# Tips for using ImageJ

by G. von Dassow, 2009

Please notice that the ImageJ website includes a very useful manual organized by menu command. The goal of this tip sheet is to point out the most commonly-used aspects, and some possibly non-obvious features.

## Setting up ImageJ

**Set memory usage:** As a Java program, ImageJ cannot arbitrarily request memory. It needs to request it at the start. The program has to have enough memory to hold all the images you plan to have open, plus a little more. Thus one of the first things you should do, if you expect to work with more than a few images at once, is to tell ImageJ how much memory to ask for. All of ImageJ's preferences are under **Edit→Options**. Choose "Memory & Threads"; you should give ImageJ up to half the total amount of memory in your computer. If you give it much more, your computer will likely have to start using the hard disk as RAM, which will slow it down. If you really need to open something big, you can tell ImageJ to use more memory, but it will slow down. Note that presently a gigabyte of RAM costs about as much as lunch for two. Whatever you set it to, you will have to restart the program for the new value to take effect.

Other options you might care about from the start:

- **JPEG quality** (for files saved in JPEG format by ImageJ) is set under "Input/Output". By default it is set to 75, but if you plan to use JPEG as an intermediate format for anything, you probably want to set this to 100, which is almost (but not quite) lossless.
- Under "Appearance", you might wish to check the box **"Interpolate Zoomed Images"**. This makes things look smoother as you zoom in and out, but if you want to see individual pixels (e.g. to measure them), you want this turned off.
- **"Colors"** refers to foreground and background color, which affects the behavior of fill and clear, and similar functions.
- **"Conversions"** includes a check-box for *perceptual* color weighting when converting RGB files to grayscale. If you anticipate doing such conversions often from real-world images (i.e., not false-colored micrographs or something), you may want this option turned on, because it better preserves luminosity.

**Plugins:** When you first download and install ImageJ it will include a rudimentary set of plugins, very many of which are available from the ImageJ website, from the mundane (reversing a stack, for example) to fairly sophisticated extensions. These are almost always available either as Java source code or compiled class files. In most cases, you can simply download the class file, place it in the appropriate subdirectory of the ImageJ folder (under "plugins", of course), and restart ImageJ. If you download something as java code, you need to compile it (creating a class file) by choosing **Plugins→Compile & Run...** within ImageJ. Next time you won't have to. Note that some plugins replace built-in ImageJ functions.

## Ten simple tasks that everyone does a lot

1. *Selecting areas.* There are enough complexities to selecting things in ImageJ that this is dealt with separately below, but for simple rectangles ImageJ works about like any other program. Command-A selects all; the rectangle tool in the toolbar selects, as you might expect, rectangles. Etc.

2. *Copying and pasting.* Within ImageJ, copy and paste should behave like you expect. But to copy something to another program you must choose **Edit→Copy to System**, and to copy something from another program you must use **File→New→System Clipboard** to make a new window with the contents of the system-wide clipboard.

3. *Duplicating and renaming a window.* **Image→Duplicate...** and **Image→Rename...** I know that's obvious, but it isn't necessarily obvious that it's there. Note that ImageJ will happily let you name all your open windows exactly the same thing, which can be confusing. More often, you will find that repeated use of some plugin will produce multiple windows with the same name. So renaming is useful.

4. *Scaling and rotating.* **Image→Scale...** and **Image→Rotate→Arbitrarily**. Beyond the obvious: by default ImageJ will interpolate, to make a smooth image. Sometimes you don't want it to. I avoid scaling or rotating images in ImageJ if I can, because the simple bilinear interpolation is not as good as other programs can do.

5. *Cropping or expanding the canvas.* **Image→Crop** is pretty obvious, but for some reason the converse operation is buried in **Image→Adjust→Canvas Size...**

6. *Contrast stretching.* There are two convenient ways to enhance brightness and contrast *linearly*: **Image→Adjust→Brightness/Contrast** and **Image→Adjust→Window/Level**. These are sufficiently involved to need their own page (next).

7. *Inverting images.* You can use **Image→Invert** if you actually want to change all the pixel values. However, sometimes you just want to see it inverted, not actually change the numbers. You can use lookup tables (LUTs) to do so; choose **Image→Lookup Tables→Invert LUT**. More about LUTs later on.

8. *Spatial calibration.* Many files produced by automated microscopes include spatial calibration information, which ImageJ can read, such that measurements you make are in real units. You can get to it (and set it) in **Image→Properties...** or **Analyze→Set Scale...**

9. *Zoom in and out.* See **Image→Zoom**; Command-+ and Command-–; or the magnifying glass (option-click to zoom out). Recall that you can set whether or not the image is interpolated when not at 100%; go to **Edit→Options→Appearance...**

10. *Saving image files.* Use TIFF if you plan to do any further analysis. ImageJ's JPEG, even at 100%, always causes some degradation.

## Contrast stretching

This page is concerned *solely* with linear changes in pixel values to enhance image brightness or contrast. The main way to do this is to use the controls under **Image→Adjust→Brightness/ Contrast**. A different option, **Image→Adjust→Window/Level**, is occasionally more convenient to manipulate, but behaves the same and accomplishes the same result.

The window that comes up includes a histogram, and numbers below it that tell you the current minimum and maximum value, meaning what intensity values in your data map to black or white on screen. If you open an 8-bit grayscale image (like the "Boats" sample), then prior to any changes the range will be from 0 to 255. Adjust it using any of the four sliders, or type in white and black point values by clicking "Set". You can use negative numbers or numbers larger than 255, which naturally means nothing will map to true black or white.

If you click "Auto" successively, it will make more and more contrasty versions of the image until it resets to 0–255. It is making successive choices about what fraction of pixels to consign to the all-white and all-black bins. If you have a selection within your image, the "Auto" feature will make these choices based only on the selected area.

**Important:** Nothing has changed about your data, no matter how you yank those sliders around, until you hit "Apply".

The Brightness and Contrast controls behave more or less the same with 8-bit grayscale and 24-bit RGB color images. With the latter, you only get to adjust the overall brightness and contrast, not the brightness and contrast of individual red, green, and blue components. In order to do any color correction, you need to adjust individual components, however. To do this, convert your image either to an RGB stack (**Image→Type→RGB Stack**) or to what ImageJ calls a "composite" image (**Image→Color→Make Composite**).

***RGB Stacks:*** doing it this way lets you see red, green, and blue channels as if they were separate grayscale images (which they are). Switch between them using the slider at the bottom of the image window. As you contrast-stretch each channel, you have to hit "Apply", and then convert it back (**Image→Type→RGB Color**) to see what happened. Unless you want to do it all by histogram-gazing this is a pain; hence...

***Composite images:*** doing it this way lets you see the full color image as you adjust each channel, as you would in Photoshop. The slider on the bottom flips between red, green, and blue, as with RGB stacks.

These controls *do not* remap values when working with 16-bit or 32-bit images. However, when you convert a 16- or 32-bit image to 8-bit, the white and black point values are used to make the 8-bit image. Which brings us to the next topic...

## Bit depth and dynamic range

ImageJ uses, and can convert between, several image types distinguished by the number of bits that encode the intensity value at each pixel, and whether that value is an integer, a floating-point number, or a vector (color).

The fundamental issue at work here is *dynamic range*: how many distinct tones can be distinguished within the image? Computer screens can only display 8 bits worth of different shades (256) in gray, and color on a screen is made from 8 bits each for red, green, and blue. You can't see any more than that, if you can even see that much.

But even if you can't see it all on a screen or a page, you can use more than 8 bits worth of information if you do any kind of enhancement (especially contrast stretching). Many scientific instruments and digital imaging devices (even some consumer-grade digital cameras) produce images with higher bit depth. 16 bits allows for 65,536 distinct shades. Because of limits on sensor devices, this is enough for most realistic measurement needs, and most "16-bit" devices actually deal with only 12 or 14 bits and the rest is either noise or interpolation. Even so, the dynamic range of a 12-bit camera is more than an order of magnitude greater than an 8-bit camera, and there are few imaging tasks that require or can believable fill more than four orders of magnitude.

So why would you need 32 bits? If you do arithmetic on images, you will easily run out of integers in even 16 bit depth. Furthermore, if you were to stick with integers, arithmetical operations would inevitably and irreversibly reduce true dynamic range: if you take the square root of an image in integers alone, the number of distinct shades that result is necessarily the square root of the maximum value in the original. That's why we invented fractions, and ImageJ's 32-bit image type associates each pixel not with an integer but with a floating-point number.

To convert between image types visit **Image→Type**. Some conversions are not allowed or don't make sense. (Note: you can mostly ignore the "8-bit color" type. In the modern world it has no purpose, except that whoever is in charge of the "Science Direct" electronic journal portal has sadistically decided to use it for online scientific publishing.)

If you convert an 8-bit image to a 16- or 32-bit image, each pixel will have the same intensity value it had before (but with decimal places for 32-bit images). If you convert a 32-bit or 16-bit image to an 8-bit image, by default ImageJ uses the current minimum and maximum as the white and black point, stretching out or compressing everything in between. If you need to, you can change this behavior in **Edit→Options→Conversions**.

Note that when ImageJ converts between 8-bit and 16-bit types, you may see your image apparently change. This is because higher bit depth images seem to be automatically contrast stretched (sometimes). However, the underlying numerical values haven't changed, just the way they are mapped to the screen.

## Selection

ImageJ provides a standard array of selection tools (rectangles, freehand blobs, etc.) which are mostly pretty intuitive, and if they aren't, the online manual describes them. Some non-obvious, potentially-useful behaviors:

- ***Composite selections*** (of multiple or overlapping shapes, or shapes with holes in them) can be made by holding down shift to add to the selected area, or option (alt) to subtract from it.
- ***Line selections can have width.*** This is useful for a number of reasons, one of which is that you might like to make intensity measurements along a broad band instead of a one-pixel line. To regulate the width of a line selection, either double-click on the line selection tool in the toolbar, or choose **Image→Adjust→Line Width**, or choose **Edit→Options→Line Width**.
- ***Selections can be specified numerically.*** Choose **Edit→Selection→Specify**. Positional numbers increase from the top left.
- ***Selections can be transferred*** between images. Given a selection in one image, choose another image window and then choose **Edit→Selection→Restore Selection**.
- ***Selections can be remembered.*** ImageJ has an "ROI manager" (for "region of interest") which will keep track of, transfer, save, and re-load numerous selections from a single image. Press T to add the current selection to the manager, and bring up its window, or open it from **Analyze→Tools→ROI Manager**. Click on items in the list to activate them within the image. You can even give them names.
- ***Turn a line into a curve:*** if you make a freehand selection, then chances are it will look a little sloppy. Choose **Edit→Selection→Fit Spline**. If you do this using the segmented line selection tool, the spline anchors are at the points you clicked originally. If you do it with freehand lines or curves, you may get more anchor points than you want; alt(option)-click them to get rid of them. Anchor points can be repositioned.
- ***Straighten a curve:*** I find it incredibly useful to be able to straighten some curvy thing (like the edge of a cell), especially if I want to do something like make intensity measurements as a function of distance along a curved profile. Trace the edge, e.g. with a segmented line, then double-click the line tool and give the line a width, and fit a spline. Then choose **Edit→Selection→Straighten** to make a new image containing the region covered by the straightened curve.
- ***Create masks:*** To convert a selection to a black-and-white *mask* (which you might use it to blot out or modulate some areas), choose **Edit→Selection→Create Mask**.
- Miscellaneous things you can do with RsOI: Enlarge or shrink selections by defined amount (**Edit→Selection→Enlarge**); Fit an ellipse or the tightest convex hull (also in **Edit→Selection** submenu); Rotate selections (guess where...).

## Five filters everyone should get to know

The functions discussed on this page fall into the category of image manipulations that a) require your judgement, b) you should disclose when you publish, and c) are hard to reconstruct if you don't keep track of the steps. All are found under the Process menu.

If the linear brightness and contrast tools don't do it for you, the next stop is **Process→Math→Gamma**. The **Gamma function** makes a concave-up (values greater than one) or concave-down (values less than one) map between input pixel values and output values. The gamma value is an exponent: *A gamma of 0.5 is the square root function, and a value of 2.0 corresponds to a square* (multiplying the image by itself). Useful values are between about 0.6 and 2.5; values less than 1.0 brighten the dim elements without saturating the rest, whereas values greater than one suppress dim elements while increasing contrast on bright elements.

The **Unsharp Mask** filter implemented in ImageJ (**Process→Filters→Unsharp Mask**) is simple but accurate and quick. The radius in question refers to the standard deviation, in pixels, of the Gaussian blur applied to simulate the out-of-focus component. Useful values for the weight are usually between 0.3 (mild) and 0.7 (strong).

The **Process→FFT** submenu contains all the commands for **Fourier filtering**. If this is of interest to you, as it should be if you need to remove regular pollutants from an image, or isolate details, subtract background, etc., the online ImageJ documentation has some simple examples of how to use these functions.

**Noise reduction** is one of the biggest problems in image processing; the best solution is usually to take another picture, if possible. Sadly that isn't always possible, and the most conservative thing to do (because it has been around the longest, not because it is the best) is the **Median** filter (under **Process→Filters**) which, as the name implies, replaces each pixel with the median value in the neighborhood (defined by the radius).

Why the median, and not the mean? There is a **Mean** filter too, of course, and the alluring (but simplistic) **Process→Smooth** command just does the mean of a 3x3 area. But for eliminating sparse, bright noise – that is, outliers – the median works better. There is a more sophisticated way to do it (**Process→Noise→Remove Outliers**) which is an improvement over a simple median filter because it allows you to set a threshold (that is, only change things if they are way different from the median) and choose to change only bright or dark pixels.

Usually I find myself longing for more sharpness, not less, but on those occasions that I need to **blur (er, "smooth") things**, **Process→Filters→Gaussian Blur** is much more flexible than the simple neighborhood mean. The parameter required is the standard deviation describing the Gaussian; fractional values are most useful.

## Arithmetic

ImageJ provides basic functions for arbitrary arithmetic on images. For the most part, these have to be done one at a time. Simple operations that apply a constant to each pixel are in the **Process→Math** submenu, and most are self-explanatory. A few things to pay attention to:

- Consider whether you have, or need, sufficient bit depth to accommodate the result of the operation. If you start with an 8-bit image and multiply by some factor, ImageJ will just set every pixel whose new value exceeds the available range to the maximum number. You may want this to happen. If not, obviously, convert your image to 16-bit or 32-bit.
- Conversely, you need to have sufficient dynamic range to accommodate the result. If you start with an 8-bit image and compute the square root, you will have instantly reduced the number of distinct levels from 256 to 16, more than an order of magnitude. This is true even if you convert to 16-bit beforehand. Therefore you may want to do such things on 32-bit, floating-point images.
- A few only work on 32-bit images; why compute the reciprocal in integer space?
- All operate on a selection within an image as well as the whole image.

The **Process→Image Calculator...** contains arithmetic functions that apply images to each other to produce a third, in which each pixel in the result is a combination of exactly and only the corresponding pixels in the originals. The same considerations listed above for arithmetic with constants apply to pairwise image calculations.

One of the most common uses for the image calculator, aside from image processing *per se*, is to evaluate the *results* of subtle image manipulations. For example, you might like to know what pixels changed when you blurred an image; use **Process→Image Calculator...** with "Difference" selected.

ImageJ includes only a few simple operators. If you need something sophisticated (like sigmoid functions) you will have to look for a plugin or write one yourself. You will need to chain these together as successive operations if you are trying to apply some formula. Macros are useful for assembling formulas; the macro language is fully documented online, and many examples are available, should you need to learn it.

There is one exception to the limit to pairwise operations: you can sum, average, or compute the median, minimum, maximum or standard deviation of more than two images by combining them into a stack and using one of the 3D projection functions. To combine all open images into a stack, choose **Image→Stacks→Images to Stack**, or start with the top image and successively use **Image→Stacks→Add Slice** and paste to build up a stack. Then choose **Image→Stacks→Z Project...** and the desired operation.

(Incidentally, you can often make fascinating artistic effects by combining multiple versions of the same image like this; try making five or six successively-stronger blurs of a photo, then compute the standard deviation.)

## Color

ImageJ does not (and I presume is not intended to) replace the variety of fluid, intuitive color manipulation functions that programs like Photoshop provide, and once you arrive at the stage of caring *artistically* about colors, you are much better off in Photoshop.

### False or true?

One of many generic confusions about color in image processing has to do with the distinction between *true color* and *false color*. As far as I am concerned this is an unfortunate word choice: all color is equally false. The relevant distinction is, how many numbers are associated with each pixel? What most people think of as a "false" color image uses a look-up table to assign a display color to each intensity value – a single number is still associated with each pixel – whereas a "true" color image has a vector, e.g. in the RGB model each pixel has three numbers, one for red intensity, one for green, and one for blue. RGB color uses Cartesian coordinates, and this is the space that ImageJ works with. There are many other ways of making a "color space", such as CMYK (a 4-vector, for printer's pigments) and Hue, Saturation, and Brightness (a three-vector like RGB, but in spherical coordinates).

The confusion arises because one often wants to apply colors that do not reflect the real-world color of the original phenomenon. A common example would be combining distinct fluorescence channels into a single image in which each color corresponds to one label. The colors chosen rarely match (or should match) the "true" spectral properties of the original phenomenon, or even the detector, and therefore people often refer to this process as "false coloring". I don't like this because it implies that I spend a good fraction of my life in the process of falsifying my data somehow. Bleagh! So I suggest the following distinction:

- ***Color coding:*** using a lookup table to assign displayed colors to a single intensity range.
- ***Color balancing:*** modulating the intrinsic, measured color within some kind of color space.
- ***Color combination:*** arranging several sets of data, each with a single intensity range, to cohabit within a color space.

Obviously, these are interconvertible, and language being what it is, we can't avoid confusion, but the following discussion is divided into these categories.

### Color coding with lookup tables

When you install ImageJ it includes a number of lookup tables (LUTs), and you can download more or make your own. Each LUT matches every number from 0 to 255 to a display value. Of course, the default LUT is the gray scale. Note that you can choose whether 0 corresponds to white or black; choose **Image→Lookup Tables→Invert LUT**. The line below the window bar will indicate "Inverting LUT".

All of the other LUTs in the **Image→Lookup Tables** submenu likewise paint the scene without changing the numbers in the image. The "Boats" sample image is a good one to use for exploring the effects of various LUTs. When you have applied a LUT to an image, choose **Image→Color→Show LUT** to see the spectrum applied.

Some of the more useful LUTs:

- **Fire** makes bright things orange and dim things blue
- **Red Hot** makes dim things red, intermediates yellow, and the brightest things white
- **Spectrum** codes things along the color wheel, with both dimmest and brightest things mapped to red
- **Rainbow RGB** does something similar
- **HiLo** codes the very dimmest pixels blue, and the very brightest, red; this is useful for showing which areas in the image are saturated.

LUT files for the ones that appear below the line in the menu can be found in the "luts" subdirectory where you installed ImageJ. Obviously you can throw away any that you don't use.

You can also edit them or make your own: choose **Image→Color→Edit LUT...** and you will see a panel of all 256 entries, each of which you can click on individually and change numerically. Of course that is a total pain and no one does it that way unless you just want to highlight a single value or a narrow range thereof. Instead, click the "Set..." button. You are asked how many colors, which means *not* the end result, but how many you wish to start with, and how you wish to interpolate among them to create the final 256-color table. There is no easy way to get good at this except by trial and error, but as a starting point, to see how it works, do this:

1. Open the "Boats" sample image
2. Choose **Image→Color→Edit LUT...**
3. Click "Set"
4. Type "4" for Number of Colors
5. Choose "Replication"
6. Click the first color in the array of four that now appears in the table; it should be black (zeros for all)
7. Click the last color and make it white (255's for all)
8. Make the second one 127,0,0 (medium red) and the third one 195,127,0 (an unappetizing sort of mud)
9. At this point the image will look rather horrible, because "replication" means there are literally just four distinct colors, so click "Set" again and choose "Interpolation" or "Spline Fitting" (the difference is subtle in this case).

Now you have a rather exaggerated sepia-tone LUT. To see how it filled in the colors, choose **Image→Color→Show LUT**, and to save it for later use, choose **Image→Color→Edit LUT...** again, and do the obvious.

Having color-coded some image with a lookup table, it remains, despite appearances, a grayscale image (whether 8-bit, 16-bit, or 32-bit). You can turn it into an RGB image in the obvious way.

**RGB images – color balancing**

ImageJ is not the ideal program in which to perform color adjustments, if that is all you are up to. However it is sometimes necessary as an intermediate in some other process. For practice the "Lena" sample image is ideal.

Earlier I mentioned that there are two ways to manipulate the red, green, and blue channels using linear brightness and contrast adjustment. There is also a third: choose **Image→Adjust→Color Balance...** This thing is a pain to use on straight RGB images, and it's much more convenient if you first convert it to composite (**Image→Color→Make Composite**), and even then there is a gotcha: the "Apply" button doesn't do what one expects, and you have to convert it back to a straight RGB image (**Image→Type→RGB Color**) to re-map the color values.

Infuriatingly, if you want to apply a gamma adjustment or some other filter to *one* of the channels but not another, you have to do it on an RGB stack, not on an image of type "RGB Color" or on the composite equivalent. See why I recommend using Photoshop for this sort of thing?

**Combining colors, a.k.a. merging channels**

Since an RGB image represents color as if there were a separate grayscale image for red, green, and blue channels, naturally if you have three grayscale images you can merge them together, assigning one to the red, one to the green, one to blue. Choose **Image→Color→Merge Channels...** and specify which one is which. In most cases, choose "none" for the gray channel; this channel is added, if you include it, to all channels, and it has only a few specific uses. Also make sure you click the check-box "Keep Source Images", since nine times out of ten you will want to try it again in different ways.

If you only have two images to merge, of course, you have to choose which colors are appropriate, and one might be empty. Or, you can use a single image for two different channels, and thus get an intermediate color. For example, putting one image in the red channel and the other in both the green and blue channels often gives excellent detail for both colors, and good color contrast. If you put one image in the green channel and the other in both red and blue, you will get a merge that looks hideous to most of us, but which journal editors will love because it is accessible to the most prevalent form of color-blindness. And so on.

> ***About choosing colors:*** *Not all merges are created equal, or end up being equally good for the same data*, and you need to try many different combinations to find the right balance between detail and perceived brightness. The reasons for this have mostly to do with visual psychology – for one thing, green seems brightest to most people, and most people see the most detail in green – and, as the forgoing statement implies, depend significantly on the individual who views the result. Unfortunately, there is also the aspect of printability to consider.

*There simply is no right choice* about whether the nuclei should be green and the actin red, or vice versa. It is often no help to try to be faithful to the original or "true" color of your data, if it has one (e.g. if you are merging fluorescence micrographs or something similar); *don't* simply assume that since you stained the nuclei with a blue dye, you ought to put it in the blue channel. This is an incredibly common failing of many published scientific images which means that the data simply can't be seen as well as they could be.

The method above preserves the independence of the original data in the merge: you could take it apart again and un-mix everything. However it rather limits your options. What if you want a yellow-cyan merge? This happens to maximize perceptual detail for most humans, while being more or less accessible to the color-blind. To do this you somehow have to stuff both original images into the green channel, in addition to putting one in the red and the other in the blue.

To do this you need to calculate an appropriate combination. The simplest thing to do is use the Image Calculator function to compute a new image that consists of either the maximum or the sum of the original images. Which you choose – max or sum – should be determined by trial and error because it depends greatly on how much your originals overlap. Once you have a combined image, just use it as if it was an independent color.

(I just realized, after all these years, that another way to do this is to apply LUTs to each of your images, then convert them to RGB images, then use the Image Calculator function to compute either the sum or the maximum. I'm not sure this saves steps, but maybe it makes it easier to work with.)

These *still* preserve the independence of the original data; you could recover each channel from the merge. This is a good thing. However, there are instances in which you may need to use color coding (in the sense described a few pages ago) as well as color combination. Worse, you might have three images to merge, and not want *any* of them to come out as pure blue. These choices all require you to compute blended images, then use the blends as RGB channels.

Besides blending distinct channels, RGB merges provide a higher-quality means to color code a single image, compared to using LUTs on 8-bit images. You might, for example, put the original in the green channel, a version subjected to gamma of 0.7 in the red channel, and another subjected to gamma of 1.4 in the blue channel. Or you might use a sharpened version of the image for one color, and the original for another...

## Image Stacks

Stacks represent three-dimensional data sets, whether from a confocal microscope (Z-series) or video, as a series of co-aligned images indexed by the third dimension (the focal axis, or time). Another application for stacks is to prepare several versions of the same image for combination, much as Photoshop uses layers. The stack functions in ImageJ overlap little with Photoshop's layer compositing tools, and indeed there are very few tools that deal with stacks exclusively. However, most ImageJ functions operate on image stacks exactly as they would on single images.

### *Where do stacks come from?*

ImageJ opens certain kinds of 3-dimensional or 4-dimensional data natively: multi-frame TIFFs, AVI files, and sequences of numbered images. For the latter, choose **File→Import→Image Sequence...**, navigate to the first image in the series, and open it; in the box that appears, you may specify a range and increment, which is useful if you are opening, say, the series of images created by exporting a QuickTime movie, and only really want every 10th frame.

Other sources of stack data, such as the various kinds of confocal microscope, may require plugins. The LOCI BioFormats library is a good place to start for a catch-all solution; once installed, these plugins will allow ImageJ to open by drag-and-drop most of the microscope-specific formats in use, plus many others.

You may need to assemble stacks by hand. **File→New→Image...** will make you a blank stack with however many slices you want. Or on any image choose **Image→Stacks→Add Slice**. If you have a series of images open that you wish to make into a stack, choose **Image→Stacks→Stack to Images**.

### *Viewing stacks*

You can scroll through stacks with the bar at the bottom, or move slice by slice using the ">" and "<" keys (not the arrow keys). To play through a stack rapidly, press "\"; choose **Image→Stacks→Animation Options...** to set the frame rate; click to stop.

### *Shuffling stacks*

Because the core ImageJ provides only a few basic stack functions, you will probably want to install at least these plugins:

*Concatenator* appends one stack to another

*Stack reverser* does the obvious

*Slice remover* also does the obvious: remove a contiguous series of frames, or every Nth frame

*Stack combiner* puts two stacks side-by-side

*Substack select* will extract a range of slices into a new stack

## Projecting Stacks for a Three-dimensional View

Having gone to the trouble to collect a series of optical sections using a confocal microscope, most of us promptly want to see everything piled up on top of itself again. A moment's reflection shows that there is no simple answer to how this should be done. For example, when you look at a real transparent 3D object, does the light from all those successive slices add up, or do you just see the brightest points within? There isn't any universal answer, but whatever you do it amounts to compiling a geometric projection of the 3D data into some plane image. In principle the plane could be oriented arbitrarily compared to the edges of the volume.

### *Z-projection*

With confocal Z-series a good starting point is to project an imaginary ray through the data at each point in the XY plane, and pick the value of the brightest voxel that you run into along the Z axis as the new value for the pixel corresponding to this ray in the projection plane. Go to **Image→Stacks→Z Project...**, choose which frames to include, and for "Projection Type" choose "Max Intensity".

If you have only a few slices, you may want to average (or sum) the voxels along the ray instead. The sum projection produces a 32-bit result, as do the standard deviation and median.

### *3D projection*

With any single projection it is impossible to see what is in front and what is in back. By projecting at several different angles and playing the frames as a movie, you can simulate a 3D view of the object. Choose **Image→Stacks→3D Project...** This command has numerous options, not all of which I understand. Here's what I do:

- From the menu, choose "Brightest Point" and either the X or Y axis.
- The slice spacing should be set already (see below); you can change it anyway if, for example, you need to stretch things out or compress them.
- Set the initial angle, the total swing, and the number of degrees: a very useful set of numbers is 345, 30, and 1, which makes a gentle swing around the straight-on view.
- The lower transparency bound lets you cut out background below some value; set it to 1 until you know you need it.
- Opacity and depth cueing should be turned off (set to 0); either I don't know how to use them effectively, or they don't work, I'm not sure which.
- If the voxels that comprise your data are about as tall (in Z) as they are wide (in X and Y) you won't need interpolation until you swing past about 30 degrees, and it slows the rendering down considerably. If your data consists of very tall voxels, this can help... although it helps more to collect finer sections.

*Note:* before using the 3D project function, you must first know the dimensions of the voxels in your data set. ImageJ will learn this from many confocal files, but if you made your stack some other way, you will have to enter it manually. Go to **Image→Properties...** to do so.

The 3D projector will compute a stack which, if you use the numbers above, contains 31 images. To animate it, go to **Image→Stacks→Animation Options...** and choose a frame rate (between 15 and 30 is usually good) and click "Loop Back and Forth".

Of course you can also ask ImageJ to render an entire tour around the axis of your choice. If your slices are further apart than one pixel diameter, then as you approach 90 degrees you will be looking through a row of slices with black space between unless you check "Interpolate"; if you *do* check "Interpolate", you will be looking at an intrinsically lower, blurrier resolution, but at least it's something.

### *Stereo pairs*

Better than animations – because they can be printed on a page – are stereo pairs. If you see in stereo in the real world (and not everyone does), then you can learn the trick of fusing two slightly-tilted images into a true 3D view. I made a separate handout for the stereo viewing trick, but to render the stereo pairs in the first place, do this:

1. Go to **Image→Stacks→3D Project...** and do everything as above but for the initial angle enter 356, for the total rotation enter 8, and for the increment enter 8.
2. This makes a stack with two images. To make a divergent-eyed stereo pair you will need to reverse the order; use the Stack Reverser plugin (or copy and paste).
3. Choose **Image→Stacks→Make Montage...** and specify two columns, one row, and no scaling. Now you've got a stereo pair.

You may find these settings too extreme, but it will exaggerate the depth to start with; you may get a more accurate view by using a 6-degree angle.

And better than stereo pairs are stereo triplets: arrange left-eye and right-eye images in a row with left-right-left; you will be able to see from the front and back at the same time.

### *Grouped Z-projection*

Sometimes you still want sections, you just want thicker ones. Install both the "Grouped Z Projector" and "Running Z Projector" plugins. The former does any of the built-in Z projection routines on regular subsets of the original stack. The latter produces a derivative stack in which each image is the projection of some number of adjacent ones.

**Five other useful things you can do with stacks**

1. **Make a montage.** To lay out some or all of the slices side by side, go to **Image→Stacks→Make Montage...**, specify the number of columns and rows and whether you want them all included, etc.

2. **Reslice in another axis.** Choose **Image→Stacks→Reslice...** If the voxels are not cubical, you will need to adjust the input and output slicing to preserve accurate aspect ratios. You can re-slice a rectangular selection, or a line selection at any angle. For a line selection you can specify a number of slices; based on the input and output slice spacings, it will re-slice a band that ends at your original line selection. To reslice at an arbitrary angle, then, it is usually better to rotate the stack and use a rectangular selection.

3. **Extended Depth-of-Field images.** What if you could erase all the out-of-focus information in each slice within a stack, and then pile up the in-focus parts? This is a computationally-challenging task to do well, but there is a quick-and-dirty way implemented by the "Stack Focuser" plugin. Two other plugins that implement more sophisticated methods are available (both are from the Biomedical Imaging Group at the École Polytechnique Fédérale de Lausanne, http://bigwww.epfl.ch/demo/edf/); these are slow, but with high-quality input they produce high-quality output.

4. Numerous **volume rendering** plugins exist. If you find one that works, please show me. Volume rendering uses more sophisticated methods than brightest-point projection, often to produce a semi-solid object. Most ImageJ renderers I have tried are either very slow, or don't work, or I haven't learned how to use them effectively. The problem is, there are very many ways to do this, which depend on the nature of the data, and it is not trivial to write fast code in Java. The least-useless volume renderer I have found so far is Kai Barthel's "Volume Viewer", which also shows an arbitrary orthogonal slice. For surface rendering, try "VolumeJ" by Michael Abramoff. Benjamin Schmid's "ImageJ 3D Viewer" looks promising but, as of this writing, does not work for me except as a component of Fiji (which is an ImageJ distribution bundle; see http://pacific.mpi-cbg.de)

5. **Save a movie.** ImageJ will save uncompressed AVI movies (**File→Save As→AVI...**) whose frame rate is set in **Image→Stacks→Animation Options...** This AVI file can be opened of course and recompressed using the QuickTime player or other program.

   Several plugins which promise to export QuickTime movies directly; the only one I have found useful is the "QT Stack Writer" by the Hardins (Windows users need to install QuickTime and a QuickTime-for-Java extension).

   ImageJ can also save a stack as a sequence of TIFFs, which the QuickTime player (or other movie making programs) can import. This is the safest and most flexible way to encode a movie without losing anything in the process.

   But what frames to use? Do you want your movie to auto-reverse? If so, the QuickTime player can do that; few others can, and programs like PowerPoint in which you might wish to display your movies often ignore the loop settings. To solve this, render both the forward and reverse frames: duplicate the stack (**Image→Duplicate...**), reverse the order, delete the first and last frames from the copy, and then concatenate the two stacks.

## Illustrating and measuring directed motion – making kymographs

Video sequences are just image stacks to ImageJ. One of the common needs in the analysis of time-series data is to show motion, or patterns of motion, on the page. If you are lucky enough to deal with motion that takes place exclusively or even predominantly in one direction, then what you want is a *kymograph*. Everyone is already familiar with this idea: seismographs, lie detectors, and those heartbeat monitors in the ICU all plot some kind of spatial axis versus time. In such plots one can not only illustrate motion, but also measure velocities directly.

***The quick-and-dirty way:***

If you have relatively few frames (say, less than a hundred), few but well-defined objects of interest, and if the object of interest moves more than its own diameter from one frame to the next, then one can make a kymograph by cutting strips containing the object(s) out of the movie, and laying them out side by side.

First determine the approximate axis of motion of the object using the angle tool in the toolbar. To measure this accurately, project a subset of frames in which the object of interest moves. Next rotate the image so that the axis of motion is either horizontal or vertical. Select the thinnest possible rectangle that surrounds the object throughout its full range of motion, then choose **Image→Duplicate...**, checking the box to duplicate the entire stack. This excises the selected strip at every time point. Arrange all the strips in a single column (for a horizontal strip) or row (vertical strips) using **Image→Stacks→Make Montage...**

***If you have more frames and relatively little motion per frame:***

In this case, objects will make continuous streaks, like the point of a pencil on paper. Analogous to a common need for 3D data sets, we need to view a slice in a plane orthogonal to the original focal plane. Therefore, once you rotate the original stack to align the motion horizontally or vertically, you can use **Image→Stacks→Reslice...** to create a new stack. In fact, if you have convection taking place across a large area, you can reslice the entire area; each slice in the new stack will show motion along a single one-pixel-wide line.

One usually needs more than a one-pixel line because if objects stray from straight paths even slightly then they move to an adjacent slice. On the new stack, then, use the Z-projection functions to compile a slab wide enough to capture long tracks. It is a matter of trial and error how wide a slab to make and whether to use averaging or maximum projection.

***If objects don't move on straight lines***

All is not necessarily lost. By reslicing, and then using three-dimensional projection or volume rendering of some kind, you can make a "kymocube" animation, or stereo pairs. What you will lose is the ability to measure velocities.

But all is *still* not lost. If you can follow one object at a time, and you are willing to go through the frames and lay down a segmented line, either the reslicing or straightening functions will extract a stretched-out track which is quantifiable (ignoring direction changes).

***If you have many objects that move more than their own diameter from frame to frame:***

Give up. O.K., I don't really mean that; what I mean is, you have to be able to tell which speck is which from frame to frame. Even if you can identify objects yourself by looking at them, kymographs will not give you any kind of satisfying representation and you are better off with a pencil. Or you need to figure out how to collect images at a faster frame rate.