

TI – Traitement d'Images

Introduction à ImageJ

Olivier Losson

Master ASE : <http://master-ase.univ-lille1.fr>
Master Informatique : <http://www.fil.univ-lille1.fr>
Spécialité IVI : <http://master-ivi.univ-lille1.fr>

Sommaire

- **1 – Présentation d'ImageJ**
 - ImageJ en quelques mots
 - Types d'images
 - Représentation des images
- **2 – Développement de *Macros***
 - Présentation des macros
 - Fonctions disponibles
 - Limites et préconisation d'emploi
- **3 – Développement de *Plugins***
 - Présentation des plugins
 - Classes fondamentales de l'API
 - Types de plugins et entrées d'un `PlugInFilter`
 - Accès aux pixels
 - Exemples
- **Sélection de références**

ImageJ en quelques mots

• Généralités

- Développé par le *National Institutes of Health* : <http://rsb.info.nih.gov/ij/>
- Logiciel libre, écrit en Java, dédié au traitement d'images
 - conseillé : JRE \geq 1.6
 - multi-plateformes, multi-threaded, applet ou application autonome
 - sources disponibles et architecture ouverte (extensibilité par *plugins* en Java)
 - communauté très active, surtout en imagerie biomédicale

• Fonctionnalités en analyse d'images

- Lit/écrit de nombreux formats d'images et vidéos
 - types 8, 16, et 32 bits pour la plupart des formats d'images disponibles
 - manipulation des piles d'images (« images 3D »)
- Possibilités étendues d'analyse des images en standard
 - outils d'analyse : histogrammes, profils, ...
 - opérations ponctuelles et de voisinage, morphologiques, FFT, segmentation, ...
- Développement aisé de scripts (*macros*), *plugins* et interfaces graphiques

Types d'images (1/3)

• Profondeur (*Image/Type*)

■ Images en niveaux de gris

- entiers positifs sur 8 bits ($0 \leq I(x,y) \leq 255$)

- entiers positifs sur 16 bits ($0 \leq I(x,y) \leq 65535$)

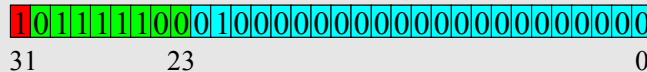
- flottants (réels) **signés** sur 32 bits (IEEE-754): $I(x,y) = (-1)^s \times 1.f \times 2^{(e-127)}$, où :

s : signe sur 1 bit ($0 \Leftrightarrow$ nombre positif, $1 \Leftrightarrow$ nombre négatif)

f : fraction (ou *mantisse*) sur 23 bits (8 388 608 valeurs possibles)

e : exposant sur 8 bits (256 valeurs possibles)

Soit 2^{23} valeurs entre 1 et 2, et presque une infinité entre 0 et 1

Ex.: s e f

 $I(x,y) = (-1)^1 \times 1.(2^{-2}) \times 2^{(124-127)} = -1.25 \times 2^{-3} = 0.15625$

■ Images couleur

- 8 bits avec couleurs indexées (table de couleurs)

- 32 bits (8 bits pour chaque canal $k \in \{\alpha, R, G, B\}$, soit $0 \leq I^k(x,y) \leq 255$)

■ Piles d'images

- Plusieurs images (mêmes type et dimensions) superposées dans la même fenêtre

Types d'images (2/3)

- Utilité des images 32 bits (float)

→ Seule possibilité pour représenter des valeurs négatives

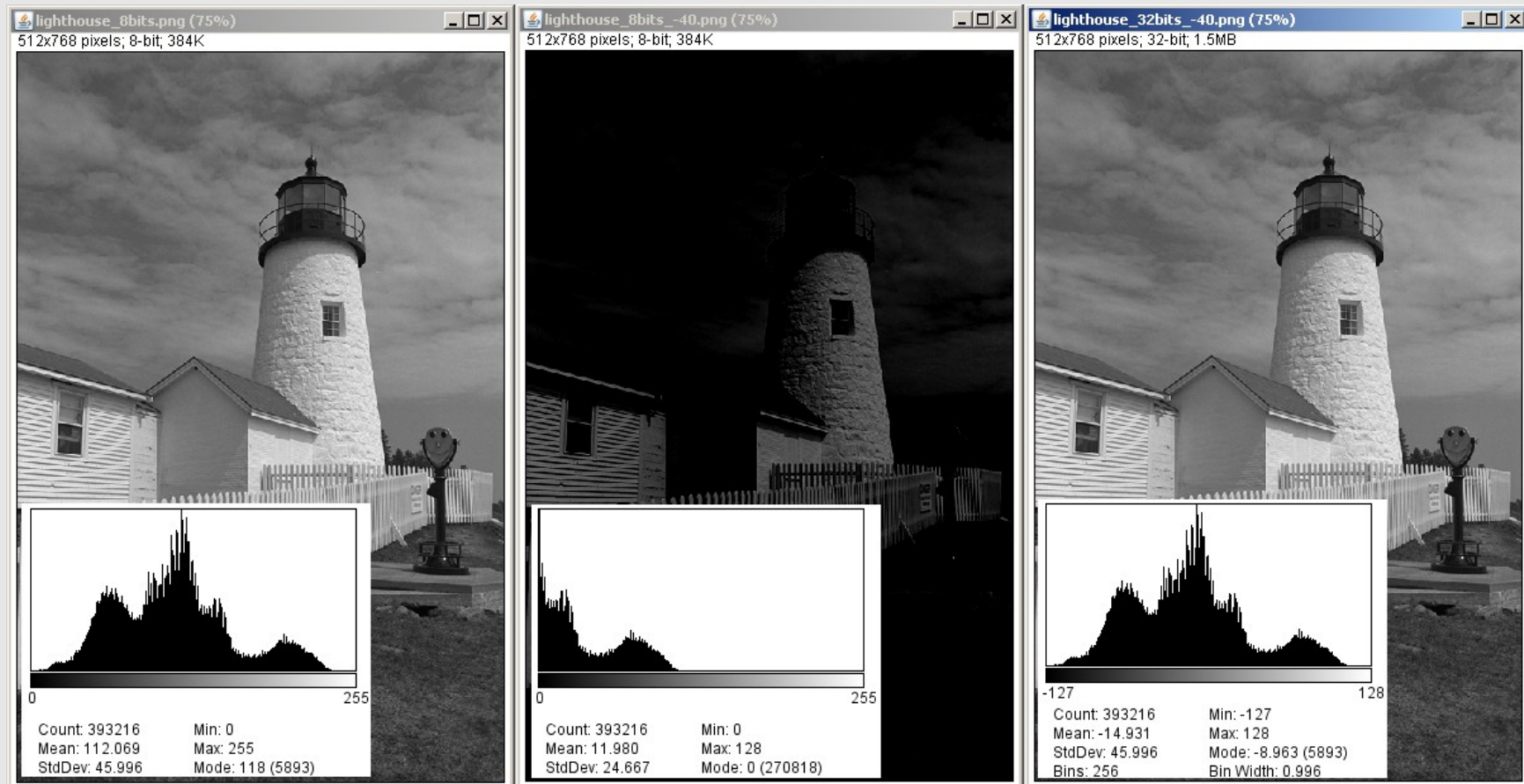


Image I sur 8 bits
 $(0 \leq I(x,y) \leq 255)$

$I_2(x,y) \equiv I(x,y) - 127$
 $I(x,y) \leq 127 \Rightarrow I_2(x,y) = 0$

$I_3(x,y) \equiv (\text{float})I(x,y) - 127$
 les valeurs ≤ 0 sont représentées

Types d'images (3/3)

• Utilité des images 32 bits (float) (suite)

→ Seule possibilité pour représenter des valeurs non entières

■ Si $I_2(x,y) \equiv I(x,y)/2$ et $I_3(x,y) \equiv (\text{float})I(x,y)/2$,

■ $I(x,y)=27 \Rightarrow I_2(x,y)=13$ mais $I_3(x,y)=13.5$

• Quel type d'image utiliser pour éviter une perte de précision ?

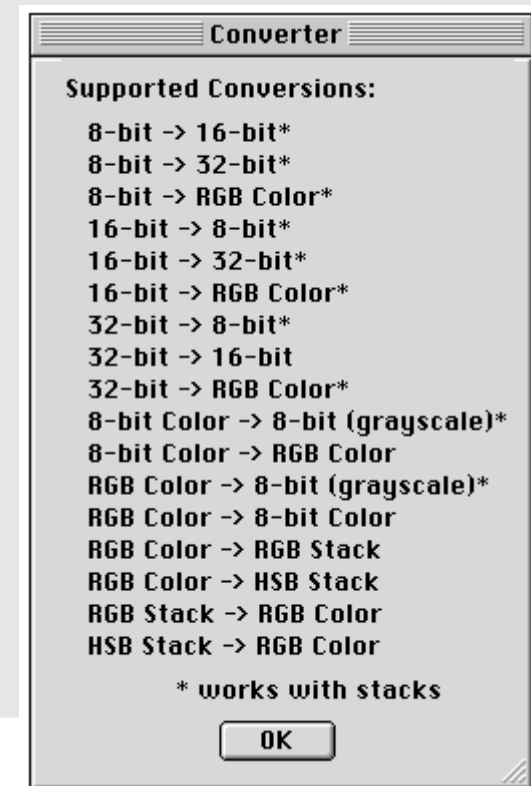
→ Maintenir le « *type* » (= la profondeur) si possible :
si entrée sur 16 bits, analyser les données sur 16 bits.

→ Si l'analyse requiert des valeurs signées
ou non entières, utiliser des images sur 32 bits.

→ Éviter autant que possible les
conversions répétées d'un type à l'autre.

→ Pour le stockage, ne pas utiliser de formats mettant
en œuvre une compression avec pertes (tels que JPG).

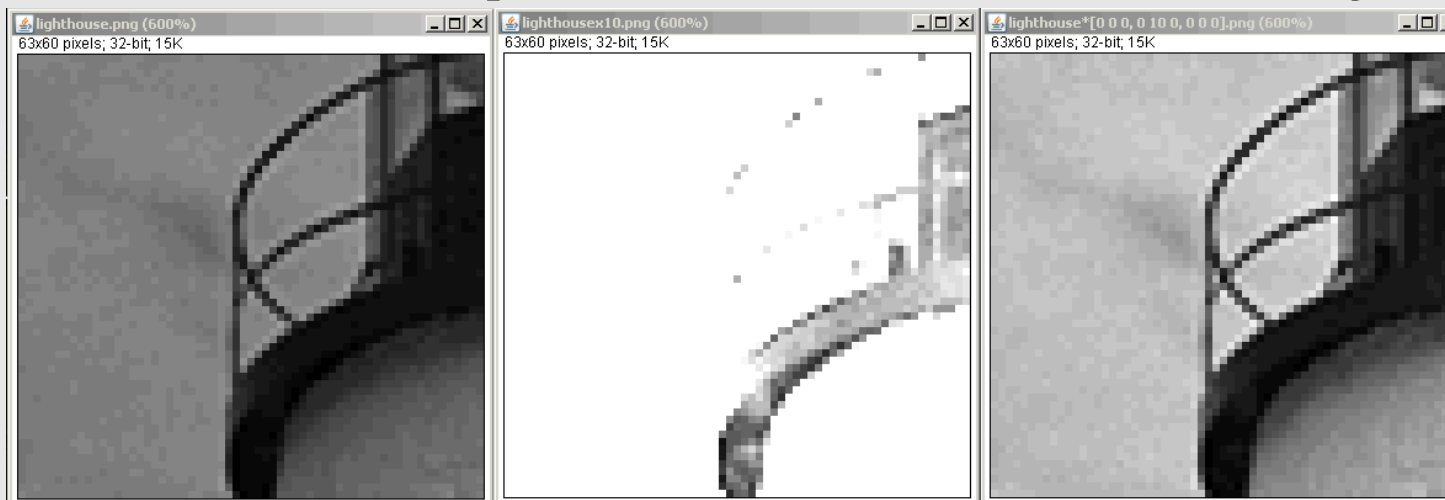
• Conversions autorisées dans ImageJ :



Représentation des images (1/5)

Affichage des images 16 et 32 bits

- Pour être affichées, ces données sont converties en 8 bits par fenêtrage.
 - La *fenêtre* définit la plage des niveaux de gris affichés (0 ou 255 hors plage).
- En pratique, pour afficher une image 16 ou 32 bits
 - les niveaux min et max sont calculés ; ils correspondent resp. à 0 et 255.
 - les niveaux de l'image sont remis à l'échelle sur 0..255.
 - *Attention* : certaines opérations saturent les niveaux à l'affichage, d'autres pas :



$$I \text{ sur 8 bits} \Rightarrow 32 \text{ bits} \quad I_2(x,y) \equiv I(x,y) \times 10$$

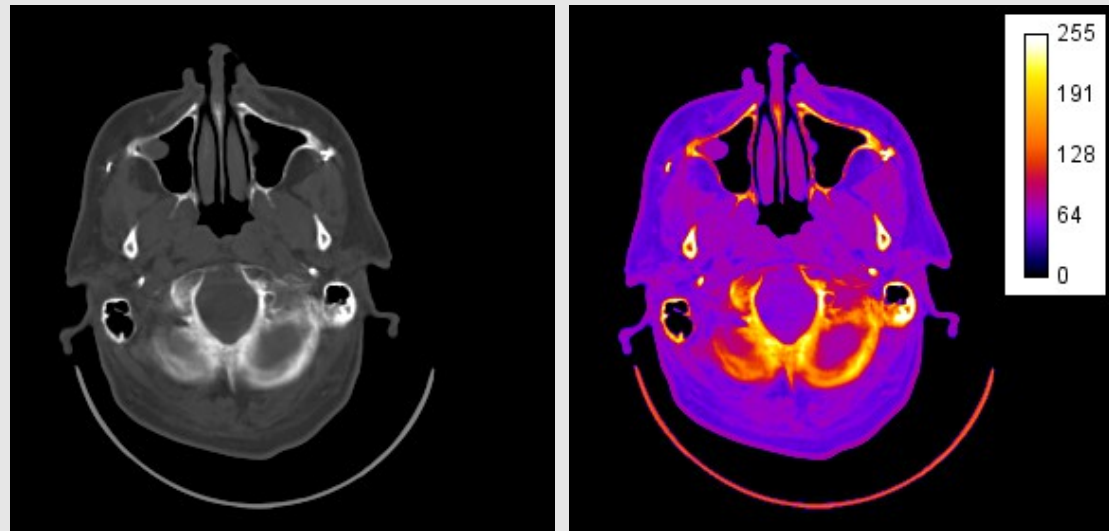
$$(0.0 \leq I(x,y) \leq 255.0) \quad I(x,y) \geq 26 \Rightarrow I_2(x,y) = 255 \quad I_3 = I * \begin{bmatrix} 0 & 0 & 0 \\ 0 & 10 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

cf. ImageProcessor, méthode `resetMinAndMax()`

Représentation des images (2/5)

• Tables de couleur (LUT)

- ➔ Permet de représenter une image mono-composante en **pseudo-couleurs**.
 - Une différence de couleur dans une telle image représente une différence d'intensité, et non une différence de couleur, entre objets de la scène.
 - But : mettre en évidence des caractéristiques de l'image, car l'œil humain ne peut distinguer que 60 nuances de gris environ, mais plusieurs millions de couleurs.
 - Principe : à chaque niveau de gris est associée une couleur (**R**, **G**, **B**) dans une *table de couleurs* ou *palette* (ang. *Look-Up Table*, *LUT*).
 - Les **données** de l'image ne sont **pas modifiées** ; seule la représentation change.

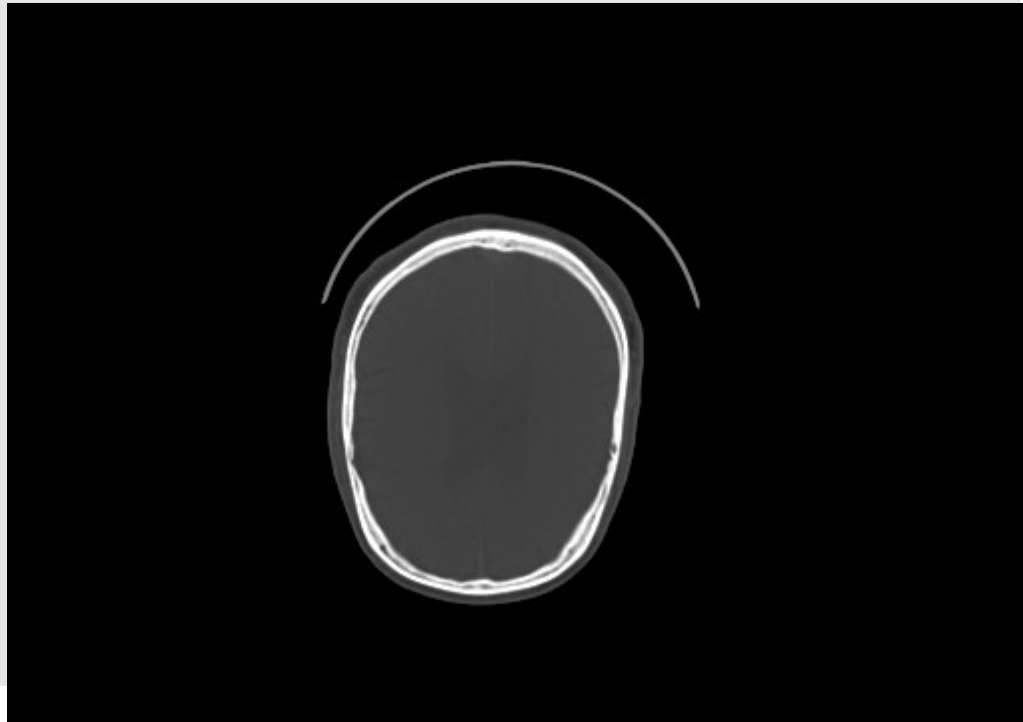
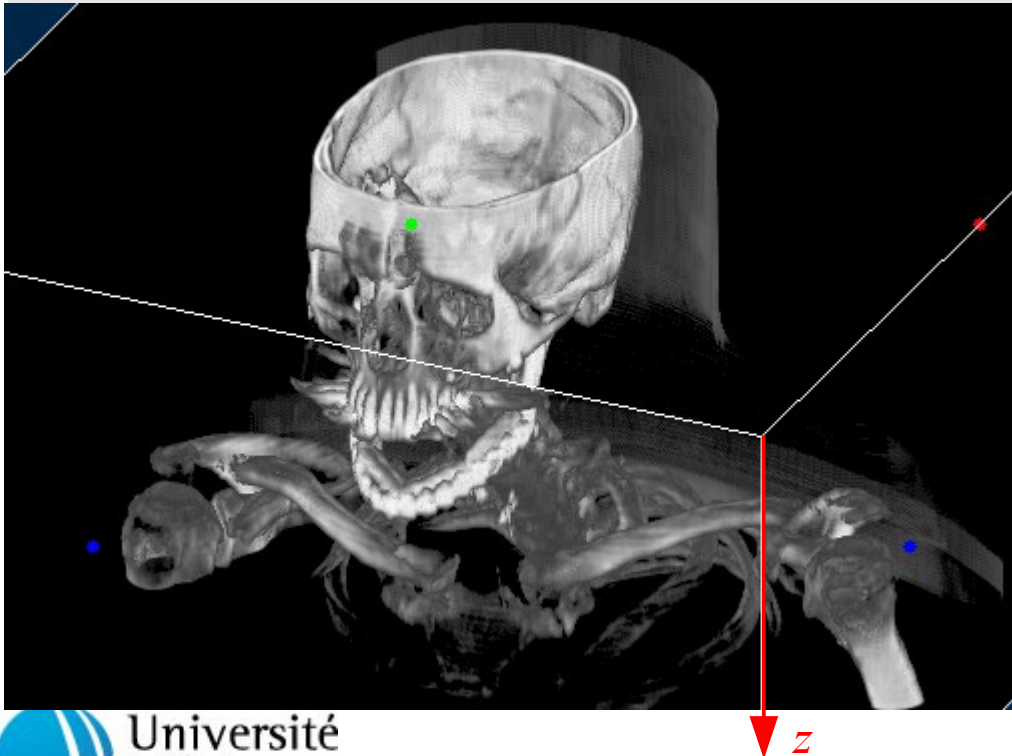


Plus d'info : http://www.macbiophotonics.ca/imagej/colour_image_processi.htm

Représentation des images (3/5)

• Piles d'images (*stack*)

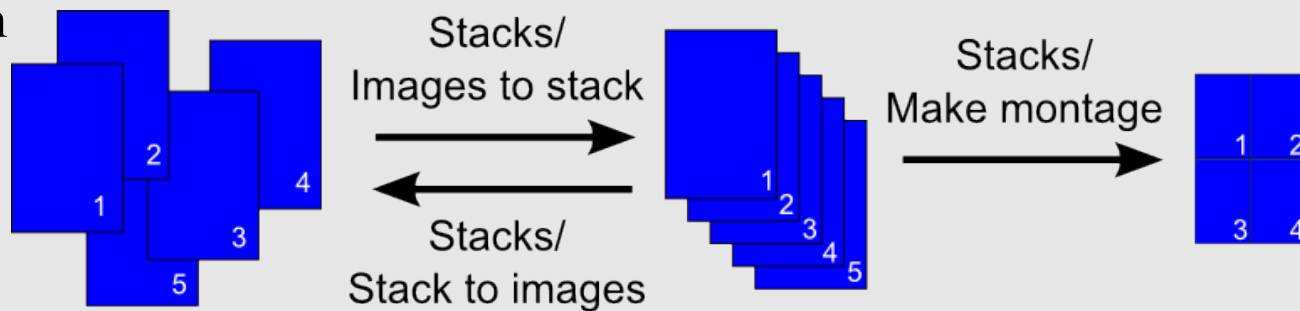
- Principe : n images (« *slices* ») de mêmes type et dimensions, dans une seule fenêtre (munie d'un sélecteur).
- Utilité : représentation
 - de l'évolution d'une scène dans le temps (t), ou
 - d'une série d'images acquises à différentes profondeurs (z)



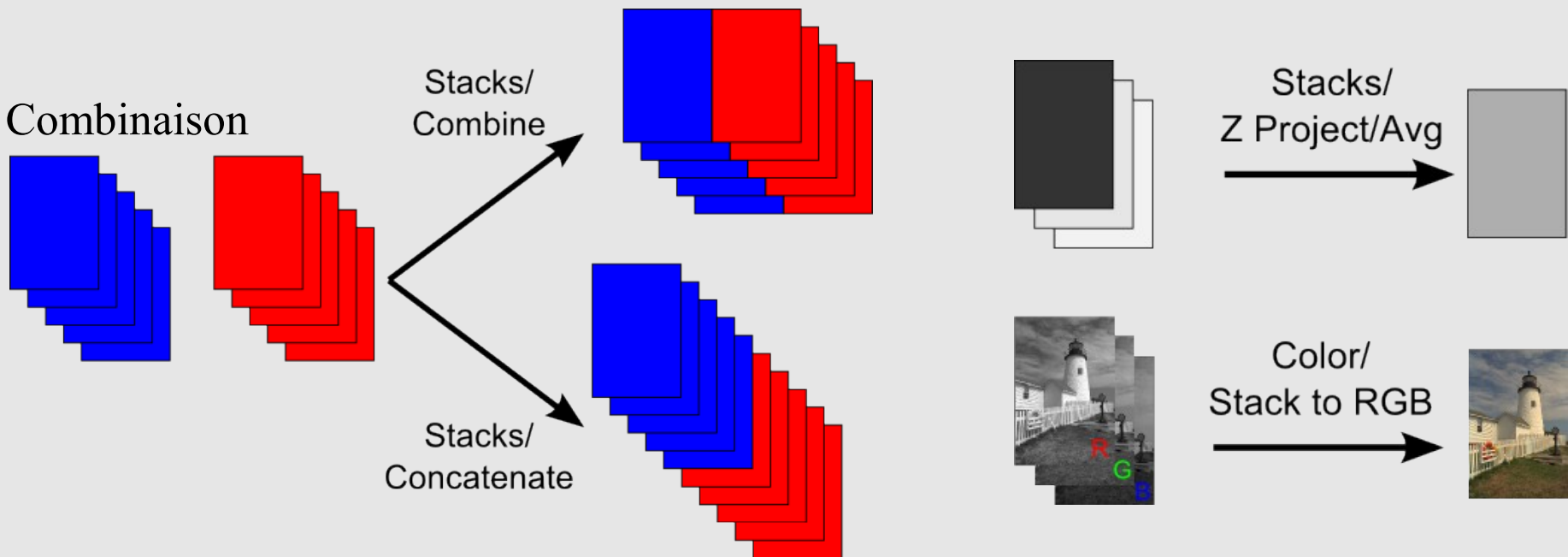
Représentation des images (4/5)

• Piles d'images : utilisation

→ Création



→ Combinaison



Représentation des images (5/5)

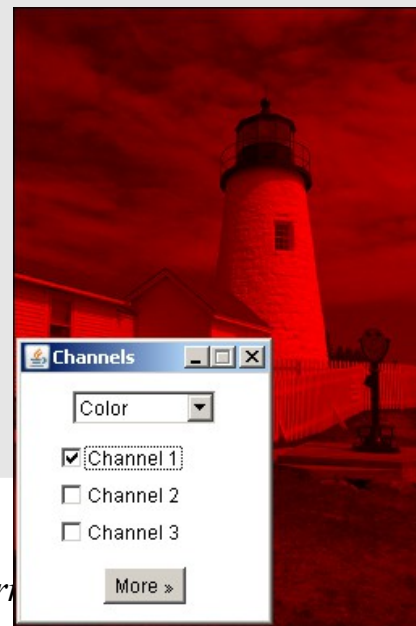
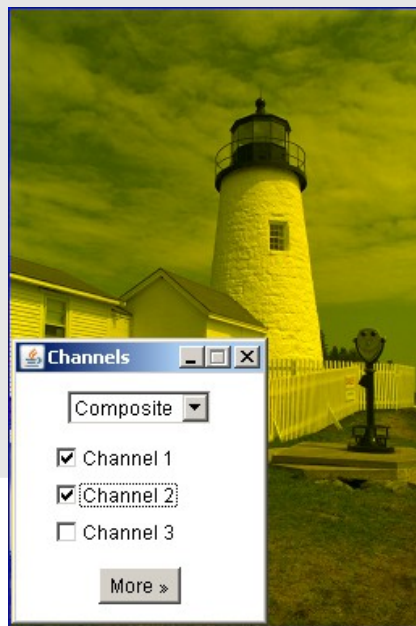
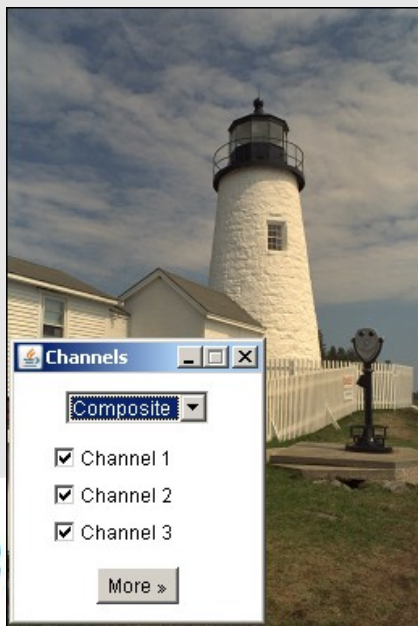
• Images composites

- Récemment introduit (ImageJ 1.38n, mars 07), et pas exploité par les plugins anciens.
- Principe : semblable aux « couches » ou « calques ».

• Avantages

- chaque canal est indépendant et peut être pris en compte ou pas pour l'affichage.
- chaque canal est clairement identifié par une couleur propre.
- chaque canal peut stocker des données 16 bits (8 en RGB).
- possibilité de fusionner plus de 3 canaux.
- visualisation directe de la fusion des canaux.

1/3 (Red); 256x384 pixels; 16-bit; 5
2/3 (Green); 256x384 pixels; 16-bit; 5
3/3 (Blue); 256x384 pixels; 16-bit; 5



Présentation des macros (1/2)

- **Principe : écrire un programme (*macro*) en langage de script offrant :**
 - les structures de contrôle classiques (boucles, tests) ;
 - toutes les opérations accessibles depuis les menus d'ImageJ ;
 - une écriture modulaire (fonctions utilisateur) ;
 - des fonctionnalités de débogage.
- **But : Automatiser un ensemble d'opérations**
 - Enregistrer une séquence d'opérations (*Plugins/Macros/Record...*)
 - Répéter ces opérations sur une autre image d'entrée (*Plugins/Macros/Run...*)
 - Systématiser ces opérations sur un ensemble de fichiers images
 - Paramétrer ces opérations (*Plugins/Macros/Edit...*)
 - en fonction des saisies de l'utilisateur
 - en fonction du contenu des images

Présentation des macros (2/2)

- **But : automatiser un ensemble d'opérations (*exemples*)**

- Enregistrer une séquence d'opérations

```
open( "./images/photo.tif" );  
saveAs( "jpeg", "../images/photo.jpg" );  
run( "Close" );
```

- Systématiser ces opérations sur un ensemble de fichiers images

```
liste = getFileList("./images/");  
for ( i=0; i<liste.length; i++)  
    if ( endsWith(liste[i], ".tif" ) ) {  
        open( liste[i] );  
        saveAs( "jpeg", "../images/"+File.nameWithoutExtension+".jpg" );  
        run("Close");  
    }
```

- Paramétrer ces opérations en fonction des saisies de l'utilisateur

```
rep = getDirectory("Choix d'un répertoire");  
liste = getFileList( rep );  
for (i=0; i<liste.length; i++)  
    if (endsWith(liste[i], ".gif")) {  
        open( liste[i] );  
        saveAs( "jpeg", rep+File.nameWithoutExtension+".jpg" );  
        run("Close");  
    }
```

Fonctions disponibles dans les macros

- **Liste exhaustive :** <http://rsbweb.nih.gov/ij/developer/macro/functions.html>
- **Quelques exemples**

- ➔ **Image**

```
nImages; nSlices; // Nombre d'images ouvertes ; nombre d'images dans pile courante
getImageID(); getWidth(); getHeight(); // Identifiant, largeur, hauteur fenêtre
selectImage(id); selectWindow(name); // Sélection fenêtre par identifiant ou nom
pixelValue = getPixel(x,y); // Lit la valeur du pixel (x,y) (retourne un int)
setPixel(x,y); // Écrit valeur (màj écran avec updateDisplay() ou en fin de macro)
getSelectionBounds(x,y,width,height); // Lire le rectangle englobant la sélection
```

- ➔ **I/O, commandes ImageJ**

```
print(chaine); // Affiche chaîne dans fenêtre Log (concaténation : +, cast: auto)
getNumber(prompt,defaultVal); getString(prompt,defaultStr); //Saisies
run(command[,options]); // Exécute une commande du menu ImageJ
setBatchMode(bool); // Mode « batch » (sans affichage) ou non
Ne pas afficher les fenêtres pendant l'exécution (bool=true) accélère jusqu'à 20x.
```


Limites et préconisations d'emploi

• Limites des macros

- Médiocre temps d'accès aux pixels, même en mode « batch ».
- Nombre de fonctionnalités (opérations, interaction, débogage) limité.

• Préconisations d'emploi

- Solution adaptée pour :
 - effectuer des opérations de mesures sur un ensemble de fichiers
 - plus généralement, systématiser des traitements pré-implémentés
 - développer rapidement des programmes prototypes avec interaction limitée :
 - utiliser l'**enregistreur** (*Plugins/Macros/Record...*) pour trouver les fonctions.
 - s'inspirer des nombreuses macros **exemples** (<ij>/macros).
- Solution peu adaptée pour :
 - effectuer des traitements systématiques (pixel à pixel), nécessitant un développement spécifique, d'images de grande taille
 - et/ou des applications avec interfaces utilisateur complexes.
- Envisager alors le développement de *plugins*.

Présentation des plugins

• Définition

- Module écrit en Java (classe) permettant d'étendre les fonctionnalités d'ImageJ.
- Utilise les classes de l'API ImageJ (<http://rsbweb.nih.gov/ij/developer/api/>).

• Installation

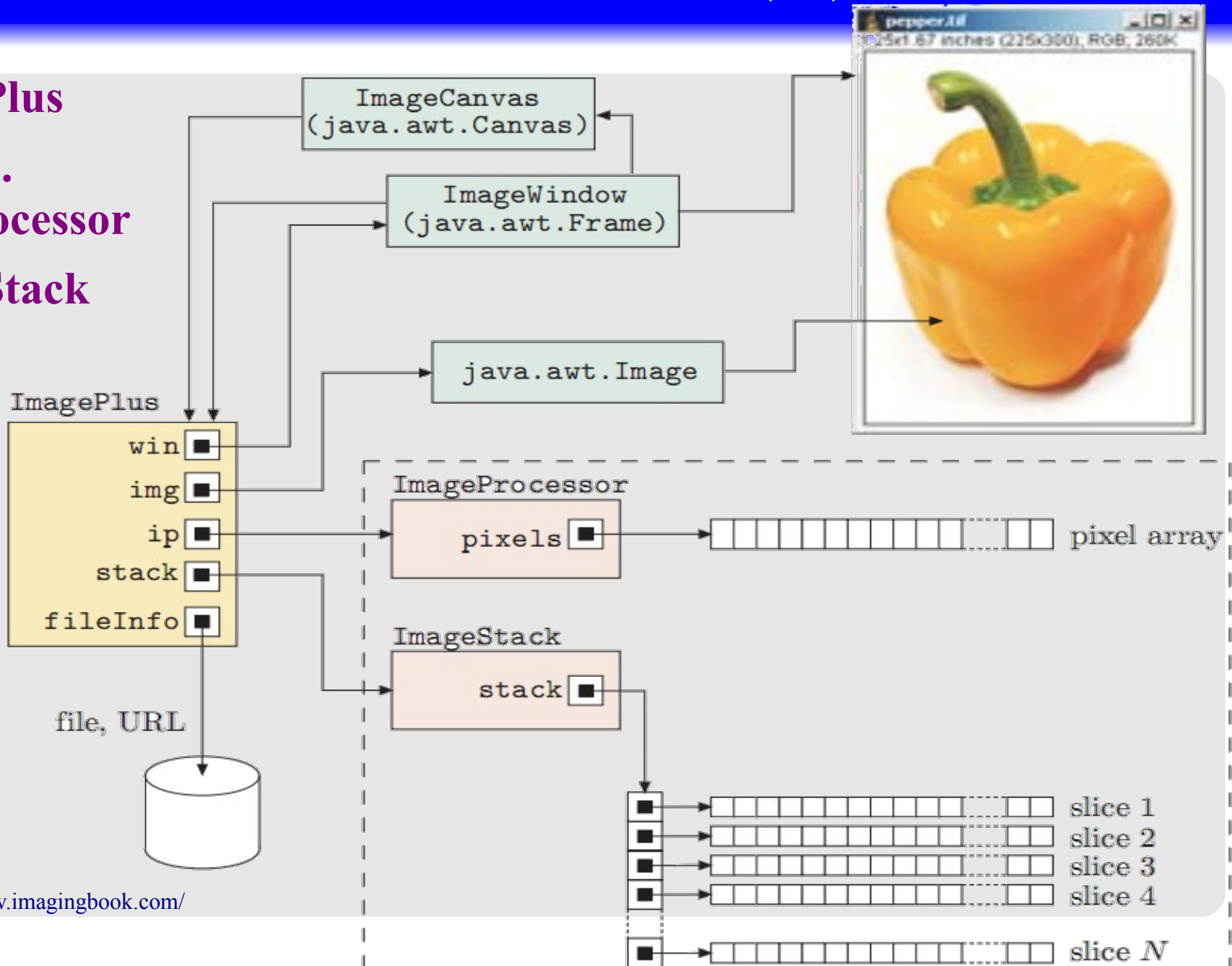
- Télécharger l'un des nombreux plugins sur <http://rsbweb.nih.gov/ij/plugins>.
- Copier simplement le fichier .class ou .jar dans (un sous-répertoire de) <ij>/plugins.
- Redémarrer ImageJ pour que l'installation soit prise en compte (*cf.* menu *Plugins*)
- Remarques
 - Des plugins sont pré-installés dans <ij>/plugins.
 - Tout plugin doit contenir un **underscore** (`_`) dans son nom.

• Développement

- Écrire un fichier .java contenant
 - un underscore dans son nom ;
 - une classe de même nom, implémentant l'interface `PlugIn` ou `PlugInFilter`.
- Compiler ce fichier (*Plugins/Compile and Run...*) pour générer le fichier .class.
- Pour un développement et un débogage plus aisé, *cf.* wiki dans les [références](#).

Classes fondamentales de l'API (1/2)

- **ij.ImagePlus**
- **ij.process.
ImageProcessor**
- **ij.ImageStack**
- **ij.gui.Roi**



Source : <http://www.imagingbook.com/>

Classes fondamentales de l'API (2/2)

• **ij.ImagePlus**

- Représente une fenêtre contenant une image et permet d'interagir avec elle.
- Ses éléments sont accessibles par les méthodes d'instance, par exemple :
 - `ImageWindow getWindow()` : la fenêtre elle-même
 - `ImageProcessor getProcessor()` : le processeur traitant les données de l'image
 - `ImageStack getStack()` : la pile d'images (éventuellement réduite à 1 image)
 - `Roi getRoi()` : la région d'intérêt courante (zone sélectionnée)
 - `ImageCanvas getCanvas()` : le « canevas » utilisé pour représenter l'image dans la fenêtre (rectangle, facteur de zoom, ...) et en traiter les événements
 - `ColorModel getColorModel()` : le modèle couleur (ou la LUT) de représentation de l'image
 - `Overlay getOverlay()` : les éléments superposés à l'image
 - `FileInfo getFileInfo()` : le fichier image

• **ij.process.ImageProcessor**

- Classe abstraite permettant de traiter ou convertir une image.
- Ses sous-classes sont adaptées aux données contenues : `ByteProcessor` (et `BinaryProcessor`), `ShortProcessor`, `FloatProcessor`, `ColorProcessor`.
- *Rem.*: l'image traitée n'est pas forcément affichée à l'écran.

Classes fondamentales de l'API (2/2)

• **ij.ImagePlus**

- Représente une fenêtre contenant une image et permet d'interagir avec elle.
- Ses éléments sont accessibles par les méthodes d'instance, par exemple :
 - `ImageWindow getWindow()` : la fenêtre elle-même
 - `ImageProcessor getProcessor()` : le processeur traitant les données de l'image
 - `ImageStack getStack()` : la pile d'images (éventuellement réduite à 1 image)
 - `Roi getRoi()` : la région d'intérêt courante (zone sélectionnée)
 - `ImageCanvas getCanvas()` : le « canevas » utilisé pour représenter l'image dans la fenêtre (rectangle, facteur de zoom, ...) et en traiter les événements
 - `ColorModel getColorModel()` : le modèle couleur (ou la LUT) de représentation de l'image
 - `Overlay getOverlay()` : les éléments superposés à l'image
 - `FileInfo getFileInfo()` : le fichier image

• **ij.process.ImageProcessor**

- Classe abstraite permettant de traiter ou convertir une image.
- Ses sous-classes sont adaptées aux données contenues : `ByteProcessor` (et `BinaryProcessor`), `ShortProcessor`, `FloatProcessor`, `ColorProcessor`.
- *Rem.*: l'image traitée n'est pas forcément affichée à l'écran.

Types de plugins

- **2 interfaces définissant 2 types de plugins**

- ➔ Un plugin implémentant l'interface **PlugIn** ne nécessite pas d'image en entrée.
- ➔ Un plugin implémentant l'interface **PlugInFilter** nécessite une image en entrée.

- **Plugin implémentant PlugIn**

- ➔ 1 seule méthode appelée, qui implémente ce que réalise effectivement le plugin :
`void run(java.lang.String arg)`

- **Plugin implémentant PlugInFilter**

- ➔ la première méthode appelée initialise le plugin :
`int setup(java.lang.String arg, ImagePlus imp)`
 - `imp` est l'image (*i.e.* la fenêtre contenant l'image) sur laquelle travaille le plugin
 - retourne ce que le plugin attend en entrée (type(s) d'image, région d'intérêt, ...)
- ➔ la seconde méthode implémente ce que réalise effectivement le plugin :
`void run(ImageProcessor ip)`
 - `ip` est l'image (*i.e.* le processeur accédant aux pixels) sur laquelle travaille le plugin
 - la **fenêtre** contenant l'image n'est accessible dans `run()` que si elle a été préalablement stockée dans une variable d'instance à l'exécution de `setup()`

Entrées d'un `PlugInFilter`

- La méthode `setup()` retourne une combinaison des constantes (`int`) :

- Types d'images : le plugin traite ...

- `DOES_8G` : des images en niveaux de gris sur 8 bits (entiers positifs)
- `DOES_16` : des images en niveaux de gris sur 16 bits (entiers positifs)
- `DOES_32` : des images en niveaux de gris sur 32 bits (flottants signés)
- `DOES_RGB` : des images couleur RGB
- `DOES_8C` : des images couleur indexées (256 couleurs)
- `DOES_ALL` : des images de tous les types précédents
- `DOES_STACK` : toutes les images d'une pile (applique `run()` sur chaque image)

- Type d'action : le plugin ...

- `DONE` : effectue seulement son initialisation `setup()`, sans appeler la méthode `run()`
- `NO_CHANGES` : n'effectue aucune modification sur les valeurs des pixels
- `NO_UNDO` : ne nécessite pas que son action puisse être annulée

- Paramètres d'entrées supplémentaires : le plugin ...

- `STACK_REQUIRED` : exige en entrée une pile d'images
- `ROI_REQUIRED` : exige qu'une région d'intérêt (*RoI*) soit sélectionnée
- `SUPPORTS_MASKING` : demande à ImageJ de rétablir, pour les *RoI* non rectangulaires, la partie de l'image comprise dans la boîte englobante mais à l'extérieur de la *RoI*

Exemple (1/2)

• Inversion d'une image

```
// Importation des paquets nécessaires. Le plugin n'est pas lui-même un paquet (pas de mot-clé package)
import ij.*;    // pour classes ImagePlus et IJ
import ij.plugin.filter.PlugInFilter;    // pour interface PlugInFilter
import ij.process.*;    // pour classe ImageProcessor
import java.awt.*;    // pour classe Rectangle
// Nom de la classe = nom du fichier. Implémente l'interface PlugInFilter
public class Image_Inverter implements PlugInFilter {

    public int setup(String arg, ImagePlus imp) {
        if (IJ.versionLessThan("1.37j")) // Requiert la version 1.37j d'ImageJ
            return DONE;    // Ne pas appeler la méthode run()
        else    // Accepte tous types d'images, piles d'images et RoIs, même non rectangulaires
            return DOES_8G+DOES_RGB+DOES_STACKS+SUPPORTS_MASKING;
    }

    public void run(ImageProcessor ip) {
        Rectangle r = ip.getRoi();    // Région d'intérêt sélectionnée (r.x=r.y=0 si aucune)
        for (int y=r.y; y<(r.y+r.height); y++)
            for (int x=r.x; x<(r.x+r.width); x++)
                ip.set(x, y, ~ip.get(x,y));    // Complément bit à bit des valeurs des pixels
    }
}
```

Accès aux pixels (1/3)

• Accès ponctuel à un pixel dans un `ImageProcessor`

- ➔ Avec vérification des coordonnées : `int getPixel(int x, int y)`
 - Retourne un **entier** sur 4 octets :
 - pour `ColorProcessor`, l'entier stocke les 4 composantes dans l'ordre **αRGB**.
ou : `int[] getPixel(int x, int y, int[] iArray)`. **R** alors en [0] (**α** inutilisé).
 - pour `FloatProcessor`, l'entier *p* contient les bits correspondant au flottant lu et doit être converti avec `Float.intBitsToFloat(p)`.
Ou : utiliser `float getPixelValue(int x, int y)`.
 - Si *x* ou *y* hors limites, retourne 0 (et `putPixel(x,y)` n'a pas d'effet).
- ➔ Sans vérification des coordonnées (plus rapide) : `int get(int x, int y)`
- ➔ Accès par coordonnée unique : `int get(int pix_index)`
 - utile si les coordonnées ne sont pas utilisées, *ex.* pour des opérations ponctuelles
 - $0 \leq \text{pix_index} \leq \text{getPixelCount}()$
 - pour `FloatProcessor`, utiliser `anImageProcessor.getf(pix_index)`.
- ➔ Accès en écriture :
 - `void putPixel(int x, int y, int value), putPixelValue(int x, int y, double value)`
 - `void set(int pix_index, int value), setf(int pix_index, float value)`

Accès aux pixels (2/3)

• Accès global aux pixels d'un `ImageProcessor`

→ Accès aux pixels dans un tableau 1D : `Object getPixels()`

- Nécessite une conversion, car le type du tableau dépend du type de processeur :

```
if (anImageProcessor instanceof ColorProcessor)
    int[] pixels = (int[]) anImageProcessor.getPixels();
```

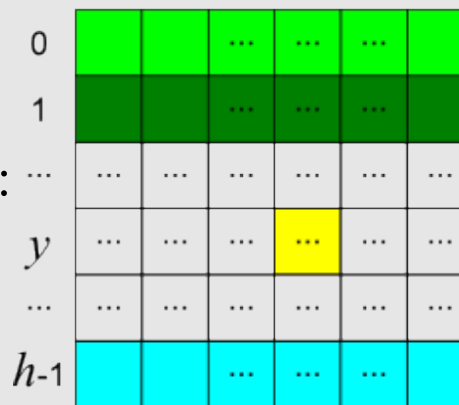
0 1 ... x ... w-1

- Le tableau

retourné est

mono

dimensionnel :



`getPixels()`



- Plus performant qu'un accès ponctuel (*ex.* `getPixel()`) à chacun des pixels.
- Cas part. d'un `ColorProcessor` : `void getRGB (byte[] R, byte[] G, byte[] B)`
- `Object getPixelsCopy()` retourne une **copie** des pixels (tableau *undo*).

→ Accès aux pixels dans un tableau 2D

- Pour un `{Byte|Short|Color}Processor` : `int[][] getIntArray()` // coords : `[x][y]`
- Pour un `FloatProcessor` : `float[][] getFloatArray()`

Accès aux pixels (3/3)

Élimination du bit de signe

→ Problème

- Les méthodes d'accès (ponctuel ou global) aux pixels retournent des valeurs entières **signées** (byte, short, int) ; *ex.* byte $\in [-128..+127]$.
- Or le plus souvent, on souhaite des valeurs **positives** de luminance ($[0..255]$, etc.).

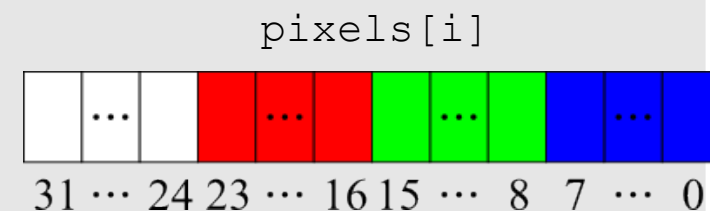
→ Solution : conversion en entier (int) **non signé** (positif) par **masquage**.

- Exemple pour une image *ip* en niveaux de gris sur 8 bits (8G)

```
byte[] pixels=(byte[]) ip.getPixels();
...
int grey=pixels[i] & 0xFF;      // pixels[i] & 0xFFFF pour short[]
```

- Exemple pour une image *ip* couleur sur 32 bits (RGB)

```
int[] pixels=(int[]) ip.getPixels();
...
int r=(pixels[i] & 0xFF0000)>>16; // rouge
int g=(pixels[i] & 0x00FF00)>>8;  // vert
int b=(pixels[i] & 0x0000FF);     // bleu
```



Exemple (2/2)

• Compter le nombre de couleurs différentes présentes dans une image

// Importation des paquets (non détaillée)

```
public class Color_Counter implements PlugInFilter {
    ImagePlus imp;
    int colors;
    static int MAX_COLORS = 256*256*256;
    int[] counts = new int[MAX_COLORS]; // Histogramme des couleurs
    public int setup(String arg, ImagePlus imp) {
        this.imp = imp; // Rem.: pas utilisé dans run()
        return DOES_RGB+NO_UNDO+NO_CHANGES; // Traite les images couleurs ; pas d'annulation
    }
    public void run(ImageProcessor ip) {
        int[] pixels = (int[])ip.getPixels();
        for (int i=0; i<pixels.length; i++)
            counts[pixels[i]&0xffffffff]++; // Masquage nécessaire car pixels[i] est signé
        for (int i=0; i<MAX_COLORS; i++) { // Comptage des couleurs différentes
            if (counts[i]>0) colors++;
        }
        IJ.log("Couleurs différentes : "+colors);
        if (colors<=64) { // Affichage des couleurs trouvées dans fenêtre Log (si elles sont moins de 64)
            IJ.log("Counts");
            for (int i=0; i<MAX_COLORS; i++) {
                if (counts[i]>0) IJ.log("    "+Integer.toHexString(i)+" : "+counts[i]);
            }
        }
    }
}
```


Sélection de références

• Livre et wiki

- ➔ W. Burger, M. J. Burge, *Digital Image Processing – An Algorithmic Introduction using Java*, Springer 2008
<http://www.imagingbook.com/>
- ➔ wiki : nombreuses ressources (FAQ, etc.), dont :
P. Pirotte, « [Include ImageJ into Eclipse to Develop Plugins](#) »

• Tutoriels

- ➔ plusieurs tutoriels, dont « *Programmation ImageJ* » (avec intro à Java)
<http://www.ijm.fr/imagerie/equipements-disponibles/traitement-et-analyse-dimages/imagej/>
- ➔ A. Podlasov, E. Aggenko, *Working and Development with ImageJ – A student reference*, 2003.
<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.78.4181&rep=rep1&type=pdf>
- ➔ W. Bailer, *Writing ImageJ PlugIns – A Tutorial*, 2006
W. Burger, M. J. Burge, *ImageJ Short Reference*, 2007
<http://www.imagingbook.com/index.php?id=102>
- ➔ ressources par J. Ross, dont « *Using and Writing Macros in ImageJ* », 2007
http://www.fmhs.auckland.ac.nz/sms/biru/facilities/analysis_resources.aspx