Fiji Training Notes

6th Edition
June, 2016

MONASH University

Contents

About this Manual 1
Introduction 1
A Note on Image Analysis 1
Distribution 1
Authors 2
Requirements 2
Initial Setup 2
Image Contributors 4
Introduction to Fiji 1
Introduction 1
Where to get Fiji 1
Configuration 2
Memory and Processor Threads 2
Proxy Settings 4
Updates 5
Adding an Additional Update Site 7
Monitoring and Clearing up Excess Memory Usage 9
Monitoring Memory 9
Finding Commands 10
Basic Module 12
Basic Navigation in Fiji 19
Aim 19
Opening Files 19
Zooming In and Out 20
Panning the Image 20
Duplicating an Image 21
Regions of Interest (ROI) 22
Types of ROIs 22
Drawing ROIs 29
Removing ROIs 29

Copying/Moving ROIs to Another Image 29
Moving ROIs 29
Resizing ROIs 30
Using the ROI Manager to add Multiple ROIs 30
Adjusting Brightness and Contrast 32
Aim 32
Adjusting Brightness and Contrast – Colour Images 32
Adjusting the Colour Balance – Colour Images 34
Adjusting the Brightness and Contrast – Monochrome Images 36
Calibrating and Adding a Scale Bar 39
Aim 39
Calibrating an Image 39
Adding a Scale Bar 41
Fluorescent Images 42
Aim 42
Changing Bit Depth – 24bit RGB 43
Adding Colour – Using Lookup Tables 44
Single Colour LUTs 44
Multicolour LUTs 45
Intensity Distribution 46
Merging Images 48
Merging Up To 7 Channels and DIC/Phase/Brightfield – Standard Colours 48
Merging Up To 7 Channels and DIC/Phase/Brightfield – Non-standard Colours 50
Merging 8 or more colour channels 53
Simulating Colour Blindness 56
Opening Advanced Data Sets 58
Aim 58
LOCI Tools 58
Initial File Opening 58
Displaying Metadata 59
Default Opening States 59

Opening Confocal Data 61
Leica – LIF Format 61
Nikon – ND2 File Format 66
Olympus – OIB and OIF File Format 70
Zeiss – LSM File Format 72
Opening DeltaVision Data 73
Visualising 3D and Live Data 75
Aim 75
Visualising 3D Data 75
Z Projection 75
Orthogonal Projection 77
Orthogonal Stack/Reslice Stack 78
Slice Montage 79
3D Volume – Simple 81
Kymograms 84
Making Movies 87
Simple Analysis – Area and Intensity of Stain 90
Aim 90
Measuring the Area and Intensity of Stain - Fluorescence 90
Separating Merged Multichannel Images 90
Threshold Image 91
Automatic Threshold Algorithms 93
Measure Thresholded Area 94
Measuring the Area of Stain – Brightfield/Chromagens 97
Make a Brightfield Image Look like a Fluorescent One 99
Cell Counting 101
Aim 101
Initial Binary Creation 101
Counting Cells – Basic Count 104
Counting Cells – Detailed Information 107
Cell Segmentation and Analysis 111

Aim 111
Creating Whole Cell Binary Masks 111
Analysing the Image 119
Checking the Effectiveness of Segmentation 121
Method 1 – ROI Manager 121
Method 2 – Create Colour Overlays 124
Complex Segmentation 127
Refinement of Compartment Masks 129
Working with Tricker Stains 131
Cell Morphology and Shape Change 134
Aim 134
Measuring Cell Complexity 134
Intensity over Time 142
Aim 142
Single Point Analysis 142
Multi Point Analysis – Manual ROI Entry 145
Multi Point Analysis – Automatic ROI Entry 149
Image Filters – Fixed and Live Data 154
Aim 154
Fixed Sample Analysis – Fluorescence 154
Selecting the Scratch 154
Measuring the Scratch and Logging the Data 156
Removing Unwanted Area from the Analysis 158
Method 1 – Size exclusion 158
Method 2 – Using Image Filters 160
Binary Reconstruction of the Mask 163
Create an Outline of the Result 166
Fixed Sample Analysis – Brightfield 171
Live Sample Analysis 175
Selecting the Scratch 175
Measuring Scratch Area 179

Creating Overview Movie of Result 181
Euclidean Distance Measurements 184
Aim 184
Measuring Euclidean Distance – Object Distribution 184
Creating a Distance Map 184
Measuring Distances 188
Measuring Euclidean Distance – Distance between Different Objects 190
Initial Image and Masks 190
Generating an EDM Map 192
Measuring the Distances 194
Skeletonize and Branching 196
Aim 196
Measuring Length using Skeleton Masks – Discrete Objects 196
Generating Masks 196
Measuring Masks 199
Measuring Length using Skeleton Masks – Connected or Networked Objects 202
Generating Skeleton Mask 202
Measuring the Mask 204
Generating color infographics of the results 205
Image Stitching 210
Aim 210
Tips for Capturing Your Images 210
Manually Creating a Montage 210
Automatically Creating a Montage – Brightfield 214
Automatically Creating a Montage – Fluorescence 217
Image Stack Alignment 220
Aim 220
Auto Alignment of Confocal Data 220
Auto Alignment of Serial Sections 222
Extended Depth of Focus 224
Aim 224

Extended Depth of Focus – Macro Images 224
Extended Depth of Focus – Microscope Images 227
3D Visualisation and Measurement 231
Aim 231
3D Visualisation 231
Visualisation 231
Making Movies 234
3D Measurement 238
Measuring Objects in 3D 238
Generating 3D models of the Results 242
Advanced Image Segmentation 245
Aim 245
Training the Module 245
Running the Analysis 253
Reviewing the Data 256
Analysing the Data 260
Advanced Module 261
About the Advanced Module 263
About 263
Basic Macro Recording 264
Aim 264
The Macro Recorder 264
Recording your First Macro – Threshold Measurement 265
Re-running the Macro 269
Recording your Second Macro – Cell Counting 271
An Introduction to Variables 275
Aim 275
The Macro Editor 276
Variables 276
Writing your first Macro – Automatically Merge Images 277
Adding Custom Naming 283
Merge DAPI, FITC and DIC.ijm – Final Code 285

Writing Your Second Macro – Channel Separate and Threshold Measurement 286
Vessel and Hypoxia Measurment.ijm – Final Code 290
Outputting a Result Overview 291
Vessel and Hypoxia Measurement (with Result Output).ijm – Final Code 293
Batch Processing 295
Aim 295
Batch Processing Concepts 295
Batch Process - Cell Counting 295
Cell Count (Batch).ijm – Final Code 300
Batch Process - Cell Counting – Contaminated Directory 301
Cell Count (Batch – Tiff Only).ijm – Final Code 302
Batch Processing and Outputting Data 303
Vessel and Hypoxia Measurement (with Result Output).ijm – Final Code 305
Custom Data Logging 307
Aim 307
Creating a Custom Log Table 307
First Steps – Batch Code 308
Custom Log Creation 308
The Analysis Code 310
Ki67 and PH3 Count.ijm – Final Code 314
Adding Refinements – Positive Cell Selection 316
Ki67 and PH3 Count (Refined).ijm – Final Code 317
Outputting Results Images to a Macro Created Folder 319
Ki67 and PH3 Count (Refined with Output).ijm – Final Code 321
Looping for ROIs 324
Aim 324
Selecting Files 324
Generating a Custom Table 325
Defining Measurements 325
Batch Processing 326
Working with ROIs 326
Opening Images with the Bio-Formats/LOCI Tools 326

Getting Images Ready for Processing 327
Segmenting the Cells 327
Preparing the Images 328
Segmenting the Cells 328
Cleaning Up Open Images 330
Looping for All ROIs 330
Measuring the Selected ROI 331
Final Calculation and Logging Data 332
Finalising the Code 333
Saving the Data Table 333
Run the Code 333
Running it Faster – Batch Mode 334
Cell Complexity Measurement with Actin Texture.ijm – Final Code 334
Custom Dialog Boxes 338
Aim 338
Defining File Sources and Save Paths 338
Creating a Custom Dialog Box 340
Collecting Information from the Dialog 342
Running Code Based on Dialog Choices 342
Automatic Montage – Hard-coded.ijm – Final Code 344
Adaptable Version of Tile Merge 346
Automatic Montage – Adaptable.ijm – Final Code 348
Arrays 350
Aim 350
Collecting File Information 350
Dialog Creation 352
Collecting the Information from the Dialog 353
Calculating Tile Size and Number 354
Creating New Arrays 354
Calculate Coordinates 355
Checking the Calculated Coordinate Lists 356
Combining the Coordinates into an Array 356

Checking the Paired Coordinates 357
Duplicating the Regions 358
File Cropper – Adaptable.ijm – Final Code 359
Functions 362
Aim 362
Global Variables 362
Vessel and Hypoxia Measurement 363
Directory Selection and Array Function 363
Split Images Function 366
Measure Function 368
Save Overview Images Function 369
Vessel and Hypoxia Measurement (with Result Output – functions).ijm – Final Code 370
Ki67 and PH3 Count 372
Global Variables 372
File Function 372
Make Table Function 373
Channel Function 374
Count DAPI Function 374
Count Ki67 Function 375
Count PH3 Function 376
Calculate Percentage Function 376
Log Data Function 377
Save Image Output Function 377
Ki67 and PH3 Count (Refined with Output – functions).ijm – Final Code 378
Built-in ImageJ Macro Functions 383
Macro Functions 383
Macro Functions List 383

About this Manual

Introduction

This manual is designed to be used in conjunction with a series of demonstration images to show users how to perform some basic and complicated analysis using the Fiji distribution of ImageJ. It is not an exhaustive list of techniques and methods by any stretch. It does however touch on many of the key components that most users will need in carrying out image analysis using Fiji.

Additionally there is an advanced section on recording and writing macros for advanced and automated analysis.

The techniques presented in this manual, while biology based, can be applied to any type of image analysis.

This manual has evolved over six editions gaining extra techniques, extra demo images and the section on programing. The current 6th Edition has extra detail added to the basic module sections and notes on creating Kymographs and cell complexity measurement. More programing elements have been added to the advanced module section showing how to loop for regions, creating custom dialog boxes and working with arrays

A Note on Image Analysis

This manual will demonstrate how to perform specific types of image analysis. The rationale behind the choices made in each method (where applicable) will be explained. Knowing which filters, methods, techniques etc. to use for a given analysis is something that cannot be easily taught. It comes from experience and a full understanding of the underlying mathematics of image analysis.

Distribution

You are welcome to distribute this manual and its demonstration images provided it is not altered in anyway and the acknowledgements to author's and image providers' contributions remain intact. Individual copies of each of the Technical Notes can be obtained from Cameron Nowell (cameron.nowell@monash.edu).

Authors

This manual was written by Cameron J. Nowell over the course of several years (beginning in 2010) initially while at the Centre for Advanced Microscopy at the Ludwig Institute for Cancer Research, further expanded while working at the Centre for Dynamic Imaging at the Walter and Eliza Hall Institute of Medical Research with the current and previous ($4^{th,}$ 5^{th} and 6^{th}) editions at the Monash Institute for Pharmaceutical Sciences, Monash University.

While written by Cameron the content of this manual has evolved and been enhanced by the numerous suggestions of those people that have attended the many workshops that this manual has been the basis of.

Requirements

This manual relies on the users having Fiji installed. Fiji can be obtained from here

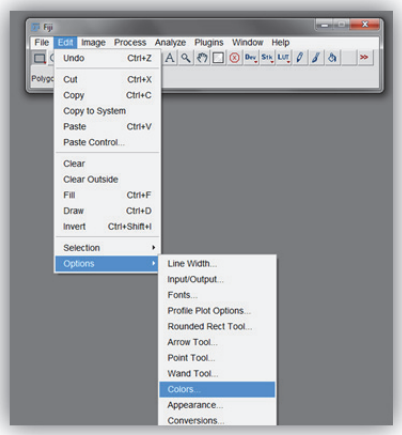
http://fiji.sc/

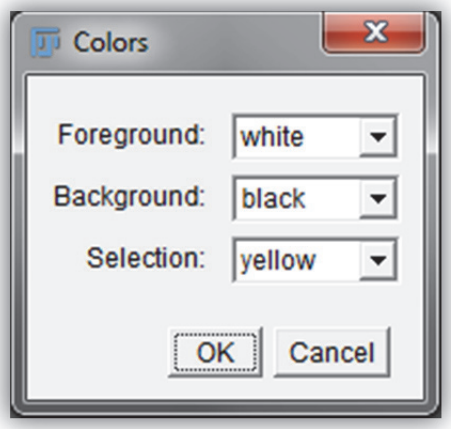
Several parts also rely on having the Morphology Operators for ImageJ plugin installed which can be added using the updates site feature of Fiji (see notes in the following introduction to Fiji chapter for this).

Initial Setup

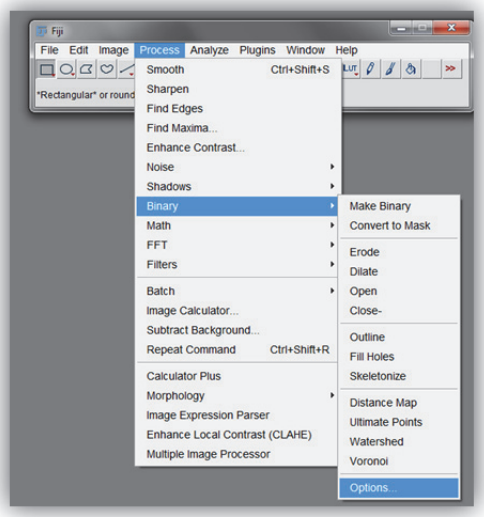
To be able to have results that match the screen shots in this manual, some of the settings within Fiji need to be set differently.

1. Set the colours of foreground, background and selection to white, black and yellow respectively. Go to **Edit → Options → Colours** and configure the dialog box as follows





2. Set the binary image options so that all images are assumed to have a black background (this stops most of the binary images that are generated looking black in white parts and vice versa). Go to **Process → Binary → Options** and set the dialog box as follows.



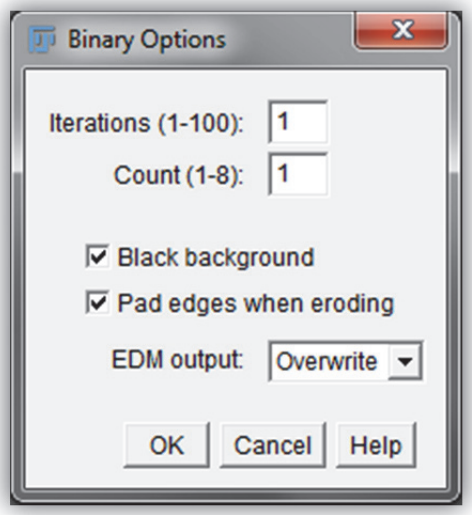


Image Contributors

This manual would not have been possible without the contribution of demonstration images from researchers from the various institutes in Melbourne and abroad.

Adam Parslow	*DIC Fish*	Ludwig Institute
Andrew Badrock	*Fish – EdU*	Ludwig Institute
Ben Williams	*Olympus.oib*	Ludwig Institute
	Colour Blindness.tif	
Cameron Nowell	*4 Channel Fluorescence*	Ludwig Institute
	Advanced Segmentation	
	Fluorescent Montage	
	H and E	
	Live Wound Assay	
Daniel Brown	*Fixed Wound Assay*	Peter MacCallum Cancer Centre
Group 1	*DeltaVision files*	Monash Live Imaging Course 2010
Group 4	*Nikon ND2 files*	Monash Live Imaging Course 2011
	Calcium Sensing	
Kelly Rogers	*Zeiss.lsm*	Walter and Eliza Hall Institute
Kerryn Elliott	*Phase Wound Healing*	University of Gothenburg
Kim Pham	*Leica.lif*	Peter MacCallum Cancer Centre
Maree Faux	*Nikon IDS Files*	Ludwig Institute
Mohammad Azad	*Mitochondria*	Monash University
Molecular Devices	*Calcium Flux*	
Paul Rigby	*Fish – MP Stack*	University of Western Australia
Rae Farnsworth	*DIC and Fluorescence*	Ludwig Institute
Richard Young	*DAB*	Peter MacCallum Cancer Centre
	Fluorescence Measurement	
Rik Littlefield	*Deer*	
Steve Williams	*Segmentation*	Peter MacCallum Cancer Centre
	Cell Scoring and Cycle	
	Batch Processing	
Tae-Hyung Kim	*Cell Complexity*	UCLA School of Medicine
TrakEM2 Team	*TEM Stack*	University of Zurich
Xiang Liu	*Tracking*	University of Melbourne

Introduction to Fiji

Introduction

Fiji is a distribution of the free image analysis package ImageJ, Fiji stands for **F**iji **I**s **J**ust **I**mageJ. Fiji comes with a large collections of plugins already installed. It also has a built in auto update system to make sure you always have the current versions of the plugins. Fiji has a biology slant to most of the plugins provided but they can be used for analysis of a range of samples.

Where to get Fiji

Fiji can be downloaded from the Fiji website located here

http://fiji.sc/

This site also contains a comprehensive wiki site of what the various plugins do and some tutorials on doing simple analysis

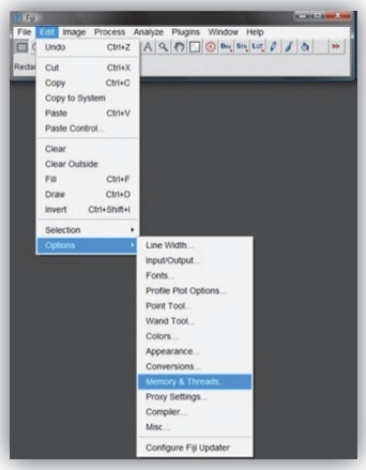
Configuration

Several aspects of Fiji need to be configured correctly for it to perform optimally.

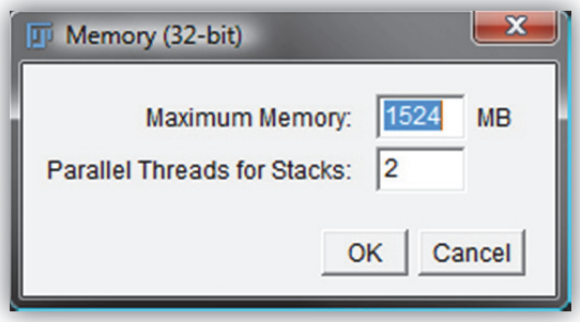
Memory and Processor Threads

By default Fiji allocates a maximum amount of system memory to itself. This value can be increased if required. Also the amount of simultaneous threads that can be ran at the same time (limited by processor cores) can be adjusted.

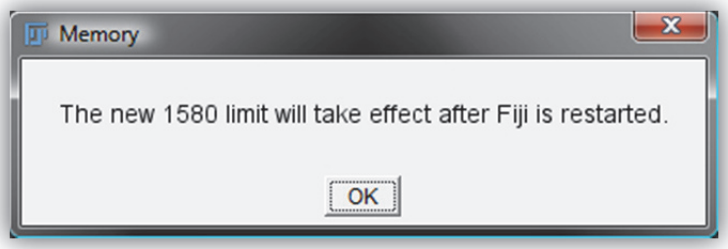
1. Go to **Edit → Options → Memory** and Threads



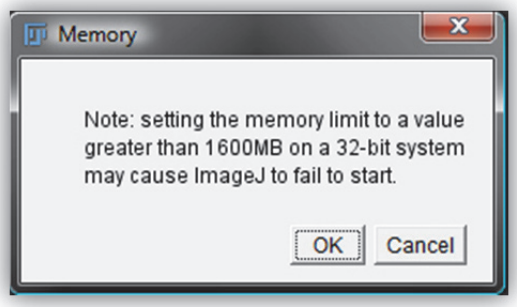
2. Set the amount of memory to what your system can handle. Don't allocate all your system memory to Fiji, usually around 60-70% is a good amount. Also set the number of threads to the number of processor cores you have.



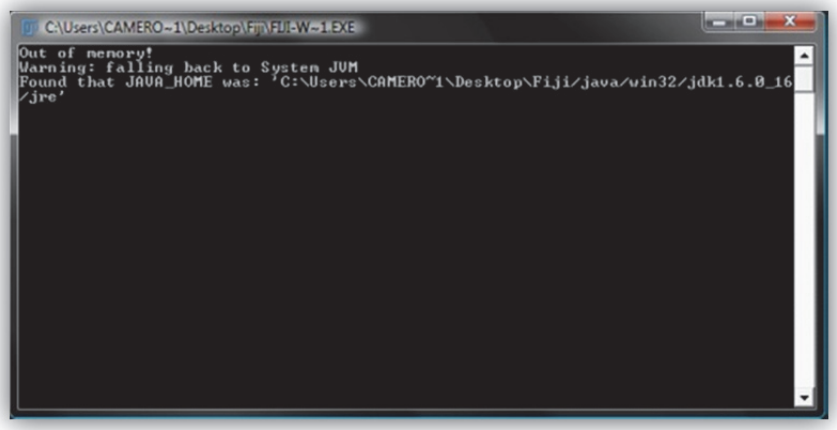
3. Fiji will then tell you that the changes will take effect after you restart.



NOTE: On 32bit systems the maximum memory cannot be set higher than 1600MB. If you set it higher you will receive this warning:



If you choose to continue Fiji may not start next time you try to use it and you will receive error messages like this

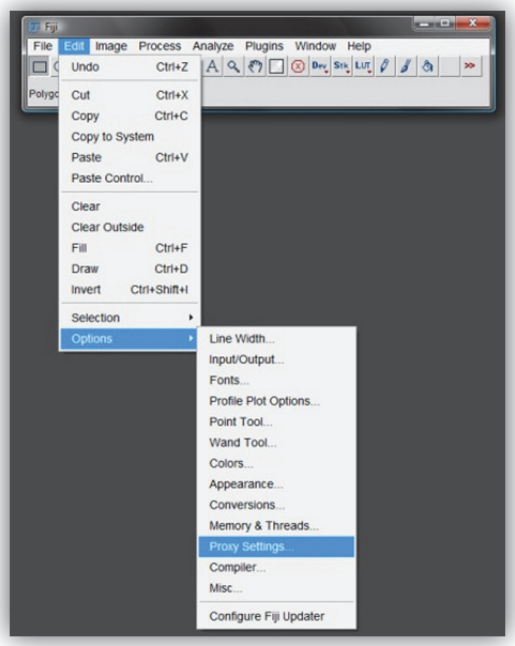


To fix the error simply delete the jvm.cfg file located in the Fiji directory, then restart Fiji.

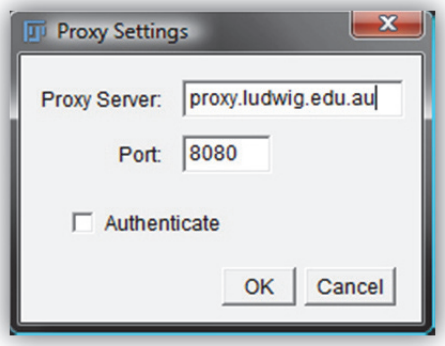
Proxy Settings

Depending on where you are connecting to the Internet you may need to have a proxy server configured to be able to access updates etc. If you are unsure about this check with your IT department.

1. Go to **Edit → Options → Proxy Settings**



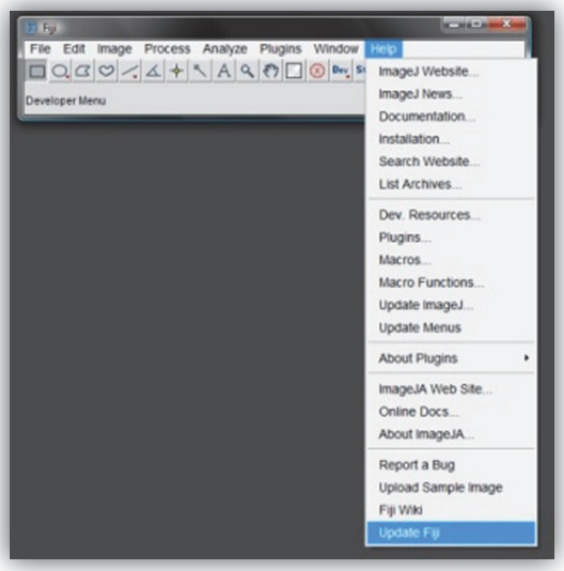
2. Enter the address and port for your proxy server (in this case proxy.ludwig.edu.au port 8080). Tick the Authenticate box if your proxy requires you to log in.



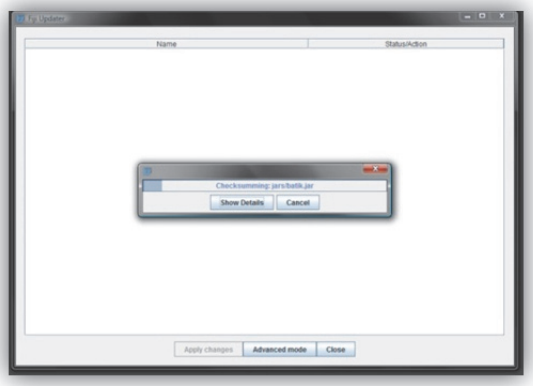
Updates

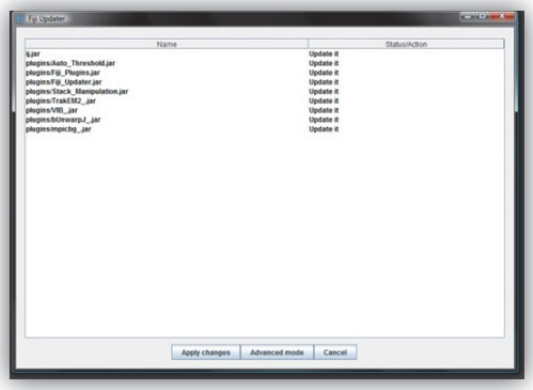
Fiji has the ability to automatically update the installed plugins and core files.

1. Go to **Help → Update Fiji**

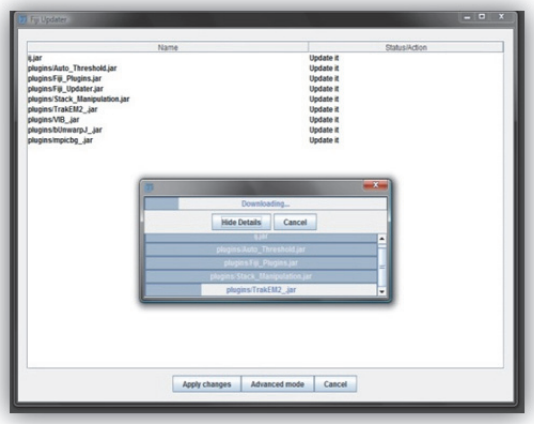


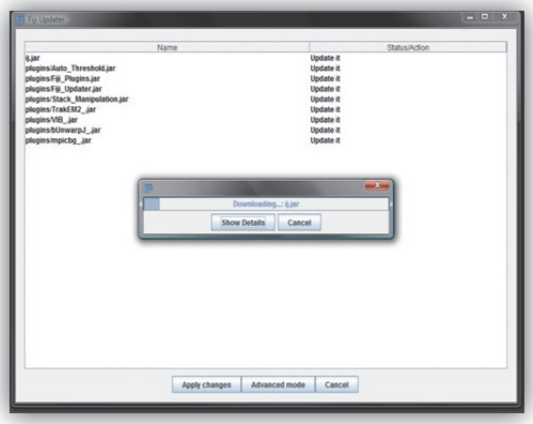
2. The updater will then check what needs to be updated and provide you with a list



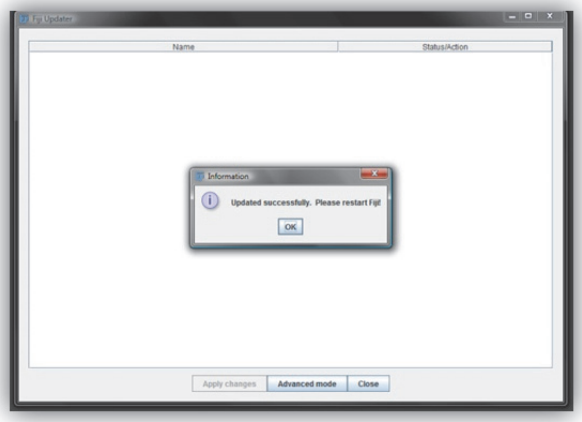


3. Press the Apply Changes button to start downloading the updates. Pressing the advanced button when downloading begins will give you more detail of what is happening





4. When the downloading is complete Fiji will ask you to restart to apply the changes

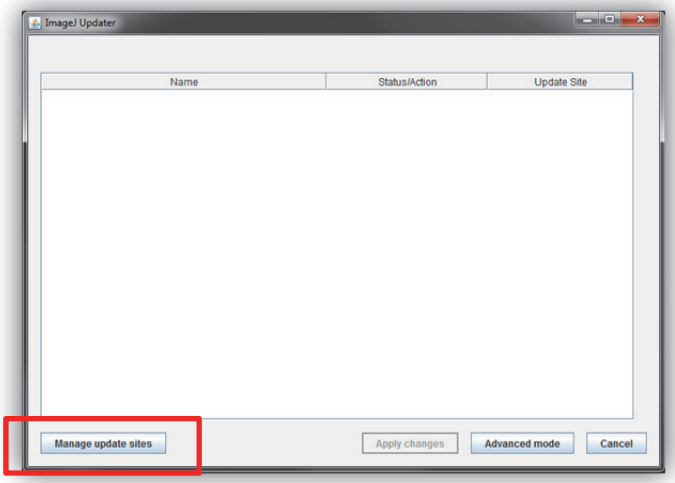


Adding an Additional Update Site

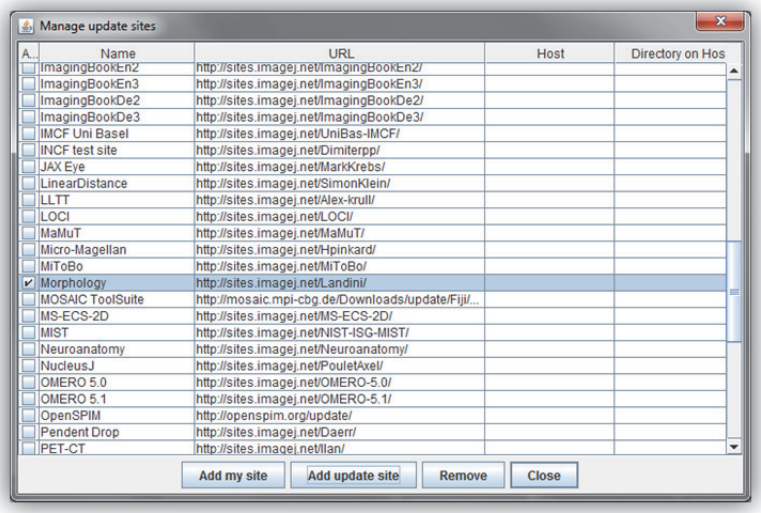
There is a large range of plugins and features that can be added to Fiji. A lot of these can be automatically added and kept up to date using the Update Sites function.

NOTE: The addition of the Morphology update site is required for some of the examples in this manual.

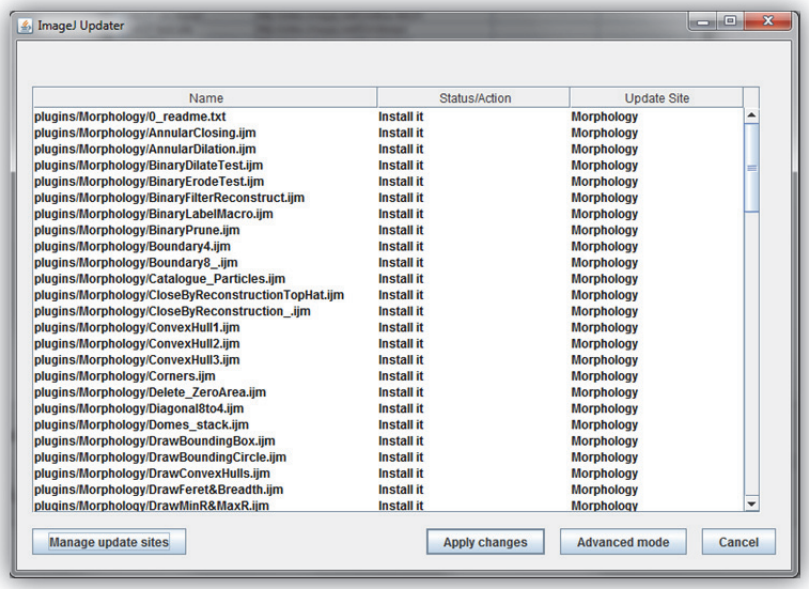
1. In the Update Fiji window from above click the **Manage update sites** button at the bottom left



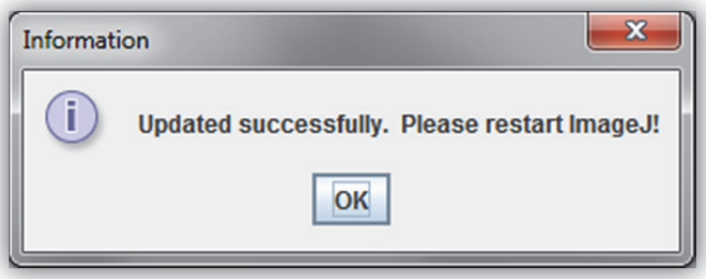
2. Select the update sites you want to add by ticking the relevant boxes. In this example the **Morphology** update site has been added, this is the one that is essential for certain examples in this manual.



3. Once all required sites have been ticked press the close button and any required updates will be added to the list. Press **Apply Changes** to add these new plugins.



4. As before Fiji will need to be restarted to install the updates

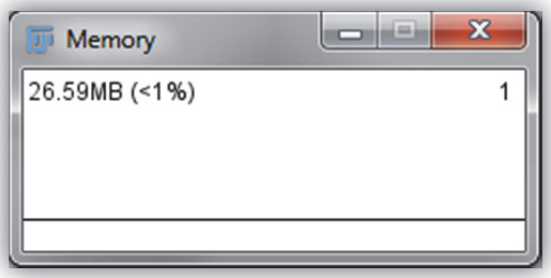


Monitoring and Clearing up Excess Memory Usage

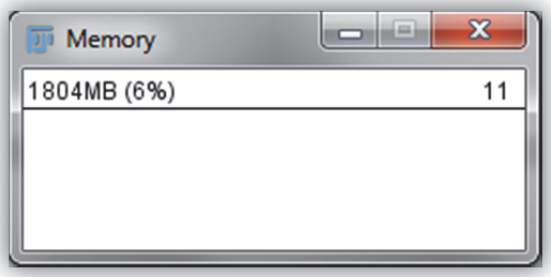
As Fiji is used memory is taken up by Java. Even though you may not have any images open it may be using the bulk of the available resources.

Monitoring Memory

To monitor the currently used memory and the number of opened images go to **Plugins → Tools → Monitor Memory**

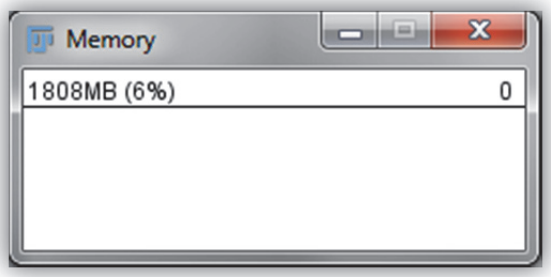


This will open the memory monitor that will show the amount of RAM currently being used (top left number), the percentage of the total available RAM that represents (value in brackets) and the number of open images (top right number).

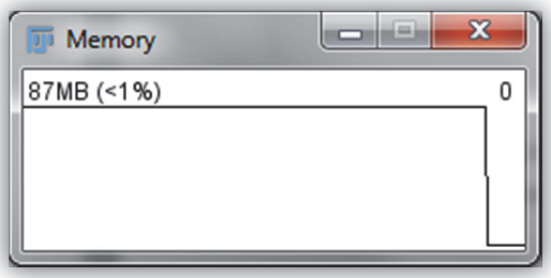


As more images are opened the available memory will fill up and the number of open images number will increase

When all open images are closed the memory is not always automatically given back to the system. In the example below it can be seen that there are no images open yet still 1800MB of memory is being used.

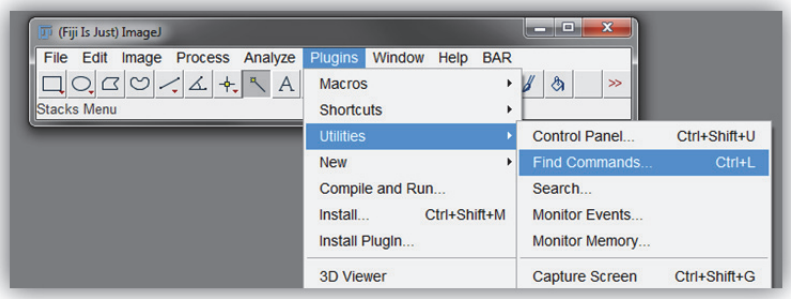


To release the memory back to the system either click on the **Memory** window or click on the bottom of the main Fiji task bar

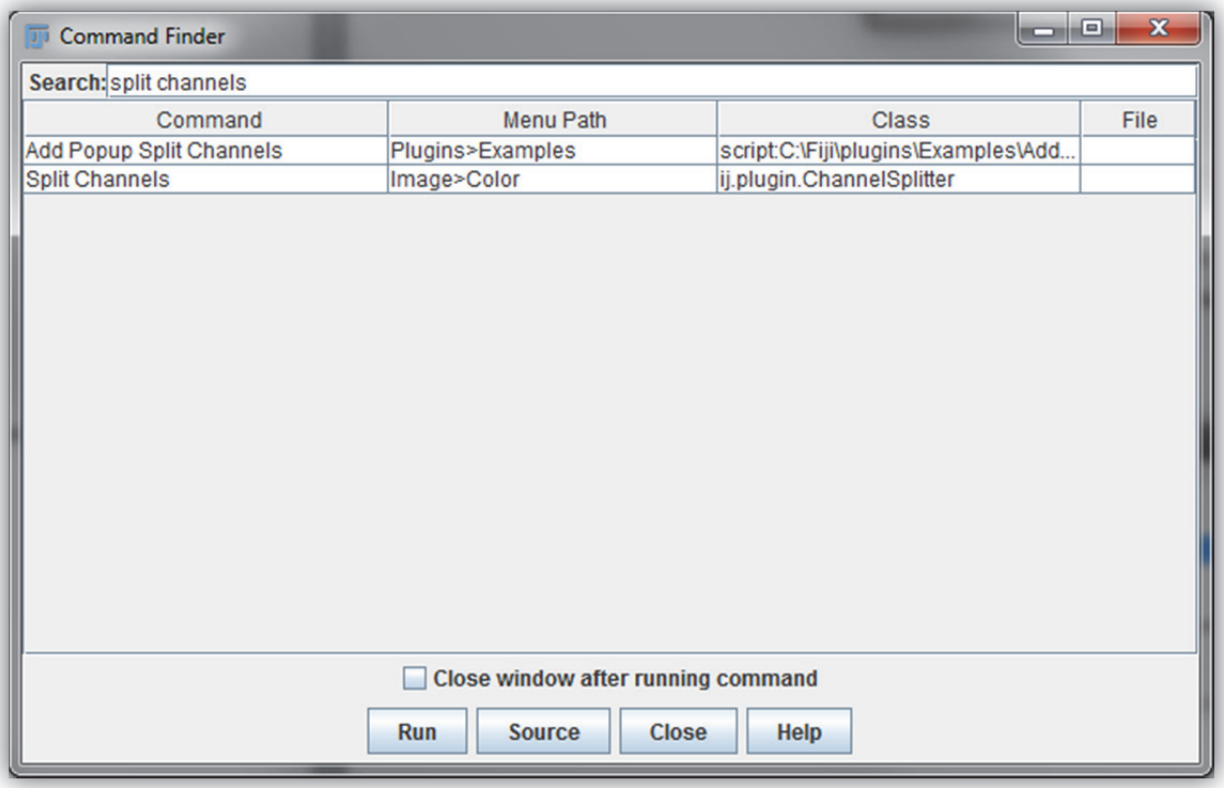


Finding Commands

There are many commands available within Fiji and it can be hard to keep track of where they are all located. To locate a command (assuming you know the name of it) go to **Plugins → Utilites → Find Commands…** or press **Ctrl + L**



You can then search for a command and see where it is located in the menu structure or double click the entry to run the command.



Basic Module

Basic Navigation in Fiji

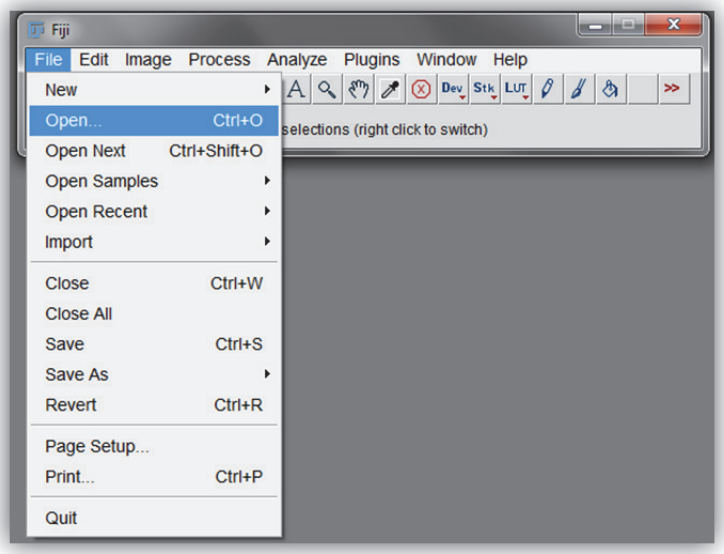
Aim

Fiji contains many different features, menus and plugins. The following is a brief overview of getting around in the Fiji interface.

NOTE: All examples in this chapter use the image **H and E 01.tif** that can be found in the folder **Demo Images\Widefield Images\H and E**

Opening Files

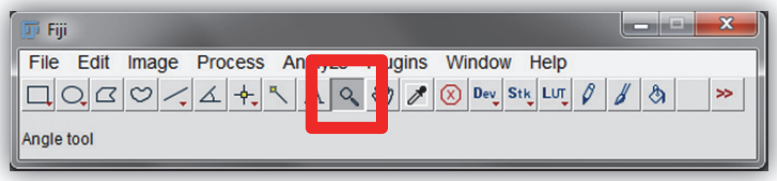
This is pretty straight forward. You can either go to **File → Open** and then navigate to the file you want.



Alternatively you can just drag and drop a file from explorer/finder etc. onto the Fiji window and it will open.

Zooming In and Out

To zoom in or out select the **Zoom Tool**

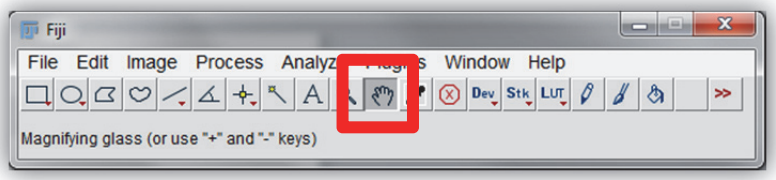


To **Zoom In** left click on the image, to **Zoom Out** right click on the image

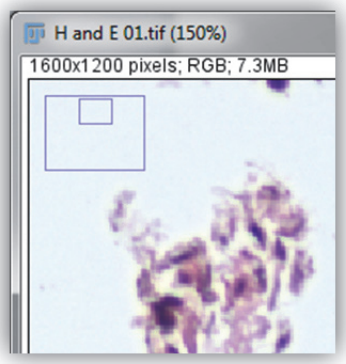
Panning the Image

If you have zoomed in a lot, or your image is a higher resolution than your screen can show, you will only be seeing part of the image on your screen. To move to other parts you need to use the **Hand Tool.**

Select the **Hand Tool** and hold down the left button on your image and move the mouse to pan around. You can also hold down the spacebar to activate the hand tool.



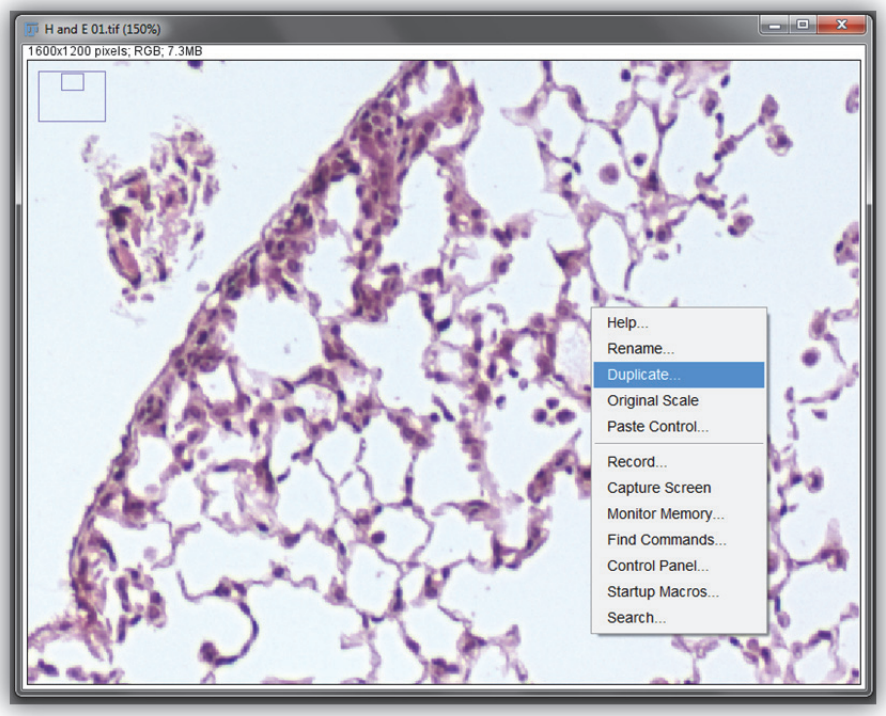
When you are zoomed into an image more than what the screen/window can show an overview icon is shown in the top left corner to let you know where you are in the original image.



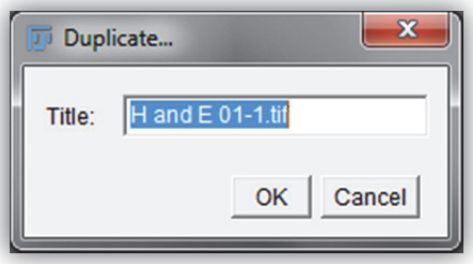
Duplicating an Image

Sometimes it is useful to work on a copy of an image instead of the original as Fiji doesn't always create result images; instead it overwrites the original image. Unless the result is saved the original data stays intact but it can be useful to have to original to refer back to.

To duplicate an image just right click on it and select **Duplicate….** from the menu



In the resulting dialog you can give the image a new name or accept the default which is the original image name with –x on the end of it.

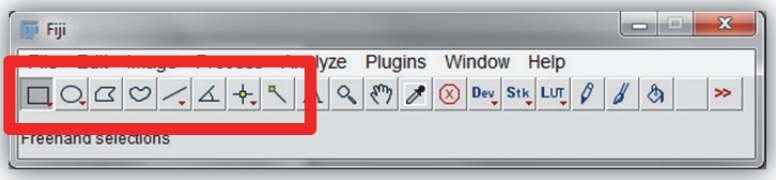


NOTE: If a region of interest (see below) is drawn on the image first only the part of the image in the region will be duplicated. This is useful for cropping images.

Regions of Interest (ROI)

There are many instances where you may need to draw a ROI on an image. They can be used to crop an image (by drawing one and duplicating the image) or to highlight a specific area for further analysis.

To draw a ROI select the desired shape. ROI buttons that have a red triangle on them have several variants of that shape available. To access these right click on the ROI button and select the variant from the menu.

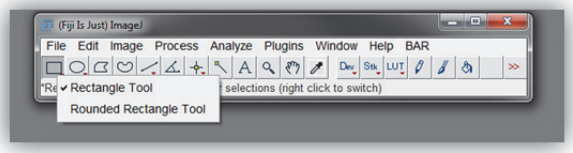


Types of ROIs

There is a range of ROI shapes and functions available. Those that have a red triangle at the bottom right of their button have several variations available. To access these you need to **Right Click** on the button. Some ROI tools such as the **line tool** and **wand tool** have additional functions that can be accessed by **double clicking** on the button.

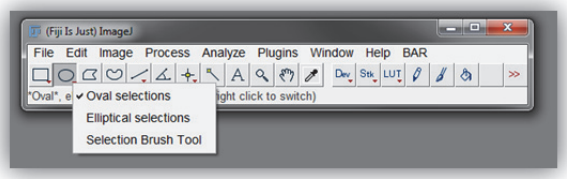
Rectangle Tool

Useful for outlining large areas, extracting or cropping part of an image.

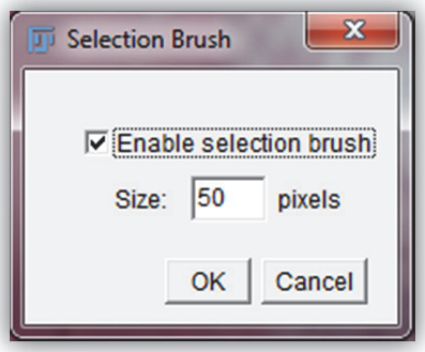


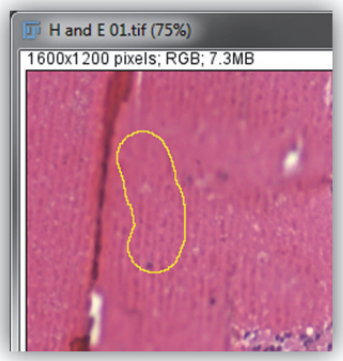
Circle Tool

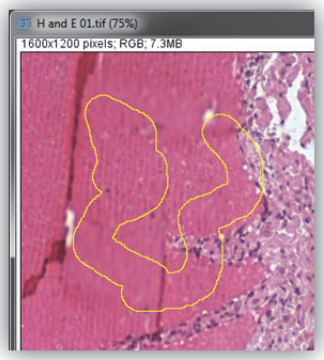
Similar to the rectangle tool but for circular selections



The circle tool also contains an option called **Selection Brush Tool**. This tool has additional settings that can be accessed by double clicking on the button. The number in the **Size:** box indicates the width in pixels of the circle that will be draw. This circle can then be used to “paint” on a selection

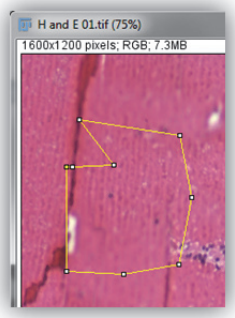






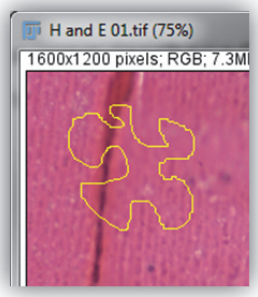
Polygon Tool

The polygon tool allows you to draw irregular shaped ROIs. Just click all the points you need around an area, double click to close the shape off



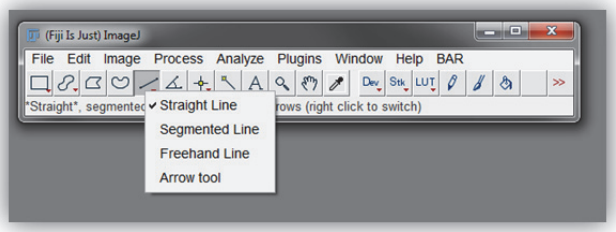
Freehand Tool

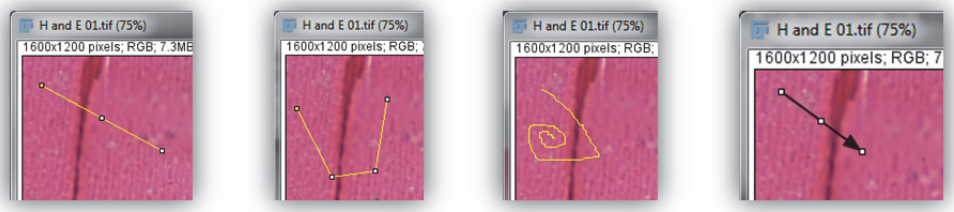
The freehand tool allows you to draw any shape you like. Just hold down the mouse button and draw away.



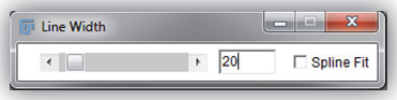
Line Tool

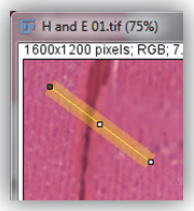
The line tool allows you to draw line selection as either a straight single line, a segmented line (like the polygon tool but not closed off, a free hand line (like the freehand tool but not closed off) or an arrow for annotations.





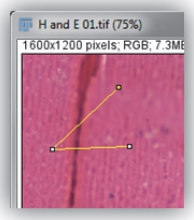
The line tool also has an option when you double click on the button to set the width of the line. This is useful when you are measuring intensities along a line as everything is averaged across the width you set to smooth out noise and inconsistencies.





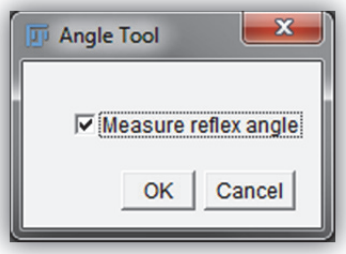
Angle Tool

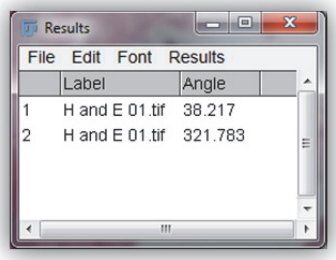
The angle tool allows you to measure the angle between two lines. Clicking the mouse adds the first point of the ray, the second click marks the vertex and the third click marks the end of the second ray.



To measure the angle you need to use the measure function by going to **Analyse → Measure** or pressing **Control + M**

Double clicking on the angle button brings up an option window that allows you to choose to measure the reflex angle instead (the larger of the two angles).

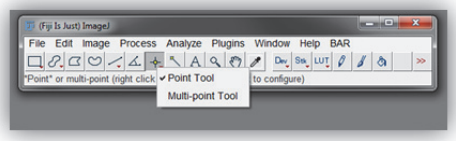


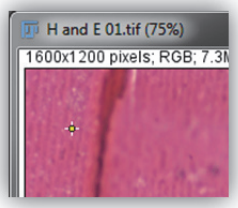


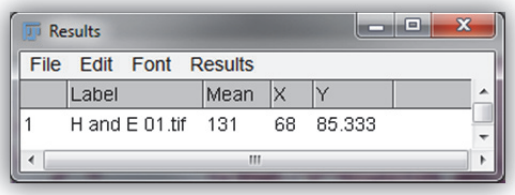
Point Tool

The point tool allows you to mark an individual or multiple points on an image to measure the XY coordinates and the intensity under the point.

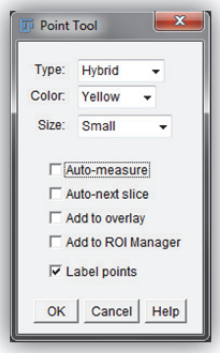
The tool has a right click option to choose between single point and multi points. Once points are marked they can be measured using the **Analyse → Measure** command or the **Ctrl + M** shortcut

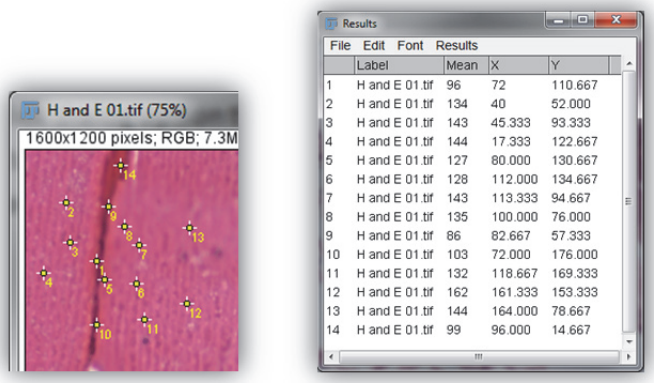




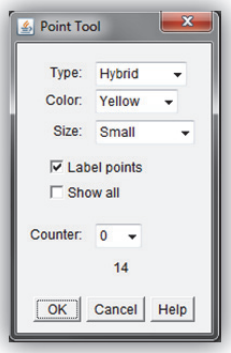


The Single point tool has a double click dialog option that allows you to change the style and colour of the pointer as well as the option to automatically add the results of a click to the result table, move to the next slice, add to overlay and ROI manager.



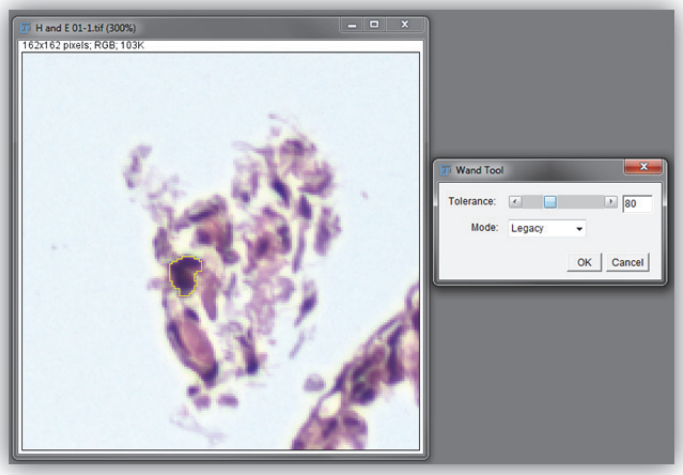


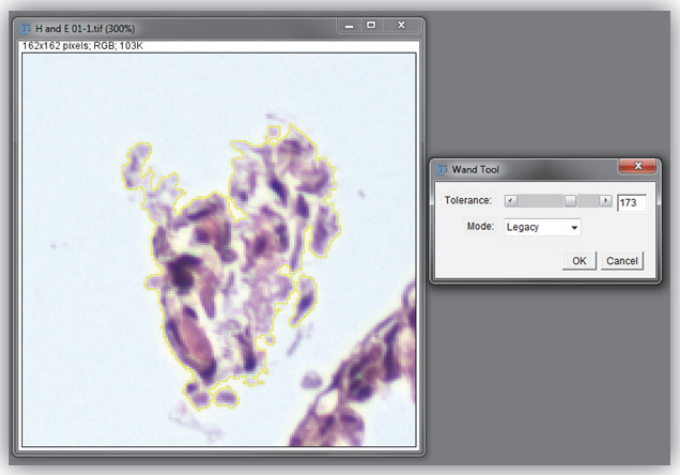
The multipoint tool also has a double click menu that has less options but can be used as a simple click counter if required (instead of logging data out to a table)



The Wand Tool

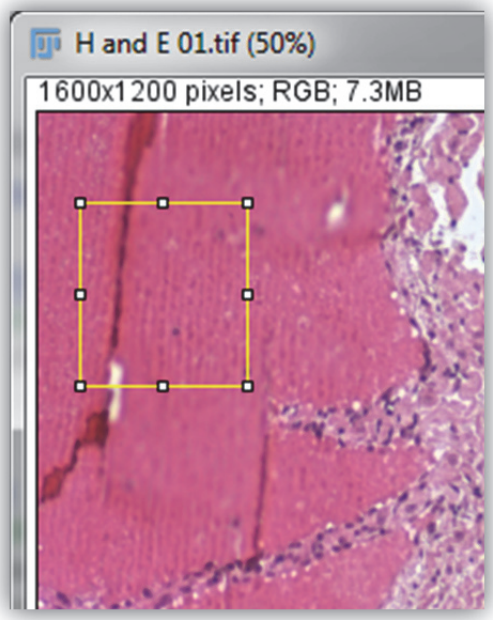
The wand tool allows automatic selection of an area based on a tolerance or threshold value. To use the tool click on an area you wish select. It may not select it correctly initially but its sensitivity can be adjusted by changing the tolerance value in the dialog that can be accessed by double clicking on the want tool button.





Drawing ROIs

To draw the selected ROI hold down the left button on the image and draw as required. If you require more than one ROI hold down the shift key and draw any additional ROIs required.



NOTE: When additional ROIs are added using the shift key they essentially become one big ROI. This means that anything within them will be analysed together. If individual ROIs are needed you need to use the ROI manager to add multiple, separate ROIs.

Removing ROIs

To remove the most recent ROI you have drawn just click away from it using any of the ROI tools or use the shortcut key **Ctrl + Shift + A**

Copying/Moving ROIs to Another Image

There are situations where you need to draw and ROI on one image/channel and then use it to measure something in a separate image/channel. To do this draw the ROI then select the image you with to copy/move it to and press **Ctrl + Shift + E**

Moving ROIs

To move an ROI have any of the ROI tools selected and click and hold inside the ROI

Resizing ROIs

To resize an ROI have an ROI tool selected and click and hold on any of the anchor point around the ROI to adjust its size.

Holding down different keys while resizing will have different effects on the resizing.

Shift – Will resize the ROI symmetrically. If the ROI was a rectangle or an oval it will become a square or circle with an initial width the same as the width of the ROI.

Ctrl – Free resize around the centre of the ROI

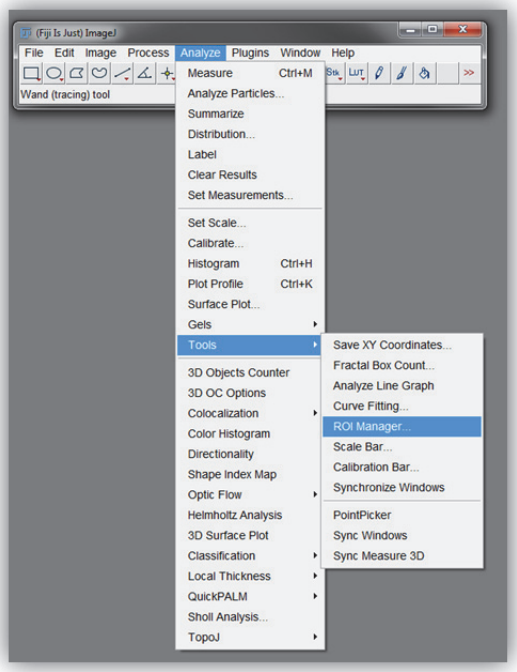
Alt – Resize the ROI while keeping the aspect ratio the same

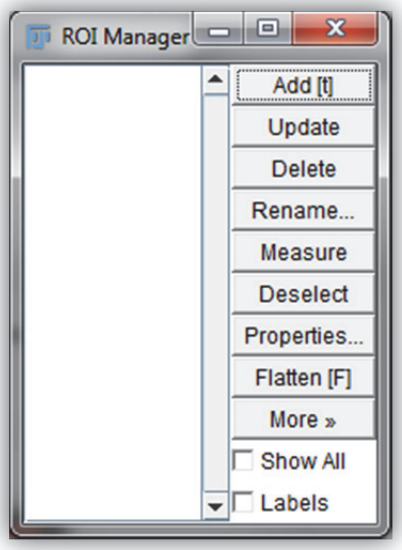
NOTE – Combinations of keys can be used too. For example hold down **Alt** and **Ctrl** to resize an ROI with the same aspect ratio around its centre.

Using the ROI Manager to add Multiple ROIs

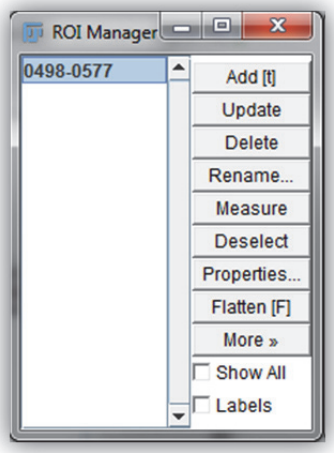
To add multiple ROIs to an image they need to be individually added (either manually or by a analysis module) to the ROI manager

1. Go to **Analyze → Tools → ROI Manager** to open the ROI manager

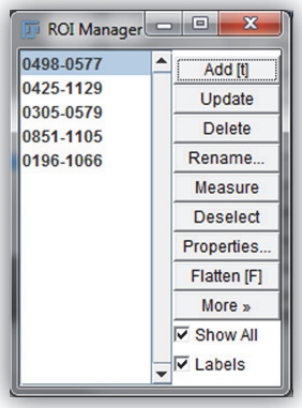




2. Draw an ROI and press the **Add** button on the **ROI Manager.** The ROI will be listed in the ROI manger. The label represents the Y and X coordinates of the centre of the ROI



3. Follow the above step to add additional ROIs. If you want to see the previously added ROIs you may need to click the **Show All** box.



Adjusting Brightness and Contrast

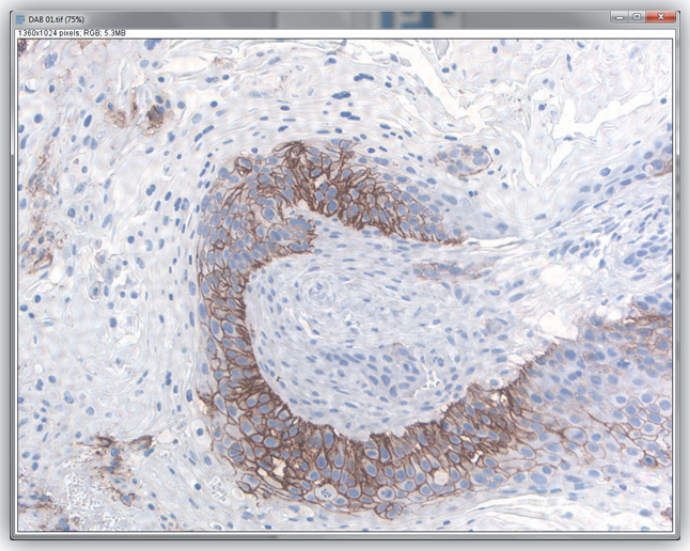
Aim

It is sometimes essential to alter the brightness and contrast of an image to make it better for publishing or highlighting an area of interest. Adjusting the balance of an image needs to be done carefully so as not to remove or add data that was not in the original image. To this end it is usually safe to adjust linear parameters such as brightness and contrast (histogram levels) as long as under and over saturation is avoided. Adjusting non-linear parameters such as gamma and curves is generally discouraged as it will change the relationship of the data in a given image. For example adjusting gamma can result in the dark values of an image not changing very much but the bright values being greatly increased.

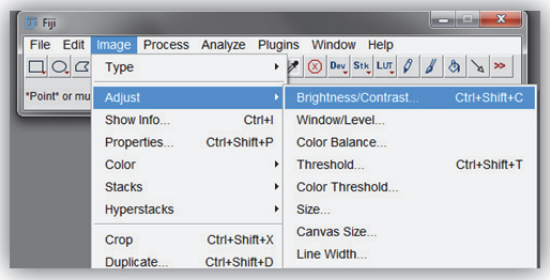
This technical note uses the **DAB 01.tif** image found in **Demo Images\Widefield Images\DAB** and the **FITC.tif** image found in **Demo Images\Widefield Images\DIC and Fluorescence**

Adjusting Brightness and Contrast – Colour Images

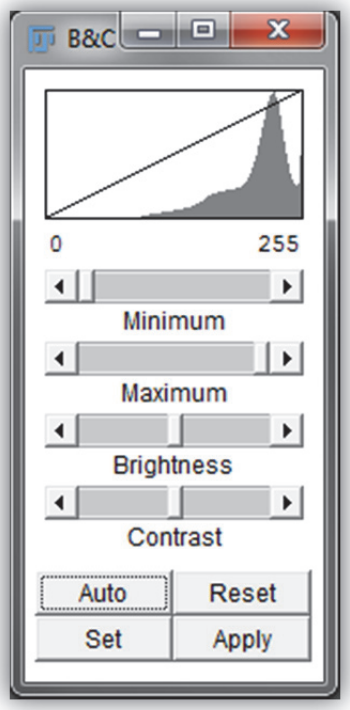
1. Open **DAB 01.tif** from **Demo Images\Widefield Images\DAB**



2. Go to **Image → Adjust → Brightness/Contrast**



3. In the resulting dialog window there is a histogram of the image data at the top with the range of intensities in the image below (in this example 0-255, 8bit).



Several options exist for adjusting the brightness and contrast (if at any point you do something weird just press the **Reset** button and everything will return to the original:

a) You can move the **Minimum** and **Maximum** sliders. This should only be done to the limit of the data and not beyond it. So in this example the maximum value should not be changed and the minimum value can be adjusted to about 80 without affecting the output.
b) You can press the **Auto** button and a histogram stretch will be calculated. The result of this can sometimes leave you with a very strange looking image.
c) You can press the **Set** button and enter in your own values directly

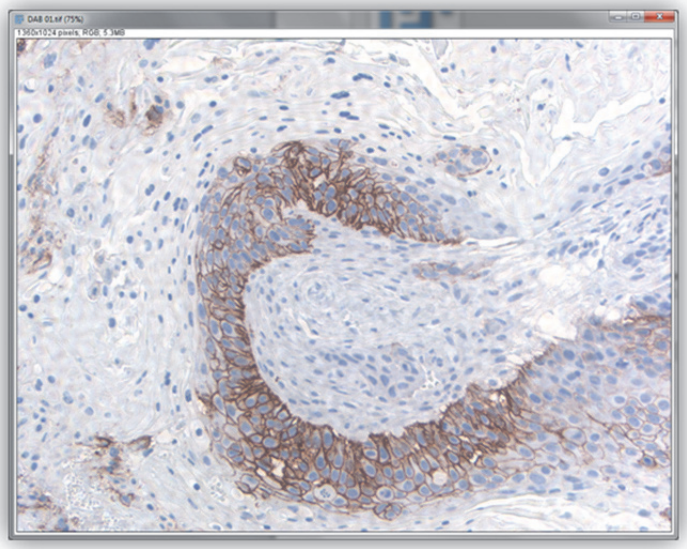
d) You can move the **Brightness** and **Contrast** sliders. The same rules apply to theses as to adjusting the **Maximum** and **Minimum** sliders.

4. Once you are happy with your settings press the **Apply** button. **NOTE:** Once the **Apply** button is pressed the original values in the image are altered to the new ones and the original data is lost. For this reason it is best to save the result as a different file so as not to lose the original data.

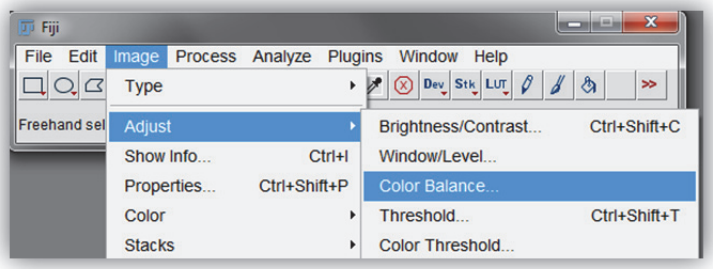
Adjusting the Colour Balance – Colour Images

It is possible to adjust the colour balance of an image to remove subtle tints that may occur due to staining or imaging artefacts. This should be done with caution as it can result in misrepresentation of data. Images that have been colour balance adjusted should not be used for further analysis.

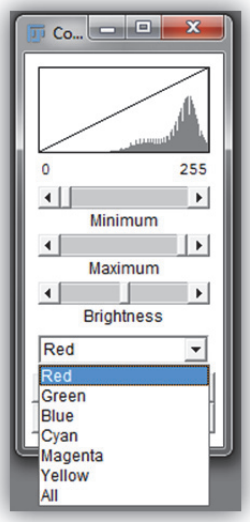
1. Open **DAB 01.tif** from **Demo Images\Widefield Images\DAB**



2. Go to **Image → Adjust → Colour Balance**



3. The resulting window contains a histogram like before. The default view is to have the red channel selected. You can select different colour channels from the pull down menu and see the resulting histograms for each



The minimum, maximum and brightness values can be adjust for each colour channel to change the balance in the image.

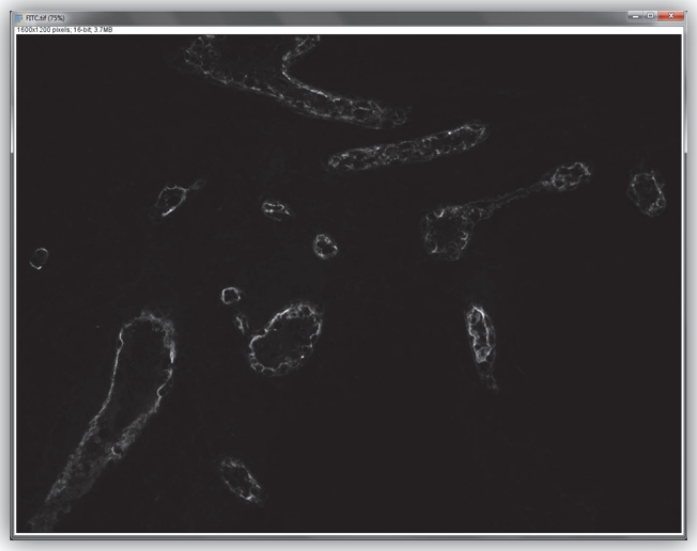
To lock in a change to a colour channel you need to press **Apply** before selecting another colour channel.

NOTE: Selecting **All** from the list will let you change all colour channels equally. This is what is being done in the brightness and contrast adjustment from before.

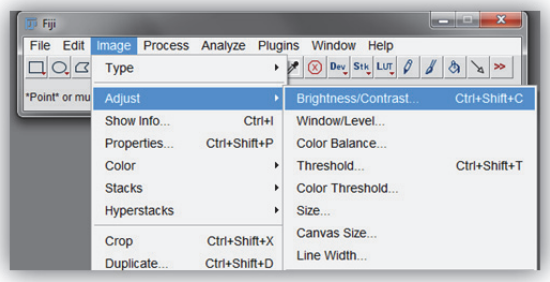
Adjusting the Brightness and Contrast – Monochrome Images

Adjusting monochrome images follows the same principals as shown above for adjusting colour images. The one major difference is the dynamic range of the images tends to be larger as they are captured on higher bit depth cameras.

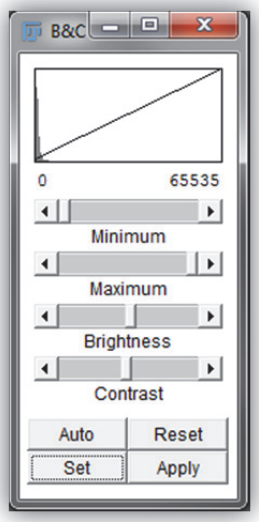
1. Open **FITC.tif** from **Demo Images\Widefield Images\DIC and Fluorescence**



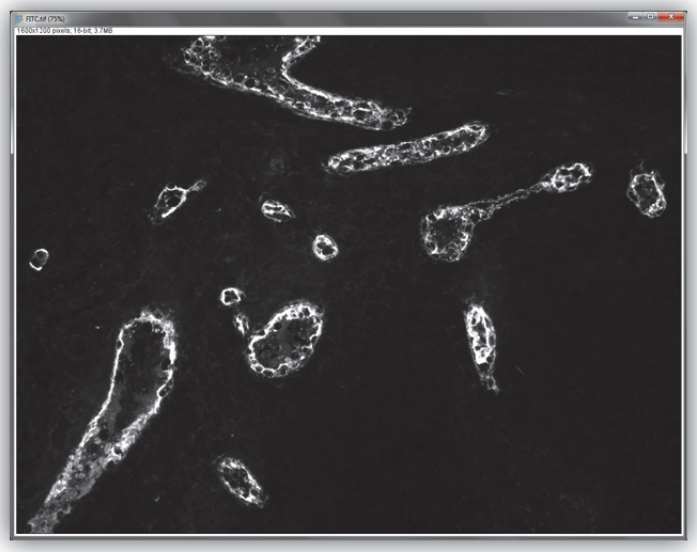
2. Go to **Image → Adjust → Brightness/Contrast**



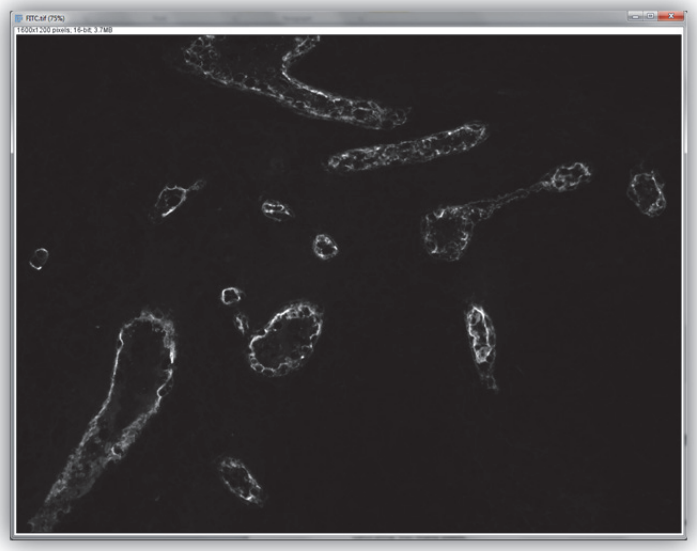
3. Notice this time the range on the bottom of the histogram is from 0 – 65535 because this is a 16-bit image. You will also notice that the bulk of the image data is in the bottom part of the histogram.



4. Very little adjustment can be made to the Minimum value before data is lost but there is some range in the Maximum for adjustment. If you press the auto button you will notice the result is too saturated in the higher end.



5. Setting the maximum value to around 45000 will give a well-balanced image without over-saturating too many pixels. As before the **Apply** button needs to be pressed before the values are applied (permanently) to the image.



Calibrating and Adding a Scale Bar

Aim

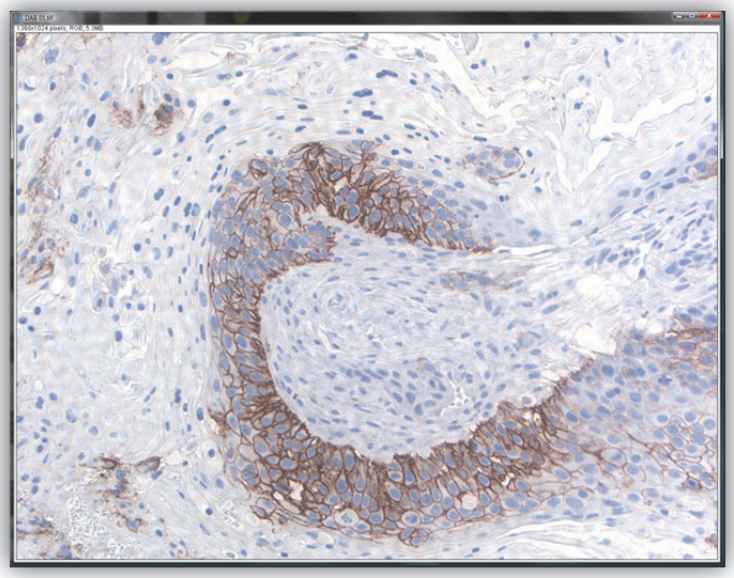
For proper presentation of images (either in a presentation or a publication) it is vitally important that they contain a scale bar. This allows the reader/viewer to see the scale of the image being presented and make their own conclusions about the data. Calibrating an image is also important if you want to get meaningful area/distance etc. measurements out of the image in any later analysis.

This technical note uses the **DAB 01.tif** image that can be found in **Demo Images\Widefield Images\DAB**

NOTE: Some images may already have calibration information embedded in them from the capture system. This should always be checked as Fiji does not always interpret this correctly.

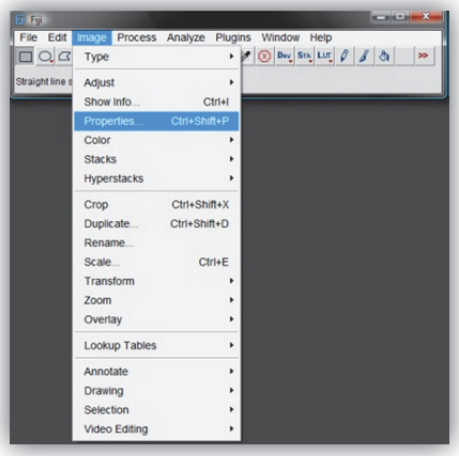
Calibrating an Image

1. Open the file **DAB 01.tif**

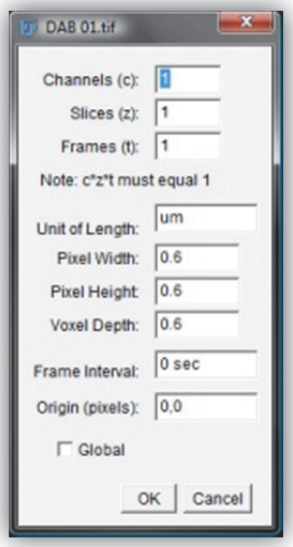


2. Check to see if the image is already calibrated. In the top left corner of the image window will be details about the image dimensions, bit depth and file size. In this example the image is 1360x1024 pixels, is in 8 bit RGB format and is 5.3MB big. If the dimensions are in pixels then the image is not calibrated. Calibrated images will have microns, um, mm, inches etc instead of pixels.

3. To calibrate the image go to **Image → Properties**



4. Enter the size of the unit of length and pixel width (in this example um and 0.6) and press **OK.**



The image is now calibrated; the information in the top left corner of the image has changed to show the size of the image in um.

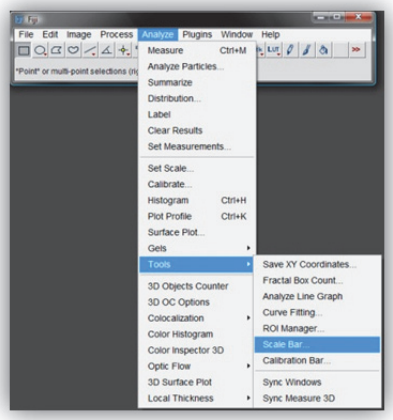
NOTE: Ticking the global box will apply the properties to all open images.

Adding a Scale Bar

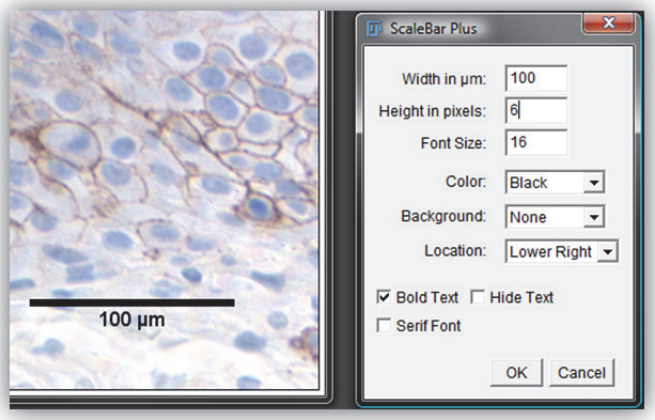
All images need a scale bar on them for publication. This gives viewers of the image and idea of the size of the objects (cells, organelles, blobs etc.) in the image.

You can add a scale bar to an uncalibrated image but it will provide no useful information as it will be in pixels and not a unit of measurement.

1. Go to **Analyse → Tools → Scale Bar**



2. Enter the detail for the scale bar in the dialog box that appears. For this example a **100um** scale bar that is **6 pixels** thick with bold text in black is created.



3. Save your file. It is usually a good idea to save your file as another copy so you always have a version of the image without a scale bar embedded in it.

NOTE: If you don't like where the scale bar is by default you can first draw a line ROI on the image and then select **At Selection** from the location list.

Fluorescent Images

Aim

Immunofluorescence experiments are almost always captured on highly sensitive monochrome cameras in 12, 14 or 16bit. This means that each fluorophore is captured as a different image with no associated colour and usually cannot be opened in the standard images viewers built into Windows, OS X or Linux. Some default viewers will display 16bit images if they are using the full range of intensities.

To make the data more presentable it is usually a good idea to add colour to each channel (usually in the colour of the fluorophores emission) and to create composite images of 2 or more of the channels combined.

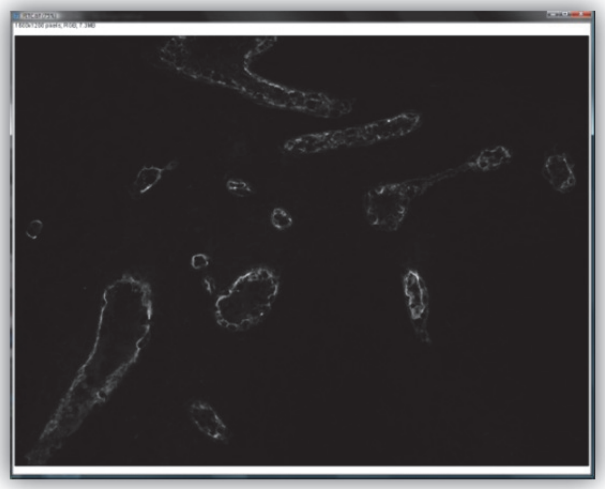
When merging fluorescent images you need to be aware that they could appear very strange to someone with colour blindness. Fiji has the ability to simulate what your image will look like to someone with colour blindness.

This method uses the images found in **Demo Images\Widefield Images\4 Chanel Fluorescence, Demo Images\Widefield Images\DIC and Fluorescent** folders and **Demo Images\Colour Blindness** folders

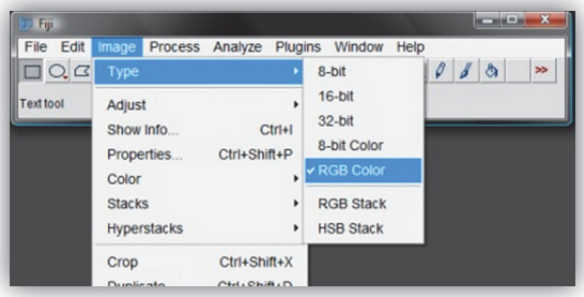
Changing Bit Depth – 24bit RGB

Most standard image viewers and presentation software will only show images that are in a 24bit RGB format (3 x 8bit colour channels). Changing a single 12, 14 or 16bit image to 24bit RGB is very simple

1. Open the **FITC.tif** image found in **Demo Images\Widefield Images\DIC and Fluorescent folders**. This is a single channel 16bit monochrome image. Earlier versions of windows will open it as a black square. It is best to convert it to a 24bit RGB image for any presentation or publication situations. **DO NOT** alter bit and colour depth if you plan to do any analysis on the image, always save the results of the conversion as a separate file to preserve your original image.



2. Go to **Image → Type → RGB Colour**



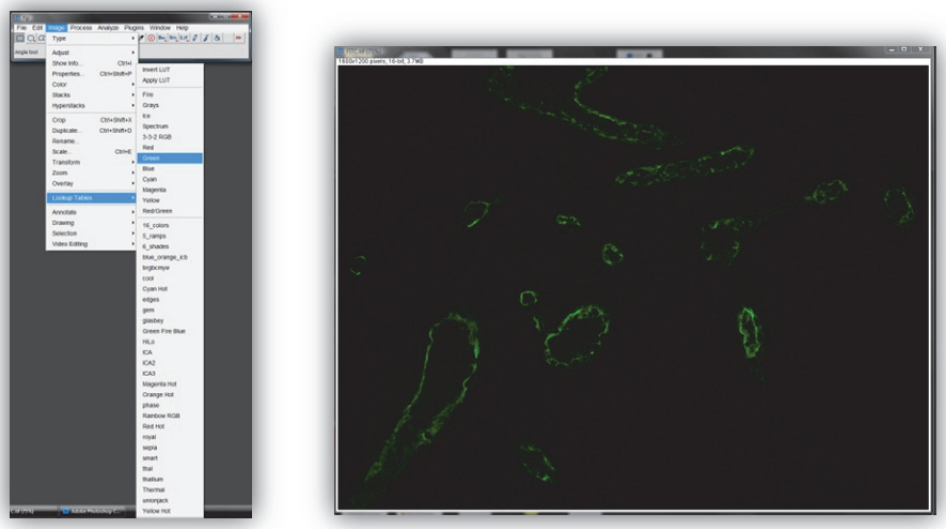
3. The image, while looking the same, is now a 24bit RGB format that is compatible with pretty much any software out there. Save the file as a copy to preserve your original data.

Adding Colour – Using Lookup Tables

The colour of a monochrome image can be changed easily by changing the lookup table (LUT) applied to it. A monochrome image has a grey look up table applied to it, meaning that each intensity is represented by a different shade of grey. This can be changed to be different shades of red, green, blue etc. LUTs that contain more than one colour can also be useful for showing things like under and over saturation or to better represent signal distribution.

Single Colour LUTs

1. Open the **FITC.tif** image from before and go **to Image → Lookup Tables**. You will see a large list of possible LUTs to apply. Apply the **Green** LUT. Alternatively you can press the **LUT** button on the toolbar.

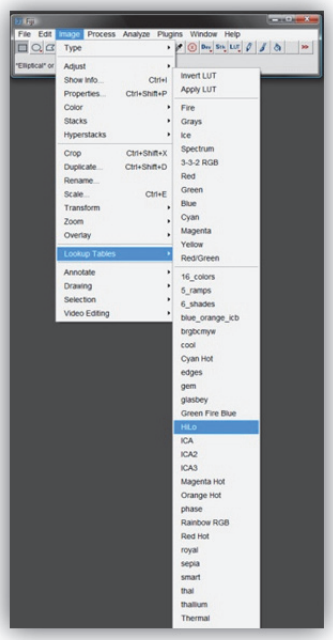


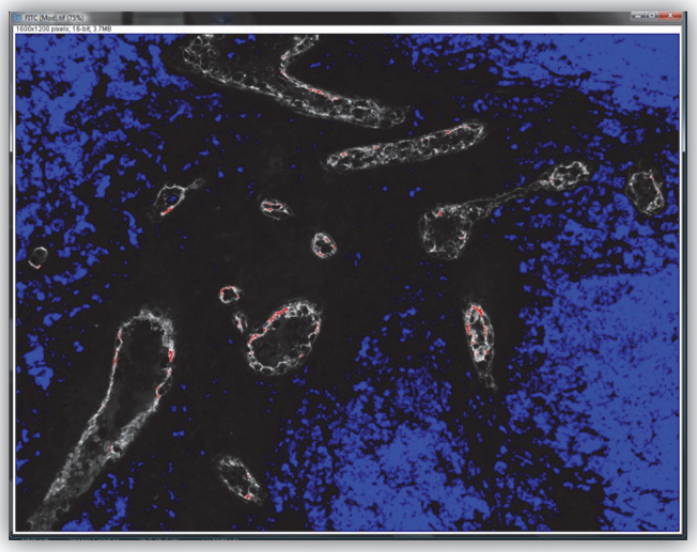
2. To save a version of the image for presentation etc. follow the steps above to convert the image to a 24bit RGB.

Multicolour LUTs

Saturation Indicator

1. Open the **FITC (Mod).tif** image. **Go to Image → Lookup Tables** and apply the **HiLo** LUT

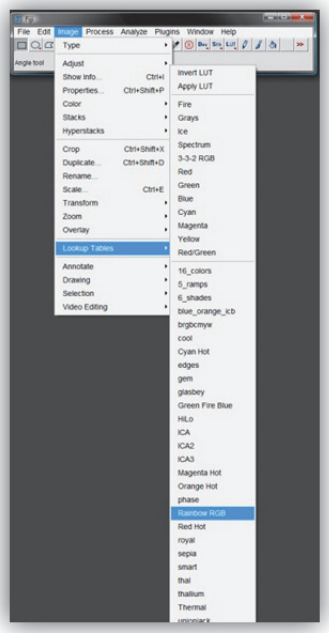


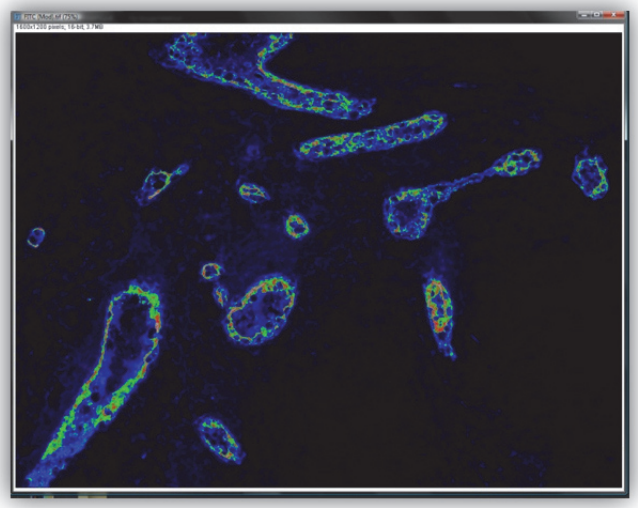


2. This LUT gives a monochrome image with some pixels in red and some pixels in blue. The red and blue pixels represent intensity values at the two extremes of the range. Red pixels are fully saturated (i.e. pure white) and blue pixels are fully under saturated (i.e. totally black). Both these types of pixels essentially contain no information as you cannot be sure how far above or below saturation they are, they are essentially a cartoon.

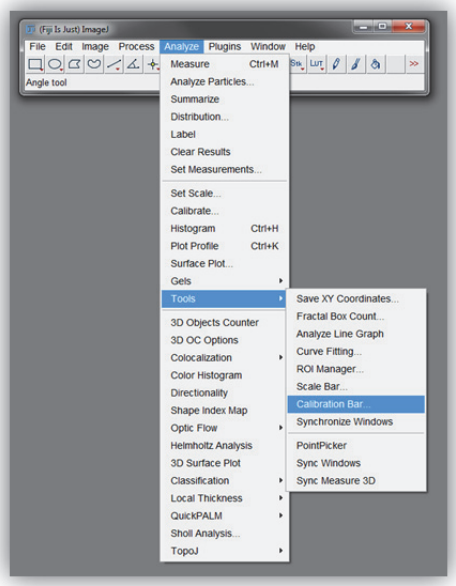
Intensity Distribution

1. Open the **FITC (Mod).tif** image. **Go to Image → Lookup Tables** and apply the **Rainbow RGB** LUT

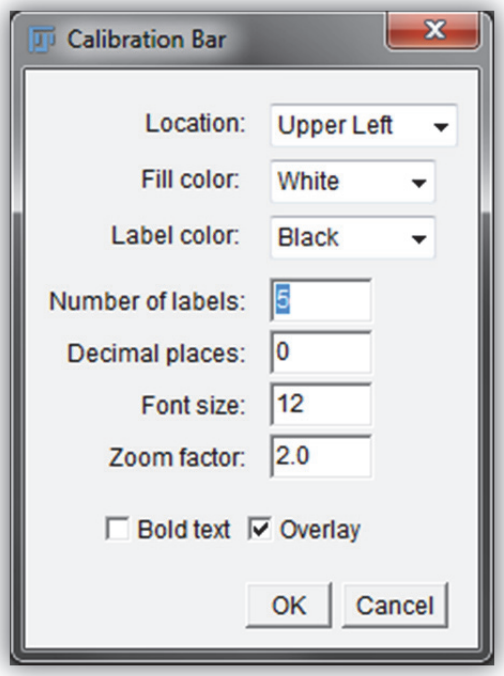


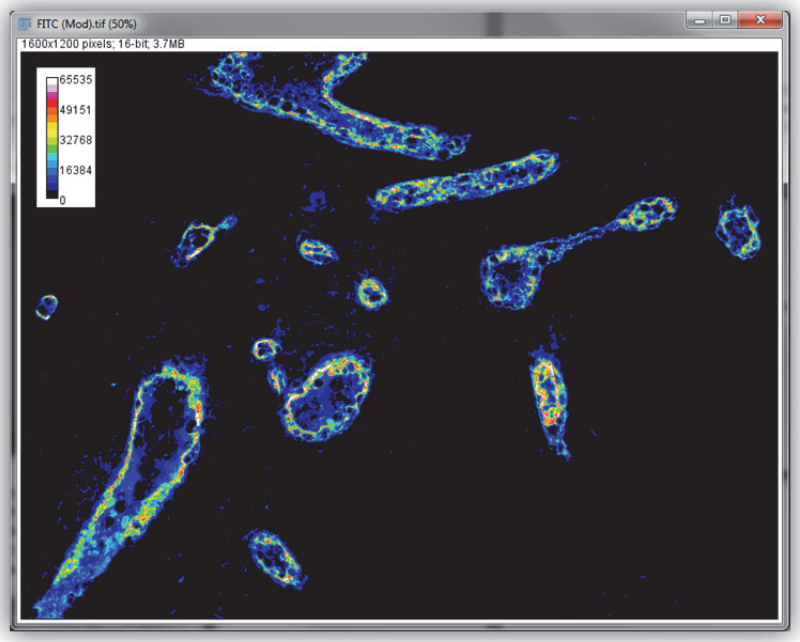


2. This LUT colours intensity in shades of red, blue or green. The highest 33% of intensities are reds, the middle 33% are greens and the bottom 33% are blues. This gives the viewer the ability to easily see the range of intensities present in the image.
3. A calibration bar of what intensities the colours cover can be added to the image by going to **Image → Tools → Calibration Bar**



4. In the **Calibration Bar** dialog you can configure the position and configuration of the calibration bar. The **Overlay** tick box allows you to create the bar as an overlay of the original image, instead of burning it into the image permanently





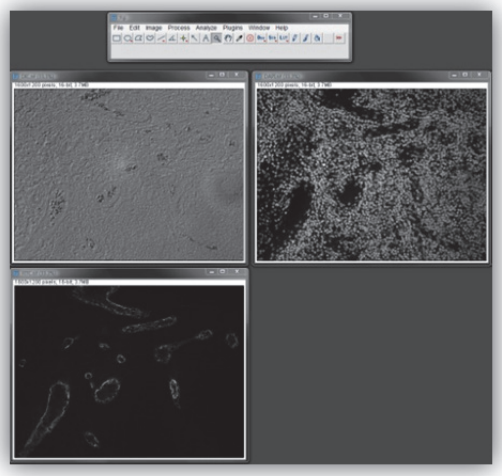
Merging Images

Merging or overlaying images of several different fluorophores can provide a very pretty picture that gives the viewer information about relative location, intensity etc. Depending on how many channels, and what colour you want them, are being merged the method is slightly different.

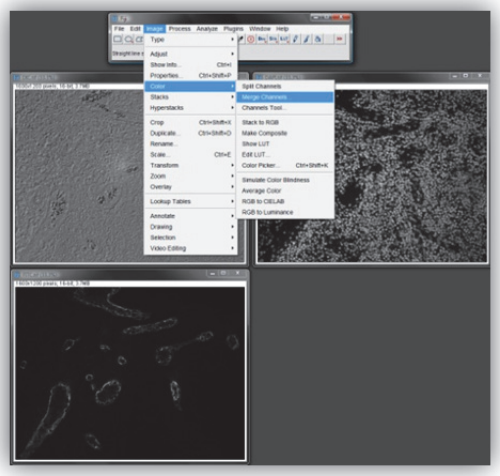
Merging Up To 7 Channels and DIC/Phase/Brightfield – Standard Colours

Fiji/ImageJ has a tool for easily merging up to six channels plus one brightfield/grey channel. This tool will however only allow you to colour your image channels red, green, blue, yellow, cyan and magenta (plus grey for the brightfield). To be able to create a merged image with "non-standard" colours (i.e. not red, green and blue) see the second method below.

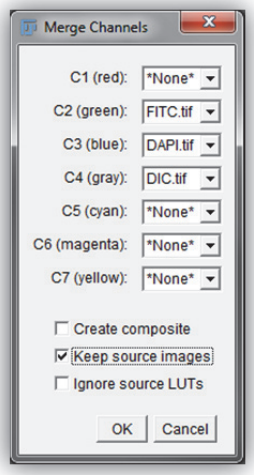
1. Open the **DAPI, FITC** and **DIC** files from the **Demo Images\Widefield Images\DIC and Fluorescence** folder.



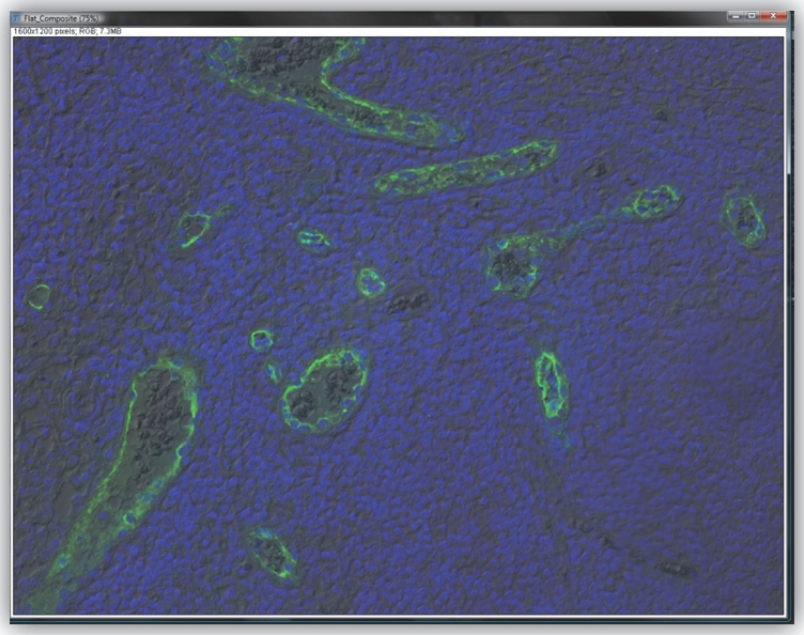
2. Go to **Image → Colour → Merge Channels**



3. Assign an image to each channel. In this example the red channel is set to none as there is no red image, **FITC** is assigned the green channel, **DAPI** the blue and **DIC** the grey channel. Un-tick the create composite box and tick the keep source images box. Press **OK**



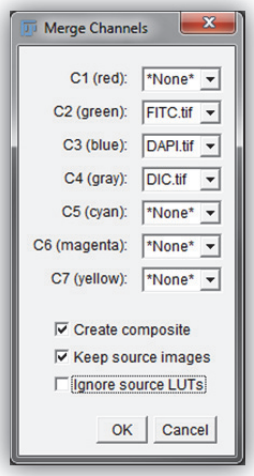
4. The resulting image will be a standard 24bit RGB colour image showing the merge of the channels.



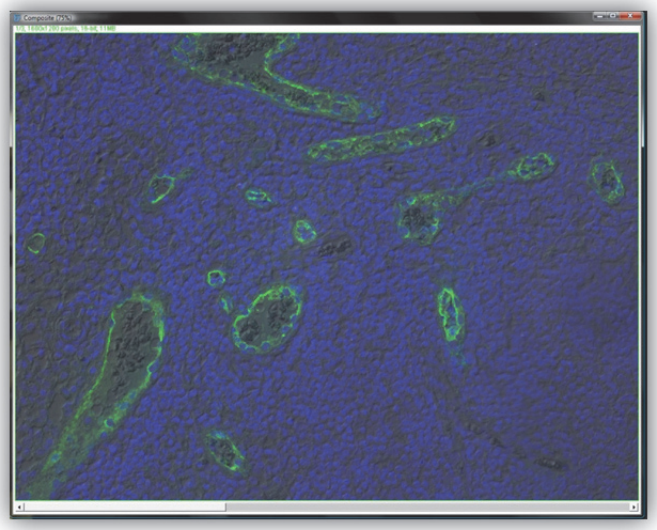
Merging Up To 7 Channels and DIC/Phase/Brightfield – Non-standard Colours

There will be some instances where you may want to merge the images as above but do not want to have them coloured the standard red, green, blue, yellow, magenta, cyan and grey combination (for example if you are presenting them to a person who is colour blind or you want one of your channels to have a multi-colour LUT applied).

1. Follow steps 1-3 in the method above, but this time make sure the **create composite** box is ticked.

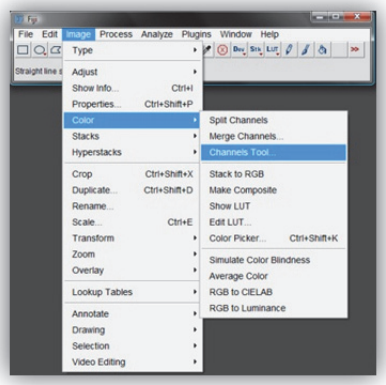


2. The resulting image will look the same on the screen as the previous merge image did but there are some differences.

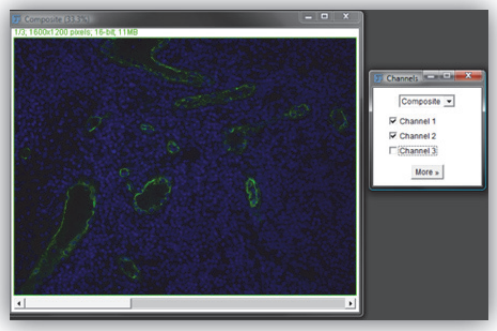


3. The image is not a 24 bit RGB image as before, it is now 3 16 bit images on top of each other. There is a slider at the bottom that lets you select the active channel. You tell which channel is selected by the colour of the border and text at the top right of the image.

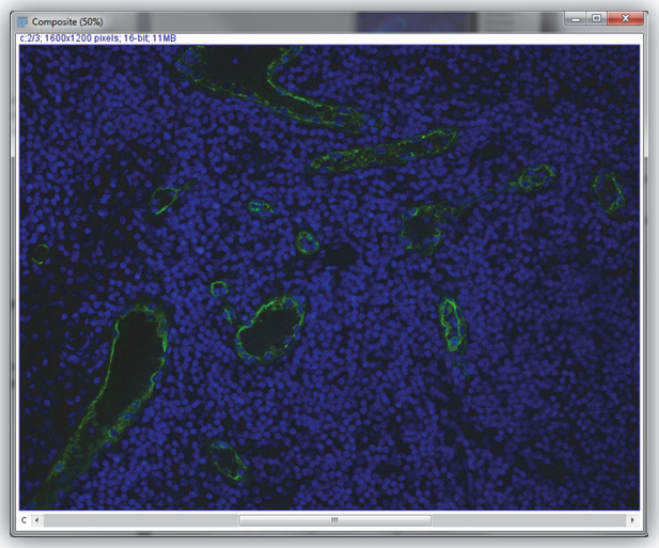
4. To change the colour of the one of the channels go to **Image → Colour → Channels Tool**



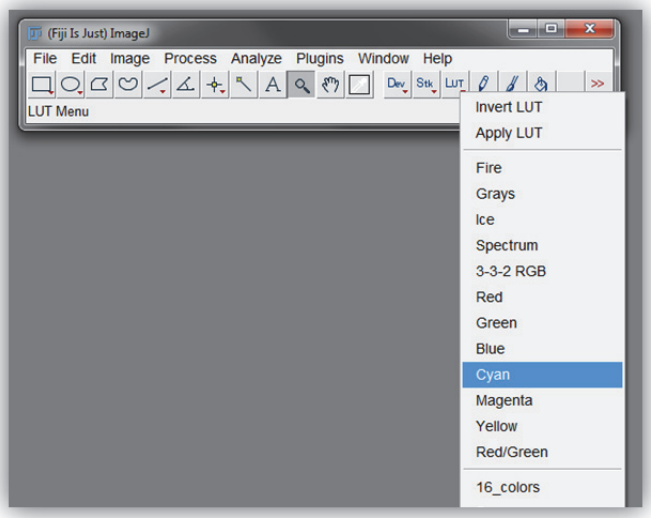
5. The resulting dialog box has several options. In its default state it will say composite in the box at the top and have all the channels ticked. Unticking the boxes next to each channel will turn that channel off. The example below shows the DIC channel (Channel 3) disabled.



6. To change to colour of a given channel use the slider at the bottom of the image to select the desired channel (in this example channel 2 – blue).



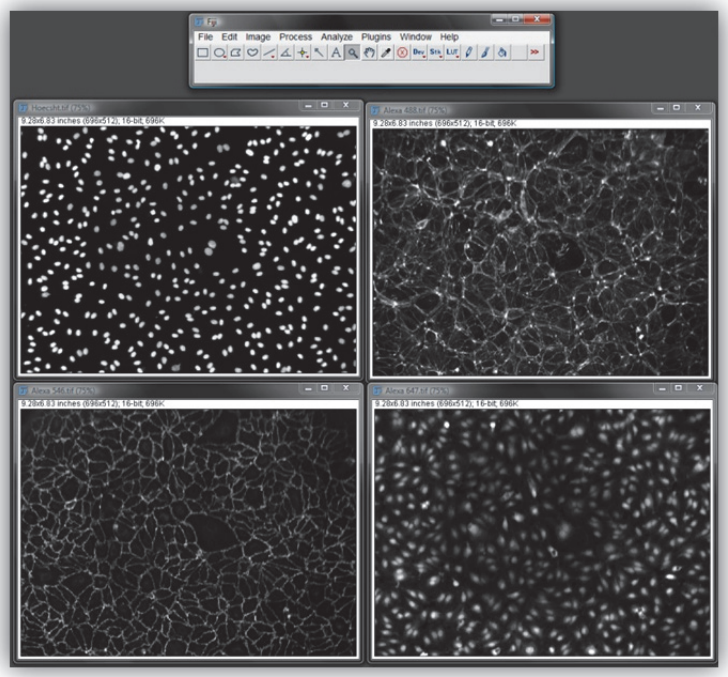
7. Click the **LUT** button on the main Fiji window and select the colour you would like to apply (Cyan and in this example)



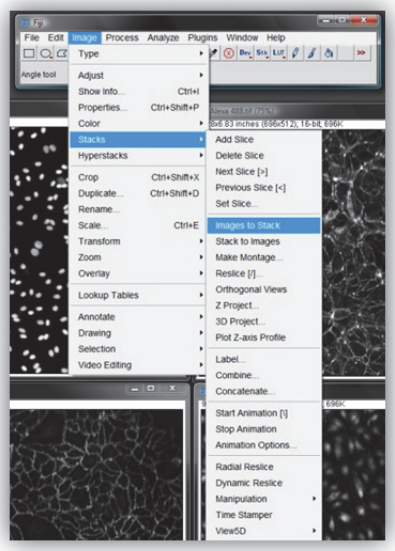
8. Once you are happy with the image yo can convert it to RGB colour by going to **Image → Type → RGB Colour**

Merging 8 or more colour channels

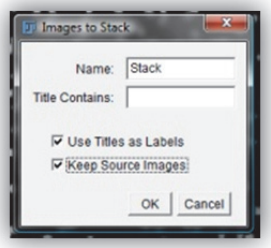
To create a merge of 8 or more colours a different method has to be used. The images that you want to colour need to be added into a stackOpen the 4 images in the Demo Images\Widefield Images\4 Channel Fluorescence folder. For this example we will only use 4 images, but this method can be used for an essentially infinite number of images



1. Go to Image → Stack → Images to Stack

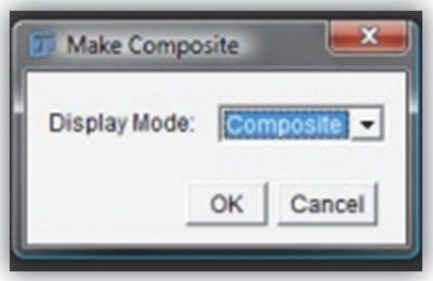


2. In the resulting dialog box make sure both boxes are ticked. You can give the stack a name, or just leave it as the default “Stack”. The result will be a stack of 4 images, one plane for each channel.

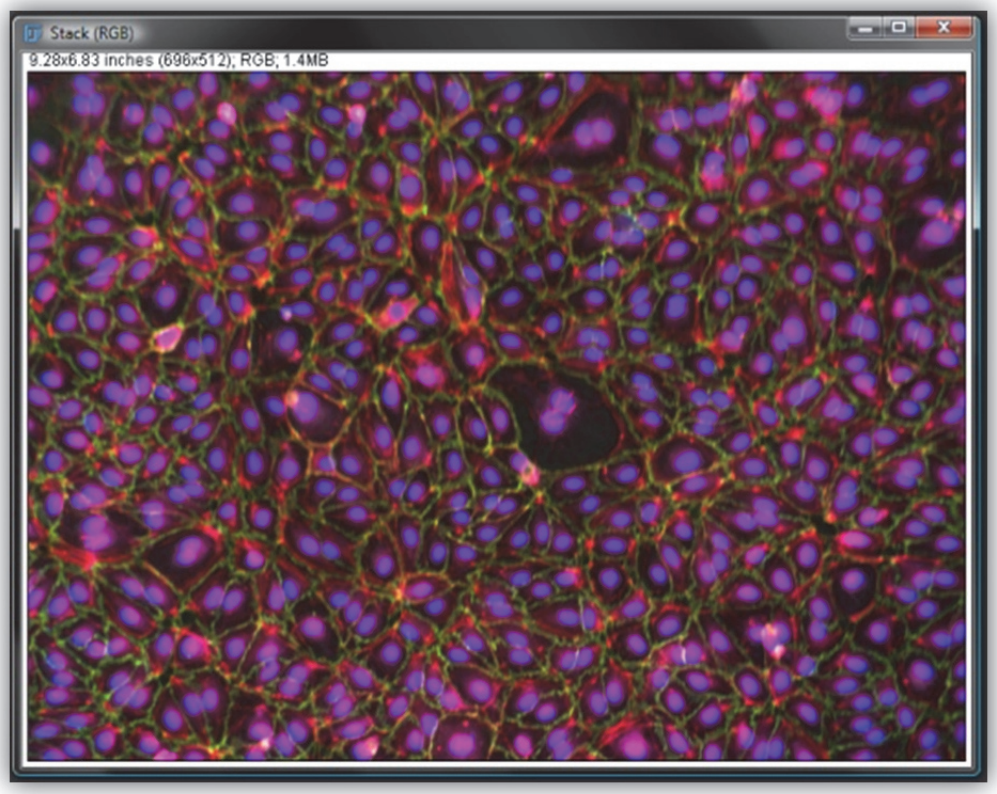


NOTE: The images will be added into the stack in alphabetical order based on their names (e.g Alexa 488 first, Hoechst last)

3. To be able to change to colour of each plane individually the stack needs to be converted to a composite image. Go to Image → Colour → Make Composite. Leave the display mode as composite and press OK in the resulting dialog box.



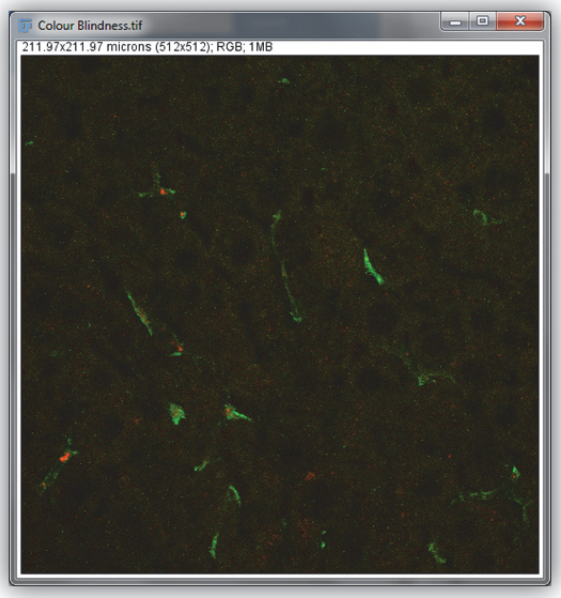
4. You now have a composite stack like before that you can assign new LUTs to



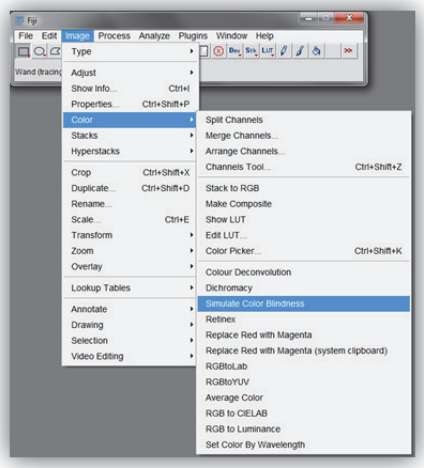
Simulating Colour Blindness

It is possible in Fiji to show what a merged image will look like to someone with varying sorts of colour blindness. The current plugin allows you to simulate Protanopia (no red), Deutanopia (no green) and Tritanopia (no blue). More advanced plugins are available that allow simulation of more subtle forms.

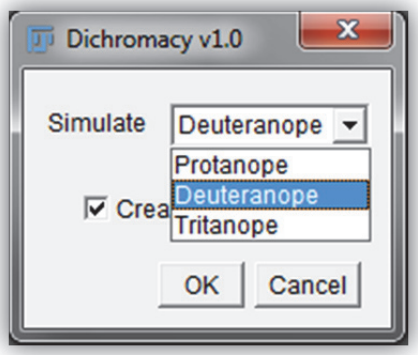
1. Open **Colour Blindness.tif** from the **Demo Images\Colour Blindness** folder. This image contains red particles within green cells. To some colour blind people this would not be distinguishable.



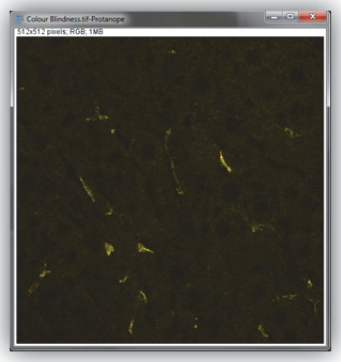
2. Go to **Image → Colour → Simulate Colour Blindness**

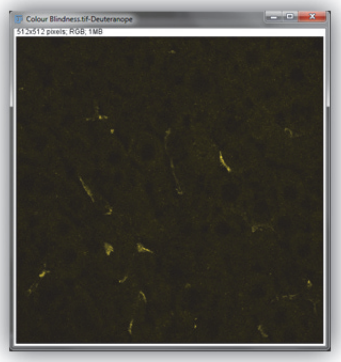


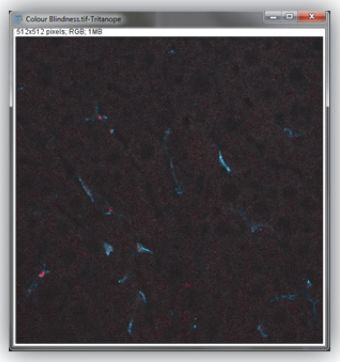
3. Select the type of colour blindness you wish to simulate from the list and press **OK.**



4. The results will show what your image may look like to someone with that type of colour blindness.







Opening Advanced Data Sets

Aim

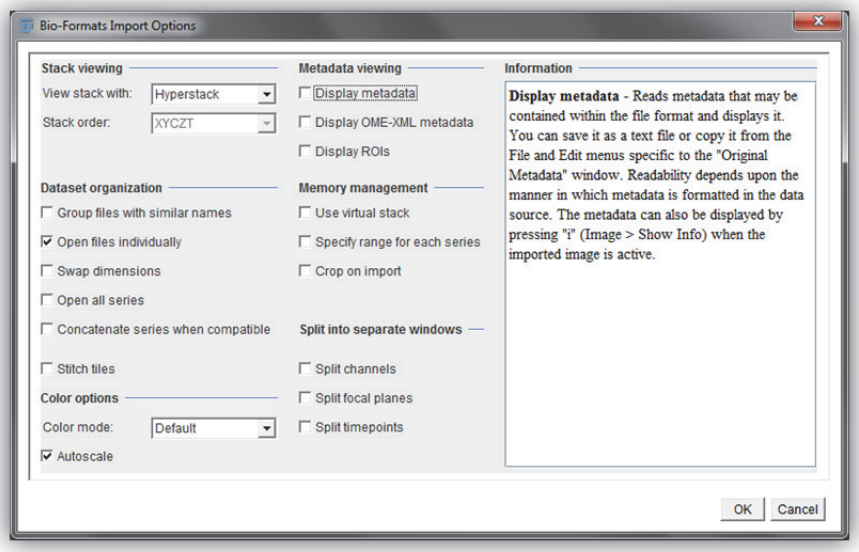
Fiji, through use of the LOCI (Laboratory for Optical and Computational Instrumentation) tools plugin, can open a large range of image formats including those from the big four confocal microscope manufacturers (Leica, Nikon, Olympus and Zeiss) and the Applied Precision DeltaVision systems. Opening each of these different formats requires different steps to open it.

The following Technical Note only covers opening confocal data. A subsequent Technical Note will cover working with the resulting stacks.

LOCI Tools

Initial File Opening

When a file is opened the LOCI tools will detect if it can open it or not and present you with the following dialog.



You can get information about each of the options by hovering the mouse pointer over it, a description will appear in the information box to the right.

In the top left is a drop down menu that lets you select how you want to view the stack. In general HyperStack is the best choice if you want to be able to easily create 3D views, make movies and change channel colours. Image5D has the same functionality with a few extra features that tend to complicate things more than they make it easy.

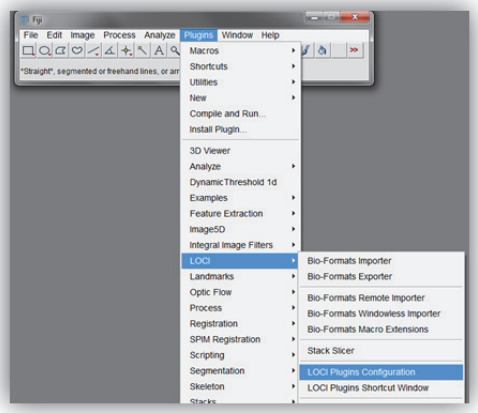
Displaying Metadata

In most cases a large amount of data about the image capture is stored in the files metadata. This includes information about the type of microscope used, calibration values, laser powers etc. To display this data you can either tick the **Display Metadata** box when opening a file or (if you only want to see the meatadata and not the image) select **Metadata Only** from the **View stack with:** pull down menu.

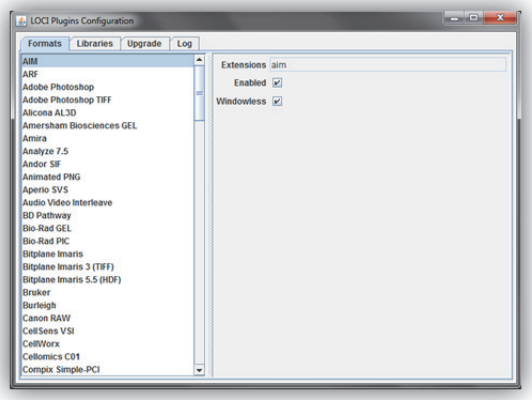
Default Opening States

Once you have decided on a standard way of opening a given file format you can set LOCI to open it the same way every time without giving you the import dialog window.

1. Go to **Plugins → LOCI Tools → LOCI Plugins Configuration**



2. In the resulting dialog window there is a list of image formats that LOCI can open on the left hand side. By selecting the desired format (In this example AIM) and checking the **Windowless** box LOCI will open the image directly without showing the import box.

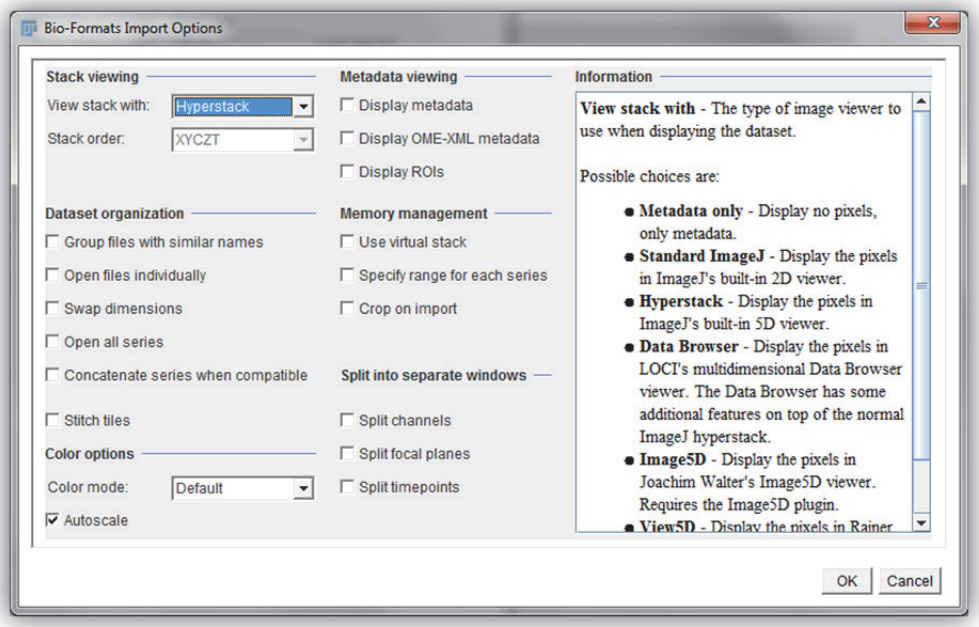


Opening Confocal Data

Leica – LIF Format

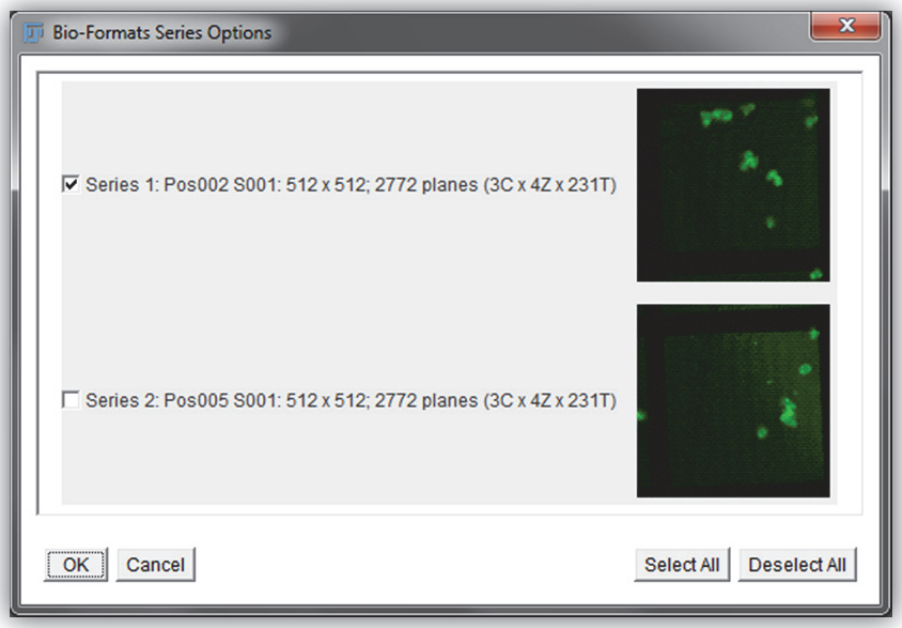
Leica confocal data (.lif files) usually contains multiple image stacks or experiments within one file. The LOCI plugin can open each of these separate data sets as individual stacks.

1. Open **Leica.lif** found in **Demo Images\Confocal.** The LOCI import window will open. Select to view the data with Hyperstack, make sure all other boxes except for the autoscale box are unticked and press **OK.**



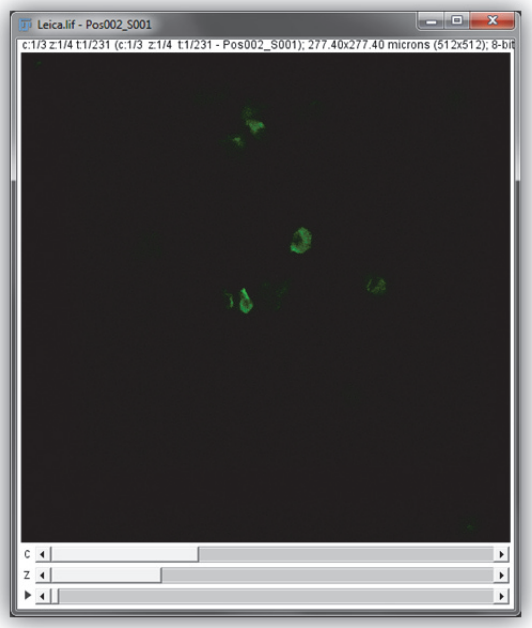
NOTE: If you have a large dataset and a computer with a small amount of RAM ticking the **Use Virtual Memory** box will only load in the files/slices/time points etc. as they are needed, instead of loading the whole stack into memory. This will however perform slower and certain plugins and filters may not work with a virtual stack.

2. You will then see a window that has thumbnails representing the different data sets contained within the file. Tick the box next to the one you want to open and press **OK**.

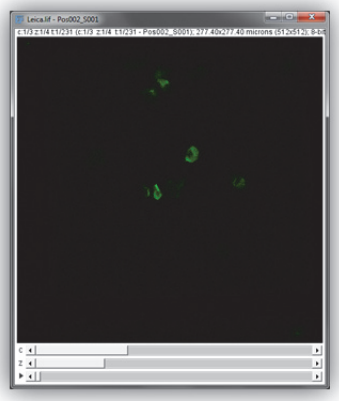
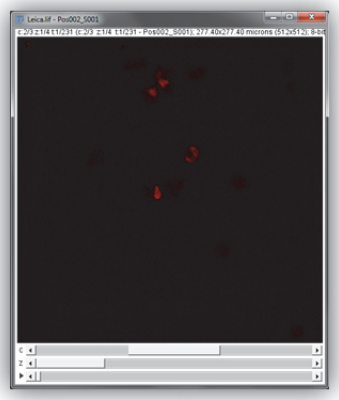
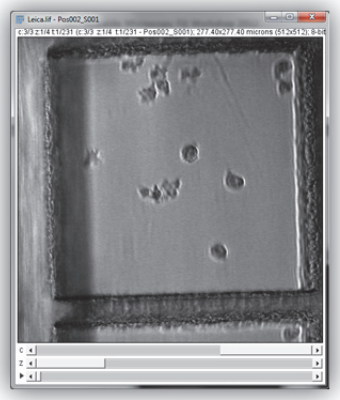


NOTE: Ticking multiple boxes will open all the ticked data sets. This could lead to stability issues.

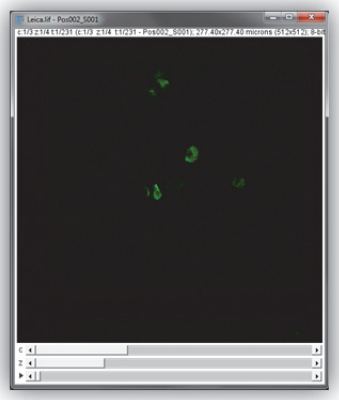
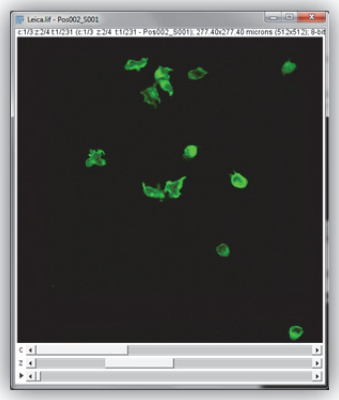
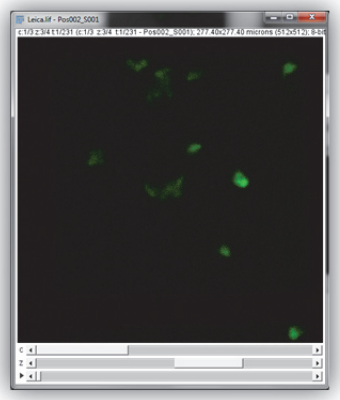
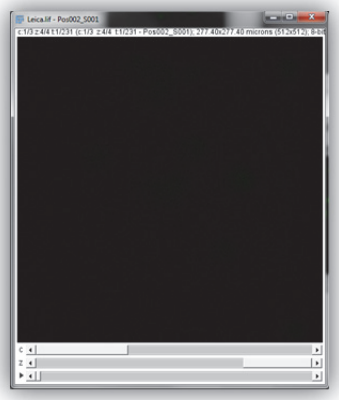
3. The resulting hyperstack should contain all the channel, z and time information from the selected data set.



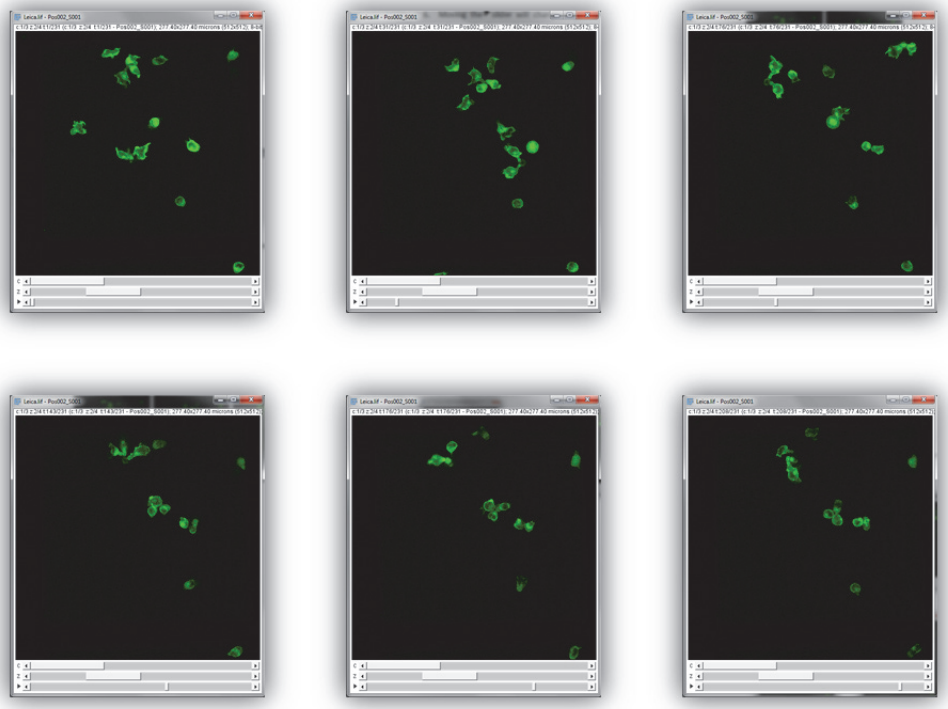
4. Moving the **C** (top) slider will select the different channels in the data set.

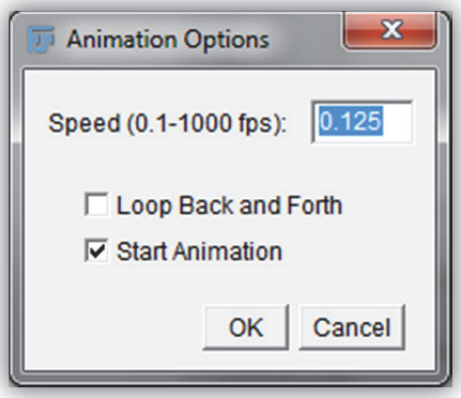
5. Moving the **Z** (middle) slider will select the different Z planes in the data set.

6. Moving the ▶ slider will change the current time point of the data set. Press the ▶ will animate the time series.

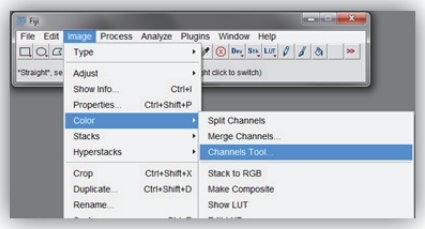


If you **Right Click** on the ▶ button you can change the speed at which the animation will play. You can also set whether or not you want the animation to loop continuously.

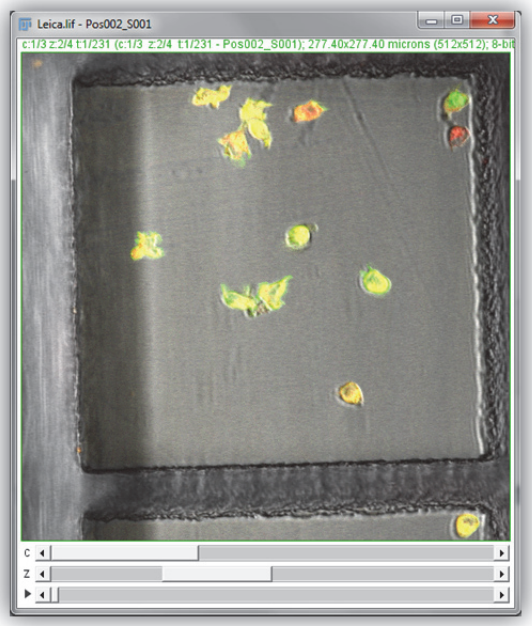


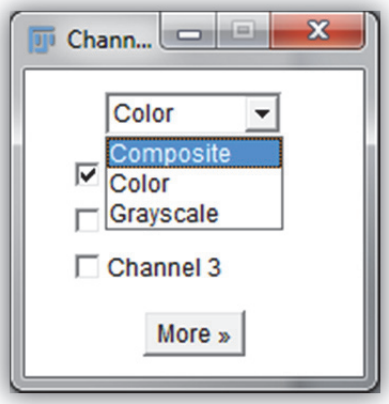
.

7. To view that stack as an overlay of all the channels go to **Image → Colour → Channels Tool**



8. In the **Channels** dialog box choose **Composite** from the drop down menu. The result will be a merge of the channels in the image using the default colours.





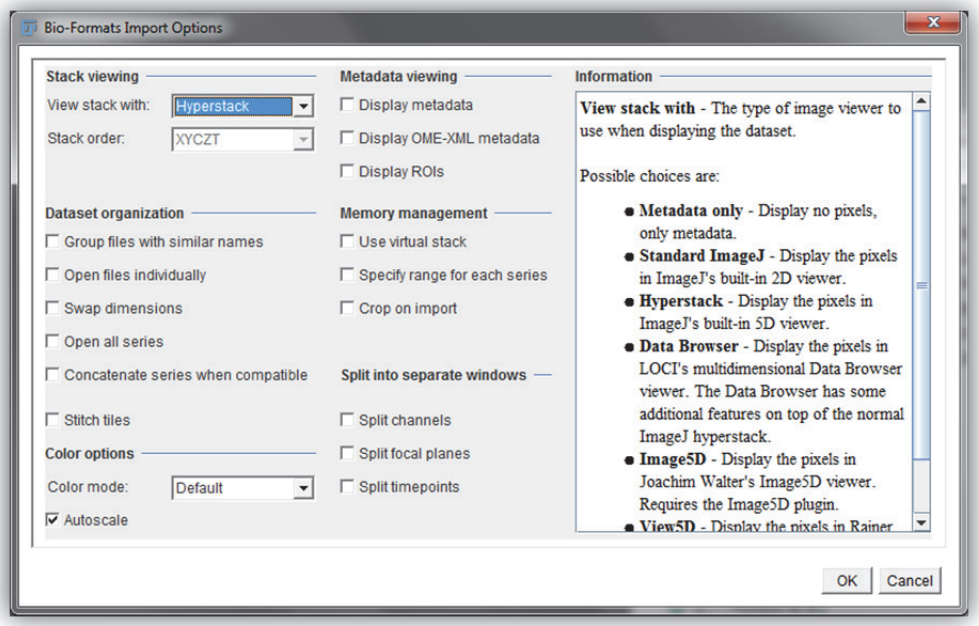
9. The colours of the channels can be changed by selecting the desired channel with the slider and changing the LUT as shown previously

Nikon – ND2 File Format

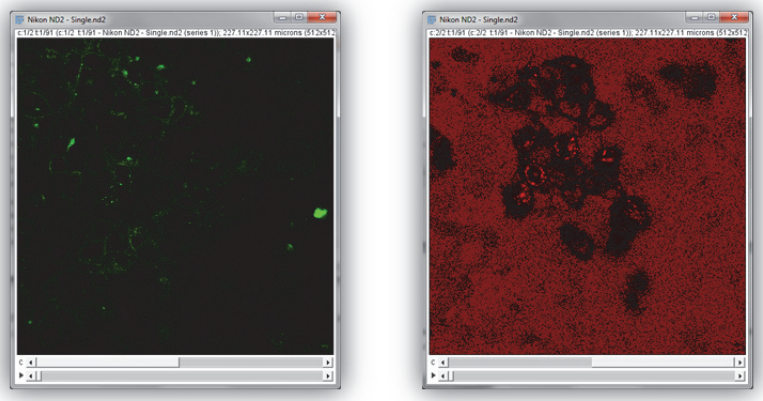
Newer Nikon Confocal Systems (C2, A1, nSTORM, nSIM) that use NIS Elements for controlling image acquisition can save data in Nikons ND2 file format. Until recently this format was quite difficult to deal with and needed a range of plugins to make it readable by Fiji. Now the LOCI tools can natively support it. The ND2 file format works similarly to the Leica LIF format in that it can contain multiple experiments/data sets in one file.

Single Data Set

1. Open **Nikon ND2 – Single.nd2** from the **Demo Images\Confocal** folder. Use the same settings as with the other data sets.

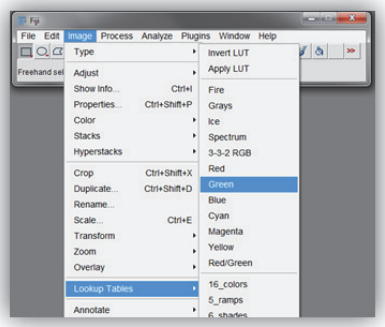


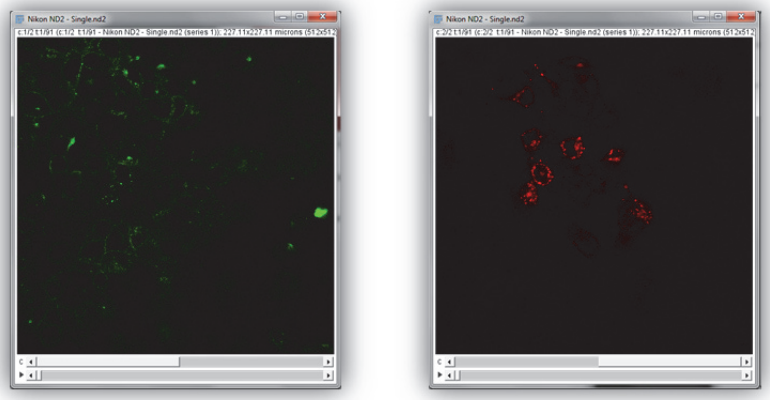
2. If you switch between the channels you will notice that the green channel has a lot of bright speckles in it and the red channel is really strange in the background.



This is due do the LUTs being corrupted on import. It can easily be fixed by re-assigning the green and red LUTs manually.

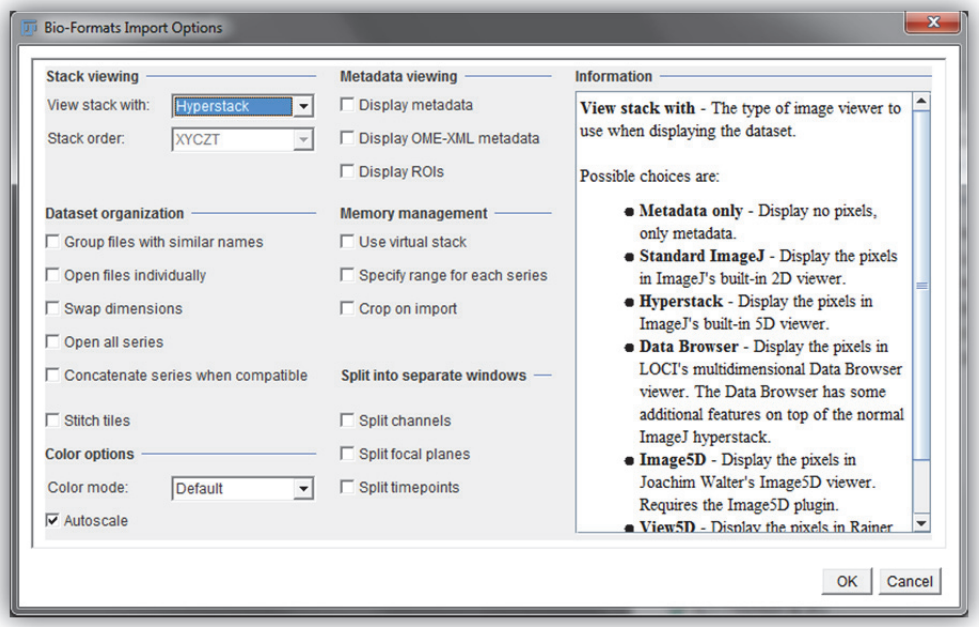
3. Select each channel separately and go to **Image → Lookup Tables → (Desired Colour)**





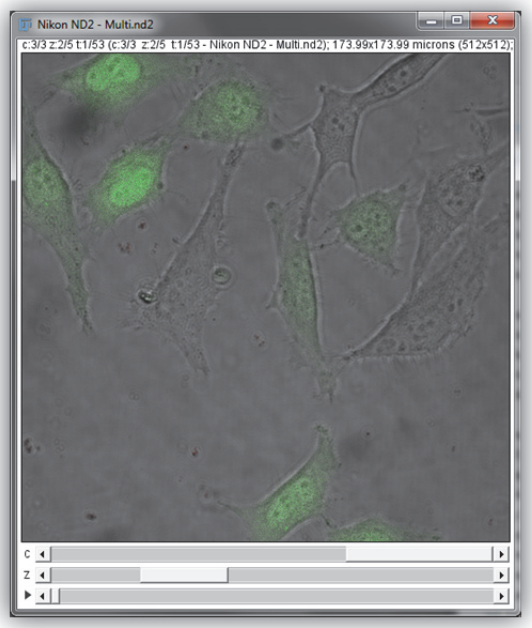
Multiple Data Set

1. Open **Nikon ND2 – Multi.nd2** from the **Demo Images\Confocal** folder. Use the same settings as with the other data sets.



2. This time you will get a second window showing the different data sets contained in the ND2 file, like what happens with Leica LIF files. Select a data set and press **OK**.

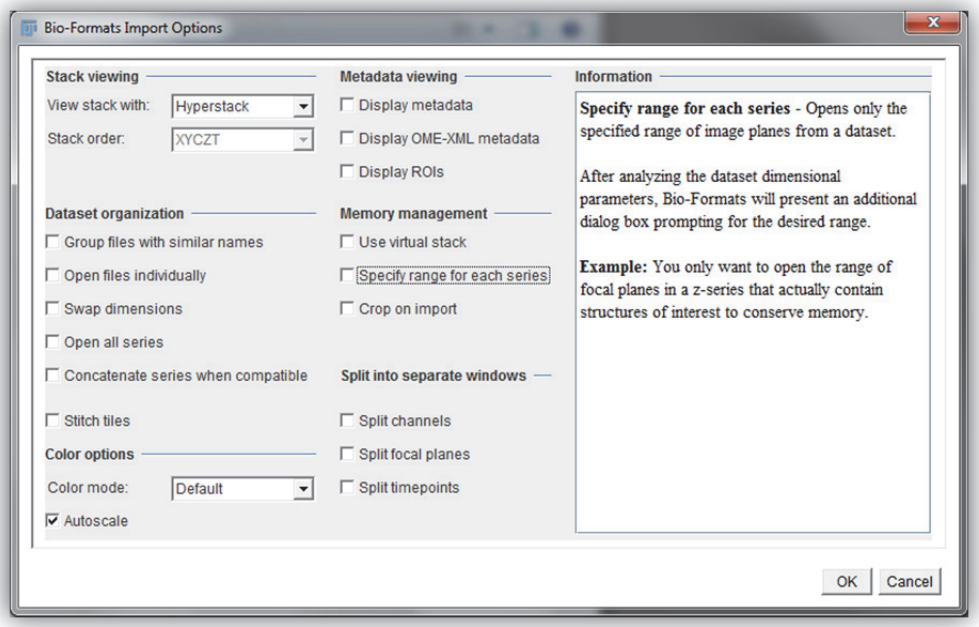
3. The image will open as a standard hyperstack that can then be changed/manipulated like the others.



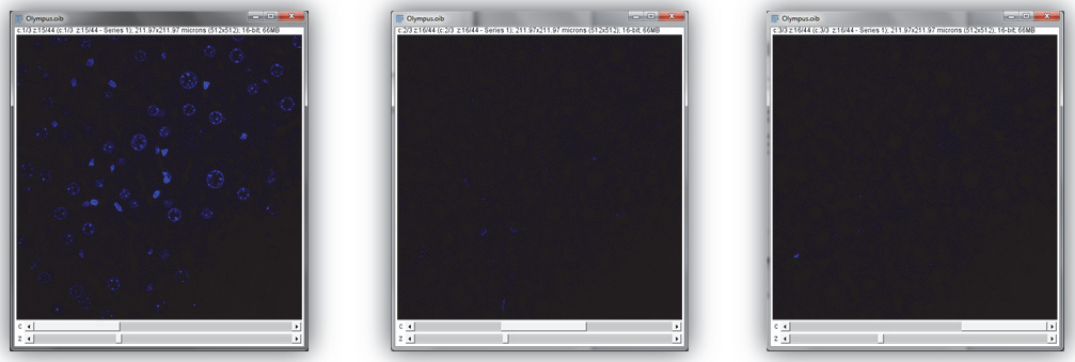
Olympus – OIB and OIF File Format

Olympus files do not always open with the right LUTs applied to the channels. But this is easily fixed with two different methods.

Firstly try opening the **Olympus.oib** file found in **Demo Data\Confocal** by setting the LOCI Import window as follows.



You will notice the resulting hyperstack has all the channels coloured in blue.



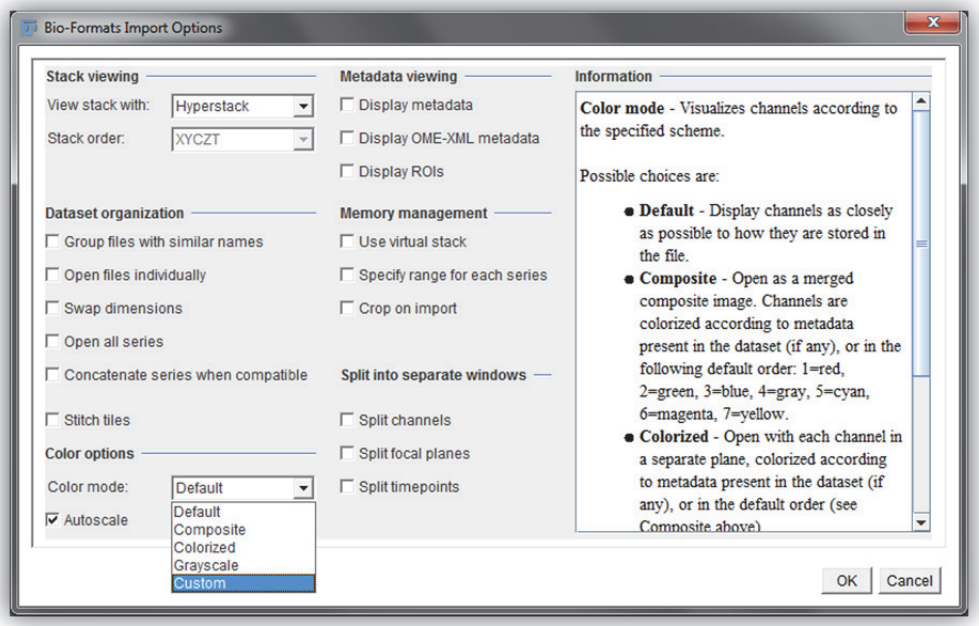
There are two methods to get around this, both are equally valid and just a matter of personal preference.

Method 1 – Reassign LUTs

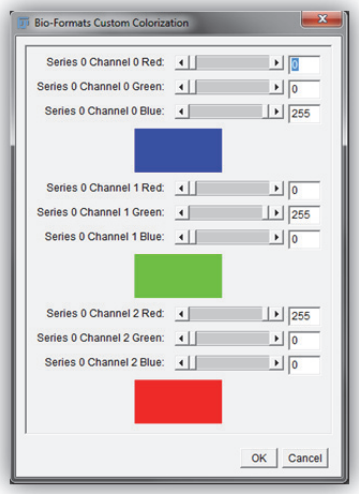
As in previous examples different LUTs can be selected from the LUT menu for each of the channels/

Method 2 – Applying Colour at Opening

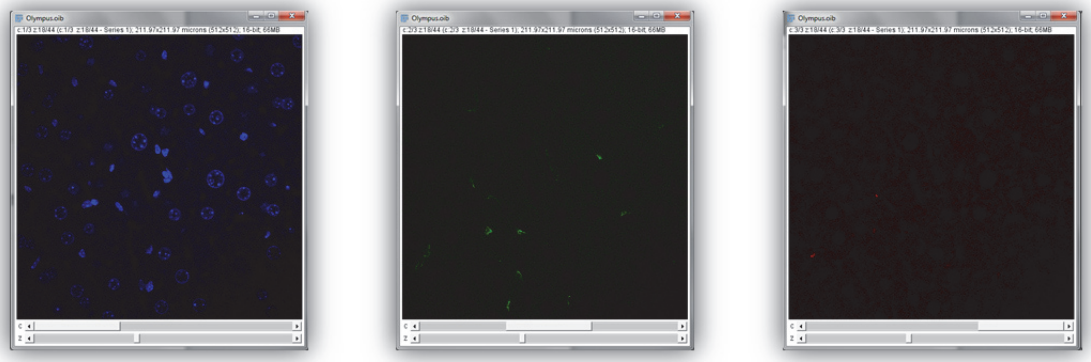
1. Open the **Olympus.oib** file but this time select **Custom** as the **Colour Mode** and press **OK**.



2. In the resulting **Bio-Formats Custom Colourization** window you can set the RGB colour values for each of the channels. Once you are happy press **OK.**



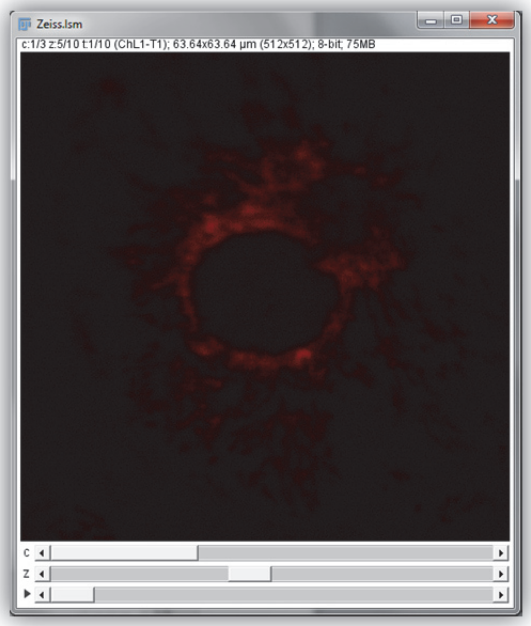
3. The result will be the same as using the channels tool above.



Zeiss – LSM File Format

Zeiss lsm files will open as a hyperstack directly without any dialog box being shown.

1. Open **Zeiss.lsm** from the **Demo Images\Confocal** folder



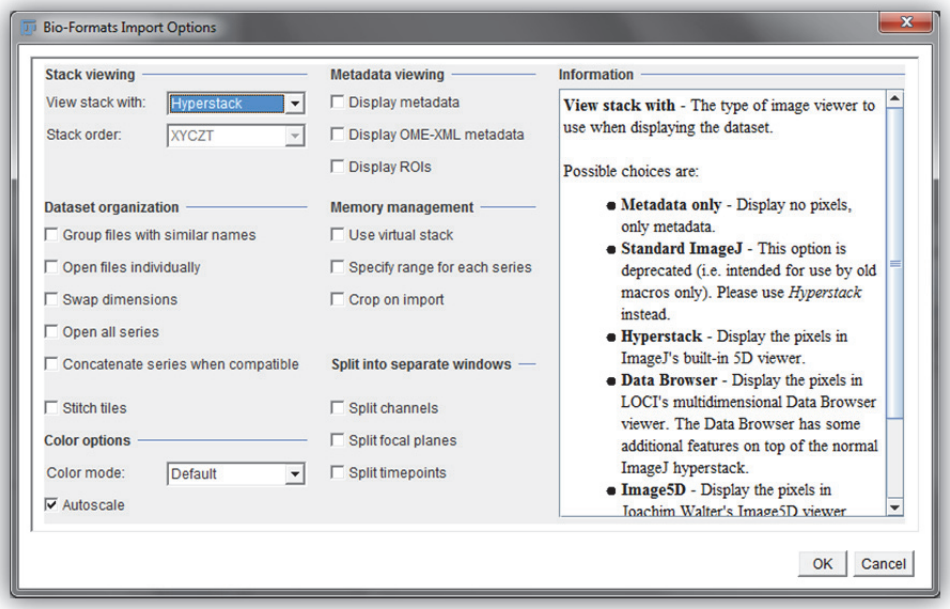
.

2. As with the other hyperstacks the channel tools can be used to create composite images, change LUT colours etc.

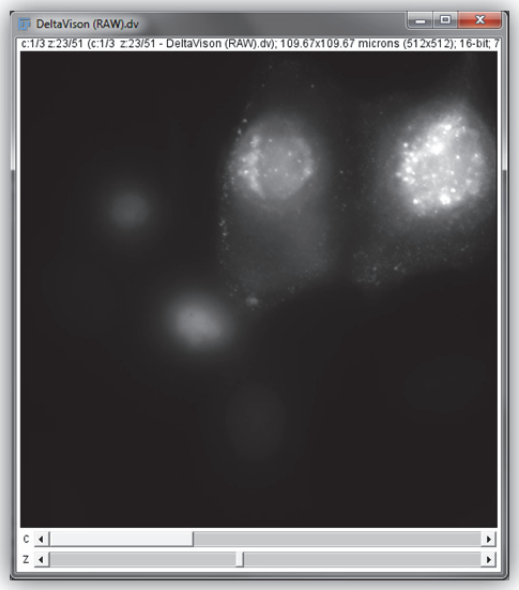
Opening DeltaVision Data

The .DV file format produced by the DeltaVision systems (both deconvolution and SIM) can be opened using the LOCI tools. They will open in grey scale and will need to be false coloured if required.

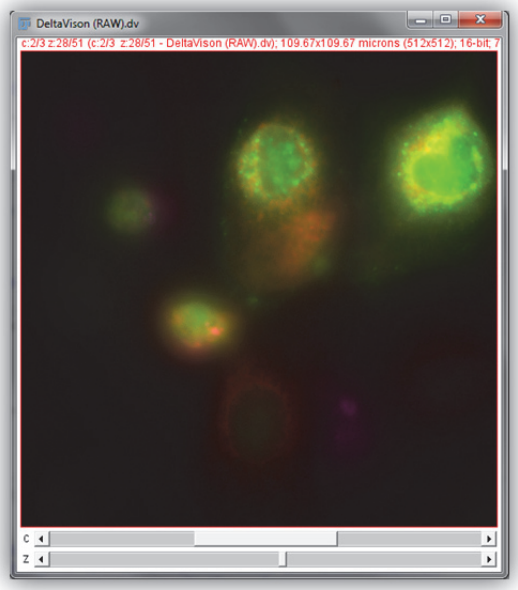
1. Open **DeltaVision(RAW).dv** from the **Demo Images\Deconvolution Folder.** Once again leave settings as before and press **OK.**

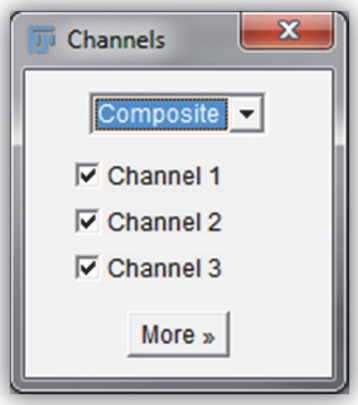


2. The stack will open in grey scale.



3. As in the previous examples the colours can be changed using the **Channels Tool**





Visualising 3D and Live Data

Aim

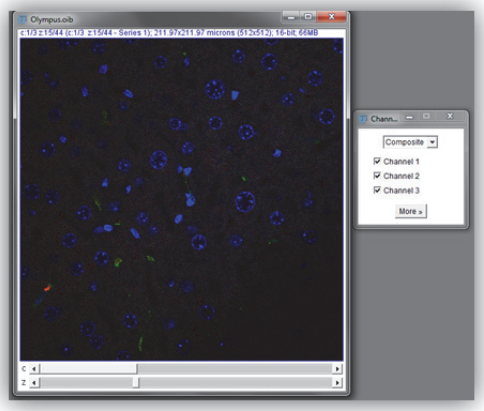
Data that is captured on a confocal microscope or live imaging system can be represented in a number of ways that can best address the question being asked. For confocal data there are several ways of showing the 3D data that is captured and for live data the easiest way to represent it is to create a montage or a movie of the time series.

Visualising 3D Data

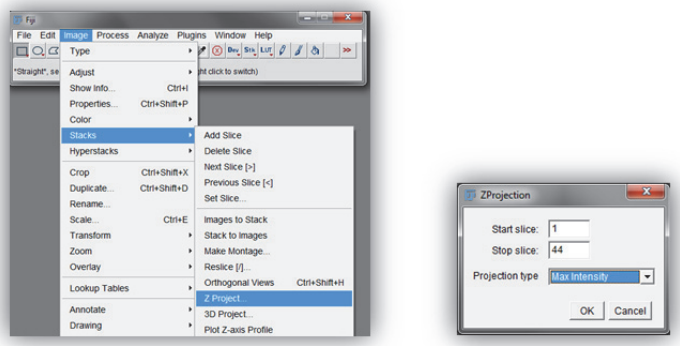
Z Projection

A Z projection is created by combining all the Z slices in a data set together to generate an image. The usual type of projection is a maximum intensity projection, this means that the highest intensity pixel from each slice is projected to the final image. Other projections are average, minimum, sum, standard deviation and median. Each has its uses, but a maximum projection is used most often in fluorescent imaging.

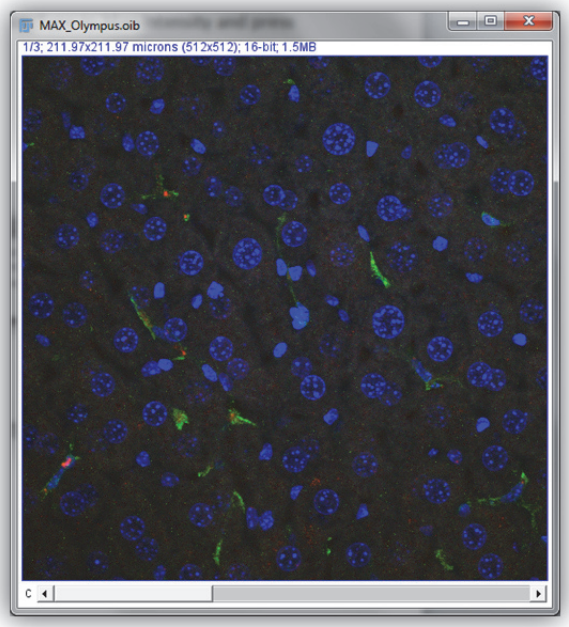
1. Open the **Particles in Cells.tif** file found in the **Demo Images\Confocal Folder** and reconfigure the colours as you did before.



2. Go to **Image → Stacks → Z Project**.. and set the Projection Type to Max Intensity and press OK.



3. The resulting image will represent the brightest pixel (in Z) at each x,y position.

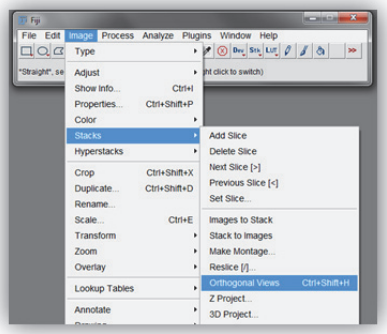


4. Try again with different Projection Types.

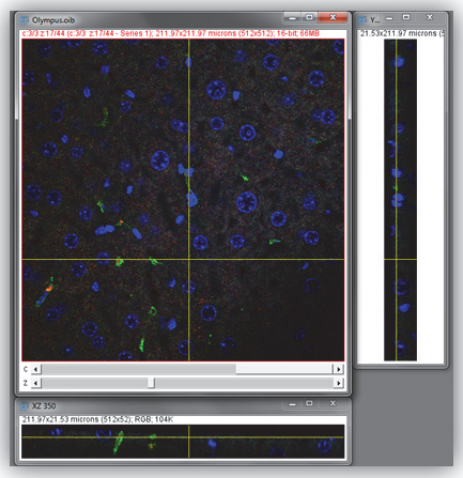
Orthogonal Projection

An orthogonal projection is a view created in the YZ or XZ dimension of an image stack. An orthogonal projection allows you to visualise depth information one slice at a time in your sample.

1. Open the **Particles in Cells.tif** file found in the **Demo Images\Confocal folder**. Go to **Image → Stacks → Orthogonal Views.**



2. Two windows will open up, one to the right and one below the original stack. These windows show the orthogonal projections in XZ and YZ. To change the view seen in each window move the yellow cross hair in the original stack.

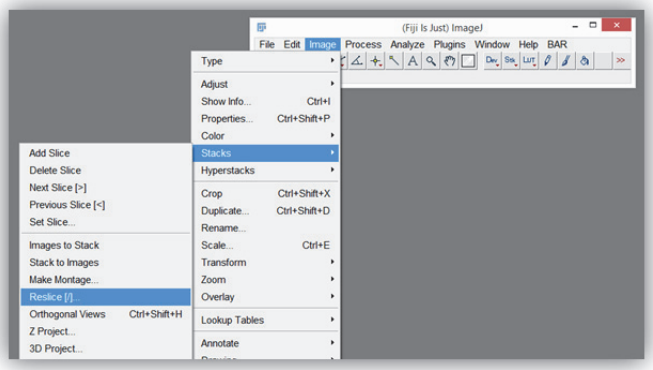


NOTE: Each of the individual XZ and YZ images can be saved for later use.

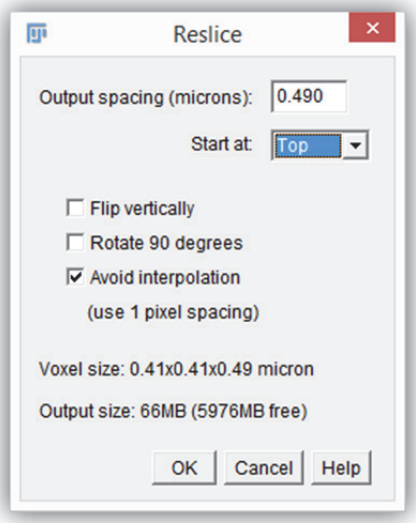
Orthogonal Stack/Reslice Stack

The orthogonal projection only lets you see one slice at a time and it has the overlay lines in the way. Sometimes it can be useful to generate a stack of orthogonal slices instead. This is easily achieved by reslicing the stack along a different axis.

1. Open the **Particles in Cells.tif** found in the **Demo Images\Confocal** folder. Go to

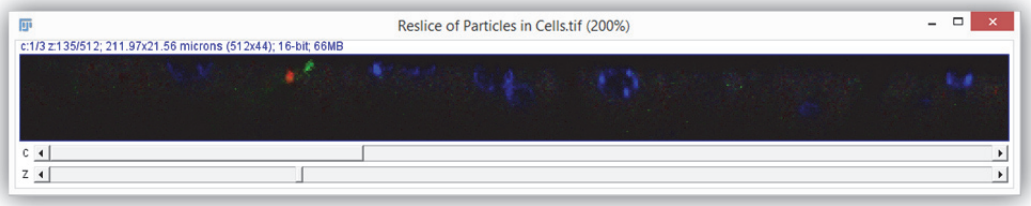


2. The dialog that opens lest you set how the stack should be resliced (top, bottom, left, right) and to set the distance between slices. In this case the stack is already calibrated so there is no need to enter a value, if it is not calibrated and you know the calibration you can enter it here. **NOTE:** This is the z spacing calibration in this example, not the xy calibration.



There is a tick box to select that says **Avoid Interpolation**. This should usually be ticked. While interpolation will make the end result look better, it does this by adding extra data that wasn't in the original image.

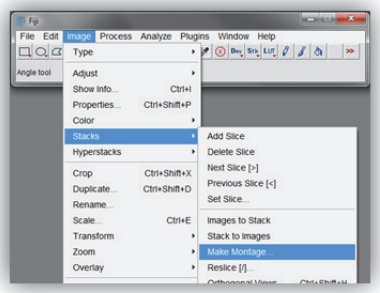
3. A new stack is generated that represents the orthogonal planes of the image. This stack can be treated as any other stack and can be made into a movie or a montage.



Slice Montage

Creating a montage allows you to show all (or a sub set) of the slices from a confocal series in one image. This technique can also be used for a time stack to show different time points in a data set.

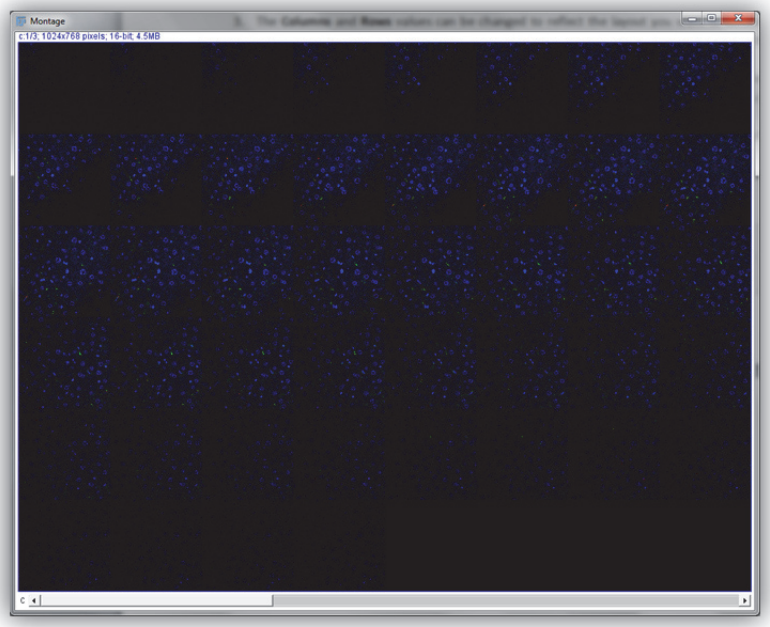
1. Open the **Particles in Cells.tif** file found in the **Demo Images\Confocal folder**. Go to **Image → Stacks → Make Montage…**



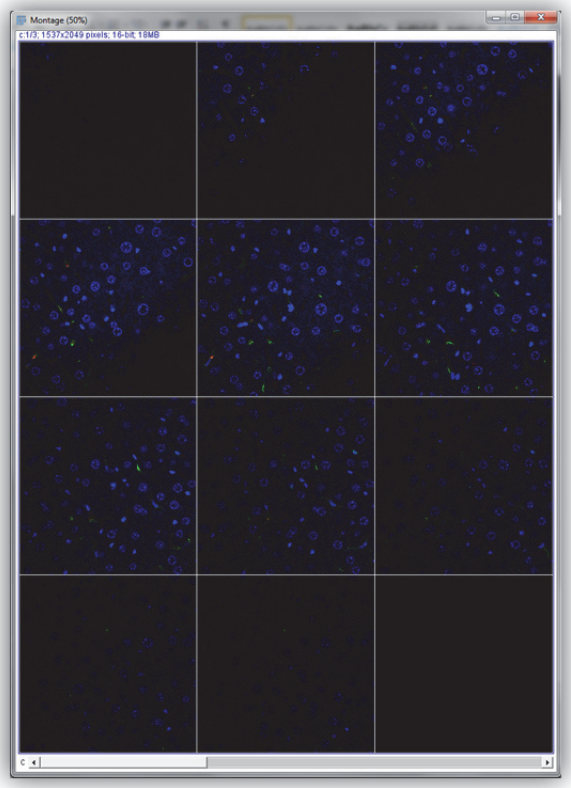
2. The **Make Montage** widow will open. This can be configured to generate the montage you want. The default settings should be something like those below

3. The **Columns** and **Rows** values can be changed to reflect the layout you want to achieve. **NOTE**: These values are not dynamically linked to the data so it is possible to change the values so that all your data will not fit in the montage.
   The **Scale Factor** is the amount each individual slice will be downsized to reduce the final size of the montage. If the montage is not going to be too big it is probably best to set this to 1 and resize the image later if required.
   The **Increment** value is the number of slices that will be used for the montage. A value of 1 means all slices will be used, a value of 2 means every second slice will be sued etc.
   The **Border Width** sets the thickness of the border line that will be drawn around each frame, a value of 0 gives no border.

4. Leave the values at their default for now and press **OK.**



5. Try creating a montage with the following settings.

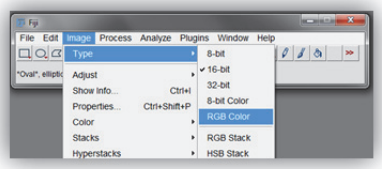


3D Volume – Simple

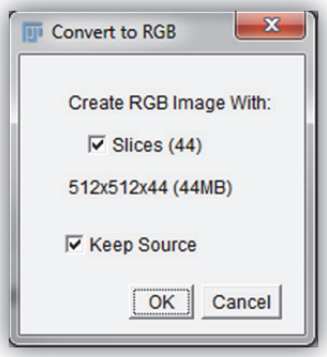
There are various ways to project 3D data in Fiji, some of which will be covered later in the course. The simplest method is a standard 3D projection.

1. Open **Particles in Cells.tif** from the **Demo Images\Confocal** folder. Before proceeding the stack needs to be converted to a standard RGB stack as the 3D projection module cannot work with 16bit data.

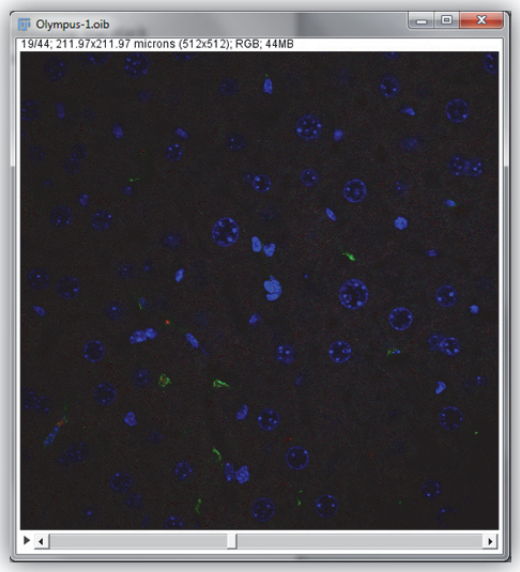
   Go to **Image → Type → RGB Colour**



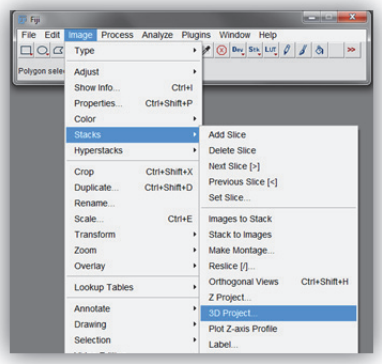
2. In the next window leave the setting as they are below and press **OK.**

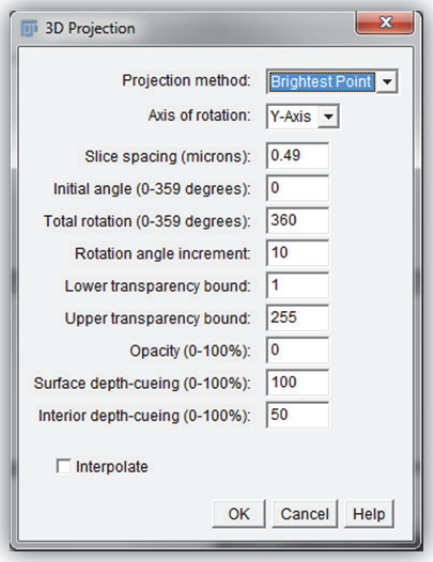


You should now have an RGB stack of the data

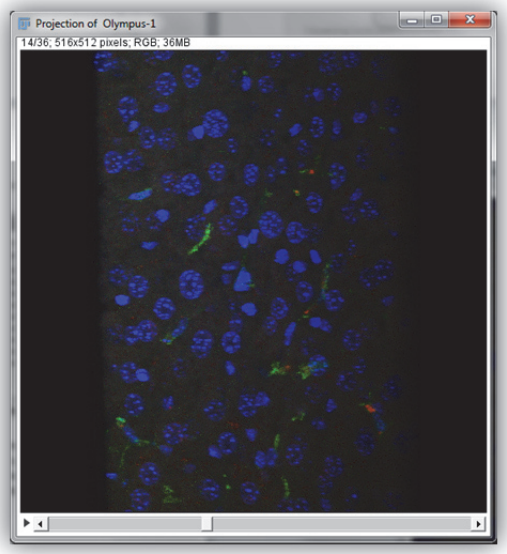


3. Now go to **Image → Stacks →3D Project…** Leave the settings as default and press **OK.**





4. The resulting image will have a slider at the bottom like a normal stack but when you move it the image will rotate around the Y axis. You may notice when you look end on to the stack you can see lines between each of the slices. This is because the data has not been Interpolated (essentially had the missing information guessed at). You can fix this by building the volume again, but this time tick the Interpolate box.

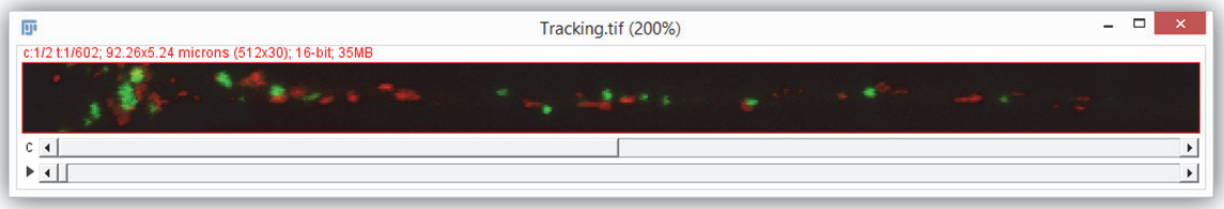


5. Try generating other models by changing the options in the 3D projection dialog box, try different rotation axis, angles or projection methods.

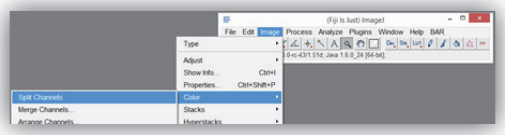
Kymograms

A kymogram is a way of representing live data in a kind of graphical way. Instead of showing live data as a movie it is represented as a 2D image with one dimension of the image representing a physical dimension (x, x or Z) with the other dimension representing time. They are a powerful way of easily representing what is going on in a live data set.

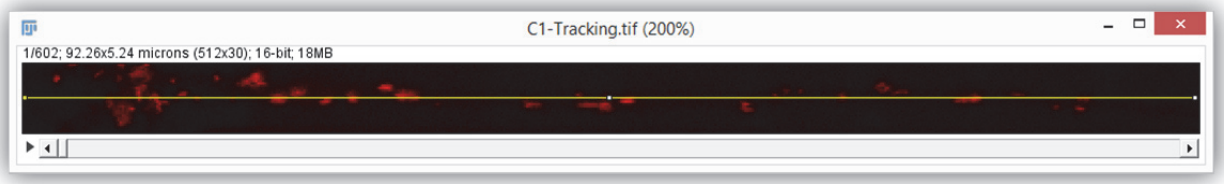
1. Open **Tracking.tif** from the **Demo Images\Confocal** folder



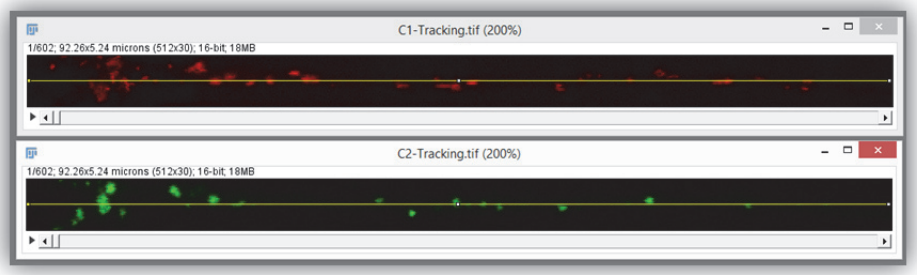
2. Split the channels by going to **Image → Colour → Split Channels**



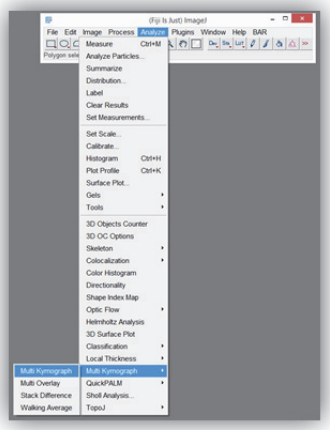
3. Use the line tool to draw a line down the middle of **channel 1** image. **TIP** hold down the **Shift** key to keep the line perfectly straight.



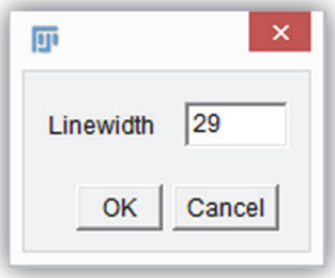
4. Select **Channel 2** and copy the line region to it by pressing **Ctrl + Shift + E**



5. Select **Channel** 1and go to **Analyze → Multi Kymograph → Multi Kymograph**



6. In the dialog that comes up you need to define a line width. This is the width of the line that will be used to average out the intensities to give you your kymograph. A higher line width will average out some noise in the resulting image, but too big a width will blur out real data too.

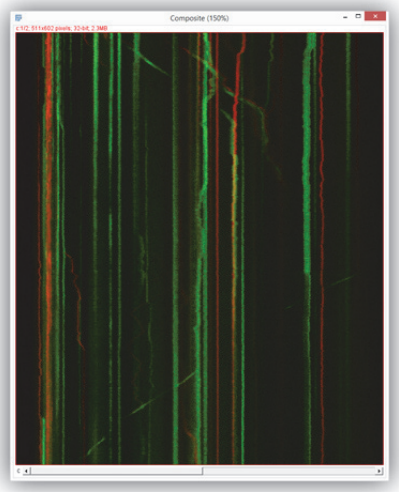


NOTE: Line width values have to be an odd number for this module to work

7. You will get an image like the following. This image show the x coordinates across the image and time, from start to end, down the image vertically. The angle of a track shows you its speed, a more horizontal line is from a particle traveling faster. The vertical lines represent particles that are not moving.

8. Rename the image to **Channel 1** by going to **Image → Rename**
9. Repeat for channel 2 so you end up with kymographs for each channel

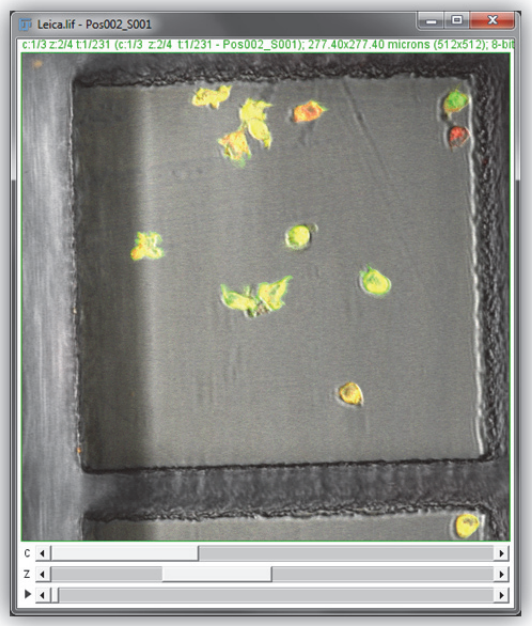
10. Merge the two channels together to see the difference in the tracks of the two markers (in this example channel 1 is green, channel 2 is red)



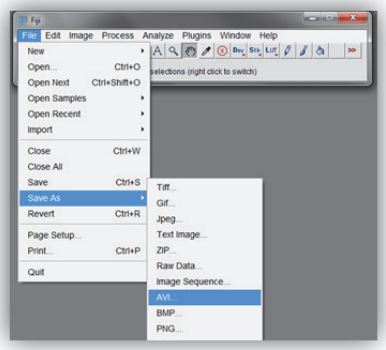
Making Movies

Any stack (Z series, time series, 3D rotation etc.) can be easily exported as a movie to show or play in PowerPoint.

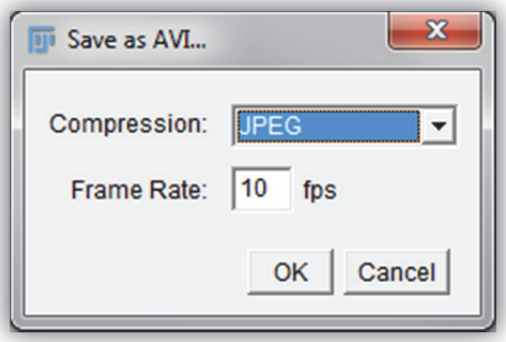
1. Open one of the data sets from the **Leica.lif** file from the **Demo Images\Confocal** folder. Set it to be a composite image and select one of the middle z slices.



2. Go to **File → Save As → AVI..**

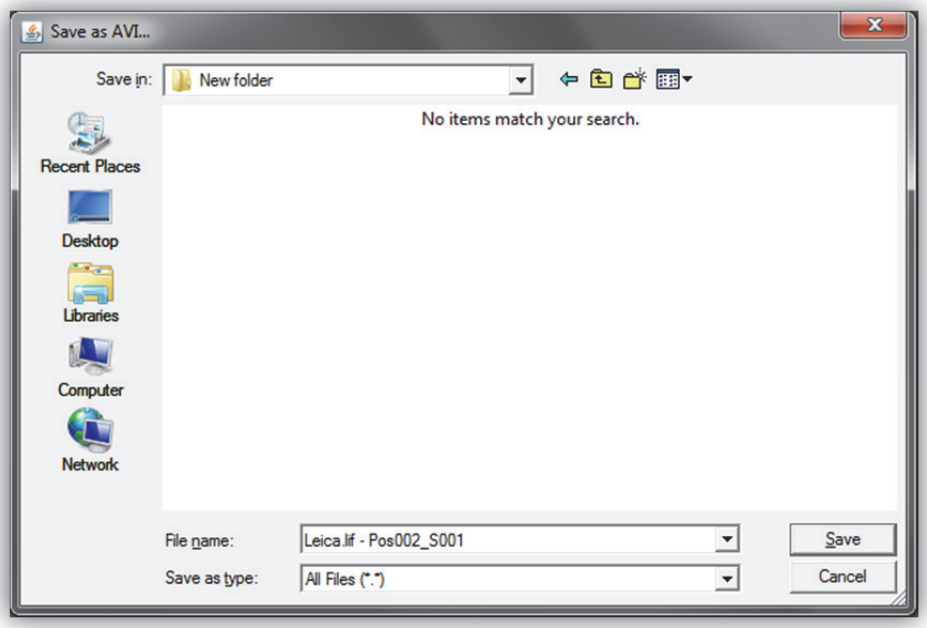


3. In the **Save as AVI** window select **JPEG** as the **Compression:** type and set the **Frame Rate** to **10.** Press **OK.**



The setting for the frame rate will depend on several factors: the length you want your movie to be, the number of frames/slices etc.

4. In the next window choose a name (or leave it as default) and a location to save your AVI file.



5. Open the file you just saved to see the movie in the default media player. **NOTE:** if you wish to save the file in a different format (e.g. MOV, mp4 etc.), save it as an AVI with no compression and then use third party software to convert it.

Simple Analysis – Area and Intensity of Stain

Aim

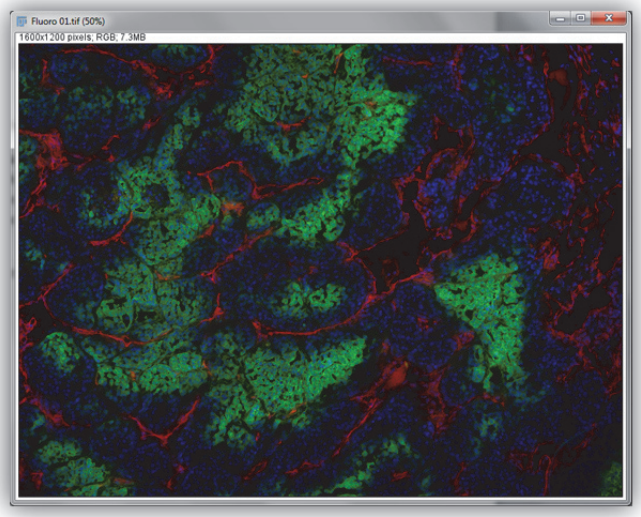
Analysis of the area covered and/or the intensity of a given stain can be a powerful tool in image analysis. It can also form the basis for more complex analysis.

Measuring the Area and Intensity of Stain - Fluorescence

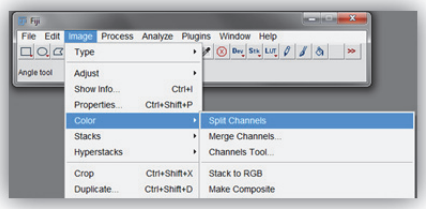
Separating Merged Multichannel Images

Before analysis of an individual stain/channel can occur it may be necessary to separate out the colour channels as individual images. This will only work for an image that was created by merging 3 fluorescent images together. It is not possible on standard colour images (H and E, DAB, etc.) or on merges of more than 3 fluorescent channels (unless it has been saved as a hyperstack/multi plane tiff).

1. Open **Fluoro 01.tif** from the **Demo Images\Widefield Images\Fluorescence Measurement** folder



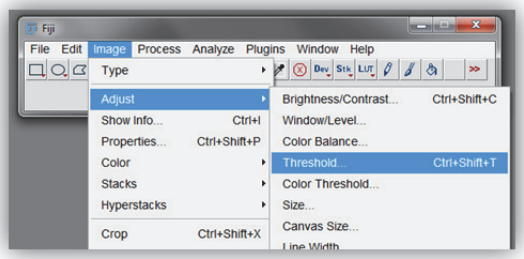
2. To separate the channels go to **Image → Colour → Split Channels**



3. The image will be split into 3 images. Each image in this case is only an 8bit monochrome image but will be suitable for basic measurement

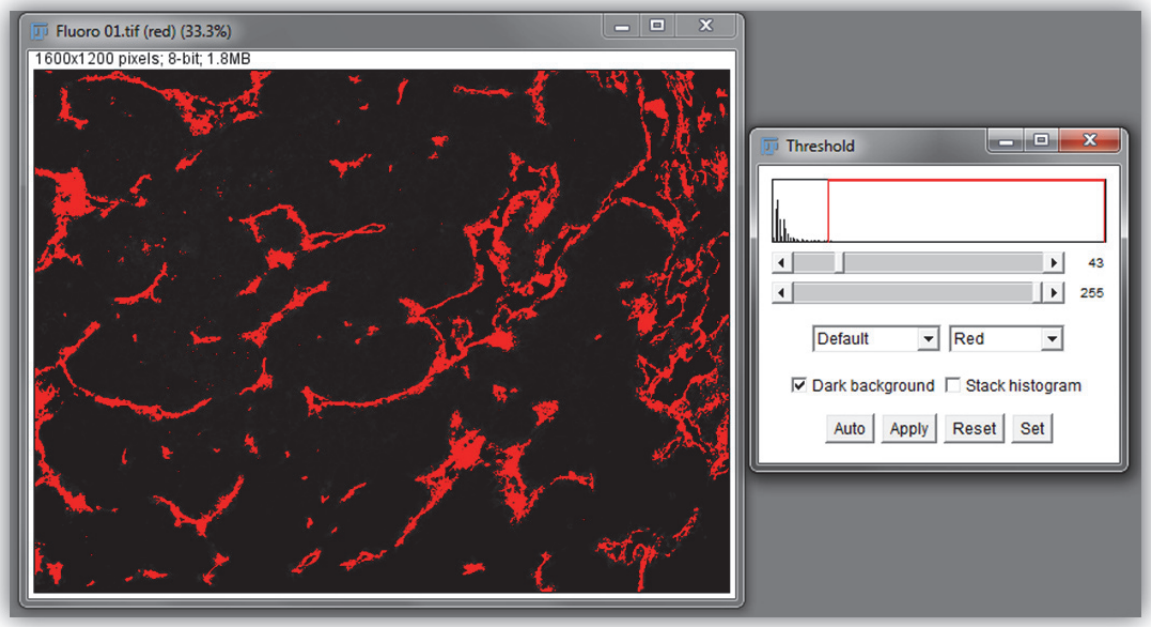
Threshold Image

1. For an image to be able to be measured it first needs to have a threshold applied to it. Select the red channel image (blood vessels). Go to **Image → Adjust → Threshold**



2. Tick the dark background box and leave all other settings at default. A red mask will be placed on the parts of the image that are selected by the threshold. You can adjust the threshold by moving the sliders. The top slider sets the bottom range of the threshold and the bottom slider adjusts the top range of the threshold. The red box on the histogram shows which parts of the histogram are being thresholded.

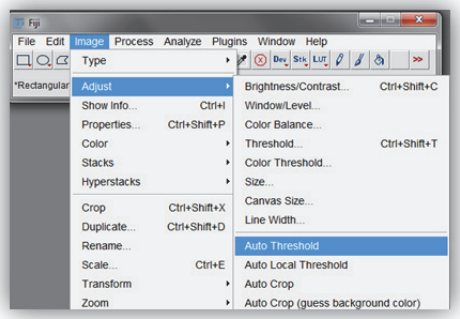
   If you press the auto button Fiji will attempt to automatically threshold the image. The drop down box on the left (the one that says default) has 16 different auto threshold algorithms to choose from.



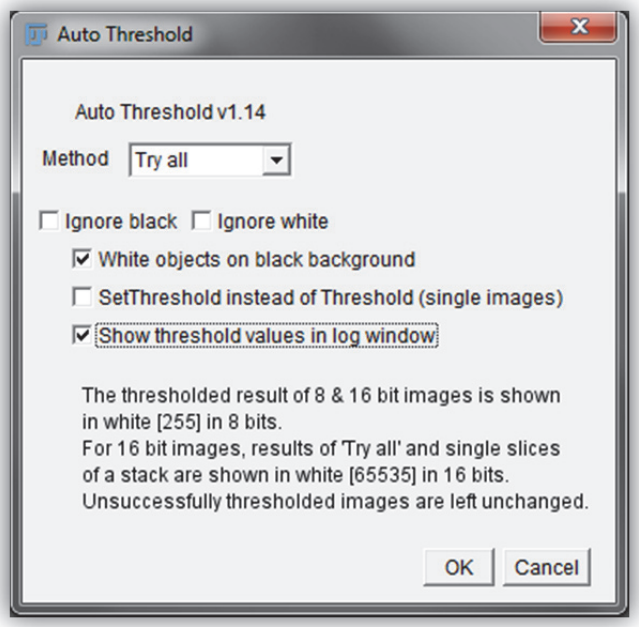
Automatic Threshold Algorithms

Fiji has a module that will allow you to easily test each auto threshold algorithm so you can decide which the best to use for your sample is.

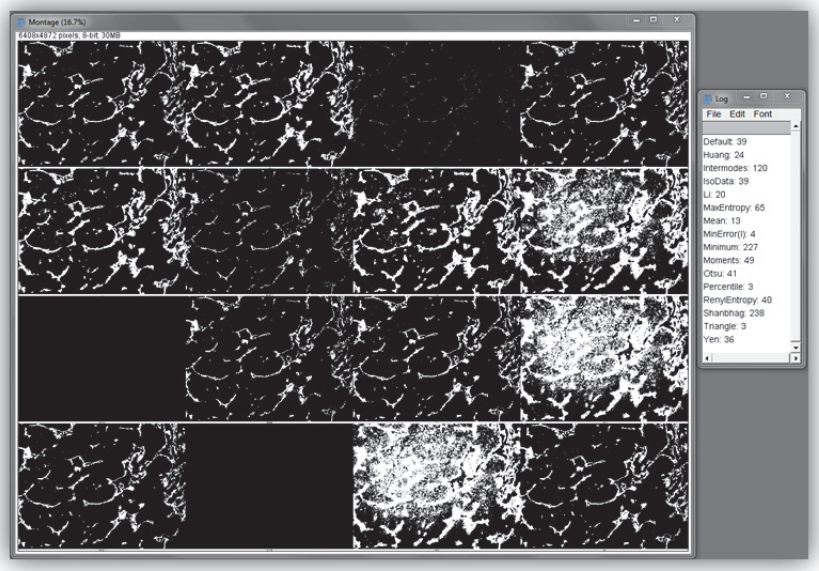
1. Go to **Image → Adjust → Auto Threshold**



2. In the dialog box that opens up set the method to Try All, make sure the **White objects on black background** and **Show threshold values in log window** boxes are ticked. Press **OK**.

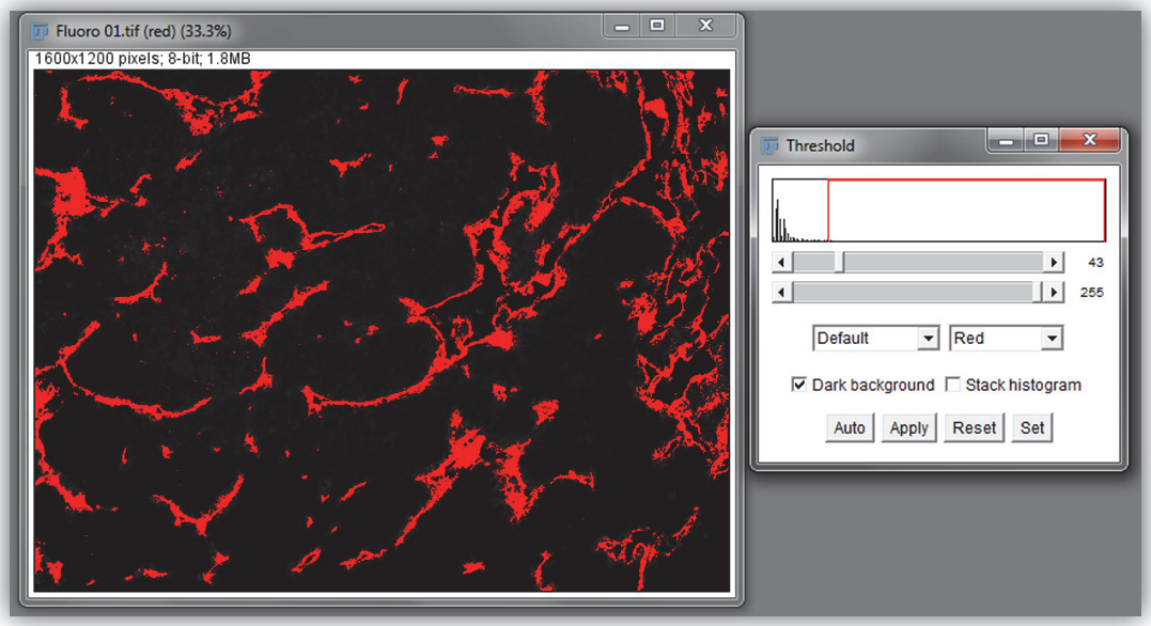


3. The resulting montage shows what each auto threshold algorithm would produce on the image. Each frame of the montage represents a different algorithm which is listed in the log that was created.

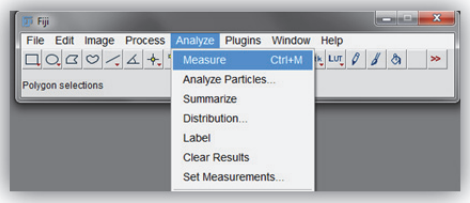


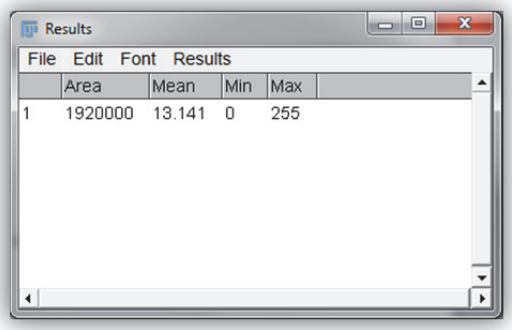
Measure Thresholded Area

1. Go back to the original red channel image and set a threshold on it. The default one is fine.



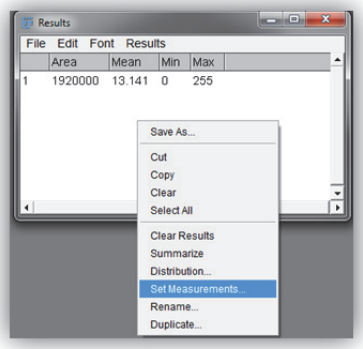
2. Go to **Analyse → Measure**. A window similar to the one below will open up





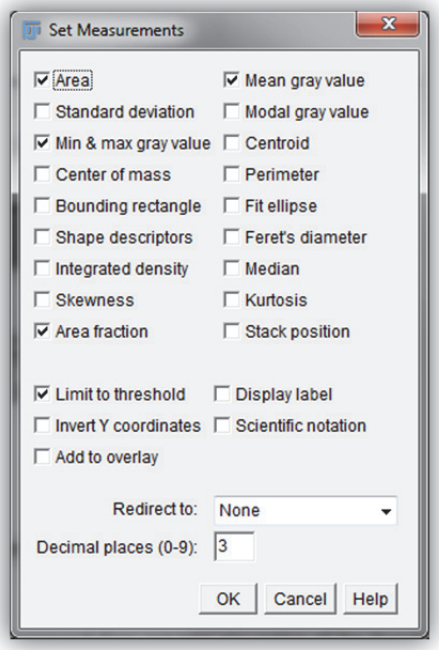
By default the measure command does not take any notice of the threshold. The current data measured shows the area of the whole image (since this is an uncalibrated image we can tell it was taken with a 1.92 megapixel camera), the average intensity of the whole image and the minimum and maximum intensity values.

3. Right click on the results window and select **Set Measurements**.

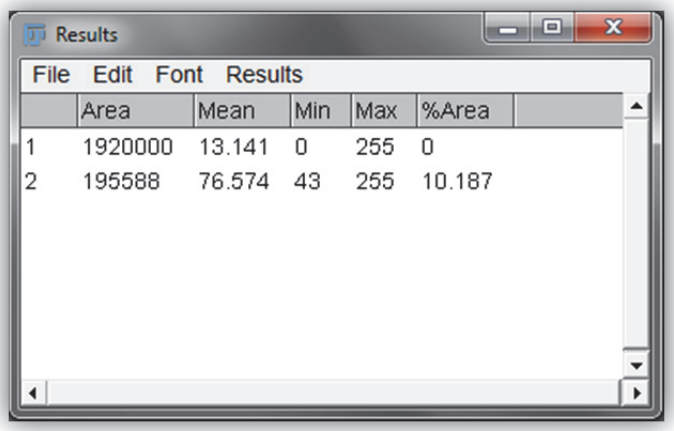


NOTE: You can also go to **Analyze → Set Measurements…**

4. The resulting window allows you to select what measurements you wish to make. It also allows you to limit the measurements to only thresholded areas. Select **Limit to Threshold** and **Area Fraction.** Set as below and press **OK.**



5. Now remeasure the image. The values for area, mean grey value and min and max grey values that are measured this time apply only to the areas under the threshold. The results can be saved to excel or a txt file if required.



NOTE: The value for area is in pixels as this image is uncalibrated.

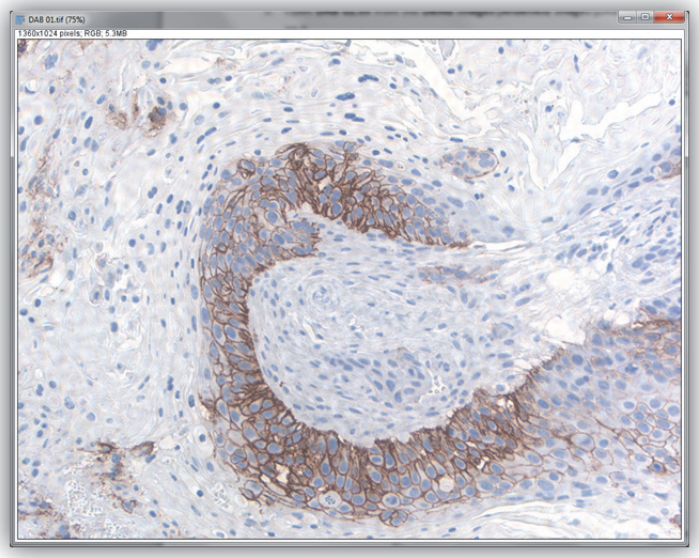
Measuring the Area of Stain – Brightfield/Chromagens

For brightfield images, like those stained with DAB chromogen or H and E, the area of stain in an image can be measured by thresholding like above. It is a little more complicated as chromogens, especially DAB, are not a pure colour but made up of a mix of red, green and blue.

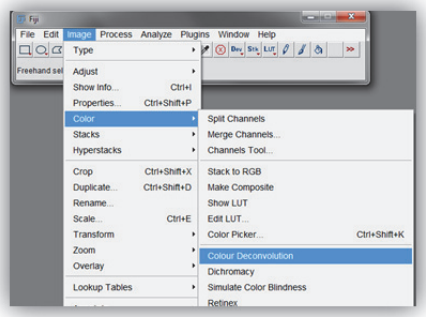
The easiest way to be able to measure this brown channel is to extract it out into its own image using colour deconvolution. Colour deconvolution allows the separation of colours by an automated threshold method. The result is not a threshold you can measure but 3 separate images representing the "channels" of the original image. These "channel" images can then be treated like single channel fluorescent images.

NOTE This technique is only good for counting objects or measuring areas, it cannot be used to measure intensities of the stain.

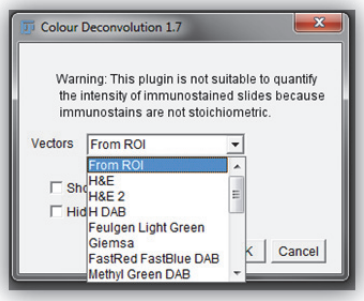
1. Open **DAB 01.tif** from the **Demo Images\Widefield Images\DAB**



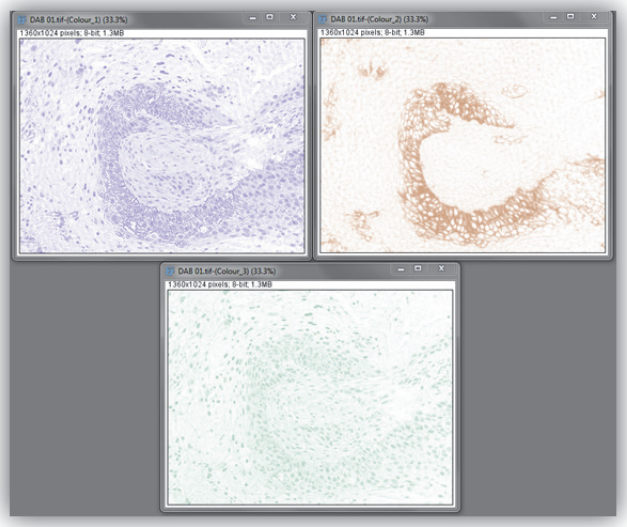
2. Go to **Image → Colour → Colour Deconvolution**



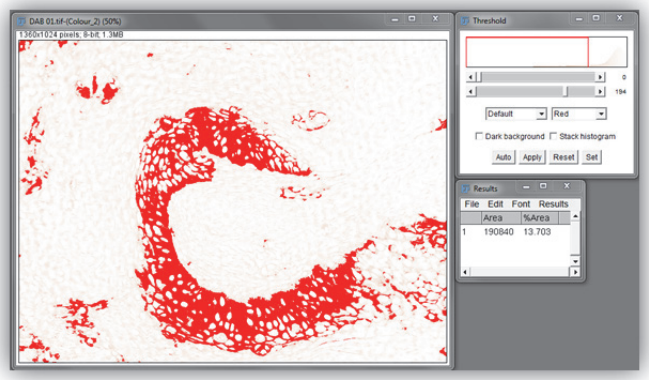
3. In the resulting window there is a choice of options in the pull down menu. The first option, **From ROI**, lets you specify the different stains in your image manually. The others are preconfigured to work with most standard histological stains. For this example select **H DAB** and press **OK.**



4. The original image will be split into three images.



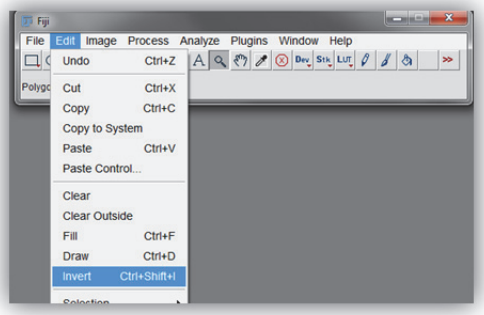
5. The three images represent the haematoxylin stained nuclei (Blue, Colour_1), the DAB chromogen (Brown, Colour_2) and what is essentially a mathematical remainder but may be useful for some things. This third channel may contain another stain if the original image was a 3 channel stain (e.g Mason' Trichrome).
6. The brown channel image can now be thresholded and measured as if it was a single fluorescence channel. Make sure that **Dark Background** is not checked this time though.

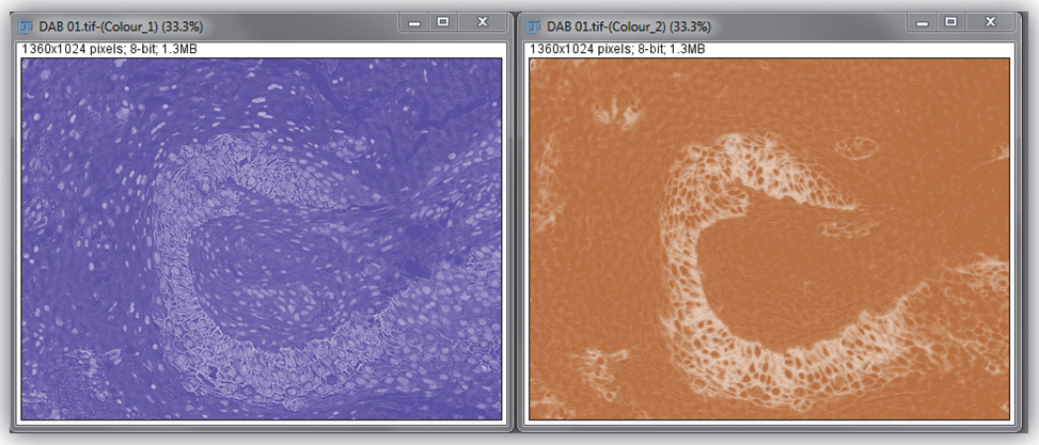


Make a Brightfield Image Look like a Fluorescent One

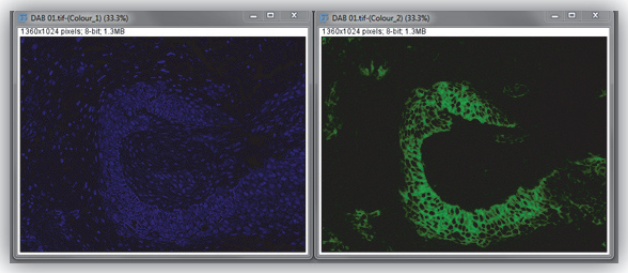
Since colour deconvolution essentially separates the chromogen stain from the nuclei signal it is possible to turn the results into something that resembles a fluorescent image that may make it easier for people to see the results of the stain.

1. The first step is to invert the colours so that dark becomes light like in a fluorescent image. Select the nuclei and brown channels in turn and go to **Edit → Invert**

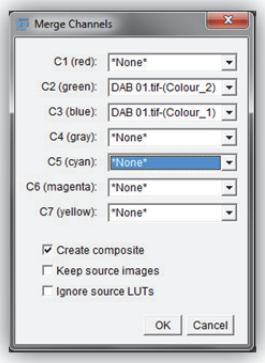




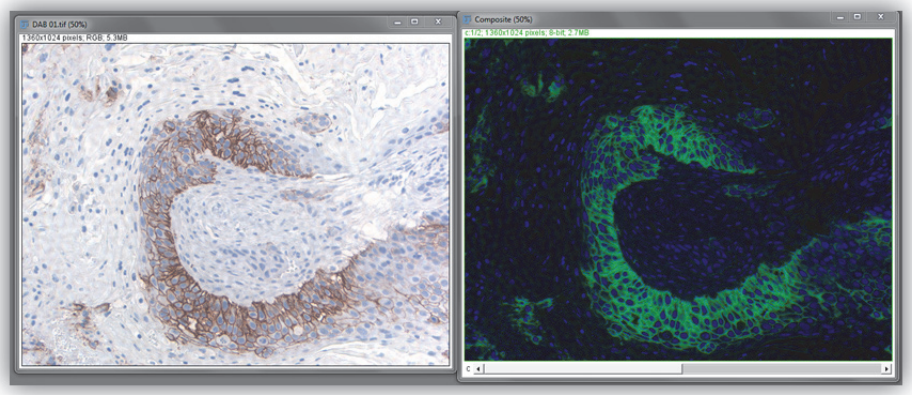
2. Now apply a **blue** LUT to the blue image and a **green** LUT to the brown image



3. These images can now be combined into a composite image like shown before



4. The result is a pseudo fluorescent image of the original DAB stain



Cell Counting

Aim

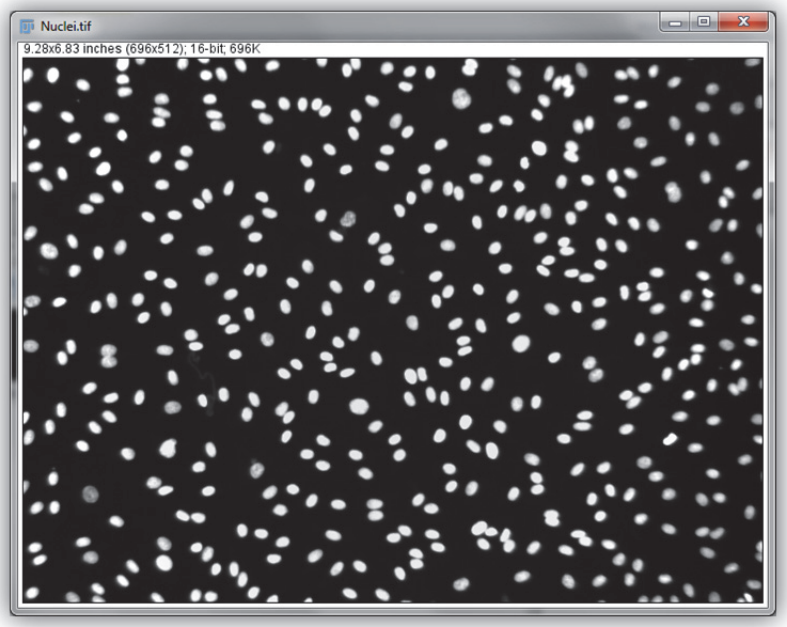
Counting fluorescently (or clearly defined brightfield) stained cells is a relatively easy process in Fiji. Information such as shape, area, intensity etc. can also be extracted on top of just a raw count number.

This example uses just a nuclear stain (like DAPI) to count the number of cells. A later technical note will show how to segment out individual whole cells and measure them.

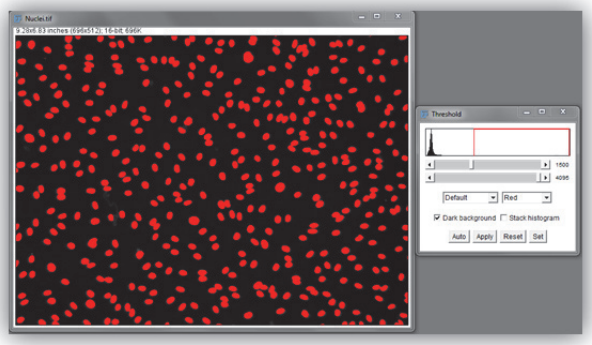
Initial Binary Creation

Before anything can be counted a binary image of it needs to be created.

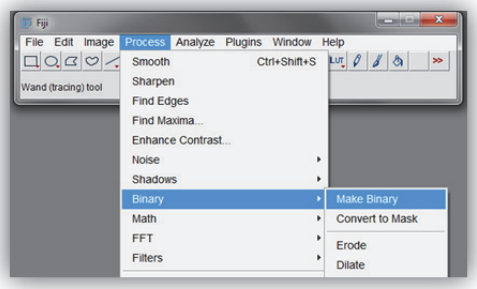
1. Open **Nuclei.tif** from the **Demo Images\Widefield\Segmentation** folder



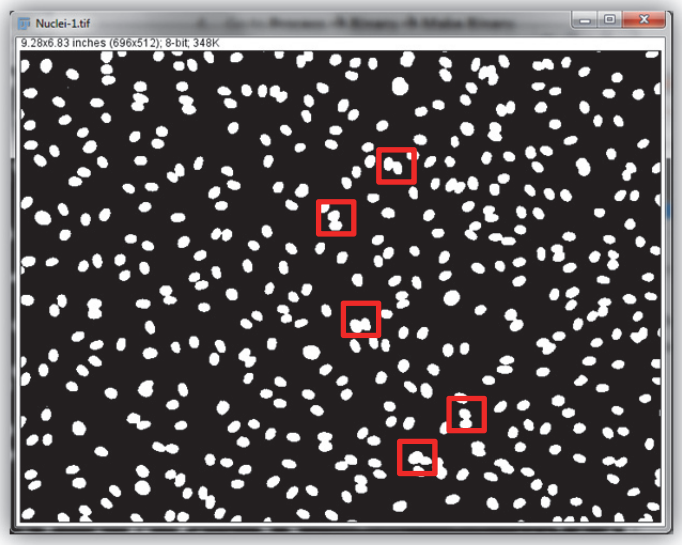
2. Create a duplicate of the **nuclei** image to work with
3. Threshold the image



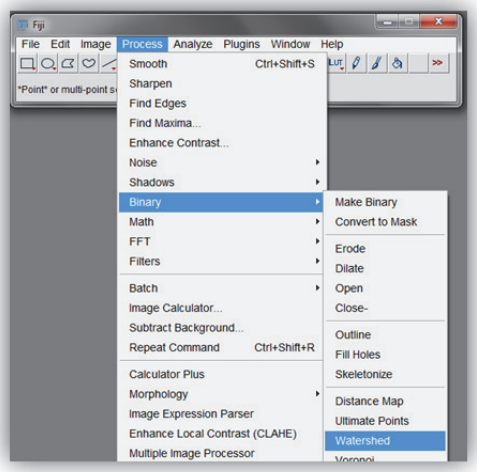
4. Go to **Process → Binary → Make Binary** or press the **Apply** button in the threshold window



5. If you look at the resulting binary image you will notice that there are several nuclei that are fused together. If we were to count this image now these nuclei would be counted as one cell, not two.

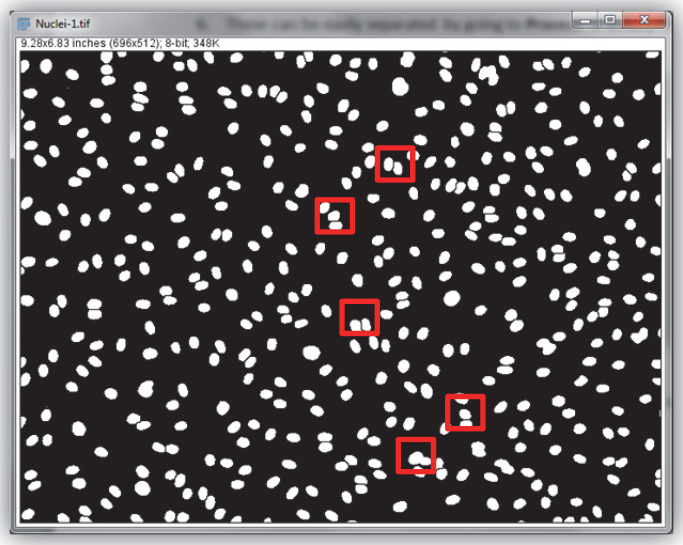


6. These can be easily separated by going to **Process → Binary → Watershed**



NOTE The binary watershed function works fine on these images as they are basically round objects joined together. The watershed function essentially draws a dividing line between the two objects. If the objects are not rounded (e.g cells with large processes) the watershed function may give a strange result.

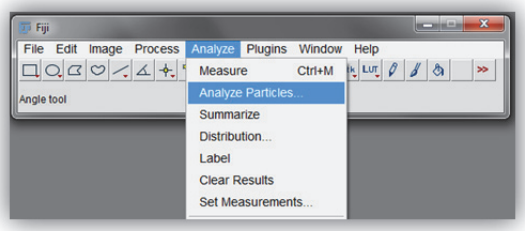
7. The result will be the majority of the joined nuclei being separated. Some will not be but this is better than nothing.



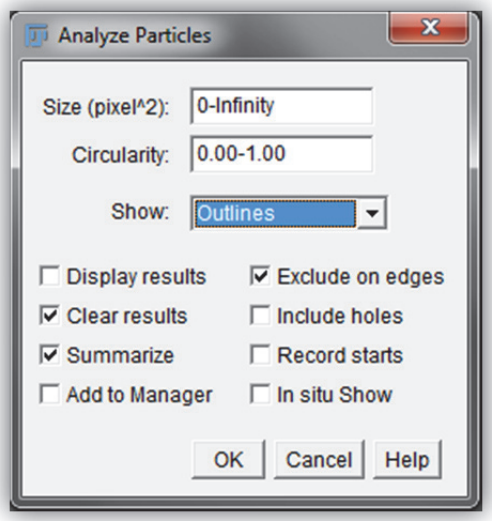
This binary image can now be used for further analysis

Counting Cells – Basic Count

1. Go to **Analyse → Analyse Particles…**

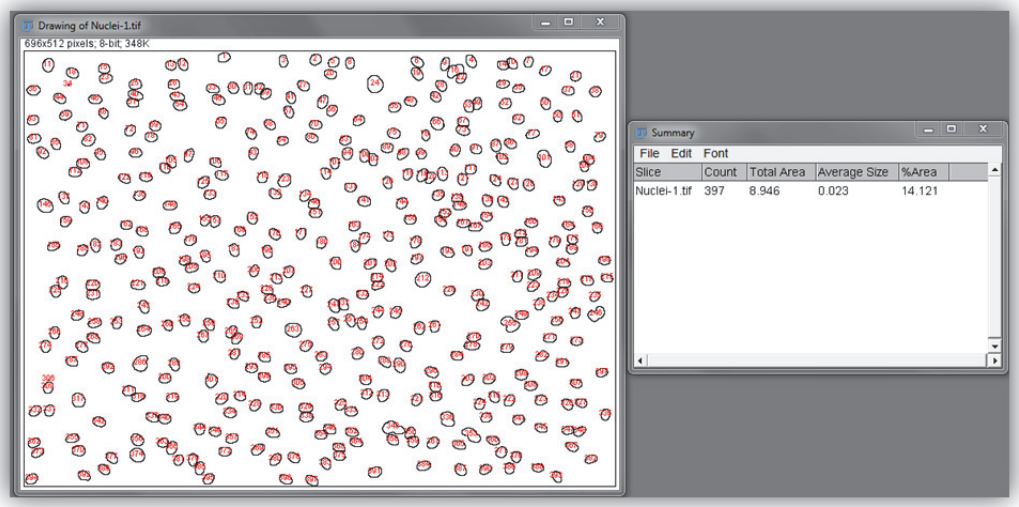


2. In the **Analyse Particles** dialog leave **Size** and **Circularity** at their default values. Set **Show:** to **Outlines** and tick the boxes as below. Press **OK.**

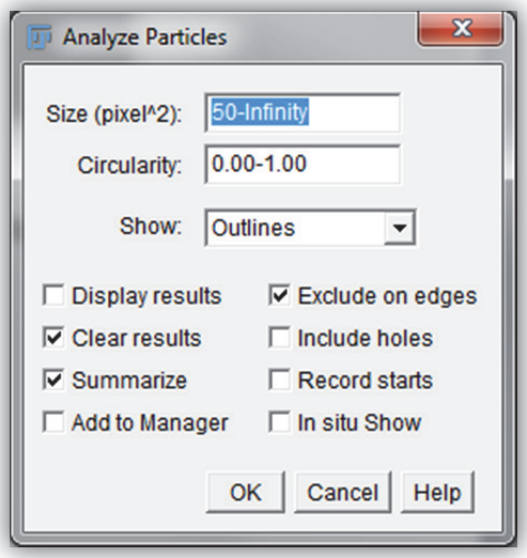


This sets the module to count everything (except for objects touching the edge of the image), display a summary table of the results and show a result image of the outlines that were measured.

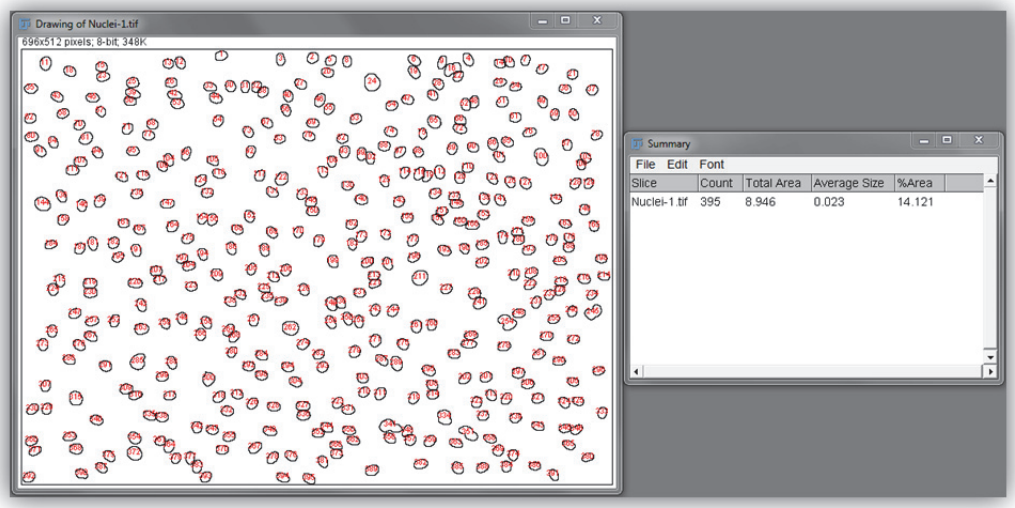
3. The results are shown in the table and as the outline image. The results show that there are 397 cells in the image, but if you look closely (for example in the top left corner) there are some things being counted that are too small to be a nuclei.



4. To filter out these results close the outline image and the summary table. Open the **Analyse Particles** module again but this time set the **Size** value to **50-Infinity**. Press **OK.**



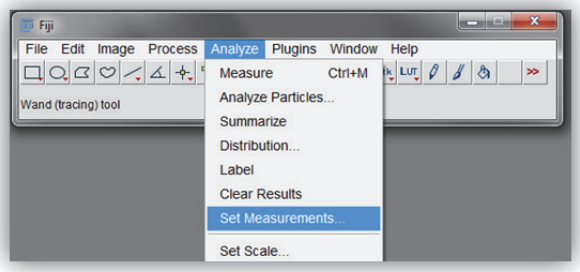
5. Now any objects smaller than 50 square pixels are not being counted and adding error to the result.



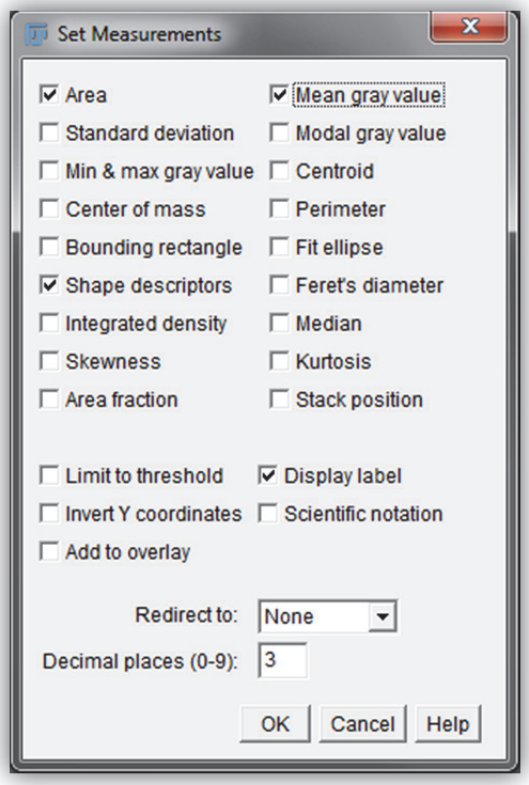
Counting Cells – Detailed Information

If you look at the outline results from the previous analysis you will notice that each out line has a unique number associated with it. This is because each object is measured individually and we can get other parameters out about it as well.

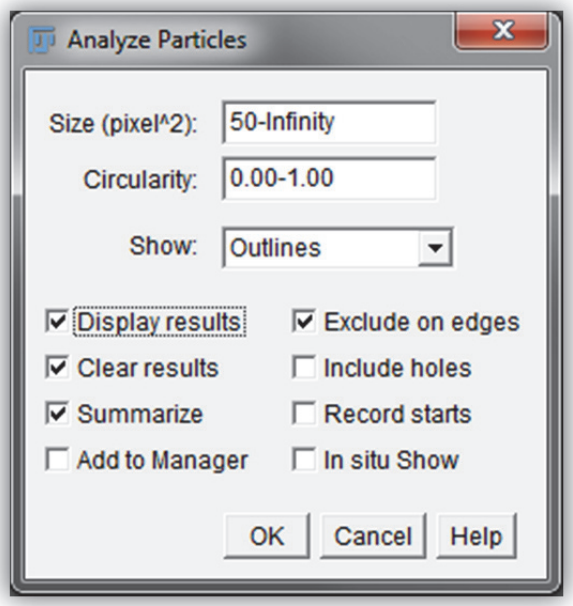
1. Close the outline image and the summary results table. Then go to **Analyse → Set Measurements…**



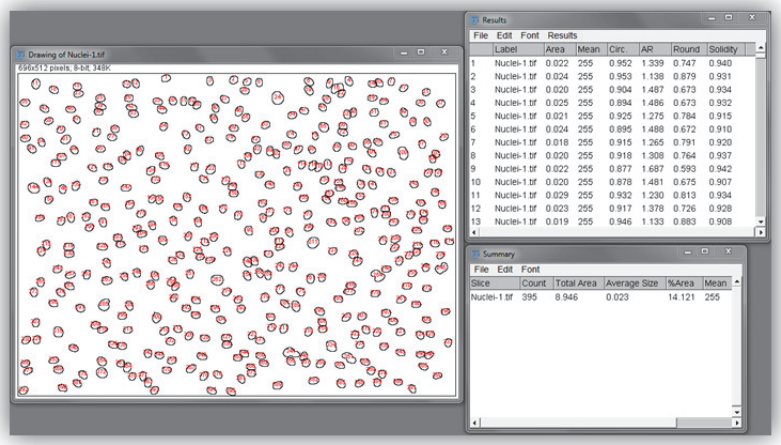
2. In this window the different measurements you want to make can be selected. For this example select the options as shown below and press **OK.**



3. Go to the **Analyse Particles** module again but this time also tick the **Display Results** checkbox and press **OK.**

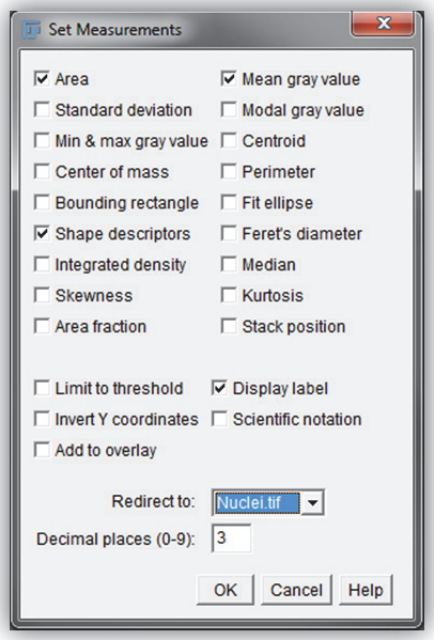


4. You will now get an extra results table with detailed information about each of the cell objects in the image. This table can be saved as a text file or exported to excel for further analysis.

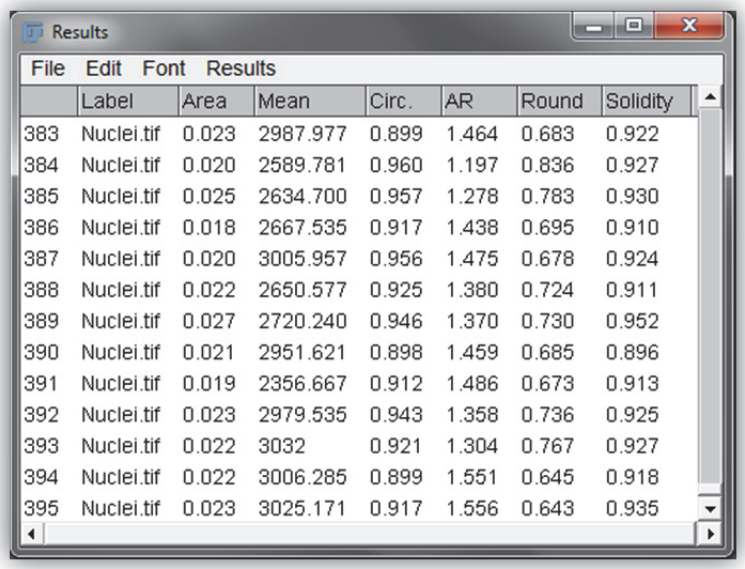


The values for **Circ, AR, Round and Solidity** describe the shape of the objects measured. For example a **Circ** value of 1.0 is a perfect circle.

5. If you look at the **Mean** column you will notice that all the objects have the same values, in this case **255** because we measured the binary image in which all objects were pure white. We can instead measure this intensity of the original nuclei image by using the binary image as a mask.
6. Close the outline mage and two results tables. Go back into **Set Measurements…** and set the **Redirect to:** value to the original nuclei image and press **OK.**

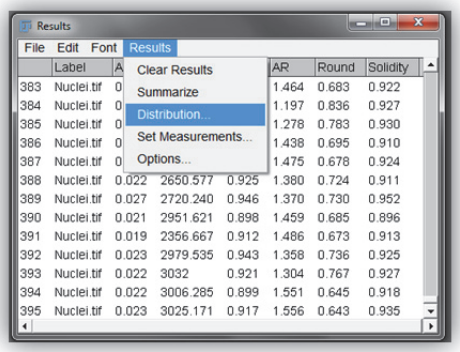


7. Re run **Analyse Particles** on the binary image. Notice that there are now different **Mean** values for each object

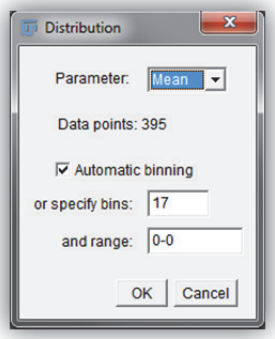


	Label	Area	Mean	Circ.	AR	Round	Solidity
383	Nuclei.tif	0.023	2987.977	0.899	1.464	0.683	0.922
384	Nuclei.tif	0.020	2589.781	0.960	1.197	0.836	0.927
385	Nuclei.tif	0.025	2634.700	0.957	1.278	0.783	0.930
386	Nuclei.tif	0.018	2667.535	0.917	1.438	0.695	0.910
387	Nuclei.tif	0.020	3005.957	0.956	1.475	0.678	0.924
388	Nuclei.tif	0.022	2650.577	0.925	1.380	0.724	0.911
389	Nuclei.tif	0.027	2720.240	0.946	1.370	0.730	0.952
390	Nuclei.tif	0.021	2951.621	0.898	1.459	0.685	0.896
391	Nuclei.tif	0.019	2356.667	0.912	1.486	0.673	0.913
392	Nuclei.tif	0.023	2979.535	0.943	1.358	0.736	0.925
393	Nuclei.tif	0.022	3032	0.921	1.304	0.767	0.927
394	Nuclei.tif	0.022	3006.285	0.899	1.551	0.645	0.918
395	Nuclei.tif	0.023	3025.171	0.917	1.556	0.643	0.935

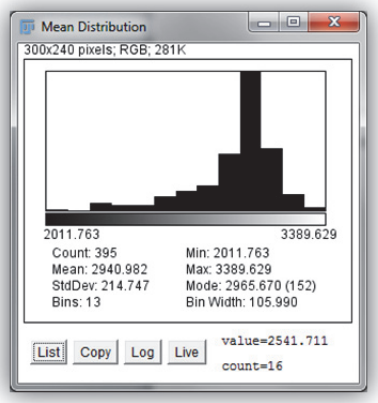
8. To view a distribution of the data select the **Results** window and go to **Results → Distribution…**



9. In the **Distribution** dialog set the **Parameter:** to what you want to graph, in this example the mean intensity, leave the other settings as default and press **OK.**



10. The result is a histogram showing the distribution of the average intensities in the sample



If you press the **List** button you will get a list that can be saved to recreate the histogram in Excel.

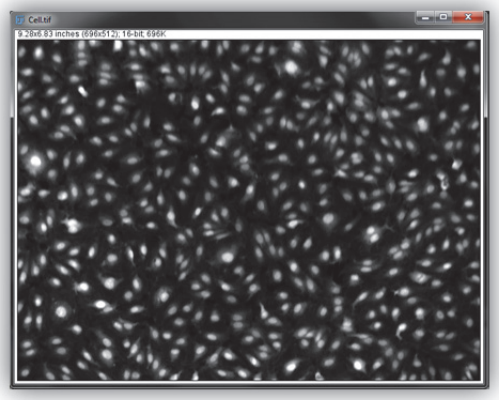
Cell Segmentation and Analysis

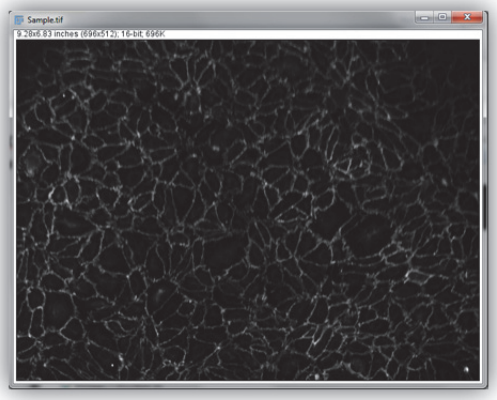
Aim

Segmentation of cells is creating masks the represent their shapes based on whole cell dyes, and sometimes nuclear dyes. These masks can be used to analyse morphology parameters such as area, shape etc. The masks can also be overlayed onto other channels of the image to measure intensities or other parameters.

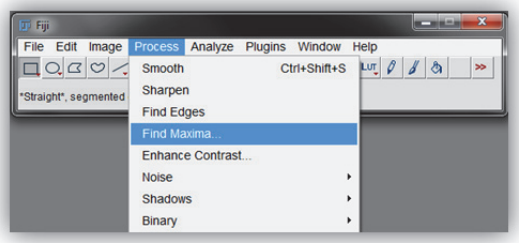
Creating Whole Cell Binary Masks

1. Open **Cell.tif** and **Sample.tif** from the **Demo Images\Widefield\Segmentation** folder. Create a duplicate of **Cell.tif**.



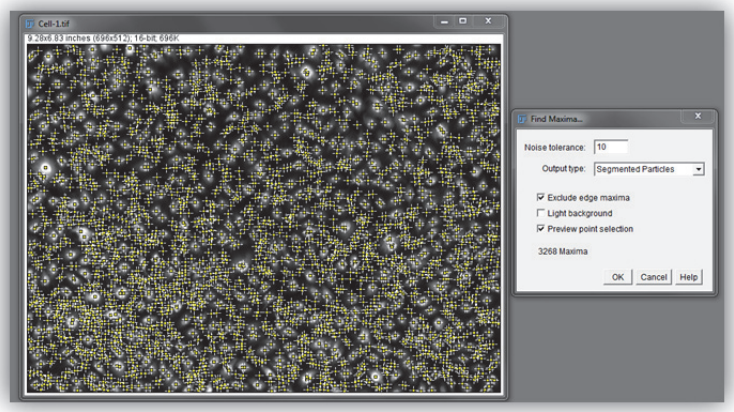


2. Select the duplicated **Cell.tif** image (usually Cell-1.tif). Go to **Process → Find Maxima…**

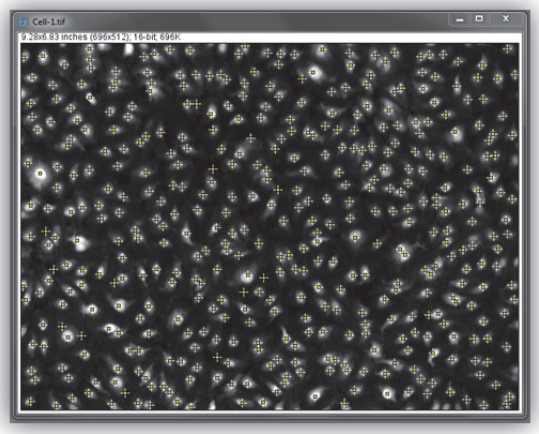


3. In the **Find Maxima** dialog box leave **Noise Tolerance** at **10**, set **Output Type:** to **Segmented Particles.** Check the **Exclude edge maxima** and **Preview point selection.** Don't press **OK** yet.

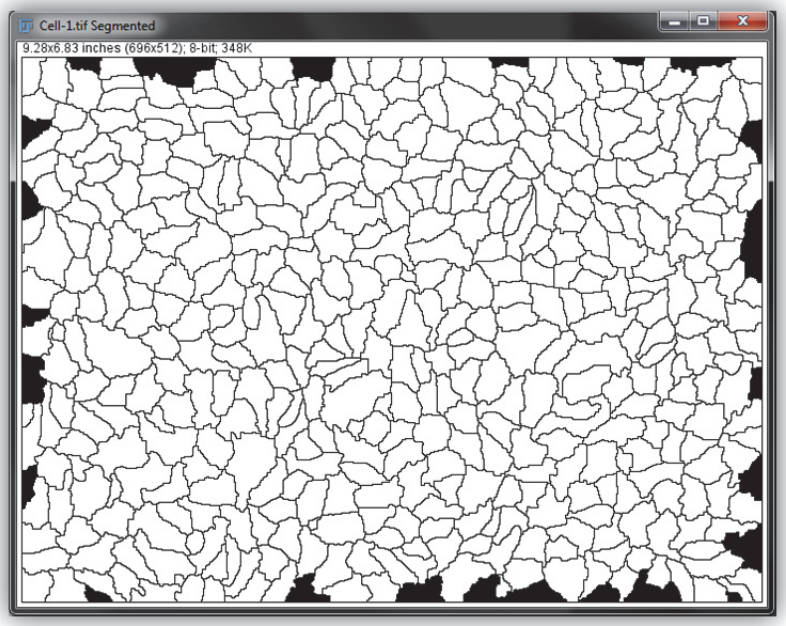
   The point is to get one maxima point per cell, to change the sensitivity the **Noise Tolerance** value needs to be increased.



4. Change the **Noise Tolerance** value until you get one maxima per cell. A value of **400** works well for this image. Press **OK.**

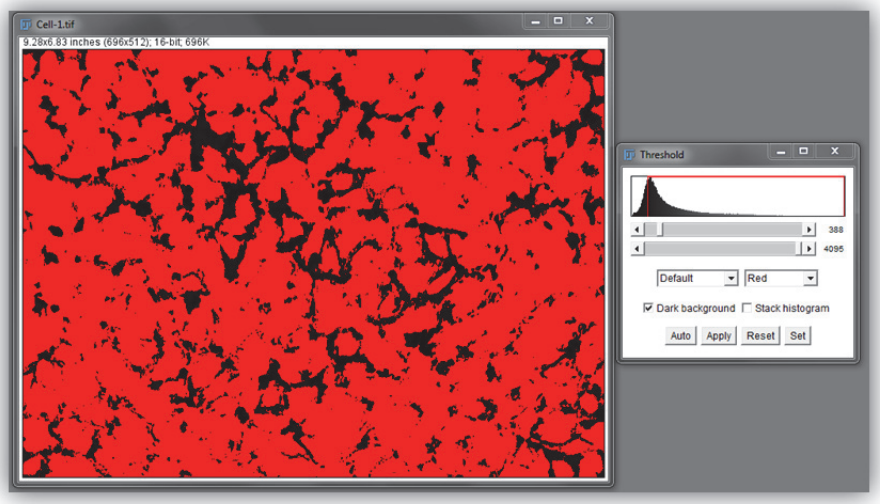


5. The result is a watershed binary mask of the cell image. The lines between the “cells” represent the lowest intensity point between each of the cells based on the seed point of the maxima point set before.



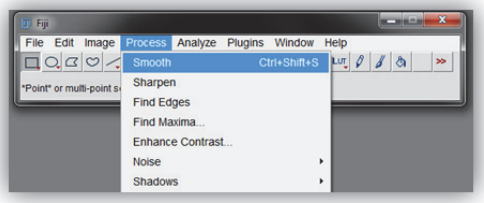
6. This mask can be used to create outlines of the individual cells. To be able to do this we need to also create a binary mask that represents the stain for the whole cells.

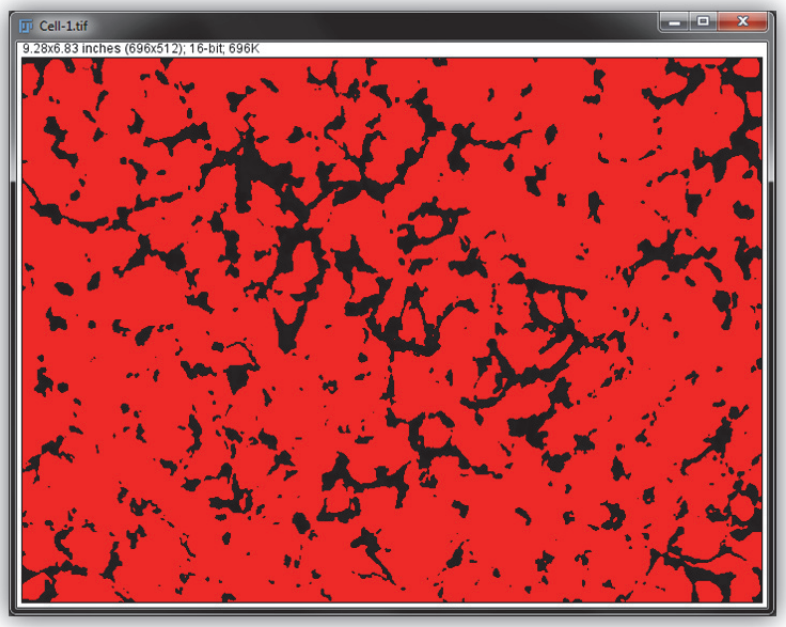
   Select the duplicated cell image and set a threshold on it **but do not apply it yet**



7. You will notice that the threshold is a little “shaggy” in places. For good quality cell segmentation it is better to have a smooth mask. The easiest way to do this is to smooth the cell image.

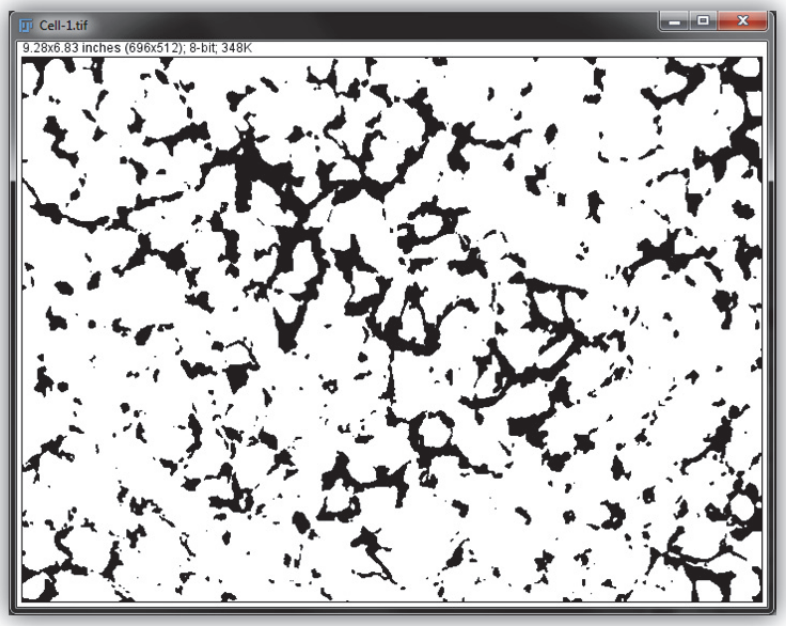
   Go to **Process → Smooth**



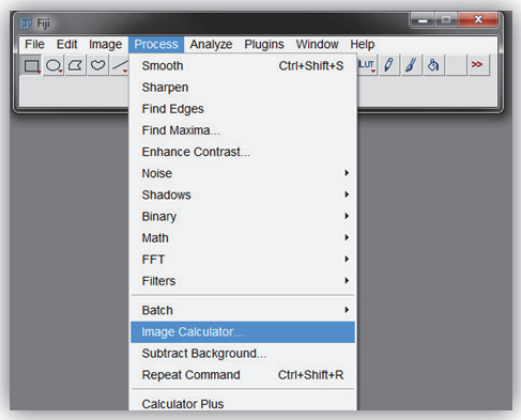


   The result smooths out some of the noise in the image and makes the threshold cleaner.

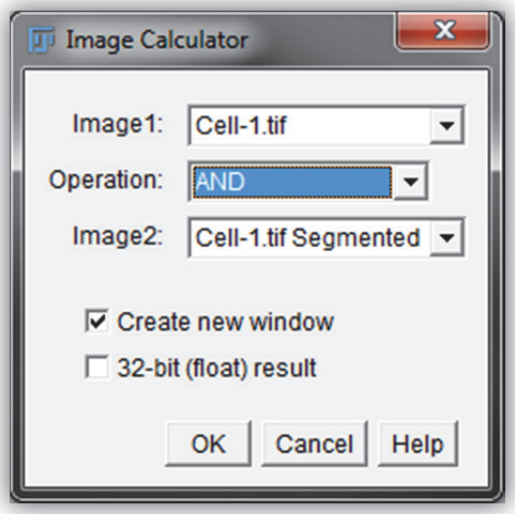
8. Apply the threshold to give a binary image of all the cell stain.



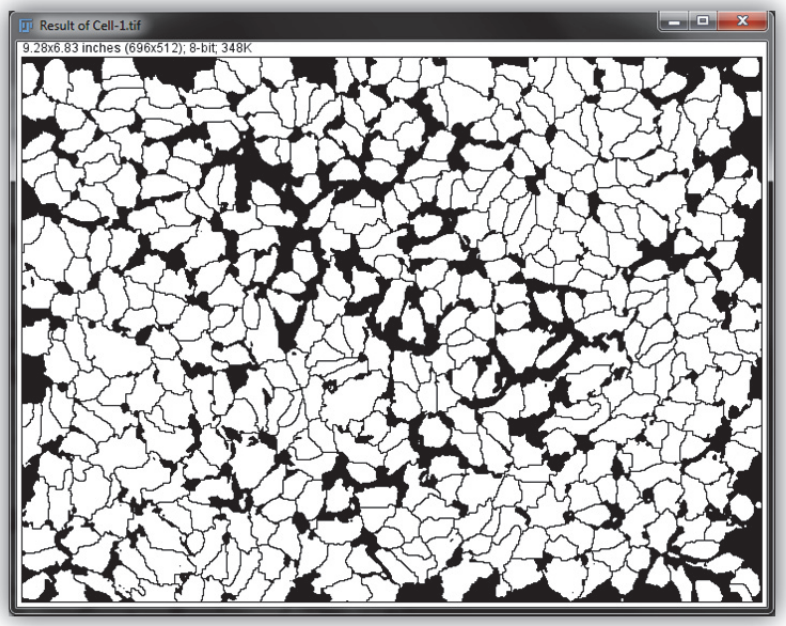
9. To get masks that resemble cell all we need to do is some simple Boolean logic between the two binary images. To do this go to **Process → Image Calculator…**



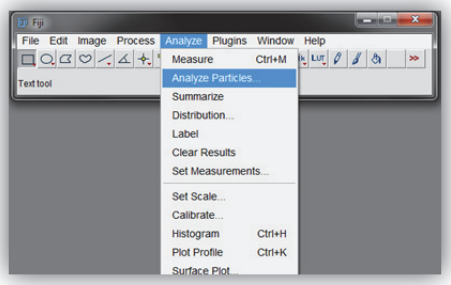
10. In the **Image Calculator** window set **Image 1:** and **Image 2:** to the two binary masks. Set the operation to **AND**. Tick the **Create new window** box and press **OK**.



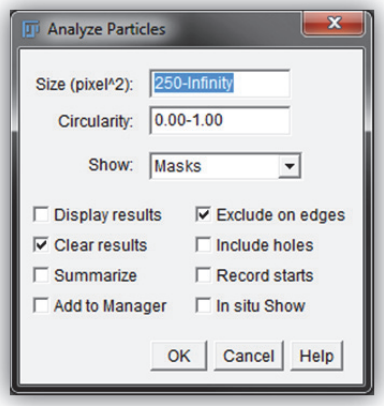
11. The resulting image is now looking like the outlines of cells but still needs to be cleaned up some.



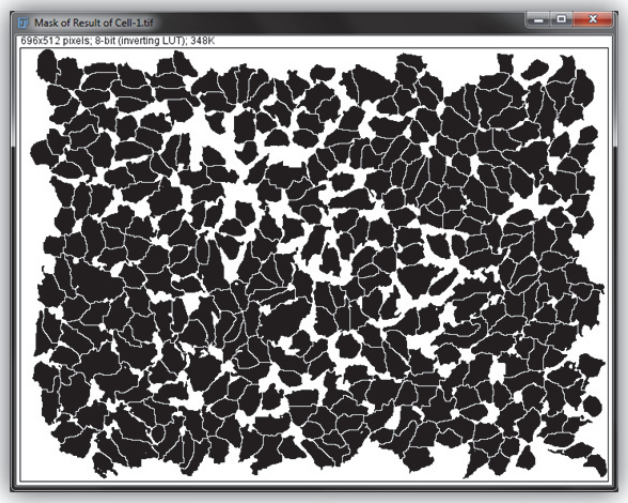
12. To clean up the cell masks use the **Analyse Particles…** module.



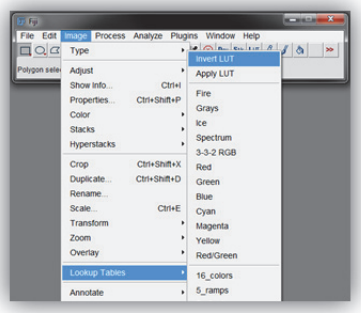
13. To remove the smaller chunks in the image set the **Size** to **250-Infinity**, set the **Show:** value to **Masks,** tick the **Exclude on edges** box and press **OK.**

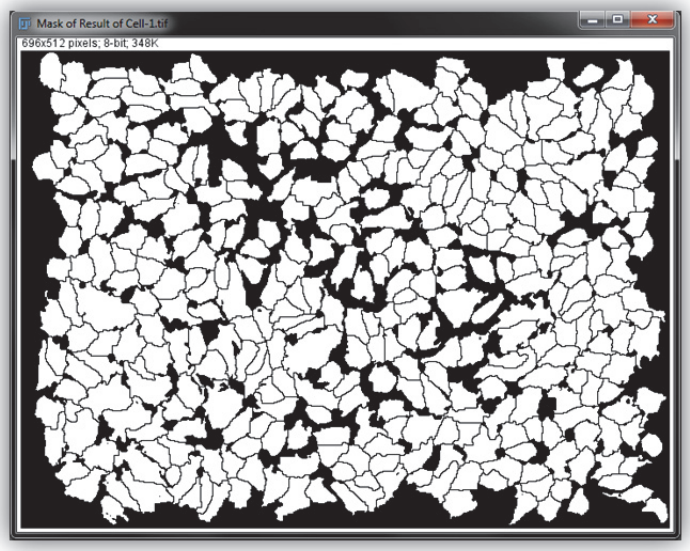


14. The resulting mask will look a bit funny. This is because the LUT is inverted.

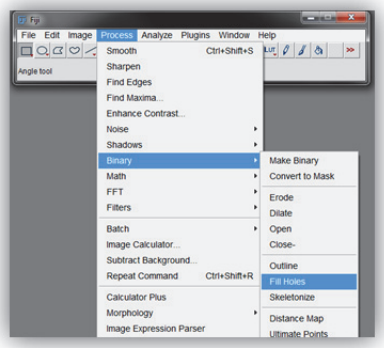


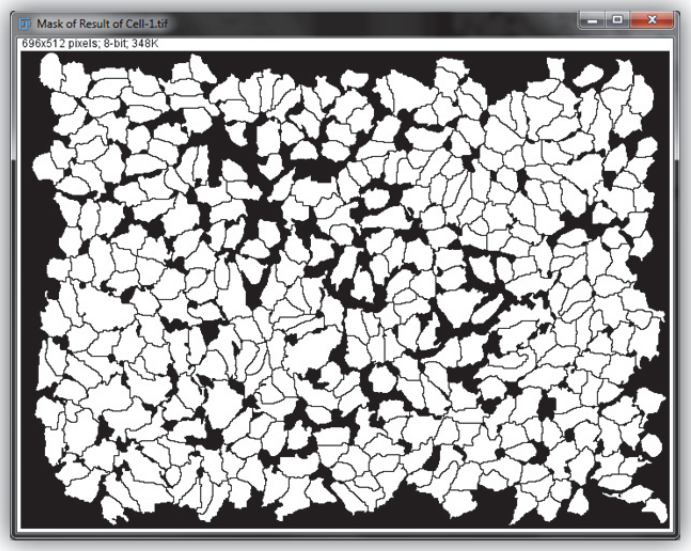
15. To turn the LUT inversion off go to **Image → Lookup Tables → Invert LUT**





16. If you look at some of the cells you will notice that they have small black holes in them. These can be filled by going to **Process → Binary → Fill Holes**



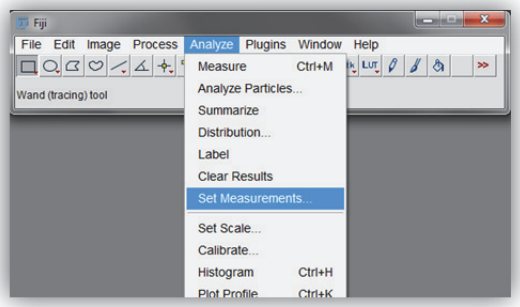


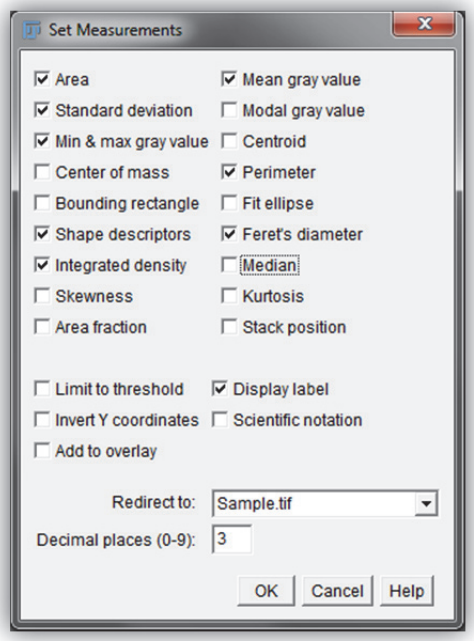
17. This final mask can be used for further analysis. The cell shapes should correspond to the outlines of the cells, how to check this will be shown later.

Analysing the Image

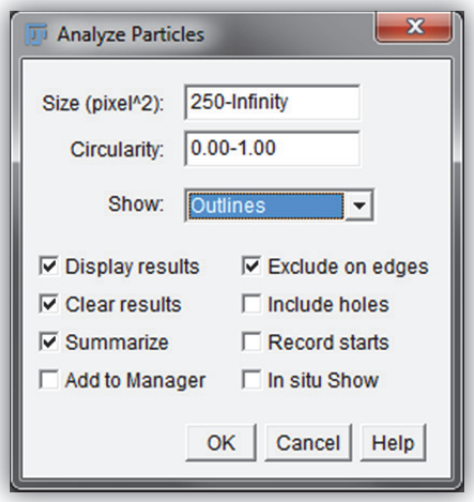
1. To analyse the **Sample.tif** image we need to redirect the measurement to use the mask on the **Sample.tif** image as we did previously.

   Go to **Analyse → Set Measurements…** and configure it as below and press **OK**

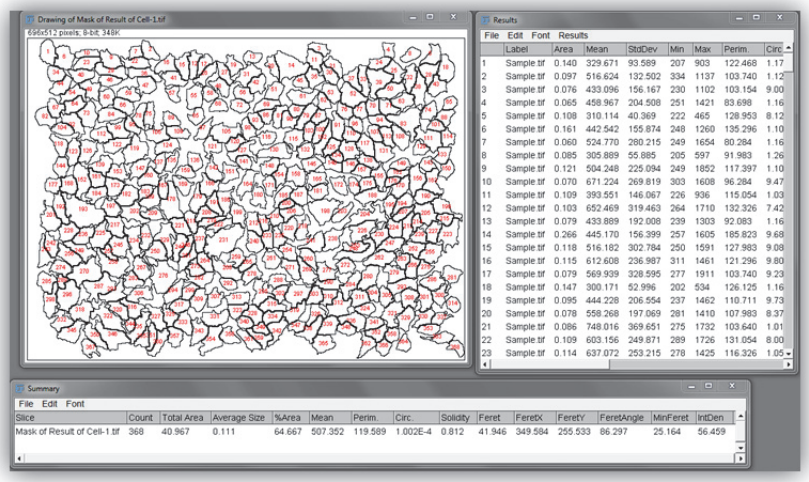




2. Use the **Analyse Particles…** module as below to measure the cells.

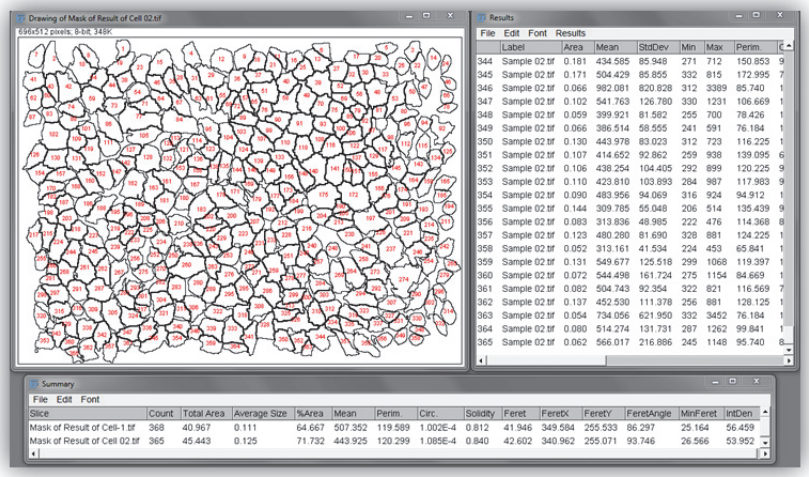


3. The results will be displayed as the outline image, summary table and detailed results table. Each of these can be saved for further analysis and presentation.



Save the image and the two tables as we will be using these for comparison to another data set.

4. Re run the mask creation and analysis on **Cell 02.tif** and **Sample 02.tif** from the **Demo Images\Widefield Images\Segmentation** and compare the results.



You will notice when looking at the summary data that in general the samples are the same. But the average size of the cells in Sample 02 is a little bit bigger and the average intensity is lower.

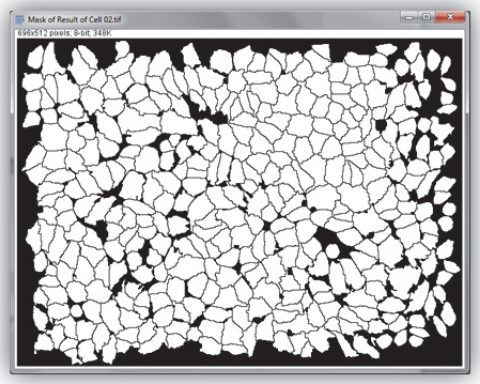
Checking the Effectiveness of Segmentation

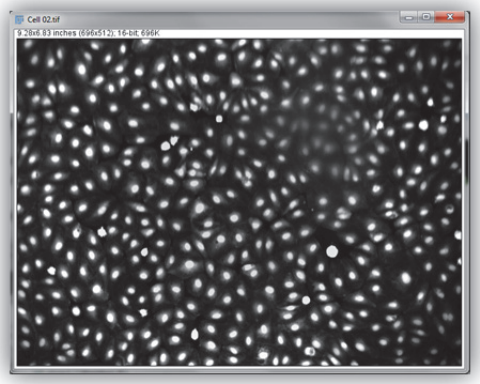
After the last analysis you should have quite a few windows open. Close all of them except for the final binary mask of the cells. We can use this mask to create outlines to check if the segmentation is accurate.

There are two methods for doing this. One involves overlaying the measurement ROIs onto the image and check them. This will also give you a list of ROIs that can be saved for other uses but can be a bit cluttered and not amenable to presentation. The second is to make an outline of the cell based on the mask and create a composite colour image of the outline and cell stain. This methods gives a picture that is easier to present or show people what has happened.

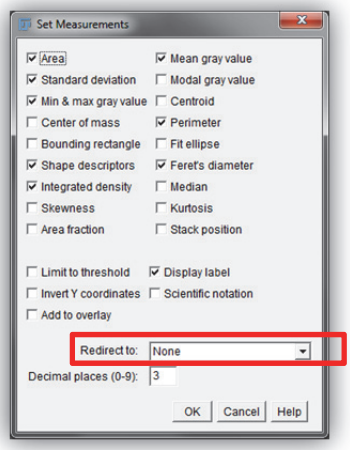
Method 1 – ROI Manager

1. Re open **Cell 02.tif** from the **Demo Images\Widefield\Segmentation** folder.

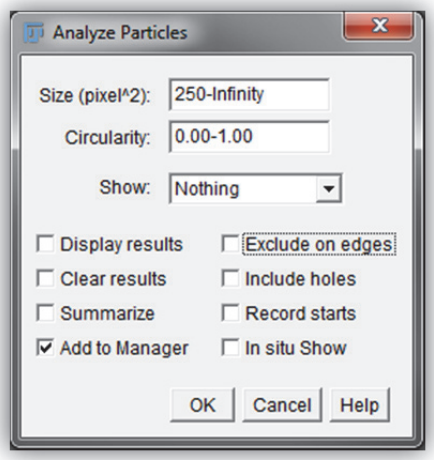




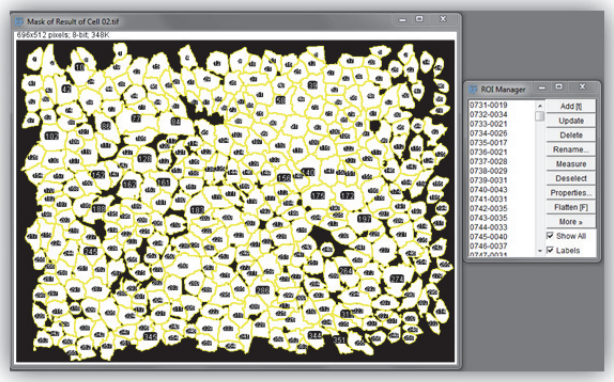
2. Go into **Set Measurements** and make sure the **Redirect to:** option is set to **None** and press **OK.**



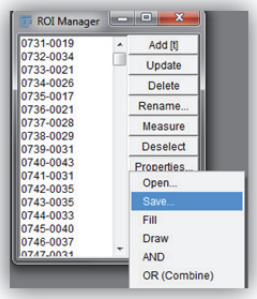
3. Select the binary mask image and go to **Analyse Particles** again. This time configure it as below and press **OK.**



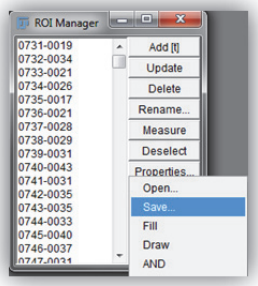
4. The result will be yellow outlines around each of the objects in the mask and a list will open up in the **ROI Manager.**



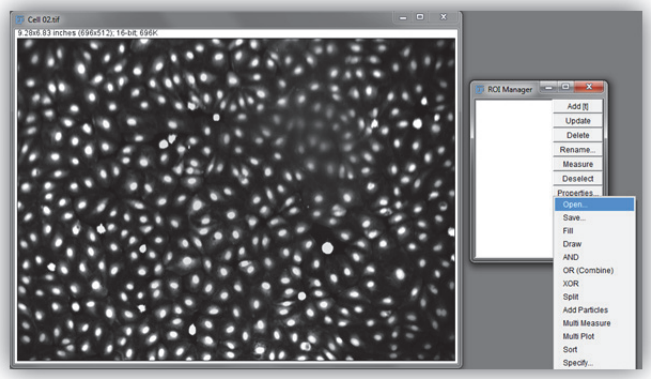
5. The ROIs can be saved and reapplied to another image, say the **Cell 02.tif** image. To save the ROIs click the **More>** button in the **ROI Manager** and select **Save.**



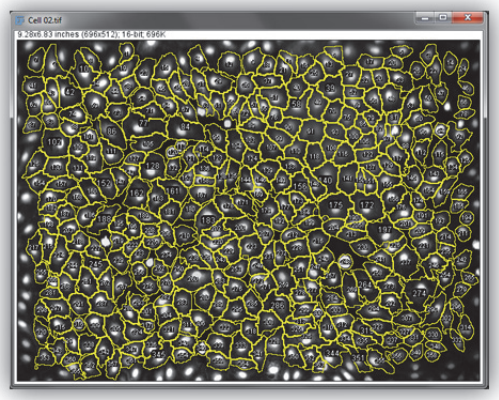
6. Save the ROI file somewhere you can find it again.

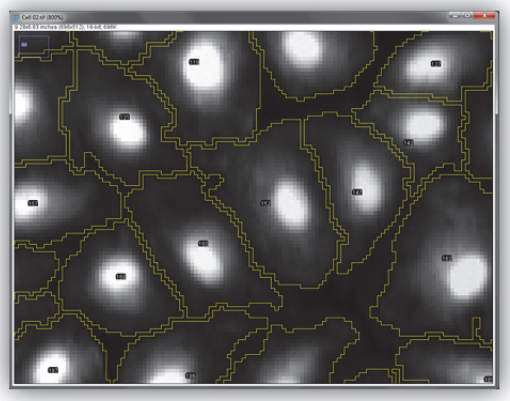


7. We don't need the ROIs on the mask image anymore so they can be deleted by pressing the **Delete** button in the **ROI Manager**.
8. Now select the **Cells 02.tif** image, press the **More>>** button in the **ROI Manager** and **Open** the ROI set you just saved.



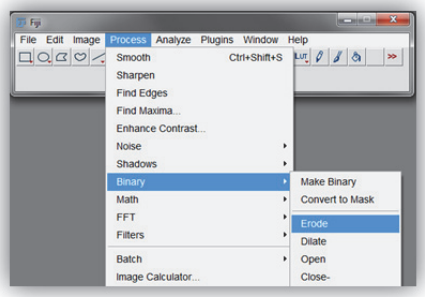
9. You will now have the ROIs loaded onto the cell stain image. You can now zoom in and see if the lines match up with the edge of the cells. You also have these regions saved if you ever need them again for other analyses.



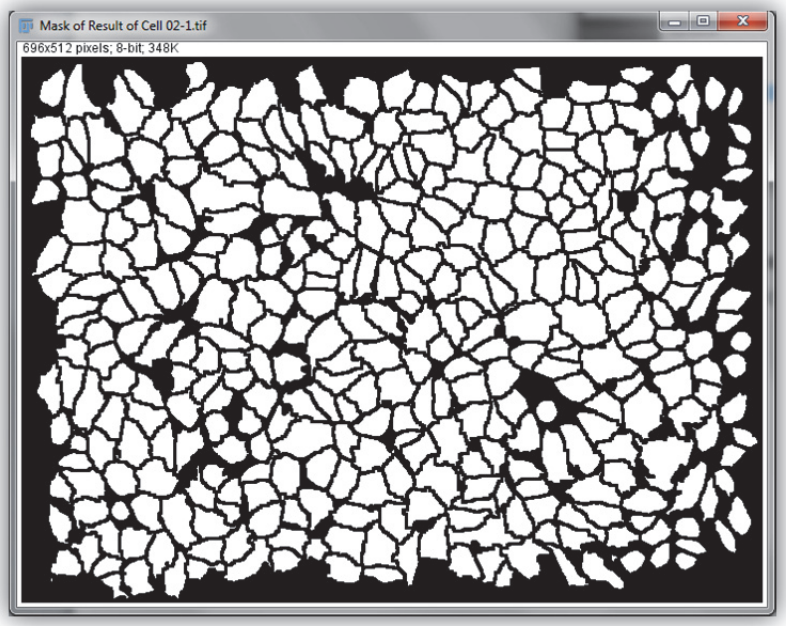


Method 2 – Create Colour Overlays

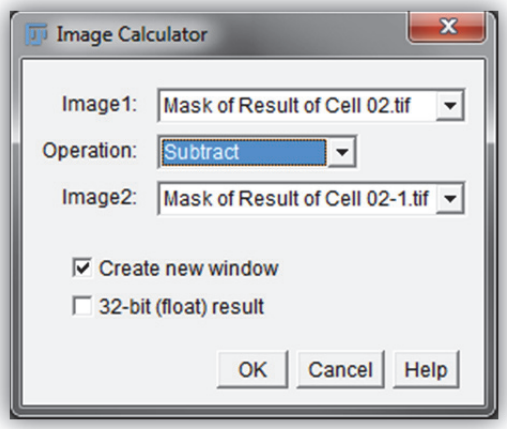
1. Delete the ROIs from the **Cell 02.tif** image. Select the binary mask image, duplicate it and go to **Process → Binary → Erode**



2. The result will be a version of the binary mask that has been eroded around the edges. This can now be used with the other binary mask to create outlines.

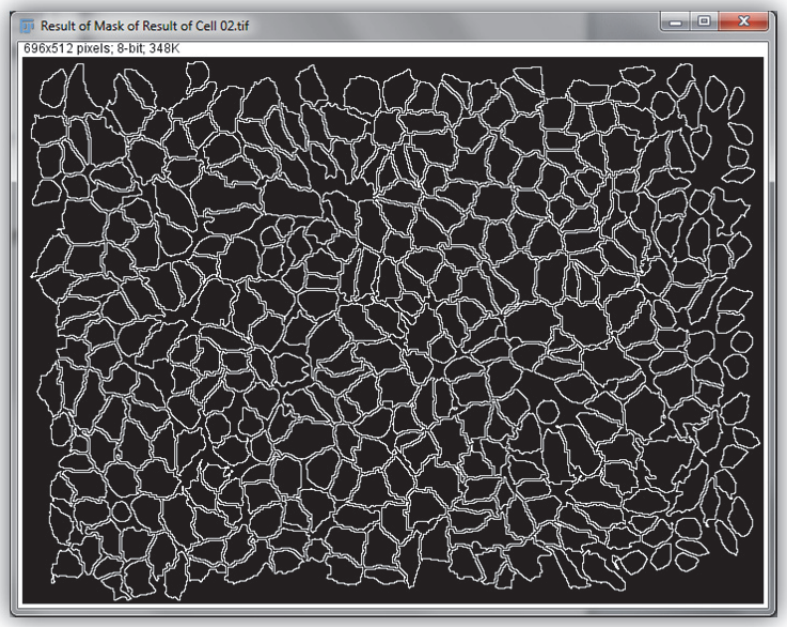


3. Use the **Image Calculator** to **Subtract** the eroded image from the original mask.



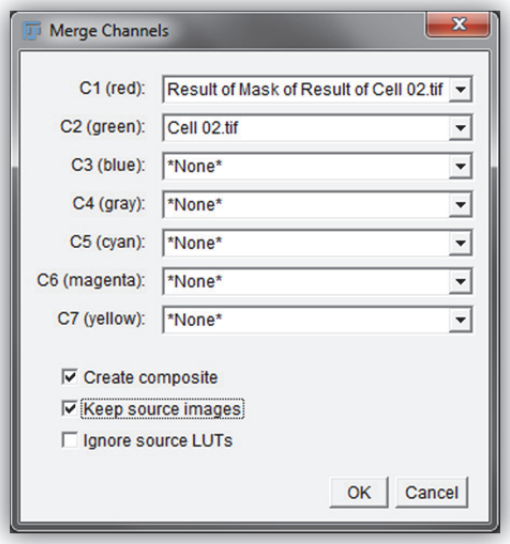
NOTE: Make sure you have the original mask image (the "bigger" one) set as **Image 1** otherwise the subtraction will not work.

4. The result is rings that represent the outside of the cell mask.

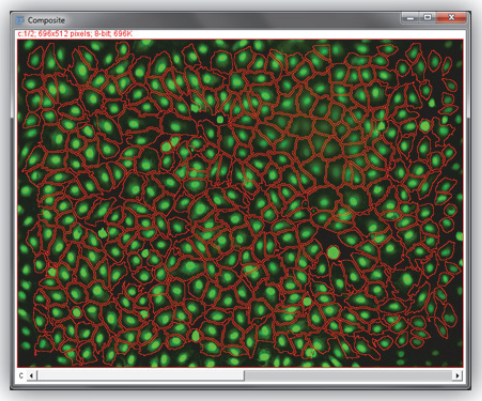


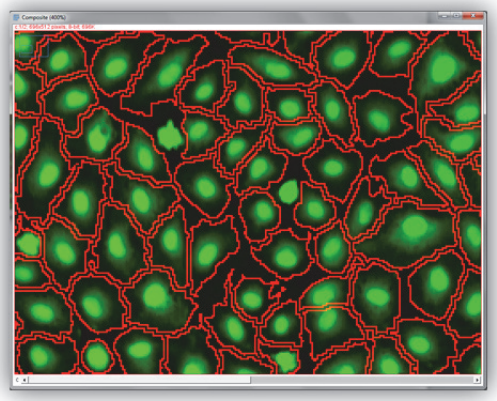
NOTE: This mask could be used as a measurement mask to analyse membrane expression.

5. Convert **Cell 02.tif** to a 8 bit image and use the merge channels function to combine the cell stain image and the rings.



6. You now have a colour composite image of the result for display and other uses.

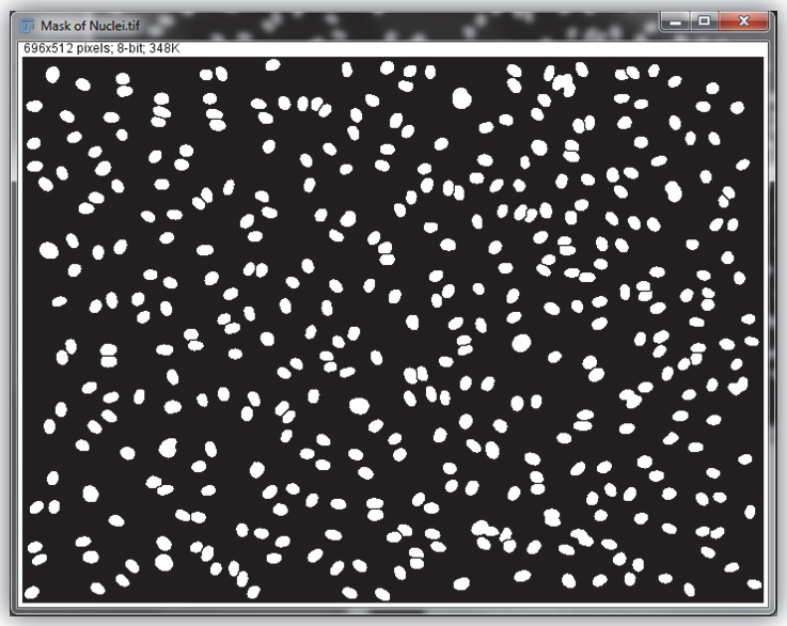




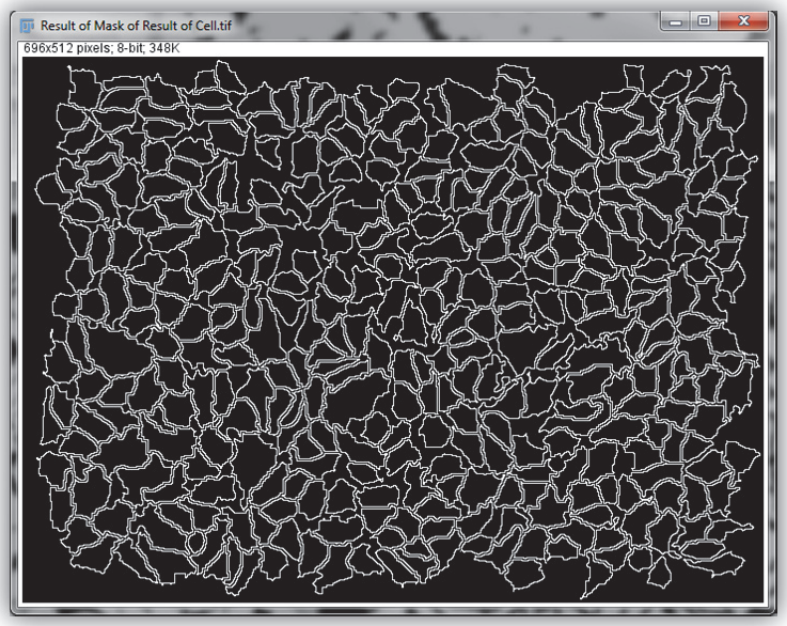
Complex Segmentation

Using what was shown in the above example and in the cell counting example it is possible to create masks that represent different cellular compartments.

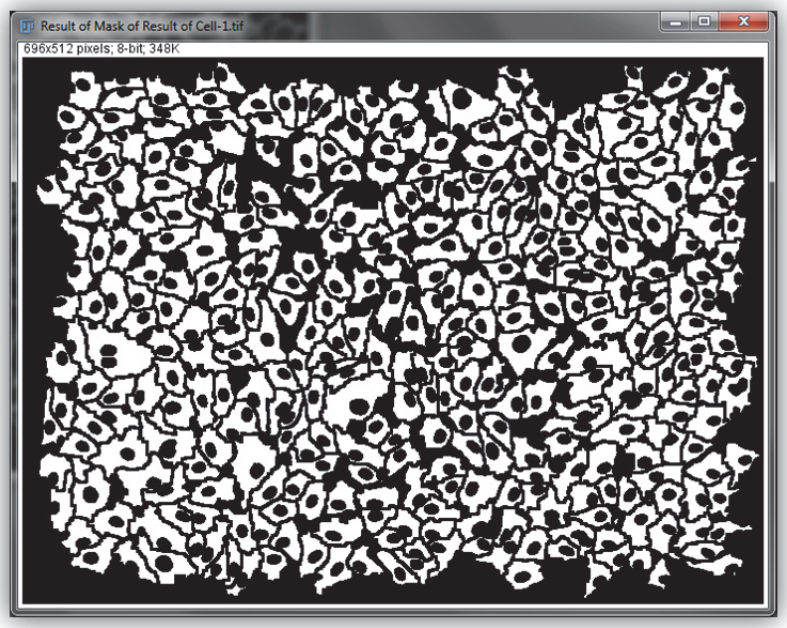
Using the cell counting example we can create a mask of the nuclei



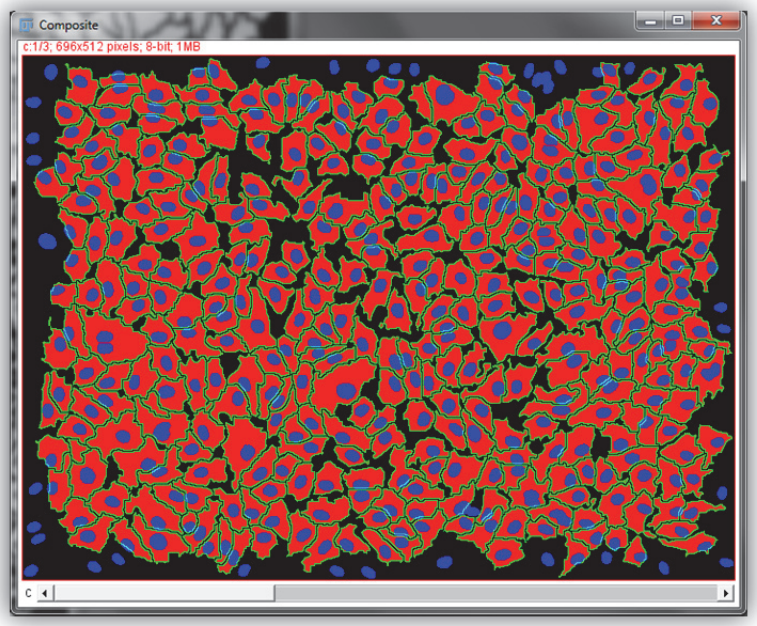
We can create a mask of the membrane of the cell using the ring method above



Now if we subtract the nuclei mask from the eroded mask used to create the rings above we are left with a mask that represents the cytoplasm of the image.



We now have three separate masks that can be used to measure the membrane, cytoplasm and nuclei of the sample image, for example to measure nuclear translocation.



Refinement of Compartment Masks

If you look at the above final mask set example there are two things wrong with it. One, there are nuclei around the edge of the image that are not associate with a cell mask. This is because the step to remove objects in the analyse particles module has removed the edge cell masks. Secondly if you look very closely there are some bits of nuclear mask that overlap with membrane mask. When used to measure the final image these masks would result in some pixels being measure twice and being classified as both membrane and nuclear.

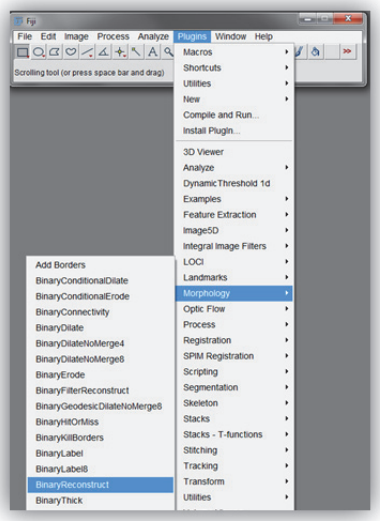
Both of these problems can be easily fixed.

Removing Extra Nuclear Masks

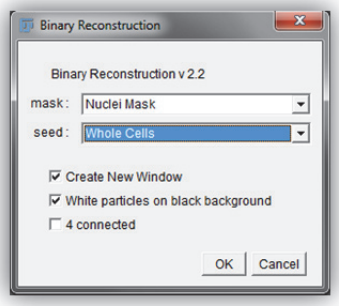
This is simple enough to do but it does require an extra plugin that is not part of the Fiji default set. This plugin is already installed on the computers you are using and there are instructions in the front of this manual that tell you how to add it to your own copy.

All we need to do is reconstruct the nuclei mask to keep only the nuclei associated with cell masks.

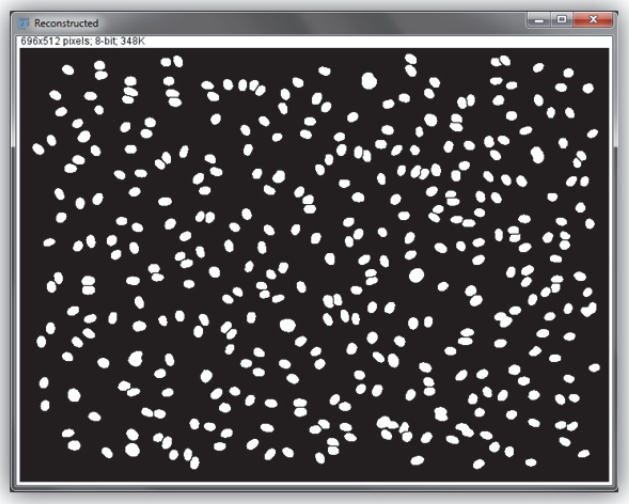
1. To do this go to **Plugins → Morphology → BinaryReconstruct**



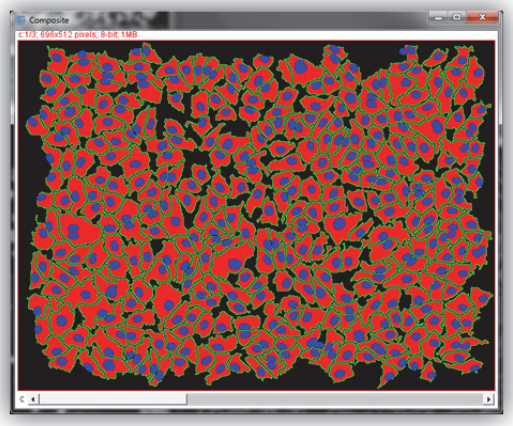
2. Configure the **Binary Reconstruction** window as below and press **OK.**



3. The result is a nuclei mask with only nuclei that overlapped with a cell being kept.



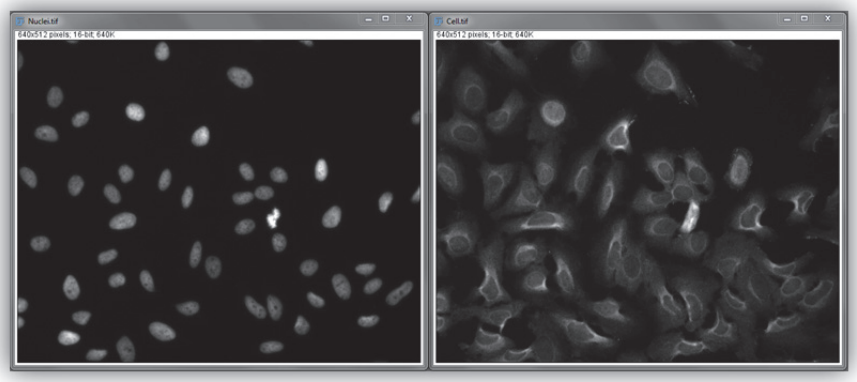
4. This mask can then be subtracted from the ring mask to remove any double measurements leaving you with a more accurate set of masks.



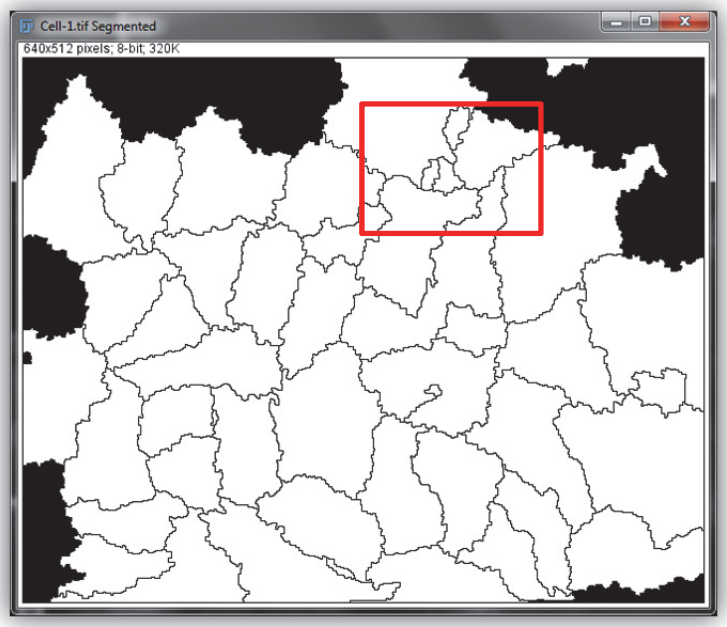
Working with Tricker Stains

Sometimes the dyes used are not the best for doing cell segmentation but it is possible to combine several dyes to achieve a good result.

1. Open **Cell.tif** and **Nuclei.tif** form the **Demo Images\Widefield\Segmentation\Cytoplasm**

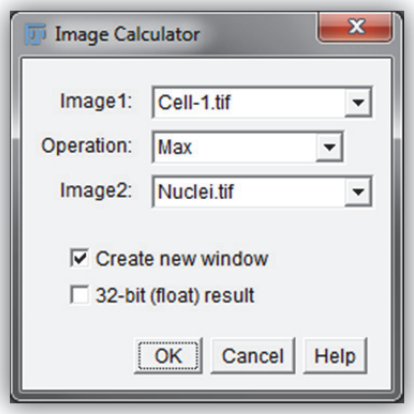


2. If you try to use the **Find Maxima** water shedding on the Cell image you will find that it will not work very well as each cell has multiple bright points in it. There will be a few broken chunks of cell and the edge detection may not be very accurate.

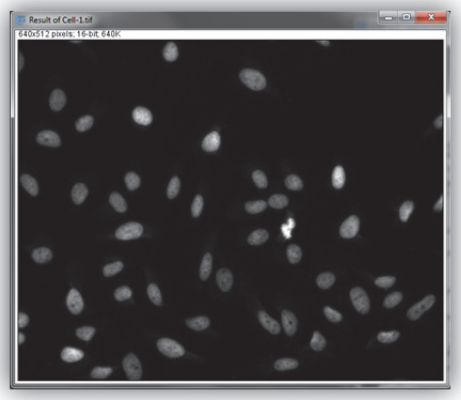


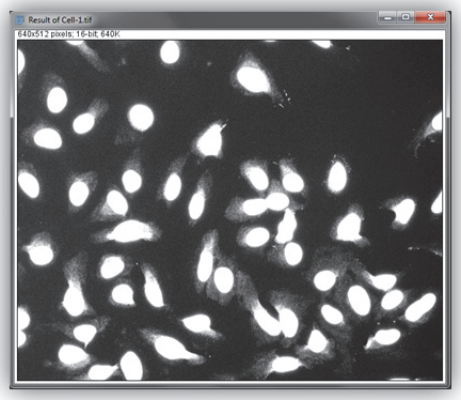
3. This can be easily fixed by combining the nuclear and whole cell stains together to give a more complete image for segmentation.

   To avoid issues with oversaturation it is best to use the **MAX** command in the image calculator and not the **Add** command.



4. The resulting image won't look very much different to the original nuclei image as the cell stain image is not as bright. But if you boost the brightness you will be able to see whole cell stain with the nuclear stain.





5. The resulting image can now be used like the previous examples to generate masks for further analysis.

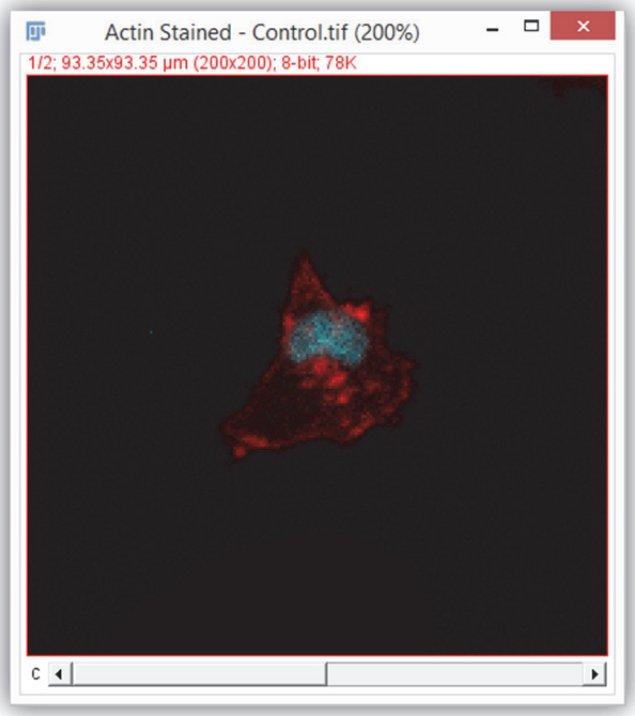
Cell Morphology and Shape Change

Aim

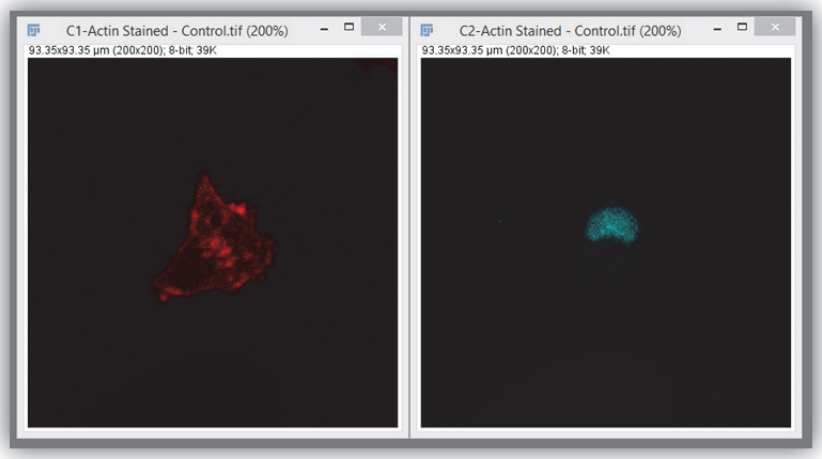
By combining several parameters you can generate outputs that can provide insight into what is happening to your sample beyond what the base measurements will show. In this example we will use the ratio of the perimeter of a cell to its convex hull to represent the complexity or “spikiness” of the cell. If the convex hull and perimeter are the same value, the ratio will be 1 and would represent a smooth cell. As the cell becomes more complex or spiky the ratio between the two deviates more and more. Additionally several intensities measures (Standard deviation, Skewness and Kurtosis) will be measured to look at changes in actin staining. In the advanced module of this manual there is a section on automating this process to batch analyse multiple cells.

Measuring Cell Complexity

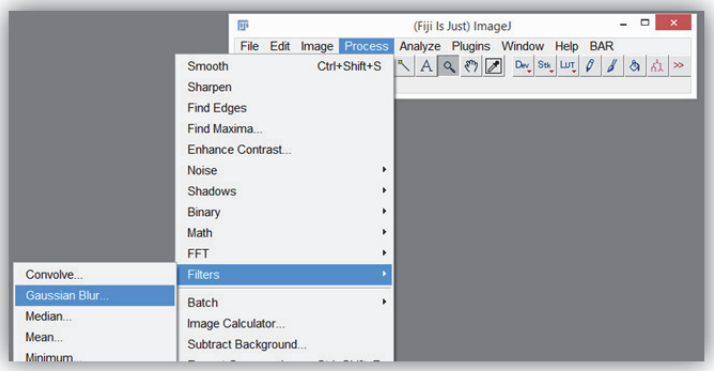
1. Open **Actin Stained – Control.tif** from the **Demo Images\Widefield\Cell Complxity** folder



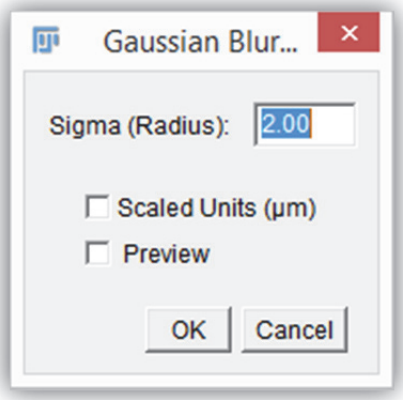
2. Separate the channels (**Image → Colour → Split Channels**)



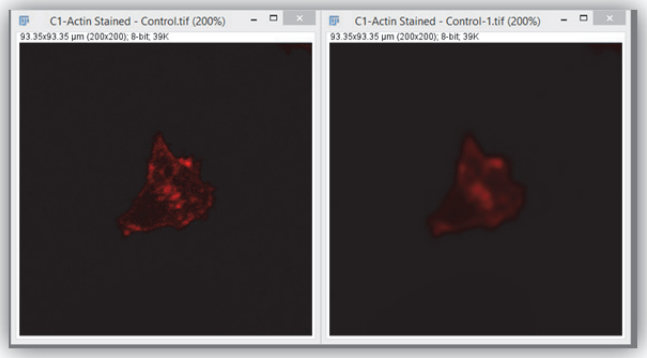
3. Make a copy of the red actin channel (right click and choose **Duplicate**). Select the duplicate and go to **Process → Filters → Gaussian Blur…**



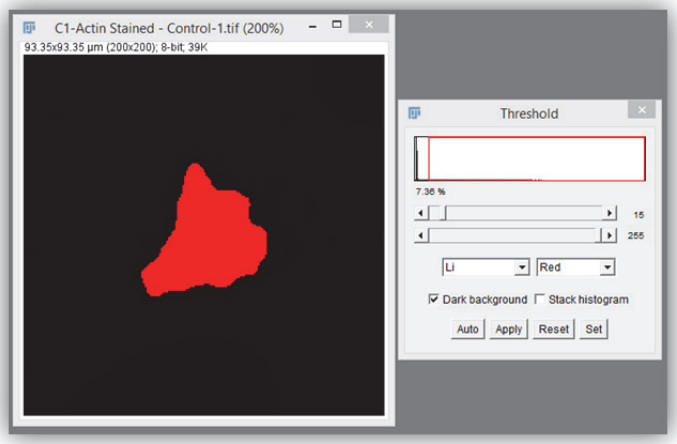
4. In the dialog that comes up enter a **Sigma** value of **2** and press **OK**

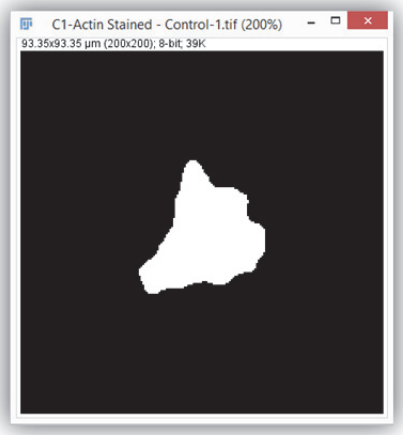


5. You will now have two versions of the actin channel .One original and one blurry. The blurry one will be used for the next couple of steps.

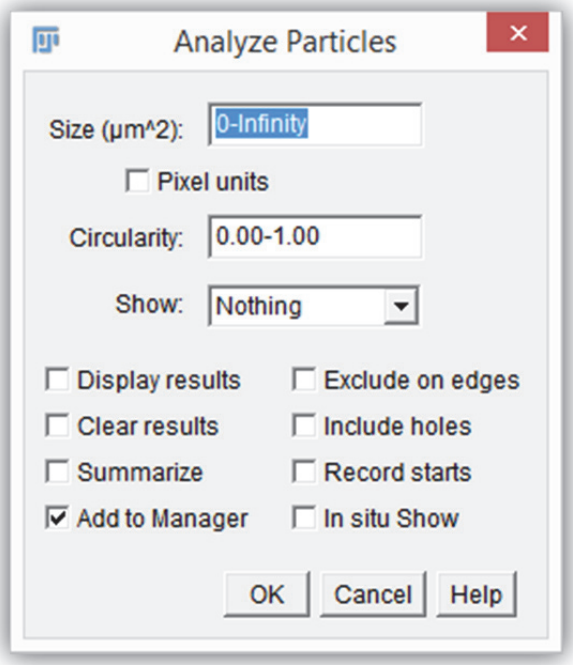


6. Select the blurred image and apply a threshold (**Image → Adjust → Threshold**) using the **Li** auto setting to create a binary image.

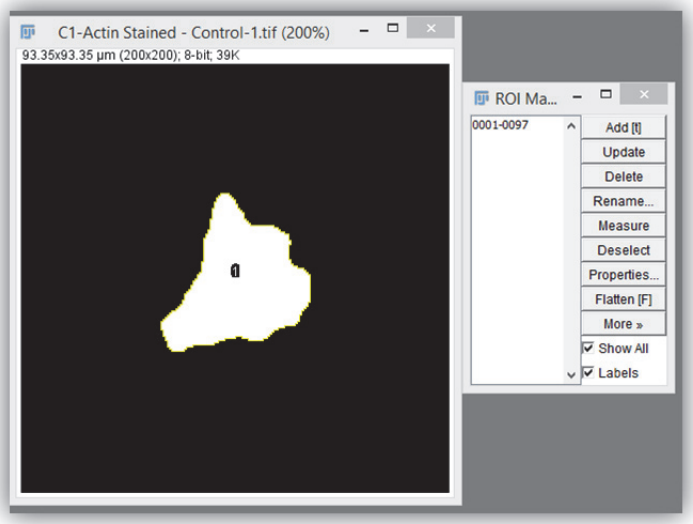




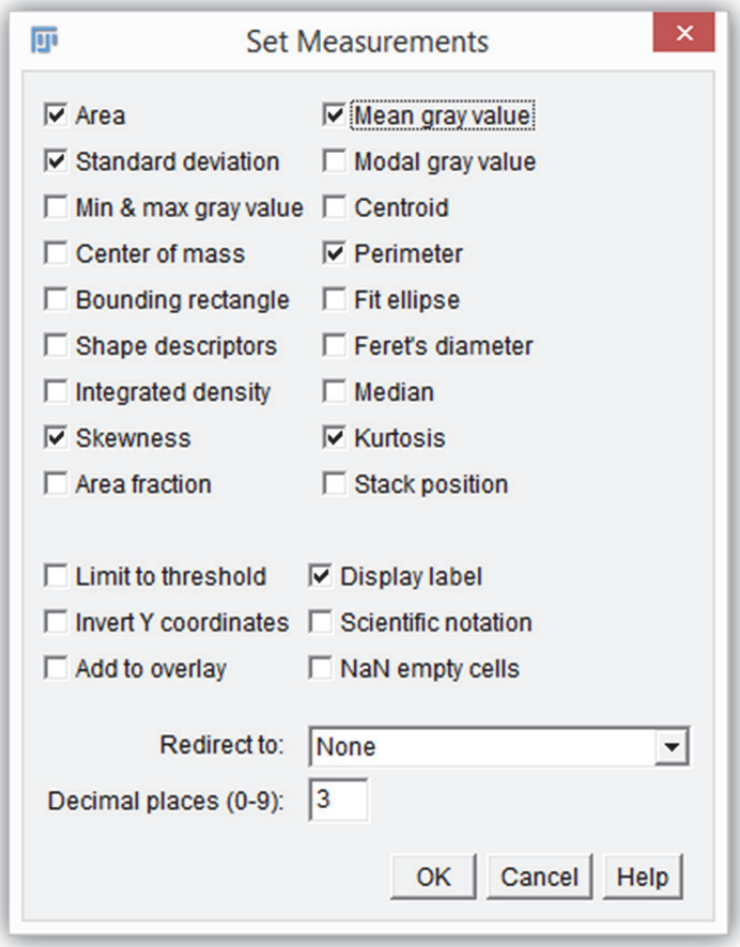
7. Select the binary image and go to **Analyze → Analyze Particles** and configure it only to add the generated ROIs into the ROI manager and press **OK**



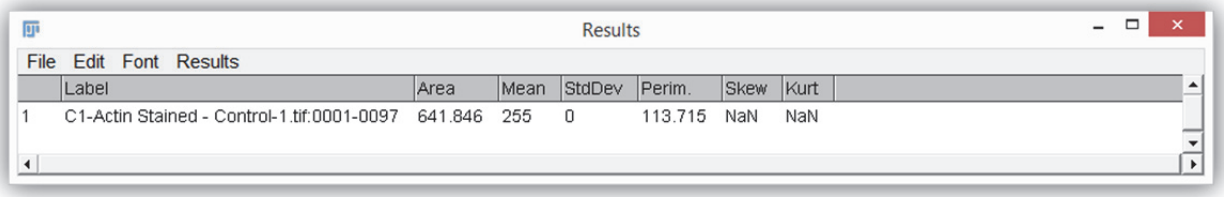
You should get a single ROI around the cell and one entry in the ROI manager



8. Go to **Analyse → Set Measurements** and configure it to measure Area, Mean gray value, Standard deviation, Perimeter, Skewness and Kurtosis as shown below

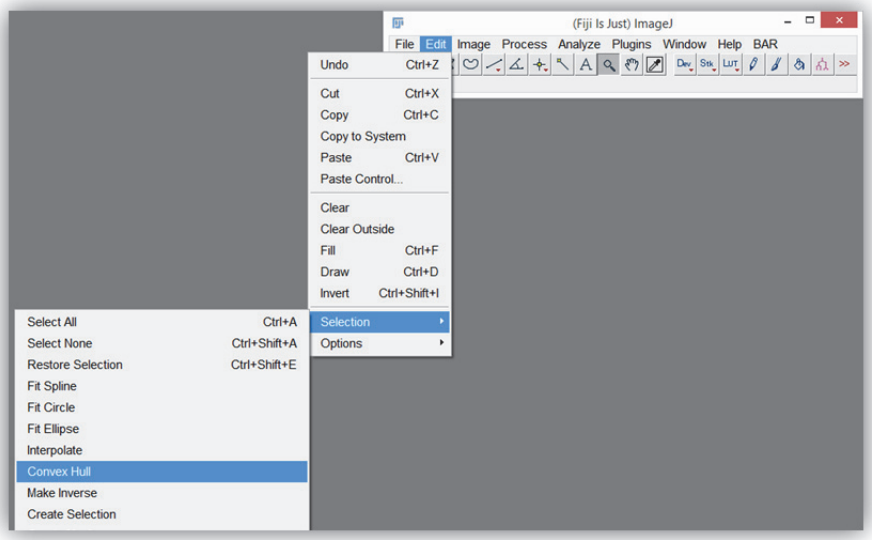


9. Select the ROI from the ROI manager and got to **Analyze → Measure** or press **Ctrl + M**

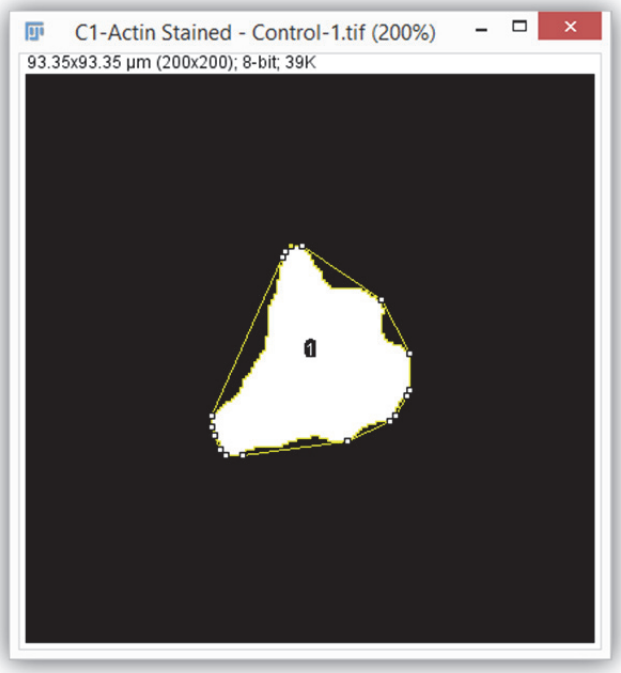


The only number of importance from this measurement is the **Perimeter** value

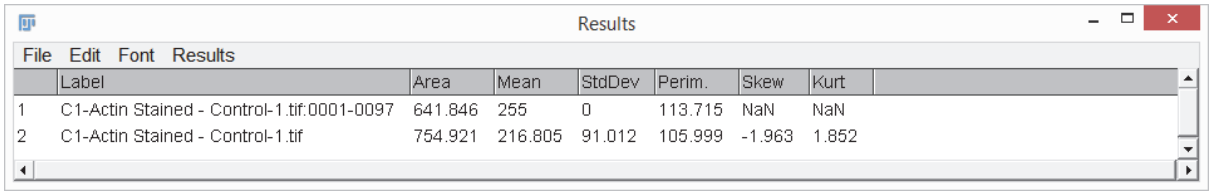
10. Select the binary image again and make sure the ROI is selected. Go to **Edit → Selection → Convex Hull**



11. This will create a convex hull selection around the cell. The convex hull is the result of joining the other most points of an object together.

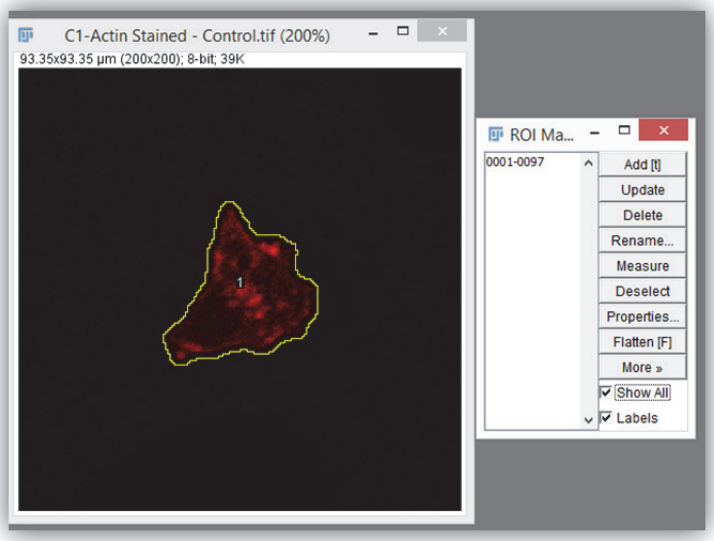


12. Once again run the **Measure** command

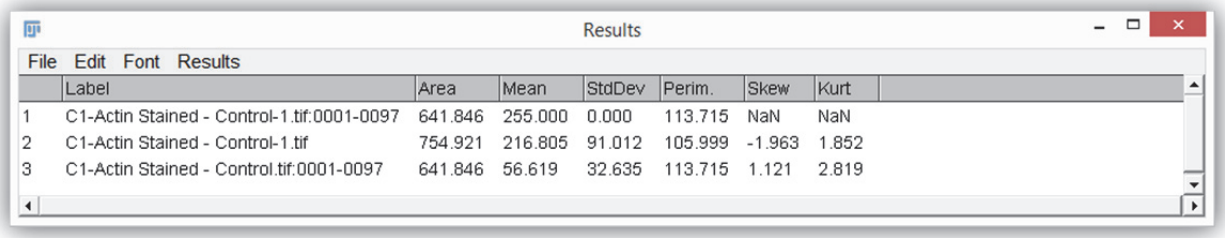


As before the only number of interest is the **Perimeter** value. The other values (mean, stdev etc) are a bit strange because the region of the convex hull covers both the white of the binary object and the black of the background.

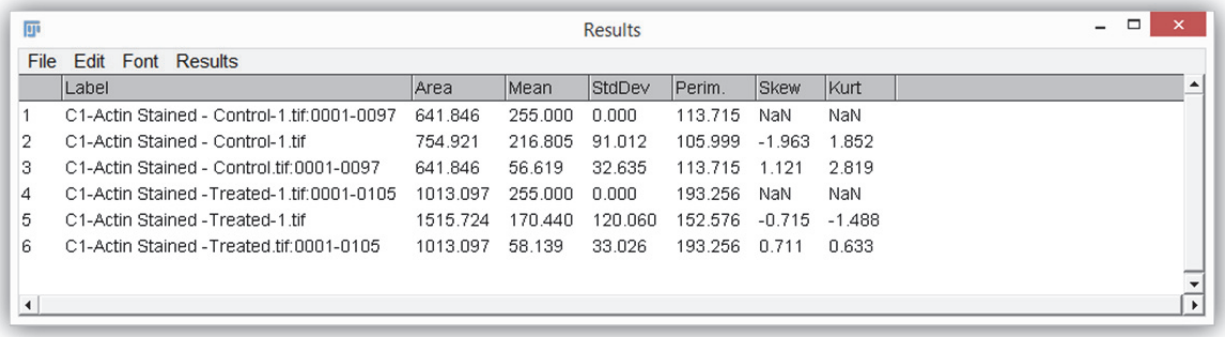
13. Finally select the original actin channel image, go to the ROI manager and tick the show all box to copy the ROI to the selected image.



14. For the last time select the ROI in the list and run the **Measure** command



15. You now have all the information you need for this cell. We know its **Area** (641.846um^2) from the first row. Its **Perimeter** (113.715um) from the first row. Its **Convex Hull** perimeter (105.999) from the second row and the Skewness and Kurtosis values (1.121, 2.819) from the last row. We can also calculate the **Perimeter** to **Convex Hull Ratio** 105.999/113.715=0.93
16. Close all the open images, delet the ROI from the ROI manager and repeat these steps for the other image in the folder (**Actin Stain – Treated.tif**).



	Label	Area	Mean	StdDev	Perim.	Skew	Kurt
1	C1-Actin Stained - Control-1.tif:0001-0097	641.846	255.000	0.000	113.715	NaN	NaN
2	C1-Actin Stained - Control-1.tif	754.921	216.805	91.012	105.999	-1.963	1.852
3	C1-Actin Stained - Control.tif:0001-0097	641.846	56.619	32.635	113.715	1.121	2.819
4	C1-Actin Stained -Treated-1.tif:0001-0105	1013.097	255.000	0.000	193.256	NaN	NaN
5	C1-Actin Stained -Treated-1.tif	1515.724	170.440	120.060	152.576	-0.715	-1.488
6	C1-Actin Stained -Treated.tif:0001-0105	1013.097	58.139	33.026	193.256	0.711	0.633

The treated cell gives a ratio of 0.79 meaning it is more spikey. It also gives quite a different Kurtosis value (while the other values don't change much). A lower kurtosis value means the intensities are flatter and more smooth with a less pronounced peak.

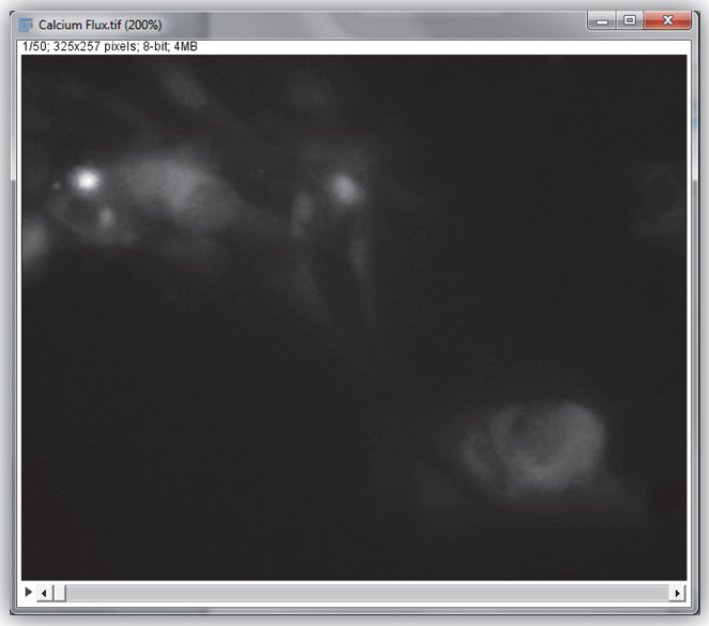
Intensity over Time

Aim

Live imaging can provide a wealth of information that cannot be achieved form fixed samples alone. In this technical note you will be shown how to measure intensity spikes in a calcium flux experiment but the principals demonstrated can be applied to any time series analysis.

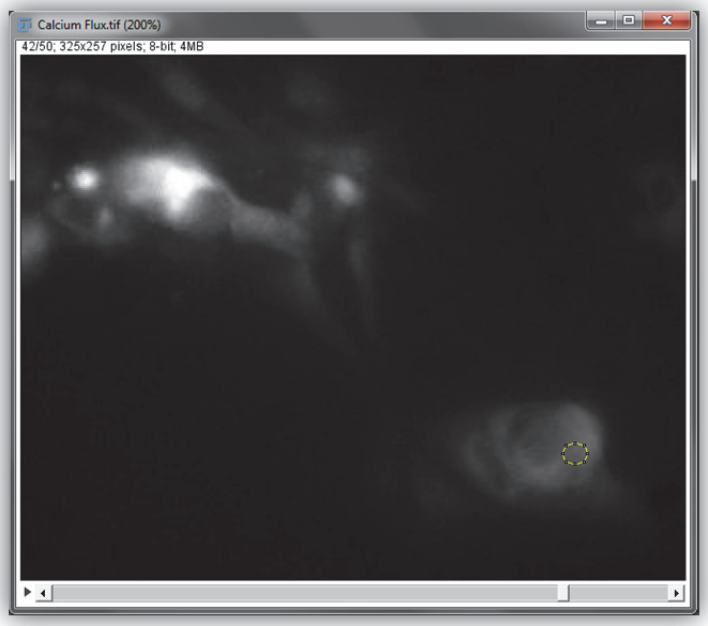
Single Point Analysis

1. Open **Calcium Flux.tif** from the **Demo Images\Widefield\Calcium Flux** folder.

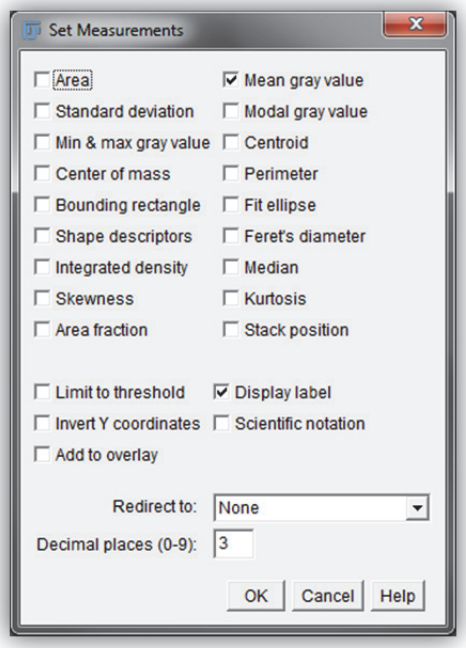


If you play through the stack you will notice that the cells are flashing and flashing at different rates.

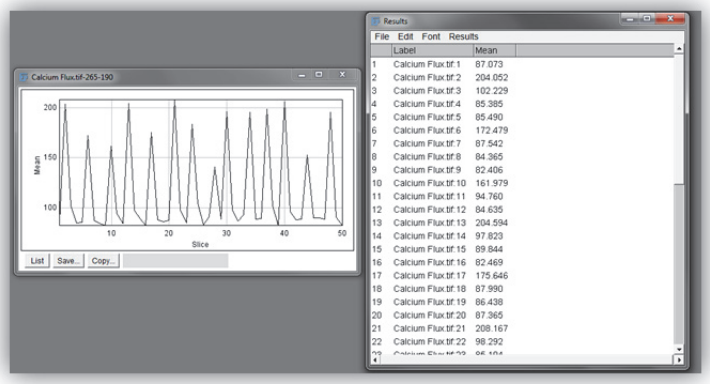
2. To measure one point draw a ROI (a circle in this example) on the area you want to measure



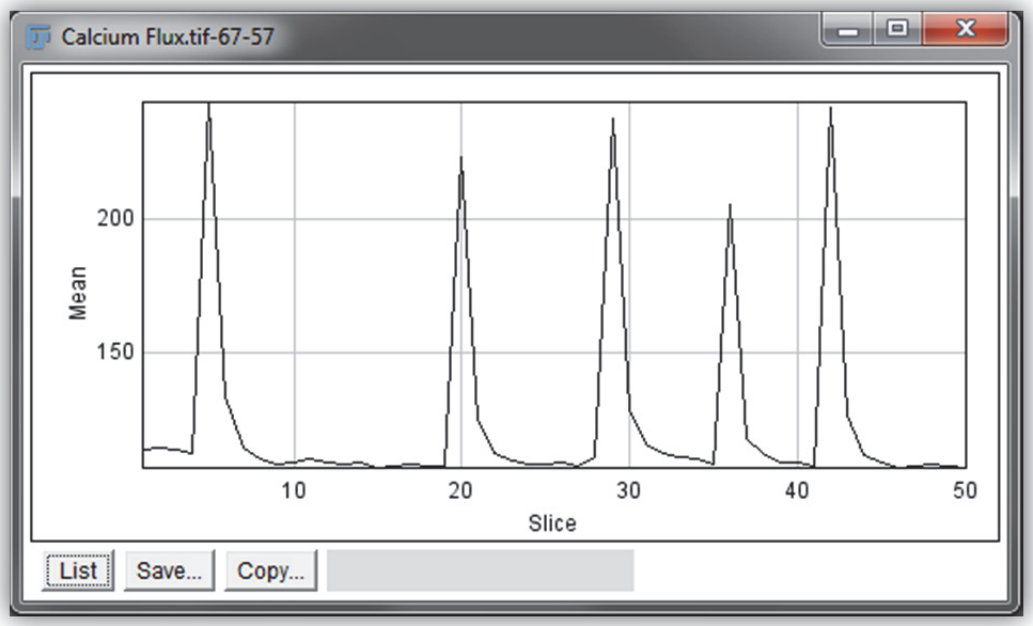
3. Before measuring we need to reconfigure the **Set Measurements** options to only show **Mean Grey Level**.



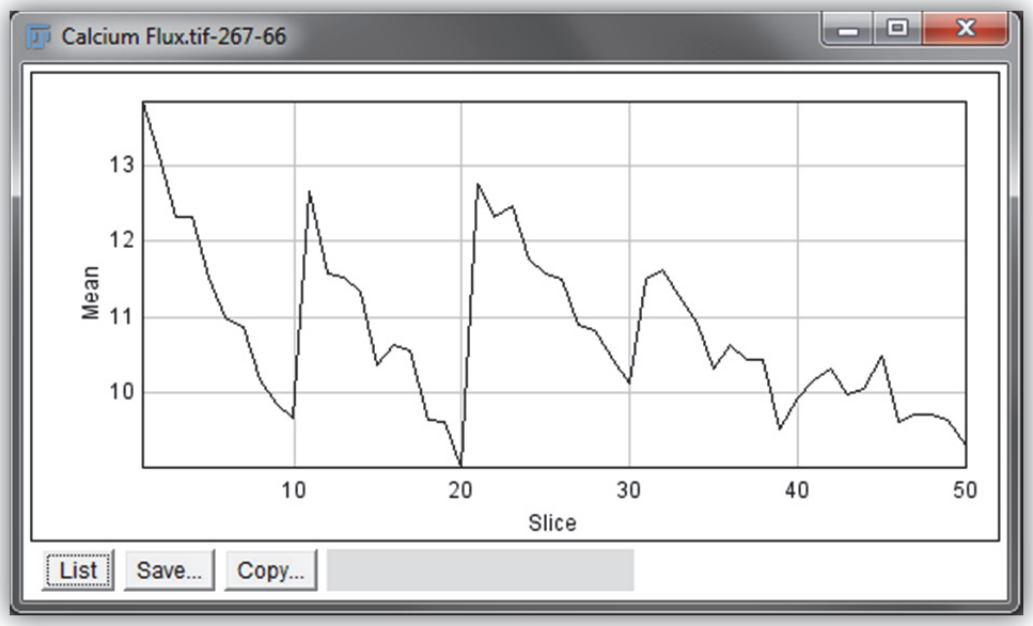
4. No go to **Image → Stacks → Plot Z Axis Profile.** The output will be a trace graph of the intensities and a table of the raw values that can be saved for later.



5. Move the ROI to a different cell and plot the profile again. Notice that different cells have different responses.



6. Also try moving the ROI to the background where there are no cells and plot the profile.

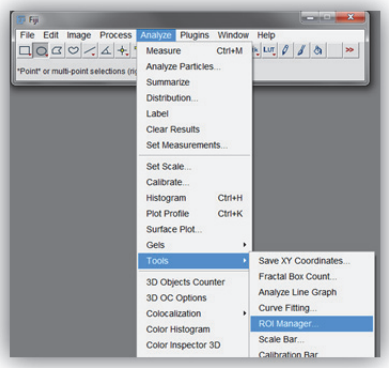


While the scale of the spikes is very low (13 vs. 250 grey levels) they need to be taken into consideration when calculating final values.

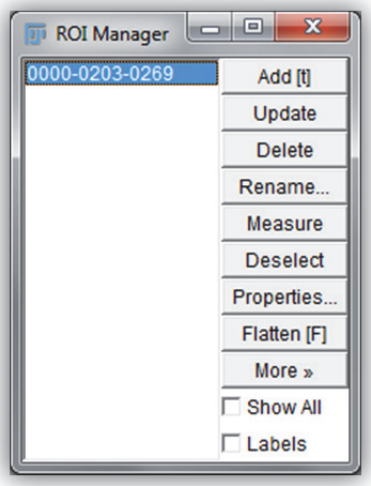
Multi Point Analysis – Manual ROI Entry

Analysing single points at a time is ok if there are not too many, but if you need to measure many points there is an alternative.

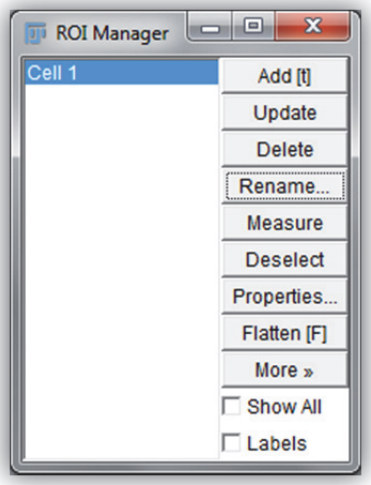
1. Close all the result and graph windows so you are only left with the original calcium flux image. Go to **Analyse → Tools → ROI Manager**



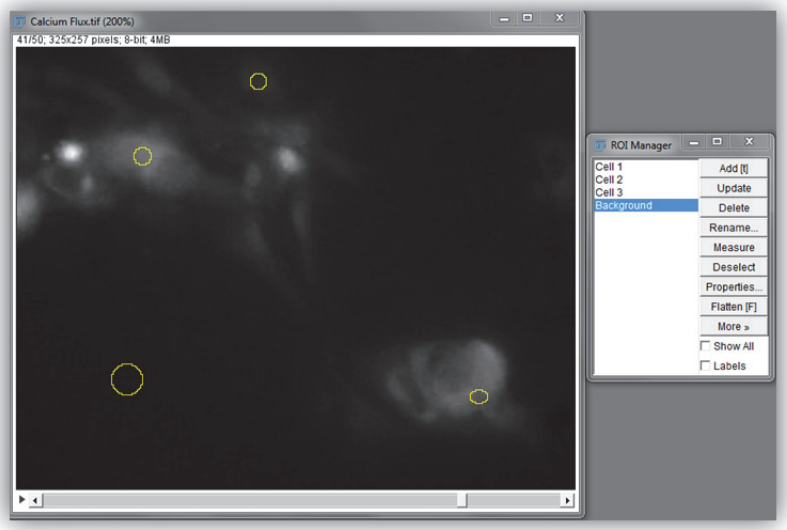
2. Now draw your first ROI on the calcium flux image. Once you have it drawn press the **Add** button in the **ROI Manager**. The ROI will be added to the list.



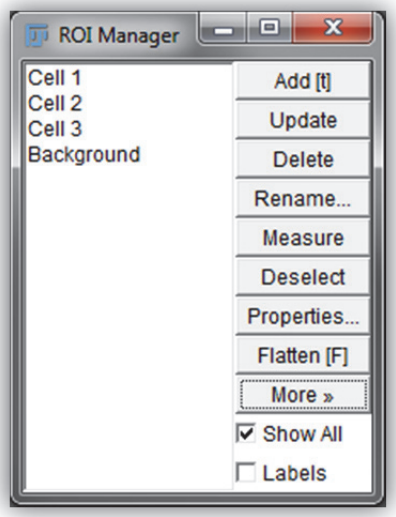
3. If you want to give it a more meaningful name just press the **Rename…** button and type in what you want.

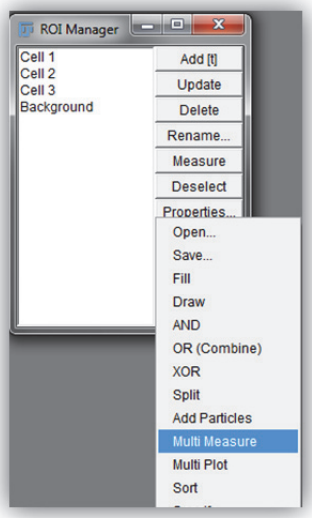


4. Add more ROIs by holding down the **shift** key and then adding (and renaming them) to the **ROI Manager.** Don't forget to include one for the background.

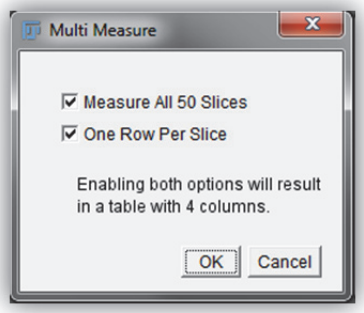


5. Tick the **Show All** box. Then click on the **More>>** button and select **Multi Measure**.

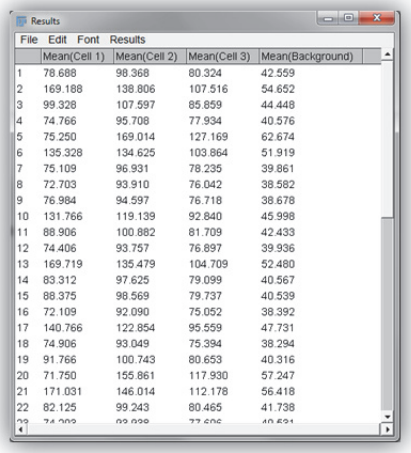




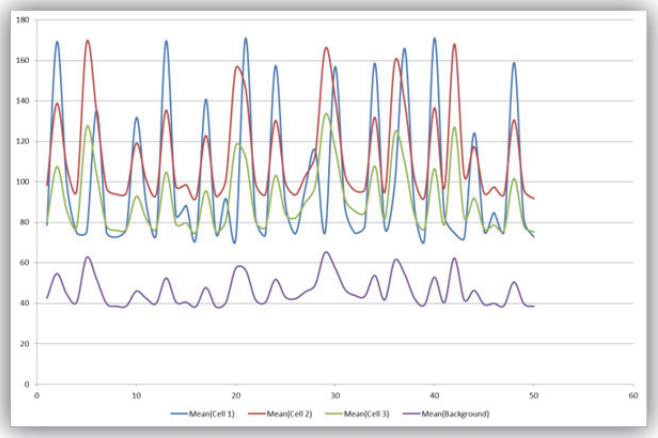
6. Make sure both boxes are ticked in the next window and press **OK.**



7. You won't get graphs this time, just a results table that can be saved and graphed later in Excel.



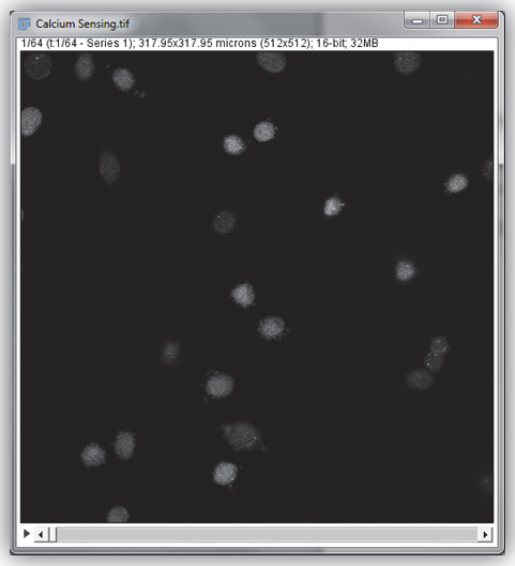
	Mean(Cell 1)	Mean(Cell 2)	Mean(Cell 3)	Mean(Background)
1	78.688	98.368	80.324	42.559
2	169.188	138.806	107.516	54.652
3	99.328	107.597	85.859	44.448
4	74.766	95.708	77.934	40.576
5	75.250	169.014	127.169	62.674
6	135.328	134.625	103.864	51.919
7	75.109	96.931	78.235	39.861
8	72.703	93.910	76.042	38.582
9	76.984	94.597	76.718	38.678
10	131.766	119.139	92.840	45.998
11	88.906	100.882	81.709	42.433
12	74.406	93.757	76.897	39.936
13	169.719	135.479	104.709	52.480
14	83.312	97.625	79.099	40.567
15	88.375	98.569	79.737	40.539
16	72.109	92.090	75.052	38.392
17	140.766	122.854	95.559	47.731
18	74.906	93.049	75.394	38.294
19	91.766	100.743	80.653	40.316
20	71.750	155.861	117.930	57.247
21	171.031	146.014	112.178	56.418
22	82.125	99.243	80.465	41.738



Multi Point Analysis – Automatic ROI Entry

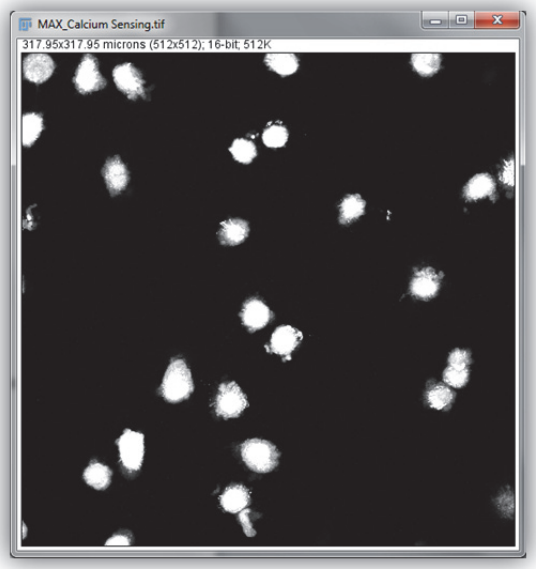
If the image has the right information in it, or there is a separate channel that marks out the cells, the ROIs can be made automatically by segmenting the cells out.

1. Open **Calcium Sensing.tif** from the **Demo Image\Confocal\Calcium Sensing** folder.

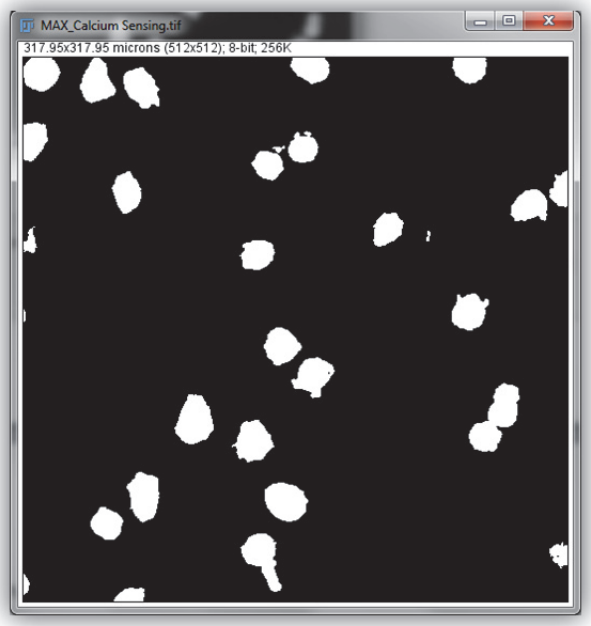


   If you play through the stack you will notice that the cells stack in one place so we don't need a secondary channel to be able to measure the cells automatically.

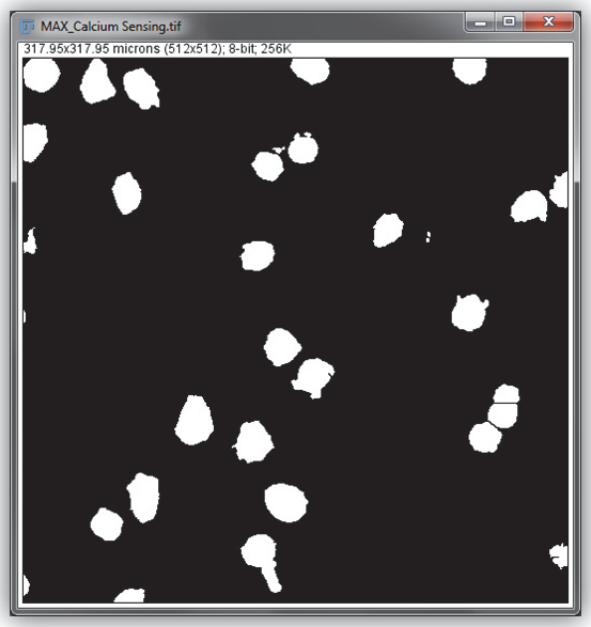
2. Create a maximum intensity **Z Projection** of the stack.



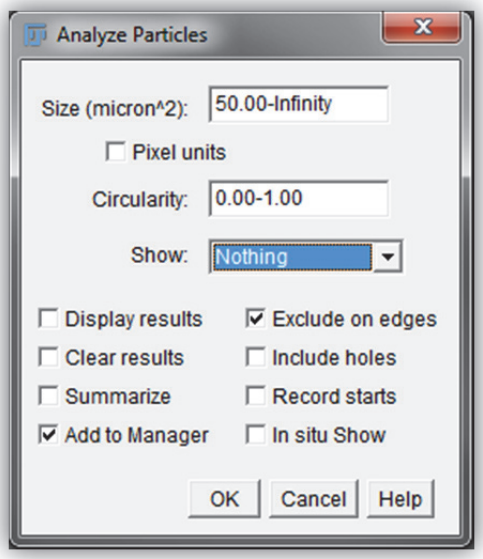
3. Smooth the projection, threshold it and make a binary image.



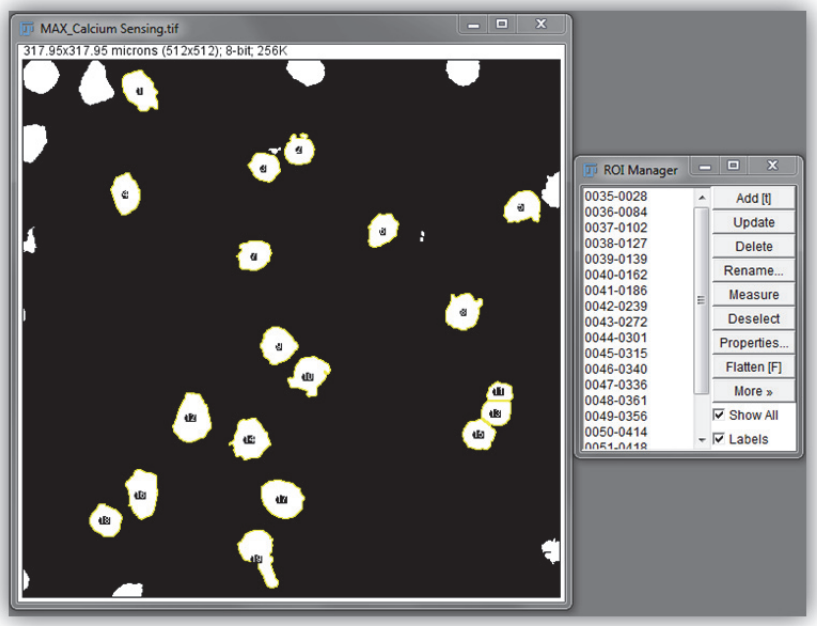
4. Some of the cells have joined together so use the **Binary Watershed** function to spate them.



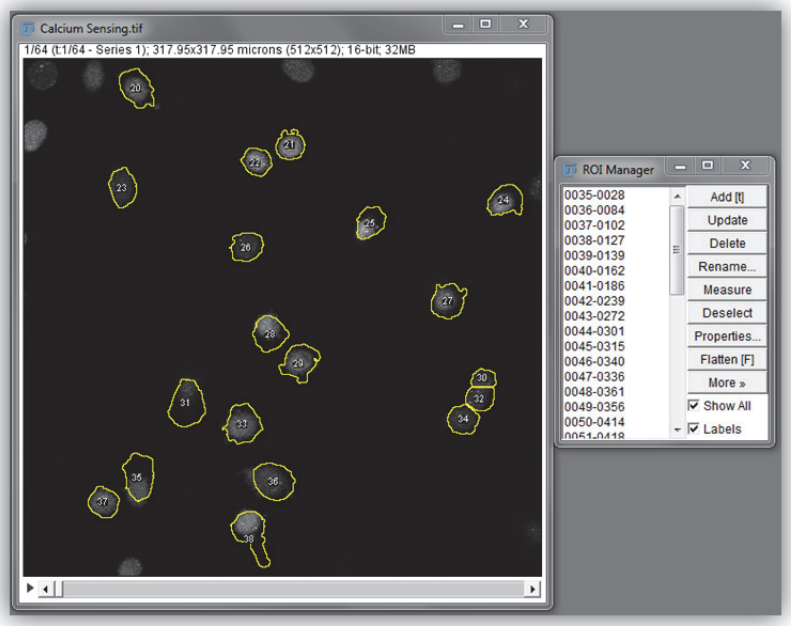
5. Configure the **Analyse Particles** module to create a mask of the individual cells, remove edge cells and small debris. Also make sure the **Add to Manager** box is ticked so we can save the ROIs created.



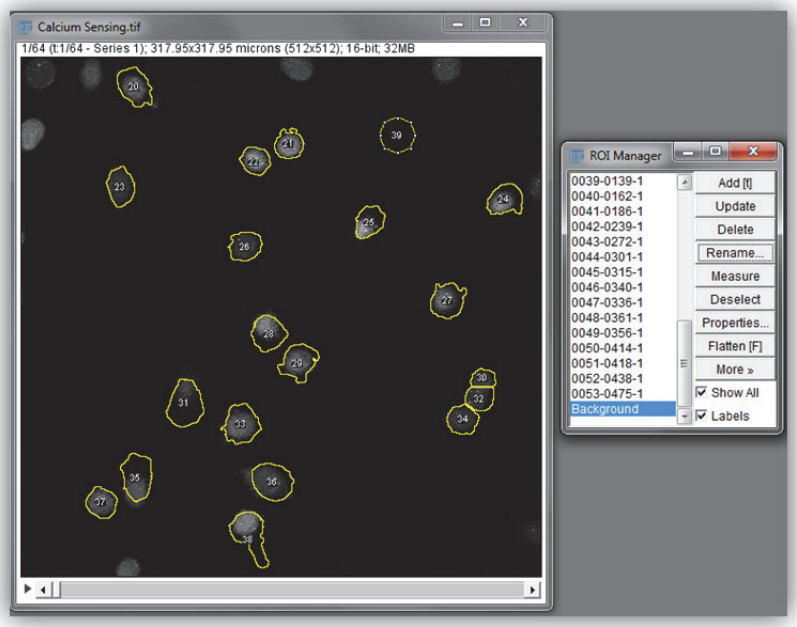
6. Save the ROIs and close the binary mask image.



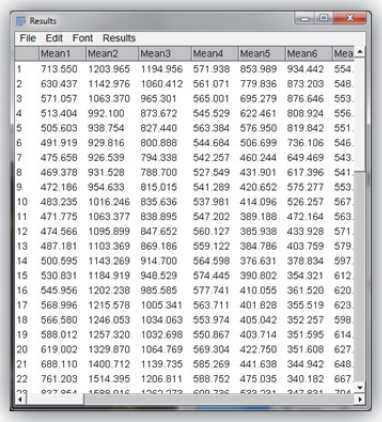
7. Load the saved ROIs onto the original image stack.



8. Manually draw a ROI on the background and add it to the ROI list.



9. Use **Multi Measure** to get the results for all the ROIs. Once again the results can be saved and graphed in excel later.



	Mean1	Mean2	Mean3	Mean4	Mean5	Mean6	Mea
1	713.550	1203.965	1194.956	571.938	853.989	934.442	554.
2	630.437	1142.976	1060.412	561.071	779.836	873.203	548.
3	571.057	1063.370	965.301	565.001	695.279	876.646	553.
4	513.404	992.100	873.672	545.529	622.461	808.924	556.
5	505.603	938.754	827.440	563.384	576.950	819.842	551.
6	491.919	929.816	800.888	544.684	506.699	736.106	546.
7	475.658	926.539	794.338	542.257	460.244	649.469	543.
8	469.378	931.528	788.700	527.549	431.901	617.396	541.
9	472.186	954.633	815.015	541.289	420.652	575.277	553.
10	483.235	1016.246	835.636	537.981	414.096	526.257	567.
11	471.775	1063.377	838.895	547.202	389.188	472.164	563.
12	474.566	1095.899	847.652	560.127	385.938	433.928	571.
13	487.181	1103.369	869.186	559.122	384.786	403.759	579.
14	500.595	1143.269	914.700	564.598	376.631	378.834	597.
15	530.831	1184.919	948.529	574.445	390.802	354.321	612.
16	545.956	1202.238	985.585	577.741	410.055	361.520	620.
17	568.996	1215.578	1005.341	563.711	401.828	355.519	623.
18	566.580	1246.053	1034.063	553.974	405.042	352.257	598.
19	588.012	1257.320	1032.698	550.867	403.714	351.595	614.
20	619.002	1329.870	1064.769	569.304	422.750	351.608	627.
21	688.110	1400.712	1139.735	585.269	441.638	344.942	648.
22	761.203	1514.395	1206.811	588.752	475.035	340.182	667.

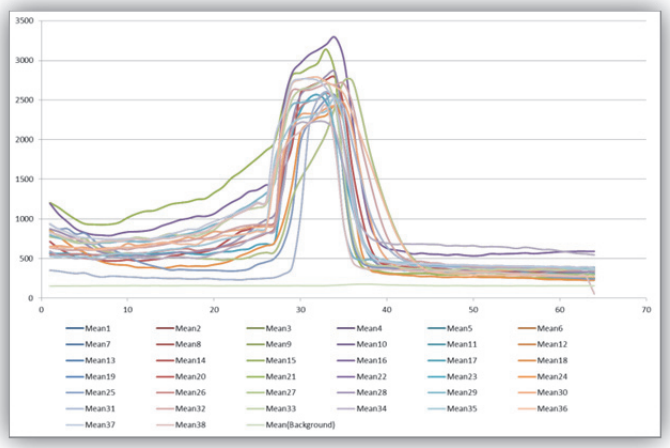


Image Filters – Fixed and Live Data

Aim

This technical note will show you how to analyse the area or a wound/scratch from a wound/scratch assay. The first part will take you through analysis of a fixed, high quality image. The second will take you through the analysis of a time-lapse data set.

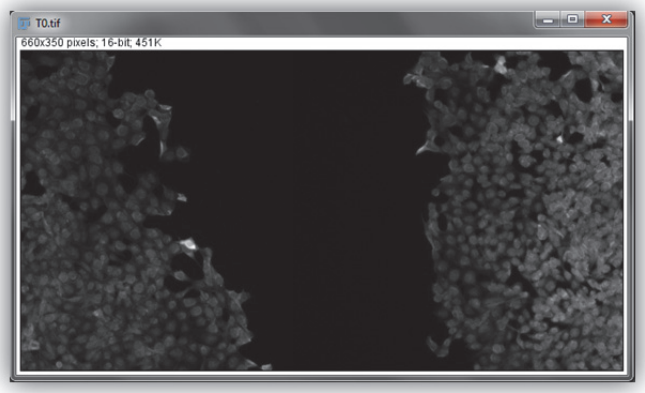
Time-lapse capture and analysis of wound assays can provide much more information than a simple end point readout. It can provide information about the dynamics of the wound closure over time. Due to the compromises that have to be made when capturing live cells (low light intensities, short exposures etc.) the quality of the images for analysis are usually never as good as a fixed sample. Because of this several extra filtering steps need to be incorporated in the analysis to extract the area of interest for analysis.

This technical note is a useful demonstration in the use of image filters (background subtraction, binary open etc.) in making an image amiable to analysis

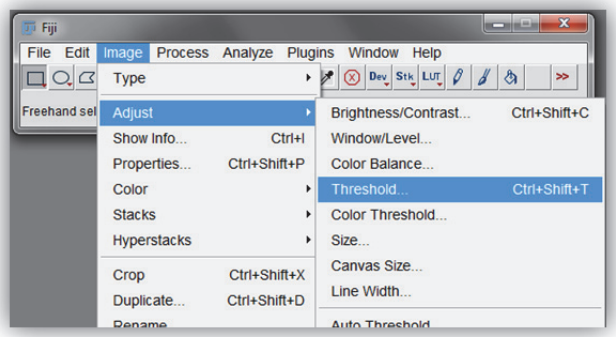
Fixed Sample Analysis – Fluorescence

Selecting the Scratch

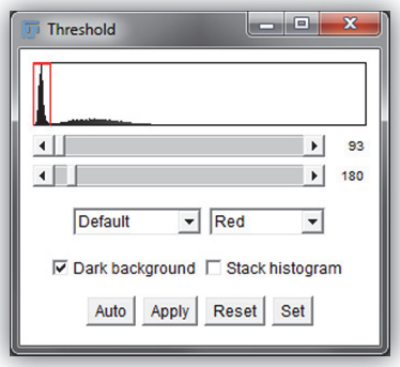
1. Open **T0.tif** from the **Demo Images\Widefield\Wounding\Fixed** folder



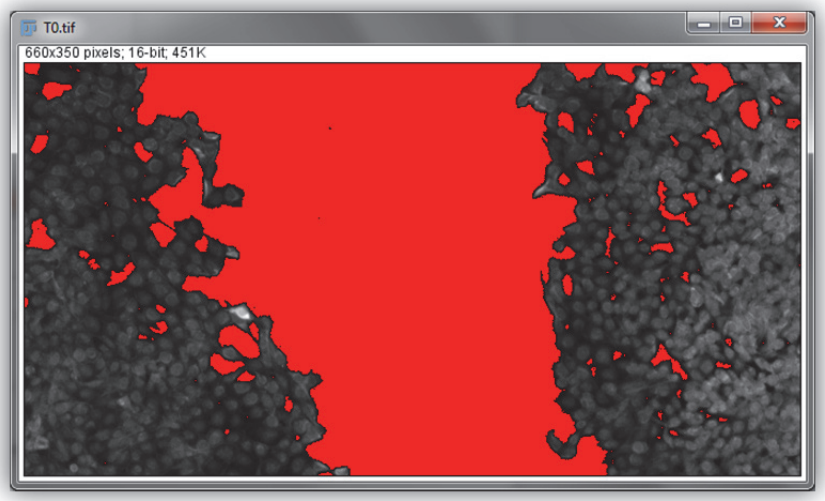
2. To be able to measure the wound it needs to be thresholded. Go to **Image→Adjust→Threshold**



3. Configure the **Threshold** dialog box as follows

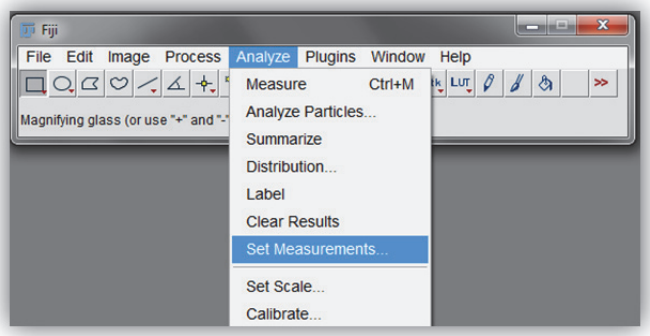


4. The result should be the area of the scratch selected in red

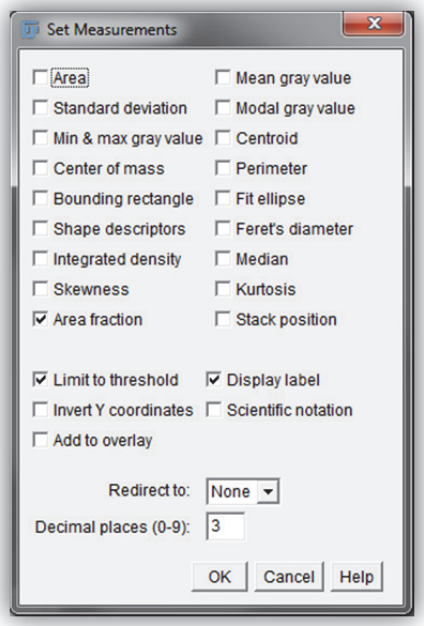


Measuring the Scratch and Logging the Data

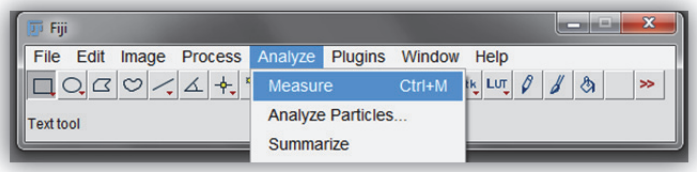
1. To measure the area of the scratch the right measurements first need to be set. Go to **Analyse → Set Measurements…**



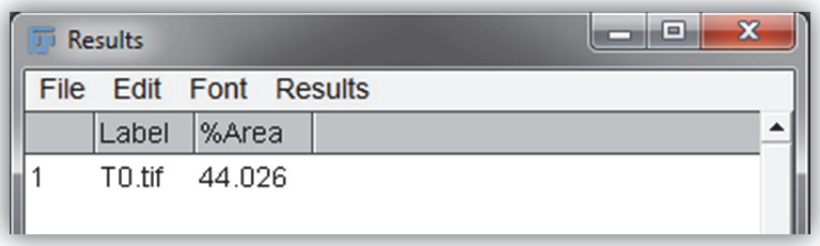
2. Select **Area Fraction, Limit to Threshold** and **Display Label** as shown below and press **OK**



3. Go to **Analyse → Measure** or press **Ctrl + M**



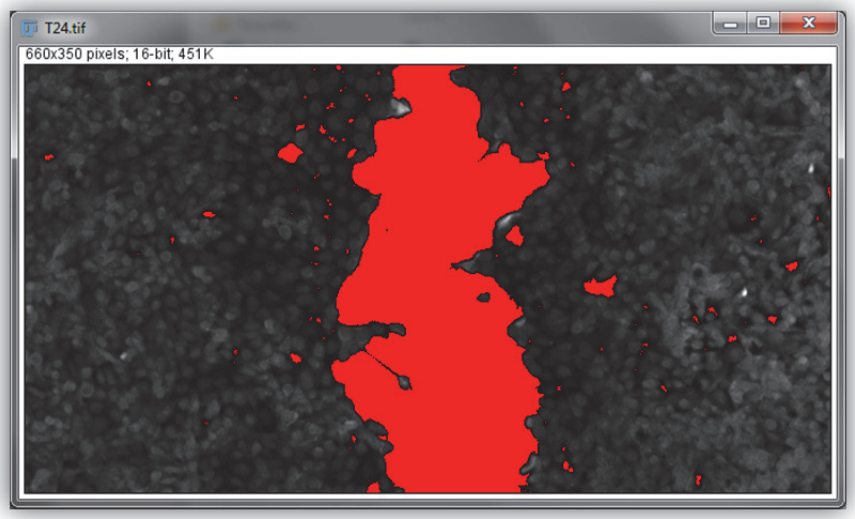
4. The percentage area of the wound will displayed in the **Result** log window

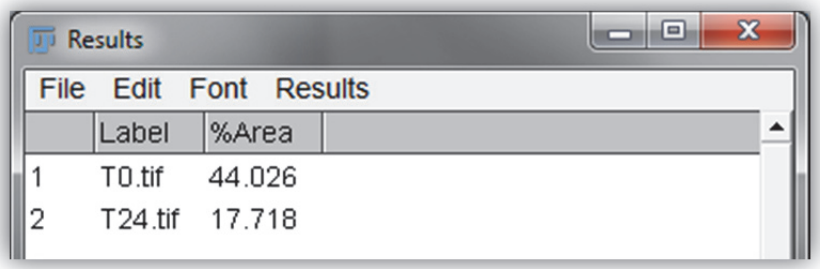


NOTE: The results in the result window can be saved to an **Excel** or **TXT** file

5. Close the open image and repeat the previous steps with **T24.tof** from the **Demo Images\Widefield\Wounding\Fixed** folder

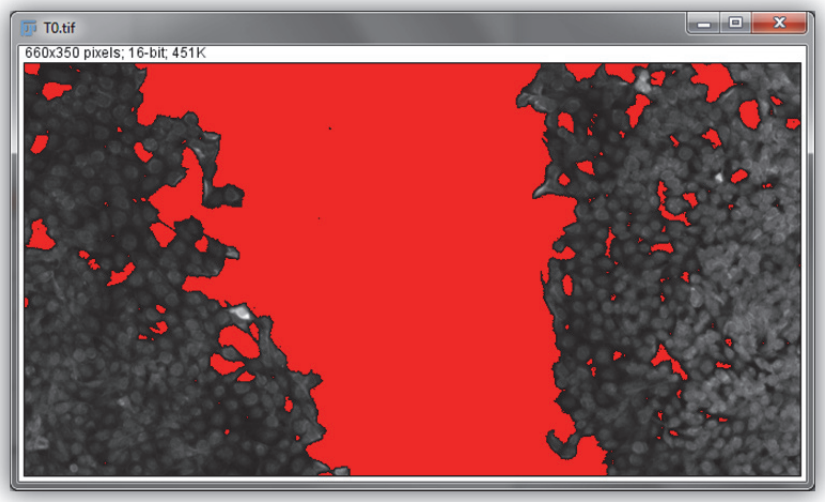
NOTE: The threshold required to select the wound may be slightly different





Removing Unwanted Area from the Analysis

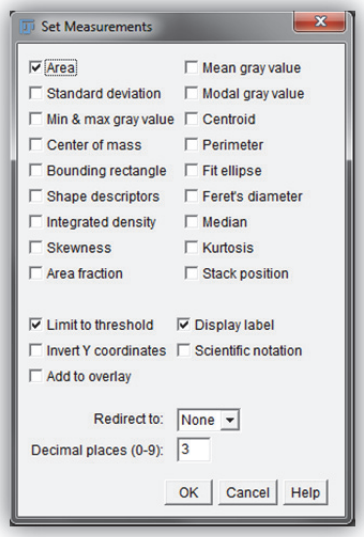
Close all previously opened images. Open and threshold **T0.tif** again. Notice that there are areas that are not part of the scratch being selected as well. This may be real gaps that should be analysed, but if they are not they can be removed in two different ways.



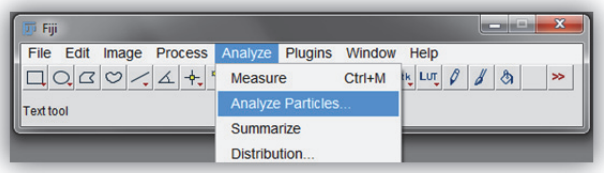
Method 1 – Size exclusion

By using a variation of the **Measure** option from above, called **Analyse Particles**, objects can be excluded from the analysis based on their size or shape.

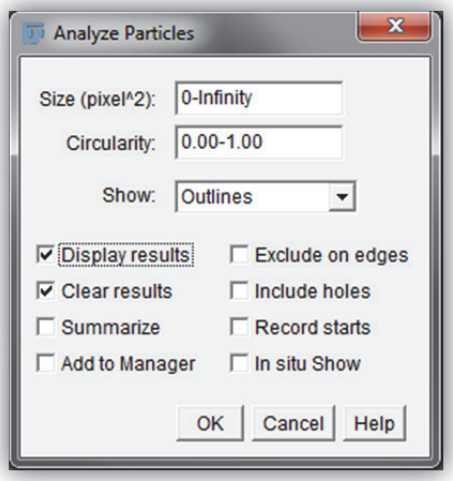
1. Go to **Analyse → Set Measurements…** deselect **Area Fraction** and select **Area.** Press **OK**



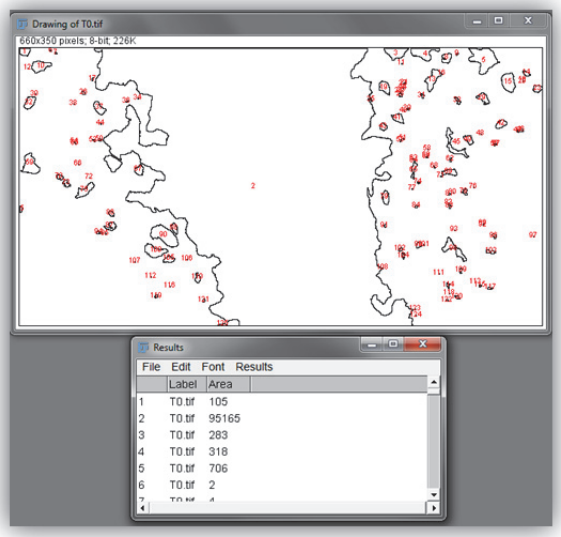
2. Go to **Analyse → Analyze Particles**



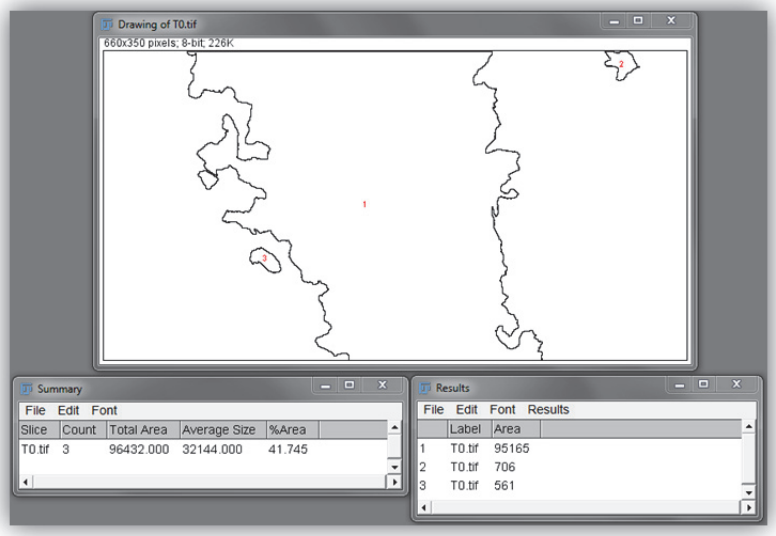
3. Configure the **Analyse Particles** dialog as below and press **OK**



4. You will get a new image that will show the outlines (and associated object number) of each object counted. The data will also be in the results table.



5. If you scroll through the results list you will notice that all the smaller chunks have an area of less than 500 pixels. Close the outlines image and go to **Analyse Particles** again. This time change the **Size (pixel^2):** value to **500-Infinity**. Also tick the **Summarise** box. Press **OK**

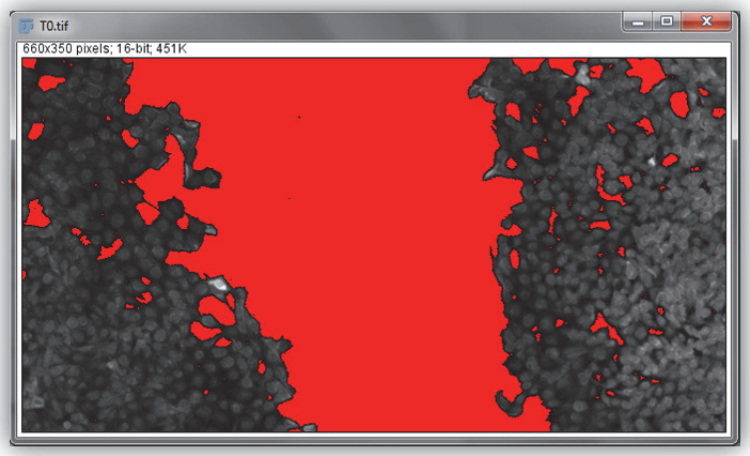


The result this time is only 3 parts of the image measured that are greater than 500 pixels in area. Also the summary results list the percentage area covered by the selected bits.

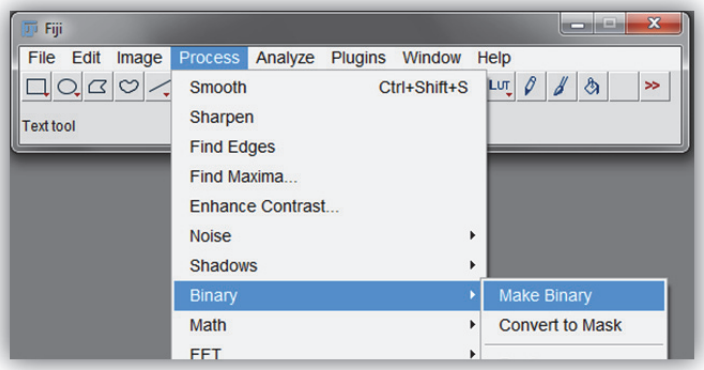
Method 2 – Using Image Filters

Instead of excluding objects from the analysis based on their size. They can be removed prior to analysis with morphology filters.

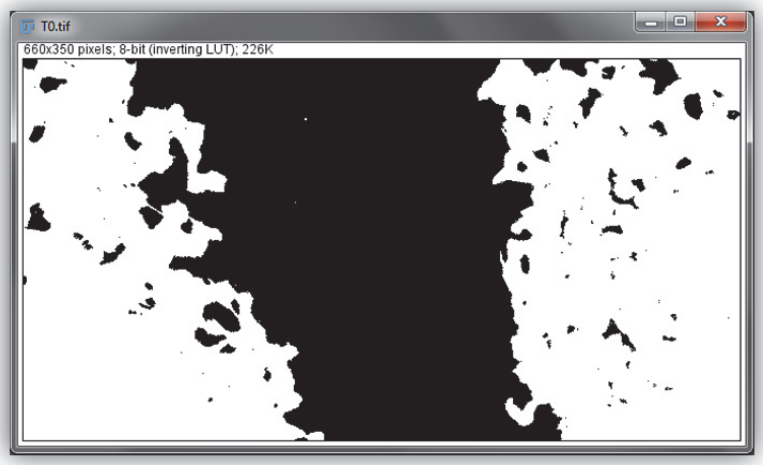
1. Open and threshold **T0.tif** again



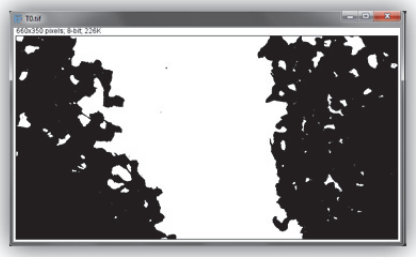
2. Go to **Process → Binary → Make Binary**



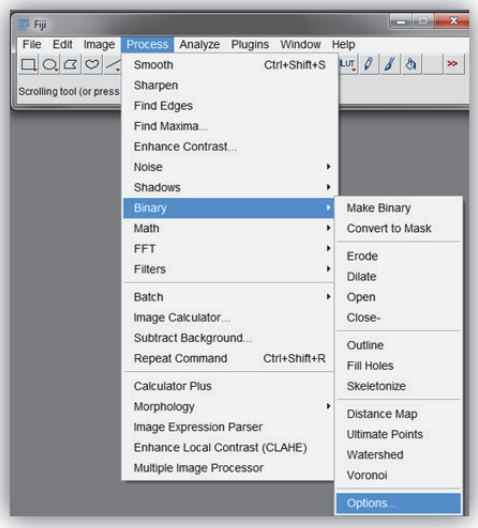
3. The result will be a black mask on a white background of the thresholded area (this may be reversed on some versions of Fiji).



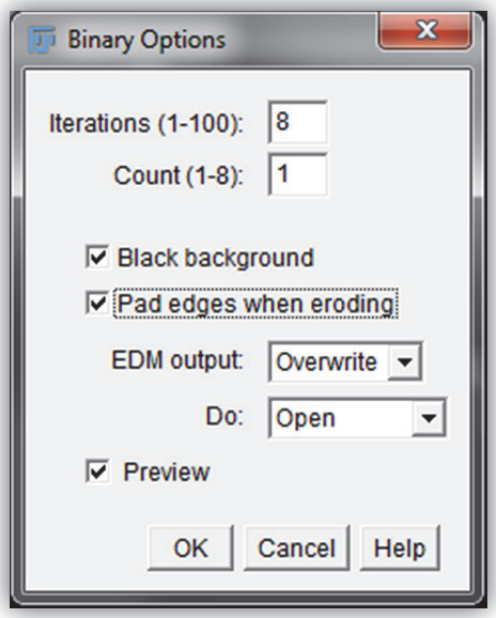
4. If you want to make the mask white for the positive areas, which most people prefer. Press the **LUT** button and select **Invert LUT**



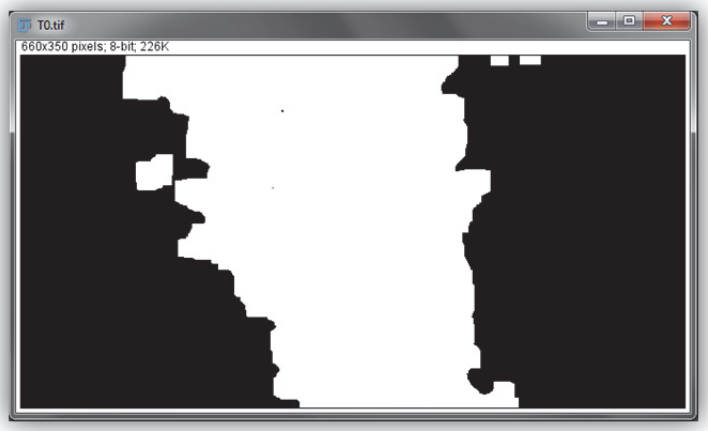
5. The best filter for removing small round blobs (like the falsely selected regions in this image) is an **Close** filter. Go to **Process → Binary → Options**



6. In the **Binary Options** dialog, set it as follows and press **OK.** Play around with the settings to see what different filters and setting do to the image.



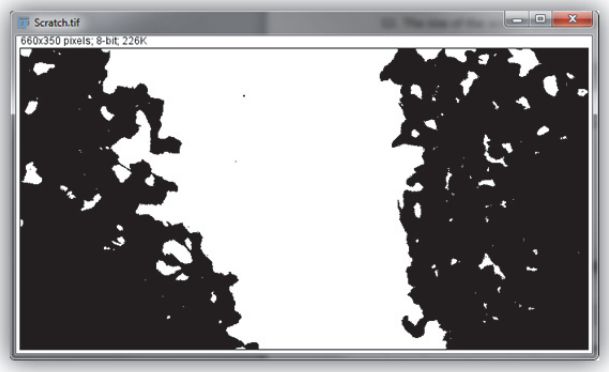
1. The resulting image will be a crude mask that represents the scratch with the smaller erroneous parts removed



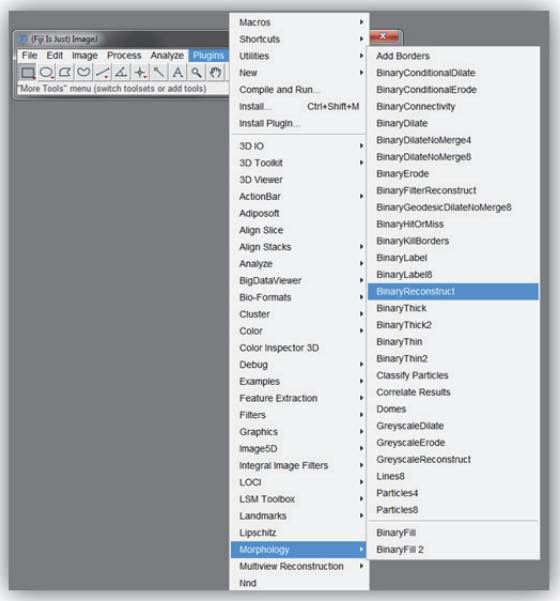
7. This image can now be thresholded and measured as before.

Binary Reconstruction of the Mask

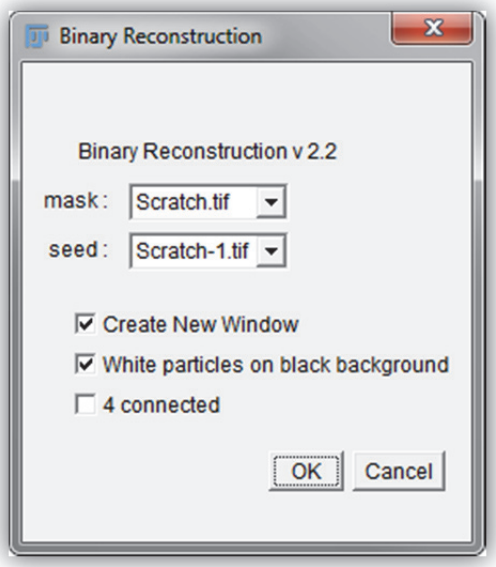
1. Keep the crude mask open. Reopen **Scratch.tif** and recrate the initial mask with all the erroneous fragments



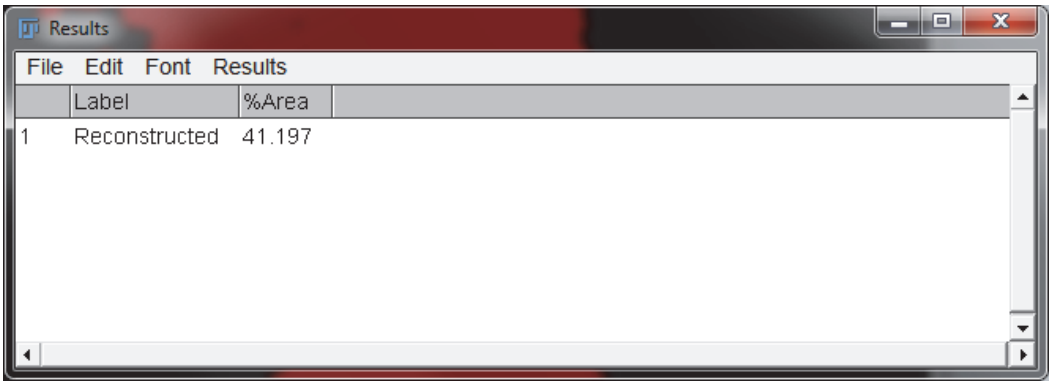
2. Go to **Plugins → Morphology → Binary Reconstruct**



3. Set the **Mask** to the original mask with the fragments and the **Seed** as the open filtered mask. Configure the other options as below.



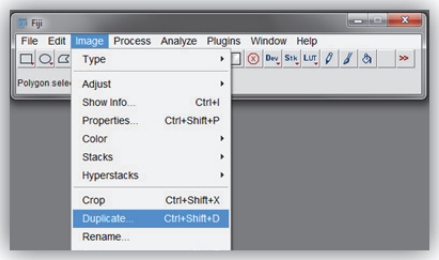
4. The result will be a clean mask that accurately represents the scratch that can be measured as before



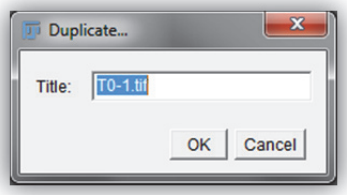
Create an Outline of the Result

Sometimes it is useful to have an outline of the segmented result placed on the original image so you can show what was actually measured from the original.

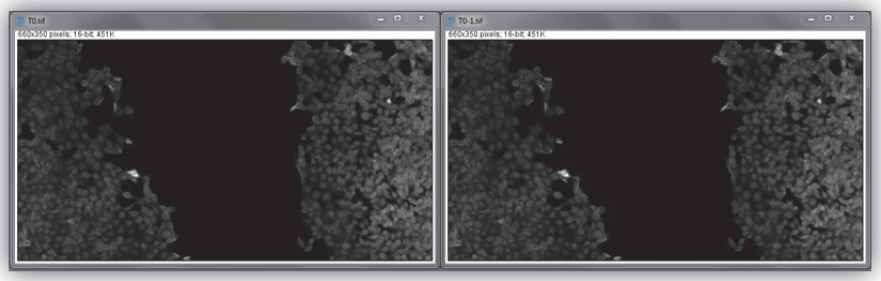
1. Open **T0.tif** again and go to **Image → Duplicate**



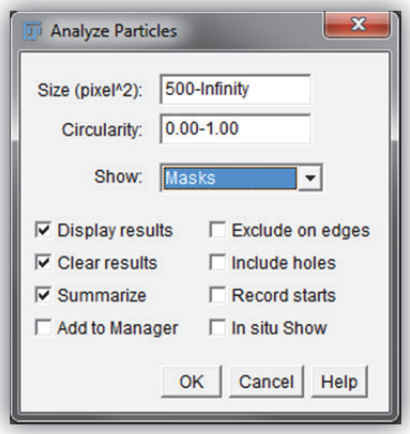
2. Press **OK** in the resulting **Duplicate…** dialog



3. The result should be two copies of the original image

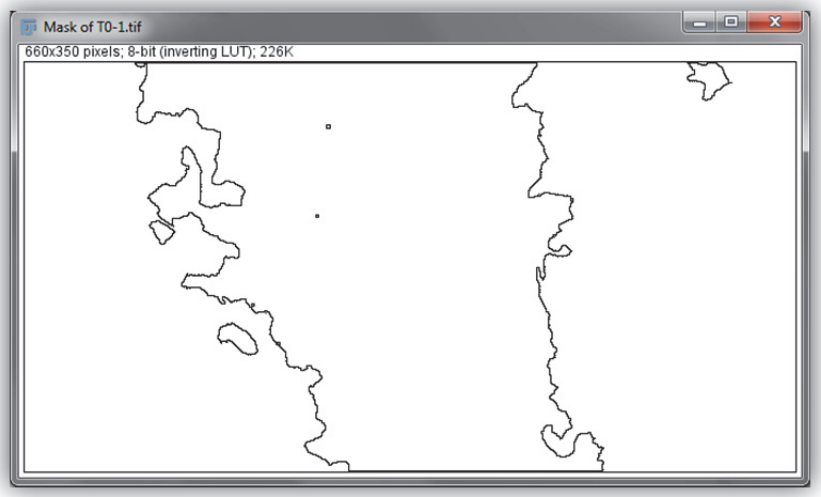


4. Select one of the image (in this case **T0-1.tif**) and threshold it as before. Use the analyse particles command again with the size exclusion filter on. But this time change the **Show** value to **Masks.** Press **OK.**



5. The result should be a binary mask of the analysed area. It may be inverted depending on your version of Fiji. If it is not inverted (Black wound, white cells) invert it now.

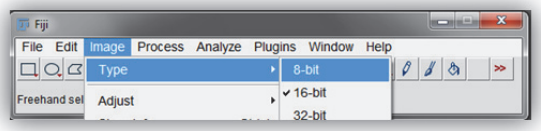
6. Select the mask and go to **Process → Binary → Outline**



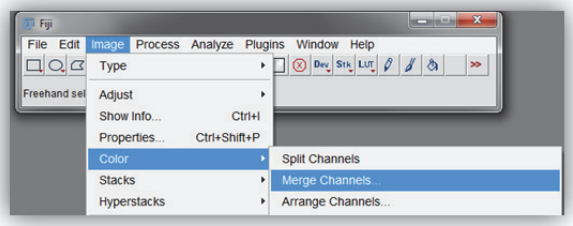
7. Invert the outline image to get a white outline representing the edges of the wound.

8. The next steps will involve merging the outline and original images together. For this to work the two images need to have the same bit depth. As this will be just an overview image to show a result, dropping the original image down to 8bit is the best option.

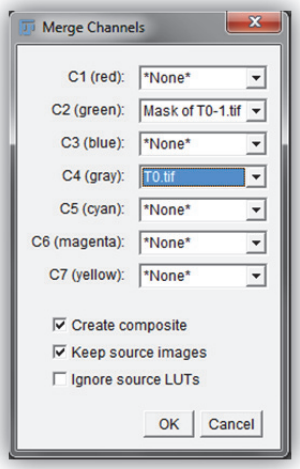
   Select the **T0.tif** image and go to **Image → Type → 8 bit**

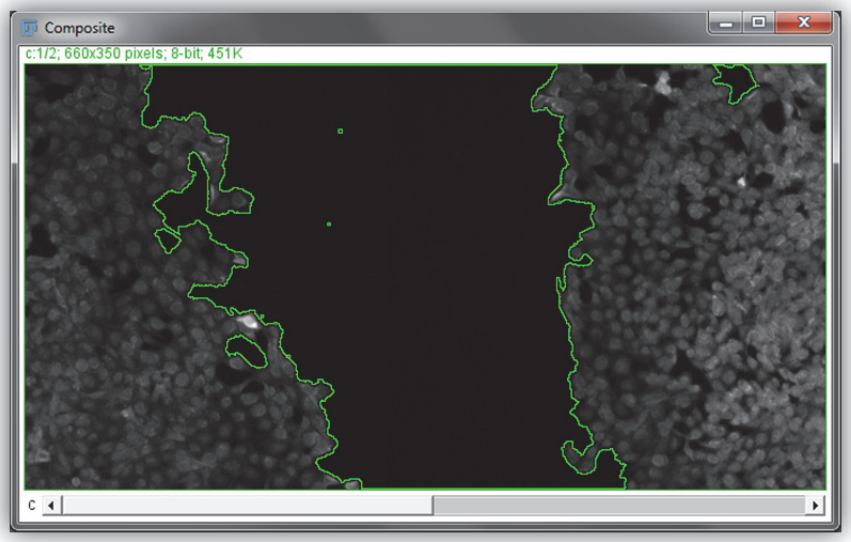


9. Go to **Image → Colour → Merge Channels…**

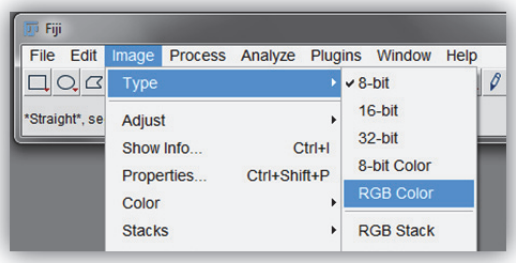


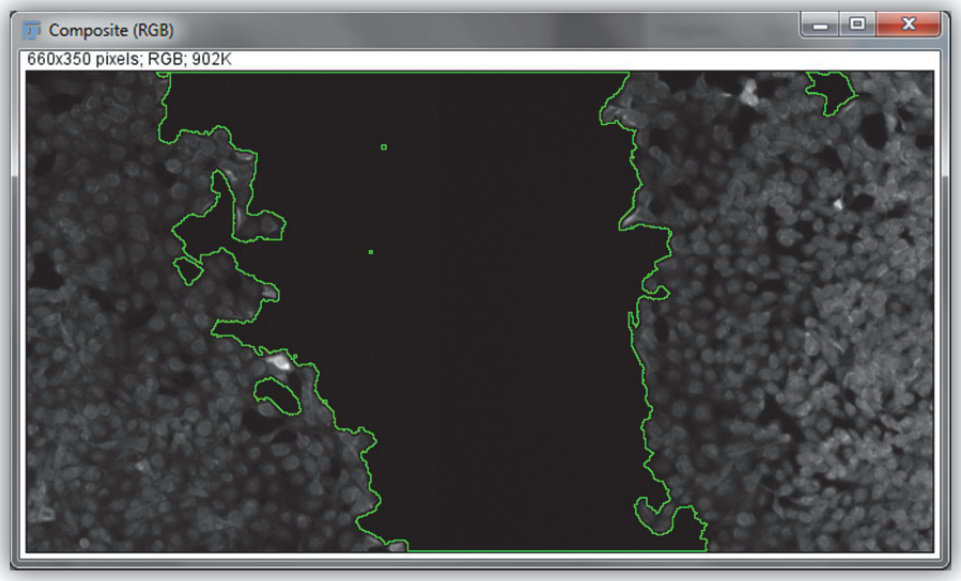
10. In the **Merge Channels** dialog, configure it as below and press **OK.** This will give you a composite image with a green outline around the measured wound. If you want a different coloured outline just put the mask image into a different colour channel.





11. The resulting image is a composite (2 channels in a stack) image. To make it compatible with PowerPoint etc. it needs to be converted to RGB. Select the merged image and go to **Image → Type → RGB Colour**



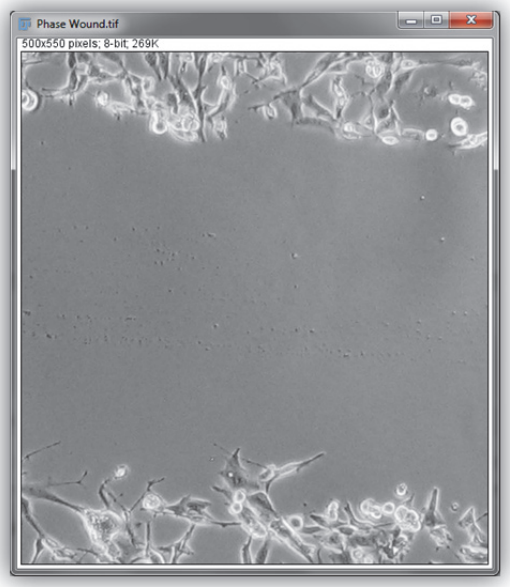


The resulting RGB image can be saved for later use.

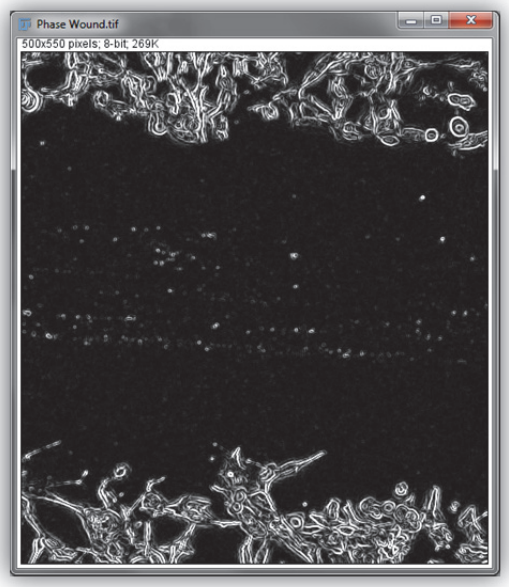
Fixed Sample Analysis – Brightfield

Sometimes it is not possible to fluorescently label a sample, in these cases it is still possible to analyse the scratch. To do this in must first be turned into a pseudo fluorescent image.

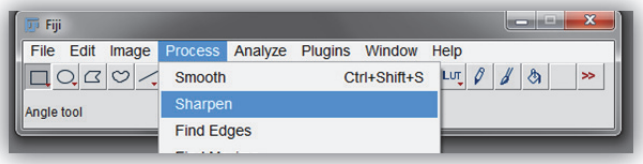
1. Open **Phase Wound.tif** from the **Demo Images\Widefield\Fixed** folder

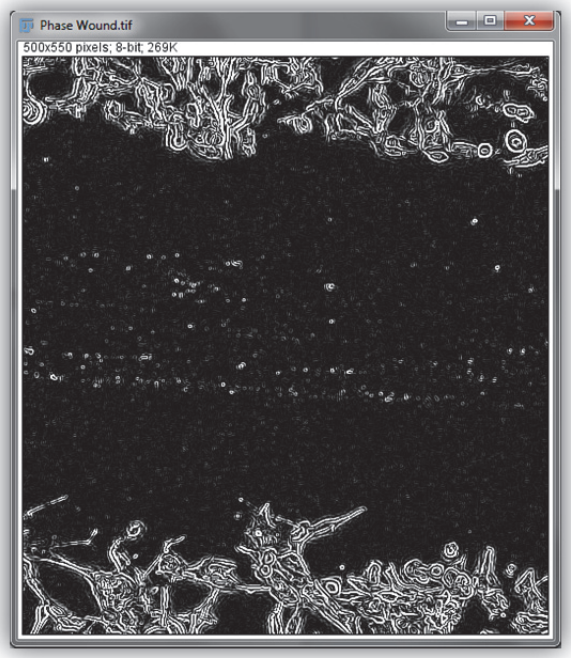


2. The first step is to define the edges of the cells. Go to **Process → Find Edges**

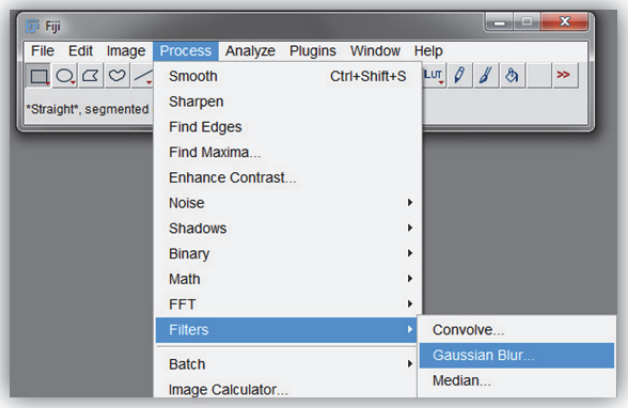


3. To sharpen the image up a bit go to **Process → Sharpen**

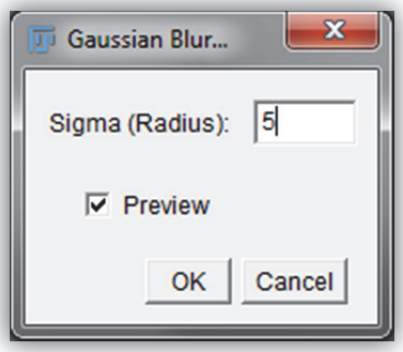


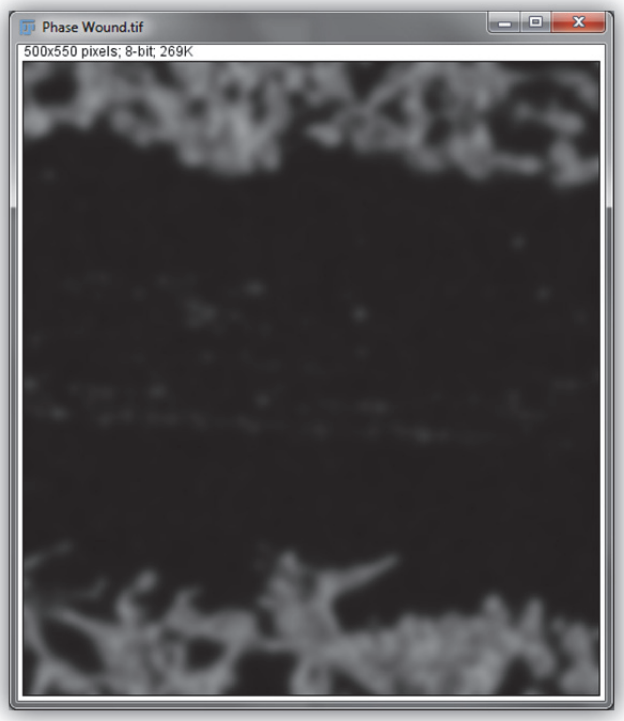


4. To remove a lot of the roughness in the image and make it easier to work with go to **Process → Filters → Gaussian Blur…**



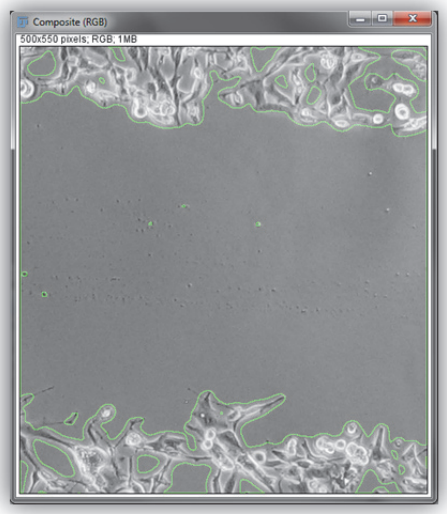
5. In the **Gaussian Blur…** dialog tick the **Preview** box and enter a value into the **Sigma (Radius)** box. Try different values to see what happens, for this image a value of **5** works quite well. Press **OK**





6. The resulting image, while looking blurry, can be now used like the previous fluorescent images to select the wound and measure it.

7. You can also generate and outline and place it back on the original image to show the validity of the analysis



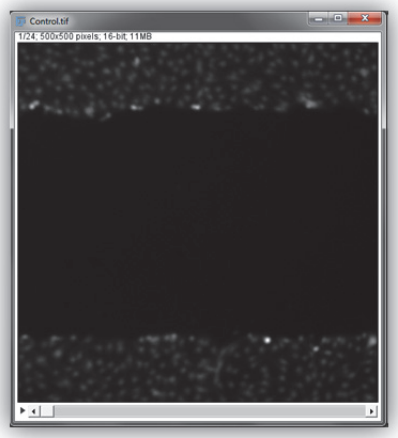
Live Sample Analysis

Measuring the wound closure rate in live imaged sample can provide a lot of power to any analysis but the images used are not always the best due to compromises that must be made to achieve a live image in the first place.

In this example we will measure the wound closure rate over 24 hours, but the image quality is not great. The image has an uneven background and bleaches over time.

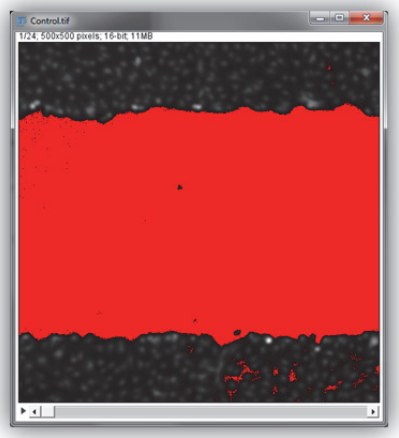
Selecting the Scratch

1. Open **Control.tif** from the **Demo Images\Widefield\Wounding\Live** folder

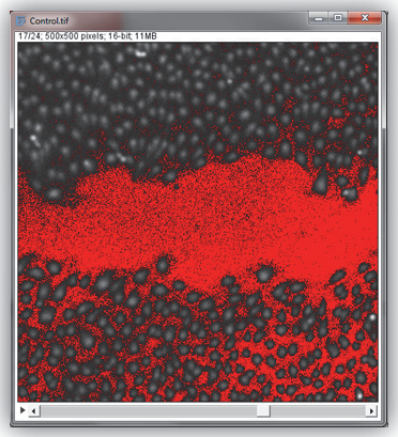


   If you play through the stack you will see the cells migrate but you will also notice that the image dims over time.

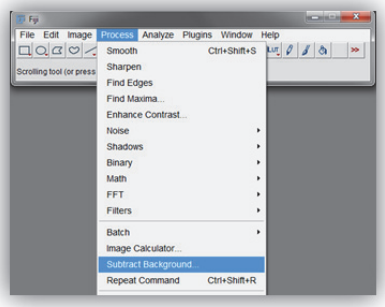
2. Threshold the first plane. You will notice that it is fairly good at picking up the scratch of the first plane (values used in this example are 270 to 509).



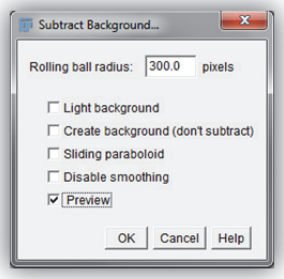
3. Select plane 17 and try to adjust the threshold to select the wound. Notice that the threshold fades as you change planes. This is because the image gets dimmer over time due to photobleaching during capture.
   You will also notice that it is not possible to evenly select the scratch due to uneven illumination.



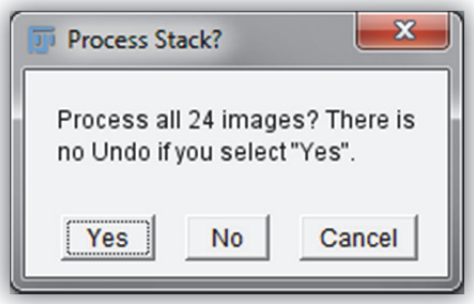
4. Reset the threshold and go to **Process → Subtract Background…**



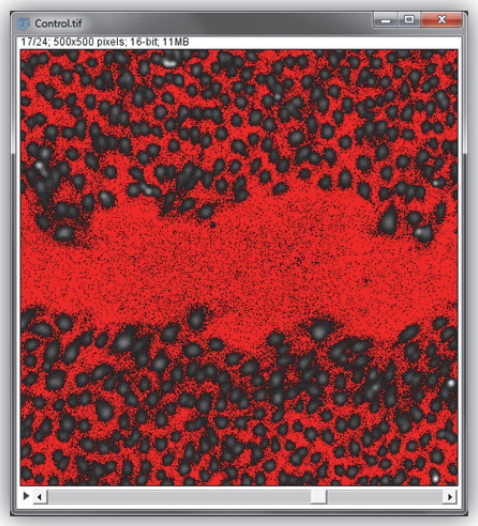
5. In the **Subtract Background…** dialog box tick the **Preview** box and try different **Rolling Ball Radius** sizes. The larger the rolling ball the larger the objects that will be left behind but the less dramatic the background subtraction. For this image a value of 300 works well.



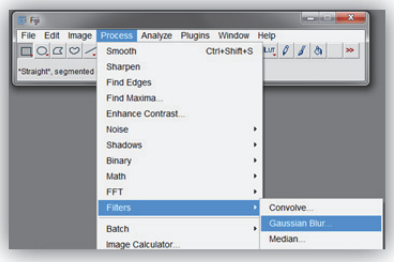
6. Press **OK** and press yes on the next window that opens asking if you want to process all planes.



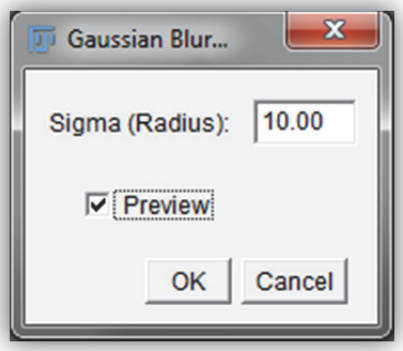
7. If you now go to plane 17 and try the threshold again you will see that while it can now be evenly applied it selects a lot of space between all the cells. This is because this information was removed by the background subtraction step.



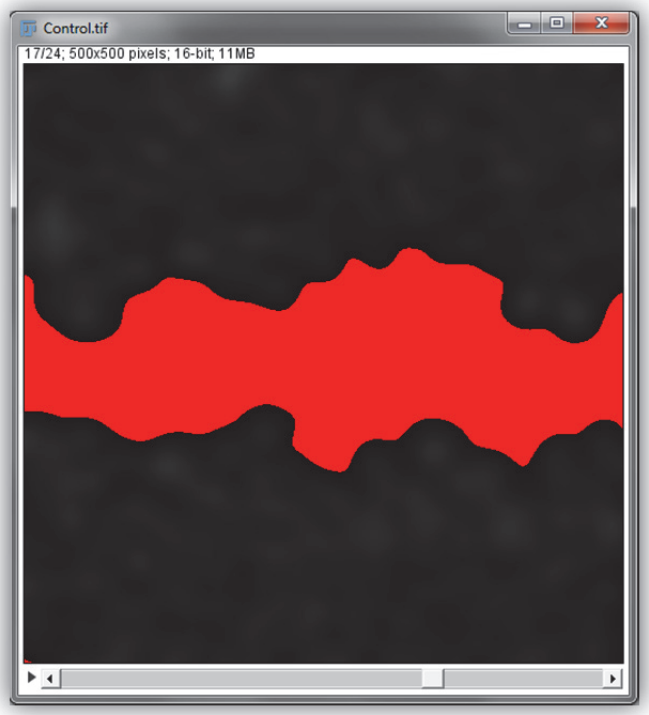
8. Reset the threshold and got to **Process → Filters → Gaussian Blur…**



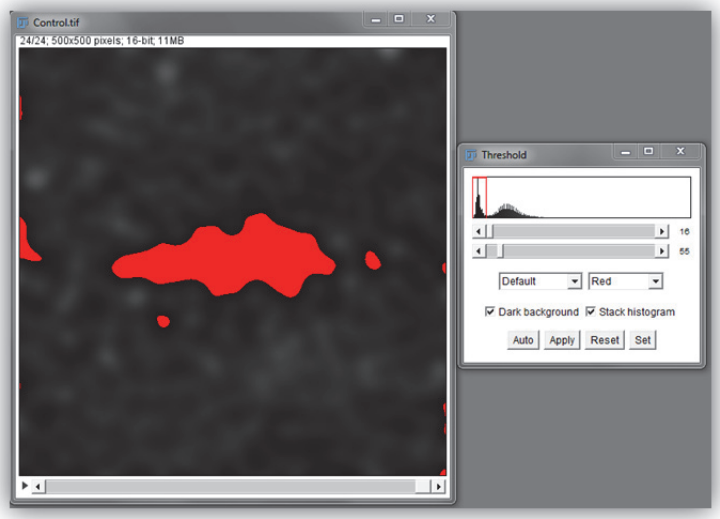
9. Set the filter size to **10** and press **OK.** Once again say yes when asked to process all planes.



10. Now you should be able to evenly threshold the wound on plane 17.



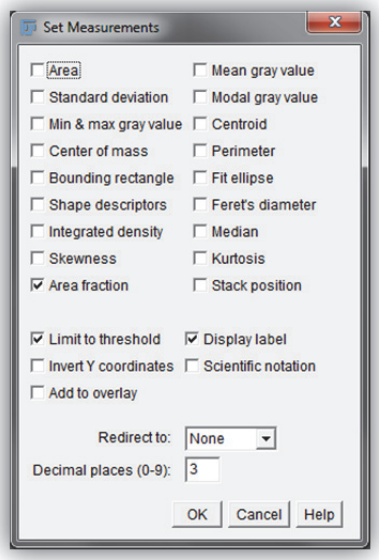
11. Now navigate to the last plane and set the threshold to select the wound. Make sure the **Stack Histogram** box is ticked otherwise things can go funny. A range of 16 to 55 works well.



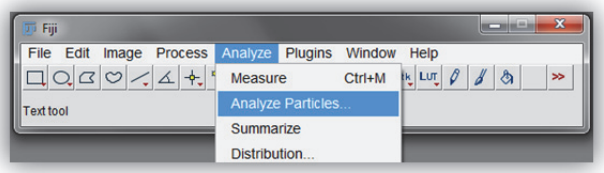
12. Play through the stack and notice that the threshold is now selecting the wound in all the planes of the image evenly.

Measuring Scratch Area

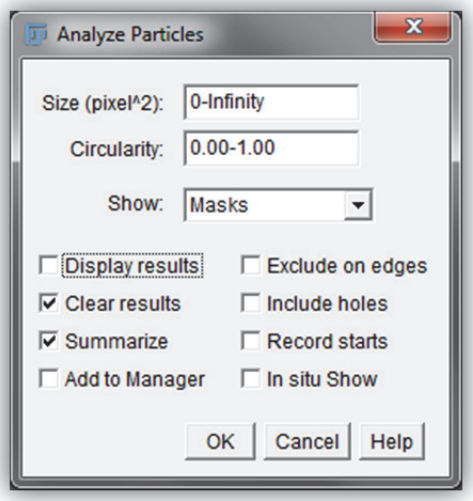
1. Go to **Analyse → Set Measurements…** and configure it to measure the **Area Fraction** and make sure it is set to **Limit to Threshold.** Press **OK**



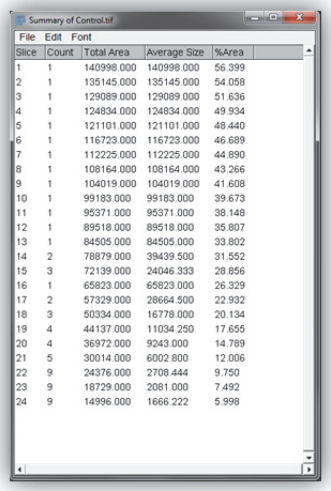
2. Go to **Analyse → Analyse Particles…**



3. In the **Analyse Particles** dialog Tick the **Clear Results** and **Summarise** results boxes. Also make sure the **Show:** value is set to **Masks** as we will use this later. Press **OK.** Once again agree to process all planes.

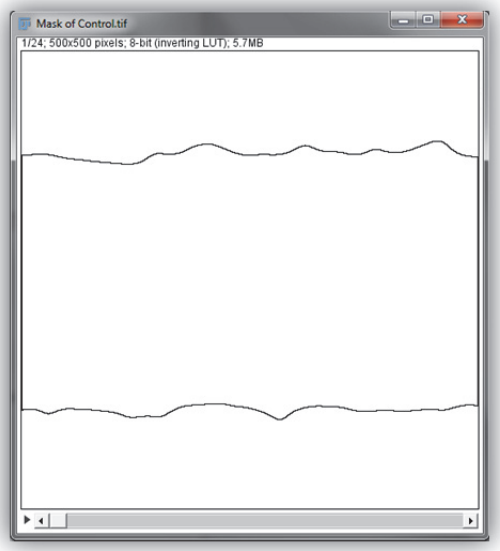


4. You will get the result for all the planes in a summary table

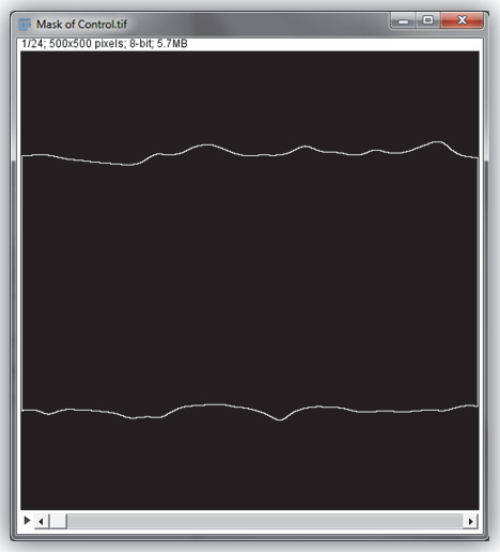


Creating Overview Movie of Result

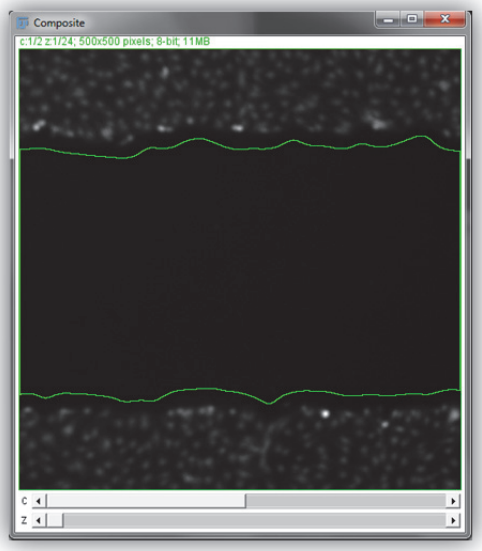
1. To create an outline movie of the scratch select the output mask and go to **Process → Binary → Outline** as before. Agree to processing all planes.



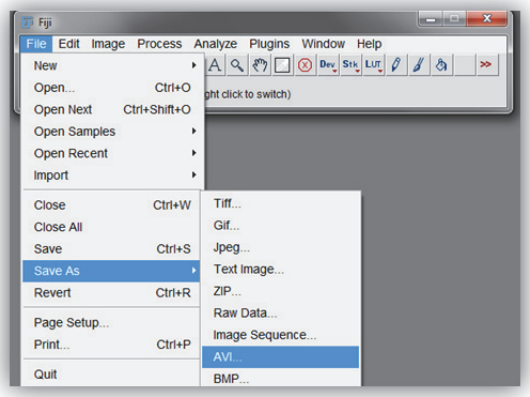
2. Invert the LUT



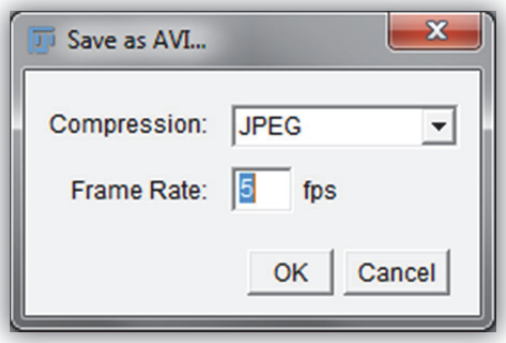
3. Reopen **Control.tif**, convert it to 8 bit and merge the outline and 8 bit converted version



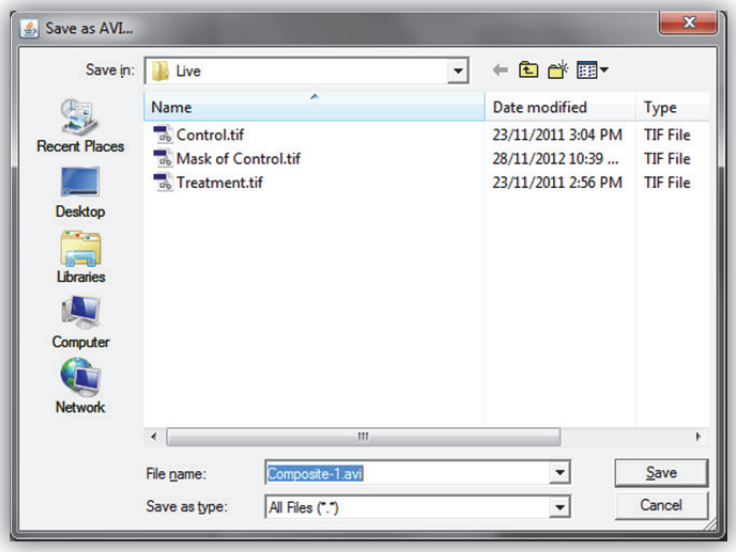
4. Convert the Composite image to RGB and go to **File → Save As → AVI…**



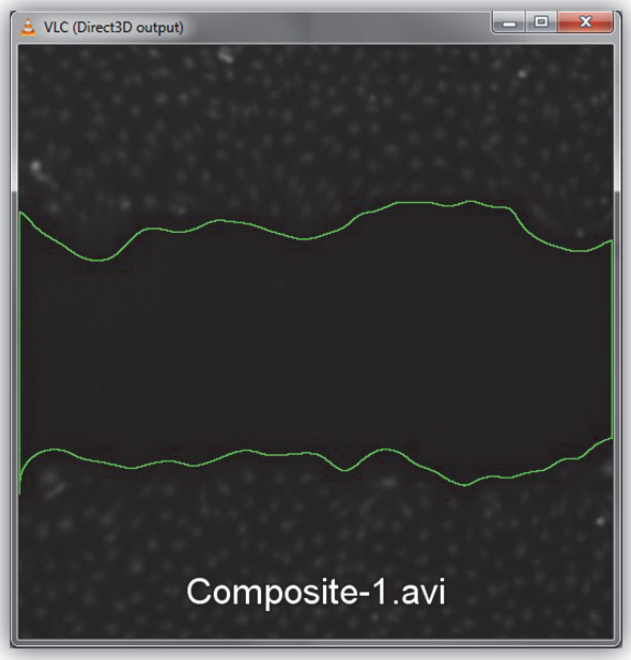
5. In the **Save as AVI…** dialog set the **Compression** to **JPG** and the **Frame Rate** to **5**



6. Choose somewhere to save the file and press **Save**



7. You should end up with a movie that can be played in your favourite video player or embedded into a presentation



Euclidean Distance Measurements

Aim

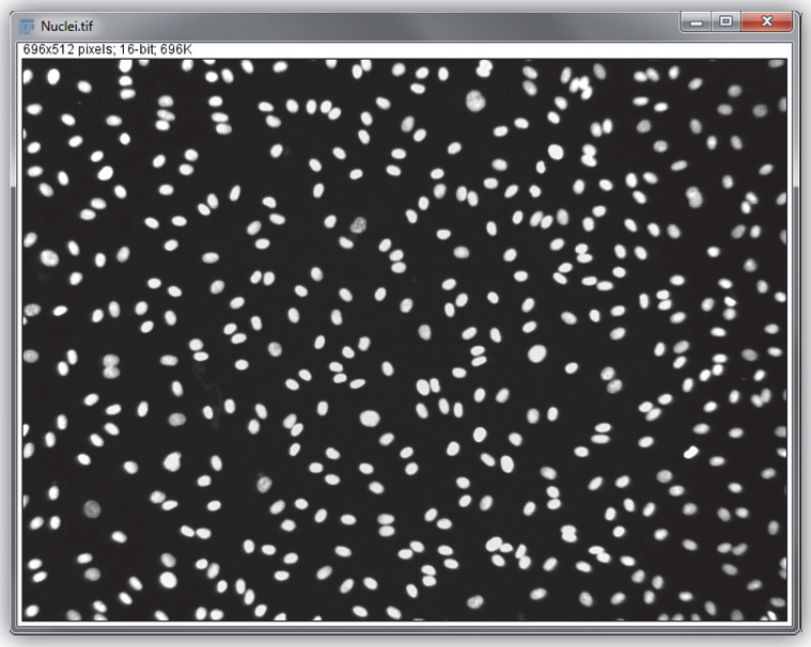
This technical note will show you how to measure the average distance between objects, in this case nuclei. Euclidean distance is a simple measurement that uses intensity as a measure of distance. Individual objects are created as binary objects. A single intensity unit is added for each pixel away from the original object.

Additionally this technique can be used to measure the distance between two separate stains. For example the distance vessels are in a tissue from hypoxic regions.

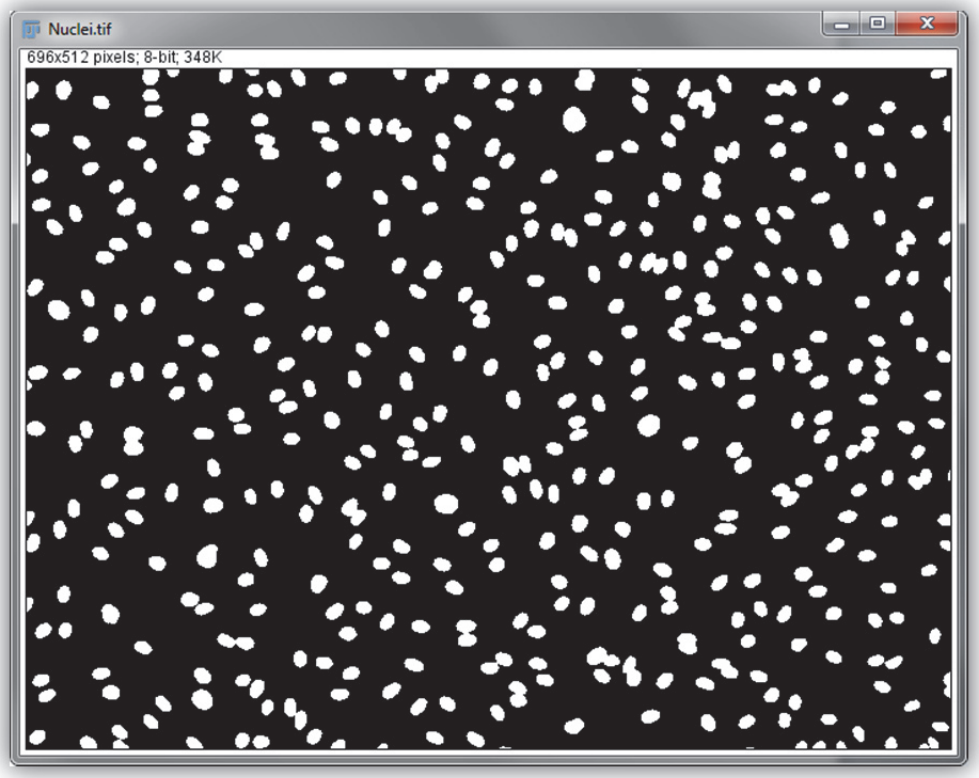
Measuring Euclidean Distance – Object Distribution

Creating a Distance Map

1. Open **Nuclei.tif** from the **Demo Images\Nuclei** folder

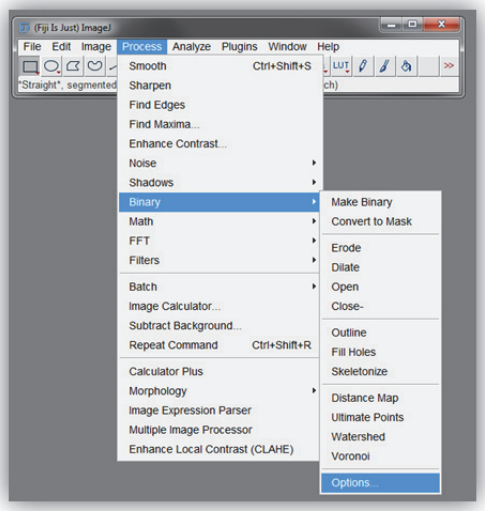


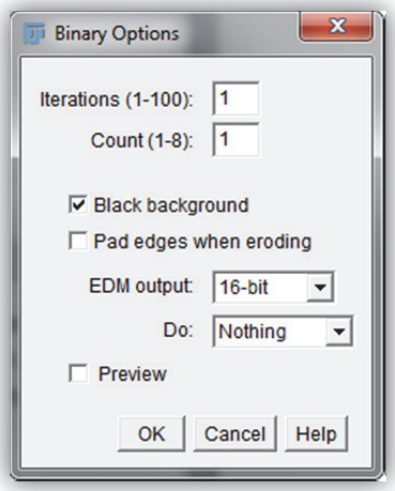
2. Apply a threshold and generate a binary image of the nuclei



3. To avoid potential issues in subsequent analysis the Euclidean distance map that is generated must be set to a bit depth of 16. The default of 8bits will only allow you measure a maximum distance of 256 pixels. A 16 bit EDM will allow a maximum distance of 65,536.

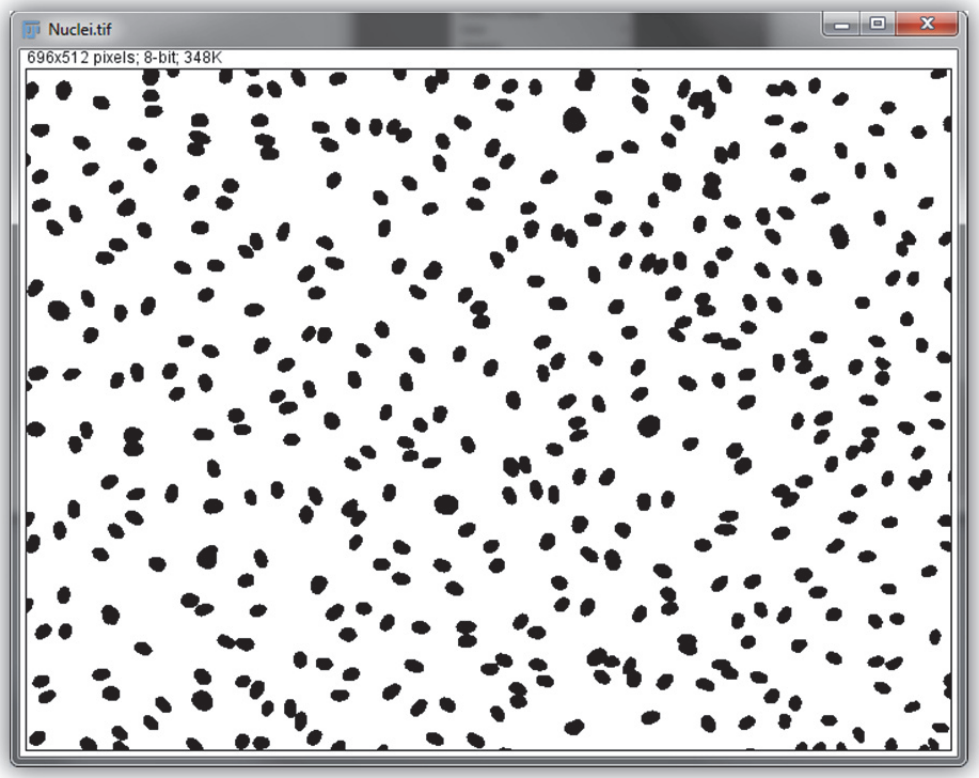
   Go to **Process → Binary → Options** and configure the EDM Output to 16 bit



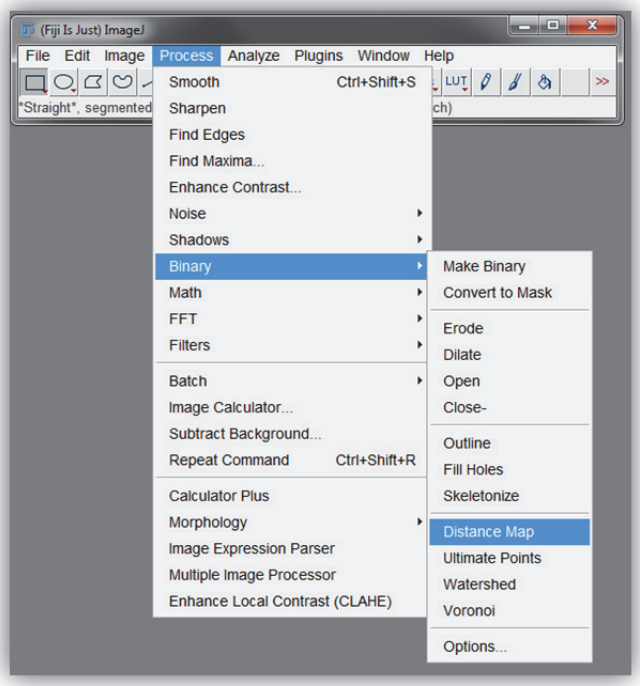


4. To generate a distance map of the distances between cells the image needs to be inverted. Go to **Edit → Invert**

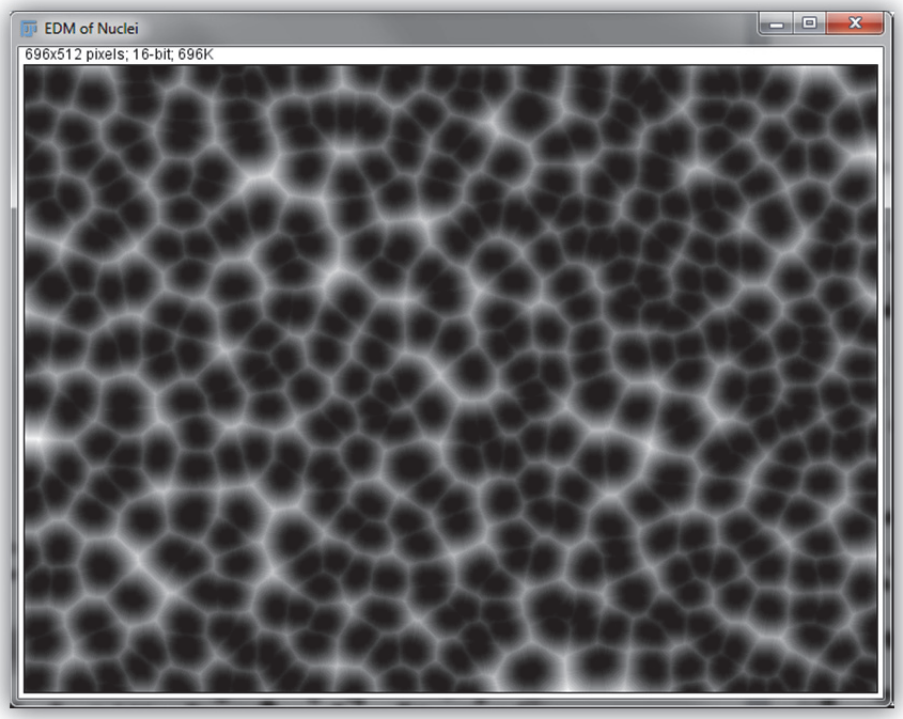
   If the image isn't inverted the resulting distance map will measure the stance from the edge to the centre of each nuclei



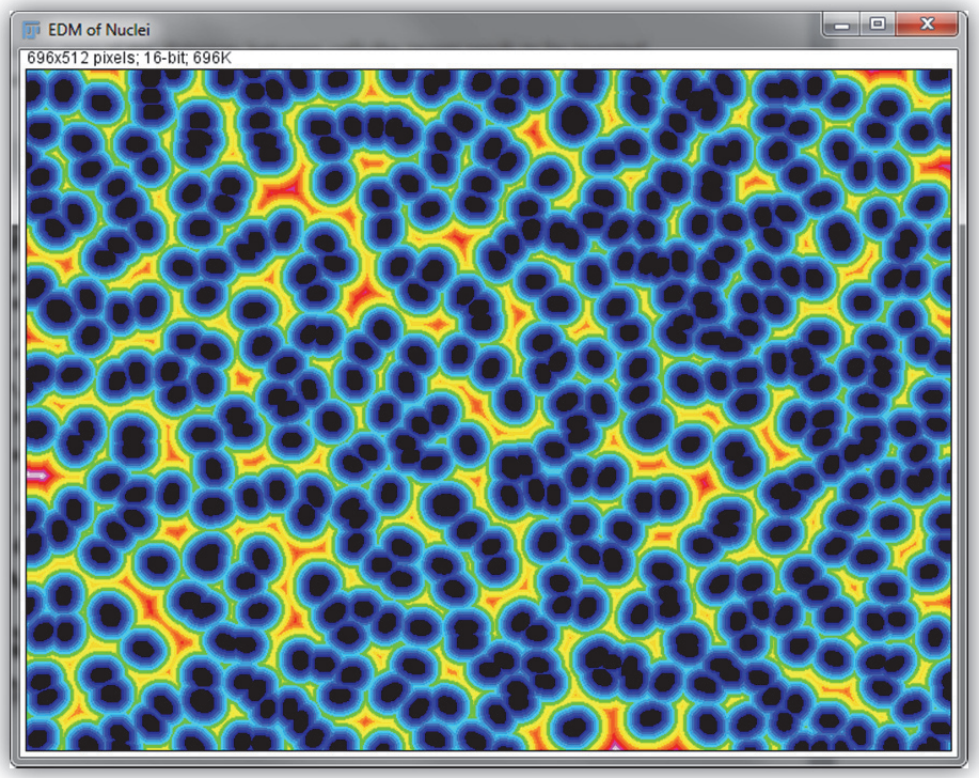
5. Go to **Process → Binary → Distance Map**



6. The resulting image is a distance map that shows distances from the original seed points (the nuclei) as an increase in intensity.

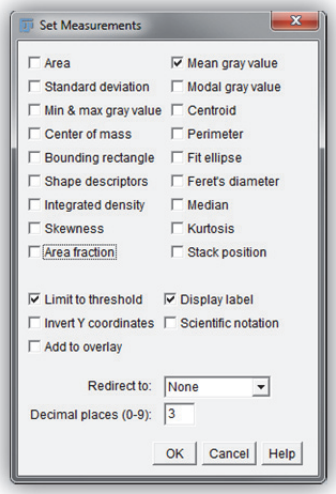


7. To make it look pretty press the **LUT** button and select 16 colours

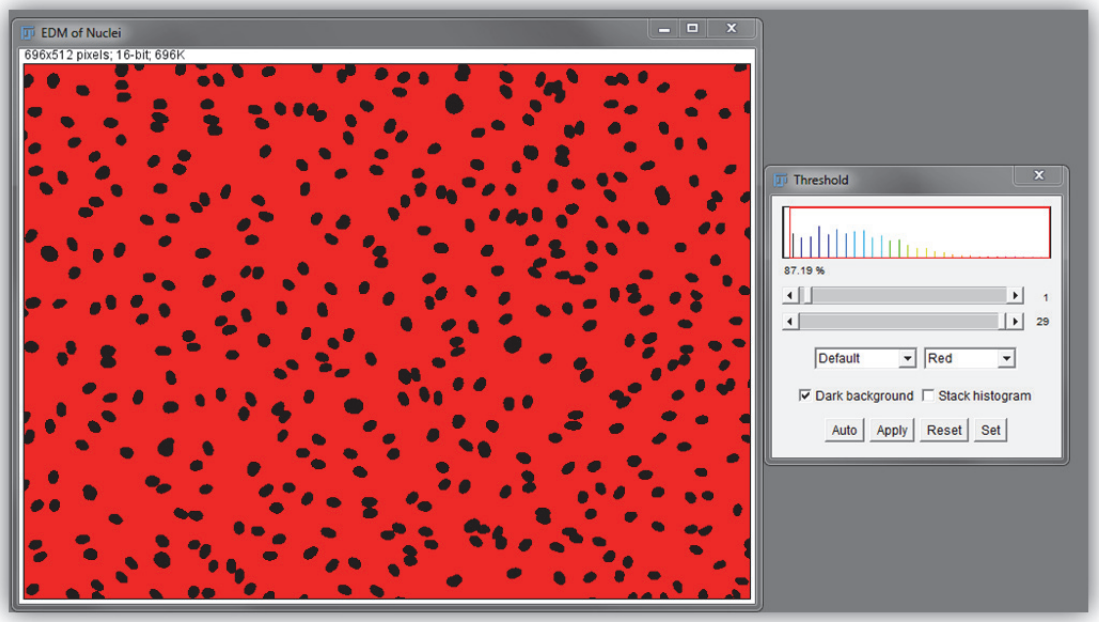


Measuring Distances

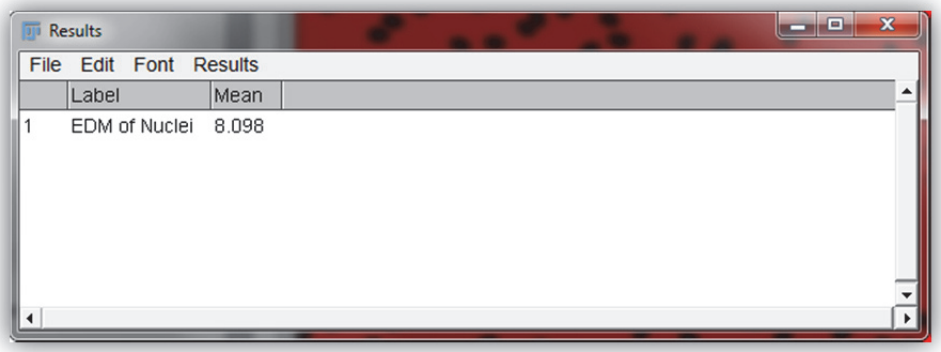
1. To measure the distances in the image firstly configure the measurements to measure the mean gray value and limit the measurement to the threshold



2. Threshold the EDM image so as to select only the background and leave the nuclei free (so all values from 1 and up)



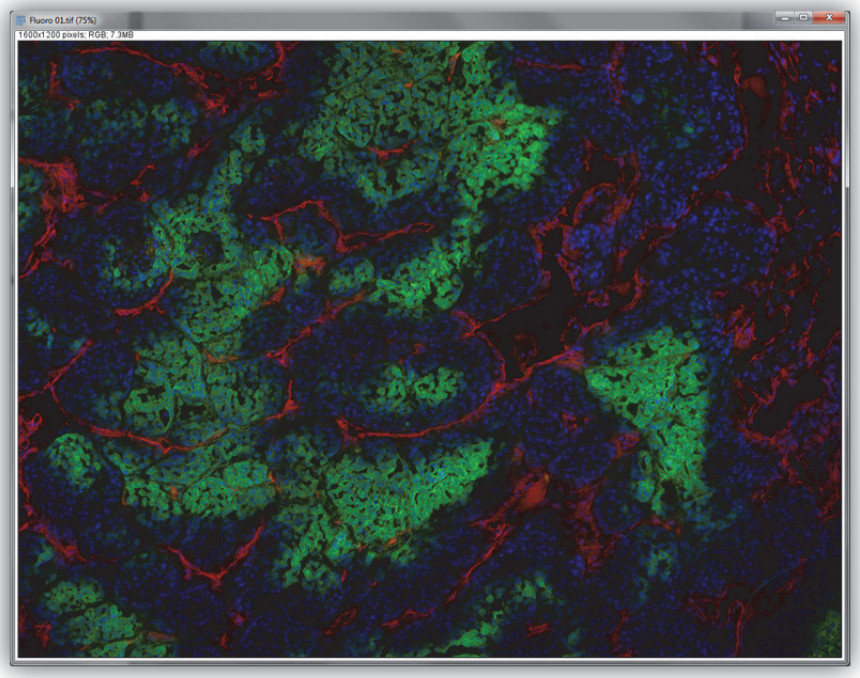
3. Measuring the image will give a mean intensity result that can be interpreted as the average distance. In this example the average distance between the nuclei is 8.098 pixels. If you know the size of a pixel in your given image you can then translate this value into a calibrated distance.



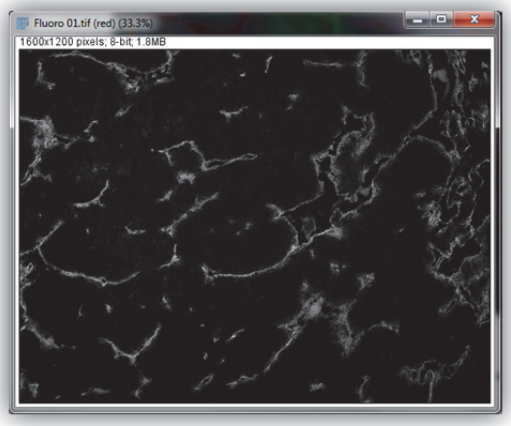
Measuring Euclidean Distance – Distance between Different Objects

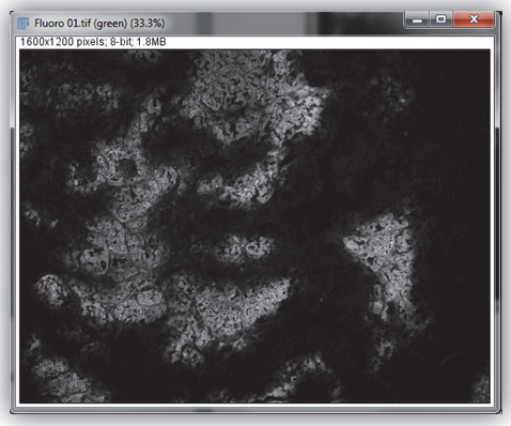
Initial Image and Masks

1. Open **Fluoro 01.tif** from the **Demo Images\Widefield\Fluorescent Measurement** folder

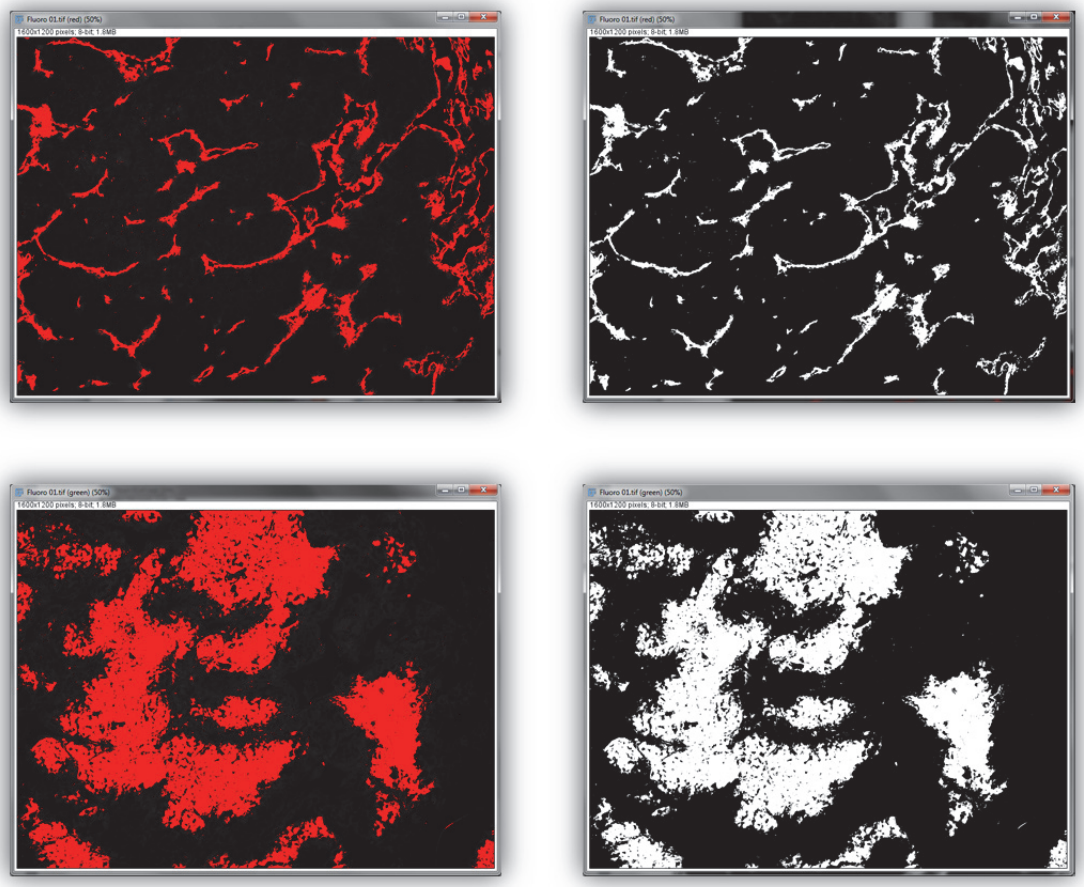


2. Separate the channels using the **Image → Colour → Split Channels** command. Only the **Red** (blood vessels) and **Green** (hypoxia) channels are required for the rest of the analysis.





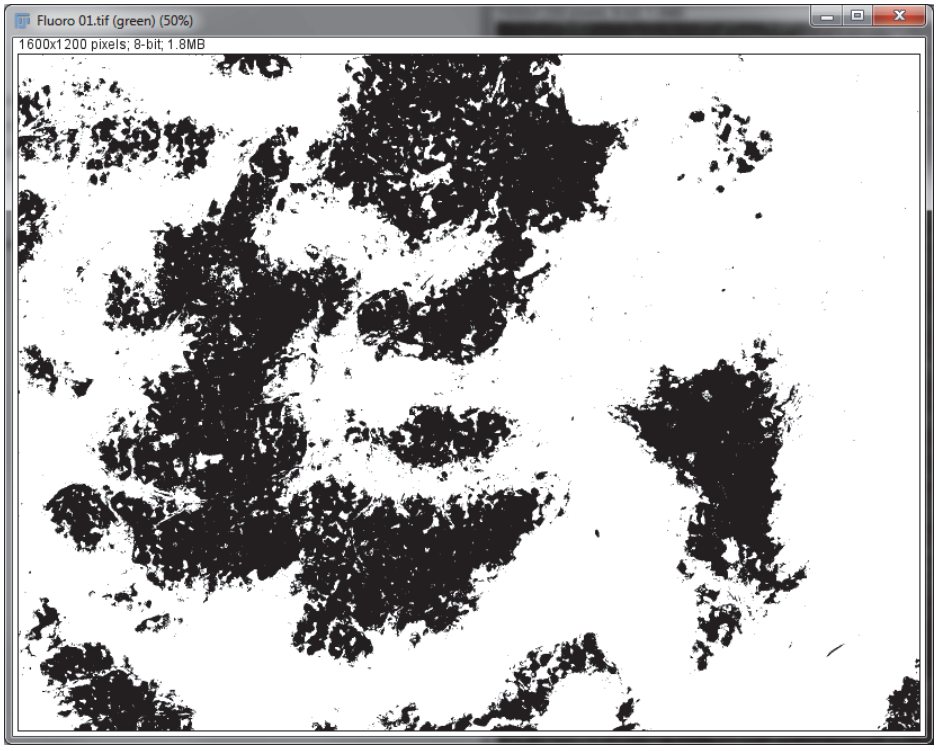
3. Apply a threshold to each channel to generate binary masks. The **default** auto threshold is fine for the **Red** channel and the **Li** auto threshold is fine for the **Green** channel



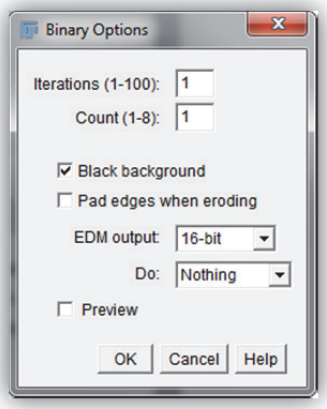
Generating an EDM Map

To measure the distance of objects from different channels from each other we first have to make one of the channels a mask and one of them a distance map. In this example we want to measure the distance of the blood vessels from the hypoxia region. For this we need a binary mask of the blood vessels (already created in previous step) and a distance map representing the distances away from the edge of the hypoxia regions.

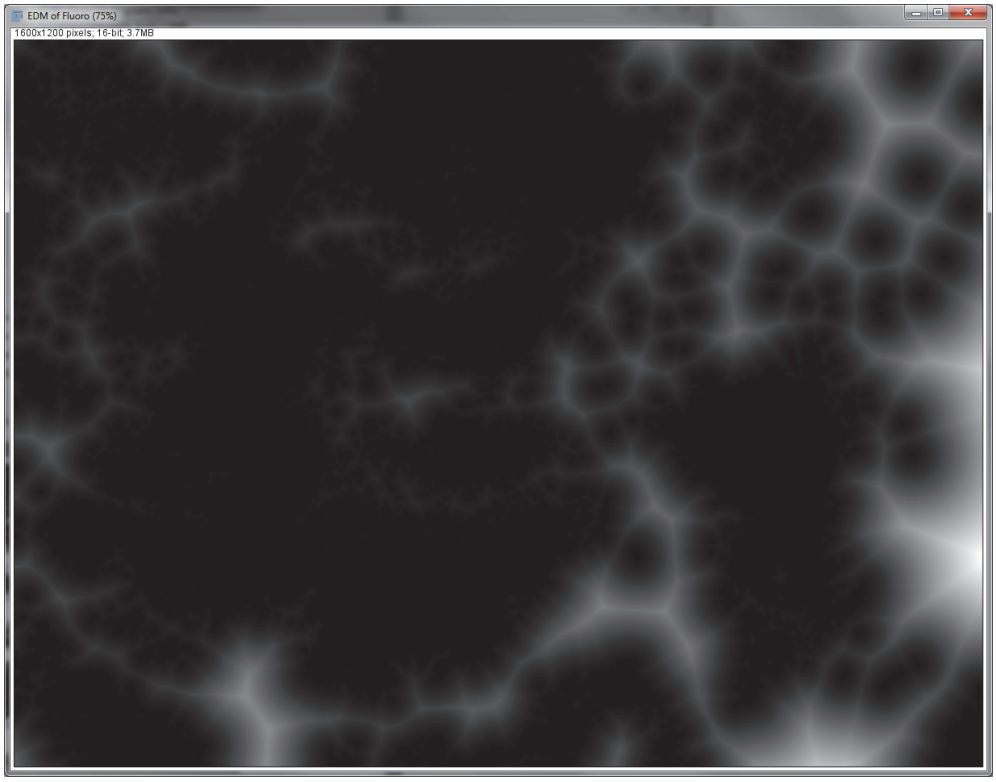
1. Select the binary image of the hypoxia stain and invert it (**Edit → invert)**



2. Check that the EDM Output is set to 16 bit (**Process → Binary → Options)**



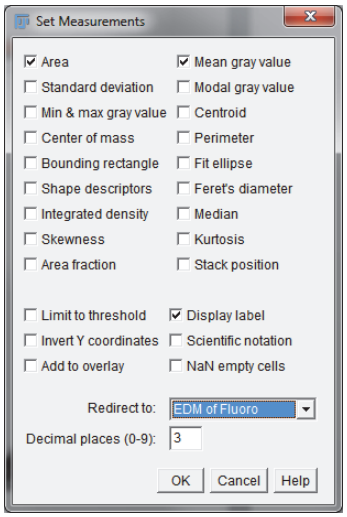
3. Generate a distance map by going to **Process → Binary → Distance Map**



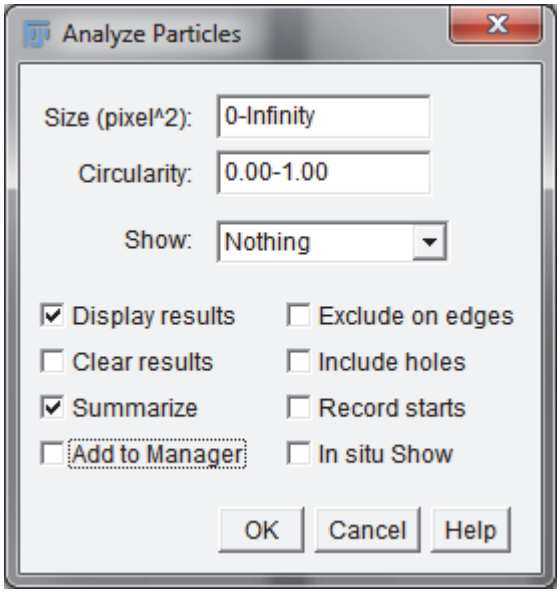
Measuring the Distances

To measure the average and individual distances of the blood vessels from the hypoxic areas we need to redirect the measurements from vessel binary when it is measured to the EDM Map we just generated.

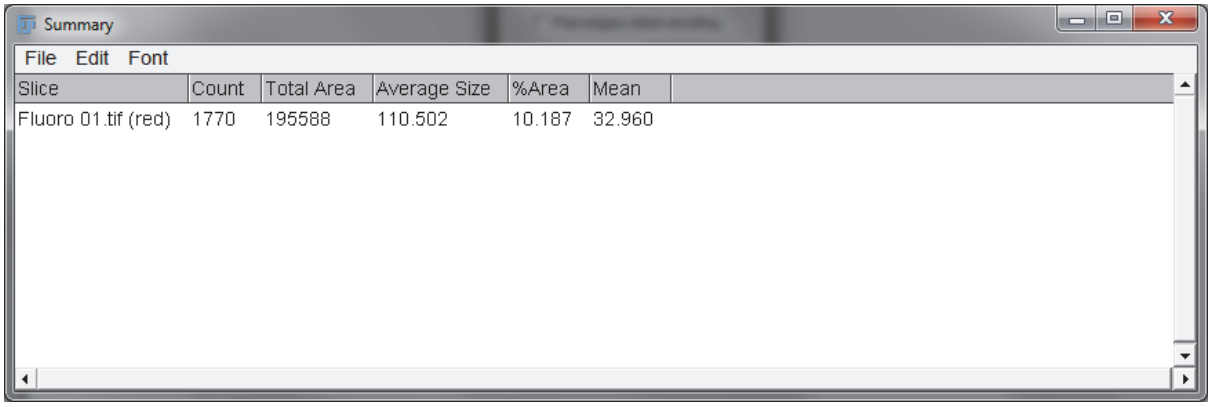
1. Go to **Analyse → Set Measurements** Select Area and Mean Grey Value. Make sure the **Redirect To** is set to **EDM of Fluoro** (the distance map)

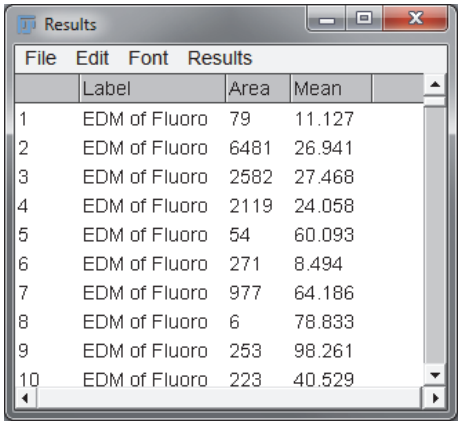


2. Select the binary mask of the vessels and go to **Analyse → Analyse Particles.** Configure it to **Display Results** and **Summarise**



3. The resulting tables will give a summary of all the measurements or detailed information about each vessel. The mean intensity value represents the number of pixels from the object. So in this example on average the vessels are 32.9 pixel away from the hypoxic regions





Skeletonize and Branching

Aim

This technical note will show you how to create a binary skeleton of a binary mask. This technique is useful for measuring the length of rod shaped, or long skinny, objects such as bacteria, mitochondria or nanoparticles. It can also be used to measure the number of branches in a networked structure.

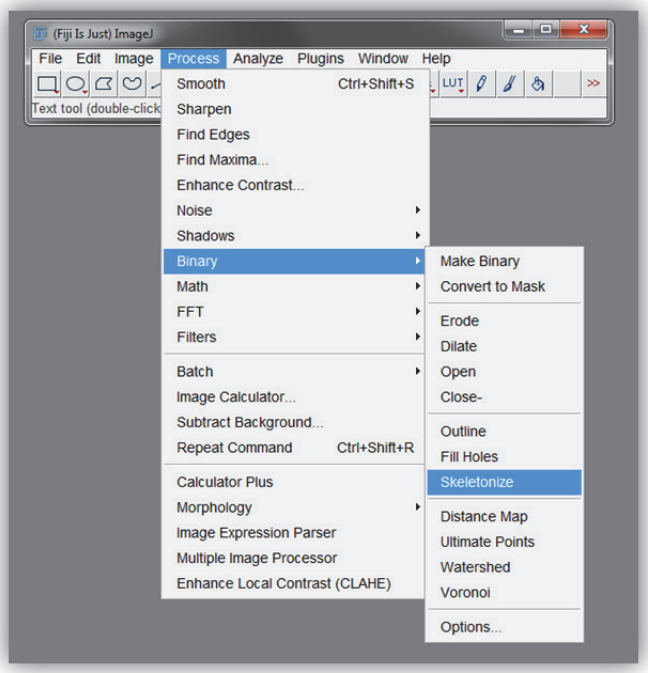
Measuring Length using Skeleton Masks – Discrete Objects

Generating Masks

1. Open **Bacteria.tif** from the **Demo Images\Widefield\Bacteria** folder

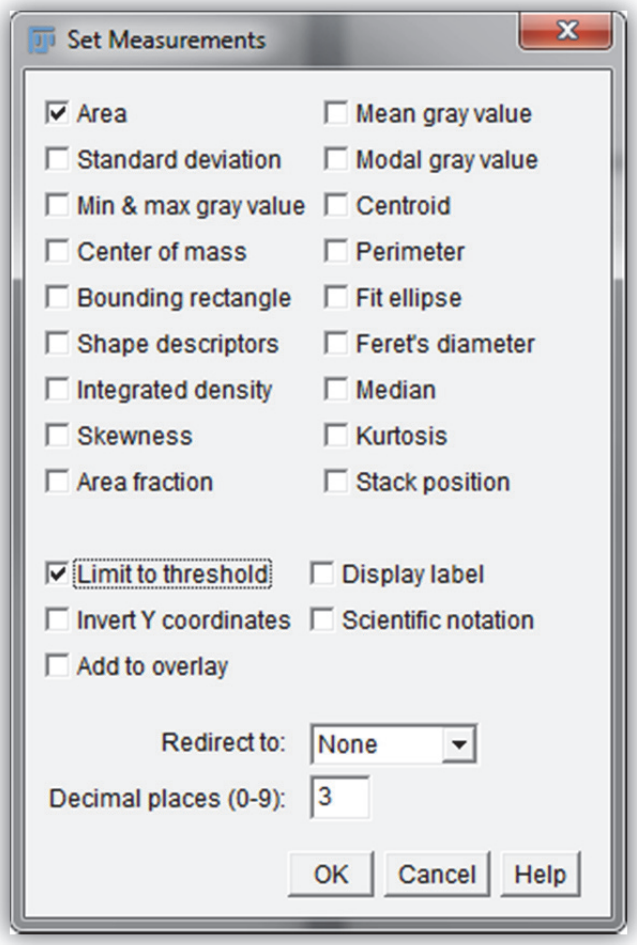
2. Apply a threshold to the image and generate a binary mask. Make sure the thresholds do not bleed into each other so as to select individual bacteria

3. Go to **Process → Binary → Skeletonize**



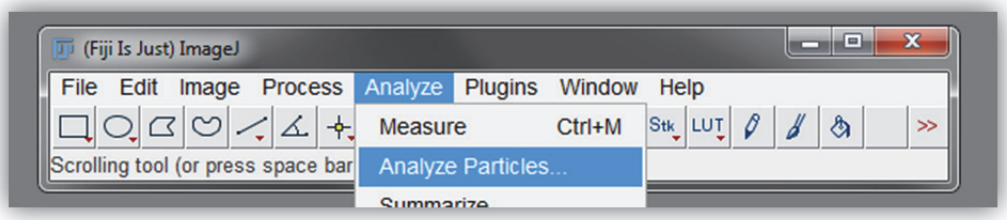
Measuring Masks

1. To measure the length of the lines set the measurements to measure Area and limit the measurements to the threshold. As the lines are a single pixel wide the area will represent the length.

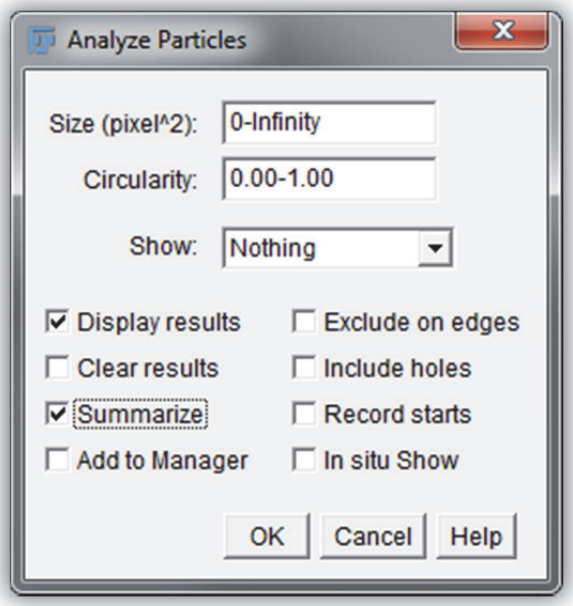


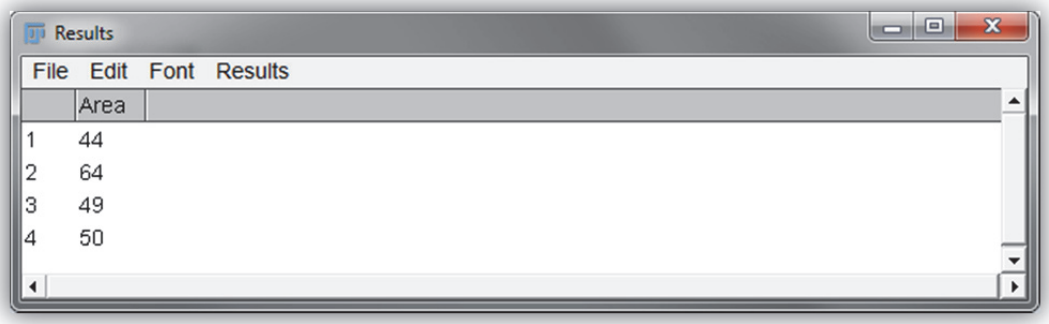
2. Place a threshold on the image

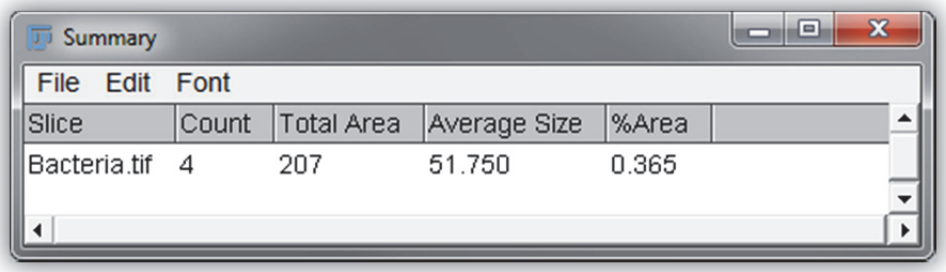
3. To measure this time we will use a slightly different measurement module. This module will allow us to get values for each object in the image, as opposed to a global average we have been getting before. Go to **Analyse → Analyse Particles**



4. Configure the resulting dialog box as follow. This will measure the thresholded objects and display two results tables. One that shows the values for each object and one that shows summary information about all the objects.





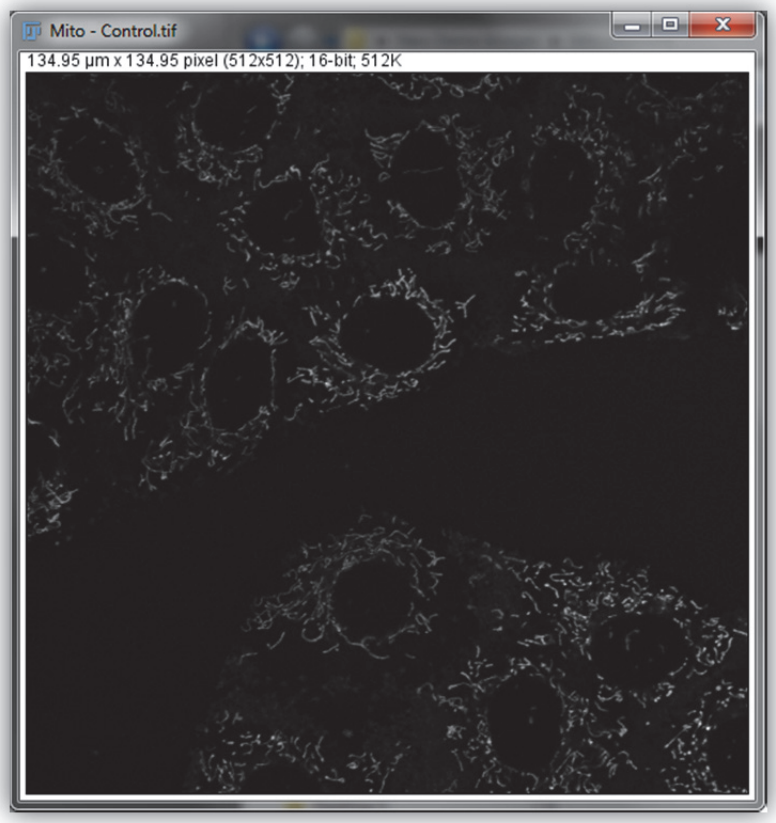

Summary
File Edit Font
Slice Count Total Area Average Size %Area
Bacteria.tif 4 207 51.750 0.365

Measuring Length using Skeleton Masks – Connected or Networked Objects

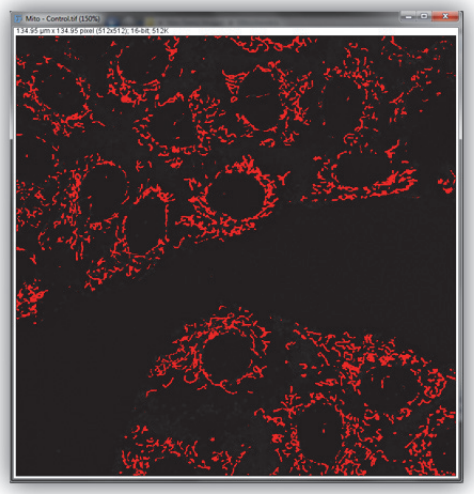
The same principals above can be applied to connected objects or networks. The resulting distance measurement is not specific to a single object but instead is a value representing the complete network.

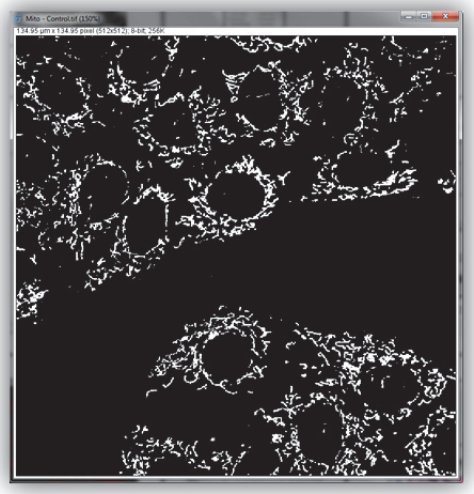
Generating Skeleton Mask

1. Open **Mito – Control.tif** from **Demo Images\Confocal\Mitochondria**

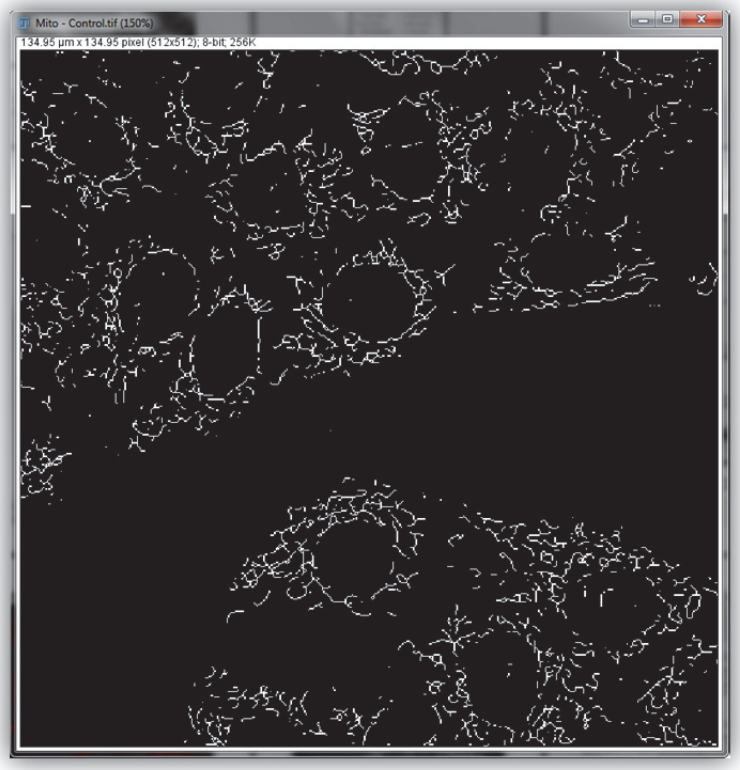


2. Use the **Default** setting to place and apply a threshold to the mitochondria (**Image → Adjust → Threshold**)



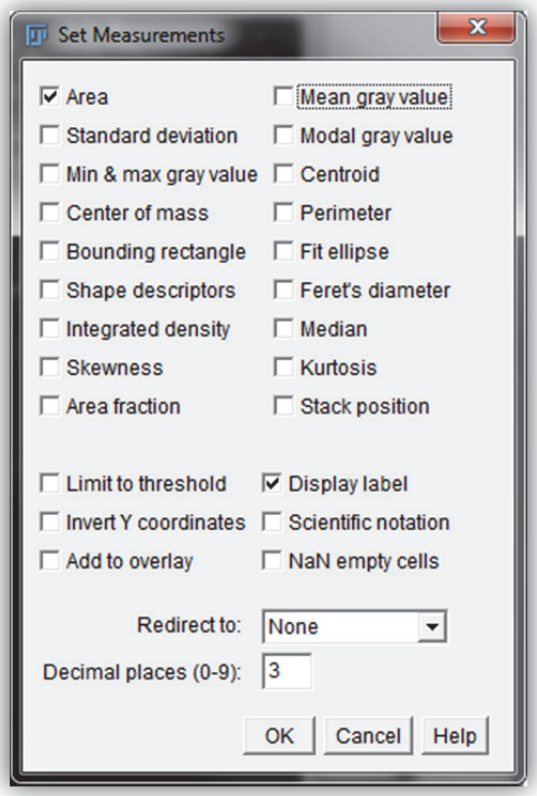


3. Convert the binary mask to a skeleton using the **Process → Binary → Skeletonise** command

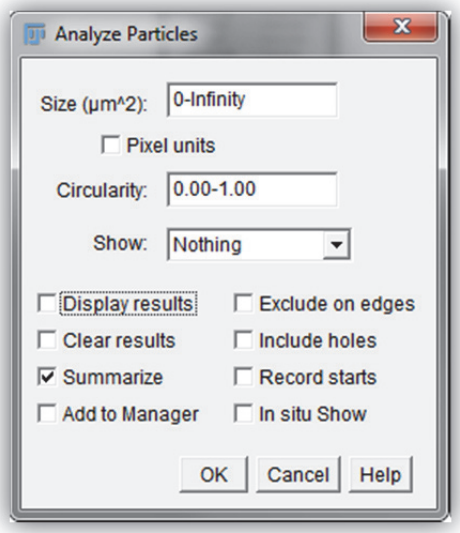


Measuring the Mask

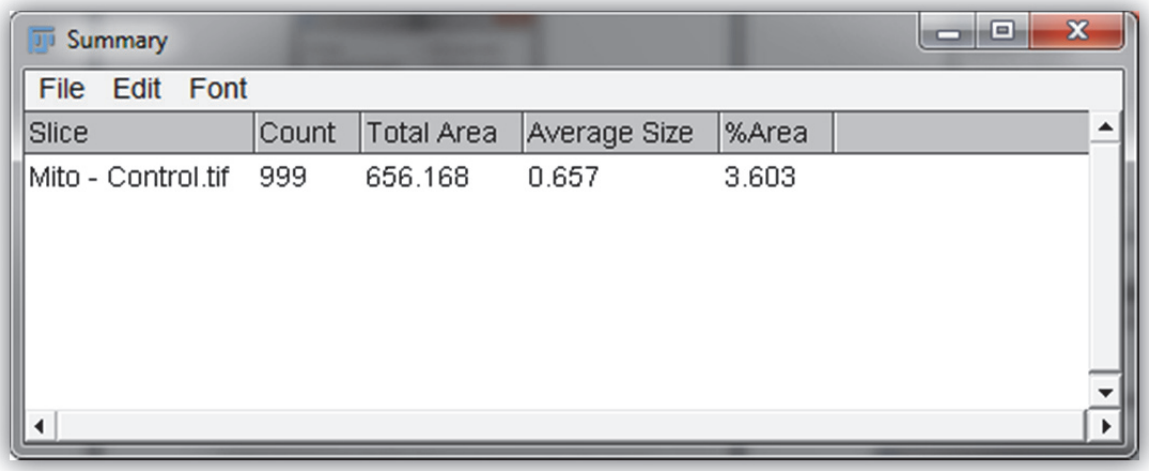
1. Go to **Analyze → Set Measurements…** and configure the settings to measure **Area** only



2. Use the **Analyze → Analyze Particles** command to get a summary of the lengths of the mitochondria



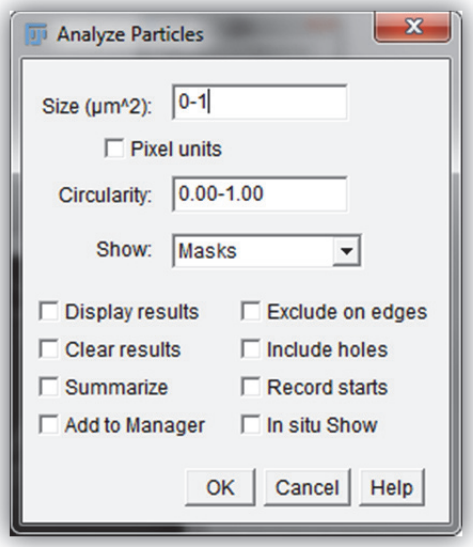
The average size shows the average length of all the mitochondria in the image. If information on individual cells is required an additional whole cell stain would need to be included.



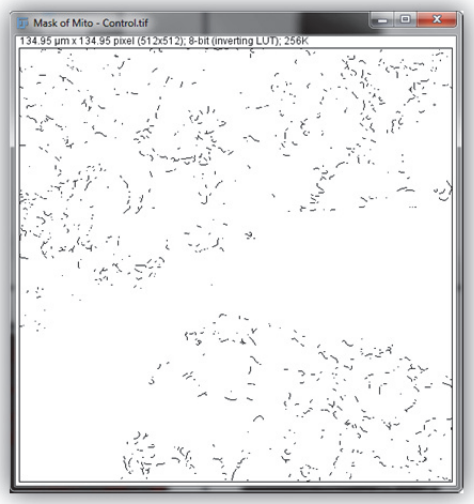
Generating color infographics of the results

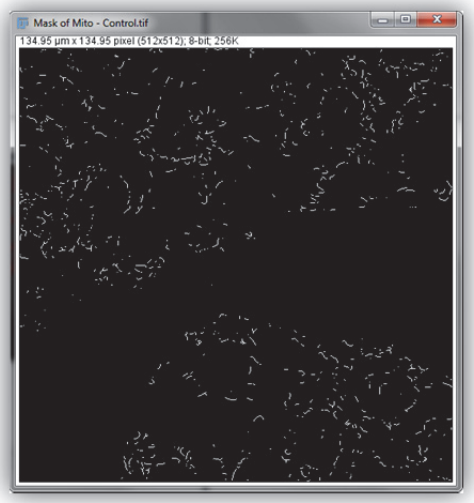
Having a color overview image of result can help show the result more clearly. In this example one will create a color overview image that shows the short and longer mitochondria in different colors.

1. Select the final skeletonize mask and use the **Analyze Particles** command again, but this time set a size range of 0-1 and set **Show:** to **Masks**, there is no need to have any of the boxes ticked

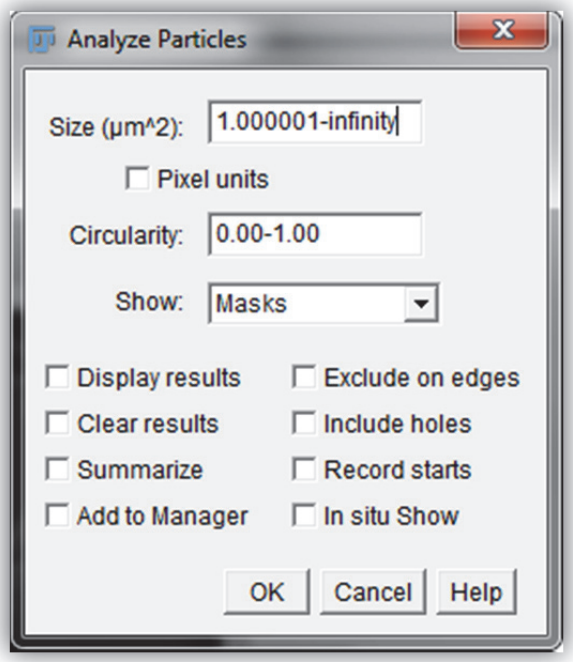


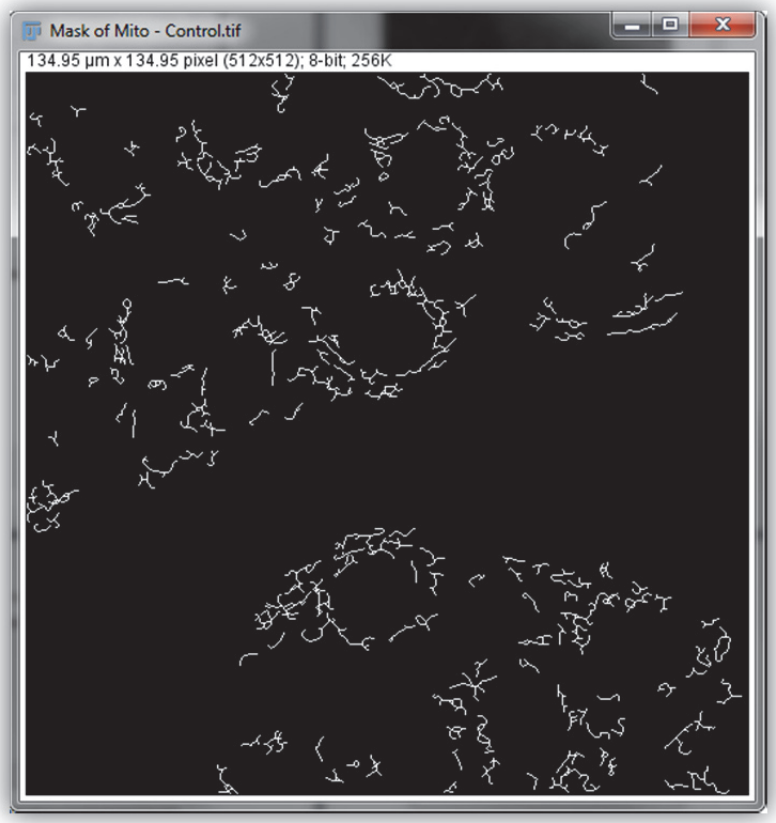
2. The resulting mask needs to be inverted to give white objects on a black background. This mask represents all the short objects (below the completely arbitrary cut off of 1um)



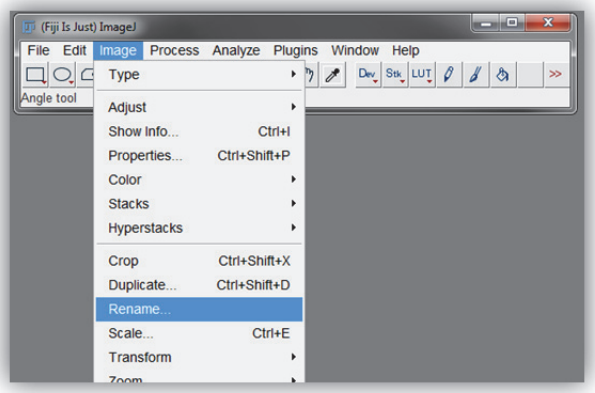


3. To generate a mask for the longer (>1um) objects follow the steps above but this time set the size cut off to 1.000001-Infinity. The high precision number removes the chance of double counting an object as both small and large (i.e. there is no overlap between the two measurements)

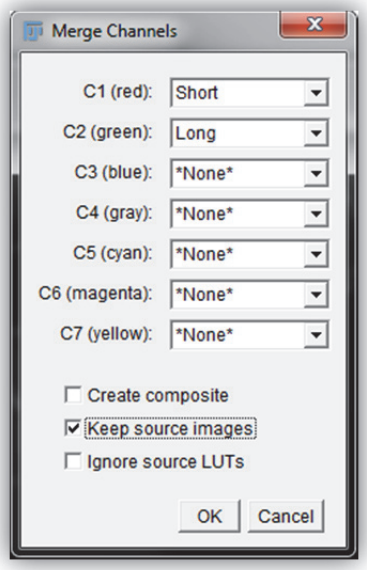


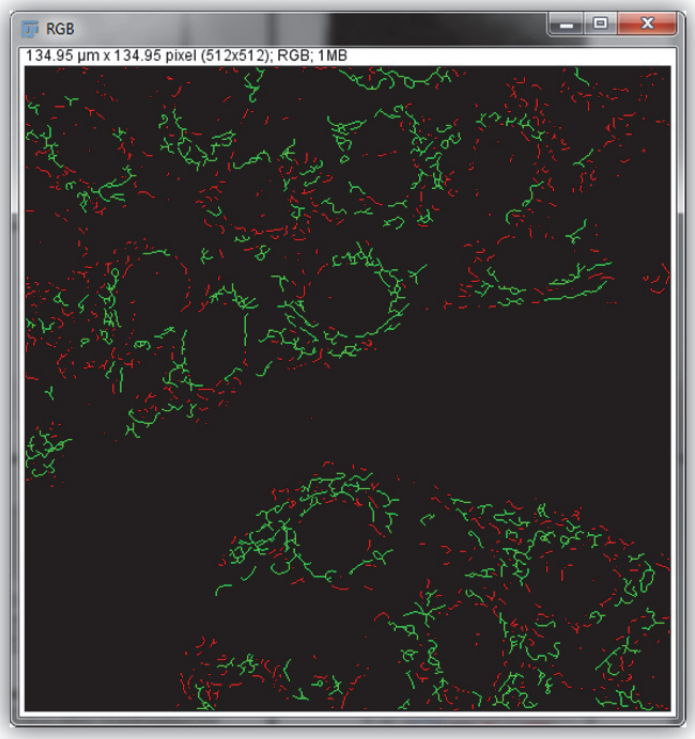


4. Both masks will have the same name (**Mask of Mito – Control.tif**). To make the next step easier it is a good idea to rename the images to different names (e.g. Short and Long)

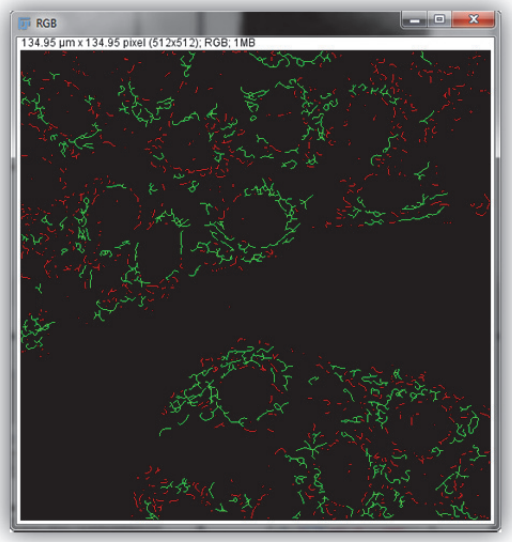


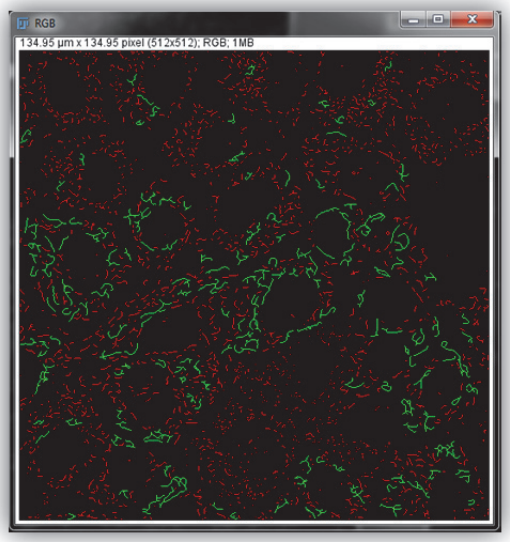
5. Merge the two resulting masks





6. Repeat all the previous steps for **Mito – Treatment.tip** and compare the results.





Summary

File Edit Font

Slice	Count	Total Area	Average Size	%Area
Mito - Control.tif	999	656.168	0.657	3.603
Mito - TreatmentI.tif	2302	867.365	0.377	4.763

Image Stitching

Aim

There are some situations where you need to capture a large field of view but do not want to sacrifice resolution by using a low powered objective. Some microscopes have the ability to automatically create montage images by controlling the stage but if you only have a manual stage there is are two plugins in Fiji to either manually or automatically create a montage from pictures you capture.

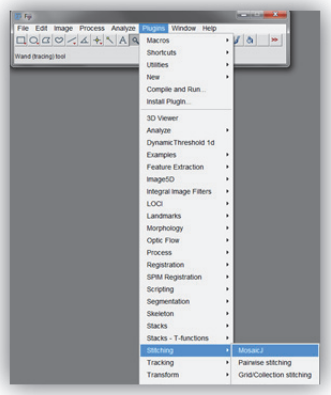
Tips for Capturing Your Images

When you capture your images to create a montage make sure they overlap by about 10-15% so the plugin can have some common features to reference. If your sample is something like sparsely dispersed cells you may need to make the overlap higher to remove any potential errors in the final montage.

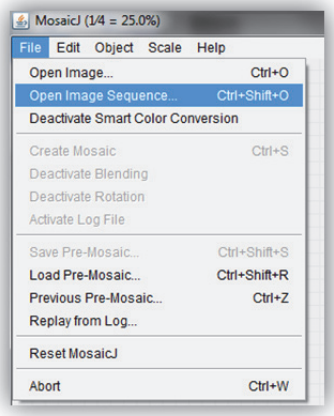
Manually Creating a Montage

You can assemble a montage manually, like a jigsaw, using the MosaicJ plugin. This works well for small numbers of images but can be very intensity for montages made up of many images.

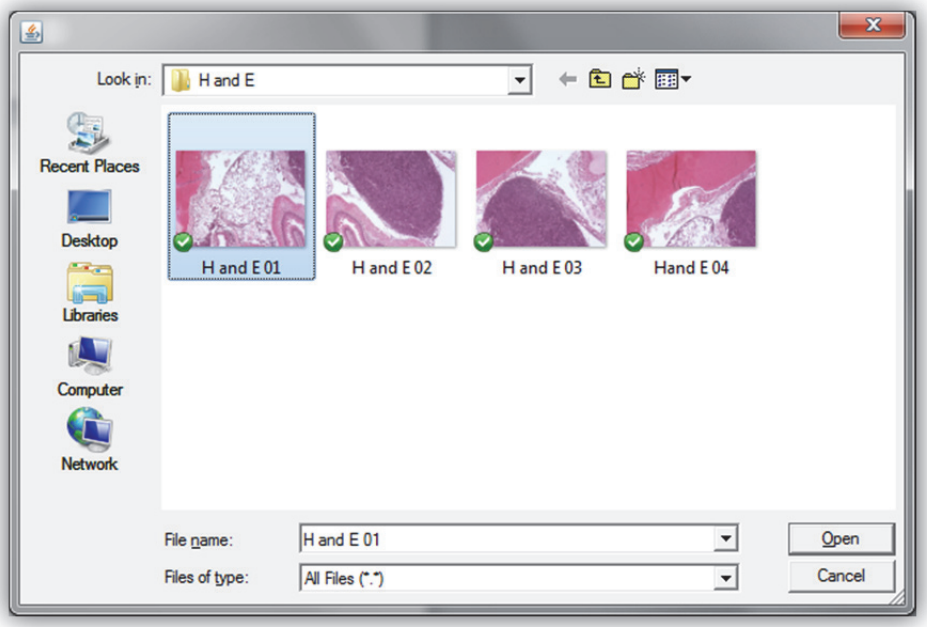
1. Go to **Plugins → Stitching → MosaicJ**



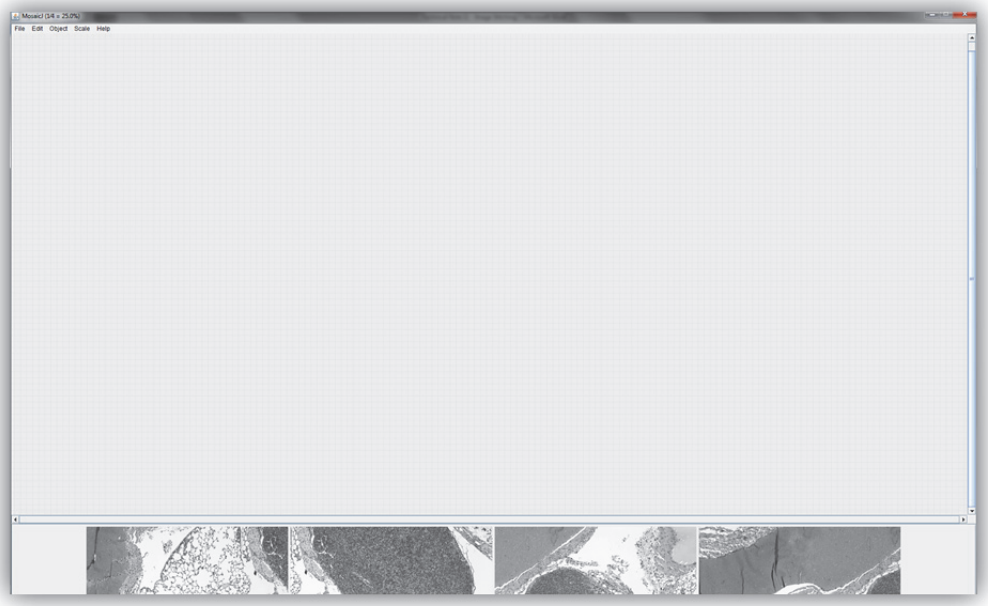
2. In the **MosaicJ** window go to **File → Open Image Sequence…**



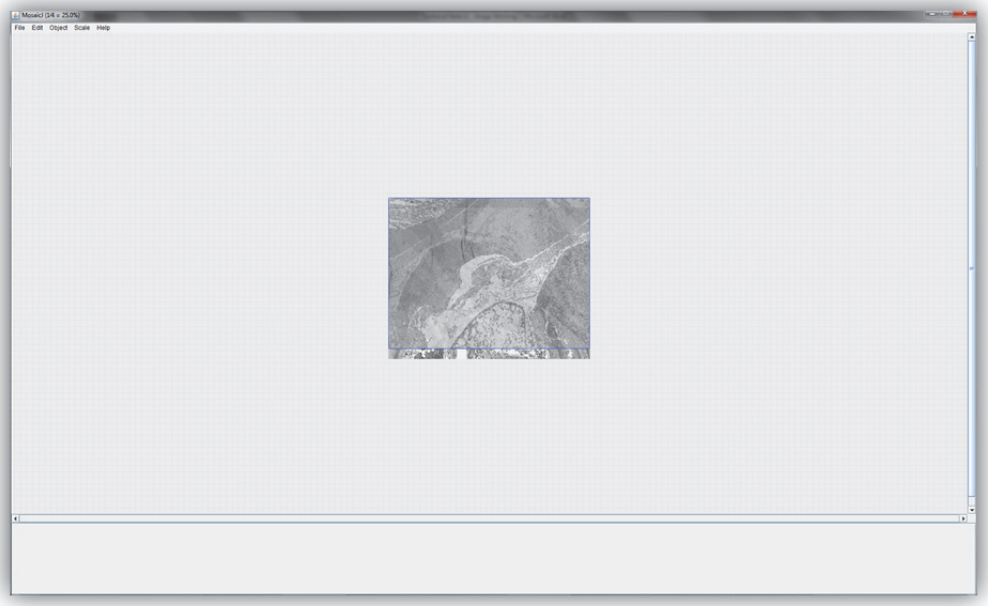
3. Go to the folder **Demo Images\Widefield\H and E** and highlight the file **H and E 01.tif** and press **Open.**



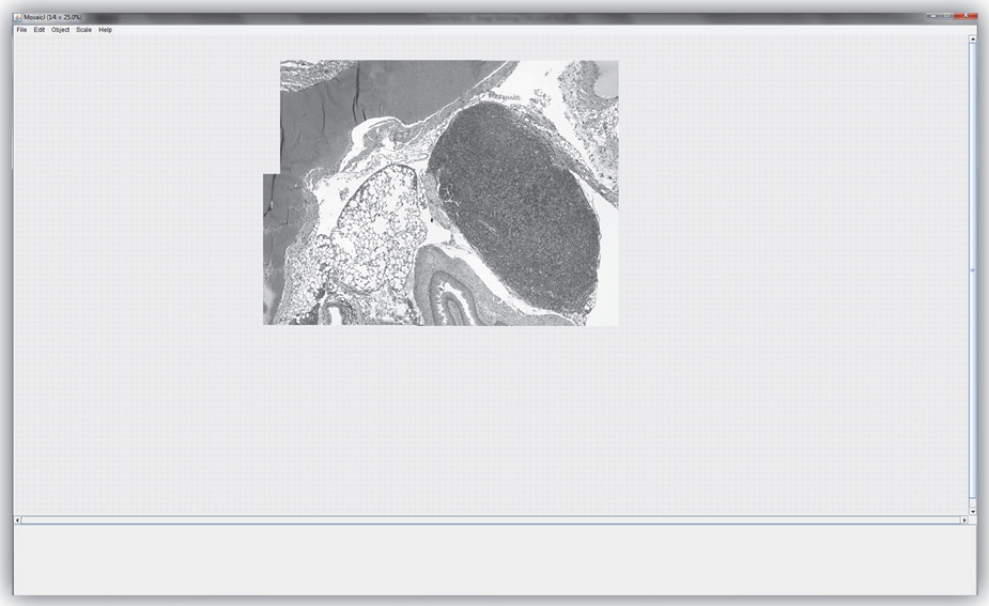
4. The four H and E images should be placed in the bottom of the **MosaicJ** window.



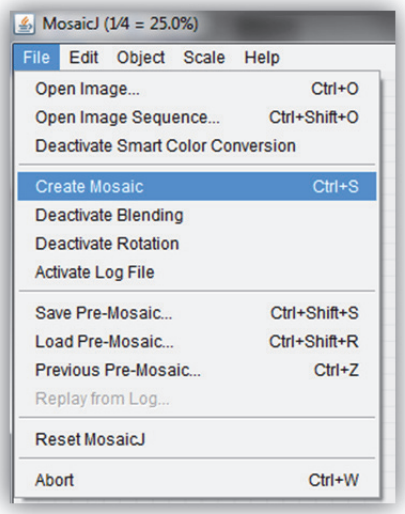
5. Click on each image to get it to move to the main part of the window.



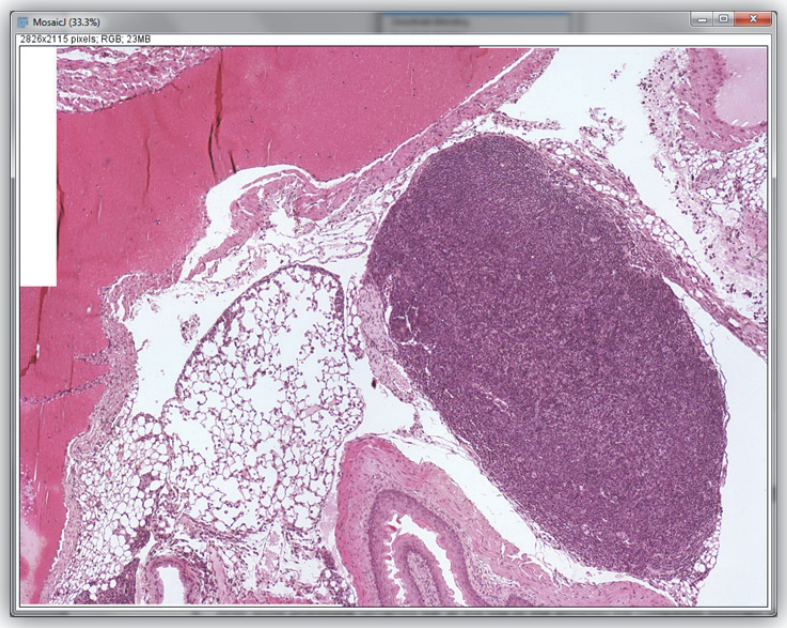
6. Now drag the pieces around and assemble them like a jig saw. When a piece is selected it has a blue outline around it. To nudge a piece hold down the **Ctrl** key and use the **Arrow** keys.



7. When you are happy with the montage go to **File → Create Mosaic**

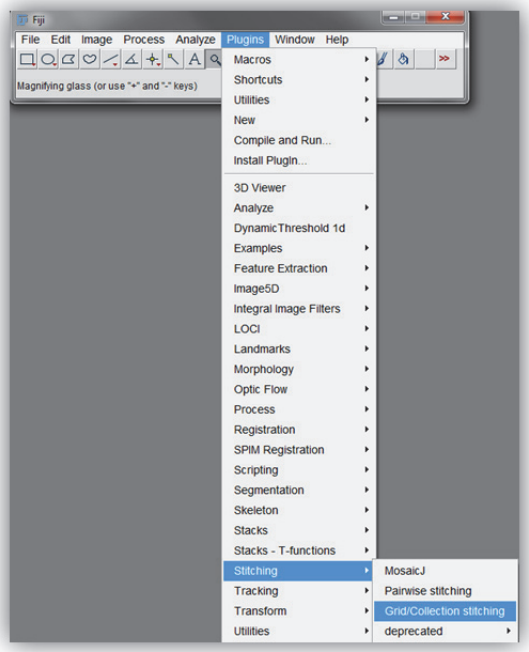


8. After some processing (progress bar at the top of the window) the completed montage will open in Fiji and MosaicJ will close.



Automatically Creating a Montage – Brightfield

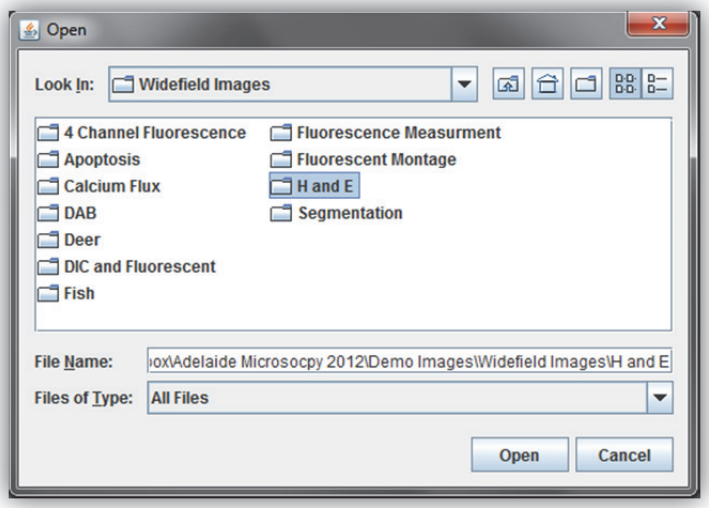
1. Go to **Plugins → Stitching → Grid/Collection Stitching**



2. In the next window set **Type** to **Unknown Position** and press **OK.**

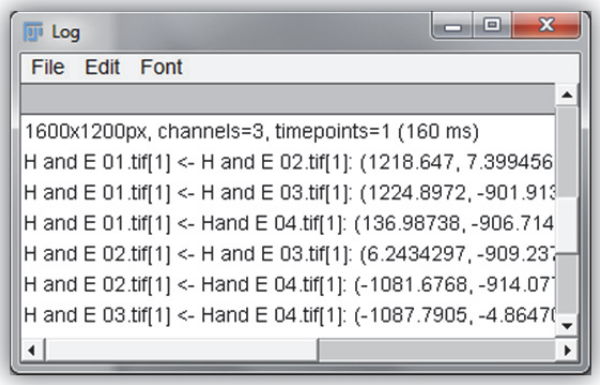
NOTE: If you know the order (number of rows, columns etc.) the images were captured in you can choose the appropriate **Type** from the list to cut down on processing time.

3. Set the **Directory** to the **Demo Images\Widefield\H and E** folder

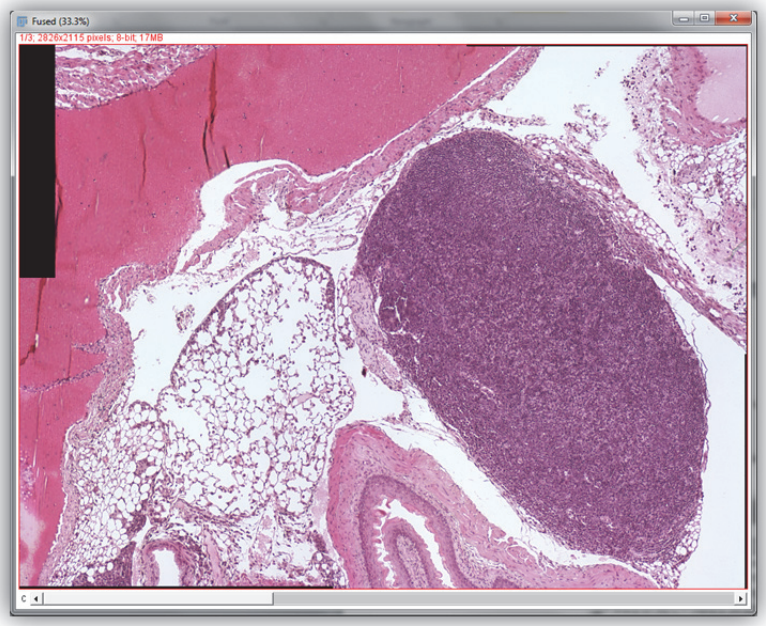


4. Leave all the other settings at their default values and press **OK.** Press **OK** the window that opens up next.

   You can watch the progress of the stitching in the **Log** window.



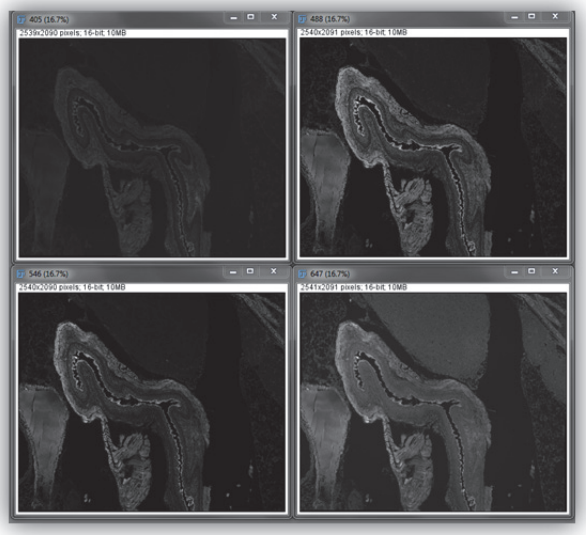
5. Once finished the completed montage will be displayed.



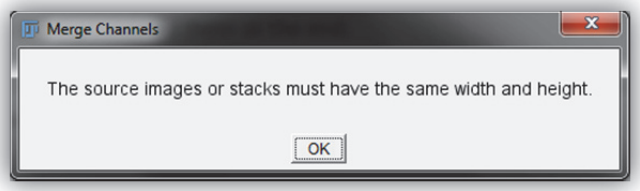
Automatically Creating a Montage – Fluorescence

Creating merges of fluorescent images works the same way but you will need to merge each of the channels separately and then overlay them at the end.

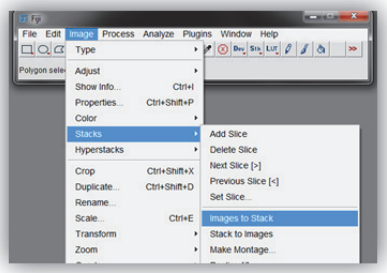
1. Use the automatic method above to merge the 4 channels of images in the **Demo Images\Widefield\Fluorescent Montage** folder



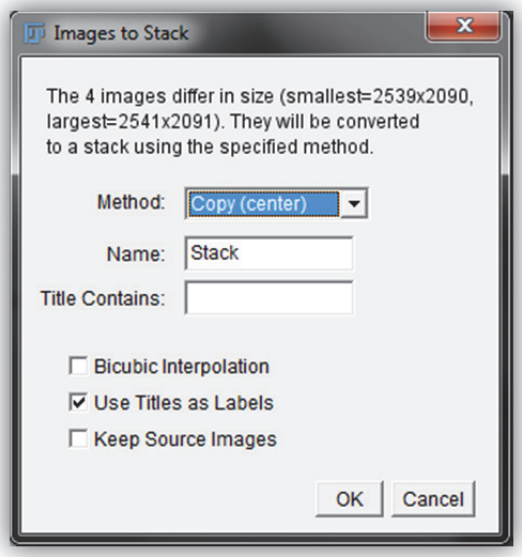
2. Try to use the **Merge Channels** command to create a composite image. You will receive the following error message.



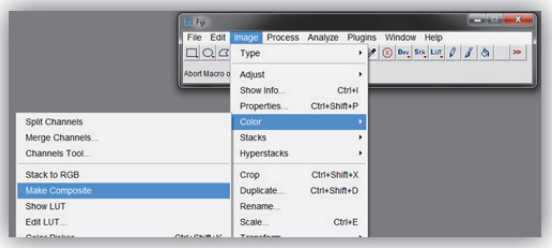
3. To get around this error all we need to do is put the images into a stack. Go to **Image → Stacks → Images to Stack…**

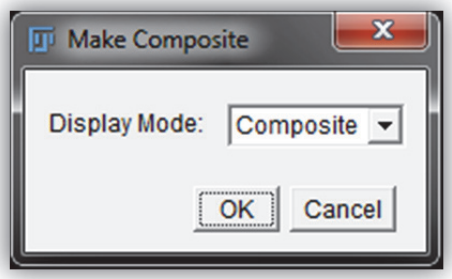


4. In the window that comes up make sure the **Method:** is set to **Copy (Centre)** and press **OK.**



5. Now that you have a stack it needs to be converted to a composite image so that colours can be easily assigned to each channel. Go to **Image → Colour → Make Composite.** Select **Composite** as the **Display Mode** and press **OK.**





6. You now have a composite stack that can be coloured and converted as needed.

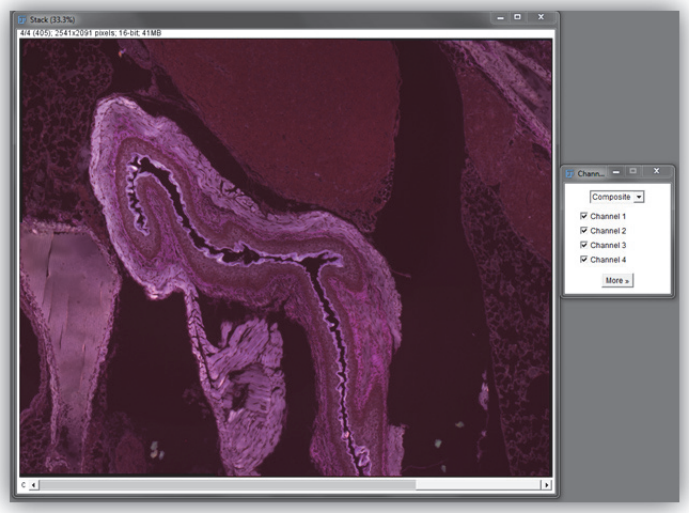


Image Stack Alignment

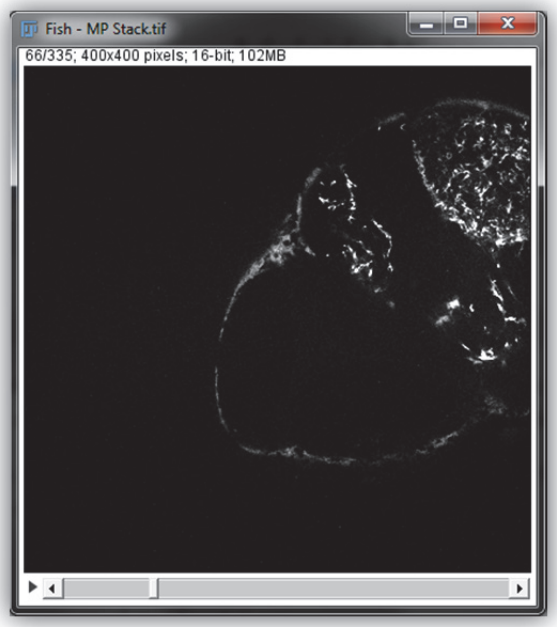
Aim

When taking large confocal stacks of unstable specimens it is possible for the sample to move and result in slices that do not align. This can be corrected with a plugin found in Fiji.

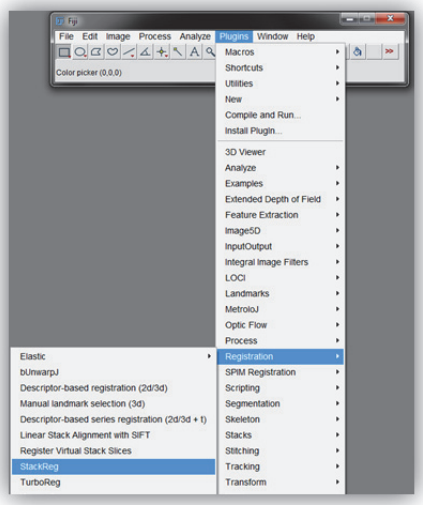
It is also possible to apply these methods to serial sections from either histology or transmission electron microscopy. Additionally is can be used to correct drift in a live imaging data set as well.

Auto Alignment of Confocal Data

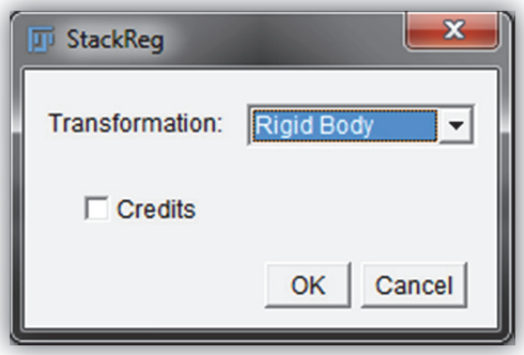
1. Open **Fish – MP Stack.tif** from the **Demo Images\Confocal\Fish** folder



2. If you play through the stack you will notice, especially between slices 70 and 100, that the images shift to the left of the frame. First duplicate the stack as the plugin will overwrite the original data.
   To correct the alignment go to **Plugins → Registration → StackReg**



3. In the next window set the **Transformation** type to **Rigid Body** and press **OK.**



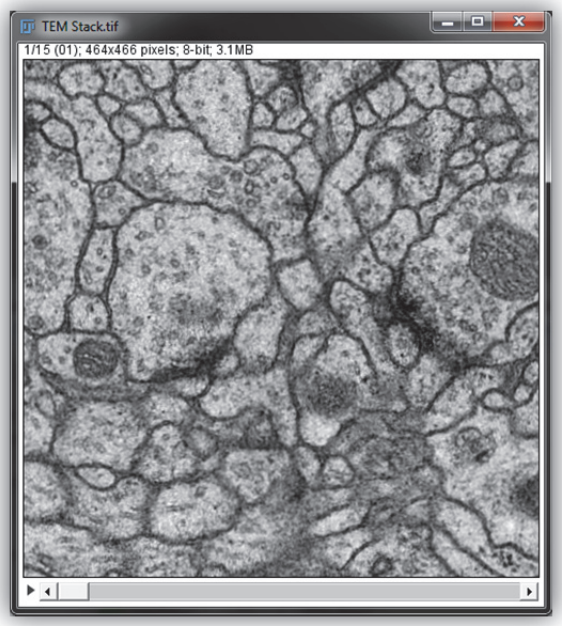
NOTE: There are four selections for the type of transformation.
Translation: Will move planes in X and Y
Rigid Body: Will move planes in X and Y as well as rotate
Scaled Rotation: Same as rigid body but will scale/zoom planes as well
Affine: Same as scaled rotation but can also deform images into trapezoid shapes

4. The stack will process to align. When it is finished play through the stack and notice that the drift that was present before is now gone.

Auto Alignment of Serial Sections

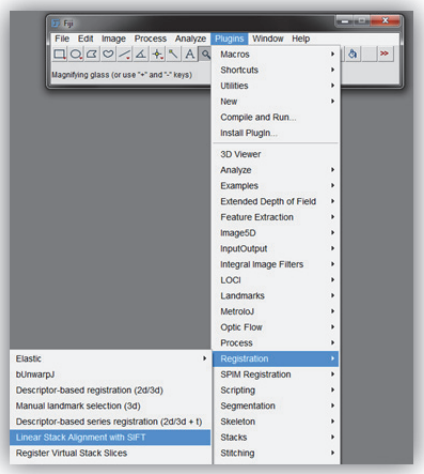
Sometimes it is necessary to align serial sections (either histology or TEM). In this example we will align a TEM stack but the same principals hold true for a histology stack.

1. Open **TEM Stack.tif** from the **Demo image\TEM** folder

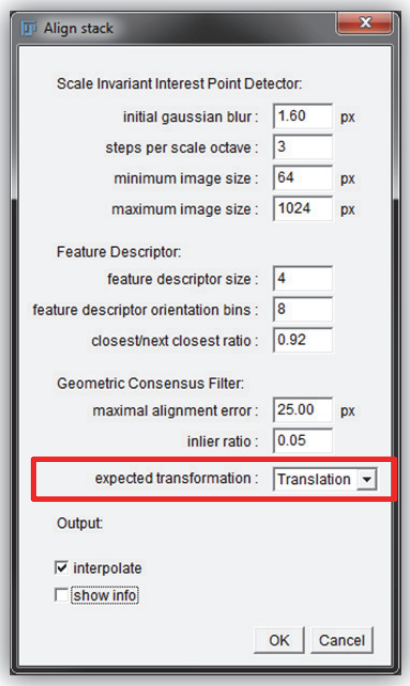


2. If you play through the stack you will notice that the slices do not align, but that there are common features between them that could be aligned. First do the same as above, duplicate the stack and use **StackReg** with **Rigid Body** transformation.
3. When you play through the stack you will notice that towards the end it wanders off a bit. Try again but this time with the **Scaled Rotation** transformation selected.
4. The result will be a little better but the data will be a bit distorted as the scaled rotation transformation will change the zoom level of some slices.
5. Try again with the **Affine** transformation selected. This probably gives the best visual alignment but notice that the last few slices have been heavily warped by the algorithm. So while looking quite good, this data has been totally destroyed.

6. To achieve a better alignment without introducing artefacts it may be necessary to use a different plugin. Duplicate the stack again and go to **Plugins → Registration → Linear Stack Alignment with SIFT.** This plugin is more computer intensive and will take longer to run but can give much more accurate alignment of a stack.



7. Leave the settings as default in the window that opens up but change the **Expected Transformation** method to **Translation**. Press **OK.**



8. The result should be a fairly well aligned stack without any introduced artefacts.

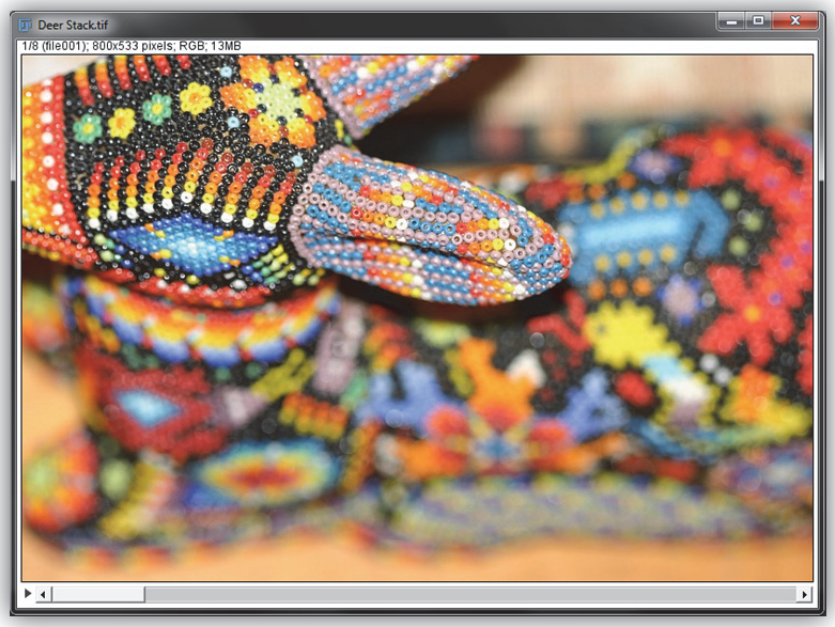
Extended Depth of Focus

Aim

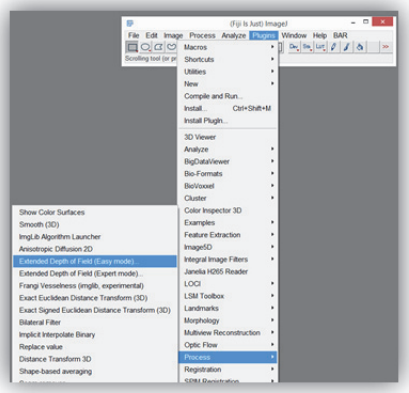
In some situations it may not be possible to get the whole sample in focus. This could be a whole mouse or a sample taken on a microscope. For these types of samples different focal points may need to be collected to get all points of interest in focus. To be able to present these images it is best to project them together as a stack and keep only the in focus parts. This can be achieved using the extended depth of focus plugin.

Extended Depth of Focus – Macro Images

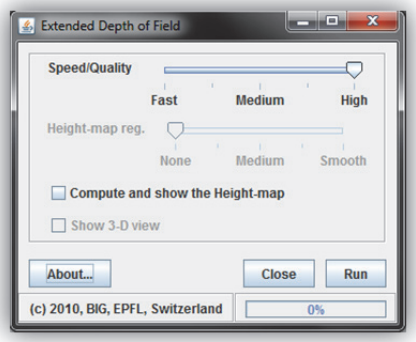
1. Open **Deer Stack.tif** from the **Demo Images\Widefield\Deer** folder. If you play through the stack you will notice that each plane has a different part of the deer in focus.



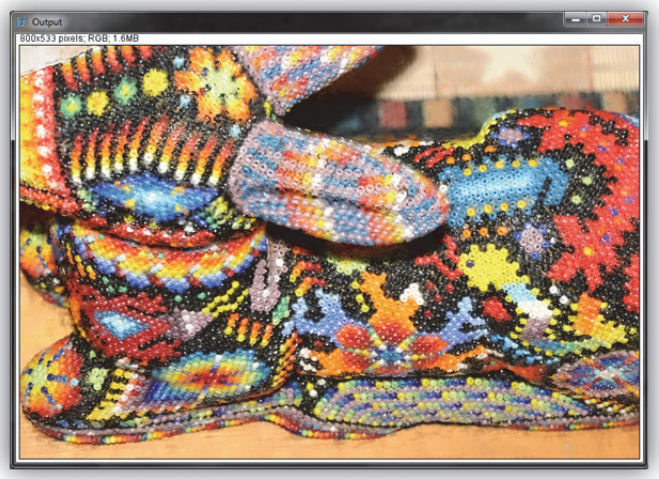
2. Go to **Plugins → Process → Extended Depth of Field → Easy Mode**



3. In the window that opens move the **Speed/Quality** slider all the way to the right. Untick the boxes and press **OK.**



4. When the processing is finished an in focus image of the deer will be displayed.

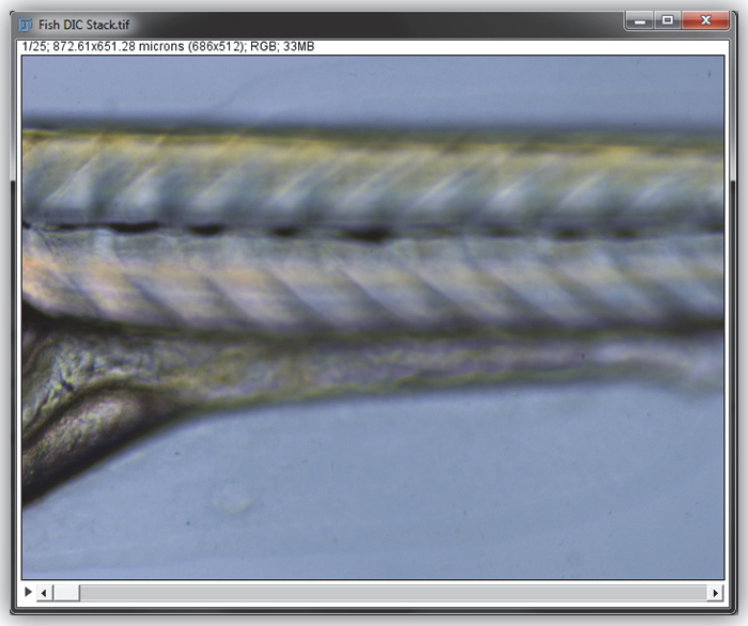


5. If you look at the final image it is a bit fuzzy in places. This is due to the images in each plane not being aligned properly; you can see this when playing through the stack. Run the **StackReg** plugin with the **Transformation** set to **Translation.**
6. The result will be a much more in focus image.

Extended Depth of Focus – Microscope Images

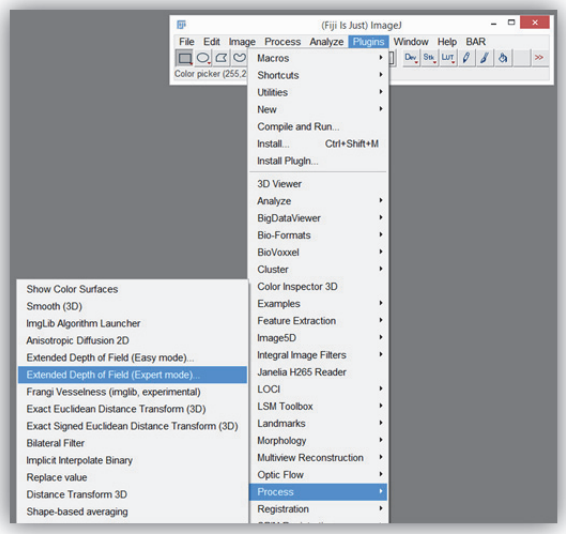
In the following example the image stack was taken on a widefield microscope with DIC optics. With DIC contrast only parts of a thick specimen are in focus on a given plane. The result of this is that when an image stack is captured it is amenable to extended depth of field projections.

1. Open **Fish DIC Stack.tif** from the **Demo Images\Widefield Images\Fish** folder

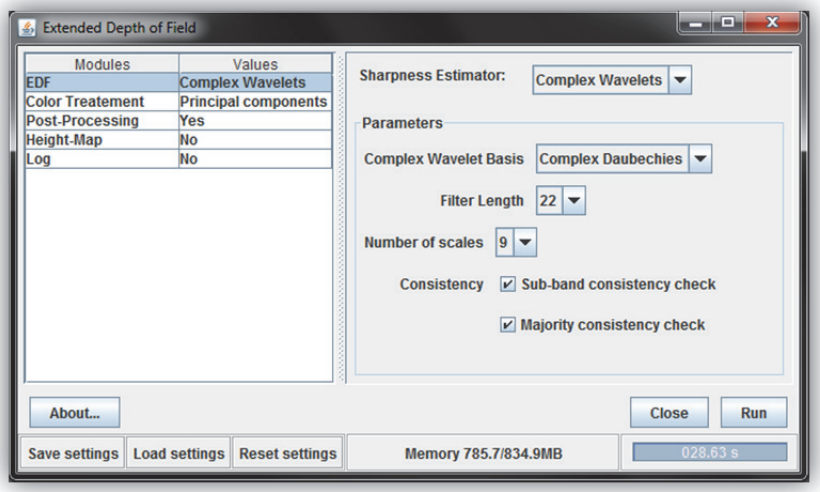


2. If you play through the stack you will notice that there are different parts of the image in focus on different planes just like the deer sample before. Run the Easy EDF on the stack as before.

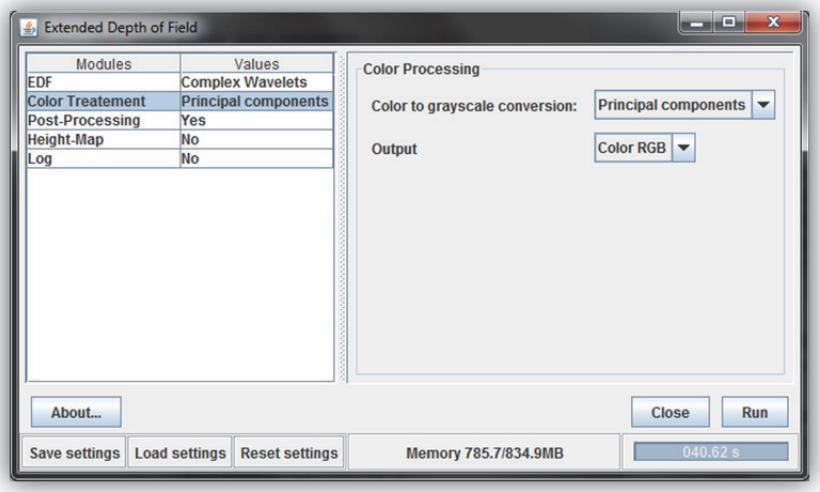
3. To get the best possible quality projection you will need to use the expert mode of the plugin. Go to **Plugins → Process → Extended Depth of Field → Expert Mode…**



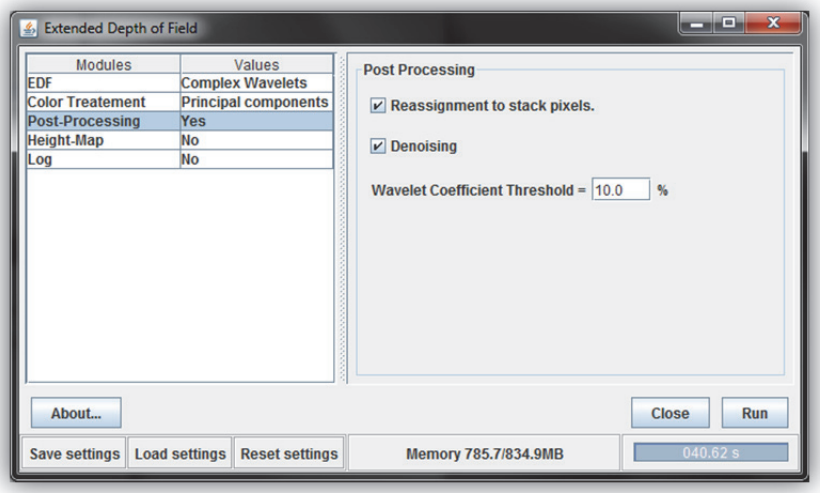
4. Select **EDF** on the left and configure the window as below



5. Select **Colour Treatment** and configure it as below.



6. Select **Post-Processing** and configure it as below.



7. Press **Run** to get the final EDF image. The differences are subtle but there will be less noise in the final image.

3D Visualisation and Measurement

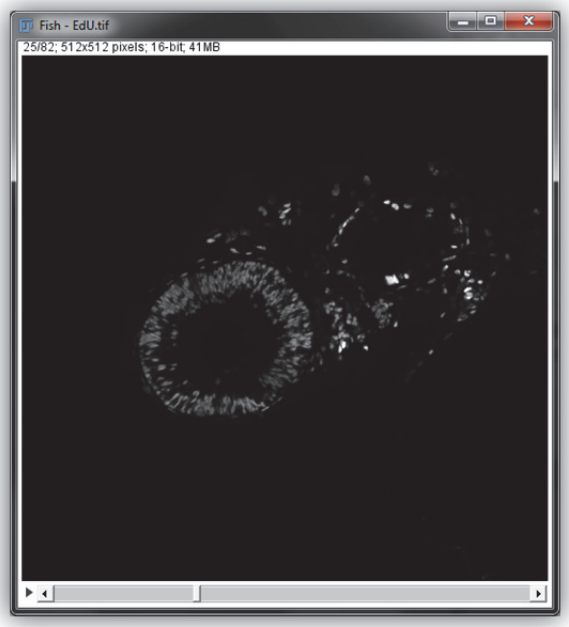
Aim

Fiji is capable of performing standard 3D visualisation and some rudimentary 3D measurements. For advanced 3D visualisation, measurement and segmentation there are really no free packagage available that are easy to use. Commercial packages such as Imaris, Volocity and Amira are more user friendly to beginners.

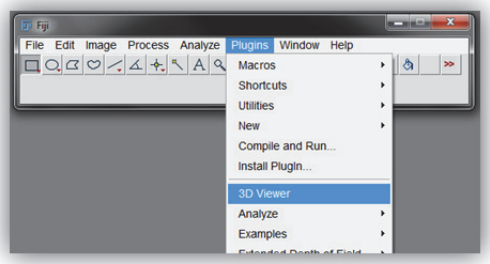
3D Visualisation

Visualisation

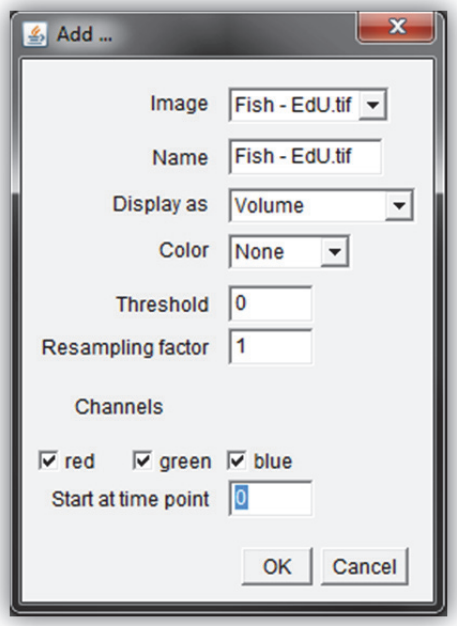
1. Open **Fish –EdU.tif** from the **Demo Images\Confocal\Fish** folder



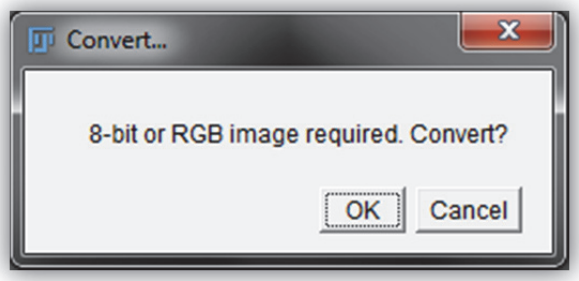
2. To view the data in 3D go to **Plugins → 3D Viewer**



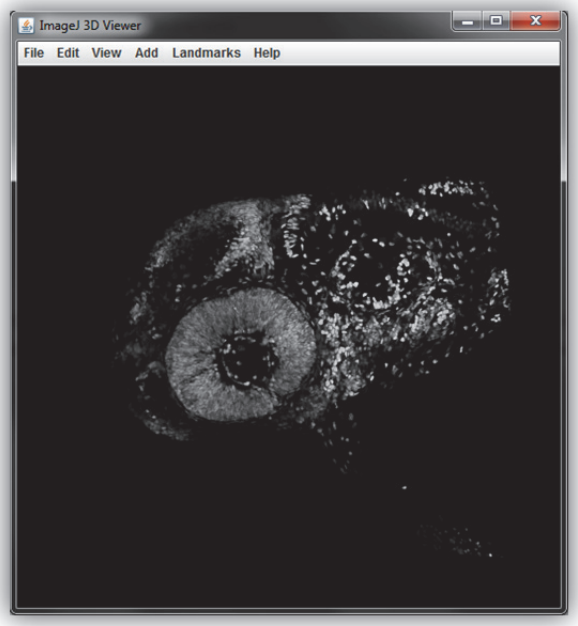
3. In the window that opens leave all the settings at default (like below) but first try changing the **Resampling Factor** to **1.** Setting the resampling factor to 1 makes a 3D model at the full resolution of the original data set. This may be too much for your computer system to handle, so test it fist.



4. When you press **OK** you will receive the following message, this is because the 3D viewer module is not compatible with 16 bit images. Just press **OK** to continue.



5. The **ImageJ 3D Viewer** will eventually show you data in 3D.

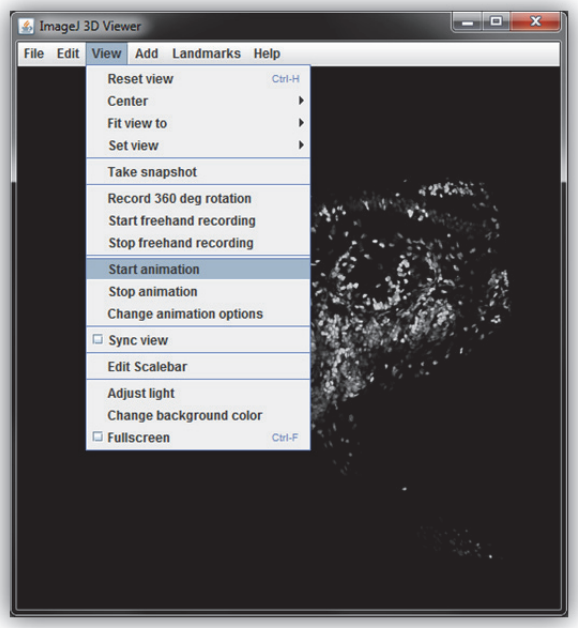


6. To navigate through the model use the following controls

   NOTE: If the performance is very slow (stuttering, jumping etc.) close the 3D viewer and reopen the data set with the **Resampling Factor** set to 2.

Zoom In and Out	Mouse Wheel or Page Up and Down Keys
Rotate Image	Click left button and hold or arrow keys

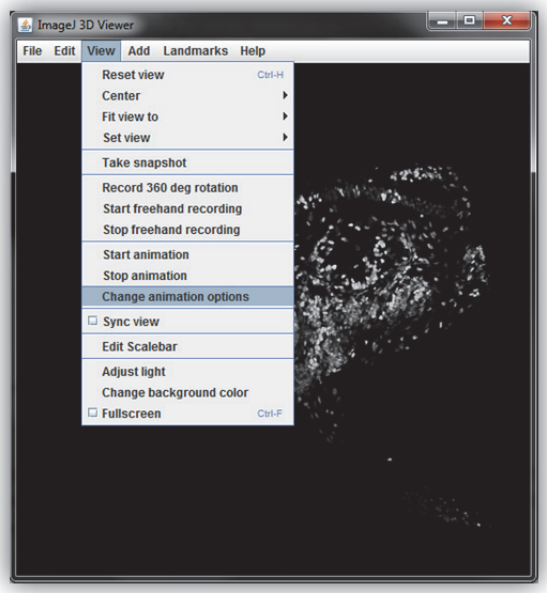
7. To animate the data set go to **View → Start Animation** (to stop it go to **View → Stop Animation**



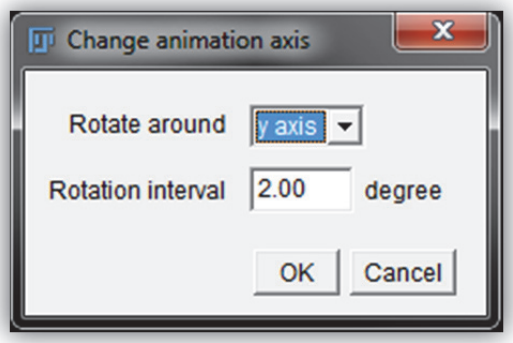
Making Movies

Predefined

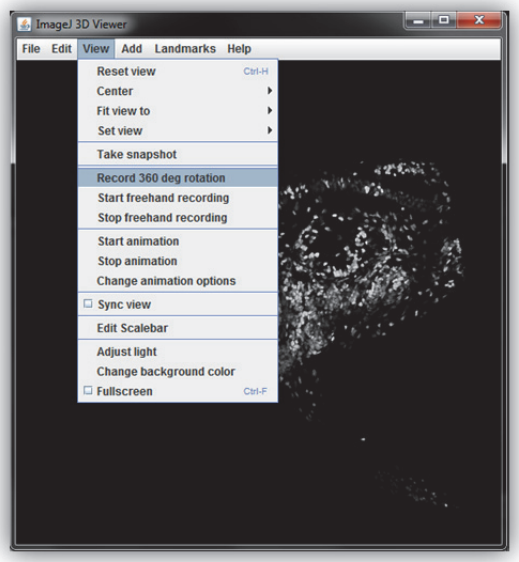
1. Firstly we need to configure the how we want the rotation to be performed. Go to **View → Change Animation Options**



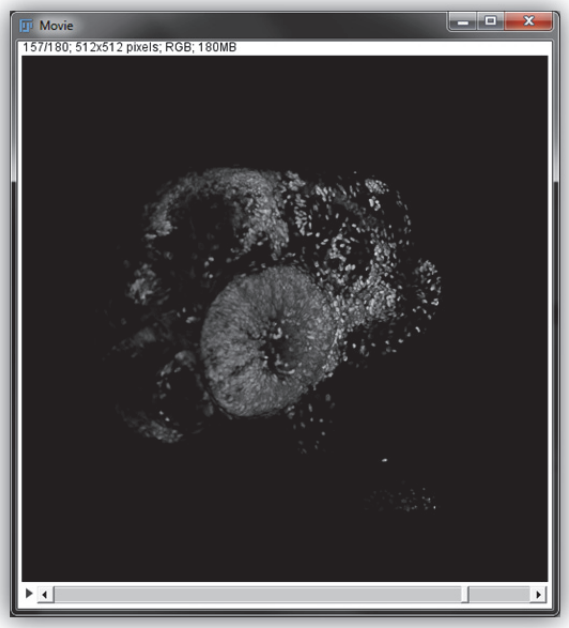
2. Choose which axis you want to rotate the image on. This is the axis plane of the window not the axis of the data set. You can also set the **Rotation Interval** the lower the number the smoother the rotation will be, but the larger the resulting file will be. When you are happy with the settings press **OK.**



3. To create the rotation go to **View → Record 360 Deg Rotation**



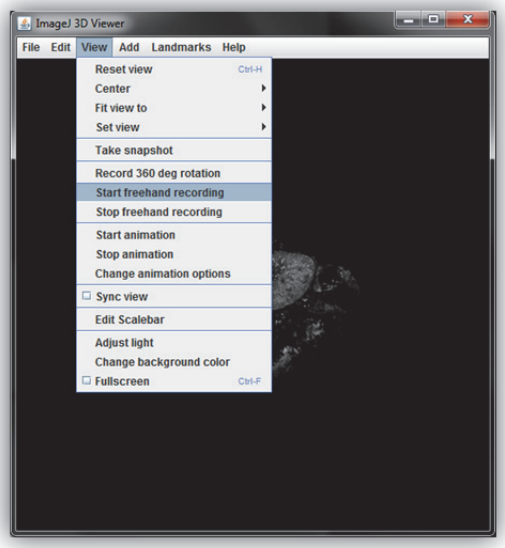
4. The result will be an image stack the you can play through and save for later or convert to an AVI as shown earlier



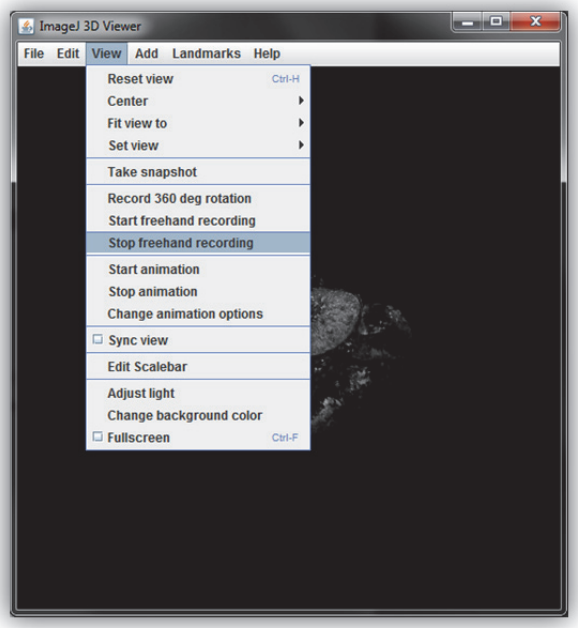
Manually Set

Sometimes a standard single axis rotation is not enough and a more complex rotation or "fly through" needs to be created.

1. Go to **View → Start Freehand Recording**

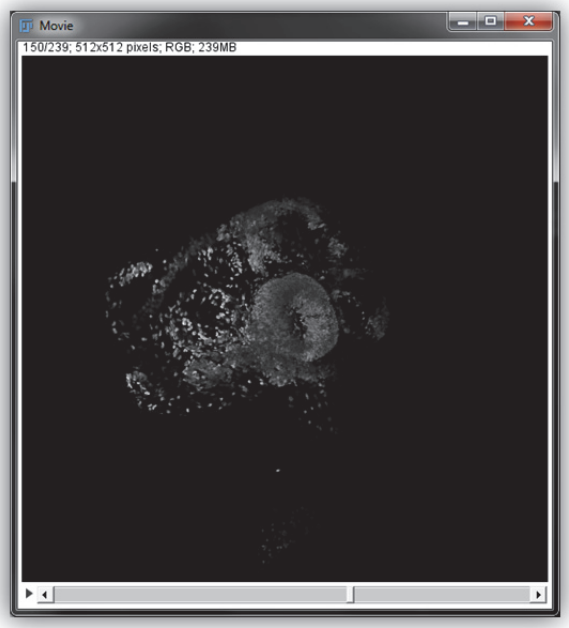


2. Now everything you do in the **ImageJ 3D Viewer.** Move the data set around, zoom in and out etc. When you have finished go to **View → Stop Freehand Recording**



NOTE: The results will vary depending on how well your computer performs. If the animation is stuttering try re opening the model with a higher **Resampling Factor** set.

3. You now have an image stack that can be played and saved as before.



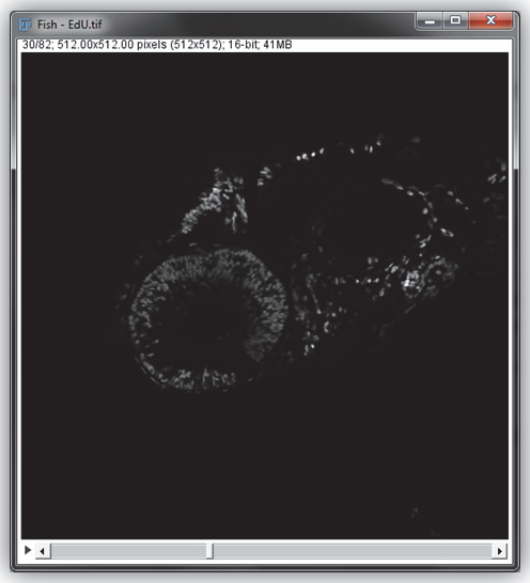
3D Measurement

The 3D measurement functions in Fiji are rather rudimentary. There are some very powerful plugins such as Neurite Tracer or TrakEM2 that can generate very detailed 3D models and measurements but does rely on the user manually outlining regions of interest.

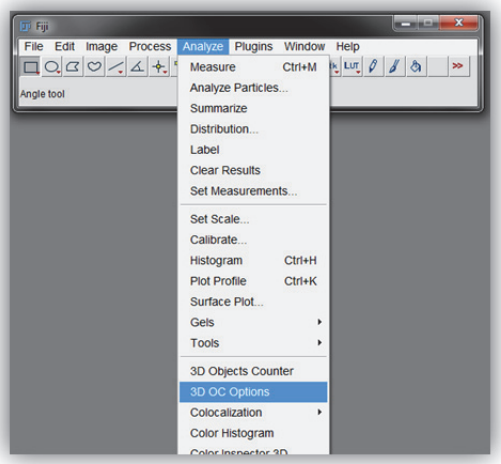
For this demonstration we will only look at a simple, automatic way of measuring objects/volumes in 3D.

Measuring Objects in 3D

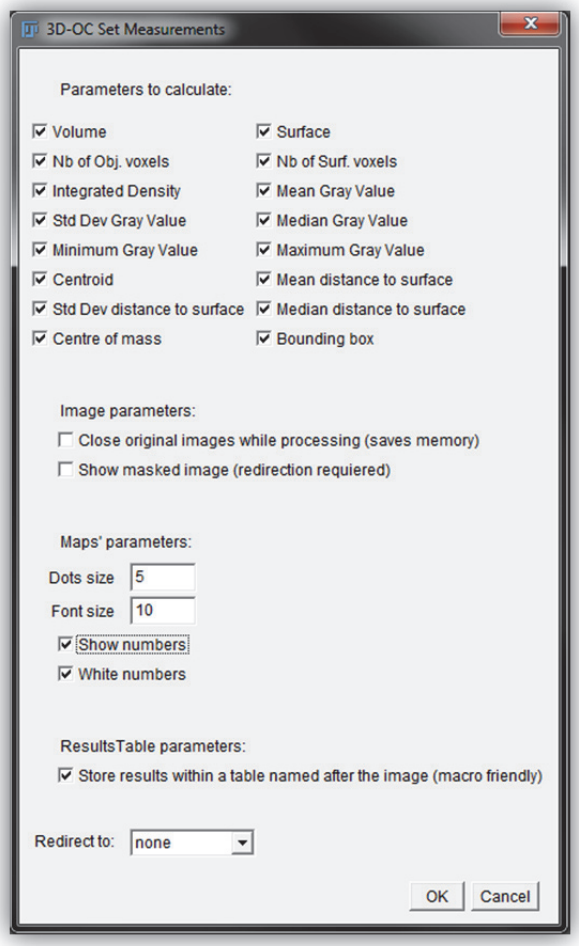
1. Open **Fish – EdU.tif** from the **Demo Images\Confocal\Fish** folder



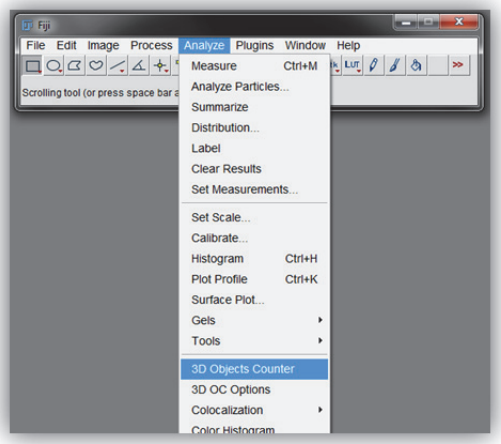
2. Go to **Analyse → 3D OC Options…**



3. In the **3D-OC Set Measurements** window you can set what you want to have measured and the outputs that will be generated. Leave the settings at default and press **OK.**



4. Go to **Analyse → 3D Objects Counter**

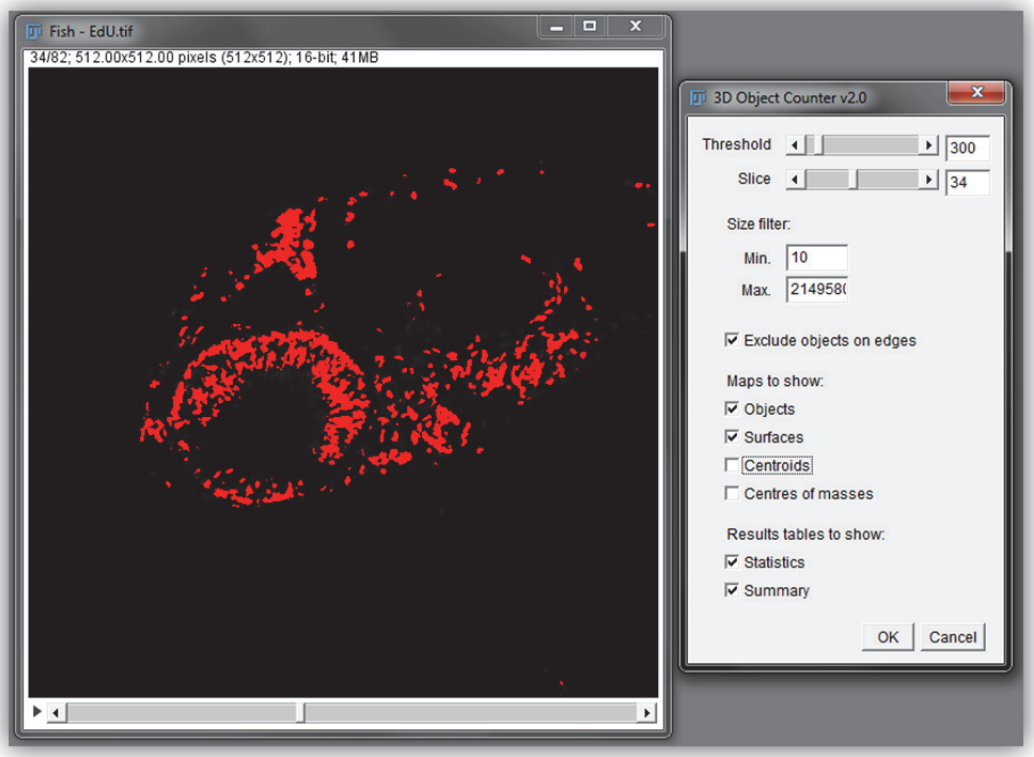


5. In the **3D Object Counter v2.0** window set the threshold to select the stain you want to measure. You can move the **Slice** slider to select a different slice to see how the threshold is working.

   You can set the **Min** and **Max** values in the **Size Filter:** to remove objects from your final analysis. The size values are in cubic measurements.

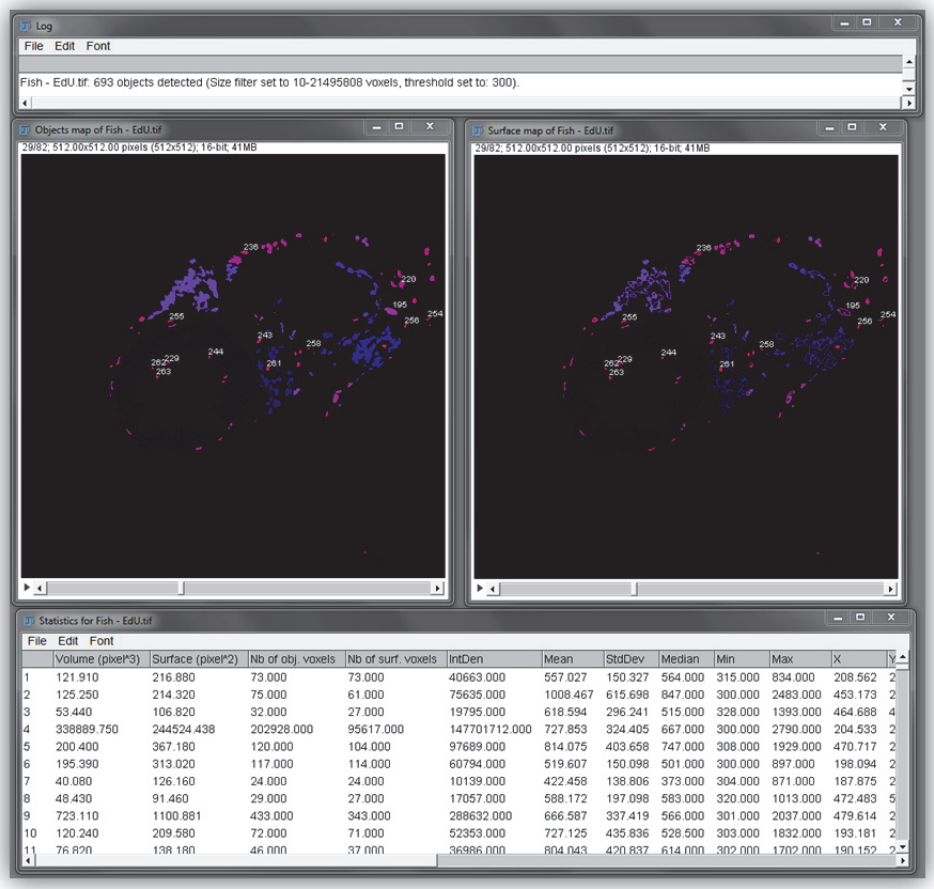
   Configure the **Maps to Show** as shown below.

   Press **OK** and wait for the processing to complete.



6. After a decent amount of processing time the results will be shown as below.

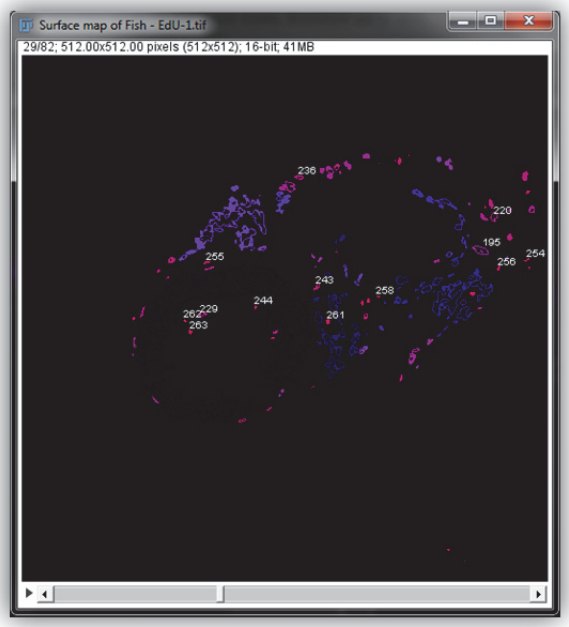
Log – shows the number of objects counted and the threshold used to generate them
Objects Map – A labelled coloured binary map of each of the objects
Surface Map – A labelled coloured binary of the outlines of each of the objects
Statistics – Detailed information about each of the objects counted in the image.



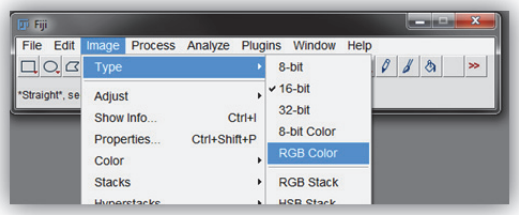
	Volume (pixel^3)	Surface (pixel^2)	Nb of obj. voxels	Nb of surf. voxels	IntDen	Mean	StdDev	Median	Min	Max	X	Y
1	121.910	216.880	73.000	73.000	40663.000	557.027	150.327	564.000	315.000	834.000	208.562	2
2	125.250	214.320	75.000	61.000	75635.000	1008.467	615.698	847.000	300.000	2483.000	453.173	2
3	53.440	106.820	32.000	27.000	19795.000	618.594	296.241	515.000	328.000	1393.000	464.688	4
4	338889.750	244524.438	202928.000	95617.000	147701712.000	727.853	324.405	667.000	300.000	2790.000	204.533	2
5	200.400	367.180	120.000	104.000	97689.000	814.075	403.658	747.000	308.000	1929.000	470.717	2
6	195.390	313.020	117.000	114.000	60794.000	519.607	150.098	501.000	300.000	897.000	198.094	2
7	40.080	126.160	24.000	24.000	10139.000	422.458	138.806	373.000	304.000	871.000	187.875	2
8	48.430	91.460	29.000	27.000	17057.000	588.172	197.098	583.000	320.000	1013.000	472.483	5
9	723.110	1100.881	433.000	343.000	288632.000	666.587	337.419	566.000	301.000	2037.000	479.614	2
10	120.240	209.580	72.000	71.000	52353.000	727.125	435.836	528.500	303.000	1832.000	193.181	2
11	76.820	138.180	46.000	37.000	36986.000	804.043	420.837	614.000	302.000	1702.000	190.152	2

Generating 3D models of the Results

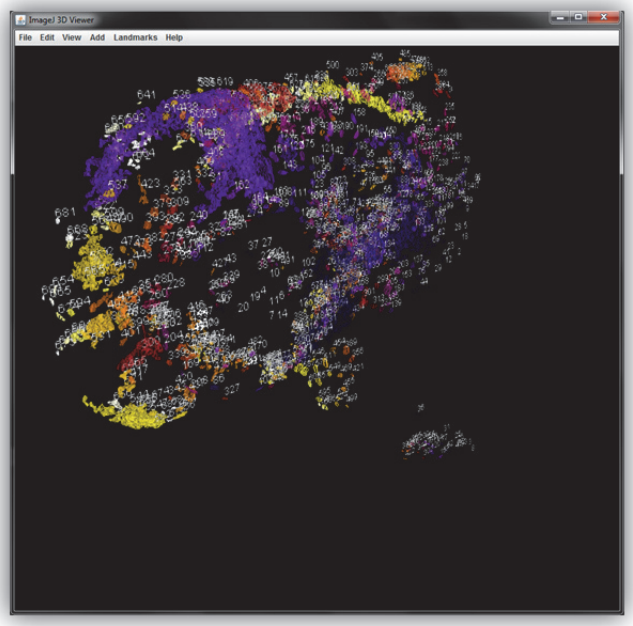
1. It is possible to generate a 3D model of the data for presentation and interpretation. Select one of the result stacks and duplicate it, the **Surface** stack is the better one for this.



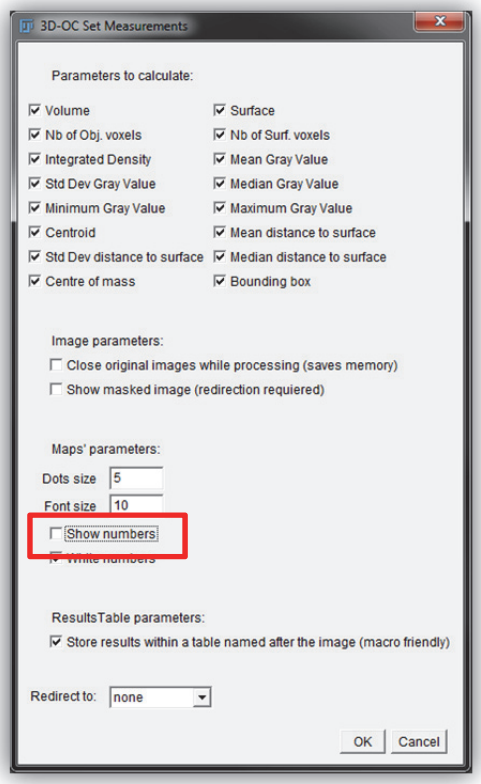
2. Convert the stack to RGB colour.



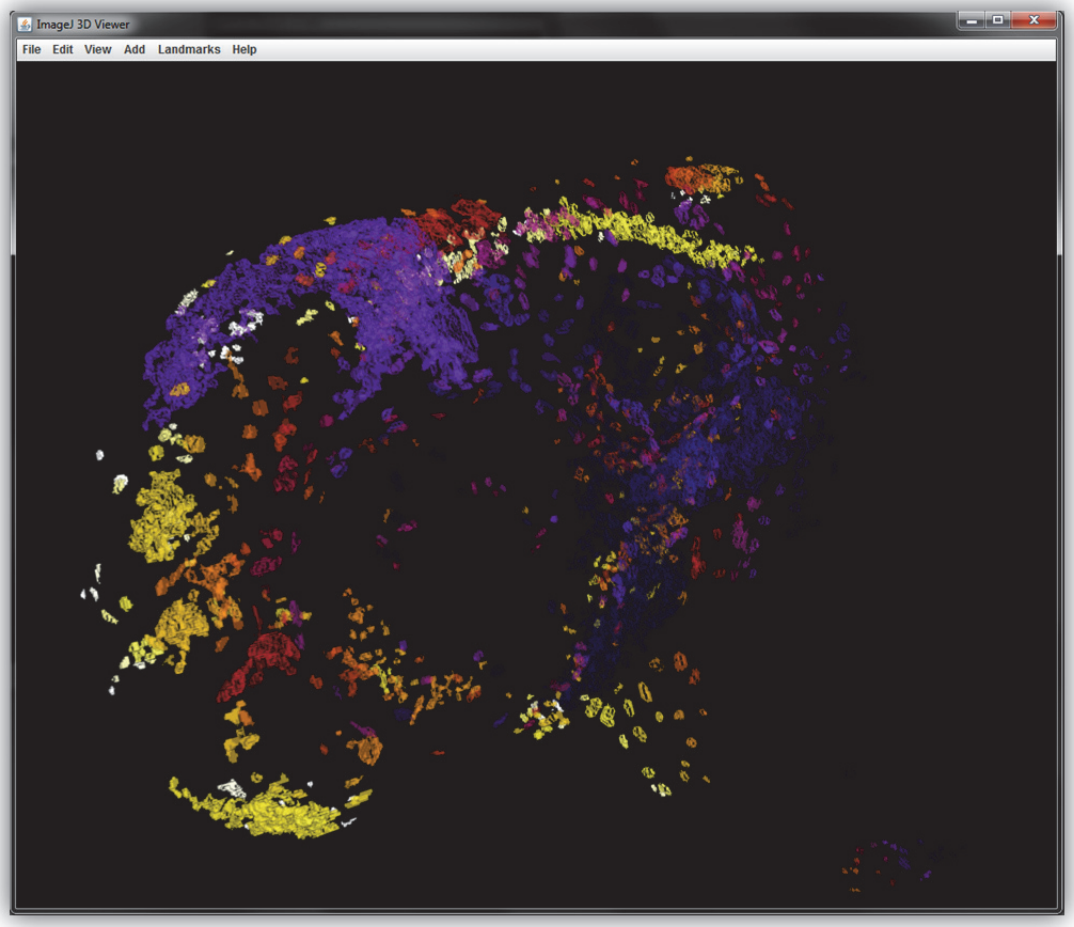
3. Open the converted stack with the **3D Viewer** as shown above.



4. If you don't want all the white numbers on the final image you can remove them by reconfiguring the **3D Objects Counter** to not show the numbers in the output images.



5. With the numbers turned off rerun the analysis and recreate the 3D model. The result is a much cleaner model.



Advanced Image Segmentation

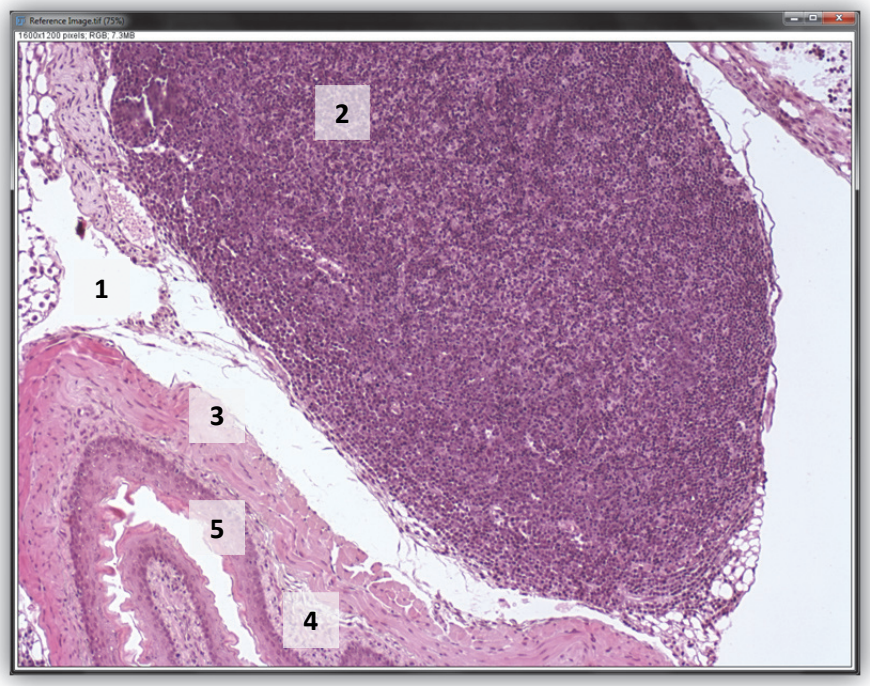
Aim

Fiji contains a very powerful trainable segmentation tool called **Trainable Weka Segmentation.** This tools works by training the module to recognise the difference in a given image by looking at a range of parameters (colour, texture, edges etc.)

This can be a very powerful tool in analysing images, such as H and E stains or TEM images, to detect differences in tissue/cell/organelle type distributions.

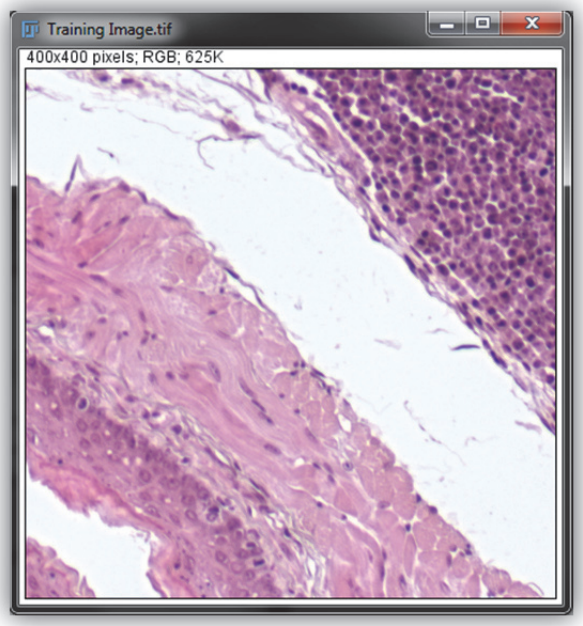
Training the Module

1. Firstly open up **Reference Image.tif** from the **Demo Images\Widefield Images\Advanced Segmentation** folder. If you look at the image we can identify several different tissue types.



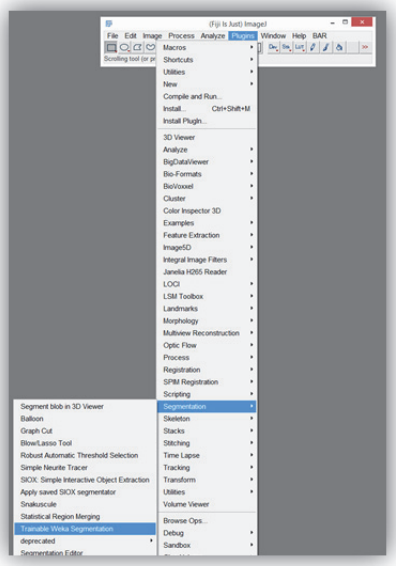
1. Background
2. Dense Tissue
3. Stromal Tissue
4. Dermal Tissue
5. Epidermal Tissue

2. Open **Training Image.tif** from the **Demo Images\Widefield Images\Advanced Segmentation**

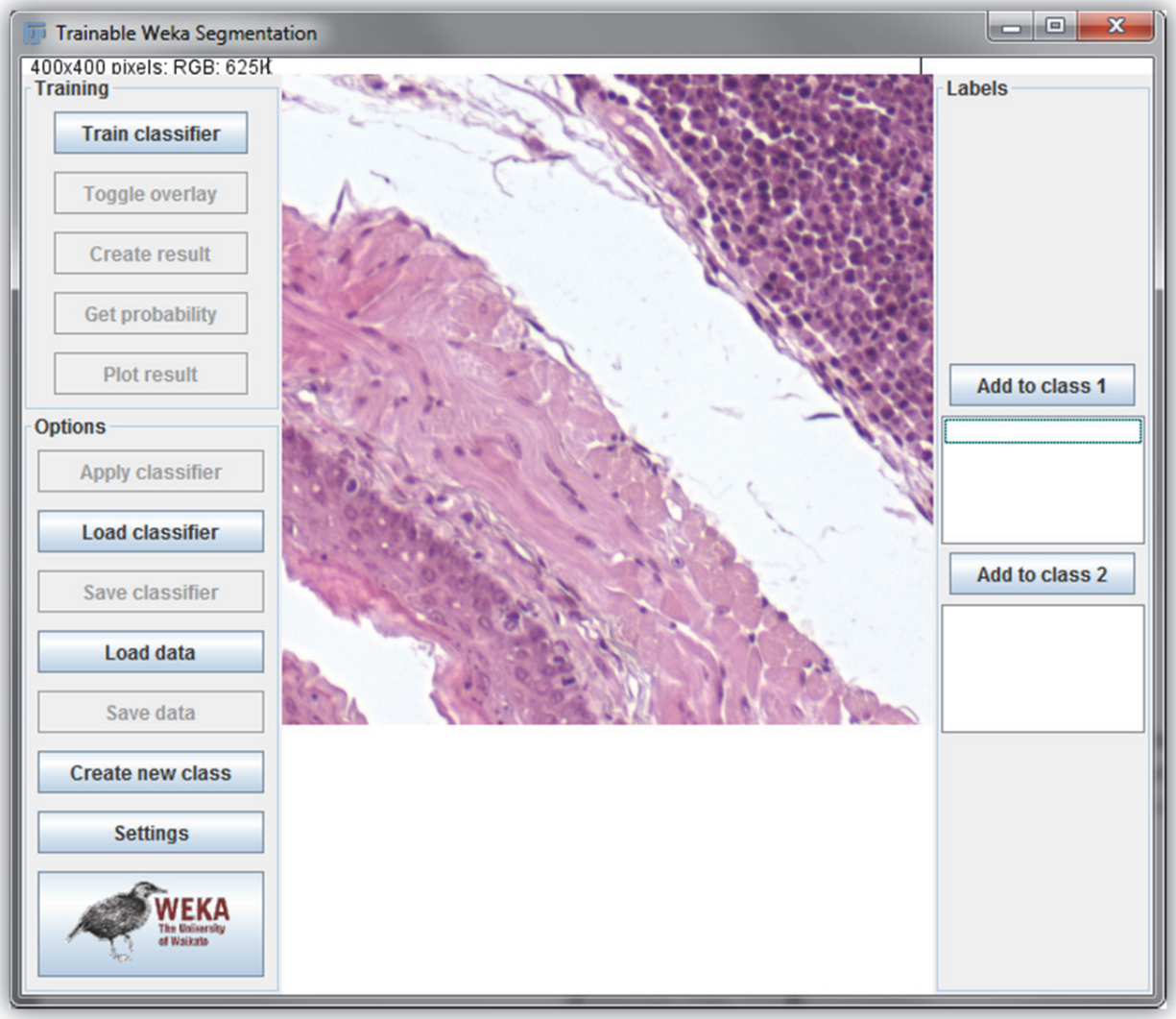


This image is a smaller section of the original image that contains representative bits of each of the tissue types. This image (and the subsequent ones we will use for analysis) are all rather small. This is because advanced segmentation requires large amounts of computer resources and so working on smaller images helps keep resource use down.

3. Go to **Plugins → Segmentation → Trainable Weka Segmentation**



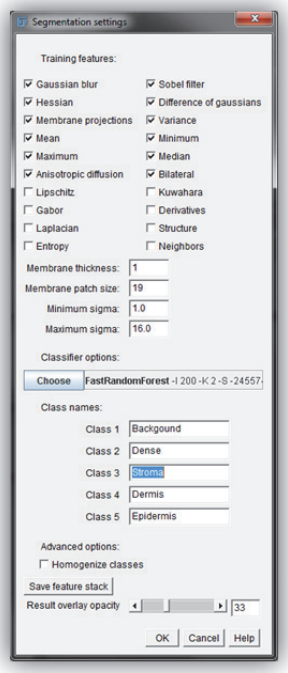
4. You can resize the window and zoom in on the image to make the subsequent steps easier.



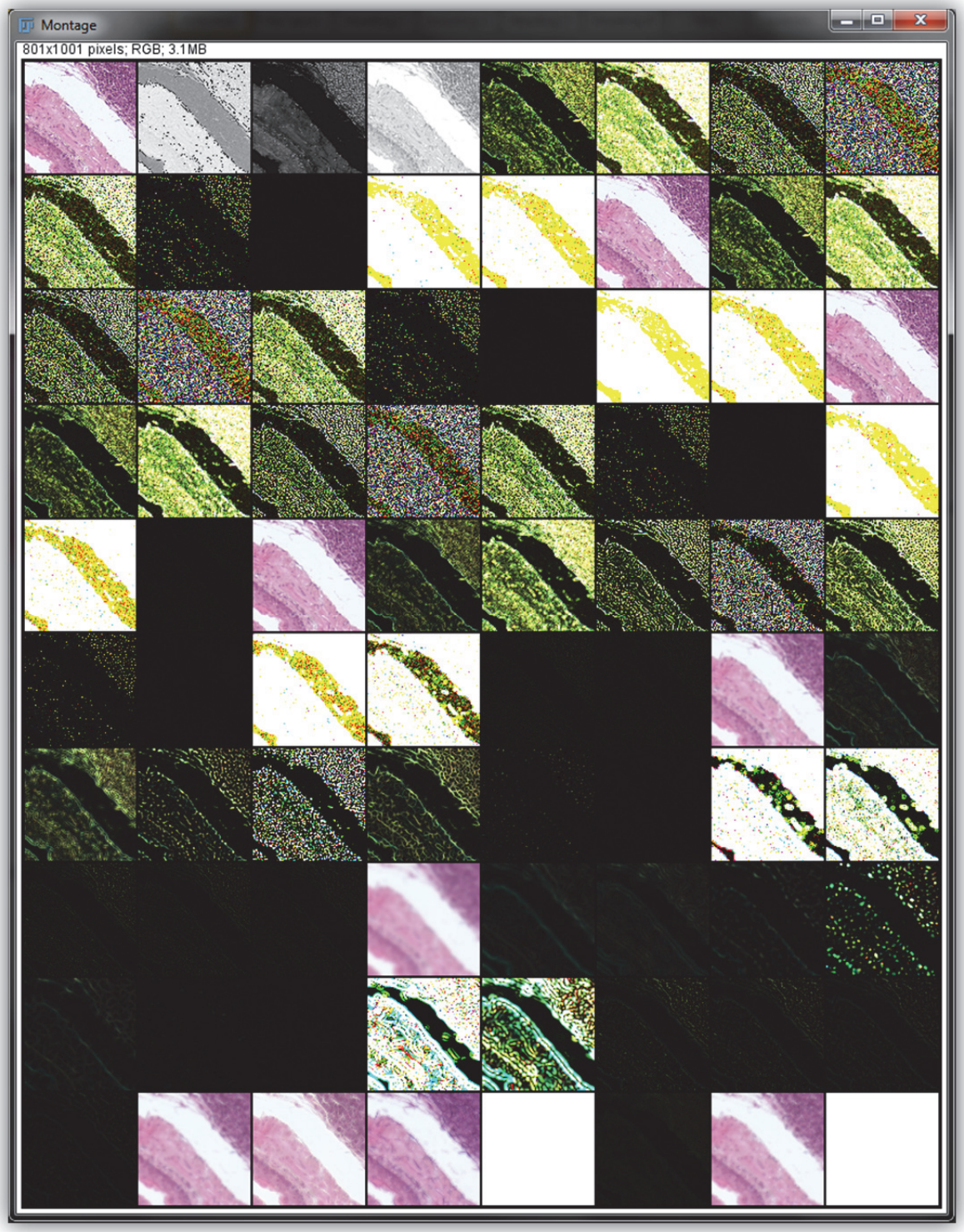
5. Firstly press the **Create New Class** and give the class a name (don't worry we will rename class 1 and 2 later).

6. Create 2 more classes to bring the total up to 5 classes.

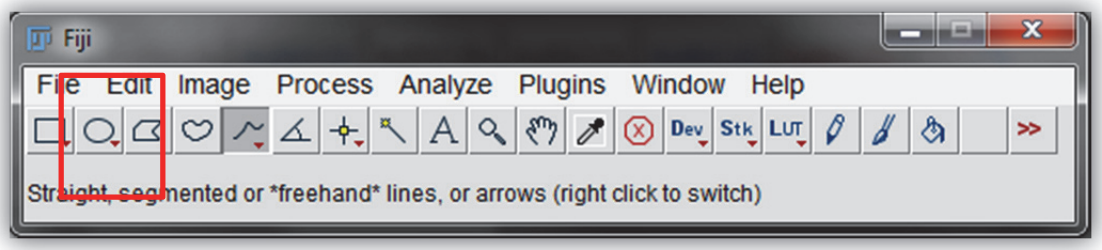
7. Press the **Settings** button and configure it as below. The more **Training Features** selected the more accurate the segmentation can be, but the more computer intensive the analysis will be (it could even become impossible to process on certain machines). The selection below is a good balance for an average computer.

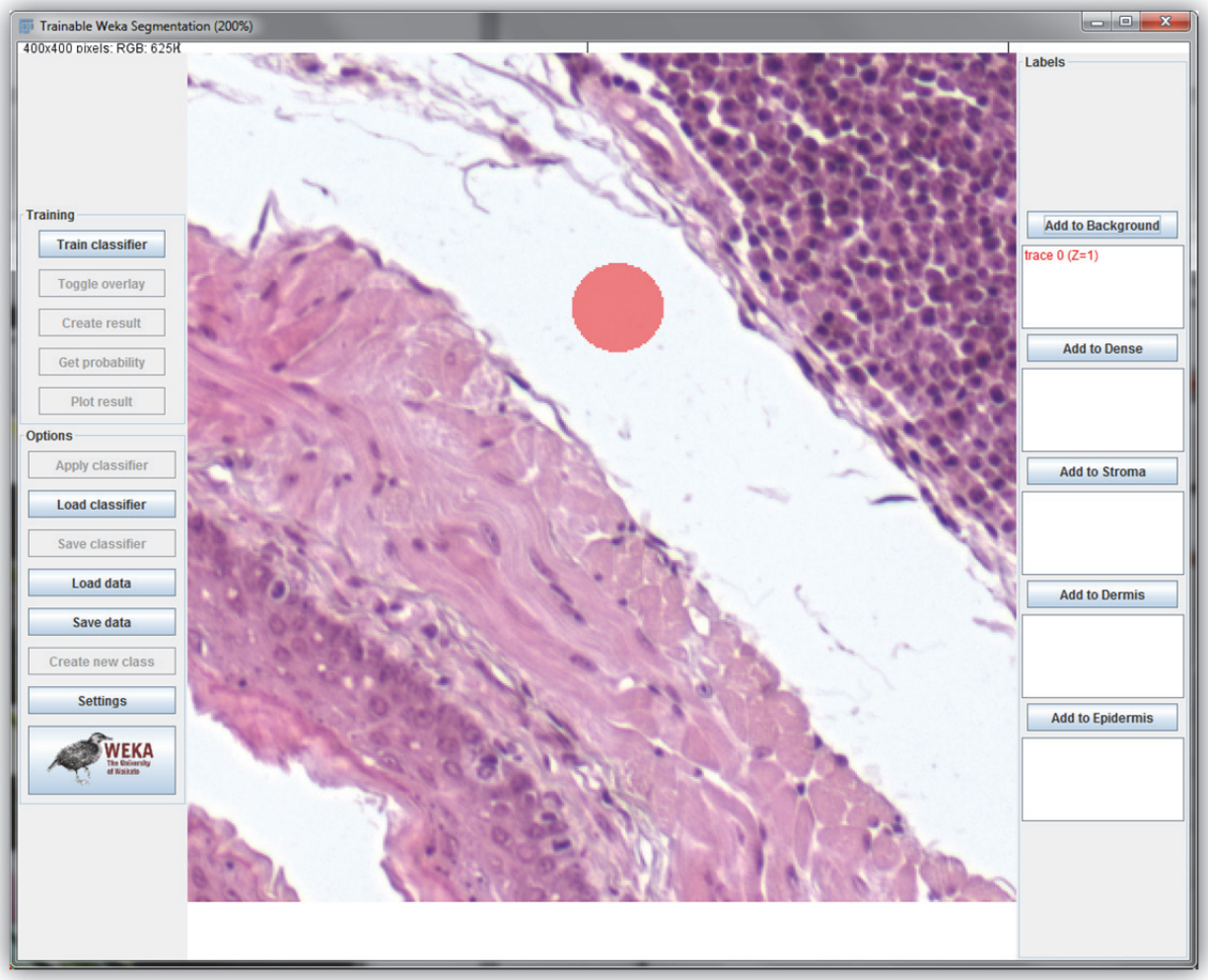


8. If you press the **Save Feature Stack** button near the bottom you can save a stack that will show you what will be used for the subsequent segmentation. The number and types of images will depend on which settings are ticked.

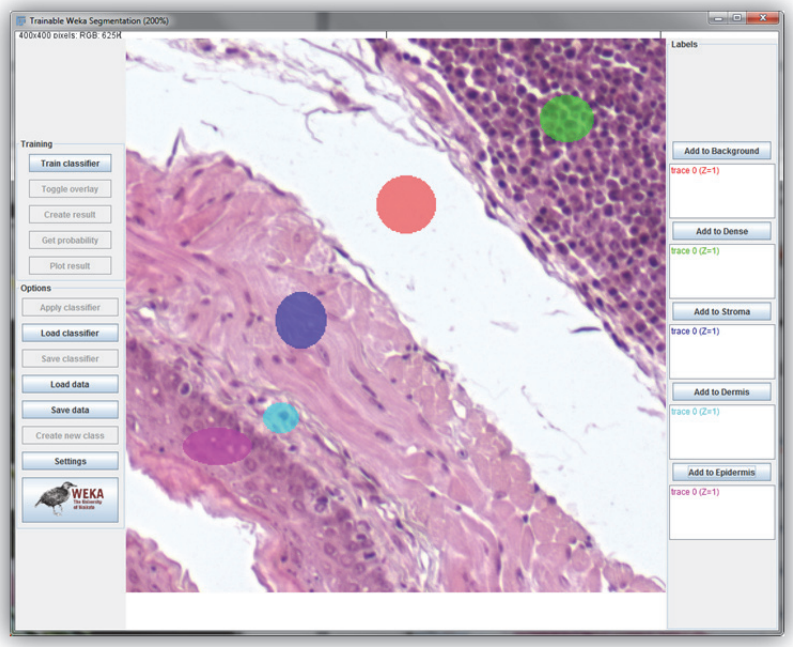


9. To train the module you need to assign the different parts of the image to the given classes. This is carried out with the ROI tools. For large homogenous areas you can use the circle, square, freehand or polygon tools. For smaller areas where more accuracy is required (and for later refinement) you should use the freehand line tool. Select the **circle** ROI tool and draw a circle ROI on the background of the image. When you are happy with what you have selected press the **Add to Background** button.



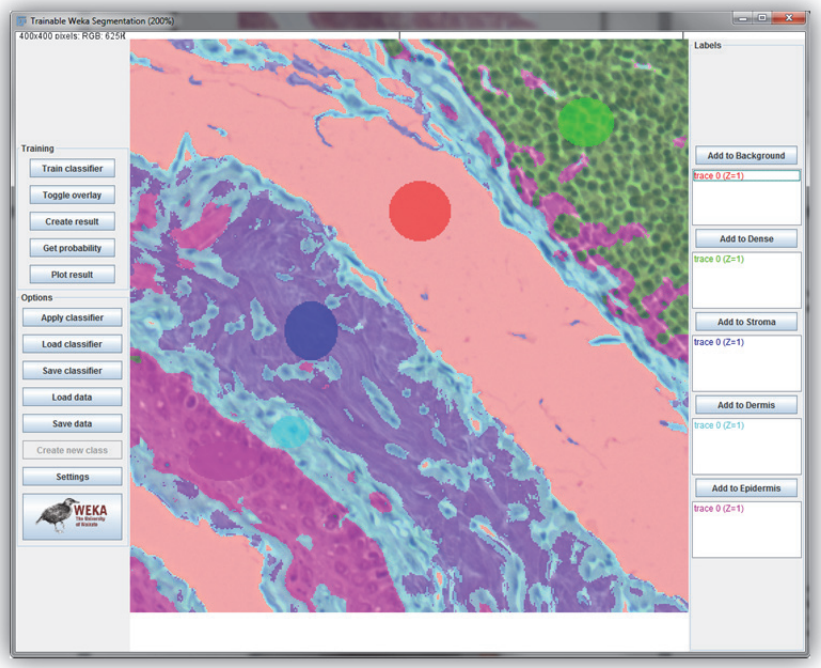


10. Repeat this for the other classes. You can add more than one example of each class if you want.

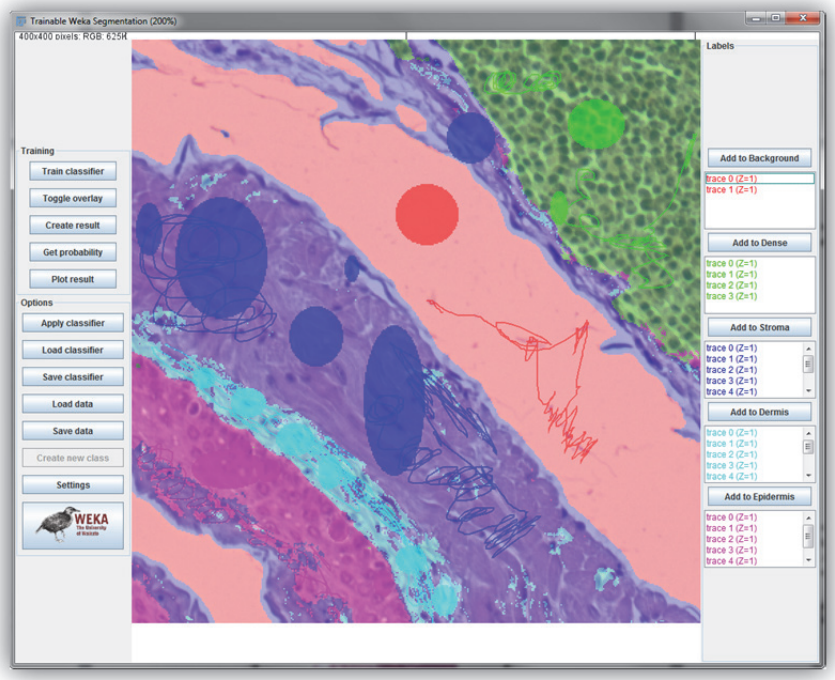


11. When done press the **Train Classifier** button. Be prepared to wait for a while.

    When it is complete you will see colour overlays put on the image representing the different classification. You can toggle the overlay on and off with the **Toggle Overlay** button.

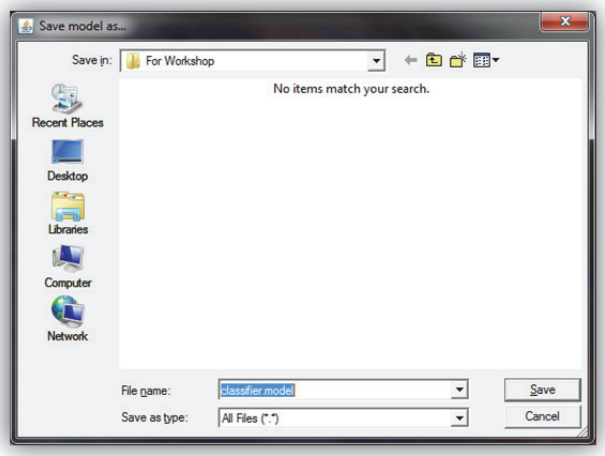


12. You will notice that the segmentation is not 100% accurate. Using the **freehand** ROI tool you can scribble over the bits that are wrong and retrain the classifier. Go through as many rounds of this as required to get a segmentation you are happy with. **NOTE:** it is very likely you will never get 100% perfect segmentation but remember that any errors will be uniform to all images and so will be "lost in the noise" of analysis.

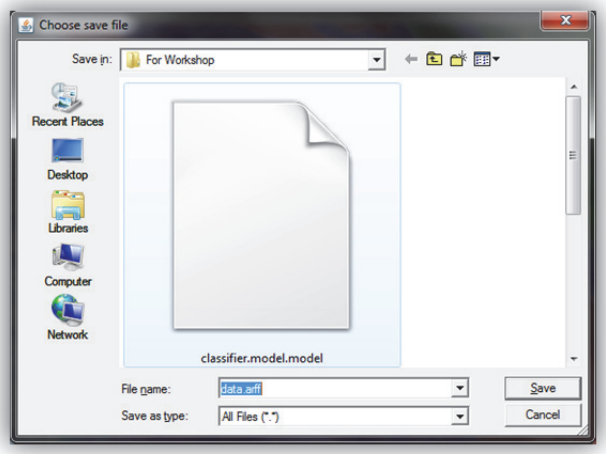


NOTE: If you add a ROI that you want to delete, just double click it in the class lists

13. Once you are happy with the segmentation we need to save the files that let the rules you created be applied to other images. Press the **Save Classifier** button and save the file somewhere you can find it again.

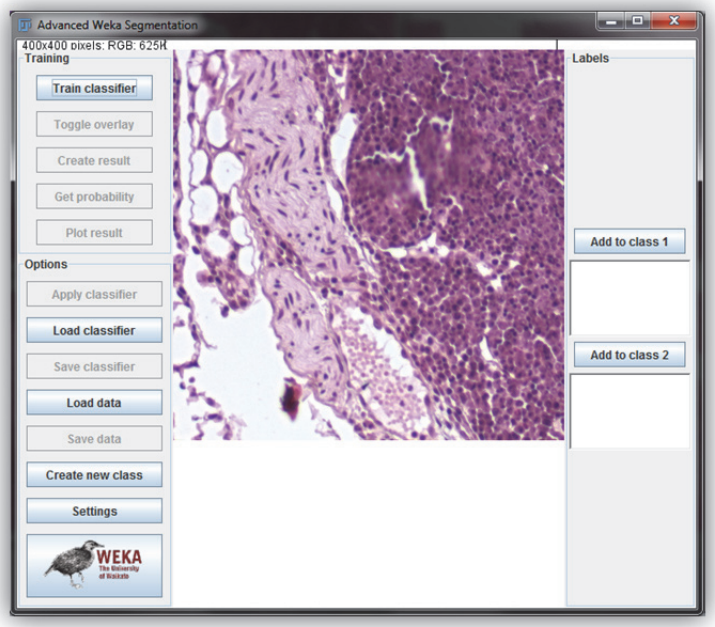


14. Do the same for the **Save Data** button.

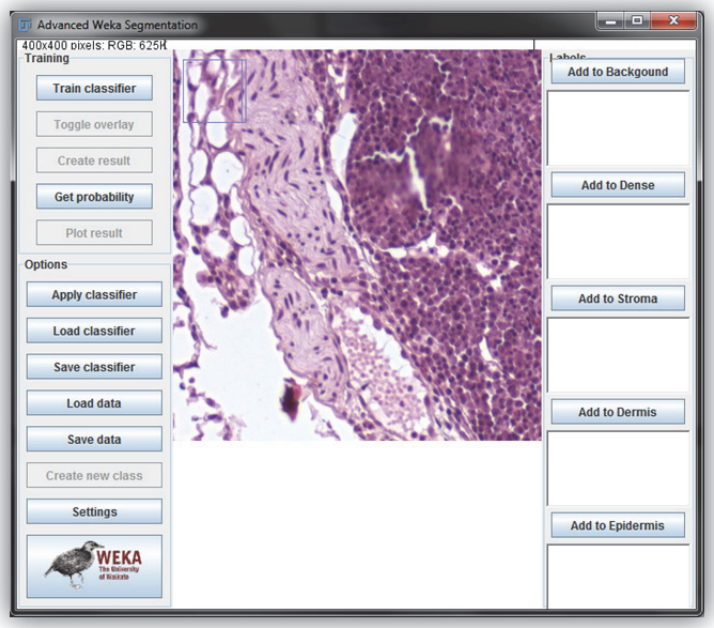


Running the Analysis

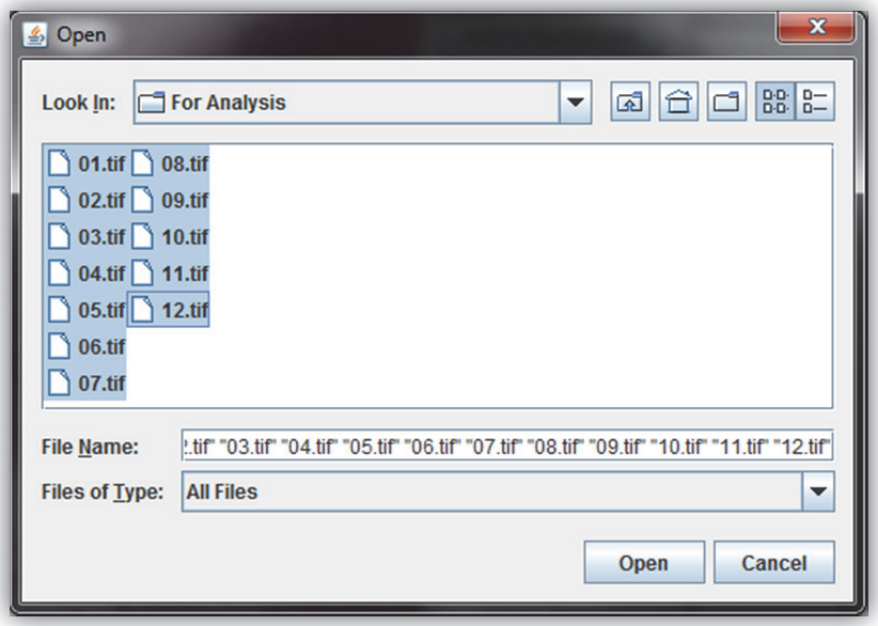
15. We are now ready to batch analyse some images. Because the workup is fairly intensive on the computer, and Java is not the best resource manager, it is best to close Fiji and restart it.
16. Once you have restarted Fiji open **01.tif** from the **Demo Images\Widefield Images\Advanced Segmentation\For Analysis**. And open up the **Weka Segmentation** module.



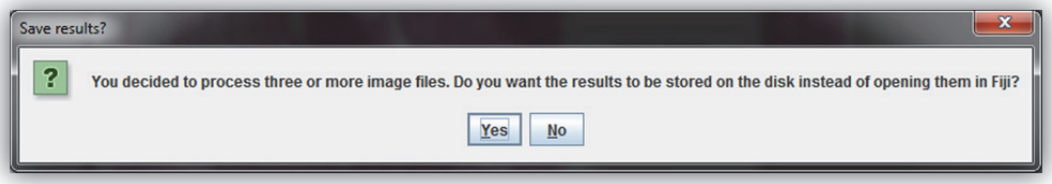
17. Press the **Load Classifier** button and load the previously save classifier file. Do the same with the **Load Data** button and the saved data file.

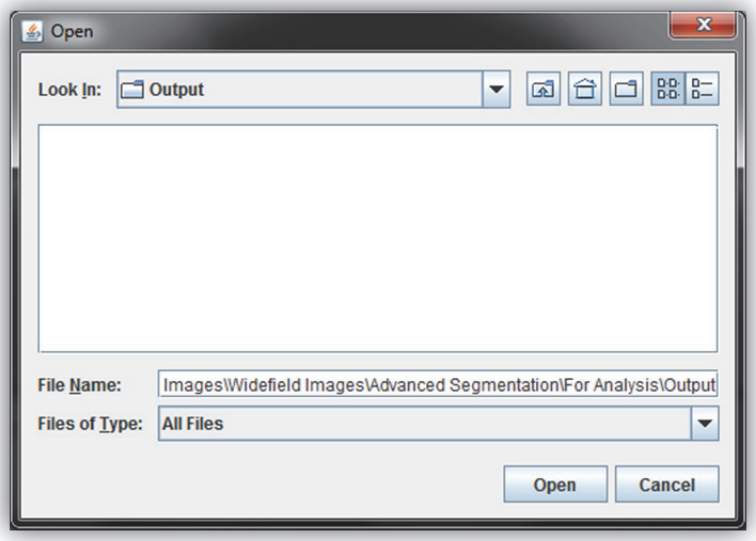


18. Now press the **Apply Classifier** button and navigate to the **Demo Images\Widefield Images\Advanced Segmentation\For Analysis** and select all the files in that folder. This images are 400 x 400 pixel regions extracted from the original image looked at at the beginning.

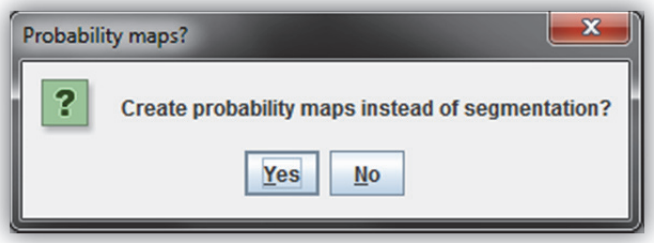


19. You will get the following message when you press **Open.** Select **Yes** and chose where to save the output images.

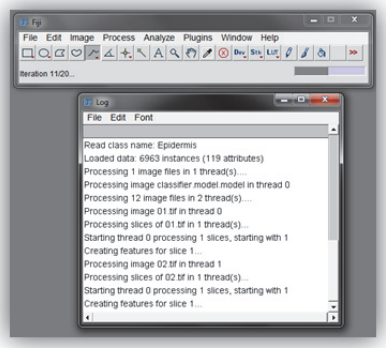




20. On the next message press **No**. This will generate cartoon type outputs instead of probability maps.

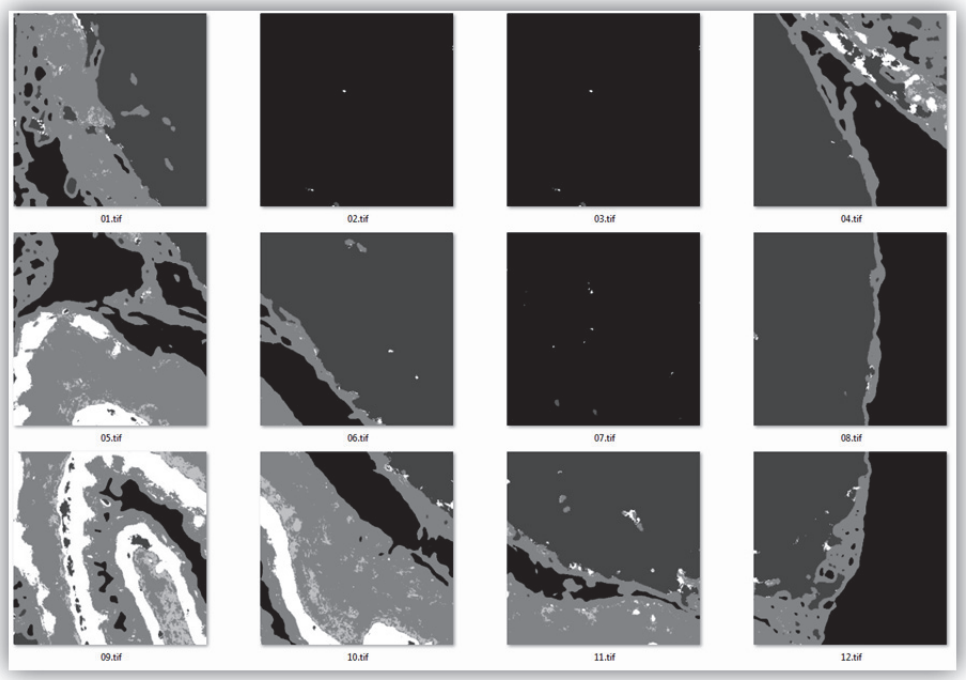


21. The processing of all the images can take some time and will more than likely bring you computer to a grinding halt. You can monitor the progress in the log and the Fiji status bar.

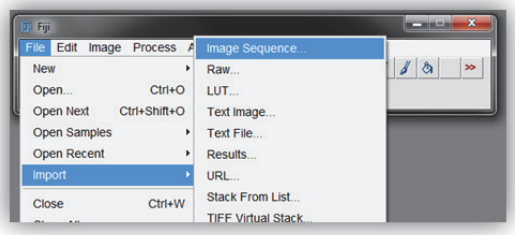


Reviewing the Data

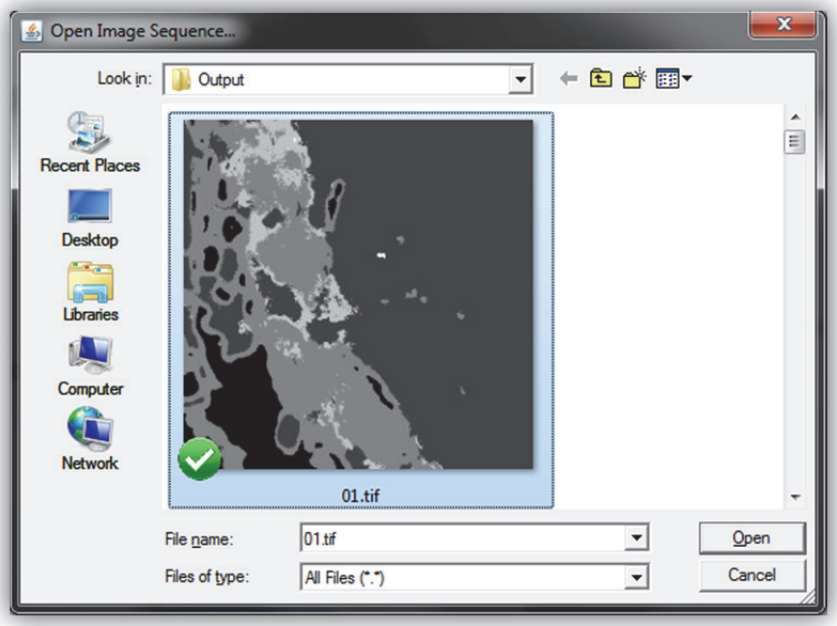
22. When the processing is complete you will have a folder with masks representing the different segmentation classes.



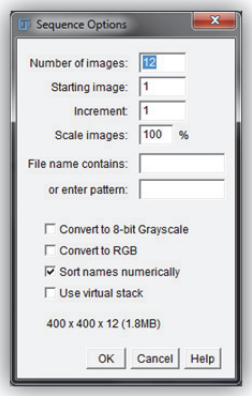
23. To create a montage of the images close the Weka module and go to **File → Import → Image Sequence…**



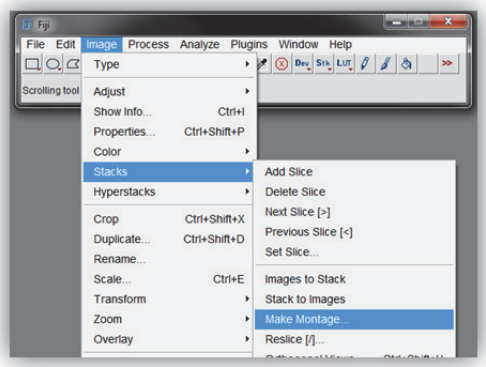
24. Select the first output mask (**01.tif** in this example).



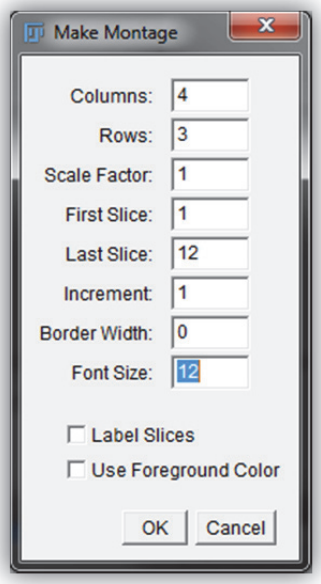
25. In the next window leave the settings at default and press **OK.**



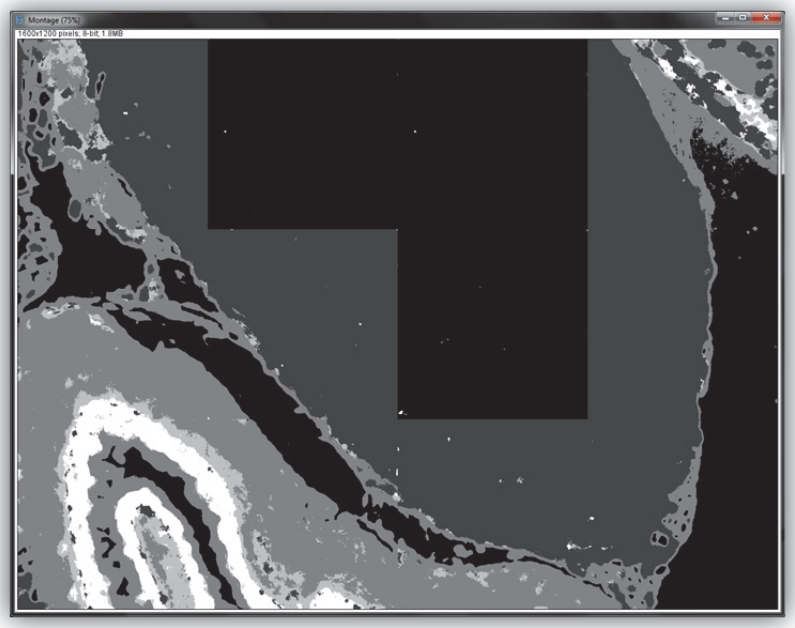
26. You will end up with a stack of images that can now be made into a montage by going to **Image → Stacks → Make Montage**…



27. Configure the **Make Montage** box for 4 **Columns** and 3 **Rows**. Also make sure the **Scale Factor** is set to 1 so the final image is not scaled down. Press **OK.**



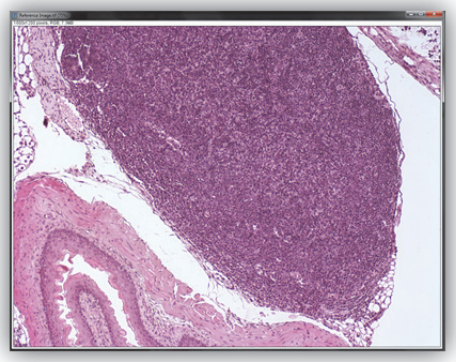
28. The final montage shows the results of the segmentation. The colours represent the different classes that were created. Class 1 (Background) is black, Class 5 (Epithelium) is white and the other classes are shades of grey in between: Class 2 (Dense Tissue) is dark grey, Class 3 (Stromal Tissue) is mid grey and Class 4 (Dermal Tissue) is light grey.

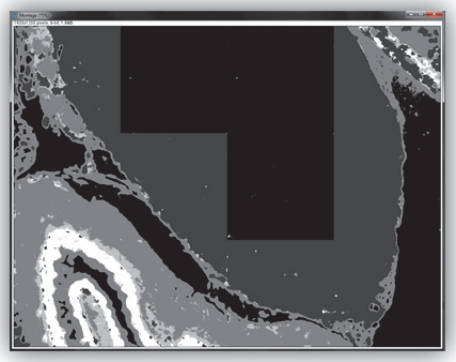


29. If you compare the results montage to the original image you will notice that there is one glaring error. 3 sections of the dense tissue have been identified at background. This is because the images used for these sections contained only dense tissue; because of this algorithms do not have enough to compare the different classes to and can result in errors.

    But in general the segmentation is fairly accurate and the few errors can be easily worked around.

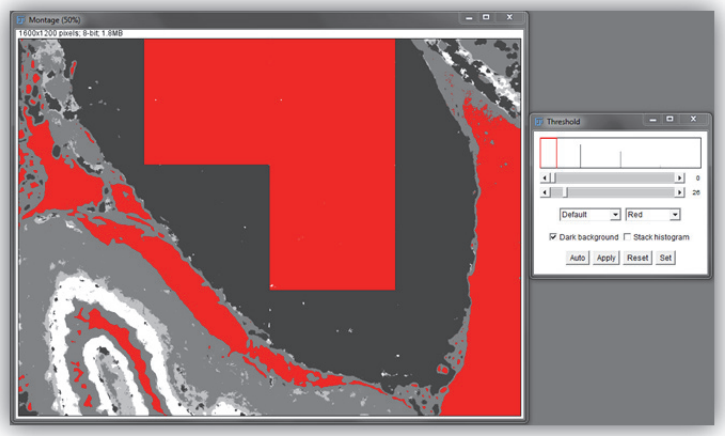
    You could also go through and retrain the segmentation on more images to get a more accurate segmentation.

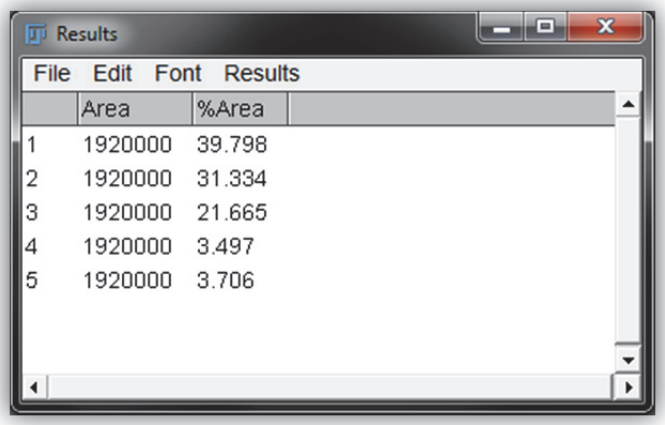




Analysing the Data

30. To analyse the data all that needs to be done is to threshold each of the colours in the montage image and measure it as any other thresholded image.





Advanced Module

About the Advanced Module

About

This section of the manual will show various examples of macro programing or scripting for Fiji. This examples only use the macro programing language of ImageJ which is essentially a derivative of Java. The Fiji scripting tool can be used to write code in other languages for use in Fiji such as Ruby, Clojure and Python.

The examples in this section start very simply by just recording steps in the macro recorder and progress to more advanced concepts such as batch processing, custom variables, data logging and looping.

The examples in this section are not an exhaustive list but should give a grounding in the basic concepts required to be able to write simple code. When developing your own code it is always best to have a solid understanding of what you are trying to automate before leaping in to coding it up. With that in mind the examples in this section are based around the manual tasks performed in the earlier part of the manual.

The end of each chapter contains the complete code of each script.

Basic Macro Recording

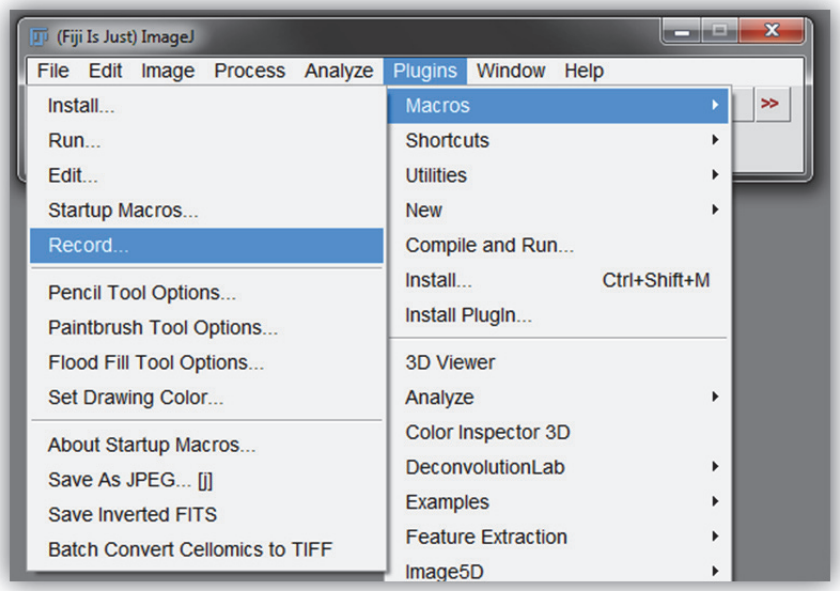
Aim

The simplest way to setup a macro in Fiji is to record the steps using the macro recorder. This can work for a simple macro but will not give a macro that is functional for multiple images. The macro recorder can be used for developing advanced macros and plug-ins to find which commands to use as they are not always intuitive or follow the same syntax.

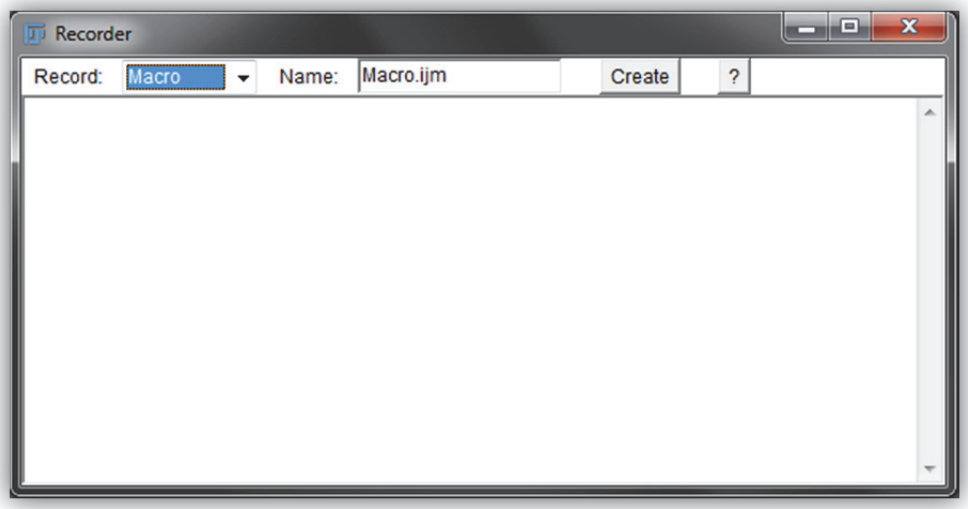
This technical note will show you how to record simple macros to measure the area of a thresholded stain and count the cells in an image.

The Macro Recorder

1. Open the macro recorder by going to **Plugins → Macros → Record…**

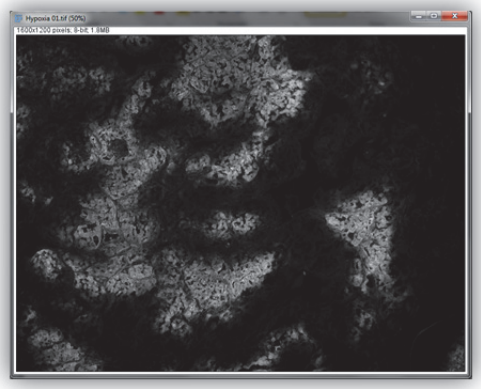


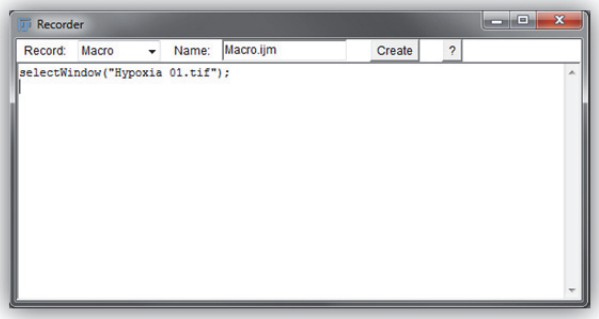
2. The **Recorder** window will open. Set **Record** to **Macro.** Everything now done in Fiji will be recorded by the recorder. Any steps you do not want can be highlighted and deleted.



Recording your First Macro – Threshold Measurement

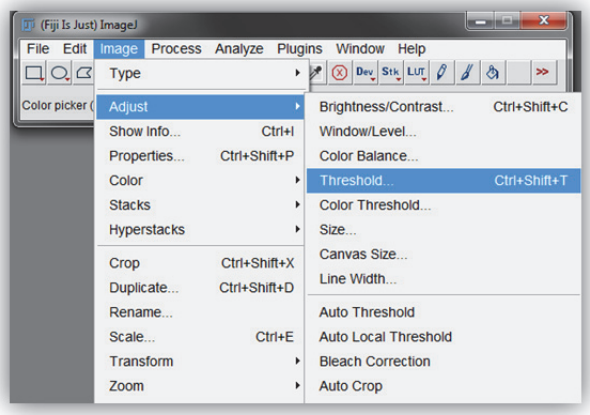
1. Open **Hypoxia 01.tif** from the **Demo Images\Widefield Images\Threshold Measurement** folder



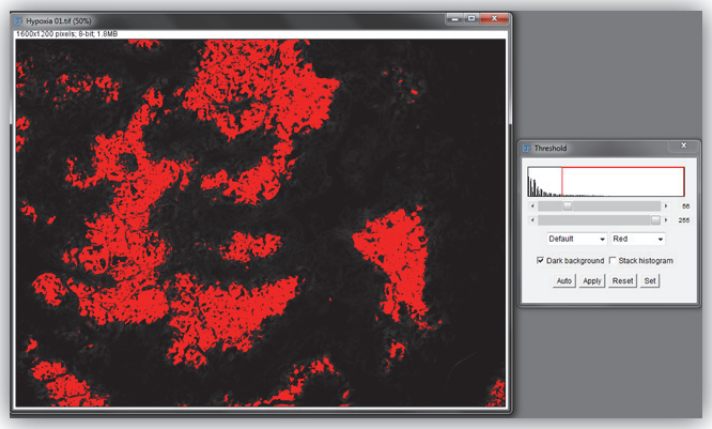


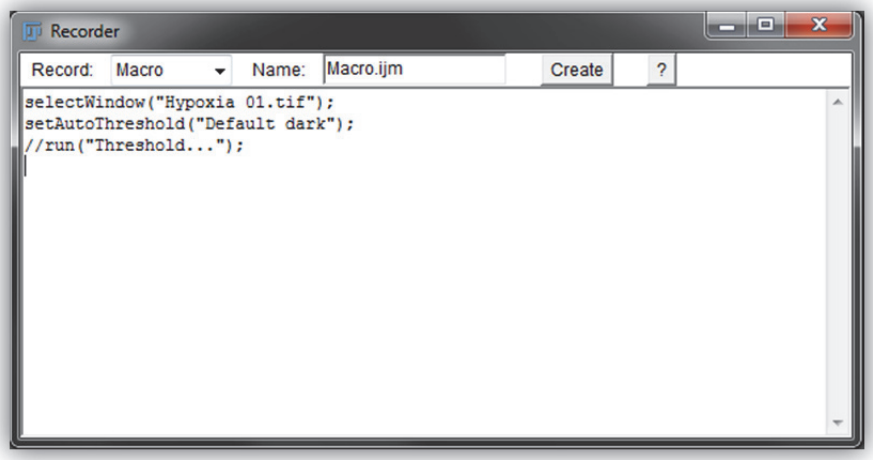
NOTE: If you drag and drop to open the image, the image recorder will record the step as **selectWindow**. If you use the open command, the image recorder will record the step as **open** with the file path.

2. Go to **Image → Adjust → Threshold**

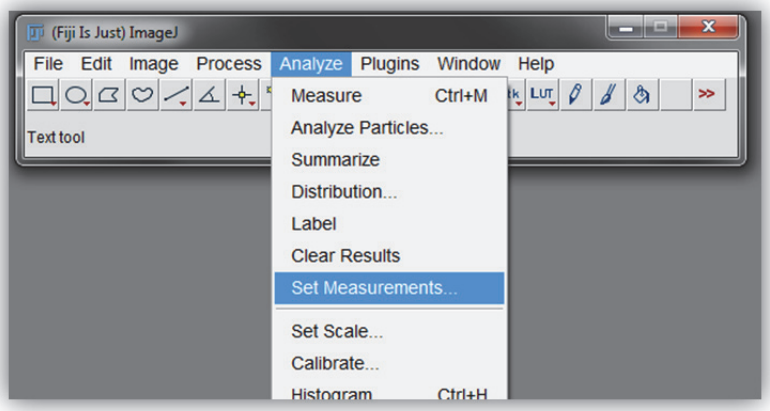


3. Leave the auto threshold setting as **Default.** The steps of setting the threshold will be added to the **Recorder** window.

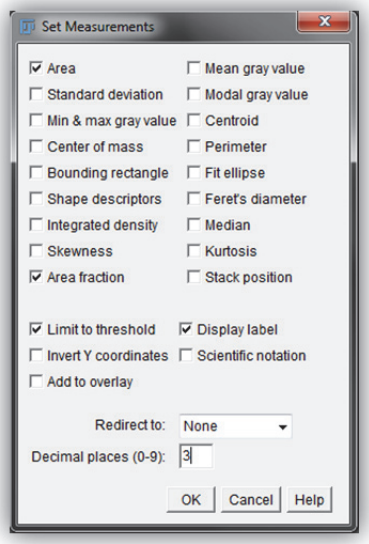


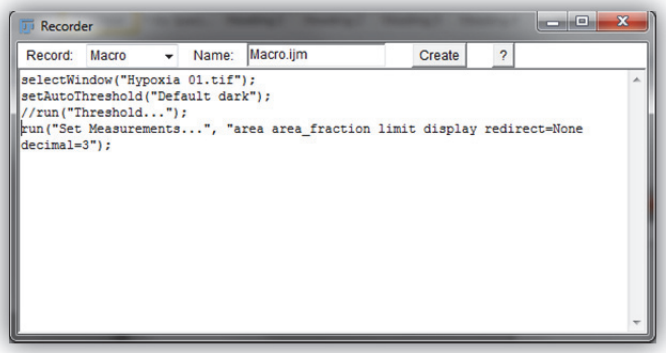


4. Go to **Analyze → Set Measurements…**

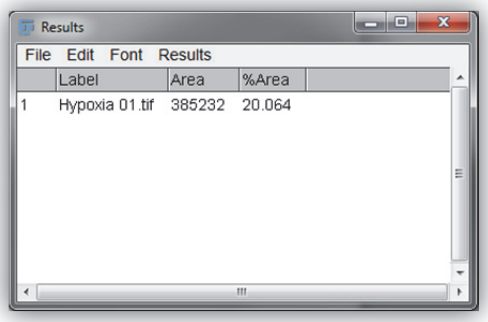


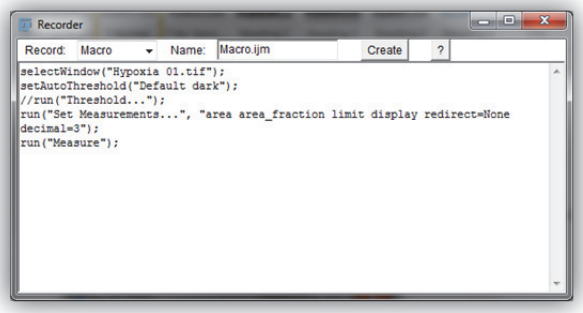
5. Configure the **Set Measurements** dialog as follows and press **OK.** The configuration should be saved into the **Recorder** window.



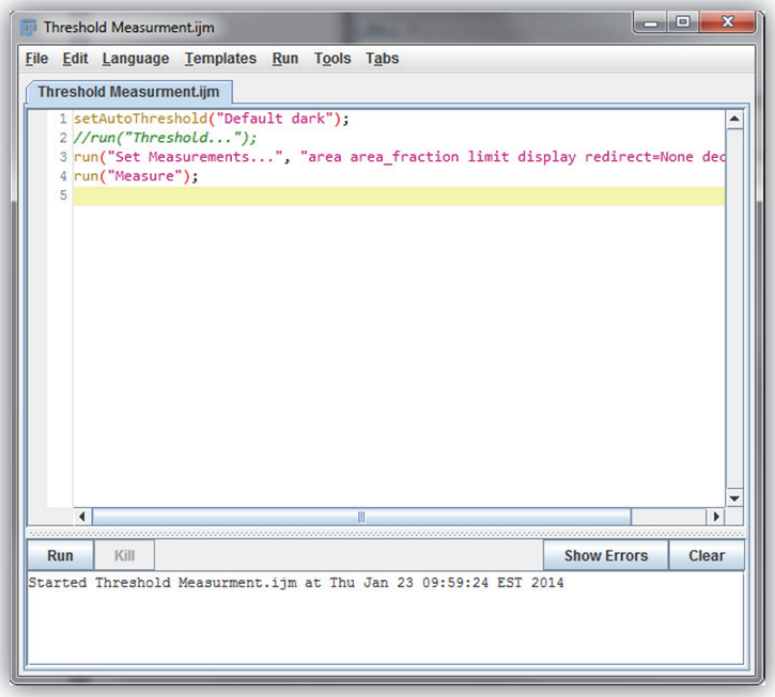


6. Go to **Analyze → Measure.** A **Results** table will be generated and the last step in the macro (run("Measure");) will be added to the **Recorder**



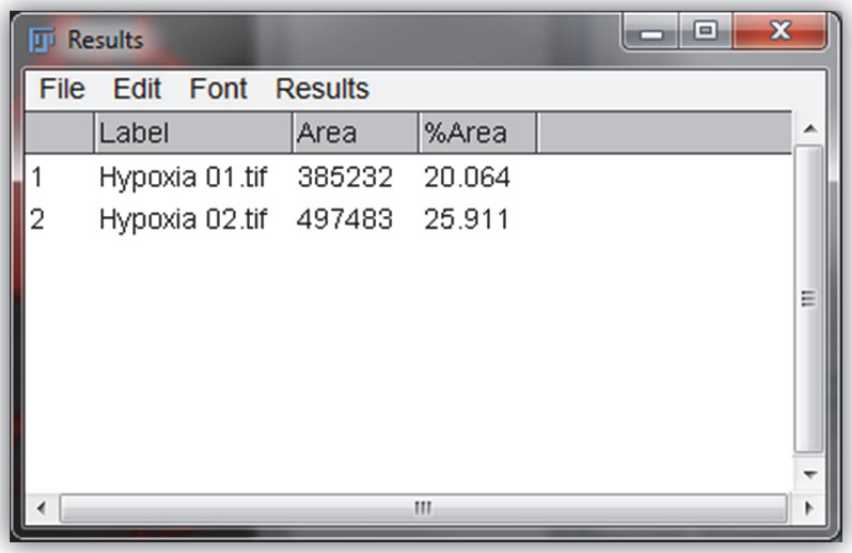


7. To complete the macro give it a name in the **Name:** box (e.g. Threshold Measurement – make sure you leave the .ijm extension on). You will also need to delete the first step that was created (selectWindow…) otherwise the macro will not work on any other images (unless they are called Hypoxia 01.tif). Press the **Create** button. It will create the macro in Fiji's macro editor (the function of which will be detailed later).



8. Save the macro by going to **File → Save As..**

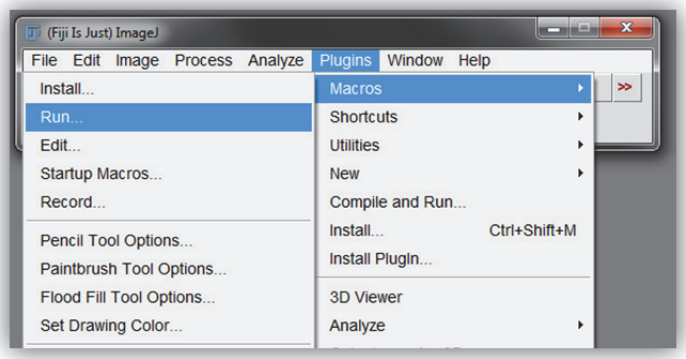
9. To run the macro select the **Hypoxia 01** image and press the **Run** button on the macro editor.
10. Open **Hypoxia 02.tif** from the **Demo Images\Widefield Images\Threshold Measurement** folder and press the **Run** button again.



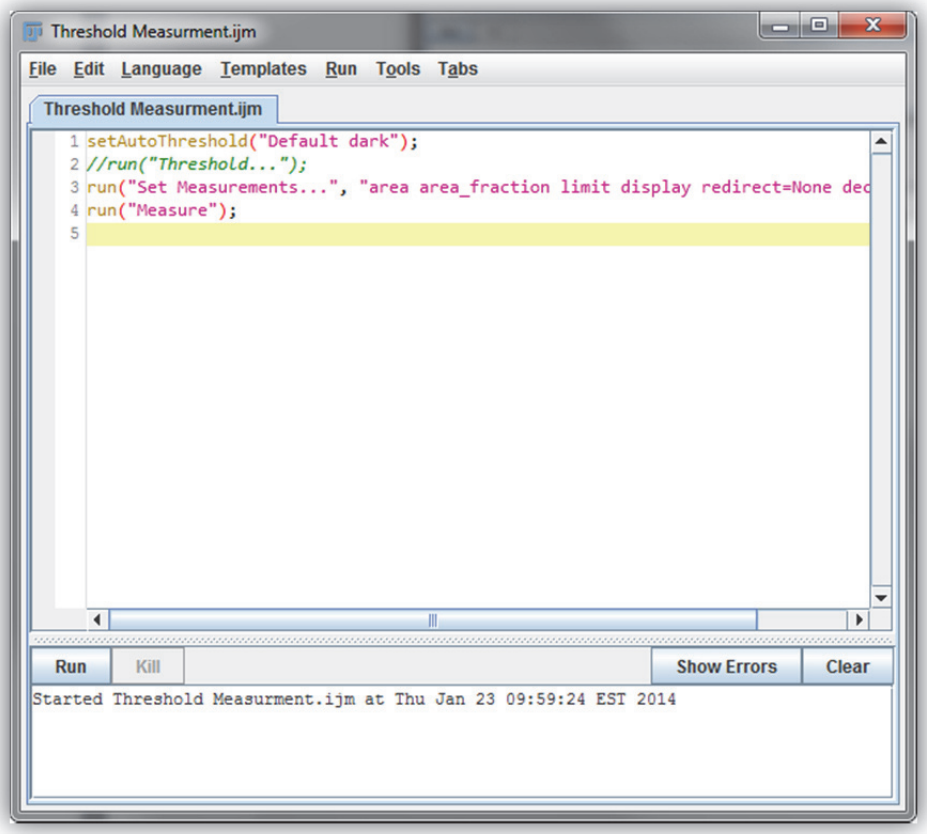
Re-running the Macro

To run the macro in the future you have several options

1. Open your image and go to **Plugins → Macro → Run.** Select the macro in the file dialog and press **Open**



2. Open the macro in Fiji's **Macro Editor** and press the **Run** button. To do this either drag and drop the macro file into Fiji or open it using the **File → Open** command.

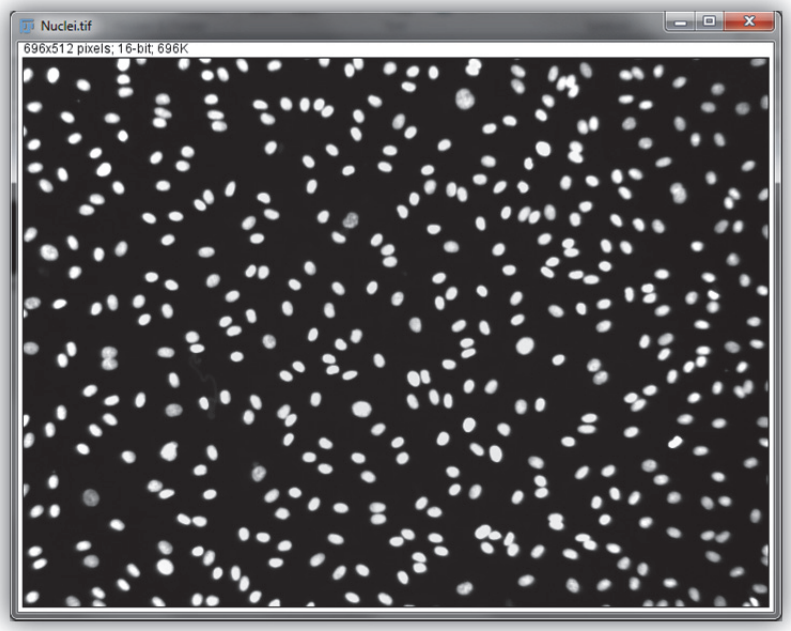


3. Open the image to analyse in Fiji and then double click the macro file in explorer/finder. **NOTE:** the first time you do this you will have to associate .ijm files with Fiji

Recording your Second Macro – Cell Counting

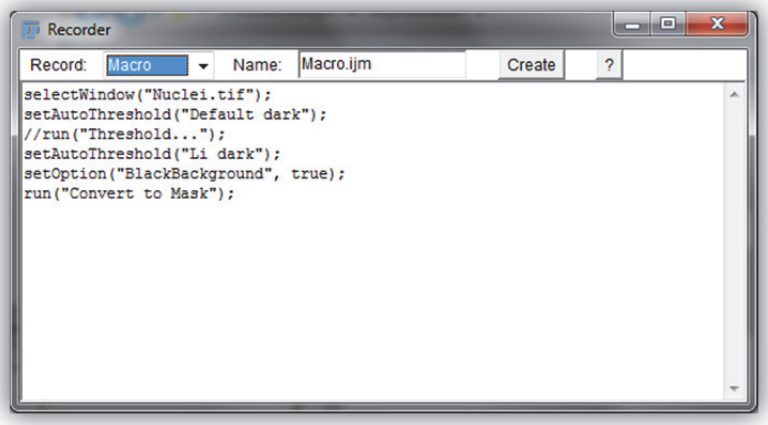
To be able to count cells we will use a similar process of recording steps as above.

1. Start the macro recorder and open **Nuclei.tif** from the **Widefield Images → Segmentation** folder

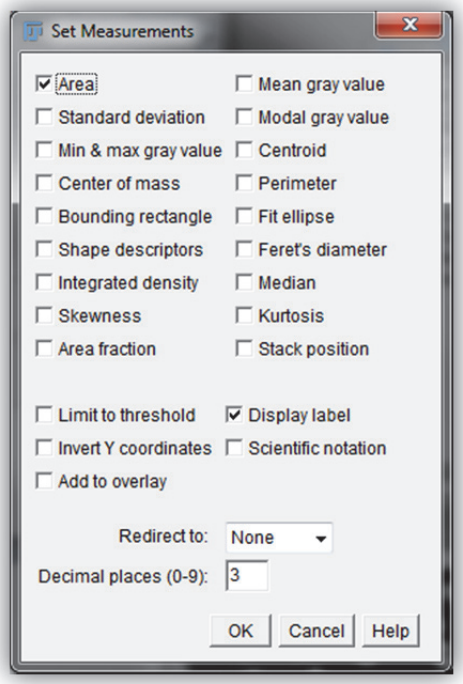


2. Apply a threshold by going to **Image → Adjust → Threshold.** The **Default** auto threshold should be applied, in this case though we want to use the **Li** algorithm as it will work more robustly on images with varying intensities. Select the **Li** algorithm and press the **Apply** button.

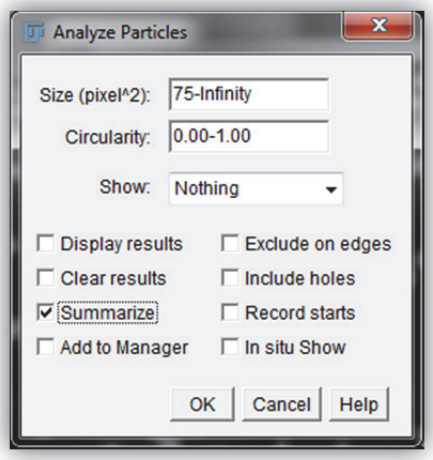
   You should end up with the **Recorder** window looking something like that below.



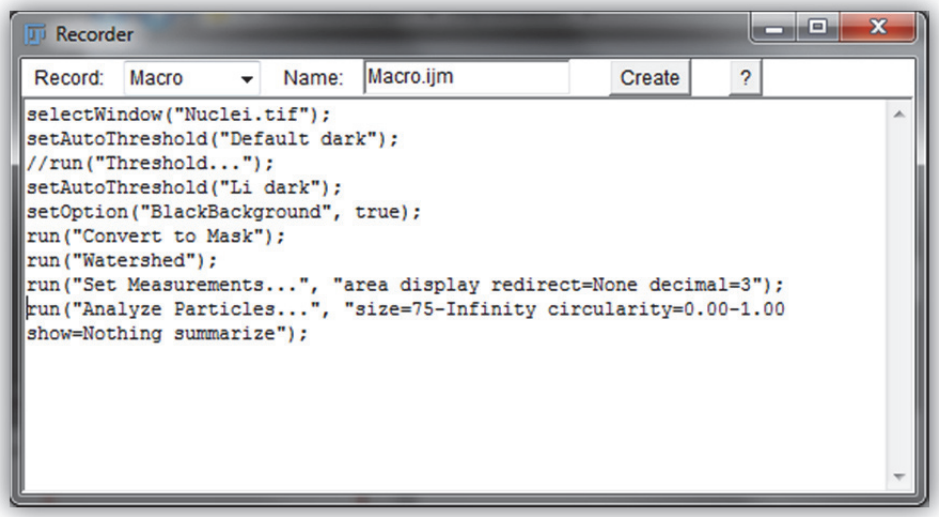
3. Next we need to run a watershed function to separate any touching nuclei. Go to **Process → Binary → Watershed**
4. Configure the **Set Measurements…** window as follows and press the **OK** button



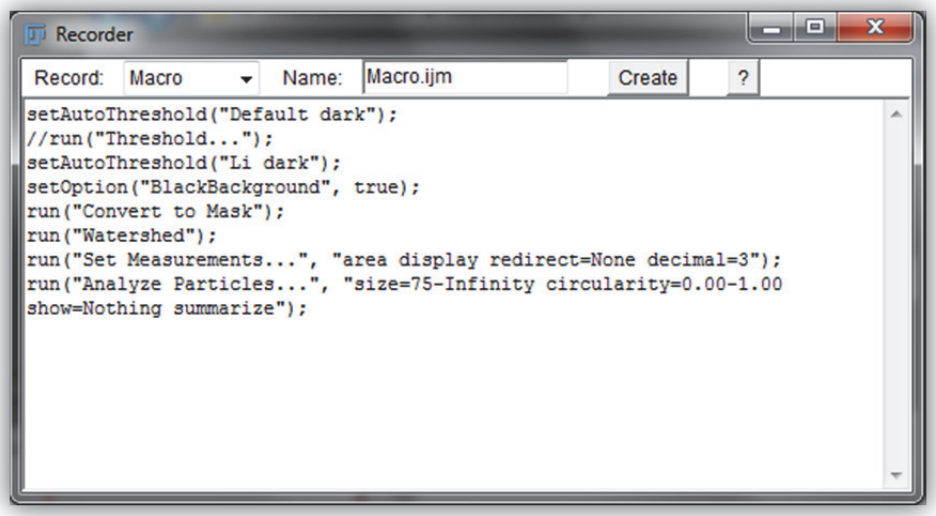
5. Configure **Analyse Particles** (**Analyze → Analyse Particles…)** as follows and press **OK**



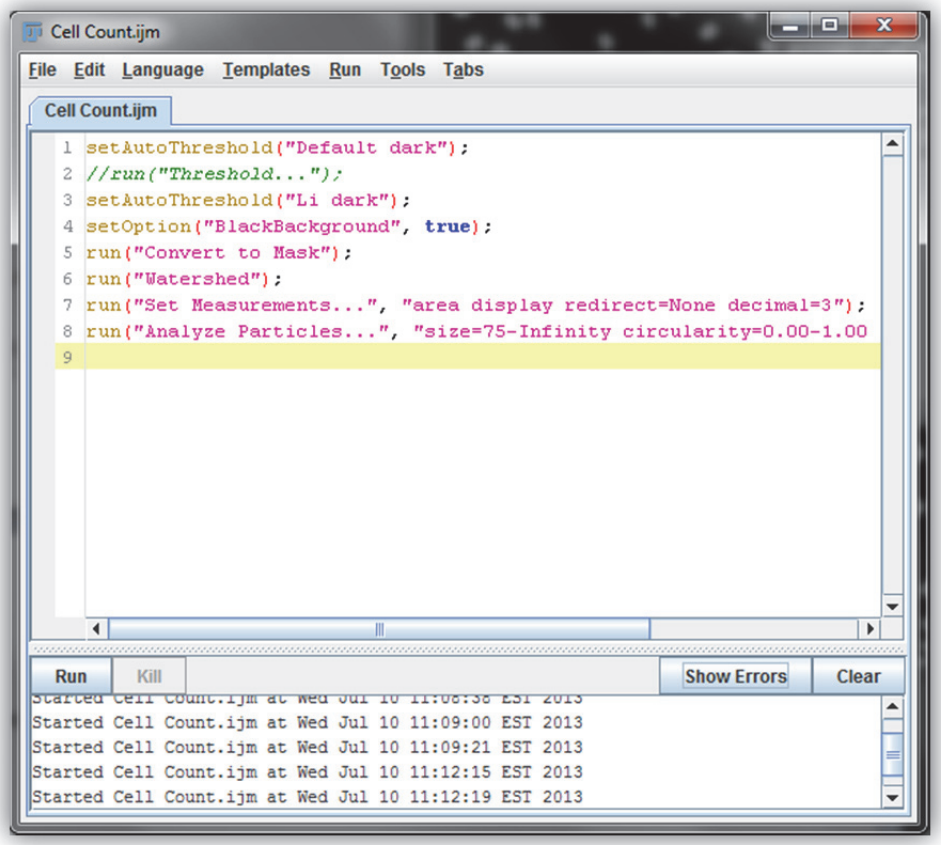
6. The macro is now complete and you should have a recorder window that looks something like the following.



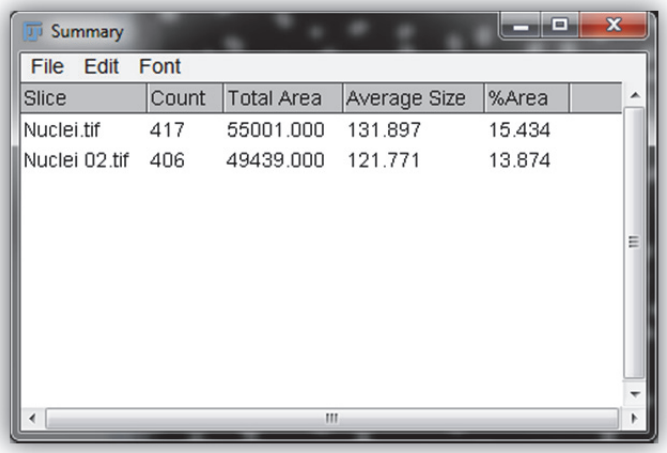
7. Delete the first step as before (**selectWindow…)** and any other steps that have been recorded that are not needed until you have only the following in the recorder window.



8. Name the Macro (e.g. **Cell Count.ijm**) and press the **Create** button. Then save the macro as previously



9. You can now run the macro on **Nuclei.tif** and **Nuclei 02.tif** from the **Demo Images\Widefield Images\Segmentation** folder to get a count of the cells.



An Introduction to Variables

Aim

In the previous technical note two macros were created by recording the steps of the functions applied to them. These macros could then be applied to any image as all the modifications and measurements were performed on the originally opened image.

If you need to perform functions on multiple images, or create copies/variations of the originals the only way this can be achieved is to write actual macro code (that can be copied and modified from the macro recorder if required) and assign custom variables.

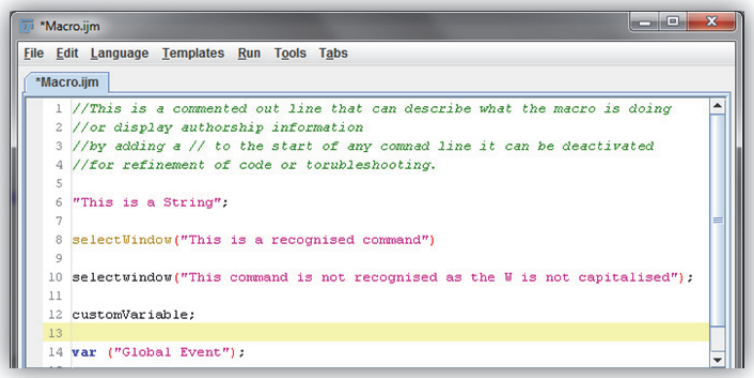
In this technical note we will write a macro to merge fluorescent and DIC images together. This will require the use of some user prompts and variables as Fiji will need to be told which images to work with.

We will also make a macro to split a merged image and measure the percentage area of two different stains in it. This will involve chaining variables together to achieve what we need.

The Macro Editor

The Fiji macro editor makes writing macros somewhat easier as it will recognise (usually) known ImageJ macro commands and colour them accordingly. This gives you the ability to know if the command you entered will be accepted as you planned. In the example below you can see how the editor will handle some of the functions

Green = Comment. These can be instructions on what the macro is doing, or the // can be used to disable a given step
Pink = String. Strings are contiguous lines of text that maybe image names, commands or other functions.
Yellow = Recognised command (note the example below it is black as it is not recognised)
Black = Custom variable or unrecognised command
Blue = Global event/command/variable. Global events apply to the whole macro code, not just the part they are contained in.



NOTE: Each line ends with a semi-colon; this indicates the end of a line to the macro language. Also note that the commands are case sensitive selectWindow is recognised while selectwindow is not.

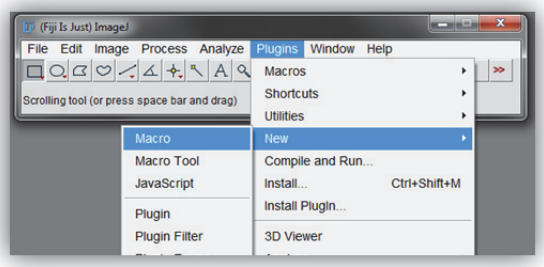
Variables

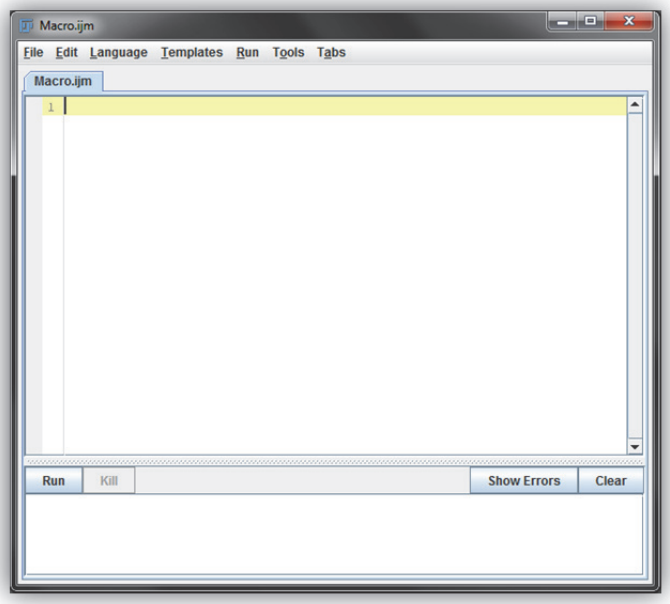
Variables are a common element of nearly every programing language. They are elements of code that either contain a value created by the program (e.g. the number of open images, or the result of a cell count). They are also used to store values (numbers, strings, other results) for further use in the code. These are commonly referred to as custom variables as they are assigned by the creator of the code and can be customised to do what is needed.

Once something is stored in a variable it can be recalled at any time it is required in the code.

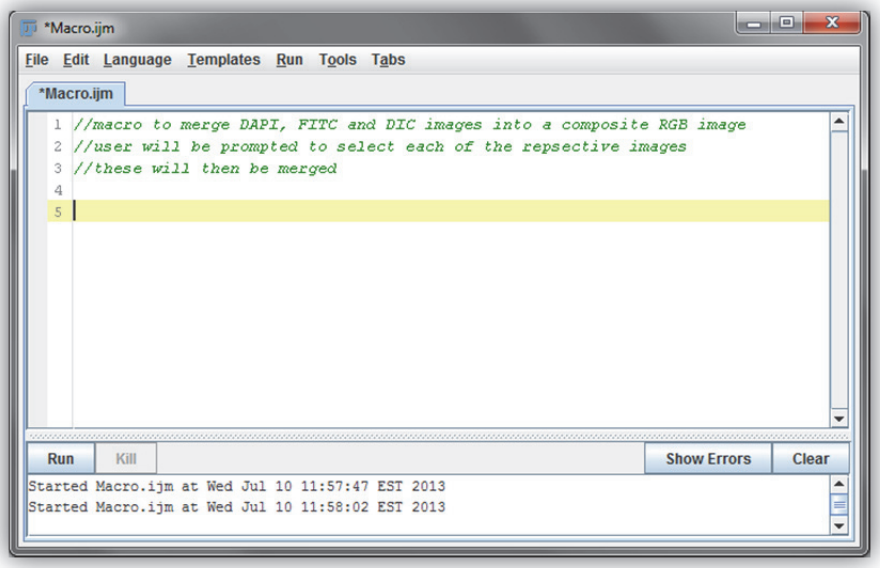
Writing your first Macro – Automatically Merge Images

1. Open the Fiji Macro editor by going to **Plugins → New → Macro**





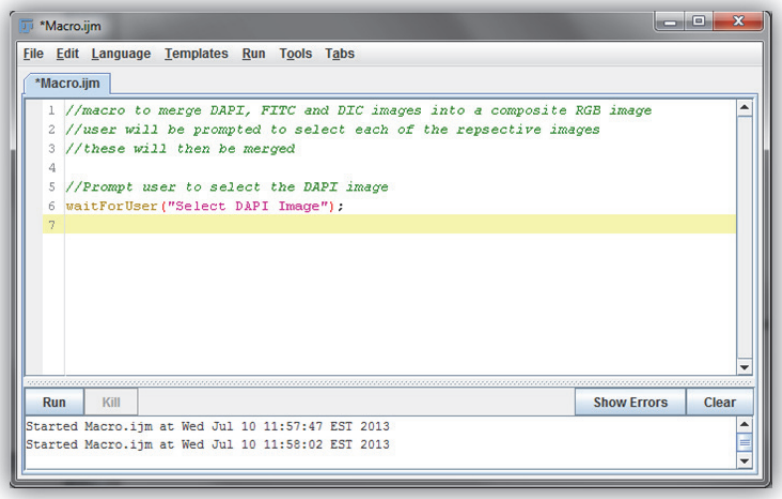
2. Before going any further it is good practice to add comments at the start of the code explaining what it is meant to do, who wrote it and any other information such as required plugins etc.



3. This macro will work by asking the user to select the DAPI, FITC and DIC images in turn and then merging them. To start with we need to add the steps to prompt the user to select each image. Enter a comment explaining the first step and then enter

waitForUser("Select DAPI Image");

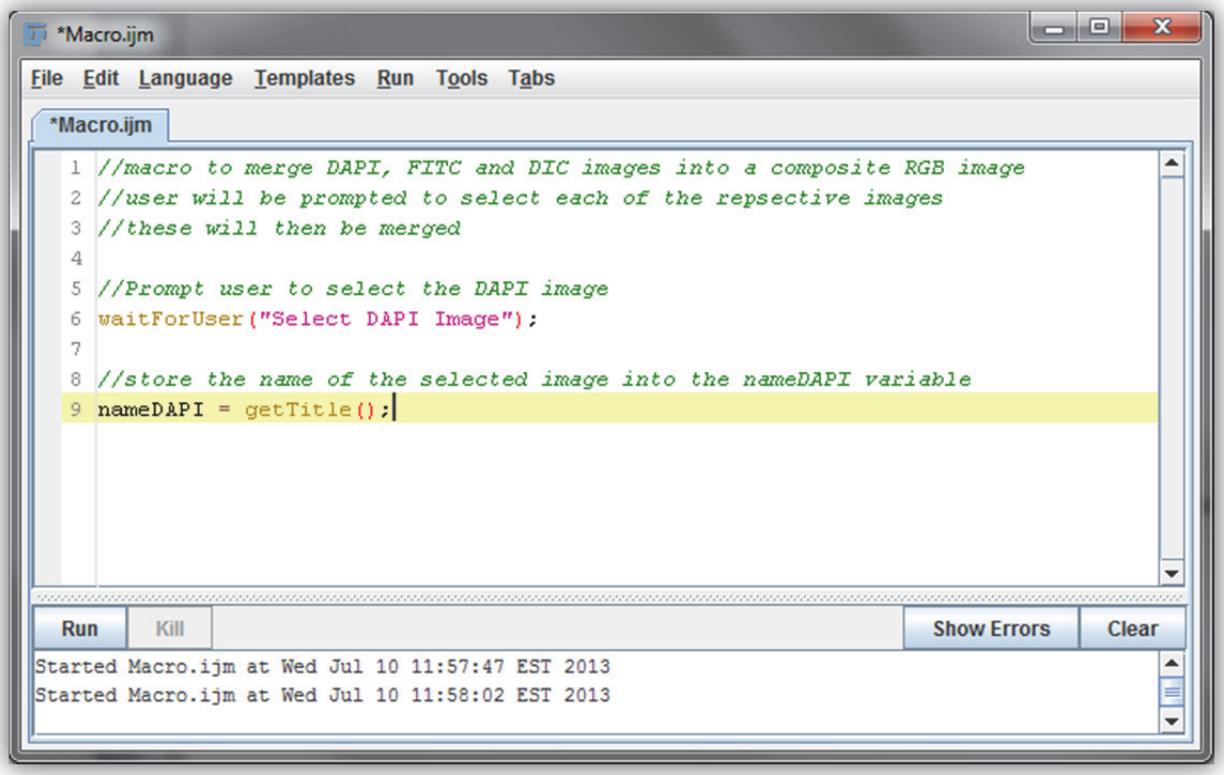
This will create a dialog box that will pause the macro, it will have "Select DAPI Image" written in it (allowing the user to select the DAPI image) and will have an OK button to click once the user has performed the action



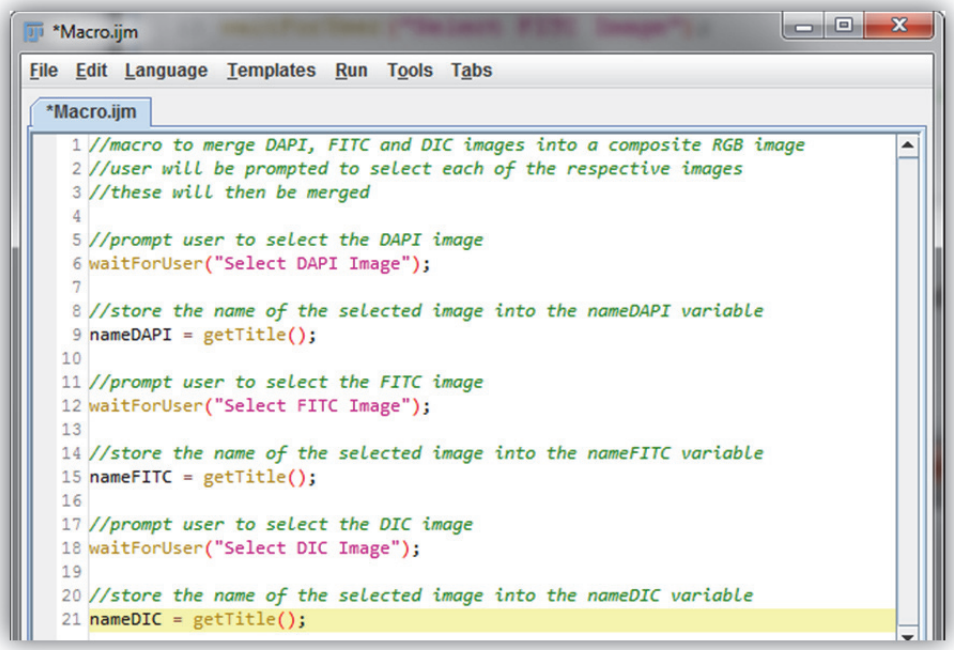
4. The next step is to store the name of the selected image into a custom variable that we can recall later on. Add another comment about the next step and then enter

   nameDAPI = getTitle();

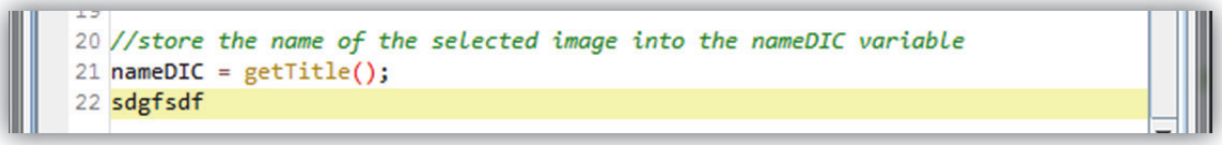
   The getTitle() command will return the title of the currently active window. It can also be used, by placing a name or variable between the brackets, to get the title of a non-active window



5. Repeat the above steps to add selection commands for the FITC and DIC images. In this example nameFITC and nameDIC have been used for the custom variables to store the image names.

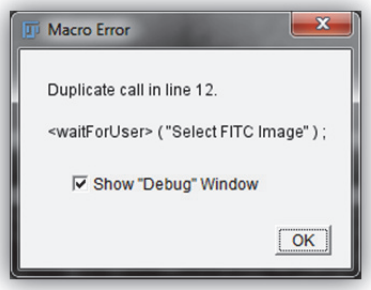


6. To test the functionality of the macro to this point add a string of garbage characters and the end of the macro. This will cause a crash that will let you open a debug window to see if variables are being populated properly.



7. Open **DAPI.tif, FITC.tif** and **DIC.tif** from the **Demo Images\Widefield Images\DIC and Fluorescent** folder. Run the macro and follow the instructions you created.

8. You will likely get this error at some point. This is because the macro has tried to move on to the next step without waiting long enough for the previous step to finish.



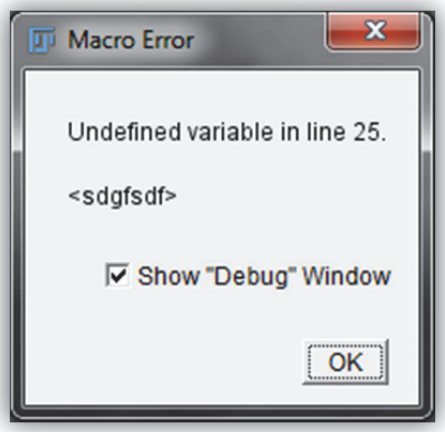
9. This can be easily fixed by adding a very short delay after the name storing steps. Add

wait(100);

after each of the nameXXX steps. This pauses the macro for 100ms to give it a chance to catch up.

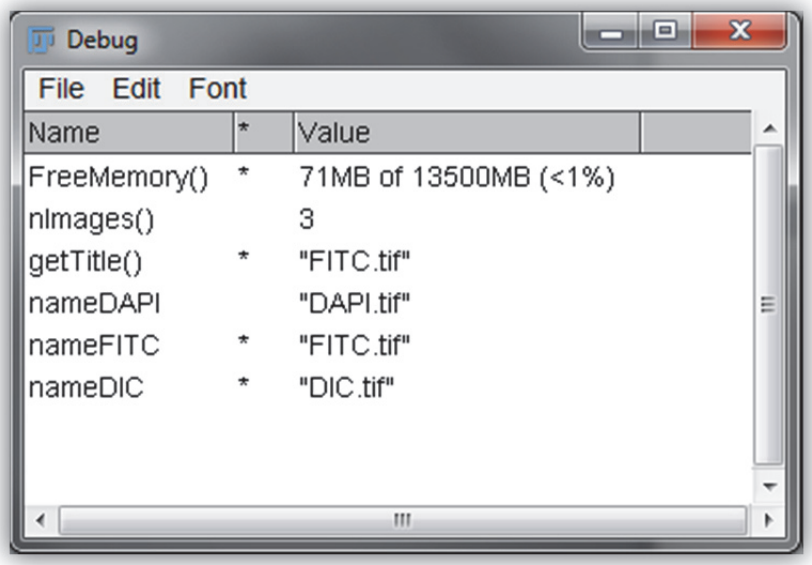
```
//store the name of the selected image into the nameDAPI variable
nameDAPI = getTitle();
wait(100);

//prompt user to select the FITC image
waitForUser("Select FITC Image");
```

10. Run the macro again, this time it should make it to the end, hit the garbage characters and throw up this message



Tick the **Show "Debug" Window** box and press **OK**

11. The **Debug** window will show you information stored by the macro when it ran to the point of crashing



For this example you can see the amount of free memory, the number of open images (nImages) the title of the currently active window (getTitle) and the names assigned to the three custom variables that were created.

If everything matches up we are good to continue.

12. Delete the garbage characters from the end of the script. Add a comment about the next step of merging the images and add the following command

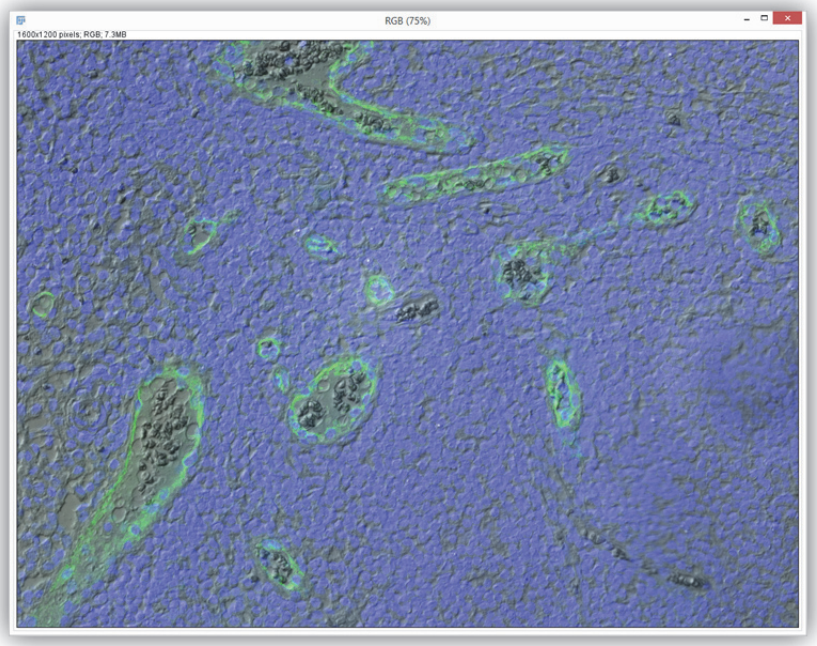
run("Merge Channels...", "c2="+nameFITC+" c3="+nameDAPI+" c4="+nameDIC+" ignore");

```
25
26 //merge the DAPI, FITC and DIC images into the Blue, Green and Grey Channels
27 run("Merge Channels...", "c2="+nameFITC+" c3="+nameDAPI+" c4="+nameDIC+" ignore");
28
```

This command will run the merge channels command and put the images whose names match the created variables into channels 2, 3 and 4.

A lot of commands in ImageJ are created as one long text string. To be able to insert our custom variable into this string we need to break it, add the variable and then continue on. This can be seen for example in the “C2=”+nameFITC+” c3… part of the command. A normal string would have the image name contained in the string (e.g. “c2=FITC.tif c3=…) but as we want to be able to address the custom variables it has to be broken.

13. Open the three images again if you haven't already got them open and run the macro. The end result should be a single RGB merge image of the three channels.



14. If it all worked, save the macro as **Merge DAPI, FITC and DIC.ijm**. Congratulations you have coded your first ImageJ macro.

Adding Custom Naming

The code written to this point works fine, but the end result is an image called RGB. This probably doesn't matter as the name can be changed when you save the image but for this exercise let's add some additional code to allow the entry of a name

1. Add the following code to the top of your existing code (below the initial comments about the scripts function). This will put up a box with the prompt of "Enter Name for Final Merge Image" on it. The box will be blank, if you wanted a default value in it you can add it between the second set of quote marks. The name entered will be stored in the **nameStore** variable for use later

nameStore=getString("Enter Name for Final Merge Image", "");

```
//these will then be merged

//prompt the user for a name for the final image
nameStore=getString("Enter Name for Final Merge Image", "");

//prompt user to select the DAPI image
waitForUser("Select DAPI Image");
```

2. To make use of this stored name we now need to put some extra code at the end of the script.

   selectWindow("RGB");

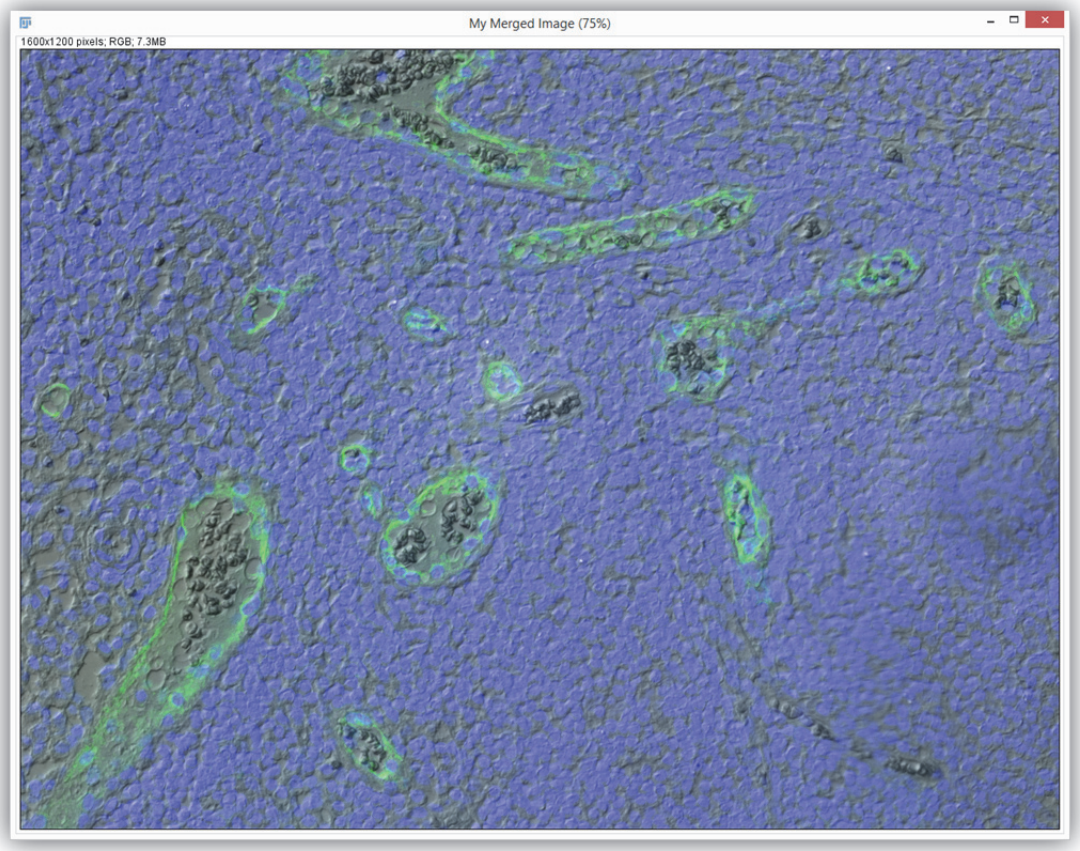
   This will make sure the merged image is the active window

   rename(nameStore);

   This will rename the image to the text stored in the **nameStore** variable

```
29 //merge the DAPI, FITC and DIC images into the Blue, Green and Grey Channels
30 run("Merge Channels...", "c2="+nameFITC+" c3="+nameDAPI+" c4="+nameDIC+" ignore");
31
32 //select the RGB image and rename it to the custom entered name
33 selectWindow("RGB");
34 rename(nameStore);
```

3. Now open the images again and rerun the macro to give your image a custom defined name



Merge DAPI, FITC and DIC.ijm – Final Code

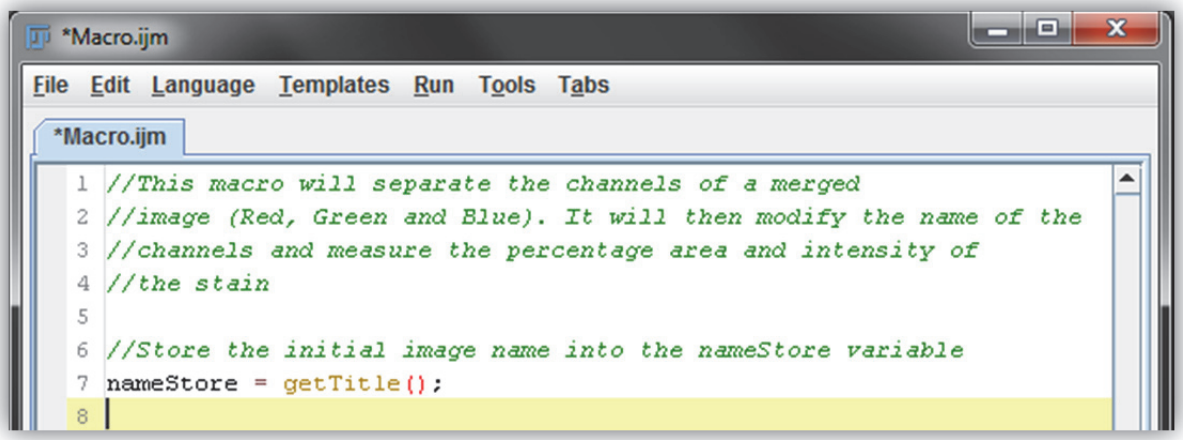
```
//macro to merge DAPI, FITC and DIC images into a composite RGB image
//user will be prompted to select each of the respective images
//these will then be merged

//prompt the user for a name for the final image
nameStore=getString("Enter Name for Final Merge Image", "");

//prompt user to select the DAPI image
waitForUser("Select DAPI Image");

//store the name of the selected image into the nameDAPI variable
nameDAPI = getTitle();
wait(100);

//prompt user to select the FITC image
waitForUser("Select FITC Image");

//store the name of the selected image into the nameFITC variable
nameFITC = getTitle();
wait(100);

//prompt user to select the DIC image
waitForUser("Select DIC Image");
wait(100);

//store the name of the selected image into the nameDIC variable
nameDIC = getTitle();

//merge the DAPI, FITC and DIC images into the Blue, Green and Grey Channels
run("Merge Channels...", "c2="+nameFITC+" c3="+nameDAPI+" c4="+nameDIC+" ignore");

//select the RGB image and rename it to the custom entered name
selectWindow("RGB");
rename(nameStore);
```

Writing Your Second Macro – Channel Separate and Threshold Measurement

1. Close any open windows and tables. Then open a new blank macro in the macro editor.
2. Add comments at the start explaining the function of the macro and then add the first step

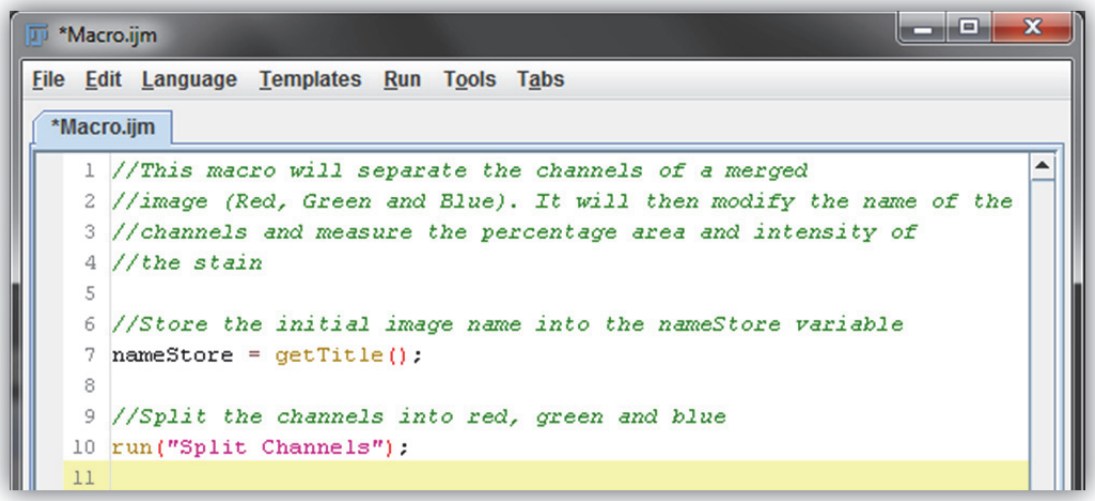
   nameStore = getTitle();

   to store the name of the initial open image into the nameStore variable for later use



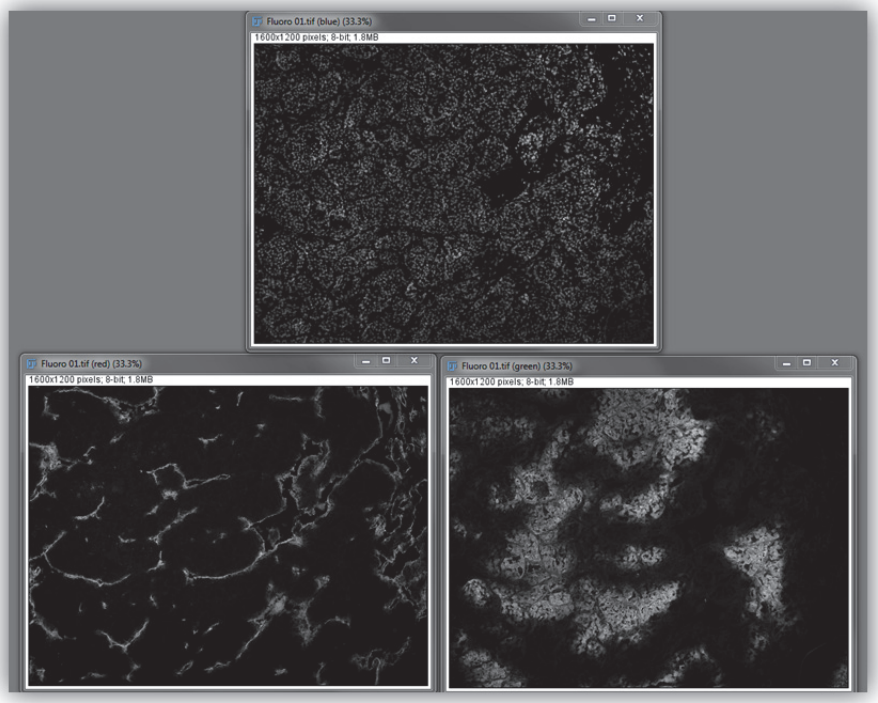
3. The next step is to split the channels of the image with the command

   run("Split Channels")



   This will close the original image and create three new images with the name of the original and a suffix describing the channel it was (red, green or blue).

4. Test the macro at this point by opening **Fluoro 01.tif** from the **Demo Images\Widefield Images\Fluorescent Measurement** folder and running the macro. The result should be three grey scale images representing the different channels



5. The next step is to rename the images to something that will make more sense in the results table that just image name (red). Add a comment about the next step and enter the following three commands

```
selectWindow(nameStore+" (red)");
rename(nameStore+" – Vessels");
redImage = getTitle();
```

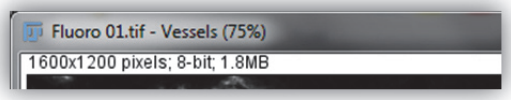
The first step will select the image that has the original file name with (red) on the end of it (e.g. Fluoro 01 (red)). The second step renames that image to the original name with – Vessels on the end (e.g. Fluoro 01 – Vessels). The third step saves the new name into the variable redImage for later recall. The third step isn't necessary as you can always recall the image with nameStore+" – Vessel" but it makes it easier to work with.

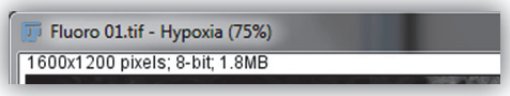
```
12 //Rename Red Image to original name + vessels and green image to
13 //original name + hypoxia
14 selectWindow(nameStore+" (red)");
15 rename(nameStore+" - Vessels");
16 redImage = getTitle();
```

6. Repeat the above steps for the green channel image, renaming it to nameStore+" – Hypoxia"

```
12 //Rename Red Image to original name + vessels and green image to
13 //original name + hypoxia
14 selectWindow(nameStore+" (red)");
15 rename(nameStore+" - Vessels");
16 redImage = getTitle();
17
18 selectWindow(nameStore+" (green)");
19 rename(nameStore+" - Hypoxia");
20 greenImage = getTitle();
```

7. If you close any open images, reopen **Fluoro 01.tif** and run the macro it should rename the images as follows





8. The next step is to threshold and measure the images. Add comments and the following steps to configure the measurements

```
run("Set Measurements...", "area mean area_fraction limit display redirect=None
decimal=3");
```

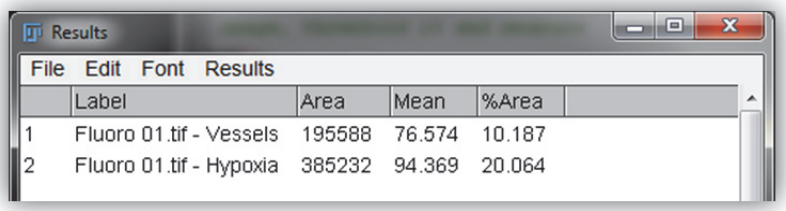
threshold the images and measure them.

```
selectWindow(redImage);
setAutoThreshold("Yen dark");
run("Measure");

selectWindow(greenImage);
setAutoThreshold("Default dark");
run("Measure");
```

NOTE: Two different auto threshold algorithms are used for the different channels. The Yen algorithm is used for the vessel image as the default one will result in false positive selection when the image has only small areas of signal.

```
//configure measurments for threshold area and average intenisty
run("Set Measurements...", "area mean area_fraction limit display redirect=None decimal=3");

//Select the vessel image, threshold it and measure
selectWindow(redImage);
setAutoThreshold("Yen dark");
run("Measure");

//select the hypoxia image, threshold it and measure
selectWindow(greenImage);
setAutoThreshold("Default dark");
run("Measure");
```

9. If you close all images, reopen Fluoro 01.tif and run the macro you should get a result table like this



10. All that is left to do is add one more step to the macro to close all open images. Add the

    run("Close All");

    command to the end of the macro

11. Save the macro as **Vessel and Hypoxia Measurement.ijm** and try it out on **Fluoro 02.tif**

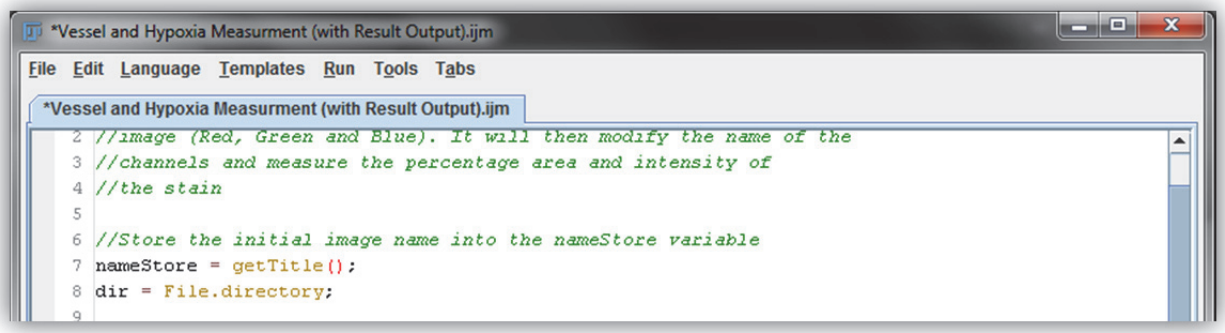
Vessel and Hypoxia Measurment.ijm – Final Code

```
//This macro will separate the channels of a merged
//image (Red, Green and Blue). It will then modify the name of the
//channels and measure the percentage area and intensity of
//the stain


//Store the initial image name into the nameStore variable
nameStore = getTitle();

//Split the channels into red, green and blue
run("Split Channels");

//Rename red image to orignal name + vessels and green image to
//original name + hypoxia
selectWindow(nameStore+" (red)");
rename(nameStore+" - Vessels");
redImage = getTitle();

selectWindow(nameStore+" (green)");
rename(nameStore+" - hypoxia");
greenImage = getTitle();

//configure measurments for threshold area and average intenisty
run("Set Measurements...", "area mean area_fraction limit display redirect=None decimal=3");

//Select the vessel image, threshold it and measure
selectWindow(redImage);
setAutoThreshold("Yen dark");
run("Measure");

//select the hypoxia image, threshold it and measure
selectWindow(greenImage);
setAutoThreshold("Default dark");
run("Measure");

//close all open images
run("Close All");
```

Outputting a Result Overview

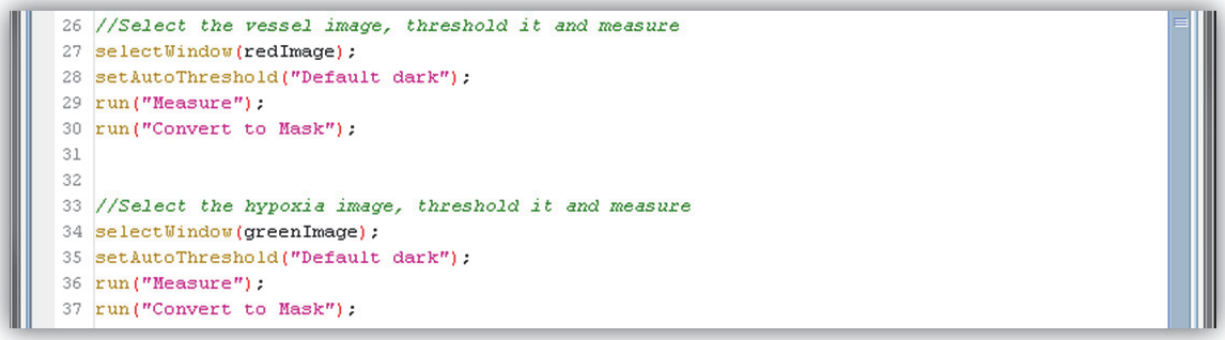
With most analysis it is good practice to output an overview image of what was analysed. This way any aberrant results can be easily investigated.

1. Firstly save the current macro again with the name **Vessel and Hypoxia Measurment (with Result Output).ijm**. This will prevent you accidently overwriting the previous macro.
2. Now add the following steps to the start of the macro



This will store the directory path of the initial image into the **dir** variable for use later. Also note that File.directory does not follow the same syntac as th other commands we have been using

3. Add the **run("Convert to Mask");** command after the measurements of the vessel and hypoxia images to generate binary masks of what was thresholded



4. Add the following merge channels command to merge the vessel and hypoxia binary masks into an RGB image

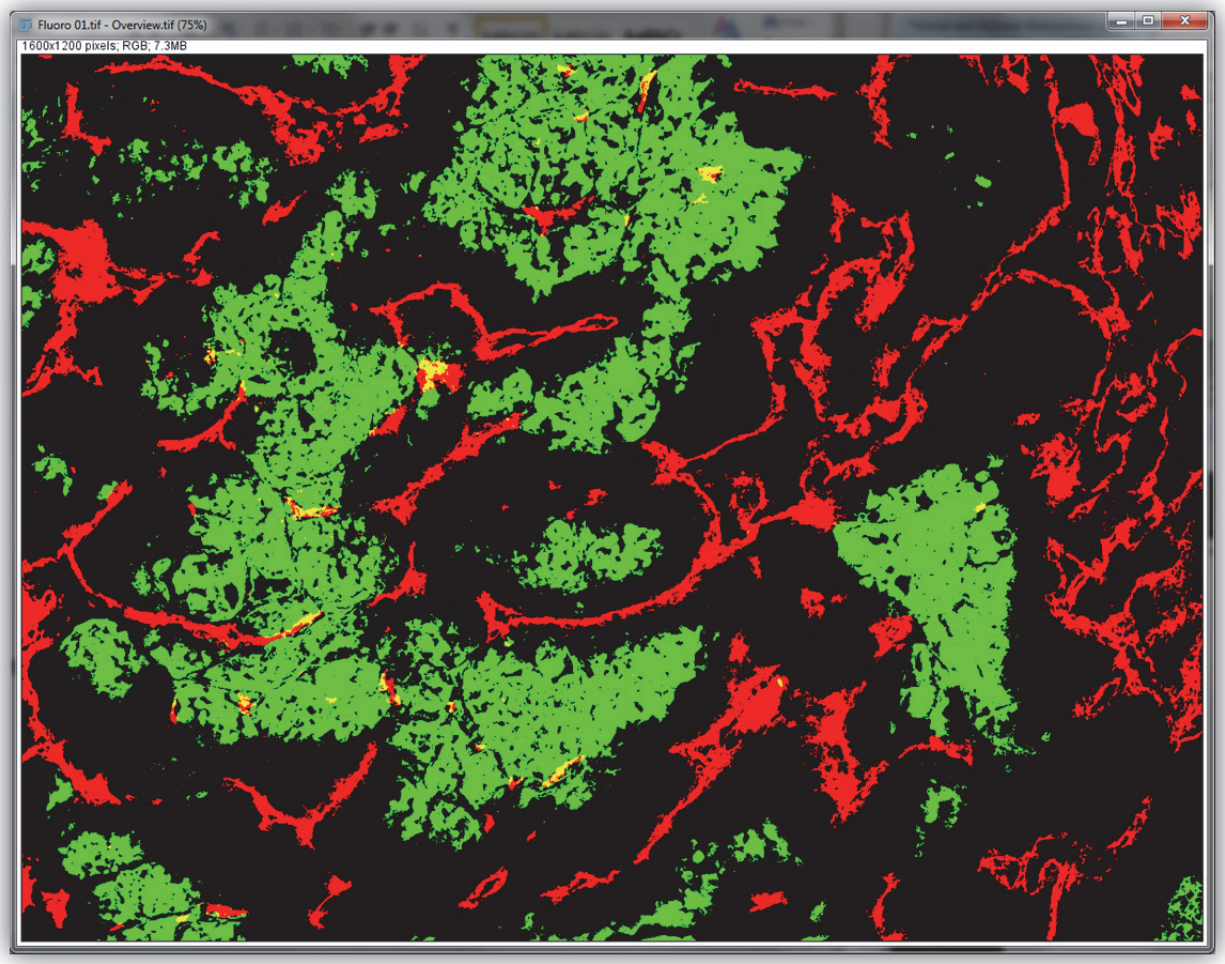
run("Merge Channels...", "c1=["+redImage+"] c2=["+greenImage+"]");

```
39 //merge Binary masks for result image
40 run("Merge Channels...", "c1=["+redImage+"] c2=["+greenImage+"]");
41
```

5. Finally add the following commands to rename the image and save it to the directory the original image was opened from

```
42 //save merged image into the folder of the original image
43 //append name with " - Overview" on the end
44 selectWindow("RGB");
45 rename(nameStore+" - Overview");
46 saveAs("Tiff", dir+nameStore+" - Overview.tif");
```

6. If you run the macro now it should output an image like the following into the original image directory



Vessel and Hypoxia Measurement (with Result Output).ijm – Final Code

```
//This macro will separate the channels of a merged
//image (Red, Green and Blue). It will then modify the name of the
//channels and measure the percentage area and intensity of
//the stain

//Store the initial image name into the nameStore variable
nameStore = getTitle();
dir = File.directory;

//Split the channels into red, green and blue
run("Split Channels");

//Rename red image to orignal name + vessels and green image to
//original name + hypoxia
selectWindow(nameStore+" (red)");
rename(nameStore+" - Vessels");
redImage = getTitle();

selectWindow(nameStore+" (green)");
rename(nameStore+" - hypoxia");
greenImage = getTitle();

//configure measurments for threshold area and average intenisty
run("Set Measurements...", "area mean area_fraction limit display redirect=None decimal=3");

//Select the vessel image, threshold it and measure
selectWindow(redImage);
setAutoThreshold("Yen dark");
run("Measure");
run("Convert to Mask");

//select the hypoxia image, threshold it and measure
selectWindow(greenImage);
setAutoThreshold("Default dark");
run("Measure");
run("Convert to Mask");

//merge binary masks for result image
run("Merge Channels...", "c1=["+redImage+"] c2=["+greenImage+"]");

//save merged image into the folder of the original image
//append name with " - Overview" on the end
selectWindow("RGB");
rename(nameStore+" - Overview");
saveAs("Tiff", dir+nameStore+" - Overview.tif");
```

```
//close all open images
run("Close All");
```

Batch Processing

Aim

Batch processing is where the true power of image analysis can shine. Processing a large directory full of images without any user intervention can generate large amounts of data in very little

For this tech note we will modify the previously made cell counting journal to be able to batch process a folder of images. We will then modify this to process only image files in the directory.

Secondly we will modify the fluorescent threshold journal to batch process and save the output images into a directory of the users choosing

Batch Processing Concepts

To be able to batch process something in Fiji several steps need to be taken

- Be able to select the directory where the images are
- Figure out how many images are in the directory
- May need to create additional directories for output
- Loop a given section of code for each image in the directory

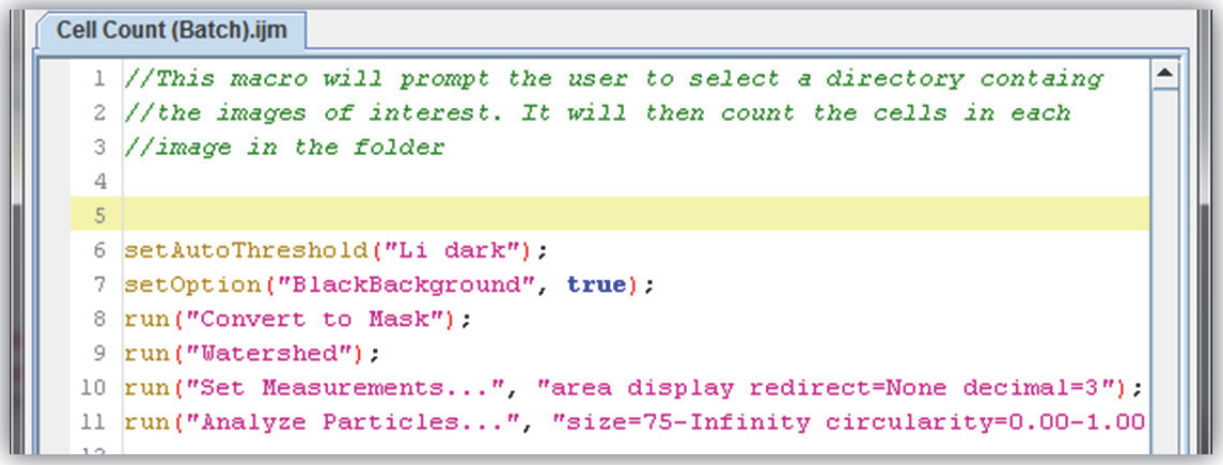
Batch Process - Cell Counting

1. Open the **Cell Count.ijm** macro you saved earlier.

Cell Count.ijm

```
1 setAutoThreshold("Default dark");
2 //run("Threshold...");
3 setAutoThreshold("Li dark");
4 setOption("BlackBackground", true);
5 run("Convert to Mask");
6 run("Watershed");
7 run("Set Measurements...", "area display redirect=None decimal=3");
8 run("Analyze Particles...", "size=75-Infinity circularity=0.00-1.00
```

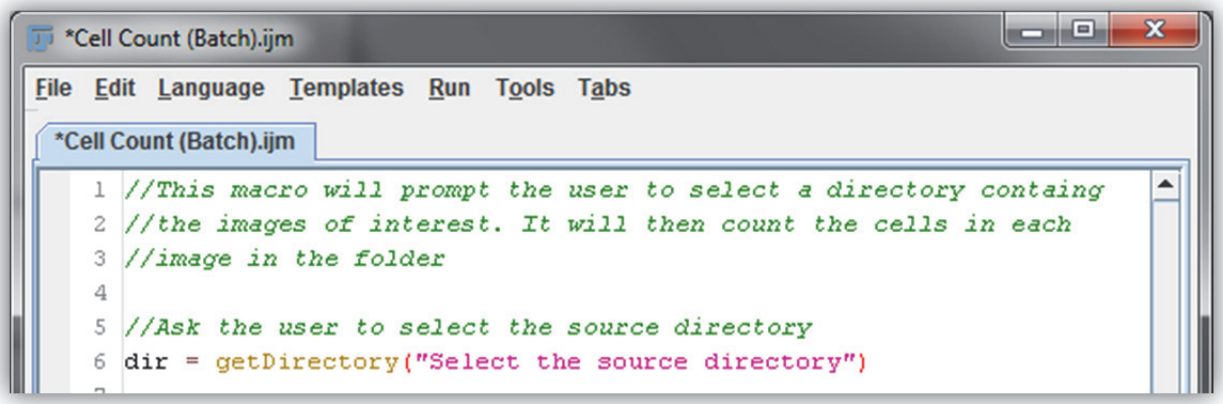
2. Firstly let's clean up the code a bit and add some comments for what the macro will do. Then save the macro as **Cell Count (Batch).ijm**



3. The next step is to add the commands to prompt the user to select the source directory. Do this by adding the following command

dir = getDirectory("Select the source directory")

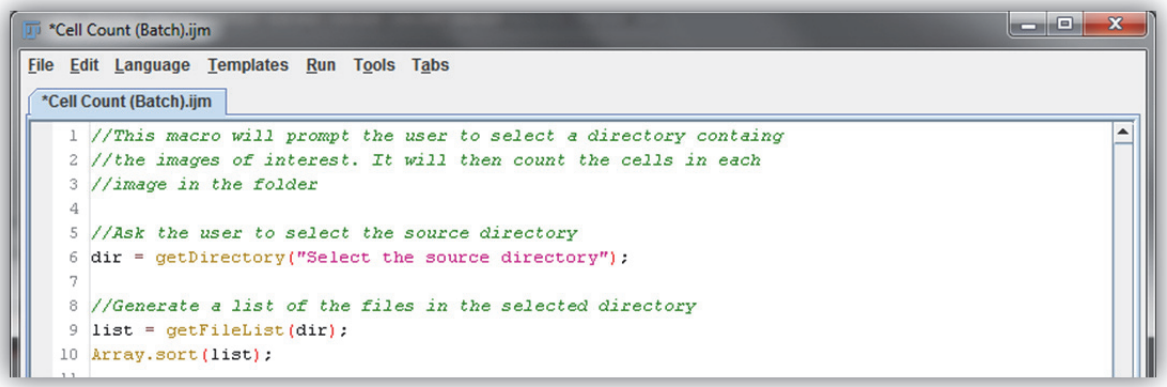
dir is a custom variable to store the selected directory in
getDirectory creates a select folder dialog (like a browse window) with the string in brackets on top of the window



4. We now need to create a list of the files in the directory so we know what, and how many, to process. This is done by using the commands

   list = getFileList(dir);
   Array.sort(list);

   The first command populates an array list of file names from the selected directory and stores them in the **list** variable. The second command makes sure the list is sorted alphabetically (certain versions of imageJ and operating systems don't always build the list alphabetically).



5. We now have a source of files and they are all in a list. We now need to tell Fiji to apply the previous code for counting cells to all images in the selected directory.

   Add the following command

   for(i=0; i<list.length; i++) {

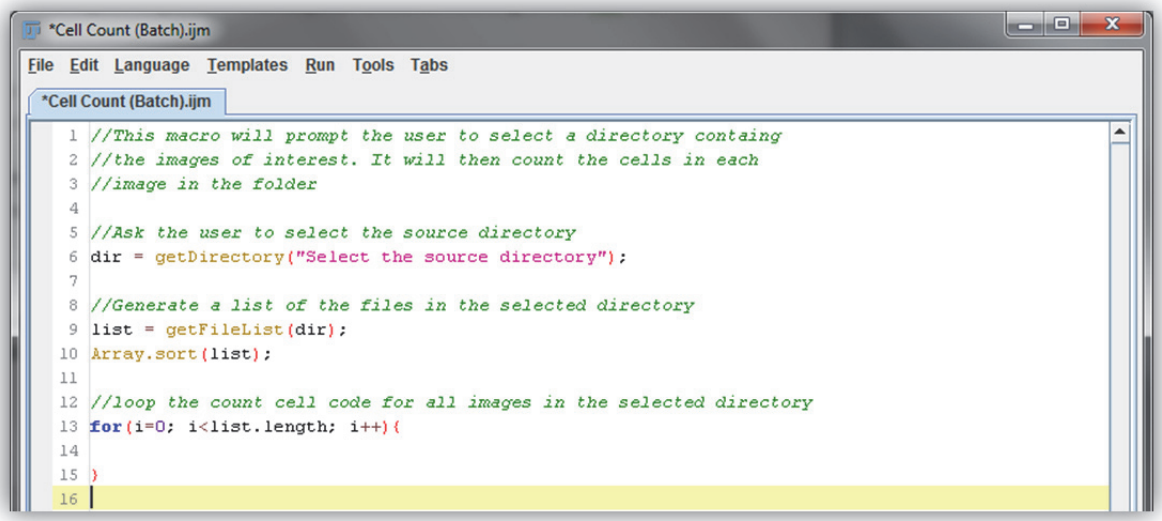
   This is called a for loop. It is a logical argument that says for while something is true run the code that follow.

   The **i** is a variable that is initially assigned to **0**. The letter i is most commonly used for for loops as it can stand for iteration, any other letter or variable name be used instead.

   The **i<list.length** tells the loop to run while the value of the i variable is lower than the length of the file list was generated.

   The **i++** tells the loop to advance the value of the **i** variable by 1 at the end of each loop.

   The **{** also called a squiggly bracket indicates the start of the for loop. The end of the loop is closed off with a **}**. So all code that will be ran by the for loop is between the **{}**.

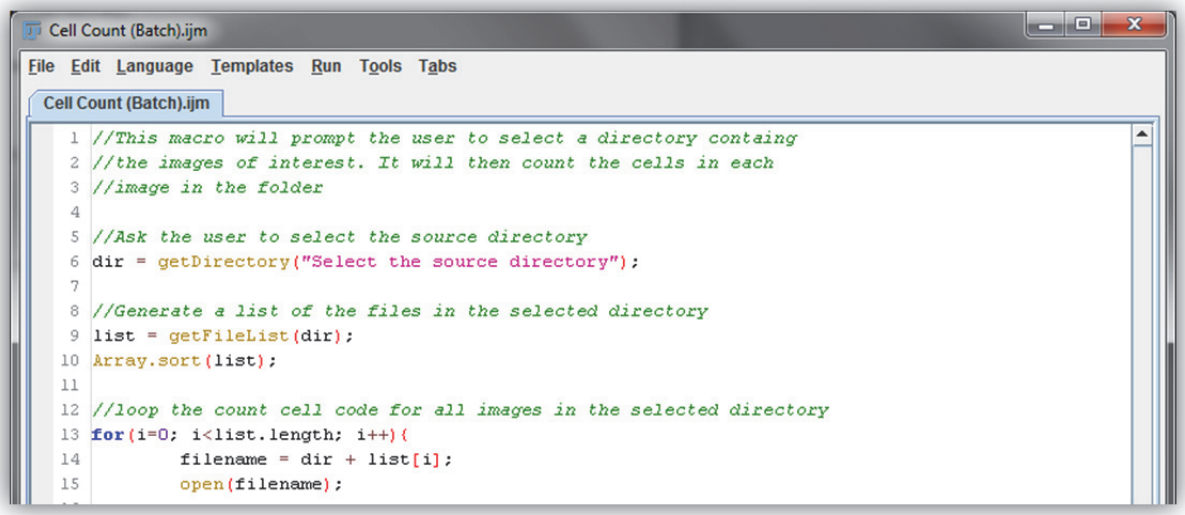


6. The next step is to tell the for loop to open a file from the selected directory. This is done with these two lines of code

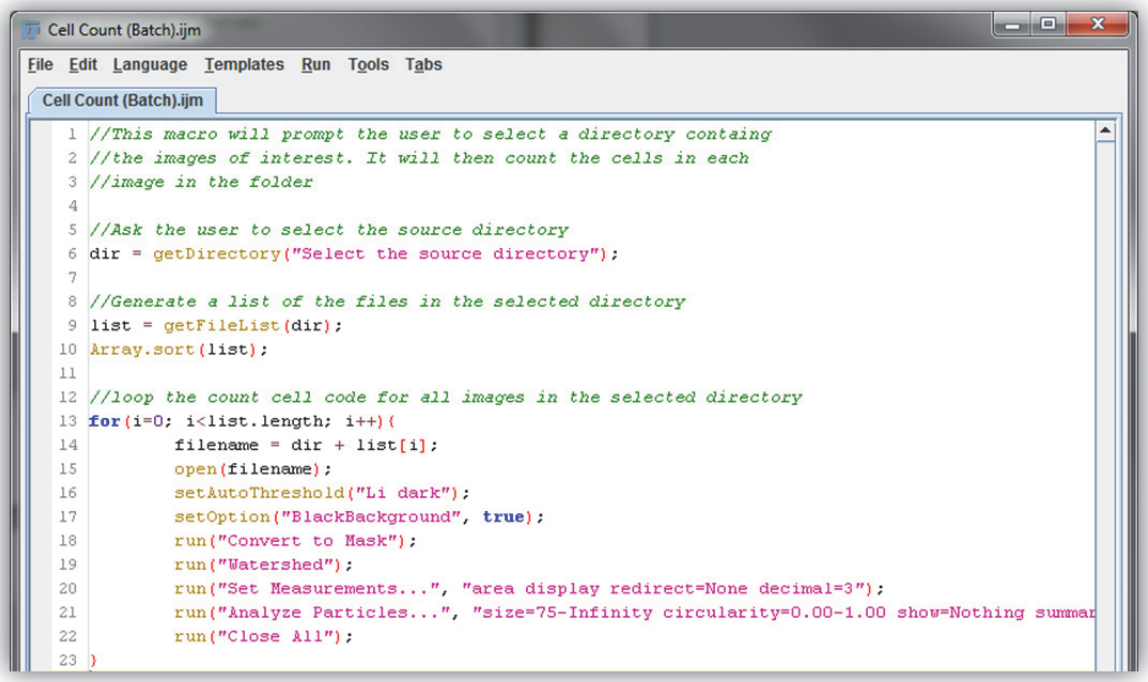
   filename = dir + list[i];
   open(filename);

   The first step populates the variable **filename** with the directory and the filename that is at position i on the list

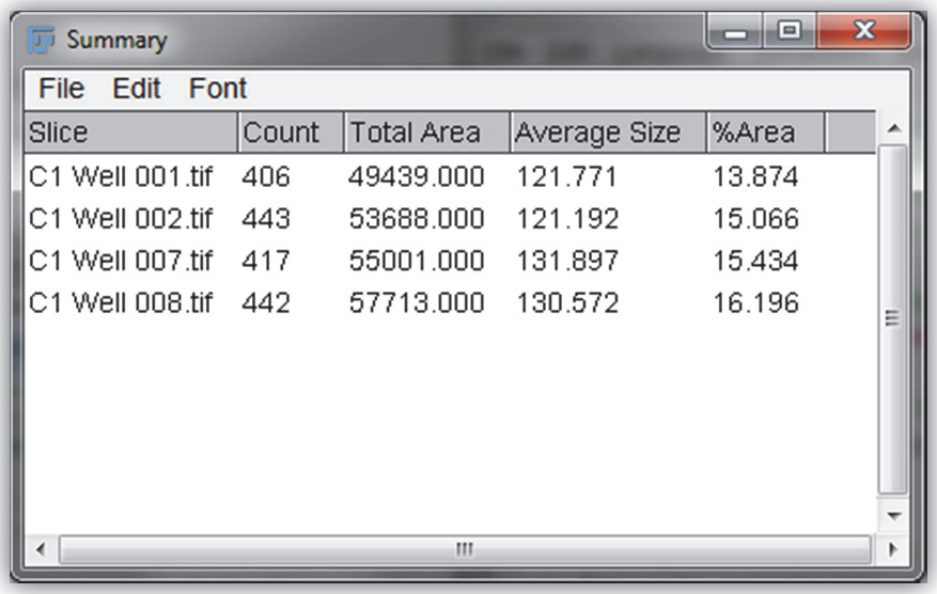
   The next step opens the file referenced in the filename variable



7. Finally add the code from the count cells macro between the **{}** after the last commands entered and add a **run("Close All");** command at the end



8. Save the macro as **Cell Count (Batch).ijm** and run it. Select the **Demo Images\Widefield Images\Batch Processing**



Cell Count (Batch).ijm – Final Code

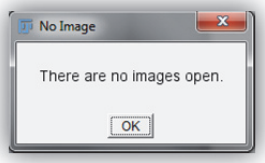
```
//This macro will prompt the user to select a directory containg
//the images of interest. It will then count the cells in each
//image in the folder

//Ask the user to select the source directory
dir = getDirectory("Select the source directory");

//Generate a list of the files in the selected directory
list = getFileList(dir);
Array.sort(list);

//loop the count cell code for all images in the selected directory
for(i=0; i<list.length; i++){
	filename = dir + list[i];
	open(filename);
	setAutoThreshold("Li dark");
	setOption("BlackBackground", true);
	run("Convert to Mask");
	run("Watershed");
	run("Set Measurements...", "area display redirect=None decimal=3");
	run("Analyze Particles...", "size=75-Infinity circularity=0.00-1.00 show=Nothing summarize");
	run("Close All");
}
```

Batch Process - Cell Counting – Contaminated Directory

The previous macro will run fine on a directory that only contains images but if there are other files are in the folder the macro will fail. This can be avoided by telling the macro to only open image files and ignore any others.

1. First open the **Cell Count (Batch).ijm** and try running on the **Demo Images\Widefield Images\Batch Processing (Contaminated)**

   You should get an error as the first file in the directory is a text file, not an image file



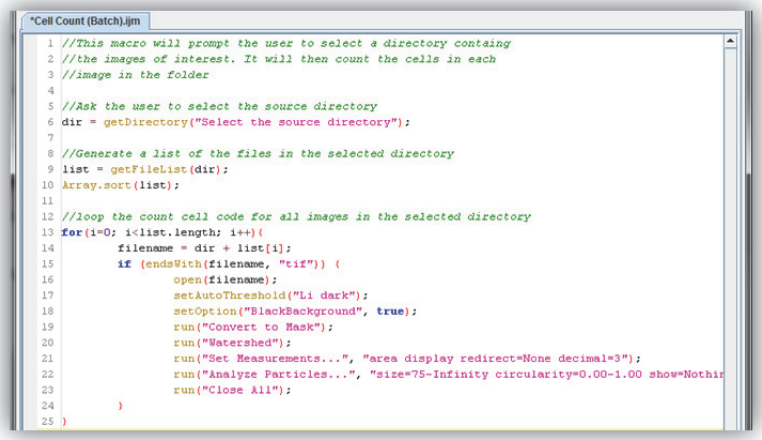
2. To get around this we need to modify the open command to check if the file is an image file before running (in this case a Tiff file)

   This is done with an **if** command as follows.

   if (endsWith(filename, "tif")) {

   This command looks at the end of the text in the filename variable and checks if it ends with tif, indicating it is a tif file. **NOTE:** the **{** this indicates the start of the if loop and needs to be closed with a **}** at the end.

   Add the commands as below



3. Save the macro as **Cell Count (Batch – Tiff Only).ijm** and try running it again on the contaminated directory.

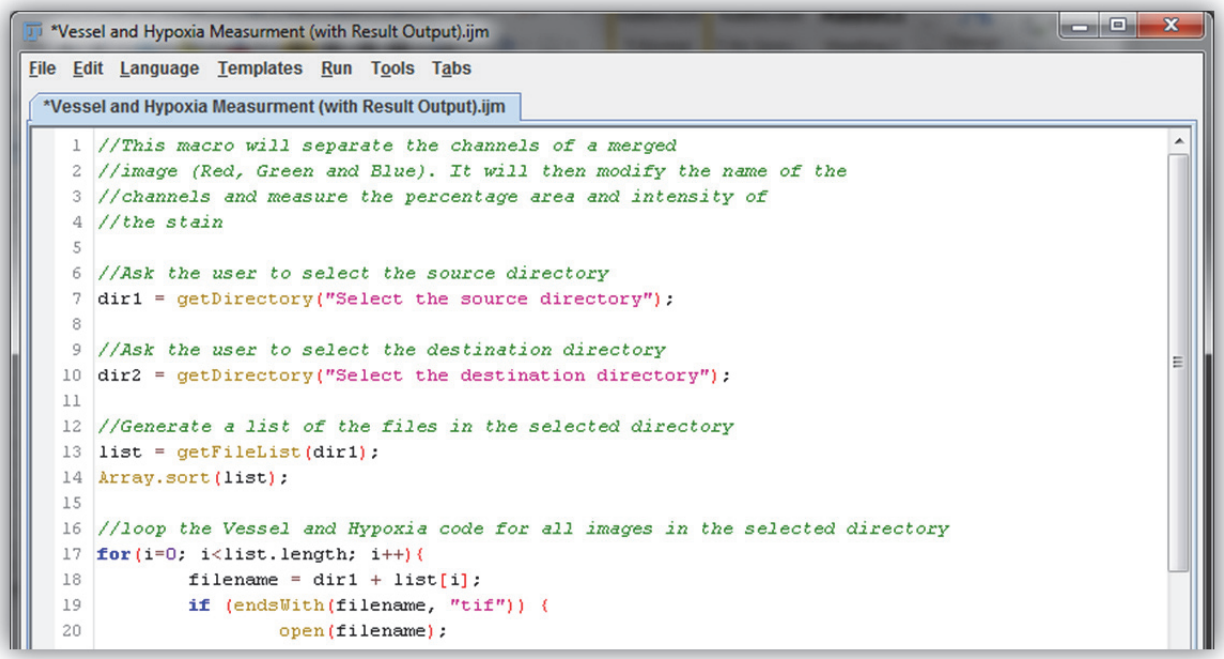
Cell Count (Batch – Tiff Only).ijm – Final Code

```
//This macro will prompt the user to select a directory containg
//the images of interest. It will then count the cells in each
//image in the folder

//Ask the user to select the source directory
dir = getDirectory("Select the source directory");

//Generate a list of the files in the selected directory
list = getFileList(dir);
Array.sort(list);

//loop the count cell code for all images in the selected directory
for(i=0; i<list.length; i++){
	filename = dir + list[i];
	if (endsWith(filename, "tif")) {
		open(filename);
		setAutoThreshold("Li dark");
		setOption("BlackBackground", true);
		run("Convert to Mask");
		run("Watershed");
		run("Set Measurements...", "area display redirect=None decimal=3");
		run("Analyze Particles...", "size=75-Infinity circularity=0.00-1.00 show=Nothing
summarize");
		run("Close All");
	}
}
```

Batch Processing and Outputting Data

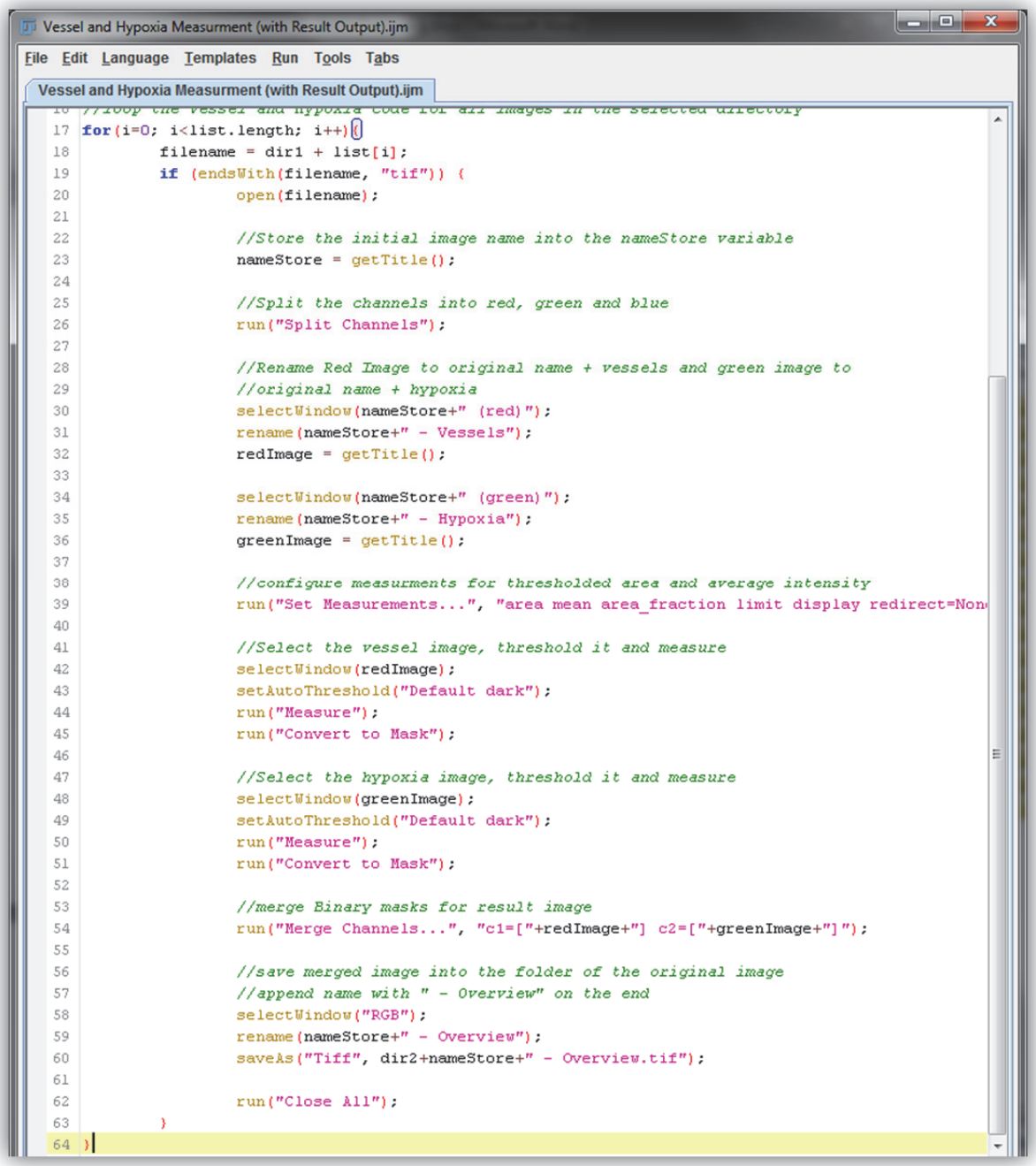
Previously we made a macro to separate images and save an output image of the thresholded areas measured. We will now modify it to batch process on the folder and prompt the user on where to save the output images instead of saving them into the source folder.

1. Open the **Vessel and Hypoxia Measurement (with Result Output).ijm** and resave it as **Vessel and Hypoxia Measurement (Batch).ijm**
2. Reconfigure the start of the macro as shown below. It's easier to copy the code from the previous Cell Count macro and then modify it.



Notice that the original dir variable has been changed to dir1 and an additional dir2 variable has been added to store the user selected destination directory. The filename variable has been changed to include dir1 as well. Otherwise the code is the same the Cell Count (Batch) macro

3. Move the rest of the code between the **{}** and change the “save as” step (line 60 in the below example) to reflect the user selected destination directory (dir2).



4. Save the macro and run it on the **Demo Images\Widefield Images\Fluorescent Measurement** folder

Vessel and Hypoxia Measurement (with Result Output).ijm – Final Code

```
//This macro will separate the channels of a merged
//image (Red, Green and Blue). It will then modify the name of the
//channels and measure the percentage area and intensity of
//the stain

//Ask the user to select the source directory
dir1 = getDirectory("Select the source directory");

//Ask the user to select the destination directory
dir2 = getDirectory("Select the destination directory");

//Generate a list of the files in the selected directory
list = getFileList(dir1);
Array.sort(list);

//loop the Vessel and Hypoxia code for all images in the selected directory
for(i=0; i<list.length; i++){
	filename = dir1 + list[i];
	if (endsWith(filename, "tif")) {
		open(filename);

		//Store the initial image name into the nameStore variable
		nameStore = getTitle();

		//Split the channels into red, green and blue
		run("Split Channels");

		//Rename Red Image to original name + vessels and green image to
		//original name + hypoxia
		selectWindow(nameStore+" (red)");
		rename(nameStore+" - Vessels");
		redImage = getTitle();

		selectWindow(nameStore+" (green)");
		rename(nameStore+" - Hypoxia");
		greenImage = getTitle();

		//configure measurments for thresholded area and average intensity
		run("Set Measurements...", "area mean area_fraction limit display redirect=None
decimal=3");

		//Select the vessel image, threshold it and measure
		selectWindow(redImage);
		setAutoThreshold("Default dark");
		run("Measure");
		run("Convert to Mask");
```

```
            //Select the hypoxia image, threshold it and measure
            selectWindow(greenImage);
            setAutoThreshold("Default dark");
            run("Measure");
            run("Convert to Mask");

            //merge Binary masks for result image
            run("Merge Channels...", "c1=["+redImage+"] c2=["+greenImage+"]");

            //save merged image into the folder of the original image
            //append name with " - Overview" on the end
            selectWindow("RGB");
            rename(nameStore+" - Overview");
            saveAs("Tiff", dir2+nameStore+" - Overview.tif");

            run("Close All");
        }
}
```

Custom Data Logging

Aim

As analysis becomes more complex you may generate results that are not part of the normal results and summary tables. For example a ratio or percentage based on two results may be calculated in the macro and needs to be logged out. This can be achieved by creating a custom log table for the data to go into which can be saved later.

For this tech note we will write a macro to count the number of positive cells in two stain channels, calculate their percentage of the total population and then log the results to a custom made table.

After that we will refine to code to be sure it is only counting true cells.

We will finally add code to save out overviews of the results into a directory within the original source directory. This is instead of asking the user to provide the output directory as in the previous macro.

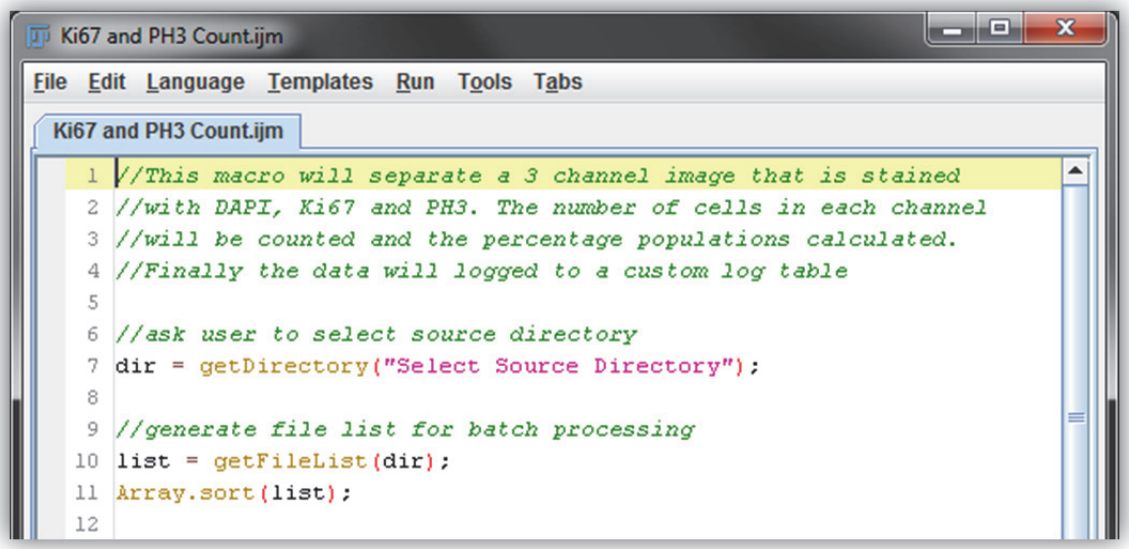
Creating a Custom Log Table

Creating a custom log table requires several steps

- The table needs a title (both for ease of use and to be able to reference it in the code)
- A size, this is just the default size of the table it can be resized manually at the end
- Titles for each of the columns you want to log data to.

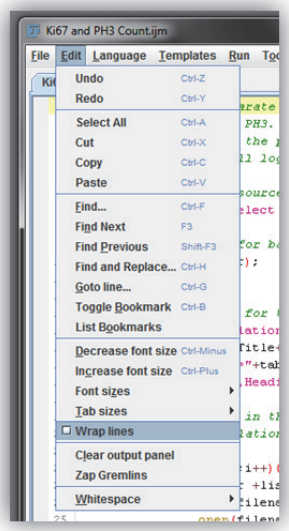
First Steps – Batch Code

As this macro will batch process a folder full of images before adding the code to create a custom log we first need to configure it to be able to batch process. Add the code below (copied from previous batch script) and save it as **Ki67 and PH3 Count.ijm**



Custom Log Creation

The next few lines of code will be quite long (as they are just a contiguous text string), to make it easier to view it all in the editor you can configure it to wrap the text to the widow by going to **Edit → Wrap Lines** in the macro editor



1. Firstly we need to configure the title for the table with the following code

   tableTitle="Cell Population";
   tableTitle2="["+tableTitle+"]";

   The reason for the two variables is due to the table title (“Cell Population” in this example) having a space in it. If a title (or other string) contains a space it must be contained in square brackets (**[]**). The second variable, tableTitle2, adds the brackets to the variable.

2. Next we need to open a table and give it the name we decided.

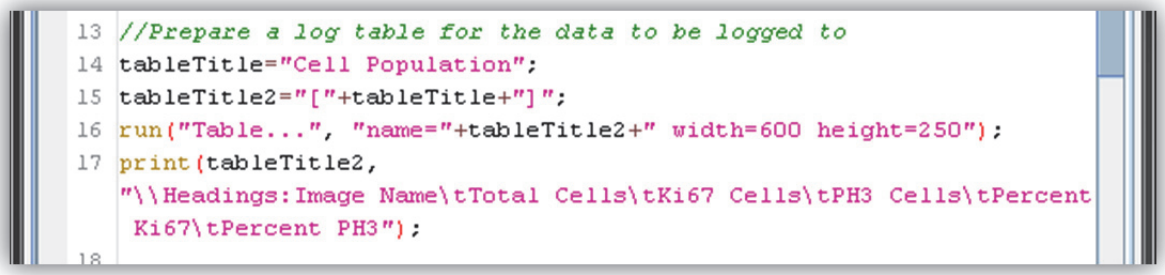
   run("Table...", "name="+tableTitle2+" width=600 height=250");

   The above code opens a table that is 600x250 pixels and has the name stored in the tableTitle2 variable

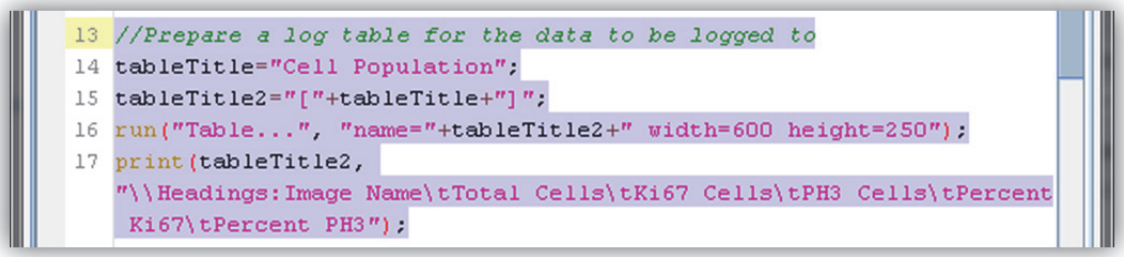
3. Finally to create the column headers add the following code

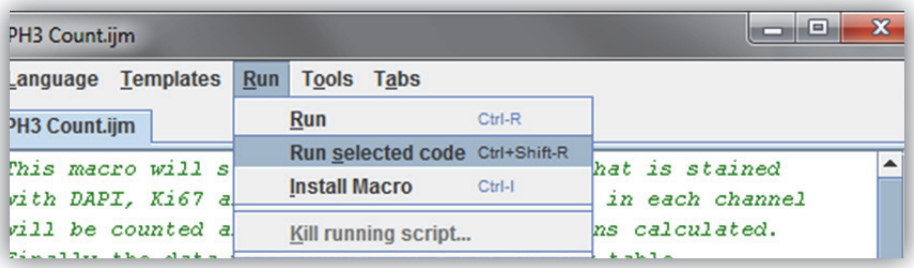
   print(tableTitle2, "\\Headings:Image Name\tTotal Cells\tKi67 Cells\tPH3 Cells\tPercent Ki67\tPercent PH3");

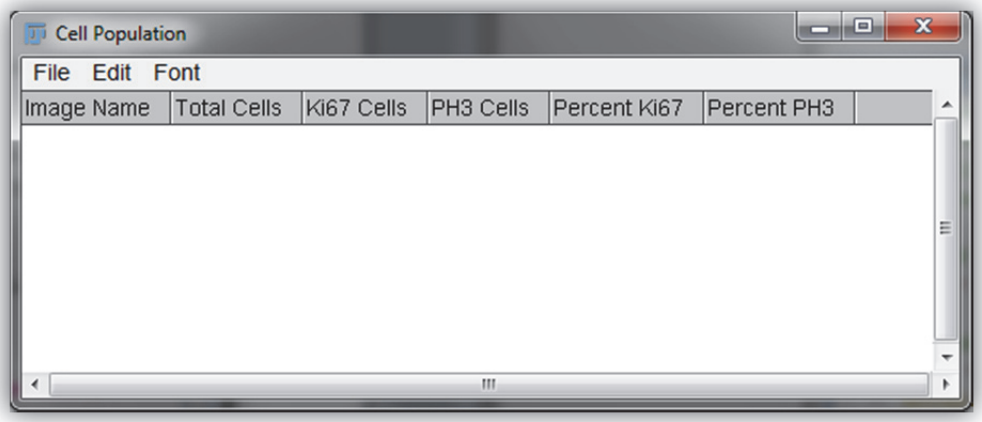
   The first part tells the code to print to a table that matches the tableTitle2 variable. Heading tell it to make the printed text column headers. The \t is recognised as a tab command to skip to the next column.



4. To test the code for the table creation you can highlight just the steps for it (lines 13-17) above and then go to **Run → Run selected code**





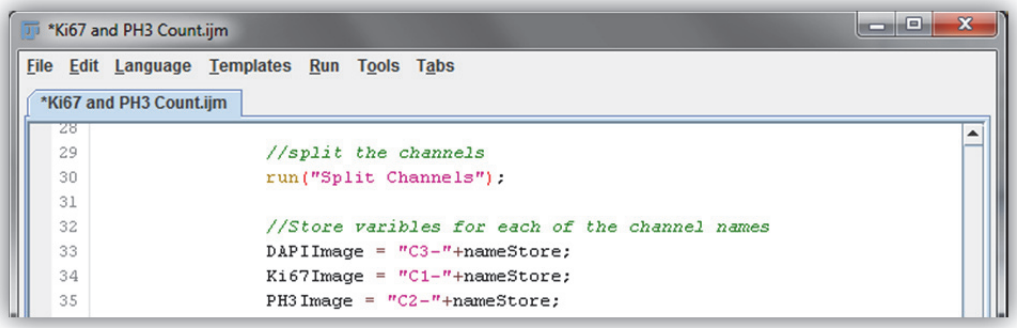


The Analysis Code

1. Finalise the batch code (once again copied from the earlier batch macro)

```
19 //loop for all images in the selected directory to count and
20 //calculate cell populations
21
22 for(i=0;i<list.length;i++){
23         filename = dir +list[i];
24         if (endsWith(filename, "tif")) {
25                 open(filename);
26                 //store the image name in the nameStore variable
27                 nameStore = getTitle();
```

2. The images that will be used for this analysis are 3 channel composite 16 bit images so firstly they need to be separated into individual channels and have their names saved into variables

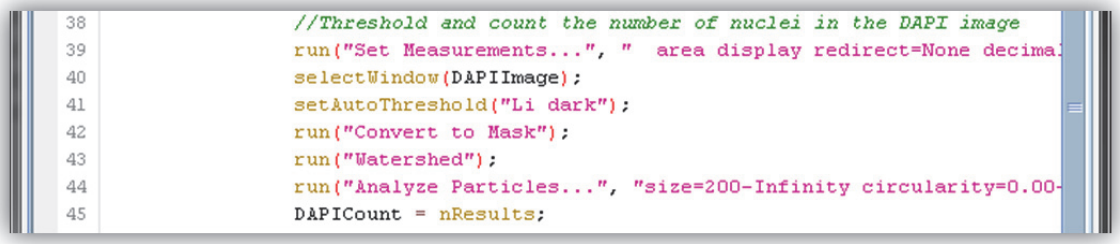


You can test this part of the code too by opening **Control.tif** from the **Demo Images\Widefield Images\Cell Scoring and Cycle** then highlight the code and running the selection. Make sure you highlight up to the definition of the **nameStore** variable.

3. To count the total number of cells (stained with DAPI) use the following code

```
run("Set Measurements...", "  area display redirect=None decimal=3");
selectWindow(DAPIImage);
setAutoThreshold("Li dark");
run("Convert to Mask");
run("Watershed");
run("Analyze Particles...", "size=200-Infinity circularity=0.00-1.00
show=Nothing display clear");
DAPICount = nResults;
```

This code works the same as the previous cell counting code but it uses the Li auto threshold algorithm. The last step counts the number of results in the results table (representing the number of cells) and stores the value in the **DAPICount** variable



4. Repeat the above for the Ki67 and PH3 Images

```
45                  //Threshold and count the number of Ki67 positive cells
46                  selectWindow(Ki67Image);
47                  setAutoThreshold("Li dark");
48                  run("Convert to Mask");
49                  run("Watershed");
50                  run("Analyze Particles...",
"size=200-Infinity circularity=0.00-1.00 show=Nothing display clear");
51                  Ki67Count = nResults;
52
53                  //Threshold and count the number of PH3 positive cells
54                  selectWindow(PH3Image);
55                  setAutoThreshold("Li dark");
56                  run("Convert to Mask");
57                  run("Watershed");
58                  run("Analyze Particles...",
"size=200-Infinity circularity=0.00-1.00 show=Nothing display clear");
59                  PH3Count = nResults;
```

5. Now that the different cell populations are counted we can calculate the percentage of each with the following code

Ki67Percent = (Ki67Count/DAPICount)*100;
PH3Percent = (PH3Count/DAPICount)*100;

```
60
61                  //calculate percentage populations
62                  Ki67Percent = (Ki67Count/DAPICount)*100;
63                  PH3Percent = (PH3Count/DAPICount)*100;
64
```

6. Now that everything is measured and calculated it can be logged out to the custom log

print(tableTitle2, nameStore + "\t" + DAPICount + "\t" + Ki67Count + "\t" + PH3Count + "\t" + Ki67Percent + "\t" + PH3Percent);

Logging out data is the same as making the headers earlier. The print command is directed to the table (tableTitle2) and the variables that are to be logged a done so in order with tab commands between them. Make sure the order you log out variables matches the original headers

```
65                  //log data out to the popultion count table
66                  print(tableTitle2, nameStore + "\t" + DAPICount + "\t" +
Ki67Count + "\t" + PH3Count + "\t" + Ki67Percent + "\t" + PH3Percent);
```

7. Finish off the code with a close all command, a close results command and two **}** to end the for and if loops.

```
68                  //close all open images
69                  run("Close All");
70                  close("Results");
71          }
72  }
```

8. Save the macro and run it on the **Demo Images\Widefield Images\Cell Scoring and Cycle** folder

Ki67 and PH3 Count.ijm – Final Code

```
//This macro will separate a 3 channel image that is stained
//with DAPI, Ki67 and PH3. The number of cells in each channel
//will be counted and the percentage populations calculated.
//Finally the data will logged to a custom log table

//ask user to select source directory
dir = getDirectory("Select Source Directory");

//generate file list for batch processing
list = getFileList(dir);
Array.sort(list);

//Prepare a log table for the data to be logged to
tableTitle="Cell Population";
tableTitle2="["+tableTitle+"]";
run("Table...", "name="+tableTitle2+" width=600 height=250");
print(tableTitle2, "\\Headings:Image Name\tTotal Cells\tKi67 Cells\tPH3 Cells\tPercent
Ki67\tPercent PH3");

//loop for all images in the selected directory to count and
//calculate cell populations

for(i=0;i<list.length;i++){
	filename = dir +list[i];
	if (endsWith(filename, "tif")) {
		open(filename);
		//store the image name in the nameStore variable
		nameStore = getTitle();

		//split the channels
		run("Split Channels");

		//Store varibles for each of the channel names
		DAPIImage = "C3-"+nameStore;
		Ki67Image = "C1-"+nameStore;
		PH3Image = "C2-"+nameStore;


		//Threshold and count the number of nuclei in the DAPI image
		run("Set Measurements...", "  area display redirect=None decimal=3");
		selectWindow(DAPIImage);
		setAutoThreshold("Li dark");
		run("Convert to Mask");
		run("Watershed");
		run("Analyze Particles...", "size=200-Infinity circularity=0.00-1.00 show=Nothing
display clear");
		DAPICount = nResults;
```

```
                //Threshold and count the number of Ki67 positive cells
                selectWindow(Ki67Image);
                setAutoThreshold("Li dark");
                run("Convert to Mask");
                run("Watershed");
                run("Analyze Particles...", "size=200-Infinity circularity=0.00-1.00 show=Nothing
display clear");
                Ki67Count = nResults;

                //Threshold and count the number of PH3 positive cells
                selectWindow(PH3Image);
                setAutoThreshold("Li dark");
                run("Convert to Mask");
                run("Watershed");
                run("Analyze Particles...", "size=200-Infinity circularity=0.00-1.00 show=Nothing
display clear");
                PH3Count = nResults;

                //calculate percentage populations
                Ki67Percent = (Ki67Count/DAPICount)*100;
                PH3Percent = (PH3Count/DAPICount)*100;

                //log data out to the popultion count table
                print(tableTitle2, nameStore + "\t" + DAPICount + "\t" + Ki67Count + "\t" +
PH3Count + "\t" + Ki67Percent + "\t" + PH3Percent);

                //close all open images
                run("Close All");
                close("Results");
        }
}
```

Adding Refinements – Positive Cell Selection

When doing cell counts based off thresholds there is the potential to count objects as cells that are not. Part of this is avoided by adding size exclusion filters to the particle counting. An added refinement is to do a reconstruction of the binary masks based off a mask you trust.

In the above example the DAPI stain is clean and can be trusted. It can be used to select only the binary object from the other channels that overlap with the nuclei masks

This step will require an additional plugin to be installed. Ths is the **Morphology** plugins that can be obtained from the **Morphology Update Site**. Instructions on adding update sites to Fiji can be found on page 7 of this manual. To add the reconstruction steps modify the previous part of the code that counted the Ki67 and PH3 images as follows

```
47                  //Threshold and count the number of Ki67 positive cells
48                  selectWindow(Ki67Image);
49                  setAutoThreshold("Li dark");
50                  run("Convert to Mask");
51                  run("Watershed");
52                  run("BinaryReconstruct ", "mask="+DAPIImage+" seed="+Ki67Image+
    " create white");
53                  selectWindow("Reconstructed");
54                  rename("Ki67 Binary");
55                  run("Analyze Particles...",
    "size=200-Infinity circularity=0.00-1.00 show=Nothing display clear");
56                  Ki67Count = nResults;
57
58                  //Threshold and count the number of PH3 positive cells
59                  selectWindow(PH3Image);
60                  setAutoThreshold("Li dark");
61                  run("Convert to Mask");
62                  run("Watershed");
63                  run("BinaryReconstruct ", "mask="+DAPIImage+" seed="+PH3Image+
    " create white");
64                  selectWindow("Reconstructed");
65                  rename("PH3 Binary");
66                  run("Analyze Particles...",
    "size=200-Infinity circularity=0.00-1.00 show=Nothing display clear");
67                  PH3Count = nResults;
```

A binary reconstruction takes the image you want to analyse and lays a seed image underneath is. Only objects in the original image that intersect the seed image will be kept, all others are discarded

1. Save the macro as **Ki67 and PH3 Count (Refined).ijm** and run it. In this case the results will be higher in the refined code; this is due to some low signal cells being excluded in the non-refined code.

Ki67 and PH3 Count (Refined).ijm – Final Code

```
//This macro will separate a 3 channel image that is stained
//with DAPI, Ki67 and PH3. The number of cells in each channel
//will be counted and the percentage populations calculated.
//Finally the data will logged to a custom log table

//ask user to select source directory
dir = getDirectory("Select Source Directory");

//generate file list for batch processing
list = getFileList(dir);
Array.sort(list);

//Prepare a log table for the data to be logged to
tableTitle="Cell Population";
tableTitle2="["+tableTitle+"]";
run("Table...", "name="+tableTitle2+" width=600 height=250");
print(tableTitle2, "\\Headings:Image Name\tTotal Cells\tKi67 Cells\tPH3 Cells\tPercent
Ki67\tPercent PH3");

//loop for all images in the selected directory to count and
//calculate cell populations

for(i=0;i<list.length;i++){
	filename = dir +list[i];
	if (endsWith(filename, "tif")) {
		open(filename);
		//store the image name in the nameStore variable
		nameStore = getTitle();

		//split the channels
		run("Split Channels");

		//Store varibles for each of the channel names
		DAPIImage = "C3-"+nameStore;
		Ki67Image = "C1-"+nameStore;
		PH3Image = "C2-"+nameStore;


		//Threshold and count the number of nuclei in the DAPI image
		run("Set Measurements...", "  area display redirect=None decimal=3");
		selectWindow(DAPIImage);
		setAutoThreshold("Li dark");
		run("Convert to Mask");
		run("Watershed");
		run("Analyze Particles...", "size=200-Infinity circularity=0.00-1.00 show=Nothing
display clear");
		DAPICount = nResults;
```

```
                //Threshold and count the number of Ki67 positive cells
                selectWindow(Ki67Image);
                setAutoThreshold("Li dark");
                run("Convert to Mask");
                run("Watershed");
                run("BinaryReconstruct ", "mask="+DAPIImage+" seed="+Ki67Image+" create
white");
                selectWindow("Reconstructed");
                rename("Ki67 Binary");
                run("Analyze Particles...", "size=200-Infinity circularity=0.00-1.00 show=Nothing
display clear");
                Ki67Count = nResults;

                //Threshold and count the number of PH3 positive cells
                selectWindow(PH3Image);
                setAutoThreshold("Li dark");
                run("Convert to Mask");
                run("Watershed");
                run("BinaryReconstruct ", "mask="+DAPIImage+" seed="+PH3Image+" create
white");
                selectWindow("Reconstructed");
                rename("PH3 Binary");
                run("Analyze Particles...", "size=200-Infinity circularity=0.00-1.00 show=Nothing
display clear");
                PH3Count = nResults;

                //calculate percentage populations
                Ki67Percent = (Ki67Count/DAPICount)*100;
                PH3Percent = (PH3Count/DAPICount)*100;

                //log data out to the popultion count table
                print(tableTitle2, nameStore + "\t" + DAPICount + "\t" + Ki67Count + "\t" +
PH3Count + "\t" + Ki67Percent + "\t" + PH3Percent);

                //close all open images
                run("Close All");
                close("Results");
        }
}
```

Outputting Results Images to a Macro Created Folder

In previous examples we have either saved output images into the source directory or had the user select where they would like them saved. There is a third option and that is for the macro to create a sub-folder within the source folder and save them there.

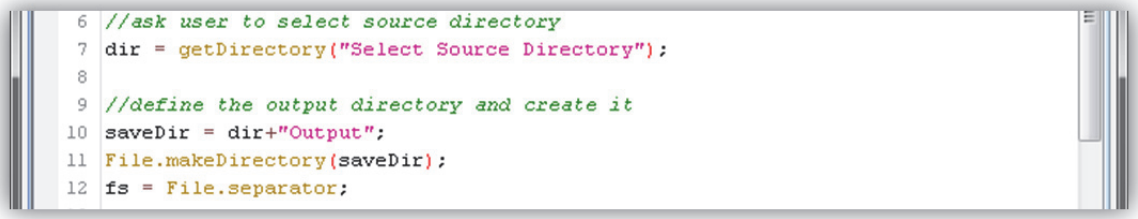
1. To create an output directory add the following two commands after the getDirectory user prompt

   saveDir = dir+"Output";
   File.makeDirectory(saveDir);
   fs = File.seperator;

   The saveDir variable is created by concatenating the user selected directory (dir) with the string "Output". This creates a text string that will be the path to the user selected source folder with a file separator at the end (e.g. c:\Images\Folder\), so when "Output" is added to the end it adds an extra directory to the string (e.g. c:\Images\Folder\Output).

   Once the saveDir variable is populated we can use it to create a directory using the File.makeDirectory command. It will also be used later on when we save the output images.

   The fs variable is populated with the file separator (/ or \) that discriminates directory level relevant to the platform the code is being run on. If you don't use this during the save step the code will be incompatible on PC, Mac and Linux platforms



2. Now we just need to add the code to merge the images and save them to the output folder. Add the following code before the close all command at the end

   run("Merge Channels...", "c1=[Ki67 Binary] c2=[PH3 Binary] c3="+DAPIImage+" ignore");
   selectWindow("RGB");
   rename(nameStore+" - Overview");
   saveAs("Tiff", saveDir+fs+nameStore+" - Overview.tif");

   The merge channels command is the same as used earlier. This time the saveAs command tells the macro to save the image in the saveDir directory that was created at the start.

```
81                  //create merged image and save to the output directory
82                  run("Merge Channels...",
"c1=[Ki67 Binary] c2=[PH3 Binary] c3="+DAPIImage+" ignore");
83                  selectWindow("RGB");
84                  rename(nameStore+" - Overview");
85                  saveAs("Tiff", saveDir+fs+nameStore+" - Overview.tif");
86
87                  //close all open images
88                  run("Close All");
89                  close("Results");
90          }
91  }
```

3. Save the macro as **Ki67 and PH3 Count (Refined with Output).ijm** and try it out

Ki67 and PH3 Count (Refined with Output).ijm – Final Code

```
//This macro will separate a 3 channel image that is stained
//with DAPI, Ki67 and PH3. The number of cells in each channel
//will be counted and the percentage populations calculated.
//Finally the data will logged to a custom log table

//ask user to select source directory
dir = getDirectory("Select Source Directory");

//define the output directory and create it
saveDir = dir+"Output";
File.makeDirectory(saveDir);
fs = File.separator;

//generate file list for batch processing
list = getFileList(dir);
Array.sort(list);

//Prepare a log table for the data to be logged to
tableTitle="Cell Population";
tableTitle2="["+tableTitle+"]";
run("Table...", "name="+tableTitle2+" width=600 height=250");
print(tableTitle2, "\\Headings:Image Name\tTotal Cells\tKi67 Cells\tPH3 Cells\tPercent
Ki67\tPercent PH3");

//loop for all images in the selected directory to count and
//calculate cell populations

for(i=0;i<list.length;i++){
	filename = dir +list[i];
	if (endsWith(filename, "tif")) {
		open(filename);
		//store the image name in the nameStore variable
		nameStore = getTitle();

		//split the channels
		run("Split Channels");

		//Store varibles for each of the channel names
		DAPIImage = "C3-"+nameStore;
		Ki67Image = "C1-"+nameStore;
		PH3Image = "C2-"+nameStore;


		//Threshold and count the number of nuclei in the DAPI image
		run("Set Measurements...", "  area display redirect=None decimal=3");
		selectWindow(DAPIImage);
		setAutoThreshold("Li dark");
```

```
		run("Convert to Mask");
		run("Watershed");
		run("Analyze Particles...", "size=200-Infinity circularity=0.00-1.00 show=Nothing
display clear");
		DAPICount = nResults;

		//Threshold and count the number of Ki67 positive cells
		selectWindow(Ki67Image);
		setAutoThreshold("Li dark");
		run("Convert to Mask");
		run("Watershed");
		run("BinaryReconstruct ", "mask="+DAPIImage+" seed="+Ki67Image+" create
white");
		selectWindow("Reconstructed");
		rename("Ki67 Binary");
		run("Analyze Particles...", "size=200-Infinity circularity=0.00-1.00 show=Nothing
display clear");
		Ki67Count = nResults;

		//Threshold and count the number of PH3 positive cells
		selectWindow(PH3Image);
		setAutoThreshold("Li dark");
		run("Convert to Mask");
		run("Watershed");
		run("BinaryReconstruct ", "mask="+DAPIImage+" seed="+PH3Image+" create
white");
		selectWindow("Reconstructed");
		rename("PH3 Binary");
		run("Analyze Particles...", "size=200-Infinity circularity=0.00-1.00 show=Nothing
display clear");
		PH3Count = nResults;

		//calculate percentage populations
		Ki67Percent = (Ki67Count/DAPICount)*100;
		PH3Percent = (PH3Count/DAPICount)*100;

		//log data out to the popultion count table
		print(tableTitle2, nameStore + "\t" + DAPICount + "\t" + Ki67Count + "\t" +
PH3Count + "\t" + Ki67Percent + "\t" + PH3Percent);

		//create merged image and save to the output directory
		run("Merge Channels...", "c1=[Ki67 Binary] c2=[PH3 Binary] c3="+DAPIImage+"
ignore");
		selectWindow("RGB");
		rename(nameStore+" - Overview");
		saveAs("Tiff", saveDir+fs+nameStore+" - Overview.tif");

		//close all open images
```

```
                run("Close All");
                close("Results");
        }
}
```

Looping for ROIs

Aim

There are sittuations where you need to measure information in a large group of regions individually and log out the measurments. In this example we will write code to batch analyse the shapes and textures of many cells. To be able to do this we need to create regions around all the cells, measure each of those regions individually and then log out the information to a custom table to be used for further image analysis.

This script will also use the LOCI tools to import confocal data captured on a Zeiss confocal.

Selecting Files

As with other examples this code will be ran on a folder of images, though these will be Zeiss LSM files and not Tiff images there is no difference in the initial code.

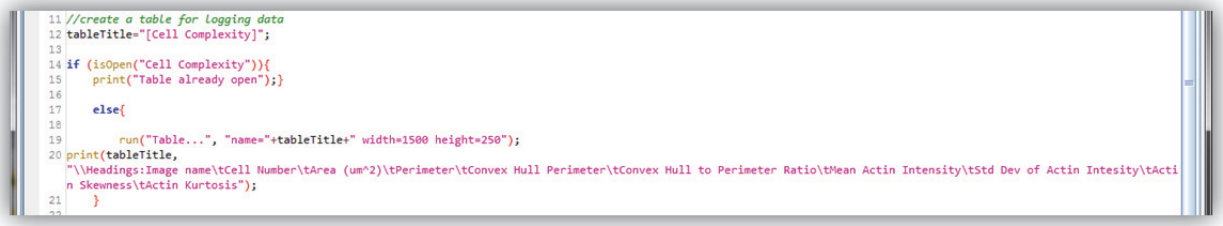
```
dir = getDirectory("Choose Source Directory ");
list = getFileList(dir);
Array.sort(list);
```

```
1  //Macro to segment cells to analyse individual cells perimeter to convex hull ratio
2  //Regions generated for individual cells will also be used to measure the textrue
3  //and variation of the actin staining in channel two by measuring mean intenisty,
4  //standard deviation of intenisty, skewness and kurtosis
5
6  //define source directory and generate a file list
7  dir = getDirectory("Choose Source Directory ");
8  list = getFileList(dir);
9  Array.sort(list);
10
```

Generating a Custom Table

The data needs to be logged into a custom table to make subsequent analysis easier.

```
tableTitle="[Cell Complexity]";

if (isOpen("Cell Complexity")){
        print("Table already open");}

        else{

                run("Table...", "name="+tableTitle+" width=1500 height=250");
print(tableTitle, "\\Headings:Image name\tCell Number\tArea
(um^2)\tPerimeter\tConvex Hull Perimeter\tConvex Hull to Perimeter
Ratio\tMean Actin Intensity\tStd Dev of Actin Intesity\tActin Skewness\tActin
Kurtosis");
        }
```



Defining Measurements

Before we loop any code we need to determine what we need to measure. For this code we need to measure the area, mean intensity, standard deviation of intensity, perimeter, skewness and kurtois of the images.

```
run("Set Measurements...", "area mean standard perimeter skewness kurtosis
redirect=None decimal=3");
```

Batch Processing

With all the initial parameters set we can now proceed to batch process the data as we normally would. Everything is placed within a **for** loop

```
for (i=0; i<list.length; i++){
```

```
26
27 //loop analysis code for all file in the list
28 for (i=0; i<list.length; i++){
29
```

Working with ROIs

When ROIs are used in the code we need to make sure there are none on the image before running any analysis. This is important on any code that loops for multiple files to make sure the next image doesn't inherit the ROIs from the previous image. This is easily done with a simple check to see if there any ROIs in the ROI Manager list and deleting them if there are.

```
roiCount=roiManager("count");
        if(roiCount>0){
        roiManager("Deselect");
        roiManager("Delete");
        }
```

```
29
30    //check to see if there are currently any ROIs in the manager. If there are delete them
31    roiCount=roiManager("count");
32    if(roiCount>0){
33    roiManager("Deselect");
34    roiManager("Delete");
35    }
36
```

Opening Images with the Bio-Formats/LOCI Tools

The Bio-Formats plugin has a range of macro functions that can be seen by going to **Plugins → Bio-Formats → Bio-Formats Macro Extensions** this will give a complete list of all the functions available in the macro language. These include things like the number of files in a series like a LIF or ND2 file, names of series etc.

For this example we won't be calling any of the extensions but will be feeding the file through the Bio-Formats plugin to open it.

We need to build a file name as usual by using the created file list and then we use that file name in the Bio-Formats importer instead of the usual Fiji Open command.

```
fileName = dir + list[i];

run("Bio-Formats Importer", "open=["+fileName+"] autoscale
color_mode=Composite view=Hyperstack stack_order=XYCZT");
```

```
37  //build file name path from the file list
38  fileName = dir + list[i];
39
40  //use the Bio-FOrmats importer to open the file
41  run("Bio-Formats Importer", "open=["+fileName+"] autoscale color_mode=Composite view=Hyperstack stack_order=XYCZT");
42
```

Getting Images Ready for Processing

We need to store the name of the original image to be able to find it later, split it all into its component channels and close ones we don't need

```
nameStore=getTitle();

run("Split Channels");

selectWindow("C1-"+nameStore);
close();
```

```
43  //store the name of the opened image
44  nameStore=getTitle();
45
46  //split channels into separate images
47  run("Split Channels");
48
49  //close channel 1 as it is not needed
50  selectWindow("C1-"+nameStore);
51  close();
```

Segmenting the Cells

For segmenting these cells we have to do things a little different to previously. The actin channel shows no nuclear staining to use to as an anchor for segmentation. We can create this though by multiplying the nuclear channel image by the actin image. This gives us a cell object with a strong central signal to use he find maxima command on.

Preparing the Images

1. To be able to multiply the two images together we first need to make sure there is enough intensity range to multiply into without saturating anything. The images in this example were captured in 8 bit so we first need to give them a 16 bit intensity range

```
selectWindow("C2-"+nameStore);
run("16-bit");
selectWindow("C3-"+nameStore);
run("16-bit");
```

```
53  //upscale chanels 2 and 3 from 8 bit to 16 bit to allow the following multiplication step not saturate the images
54  selectWindow("C2-"+nameStore);
55  run("16-bit");
56  selectWindow("C3-"+nameStore);
57  run("16-bit");
```

2. Now we can multiply the two images together using the image calculator

```
imageCalculator("Multiply create", "C2-"+nameStore,"C3-"+nameStore);
```

```
59  //multiply channels 2 and 3 together to create an image suitible for segmentation.
60  imageCalculator("Multiply create", "C2-"+nameStore,"C3-"+nameStore);
```

Segmenting the Cells

With an image that now contains a central nuclear signal and cell boundaries we can use the find maxima command to create a segmented watershed map

1. First apply a slight blur to remove noise and then use the find maxima command to create segmented particles

```
run("Gaussian Blur...", "sigma=2");

run("Find Maxima...", "noise=3000 output=[Segmented Particles]");
```

```
62  //apply a Gaussian blur to remove noise
63  run("Gaussian Blur...", "sigma=2");
64
65  //create segmented particles of the cell objects
66  run("Find Maxima...", "noise=3000 output=[Segmented Particles]");
```

2. Before creating a binary of the actin stain to finalise the segmentation we need to make a copy of the actin channel for later measurements so we are not measuring a filtered or binary image.

```
selectWindow("C2-"+nameStore);
run("Duplicate...", "title=Actin");
```

```
68  //duplicate channel 2 for later analysis
69  selectWindow("C2-"+nameStore);
70  run("Duplicate...", "title=Actin");
```

3. Now we can create a binary image of the actin stain to be used to create single cell objects.

```
selectWindow("C2-"+nameStore);
run("Gaussian Blur...", "sigma=2");
setAutoThreshold("Li dark");
setOption("BlackBackground", true);
run("Convert to Mask");
```

```
72  //blur, threshold and binarise the Actin channel for segmentation
73  selectWindow("C2-"+nameStore);
74  run("Gaussian Blur...", "sigma=2");
75  setAutoThreshold("Li dark");
76  setOption("BlackBackground", true);
77  run("Convert to Mask");
78
```

4. Single cell masks can now be generated using the image calculator and AND command.

```
imageCalculator("AND create", "Result of C2-"+nameStore+" Segmented","C2-"+nameStore);
```

```
79  //create raw cell mask of the cells
80  imageCalculator("AND create", "Result of C2-"+nameStore+" Segmented","C2-"+nameStore);
81
```

5. Finally measure the cell masks with the analyse particles command to create ROIs in the manager and to clean up any erroneous chunks of cell.

```
selectWindow("Result of Result of C2-"+nameStore+" Segmented");
run("Analyze Particles...", "size=250.00-Infinity show=Masks display exclude add");
selectWindow("Mask of Result of Result of C2-"+nameStore+" Segmented");
run("Invert LUT");
rename("Cell Mask");
```

```
//clean up raw cell mask by removing small objects that are not cells adn create ROIs into the ROI manager
selectWindow("Result of Result of C2-"+nameStore+" Segmented");
run("Analyze Particles...", "size=250.00-Infinity show=Masks display exclude add");
selectWindow("Mask of Result of Result of C2-"+nameStore+" Segmented");
run("Invert LUT");
rename("Cell Mask");
```

Cleaning Up Open Images

Now that all the required images have been created it is good practice to close those that are no longer needed.

```
selectWindow("Result of Result of C2-"+nameStore+" Segmented");
close();
selectWindow("C2-"+nameStore);
close();
selectWindow("Result of C2-"+nameStore+" Segmented");
close();
selectWindow("Result of C2-"+nameStore);
close();
selectWindow("C3-"+nameStore);
close();
```

```
//Close any unwanted windows so only the cell mask and orignial actin stains are left
selectWindow("Result of Result of C2-"+nameStore+" Segmented");
close();
selectWindow("C2-"+nameStore);
close();
selectWindow("Result of C2-"+nameStore+" Segmented");
close();
selectWindow("Result of C2-"+nameStore);
close();
selectWindow("C3-"+nameStore);
close();
```

Looping for All ROIs

Before looping for all the ROIs we need to put them where they need to be (the raw Actin image) and count how many there are to know how many time to loop the analysis code.

1. Copy the ROIs in the manager to the raw Actin image by selecting it and using the ROI manager show all command.

```
selectWindow("Actin");
roiManager("Show All");
```

```
//select the original actin channel and activate the ROIs on it
selectWindow("Actin");
roiManager("Show All");
```

2. Get a count of the number of ROIs in the manager using the count command

   numROIs=roiManager("count");

```
105    //count the number of ROIs to know how many to loop for
106    numROIs=roiManager("count");
107
```

3. With all the information collected we can now loop the measurements for all ROIs in the list. This is done with a for loop that is incremented based on the ROIs in the manager list

   for(r=0;r<numROIs;r++){

   The first step of the loop is to select an ROI to process. The ROIs in the manager list are indexed from 0 (i.e. region 0 is the first in the list). As r will start at 0 and count up by one on each loop its value can be used to select the next region on the list in each loop

   roiManager("Select", r);

```
108    //loop the following code for the number of ROIs
109    for(r=0;r<numROIs;r++){
110
111        //select the ROI designated by the current value of r. ROIs start form 0
112        roiManager("Select", r);
```

Measuring the Selected ROI

With an ROI selected and active we can now measure and store the various values required. These steps will get the area, perimeter and intensity values of the cell based on the cell outline.

cellArea=getResult("Area");
cellPeri=getResult("Perim.");
meanInt=getResult("Mean");
stdInt=getResult("StdDev");
skew=getResult("Skew");
kurt=getResult("Kurt");

```
114        //measure the ROI to get the area, perimeter and various intenisty values
115        run("Measure");
116        cellArea=getResult("Area");
117        cellPeri=getResult("Perim.");
118        meanInt=getResult("Mean");
119        stdInt=getResult("StdDev");
120        skew=getResult("Skew");
121        kurt=getResult("Kurt");
```

With those parameters measured the only one left to measure is the perimeter of the convex hull of the cell.

```
run("Convex Hull");
run("Measure");
cellConvex=getResult("Perim.");
```

```
125  //genrate the convex hull of the ROI and measure its perimenter
126  run("Convex Hull");
127  run("Measure");
128  cellConvex=getResult("Perim.");
129
```

Final Calculation and Logging Data

With all parameters measured the last steps are to calculate the perimeter to convex hull ratio and log the data

1. Calculate the ratio using the following code

```
ratio=cellPeri/cellConvex;
```

```
128  //calculate the cell perimeter to convex hull ratio
129  ratio=cellPeri/cellConvex;
```

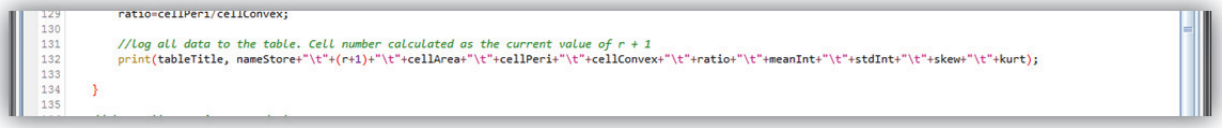
2. Log the collected data for the cell to the table

```
print(tableTitle,
nameStore+"\t"+(r+1)+"\t"+cellArea+"\t"+cellPeri+"\t"+cellConvex+"\t"+ratio+"\t
"+meanInt+"\t"+stdInt+"\t"+skew+"\t"+kurt);
```

```
131  //Log all data to the table. Cell number calculated as the current value of r + 1
132  print(tableTitle, nameStore+"\t"+(r+1)+"\t"+cellArea+"\t"+cellPeri+"\t"+cellConvex+"\t"+ratio+"\t"+meanInt+"\t"+stdInt+"\t"+skew+"\t"+kurt);
```

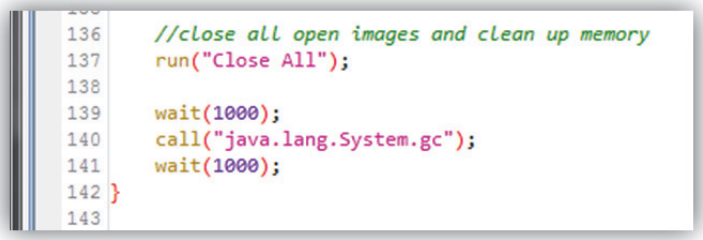
Finalising the Code

With everything in place we can close of the ROI measurement loop with a } to let it keep looping for all regions.



Outside this loop we can close all open images using the close all command and clear up any wasted memory ready for the next loop on the next image

```
        run("Close All");

        wait(1000);
        call("java.lang.System.gc");
        wait(1000);
}
```



Saving the Data Table

At the end of everything the table can be saved automatically into the originally selected source directory

```
selectWindow("Cell Complexity");
saveAs("Text", dir+"Cell Complexity.xls");
```

Run the Code

Save the script as “Cell Complexity Measurement with Actin Texture.ijm” and run it on the folder **Demo Images\Confocal\Cell Complexity\For Batch**

Running it Faster – Batch Mode

When you run the code it will open all the images and flash a lot on screen. To save a lot of time and avoid the risk of accidentally interrupting the code as it is running you can put the code into batch mode. This means no images will be displayed and all processing will occur off screen, greatly speeding up the analysis.

Add the following command just before the initiation of the loop for all the files in the directory

setbatchMode(true);

```
//define required measurments for analysis
run("Set Measurements...", "area mean standard perimeter skewness kurtosis redirect=None decimal=3");

//enter batch mode to speed up processing
setBatchMode(true);

//loop analysis code for all file in the list
for (i=0; i<list.length; i++){

    //check to see if there are currently any ROIs in the manager. If there are delete them
    roiCount=roiManager("count");
    if(roiCount>0){
    roiManager("Deselect");
    roiManager("Delete");
    }
```

Run the code again. It should run much faster with just the data table filling up with numbers and no images open.

Cell Complexity Measurement with Actin Texture.ijm – Final Code

```
//Macro to segment cells to analyse individual cells perimeter to convex hull ratio
//Regions generated for individual cells will also be used to measure the textrue
//and variation of the actin staining in channel two by measuring mean intenisty,
//standard deviation of intenisty, skewness and kurtosis

//define source directory and generate a file list
dir = getDirectory("Choose Source Directory ");
list = getFileList(dir);
Array.sort(list);

//create a table for logging data
tableTitle="[Cell Complexity]";

if (isOpen("Cell Complexity")){
        print("Table already open");}

        else{

                run("Table...", "name="+tableTitle+" width=1500 height=250");
```

```
print(tableTitle, "\\Headings:Image name\tCell Number\tArea (um^2)\tPerimeter\tConvex Hull Perimeter\tConvex Hull to Perimeter Ratio\tMean Actin Intensity\tStd Dev of Actin Intesity\tActin Skewness\tActin Kurtosis");
	}

//define required measurments for analysis
run("Set Measurements...", "area mean standard perimeter skewness kurtosis redirect=None decimal=3");

//enter batch mode to speed up processing
setBatchMode(true);

//loop analysis code for all file in the list
for (i=0; i<list.length; i++){

	//check to see if there are currently any ROIs in the manager. If there are delete them
	roiCount=roiManager("count");
	if(roiCount>0){
	roiManager("Deselect");
	roiManager("Delete");
	}

	//build file name path from the file list
	fileName = dir + list[i];

	//use the Bio-FOrmats importer to open the file
	run("Bio-Formats Importer", "open=["+fileName+"] autoscale color_mode=Composite view=Hyperstack stack_order=XYCZT");

	//store the name of the opened image
	nameStore=getTitle();

	//split channels into separate images
	run("Split Channels");

	//close channel 1 as it is not needed
	selectWindow("C1-"+nameStore);
	close();

	//upscale chanels 2 and 3 from 8 bit to 16 bit to allow the following multiplication step not saturate the images
	selectWindow("C2-"+nameStore);
	run("16-bit");
	selectWindow("C3-"+nameStore);
	run("16-bit");

	//multiply channels 2 and 3 together to create an image suitible for segmentation.
	imageCalculator("Multiply create", "C2-"+nameStore,"C3-"+nameStore);
```

```
	//apply a Gaussian blur to remove noise
	run("Gaussian Blur...", "sigma=2");

	//create segmented particles of the cell objects
	run("Find Maxima...", "noise=3000 output=[Segmented Particles]");

	//duplicate channel 2 for later analysis
	selectWindow("C2-"+nameStore);
	run("Duplicate...", "title=Actin");

	//blur, threshold and binarise the Actin channel for segmentation
	selectWindow("C2-"+nameStore);
	run("Gaussian Blur...", "sigma=2");
	setAutoThreshold("Li dark");
	setOption("BlackBackground", true);
	run("Convert to Mask");

	//create raw cell mask of the cells
	imageCalculator("AND create", "Result of C2-"+nameStore+" Segmented","C2-
"+nameStore);

	//clean up raw cell mask by removing small objects that are not cells adn create ROIs into
the ROI manager
	selectWindow("Result of Result of C2-"+nameStore+" Segmented");
	run("Analyze Particles...", "size=250.00-Infinity show=Masks display exclude add");
	selectWindow("Mask of Result of Result of C2-"+nameStore+" Segmented");
	run("Invert LUT");
	rename("Cell Mask");

	//Close any unwanted windows so only the cell mask and orignial actin stains are left
	selectWindow("Result of Result of C2-"+nameStore+" Segmented");
	close();
	selectWindow("C2-"+nameStore);
	close();
	selectWindow("Result of C2-"+nameStore+" Segmented");
	close();
	selectWindow("Result of C2-"+nameStore);
	close();
	selectWindow("C3-"+nameStore);
	close();

	//select the original actin channel and activate the ROIs on it
	selectWindow("Actin");
	roiManager("Show All");

	//count the number of ROIs to know how many to loop for
	numROIs=roiManager("count");
```

```
	//loop the following code for the number of ROIs
	for(r=0;r<numROIs;r++){

		//select the ROI designated by the current value of r. ROIs start form 0
		roiManager("Select", r);

		//measure the ROI to get the area, perimeter and various intenisty values
		run("Measure");
		cellArea=getResult("Area");
		cellPeri=getResult("Perim.");
		meanInt=getResult("Mean");
		stdInt=getResult("StdDev");
		skew=getResult("Skew");
		kurt=getResult("Kurt");

		//genrate the convex hull of the ROI and measure its perimenter
		run("Convex Hull");
		run("Measure");
		cellConvex=getResult("Perim.");

		//calculate the cell perimeter to convex hull ratio
		ratio=cellPeri/cellConvex;

		//log all data to the table. Cell number calculated as the current value of r + 1
		print(tableTitle,
nameStore+"\t"+(r+1)+"\t"+cellArea+"\t"+cellPeri+"\t"+cellConvex+"\t"+ratio+"\t"+meanInt+"\t"+std
dInt+"\t"+skew+"\t"+kurt);

	}

	//close all open images and clean up memory
	run("Close All");

	wait(1000);
	call("java.lang.System.gc");
	wait(1000);
}

//save output table into the folder the orignal images were in
selectWindow("Cell Complexity");
saveAs("Text", dir+"Cell Complexity.xls");
```

Custom Dialog Boxes

Aim

With some script user input is required at the start to fill in information for missing parameters or those that cannot be easily interpreted automatically. To gather this information a custom dialog window can be created for the user to fill out, the data from this window can then be stored in variables for later use in the code.

Dialog windows can contain boxes for text and numbers, check boxes, sliders, drop down boxes and radio buttons. The values contained in these entries can be forwarded to variables.

These notes make use of the images found in the **Demo Images\Widefield\Montage** folder. The following examples will use custom dialogs to ask the user how an image was separated into tiles to be able to stich it back together again. Another example will allow the user to enter any number of rows and columns and have the script chop the image up into tiles and save them.

Defining File Sources and Save Paths

This first example will ask the user to select the first file of a tiled set of data and then ask for the user to select the type of grid array it is (choices will be 2x2, 3x3 and 4x4).

This example will use the files in the **2x2, 3x3 and 4x4** folders found in **Demo Images\Widefield\Montage** folder.

1. Open up the script editor and enter comments at the top about what the script will do

```
1 //Macro to automatically merge tile scan data into a single image based on information
2 //gathered from the user that is entered in a custom dialog box
```

2. Define a variable for the file separator to allow consistency across operating systems

   Fs=File.separator;

```
//store the OS specific file seperator into the fs variable for later file saving and directory creeation
fs=File.separator;
```

3. Next add a step to ask the user to select the first file of the montage and store its location in the **filePath** variable

   filePath=File.openDialog("Select First File");

```
//as the user to select the first file of the montage and store its path in the filePath variable
filePath=File.openDialog("Select First File");
```

4. Now we can load all the images in the series into a stack using the Load Image Sequence command

   run("Image Sequence...", "open=["+filePath+"] sort");

```
//load the images a sequence into a new stack
run("Image Sequence...", "open=["+filePath+"] sort");
```

5. Once the stack is created we need to get the path it came from and create an output directory to save the merged image into.

   pathStore=getInfo("image.directory");
   savePath=pathStore+fs+"Merged";
   File.makeDirectory(savePath);

   We also need to store the name of the opened stack in the nameStore variable

```
//find the folder the image was opened from and create a sub directory in it called Merged
pathStore=getInfo("image.directory");
savePath=pathStore+fs+"Merged";
File.makeDirectory(savePath);

//store the name of the opened stack into the nameStore variable
nameStore=getTitle();
```

Creating a Custom Dialog Box

We now need to generate the custom dialog window. Custom dialogs always require three steps. A create step where you give it a name and create all the entries for the fields you will need. A show step that generates the dialog and displays it for the user to interact with it. Finally a collection step where the data in the dialog is taken and stored in variables to be used in later functions.

1. Give the dialog a name using the Dialog.create command

   Dialog.create("Tile Merger");

2. Next we need to add some fields for the user to interact with. Firstly we need to be able to enter the name we want the resulting merge saved as. This is done with the following command. The first bit in quotes is the label that will be next to the box to fill in. The second double quote is the default value you want the box populated with, in this case we want it blank hence the "".

   Dialog.addString("Enter Name for Final Merged Image", "");

3. Finally we need an option to select what kind of montage to do. The options we are going to have are 2x2, 3x3 and 4x4. To be able to have these options we first need to add them to an array that we can list in the next step.

   Items=newArray("2x2", 3x3", "4x4");

4. Now we can make them into some entries with the following code. This code makes a list of radio buttons labelled Grid Layout that contains the entries from the previous array in a 3 row 1 column layout with the 2x2 entry selected as default.

   Dialog.addRadioButtonGroup("Grid Layout", items, 3,1, "2x2");

The end result of this should look something like this

```
//create a dialog called Tile Merger
Dialog.create("Tile Merger");

//add an option to the dialog for the user to enter the name of the file to save
Dialog.addString("Enter Name for Final Merged Image","");

//create an array for the radio button names. These are the types of merge that can be performed
items=newArray("2x2","3x3","4x4");

//create a part of the dialog with the list of raio buttons calling it Grid Layout and maxe the buttons in one column
Dialog.addRadioButtonGroup("Grid Layout", items,3,1,"2x2");
```

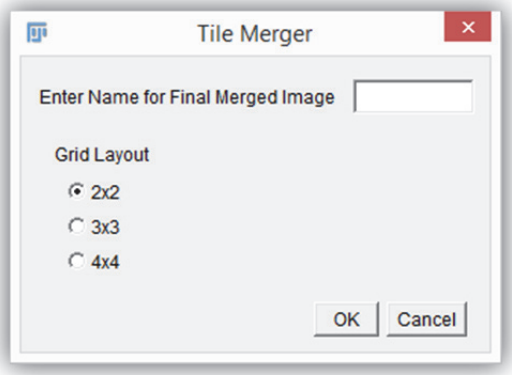
5. Now that we have a dialog built we need it to display. This is done with the following command.

Dialog.show();

```
32
33 //display the dialog box
34 Dialog.show();
35
```

6. You can now test to see if the dialog displays correctly. Select the code from the Dialog.create step through to the dialog.show step and go to **Run → Run Selected Code** or press **Ctrl + Shift + R**

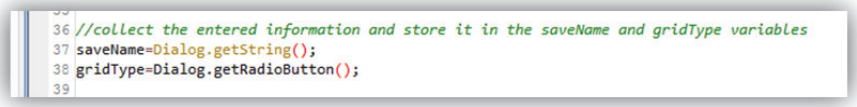
```
20
21 //create a dialog called Tile Merger
22 Dialog.create("Tile Merger");
23
24 //add an option to the dialog for the user to enter the name of the file to save
25 Dialog.addString("Enter Name for Final Merged Image","");
26
27 //create an array for the radio button names. These are the types of merge that can be performed
28 items=newArray("2x2","3x3","4x4");
29
30 //create a part of the dialog with the list of raio buttons calling it Grid Layout and maxe the buttons in one column
31 Dialog.addRadioButtonGroup("Grid Layout", items,3,1,"2x2");
32
33 //display the dialog box
34 Dialog.show();
35
```



Collecting Information from the Dialog

Now the information entered needs to be stored in variables. This is done with the corresponding Dialog.get commands. The variables are collected in the order they were entered into the table, so in this example we will get the save name and then the type of grid layout.

```
saveName=Dialog.getString();
gridType=Dialog.getRadioButton();
```



Running Code Based on Dialog Choices

Now that all the required information has been collected the stitching code can be ran. The montage code for all option is essentially the same with the only difference being the number of columns and rows. These options are hard-coded (can't be changed) so only options will be available for 2x2, 3x3 or 4x4 merging.

1. Firstly a test needs to be performed to see which grid type was selected. The following code is for a 2x2 grid. Notice the double equals sign in the if test, this is required for the code to work

```
if(gridType=="2x2"){
        selectWindow(nameStore);
        run("Make Montage...", "columns=2 rows=2 scale=1");
        selectWindow("Montage");
        saveAs("Tiff", savePath+fs+saveName);
        close();
        selectWindow(nameStore);
        close();

}
```

2. For the 3x3 and 4x4 grids the following code is used

```
        if(gridType=="3x3"){
                selectWindow(nameStore);
                run("Make Montage...", "columns=3 rows=3 scale=1");
                selectWindow("Montage");
                saveAs("Tiff", savePath+fs+saveName);
                close();
                selectWindow(nameStore);
```

```
        close();
}


if(gridType=="4x4"){
        selectWindow(nameStore);
        run("Make Montage...", "columns=4 rows=4 scale=1");
        selectWindow("Montage");
        saveAs("Tiff", savePath+fs+saveName);
        close();
        selectWindow(nameStore);
        close();
}
```

```
//if a 2x2 grid was selected then run the appropriate code to stich it and save the result
if(gridType=="2x2"){
    selectWindow(nameStore);
    run("Make Montage...", "columns=2 rows=2 scale=1");
    selectWindow("Montage");
    saveAs("Tiff", savePath+fs+saveName);
    close();
    selectWindow(nameStore);
    close();

}

//if a 3x3 grid was selected then run the appropriate code to stich it and save the result
if(gridType=="3x3"){
    selectWindow(nameStore);
    run("Make Montage...", "columns=3 rows=3 scale=1");
    selectWindow("Montage");
    saveAs("Tiff", savePath+fs+saveName);
    close();
    selectWindow(nameStore);
    close();

}

//if a 4x4 grid was selected then run the appropriate code to stich it and save the result
if(gridType=="4x4"){
    selectWindow(nameStore);
    run("Make Montage...", "columns=4 rows=4 scale=1");
    selectWindow("Montage");
    saveAs("Tiff", savePath+fs+saveName);
    close();
    selectWindow(nameStore);
    close();

}
```

1. Save the script as “Automatic Montage – Hard-coded.ijm” and test it out

Automatic Montage – Hard-coded.ijm – Final Code

```
//Macro to automatically merge tile scan data into a single image based on information
//gathered from the user that is entered in a custom dialog box

//store the OS specific file seperator into the fs variable for later file saving and directory creation
fs=File.separator;

//as the user to select the first file of the montage and store its path in the filePath variable
filePath=File.openDialog("Select First File");

//load the images a sequence into a new stack
run("Image Sequence...", "open=["+filePath+"] sort");

//find the folder the image was opened from and create a sub directory in it called Merged
pathStore=getInfo("image.directory");
savePath=pathStore+fs+"Merged";
File.makeDirectory(savePath);

//store the name of the opened stack into the nameStore variable
nameStore=getTitle();

//create a dialog called Tile Merger
Dialog.create("Tile Merger");

//add an option to the dialog for the user to enter the name of the file to save
Dialog.addString("Enter Name for Final Merged Image","");

//create an array for the radio button names. These are the types of merge that can be performed
items=newArray("2x2","3x3","4x4");

//create a part of the dialog with the list of raio buttons calling it Grid Layout and maxe the buttons
in one column
Dialog.addRadioButtonGroup("Grid Layout", items,3,1,"2x2");

//display the dialog box
Dialog.show();

//collect the entered information and store it in the saveName and gridType variables
saveName=Dialog.getString();
gridType=Dialog.getRadioButton();

//if a 2x2 grid was selected then run the appropriate code to stich it and save the result
if(gridType=="2x2"){
        selectWindow(nameStore);
        run("Make Montage...", "columns=2 rows=2 scale=1");
        selectWindow("Montage");
        saveAs("Tiff", savePath+fs+saveName);
        close();
```

```
	selectWindow(nameStore);
	close();

}

//if a 3x3 grid was selected then run the appropriate code to stich it and save the result
if(gridType=="3x3"){
	selectWindow(nameStore);
	run("Make Montage...", "columns=3 rows=3 scale=1");
	selectWindow("Montage");
	saveAs("Tiff", savePath+fs+saveName);
	close();
	selectWindow(nameStore);
	close();

}

//if a 4x4 grid was selected then run the appropriate code to stich it and save the result
if(gridType=="4x4"){
	selectWindow(nameStore);
	run("Make Montage...", "columns=4 rows=4 scale=1");
	selectWindow("Montage");
	saveAs("Tiff", savePath+fs+saveName);
	close();
	selectWindow(nameStore);
	close();

}
```

Adaptable Version of Tile Merge

The above example was limited in the type of merges it could perform. Only grid types of 2x2, 3x3 and 4x4 could be merged. The following code shows how to make it adaptable to any number of rows and columns (assuming the user knows how many of each there are).

Open the previous macro and resave it as “Automatic Montage – Adaptable.ijm”

1. The initial steps for selecting the files to process and building the stack are identical and can be left.

```
//macro to merge any array of grid collected images

//store the OS specific file seperator into the fs variable for later file saving and directory creeation
fs=File.separator;

//as the user to select the first file of the montage and store its path in the filePath variable
filePath=File.openDialog("Select First File");

//load the images a sequence into a new stack
run("Image Sequence...", "open=["+filePath+"] sort");

//find the folder the image was opened from and create a sub directory in it called Merged
pathStore=getInfo("image.directory");
savePath=pathStore+fs+"Merged";
File.makeDirectory(savePath);

//store the name of the opened stack into the nameStore variable
nameStore=getTitle();

```

2. The dialog is similar to the last one but instead of hard set options for the grid the user will be asked how many rows and columns were collected.

Dialog.create("Tile Merger");

Dialog.addString("Enter Name for Final Merged Image","");

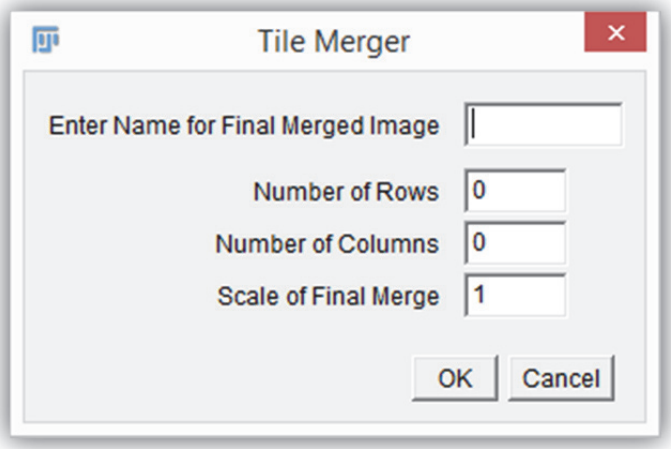
Dialog.addNumber("Number of Rows", 0);
Dialog.addNumber("Number of Columns", 0);

Dialog.addNumber("Scale of Final Merge", 1);

```
//create a dialog called Tile Merger
Dialog.create("Tile Merger");

//add an option to the dialog for the user to enter the name of the file to save
Dialog.addString("Enter Name for Final Merged Image","");

//Ask user for the number of rows and columns of data collected
Dialog.addNumber("Number of Rows", 0);
Dialog.addNumber("Number of Columns", 0);

//ask user for the scale of the final output image. 1=full resolution
Dialog.addNumber("Scale of Final Merge", 1);
```

3. As before we need to add the dialog.show option and then you can test to see if the dialog box looks ok.

```
Dialog.show();
```



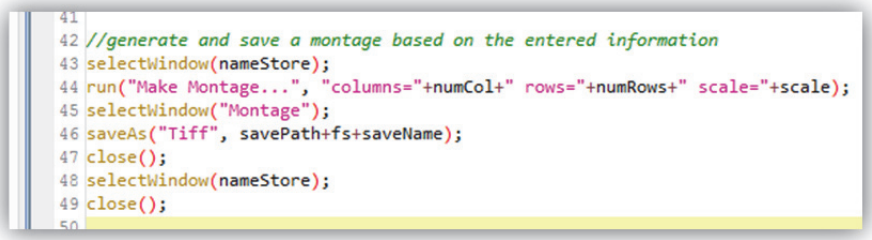
4. The entered information needs to be logged into variables.

```
saveName=Dialog.getString();
numRows=Dialog.getNumber();
numCol=Dialog.getNumber();
scale=Dialog.getNumber();
```

```
36 //collect the entered information and store it in the saveName and gridType variables
37 saveName=Dialog.getString();
38 numRows=Dialog.getNumber();
39 numCol=Dialog.getNumber();
40 scale=Dialog.getNumber();
41
```

5. The final code to create the montage is the same as last time but instead of being hard-coded to a limited number of array sizes the variables collected are entered into the montage creation string as well as the scale value.

```
selectWindow(nameStore);
run("Make Montage...", "columns="+numCol+" rows="+numRows+"
scale="+scale);
selectWindow("Montage");
saveAs("Tiff", savePath+fs+saveName);
close();
selectWindow(nameStore);
close();
```



```
41
42 //generate and save a montage based on the entered information
43 selectWindow(nameStore);
44 run("Make Montage...", "columns="+numCol+" rows="+numRows+" scale="+scale);
45 selectWindow("Montage");
46 saveAs("Tiff", savePath+fs+saveName);
47 close();
48 selectWindow(nameStore);
49 close();
50
```

6. Save the code as “Automatic Montage – Adaptable.ijm” as test it out.

Automatic Montage – Adaptable.ijm – Final Code

```
//macro to merge any array of grid collected images

//store the OS specific file seperator into the fs variable for later file saving and directory creeation
fs=File.separator;

//as the user to select the first file of the montage and store its path in the filePath variable
filePath=File.openDialog("Select First File");

//load the images a sequence into a new stack
run("Image Sequence...", "open=["+filePath+"] sort");

//find the folder the image was opened from and create a sub directory in it called Merged
pathStore=getInfo("image.directory");
savePath=pathStore+fs+"Merged";
File.makeDirectory(savePath);

//store the name of the opened stack into the nameStore variable
nameStore=getTitle();

//create a dialog called Tile Merger
Dialog.create("Tile Merger");

//add an option to the dialog for the user to enter the name of the file to save
Dialog.addString("Enter Name for Final Merged Image","");

//Ask user for the number of rows and columns of data collected
Dialog.addNumber("Number of Rows", 0);
Dialog.addNumber("Number of Columns", 0);

//ask user for the scale of the final output image. 1=full resolution
Dialog.addNumber("Scale of Final Merge", 1);

//display the dialog box
Dialog.show();
```

```
//collect the entered information and store it in the saveName and gridType variables
saveName=Dialog.getString();
numRows=Dialog.getNumber();
numCol=Dialog.getNumber();
scale=Dialog.getNumber();

//generate and save a montage based on the entered information
selectWindow(nameStore);
run("Make Montage...", "columns="+numCol+" rows="+numRows+" scale="+scale);
selectWindow("Montage");
saveAs("Tiff", savePath+fs+saveName);
close();
selectWindow(nameStore);
close();
```

Arrays

Aim

Entering data into arrays has already been used previously in this manual but it has been handled automatically by commands like the getDirectory command. There are times though when you need to generate a custom array to be able to generate the data required.

This example will let a user chop an image up into smaller tiles in any number of rows and columns taken from a custom dialog and then processed through several arrays to calculate the required coordinates for cropping each tile. The chopping is done by creating regions on the image and duplicating them to generate the smaller tile images.

This example uses the **Original.tif** file found in the **Demo Images\Widefield\Montage** folder. It needs to be manually opened before running or testing the code.

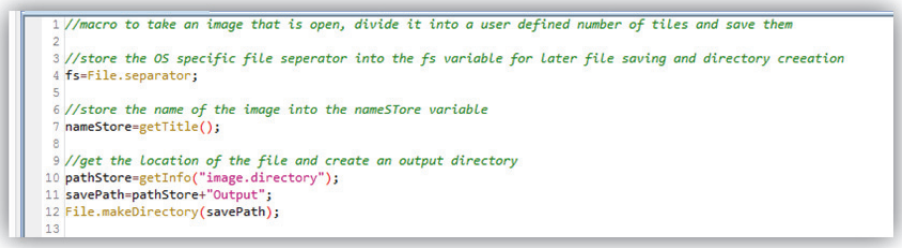
NOTE: A lot of the array commands do not change colour in the script editor so be careful when entering them to make sure they are correct.

Collecting File Information

As with other scripts we need to find the location of the file and generate an output directory for the resulting files to be saved in. **NOTE:** This script works on an image that has already been manually opened

1. Enter the usual file.separator, name storage and path creation code

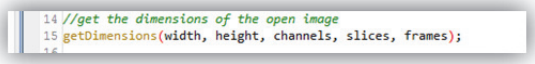
```
fs=File.separator;

nameStore=getTitle();

pathStore=getInfo("image.directory");
savePath=pathStore+"Output";
File.makeDirectory(savePath);
```



```
//macro to take an image that is open, divide it into a user defined number of tiles and save them

//store the OS specific file seperator into the fs variable for later file saving and directory creeation
fs=File.separator;

//store the name of the image into the nameSTore variable
nameStore=getTitle();

//get the location of the file and create an output directory
pathStore=getInfo("image.directory");
savePath=pathStore+"Output";
File.makeDirectory(savePath);

```

2. The next step is to find out how big the image is to be able to work out how to chop it up. This is done with the **getDimensions** command. This command needs to collect all dimension information (width, height, number of channels, number of slices and number of frames) and store them all in variables even if they won't be used. For this example we only need the width and height of the image.

getDimensions(width, height, channels, slices, frames);



Dialog Creation

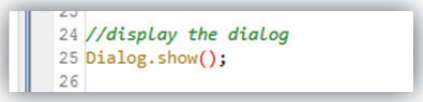
We now need to create a dialog box for collecting the data. This dialog is very similar to the ones in the custom dialogs section. It will ask the user for a base name and the number of rows and columns to chop the image up into. There is one addition to this one in that a message has been added to the top of the dialog to explain what it does. Take note that the addMessage part contains \n characters. These characters create a line break to keep text clumped together instead of one big long line.

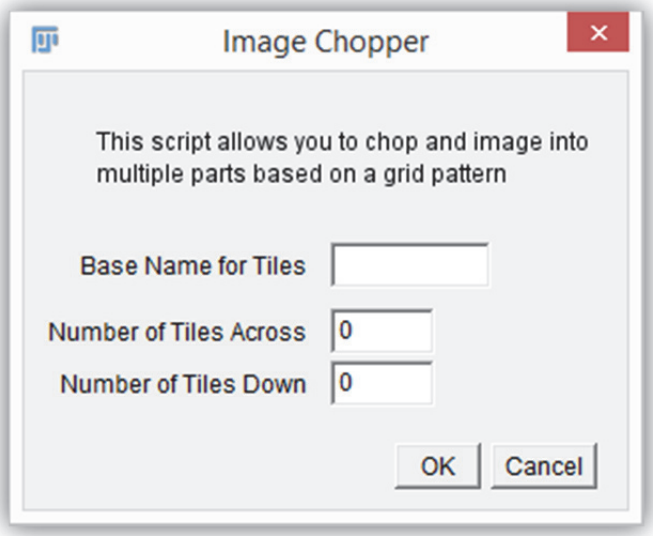
1. Enter the options for making a dialog box

```
Dialog.create("Image Chopper");
Dialog.addMessage("This script allows you to chop and image into \nmultiple parts based on a grid pattern\n \n");
Dialog.addString("Base Name for Tiles", "");
Dialog.addNumber("Number of Tiles Across", 0);
Dialog.addNumber("Number of Tiles Down", 0);
```

2. Show and test the dialog to make sure it is ok

```
Dialog.show();
```





Collecting the Information from the Dialog

As before we need to collect the dialog values into variables

```
tileName=Dialog.getString();
tilesAcross=Dialog.getNumber();
tilesDown=Dialog.getNumber();
```

```
27 //collect the dialog information into varibales
28 tileName=Dialog.getString();
29 tilesAcross=Dialog.getNumber();
30 tilesDown=Dialog.getNumber();
31
```

Calculating Tile Size and Number

The size and number of the regions is calculated based of the size of the image and the number of entered regions. The total number of regions is calculated by multiplying the number of rows and columns together. The width and height of the required regions is calculated by divding the dimensions of the image by the number of tiles across or down. **NOTE:** in the code the **floor** command has been used. This rounds any decimal values down to the nearest whole number. This avoids having decimals in region sizes and stops the last region falling off the edge of the image and crashing the code.

```
totalTiles=tilesAcross*tilesDown;

regionWidth=floor(width/tilesAcross);
regionHeight=floor(height/tilesDown);
```

```
32 //calculate the total number of tiles to be processed
33 totalTiles=tilesAcross*tilesDown;
34
35 //determine the width and height required for the crop regions based on the number of rows and columns defined
36 regionWidth=floor(width/tilesAcross);
37 regionHeight=floor(height/tilesDown);
38
```

Creating New Arrays

To be able to easily process the regions into tiles we need the coordinates need to be easily accessible. The easiest way to do this is to put them into an array. For this example we will generate an array for the x coordinates and one for the y coordinates. These will then be processed into a third array in pairs of coordinates to use with the specify selection command.

```
xPosList=newArray();
yPosList=newArray();
PosList=newArray();
```

```
39 //generate blank arrays for storing the region coordinates
40 xPosList=newArray();
41 yPosList=newArray();
42 PosList=newArray();
43
```

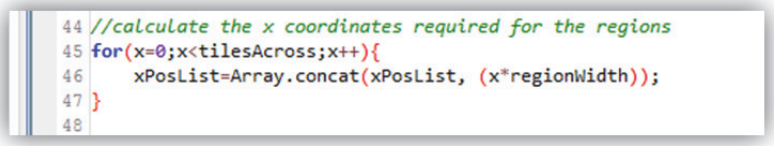
Calculate Coordinates

We now know how big a region should be, how many there should be and some arrays to store them in. We now need to calculate the coordinates of each in x and y and add them to the array lists xPosList and yPosList.

This will make use of the array concatenate command to add each calculated value onto the end of the respective array list.

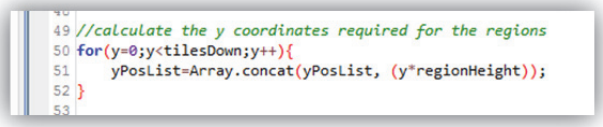
1. Firstly calculate the x coordinates needed. By starting at 0 (defined by the x value) and looping for the number of define tiles across each of the required x coordinates are achieved by multiplying the value of x by the calculated width of the required region.

```
for(x=0;x<tilesAcross;x++){
        xPosList=Array.concat(xPosList, (x*regionWidth));
}
```



2. The same concept can be repeated for the y coordinates.

```
for(y=0;y<tilesDown;y++){
        yPosList=Array.concat(yPosList, (y*regionHeight));
}
```



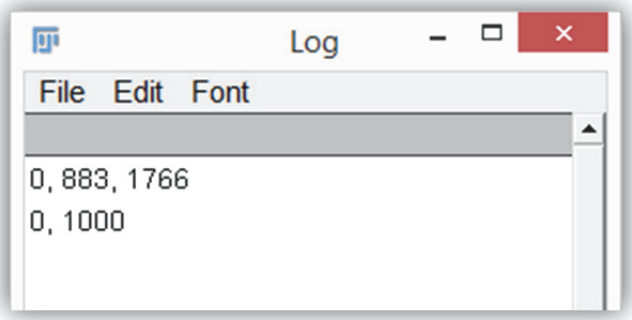
Checking the Calculated Coordinate Lists

You can check to see if the calculated coordinates make sense by adding an Array.print command for the x and y array.

Array.print(xPosList);
Array.print(yPosList);

```
54 //print the xPosList and yPoslist arrays to check
55 Array.print(xPosList);
56 Array.print(yPosList);
```

Open the **Original.tif** image and run the code written so far to see what output you get. The example shown here was ran on a 3 across, 2 down setting. The entries for each array should be printed into the log window.



Combining the Coordinates into an Array

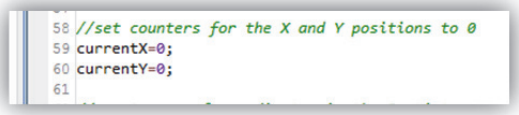
The coordinates calculated above now need to be combined into another array to make it easier to process. We need the coordinates of each roi in the terms of x and y coordinates. Using the output from the test above we need the following pairs of coordinates.

0,0
883,0
1766,0
0,1000
883,1000
1766,1000

To do this we carry out another concatenation where we take alternating values from the x and y lists as follows

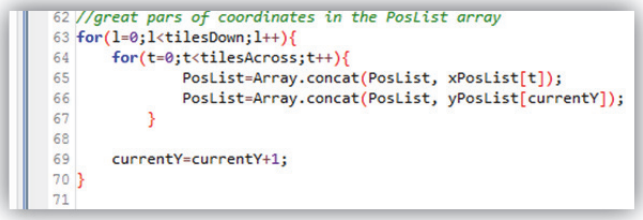
1. We need a couple of counters to keep track of where we are in the lists. These are initially set to 0

```
currentX=0;
currentY=0;
```



2. Now the coordinate pairs can be worked out and added to the PosList array

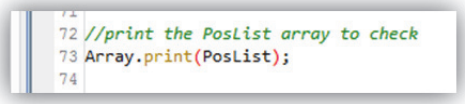
```
for(l=0;l<tilesDown;l++){
        for(t=0;t<tilesAcross;t++){
                PosList=Array.concat(PosList, xPosList[t]);
                PosList=Array.concat(PosList, yPosList[currentY]);
            }

        currentY=currentY+1;
}
```



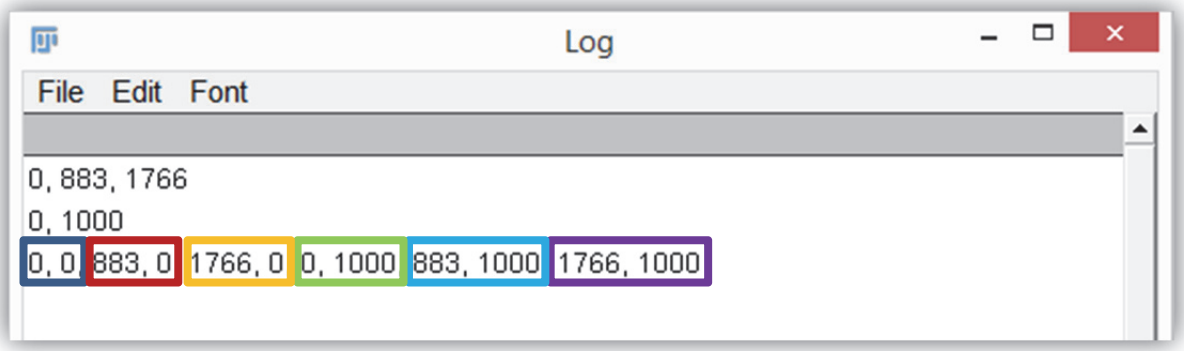
Checking the Paired Coordinates

As before we can use the Array.print command to see if the concatenation of the two lists has worked.

```
Array.print(PosList);
```



As previously run the code up until this point and look at the log output. If you look at the data in pairs they should be the coordinates we need to do the final part.

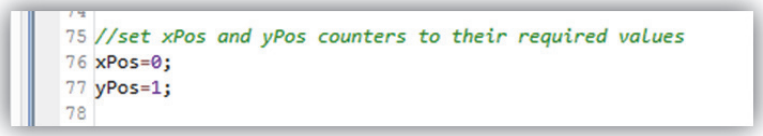


Duplicating the Regions

The final step is to use all the information we now have to duplicate the regions as tiles and save them to the output directory.

1. We need some counters again to be able to pull the correct entries out of the array list

```
xPos=0;
yPos=0;
```



2. Now we loop the following code for the number of title we want in the end. In each loop the positions in the PosList array for the coordinates are read out then have a value of 2 added to them moving them down the list to the next pair.

```
for(t=0;t<totalTiles;t++){
                selectWindow(nameStore);
                run("Specify...", "width="+regionWidth+"
height="+regionHeight+" x="+PosList[xPos]+" y="+PosList[yPos]);
                run("Duplicate...", "title="+tileName+"-"+(t+1));
                saveAs("Tiff", savePath+fs+tileName+"-"+(t+1));
                close();
                xPos=xPos+2;
                yPos=yPos+2;
}
```

```
//duplicate regions and save them based on the coordinates calculated
for(t=0;t<totalTiles;t++){
        selectWindow(nameStore);
        run("Specify...", "width="+regionWidth+" height="+regionHeight+" x="+PosList[xPos]+" y="+PosList[yPos]);
        run("Duplicate...", "title="+tileName+"-"+(t+1));
        saveAs("Tiff", savePath+fs+tileName+"-"+(t+1));
        close();
        xPos=xPos+2;
        yPos=yPos+2;
}
```

3. Add a **run("Close All");** command to the end of your code. Save it as "File Cropper – Adaptable.ijm", open the original.tif image if it isn't already and test it out. Try different combinations of rows and columns, it should cope with them all.

File Cropper – Adaptable.ijm – Final Code

```
//macro to take an image that is open, divide it into a user defined number of tiles and save them

//store the OS specific file seperator into the fs variable for later file saving and directory creeation
fs=File.separator;

//store the name of the image into the nameSTore variable
nameStore=getTitle();

//get the location of the file and create an output directory
pathStore=getInfo("image.directory");
savePath=pathStore+"Output";
File.makeDirectory(savePath);

//get the dimensions of the open image
getDimensions(width, height, channels, slices, frames);

//create a dialog box asking for a base name and the number of rows and columns
Dialog.create("Image Chopper");
Dialog.addMessage("This script allows you to chop and image into \nmultiple parts based on a grid pattern\n \n");
Dialog.addString("Base Name for Tiles", "");
Dialog.addNumber("Number of Tiles Across", 0);
Dialog.addNumber("Number of Tiles Down", 0);

//display the dialog
Dialog.show();

//collect the dialog information into varibales
tileName=Dialog.getString();
tilesAcross=Dialog.getNumber();
tilesDown=Dialog.getNumber();

//calculate the total number of tiles to be processed
```

```
totalTiles=tilesAcross*tilesDown;

//determine the width and height required for the crop regions based on the number of rows and columns defined
regionWidth=floor(width/tilesAcross);
regionHeight=floor(height/tilesDown);

//generate blank arrays for storing the region coordinates
xPosList=newArray();
yPosList=newArray();
PosList=newArray();

//calculate the x coordinates required for the regions
for(x=0;x<tilesAcross;x++){
	xPosList=Array.concat(xPosList, (x*regionWidth));
}

//calculate the y coordinates required for the regions
for(y=0;y<tilesDown;y++){
	yPosList=Array.concat(yPosList, (y*regionHeight));
}

//print the xPosList and yPoslist arrays to check
Array.print(xPosList);
Array.print(yPosList);

//set counters for the X and Y positions to 0
currentX=0;
currentY=0;

//great pars of coordinates in the PosList array
for(l=0;l<tilesDown;l++){
	for(t=0;t<tilesAcross;t++){
			PosList=Array.concat(PosList, xPosList[t]);
			PosList=Array.concat(PosList, yPosList[currentY]);
		}

	currentY=currentY+1;
}

//print the PosList array to check
Array.print(PosList);

//set xPos and yPos counters to their required values
xPos=0;
yPos=1;

//duplicate regions and save them based on the coordinates calculated
```

```
for(t=0;t<totalTiles;t++){
                    selectWindow(nameStore);
                    run("Specify...", "width="+regionWidth+" height="+regionHeight+"
x="+PosList[xPos]+" y="+PosList[yPos]);
                    run("Duplicate...", "title="+tileName+"-"+(t+1));
                    saveAs("Tiff", savePath+fs+tileName+"-"+(t+1));
                    close();
                    xPos=xPos+2;
                    yPos=yPos+2;
}

run("Close All");
```

Functions

Aim

With longer and more complex macro code it can become unwieldy to easily edit it or port it across to other uses. This can be made easier by adding functions to the code.

Functions are stubs of code that carry out a set bunch of code, for example split channels and rename them. This stub is referenced by a variable and can be called and ran anytime within the code by calling just the single variable.

For this technical note we will modify two of the previous macros we created (**Vessel and Hypoxia Measurement (with Result Output).ijm** and **Ki67 and PH3 Count (Refined with Output).ijm**) to use functions.

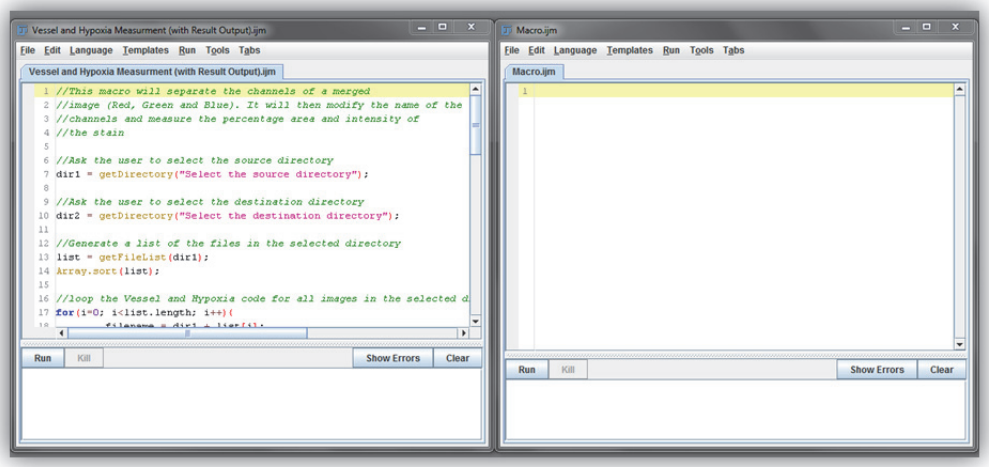
Global Variables

When using functions any variables that are created within the functions will not be available to other functions or the rest of the macro. To avoid this the variables need to be assigned globally at the start of the macro. The command defining a global variable is **var** and then the variable. For example

```
var cellCount = 0;
```

Vessel and Hypoxia Measurement

For the following steps we will be copying blocks of code from the original code (**Vessel and Hypoxia Measurement (with Result Output).ijm)** and pasting it into the new code. To make this easier it is best to have two separate macro editor windows open. To do this open the original macro and then go to **Plugins → New → Macro** to open a separate macro editor window.

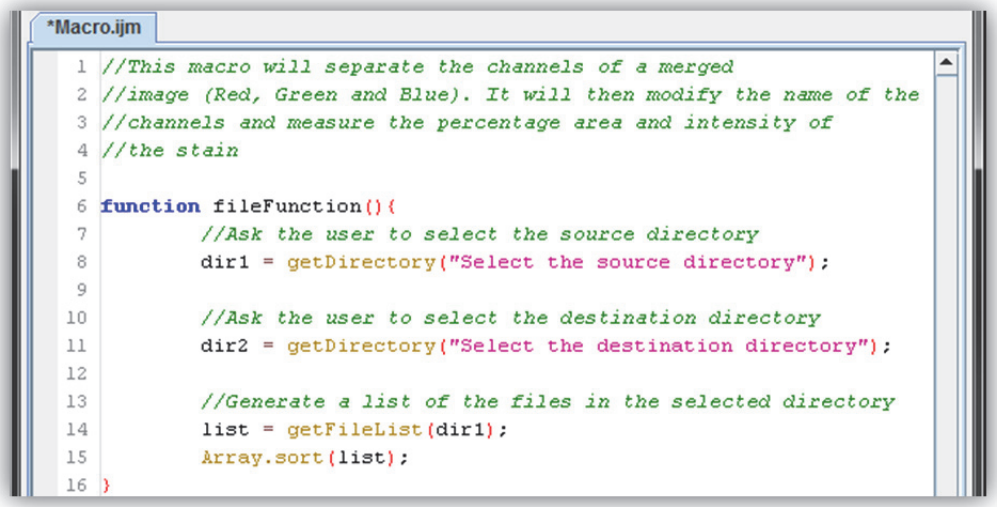


Directory Selection and Array Function

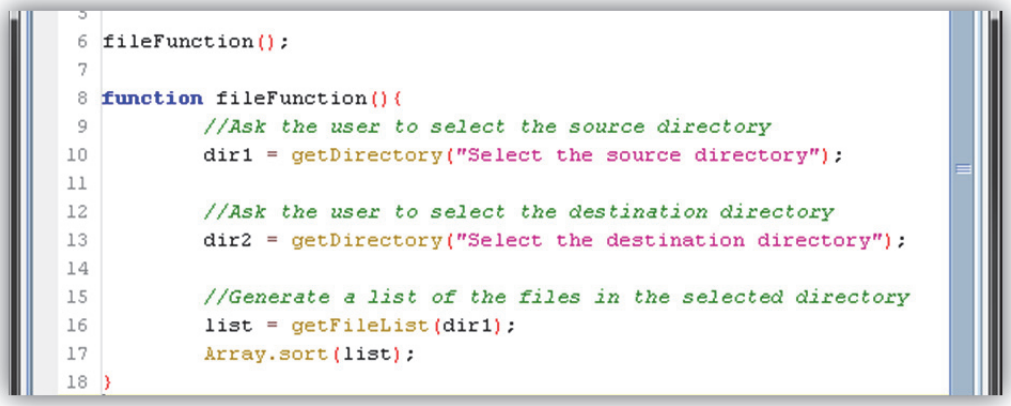
1. Take the first steps from the original macro and paste them into the new blank macro as follows

```
function fileFunction(){
        //Ask the user to select the source directory
        dir1 = getDirectory("Select the source directory");

        //Ask the user to select the destination directory
        dir2 = getDirectory("Select the destination directory");

        //Generate a list of the files in the selected directory
        list = getFileList(dir1);
        Array.sort(list);
        }
```

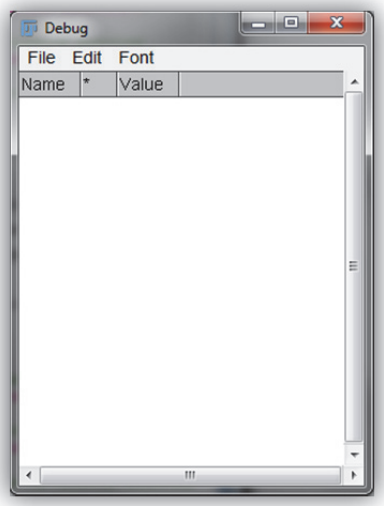
The **function** command defines the function the variable afterwards (in this case **fileFunction()**) is the variable that will be used to call the function when required. Also note that the function is contained within **{}**



2. To be able to run the function you need to call it up in the code. It doesn't matter where you call it, but the general rule is to define functions at the bottom of the code and run them from the top. Add **fileFunction();** to the top of the code



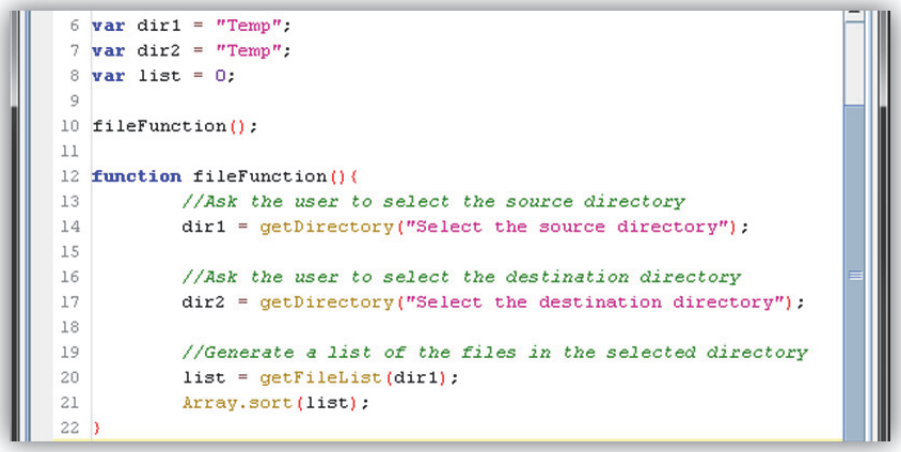
3. Save the macro as **Vessel and Hypoxia Measurement (with Result Output – functions).ijm**, add a string of garbage characters at the end after the **}** and try running it so that you can call up the debug window. You will notice that the debug window has nothing in it, though it should have the variables for dir1, dir2 and list in it. This is because the variables are only defined within the function and are therefore not available to the rest of the macro.



4. Define the required variables as global variables by putting the following code at the top of the macro

```
var dir1 = “Temp”;
var dir2 = “Temp”;
var list = 0;
```

It doesn’t matter what the variables are defined as they just need to have something assigned. It will be over written with what I needed when the full code runs.



```
6  var dir1 = "Temp";
7  var dir2 = "Temp";
8  var list = 0;
9
10 fileFunction();
11
12 function fileFunction(){
13         //Ask the user to select the source directory
14         dir1 = getDirectory("Select the source directory");
15
16         //Ask the user to select the destination directory
17         dir2 = getDirectory("Select the destination directory");
18
19         //Generate a list of the files in the selected directory
20         list = getFileList(dir1);
21         Array.sort(list);
22 }
```

5. Save the code, make sure the garbage characters are still at the end and run it again. You should now get a debug window that contains populated dir1, dir2 and list variables.

6. Remove the garbage characters and copy the start of the for loop function across as below

```
6  var dir1 = "Temp";
7  var dir2 = "Temp";
8  var list = 0;
9
10 fileFunction();
11
12 //loop the Vessel and Hypoxia code for all images in the selected d
13 for(i=0; i<list.length; i++){
14         filename = dir1 + list[i];
15         if (endsWith(filename, "tif")) {
16                 open(filename);
17
18         }
19
20 function fileFunction(){
```

Split Images Function

1. Next we will define the function to spilt the image and load the names into variables

```
function channelFunction(){
        //Store the initial image name into the nameStore variable
        nameStore = getTitle();

        //Split the channels into red, green and blue
        run("Split Channels");

        //Rename Red Image to original name + vessels and green image to
        //original name + hypoxia
        selectWindow(nameStore+" (red)");
        rename(nameStore+" - Vessels");
        redImage = getTitle();

        selectWindow(nameStore+" (green)");
        rename(nameStore+" - Hypoxia");
        greenImage = getTitle();
}
```

```
function channelFunction(){
        //Store the initial image name into the nameStore variable
        nameStore = getTitle();

        //Split the channels into red, green and blue
        run("Split Channels");

        //Rename Red Image to original name + vessels and green ima
        //original name + hypoxia
        selectWindow(nameStore+" (red)");
        rename(nameStore+" - Vessels");
        redImage = getTitle();

        selectWindow(nameStore+" (green)");
        rename(nameStore+" - Hypoxia");
        greenImage = getTitle();
}
```

2. Now we need to add to the global variable list the new variables that were called in this function

var nameStore = "Temp";
var redImage = "Temp";
var greenImage = "Temp";

```
var dir1 = "Temp";
var dir2 = "Temp";
var list = 0;
var nameStore = "Temp";
var redImage = "Temp";
var greenImage = "Temp";

fileFunction();

```

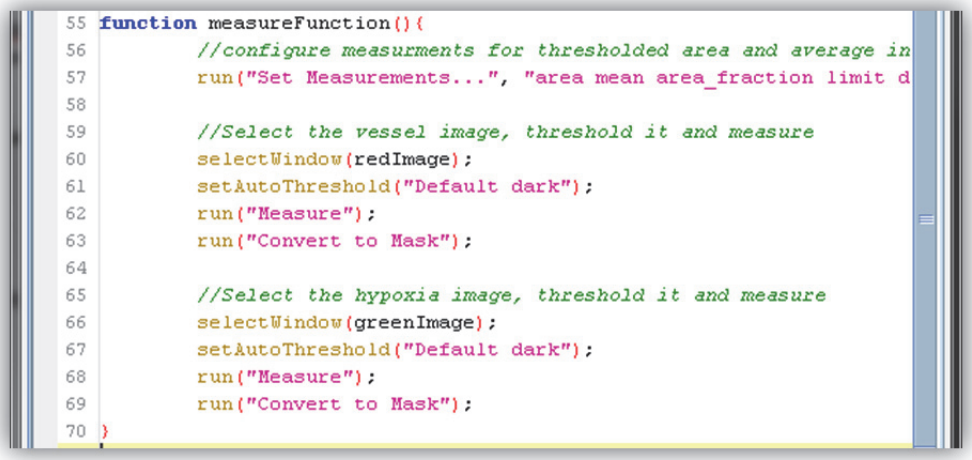
3. Add the **fileFunction** call inside the for/if loop

```
//loop the Vessel and Hypoxia code for all images in the selected d
for(i=0; i<list.length; i++){
        filename = dir1 + list[i];
        if (endsWith(filename, "tif")) {
                open(filename);

                channelFunction();
        }
```

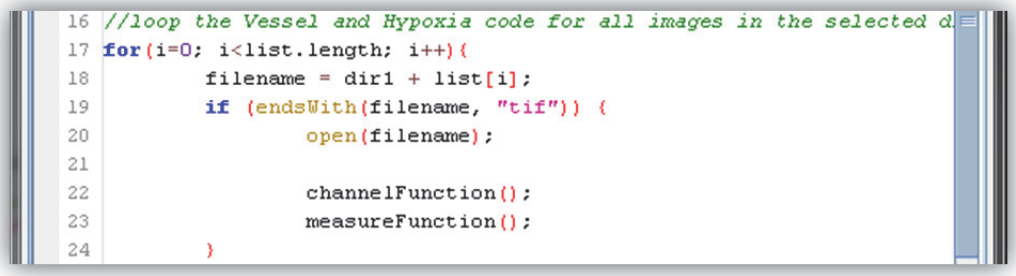
Measure Function

1. Next we will define the function to measure the images

```
function measureFunction(){
        //configure measurments for thresholded area and average intensity
        run("Set Measurements...", "area mean area_fraction limit display
        redirect=None decimal=3");

        //Select the vessel image, threshold it and measure
        selectWindow(redImage);
        setAutoThreshold("Default dark");
        run("Measure");
        run("Convert to Mask");

        //Select the hypoxia image, threshold it and measure
        selectWindow(greenImage);
        setAutoThreshold("Default dark");
        run("Measure");
        run("Convert to Mask");
}
```



2. There are no new variables in this function so there is no need to add anything to the global variable list. Just add the **measureFunction** to the for/if loop

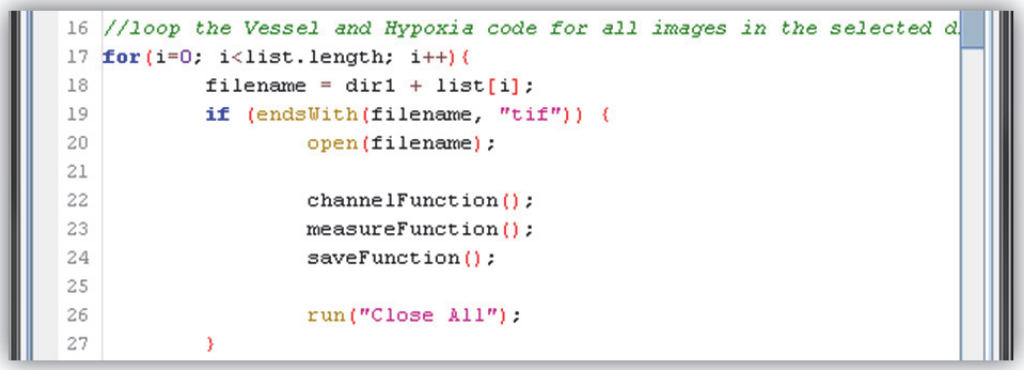


Save Overview Images Function

1. Lastly define the function to save the overview images

```
function saveFunction(){
        //merge Binary masks for result image
        run("Merge Channels...", "c1=["+redImage+"] c2=["+greenImage+"]");

        //save merged image into the folder of the original image
        //append name with " - Overview" on the end
        selectWindow("RGB");
        rename(nameStore+" - Overview");
        saveAs("Tiff", dir2+nameStore+" - Overview.tif");
}
```

2. Add the function into the for/if loop and also add a close all command as well.



3. Save the macro and run it. The result should be the same as the original. In the future if it needs to be modified you can just edit the required function

Vessel and Hypoxia Measurement (with Result Output – functions).ijm – Final Code

```
//This macro will separate the channels of a merged
//image (Red, Green and Blue). It will then modify the name of the
//channels and measure the percentage area and intensity of
//the stain

var dir1 = "Temp";
var dir2 = "Temp";
var list = 0;
var nameStore = "Temp";
var redImage = "Temp";
var greenImage = "Temp";

fileFunction();


//loop the Vessel and Hypoxia code for all images in the selected directory
for(i=0; i<list.length; i++){
	filename = dir1 + list[i];
	if (endsWith(filename, "tif")) {
		open(filename);

		channelFunction();
		measureFunction();
		saveFunction();

		run("Close All");
	}
}
function fileFunction(){
	//Ask the user to select the source directory
	dir1 = getDirectory("Select the source directory");

	//Ask the user to select the destination directory
	dir2 = getDirectory("Select the destination directory");

	//Generate a list of the files in the selected directory
	list = getFileList(dir1);
	Array.sort(list);
}

function channelFunction(){
	//Store the initial image name into the nameStore variable
	nameStore = getTitle();

	//Split the channels into red, green and blue
	run("Split Channels");
```

```
	//Rename Red Image to original name + vessels and green image to
	//original name + hypoxia
	selectWindow(nameStore+" (red)");
	rename(nameStore+" - Vessels");
	redImage = getTitle();

	selectWindow(nameStore+" (green)");
	rename(nameStore+" - Hypoxia");
	greenImage = getTitle();
}

function measureFunction(){
	//configure measurments for thresholded area and average intensity
	run("Set Measurements...", "area mean area_fraction limit display redirect=None
decimal=3");

	//Select the vessel image, threshold it and measure
	selectWindow(redImage);
	setAutoThreshold("Default dark");
	run("Measure");
	run("Convert to Mask");

	//Select the hypoxia image, threshold it and measure
	selectWindow(greenImage);
	setAutoThreshold("Default dark");
	run("Measure");
	run("Convert to Mask");
}

function saveFunction(){
	//merge Binary masks for result image
	run("Merge Channels...", "c1=["+redImage+"] c2=["+greenImage+"]");

	//save merged image into the folder of the original image
	//append name with " - Overview" on the end
	selectWindow("RGB");
	rename(nameStore+" - Overview");
	saveAs("Tiff", dir2+nameStore+" - Overview.tif");
}
```

Ki67 and PH3 Count

Apply what you learnt above with the **Ki67 and PH3 Count (Refined with Output).ijm** by creating functions for each of the major components of the macro

Global Variables

```
var dir = "Temp";
var saveDir = "Temp";
var fs = "Temp";
var list = 0;
var tableTitle = "Temp";
var tableTitle2 = "Temp";
var filename = "Temp";
var nameStore = "Temp";
var DAPIImage = "Temp";
var Ki67Image = "Temp"
var PH3Image = "Temp";
var DAPICount = "Temp";
var Ki67Count = "Temp";
var PH3Count = "Temp";
var Ki67Percent = "Temp";
var PH3Percent = "temp";
```

File Function

```
function fileFunction(){
	//ask user to select source directory
	dir = getDirectory("Select Source Directory");

	//define the output directory and create it
	saveDir = dir+"Output";
	File.makeDirectory(saveDir);
	fs = File.separator;

	//generate file list for batch processing
	list = getFileList(dir);
	Array.sort(list);
}
```

```
46 function fileFunction(){
47         //ask user to select source directory
48         dir = getDirectory("Select Source Directory");
49
50         //define the output directory and create it
51         saveDir = dir+"Output";
52         File.makeDirectory(saveDir);
53         fs = File.separator;
54
55         //generate file list for batch processing
56         list = getFileList(dir);
57         Array.sort(list);
58 }
```

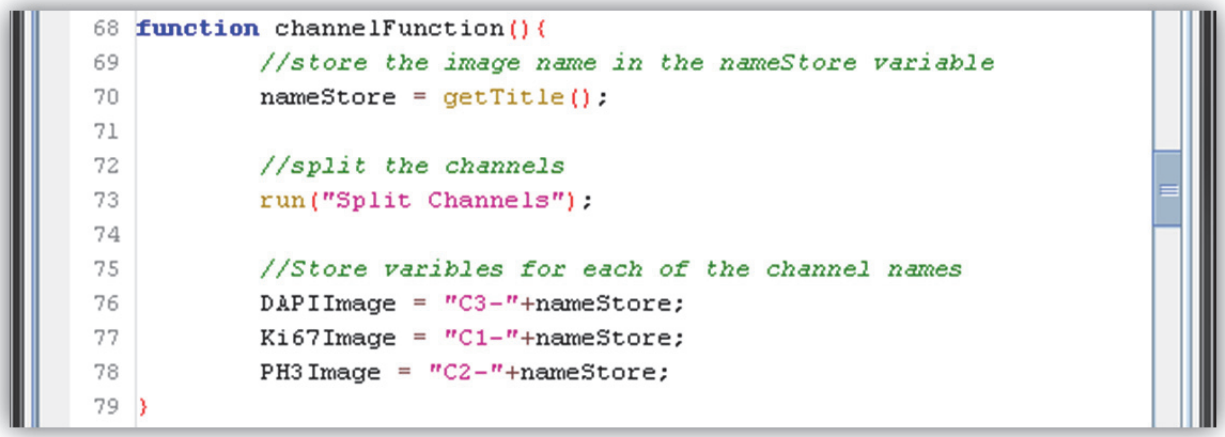
Make Table Function

```
function makeTable(){
        //Prepare a log table for the data to be logged to
        tableTitle="Cell Population";
        tableTitle2="["+tableTitle+"]";
        run("Table...", "name="+tableTitle2+" width=600 height=250");
        print(tableTitle2, "\\Headings:Image Name\tTotal Cells\tKi67 Cells\tPH3 Cells\tPercent
        Ki67\tPercent PH3");
}
```

```
60 function makeTable(){
61         //Prepare a log table for the data to be logged to
62         tableTitle="Cell Population";
63         tableTitle2="["+tableTitle+"]";
64         run("Table...", "name="+tableTitle2+" width=600 height=250
65         print(tableTitle2, "\\Headings:Image Name\tTotal Cells\tKi
66 }
```

Channel Function

```
function channelFunction(){
	//store the image name in the nameStore variable
	nameStore = getTitle();

	//split the channels
	run("Split Channels");

	//Store varibles for each of the channel names
	DAPIImage = "C3-"+nameStore;
	Ki67Image = "C1-"+nameStore;
	PH3Image = "C2-"+nameStore;
}
```



```
68 function channelFunction(){
69         //store the image name in the nameStore variable
70         nameStore = getTitle();
71
72         //split the channels
73         run("Split Channels");
74
75         //Store varibles for each of the channel names
76         DAPIImage = "C3-"+nameStore;
77         Ki67Image = "C1-"+nameStore;
78         PH3Image = "C2-"+nameStore;
79 }
```

Count DAPI Function

```
function countDAPI(){
	//Threshold and count the number of nuclei in the DAPI image
	run("Set Measurements...", "  area display redirect=None decimal=3");
	selectWindow(DAPIImage);
	setAutoThreshold("Li dark");
	run("Convert to Mask");
	run("Watershed");
	run("Analyze Particles...", "size=200-Infinity circularity=0.00-1.00 show=Nothing display
clear");
	DAPICount = nResults;
}
```

```
81 function countDAPI(){
82         //Threshold and count the number of nuclei in the DAPI ima
83         run("Set Measurements...", "  area display redirect=None de
84         selectWindow(DAPIImage);
85         setAutoThreshold("Li dark");
86         run("Convert to Mask");
87         run("Watershed");
88         run("Analyze Particles...", "size=200-Infinity circularity=
89         DAPICount = nResults;
90 }
```

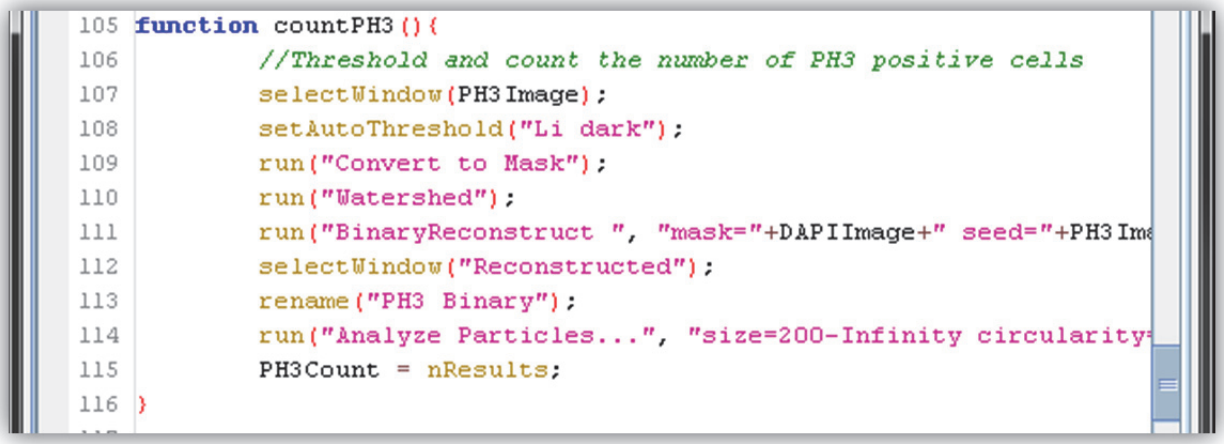
Count Ki67 Function

```
function countKi67(){
        //Threshold and count the number of Ki67 positive cells
        selectWindow(Ki67Image);
        setAutoThreshold("Li dark");
        run("Convert to Mask");
        run("Watershed");
        run("BinaryReconstruct ", "mask="+DAPIImage+" seed="+Ki67Image+" create white");
        selectWindow("Reconstructed");
        rename("Ki67 Binary");
        run("Analyze Particles...", "size=200-Infinity circularity=0.00-1.00 show=Nothing display
        clear");
        Ki67Count = nResults;
}
```

```
92 function countKi67(){
93         //Threshold and count the number of Ki67 positive cells
94         selectWindow(Ki67Image);
95         setAutoThreshold("Li dark");
96         run("Convert to Mask");
97         run("Watershed");
98         run("BinaryReconstruct ", "mask="+DAPIImage+" seed="+Ki67Im
99         selectWindow("Reconstructed");
100        rename("Ki67 Binary");
101        run("Analyze Particles...", "size=200-Infinity circularity=
102        Ki67Count = nResults;
103 }
```

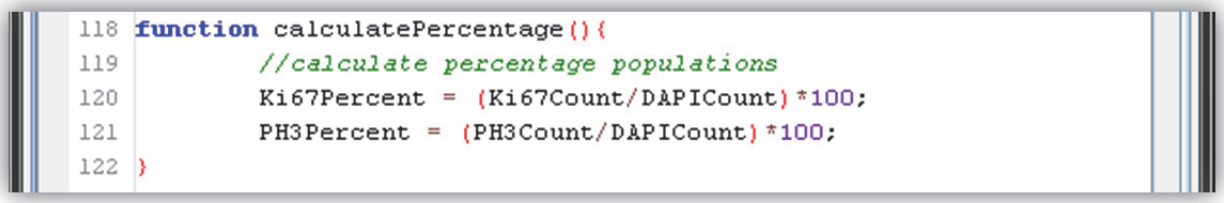
Count PH3 Function

```
function countPH3(){
        //Threshold and count the number of PH3 positive cells
        selectWindow(PH3Image);
        setAutoThreshold("Li dark");
        run("Convert to Mask");
        run("Watershed");
        run("BinaryReconstruct ", "mask="+DAPIImage+" seed="+PH3Image+" create white");
        selectWindow("Reconstructed");
        rename("PH3 Binary");
        run("Analyze Particles...", "size=200-Infinity circularity=0.00-1.00 show=Nothing display
        clear");
        PH3Count = nResults;
}
```



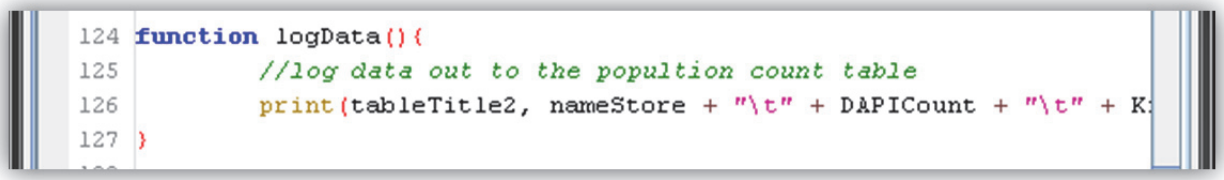
Calculate Percentage Function

```
function calculatePercentage(){
        //calculate percentage populations
        Ki67Percent = (Ki67Count/DAPICount)*100;
        PH3Percent = (PH3Count/DAPICount)*100;
}
```



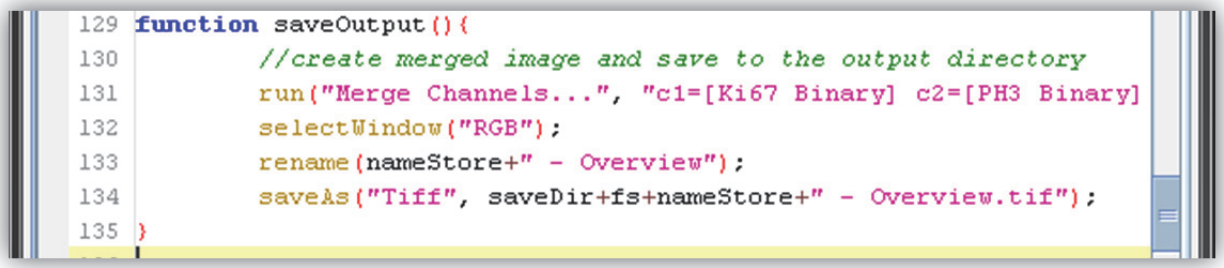
Log Data Function

```
function logData(){
	//log data out to the popultion count table
	print(tableTitle2, nameStore + "\t" + DAPICount + "\t" + Ki67Count + "\t" + PH3Count +
	"\t" + Ki67Percent + "\t" + PH3Percent);
}
```



Save Image Output Function

```
function saveOutput(){
	//create merged image and save to the output directory
	run("Merge Channels...", "c1=[Ki67 Binary] c2=[PH3 Binary] c3="+DAPIImage+" ignore");
	selectWindow("RGB");
	rename(nameStore+" - Overview");
	saveAs("Tiff", saveDir+fs+nameStore+" - Overview.tif");
}
```



Ki67 and PH3 Count (Refined with Output – functions).ijm – Final Code

```
//This macro will separate a 3 channel image that is stained
//with DAPI, Ki67 and PH3. The number of cells in each channel
//will be counted and the percentage populations calculated.
//Finally the data will logged to a custom log table

var dir = "Temp";
var saveDir = "Temp";
var fs = "Temp";
var list = 0;
var tableTitle = "Temp";
var tableTitle2 = "Temp";
var filename = "Temp";
var nameStore = "Temp";
var DAPIImage = "Temp";
var Ki67Image = "Temp"
var PH3Image = "Temp";
var DAPICount = "Temp";
var Ki67Count = "Temp";
var PH3Count = "Temp";
var Ki67Percent = "Temp";
var PH3Percent = "temp";

fileFunction();
makeTable();


for(i=0;i<list.length;i++){
	filename = dir +list[i];
	if (endsWith(filename, "tif")) {
		open(filename);

		channelFunction();
		countDAPI();
		countKi67();
		countPH3();
		calculatePercentage();
		logData();
		saveOutput();

		//close all open images
		run("Close All");
		close("Results");
	}
}

function fileFunction(){
	//ask user to select source directory
```

```
	dir = getDirectory("Select Source Directory");

	//define the output directory and create it
	saveDir = dir+"Output";
	File.makeDirectory(saveDir);
	fs = File.separator;

	//generate file list for batch processing
	list = getFileList(dir);
	Array.sort(list);
}

function makeTable(){
	//Prepare a log table for the data to be logged to
	tableTitle="Cell Population";
	tableTitle2="["+tableTitle+"]";
	run("Table...", "name="+tableTitle2+" width=600 height=250");
	print(tableTitle2, "\\Headings:Image Name\tTotal Cells\tKi67 Cells\tPH3 Cells\tPercent
Ki67\tPercent PH3");
}

function channelFunction(){
	//store the image name in the nameStore variable
	nameStore = getTitle();

	//split the channels
	run("Split Channels");

	//Store varibles for each of the channel names
	DAPIImage = "C3-"+nameStore;
	Ki67Image = "C1-"+nameStore;
	PH3Image = "C2-"+nameStore;
}

function countDAPI(){
	//Threshold and count the number of nuclei in the DAPI image
	run("Set Measurements...", "  area display redirect=None decimal=3");
	selectWindow(DAPIImage);
	setAutoThreshold("Li dark");
	run("Convert to Mask");
	run("Watershed");
	run("Analyze Particles...", "size=200-Infinity circularity=0.00-1.00 show=Nothing display
clear");
	DAPICount = nResults;
}

function countKi67(){
	//Threshold and count the number of Ki67 positive cells
```

```
	selectWindow(Ki67Image);
	setAutoThreshold("Li dark");
	run("Convert to Mask");
	run("Watershed");
	run("BinaryReconstruct ", "mask="+DAPIImage+" seed="+Ki67Image+" create white");
	selectWindow("Reconstructed");
	rename("Ki67 Binary");
	run("Analyze Particles...", "size=200-Infinity circularity=0.00-1.00 show=Nothing display
clear");
	Ki67Count = nResults;
}

function countPH3(){
	//Threshold and count the number of PH3 positive cells
	selectWindow(PH3Image);
	setAutoThreshold("Li dark");
	run("Convert to Mask");
	run("Watershed");
	run("BinaryReconstruct ", "mask="+DAPIImage+" seed="+PH3Image+" create white");
	selectWindow("Reconstructed");
	rename("PH3 Binary");
	run("Analyze Particles...", "size=200-Infinity circularity=0.00-1.00 show=Nothing display
clear");
	PH3Count = nResults;
}

function calculatePercentage(){
	//calculate percentage populations
	Ki67Percent = (Ki67Count/DAPICount)*100;
	PH3Percent = (PH3Count/DAPICount)*100;
}

function logData(){
	//log data out to the popultion count table
	print(tableTitle2, nameStore + "\t" + DAPICount + "\t" + Ki67Count + "\t" + PH3Count + "\t" +
Ki67Percent + "\t" + PH3Percent);
}

function saveOutput(){
	//create merged image and save to the output directory
	run("Merge Channels...", "c1=[Ki67 Binary] c2=[PH3 Binary] c3="+DAPIImage+" ignore");
	selectWindow("RGB");
	rename(nameStore+" - Overview");
	saveAs("Tiff", saveDir+fs+nameStore+" - Overview.tif");
}
```

Appendix

Fiji Training Notes

Appendix A

Built-in ImageJ Macro Functions

Macro Functions

The following list shows the vast majority (some are undocumented) of the macro functons built into the ImageJ macro language. The list can also be accessed from here

http://rsbweb.nih.gov/ij/developer/macro/functions.html

The hyperlinks in the following list should work if you are online.

Macro Functions List

A

abs(n)
Returns the absolute value of *n*.

acos(n)
Returns the inverse cosine (in radians) of *n*.

Array Functions
These functions operate on arrays. Refer to the ArrayFunctions macro for examples.

Array.concat(array1,array2) - Returns a new array created by joining two or more arrays or values (examples). Requires 1.46c.
Array.copy(array) - Returns a copy of *array*.
Array.fill(array, value) - Assigns the specified numeric value to each element of *array*.
Array.findMaxima(array, tolerance) - Returns an array holding the peak positions (sorted with descending strength). Tolerance is the minimum amplitude difference to needed to separate two peaks. There is an optional 'excludeOnEdges' argument that defaults to 'true'. Examples. Requires 1.48c.
Array.findMinima(array, tolerance) - Returns an array holding the minima positions. Requires 1.48c.
Array.fourier(array, windowType) - Calculates and returns the Fourier amplitudes of *array*. *WindowType* can be "none", "Hamming", "Hann", or "flat-top", or may be omitted (meaning "none"). See the TestArrayFouriermacro for an example and more documentation. Requires 1.49i.
Array.getSequence(n) - Returns an array containing the numeric sequence 0,1,2...n-1. Requires 1.49u.

Array.getStatistics(array, min, max, mean, stdDev) - Returns the *min*,*max*, *mean*, and *stdDev* of *array*, which must contain all numbers.
Array.print(array) - Prints the array on a single line. Requires 1.46c.
Array.rankPositions(array) - Returns, as an array, the rank position indexes of *array*, starting with the index of the smallest value (example).
Array.resample(array,len) - Returns an array which is linearly resampled to a different length. Requires 1.47j.
Array.reverse(array) - Reverses (inverts) the order of the elements in*array*. Requires 1.46c.
Array.show(array) - Displays the contents of *array* in a window. Requires 1.48d.
Array.show("title", array1, array2, ...) - Displays one or more arrays in a Results window (examples). If *title* (optional) is "Results", the window will be the active Results window, otherwise, it will be a dormant Results window (see also IJ.renameResults). If *title* ends with "(indexes)", a 0-based Index column is shown. If *title* ends with "(row numbers)", the row number column is shown. Requires 1.48d.
Array.slice(array,start,end) - Extracts a part of an array and returns it. (examples). Requires 1.46c.
Array.sort(array) - Sorts *array*, which must contain all numbers or all strings. String sorts are case-insensitive in v1.44i or later.
Array.trim(array, n) - Returns an array that contains the first *n* elements of *array*.

asin(n)
Returns the inverse sine (in radians) of *n*.

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

atan2(y, x)
Calculates the inverse tangent 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.

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 the CallJavaDemo macro and the ImpPropsplugin for examples. Note that the call() function does not work when ImageJ is running as an unsigned applet.

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. Use *run("Close")* to close non-image windows.

close(pattern)
Closes all image windows whose title matches *pattern*. *Pattern* may contain the wildcard characters "*" (matches any character sequence) or "?" (matches any single character). For example, close("Histo*") could be used to dispose all histogram windows. The front image remains in front if it still exists. *Pattern* is not case sensitive. Use close("\\Others") to close all except the front image. Requires 1.47g.

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. Uses scientific notation if 'decimalPlaces is negative. Note that d2s stands for "double to string".

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.

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, initialText) - Adds a text field to the dialog, using the specified label and initial text.
Dialog.addString(label, initialText, columns) - Adds a text field to the dialog, where *columns* specifies the field width in characters.
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.
Dialog.addSlider(label, min, max, default) - Adds a slider controlled numeric field to the dialog, using the specified label, and min, max and default values (example). Values with decimal points are used when (max-min)<=5 and min, max or default are non-integer.
Dialog.addCheckbox(label, default) - Adds a checkbox to the dialog, using the specified label and default state (true or false).
Dialog.addCheckboxGroup(rows, columns, labels, defaults) - Adds a*rows*x*columns* grid of checkboxes to the dialog, using the specified labels and default states (example).
Dialog.addRadioButtonGroup(label, items, rows, columns, default) - Adds a group of radio buttons to the dialog, where 'label' is the group label, 'items' is an array containing the button labels, 'rows' and 'columns' specify the grid size, and 'default' is the label of the default button. (example). Requires 1.47r.
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.addHelp(url) - Adds a "Help" button that opens the specified URL in the default browser. This can be used to supply a help page for this dialog or macro. With v1.46b or later, displays an HTML formatted message if 'url' starts with "<html>" (example).
Dialog.setInsets(top, left, bottom) - Overrides the default insets (margins) used for the next component added to the dialog.
Default insets:
addMessage: 0,20,0 (empty string) or 10,20,0
addCheckbox: 15,20,0 (first checkbox) or 0,20,0
addCheckboxGroup: 10,0,0
addNumericField: 5,0,3 (first field) or 0,0,3
addStringField: 5,0,5 (first field) or 0,0,5
addChoice: 5,0,5 (first field) or 0,0,5
Dialog.setLocation(x,y) - Sets the screen location where this dialog will be displayed.
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.
Dialog.getRadioButton() - Returns the selected item (a string) from the next radio button group.

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("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.

doWand(x, y, tolerance, mode)
Traces the boundary of the area with pixel values within 'tolerance' of the value of the pixel at (x,y). 'mode' can be "4-connected", "8-connected" or "Legacy". "Legacy" is for compatibility with previous versions of ImageJ; it is ignored if 'tolerance' > 0.

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. See also: Overlay.drawLine.

drawOval(x, y, width, height)
Draws the outline of an oval using the current color and line width. See also: fillOval,setColor, setLineWidth, autoUpdate and Overlay.drawEllipse.

drawRect(x, y, width, height)
Draws the outline of a rectangle using the current color and line width. See also:fillRect, setColor, setLineWidth, autoUpdate and Overlay.drawRect

drawString("text", x, y)
Draws text at the specified location. Call setFont() to specify the font. CallsetJustification() to have the text centered or right justified. Call getStringWidth() to get the width of the text in pixels. Refer to the TextDemo macro for examples and toDrawTextWithBackground to see how to draw text with a background.

drawString("text", x, y, background)
Draws text at the specified location with a filled background (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,matches.

eval(macro)
Evaluates (runs) one or more lines of macro code. An optional second argument can be used to pass a string to the macro being evaluated. See also: EvalDemo macro andrunMacro function.

eval("script", javascript)
Evaluates the JavaScript code contained in the string *javascript*, for example*eval("script","IJ.getInstance().setAlwaysOnTop(true);")*. See also: runMacro(path,arg).

eval("bsh", script)
Evaluates the BeanShell code contained in the string *script*. Requires 1.47n.

eval("python", script)
Evaluates the Python code contained in the string *script*. Requires 1.47n.

exec(string or strings)
Executes a native command and returns the output of that command as a string. Also opens Web pages in the default browser and documents in other applications (e.g., Excel). With commands with multiple arguments, each argument should be passed as a separate string. For example
exec("open", "/Users/wayne/test.jpg", "-a", "/Applications/Gimp.app");
Refer to the ExecExamples macro for examples.

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*.

Ext (Macro Extension) Functions
These are functions that have been added to the macro language by plugins using the MacroExtension interface. The Image5D_Extensions plugin, for example, adds functions that work with Image5D. The Serial Macro Extensions plugin adds functions, such as Ext.open("COM8",9600,"") and Ext.write("a"), that talk to serial devices.

F

File Functions
These functions allow you to get information about a file, read or write a text file, create a directory, or to delete, rename or move a file or directory. Note that these functions return a string, with the exception of *File.length*, *File.exists*, *File.isDirectory*,*File.rename* and *File.delete* when used in an assignment statement, for example "length=File.length(path)". The FileDemo macro demonstrates how to use these functions. See also: getDirectory and getFileList.

File.append(string, path) - Appends *string* to the end of the specified file.
File.close(f) - Closes the specified file, which must have been opened using File.open().
File.copy(path1, path2) - Copies a file. Requires 1.47j.
File.dateLastModified(path) - Returns the date and time the specified file was last modified.
File.delete(path) - Deletes the specified file or directory. With v1.41e or later, returns "1" (true) if the file or directory was successfully deleted. If the file is a directory, it must be empty. The file must be in the user's home directory, the ImageJ directory or the temp directory.
File.directory - The directory path of the last file opened using a file open dialog, a file save dialog, drag and drop, or the open(path) function.
File.exists(path) - Returns "1" (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.isDirectory(path) - Returns "1" (true) if the specified file is a directory.
File.lastModified(path) - Returns the time the specified file was last modified as the number of milliseconds since January 1, 1970.
File.length(path) - Returns the length in bytes of the specified file as a string, or as a number when used in an assignment statement, for example "length=File.length(path)".

File.makeDirectory(path) - Creates a directory.
File.name - The name of the last file opened using a file open dialog, a file save dialog, drag and drop, or the open(path) function.
File.nameWithoutExtension - The name of the last file opened with the extension removed.
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. Currently, only one file can be open at a time. For an example, refer to the SaveTextFileDemo macro.
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.
File.openAsRawString(path) - Opens a file and returns up to the first 5,000 bytes as a string. Returns all the bytes in the file if the name ends with ".txt". Refer to the First10Bytes and ZapGremlins macros for examples.
File.openAsRawString(path, count) - Opens a file and returns up to the first *count* bytes as a string.
File.openUrlAsString(url) - Opens a URL and returns the contents as a string. Returns an emptly string if the host or file cannot be found. With v1.41i and later, returns "<Error: message>" if there any error, including host or file not found.
File.openDialog(title) - Displays a file open dialog and returns the path to the file choosen by the user (example). The macro exits if the user cancels the dialog.
File.rename(path1, path2) - Renames, or moves, a file or directory. Returns "1" (true) if successful.
File.saveString(string, path) - Saves *string* as a file.
File.separator - Returns the file name separator character ("/" or "\").

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.

fillRect(x, y, width, height)
Fills the specified rectangle with the current drawing color. See also: drawRect,setColor, autoUpdate.

Fit Functions
These functions do curve fitting. The CurveFittingDemo macro demonstrates how to use them.

Fit.doFit(equation, xpoints, ypoints) - Fits the specified equation to the points defined by *xpoints*, *ypoints*. *Equation* can be either the equation name or an index. The equation names are shown in the drop down menu in the *Analyze>Tools>Curve Fitting* window. With ImageJ 1.42f or later,*equation* can be a string containing a user-defined equation (example).
Fit.doFit(equation, xpoints, ypoints, initialGuesses) - Fits the specified equation to the points defined by *xpoints*, *ypoints*, using initial parameter values contained in *initialGuesses*, an array equal in length to the number of parameters in *equation* (example).
Fit.rSquared - Returns R^2=1-SSE/SSD, where SSE is the sum of the squares of the errors and SSD is the sum of the squares of the deviations about the mean.
Fit.p(index) - Returns the value of the specified parameter.
Fit.nParams - Returns the number of parameters.
Fit.f(x) - Returns the y value at *x* (example).

Fit.nEquations - Returns the number of equations.
Fit.getEquation(index, name, formula) - Gets the name and formula of the specified equation.
Fit.plot - Plots the current curve fit.
Fit.logResults - Causes doFit() to write a description of the curve fitting results to the Log window.
Fit.showDialog - Causes doFit() to display the simplex settings dialog.

floodFill(x, y)
Fills, with the foreground color, pixels that are connected to, and the same color as, the pixel at *(x, y)*. 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.

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. UsetoScaled(x,y) to scale the coordinates. For examples, see the GetCursorLocDemo and the GetCursorLocDemoTool macros.

getDateAndTime(year, month, dayOfWeek, dayOfMonth, hour, minute, second, msec)
Returns the current date and time. Note that 'month' and 'dayOfWeek' are zero-based indexes. For an example, refer to the GetDateAndTime macro. See also: getTime.

getDimensions(width, height, channels, slices, frames)
Returns the dimensions of the current image.

getDirectory(string)
Displays a "choose directory" dialog and returns the selected directory, or returns the path to a specified directory, such as "plugins", "home", etc. The returned path ends with a file separator, either "\" (Windows) or "/". Returns an empty string if the specified directory is not found or aborts

the macro if the user cancels the "choose directory" dialog box. For examples, see the GetDirectoryDemo andListFilesRecursively macros. See also: getFileList and the File functions.

getDirectory("Choose a Directory") - Displays a file open dialog, using the argument as a title, and returns the path to the directory selected by the user.
getDirectory("plugins") - Returns the path to the plugins directory.
getDirectory("macros") - Returns the path to the macros directory.
getDirectory("luts") - Returns the path to the luts directory.
getDirectory("image") - Returns the path to the directory that the active image was loaded from.
getDirectory("imagej") - Returns the path to the ImageJ directory.
getDirectory("startup") - Returns the path to the directory that ImageJ was launched from.
getDirectory("home") - Returns the path to users home directory.
getDirectory("temp") - Returns the path to the temporary directory (/tmp on Linux and Mac OS X).

getDisplayedArea(x, y, width, height)
Returns the pixel coordinates of the actual displayed area of the image canvas. For an example, see the Pixel Sampler Tool.

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 theListFilesRecursively macro.

getFontList()
Returns an array containing the names of available system fonts (example).

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

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("window.contents"). For an example, see the ListDicomTags macros. See also: getMetadata.

getInfo("command.name")
Returns the name of the most recently invoked command. The names of commands invoked using keyboard shortcuts are preceded by "^" (example).

getInfo(DICOM_TAG)
Returns the value of a DICOM tag in the form "xxxx,xxxx", e.g. getInfo("0008,0060"). Returns an empty string if the current image is not a DICOM or if the tag was not found.

getInfo("font.name")
Returns the name of the current font.

getInfo("image.description")
Returns the TIFF image description tag, or an empty string if this is not a TIFF image or the image description is not available.

getInfo("image.directory")
Returns the directory that the current image was loaded from, or an empty string if the directory is not available.

getInfo("image.filename")
Returns the name of the file that the current image was loaded from, or an empty string if the file name is not available.

getInfo("image.subtitle")
Returns the subtitle of the current image. This is the line of information displayed above the image and below the title bar.

getInfo("log")
Returns the contents of the Log window, or "" if the Log window is not open.

getInfo("macro.filepath")
Returns the filepath of the most recently loaded macro or script.

getInfo("micrometer.abbreviation")
Returns "µm", the abbreviation for micrometer.

getInfo("os.name")
Returns the OS name ("Mac OS X", "Linux" or "Windows").

getInfo("overlay")
Returns information about the current image's overlay.

getInfo("selection.name")
Returns the name of the current selection, or "" if there is no selection or the selection does not have a name. The argument can also be "roi.name".

getInfo("selection.color")
Returns the color of the current selection. Requires 1.47a.

getInfo("slice.label")
Return the label of the current stack slice. This is the string that appears in parentheses in the subtitle, the line of information above the image. Returns an empty string if the current image is not a stack or the current slice does not have a label.

getInfo("threshold.method")
Returns the current thresholding method ("IsoData", "Otsu", etc).

getInfo("threshold.mode")
Returns the current thresholding mode ("Red","B&W" or"Over/Under").

getInfo("window.contents")
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. Replaces the getInfo() function.

getInfo("window.type")
Returns the type ("Image", "Text", "ResultsTable", "Editor", "Plot", "Histogram", etc.) of the front window. Requires 1.48g.

getInfo(key)
Returns the Java property associated with the specified key (e.g., "java.version", "os.name", "user.home", "user.dir", etc.). Returns an empty string if there is no value associated with the key. See also: getList("java.properties").

getLine(x1, y1, x2, y2, lineWidth)
Returns the starting coordinates, ending coordinates and width of the current straight line selection. The coordinates and line width are in pixels. Sets x1 = -1 if there is no line selection. Refer to the GetLineDemo macro for an example. See also: makeLine.

getList("image.titles")
Returns a list (array) of image window titles. For an example, see theDisplayWindowTitles macro.

getList("window.titles")
Returns a list (array) of non-image window titles. For an example, see theDisplayWindowTitles macro.

getList("java.properties")
Returns a list (array) of Java property keys. For an example, see theDisplayJavaProperties macro. See also: getInfo(key).

getList("threshold.methods")
Returns a list of the available automatic thresholding methods (example).

getList("LUTs")
Returns, as an array, a list of the LUTs in the *Image>Lookup Tables* menu (example). Requires 1.47r.

getLocationAndSize(x, y, width, height)
Returns the location and size, in screen coordinates, of the active image window. UsegetWidth and getHeight to get the width and height, in image coordinates, of the active image. See also: setLocation,

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("Info")
Returns the metadata (a string) from the "Info" property of the current image. With DICOM images, this is the information (tags) in the DICOM header. See also:setMetadata.

getMetadata("Label")
Returns the current slice label. The first line of the this label (up to 60 characters) is display as part of the image subtitle. With DICOM stacks, returns the metadata from the DICOM header. See also: setMetadata.

getMinAndMax(min, max)
Returns the minimum and maximum displayed pixel values (display range). See theDisplayRangeMacros 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. See also:Plot.getValues().

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: calibrateand List.setMeasurements

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.) Omit the second argument and the row defaults to nResults-1 (the last row in the results table). See also: nResults, setResult, isNaN, getResultLabel.

getResultString("Column", row)
Returns a string from the ImageJ results table or "null" if the specified column is not found. The first argument specifies a column in the table. The second specifies the row, where 0<=*row*<nResults. Requires 1.47o.

getResultLabel(row)
Returns the label of the specified row in the results table, or an empty string if *Display Label* is not checked in *Analyze>Set Measurements*.

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 and setSelectionLocation.

getSelectionCoordinates(xpoints, ypoints)
Returns two arrays containing the X and Y coordinates, in pixels, 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 image, an integer between 1 andnSlices. Returns 1 if the active image is not a stack. See also: setSlice,Stack.getPosition.

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 and List.setMeasurements

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.

getStringWidth(string)
Returns the width in pixels of the specified string. See also: setFont, drawString.

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.

getValue("color.foreground")
Returns the current foreground color as a value that can be passed to thesetColor(value) function. The value returned is the pixel value used by the *Edit>Fill*command and by drawing tools.

getValue("color.background")
Returns the current background color as a value that can be passed to thesetColor(value) function. The value returned is the pixel value used by the *Edit>Clear*command.

getValue("rgb.foreground")
Returns the current foregound color as an RGB pixel value (example). Requires 1.47g.

getValue("rgb.background")
Returns the current backgound color as an RGB pixel value. Requires 1.47g.

getValue("font.size")
Returns the size, in points, of the current font.

getValue("font.height")
Returns the height, in pixels, of the current font.

getValue("selection.width")
Returns the stroke width of the current selection. Requires 1.47a.

getValue("results.count")
Returns the number of lines in the current results table. Unlike nResults, works with tables that are not named "Results". Requires 1.49t.

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"). 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

IJ Functions
These functions provide access to miscellaneous methods in ImageJ's IJ class.

IJ.deleteRows(index1, index2) - Deletes rows *index1* through *index2* in the results table.
IJ.getToolName() - Returns the name of the currently selected tool. See also: setTool.
IJ.freeMemory() - Returns the memory status string (e.g., "2971K of 658MB (<1%)") that is displayed when the users clicks in the ImageJ status bar.
IJ.currentMemory() - Returns, as a string, the amount of memory in bytes currently used by ImageJ.
IJ.log(string) - Displays *string* in the Log window.
IJ.maxMemory() - Returns, as a string, the amount of memory in bytes available to ImageJ. This value (the Java heap size) is set in the*Edit>Options>Memory & Threads* dialog box.
IJ.pad(n, length) - Pads 'n' with leading zeros and returns the result (example).
IJ.redirectErrorMessages() - Causes next image opening error to be redirected to the Log window and prevents the macro from being aborted (example).
IJ.renameResults(name) - Changes the title of the "Results" table to the string *name*.
IJ.renameResults(oldName,newName) - Changes the title of a results table from *oldName* to *newName*. Requires 1.46g.

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 theImageCalculatorDemo macros for examples.

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

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

is("animated")
Returns *true* if the current image is an animated stack.

is("applet")
Returns *true* if ImageJ is running as an applet.

is("Batch Mode")
Returns *true* if the macro interpreter is running in batch mode.

is("binary")
Returns *true* if the current image is binary (8-bit with only 0 and 255 values).

is("Caps Lock Set")
Returns *true* if the caps lock key is set. Always return *false* on some platforms.

is("changes")
Returns *true* if the current image's 'changes' flag is set.

is("composite")
Returns *true* if the current image is a a multi-channel stack that uses the CompositeImage class.

is("global scale")
Returns *true* if there is global spatial calibration.

is("grayscale")
Returns *true* if the current image is grayscale, or an RGB image with identical R, G and B channels. Requires 1.46i.

is("Inverting LUT")
Returns *true* if the current image is using an inverting (monotonically decreasing) lookup table.

is("locked")
Returns *true* if the current image is locked.

is("Virtual Stack")
Returns *true* if the current image is a virtual stack.

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. Comparison with a NaN always returns *false* so "if (n!=n)" is equilvalent to (isNaN(n))". Note that the numeric constant NaN is predefined in the macro language.
The NaNs macro demonstrates how to remove NaNs from an image.

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*) . See also: Overlay.lineTo.

List Functions
These functions work with a list of key/value pairs. The ListDemo macro demonstrates how to use them.

List.set(key, value) - Adds a key/value pair to the list.
List.get(key) - Returns the string value associated with *key*, or an empty string if the key is not found.
List.getValue(key) - When used in an assignment statement, returns the value associated with *key* as a number. Aborts the macro if the value is not a number or the key is not found.
List.size - Returns the size of the list.
List.clear() - Resets the list.
List.setList(list) - Loads the key/value pairs in the string *list*.
List.getList - Returns the list as a string.
List.setMeasurements - Measures the current image or selection and loads the resulting keys (Results table column headings) and values into the list. All parameters listed in the *Analyze>Set Measurements* dialog box are measured, including those that are unchecked. Use List.getValue() in an assignment statement to retrieve the values. See the DrawEllipse macro for an example.
List.setCommands - Loads the ImageJ menu commands (as keys) and the plugins that implement them (as values).

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

M

makeArrow(x1, y1, x2, y2, style)
Creates an arrow selection, where 'style' is a string containing "filled", "notched", "open", "headless" or "bar", plus the optionial modifiers "outline", "double", "small", "medium" and "large" (example). See also: Roi.setStrokeWidth andRoi.setStrokeColor. Requires 1.49a.

makeEllipse(x1, y1, x2, y2, aspectRatio)
Creates an elliptical selection, where *x1,y1,x2,y2* specify the major axis of the ellipse and *aspectRatio* (<=1.0) is the ratio of the lengths of minor and major axis.

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. You can create segmented line selections by specifying more than two coordinate pairs, for example*makeLine(25,34,44,19,69,30,71,56)*.

makeLine(x1, y1, x2, y2, lineWidth)
Creates a straight line selection with the specified width. See also: getLine.

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

makePoint(x, y)
Creates a point selection at the specified location. Create a multi-point selection by using *makeSelection("point",xpoints,ypoints)*. Use *setKeyDown("shift"); makePoint(x, y);* to add a point to an existing point selection.

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.

makeRectangle(x, y, width, height, arcSize)
Creates a rounded rectangular selection using the specified corner arc size.

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", or 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.

makeText(string, x, y)
Creates a text selection at the specified coordinates. The selection will use the font and size specified by the last call to setFont(). The CreateOverlay macro provides an example.

matches(string, regex)
Returns *true* if *string* matches the specified regular expression. See also: startsWith,endsWith, indexOf, replace.

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. Precede with *call("ij.gui.ImageWindow.setNextLocation", x, y)* to set the location of the new image. *Width* and *height* specify the width and height of the image in pixels. *Depth* specifies the number of stack slices.

newMenu(macroName, stringArray)
Defines a menu that will be added to the toolbar when the menu tool named*macroName* is installed. Menu tools are macros with names ending in "Menu Tool".*StringArray* is an array containing the menu commands. Returns a copy of *stringArray*. For an example, refer to the Toolbar Menus toolset.

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

nResults
Returns the current measurement counter value. The parentheses "()" are optional. See also: getValue("results.count").

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

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. With 1.41k or later, opens images specified by a URL, for example*open("http://imagej.nih.gov/ij/images/clown.gif").* With 1.49v or later, opens a folder of images as a stack. Use open("path/to/folder","virtual") to open a folder of images as a virtual stack.

open(path, n)
Opens the *n*th image in the TIFF stack specified by *path*. For example, the first image in the stack is opened if *n*=1 and the tenth is opened if *n*=10.

Overlay Functions
Use these functions to create and manage non-destructive graphic overlays. For an exmple, refer to the OverlayPolygons macro. See also: setColor, setLineWidth andsetFont.

Overlay.moveTo(x, y)
Sets the current drawing location.
Overlay.lineTo(x, y)
Draws a line from the current location to (*x,y*) .
Overlay.drawLine(x1, y1, x2, y2)
Draws a line between (*x1,y1*) and (*x2,y2*)).
Overlay.add
Adds the drawing created by Overlay.lineTo(), Overlay.drawLine(), etc. to the overlay without updating the display.
Overlay.setPosition(n)
Sets the stack position (slice number) of the last item added to the overlay (example).
Overlay.setPosition(c, z, t)
Sets the hyperstack position (channel, slice, frame) of the last item added to the overlay. Requires 1.47n.
Overlay.drawRect(x, y, width, height)
Draws a rectangle, where (*x,y*) specifies the upper left corner.
Overlay.drawEllipse(x, y, width, height)
Draws an ellipse, where (*x,y*) specifies the upper left corner of the bounding rectangle.
Overlay.drawString("text", x, y)
Draws text at the specified location and adds it to the overlay. UsesetFont() to specify the font and setColor to set specify the color (example).
Overlay.drawString("text", x, y, angle)
Draws text at the specified location and angle and adds it to the overlay (example). Requires 1.48q.
Overlay.show
Displays the current overlay.
Overlay.hide
Hides the current overlay.
Overlay.hidden
Returns *true* if the active image has an overlay and it is hidden. Requires 1.46a.
Overlay.remove
Removes the current overlay.
Overlay.clear
Resets the overlay without updating the display. Requires 1.48r.
Overlay.size
Returns the size (selection count) of the current overlay. Returns zero if the image does not have an

overlay.
Overlay.addSelection
Adds the current selection to the overlay. Requires v1.47i.
Overlay.addSelection(strokeColor)
Sets the stroke color ("red", "green", "ff8800", etc.) of the current selection and adds it to the overlay. Requires v1.47j.
Overlay.addSelection(strokeColor, strokeWidth)
Sets the stroke color ("blue", "yellow", "ffaa77" etc.) and stroke width of the current selection and adds it to the overlay. Requires v1.47j.
Overlay.addSelection("", 0, fillColor)
Sets the fill color ("red", "green", etc.) of the current selection and adds it to the overlay. Requires v1.47j.
Overlay.activateSelection(index)
Activates the specified overlay selection. Requires v1.47i.
Overlay.moveSelection(index, x, y)
Moves the specified selection to the specified location. Requires v1.48c.
Overlay.removeSelection(index)
Removes the specified selection from the overlay.
Overlay.copy
Copies the overlay on the current image to the overlay clipboard.
Overlay.paste
Copies the overlay on the overlay clipboard to the current image.
Overlay.drawLabels(boolean)
Enables/disables overlay labels. Requires 1.47n.

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.

PI
Returns the constant π (3.14159265), the ratio of the circumference to the diameter of a circle.

Plot Functions
Use these functions to generate and display plots. For examples, check out theExample Plot, More Example Plots, AdvancedPlots, Semi-log Plot and Arrow Plotmacros.

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.
Plot.add(type, 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 (*type*) can be "line", "circles", "boxes", "triangles", "crosses", "dots", "x", "connected" (requires 1.49t), "error bars" (in y direction) or "xerror bars". In 1.49t or later, error bars apply to the last dataset provided by *Plot.create*or *Plot.add*.
Plot.drawVectors(xStarts, yStarts, xEnds, yEnds)
Draws arrows from the starting to ending coordinates contained in the arrays.
Plot.drawLine(x1, y1, x2, y2)
Draws a line between *x1,y1* and *x2,y2*, using the coordinate system defined by *Plot.setLimits()*.
Plot.drawNormalizedLine(x1, y1, x2, y2)
Draws a line using a normalized 0-1, 0-1 coordinate system, with (0,0) at the top left and (1,1) at the lower right corner.
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.setLimits(xMin, xMax, yMin, yMax)**
Sets the range of the x-axis and y-axis of plots. With version 1.50g and later, when 'NaN' is used as a limit, the range is calculated automatically from the plots that have been added so far.
Plot.getLimits(xMin, xMax, yMin, yMax)
Returns the current axis limits. Note that min>max if the axis is reversed. Requires 1.49t.
Plot.setLimitsToFit()
Sets the range of the x and y axes to fit all data. Requires 1.49t.
Plot.setColor(color)
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", "yellow", or a hex value like "#ff00ff". Note that the curve specified in *Plot.create()* is drawn last.
Plot.setColor(color1, color2)
This is a two argument version of Plot.setColor, where the second argument is used for filling symbols or as the line color for connected points. Requires 1.49t.
Plot.setBackgroundColor(color)
Sets the background color of the plot frame (example). Requires 1.49h.
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.setJustification("center")
Specifies the justification used by *Plot.addText()*. The argument can be "left", "right" or "center". The default is "left".
Plot.setLegend("label1\tlabel2...", "options")
Creates a legend for each of the data sets added with *Plot.create* and*Plot.add*. In the first argument,

the labels for the data sets should be separated with tab or newline characters. The optional second argument can contain the legend position ("top-left", "top-right", "bottom-left", "bottom-right"; default is automatic positioning), "bottom-to-top" for reversed sequence of the labels, and "transparent" to make the legend background transparent. Requires 1.49t.

Plot.setFrameSize(width, height)

Sets the plot frame size in pixels, overriding the default size defined in the*Edit>Options>Profile Plot Options* dialog box.

Plot.setLogScaleX(boolean)

Sets the x axis scale to Logarithmic, or back to linear if the optional boolean argument is false. In versions up to 1.49s, it must be called immediately after *Plot.create* and before *Plot.setLimits*. See theLogLogPlot macro for an example.

Plot.setLogScaleY(boolean)

Sets the y axis scale to Logarithmic, or back to linear if the optional boolean argument is false.

Plot.setXYLabels("x Label", "y Label")

Sets the axis labels. Requires 1.49t.

Plot.setFontSize(size, "options")

Sets the default font size in the plot (otherwise specified in *Profile Plot Options*), used e.g. for axes labels. Can be also called prior to*Plot.addText*. The optional second argument can include "bold" and/or "italic". Requires 1.49t.

Plot.setAxisLabelSize(size, "options")

Sets the fort size of the axis labels. The optional second argument can include "bold" and/or "italic". Requires 1.49t.**Plot.setFormatFlags("11001100001111")**

Controls whether to draw axes labels, grid lines and ticks (major/minor/ticks for log axes). Use the macro recorder and *More>>Axis Options* in any plot window to determine the binary code. Requires 1.49t.

Plot.useTemplate("plot name" or id)

Transfers the formatting of an open plot window to the current plot. Requires 1.49t.

Plot.makeHighResolution(factor)

Creates a high-resolution image of the plot enlarged by *factor*. Add the second argument "disable" to avoid antialiased text. Requires 1.49t.**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.

Plot.getValues(xpoints, ypoints)

Returns the values displayed by clicking on "List" in a plot or histogram window (example).

Plot.showValues()

Displays the plot values in the Results window. Must be preceded by*Plot.show*.

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. The print() function accepts multiple arguments. For example, you can use*print(x,y,width, height)* instead of *print(x+" "+y+" "+width+" "+height)*. If the first argument is a file handle returned by File.open(path), then the second is saved in the refered file (see SaveTextFileDemo).

Numeric expressions are automatically converted to strings using four decimal places, or use the d2s function to specify the decimal places. For example, *print(2/3)* outputs "0.6667" but *print(d2s(2/3,1))* outputs "0.7".

The print() function accepts commands such as *"\\Clear"*, *"\\Update:<text>"* and*"\\Update<n>:<text>"* (for n<26) that clear the "Log" window and update its contents. For example, *print("\\Clear")* erases the Log window, *print("\\Update:new text")*replaces the last line with "new text" and *print("\\Update8:new 8th line")* replaces the 8th line with "new 8th line". Refer to the LogWindowTricks macro for an example.

The second argument to print(arg1, arg2) is appended to a text window or table if the first argument is a window title in brackets, for example *print("[My Window]", "Hello, world")*. With text windows, newline characters ("\n") are not automatically appended and text that starts with *"\\Update:"* replaces the entire contents of the window. Refer to the PrintToTextWindow, Clock and ProgressBar macros for examples.

The second argument to print(arg1, arg2) is appended to a table (e.g., ResultsTable) if the first argument is the title of the table in brackets. Use the *Plugins>New* command to open a blank table. Any command that can be sent to the "Log" window (*"\\Clear"*,*"\\Update:<text>"* , etc.) can also be sent to a table. Refer to the SineCosineTable2 andTableTricks macros for examples.

R

random
Returns a random number between 0 and 1.

random("seed", seed)
Sets the seed (a whole number) used by the *random()* function.

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. If *old* or *new* are longer than one character, each substring of *string* that matches the regular expression *old* is replaced with *new*. When doing a simple string replacement, and *old* contains regular expression metacharacters ('.', '[', ']', '^', '$', etc.), you must escape them with a "\\". For example, to replace "[xx]" with "yy", use *string=replace(string,"\\[xx\\]","yy")*. See also:matches.

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
With 16-bit and 32-bit images, resets the minimum and maximum displayed pixel values (display range) to be the same as the current image's minimum and maximum pixel values. With 8-bit images, sets the display range to 0-255. With RGB images, does nothing. See the DisplayRangeMacros for examples.

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

restoreSettings
Restores *Edit>Options* submenu settings saved by the saveSettings function.

ROI Functions
Use these functions to get information about the current selection or to get or set selection properties. Refer to the RoiFunctionsDemo macro for examples. These functions require ImageJ 1.48h or later.

Roi.contains(x, y)
Returns "1" if the point *x,y* is inside the current selection or "0" if it is not. Aborts the macro if there is no selection. Requires 1.49h. See also:selectionContains.
Roi.getBounds(x, y, width, height)
Returns the location and size of the selection's bounding rectangle.
Roi.getCoordinates(xpoints, ypoints)
Returns, as two arrays, the x and y coordinates that define this selection.
Roi.getDefaultColor
Returns the current default selection color.
Roi.getStrokeColor
Returns the selection stroke color.
Roi.getFillColor
Returns the selection fill color.
Roi.getName
Returns the selection name or an empty string if the selection does not have a name.
Roi.getProperty(key)
Returns the value (a string) associated with the specified key or an empty string if the key is not found.
Roi.setProperty(key, value)
Adds the specified key and value pair to the selection properties. Assumes a value of "1" (true) if there is only one argument.
Roi.getProperties
Returns all selection properties or an empty string if the selection does not have properties.
Roi.getSplineAnchors(x, y)
Returns the x and y coordinates of the anchor points of a spline fitted selection (example).
Roi.setPolygonSplineAnchors(x, y)
Creates or updates a spline fitted polygon selection (example).
Roi.setPolylineSplineAnchors(x, y)
Creates or updates a spline fitted polyline selection (example).
Roi.getType
Returns, as a string, the type of the current selection.

Roi.move(x, y)
Moves the selection to the specified location.
Roi.setStrokeColor(color)
Sets the selection stroke color ("red", "5500ff00". etc.).
Roi.setFillColor(color)
Sets the selection fill color ("red", "5500ff00". etc.).
Roi.setName(name)
Sets the selection name.
Roi.setStrokeWidth(width)
Sets the selection stroke width.

ROI Manager Functions
These function run ROI Manager commands. The ROI Manager is opened if it is not already open. Use *roiManager("reset")* to delete all ROIs on the list. UsesetOption("Show All", boolean) to enable/disable "Show All" mode. For examples, refer to the RoiManagerMacros, ROI Manager Stack Demo and RoiManagerSpeedTestmacros.
roiManager("and")
Uses the conjunction operator on the selected ROIs, or all ROIs if none are selected, to create a composite selection.

roiManager("add")
Adds the current selection to the ROI Manager.

roiManager("add & draw")
Outlines the current selection and adds it to the ROI Manager.

roiManager("combine")
Uses the union operator on the selected ROIs to create a composite selection. Combines all ROIs if none are selected.

roiManager("count")
Returns the number of ROIs in the ROI Manager list.

roiManager("delete")
Deletes the selected ROIs from the list, or deletes all ROIs if none are selected.

roiManager("deselect")
Deselects all ROIs in the list. When ROIs are deselected, subsequent ROI Manager commands are applied to all ROIs.

roiManager("draw")
Draws the selected ROIs, or all ROIs if none are selected, using the equivalent of the *Edit>Draw* command.

roiManager("fill")
Fills the selected ROIs, or all ROIs if none are selected, using the equivalent of the *Edit>Fill* command.

roiManager("index")
Returns the index of the currently selected ROI on the list, or -1 if the list is empty or no ROIs are selected. Returns the index of the first selected ROI if more than one is selected

roiManager("measure")
Measures the selected ROIs, or if none is selected, all ROIs on the list.

roiManager("multi measure")
Measures all the ROIs on all slices in the stack, creating a Results Table with one row per slice.

roiManager("multi-measure append")
Measures all the ROIs on all slices in the stack, appending the measurements to the Results Table, with one row per slice.

roiManager("multi-measure one")
Measures all the ROIs on all slices in the stack, creating a Results Table with one row per measurement.

roiManager("multi plot")
Plots the selected ROIs, or all ROIs if none are selected, on a single graph.

roiManager("open", file-path)
Opens a .roi file and adds it to the list or opens a ZIP archive (.zip file) and adds all the selections contained in it to the list.

roiManager("remove slice info")
Removes the information in the ROI names that associates them with particular stack slices.

roiManager("rename", name)
Renames the selected ROI. You can get the name of an ROI on the list using *call("ij.plugin.frame.RoiManager.getName", index)*.

roiManager("reset")
Deletes all ROIs on the list.

roiManager("save, file-path)
Saves all the ROIs on the list in a ZIP archive.

roiManager("save selected", file-path)
Saves the selected ROI as a .roi file.

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")*. Note that macros that use this function sometimes run orders of magnitude faster in batch mode. Use *roiManager("deselect")* to deselect all items on the list. For an example, refer to the ROI Manager Stack Demo macro.

roiManager("select", indexes)
Selects multiple items in the ROI Manager list, where *indexes* is an array of integers, each of which must be greater than or equal to 0 and less than the value returned by *roiManager("count")*. The selected ROIs are not highlighted in the ROI Manager list and are no longer selected after the next ROI Manager command is executed.

roiManager("show all")
Displays all the ROIs as an overlay.

roiManager("show all with labels")
Displays all the ROIs as an overlay, with labels.

roiManager("show all without labels")
Displays all the ROIs as an overlay, without labels.

roiManager("show none")
Removes the ROI Manager overlay.

roiManager("sort")
Sorts the ROI list in alphanumeric order.

roiManager("split")
Splits the current selection (it must be a composite selection) into its component parts and adds them to the ROI Manager.

roiManager("update")
Replaces the selected ROI on the list with the current selection.

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. With ImageJ 1.43 and later, variables can be passed without using string concatenation by adding "&" to the variable name. For examples, see the ArgumentPassingDemo macro.

runMacro(name)
Runs the specified macro or script, which is assumed to be in the Image macros folder. A full file path may also be used. Returns any string argument returned by the macro or the last expression evaluated in the script. For an example, see the CalculateMeanmacro. See also: eval.

runMacro(name, arg)
Runs the specified macro or script, which is assumed to be in the macros folder, or use a full file path. The string argument 'arg' can be retrieved by the macro or script using the getArgument()

function. Returns the string argument returned by the macro or the last expression evaluated in the script. See also: getArgument.

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", ".fits", ".png", ".pgm", ".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", "fits", "png", "pgm", "text image", "lut", "selection", "results", "xy Coordinates" or "text". Use *saveAs(format)* to have a "Save As" dialog displayed.

saveSettings()
Saves most *Edit>Options* submenu settings so they can be restored later by callingrestoreSettings.

screenHeight
Returns the screen height in pixels. See also: getLocationAndSize, setLocation.

screenWidth
Returns the screen width in pixels.

selectionContains(x, y)
Returns *true* if the point *x,y* is inside the current selection. Aborts the macro if there is no selection.

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. See also: setSelectionName andselectionType.

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 theShowImageInfo 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. The *id* can also be an image title (a string).

selectWindow("name")
Activates the window with the title "name".

setAutoThreshold()
Uses the "Default" method to determine the threshold. It may select dark or bright areas as thresholded, as was the case with the *Image>Adjust>Threshold* "Auto" option in ImageJ 1.42o and earlier. See also: setThreshold, getThreshold, resetThreshold.

setAutoThreshold(method)
Uses the specified method to set the threshold levels of the current image. Use the getList("threshold.methods") function to get a list of the available methods. Concatenate " dark" to the method name if the image has a dark background. For an example, see the AutoThresholdingDemo macro.

setBackgroundColor(r, g, b)
Sets the background color, where *r*, *g*, and *b* are >= 0 and <= 255. See also:setForegroundColor.

setBackgroundColor(rgb)
Sets the background color, where *rgb* is an RGB pixel value. See also:getValue("rgb.background"). Requires 1.47g.

setBatchMode(arg)
Controls whether images are visible or hidden during macro execution. If *arg* is 'true', the interpreter enters batch mode and newly opened images are not displayed. If *arg* is 'false', exits batch mode and disposes of all hidden images except for the active image, which is displayed in a window. The interpreter also exits batch mode when the macro terminates, disposing of all hidden images.

With ImageJ 1.48h or later, you can use 'show' and 'hide' arguments to individually show or hide images. By not displaying and updating images, batch mode macros run up to 20 times faster. Examples: BatchModeTest, BatchMeasure, BatchSetScale andReplaceRedWithMagenta.

setBatchMode("exit and display")
Exits batch mode and displays all hidden images.

setBatchMode("show")
Displays the active hidden image, while batch mode remains in same state. Requires 1.48h.

setBatchMode("hide")
Enters (or remains in) batch mode and hides the active image Requires 1.48h.

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. SetColor() is faster than setForegroundColor(), and it does not change the system wide foreground color or repaint the color picker tool icon, but it cannot be used to specify the color used by commands called from macros, for example *run("Fill")*.

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")*.

setColor(string)
Sets the drawing color, where 'string' can be "black", "blue", "cyan", "darkGray", "gray", "green", "lightGray", "magenta", "orange", "pink", "red", "white", "yellow", or a hex value like "#ff0000".

setFont(name, size[, style])
Sets the font used by the drawString function. The first argument is the font name. It should be "SansSerif", "Serif" or "Monospaced". The second is the point size. The optional third argument is a string containing "bold" or "italic", or both. The third argument can also contain the keyword "antialiased". For examples, run the TextDemomacro.

setFont("user")
Sets the font to the one defined in the *Edit>Options>Fonts* window. See also:getInfo("font.name"), getValue("font.size") and getValue("font.height").

setForegroundColor(r, g, b)
Sets the foreground color, where *r*, *g*, and *b* are >= 0 and <= 255. See also: setColorand setBackgroundColor.

setForegroundColor(rgb)
Sets the foreground color, where *rgb* is an RGB pixel value. See also:getValue("rgb.foreground"). Requires 1.47g.

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. Call *setKeyDown("esc")* to abort the currently running macro or plugin. For examples, see theCompositeSelections, DoWandDemo and AbortMacroActionTool 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 window to a new location. Use*call("ij.gui.ImageWindow.setNextLocation", x, y)* to set the location of the next opened window. See also: getLocationAndSize, screenWidth, screenHeight.

setLocation(x, y, width, height)
Moves and resizes the active image window. The new location of the top-left corner is specified by *x* and *y*, and the new size by *width* and *height*.

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("Info", string)
Assigns the metadata in *string* to the "Info" image property of the current image. This metadata is displayed by *Image>Show Info* and saved as part of the TIFF header. See also: getMetadata.

setMetadata("Label", string)
Sets *string* as the label of the current image or stack slice. The first 60 characters, or up to the first newline, of the label are displayed as part of the image subtitle. The labels are saved as part of the TIFF header. See also: getMetadata.

setMinAndMax(min, max)
Sets the minimum and maximum displayed pixel values (display range). See theDisplayRangeMacros for examples.

setMinAndMax(min, max, channels)
Sets the display range of specified channels in an RGB image, where 4=red, 2=green, 1=blue, 6=red+green, etc. Note that the pixel data is altered since RGB images, unlikecomposite color images, do not have a LUT for each channel.

setOption(option, boolean)
Enables or disables an ImageJ option, where *option* is one of the following options and*boolean* is either *true* or *false*.

"AutoContrast" enables/disables the *Edit>Options>Appearance* "Auto contrast stacks" option. You can also have newly displayed stack slices contrast enhanced by holding the shift key down when navigating stacks.

"Bicubic" provides a way to force commands like*Edit>Selection>Straighten*, that normally use bilinear interpolation, to use bicubic interpolation.

"BlackBackground" enables/disables the *Process>Binary>Options* "Black background" option.

"Changes" sets/resets the 'changes' flag of the current image. Set this option *false* to avoid "Save Changes?" dialog boxes when closing images.

"DebugMode" enables/disables the ImageJ debug mode. ImageJ displays information, such as TIFF tag values, when it is in debug mode.

"DisablePopupMenu" enables/disables the the menu displayed when you right click on an image.

"DisableUndo" enables/disables the *Edit>Undo* command. Note that a*setOption("DisableUndo",true)* call without a correspondingsetOption("DisableUndo",false) will cause *Edit>Undo* to not work as expected until ImageJ is restarted.

"Display label", *"Limit to threshold"*, *"Area"*, *"Mean"* and *"Std"*, *"Stack position"* and *"Add to overlay"* (requires 1.48r), enable/disable the corresponding *Analyze>Set Measurements* options.

"ExpandableArrays" enables/disables support for auto-expanding arrays (example). Note that macros that use auto-expanding arrays will not be compatible with Image 2.0.

"JFileChooser" enables/disables use of the Java JFileChooser to open and save files instead of the native OS file chooser. Requires v1.47d.

"Loop" enables/disables the *Image>Stacks>Tools>Animation Options*"Loop back and forth" option.

"OpenUsingPlugins" controls whether standard file types (TIFF, JPEG, GIF, etc.) are opened by external plugins or by ImageJ (example).

"QueueMacros" controls whether macros invoked using keyboard shortcuts run sequentially on the event dispatch thread (EDT) or in separate, possibly concurrent, threads (example). In "QueueMacros" mode, screen updates, which also run on the EDT, are delayed until the macro finishes.

"Show All" enables/disables the the "Show All" mode in the ROI Manager.

"ShowAngle" determines whether or not the "Angle" value is displayed in the Results window when measuring straight line lengths. Requires 1.49c.

"ShowMin" determines whether or not the "Min" value is displayed in the Results window when "Min & Max Gray Value" is enabled in the*Analyze>Set Measurements* dialog box.

"ShowRowNumbers" enables/disables display of Results table row numbers (example).

setPasteMode(mode)
Sets the transfer mode used by the *Edit>Paste* command, where *mode* is "Copy", "Blend", "Average", "Difference", "Transparent-white", "Transparent-zero", "AND", "OR", "XOR", "Add", "Subtract", "Multiply", "Divide, "Min" or "Max". TheGetCurrentPasteMode macro demonstrates how to get the current paste mode. In ImageJ 1.42 and later, the paste mode is saved and restored by the saveSettings andrestoreSettings.

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. With v1.47o or later, it can be a string. Call*setResult("Label", row, string)* to set the row label. Call *updateResults()* to display the updated table in the "Results" window. For examples, see the SineCosineTable andConvexitySolidarity 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, ExtractRGBChannels andRGB_Histogram macros.

setSelectionLocation(x, y)
Moves the current selection to *(x,y)*, where *x* and *y* are the pixel coordinates of the upper left corner of the selection's bounding rectangle. TheRoiManagerMoveSelections macro uses this function to move all the ROI Manager selections a specified distance. See also: getSelectionBounds.

setSelectionName(name)
Sets the name of the current selection to the specified name. Aborts the macro if there is no selection. 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.

setThreshold(lower, upper)
Sets the lower and upper threshold levels. There is an optional third argument that can be "red", "black & white", "over/under", "no color" or "raw". With density calibrated images, *lower* and *upper* must be calibrated values unless the optional third argument is "raw". See also: setAutoThreshold, getThreshold, resetThreshold.

setTool(name)
Switches to the specified tool, where *name* is "rectangle", "roundrect", "elliptical", "brush", "polygon", "freehand", "line", "polyline", "freeline", "arrow", "angle", "point", "multipoint", "wand", "text", "zoom", "hand" or "dropper". Refer to the SetToolDemo,ToolShortcuts or ToolSwitcher, macros for examples. See also: IJ.getToolName.

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=unused, 11=zoom, 12=hand, 13=dropper, 14=angle, 15...21=custom tools. 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 theDrawingTools tool set.

setVoxelSize(width, height, depth, unit)
Defines the voxel dimensions and unit of length ("pixel", "mm", etc.) for the current image or stack. A "um" unit will be converted to "µm". The *depth* argument is ignored if the current image is not a stack. See also: getVoxelSize.

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. Can display HTML formatted text (example).

showMessage("title", "message")
Displays "message" in a dialog box using "title" as the the dialog box title. Can display HTML formatted text (example).

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. See also:getBoolean.

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. Use negative values to show subordinate progress bars as moving dots (example).

showProgress(currentIndex, finalIndex)
Updates the progress bar, where the length of the bar is set to *currentIndex*/*finalIndex*of the maximum bar length. The bar is erased if *currentIndex*>*finalIndex* or*finalIndex*==0.

showStatus("message")
Displays a message in the ImageJ status bar.

showText("string")
Displays a string in a text window.

showText("Title", "string")
Displays a string in a text window using the specified title. Requires 1.48j.

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 resetfunction. 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. Multiple delimiters in the *string* are merged (taken as one) and delimiters at the start or end of the *string* are ignored unless the delimiter is a single comma, a single semicolon or a regular expression. With ImageJ 1.49f or later, *delimiters* can be also a regular expression enclosed in parentheses, e.g. *delimiters*="(\n\n)" splits only at empty lines (two newline characters following each other).

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

Stack (hyperstack) Functions
These functions allow you to get and set the position (channel, slice and frame) of a hyperstack (a 4D

or 5D stack). The HyperStackDemo demonstrates how to create a hyperstack and how to work with it using these functions

Stack.isHyperstack - Returns true if the current image is a hyperstack.
Stack.getDimensions(width, height, channels, slices, frames) Returns the dimensions of the current image.
Stack.setDimensions(channels, slices, frames) - Sets the 3rd, 4th and 5th dimensions of the current stack.
Stack.setChannel(n) - Displays channel *n* .
Stack.setSlice(n) - Displays slice *n* .
Stack.setFrame(n) - Displays frame *n* .
Stack.getPosition(channel, slice, frame) - Returns the current position.
Stack.setPosition(channel, slice, frame) - Sets the position.
Stack.getFrameRate() - Returns the frame rate (FPS).
Stack.setFrameRate(fps) - Sets the frame rate.
Stack.getFrameInterval() - Returns the frame interval in time (T) units.
Stack.setFrameInterval(interval) - Sets the frame interval in time (T) units.
Stack.getUnits(X, Y, Z, Time, Value) - Returns the x, y, z, time and value units.
Stack.setTUnit(string) - Sets the time unit.
Stack.setZUnit(string) - Sets the Z-dimension unit.
Stack.setDisplayMode(mode) - Sets the display mode, where *mode* is "composite", "color" or "grayscale". Requires a multi-channel stack and v1.40a or later.
Stack.getDisplayMode(mode) - Sets the string *mode* to the current display mode.
Stack.setActiveChannels(string) - Controls which channels in a composite color image are displayed, where *string* is a list of ones and zeros that specify the channels to display. For example, "101" causes channels 1 and 3 to be displayed.
Stack.getActiveChannels(string) - Returns a string that represents the state of the channels in a composite color image, where '1' indicates a displayed channel and '0' indicates an inactive channel.
Stack.swap(n1, n2) - Swaps the two specified stack images, where *n1* and*n2* are integers greater than 0 and less than or equal to nSlices.
Stack.getStatistics(voxelCount, mean, min, max, stdDev) - Calculates and returns stack statistics.
Stack.setOrthoViews(x, y, z) - If an *Orthogonal Views* is active, its crosspoint is set to x, y, z (example).
Stack.getOrthoViewsID - Returns the image ID of the current*Orthogonal Views*, or zero if none is active.
Stack.stopOrthoViews - Stops the current *Orthogonal Views* and closes the "YZ" and "XZ" windows.

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

String Functions
These functions do string buffering and copy strings to and from the system clipboard. The CopyResultsToClipboard macro demonstrates their use. See also: endsWith,indexOf, lastIndexOf, lengthOf, startsWith and substring.

String.resetBuffer - Resets (clears) the buffer.
String.append(str) - Appends *str* to the buffer.
String.buffer - Returns the contents of the buffer.

String.copy(str) - Copies *str* to the clipboard.
String.copyResults - Copies the Results table, or selected rows in the Results table (1.47i or later), to the clipboard.
String.getResultsHeadings - Returns the Results window headers.Example. Requires 1.46h.
String.paste - Returns the contents of the clipboard. Requires 1.48j.
String.show(str)
Displays *str* in a text window. Requires 1.48j.
String.show(title, str)
Displays *str* in a text window using *title* as the title.

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.

substring(string, index)
Returns a substring of *string* that begins at *index* and extends to the end of *string*.

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.

toolID
Returns the ID of the currently selected tool. See also: setTool, IJ.getToolName.

toScaled(x, y)
Converts unscaled pixel coordinates to scaled coordinates using the properties of the current image or plot. Also accepts arrays.

toUnscaled(x, y)
Converts scaled coordinates to unscaled pixel coordinates using the properties of the current image or plot. Also accepts arrays. Refer to the AdvancedPlots macro set for examples.

toScaled(length)
Converts (in place) a length in pixels to a scaled length using the properties of the current image.

toUnscaled(length)
Converts (in place) a scaled length to a length in pixels using the properties of the current image.

toString(number)
Returns a decimal string representation of *number*. See also: toBinary, toHex,parseFloat and parseInt.

toString(number, decimalPlaces)
Converts *number* into a string, using the specified number of decimal places.

toUpperCase(string)
Returns a new string that is a copy of *string* with all the characters converted to upper case.

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.

waitForUser(string)
Halts the macro and displays *string* in a dialog box. The macro proceeds when the user clicks "OK". Unlike showMessage, the dialog box is not modal, so the user can, for example, create a selection or adjust the threshold while the dialog is open. To display a multi-line message, add newline characters ("\n") to *string*. This function is based on Michael Schmid's Wait_For_User plugin. Example: WaitForUserDemo.

waitForUser(title, message)
This is a two argument version of *waitForUser*, where *title* is the dialog box title and*message* is the text dispayed in the dialog.

waitForUser
This is a no argument version of *waitForUser* that displays "Click OK to continue" in the dialog box.