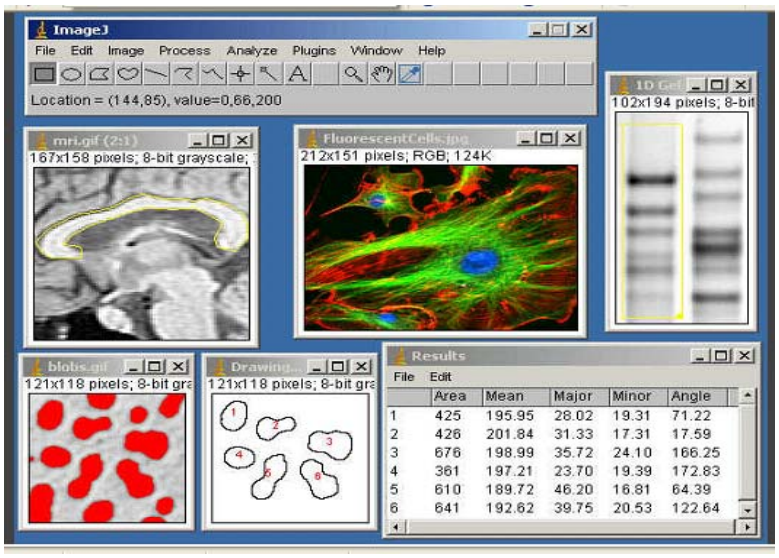


***Figure 7** : Fenêtre d'ImageJ.*

Il peut faire des calculs statistiques sur des valeurs de pixel définis par l'utilisateur, mesurer des distances et des angles, créer des histogrammes etc. Un calibrage spatial est disponible afin de fournir des mesures dimensionnelles dans les unités telles que le millimètre. Il supporte des fonctions standard de traitement d'images telles que la manipulation de contraste, le lissage, la détection de bord et le filtre médian. Il fait des transformations géométriques telles que la graduation, la rotation etc. On peut zoomer et dé-zoomer l'image. Toutes les analyses et les fonctions de traitement d'image sont disponibles à n'importe quel facteur de rapport optique.

ImageJ est un logiciel dont les fonctions peuvent être étendues par le biais des plugins et des macros. Ainsi, les plugins et les macros écrits par l'utilisateur lui permettent de résoudre presque n'importe quel problème de traitement ou d'analyse d'image.

1) Les plugins

Les plugins sont des classes Java placées dans le dossier plugins d'*ImageJ* ou dans l'un de ses sous dossiers. Le terme plugin peut se traduire par « module » en français, mais nous utiliserons le terme anglais.

De façon très simple, un plugin peut se créer à partir de l'enregistreur. C'est-à-dire qu'à partir de la barre de menu, cliquer sur Plugins ensuite sur Macros puis sur Record. Le fichier texte s'obtient en cliquant sur « Create ». A partir de ce mini-éditeur de texte, le fichier texte peut être converti en plugin en cliquant sur Edit, puis sur Convert to plugin. Cela entraîne l'ouverture d'une nouvelle fenêtre qui porte le même nom que le fichier texte assorti d'un « _ » avec l'extension « .Java ».

En dehors du fait que cette fonctionnalité permet de développer très rapidement un plugin, elle permet en plus de créer le squelette de base des futurs plugins.

```
import ij.* ;
import ij.process* ;
import ij.gui* ;
import java.awt.* ;
import ij.plugin* ;
```

Où ij est la classe principale à partir de laquelle dérive tout le reste (voir annexe 4).

L'exemple ci-dessous montre la structure d'un plugin.

```
import ij.*;
import ij.process* ;
import ij.gui* ;
import java.awt.* ;
import ij.plugin.*;
public class Hello_ implements PlugIn {
public void run(String str) {
IJ.showMessage("Hello World en Java pour ImageJ !");
                    }
                                    }
```

Dans ce code nous importons d'abord les paquets dont nous aurons besoin (mot clé : *import*). Nous déclarons ensuite la classe Hello en n'oubliant pas le nécessaire « _ » dans son nom ; cette classe implémente l'interface *PlugIn* (d'où l'import de *ij.plugin*.* qui contient notre interface). Cette interface n'a qu'une méthode publique : *run*. L'import de *ij* est nécessaire pour la méthode *IJ.showMessage.*

Plus de 100 plugins sont disponibles sur le site Web d'*ImageJ*. De plus amples informations sur les plugins sont disponibles à l'adresse http://mtd.fh-hagenberg.at/depot/imaging/imagej/. Les plugins sont faciles à lancer et sont plus flexibles, mais il est plus facile d'écrire et de corriger une macro.

2) Les macros

Les macros sont des programmes simples qui automatisent une série de commandes d'*ImageJ*. Une macro est sauvegardée dans un simple fichier texte. Elle peut être rappelée ou lancée à partir d'une commande accessible par les menus, à l'aide d'une touche de raccourci, en cliquant sur une icône, ou même automatiquement au lancement de l'application.

La façon la plus simple de créer une macro est d'enregistrer une série de commandes à l'aide de l'enregistreur ou recorder. Un fichier macro peut contenir plus d'une macro. Pour cela, les différentes macros doivent être déclarées par l'instruction macro délimitées par des accolades. Mais on peut aussi créer une macro en écrivant soit même le script à partir des fonctions appropriées présentées dans l'annexe 5. Il est aussi possible de définir ses propres fonctions.

De plus amples informations sur les macros sont disponibles à l'adresse http://rsb.info.nih.gov/ij/docs/intro.html. Il y a plus de 200 macros disponibles sur le site Web d'*ImageJ*.

III.2 Les macros réalisées

1) Premier type de macros créées : macros utilisant des plugins existants

a) Macro Snake

Cette macro utilise le plugin SplineSnake qui permet de sélectionner les pixels d'une image en fonction du niveau de gris. Avant de lancer ce plugin, il faut sélection une partie de l'image contenant l'objet à l'intérieur. Au cours de son exécution la courbe de sélection est attirée aux frontières de l'objet (qui présentent un fort gradient du niveau de gris), cela permet de déterminer le contour de l'objet. Ce plugin ne pouvant s'exécuter que sur une image, le but de cette macro est d'automatiser la segmentation pour la série d'images contenue dans le volume reconstruit.

```
// OUVERTURE DU FICHIER
open();
// RECUPERATION DU NOM DE L'IMAGE
Image = getTitle();
// CHANGEMENT DE LA COULEUR DE FOND DE L'IMAGE
setBackgroundColor(160,160,160);
// AUGMENTATION DES DIMENSIONS DE L'IMAGE
run("Canvas Size...", "width=500 height=300 position=Center");
// CHANGEMENT DE CONTRASTE
run("Enhance Contrast", "saturated=0.5 normalize normalize_all");
// CREATION DE DEUX NOUVELLES SERIES D'IMAGES
newImage("Contours D4-05-50k","RGB Color",getWidth(),getHeight(),nSlices);
newImage("Segmentation D4-05-50k","8-bit",getWidth(),getHeight(),nSlices);
// AUTOMATISATION DE LA SEGMENTATION DE TOUTES LES IMAGES DE LA SERIE AVEC
```

```
// LE PLUGIN SPLINE SNAKE
for(i=20;i<=60;i++) {
// SELECTION DE L'IMAGE DE DEPART
                    selectWindow(Image);
// POSITIONNEMENT SUR L'IMAGE i ET COPIAGE DE CELLE-CI
                    setSlice(i);run("Copy");
// CREATION D'UNE NOUVELLE IMAGES ET COLLAGE DE L'IMAGE COPIEE CI DESSUS
                    newImage("Slice" +i+"","8-bits",getWidth(),getHeight(),1);run("Paste");
// SELECTION D'UNE PARTIE CONTENANT L'OBJET SUR L'IMAGE RECEMMENT COLLEE
                    makeRectangle(150,67,137,125);
// APPLICATION DU PLUGIN SPLINE SNAKE
                     run("SnakeD ", "gradient=40 regularization=0.4 number=1000 step=5
                     first=1 last=1 draw=Red alpha=0.2 create");
// COPIAGE DU RESULTAT OBTENU PUIS COLLAGE DANS LE S DEUX SERIES D'IMAGES CREEES
// AVANT LA BOUCLE
                     run("Copy");close();selectWindow("Slice"+i+"");close();
                     selectWindow("Segmentation D4-05-50k");setSlice(i);run("Paste");
                     wait(3000);
                     selectWindow("Slice"+i+"_snake_deriche"); run("Copy");close();
                     selectWindow("Contours D4-05-50k");setSlice(i);run("Paste");
                     }
```

b) Macro Sélection des objets

Cette macro est une macro outil, c'est-à-dire une macro qui doit être ajoutée à la barre d'outils d'*ImageJ* et qui s'exécute lorsque l'utilisateur clique sur l'image. Elle utilise le plugin Crop qui permet d'extraire un volume donné d'un volume beaucoup plus grand. Cette macro permet d'exécuter le plugin Crop (après que l'utilisateur ait cliqué sur l'objet à extraire) et d'enregistrer le nouveau volume dans un répertoire de son choix, afin de pouvoir économiser la mémoire d'*ImageJ* si on a plusieurs objets à extraire. En cliquant plusieurs fois dans la pile d'images, plusieurs petits volumes seront extraits successivement autour des positions choisies.

```
// OUVERTURE DU FICHIER
```

```
open();
// POSITIONNEMENT A L'INTERIEUR DE LA SERIE
setSlice(nSlices/3);
// UTILISATION D'UNE BOITE DE DIALOGUE POUR  CREER UN REPERTOIRE
parent = getDirectory("choisir repertoire parent");
Dialog.create("Création repertoire");
Dialog.addString("Nom repertoire :", "saisir le nom");
Dialog.show();
repertoire = parent+Dialog.getString+File.separator;
File.exists(repertoire);
    if (File.exists(repertoire) ==true)
        exit("ce repertoire existe déjà");
    else
        File.makeDirectory(repertoire);
// SELECTION DE LA MACRO OUTIL 10
setTool(10);
// SELECTION D'UN TYPE D'OBJET SUR LA STACK DE DEPART
var x,y,z,dx,dy,dz,X1,X2,Y1,Y2,Z1,Z2,width, height,depth,nomImage,emplacement;
macro "cercle Tool - C000 O11cc T5b10+" {
// UTILISATION D'UNE BOITE DE DIALOGUE POUR CARACTERISER LES IMAGES
ImgDepart= getTitle();
Dialog.create("Caractérisation des images");
Dialog.addString("Nom de l'image : ", nomImage);
Dialog.addNumber("Axe sur  x", dx);
Dialog.addNumber("Axe sur y", dy);
Dialog.addChoice("format : ", newArray("jpeg", "gif", "tiff", "bmp"), "tiff");
Dialog.addString("Path : ", repertoire);
Dialog.show();
nomImage = Dialog.getString();
dx = Dialog.getNumber();
dy = Dialog.getNumber();
dz = dx/2+dy/2;
format = Dialog.getChoice();
repertoire = Dialog.getString();
```

```
getCursorLoc( x, y, z, flags);
d(); // FONCTION
// INFO SUR LES DIMENSIONS DE LA STACK
 width=getWidth();
height=getHeight();
depth=nSlices;
// PARAMETRAGE DE LA LARGEUR DE LA SELECTION
X1=x-dx/2; if (X1< 1) X1=1;
X2=x+dx/2; if (X2>width) X2=width;
// PARAMETRAGE DE LA HAUTEUR DE LA SELECTION
Y1=y-dy/2; if (Y1<1) Y1=1;
Y2=y+dy/2; if (Y2>height) Y2=height;
// PARAMETRAGE DU NOMBRE DE SECTIONS DE LA SELECTION
Z1=z-dz/2;  if (Z1<1) Z1=1;
Z2=z+dz/2; if (Z2>depth) Z2=depth;
h();         // FONCTION
q();         // FONCTION
                }
// d() permet de tracer un oval, de choisir la couleur blanche pour foreground, tracer l'oval et
// entrer les données de la selection dans le Roi Manager
function d() {makeOval (x-dx/2, y-dy/2, dx, dy);
              setForegroundColor(255,255,255);run("Draw");}
// h() permet de découper l'objet selectionné
function h() {run("TJ Crop", "x-range="+X1+","+X2+" y-range="+Y1+","+Y2+"
                z-range="+Z1+","+Z2+" t-range=1,1 ch-range=1,1");
                rename(nomImage);saveAs(format, emplacement+nomImage);img = getTitle();
                selectWindow(img);close();}
// q() permet de revenir sur la stack de départ
function q() {selectWindow(ImgDepart);}
```

c) Marco « Calcul de la porosité »

Elle utilise le plugin Object 3D counter qui compte le nombre d'objets 3D dans une pile d'images et renvoie comme résultat le volume, la surface, le centre de masse et la

position de l'intensité maximale de chaque objet. Cette macro automatise la détection de contour d'un objet par seuillage et binarisation sur une série d'images, puis exécute le plugin Object 3D counter afin de détecter les objets en trois dimensions et permet dans le cas d'un matériau poreux d'évaluer la porosité. Le code source de cette macro est présenté dans l'annexe 6. Pour l'échantillon étudié, la porosité déterminée par cet algorithme est de l'ordre de 10,34 %.

2) Macro utilisant des fonctions prédéfinies dans le logiciel *ImageJ*

a) Marco Séparation des faces

Cette macro permet de calculer les proportions des différentes faces exposées ainsi que de séparer la partie latérale des parties extrêmes d'un catalyseur d'hydrogénation sélective (Pd) en forme de bâtonnet à section pentagonale qui présente des faces non équivalents. En effet, ces catalyseurs utilisés dans les réactions d'hydrogénation des coupes pétrolières ont une réactivité sélective due au fait que les faces n'ont pas la même réactivité. Ainsi, estimer la proportion des faces exposées catalyseur peut permettre de comprendre et de prévoir la réactivité du catalyseur.

```
//OUVERTURE DU FICHIER
open("CONTOUR CYLINDRE.tif");
run("8-bit");
//FIXATION DES PARAMETRES DE MESURE + ANALYSE DE PARTICULES
run("Set Measurements...", "area centroid perimeter slice redirect=None decimal=3");
run("Analyze Particles...", "size=0-Infinity circularity=0.00-1.00 show=Nothing display stack");
print(nResults);
//CALCUL DU CONTOUR TOTAL
var Contour_Total;
a = 0;
for (i=0; i< nResults; i++) {
        a1 = getResult("Perim.", i);
        a = a + a1;
```

```
        }
print("Surface Totale = "+a);
setResult("Contour_Total", nResults, a);
//CALCUL DU CONTOUR MOYEN DE LA PARTIE CENTRALE
N = nSlices;
b = 0;T = 0;
for (i=N/2 -75; i<=N/2 +75; i++) {
            b1= getResult("Perim.", i);
            b = b + b1;
            T = T + 1;
             }
M = b/T;
setResult("ContourLatMoy", nResults, M);
// CAL CUL DE LA SURFACE LATERALE
var ContLat, SurfaceLat;
SurfaceLat = 0;
for (i=0; i < nResults; i++) {
                 h = getResult("Perim.", i);
                if ( h >= 0.9*M && h <= 1.10*M)   {
                                                    surface = getResult("Perim.", i);
                                                    SurfaceLat = SurfaceLat + surface;
                                                  }
                                    }
print("Surface Laterale = "+SurfaceLat);
setResult("SurfaceLat", nResults, SurfaceLat);
updateResults();
var SurfaceExtrem, PartieLat, PartieExtrem;
SurfaceExtrem = a - SurfaceLat; print("SurfaceExtrem = "+SurfaceExtrem);
PartieLat = (SurfaceLat/a)*100; print("Partie Laterale = "+PartieLat+" %");
PartieExtrem = (SurfaceExtrem/a)*100; print("Parties Extremes = "+PartieExtrem+" %");
//COLORATION DE LA SURFACE LATERALE
selectWindow("CONTOUR CYLINDRE.tif");
newImage("Coloration","RGB Color",getWidth(),getHeight(),nSlices);
for (i = 7; i <= 354 ; i++) {
```

```
                        selectWindow("CONTOUR CYLINDRE.tif");
                        setSlice(i);run("Copy");
                        selectWindow("Coloration");setSlice(i);run("Paste");
                        h = getResult("Perim.", i);
                        if ( h >= 0.9*Y && h <= 1.10*Y)  changeValues(0,175,200);
                          }
selectWindow("Coloration");
run("8-bit");
// SEPARATION
selectWindow("CONTOUR CYLINDRE.tif");
newImage("MILIEU_CYLINDRE","8-bit",getWidth(),getHeight(),nSlices);
newImage("EXTREMITES_CYLINDRE","8-bit",getWidth(),getHeight(),nSlices);
for (i = 7; i<=354 ; i++) {
                        selectWindow("CONTOUR CYLINDRE.tif");
                        setSlice(i);run("Copy");
                        h = getResult("Perim.", i);
            if ( h >= 0.9*M && h <= 1.20*M)
                {selectWindow("MILIEU_CYLINDRE");setSlice(i);run("Paste");}
            else {selectWindow("EXTREMITES_CYLINDRE");setSlice(i);run("Paste");}
                                }
```

b) Macro Détection de contours par seuillage et binarisation

C'est une macro qui permet d'une part de détecter les contours d'un objet 3D, comme la macro Snake, mais cette fois-ci par seuillage en fonction des niveaux de gris des pixels, binarisation et ensuite lissage des contours obtenus. D'autre part, elle permet d'éliminer les objets secondaires (billes d'or utilisées pour la calibration) et de calculer la surface et le volume de l'objet qui nous intéresse. Le code source de cette macro est présenté dans l'annexe 7.

c) Macro Calcul du pourcentage des différents objets

Toujours en catalyse, les catalyseurs d'hydrogénation sélective au palladium peuvent se présenter sous plusieurs formes géométriques (cubique, tétraèdre, bâtonnet de section pentagonale et ceux que nous nommerons MTP (Multiple Twin Particle)). Les différentes facettes de chaque objet sont du même type en dehors des bâtonnets pour lesquels les faces latérales ont une orientation différente des faces des extrémités. Ainsi, dans un mélange (supposé homogène) de ces catalyseurs d'hydrogénation sélective, le fait de connaître les proportions de chaque catalyseur de forme géométrique déterminée permettra de prévoir la réactivité totale. Cette macro permet donc d'estimer, à partir d'un échantillonnage représentatif et contenant tous les types de catalyseurs, la proportion surfacique de chacun.

```
// OUVERTURE DU FICHIER ET POSITIONNEMENT A L'INTERIEUR DE LA SERIE
open(Filt3b_Sc025_MTP_B2_29kX_d.img);
setSlice(41);
// DUPPLICATION DE LA SERIE D'IMAGES
run("Duplicate...", "title=[Etape Intermédiaire-1] duplicate");setSlice(41);
// SEUILLAGE DE LA SERIE DES IMAGES
selectWindow("Filt3b_Sc025_MTP_B2_29kX_d.img");
setSlice(41);setThreshold(0, 128);
run("Threshold", "thresholded remaining black stack");
run("Options...", "iterations=1 count=1");
run("Fill Holes", "stack");
run("Open", "stack");
run("Options...", "iterations=3 count=1");
run("Erode","stack");
run("Dilate","stack");
run("Options...", "iterations=1 count=1");
run("Close-", "stack");
run("Fill Holes", "stack");
//VERIFICATION DE LA SEGMENTATION
run("Duplicate...", "title=[Etape Duplic-1] duplicate");
run("Find Edges", "stack");
imageCalculator("Max create stack", "Etape Intermédiaire-1","Etape Duplic-1");
selectWindow("Etape Intermédiaire-1"); close();
selectWindow("Etape Duplic-1"); close();
```

```
// ANALYSE DES PARTICULES
selectWindow("Filt3b_Sc025_MTP_B2_29kX_d.img");
setAutoThreshold();
//run("Threshold...");
setThreshold(0, 128);
run("Threshold", "thresholded remaining black stack");
run("Set Measurements...", "area centroid center perimeter redirect=None decimal=3");
run("Analyze Particles...", "size=0-Infinity circularity=0.00-1.00 show=Nothing display clear record stack");
//CALCUL DU PERIMETRE TOTAL DES DIFFERENTS TYPE D'OBJETS
var Perim_Total_Cylindre,Perim_Total_Triangle, Perim_Total_Quelconque, Perim_Total_Cube;
a = 0; b = 0; c = 0; d= 0;
for (i=0; i< nResults; i++) {
                                selectWindow("Filt3b_Sc025_MTP_B2_29kX_d.img");
                                a1 = getResult("Perim.", i);
//Calcul du perimètre total des cylindres
                                if (a1>0.98*111  && a1<1.16*111) { a = a + a1;}
//Calcul du perimètre total des triangles
                                else if (a1>0.955*91.5 && a1<1.19*91.5) {b = b + a1;}
//Calcul du perimètre total des objets de forme quelconque
                                 else if (a1>0.9*80.5  && a1<1.085*80.5) {c = c + a1;}
 //Calcul du perimètre total des cubes
                                 else if (a1>0.87*42.5  && a1<1.18*42.5) {d = d + a1;}
                                 }
setResult("Perim_Total_Cylindre", nResults, a);
setResult("Perim_Total_Triangle", nResults, b);
setResult("Perim_Total_Quelconque", nResults, c);
setResult("Perim_Total_Cube", nResults, d);
updateResults();
// RECUPERATION DES DONNEES DANS LE TABLEAU DES RESULTATS
// Recuperation de la valeur du perimetre total des cylindres
for (i=0; i< nResults-3; i++) {
                                Perim_Total_Cylindre = getResult("Perim_Total_Cylindre", i);
```

```
                                Perim_Total_Cylindre = Perim_Total_Cylindre + 0;
                                }
print("Perim_Total_Cylindre="+Perim_Total_Cylindre);
// Recuperation de la valeur du perimetre total des triangles
for (i=0; i< nResults-2; i++) {
                                    Perim_Total_Triangle = getResult("Perim_Total_Triangle", i);
                                    Perim_Total_Triangle = Perim_Total_Triangle + 0;
                                    }
print("Perim_Total_Triangle="+Perim_Total_Triangle);
// Recuperation de la valeur du perimetre total des objets de forme quelconque
for (i=0; i< nResults-1; i++) {
                            Perim_Total_Quelconque = getResult("Perim_Total_Quelconque", i);
                            Perim_Total_Quelconque = Perim_Total_Quelconque + 0;
                                }
print("Perim_Total_Quelconque="+Perim_Total_Quelconque);
// Recuperation de la valeur du perimetre des cubes
for (i=0; i< nResults; i++) {
                                Perim_Total_Cube = getResult("Perim_Total_Cube", i);
                                Perim_Total_Cube = Perim_Total_Cube + 0;
                               }
print("Perim_Total_Cube="+Perim_Total_Cube);
// CALCUL DU PERIMETRE TOTAL DES OBJETS
Perim_Total = Perim_Total_Cube + Perim_Total_Quelconque + Perim_Total_Triangle +
Perim_Total_Cylindre;
print("Perim_Total ="+Perim_Total);
// CALCUL DU POURCENTAGE DE CHAQUE FORME D'OBJET
// Pourcentage des cylindres
Pourcent_Cylindre = (Perim_Total_Cylindre/Perim_Total)*100;
// Pourcentage des triangles
Pourcent_Triangle = (Perim_Total_Triangle/Perim_Total)*100;
// Pourcentage des objets de forme quelconque
Pourcent_Quelconque = (Perim_Total_Quelconque/Perim_Total)*100;
// Pourcentage des cubes
Pourcent_Cube = (Perim_Total_Cube/Perim_Total)*100;
```

```
// Impression des pourcentages obtenus
print("Pourcent_Cylindre="+Pourcent_Cylindre+"%");
print("Pourcent_Triangle ="+Pourcent_Triangle+"%");
print("Pourcent_Quelconque ="+Pourcent_Quelconque+"%");
print("Pourcent_Cube ="+Pourcent_Cube+"%");

selectWindow("Results");close();
```

d) Exemple de traitement complet des données en utilisant une seule macro

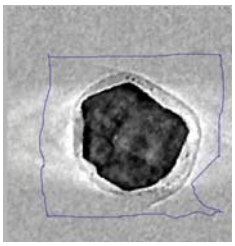
Le code source de cette macro est présenté dans l'annexe 8. C'est une macro globale, constituée de plusieurs macros décrites auparavant, qui permet d'effectuer une traitement intégral des données volumiques en commençant par la segmentation des données par seuillage et binarisation des images et en se terminant par la répartition des différentes faces présentes et le calcul des pourcentages correspondants. Ce traitement a été appliqué à une particule de catalyseur d'hydrogénation sélective au Pd en forme de bâtonnet à section pentagonale.

3) Macro utilisant des fonctions que nous avons définies : macro Rotation d'un objet cylindrique pour le mettre dans l'axe

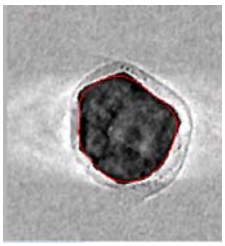
Il est possible, avec *ImageJ*, de définir ses propres fonctions. Ainsi, cette macro utilise certaines fonctions que nous avons définies. C'est aussi une macro outil, et elle utilise le plugin Rotate qui permet de faire la rotation d'un objet d'un angle arbitraire autour de l'axe X, Y ou Z. Elle nous permet d'exécuter ce plugin en cliquant sur l'image, et par rotations suivant les axes nécessaires de ramener l'axe du cylindre des catalyseurs d'hydrogénation sélective au Pd en forme de bâtonnet à section pentagonale, pour le faire coïncider avec l'axe Z. Le code source est présenté dans l'annexe 8. Un autre code source a été crée pour réaliser directement la rotation, en se basant sur le fait que, si l'objet est dans l'axe, la surface d'une section terminale de l'objet doit être minimale.

III.3 Résultats obtenus en utilisant les macros

1) Macro Snake

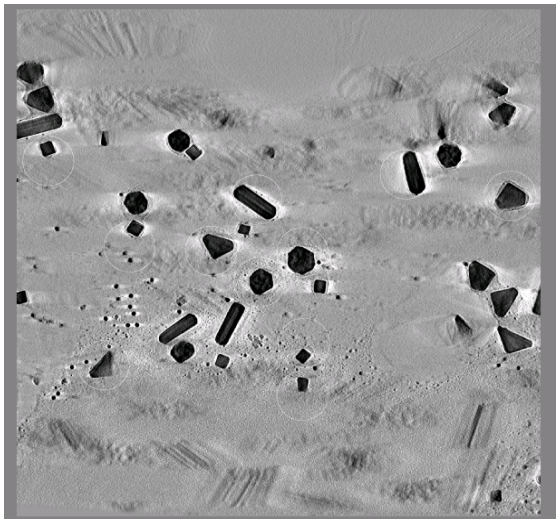


a) Sélection d'une partie de l'image (une section dans le volume reconstruit) contenant l'objet avant la segmentation.

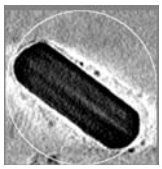


b) Résultat obtenu après segmentation. L'objet est bien identifié : ce traitement a été appliqué de manière automatique sur toutes les sections du volume contenant l'objet.

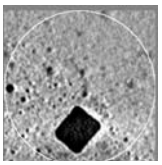
2) Macro Sélection des objets



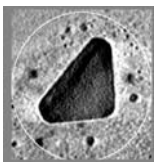
a) Volume à partir duquel on veut extraire des zones biens précises.



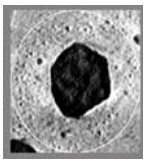
Objet Bâtonnet



Objet Cubique



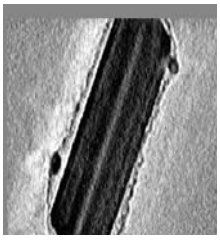
Objet Tétraédrique



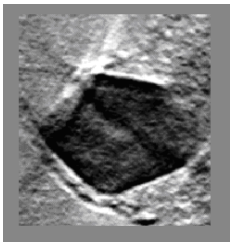
Objet « MTP »

b) *Volumes extraits autour des objets sélectionnés avec la souris.*

3) Macro Rotation d'un objet cylindrique pour le mettre dans l'axe

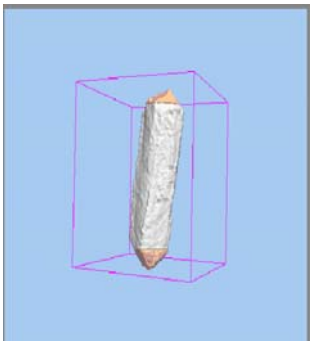


a) Image de départ



c) Image finale, après rotation

4) Macro Séparation des facettes



a) Coloration des différentes faces d'une particule de Pd en forme de bâtonnet

Surface Totale = $1{,}48*10^5$ $Pixels^2$
Surface Latérale = $1{,}33*10^5$ $Pixels^2$
Surfaces extrêmes = $0{,}14*10^5$ $Pixels^2$

Partie latérale = 90,4 %
Parties extrêmes = 9,6 %

Remarque : 1 Pixel = 1,12 nm

b) Résultat des estimations sur cet échantillon

5) Macro Calcul du pourcentage des différents objets

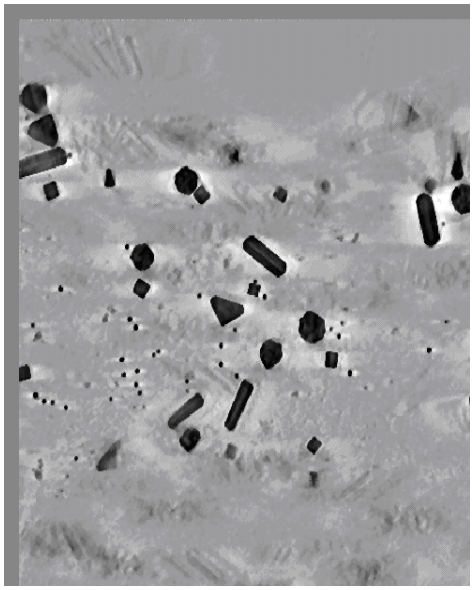
Résultat des estimations sur un échantillon contenant plusieurs types de particules de catalyseurs

Surface totale des bâtonnets = $1{,}72*10^4$ $Pixels^2$
Surface totale des tétraèdres = $4{,}05*10^4$ $Pixels^2$
Surface totale des MTP = $3{,}71*10^4$ $Pixels^2$
Surface totale des cubes = $2{,}23*10^4$ $Pixels^2$

Surface totale des objets = $1{,}17*10^5$ $Pixels^2$

Remarque : 1 Pixel = 1,932 nm

Pourcentage surfacique des bâtonnets = 14,7 %
Pourcentage surfacique des tétraèdres = 34,6 %
Pourcentage surfacique des MTP = 31,7 %
Pourcentage surfacique des cubes = 19,0 %



Section d'un échantillon type contenant plusieurs catalyseurs

6) Macro « Calcul de la porosité »

a) Image de départ

b) Segmentation de l'objet

c) Segmentation des pores

Porosité de l'échantillon 10,34 %

IV. Conclusion

Nous avons crée dans ce langage macro des routines qui, implémentées dans le logiciel de traitement d'images ImageJ, permettent d'analyser les données volumiques obtenus par reconstruction d'une série de projections enregistrées en microscopie électronique à transmission. Ces routines ont été appliquées à l'analyse des échantillons d'intérêt en catalyse hétérogène, thématique d'étude qui présente un intérêt majeur pour l'Institut Français du Pétrole qui était directement concerné par ce travail de stage.

Pour ces échantillons, la détermination des informations volumiques telles que la morphologie, la distribution et la proportion des faces non équivalentes du point de vue de la réactivité, la porosité (ouverte ou fermée) etc., est d'une importance cruciale pour comprendre leurs propriétés et leur comportement dans des conditions bien particulières.

Une première classe des macros a été réalisée dans le but de faciliter la manipulation et la gestion de ces données volumiques. Ensuite, nous nous sommes concentrés sur la création des macros permettant de détecter de manière automatique (sur toutes les sections d'un volume) les contours des objets d'intérêt ou des inhomogénéités contenues dans l'objet. Enfin, une fois les objets identifiés, la détermination des paramètres quantitatifs et leur présentation ont été réalisés. Les macros peuvent être utilisées individuellement ou contenue dans une macro globale permettant d'effectuer un traitement complet ayant comme fichier d'entrée, les données brutes et comme fichier de sortie le tableau des résultats.

Nous avons réussi à obtenir des premières valeurs pour les paramètres quantitatifs d'intérêt, et ceci pour plusieurs types d'échantillons. Ces valeurs seront comparées à celles obtenues en utilisant d'autres techniques d'investigation indirecte de la structure interne et de la morphologie de ces échantillons, comme par exemple la mesure de la porosité.

Ce type d'analyse des données en utilisant des macros implémentées dans un logiciel de traitement d'images présente plusieurs avantages : il permet l'extraction des paramètres quantitatifs à partir d'un volume et non d'une seule image ; l'analyse des données présente un caractère interactif grâce à la possibilité de création à tout moment et de manière simple des boîtes de dialogue ; les routines sont simples à écrire dans ce langage et faciles à comprendre et à modifier pour un autre utilisateur.

Ce stage était la première approche d'un travail qui sera continué par les chercheurs de l'IPCMS-GSI et l'IFP, afin de réaliser des routines encore plus développées en utilisant des algorithmes mathématiques complexes. Le but final étant d'obtenir de manière encore plus simple des valeurs tout à fait fiables pour les paramètres quantitatifs qui les intéressent.

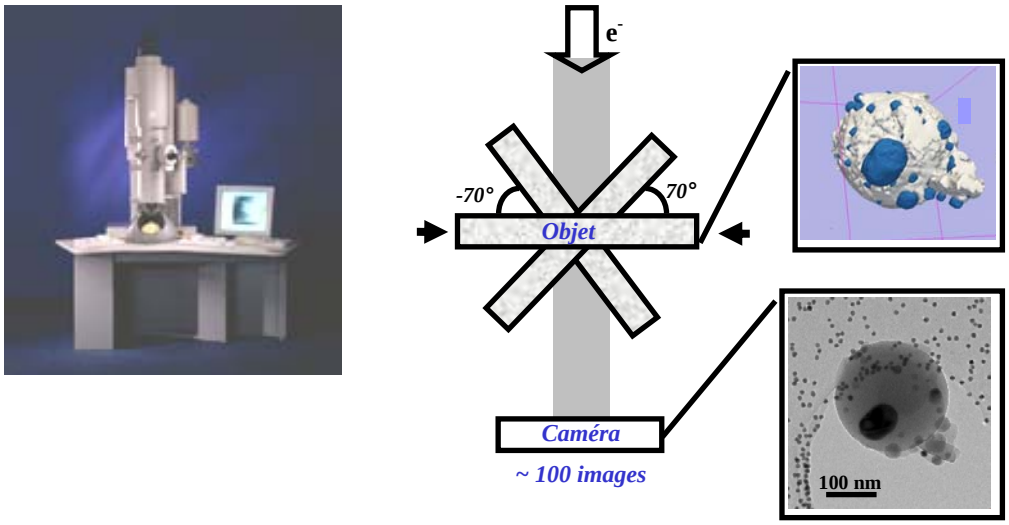
ANNEXES

Annexe 1 : Acquisition de la série

L'acquisition des données peut s'effectuer sur tout microscope qui possède un angle de tilt maximum élevé, une caméra et de préférence un canon FEG.

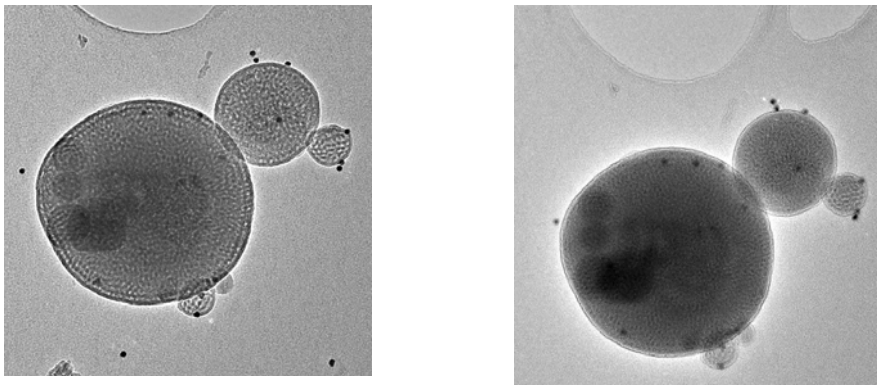
Tout d'abord, en utilisant la fonction de corrélation croisée entre deux images successives, le logiciel d'acquisition corrige les déplacements de l'objet dans les directions perpendiculaire et parallèle à l'axe de tilt provenant du fait que l'échantillon n'est pas tout à fait à la hauteur eucentrique et que le goniomètre présente des irrégularités mécaniques à cette échelle. Ensuite il corrige la mise au point sur l'échantillon en utilisant le principe du « *wobbler* » (en inclinant le faisceau, l'image se déplace et le déplacement est proportionnel à l'écart par rapport à la mise au point). Finalement il enregistre l'image et passe à l'angle de tilt suivant.

Pour pouvoir être corrigés par le logiciel d'acquisition automatique, ces déplacements en x, y (translation et rotation de l'image) et z (mise au point) doivent être relativement faibles, il faut donc tout d'abord déterminer le comportement général du porte-échantillon, lorsque l'angle de tilt varie, en utilisant une grille de référence (calibration du goniomètre).



a) Microscope FEI Tecnai G20 *b) Procédure d'acquisition*

Figure A1 : *Photo d'un microscope et illustration de la procédure d'acquisition.*



a) Première image acquise *b) Image suivante, sans corriger le déplacement et la mise au point*

Figure A2 : *Images acquises l'une après l'autre sans correction du déplacement et de la mise au point entre les deux.*

Annexe 2 : Alignement et corrections

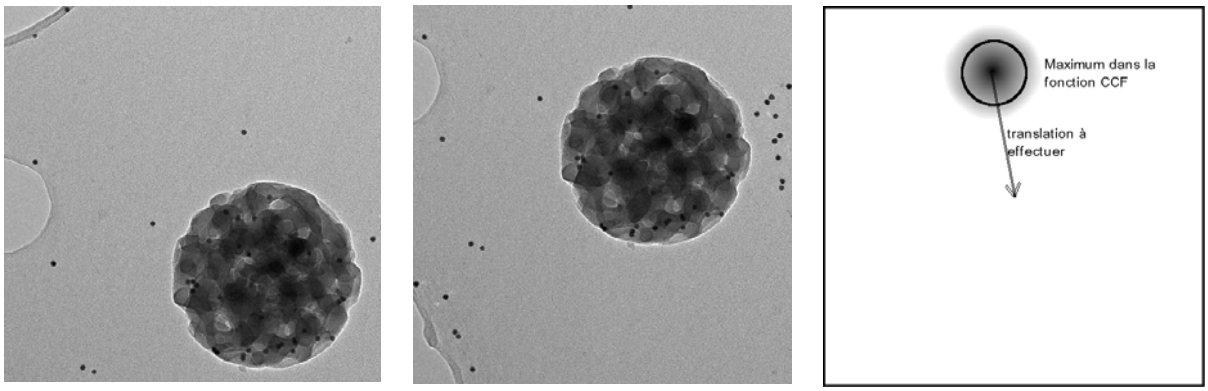
a) Alignement brut : Corrélation croisée

Il existe plusieurs méthodes pour aligner des images, la plus connue et la plus utilisée étant celle qui fait appel au calcul de la fonction de corrélation croisée (CCF) entre deux images. Cette fonction, matérialisée du point de vue mathématique par le produit de convolution entre deux images, est dans notre cas une mesure de la similarité entre deux images successives appartenant à la même série de projections (comme l'incrément angulaire est faible, les images sont presque identiques). Maximiser la fonction de corrélation croisée revient à chercher la position relative (en translation et en rotation) de la deuxième image par rapport à la première (considérée comme image de référence) pour avoir la meilleure similitude entre les deux. La position du pic correspondant au maximum dans la représentation bi-dimensionnelle de la fonction fournit le déplacement à effectuer pour que les deux images soient alignées.

La fonction de corrélation croisée pour l'opération de translation est définie de la manière suivante : $CCF(\vec{r}) = \sum_{j=1}^{N} I_1(\vec{R}_j) I_2(\vec{R}_j + \vec{r})$

Où : $I_1(\vec{R}_j)$ est l'image de référence,

$I_2(\vec{R}_j + \vec{r})$ l'image à aligner et *N* le nombre de pixels dans chaque image.



a) Image de référence *b) Image à aligner* *c) Fonction de corrélation*

***Figure A3** : Deux images acquises consécutivement et leur fonction de corrélation.*

b) Alignement fin et corrections : Modèle géométrique

Il est important de préciser que cet alignement effectué à l'aide de la fonction CCF n'est jamais parfait.

En effet ; d'une part, les projections successives sont légèrement différentes et les maxima de la série de fonctions CCF ne donnent pas nécessairement une origine commune pour toute la série. D'autre part, quelquefois ces maxima sont imposés par des détails existant sur l'image (comme les zones très contrastées) qui ne font pas partie de l'objet à étudier. Dans ce cas, les images seront alignées en fonction de ces détails et non de l'objet. Il faut donc mettre en place une procédure d'alignement plus fin à l'aide d'un modèle géométrique ancré aux images.

Du point de vue pratique, ceci revient à se fixer des repères sur une image (de préférence l'image centrale) et à les suivre tout au long de la série. La manière la plus simple est de déposer sur l'objet et sur la membrane qui supporte l'objet des billes d'or de taille nanométrique. En choisissant certaines billes nous pouvons définir le modèle à suivre qui nous permettra finalement d'aligner les images d'une manière plus précise. La taille des billes est choisie en fonction du grandissement auquel l'acquisition sera faite. L'avantage de ce type de repère est lié à la forme sphérique des billes (valable pour tout angle d'observation).

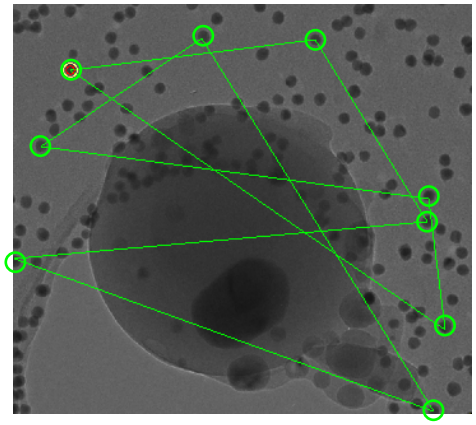


Figure A4 : *Modèle géométrique défini sur l'image centrale de la série de projections (angle de tilt zéro) en sélectionnant plusieurs billes d'or de taille 10 nm, initialement déposées sur la membrane. Cette image a été enregistrée à G = 25.000.*

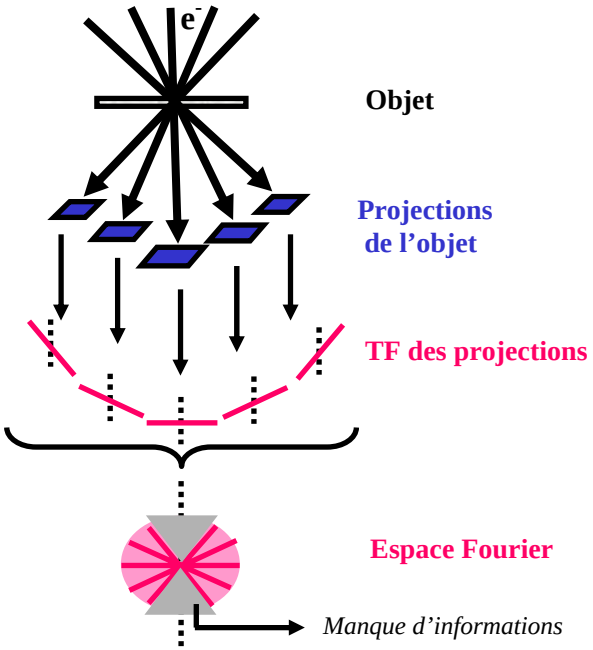
Par ailleurs, en utilisant ce même modèle, il est possible d'effectuer les autres corrections énumérées ci-dessus en obtenant à la fin une nouvelle série d'images alignées et corrigées, prête à être utilisée pour le calcul du volume.

Annexe 3 : Contraintes imposées par l'acquisition et la résolution

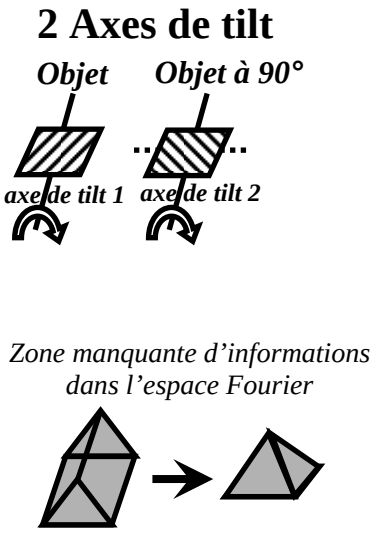
Comme l'intervalle balayé n'est jamais complet (d'une part, l'angle de tilt est limité et d'autre part, l'échantillon devient trop épais), il reste deux zones appelée « zones aveugles » où on n'aura pas d'informations sur l'objet (voir figure A5). La présence de ces zones a une influence néfaste sur la reconstruction. Cette influence est matérialisée par une perte de résolution et par une distorsion de l'objet dans la direction de l'axe de tilt. Pour diminuer la taille de ces deux zones (qui sont en fait deux prismes), il est possible d'enregistrer deux séries de projections, en tournant l'objet de 90° autour de sa normale. Cela permet de réduire ces zones aveugles à deux pyramides, mais le désavantage est qu'on obtient deux séries d'images équivalentes. Ces séries vont générer deux volumes reconstruits qu'il faut savoir recombiner par la suite pour réduire effectivement les distorsions.

Une autre méthode d'acquisition consiste à faire tourner l'échantillon autour d'un axe oblique par rapport à la direction du faisceau électronique. Dans cette géométrie les distorsions dans les reconstructions sont moins importantes car il est possible de recouvrir l'intervalle angulaire complet, mais il faut néanmoins disposer d'un goniomètre et d'un porte-échantillon adapté.

Concernant la résolution finale, elle dépend fortement du nombre de projections acquises et de l'intervalle balayé. Si la résolution dans la direction parallèle à l'axe de tilt R_x est celle des images 2D, les deux autres résolutions peuvent être approximées par la relation : $R_z = R_y = \frac{\pi \cdot D}{N}$ où D est le diamètre de l'échantillon et N le nombre de projections. De plus, si l'intervalle de tilt est limité, la résolution R_z est dégradée par un facteur supplémentaire qui dépend de l'angle de tilt maximum (*P.A. Midgley, M. Weyland, Ultramicroscopy 96 (2003), 413*).



Remplissage de l'espace de Fourier de l'objet à partir de la série de projections.



Méthode pour réduire la taille des zones « aveugles » : acquisition de deux séries d'images sur deux axes de tilt perpendiculaires.

***Figure A5** : Illustration de l'espace de Fourier et des zones « aveugles ».*

Annexe 4 : Les classes d'*ImageJ*

La classe principale est *ij*, de celle-ci dérive tout le reste.

ij

ImageJApplet

Classe nécessaire pour faire tourner ImageJ dans un navigateur. L'avantage est qu'une installation n'est pas nécessaire, le désavantage est un accès limité aux ressources de la machine à cause de l'aspect sécurité des applets Java.

ImageJ

La classe principale d'une application ImageJ. Elle contient la méthode *run*, principal point d'entrée du programme, et la fenêtre principale d'ImageJ.

Executer

Une classe pour exécuter les commandes des menus dans des threads séparés (pour ne pas bloquer le reste du programme).

IJ

Une classe qui contient beaucoup d'utilitaires.

ImagePlus

La représentation d'une image dans ImageJ, basée sur un *ImageProcessor*.

ImageStack

Une pile d'images de taille variable.

WindowManager

L'ensemble des fenêtres ouvertes.

ij.gui

ProgressBar

Une barre qui informe graphiquement de la progression d'une opération.

GenericDialog

Une boîte de dialogue modulaire pour interagir avec l'utilisateur.

NewImage

Pour créer de nouvelles images en partant de zéro.

Roi

Une classe qui représente une région d'intérêt d'une image. Si le plugin le supporte, on peut traiter la région d'intérêt seule et non pas toute l'image.

ImageCanvas

Un support pour dessiner une image, dérivé de *java.awt.Canvas*.

ImageWindow

Une fenêtre pour afficher une image, dérivée de *java.awt.Frame*.

StackWindow

Un *ImageWindow* pour les stacks.

HistogramWindow

Un *ImageWindow* pour les histogrammes.

PlotWindow

Un *ImageWindow* pour les tracés de courbes.

ij.io

Un paquet de gestion des entréessorties,
à savoir toutes les classes pour lire et écrire les images.

ij.measure

Les classes relatives aux mesures.

ij.plugin

La plupart des commandes ImageJ sont des plugins et se retrouvent donc dans cette classe ou ses dérivées.

PlugIn

Cette interface doit être implémentée par tout plugin qui n'a pas besoin d'image en entrée.

Converter

Implémentation pour convertir correctement des *ImagePlus* d'un type vers un autre.

ij.plugin.filter

PluginFilter

Cette interface doit être implémentée par tout plugin qui nécessite une image en entrée (qui effectue donc une opération sur cette image).

ij.process

ImageConverter

Une classe contenant les méthodes pour convertir les images d'un type dans un autre.

ImageProcessor

Une superclasse abstraite pour certains types d'images. Un *imageProcessor* contient les méthodes permettant de travailler réellement sur les données contenues dans l'image.

StackConverter

Le pendant de *ImageConverter* pour les stacks.

StackProcessor

Le pendant de *ImageProcessor* pour les stacks.

ij.text

Paquet qui contient les classes pour afficher et éditer du texte.

Annexe 5 : Fonctions appropriées à *ImageJ*

A

abs(n)

Returns the absolute value of *n*.

acos(n)

Returns the arc cosine (in radians) of *n*. Requires 1.34h.

asin(n)

Returns the arc sine (in radians) of *n*. Requires 1.34h.

atan(n)

Calculates the inverse tangent (arc tangent) of *n*. Returns a value in the range -PI/2 through PI/2.

atan2(y, x)

Calculates the arctangent of *y/x* and returns an angle in the range -PI to PI, using the signs of the arguments to determine the quadrant. Multiply the result by 180/PI to convert to degrees.

autoUpdate(boolean)

If *boolean* is true, the display is refreshed each time lineTo(), drawLine(), drawString(), etc. are called, otherwise, the display is refreshed only when updateDisplay() is called or when the macro terminates.

B

beep()

Emits an audible beep.

bitDepth()

Returns the bit depth of the active image: 8, 16, 24 (RGB) or 32 (float).

C

calibrate(value)

Uses the calibration function of the active image to convert a raw pixel value to a density calibrated value. The argument must be an integer in the range 0-255 (for 8-bit images) or 0-65535 (for 16-bit images). Returns the same value if the active image does not have a calibration function. Requires 1.34h.

call("class.method", arg1, arg2, ...)

Calls a public static method in a Java class, passing an arbitrary number of string arguments, and returning a string. Refer to CallJavaDemo for an example.

changeValues(v1, v2, v3)

Changes pixels in the image or selection that have a value in the range *v1-v2* to *v3*. For example, *changeValues(0,5,5)* changes all pixels less than 5 to 5, and *changeValues(0x0000ff,0x0000ff,0xff0000)* changes all blue pixels in an RGB image to red.

charCodeAt(string, index)

Returns the Unicode value of the character at the specified index in *string*. Index values can range from 0 to lengthOf(*string*). Use the fromCharCode() function to convert one or more Unicode characters to a string.

close()

Closes the active image. This function has the advantage of not closing the "Log" or "Results" window when you meant to close the active image.

cos(angle)

Returns the cosine of an angle (in radians).

D

d2s(n, decimalPlaces)

Converts the number *n* into a string using the specified number of decimal places. Note that d2s stands for "double to string". This function will probably be replaced by one with a better name.

Dialog.create("Title")

Creates a dialog box with the specified title. Call *Dialog.addString()*, *Dialog.addNumber(),* etc. to add components to the dialog. Call *Dialog.show()* to display the dialog and *Dialog.getString()*, *Dialog.getNumber()*, etc. to retrieve the values entered by the user. Refer to the DialogDemo macro for an example. Requires 1.34m.

Dialog.addMessage(string) - Adds a message to the dialog. The message can be broken into multiple lines by inserting new line characters ("\n") into the string.

Dialog.addString("Label", "Initial text") - Adds a text field to the dialog, using the specified label and intial text.

Dialog.addNumber("Label", default) - Adds a numeric field to the dialog, using the specified label and default value.

Dialog.addNumber("Label", default, decimalPlaces, columns, units) - Adds a numeric field, using the specified label and default value. *DecimalPlaces* specifies the number of digits to right of decimal point, *columns* specifies the the field width in characters and units is a string that is displayed to the right of the field. Requires 1.35b.

Dialog.addCheckbox("Label", default) - Adds a checkbox to the dialog, using the specified label and default state (true or false).

Dialog.addChoice("Label", items) - Adds a popup menu to the dialog, where *items* is a string array containing the menu items.

Dialog.addChoice("Label", items, default) - Adds a popup menu, where *items* is a string array containing the choices and *default* is the default choice.

Dialog.show() - Displays the dialog and waits until the user clicks "OK" or "Cancel". The macro terminates if the user clicks "Cancel".

Dialog.getString() - Returns a string containing the contents of the next text field.

Dialog.getNumber() - Returns the contents of the next numeric field.

Dialog.getCheckbox() - Returns the state (true or false) of the next checkbox.

Dialog.getChoice() - Returns the selected item (a string) from the next popup menu.

doCommand("Command")

Runs an ImageJ menu command in a separate thread and returns immediately. As an example, "doCommand('Start Animation [=]')" starts animating the current stack in a separate thread and the macro continues to execute. Use "run(doCommand('Start Animation [=]')" and the macro hangs until the user stops the animation.

doWand(x, y)

Equivalent to clicking on the current image at (x,y) with the wand tool. Note that some objects, especially one pixel wide lines, may not be reliably traced unless they have been thresholded (highlighted in red) using setThreshold.

drawLine(x1, y1, x2, y2)

Draws a line between (x1, y1) and (x2, y2). Use setColor() to specify the color of the line and setLineWidth() to vary the line width.

drawOval(x, y, width, height)

Draws the outline of an oval using the current color and line width. See also: fillOval, setColor, setLineWidth, autoUpdate. Requires 1.37e.

drawRect(x, y, width, height)

Draws the outline of a rectangle using the current color and line width. See also: fillRect, setColor, setLineWidth, autoUpdate. Requires 1.37e.

drawString("text", x, y)

Draws text at the specified location. Call setFont() to specify the font. Call setJustification() to have the text centered or right justified. Refer to the TextDemo macro for examples.

dump()

Writes the contents of the symbol table, the tokenized macro code and the variable stack to the "Log" window.

E

endsWith(string, suffix)

Returns *true* (1) if *string* ends with *suffix*. See also: startsWith, indexOf, substring.

eval(macro)

Evaluates (runs) one or more lines of macro code. See also: EvalDemo macro and runMacro function. Requires v1.34g.

exit() or exit("error message")

Terminates execution of the macro and, optionally, displays an error message.

exp(n)

Returns the exponential number e (i.e., 2.718...) raised to the power of *n*.

F

File functions

These functions allow you to get information about files, to delete files and directories, and to create directories. The FileDemo macro demonstrates how to use them. See also: getDirectory and getFileList. Requires 1.35g.

File.exists(path) - Returns *true* if the specified file exists.
File.getName(path) - Returns the last name in *path*'s name sequence.
File.getParent(path) - Returns the parent of the file specified by *path*.
File.getAbsolutePath(path) - Returns the full path of the file specified by *path*.
File.isDirectory(path) - Returns *true* if the specified file is a directory.
File.length(path) - Returns the length in bytes of the specified file.
File.makeDirectory(path) - Creates a directory.
File.dateLastModified(path) - Returns the date and time the specified file was last modified.
File.lastModified(path) - Returns the time the specified file was last modified as the number of milliseconds since January 1, 1970.
File.delete(path) - Deletes the specified file. If the file is a directory, it must be empty. The file must be in the ImageJ or temp directory.
File.separator - Returns the file name separator character ("/" or "\").
File.openAsString(path) - Opens a text file and returns the contents as a string. Displays a file open dialog box if *path* is an empty string. Use *lines=split(str,"\n")* to convert the string to an array of lines. Requires 1.35r.

File.open(path) - Creates a new text file and returns a file variable that refers to it. To write to the file, pass the file variable to the print() function. Displays a file save dialog box if *path* is an empty string. The file is closed when the macro exits. For an example, refer to the OpenFileDemo macro. Requires 1.35r.

File.close(f) - Closes the specified file, which must have been opened using File.open(). Requires 1.35r.

fill()

Fills the image or selection with the current drawing color.

fillOval(x, y, width, height)

Fills an oval bounded by the specified rectangle with the current drawing color. See also: drawOval, setColor, autoUpdate. Requires 1.37e.

fillRect(x, y, width, height)

Fills the specified rectangle with the current drawing color. See also: drawRect, setColor, autoUpdate. Requires 1.34g.

floodFill(x, y)

Fills, with the foreground color, pixels that are connected to, and the same color as, the pixel at *(x, y)*. With 1.37e or later, does 8-connected filling if there is an optional string argument containing "8", for example *floodFill(x, y, "8-connected")*. This function is used to implement the flood fill (paint bucket) macro tool. Requires 1.34j.

floor(n)

Returns the largest value that is not greater than *n* and is equal to an integer. See also: round.

fromCharCode(value1,...,valueN)

This function takes one or more Unicode values and returns a string.

G

getArgument()

Returns the string argument passed to macros called by runMacro(macro, arg), eval(macro), *IJ.runMacro(macro, arg)* or *IJ.runMacroFile(path, arg)*.

getBoolean("message")

Displays a dialog box containing the specified message and "Yes", "No" and "Cancel" buttons. Returns *true* (1) if the user clicks "Yes", returns *false* (0) if the user clicks "No" and exits the macro if the user clicks "Cancel".

getBoundingRect(x, y, width, height)

Replace by getSelectionBounds.

getCursorLoc(x, y, z, modifiers)

Returns the cursor location in pixels and the mouse event modifier flags. The *z* coordinate is zero for 2D images. For stacks, it is one less than the slice number. For examples, see the GetCursorLocDemo and the GetCursorLocDemoTool macros.

getDateAndTime(year, month, dayOfWeek, dayOfMonth, hour, minute, second, msec)

Returns the current date and time. For an example, refer to the GetDateAndTime macro. See also: getTime. Requires v1.34n or later.

getDirectory(title)

Returns the path to a specified directory. If *title* is "startup", returns the path to the directory that ImageJ was launched from (usually the ImageJ directory). If it is "plugins" or "macros", returns the path to the plugins or macros folder. If it is "image", returns the path to the directory that the active image was loaded from. If it is "home", returns the path to users home directory. If it is "temp", returns the path to the /tmp directory. Otherwise, displays a dialog (with *title* as the title), and returns the path to the directory selected by the user. Returns an empty string if the specified directory is not found or aborts the macro if the user cancels the dialog box. For examples, see the GetDirectoryDemo and ListFilesRecursively macros.

getFileList(directory)

Returns an array containing the names of the files in the specified directory path. The names of subdirectories have a "/" appended. For an example, see the ListFilesRecursively macro.

getImageID()

Returns the unique ID (a negative number) of the active image. Use the *selectImage(id)*, *isOpen(id)* and *isActive(id)* functions to activate an image or to determine if it is open or active.

getImageInfo()

Returns a string containing the text that would be displayed by the *Image/Show Info* command. To retrieve the contents of a text window, use getInfo(). For an example, see the ListDicomTags macros.

getInfo()

If the front window is a text window, returns the contents of that window. If the front window is an image, returns a string containing the text that would be displayed by *Image/Show Info*. Note that getImageInfo() is a more reliable way to retrieve information about an image. Use split(getInfo(),'\n') to break the string returned by this function into separate lines.

getLine(x1, y1, x2, y2, lineWidth)

Returns the starting coordinates, ending coordinates and width of the current straight line selection. Sets x1 = -1 if there is no line selection.

getLocationAndSize(x, y, width, height)

Returns the location and size of the active image window. Use getWidth and getHeight to get the width and height of the active image. See also: setLocation, Requires v1.34k or later.

getHeight()

Returns the height in pixels of the current image.

getHistogram(values, counts, nBins[, histMin, histMax])

Returns the histogram of the current image or selection. *Values* is an array that will contain the pixel values for each of the histogram counts (or the bin starts for 16 and 32 bit images), or set this argument to 0. *Counts* is an array that will contain the histogram counts. *nBins* is the number of bins that will be used. It must be 256 for 8 bit and RGB image, or an integer greater than zero for 16 and 32 bit images. With 16-bit images, the *Values* argument is ignored if *nBins* is 65536. With 16-bit and 32-bit images, and ImageJ 1.35a and later, the

histogram range can be specified using optional *histMin* and *histMax* arguments. See also: getStatistics, HistogramLister, HistogramPlotter, StackHistogramLister and CustomHistogram.

getLut(reds, greens, blues)

Returns three arrays containing the red, green and blue intensity values from the current lookup table. See the LookupTables macros for examples.

getMetadata()

Returns the metadata (a string) associated with the current image or stack slice. With stacks, the first line of the metadata is the slice label. With DICOM images and stacks, returns the metadata from the DICOM header. See also: setMetadata. Requires v1.34q.

getMinAndMax(min, max)

Returns the minimum and maximum displayed pixel values (display range). See the DisplayRangeMacros for examples.

getNumber("prompt", defaultValue)

Displays a dialog box and returns the number entered by the user. The first argument is the prompting message and the second is the value initially displayed in the dialog. Exits the macro if the user clicks on "Cancel" in the dialog. Returns *defaultValue* if the user enters an invalid number. See also: Dialog.create.

getPixel(x, y)

Returns the value of the pixel at *(x,y)*. Note that pixels in RGB images contain red, green and blue components that need to be extracted using shifting and masking. See the Color Picker Tool macro for an example that shows how to do this.

getPixelSize(unit, pixelWidth, pixelHeight)

Returns the unit of length (as a string) and the pixel dimensions. For an example, see the ShowImageInfo macro. See also: getVoxelSize.

getProfile()

Runs *Analyze/Plot Profile* (without displaying the plot) and returns the intensity values as an array. For an example, see the GetProfileExample macro.

getRawStatistics(nPixels, mean, min, max, std, histogram)

This function is similar to getStatistics except that the values returned are uncalibrated and the histogram of 16-bit images has a bin width of one and is returned as a *max*+1 element array. For examples, refer to the ShowStatistics macro set. See also: calibrate. Requires 1.34h.

getResult("Column", row)

Returns a measurement from the ImageJ results table or NaN if the specified column is not found. The first argument specifies a column in the table. It must be a "Results" window column label, such as "Area", "Mean" or "Circ.". The second argument specifies the row, where 0<=*row*<nResults. *nResults* is a predefined variable that contains the current measurement count. (Actually, it's a built-in function with the "()" optional.) With ImageJ 1.34g and later, you can omit the second argument and have the row default to nResults-1 (the last row in the results table). See also: nResults, setResult, isNaN, getResultLabel.

getResultLabel(row)

Returns the label of the specified row in the results table or an empty string if the row does not have a label. Requires v1.35r or later.

getSelectionBounds(x, y, width, height)

Returns the smallest rectangle that can completely contain the current selection. *x* and *y* are the pixel coordinates of the upper left corner of the rectangle. *width* and *height* are the width and height of the rectangle in pixels. If there is no selection, returns (0, 0, ImageWidth, ImageHeight). See also: selectionType. Requires v1.34g or later.

getSelectionCoordinates(xCoordinates, yCoordinates)

Returns two arrays containing the X and Y coordinates of the points that define the current selection. See the SelectionCoordinates macro for an example. See also: selectionType, getSelectionBounds.

getSliceNumber()

Returns the number of the currently displayed stack slice, an integer between 1 and nSlices. Returns 1 if the active image is not a stack. See also: setSlice. Requires 1.34g.

getString("prompt", "default")

Displays a dialog box and returns the string entered by the user. The first argument is the prompting message and the second is the initial string value. Exits the macro if the user clicks on "Cancel" or enters an empty string. See also: Dialog.create.

getStatistics(area, mean, min, max, std, histogram)

Returns the area, average pixel value, minimum pixel value, maximum pixel value, standard deviation of the pixel values and histogram of the active image or selection. The histogram is returned as a 256 element array. For 8-bit and RGB images, the histogram bin width is one. For 16-bit and 32-bit images, the bin width is (*max-min*)/256. For examples, refer to the ShowStatistics macro set. Note that trailing arguments can be omitted. For example, you can use *getStatistics(area)*, *getStatistics(area, mean)* or *getStatistics(area, mean, min, max)*. See also: getRawStatistics Requires 1.34h.

getThreshold(lower, upper)

Returns the lower and upper threshold levels. Both variables are set to -1 if the active image is not thresholded. See also: setThreshold, getThreshold, resetThreshold.

getTime()

Returns the current time in milliseconds. The granularity of the time varies considerably from one platform to the next. On Windows NT, 2K, XP it is about 10ms. On other Windows versions it can be as poor as 50ms. On many Unix platforms, including Mac OS X, it actually is 1ms. See also: getDateAndTime.

getTitle()

Returns the title of the current image.

getVoxelSize(width, height, depth, unit)

Returns the voxel size and unit of length ("pixel", "mm", etc.) of the current image or stack. See also: getPixelSize, setVoxelSize.

getVersion()

Returns the ImageJ version number as a string (e.g., "1.34s"). Requires v1.35m. See also: requires.

getWidth()

Returns the width in pixels of the current image.

getZoom()

Returns the magnification of the active image, a number in the range 0.03125 to 32.0 (3.1% to 3200%).

I

imageCalculator(operator, img1, img2)

Runs the *Process/Image Calculator* tool, where *operator* ("add","subtract","multiply","divide", "and", "or", "xor", "min", "max", "average", "difference" or "copy") specifies the operation, and *img1* and *img2* specify the operands. *img1* and *img2* can be either an image title (a string) or an image ID (an integer). The *operator* string can include up to three modifiers: "create" (e.g., "add create") causes the result to be stored in a new window, "32-bit" causes the result to be 32-bit floating-point and "stack" causes the entire stack to be processed. See the ImageCalculatorDemo macros for examples. Requires v1.34r.

indexOf(string, substring)

Returns the index within *string* of the first occurrence of *substring*. See also: lastIndexOf, startsWith, endsWith, substring, toLowerCase, replace.

indexOf(string, substring, fromIndex)

Returns the index within *string* of the first occurrence of *substring*, with the search starting at *fromIndex*.

isActive(id)

Returns *true* if the image with the specified ID is active.

isKeyDown(key)

Returns *true* if the specified key is pressed, where *key* must be "shift", "alt" or "space". See also: setKeyDown.

isNaN(n)

Returns *true* if the value of the number *n* is NaN (Not-a-Number). A common way to create a NaN is to divide zero by zero. This function is required because (n==NaN) is always false. Note that the numeric constant NaN is predefined in the macro language.

isOpen(id)

Returns *true* if the image with the specified ID is open.

isOpen("Title")

Returns *true* if the window with the specified title is open.

L

lastIndexOf(string, substring)

Returns the index within *string* of the rightmost occurrence of *substring*. See also: indexOf, startsWith, endsWith, substring.

lengthOf(str)

Returns the length of a string or array.

lineTo(x, y)

Draws a line from current location to (*x*,*y*) .

log(n)

Returns the natural logarithm (base e) of *n*. Note that log10(n) = log(n)/log(10).

M

makeLine(x1, y1, x2, y2)

Creates a new straight line selection. The origin (0,0) is assumed to be the upper left corner of the image. Coordinates are in pixels. With ImageJ 1.35b and letter, you can create segmented line selections by specifying more than two coordinate, for example *makeLine(25,34,44,19,69,30,71,56)*.

makeOval(x, y, width, height)

Creates an elliptical selection, where (*x*,*y*) define the upper left corner of the bounding rectangle of the ellipse.

makePolygon(x1, y1, x2, y2, x3, y3, ...)

Creates a polygonal selection. At least three coordinate pairs must be specified, but not more than 200. As an example, *makePolygon(20,48,59,13,101,40,75,77,38,70)* creates a polygon selection with five sides.

makeRectangle(x, y, width, height)

Creates a rectangular selection. The *x* and *y* arguments are the coordinates (in pixels) of the upper left corner of the selection. The origin (0,0) of the coordinate system is the upper left corner of the image.

makeSelection(type, xcoord, ycoord)

Creates a selection from a list of XY coordinates. The first argument should be "polygon", "freehand", "polyline", "freeline", "angle" or "point". In ImageJ 1.32g or later, it can also be the numeric value returned by selectionType. The *xcoord* and *ycoord* arguments are numeric arrays that contain the X and Y coordinates. See the MakeSelectionDemo macro for examples.

maxOf(n1, n2)

Returns the greater of two values.

minOf(n1, n2)

Returns the smaller of two values.

moveTo(x, y)

Sets the current drawing location. The origin is always assumed to be the upper left corner of the image.

N

newArray(size)

Returns a new array containing *size* elements. You can also create arrays by listing the elements, for example newArray(1,4,7) or newArray("a","b","c"). For more examples, see the Arrays macro.

The ImageJ macro language does not directly support 2D arrays. As a work around, either create a blank image and use setPixel() and getPixel(), or create a 1D array using a=newArray(xmax*ymax) and do your own indexing (e.g., value=a[x+y*xmax]).

newImage(title, type, width, height, depth)

Opens a new image or stack using the name *title*. The string *type* should contain "8-bit", "16-bit", "32-bit" or "RGB". In addition, it can contain "white", "black" or "ramp" (the default is "white"). As an example, use "16-bit ramp" to create a 16-bit image containing a grayscale ramp. *Width* and *height* specify the width and height of the image in pixels. *Depth* specifies the number of stack slices.

nImages

Returns number of open images. The parentheses "()" are optional.

nResults

Returns the current measurement counter value. The parentheses "()" are optional.

nSlices

Returns the number of slices in the current stack. Returns 1 if the current image is not a stack. The parentheses "()" are optional. See also: getSliceNumber,

O

open(path)

Opens and displays a tiff, dicom, fits, pgm, jpeg, bmp, gif, lut, roi, or text file. Displays an error message and aborts the macro if the specified file is not in one of the supported formats, or if the file is not found. Displays a file open dialog box if *path* is an empty string or if there is no argument. Use the *File/Open* command with the command recorder running to generate calls to this function.

P

parseFloat(string)

Converts the string argument to a number and returns it. Returns NaN (Not a Number) if the string cannot be converted into a number. Use the isNaN() function to test for NaN. For examples, see ParseFloatIntExamples.

parseInt(string)

Converts *string* to an integer and returns it. Returns NaN if the string cannot be converted into a integer.

parseInt(string, radix)

Converts *string* to an integer and returns it. The optional second argument (*radix*) specifies the base of the number contained in the string. The radix must be an integer between 2 and 36. For radixes above 10, the letters of the alphabet indicate numerals greater than 9. Set *radix* to 16 to parse hexadecimal numbers. Returns NaN if the string cannot be converted into a integer. For examples, see ParseFloatIntExamples.

Plot.create("Title", "X-axis Label", "Y-axis Label", xValues, yValues)

Generates a plot using the specified title, axis labels and X and Y coordinate arrays. If only one array is specified it is assumed to contain the Y values and a 0..n-1 sequence is used as the X values. It is also permissible to specify no arrays and use *Plot.setLimits()* and *Plot.add()* to generate the plot. Use *Plot.show()* to display the plot in a window or it will be displayed automatically when the macro exits. For examples, check out the ExamplePlots macro file.

Plot.setLimits(xMin, xMax, yMin, yMax)

Sets the range of the x-axis and y-axis of plots created using *Plot.create()*.

Plot.setLineWidth(width)

Specifies the width of the line used to draw a curve. Points (circle, box, etc.) are also drawn larger if a line width greater than one is specified. Note that the curve specified in *Plot.create()* is the last one drawn before the plot is dispayed or updated.

Plot.setColor("red")

Specifies the color used in subsequent calls to *Plot.add()* or *Plot.addText()*. The argument can be "black", "blue", "cyan", "darkGray", "gray", "green", "lightGray", "magenta", "orange", "pink", "red", "white" or "yellow". Note that the curve specified in *Plot.create()* is drawn last.

Plot.add("circles", xValues, yValues)

Adds a curve, set of points or error bars to a plot created using *Plot.create()*. If only one array is specified it is assumed to contain the Y values and a 0..n-1 sequence is used as the X values. The first argument can be "line", "circles", "boxes", "triangles", "crosses", "dots", "x" or "error bars".

Plot.addText("A line of text", x, y)

Adds text to the plot at the specified location, where (0,0) is the upper left corner of the the plot frame and (1,1) is the lower right corner. Call *Plot.setJustification()* to have the text centered or right justified.

Plot.setJustification("center")

Specifies the justification used by *Plot.addText()*. The argument can be "left", "right" or "center". The default is "left".

Plot.show()

Displays the plot generated by *Plot.create()*, *Plot.add()*, etc. in a window. This function is automatically called when a macro exits.

Plot.update()

Draws the plot generated by *Plot.create()*, *Plot.add()*, etc. in an existing plot window. Equivalent to *Plot.show()* if no plot window is open.

pow(base, exponent)

Returns the value of *base* raised to the power of *exponent*.

print(string)

Outputs a string to the "Log" window. Numeric arguments are automatically converted to strings. Starting with v1.34b, this function accepts multiple arguments. For example, you can use *print(x,y,width, height)* instead of *print(x+" "+y+" "+width+" "+height)*. Refer to the SineCosineTable macro for an example of how to save the contents of the "Log" window to a text file.

R

random

Returns a random number between 0 and 1.

rename(name)

Changes the title of the active image to the string *name*.

replace(string, old, new)

Returns the new string that results from replacing all occurrences of *old* in *string* with *new*, where *old* and *new* are single character strings. With Java 1.4 and later, replaces each substring of *string* that matches the regular expression *old* with *new*. Requires 1.34h.

requires("1.29p")

Display a message and aborts the macro if the ImageJ version is less than the one specified. See also: getVersion.

reset

Restores the backup image created by the snapshot function. Note that reset() and run("Undo") are basically the same, so only one run() call can be reset.

resetMinAndMax

Resets the minimum and maximum displayed pixel values (display range) to be the same as the current image's minimum and maximum pixel values. See the DisplayRangeMacros for examples.

resetThreshold

Disables thresholding. See also: setThreshold, setAutoThreshold, getThreshold.

restorePreviousTool

Switches back to the previously selected tool. Useful for creating a tool macro that performs an action, such as opening a file, when the user clicks on the tool icon.

restoreSettings

Restores *Edit/Options* submenu settings saved by the *saveSettings()* function.

roiManager(cmd)

Runs an ROI Manager command, where *cmd* must be "Add", "Add & Draw", "Deselect", "Measure", "Draw" or "Combine". The ROI Manager is opened if it is not already open. With v1.34k or later, use *roiManager("reset")* to delete all items on the list. For examples, refer to the RoiManagerMacros, RoiManagerAddParticles and ROI Manager Stack Demo macros.

roiManager(cmd, name)

Runs an ROI Manager I/O command, where *cmd* is "Open", "Save or "Rename", and *name* is a file path or name. With v1.35c or later, the "Save" option ignores selections and saves all the ROIs as a ZIP archive. The "Rename" option requires v1.37h or later. With v1.37i or later, you can get the selection name using *call("ij.plugin.frame.RoiManager.getName", index)*. The ROI Manager is opened if it is not already open.

roiManager("count")

Returns the number of items in the ROI Manager list. Requires v1.34k.

roiManager("index")

Returns the index of the currently selected item on the ROI Manager list, or -1 if the list is empty, no items are selected, or more than one item is selected. Requires v1.37q.

roiManager("select", index)

Selects an item in the ROI Manager list, where *index* must be greater than or equal zero and less than the value returned by *roiManager("count")*. Use *roiManager("deselect")* to deselect all items on the list. For an example, refer to the ROI Manager Stack Demo macro. Requires v1.34k.

round(n)

Returns the closest integer to *n*. See also: floor.

run("command"[, "options"])

Executes an ImageJ menu command. The optional second argument contains values that are automatically entered into dialog boxes (must be GenericDialog or OpenDialog). Use the Command Recorder (*Plugins/Macros/Record*) to generate run() function calls. Use string concatentation to pass a variable as an argument. For examples, see the ArgumentPassingDemo macro.

runMacro(name)

Runs the specified macro file, which is assumed to be in the Image macros folder. A full file path may also be used. The ".txt" extension is optional. Returns any string argument returned by the macro. May have an optional second string argument that is passed to macro. For an example, see the CalculateMean macro. See also: eval and getArgument. Requires v1.34g.

S

save(path)

Saves an image, lookup table, selection or text window to the specified file path. The path must end in ".tif", ".jpg", ".gif", ".zip", ".raw", ".avi", ".bmp", ".lut", ".roi" or ".txt".

saveAs(format, path)

Saves the active image, lookup table, selection, measurement results, selection XY coordinates or text window to the specified file path. The *format* argument must be "tiff", "jpeg", "gif", "zip", "raw", "avi", "bmp", "text image", "lut", "selection", "measurements", "xy Coordinates" or "text". With v1.34k or later, use *saveAs(format)* to have a "Save As" dialog displayed.

saveSettings()

Saves most *Edit/Options* submenu settings so they can be restored later by calling *restoreSettings()*.

screenHeight

Returns the screen height in pixels. See also: getLocationAndSize, setLocation. Requires v1.34g.

screenWidth

Returns the screen width in pixels. Requires v1.34g.

selectionName

Returns the name of the current selection, or an empty string if the selection does not have a name. Aborts the macro if there is no selection. Requires 1.35i. See also: setSelectionName and selectionType.

selectionType()

Returns the selection type, where 0=rectangle, 1=oval, 2=polygon, 3=freehand, 4=traced, 5=straight line, 6=segmented line, 7=freehand line, 8=angle, 9=composite and 10=point. Returns -1 if there is no selection. For an example, see the ShowImageInfo macro.

selectImage(id)

Activates the image with the specified ID (a negative number). If *id* is greater than zero, activates the *id*th image listed in the Window menu. With ImageJ 1.33n and later, *id* can be an image title (a string).

selectWindow("name")

Activates the image window with the title "name". Also activates non-image windows in v1.30n or later.

setAutoThreshold()

Sets the threshold levels based on an analysis of the histogram of the current image or selection. Equivalent to clicking on "Auto" in the *Image/Adjust/Threshold* window. See also: setThreshold, getThreshold, resetThreshold.

setBackgroundColor(r, g, b)

Sets the background color, where *r*, *g*, and *b* are >= 0 and <= 255.

setBatchMode(arg)

If *arg* is *true*, the interpreter enters batch mode and images are not displayed, allowing the macro to run up to 20 times faster. If *arg* is *false*, exits batch mode and displays the active image in a window. With ImageJ 1.37j or later, set *arg* to *"exit and display"* to exit batch mode and display all open batch mode images. Here are five example batch mode macros: BatchModeTest, BatchMeasure, BatchSetScale and ExpandOrShrinkSelection, ReplaceRedWithMagenta.

setColor(r, g, b)

Sets the drawing color, where *r*, *g*, and *b* are >= 0 and <= 255. With 16 and 32 bit images, sets the color to 0 if r=g=b=0. With 16 and 32 bit images, use *setColor(1,0,0)* to make the drawing color the same is the minimum displayed pixel value. This function is faster than setForegroundColor() and does not change the system wide foreground color or repaint the color picker tool icon.

setColor(value)

Sets the drawing color. For 8 bit images, 0<=*value*<=255. For 16 bit images, 0<=*value*<=65535. For RGB images, use hex constants (e.g., 0xff0000 for red). This function does not change the foreground color used by run("Draw") and run("Fill").

setFont(name, size[, style])

The first argument is the font name. It should be "SansSerif", "Serif" or "Monospaced". The second argument is the point size of the font. The optional third argument is a string containing "bold" or "italic", or both. With ImageJ 1.37e or later, the third argument can also contain the keyword "antialiased". For examples, run the TextDemo macro.

setForegroundColor(r, g, b)

Sets the foreground color, where *r*, *g*, and *b* are >= 0 and <= 255.

setJustification("center")

Specifies the justification used by *drawString()* and *Plot.addText()*. The argument can be "left", "right" or "center". The default is "left".

setKeyDown(keys)

Simulates pressing the shift, alt or space keys, where *keys* is a string containing some combination of "shift", "alt" or "space". Any key not specified is set "up". Use *setKeyDown("none")* to set all keys in the "up" position. For an examples, see the CompositeSelections and DoWandDemo macros. See also: isKeyDown.

setLineWidth(width)

Specifies the line width (in pixels) used by drawLine(), lineTo(), drawRect() and drawOval().

setLocation(x, y)

Moves the active image window to a new location. See also: getLocationAndSize, screenWidth, screenHeight.

setLut(reds, greens, blues)

Creates a new lookup table and assigns it to the current image. Three input arrays are required, each containing 256 intensity values. See the LookupTables macros for examples.

setMetadata(string)

Assigns the metadata contained in the specified string to the the current image or stack slice. With stacks, the first 15 characters, or up to the first newline, are used as the slice label. The metadata is always assigned to the "Info" image property (displayed by *Image/Show Info*) if *string* starts with "Info:". Note that the metadata is saved as part of the TIFF header when the image is saved. See also: getMetadata. Requires v1.34q.

setMinAndMax(min, max)

Sets the minimum and maximum displayed pixel values (display range). See the DisplayRangeMacros for examples.

setPasteMode(mode)

Sets the transfer mode used by the *Edit/Paste* command, where 'mode' is "Copy", "Blend", "Average", "Difference", "Transparent", "AND", "OR", "XOR", "Add", "Subtract", "Multiply", or "Divide". In v1.37a or later, 'mode' can also be "Min" or "Max".

setPixel(x, y, value)

Stores *value* at location (*x*,*y*) of the current image. The screen is updated when the macro exits or call updateDisplay() to have it updated immediately.

setResult("Column", row, value)

Adds an entry to the ImageJ results table or modifies an existing entry. The first argument specifies a column in the table. If the specified column does not exist, it is added. The second argument specifies the row, where 0<=*row*<=nResults. (*nResults* is a predefined variable.) A row is added to the table if *row=nResults*. The third argument is the value to be added or modified. Call *setResult("Label", row, string)* to set the row label (v1.33n or later). Call *updateResults()* to display the updated table in the "Results" window. For examples, see the SineCosineTable and ConvexitySolidarity macros.

setRGBWeights(redWeight, greenWeight, blueWeight)

Sets the weighting factors used by the *Analyze/Measure*, *Image/Type/8-bit* and *Analyze/Histogram* commands when they convert RGB pixels values to grayscale. The sum of the weights must be 1.0. Use *(1/3,1/3,1/3)* for equal weighting of red, green and blue. The weighting factors in effect when the macro started are restored when it terminates. For examples, see the MeasureRGB, ShowRGBChannels and RGB_Histogram macros. Requires v1.35b.

setThreshold(lower, upper)

Sets the lower and upper threshold levels. Starting with ImageJ 1.34g, there is an optional third argument that can be "red", "black & white", "over/under" or "no color". See also: setAutoThreshold, getThreshold, resetThreshold.

setTool(id)

Switches to the specified tool, where 0=rectangle, 1=oval, 2=polygon, 3=freehand, 4=straight line, 5=polyline, 6=freeline, 7=point, 8=wand, 9=text, 10=spare, 11=zoom, 12=hand, 13=dropper, 14=angle, 15...19=spare. See also: toolID.

setupUndo()

Call this function before drawing on an image to allow the user the option of later restoring

the original image using *Edit/Undo*. Note that setupUndo() may not work as intended with macros that call the run() function. For an example, see the DrawingTools tool set.

setSelectionName(name)

Sets the name of the current selection to the specified name. Aborts the macro if there is no selection. Requires 1.35i. See also: selectionName and selectionType.

setSlice(n)

Displays the *n*th slice of the active stack. Does nothing if the active image is not a stack. For an example, see the MeasureStack macros. See also: getSliceNumber, nSlices.

setVoxelSize(width, height, depth, unit)

Defines the voxel dimensions and unit of length ("pixel", "mm", etc.) for the current image or stack. The *depth* argument is ignored if the current image is not a stack. See also: getVoxelSize. Requires v1.34k.

setZCoordinate(z)

Sets the Z coordinate used by *getPixel()*, *setPixel()* and *changeValues()*. The argument must be in the range 0 to n-1, where n is the number of images in the stack. For an examples, see the Z Profile Plotter Tool.

showMessage("message")

Displays "message" in a dialog box.

showMessage("title", "message")

Displays "message" in a dialog box using "title" as the the dialog box title.

showMessageWithCancel(["title",]"message")

Displays "message" in a dialog box with "OK" and "Cancel" buttons. "Title" (optional) is the dialog box title. The macro exits if the user clicks "Cancel" button.

showProgress(progress)

Updates the ImageJ progress bar, where 0.0<=*progress*<=1.0. The progress bar is not displayed if the time between the first and second calls to this function is less than 30 milliseconds. It is erased when the macro terminates or *progress* is >=1.0.

showStatus("message")

Displays a message in the ImageJ status bar.

sin(angle)

Returns the sine of an angle (in radians).

snapshot()

Creates a backup copy of the current image that can be later restored using the reset function. For examples, see the ImageRotator and BouncingBar macros.

split(string, delimiters)

Breaks a string into an array of substrings. *Delimiters* is a string containing one or more delimiter characters. The default delimiter set " \t\n\r" (space, tab, newline, return) is used if *delimiters* is an empty string or split is called with only one argument (v1.34j or later). Returns a one element array if no delimiter is found.

sqrt(n)

Returns the square root of *n*. Returns NaN if *n* is less than zero.

startsWith(string, prefix)

Returns *true* (1) if *string* starts with *prefix*. See also: endsWith, indexOf, substring, toLowerCase.

substring(string, index1, index2)

Returns a new string that is a substring of *string*. The substring begins at *index1* and extends to the character at *index2* - 1. See also: indexOf, startsWith, endsWith, replace.

T

tan(angle)

Returns the tangent of an angle (in radians).

toBinary(number)

Returns a binary string representation of *number*.

toHex(number)

Returns a hexadecimal string representation of *number*.

toLowerCase(string)

Returns a new string that is a copy of *string* with all the characters converted to lower case. Requires v1.34b.

toolID

Returns the ID of the currently selected tool. Requires 1.35i. See also: setTool.

toString(number)

Returns a decimal string representation of *number*. See also: toBinary, toHex, parseFloat and parseInt. Requires v1.34g.

toUpperCase(string)

Returns a new string that is a copy of *string* with all the characters converted to upper case. Requires v1.34b.

U

updateDisplay()

Redraws the active image.

updateResults()

Call this function to update the "Results" window after the results table has been modified by calls to the setResult() function.

W

wait(n)

Delays (sleeps) for *n* milliseconds.

write(string)

Outputs a string to the "Results" window. Numeric arguments are automatically converted to strings.

Annexe 6 : Macro « Calcul de la porosité »

```
open("Filt3_Sc025_teyssier-780-6mrc.mrc.img");
title = getTitle(); //print(title);
setSlice(nSlices/2);
setBackgroundColor(113,113,113);
run("Canvas Size...", "width=500 height=500 position=Center");
newImage("SEUILLAGE","8-bit",getWidth(),getHeight(),nSlices);
newImage("PIXEL","RGB
Color",getWidth(),getHeight(),nSlices);setBackgroundColor(185,181,0);
//setBatchMode(true);
for(i=68;i<=145;i++) {
                selectWindow(title);
                setSlice(i);run("Copy");
                newImage("Etape","8-bit",getWidth(),getHeight(),1);
                run("Paste");
                makeOval(93, 87, 346, 291);

run("SnakeD ", "gradient=35 regularization=0.6 number=200 step=5 first=1 last=1 draw=Red
alpha=0.7 create");
close();
selectWindow("Etape");run("Clear Outside");
setThreshold(0, 112);
run("Threshold", "thresholded remaining black");
run("Options...", "iterations=1 count=1");
run("Dilate");
run("Erode");
run("Copy");//close();
selectWindow("SEUILLAGE");setSlice(i);run("Paste");
selectWindow("Etape_snake_deriche"); close();
selectWindow("Etape");makeOval(93, 87, 346, 291);
```

```
run("SnakeD ", "gradient=35 regularization=0.6 number=200 step=5 first=1 last=1 draw=Red
alpha=0.7 create");
close();
selectWindow("Etape");run("Copy");
newImage("INTERMEDIAIRE","RGB Color",getWidth(),getHeight(),1);
run("Paste");
run("Clear Outside");
run("Select None");
run("Copy");
selectWindow("PIXEL");setSlice(i);
run("Paste");wait(300);
selectWindow("Etape_snake_deriche"); close();
selectWindow("INTERMEDIAIRE");close();
selectWindow("Etape");close();
                }
// DETERMINATION DU VOLUME DE L'OBJET
// SELECTION DE LA FENETRE SEUILLAGE
selectWindow("SEUILLAGE")
// SEUILLAGE
run("Options...", "iterations=1 count=1");
run("Close-", "stack");
run("Fill Holes", "stack");
run("Options...", "iterations=3 count=1");
run("Close-", "stack");
run("Dilate", "stack");
run("Fill Holes", "stack");
run("Options...", "iterations=3 count=1");
run("Close-", "stack");
run("Dilate", "stack");
run("Fill Holes", "stack");
run("Options...", "iterations=6 count=1");
run("Erode", "stack");
// ANALYSE DE PARTICULES
run("Set Measurements...", "area center perimeter redirect=None decimal=3");
```

```
run("Analyze Particles...", "size=0-Infinity circularity=0.00-1.00 show=Nothing display clear
record stack");
// CALCUL DU VOLUME DE L'OBJET
var Volume_Objet;
Volume_Objet = 0;
for (i = 1; i < nResults; i++)
{
VO = getResult("Area", i);
Volume_Objet = Volume_Objet + VO;
}
print("Volume_Objet = "+Volume_Objet);
// FERMETURE DE LA FENETRE RESULT
 if (isOpen("Results")) {
     selectWindow("Results");
    run("Close");
     }
// DETERMINATION DE LA POROSITE
// SELECTION DE LA FENETRE SEUILLAGE
selectWindow("PIXEL");
// CONVERSION EN 8 BIT
run("8-bit");
// APPLICATION DU PLUGIN 3D OBJECTS COUNTER
setThreshold(128, 255);
run(" 3D objects counter", "threshold=128 slice=94 min=10 max=100000000 particles dot=3
font=12 log");
run("Duplicate...", "title=PorositÃ©1");
// SAUVEGARDE DES RESULTATS
if (isOpen("Results from PIXEL"))
{
     selectWindow("Results from PIXEL");
     saveAs("Text", "/home/tihay/TOMO/Ovidiu/Rodrigue/Results from PIXEL.txt");
     run("Text...", "save=[/home/tihay/TOMO/Ovidiu/Rodrigue/Results from PIXEL.txt]");
     run("Close");
}
// OUVERTURE DU FICHIER RESULT
```

```
// AVEC LA MACRO DE MUTTERER
 lineseparator = "\n";
 cellseparator = ",\t";
// copies the whole RT to an array of lines
 lines=split(File.openAsString(""), lineseparator);
// recreates the columns headers
labels=split(lines[0], cellseparator);
    if (labels[0]==" ")
      k=1; // it is an ImageJ Results table, skip first column
    else
      k=0; // it is not a Results table, load all columns
    for (j=k; j<labels.length; j++)
      setResult(labels[j],0,0);
    // dispatches the data into the new RT
    run("Clear Results");
    for (i=1; i<lines.length; i++) {
      items=split(lines[i], cellseparator);
      for (j=k; j<items.length; j++)
       setResult(labels[j],i-1,items[j]);
 //updateResults();
 // FIN MACRO MUTTERER
//run("Close", "openasstring...=[/home/tihay/TOMO/Ovidiu/Rodrigue/Results from
PIXEL.txt]");
// CALCUL DU VOLUME DES PORES
var Volume_Pores;
for (i = 1; i < nResults; i++)
{
VP = getResult("Volume", i);
Volume_Pores = Volume_Pores + VP;
}
print("Volume_Pores = "+Volume_Pores);
// CALCUL DE LA POROSITE
var Porosite;
var Volume_Pores, Volume_Objet;
```

```
Volume_Pores = 110013;
Volume_Objet = 3779688;
Porosite = (Volume_Pores/Volume_Objet)*100;
print("Porosite = "+Porosite+" %");
```

Annexe 7 : Macro Détection de contours par seuillage et binarisation

```
// OUVERTURE DU FICHIER
open("Cylindre_Crop_Retouche_Rotate_Filt1_MTP_A2_50kX_a_corrige_D.tif »);
// DUPPLICATION ET POSITIONNEMENT AU MILIEU DE LA SERIE
run("Duplicate...",title=Cylindre_Crop_Retouche_Rotate_Filt1_MTP_A2_50kX_a_corrige_D
 -1.img duplicate"); setSlice(nSlices/2);
// RETOUR A L'IMAGE DE DEPART ET POSITIONNEMENT A L'INTERIEUR DE LA SERIE
selectWindow("Cylindre_Crop_Retouche_Rotate_Filt1_MTP_A2_50kX_a_corrige_D.tif");
setSlice(nSlices/2);
// SEUILLAGE DE LA SERIE DES IMAGES
setThreshold(79, 255);
run("Threshold", "thresholded remaining black stack");
run("Options...", "iterations=2 black count=1");
run("Fill Holes", "stack");
run("Open", "stack");
run("Close-", "stack");
run("Fill Holes", "stack");
run("Options...", "iterations=7 black count=1");
run("Erode","stack");
run("Dilate","stack");
run("Smooth", "stack");
setThreshold(127, 255);
run("Threshold", "thresholded remaining black stack");


//VERIFICATION DE LA SEGMENTATION
run("Duplicate...",
"title=Cylindre_Crop_Retouche_Rotate_Filt1_MTP_A2_50kX_a_corrige_D-2.img
duplicate");
run("Find Edges", "stack");setSlice(nSlices/2);
imageCalculator("Max create stack", "Cylindre_Crop_Retouche_Rotate_Filt1_MTP_A2_
50kX_a_corrige_D-1.img","Cylindre_Crop_Retouche_Rotate_Filt1_MTP_A2_50kX_a_
```

```
corrige_D-2.img");
selectWindow("Cylindre_Crop_Retouche_Rotate_Filt1_MTP_A2_50kX_a_corrige_D-
1.img"); close();
selectWindow("Cylindre_Crop_Retouche_Rotate_Filt1_MTP_A2_50kX_a_corrige_D-
2.img"); close();
//CREATION D'UNE NOUVELLE SERIE D'IMAGES
selectWindow("Cylindre_Crop_Retouche_Rotate_Filt1_MTP_A2_50kX_a_corrige_D.tif");
newImage("CONTOUR CYLINDRE","8-bit",getWidth(),getHeight(),nSlices);
// SELECTION DU CONTOUR PRINCIPAL
for(i=1;i<=nSlices;i++) {
selectWindow("Cylindre_Crop_Retouche_Rotate_Filt1_MTP_A2_50kX_a_corrige_D.tif");
setSlice(i);run("Copy");
// CREATION D'UNE IMAGE DE LA SLICE COPIEE
newImage("CYLINDRE_Etape"+i+"","8-bit",getWidth(),getHeight(),1);run("Paste");
run("Options...", "iterations=1 count=1");
run("Dilate");
run("Watershed");
run("Erode");
// FIXATION DES PARAMETRES  ET ANALYSE DE PARTICULES
run("Set Measurements...", "area min centroid center perimeter redirect=None decimal=3");
run("Analyze Particles...", "size=0-Infinity circularity=0.00-1.00 show=Nothing display");
// DETERMINATION DE LA PLUS GRANDE SURFACE
A = 0;
for(j=0; j<=nResults-1; j++) {
                                B= getResult("Area",j);
                                if ( A < B) A=B;
                                 }
// ELIMINATION DES CONTOURS SECONDAIRES
for(j=0; j<=nResults-1; j++) {
                                 selectWindow("CYLINDRE_Etape"+i+"");
                                 B= getResult("Area",j);
                                 if ( B < A) {doWand(getResult("X", j),getResult("Y", j));
                                 setForegroundColor(255, 255, 255);fill();run("Select None");}
                                 }
```

```
// SELECTION DU CONTOUR PRINCIPAL
selectWindow("CYLINDRE_Etape"+i+"");run("Copy");
// COLLAGE DU CONTOUR PRINCIPAL DANS LA NOUVELLE IMAGE
selectWindow("CONTOUR CYLINDRE");
setSlice(i);run("Paste");
// NETTOYAGE DE LA FENETRE RESULT
run("Clear Results");
// FERMETURE DE LA FENETRE QUELCONQUE ETAPE
selectWindow("CYLINDRE_Etape"+i+"");close();
                                        }
//FIXATION DES PARAMETRES DE MESURE + ANALYSE DE PARTICULES
run("Set Measurements...", "area centroid perimeter slice redirect=None decimal=3");
run("Analyze Particles...", "size=0-Infinity circularity=0.00-1.00 show=Nothing display
stack");
n=nResults;
// CALCUL DE LA SURFACE TOTALE
var Surface_Total;
a = 0;
for (i=0; i< n; i++) {
                        a1 = getResult("Perim.", i);
                        a = a + a1;
                         }
setResult("Surface_Total", nResults, a);
updateResults();
```

Annexe 8 : Exemple de traitement des données avec une macro

```
// OUVERTURE DU FICHIER
open("Cylindre_Crop_Retouche_Rotate_Filt1_MTP_A2_50kX_a_corrige_D.tif");
// DUPPLICATION
run("Duplicate...",
"title=Cylindre_Crop_Retouche_Rotate_Filt1_MTP_A2_50kX_a_corrige_D-1.img
duplicate");
setSlice(nSlices/2);
// POSITIONNEMENT A L'INTERIEUR DE LA SERIE
selectWindow("Cylindre_Crop_Retouche_Rotate_Filt1_MTP_A2_50kX_a_corrige_D.tif");se
tSlice(nSlices/2);
// SEUILLAGE DE LA SERIE DES IMAGES
setThreshold(79, 255);
run("Threshold", "thresholded remaining black stack");
run("Options...", "iterations=2 black count=1");
run("Fill Holes", "stack");
run("Open", "stack");
run("Close-", "stack");
run("Fill Holes", "stack");
run("Options...", "iterations=7 black count=1");
run("Erode","stack");
run("Dilate","stack");
run("Smooth", "stack");
setThreshold(127, 255);
run("Threshold", "thresholded remaining black stack");
// VERIFICATION DE LA SEGMENTATION
run("Duplicate...",
"title=Cylindre_Crop_Retouche_Rotate_Filt1_MTP_A2_50kX_a_corrige_D-2.img
duplicate");
run("Find Edges", "stack");setSlice(nSlices/2);
imageCalculator("Max create stack", "Cylindre_Crop_Retouche_Rotate_Filt1_MTP_A2_
```

```
50kX_a_corrige_D-1.img","Cylindre_Crop_Retouche_Rotate_Filt1_MTP_A2_50kX_a_
corrige_D-2.img");
selectWindow("Cylindre_Crop_Retouche_Rotate_Filt1_MTP_A2_50kX_a_corrige_D-
1.img"); close();
selectWindow("Cylindre_Crop_Retouche_Rotate_Filt1_MTP_A2_50kX_a_corrige_D-
2.img"); close();
//CREATION D'UNE NOUVELLE SERIE D'IMAGES
selectWindow("Cylindre_Crop_Retouche_Rotate_Filt1_MTP_A2_50kX_a_corrige_D.tif");
newImage("CONTOUR CYLINDRE","8-bit",getWidth(),getHeight(),nSlices);
// SELECTION DU CONTOUR PRINCIPAL
for(i=1;i<=nSlices;i++) {
selectWindow("Cylindre_Crop_Retouche_Rotate_Filt1_MTP_A2_50kX_a_corrige_D.tif");
setSlice(i);run("Copy");
// CREATION D'UNE IMAGE DE LA SLICE COPIEE
newImage("CYLINDRE_Etape"+i+"","8-bit",getWidth(),getHeight(),1);run("Paste");
run("Options...", "iterations=1 count=1");
run("Dilate");
run("Watershed");
run("Erode");
// SEUILLAGE ET ANALYSE DE PARTICULES
//setThreshold(0, 128);
//run("Threshold", "thresholded remaining black");
run("Set Measurements...", "area min centroid center perimeter redirect=None decimal=3");
run("Analyze Particles...", "size=0-Infinity circularity=0.00-1.00 show=Nothing display");
// DETERMINATION DE LA PLUS GRANDE SURFACE
A = 0;
for(j=0; j<=nResults-1; j++) {
                                B= getResult("Area",j);
                                if ( A < B) A=B;
                                 }
// ELIMINATION DES CONTOURS SECONDAIRES
for(j=0; j<=nResults-1; j++) {
                                 selectWindow("CYLINDRE_Etape"+i+"");
                                 B= getResult("Area",j);
```

```
                                    if ( B < A) {doWand(getResult("X", j),getResult("Y", j));
                                    setForegroundColor(255, 255, 255);fill();run("Select None");}
                                     }
// SELECTION DU CONTOUR PRINCIPAL
selectWindow("CYLINDRE_Etape"+i+"");run("Copy");
// COLLAGE DU CONTOUR PRINCIPAL DANS LA NOUVELLE IMAGE
selectWindow("CONTOUR CYLINDRE");
setSlice(i);run("Paste");
// NETTOYAGE DE LA FENETRE RESULT
run("Clear Results");
// FERMETURE DE LA FENETRE QUELCONQUE ETAPE
selectWindow("CYLINDRE_Etape"+i+"");close();
                                        }
// FIXATION DES PARAMETRES DE MESURE + ANALYSE DE PARTICULES
run("Set Measurements...", "area centroid perimeter slice redirect=None decimal=3");
run("Analyze Particles...", "size=0-Infinity circularity=0.00-1.00 show=Nothing display
stack");
n=nResults;
// CALCUL DE LA SURFACE TOTALE
var Surface_Total;
a = 0;
for (i=0; i< n; i++) {
                        a1 = getResult("Perim.", i);
                         a = a + a1;
                           }
setResult("Surface_Total", nResults, a);
// CALCUL DU CONTOUR MOYEN DE LA PARTIE CENTRALE
N = nSlices;
b = 0;T = 0;
for (i=N/2 -75; i<=N/2 +75; i++) {
                                        b1= getResult("Perim.", i);
                                         b = b + b1;
                                         T = T + 1;
                                       }
```

```
M = b/T;
//print("M = "+M);
setResult("ContourLatMoy", nResults, M);
// CAL CUL DE LA SURFACE LATTERALE
var ContLat, SurfaceLat;
SurfaceLat = 0;
for (i=0; i < nResults; i++) {
                                h = getResult("Perim.", i);
                                if ( h >= 0.9*M && h <= 1.10*M)   {
                                                                surface = getResult("Perim.", i);
                                                                SurfaceLat = SurfaceLat + surface;
                                                                        }
                                }
print("Surface Laterale = "+SurfaceLat);
setResult("SurfaceLat", nResults, SurfaceLat);
updateResults();
var SurfaceExtrem, PartieLat, PartieExtrem;
SurfaceExtrem = a - SurfaceLat; print("SurfaceExtrem = "+SurfaceExtrem);
PartieLat = (SurfaceLat/a)*100; print("Partie Laterale = "+PartieLat+" %");
PartieExtrem = (SurfaceExtrem/a)*100; print("Parties Extremes = "+PartieExtrem+" %");
// COLORATION DE LA SURFACE LATTERALE
selectWindow("CONTOUR CYLINDRE");
newImage("Coloration","RGB Color",getWidth(),getHeight(),nSlices);
for (i = 7; i <= 354 ; i++) {
                                selectWindow("CONTOUR CYLINDRE");
                                setSlice(i);run("Copy");
                                selectWindow("Coloration");setSlice(i);run("Paste");
                                h = getResult("Perim.", i);
                               if ( h >= 0.9*M && h <= 1.10*M)  changeValues(0,175,200);
                               }
selectWindow("Coloration");
run("8-bit");

// SEPARATION
```

```
selectWindow("CONTOUR CYLINDRE");
newImage("MILIEU_CYLINDRE","8-bit",getWidth(),getHeight(),nSlices);
newImage("EXTREMITES_CYLINDRE","8-bit",getWidth(),getHeight(),nSlices);
for (i = 7; i<=354 ; i++) {
                                selectWindow("CONTOUR CYLINDRE");
                                setSlice(i);run("Copy");
                                h = getResult("Perim.", i);
            if ( h >= 0.9*M && h <= 1.20*M)
              {selectWindow("MILIEU_CYLINDRE");setSlice(i);run("Paste");}
           else {selectWindow("EXTREMITES_CYLINDRE");setSlice(i);run("Paste");}
                                }
```

Annexe 9 : Macro Rotation d'un objet cylindrique pour le mettre dans l'axe

```
// OUVERTURE DU FICHIER
open();
// POSITIONNEMENT A L'INTERIEUR DE LA SERIE
setSlice(nSlices/2);
// SELECTION D'UN TYPE D'OBJET SUR LA PILE D'IMAGES
setTool(10);
// DEFINITION DES VARIABLES
var dx, dy, dz;
macro "annot Tool- C000 T2708A T87088" {
// CREATION D'UNE BOITE DE DIALOGUE
Dialog.create("Choix de l'axe de Rotation");
Dialog.addMessage("Choisir l'axe de rotation en donnant la valeur de l'angle");
Dialog.addNumber("Rotation suivant X", dx);
Dialog.addNumber("Rotation suivant Y", dy);
Dialog.addNumber("Rotation suivant Z", dz);
Dialog.addMessage("  SENS DE ROTATION \n \n Angle négatif: sens trigonométrique \n Angle positif : sens inverse");
Dialog.show();
// RECUPERATION DES VALEURS SAISIES DANS LA BOITE DE DIALOGUE
dx = Dialog.getNumber();
dy = Dialog.getNumber();
dz = Dialog.getNumber();
print("dx = "+dx, "dy = "+dy, "dz = "+dz);
// RECUPERATION DU NOM DE L'IMAGE DE DEPART
ImageDepart = getTitle(); // print("ImageDepart = "+ImageDepart);
// APPLICATION DE LA FONCTION ROTATION
FonctionRotation();
// RECUPERATION DU NOM DE LA NOUVELLE IMAGE
ImageRotated = getTitle(); // print("ImageRotated = "+ImageRotated);
```

```
// APPLICATION D'UNE FONCTION QUI PERMET DE FERMER ImageDepart
FonctionCloseImageDepart();
// SELECTION DE L'IMAGE AYANT SUBIT LA ROTATION
selectWindow(ImageRotated);
// RECUPERATION DU NOM DE CETTE IMAGE
ImageDepart = getTitle();
// POSITIONNEMENT A L'INTERIEUR DE LA SERIE
setSlice(nSlices/2);
// APPLICATION D'UNE FONCTION QUI PERMET RENOMMER ImageRotated
//FonctionRenameImageRotated();
}
// DEFINITION DE LA FONCTION FonctionRotation()
function FonctionRotation() {run("TJ Rotate", "z-angle="+dz+" y-angle="+dy+"
                                x-angle="+dx+" interpolation=linear background=0.0 adjust");}
// DEFINITION DE LA FONCTION FonctionCloseImageDepart()
function FonctionCloseImageDepart() {selectWindow(ImageDepart);close();};
// DEFINITION DE LA FONCTION FonctionRenameImageRotated()
//function FonctionRenameImageRotated() {selectWindow(ImageRotated);
                                        rename("ImageDepart");setSlice(nSlices/2);};
```