

Analyzing fluorescence microscopy images with ImageJ

Peter Bankhead
Queen's University Belfast

May 2014

This work is made available in the hope it will help
a few more people develop an interest in image analysis.
If you have any comments, corrections or suggestions,
please contact me at [p.bankhead\[at\]qub.ac.uk](mailto:p.bankhead@qub.ac.uk),
so that it might one day get better.



© 2013, Peter Bankhead

The initial version of this text was written at the Nikon Imaging Center @ Heidelberg University, 2010 – 2012.

Most figures were created from original drawings, or from sample images obtained directly from Fiji under **File** → **Open Samples** (also available at <http://imagej.nih.gov/ij/images/>), apart from:

- Sunny cell (e.g. Figure 1.3), thanks to Astrid Marx
- Widefield cells (Figure 2.1), yeast cells (e.g. Figure 2.3), & fluorescent beads (Figure 15.3) thanks to Ulrike Engel
- XKCD cartoons, from www.xkcd.com

Contents

Contents	iii
Preface	v
I Introducing images, ImageJ & Fiji	1
1 Images & pixels	3
2 Dimensions	15
3 Types & bit-depths	23
4 Channels & colours	33
5 Files & file formats	39
II Processing fundamentals	49
6 Overview: Processing & Analysis	51
7 Measurements & regions of interest	53
8 Manipulating individual pixels	59
9 Detection by thresholding	67
10 Filters	79
11 Binary images	101
12 Processing data with higher dimensions	109
13 Writing macros	117

III Fluorescence images	125
14 From photons to pixels	127
15 Blur & the PSF	131
16 Noise	143
17 Microscopes & detectors	163
18 Simulating image formation	175
Index	185

Preface

‘To them, I said, the truth would
be literally nothing but the
shadows of the images.’

The Republic, Plato

In Plato’s *Republic*, Socrates proposes a scene in which prisoners are chained up in an underground cavern so they cannot even move their heads¹. The view of the prisoners is confined to the shadows cast upon the wall in front of them: their own shadows, and also those of people passing by behind them. If all they could ever see were the shadows, Socrates asks, then wouldn’t the shadows seem like the truth to them? Wouldn’t they interpret sounds and voices as if they are coming from the shadows themselves, rather than anyone or anything behind them, and think that they understand – failing to realize that they are only seeing reflections of reality, and not reality itself?

The purpose of bringing this up here is not so much to introduce a profound examination how our limited minds may overestimate their grasp of reality based upon what our senses can tell us, but rather to draw attention to a rather more simple and specific point: when it comes to analyzing fluorescence microscopy images in biology, it is *essential* to remember that we are not directly seeing the phenomena we normally want to measure. At best, we can record images containing information that relates in some way to the reality of what we wish to study – but which is nevertheless quite far removed from that reality.

Computers are excellent for making measurements. Armed with some relevant software, it is possible to make masses of measurements quickly – all to an intimidating number of decimal (or rather binary) places, giving an impressive semblance of precision. But to ensure that the measurements made are at all *meaningful*, and that they are properly interpreted, requires that we know the limitations of what the images can tell us. Otherwise we risk very accurately quantifying phenomena like point spread functions and noise, and reporting these as if they might have biological significance.

¹The dialogue can be read at http://en.wikisource.org/wiki/The_Republic/Book_VII.



Figure 1: Plato's cave.

The overall purpose of this book is to provide a path for busy biologists into understanding fluorescence images and how to analyze them, using the free, open-source software ImageJ (specifically the Fiji distribution) to explore the concepts. It divides into three parts:

1. *Introduction to images, ImageJ & Fiji* – The basic concepts of digital images in microscopy. While essential to know, if the idea of trying to quantitatively analyze an 8-bit RGB colour JPEG compressed AVI file already fills you with horror, you may be able to save time by skipping much of this.
2. *Processing fundamentals* – A tour of the most important and general techniques that can be used to help extract information from many types of image. After this, you should be happy with Gaussian filters, thresholding, spot detection and object analysis, with some macro-writing skills.
3. *Fluorescence images* – More detail about fluorescence image formation. This is necessary not only to properly interpret any measurements, but also when figuring how to ensure data is recorded well in the first place.

No background in image analysis or computer programming is assumed, nor is the maths here very deep. Specifically, if you are at ease with arithmetic, means, medians², standard deviations and the occasional square root, this is enough.

²Just in case: the mean is the familiar average, i.e. add up all the values, and divide by the number of values you have. The median is what you get if you sort a list of values, and then choose the one in the middle.

There are a large number of questions and practicals strewn throughout the pages. These are not supposed to steal your time (although a few could be quite tricky), but rather to make the ideas more memorable and the reading feel more worthwhile. In most cases, my suggested solutions are provided at the end of each chapter. You may well find alternative or better solutions.

Part I

Introducing images, ImageJ & Fiji

Images & pixels

Chapter outline

- *Digital images are composed of pixels*
- *Each pixel has a numeric value, often related to detected light*
- *The same pixel values can be displayed in different ways*
- *In scientific image processing, image appearance can be changed independently of pixel values by modifying a lookup table*

1.1 Introduction

The smallest units from which an image is composed are its *pixel*. The word pixel is derived from *picture element* and, as far as the computer is concerned, each pixel is just a number. When the image data is displayed, the values of pixels are usually converted into squares of particular colours – but this is only for our benefit to allow us to get a fast impression of the image contents, i.e. the approximate values of pixels and where they are in relation to one another. When it comes to processing and analysis, we need to get past the display and delve into the real data: the numbers.

This distinction between data (the pixel values) and display (the coloured squares) is particularly important in fluorescence microscopy. The pixels given to us by the microscope are measurements of the light being emitted by a sample. From these we can make deductions, e.g. more light may indicate the presence of a particular structure or substance, and knowing the exact values allows us to make comparisons and quantitative measurements. On the other hand, the coloured squares do not matter for measurements: they are just nice to look at (Figure 1.1). Still, two related facts can cause us trouble:

1. **Images that look the same can contain *different* pixel values**
2. **Images that look *different* can still contain *the same* pixel values**

This makes it quite possible to analyze two different images that *appear* identical, but to get very different results. Therefore to be sure we are observing and measuring the right things, we need to know what is happening whenever we open, adjust and save our images. It is not enough to trust our eyes.

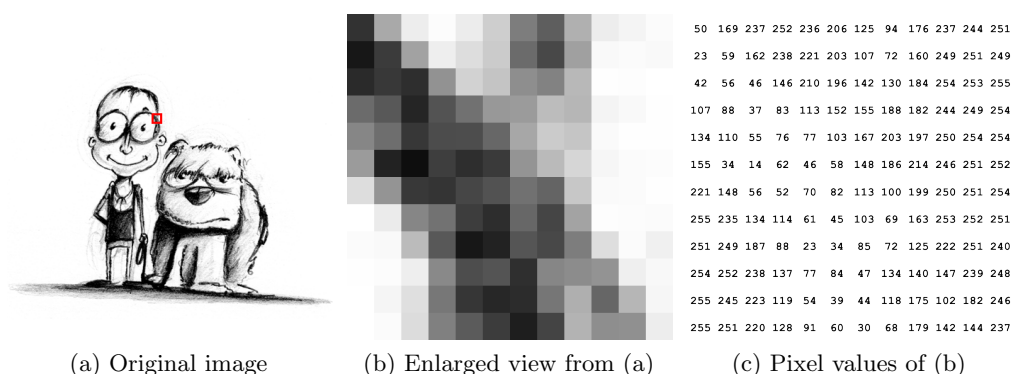


Figure 1.1: An image depicting an interestingly-matched couple I saw recently on the way home from work. (a) & (b) The image is shown using small squares of different shades of gray, where each square corresponds to a single pixel. This is only a convention used for display; the pixels themselves are stored as arrays of numbers (c) – but looking at the numbers directly it is pretty hard for us to visualize what the image contains.

1.2 ImageJ & Fiji

So to work with our digital images we do not just need any software that can handle images: we need scientific software that allows us to explore our data and to control exactly what happens to it.

ImageJ, developed at the National Institutes of Health by Wayne Rasband, is designed for this purpose. The ‘J’ stands for Java: the programming language in which it is written. It can be downloaded for free from <http://imagej.net>, and its source code is in the public domain, making it particularly easy to modify and distribute. Moreover, it can be readily extended by adding extra features in the form of *plugins*, *macros* or *scripts*¹.

This marvellous customisability has one potential drawback: it can be hard to know where to start, which optional features are really good, and where to find them all. Fiji, which stands for *Fiji Is Just ImageJ*, goes some way to addressing this. It is a distribution of ImageJ that comes bundled with a range of add-ons intended primarily for life scientists. It also includes its own additional features, such as an integrated system for automatically installing updates and bug-fixes, and extra open-source libraries that enable programmers to more easily implement sophisticated algorithms.

Therefore, everything ImageJ can do can also be accomplished in Fiji (because Fiji contains the full ImageJ inside it), but the converse is not true (because Fiji contains many extra bits). Therefore in this course we will use Fiji, which can be downloaded for free from <http://fiji.sc/>.

¹All three of these consist of some computer-readable instructions, but they are written in slightly different languages. Macros are usually the easiest and fastest to write, and we will start producing our own in Chapter 13. For more complex tasks, the others may be preferable.

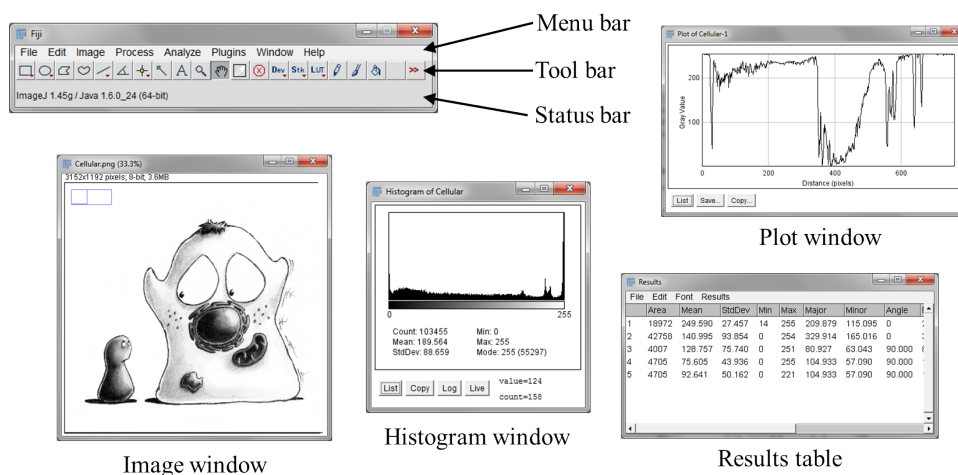


Figure 1.2: The main user interface for Fiji.

1.2.1 The user interface

It can take some time to get used to the ImageJ/Fiji² user interface, which may initially seem less friendly and welcoming than that found in some commercial software. But the good news is that, once you find your way around, it becomes possible to *do* a lot of things that would simply not be possible in many other software applications.

Some of the components we will be working with are shown in Figure 1.2. At first, only the main window containing the menu, tool and status bars is visible, and the others appear as needed when images are opened and processed. Should you ever lose this main window behind a morass of different images, you can bring it back to the front by pressing the **Enter** key.


1.2.2 Tips & tricks

Here are a few non-obvious tips that can making working with ImageJ or Fiji easier, in order of importance (to me):

- Files can be opened quickly by dragging them (e.g. from the desktop, Windows Explorer or the Mac Finder) onto the **Status bar**. Most plugins you might download can also be installed this way.
- If you know the name of a command (including plugins) but forget where it is hidden amidst the menus, type **Ctrl + L** (or perhaps just **L**) to bring up the **Command Finder** – it may help to think of it as a *List* – and start typing the name. You can run the command directly, or select

²At the risk of confusion, I will refer to ImageJ most of the time, and Fiji only whenever discussing a feature not included within ImageJ.

Show full information to find out its containing menu. Note that some commands have similar or identical names, in which case they might only be distinguishable by their menu.

- ImageJ's **Edit** → **Undo** has very limited abilities – it *may* be available if you modify a single 2D image, but will not be if you process data with more dimensions (see Chapter 2). While inconvenient if you are used to long undo-lists in software like Microsoft Word or Adobe Photoshop, there is a good rationale behind it: supporting undo could require storing multiple copies of previous versions of the image, which might rapidly use up all the available memory when using large data sets. The solution is to take care of this manually yourself by choosing **Image** → **Duplicate...** to create a copy of the image before doing any processing you may not wish to keep.
- There is a wide range of shortcut keys available. Although the menus claim you need to type **Ctrl**, you do not really unless the option under **Edit** → **Options** → **Misc...** tells you otherwise. You can also add more shortcuts under **Plugins** → **Shortcuts** → **Create Shortcut...**
- To move around large images, you can use the **scrolling tool** , or simply click and drag on the image while holding down the **spacebar**. A small rectangular diagram (visible on the top left of the image window in Figure 1.2) indicates which part of the entire image is currently being shown.
- There are more tools and options than meet the eye. Double-clicking and right-clicking on icons in the **Tool bar** can each reveal more possibilities.
- Pressing **Escape** *may* abort the operation of any currently-running command... But it requires the command's author to have implemented this functionality. So it might not do anything.

1.2.3 Finding more information

Links to more information for using ImageJ, including a detailed manual, are available at <http://imagej.net/docs/>. These resources also apply to Fiji. More specific information for Fiji's additional features, along with tutorials, can be found at <http://fiji.sc/wiki/index.php/Documentation>.

Referencing Fiji and ImageJ

Whenever publishing work using Fiji or ImageJ, you should check their respective websites for information regarding how to refer to them. Furthermore, some specific plugins have their own published papers that should be cited if the plugins are used. See <http://imagej.net/docs/faqs.html> and <http://fiji.sc/wiki/index.php/Publications> for more information.

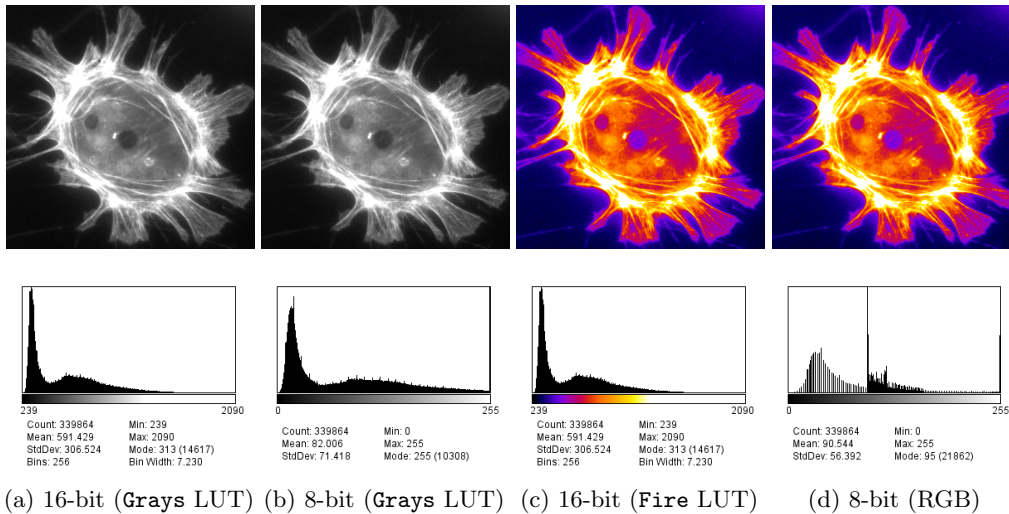


Figure 1.3: Do not trust your eyes for image comparisons: different pixel values might be displayed so that they look the same, while the same pixel values may be displayed so that they look different. Here, only the images in (a) and (c) are identical to one another in terms of their pixel values – and only these contain the original data given by the microscope. The terms used in the captions will be explained in Chapters 3 and 4.

1.3 Data & its display

1.3.1 Comparing images

Now we return to the data/display dichotomy. In the top row of Figure 1.3, you can see four images as they might be shown in ImageJ. The first and second pairs both *look* identical to one another. However, it is only actually (a) and (c) that *are* identical in terms of content. Since these contain the original pixel values given by the microscope they could be analyzed, but analyzing either (b) or (d) instead may well lead to untrustworthy results.

Reliably assessing the similarities and differences between images in Figure 1.3 would therefore be impossible just based on their appearance in the top row, but it becomes much easier if we consider the corresponding *image histograms* below. These histograms (created with `Analyze` → `Histogram`) depict the total number of pixels with each different value within the image as a series of vertical bars, displayed above some extra statistics – such as the maximum, minimum and mean of all the pixels in that image. Looking at the histograms and the statistics below make it clear that only (a) and (c) could possibly contain the same values.

Uses of histograms
Comparing images, calculating pixel statistics, identifying possible thresholds (Chapter 9)

Question 1.1

If you want to tell whether two images are identical, is comparing their histograms *always* a reliable method?

Solution

1.3.2 Mapping colours to pixels

The reason for the different appearances of images in Figure 1.3 is that the first three do not all use the same *lookup tables* (LUTs; sometimes alternatively called *colour maps*), while in Figure 1.3d the image has been *flattened*. Flattening will become relevant in Chapter 4, but for now we will concentrate on LUTs.

A LUT is essentially a table in which rows give possible pixel values alongside the colours that should be used to display them. For each pixel in the image, ImageJ finds out the colour of square to draw on screen by ‘looking up’ its value in the LUT. This means that when we want to modify how an image appears, we can simply change its LUT – keeping all our pixel values safely unchanged.

Practical 1.1

Using `Spooked.tif` as an example image, explore pixel values and LUTs in Fiji. Originally, each pixel should appear with some shade of gray (which indicates a correspondingly sombre LUT). As you move the cursor over the image, in the status bar at the top of the screen you should see `value =` beside the numerical value of whichever pixel is underneath the cursor.



If you then go to `Image` → `Lookup Tables` → ... and click on some other LUT the image appearance should be instantly updated, but putting the cursor over the same pixels as before will reveal that the actual values are unchanged.

Finally, if you want to ‘see’ the LUT you are using, choose `Image` → `Color` → `Show LUT`. This shows a bar stretching from 0 to 255 (the significance of this range will be clearer after Chapter 3) with the colour given to each value in between. Clicking `List` then shows the actual numbers involved in the LUT, and columns for `Red`, `Green` and `Blue`. These latter columns give instructions for the relative amounts of red, green and blue light that should be mixed to get the right colour (see Chapter 4).

Why use different LUTs?

The ability to change LUTs has several advantages. A simple one is that we can use LUTs to make the colours in our image match with the wavelengths of the light we have detected, such as by showing a DAPI staining in blue or GFP in green. But often this is not really optimal, and you may prefer to show an image using some multicoloured LUT (e.g. `Fire` in ImageJ) that does not otherwise have any physical relevance. This is because the eye is relatively poor at distinguishing different shades of the same colour, and presenting the identical information using many different colours can make differences between pixel values more apparent.

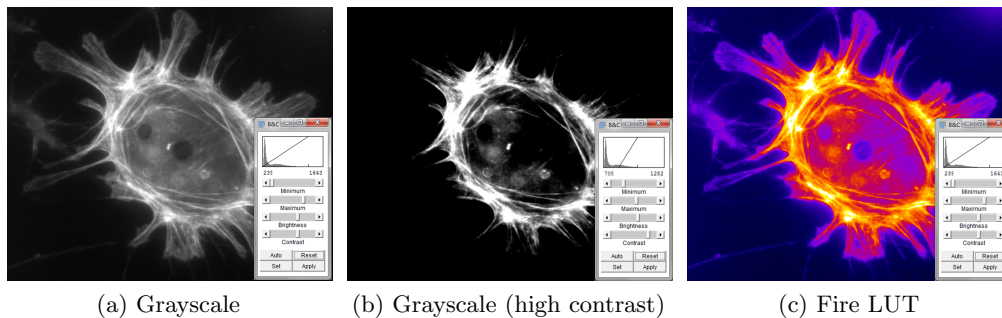


Figure 1.4: The same image can be displayed in different ways by adjusting the contrast settings or the LUT. Nevertheless, despite the different appearance, *the values of the pixels are the same in all three images.*

Modifying the LUT can help make information visible

But swapping one set of LUT colours for another is not the only way to change the appearance. We can also keep the same colours, but change which pixel values each colour is used for.

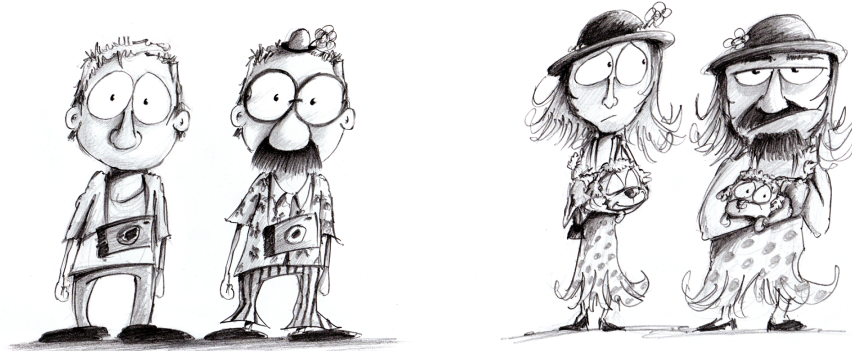
For example, suppose we have chosen a gray LUT. Most monitors can (theoretically) show us 256 different shades of gray, so we can give a different shade to pixels with values from 0–255, where 0 corresponds to black and 255 corresponds to white. But suppose our image only contains interesting values in the range 5–50. Then we will only be using 46 rather similar shades of gray to display it, and not using either our monitor or our eyesight to their full capacities. It would be easier to see what is happening if we made every pixel with a value ≤ 5 black and ≥ 50 white, and then distributed all our available shades of gray to the values in between. This would make full use of the colours we have in our LUT, and give us an image with improved *contrast*. Of course, we can also apply the same principle using any other LUT, replacing black and white with the first and last colours in the LUT respectively.

Adjusting the display range

This type of LUT adjustment is done in ImageJ using the `Image → Adjust → Brightness/Contrast...` command (quickly accessed by typing `Shift + C`; see Figure 1.4). The first two scrollbars that appear are called **Minimum** and **Maximum**, and these define the thresholds below and above which pixels are given the first or last LUT colour respectively. Modifying either of these sliders automatically changes the **Brightness** and **Contrast** sliders. Although the terms **brightness** and **contrast** are probably more familiar, it is usually easier to work with **Minimum** and **Maximum**. If you know, for example, you do not care to see anything in the darkest part of the image, you can increase the value of **Minimum** to clip it out of the picture (only for display!), and devote more different colours for the part of

LUTs & uniqueness

Note that the range of possible values in an image can easily exceed the range of colours the LUT contains, in which case *pixels with different values will be displayed using exactly the same colour.*



(a) Fundamentally the same – despite different appearances

(b) Fundamentally different – despite the same (or similar) appearance

Figure 1.5: The same person may appear very different thanks to changes in clothing and accessories (a). Conversely, quite different people might be dressed up to look very similar, and it is only upon closer inspection that important differences become apparent (b). The true identity and dressing can be seen as analogous to an image’s pixel values and its display.

the image that is really interesting.

Practical 1.2

Return again to `Spooked.tif` and investigate the contents of the image by adjusting the `Minimum` and `Maximum` sliders. What are the best settings when viewing the image?

After you have done this, explore the effects of pressing the `Auto`, `Reset`, `Set` and `Apply` buttons on the `Brightness/Contrast` panel.

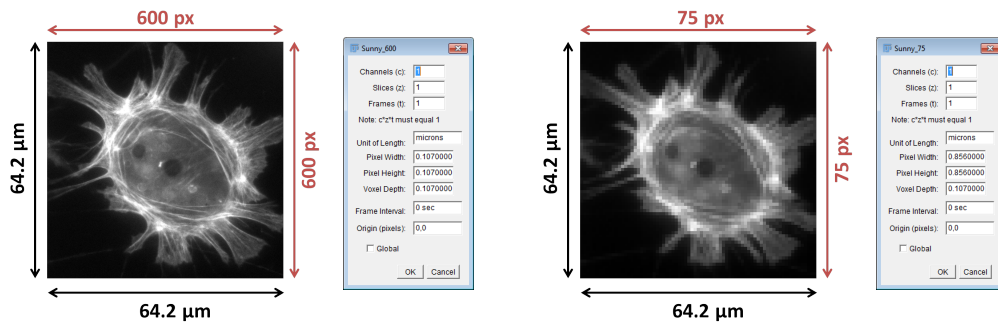
(And subsequently be wary of using the `Apply` button ever again.)

Solution

Scientific image analysis is not photo editing!

One way to imagine the distinction between pixel values and a LUT is that the former determine the real *identity* of the image, while the latter is simply the *clothing* the image happens to be wearing (Figure 1.5). Our interest is in the identity; the clothing is incidental, though might be chosen to accent certain features of interest.

It is vital for scientific analysis that changing the LUTs, either by switching the colours or adjusting `Brightness/Contrast`, does not mess up or otherwise modify the underlying data. This is the case for the normal contrast controls in ImageJ (if you avoid the tempting `Apply` button). But this feature would not



(a) 600 × 600 pixel image and its properties

(b) 75 × 75 pixel image and its properties

Figure 1.6: Two images with the same field of view, but different numbers of pixels – and therefore different pixel sizes. In (a) the pixel width and height are both $64.2/600 = 0.107 \mu\text{m}$. In (b) the pixel width and height are both $64.2/75 = 0.856 \mu\text{m}$. For display, (b) has been scaled up to look the same size as (a), so its larger pixels make it appear ‘blocky’.

be so important in other applications like photo editing, where only the final appearance is what matters. Therefore, if you adjust the brightness, contrast or ‘levels’ in Photoshop, for example, you really *do* change the underlying pixel values – and, once changed, you cannot ‘change them back’ (apart from using the **Undo** command) and expect to get your original values.

Therefore, if you have enhanced an image in Photoshop, your pixel values can easily be changed in a way that makes their reliable interpretation no longer possible!

1.4 Properties & pixel size

Hopefully by now you are appropriately paranoid about accidentally changing pixel values and therefore compromising your image’s integrity, so that if in doubt you will always calculate histograms or other measurements before and after trying out something new to check whether the pixels have been changed.

This chapter ends with the other important characteristic of pixels for analysis: their *size*, and therefore how measuring or counting them might be related back to identifying the sizes and positions of things in real life. Sizes also need to be correct for much analysis to be meaningful.

Pixel sizes are found in ImageJ under **Image** → **Properties...**, where you will see values for **Pixel width** and **Pixel height**, defined in terms of **Unit of Length**. A useful way to think of these is as proportions of the size of the total field of view contained within the image (see Figure 1.6). For example, suppose

we are imaging an area with a width of 100 μm , and we have 200 pixels in the horizontal direction of our image. Then we can treat the ‘width’ of a pixel as $100/200 = 0.5 \mu\text{m}$. The pixel height can be defined and used similarly, and is typically (but not necessarily) the same as the width.

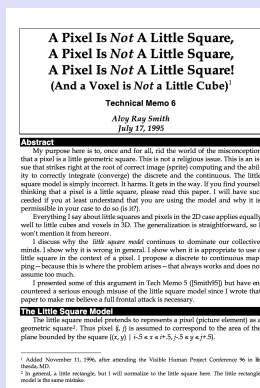
Voxels
In 3D, pixels are sometimes called *voxels* (from *volume pixels*), so **Voxel depth** is the spacing between slices in a *z*-stack (see Chapter 2).

Pixel squareness

Talking of pixels as having (usually) equal widths and heights makes them sound very square-like, but earlier I stated that pixels are not squares – they are just displayed using squares.

This slightly murky philosophical distinction is considered in Alvy Ray Smith’s technical memo (*right*), the title of which gives a good impression of the central thesis^a. In short, pushing the pixels-are-square model too far leads to confusion in the end (e.g. what would happen at their ‘edges’?), and does not really match up to the realities of how images are recorded (i.e. pixel values are not determined by detecting light emitted from little square regions, see Chapter 15). Alternative terms, such as *sampling distance*, are often used instead of pixel sizes – and are potentially less misleading. But ImageJ uses pixel size, so we will as well.

^ahttp://alvyray.com/Memos/CG/Microsoft/6_pixel.pdf



1.4.1 Pixel sizes and measurements

Knowing the pixel size makes it possible to calibrate size measurements. For example, if we measure some structure horizontally in the image and find that it is 10 pixels in length, with a pixel size of 0.5 μm , we can deduce that its actual length in reality is (roughly!) $10 \times 0.5 = 5 \mu\text{m}$.

This calibration is often done automatically when things are measured in ImageJ (Chapter 7), and so the sizes must be correct for the results to be reasonable. All being well, they will be written into an image file during acquisition and subsequently read – but this does not always work out (Chapter 5), and so `Properties...` should always be checked. If ImageJ could not find sensible values in the image file, by default it will say each pixel has a width and height of 1.0 pixel... not very informative, but at least not wrong. You can then manually enter more appropriate values if you know them.

1.4.2 Pixel sizes and detail

In general, if the pixel size in a fluorescence image is large then we cannot see very fine detail (see Figure 1.6b). However, the subject becomes complicated by the diffraction of light whenever we are considering scales of hundreds of nanometres,

so that acquiring images with smaller pixel sizes does not necessarily bring us extra information – and might actually become a hindrance (see Chapters 15 and 16).

Solutions

Question 1.1 No! It is possible for two quite different images to have identical histograms. For example, if you imagine having an image and then randomly reshuffling its pixels to get a second image, then the histogram would be unchanged but the image itself would be. Other, more subtle differences could also lead to different arrangements of pixels (and therefore different images), but similar appearances and identical histograms and statistics.

Nevertheless, histogram comparison is fast (in ImageJ, just click on each image and press H) and a pretty good guide. Non-identical histograms would least show that images are categorically *not* the same. A more reliable method of comparing images will come up in Chapter 8.

Practical 1.2 When the **Minimum** is set to 0 and the **Maximum** is greatly reduced, an eerier picture should emerge. This was always present in the pixel data, but probably could not be seen initially.

In images like this, the ‘best’ contrast setting really depends upon what it is you want to see – although Chapter 8 describes one way to try to see ‘everything at once’.

With regard to the buttons, **Auto** chooses contrast settings that might look good (by bringing **Minimum** and **Maximum** a bit closer to one another). **Reset** makes sure **Minimum** and **Maximum** match with the actual minimum and maximum pixel values in the image (or 0 and 255 for an 8-bit image – Chapter 3), while **Set** allows you to input their values manually.

Apply is altogether more dangerous. It really does change the pixel values of the image. *This loses information and so is rarely a good idea!* After pressing **Apply**, you are very likely not to be able to get your original pixel values back again.

Dimensions

Chapter outline

- *The number of dimensions of an image is the number of pieces of information required to identify each pixel*
- *In ImageJ, images with more than 2 dimensions are stored in a stack or hyperstack*

2.1 Identifying dimensions

The idea of image dimensions is straightforward: the number of dimensions is the number of pieces of information you need to know to identify individual pixels.

For example, in the most familiar 2D images, you can uniquely identify a pixel by knowing its x and y spatial coordinates. But if you needed to know x and y coordinates, a z -slice number, a colour channel and a time point then you would be working with 5D data (Figure 2.1). You could throw away one of these

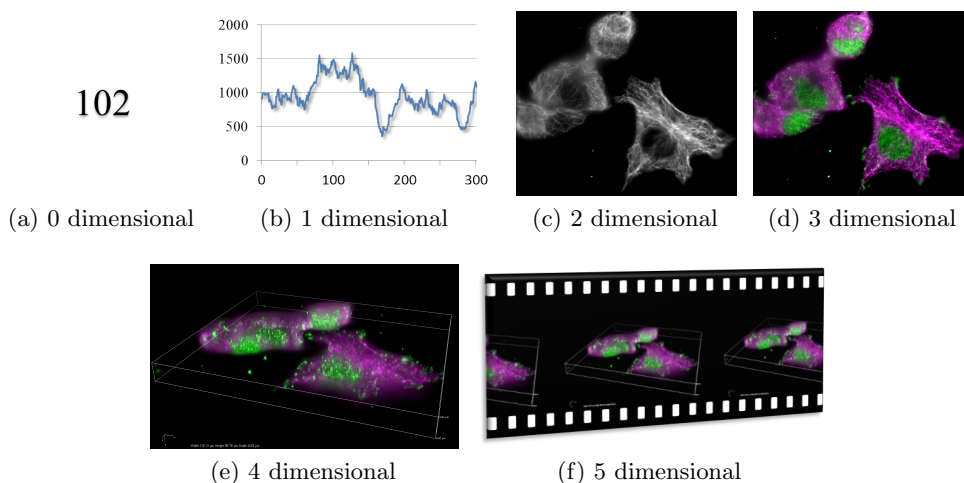


Figure 2.1: Depictions of images with different numbers of dimensions. (a) A single value is considered to have 0 dimensions. (b–f) Additional dimensions are added, here in the following order: x coordinate (1), y coordinate (2), channel number (3), z slice (4) and time point (5).

dimensions – any one at all – and get a 4D image, and keep going until you have a single pixel remaining: a 0D image. Throw away that, and you no longer have an image.

In principle, therefore, 2D images do not need to have x and y dimensions. The dimensions could be x and z , or y and time, for example. But while we may play around with the identity of dimensions, the important fact remains: an n D image requires n pieces of information to identify each pixel.

2.2 Stacks & Hyperstacks

In the beginning there were 2D images. Then ImageJ supported *stacks*, which allowed an extra dimension that could either include different time points or z -slices – but not both. Nowadays, *hyperstacks* are the more flexible derivative of stacks, and can (currently) store up to 5 dimensions without getting them confused.

Hyperstack flexibility

A hyperstack can contain 0–5 dimensions, while a stack can only contain 0–3. So why worry about stacks at all?

The main reason comes from ImageJ’s evolution over time. Some commands – perhaps originating in the pre-hyperstack era – were written only for 2D or 3D data. Trying to apply them to 4D or 5D images may then cause an error, or it may simply produce strange results. As commands become updated the chance of dimension-related errors and strangeness reduces, and the stack-hyperstack distinction is disappearing.

2.2.1 Navigating dimensions

With a stack or hyperstack, only a single 2D *slice* is ‘active’ at any one time. Extra sliders at the bottom of the image window are used to change which slice this is (Figure 2.2). In the case of multichannel images, any changes to lookup tables are only made to slices of the currently-active channel.

2.2.2 Correcting dimensions

The dimensions of an image can be seen in the top entries of `Image → Properties...`. Occasionally these can be incorrect: perhaps different z -slices were wrongly interpreted as time points when a file was opened, or the presence of multiple channels was not spotted. This can affect not only the display, but also some processing or measurements. Fortunately, dimensions can be corrected manually using the command `Image → Hyperstack → Stack to Hyperstack...` – provided you know, or can work out, the right values.

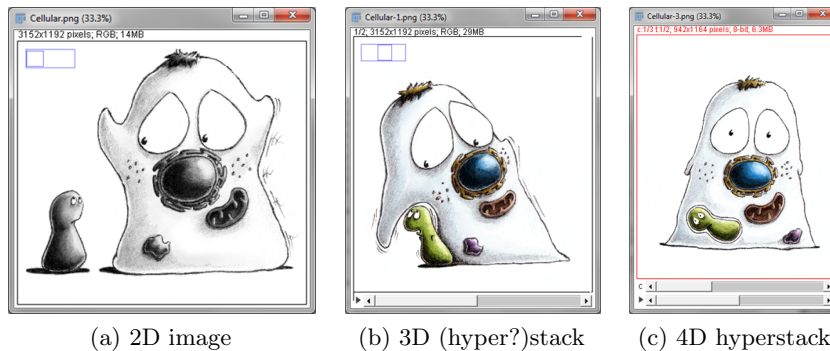


Figure 2.2: Stacks in ImageJ contain 3D data, while hyperstacks contain up to 5 dimensions. Both include additional sliders, not required for 2D images, to shift along the additional dimensions and make different slices active.

Practical 2.1

Something terrible has befallen the file `lost_dimensions.tif`, so that it is displayed as a 3D stack, when in reality it should have more dimensions. By inspecting the file, identify how many channels, z -slices and time points it originally contained, and set these using `Stack to Hyperstack...` so that it displays properly. What are the correct dimensions? *Solution*

2.3 Presenting dimensions

To the computer, an image is stored as a lot of pixel values, irrespective of the number of dimensions it should have. However, as the number of dimensions increases, providing a useful representation of all the values at once becomes tricky. The z dimension is most troublesome of all, because there are relatively natural choices for channels and time points (i.e. to use different colours and to show a movie), so we will concentrate on it.

2.3.1 Viewing angles: your data in a box

It is helpful to consider the pixels of 3D data as being densely packed into a transparent box that could be viewed from different angles (Figure 2.3a). Visualizations like this can be made with `Plugins` → `3D Viewer`. They are particularly good for generating attractive figures or impressive movies, but details can be hard to interpret because they are influenced by perspective and which pixels overlap from our current viewing angle.

To systematically explore data, therefore, it is usually preferable to look inside the box by generating 2D images from only 3 angles: from above (xy) and from two remaining sides (xz and yz). These 3 viewpoints are *orthogonal* (i.e. they

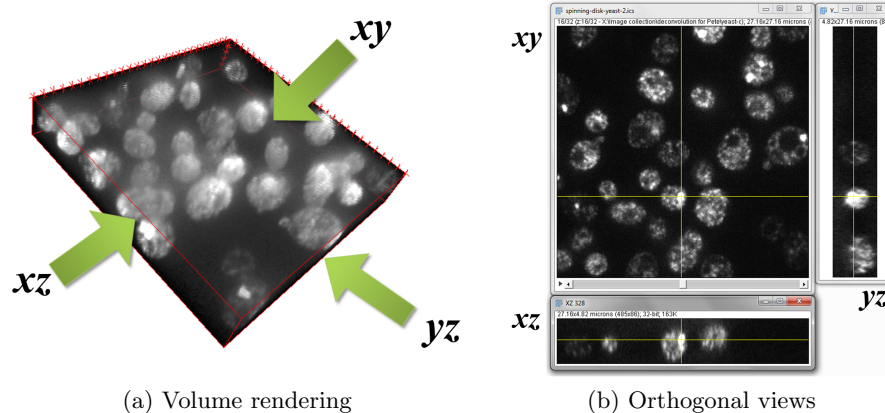


Figure 2.3: Two ways to look at 3D data, using the `3D Viewer` for ‘volume rendering’ and the `Orthogonal Views` command.

are oriented at 90° to one another), and the command `Image` \rightarrow `Stacks` \rightarrow `Orthogonal Views` makes this easy. It opens up 2 extra windows, so that when you click at any point on the original xy view, you are shown cross-sections through that point from each direction.

Reslicing

The `Orthogonal Views` command really only gives you a temporary look at the data from different angles, but you do not have full control over the extra views: you have limited influence over the brightness and contrast, for example, and all your clicks on the images get intercepted to update the display, which means you cannot draw regions of interest (Chapter 7).

If you instead want to rotate the entire stack so that you can browse through what are effectively xz or yz slices and do whatever you want to them, the command you need is `Image` \rightarrow `Stacks` \rightarrow `Reslice` `[/]`....

2.3.2 Z-projections

Another extremely useful way to collapse the data from 3 dimensions into 2 is to use a z -projection. The basic idea is of taking all the pixels in a stack that have the same x and y coordinate, applying some operation to them, and putting the result of that operation into a new 2D image at the same x and y position.

Two important operations in fluorescence imaging are to add all the pixels with the same xy coordinates (a *sum projection*), or to compare the pixels and select only the largest values (a *maximum projection*), both implemented under `Image` \rightarrow `Stacks` \rightarrow `Z Project`.... The advantage of the first is that every pixel value has an influence on the result: which is good if you plan to measure intensities in the projection image (although quantitative analysis of projections can be somewhat

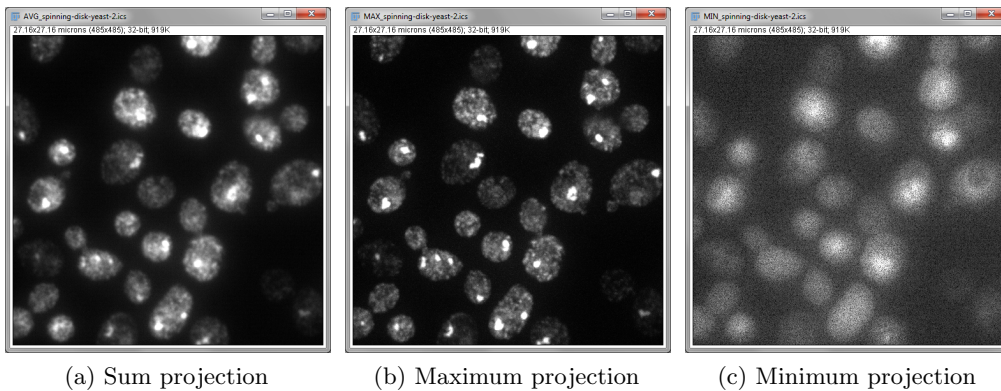


Figure 2.4: Three projections of a z -stack. Sum projections often look similar to maximum projections, but less sharp (a).

dangerous, e.g. if intensity measurements are compared between projections made from stacks with different numbers of slices). The advantage of the second is that it tends to give a nice and sharp looking image, since structures are at their brightest in the planes where they are in focus (Figure 2.4b). Naturally, you could make a *minimum intensity projection* if you liked, but a very out-of-focus-looking image is generally less desirable (Figure 2.4c).

Question 2.1

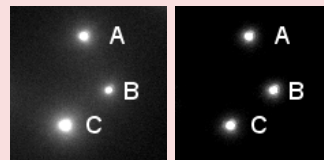
Imagine computing a sum and a maximum projection of a 10-slice stack containing a large, in-focus nucleus. How might each of these projections be affected if your stack contained:

- 4 additional, out-of-focus slices (with non-zero pixel values)
- several very bright, isolated, randomly distributed outlier pixels – with values twice what they should be (due to noise)

Solution

Question 2.2

Shown on the right are sum and maximum projections of an image containing 3 beads: A, B and C. Which projection is which? And which projection, if either, would be suitable for determining the pair of beads that are closest to one another?

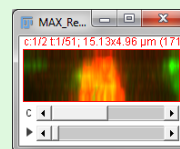
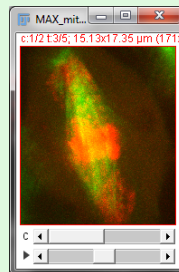
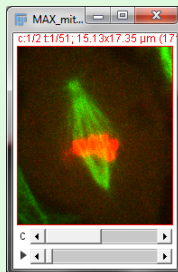


Solution

Practical 2.2

Z-projections are all very well, but how would you make an x , y or $time$ projection?

If you want to try this out, you can use the image File → Open Samples → Mitosis, which has all 5 dimensions to work with.



Max z-projection Max time-projection Max x-projection

Note: Choose File → Open Samples → Cache Sample Images to avoid needing to re-download sample images every time you want them.

Solution

Solutions

Practical 2.1 `lost_dimensions.tif` should contain 2 channels, 3 z -slices and 16 time points. The dimensions are in the default order ($xyzct$).

Question 2.1 Additional, out-of-focus planes will have an effect upon sum projections: increasing all the resulting pixel values. However, the extra planes would have minimal effects upon maximum projections, since they are unlikely to contain higher values than the in-focus planes.

Maximum projections will, however, be very affected by bright outliers: these will almost certainly appear in the result with their values unchanged. Such outliers would also influence a sum projection, but less drastically because each pixel would contain the sum of 9 reasonable values and only 1 large value (unless, by bad luck, many outliers happen to overlap at the same xy coordinate).

Question 2.2 Identifying the projections is tricky since the contrast settings could be misleading, although here they are not... the sum projection is on the left, and the maximum on the right. The sum projection looks less sharp since the regions around the beads contains out-of-focus light, which becomes more obvious when all the slices are added.

As for determining the distance between beads, neither projection is very good. Either could be used to determine the distance in x and y , but if one bead is much, much deeper in the stack then all information about this z displacement would be lost in the projection. This is one reason why it is not good to base analysis on projections alone. `Orthogonal views` would help.

Question 2.2 The `Z Project...` command will also work on time series to make a time projection, assuming there are not extra z -slices present too. If there are, you can use `Image` \rightarrow `Hyperstack` \rightarrow `Re-order Hyperstacks...` or `Stack to Hyperstack...` to switch the dimension names and trick ImageJ into doing what you want.

You can make x and y projections by running `Reslice [/]...` first, then making the projection.

Types & bit-depths

Chapter outline

- *The bit-depth & type of an image determine what pixel values it can contain*
- *An image with a higher bit-depth can (potentially) contain more information*
- *For acquisition, most images have the type unsigned integer*
- *For processing, it is often better to use floating point types*
- *Attempting to store values outside the range permitted by the type & bit-depth leads to clipping*

3.1 Possible & impossible pixels

As described in Chapter 1, each pixel has a numerical value – but a pixel cannot typically have *any* numerical value it likes. Instead, it works under the constraints of the image *type* and *bit-depth*. Ultimately the pixels are stored in some binary format: a series of bits (*binary digits*), i.e. ones and zeros. The bit-depth determines how many of these ones and zeros are available for the storage of each pixel. The type determines how these bits are interpreted.

3.2 Representing numbers with bits

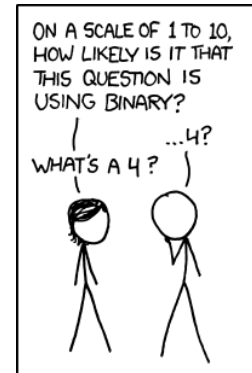
Suppose you are developing a code to store numbers, but in which you are only allowed to write ones and zeros. If you are only allowed a single one or zero, then you can only actually represent two different numbers. Clearly, the more ones and zeros you are allowed, the more unique combinations you can have – and therefore the more different numbers you can represent in your code.

3.2.1 A 4-bit example

The encoding of pixel values in binary digits involves precisely this phenomenon. If we first assume we have a 4 bits, i.e. 4 zeros and ones available for each pixel, then these 4 bits can be combined in 16 different ways:

0000 → 0	0100 → 4	1000 → 8	1100 → 12
0001 → 1	0101 → 5	1001 → 9	1101 → 13
0010 → 2	0110 → 6	1010 → 10	1110 → 14
0011 → 3	0111 → 7	1011 → 11	1111 → 15

Here, the number after the arrow shows how each sequence of bits *could* be interpreted. We do not *have* to interpret these particular combinations as the integers from 0–15, but it is common to do so – this is how binary digits are understood using an *unsigned integer* type. But we could also easily decide to devote one of the bits to giving a sign (positive or negative), in which case we could store numbers in the range $-8 - +7$ instead using precisely the same bit combinations. This would be a *signed integer* type. Of course, in principle there are infinite other variations of how we interpret our 4-bit binary sequences (integers in the range $-7 - +8$, even numbers between 39 to 73, the first 16 prime numbers etc.), but the ranges I’ve given are the most normal. In any case, knowing the type of an image is essential to be able to decipher the values of its pixels from how they are stored in binary.



<http://xkcd.com/953/>

3.2.2 Increasing bit depths

So with 4 bits per pixel, we can only represent 16 unique values. Each time we include another bit, we double the number of values we can represent. Computers tend to work with groups of 8 bits, with each group known as 1 byte. Microscopes that acquire 8-bit images are still very common, and these permit $2^8 = 256$ different pixel values, which understood as unsigned integers fall in the range 0–255. The next step up is a 16-bit image, which can contain $2^{16} = 65536$ values: a dramatic improvement (0–65535). Of course, because twice as many bits are needed for each pixel in the 16-bit image when compared to the 8-bit image, twice as much storage space is required.

3.2.3 Floating point types

Although the images we acquire are normally composed of unsigned integers, we will later explore the immense benefits of processing operations such as averaging or subtracting pixel values, in which case the resulting pixels may be negative or contain fractional parts. *Floating point* type images make it possible to store these new values in an efficient way.

Floating point pixels have variable precision depending upon whether or not they are representing very small or very large numbers. Representing a number in binary using floating point is analogous to writing it out in standard form, i.e. something like 3.14×10^8 . In this case, we have managed to represent 314000000

using only 4 digits: 314 and 8 (the 10 is already known in advance). In the binary case, the form is more properly something like $\pm 2^M \times N$: we have one bit devoted to the sign, a fixed number of additional bits for the exponent M , and the rest to the main number N (called the fraction).

A 32-bit floating point number typically uses 8 bits for the exponent and 23 bits for the fraction, allowing us to store a very wide range of positive and negative numbers. A 64-bit floating point number uses 11 bits for the exponent and 52 for the fraction, thereby allowing both an even wider range and greater precision. But again these require more storage space than 8- and 16-bit images.

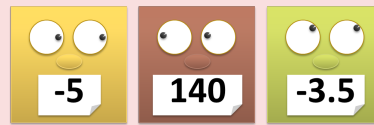
Special values
Floating point images can also contain 3 extra values:

$+\infty$, $-\infty$ and NaN (indicating *Not a Number*: the pixel has no valid value)

Question 3.1

The pixels on the right all belong to different images. In each case, identify what possible types those images could be.

Solution



3.3 Limitations of bits

The main point of Chapter 1 is that we need to keep control of our pixel values so that our final analysis is justifiable. In this regard, there are two main bit-related things that can go wrong when trying to store a pixel value in an image:

1. *Clipping*: We try to store a number outside the range supported, so that the closest valid value is stored instead, e.g. trying to put -10 and 500 into an 8-bit unsigned integer will result in the values 0 and 255 being stored instead.
2. *Rounding*: We try to store a number that cannot be represented exactly, and so it must be rounded to the closest possible value, e.g. trying to put 6.4 in an 8-bit unsigned integer image will result in 6 being stored instead.

3.3.1 Data clipping

Of the two problems, clipping is usually the more serious, as shown in Figure 3.1. A clipped image contains pixels with values equal to the maximum or minimum supported by that bit-depth, and it is no longer possible to tell what values those pixels *should* have. The information is irretrievably lost.

Clipping can already occur during image acquisition, where it may be called *saturation*. In fluorescence microscopy, it depends upon three main factors:

1. *The amount of light being emitted*. Because pixel values depend upon how much light is detected, a sample emitting very little light is less likely to require the ability to store very large values. Although it still might because of...

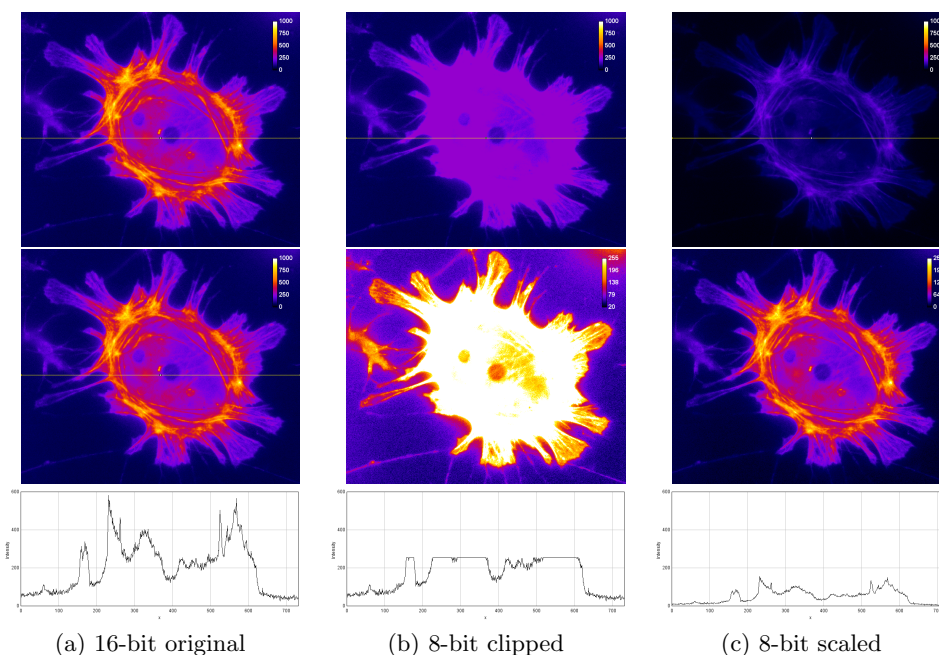


Figure 3.1: Storing an image using a lower bit-depth, either by clipping or by scaling the values. The top row shows all images with the same minimum and maximum values to determine the contrast, while the middle row shows the same images with the maximum set to the highest pixel value actually present. The bottom row shows horizontal pixel intensity profiles through the centre of each image, using the same vertical scales. One may infer that information has been lost in both of the 8-bit images, but more much horribly when clipping was applied. The potential reduction in information is only clear in (c) when looking at the profiles, where rounding errors are likely to have occurred.

2. *The gain of the microscope.* Quantifying very tiny amounts of light accurately has practical difficulties. A microscope's gain effectively amplifies the amount of detected light to help overcome this before turning it into a pixel value (see Chapter 17). However, if the gain is too high, even a small number of detected photons could end up being over-amplified until clipping occurs.
3. *The offset of the microscope.* This effectively acts as a constant being added to every pixel. If this is too high, or negative, it can also push the pixels outside the permissible range.

If clipping occurs, we no longer know what is happening in the brightest or darkest parts of the image – which can thwart any later analysis. Therefore *during image acquisition, any gain and offset controls should be adjusted as necessary to make sure clipping is avoided.*

Question 3.2

When acquiring an 8-bit unsigned integer image, is it fair to say your data is fine so long as you do not store pixel values < 0 or > 255 ? *Solution*

Question 3.3

The bit-depth of an image is probably some multiple of 8, but the bit-depth that a detector (e.g. CCD) can support might not be. For example, what is the maximum value in a 16-bit image that was acquired using a camera with a 12-bit output? And what is the maximum value in a 8-bit image acquired using a camera with a 14-bit output? *Solution*

3.3.2 Rounding errors

Rounding is a more subtle problem than clipping. Again it is relevant as early as acquisition. For example, suppose you are acquiring an image in which there really are 1000 distinct and quantifiable levels of light being emitted from different parts of a sample. These could not possibly be given different pixel values within an 8-bit image, but could normally be fit into a 16-bit or 32-bit image with lots of room to spare. If our image is 8-bit, and we want to avoid clipping, then we would need to scale the original photon counts down first – resulting in pixels with different photon counts being rounded to have the same values, and their original differences being lost.

Nevertheless, rounding errors during acquisition are usually small. Rounding can be a bigger problem when it comes to processing operations like filtering, which often involve computing averages over many pixels (Chapter 10). But, fortunately, at this post-acquisition stage we can convert our data to floating point and then get fractions if we need them.

Floating point rounding errors

Using floating point types does not completely solve rounding issues. In fact, even a 64-bit floating point image cannot store all useful pixel values with perfect precision, and seemingly straightforward numbers like 0.1 are only imprecisely represented. But this is not really unexpected: this binary limitation is similar to how we cannot write $1/3$ in decimal exactly, but rather we can get only get closer and closer for so long as we are willing to add more 3's after the decimal point. Still, rounding 0.1 to 0.100000001490116119384765625 (a possible floating point representation) is not so bad as rounding it to 0 (an integer representation), and the imprecisions of floating point numbers in image analysis are usually small enough to be disregarded.

See <http://xkcd.com/217/> for more information.

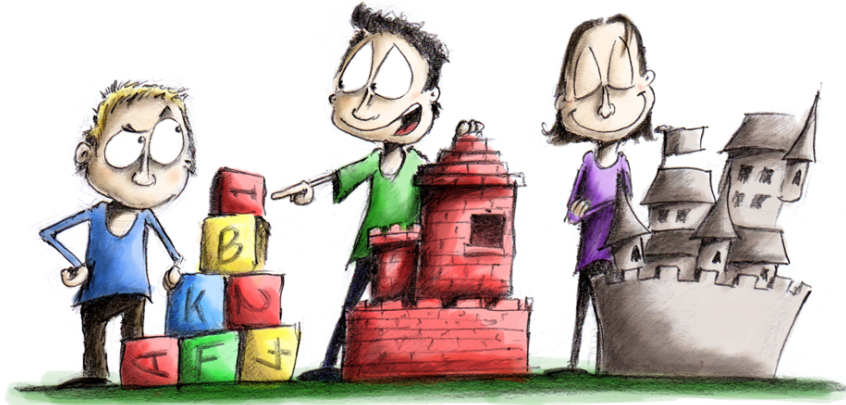


Illustration of the comparative accuracy of (*left to right*) 8-bit, 16-bit and 32-bit images.

Figure 3.2: Building blocks and bit depth. If an 8-bit image is like creating a sculpture out of large building blocks, a 16-bit image is more like using lego and a 32-bit floating point image resembles using clay. Anything that can be created with the blocks can also be made from the lego; anything made from the lego can also be made from the clay. This does not work in reverse: some complex creations can only be represented properly by clay, and building blocks permit only a crude approximation at best. On the other hand, if you only need something blocky, it's not really worth the extra effort of lego or clay. And, from a very great distance, it might be hard to tell the difference.

3.3.3 More bits are better... usually

From considering both clipping and rounding, the simple rule of bit-depths emerges: if you want the maximum information and precision in your images, more bits are better. This is depicted in Figure 3.2. Therefore, when given the option of acquiring a 16-bit or 8-bit image, most of the time you should opt for the former.

Question 3.4

Although *more bits are better* is a simple rule we can share with those who do not really get the subtleties of bit-depths, it should not be held completely rigorously. When might more bits *not* be better?

Solution

3.4 Converting images in ImageJ

For all that, sometimes it is necessary to convert an image type or bit-depth, and then caution is advised. This could be required against your better judgement,

but you have little choice because a particular command or plugin that you need has only been written for specific types of image. And while this could be a rare event, the process is unintuitive enough to require special attention.

Conversions are applied in ImageJ using the commands in the **Image** → **Type** → ... submenu. The top three options are **8-bit** (unsigned integer), **16-bit** (unsigned integer) and **32-bit** (floating point), which correspond to the types currently supported¹.

In general, *increasing* the bit-depth of an image should not change the pixel values: higher bit-depths can store all the values that lower bit-depths can store. But going backwards that is not the case, and when *decreasing* bit-depths one of two things can happen depending upon whether the option **Scale When Converting** under **Edit** → **Options** → **Conversions...** is checked or not.

- **Scale When Converting** is *not* checked: pixels are simply given the closest valid value within the new bit depth, i.e. there is clipping and rounding as needed.
- **Scale When Converting** *is* checked: a constant is added or subtracted, then pixels are further divided by another constant before being assigned to the nearest valid value within the new bit depth. Only *then* is clipping or rounding applied if it is still needed.

Perhaps surprisingly, the constants involved in scaling are determined from the **Minimum** and **Maximum** in the current **Brightness/Contrast...** settings: the **Minimum** is subtracted, and the result is divided by **Maximum - Minimum**. Any pixel value that was lower than **Minimum** or higher than **Maximum** ends up being clipped. Consequently, *converting to a lower bit-depth with scaling can lead to different results depending upon what the contrast settings were.*

Question 3.5

Why is scaling *usually* a good thing when reducing the bit-depth, and why is a constant usually subtracted before applying this scaling?

Hint: As an example, consider how a 16-bit image containing values in the range 4000–5000 might be converted to 8-bit first without scaling, and then alternatively by scaling with or without the initial constant subtraction. What constants for subtraction and division would usually minimize the amount of information lost when converting to 8-bit image, limiting the errors to rounding only and not clipping?

Solution

¹The remaining commands in the list involve colour, and are each variations on 8-bit unsigned integers (see Chapter 4).

Practical 3.1

Make sure that the **Scale when Converting** option is turned on (it should be by default). Then using a suitable 8-bit sample image, e.g. **File** → **Open Samples** → **Boats**, explore the effects of brightness/contrast settings when increasing or decreasing bit-depths. How should the contrast be set before reducing bit-depths? And can you destroy the image by simply increasing then decreasing the bit-depth?

Solution

Solutions

Practical 3.1 The possible image types, from left to right:

1. Signed integer or floating point
2. Unsigned integer, signed integer or floating point
3. Floating point

Question 3.2 No! At least, not really.

You *cannot* store pixels outside the range 0–255. But if your image contains pixels with either of those extreme values, you cannot be sure whether or not clipping has occurred. Therefore, you should ensure images you acquire do not contain any pixels with the most extreme values permitted by the image bit-depth. If you want to know for sure you can trust your 8-bit data is not clipped, the maximum range would be 1–254.

Question 3.3 The maximum value of a 16-bit image obtained using a 12-bit camera is 4095 (i.e. $2^{12}-1$). The maximum value of an 8-bit image obtained using a 14-bit camera is 255 – the extra bits of the camera do not change this. But if the image was saved in 16-bit instead, the maximum value would be 16383.

So be aware that the actual range of possible values depends upon the acquisition equipment as well as the bit-depth of the image itself. The lower bit-depth will dominate.

Question 3.4 Reasons why a lower bit depth is *sometimes* preferable to a higher one include:

- A higher bit-depth leads to larger file sizes, and potentially slower processing. For very large datasets, this might be a bigger issue than any loss of precision found in using fewer bits.
- The amount of light detected per pixel might be so low that thousands of possible values are not required for its accurate storage, and 8-bits (or even fewer) would be enough. For the light-levels in biological fluorescence microscopy, going beyond 16-bits would seldom bring any benefit.

But with smallish datasets for which processing and storage costs are not a problem, it is safest to err on the side of more bits than we strictly need.

Question 3.5 In the example given, converting to 8-bit without any scaling would result in all pixels simply becoming 255: all useful information in the image would be lost.

With scaling but without subtraction, it would make sense to divide all pixel values by the maximum in the image divided by the maximum in the new bit depth, i.e. by $5000/255$. This would then lead to an image in which pixels fall into the range 204–255. Much information has clearly been lost: 1000 potentially different values have now been squeezed into 52.

However, if we first subtract the smallest of our 16-bit values (i.e. 4000), our initial range becomes 0–1000. Divide then by $1000/255$ and the new values become scaled across the full range of an 8-bit image, i.e. 0–255. We have still lost information – but considerably less than if we had not subtracted the constant first.

Practical 3.1 It is a good idea to choose `Reset` in the `Brightness/Contrast...` window before reducing any bit-depths for 2D images (see Section 12.2 for more dimensions).

You can destroy an image by increasing its bit-depth, adjusting the brightness/contrast and then decreasing the bit-depth to the original one again. This may seem weird, because clearly the final bit-depth is *capable* of storing all the original pixel values. But ImageJ does not know this and does not check, so it will simply do its normal bit-depth-reducing conversion based on contrast settings.

Channels & colours

Chapter outline

- *Images with multiple colour channels may display differently in different software*
- *RGB images are a special case, and these look consistent across different software*
- *In ImageJ, multichannel images that are not RGB may be referred to as composite images*
- *Conversion of composite to RGB often loses information!*

4.1 Different kinds of colour image

One way to introduce colour into images is to use a suitable LUT, as described in Chapter 1. However, then the fact that different colours could be involved in the display of such images was really only incidental: at each location in the image there was still only one channel, one pixel and one value.

For images in which colour is more intrinsic – that is, when the channel number is an extra dimension (Chapter 2) and we want to display channels superimposed on top of one another – things become more complex, because the precise colour in the display for one location now depends upon a combination of multiple pixel values mixed together.

There are two main types of colour image we will consider in this chapter, and it is important to know the difference between them:

1. *Multichannel / composite images* – good for analysis, since they can contain the original pixels given by the microscope, but may appear differently (or not be readable at all) in some software
2. *RGB images* – good for display, because they have a consistent appearance, but often unsuitable for quantitative analysis because original pixel values are likely to be lost

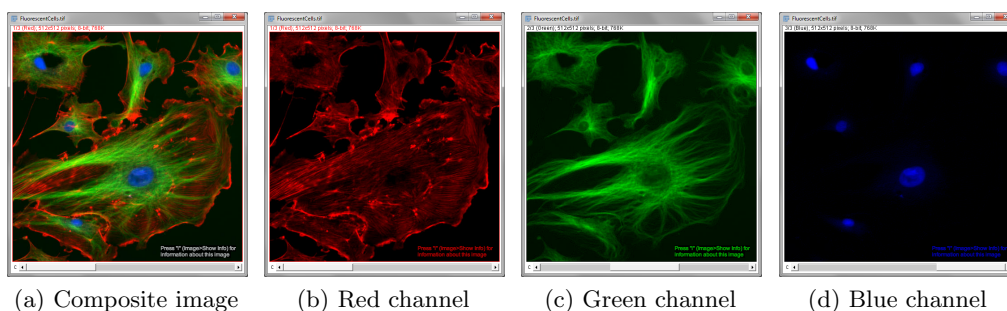


Figure 4.1: ImageJ composite image sample **Fluorescent Cells**. Using the **Channels Tool...** and the slider at the bottom of the window, you can view the channels individually or simultaneously.

4.2 Multichannel images

We consider a ‘true’ multichannel image here to be one in which the channel number is an extra dimension. Pixel values for each channel are often determined from light that has been filtered according to its wavelength. In principle, any LUT might be applied to each channel, but it makes sense to choose LUTs that somehow relate to the wavelength (i.e. colour) of light detected for the corresponding channels. Channels can then be overlaid on top of one another, and their colours further merged for display (e.g. high values in green and red channels are shown as yellow).

The important feature of these images is that the actual channel information is always retained, and so the original pixel values remain available. This means you can still extract channels or adjust their LUTs as needed.

4.2.1 Composite images in ImageJ

In ImageJ, such representations of multiple channels are sometimes known as *composite images*. An example can be opened by selecting **File** → **Open Samples** → **Fluorescent Cells (400K)** (Figure 4.1). As you move the slider at the bottom of the image, it might not look like much is happening. But if you also open the **Brightness/Contrast...** tool you can see that the colour of the histogram changes for each slider position. Adjusting the contrast then adjusts it *only for the current channel*. This is very useful because quite different contrast settings can be required for each channel to get a decent final appearance. Also, as you move the mouse over the image the ‘value’ shown in the status bar is the pixel value *only for that channel*.

Composite images allow us to see multiple channels at the same time. But sometimes this masks information and it helps to look at each channel individually. One way to do this is to choose **Image** → **Color** → **Split Channels**, which will

give you separate images for each channel. But a more powerful option is to select `Image` → `Color` → `Channels Tool...`

Practical 4.1

Using the `Fluorescent Cells (400K)` image, explore the use of the `Channels Tool...` until you are comfortable with what it does, including what happens when you click on the colours under `More>>`.

How do composite images relate to the LUTs we described before? Try out some of the commands in `Image` → `Lookup Tables` → when investigating this.

Solution

4.3 RGB images

Composite images can contain any number of channels, and these channels can have any of the bit-depths ImageJ supports (see Chapter 3) – so long as all channels have the same bit-depth. By contrast, RGB images invariably have 3 channels, and each channel is 8-bit¹. Furthermore, the channel colours in an RGB image are fixed to *red*, *green* and *blue*.

This already makes clear that composite images can (at least potentially) contain much more information. However, the inflexibility of RGB images has one important advantage: compatibility. Computer monitors generate colours for display by mixing red, green and blue light. Therefore RGB images can dictate directly how much of each colour should be used; the values in each channel really do determine the final image appearance, without any ambiguity regarding which LUTs should be applied because these are already known. Therefore all software can display RGB images in the same way.

4.3.1 RGB images in ImageJ

RGB images can be easily distinguished from composite images in ImageJ both because they do not have an additional slider at the bottom to move between channels, and because the text `RGB` appears at the top of the image window in place of the bit-depth. Also, moving the cursor over an RGB image leads to 3 numbers following the `value` label in the main status bar: the pixel values for each of the channels.

Changing the brightness/contrast of an RGB image is also different. Making adjustments using the normal `Brightness/Contrast...` tool leads to the appearance of all 3 channels being affected simultaneously. And should you

¹This is the case in ImageJ, and usually (but not always) in other software. For example, it is possible for an RGB image to be 16-bit, and some contain an extra ‘alpha’ channel (which relates to transparency), and so might be called ARGB or RGBA. But such images do not often occur in science.

happen to click another image before returning to the original RGB image, you will find that any contrast settings have been permanently applied – and the original data probably cannot be restored!

4.4 Comparison of colour images

Only composite images can therefore store more than three channels, or data with a bit-depth higher than 8. Consequently, if acquisition software provides the option to save data in an RGB format, this temptation should normally be resisted – unless you *also* save the image in the software manufacturer’s preferred, non-RGB format, and only use this latter version for analysis. The reason is that the conversion of a composite image to RGB *can* result in a significant loss of information, and if we only have the RGB version it may be impossible for us to say whether or not this loss has occurred.

Practical 4.2

You can test the compatibility of composite and RGB images by opening **File** → **Open Samples** → **HeLa Cells** image, and saving it using **File** → **Save As** → **Tiff...** both before and after running **Image** → **Type** → **RGB Color** to convert the original 16-bit composite data to RGB. Try then opening both saved images in other software (e.g. Microsoft PowerPoint) and compare their appearance.

Practical 4.3

Think of at least 2 ways in which converting a composite image to RGB can lose information.

You can explore this by comparing the images `Cell_composite.tif` and `Cell_RGB.tif`. The **Image** → **Color** → **Split Channels** command should help.

Solution

Still, there is no way of *displaying* a composite image that cannot be replicated by an RGB image, because the monitor itself works with RGB data (translated to red, green and blue light). Therefore for creating figures or presentations, converting data to RGB is a very *good* idea for the compatibility advantages it brings. In the end, it is normal to need to keep at least two versions of each dataset: one in the original (multichannel / composite) format, and one as RGB for display. This RGB image is normally created as the *final* step, after applying any processing or LUT adjustments to the original data.

4.4.1 Other colour spaces

Since monitors work with RGB images, in practice what you are actually seeing on screen is always an RGB image – even when we are actually working with the data

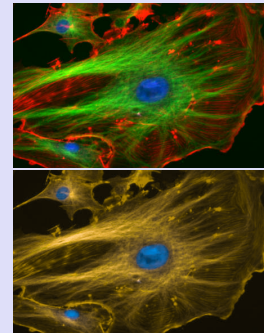
in a composite image. The RGB version is just quietly made in the background based on the composite data so that the monitor can give us something to look at. If we convert the composite image to RGB we then basically just throw away all the extra useful information the composite contained, and keep only the displayed version.

However, there are more ways to represent colours than just mixtures of red, green and blue light. One familiar example is using cyan, magenta, yellow and black ink – leading to a CMYK representation of colour. In practice, fewer colours can be faithfully reproduced using CMYK when compared to RGB, which is one reason why printed images often do not seem to have quite the same colours as the same images on screen.

The other main colour representation that turns up in ImageJ is HSB, which stands for *Hue*, *Saturation* and *Brightness*. An RGB image can be converted to HSB using `Image` → `Type` → `HSB Stack`. In practice, this representation can be useful for detecting features in colour images that were originally RGB (e.g. conventional photographs), but it is largely irrelevant for fluorescence microscopy.

Choosing channel colours

By changing the LUTs, channels can be assigned any colour you like. Although red/green images are widespread, especially for colocalization, they are particularly unhelpful for colourblind people. More accessible images can be created by switching the LUTs of at least one channel to something more suitable (e.g. Red → Magenta) – although displaying both channels separately is better still.



You can also test the effects of different colours using the Fiji command `Image` → `Color` → `Simulate Color Blindness` (note that this may require converting the image to RGB first).

More information about generating figures with suitable colours is available at <http://www.nature.com/nmeth/journal/v8/n6/full/nmeth.1618.html>.

Question 4.1

ImageJ has two different copying commands, `Edit` → `Copy` and `Edit` → `Copy to System`. Why?

Note: When investigating this, you should explore `Edit` → `Paste`, alongside two commands in the `File` → `New` → submenu. Be on the lookout for any conversions.

Solution

Solutions

Practical 4.1 Each individual channel in a composite image is really just being shown with a LUT – most often one that is predominantly red, green or blue. The final displayed colour results from mixing the channel colours together. Therefore, in principle any LUT might be used for each channel. But unconventional LUT choices can easily result in rather avant garde images that are difficult to interpret.

Practical 4.3 One way that converting an image to RGB can lose information is if the original data is 16-bit before conversion and 8-bit afterwards, causing rounding and/or clipping to occur (Chapter 3).

Another is that during conversion the channels have been forced to be purely red, green and blue. But perhaps they were not originally so, and separate channels have been merged into the same ones. If you had started with more than three channels, this will definitely have occurred, but even if you had fewer channels you can have problems (like in the `Cell_composite.tif` and `Cell_RGB.tif` examples).

The trouble with *only* having an RGB image is that it is no longer possible to know for sure what the original data looked like, to be able to figure out whether anything has been lost. For example, perhaps you have managed to generate a magnificent image consisting of turquoise, yellow and dazzling pink channels. Each pixel is displayed on screen as a combination of those colours. However, precisely the same colour for any pixel can also be represented – rather more simply – as a mixture of red, green and blue. So no matter what channels and colours you began with, the final result after merging can be replicated using red, green and blue channels. But if you *only* have the RGB version, you might never be able to extract the original 3 channels again. Their colours could be so mixed together that the original pixel values would be irretrievably lost.

And so the good things about RGB images is that they look identical to the original image you had before conversion, and other software (e.g. webbrowsers or PowerPoint) can understand them effortlessly. But the *potentially very bad* thing about RGB images is that creating them requires conversion, and after this it might very well be *impossible* to regain the original pixel values.

Question 4.1 Copy and Copy to System have quite distinct purposes. The former allows you to copy part of an individual image slice (i.e. a 2D part of a particular colour channel) and paste it into another image while exactly preserving the original image type and pixel values. However, this image type might well not be compatible with other software, therefore images copied in this way cannot be accessed outside ImageJ. If you want to copy an image and paste it into other software, you need Copy to System. This copies the image to the system clipboard, converting it to RGB in the process, thereby preserving appearance – but not necessarily the pixel values.

Files & file formats

Chapter outline

- *Image files consist of pixel values and metadata*
- *Some file formats are suitable for data to analyze, others for data only to display*
- *Metadata necessary for analysis can be lost by saving in a non-scientific file format*
- *Compression can be either lossless or lossy*
- *Original pixel values can be lost through lossy compression, conversion to RGB, or by removing dimensions*

5.1 The contents of an image file

An image file stored on computer contains two main things:

1. *Image data* – the pixel values (numbers, only numbers)
2. *Metadata* – additional information, such as dimensions, image type, bit-depth, pixel sizes and microscope settings (‘data about data’)

The image data is clearly important. But some pieces of the metadata are essential for the image data to be interpretable at all. And if the metadata is incorrect or missing, measurements can also be wrong. Therefore files must be saved in formats that preserve both the pixel values and metadata accurately if they are to be analyzable later.

5.2 File formats

When it comes to saving files, you have a choice similar to that faced when working with colour images: you can either save your data in a file format good for analysis or good for display. The former requires a specialist microscopy/scientific format that preserves the metadata, dimensions and pixels exactly. For the latter, a general file format is probably preferable for compatibility.

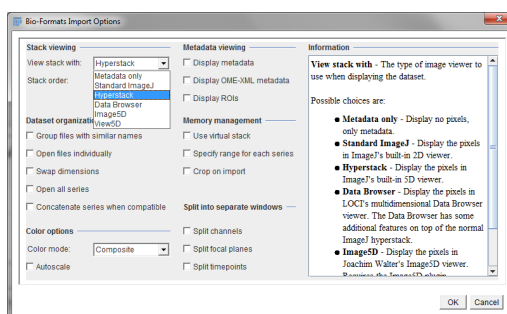


Figure 5.1: Options when opening a file using the LOCI Bioformats importer.

5.2.1 Microscopy file formats (*ICS, OME-TIFF, LIF, LSM, ND2...*)

Fortunately, much useful metadata is normally available stored within freshly-acquired images. Unfortunately, this usually requires that the images are saved in the particular format of the microscope or acquisition software manufacturer (e.g. ND2 for Nikon, LIF for Leica, OIB or OIF for Olympus, LSM or ZVI for Zeiss). And it is quite possible that other software will be unable to read that metadata or, worse still, read it incorrectly or interpret it differently.

In Fiji, the job of dealing with most proprietary file formats is given to the LOCI Bioformats plugin¹. This attempts to extract the most important metadata, and, if you like, can display its findings for you to check. When opening a file using Bioformats, the dialog box shown in Figure 5.1 appears first and gives some extra options regarding how you want the image to be displayed when it is opened.

In general, Bioformats does a very good job – but it does not necessarily get all the metadata in all formats. This is why...

You should always keep the originally acquired files – and refer to the original acquisition software to confirm the metadata

The original files should be trustworthy most of the time. But it remains good to keep a healthy sense of paranoia, since complications can still arise: perhaps during acquisition there were values that ought to have been input manually (e.g. the objective magnification), without which the metadata is incorrect (e.g. the pixel size is miscalculated). This can lead to *wrong* information stored in the metadata from the beginning. So it is necessary to pay close attention at an early stage. While in many cases an accurate pixel size is the main concern, other metadata can matter sometimes – such as if you want to estimate the blur of an image (Chapter 15), and perhaps reduce it by deconvolution.

The messy state of metadata

Despite its importance, metadata can be annoyingly slippery and easy to lose.

¹This can also be added to ImageJ, see <http://loci.wisc.edu/software/bio-formats>

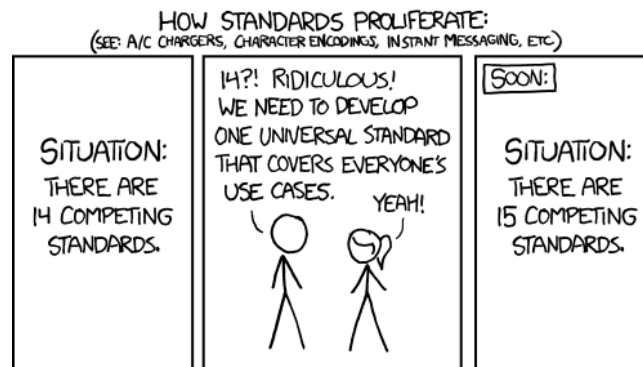


Figure 5.2: From <http://www.xkcd.com/927/>

New proprietary file formats spring up (Figure 5.2), and the specifications for the formats might not be released by their creators, which makes it very difficult for other software developers to ensure the files are read properly.

One significant effort to sort this out and offer a standardized format is being undertaken by the Open Microscopy Environment (OME), which is often encountered in the specific file format OME-TIFF. While such efforts give some room for hope, it has not yet won universal support among software developers.

5.2.2 General file formats (*JPEG, PNG, MPEG, AVI, TIFF...*)

If you save your data in a general format, it should be readable in a lot more software. However, there is a fairly good chance your image will be converted to 8-bit and/or RGB colour, extra dimensions will be thrown away (leaving you with a 2D image only) and most metadata lost. The pixel values might also be compressed – which could be an even bigger problem.

5.2.3 Compression

There are two main categories of compression: *lossy* and *lossless*.

1. *Lossy compression* (e.g. JPEG) saves space by ‘simplifying’ the pixel values, and converting them into a form that can be stored more concisely. But this loses information, since the original values cannot be perfectly reconstructed. It may be fine for photographs if keeping the overall appearance similar is enough, but is terrible for any application in which the exact values matter.
2. *Lossless compression* (e.g. PNG, BMP, most types of TIFF, some microscopy formats) saves space by encoding the patterns in the image more efficiently, but in such a way that the original data can be perfectly reconstructed. No information is lost, but the reductions in file size are usually modest.

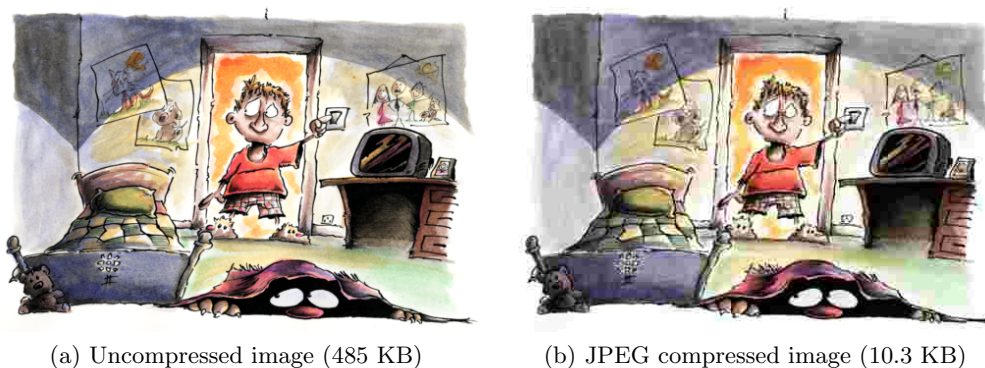


Figure 5.3: The effect of strong JPEG compression on colour images. The reduction in file size can be dramatic, but the changes to pixel values are irreversible. If you have an eerie sense that something might be wrong with an image, or the file size seems too good to be true, be sure to look closely for signs of little artificial square patterns that indicate that it was JPEG compressed once upon a time. By contrast, saving (a) as a PNG (which uses lossless compression) would give an image that looks identical to the original, but with a file size of 366 KB.

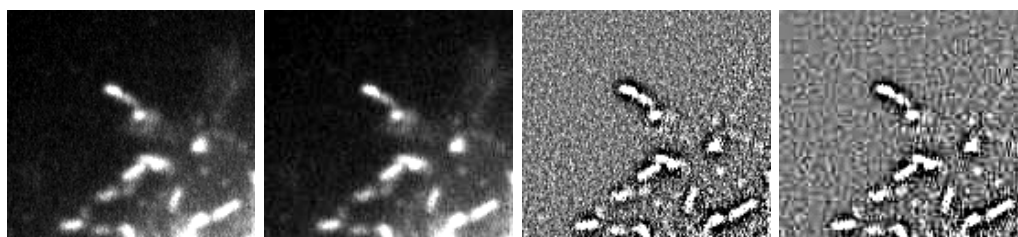


Figure 5.4: Using lossy compression can make some analysis impossible. The saved images in (a) and (b) look similar. However, after subtracting a smoothed version of the image, the square artifacts signifying the effects of JPEG compression are clearly visible in (d), but absent in (c). In later analysis of (b), it would be impossible to know the extent to which you are measuring compression artifacts instead of interesting phenomena.

Therefore lossless compression should not cause problems, but lossy compression really can. The general rule is

If in doubt, avoid compression – and really avoid JPEG

JPEG gets a special mention because it is ubiquitous, and its effects can be particularly ugly (Figure 5.3). It breaks an image up into little 8×8 blocks and then compresses these separately. Depending upon the amount of compression, you might be able to see these blocks clearly in the final image (Figure 5.4).

Question 5.1

Suppose you have an original image in TIFF format (no compression). First, you save it as JPEG (lossy compression) to reduce the file size, then close it and throw away the TIFF. Upon hearing JPEG is bad, you reopen the image and save it as TIFF once more (no compression), throwing away the JPEG. How does your final TIFF image look, and what is its file size? *Solution*

Compressed TIFFs...?

The reality of file formats is slightly muddier than this brief discussion might imply. For example, TIFF (like AVI) is really a container for data, so that a TIFF file *can* store an image that has been compressed using a lossy technique (but it usually won't), and there is also such a thing as a lossless JPEG. So in times of doubt it is wise to be cautious, and stick with specialized microscopy file formats for most data. Nevertheless, when it comes to creating movies, which should be looked at but not analyzed, lossy compression is probably needed to get reasonable file sizes.

Additionally, you can take *any* file and then compress it losslessly later, e.g. by making a ZIP archive from it. ImageJ can directly open TIFF files that have been compressed this way. It can even write them using the **File** → **Save As** → **ZIP...** command. This is safe, but if you want to open the image in another program you will probably have to unzip it first.

5.2.4 Choosing a file format

Table 5.1 summarizes the characteristics of some important file formats. Personally, I use ICS/IDS if I am moving files between analysis applications that support it, and TIFF if I am just using ImageJ or Fiji. My favourite file format for most display is PNG², while I use PDF where possible for journal figures.

²If you ever email a Windows Bitmap file (BMP) to the sort of person who has a 'favourite file format', they will judge you for it, and possibly curse you for the time it takes to download. Unless, perhaps, that person was the designer of the BMP format. It is not well-compressed like JPEG or PNG, and nor does it contain much of the useful extra information a TIFF can store.

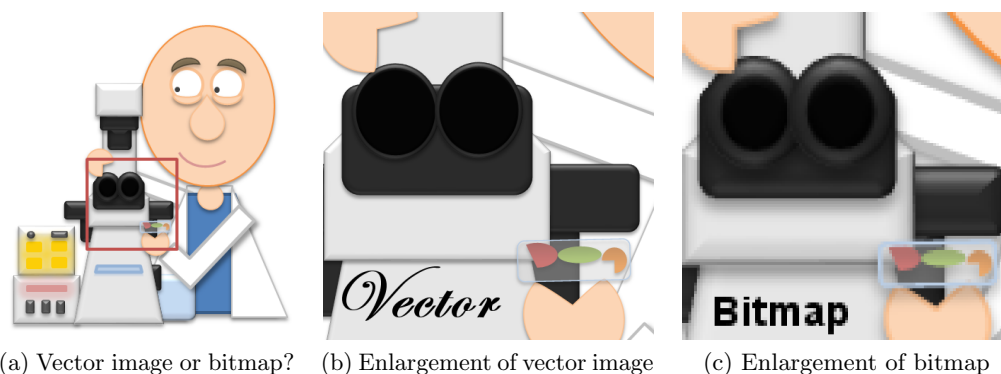


Figure 5.5: When viewed from afar, it may be difficult to know whether an image is a vector or a bitmap (a) because they can sometimes look identical (although a photograph or micrograph will always be a bitmap). However, when enlarged a vector image will remain sharp (b), whereas a bitmap will not (c).

File formats for creating figures

Preparing figures for publication can be a bewildering process. To begin with, it is necessary to know about two different types of image, one of which has not been discussed here so far:

- *Bitmaps*. These are composed of individual pixels: e.g. photographs, or all the microscopy images we are concerned with here.
- *Vector images*. These are composed of lines, curves, shapes or text. The instructions needed to draw the image (i.e. coordinates, equations, fonts) are stored rather than pixels, and then the image is recreated from these instructions when necessary.

If you scale a 2D bitmap image by doubling its width and height, then it will contain four times as many pixels. Guesses need to be made about how to fill in the extra information properly (which is the problem of *interpolation*), and the result generally looks less sharp than the original. But if you double the size of a vector image, it is just a matter of updating the maths needed to draw the image accordingly, and the result looks just as sharp as the original.

Vector images are therefore best for things like diagrams, histograms, plots and charts, because they can be resized freely and still look good. Also, they often have tiny file sizes because only a few instructions to redraw the image need to be kept, whereas a huge number of pixels might be required to store sufficiently nice, sharp text in a bitmap. But bitmaps are needed for images formed from detecting light, which cannot be reduced to a few simple equations and instructions.

Finally, some versatile file formats, such as PDF or EPS, can store both kinds of image: perhaps a bitmap with some text annotations on top. If you are including text or diagrams, these formats are generally best. But if you only have bitmaps without annotations of any kind, then TIFF is probably the most common file format for creating figures.

5.3 Dealing with large datasets

Large datasets are always troublesome, and we will not be considering datasets of tens or hundreds of gigabytes here. But when a file's size is lurking beyond the boundary of what your computer can comfortably handle, one trick for dealing with this is to use a *virtual stack*.

Chapter 2 mentioned the (in practice often ignored) distinction between *stacks* and *hyperstacks*. Virtual stacks are a whole other idea. These provide a way to browse through large stacks or hyperstacks without needing to first read all the data into the computer's memory. Rather, only the currently-displayed 2D image slice is shown. After moving the slider to look at a different time point, for example, the required image slice is read from its location on the hard disk at that moment. The major advantage of this is that it allows the contents of huge files to be checked relatively quickly, and it makes it possible to peek into datasets that are too large to be opened completely.

The disadvantage is that ImageJ can appear less responsive when browsing through a virtual stack because of the additional time required to access an image on the hard disk. Also, be aware that if you process an image that is part of a virtual stack, the processing will be lost when you move to view another part of the dataset!

Virtual stacks are set up either by **File** → **Import** → **TIFF Virtual Stack...** or choosing the appropriate checkbox when the LOCI Bioformats plugin is used to open an image (see Figure 5.1).

Question 5.2

Virtual stacks are one way to avoid out-of-memory errors when dealing with large datasets. But assuming that you can open an image completely in the normal way, what other ways might you be able to reduce memory requirements during analysis?

Note: You can base your answer here upon exploring ImageJ's menus, or experience in using other software. Some methods involve a loss of information, but this might be acceptable, depending on the application. *Solution*

Format	Comments
ICS/IDS	A quite simple format designed for microscopy images, in which the metadata is stored in a text file (<code>.ics</code>) and the image data in a separate file (<code>.ids</code>). Both files should always be kept together in the same directory. The big advantage of this is that <i>all</i> the metadata can be read – and even edited – simply by opening it in a text editor (e.g. Wordpad on Windows, TextEdit on a Mac). Not everything may be stored in the metadata, but at least you can always know precisely what is.
ICS2	Similar to ICS/IDS, but the text and image data are stored in the same file (<code>.ics</code>). While avoiding the need to keep two files together, this loses the main advantage of being able to easily read the metadata in a text editor.
TIFF	ImageJ's default format for saving. Can be read by some other software, but the metadata might not be correctly understood.
OME-TIFF	The product of an ongoing attempt to standardize the storage of microscopy images and metadata. See http://www.openmicroscopy.org/
JPEG	General, compressed image format for single-channel or RGB colour images. Good for photos, bad for science.
PNG	Suitable for the same types of images as JPEG, but always using lossless compression. Good for presentations, websites and was used for most figures in this very document.
TIFF	The kind of TIFFs written by software like Photoshop are appropriate for journal figures containing bitmaps. The file size might be larger than for the equivalent PNG.
PDF	A handy format for incorporating both bitmap images and vector annotations. But take care with the bitmaps, which may or may not be lossily compressed. PDFs usually look the same no matter which computer is displaying them. Good for journal figures.
AVI	Only for showing movies – especially on Windows, but should work on other computers. ImageJ can cope best with uncompressed AVIs. If trying to decipher a file made by some uncommon format, you can try the free (Windows) software VirtualDub (http://www.virtualdub.org/) to convert it. Handbrake (http://handbrake.fr/) is alternative free software for converting movies into different formats, and is available for Windows, Mac and Linux.
Quicktime (MOV)	Better supported than AVI on the Mac, but might not work on Windows unless Quicktime is installed.

Table 5.1: Some personal thoughts on file formats.

Practical 5.1

To get a feel for the importance of metadata, you can try opening an image in which it is completely absent. This is quite tricky, and requires some detective work (or some luck).

Try to open the file `Besenfreunde.ids` using Fiji – it depicts an odd, and as yet unexplained, scene that I passed on my way to work soon after arriving in Heidelberg. This file *only* contains pixel values, and no metadata. It can still be read using the `File` → `Import` → `Raw...` command, but to do so you will need to figure out the necessary metadata and input the appropriate values.

The following points may help:

- The file contains only a single image, and a single channel.
- The dimensions (width and height) of the image are each a multiple of 100 pixels.
- The data is in 8, 16 or 32-bit format (signed or unsigned).
- There are no offsets or gaps included.
- The total size of the file is 400 000 bytes.

Note: The option `Little-endian byte order` relates to whether the bytes of 16 or 32-bit images are arranged from the least-to-most significant, or most-to-least significant. Some types of computer prefer one order, some prefer another, so this is something else the metadata should contain. The difference is similar to how a perfectly respectable number like *twenty-three* is (quite ludicrously) spoken as *three-and-twenty* in German. *Solution*

Solutions

Question 5.1 The final image will look exactly like the JPEG version, but with the same file size as the original TIFF. As such, it has ‘the worst of both worlds’.

Question 5.2 Here are some suggestions for working with large files, not all of which would be appropriate in all circumstances:

1. Decrease the image bit-depth (often a bad idea, but possibly ok if exact pixel values are not so important)
2. Crop the image (using the LOCI Bioformats plugin, this can be done as the file is being opened; otherwise, see `Image` → `Crop`)
3. Separate channels and process each independently (see `Image` → `Color` → `Split Channels`)
4. Resize (i.e. scale down) the image (see `Image` → `Scale...`)

Practical 5.1 The file size gives you the

$$\text{File size (in bytes)} = \frac{\text{width} \times \text{height} \times \text{bit-depth}}{8}$$

where the division by 8 converts the total number of bits to bytes (since 8 bits make 1 byte). This can be used to make reasonable starting estimates for the width, height and bit-depth, but figuring out which are correct likely still requires some trial-and-error. In the end, the settings you need are:

- Image type: 16-bit unsigned
- Width: 500 pixels
- Height: 400 pixels
- Little-endian byte order: False

Now make sure never to lose your metadata, and you do not need to solve such puzzles often in real life. (Also, any explanations for what exactly was going on in that scene would be welcome.)

Part II

Processing fundamentals

Overview: Processing & Analysis

Chapter outline

- *Image analysis usually consists of three main stages: preprocessing, detection & measurement*
- *The same basic techniques reoccur in most analysis, with their combinations & parameters tailored to the particular task*

6.1 The goal of analysis

Successfully extracting useful information from fluorescence images usually requires triumphing in two main battles. The first is to overcome limitations in image quality and make the really interesting image content more clearly visible. This involves *image processing*, the output of which is another image. The second battle is to compute meaningful measurements. This is *image analysis*. Except when creating beautiful figures, the ultimate goal we are most interested in here is analysis – but processing is almost always indispensable to get us there.

One way to approach image analysis is to see it like a puzzle. In the end, one hopes to extract some kind of quantitative measurements that are justified by the nature of the experiment and the facts of image formation, but there is no fixed way to go about that. This liberating realization suggests there is room for lateral thinking and sparks of creativity. Admittedly, if no solution comes to mind after pondering for a while then such an optimistic outlook quickly subsides and the ‘puzzle’ may very well turn into an unbearably infuriating ‘problem’ – but the point here is that *in principle* image analysis *can* be enjoyable. All it takes is properly-acquired data, a modicum of enthusiasm, and the good luck not to be working on something horrendously difficult.

Despite the diversity of algorithms¹ that could be constructed to analyze an image, in practice they generally tend to be built up from the following three stages:

1. *(Pre)processing*, e.g. subtract the background, use a filter to reduce noise
2. *Detection*, e.g. apply a threshold to locate interesting features, refine the detection

¹An ‘algorithm’ can be considered to be simply a ‘sequence of steps’.

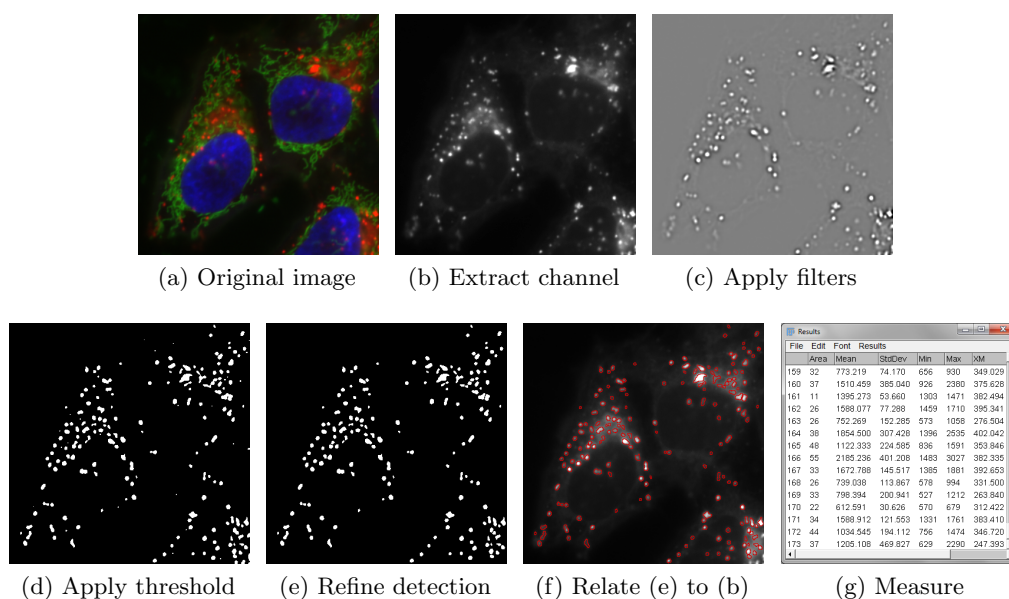


Figure 6.1: An simple image analysis workflow for detecting and measuring small spots, applied to the red channel of the sample image HeLa Cells.

3. *Measurement*, e.g. count pixels to determine areas or volumes, quantify intensities

Figure 6.1 shows an example of how these can fit together.

This part provides a tour of many of the fundamental techniques that may be used for each of the three stages. Armed with only these techniques, a vast amount is already achievable: the challenge is to figure out how to string them together for whatever application you encounter. But even if you ultimately end up doing your analysis with someone else’s automated software or plugin, knowing the main building blocks of image processing and analysis can still help explain what the algorithm you use is really doing and why.

The following chapters deal primarily with 2D images. Chapter 12 then briefly discusses how the techniques can be extended into more dimensions, before Chapter 13 provides an introduction to writing macros, which can be used to automate all or part of an analysis workflow.

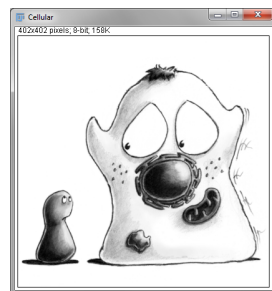
Measurements & regions of interest

Chapter outline

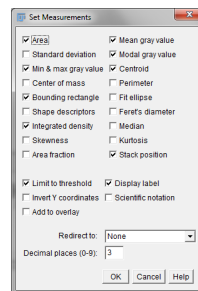
- *Regions of Interest (ROIs) can be manually drawn and measured*
- *The accuracy of measurements will depend upon the Image Properties being correct*
- *Multiple ROIs can be added to the ROI Manager or an overlay*
- *ROIs can be inverted or combined to give more complex shapes*

7.1 Measuring images

The main command for measuring in ImageJ is found under **Analyze** → **Measure** (or just press M), where **Analyze** → **Set Measurements...** determines what measurements are actually made. Possibilities include areas, perimeters, lengths, and minimum, maximum and mean pixel intensities (here referred to as ‘gray values’), as well as further measurements of shapes or intensities (Figure 7.1b).



(a) Image



(b) Set measurements dialog

Label	Area	Mean	Mode	Min	Max	X	Y	BX	BY	Width	Height	IntDen	RawIntDen	Slice	
1	Cellular	161604	227.883	255	1	255	201	201	0	0	402	402	36826846	36826846	1

(c) Results table

Figure 7.1: Measurements made on an image are added to a results table. The choice of measurements to make can be changed using the **Set Measurements...** command.

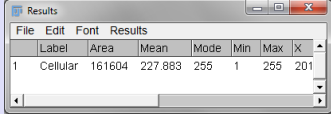
By default, measurements are made over the entire area of the currently-selected 2D image slice, and added to a *results table* (Figure 7.1c).

Important!

Size measurements are automatically scaled according to the pixel sizes in Image → Image Properties..., so these must be correct!

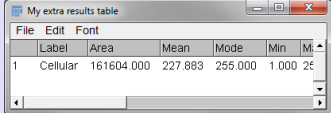
Results tables & identity crises

A small idiosyncrasy to be aware of is that, as far as ImageJ is concerned, there is only ever one ‘official’ results table – the one with the title **Results**. Different, similar-looking tables can be produced (perhaps by duplicating the official table, or internally by some other command), but any new measurements you make with the **Measure** command will *only* be added to the official table. The official table also has an extra **Results** entry in its menu bar.



Label	Area	Mean	Mode	Min	Max	X
1 Cellular	161604	227.883	255	1	255	201

*New measurements **will** be added here*



Label	Area	Mean	Mode	Min	Max
1 Cellular	161604.000	227.883	255.000	1.000	255.000

*New measurements **will not** be added here*

Caution with measurements

Set Measurements... deserves your close attention! Because all new measurements are added to the same results table, when working with multiple images it can be hard to remember which measurement refers to which image. It is therefore a very good idea to choose **Display label** under **Set Measurements...**, to ensure the image title is included in the table. When working with higher dimensions, choosing **Stack Position** lets you know which 2D slice of the entire dataset has been measured.

Also, for now, make sure that **Redirect to** is set to **None**. This is *normally* what you want, to avoid merrily measuring the wrong image by accident (see Chapter 9).

7.2 Regions Of Interest

Usually we want to measure something *within* an image and not the whole thing. Regions Of Interest (ROIs) can be used to define specific parts of an image that should be processed independently or measured, and so only pixels within any ROI we draw will be included in the calculations when we run **Measure**.

ROIs of different types (e.g. rectangles, circles, lines, points, polygons, freehand shapes) can be drawn using the commands in the tool bar (Figure 7.2), and are invariably 2D. Right-clicking the tools often provides access to related tools, while

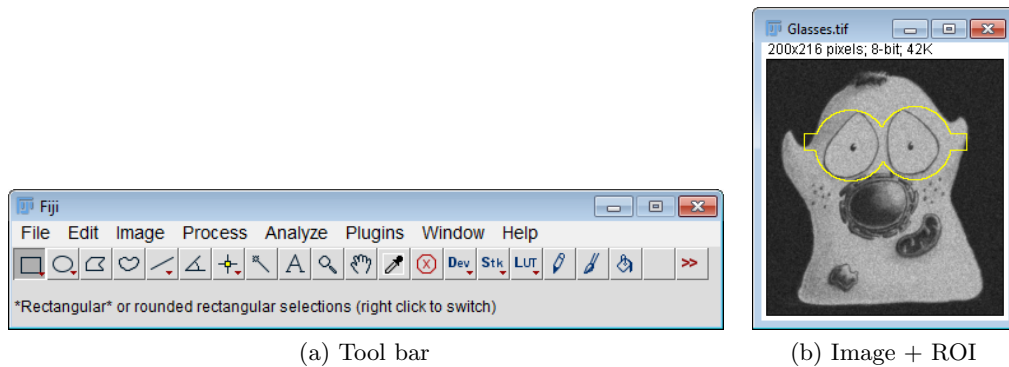


Figure 7.2: ROI drawing tools are found on the left side of the ImageJ tool bar (a). The ROI in (b) was created by drawing one rectangular and two circular ROIs, holding down the **Shift** key between each so that the regions were combined.

double-clicking may give additional options. When drawing a ROI, pressing **Shift** or **Control** before releasing the mouse button adds the ROI being drawn to any existing ROI already present.

Some extra commands to create or adjust ROIs appear under the **Edit** → **Selection** submenu, which we will make more use of later.

Measurement accuracy

Although ImageJ can measure very exactly whatever regions it is told to measure *within an image*, keep in mind that in light microscopy images any size measurements will not exactly correspond to sizes of structures *in real life*. This is especially true at very small scales (hundreds of nanometres or smaller), for blur-related reasons that will be described in Chapter 15.

7.3 Working with multiple regions

Normally, only a single ROI can be ‘active’ (i.e. affecting measurements) at any one time. If you need control over multiple ROIs, there are two places in which you can store them, differing according to purpose:

1. *The ROI Manager*: for most ROIs that you want to be able to edit and use for measurements
2. *The image overlay*: for ROIs that you only want to display

7.3.1 The ROI Manager

The *ROI Manager* provides a convenient way to store multiple ROIs in a list, allowing you to easily access, edit and measure them. The slow way to open it is

to choose **Analyze** → **Tools** → **ROI Manager**... The fast way is just to draw a ROI and press **T**¹. The additional **Measure** command within the manager is then like applying **Analyze** → **Measure** to each ROI in turn. If you happen to want to show all the ROIs simultaneously, you can select the **Show All** option².

Because ROIs in the ROI Manager are represented independently of the image on which they were defined, you can create a ROI on one image, add it to the ROI manager, select a different image and then click on the ROI in the ROI Manager to place it on the second image. Measurements made from the ROI Manager always use the most recently-selected image, so be careful if you have several images open at the same time.

Transferring ROIs

A faster way to transfer ROIs between images without using the ROI Manager is to click on the second image and press **Shift + E** (the shortcut for **Edit** → **Selection** → **Restore Selection**)

Expert ROI manipulation with the ROI Manager

Using the ROI Manager, you can craft your ROIs into more complex shapes, adding or removing other ROIs. First, add the main ROIs you want to work with to the manager. Then select them, and choose from among the options:

- **AND** – create a ROI containing only the regions where the selected ROIs overlap
- **OR** – create a single ROI composed by combining all the selected ROIs
- **XOR** – create a single ROI containing all the selected ROIs, *except* the places where they overlap ('eXclusive OR')

Adjusting overlays

You can 'reactivate' a ROI on an overlay by clicking it with the **Alt** key pressed (provided a suitable ROI tool is selected).

7.3.2 Overlays

Overlays also contain a list of ROIs that are shown simultaneously on the image, but which do *not* affect the **Measure** command. They are therefore suitable for storing annotations. You can think of them as existing on their own separate layer, so that adding and removing the overlay does not mess up the underlying pixel values (Figure 7.3). The relevant commands are found in the **Image** → **Overlay** submenu, where you can get started by drawing a ROI and choosing **Add Selection** (or simply press **B**³). The same submenu also provides commands to transfer ROIs between the overlay and the ROI Manager.

7.3.3 Saving ROIs

Individual ROIs can be saved simply by choosing **File** → **Save As** → **Selection...** The ROI Manager itself has a **Save...** command (under **More**), which will save whichever ROIs are currently selected (or, if none are selected, all of them). Overlays are fixed to specific images and do not have their own special save

¹Easily memorable as 'Take this ROI and add it to the ROI Manager'. Or 'Troy Manager'.

²If you have a stack, you also may need to explore **More** >> **Options...** to define whether all ROIs are shown on all slices, or only on the slices on which they were first created.

³For 'Boverlay'.

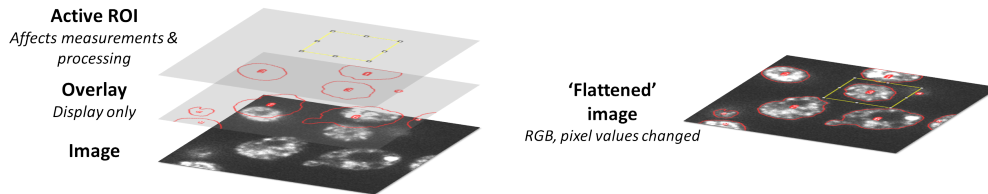


Figure 7.3: ROIs and overlays are displayed on top of images, and so can be removed easily without having any effect upon the pixel values. Flattened images may appear the same on screen, but are invariably RGB (see Chapter 4) and have had their pixel values permanently changed to show any annotations.

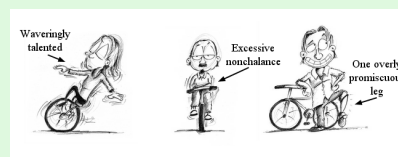
command, but will nonetheless be included if you save the image as a TIFF file (ImageJ's default format). Any currently-active ROI will also be saved in a TIFF.

This is fine if you work only in ImageJ or Fiji, but unfortunately if you try to view your ROIs in other software it is highly unlikely to work properly, since the format is specific to ImageJ. The way around this is to use the **Image** → **Overlay** → **Flatten** command. This creates an *RGB copy of the image in which the pixel values have been changed* so that any ROIs or overlays will appear whenever you open the image elsewhere. Therefore you may well want to use this command when creating figures or presentations, but you do *not* want to subsequently apply your analysis to the image you have flattened – always use the original instead. We will return to this topic in Chapter 4.

Practical 7.1

Open the images `Annotated_Cyclists_1.tif` and `Annotated_Cyclists_2.tif`, which depict the 3 main cyclist characteristics I found most disconcerting as a pedestrian in Heidelberg.

The images should initially look the same, but in one the text is an overlay, while in the other the image has been flattened. Which is which? Try to think of several ways to investigate this.



Solution

Practical 7.2

Using the cyclist image containing the overlay from the previous practical, rearrange the annotations so that they are each positioned next to different cyclists. You could do this by deleting the overlay and starting again, but there are other, faster possibilities (using a technique mentioned before, or the **Image** → **Overlay** → **To ROI Manager** command).

Solution

Solutions

Practical 7.1 `Annotated_Cyclists_1.tif` is the one with the overlay.

Five ways to determine whether an annotation is an overlay or not:

1. Zoom in very closely to the region containing the annotation. If it becomes ‘blocky’, i.e. made up of pixels, it is not an overlay. If it remains smooth, then it is.
2. Move your cursor over the region where the annotation appears, and look at the pixel values. If the values are all the same where the annotation is present, but different elsewhere, then it is unlikely to be an overlay.
3. Using the paintbrush or pencil tool from the toolbar, try putting some other colour where the annotation appears. If the annotation remains visible on top of where you drew, it must be an overlay.
4. Choose `Image` → `Overlay` → `Hide Overlay` and see if the annotation disappears.
5. Choose `Image` → `Overlay` → `To ROI Manager` and see if anything happens at all.

Practical 7.2 *Old solution (when I wrote this question):* Once the ROIs are in the ROI Manager, you can click on each and then move it. However, the original ROI will still stay in the manager – so after moving, you need to add the newly-positioned ROI to the manager, and delete the old one again.

Solution since ImageJ v1.46m: Click the annotation while holding down the `Alt` key, to bring it to life so it can be moved around again. This only works if certain tools are selected, e.g. `Rectangle` or `Text`, because some others have more overriding functions, such as zooming in or scrolling.

Manipulating individual pixels

Chapter outline

- *Point operations are mathematical operations applied to individual pixel values*
- *They can be applied using a single image, an image and a constant, or two images of the same size*
- *Some point operations improve image appearance by changing the relationships between pixel values*

8.1 Introduction

A step used to process an image in some way is called an *operation*, and the simplest examples are *point operations*. These act on individual pixels, changing each in a way that depends upon its own value, but not upon where it is or the values of other pixels. While not immediately very glamorous, point operations often have indispensable roles in more interesting contexts – and so it is essential to know where to find them and how they are used.

8.2 Point operations using a single image

8.2.1 Arithmetic

The **Process** → **Math** submenu is full of useful things to do with pixel values. At the top of the list come the arithmetic operations: **Add...**, **Subtract...**, **Multiply...** and **Divide...**. These might be used to subtract background (extremely important when quantifying intensities; see Chapter 18) or scale the pixels of different images into similar ranges (e.g. if the exposure time for one image was twice that of the other, the pixel values should be divided by two to make them more comparable) – and ought to mostly behave as you expect.

Uses of point operations:
Subtracting a background constant, normalizing to an exposure time, scaling for bit-depth conversion, adjusting contrast...

Question 8.1

Suppose you add a constant to every pixel in the image. Why might subtracting the same constant from the result not give you back the image you started with?

Solution

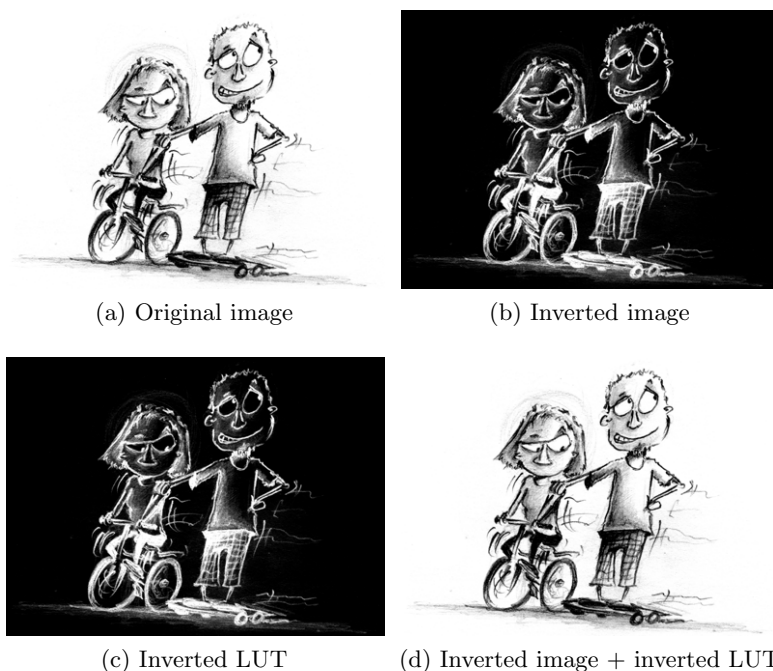


Figure 8.1: The effect of image and LUT inversion on a depiction of two young lovers, spotted on Gaisbergstraße displaying the virtues of invention and tolerance.

8.2.2 Image inversion

Inverting an image (**Edit** → **Invert**) effectively involves ‘flipping’ the intensities: making the higher values lower, and the lower values higher. In the case of 8-bit images, inverted pixel values can be easily computed simply by subtracting the original values from the maximum possible – i.e. from 255. Although this would work in principle for 16-bit images as well, it could have the slightly uncomfortable effect of making an image containing only small values suddenly now only contain huge ones.

Practical 8.1

Edit → **Invert** works differently when applied to different image types. Like in the 8-bit case, pixel values are always subtracted from some ‘maximum’ – your challenge is to work out how this maximum is determined for 16 and 32-bit images in ImageJ.

(Note that the methods used for 16 and 32-bit images are not even the same as one another.)

Solution

Why is inversion useful?

Suppose you have a good strategy designed to detect bright structures, but your images contain dark structures.

Simply invert your images first, then the structures become bright.

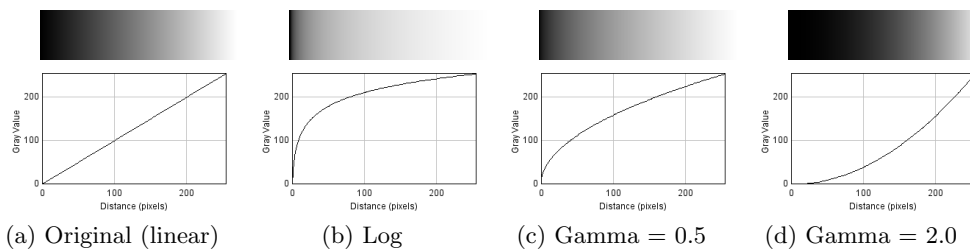


Figure 8.2: Nonlinear transforms applied to a simple ‘ramp’ image, consisting of linearly increasing pixel values. Replacing each pixel with its log or gamma-adjusted value has the effect of compressing either the lower or higher intensities closer together to free up more gray levels for the others.

Inverting LUTs

Beware! You can also invert the colours used for display with `Image` → `Lookup Tables` → `Invert LUT` – which *looks* the same as if the image is inverted, but does not change the pixel values (Figure 8.1)!

Moreover, whether the LUT is inverted is saved inside TIFF files – and so you could potentially open an image and find its LUT was inverted before you even started to do anything, and thereby misjudge whether structures are really brighter or darker than everything else. See `File` → `Open Samples` → `Blobs` for an example of this.

8.2.3 Nonlinear contrast enhancement

With arithmetic operations we change the pixel values, usefully or otherwise, but (assuming we have not fallen into the trap alluded to in a previous question) we have done so in a *linear* way. At most it would take another multiplication and/or addition to get us back to where we were. Because a similar relationship between pixel values exists, we could also adjust the `Brightness/Contrast...` so that it does not *look* like we have done anything at all.

Nonlinear point operations differ in that they affect relative values differently depending upon what they were in the first place (Figure 8.2). This turns out to be very useful for displaying images with *high dynamic ranges* – that is, a big difference between the largest and smallest pixel values (e.g. Figure 8.3). Using the `Brightness/Contrast...` tool (which assigns LUT colours linearly to all the pixel values between the minimum and maximum chosen) it might not be possible to find settings that assign enough different colours to the brightest and darkest regions simultaneously for all the interesting details to be made visible.

The `Gamma...` or `Log...` commands within the `Process` → `Math` submenu offer one type of solution. The former means that every pixel with a value p is replaced by p^γ , where γ is some constant of your choosing. The latter simply

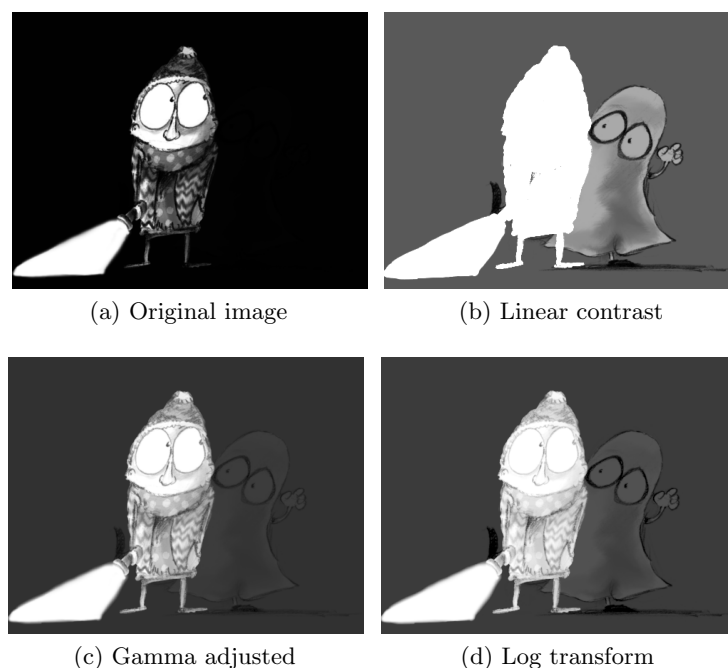


Figure 8.3: The application of nonlinear contrast enhancement to an image with a wide range of values. (*Top row*) In the original image, it is not possible to see details in both the foreground and background simultaneously. (*Bottom row*) Several examples of nonlinear techniques that make details visible throughout the image.

replaces pixel values with their natural logarithm. Examples of these are shown in Figure 8.3. Some extra (linear) rescaling is applied internally by ImageJ when using gamma and log commands, since otherwise the resulting values might fall out of the range supported by the bit-depth.

Practical 8.2

Explore the use of the nonlinear transforms in the ImageJ submenu `Process` → `Math` for enhancing the contrast of any image (possibly `Spooked_16-bit.tif`). In particular, notice how the effects change depending upon whether $gamma < 1$ or not.

Important!

When creating figures for publication, changing the contrast in some linear manner – i.e. just by scaling using the `Brightness/Contrast...` tool – is normally considered fine (assuming that it has not been done mischievously to make some inconvenient, research-undermining details impossible to discern). *But if any nonlinear operations are used, these should always be noted in the figure legend!* This is because although nonlinear operations can be very

helpful when used with care, they can also easily mislead – exaggerating or underplaying differences in brightness.

8.3 Point operations involving multiple images

Instead of applying arithmetic using an image and a constant, we could also use two images of the same size. These can readily be added, subtracted, multiplied or divided by applying the operations to the corresponding pixels.

The command to do this is found under **Process** → **Image Calculator**.... But beware of the bit-depth! If any of the original images are 8 or 16-bit, then the result might require clipping or rounding, in which case selecting the option to create a 32-bit (float) result may be necessary to get the expected output.

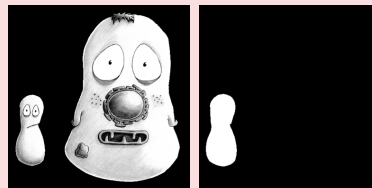
Uses of the image calculator

Subtracting varying backgrounds, comparing images, intensity ratios, masking out regions...

Question 8.2

In the two 32-bit images shown here, white pixels have values of one and black pixels have values of zero (gray pixels have values somewhere in between).

What would be the result of multiplying the images together? And what would be the result of dividing the left image by the right image?



Solution

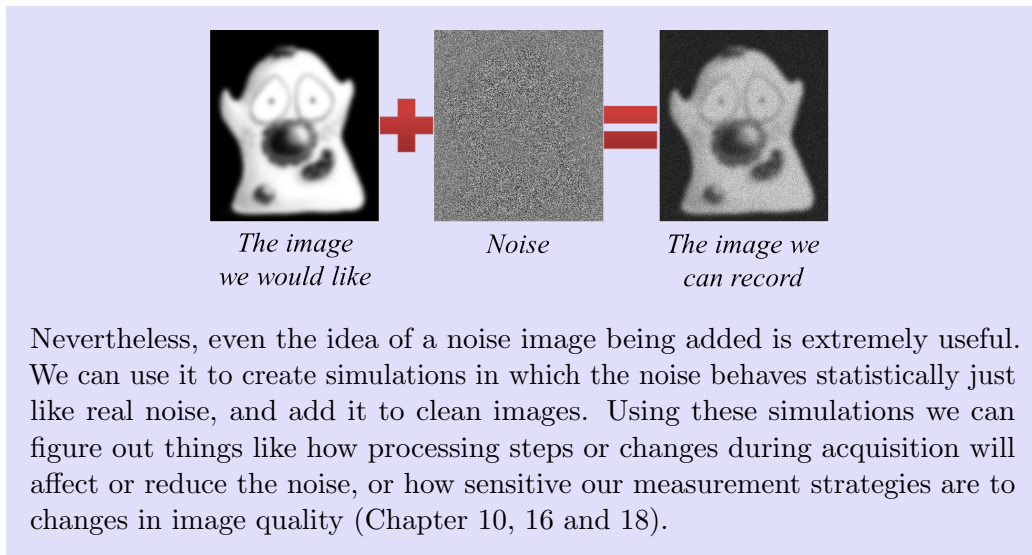
Practical 8.3

With the help of the **Image Calculator**, confirm which two of the images **Same_1.tif**, **Same_2.tif** and **Same_3.tif** are identical in terms of pixel values, and which is different.

Solution

Modelling image formation: Adding noise

Fluorescence images are invariably noisy. The noise appears as a graininess throughout the image, which can be seen as arising from a random noise value (positive or negative) being added to every pixel. This is equivalent to adding a separate ‘noise image’ to the non-existent cleaner image that we would prefer to have recorded. If we knew the pixels in the noise image then we could simply subtract it to get the clean result – but, in practice, their randomness means that we do not.



Solutions

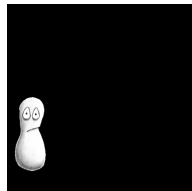
Question 8.1 If you add a constant that pushes pixel values outside the range supported by the bit-depth (e.g. 0–255 for 8-bit), then the result is clipped to the closest possible value. Subtracting the constant again does not restore the original value. Note that this is less likely to occur with a 32-bit image.

Practical 8.1 At the time of writing, to invert a 16-bit image, pixel are subtracted from *the maximum value within the original image*. This is also true for stacks: the maximum value in the entire stack is found.

For 32-bit image inversion, the pixels are subtracted from the *display maximum*, i.e. whatever maximum is set in the **Brightness/Contrast...** dialog box. Consequently, inverting a 32-bit image can give different results each time it is applied if the contrast settings are not kept the same!

One way to improve predictability when inverting a 32-bit image is simply to multiply each pixel by -1 instead of using the **Invert** command – although this would not be a good strategy for 8 or 16-bit images.

Question 8.2 Multiplying the images effectively results in everything outside the white region in the right image being removed from the left image (i.e. set to zero).



Dividing has a similar effect, except that instead of becoming zero the masked-out pixels will take one of three results, depending upon the original pixel's value in the left image:

- if it was *positive*, the result is $+\infty$
- if it was *negative*, the result is $-\infty$
- if it was zero, the result is NaN ('not a number' – indicating 0/0 is undefined)

These are special values that can be contained in floating point images, but not images with integer types.

Practical 8.3 My preferred way to check this is to subtract the images from one another, making sure that the result is 32-bit (in case there are negative values). Doing this should reveal something hidden in the image `Same_2.tif`. Note that the contrast settings differ between `Same_1.tif` and `Same_3.tif`, so they may *look* different.

(Note that the calculator's `Difference` or `Divide` commands could also be used. `XOR` would work as well, but its output is harder to interpret since it involves comparing individual bits used to store each pixel value and gives an output where all matching bits are 0 and all non-matching bits are 1. When converted back into actual decimal values and then to colours for us to look at, this can appear strange. But at least if the resulting image is not completely black then we know that the original input images were not identical.)

Detection by thresholding

Chapter outline

- *The process of detecting interesting objects in an image is called segmentation, and the result is often a binary or labeled image*
- *Global thresholding identifies pixels with values in particular ranges*
- *Thresholds can be calculated from image histograms*
- *Combining thresholding with filtering & image subtraction make it suitable for a wide range of images*
- *Binary images can be used to create ROIs or other object representations*

9.1 Introduction

Chapter 7 described how measurements can be made using manually-drawn ROIs. This may be fine in simple cases where there are not too many things to analyze, but it is preferable to find ways to automate the process of defining regions – not only because this is likely to be faster, but because it should give more reproducible and less biased results.

9.1.1 Objects, segmentation, binary & labeled images

In image processing literature, interesting image structures are frequently called *objects* (or sometimes *connected components*), and the often troublesome process of detecting them is *image segmentation*. Most of the techniques described in the following chapters can be strung together in an effort to segment an image accurately. If successful, the result may be a *binary image*, in which each pixel can only have one of two values to indicate whether it is part of an object or not, or a *labeled image*, in which all pixels that are part of the same object have the same, unique value. It is common to concentrate first on producing a binary image, and then create a labeled image only if necessary by identifying distinct clusters of object pixels and assigning the labels to these.

Binary images in ImageJ

Although only one 1 bit is really needed for each pixel in a binary image, the implementation in ImageJ currently uses 8-bits – and so the actual pixel values allowed are 0 and 255. To complicate matters, ImageJ also permits *either* of these to represent the foreground, with the choice hidden away under `Process → Binary → Options...`, and 0 taken to be ‘black’ and 255 ‘white’. Personally, I prefer for white to represent the foreground (i.e. the interesting things we have detected), and so I will assume that the `Black background` option has been checked.

Nevertheless, you should be aware that this convention is not adopted universally. Furthermore, if you choose `Invert LUT` then the colours are flipped anyway – so yet more confusion arises. Therefore if you find that any processing of binary images gives odd results, be sure to check the binary options and LUT status.

9.2 Global thresholding

The usual way to generate a binary image is by *thresholding*: identifying pixels above or below a particular threshold value. In ImageJ, the `Image → Adjust → Threshold...` command allows you to define both low and high threshold values, so that only pixels falling within a specified range are found. After choosing suitable thresholds, pressing `Apply` produces the binary image¹. Because the same thresholds are applied to every pixel in the entire image, this is an example of *global thresholding* – which is really a kind of point operation, since the output for any pixel depends only on pixel’s original value and nothing else.

9.2.1 Choosing your results with manual thresholds

The puzzle of global thresholding is how to define the thresholds sensibly. If you open `File → Open Samples → HeLa Cells` and split the channels (`Image → Color → Split Channels`), you can use `Threshold...` to interactively try out different possibilities for each channel. You should soon notice the danger in this: the results (especially in the red or green channels) can be very sensitive to the threshold you choose. Low thresholds tend to detect *more* structures, and also to make them *bigger* – until the point at which the structures merge, and then there are *fewer* detections again (Figure 9.1).

In other words, you can sometimes use manual thresholds to get more or less whatever result you want – which could completely alter the interpretation of the data. For the upstanding scientist who finds this situation disconcerting, ImageJ therefore offers a number of automated threshold determination methods

¹Since in ImageJ this replaces the original image, you might want to duplicate it first.

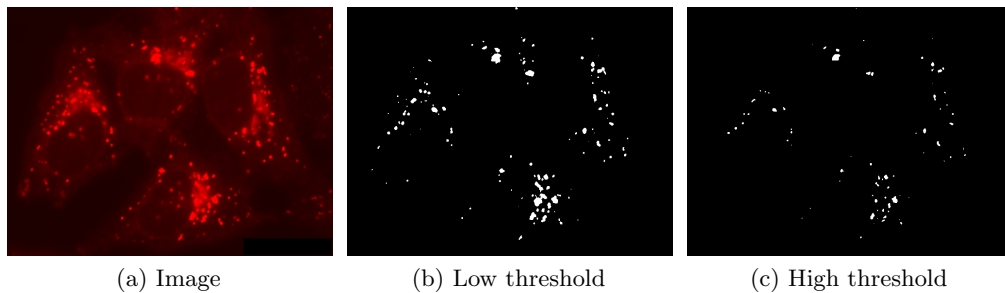


Figure 9.1: Applying manually-chosen thresholds to the red channel of HeLa Cells (a). In (b), a relatively low threshold results in 124 spots being detected with an average area of 32.9 pixels² – but in some places these look like several spots merged. Choosing a higher threshold to avoid merging leads to 74 detections in (c), with an average size of 20.3 pixels².

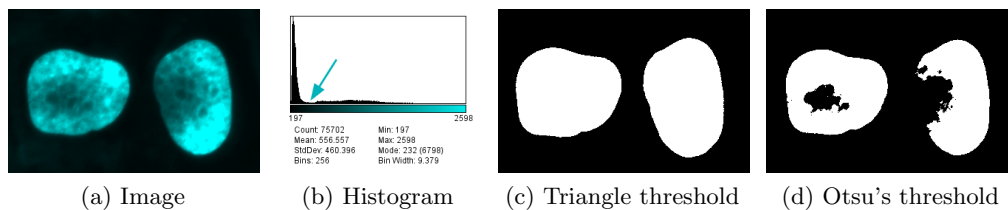


Figure 9.2: Automated thresholding to detect nuclei. (a) Detail from the blue channel of HeLa Cells. (b) In the image histogram you can see one large peak corresponding to the background, and a much longer, shallower peak corresponding to the nuclei. The arrow marks the trough between these two peaks. From inspecting the histogram, one would expect that a threshold anywhere in the range 380-480 could adequately separate the two classes. The triangle method yields a suitable threshold of 395 (c), while Otsu's method gives 762, making it inappropriate for this particular data (d).

in a drop-down list in the `Threshold...` tool. These are described at http://fiji.sc/wiki/index.php/Auto_Threshold, often with references to the original published papers upon which they are based. Fiji's `Image` → `Adjust` → `Auto Threshold` command provides additional options, including the ability to apply all the thresholds and see which one appears to provide the best results.

9.2.2 Determining thresholds from histograms

There is no always-applicable strategy to determine a threshold; images vary too much. However, by its nature, thresholding assumes that there are two classes of pixel in the image – those that belong to interesting objects, and those that do not – and pixels in each class have different intensity values². Whenever values are

²Of course there may be multiple classes for different kinds of objects, and perhaps multiple thresholds would make more sense. However, in such cases it may be possible to apply steps

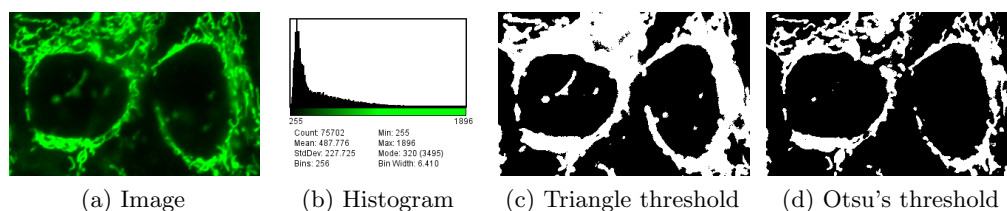


Figure 9.3: Automated thresholding of less distinct classes. (a) Detail from the green channel of HeLa Cells. (b) The background peak is still visible in the histogram, but merges without an obvious trough into the foreground. The triangle method here gives a much lower threshold than Otsu's method (461 rather than 615), although which is preferable may depend on the application (c)-(d).

significant, but their exact location in the image is either unknown or unimportant, this information is neatly summarized within the image's histogram. Therefore ImageJ's methods to find thresholds do not work on the images directly, but rather on their histograms – which is considerably simpler.

A justification for this can be seen in Figure 9.2. Looking at the image, the two nuclei are obvious: they clearly have higher values than the background (a). However, looking at the histogram alone (b) we could already have inferred that there was a class of background pixels (the tall peak on the left) and a class of 'other', clearly distinct pixels (the much shallower peak on the right). By choosing a threshold between these two peaks – somewhere around 400 – the nuclei can be cleanly separated (c). Choosing a threshold much higher or lower than this yields less impressive results (d).

Figure 9.3 gives a more challenging example. In the image itself the structures are not very clearly defined, and in many cases it is not obvious whether we would want to consider any particular pixel as 'bright enough' for detection or not (a). The histogram also depicts this uncertainty; there is a smoother transition between the background peak and the foreground (b). The results of applying two different automated thresholds are shown, (c) and (d). Both are in some sense justifiable, and deciding which is the most appropriate would require a deeper understanding of what the image contains and what is to be analyzed.

Automated thresholding and data clipping

If the data is clipped (Section 3.3), then the statistics calculated from the histogram are affected – and the theory underlying why an automated threshold should work might no longer apply. *This is another reason why clipping should always be avoided!*

such as filtering to remove some of the more confusing information, and reduce the detection problem to that of separating only two classes. Therefore although thresholding is not always appropriate and different methods of detection can be needed for complex problems, it is still useful a lot of the time.

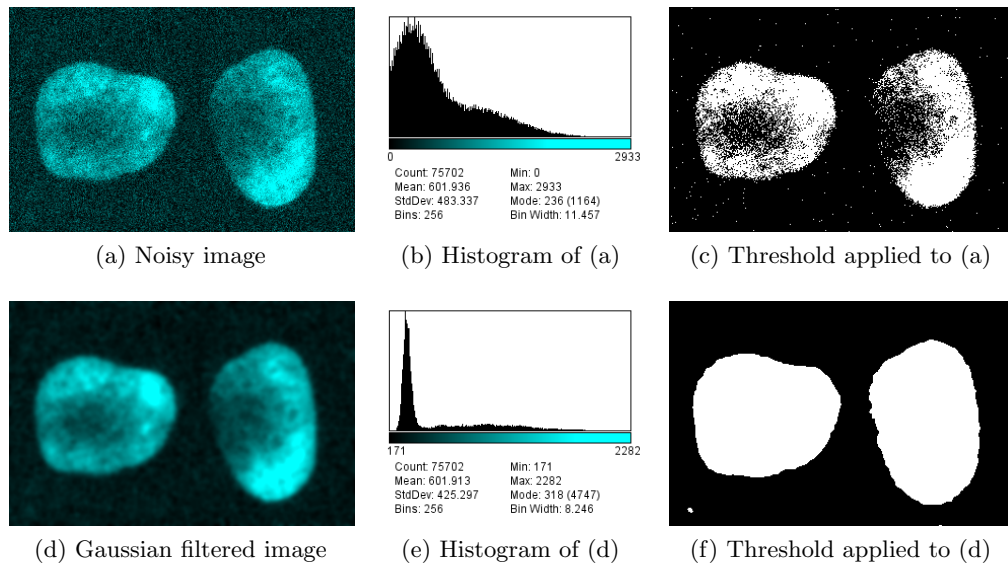


Figure 9.4: Noise can affect thresholding. After the addition of simulated noise to the image in Figure 9.2a, the distinction between nuclei and non-nuclei pixels is much harder to identify in the histogram (b). Any threshold would result in a large number of incorrectly-identified pixels. However, applying a Gaussian filter (here, $\sigma = 2$) to reduce noise can dramatically improve the situation (e). Thresholds in (c) and (f) were computed using the triangle method.

9.3 Thresholding difficult data

Applying global thresholds is all well and good in easy images for which a threshold clearly exists, but in practice things are rarely so straightforward – and often no threshold, manual or automatic, produces useable results. This section anticipates the next chapter on filters by showing that, with some extra processing, thresholding can be redeemed even if it initially seems to perform badly.

9.3.1 Thresholding noisy data

Noise is one problem that affects thresholds, especially in live cell imaging. The top half of Figure 9.4 reproduces the nuclei from Figure 9.2, but with extra noise added to simulate less than ideal imaging conditions. Although the nuclei are still clearly visible in the image (a), the two classes of pixel previously easy to separate in the histogram have now merged together (b). The triangle threshold method, which had performed well before, now gives less attractive results (c), because the noise has caused the ranges of background and nuclei pixels to overlap. However, applying a Gaussian filter smooths the image, thereby reducing much of the random noise (Chapter 10), which results in a histogram dramatically more similar to that in the original, (almost) noise-free image, and the threshold is

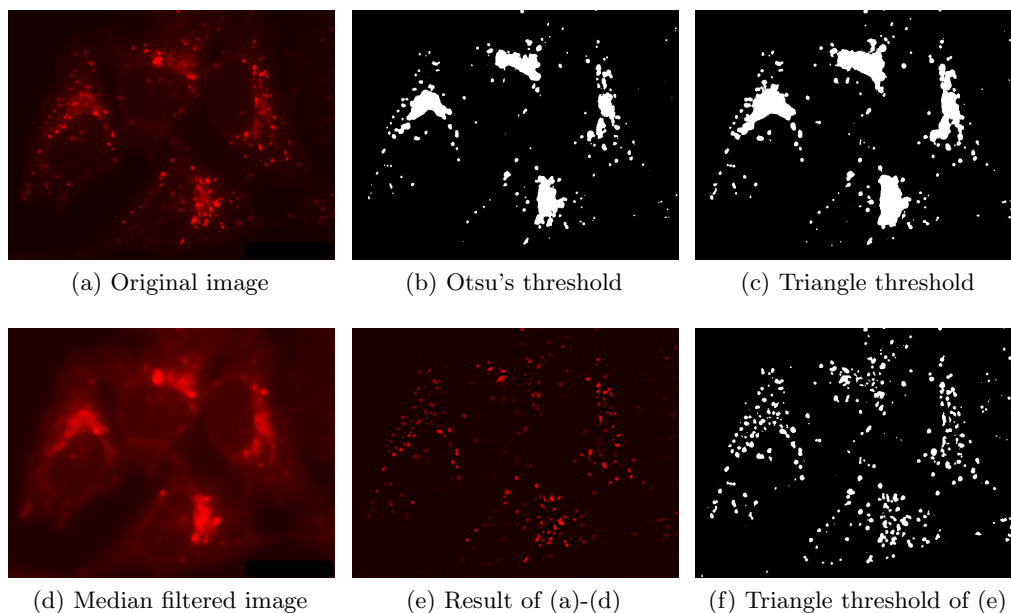


Figure 9.5: Thresholding to detect structures appearing on a varying background. No global threshold may be sufficiently selective (top row). However if a ‘background image’ can be created, e.g. by median filtering, and then subtracted, a single threshold can give better results (bottom row). This is equivalent to applying a varying threshold to the original image.

again quite successful (f).

9.3.2 Local thresholding

Another common problem is that structures that should be detected appear on top of a background that itself varies in brightness. For example, in the red channel of HeLa cells there is no single global threshold capable of identifying and separating all the ‘spot-like’ structures; any choice will miss many of the spots because a threshold high enough to avoid the background will also be too high to catch all the spots occurring in the darker regions (Figure 9.5a–c).

In such cases it would be better if we could define different thresholds for different parts of the image: a *local threshold*. A few methods to do this are implemented in Fiji’s Image → Adjust → Auto Local Threshold, and described at http://fiji.sc/wiki/index.php/Auto_Local_Threshold. However, if these are insufficient it is easy to implement our own local thresholding and get more control over the result if we think of the problem from a slightly different angle. Suppose we had a second image that contained values equal to the thresholds we want to apply, and which could be different for every pixel. If we simply *subtract* this second image from the first, we can then apply a global threshold to detect what we want.

The difficult part is creating the second image, but again filters come in useful. One option is a *median filter* (Section 10.3.1), which effectively moves through every pixel in the image, ranks the nearby pixels in order of value, and chooses the middle one – thereby removing anything much brighter or much darker than its surroundings (d). Subtracting the median-filtered image from the original gives a result to which a global threshold can be usefully applied (e).

Alternative background subtraction

ImageJ already has a **Process** → **Subtract Background...** command that does something similar to the above, but in which the background is determined using the ‘rolling ball algorithm’. This command is described in more detail upon pressing its **Help** button, and it supports previewing the background so that you can check it is doing something appropriate.

Practical 9.1

Explore several automated methods of thresholding the different channels of **File** → **Open Samples** → **HeLa Cells**, using **Subtract Background...** if necessary.

9.4 Practicalities: bit-depths & types

9.4.1 Using NaNs

Although not obviously integral to the idea of thresholding, bit-depths and image types are relevant in two main ways. The first appears when you click **Apply** in the **Threshold...** dialog box for a 32-bit image. This presents the option **Set Background Pixels to NaN**, which instead of making a binary image would give an image in which the foreground pixels retain their original values, while background pixels are *Not A Number*. This is a special value that can only be stored in floating point images, which ImageJ ignores when making measurements later. It is therefore used to mask out regions.

Question 9.1

Through experiment or guesswork, what do you suppose happens to NaNs with a 32-bit image is converted to 8-bit or 16-bit? *Solution*

Practical 9.2

Create an image including NaN pixels, then measure some ROIs drawn on it. Are area measurements affected by whether NaNs are present or not? *Solution*

9.4.2 Histogram binning

The second way in which bit-depths and types matter is that histograms of images > 8-bit involve *binning* the data. For example, with a 32-bit image it would probably not make sense to create a histogram that has separate counts for all possible pixel values: in addition to counts for pixels with exact values 1 and 2, we would have thousands of counts for pixels with fractions in between (and most of these counts would be 0). Instead, the histogram is generated by dividing the total data range (maximum – minimum pixel values) into 256 separate *bins* with equal widths, and counting how many pixels have values falling into the range of each bin. It is therefore like a subtle conversion to 8-bit precision for the threshold calculation, but without actually changing the original data. The same type of conversion is used for 16-bit images – *unless* you use Fiji's `Image → Adjust → Auto Threshold` command, which uses a full 16-bit histogram with 65536 bins.

Although binning effects can often be ignored, if the total range of pixel values in an image is very large then it is worth keeping in mind.

Practical 9.3

What are the implications of using a 256-bin histogram for thresholding a 32-bit image? In particular, how might any outlier pixels affect the accuracy with which you can define a threshold – automatically or manually?

To explore this, you can use the extreme example of `cell_outlier.tif` along with the `Threshold...` command. `Analyze → Histogram` lets you investigate the image histogram with different numbers of bins – but any changes you make here will not be reflected in the histogram actually used for thresholding.


How could you (manually) reduce the impact of any problems you find?

Solution

9.5 Measuring objects in binary images

9.5.1 Generating & measuring ROIs

Once we have a binary image, the next step is to identify objects within it and measure them. In 2D, there are several options:

- Click on an object with the Wand tool  to create measurable ROI from it
- `Edit → Selection → Create Selection` makes a single ROI containing all the foreground pixels. Disconnected regions can be separated by adding the ROI to the ROI Manager and choosing `More >> Split`.
- `Analyze → Analyze Particles...` detects and measures all the foreground regions as individual objects.

Creating ROIs

The Wand tool, `Create Selection & Analyze`

`Particles...` can also be used when a threshold is being previewed on an image, but it has not yet been converted to binary.

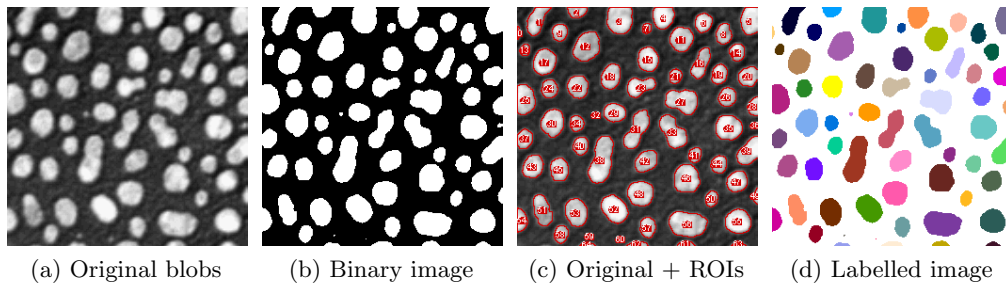


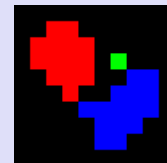
Figure 9.6: Examples of a grayscale (`Blobs.gif`), binary and labelled image. In (c), ROIs have been generated from (b) and superimposed on top of (a). In (d), each label has been assigned a unique colour for display.

`Analyze Particles...` is the most automated and versatile option, making it possible to ignore regions that are particularly small or large, straight or round (using a `Circularity` metric). It can also output summary results and add ROIs for each region to the ROI Manager. With the `Show: Count Masks` option, it will generate a labeled image, in which each pixel has a unique integer value indicating the number of the object it is part of – or zero if it is in the background. With a suitably colourful LUT, this can create a helpful and cheerful display (Figure 9.6d).

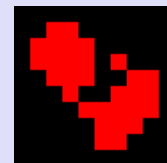
Connectivity

Identifying multiple objects in a binary image involves separating distinct groups of pixels that are considered ‘connected’ to one another, and then creating a ROI or label for each group. Connectivity in this sense can be defined in different ways. For example, if two pixels have the same value and are immediately beside one another (above, below, to the left or right, or diagonally adjacent) then they are said to be *8-connected*, because there are 8 different neighbouring locations involved. Pixels are *4-connected* if they are horizontally or vertically adjacent, but *not* only diagonally.

The choice of connectivity can make a big difference in the number and sizes of objects found, as the example on the right shows (distinct objects are shown in different colours). An option to specify the connectivity used by the `Wand` tool can be found by double-clicking its button.



4-connected



8-connected

Question 9.2

What do you suppose *6-connectivity* and *26-connectivity* refer to? *Solution*

Practical 9.4

Work out what kind of connectivity is used by the `Analyze Particles...` command.

Solution

9.5.2 Redirecting measurements

Although binary images can show the shapes of things to be measured, pixel intensity measurements made on a binary image are not very helpful. You could use the above techniques to make ROIs from binary images, then apply those to the original image to get meaningful measurements. However, it is possible to avoid this extra step by changing the `Redirect to:` option under `Set Measurements...`. This allows you to measure ROIs or run `Analyze Particles...` with one image selected and used to define the regions, while redirecting your measurements to be made on a completely different image of your choice.

If you use this, just be sure to reset the `Redirect to:` option when you are done, to avoid accidentally measuring the wrong image for so long as it is open.

Solutions

Question 9.1 Since NaN is not an integer, it cannot be stored in an 8-bit or 16-bit unsigned integer image. Instead, all NaNs simply become zero.

Question 9.2 They are. If you measure the area of an image containing NaNs, the result is less than if you measure the area of the same image converted to 8-bit – since only the non-NaN parts are included. If you measure a region containing NaNs only, the area is 0.

Practical 9.3 First, a positive implication of using a 256-bit histogram for thresholding is that it can be fast: more bins add to the computations involved. Also, creating too many bins has the result of making most of them zero – potentially causing some automated threshold-determination algorithms to fail.

A negative implication is that using 256 bins means that only 256 different thresholds are possible: that is, if your image range is 0–25500, then the thresholds you could get are 0, 100, 200, . . . 25500. If the optimal threshold is really 150, this will not be found. But usually if your range of pixel values is this large, you do not need a very fine-grained threshold for acceptable results anyway.

This changes if you have outliers. A single extreme pixel – as occurs when a pixel in a CCD camera is somehow ‘broken’ – can cause most other pixels in the image to be squeezed into only a few bins. Then the histogram resolution might really be too small for reasonable thresholding.

Two possible ways to overcome this are:

1. Apply a provisional threshold to detect the outliers only, switch the **Dark background** option if necessary, and use the trick of making background values NaN in thresholded 32-bit images. This eliminates the outlier so that it cannot influence the results. Recomputing the threshold will simply ignore it.
2. Convert the image to 8-bit manually yourself. This allows you to effectively choose the range of the histogram bins (using **Brightness/Contrast...**; see Section 3.4) Since the threshold is made using 256 bins, you are not really losing any information that was not going to be lost anyway.

Question 9.2 6-connectivity is similar to 4-connectivity, but in 3D. If all 3D diagonals are considered, we end up with each pixel having 26 neighbours.

Question 9.4 At the time of writing, **Analyze Particles...** uses 8-connectivity.

Filters

Chapter outline

- *Filters can be used to reduce noise and/or enhance features, making detection & measurement much easier*
- *Linear filters replace each pixel by a weighted sum of surrounding pixels*
- *Nonlinear filters replace each pixel with the result of some other computation using surrounding pixels*
- *Gaussian filters have various advantages that make them a good choice for many applications with fluorescence images*

10.1 Introduction

Filters are phenomenally useful. Almost all interesting image analysis involves filtering of some sort at some stage. In fact, the analysis of a difficult image sometimes becomes trivial once a suitable filter has been applied to it. It is therefore no surprise that much of the image processing literature is devoted to the topic of designing and testing filters.

The basic idea of filtering here is that each pixel in an image is assigned a new value depending upon the values of other pixels within some defined region (the pixel's *neighbourhood*). Different filters work by applying different calculations to the neighbourhood to get their output. Although the plethora of available filters can be intimidating at first, knowing only a few of the most useful is already a huge advantage.

This chapter begins by introducing several extremely common *linear* and *nonlinear* filters for image processing. It ends by considering in detail some techniques based on one particularly important linear filter.

10.2 Linear filtering

Linear filters replace each pixel with a *linear combination* ('sum of products') of other pixels. Therefore the only mathematical background they require is the ability to add and multiply.

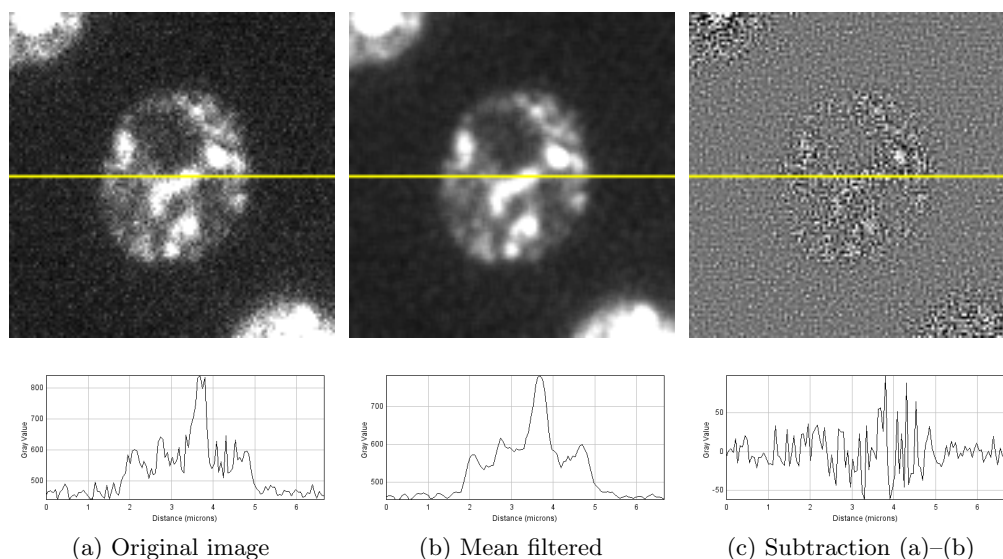


Figure 10.1: Filters can be used to reduce noise. (a) A spinning disc confocal image of a yeast cell. (b) Applying a small mean filter makes the image smoother, as is particularly evident in the fluorescence plot made through the image centre. (c) Computing the difference between images shows what the filter removed, which was mostly random noise.

10.2.1 Mean filters

To begin, consider the somewhat noisy image of a yeast cell in Figure 10.1a. The noise can be seen in the random jumps in the fluorescence intensity profile shown. One way to improve this is to take each pixel, and simply replace its value with the mean (average) of itself and the 8 pixels immediately beside it (including diagonals). This ‘averages out’ much of this noisy variation, giving a result that is considerably smoother (b). Subtracting the smoothed image from the original shows that what has been removed consists of small positive and negative values, mostly (but not entirely) lacking in interesting structure (c).

This smoothing is what a 3×3 *mean filter*¹ does. Each new pixel now depends upon the average of the values in a 3×3 pixel region: the noise is reduced, at a cost of only a little spatial information. The easiest way to apply this in ImageJ is through the `Process` → `Smooth` command². But this simple filter could be easily modified in at least two ways:

1. Its size could be increased. For example, instead of using just the pixels immediately adjacent to the one we are interested in, a 5×5 mean filter replaces each pixel by the average of a square containing 25 pixels, still centred on the main pixel of interest.

¹Also called an *arithmetic mean*, *averaging* or *box-car filter*.

²Note that the shortcut is `Shift + S` – a fact I rediscover regularly when intending to save my images. Be careful!

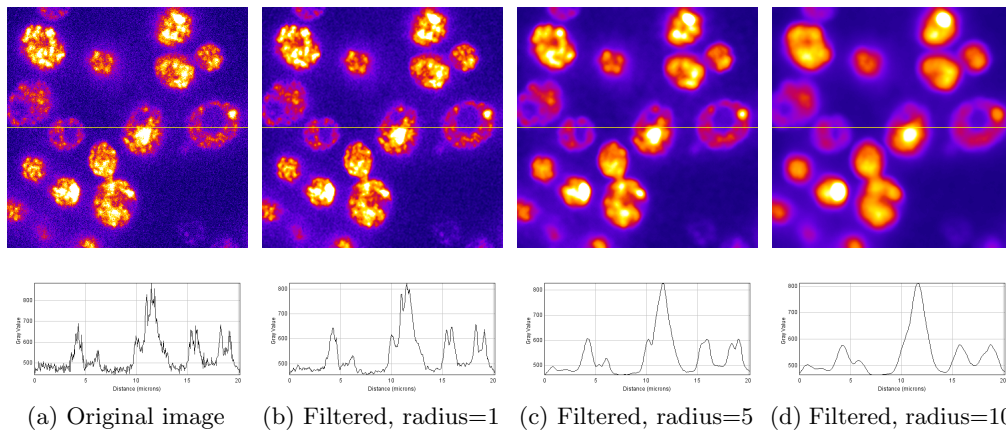


Figure 10.2: Smoothing an image using mean filters with different radii.

2. The average of the pixels in some other shape of region could be computed, not just an $n \times n$ square.

Process \rightarrow **Filters** \rightarrow **Mean...** is ImageJ's general command for mean filtering. It uses approximately circular neighbourhoods, and the neighbourhood size is adjusted by choosing a **Radius** value. The **Show Circular Masks** command displays the neighbourhoods used for different values of **Radius**. If you happen to choose **Radius** = 1, you get a 3×3 filter – and the same results as using **Smooth**.

Figure 10.2 shows that as the radius increases, the image becomes increasingly smooth – losing detail along with noise. This causes the result to look blurry. If noise reduction is the primary goal, it is therefore best to avoid unnecessary blurring by using the smallest filter that gives acceptable results. More details on *why* mean filters reduce noise, and by how much, will be given in Chapter 16.

Question 10.1

Setting **Radius** = 6 gives a circular filter that replaces each pixel with the mean of 121 pixels. Using a square 11×11 filter would also replace each pixel with the mean of 121 pixels. Can you think of any advantages in using the circular filter rather than the square filter?

Solution

10.2.2 General linear filters

There are various ways to compute a mean of N different pixels. One is to add up all the values, then divide the result by N . Another is to multiply each value by $1/N$, then add up the results. The second approach has the advantage that it is easy to modify to achieve a different outcome by changing the weights used to scale each pixel depending upon where it is. This is how a *linear filter* works in general, and mean filters are simply one specific example.

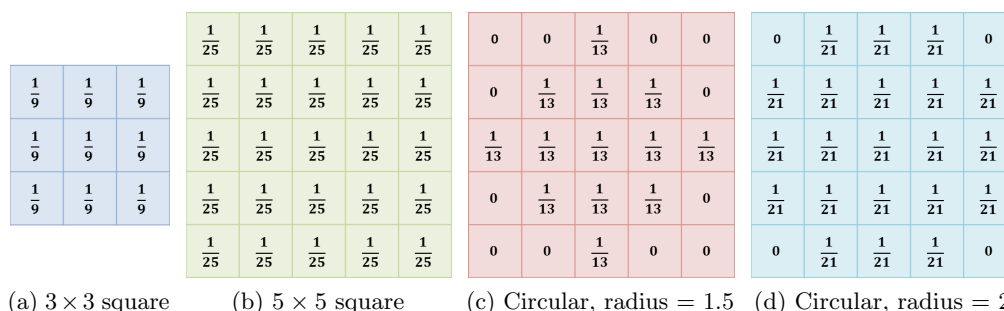


Figure 10.3: The kernels used with several mean filters. Note that (c) and (d) are the ‘circular’ filters used by ImageJ’s `Mean...` command for different radii.

A linear filter is defined by a *filter kernel* (or *filter mask*). This resembles another (usually small and rectangular) image in which each pixel is known as a *filter coefficient* and these correspond to the weights used for scaling. In the case of a mean filter, the coefficients are all the same (or zero, if the pixel is not part of the neighbourhood), as shown in Figure 10.3. But different kernels can give radically different results, and be designed to have very specific properties.

An algorithm to apply the filtering is shown in Figure 10.4.

Question 10.2

When filtering, the output for each pixel is usually put into a new image – so that the original image is unchanged (although ImageJ might switch the new image in place of the old as soon as it is finished, so that it *looks* like the image was changed).

Is the creation of a new image really necessary for the algorithm in Figure 10.4 to work, or does it just prevent the old image being lost – allowing you to retrieve it by pressing `Undo`?

Solution

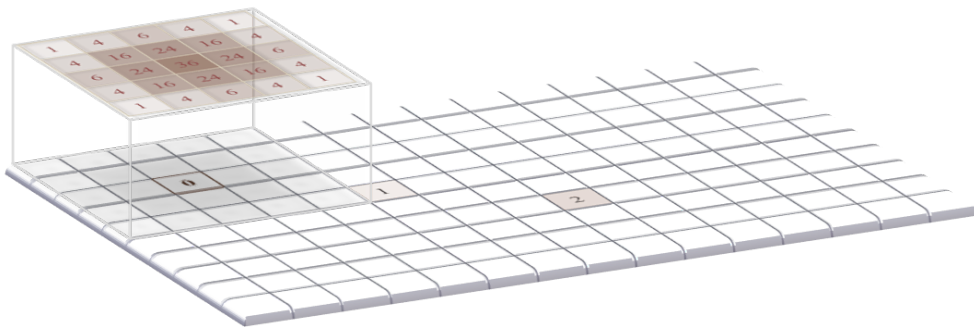
10.2.3 Defining your own filters

The application of such filtering is often referred to as *convolution*, and if you like you can go wild inventing your own filters using the `Process` → `Filters` → `Convolve...` command. This allows you to choose which specific coefficients the filter should have, arranged in rows and columns. If you choose the `Normalize Kernel` option then the coefficients are scaled so that they add to 1 (if possible), by dividing by the sum of all the coefficients.

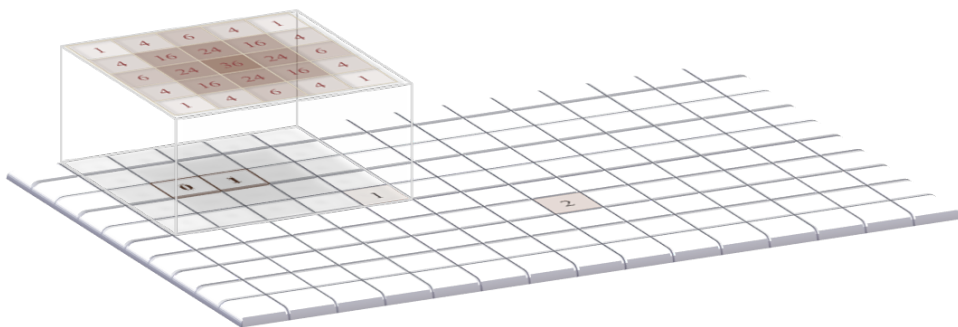
Question 10.3

When defining an $n \times n$ filter kernel with `Convolve...`, ImageJ insists that n is an odd number. Why?

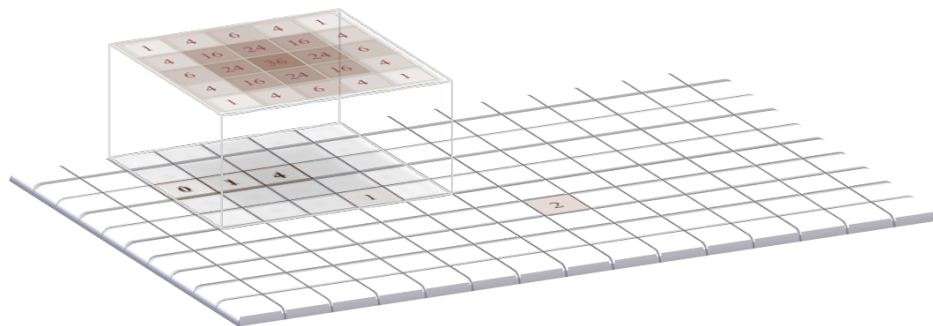
Solution



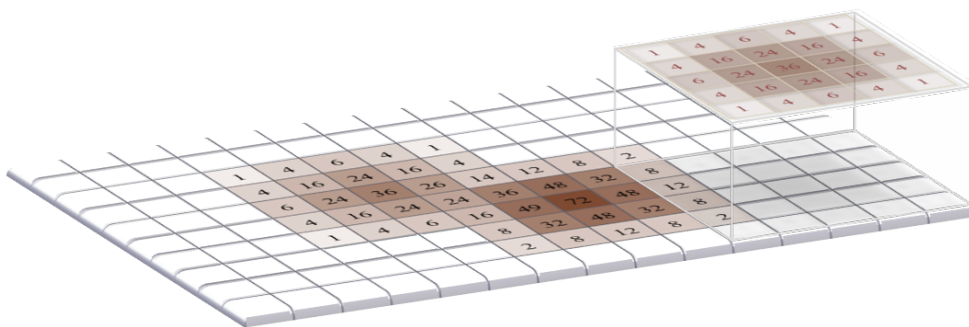
(a) The filter is positioned over the top corner of the image. The products of the filter coefficients and the corresponding image pixel values are added together, and the result inserted in a new output image (although here the output is displayed in the original image to save space).



(b) The filter is shifted to the next pixel in line, and the process repeated.



(c) The filtering continues into the third pixel.



(d) The filtering operation is applied to all pixels in the image to produce the final output.

Figure 10.4: Applying a linear filter to an image containing two non-zero pixels using the sum-of-products algorithm. The result is an image that looks like it contains two (scaled) versions of the filter itself, which in this case overlap with one another.

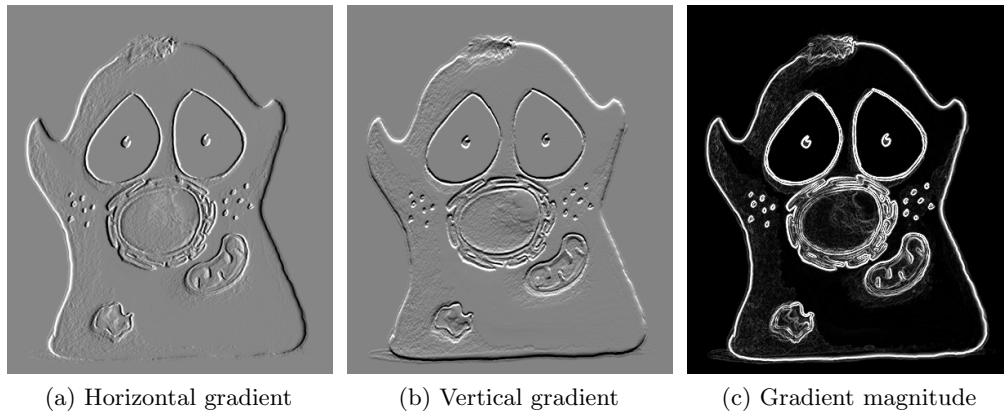


Figure 10.5: Using gradient filters and the gradient magnitude for edge enhancement.

Question 10.4

Predict what happens when you convolve an image using a filter that consists of a single coefficient with a value -1 in the following cases:

1. `Normalize Kernel` is checked
2. You have a 32-bit image (`Normalize Kernel` is unchecked)
3. You have an 8-bit image (`Normalize Kernel` is unchecked)

Solution

10.2.4 Gradient filters

Often, we want to detect structures in images that are distinguishable from the background because of their edges. So if we could detect the edges we would be making good progress. Because an edge is usually characterized by a relatively sharp transition in pixel values – i.e. by a steep increase or decrease in the profile across the image – *gradient filters* can be used to help.

A very simple gradient filter has the coefficients $-1, 0, 1$. Applied to an image, this replaces every pixel with the difference between the pixel to the right and the pixel to the left. The output is positive whenever the fluorescence is increasing horizontally, negative when the fluorescence is decreasing, and zero if the fluorescence is constant – *no matter what the original constant value was*, so that flat areas are zero in the gradient image irrespective of their original brightness. We can also rotate the filter by 90° and get a vertical gradient image (Figure 10.5).

Having two gradient images with positive and negative values can be somewhat hard to work with. If we square all the pixels in each, the values become positive.

Finding edges

For more sophisticated edge detection, see `FeatureJ Edges & FeatureJ Laplacian`, or search for *Canny-Deriche filtering*.

Then we can add both the horizontal and vertical images together to combine their information. If we compute the square root of the result, we get what is known as the *gradient magnitude*³, which has high values around edges, and low values everywhere else. This is (almost) what is done by the command **Process** → **Find Edges**.

Practical 10.1

Try calculating the gradient magnitude using **Duplicate...**, **Convolve...**, **Image Calculator...** and several commands in the **Process** → **Math** → submenu. If you need a sample image, you can use **File** → **Open Samples** → **Blobs (25K)**. (*Be sure to pay attention to the bit-depth!*) *Solution*

Question 10.5

Suppose the mean pixel value of an image is 100. What will the mean value be after applying a horizontal gradient filter? *Solution*

Practical 10.2

There is a LUT called **edges** in ImageJ. Applied to **File** → **Open Samples** → **Blobs (25K)**, it does a rather good job of highlighting edges – without actually changing the pixels at all. How does it work? *Solution*

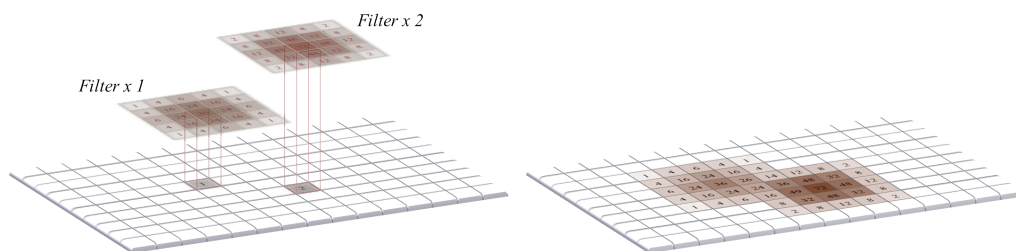
10.2.5 Convolution & correlation

Although linear filtering and convolution are terms that are often used synonymously, the former is a quite general term while the latter can be used in a somewhat more restricted sense. Specifically, for convolution the filter should be rotated by 180° before applying the algorithm of Figure 10.4. If the algorithm is applied without the rotation, the result is really a *correlation*. However, this distinction is not always kept in practice; convolution is the more common term, and often used in image processing literature whenever no rotation is applied. Fortunately, much of the time we use symmetric filters, in which case it makes absolutely no difference which method is used. But for gradient filters, for example, it is good to be aware that the sign of the output (i.e. positive or negative) would be affected.

Why rotate a filter for convolution?

It may not be entirely clear why rotating a filter for convolution would be worthwhile. One partial explanation is that if you convolve a filter with an image containing only a single non-zero pixel that has a value of one, the result

³The equation then looks like Pythagoras' theorem: $G_{mag} = \sqrt{G_x^2 + G_y^2}$



(a) A copy of the filter is centred on every non-zero pixel in the image, and its coefficients are multiplied by the value of that pixel. (b) The coefficients of the scaled filters are assigned to the pixels of the output image, and overlapping values added together.

Figure 10.6: An alternative view of convolution as the summation of many scaled filters. Here, only two pixels in the original image have non-zero values so only two copies of the filter are needed, but often all pixels are non-zero – resulting in the addition of as many scaled filters as there are pixels. The final image computed this way is the same as that obtained by the method in Figure 10.4 – assuming either symmetrical filters, or that one of them has been rotated.

is an exact replica of the filter. But if you correlate the filter with the same image, the result is a rotated version of the filter. This can be inferred from Figures 10.4a and 10.4b: you can see that when the bottom right value of the filter overlaps with the first non-zero pixel, it results in the filter coefficient’s value being inserted in the top left of the image. Thus the application of the algorithm in Figure 10.4 inherently involves a rotation, and by rotating the filter first this is simply ‘corrected’.

This leads to an equivalent way to think of convolution: each pixel value in an image scales the filter, and then these scaled filters replace the original pixels in the image – with overlapping values added together (Figure 10.6). This idea reappears in Chapter 15, because convolution happens to also describe the blur inherent in light microscopy.

Question 10.6

Does ImageJ’s `Convolve...` command really implement convolution – or is it actually correlation?

Solution

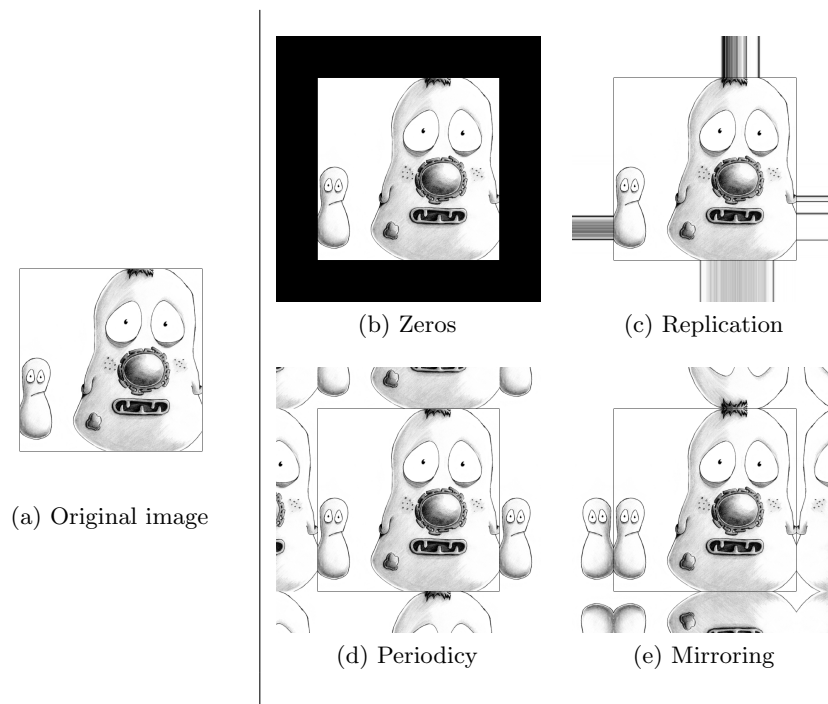
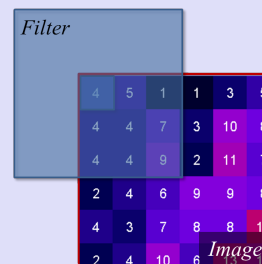


Figure 10.7: Methods for determining suitable values for pixels beyond image boundaries when filtering.

Filtering at image boundaries

If a filter consists of more than one coefficient, the neighbourhood will extend beyond the image boundaries when filtering some pixels nearby. These boundary pixels could simply be ignored and left with their original values, but for large neighbourhoods this would result in much of the image being unfiltered. Alternative options include treating every pixel beyond the boundary as zero, replicating the closest valid pixel, treating the image as if it is part of a periodic tiling, or mirroring the internal values (Figure 10.7).



Practical 10.3

Using any image, work out which of the methods for dealing with boundaries shown in Figure 10.7 is used by ImageJ's `Convolve...` command. *Solution*

Figure 10.8: Results of different 3×3 rank filters when processing a single neighbourhood in an image. The output of a 3×3 mean filter in this case would also be 15.

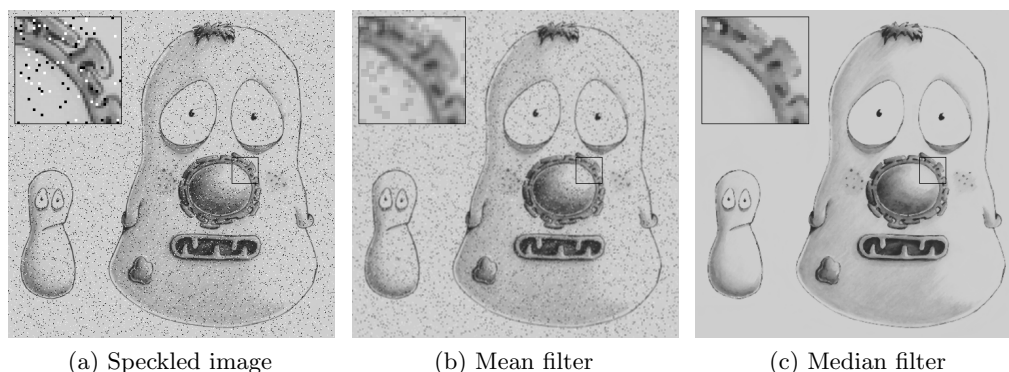
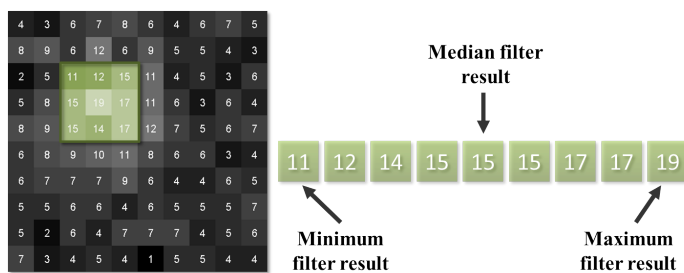


Figure 10.9: Applying mean and median filters (radius = 1 pixel) to an image containing isolated extreme values (known as *salt and pepper noise*). A mean filter reduces the intensity of the extreme values but spreads out their influence, while a small median filter is capable of removing them completely with a minimal effect upon the rest of the image.

10.3 Nonlinear filters

Linear filters involve taking neighbourhoods of pixels, scaling them by specified constants, and adding the results to get new pixel values. Nonlinear filters also make use of neighbourhoods of pixels, but with different calculations to obtain the output. Here we will consider one especially important family of nonlinear filters.

10.3.1 Rank filters

Rank filters effectively sort the values of all the neighbouring pixels in ascending order, and then choose the output based upon this ordered list. The most common example is the *median filter*, in which the pixel value at the centre of the list is used for the filtered output. The result is often similar to that of a mean filter, but has the major advantage of removing extreme isolated values completely, *without allowing them to have an impact upon surrounding pixels*. This is in contrast to a mean filter, which cannot ignore extreme pixels but rather will smooth them out into occupying larger regions (Figure 10.9). However, a disadvantage of a median filter is that it can seem to introduce patterns or textures that were not present in the original image, at least whenever the size of the filter increases (see

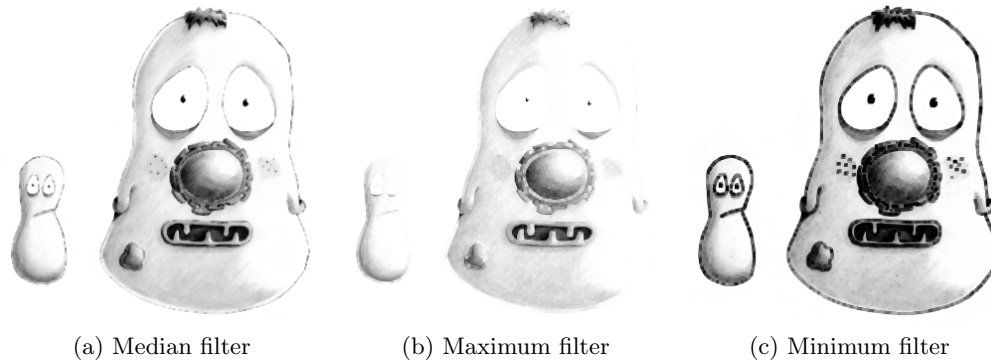


Figure 10.10: The result of applying three rank filters (radius = 1 pixel) to the noise-free image in Figure 10.13a.

Figure 10.13d). Another disadvantage is that large median filters tend to be slow.

Other rank filters include the *minimum* and *maximum* filters, which replace each pixel value with the minimum or maximum value in the surrounding neighbourhood respectively (Figure 10.10). They will become more important in Chapter 11.

Question 10.7

What would happen if you subtract a minimum filtered image (e.g. Figure 10.10c) from a maximum filtered image (Figure 10.10b)? *Solution*

Removing outliers

Figure 10.9 shows that median filtering is much better than mean filtering for removing outliers. We might encounter this if something in the microscope is not quite functioning as expected or if dark noise is a problem, but otherwise we expect the noise in fluorescence microscopy images to produce few really extreme values (see Chapter 16).

Nevertheless, `Process` → `Noise` → `Remove Outliers...` provides an alternative if isolated bright values are present. This is a nonlinear filter that inserts median values *only whenever a pixel is found that is further away from the local median than some adjustable threshold*. It is therefore like a more selective median filter that will only modify the image at pixels where it is considered really necessary. The main difficulty is then choosing a sensible threshold.

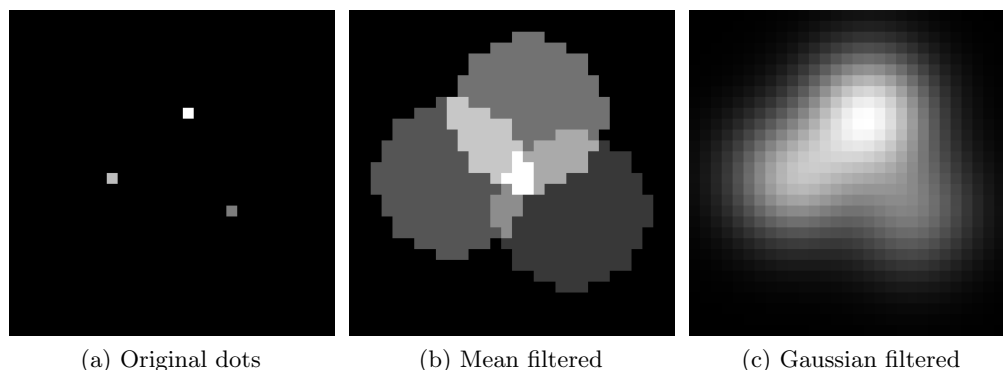
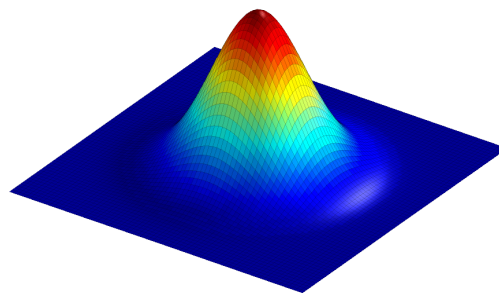


Figure 10.11: Comparing a mean and Gaussian filter. The mean filter can introduce patterns and maxima where previously there were none. For example, the brightest region in (b) is one such maximum – *but the values of all pixels in the same region in (a) were zero!* By contrast, the Gaussian filter produced a smoother, more visually pleasing result, less prone to this effect (c).

Figure 10.12: Surface plot of a 2D Gaussian function, calculated using the equation

$$g(x, y) = Ae^{-\left(\frac{x^2+y^2}{2\sigma^2}\right)}$$

The scaling factor A is used to make the entire volume under the surface equal to 1, which in terms of filtering means that the coefficients add to 1 and the image will not be unexpectedly scaled. The size of the function is controlled by σ .



10.4 Gaussian filters

10.4.1 Gaussian filters from Gaussian functions

We end this chapter with one fantastically important linear filter, and some variants based upon it. A *Gaussian filter* is a linear filter that also smooths an image and reduces noise. However, unlike a mean filter – for which even the furthest away pixels in the neighbourhood influence the result by the same amount as the closest pixels – the smoothing of a Gaussian filter is weighted so that the influence of a pixel decreases with its distance from the filter centre. This tends to give a better result in many cases (Figure 10.11).

The coefficients of a Gaussian filter are determined from a Gaussian function (Figure 10.12), and its size is controlled by a σ value – so when working with ImageJ’s `Gaussian Blur...` command, you will need to specify this rather than the filter radius. σ is equivalent to the standard deviation of a normal (i.e. Gaussian) distribution. A comparison of several filters is shown in Figure 10.13.

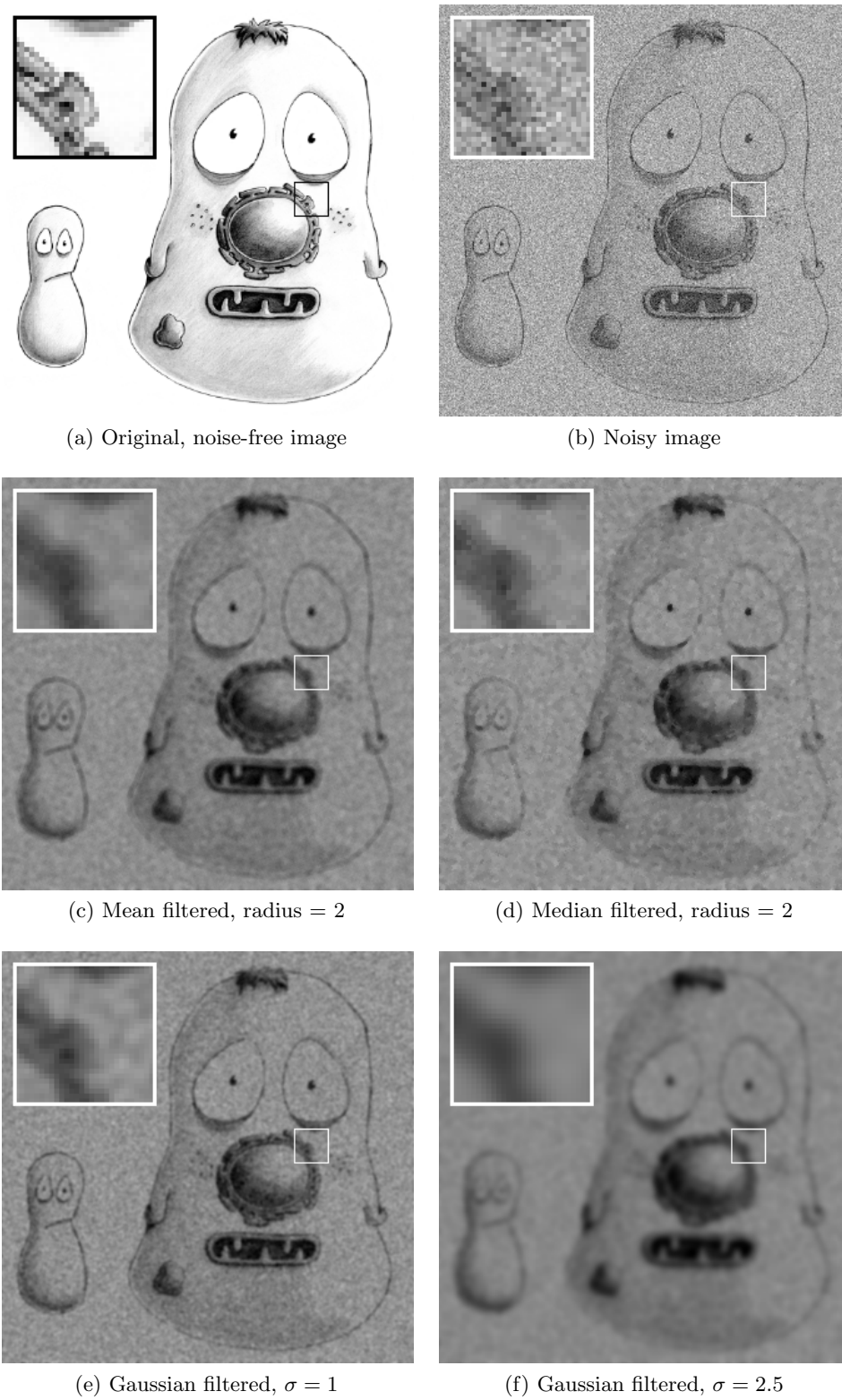


Figure 10.13: The effects of various linear and nonlinear filters upon a noisy image of fixed cells.

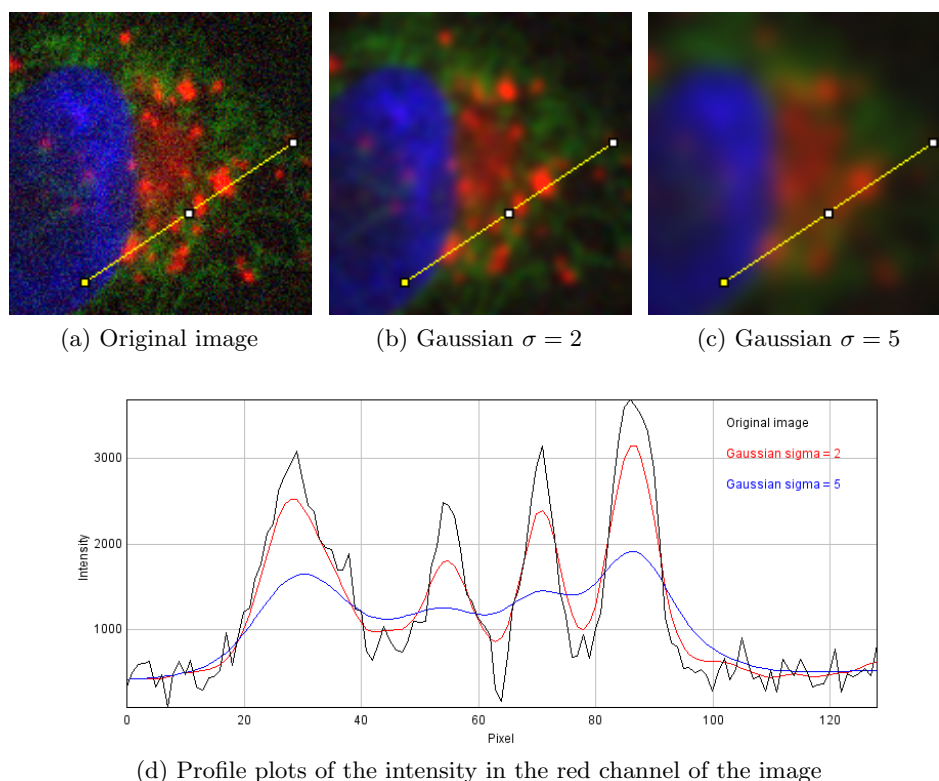


Figure 10.14: The effect of Gaussian filtering on the size and intensity of structures. The image is taken from File \rightarrow Open Samples \rightarrow HeLa Cells, with some additional simulated noise added to show that this is also reduced by Gaussian filtering.

10.4.2 Filters of varying sizes

Gaussian filters have useful properties that make them generally preferable to mean filters, some of which will be mentioned in Chapter 15 (others require a trip into Fourier space, beyond the scope of this book). Therefore if in doubt regarding which filter to use for smoothing, Gaussian is likely to be the safer choice.

A small filter will mostly suppress noise, because noise masquerades as tiny fluorescence fluctuations at individual pixels. As the filter size increases, Gaussian filtering starts to suppress larger structures occupying multiple pixels – reducing their intensities and increasing their sizes, until eventually they would be smoothed into surrounding regions (Figure 10.14). By varying the filter size, we can then decide the *scale* at which the processing and analysis should happen.

Figure 10.15 shows an example of when this is useful. Here, gradient magnitude images are computed similar to that in Figure 10.5, but because the original image is now noisy the initial result is not very useful – with even strong edges being buried amid noise (b). Applying a small Gaussian filter prior to computing the gradient magnitude gives much better results (c), but if we only wanted the very

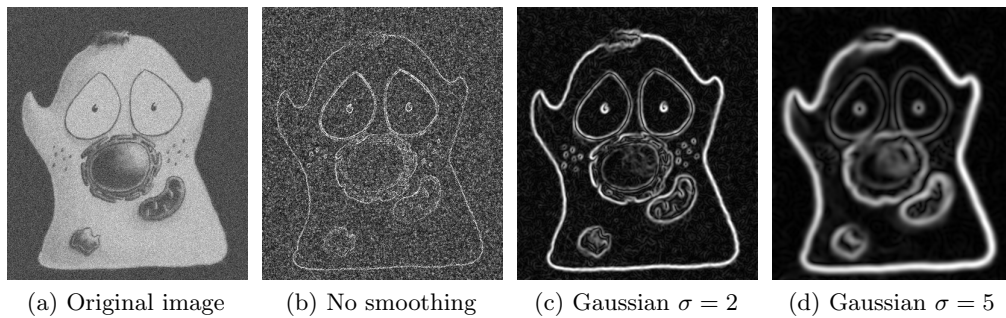


Figure 10.15: Applying Gaussian filters before computing the gradient magnitude changes the scale at which edges are enhanced.

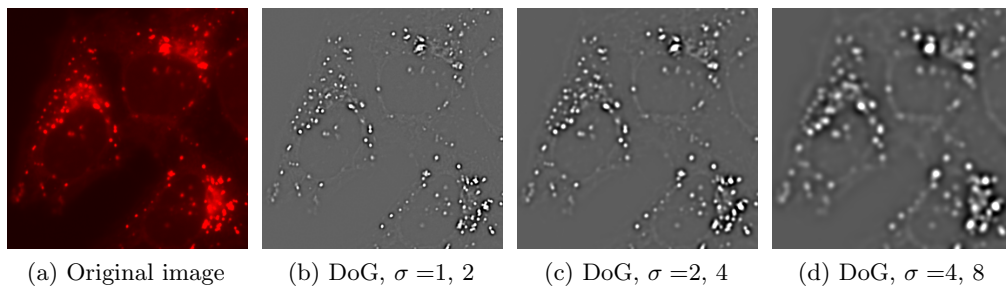


Figure 10.16: Difference of Gaussian filtering of the same image at various scales.

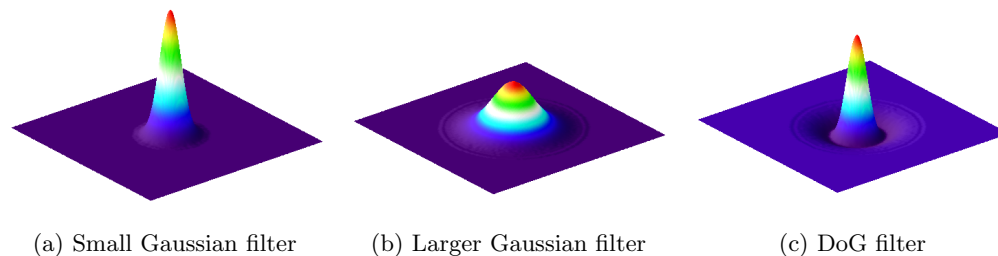


Figure 10.17: Surface plots of two Gaussian filters with small and large σ , and the result of subtracting the latter from the former. The sum of the coefficients for (a) and (b) is one in each case, while the coefficients of (c) add to zero.

strongest edges then a larger filter would be better (d).

10.4.3 Difference of Gaussians filtering

So Gaussian filters can be chosen to suppress small structures. But what if we also wish to suppress large structures – so that we can concentrate on detecting or measuring structures with sizes inside a particular range?

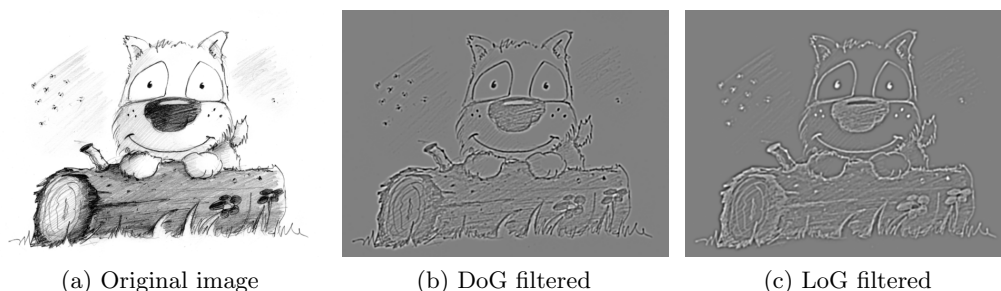


Figure 10.18: Application of DoG and LoG filtering to an image. Both methods enhance the appearance of spot-like structures, and (to a lesser extent) edges, and result in an image containing both positive and negative values with an overall mean of zero. In the case of LoG filtering, inversion is involved: darker points become bright after filtering.

We already have the pieces necessary to construct one solution. Suppose we apply one Gaussian filter to reduce small structures. Then we apply a second Gaussian filter, bigger than the first, to a duplicate of the image. This will remove even more structures, while still preserving the largest features in the image. But if we finally subtract this second filtered image from the first, we are left with an image that contains the information that ‘falls between’ the two smoothing scales we used. This process is called difference of Gaussians (DoG) filtering, and it is especially useful for detecting small spots or as an alternative to the gradient magnitude for enhancing edges (Figure 10.16).

DoG filters

In fact, to get the result of DoG filtering it is not necessary to filter the image twice and subtract the results. We could equally well subtract the coefficients of the larger filter from the smaller first (after making sure both filters are the same size by adding zeros to the edges as required), then apply the resulting filter to the image only once (Figure 10.17).

10.4.4 Laplacian of Gaussian filtering

One minor complication with DoG filtering is the need to select two different values of σ . A similar operation, which requires only a single σ and a single filter, is Laplacian of Gaussian (LoG) filtering. The appearance of a LoG filter is like an upside-down DoG filter (Figure 10.19), but if the resulting filtered image is inverted then the results are comparable⁴. You can test out LoG filtering in Fiji using `Plugins → Feature Extraction → FeatureJ → FeatureJ Laplacian`.

⁴A LoG filter is also often referred to as a *mexican-hat filter*, although clearly the filter (or the hat-wearer) should be inverted for the name to make more sense.

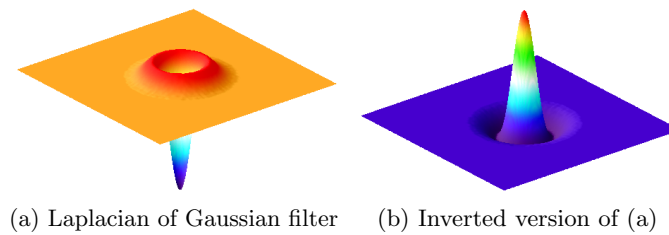


Figure 10.19: Surface plot of a LoG filter. This closely resembles Figure 10.18, but inverted so that the negative values are found in the filter centre.

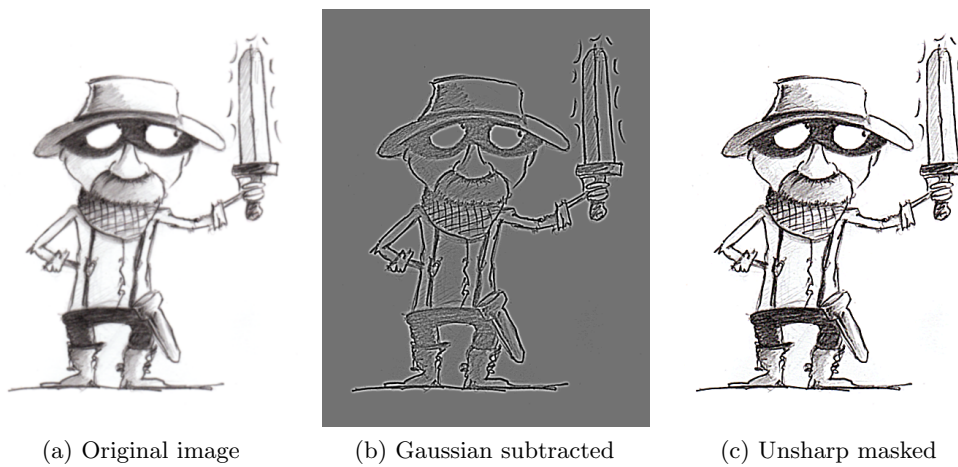


Figure 10.20: The application of unsharp masking to a blurred image. First a Gaussian-smoothed version of the image ($\sigma = 1$) is subtracted from the original, scaled (*weight* = 0.7) and added back to the original.

Practical 10.4

Why does the `FeatureJ Laplacian` command have a `Detect zero-crossings` option? If you are unsure, you can try it out on `DoG_on_Log.png`. You should also investigate the effects of changing the `Smoothing scale` option. *Solution*

10.4.5 Unsharp masking

Finally, a related technique widely-used to enhance the visibility of details in images – although *not* advisable for quantitative analysis – is *unsharp masking* (`Process` → `Filters` → `Unsharp mask...`).

This uses a Gaussian filter first to blur the edges of an image, and then subtracts it from the original. But rather than stop there, the subtracted image is multiplied by some weighting factor and *added back* to the original. This gives

an image that looks much the same as the original, but with edges sharpened by an amount dependent upon the chosen weight.

Unsharp masking can improve the visual appearance of an image, but it is important to remember that it modifies the image content in a way that might well be considered suspicious in scientific circles. Therefore, if you apply unsharp masking to any image you intend to share with the world you should have a good justification and certainly admit what you have done. If you want a more theoretically justified method to improve image sharpness, it may be worth looking into ‘(maximum likelihood) deconvolution’ algorithms.

Solutions

Question 10.1 Circles are more ‘compact’. Every point on the perimeter of a circle is the same distance from the centre. Therefore using a circular filter involves calculating the mean of all pixels a distance of $\leq \text{Radius}$ pixels away from the centre. For a square filter, pixels that are further away in diagonal directions than horizontal or vertical directions are allowed to influence the results. If a pixel is further away, it is more likely to have a very different value because it is part of some other structure.

Question 10.2 A new image is needed for the algorithm to work. Otherwise, if we put the results directly into the image we are still filtering then we would only be guaranteed to get the correct result for the first pixel. For later pixels, computations would involve a combination of original and already-filtered pixel values, which would most likely give a different final result.

Question 10.3 If n is an odd number, the filter has a clear central pixel. This makes things easier whenever using the approach outlined in Figure 10.4.

Question 10.4 The results of convolving with a single -1 coefficient in different circumstances:

1. *Normalize Kernel is checked*: Nothing at all happens. The normalization makes the filter just a single 1... and convolving with a single 1 leaves the image unchanged.
2. *You have a 32-bit image (Normalize Kernel unchecked)*: The pixel values become negative, and the image looks inverted.
3. *You have an 8-bit image (Normalize Kernel unchecked)*: The pixel values would become negative, but then cannot be stored in an 8-bit unsigned integer form. Therefore, all pixels simply become clipped to zero.

Practical 10.1 The process to calculate the gradient magnitude is:

1. Convert the image to 32-bit and duplicate it
2. Convolve one copy of the image with the horizontal gradient filter, and one with the vertical
3. Compute the square of both images (**Process** \rightarrow **Math** \rightarrow **Square**)

4. Use the image calculator to add the images together
5. Compute the square root of the resulting image (**Process** → **Math** → **Square Root**)

Note that this process goes wrong if the image is stored in some unsigned integer format, since it needs negative values.

Question 10.5 After applying a gradient filter, the image mean will be 0: every pixel is added once and subtracted once when calculating the result. If you are sceptical about this: test it, making sure the image is 32-bit first.

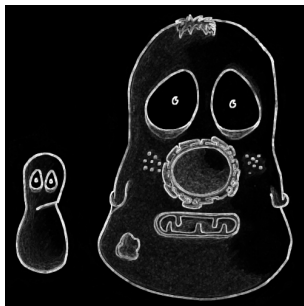
Practical 10.2 The **edges** LUT shows most low and high pixel values as black – and uses lighter shades of gray only for a small range of values in between (see **Image** → **Color** → **Edit LUT...**). In any image with a good separation of background and foreground pixels, but which still has a somewhat smooth transition between them, this means everything but the edges can appear black.

Question 10.6 At the time of writing, the **Convolve...** command actually applies correlation!

Practical 10.3 Replication of boundary pixels is the default method used by **Convolve...** in ImageJ (although other filtering plugins by different authors might use different methods).

My approach to test this involved using **Convolve...** with a filter that consisting of a 1 followed by a lot of zeros (e.g. 1 0 0 0 0 0 0 0 0 0 0 0). This basically shifts the image to the right, bringing whatever is outside the image boundary into view.

Question 10.7 Subtracting a minimum from a maximum filtered image would be another way to accent the edges:



Practical 10.4 After computing the LoG filtered image, **Detect zero-crossings** identifies locations where the sign of adjacent pixels changes from negative-to-positive (or vice versa). Thanks to the properties of the Laplacian operator, these correspond to edges. Unfortunately, they often give rather a lot more edges than you might like – but by increasing the **Smoothing scale** parameter, the sigma value is increased to cause more smoothing and thereby reduce the edges caused by small structures or noise (see Figure 10.15).

Binary images

Chapter outline

- *Morphological operations can be used to refine or modify the shapes of objects in binary images*
- *The distance & watershed transforms can be used to separate close, round structures*

11.1 Introduction

By means of filtering and thresholding, we can create binary images to detect structures of various shapes and sizes for different applications. Nevertheless, despite our best efforts these binary images often still contain inaccurate or undesirable detected regions, and could benefit from some extra cleaning up. Since at this stage we have moved away from the pixel values of the original image and are working only with shapes – morphology – the useful techniques are often called *morphological operations*. In ImageJ, several of these are to be found in the `Process` → `Binary` submenu.

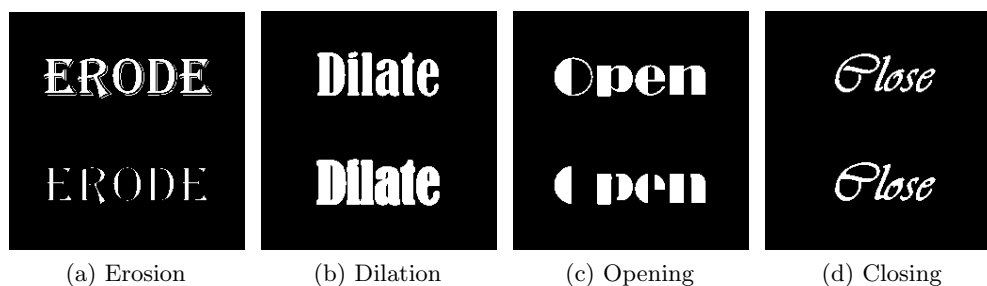


Figure 11.1: The effects of the `Erode`, `Dilate`, `Open` and `Close`- commands. The original image is shown at the top, while the processed part is at the bottom in each case.

11.2 Morphological operations using rank filters

11.2.1 Erosion & dilation

Our first two morphological operations, *erosion* and *dilation*, are actually identical to minimum and maximum filtering respectively, described already in Section 10.3.1. The names erosion and dilation are used more often when speaking of binary images, but the operations are the same irrespective of the kind of image. Erosion will make objects in the binary image smaller, because a pixel will be set to the background value if *any* other pixels in the neighbourhood are background. This can split single objects into multiple ones. Conversely, dilation makes objects bigger, since the presence of a single foreground pixel anywhere in the neighbourhood will result in a foreground output. This can also cause objects to merge.

11.2.2 Opening & closing

The fact that erosion and dilation alone affect sizes can be a problem: we may like their abilities to merge, separate or remove objects, but prefer that they had less impact upon areas and volumes. Combining both operations helps achieve this.

Opening consists of an erosion followed by a dilation. It therefore first shrinks objects, and then expands them again to an *approximately* similar size. Such a process is not as pointless as it may first sound. If erosion causes very small objects to completely disappear, clearly the dilation cannot make them reappear: they are gone for good. Barely-connected objects separated by erosion are also not reconnected by the dilation step.

Closing is the opposite of opening, i.e. a dilation followed by an erosion, and similarly changes the shapes of objects. The dilation can cause almost-connected objects to merge, and these often then remain merged after the erosion. If you wish to count objects, but these are wrongly subdivided in the segmentation, closing may therefore help make the counts more accurate.

These operations are implemented with the **Erode**, **Dilate**, **Open** and **Close** commands – but only using 3×3 neighbourhoods. To perform the operations with larger neighbourhoods, you can simply use the **Maximum...** and **Minimum...** filters, combining them to get opening and closing if necessary. Alternatively, in Fiji you can explore **Process** → **Morphology** → **Gray Morphology**.

11.3 Outlines, holes & maximum

The **Outline** command, predictably, removes all the interior pixels from 2D binary objects, leaving only the perimeters (Figure 11.2a). **Fill Holes** would then fill these interior pixels in again, or indeed fill in any background pixels that are completely surrounded by foreground pixels (Figure 11.2b). **Skeletonize** shaves off all the outer pixels of an object until only a connected central line remains

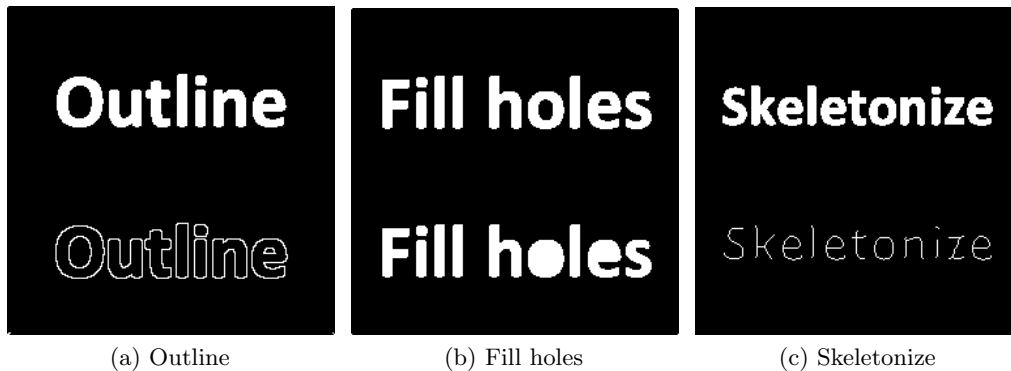


Figure 11.2: The effects of the `Outline`, `Fill holes` and `Skeletonize` commands.

(Figure 11.2c). If you are analyzing linear structures (e.g. blood vessels, neurons), then this command or those in Fiji's `Plugins` → `Skeleton` submenu may be helpful.

Question 11.1

The outline of an object in a binary image can also be determined by applying one other morphological operation to a duplicate of the image, and then using the `Image Calculator`. How?

Solution

11.4 Using image transforms

An image transform converts an image into some other form, in which the pixel values can have a (sometimes very) different interpretation. Several transforms are relevant to refining image segmentation.

11.4.1 The distance transform

The distance transform replaces each pixel of a binary image with the distance to the closest background pixel. If the pixel itself is already part of the background then this is zero (Figure 11.3c). It can be applied using the `Process` → `Binary` → `Distance Map` command, and the type of output given is determined by the `EDM output` option under `Process` → `Binary` → `Options...` (where EDM stands for 'Euclidean Distance Map'). This makes a difference, because the distance between two diagonal pixels is considered $\sqrt{2} \approx 1.414$ (by Pythagoras' theorem), so a 32-bit output can give more exact straight-line distances without rounding.

A natural question when considering the distance transform is: why? Although you may not have a use for it directly, with a little creative thinking it can help solve some other problems rather elegantly. For example, `Ultimate Points` uses

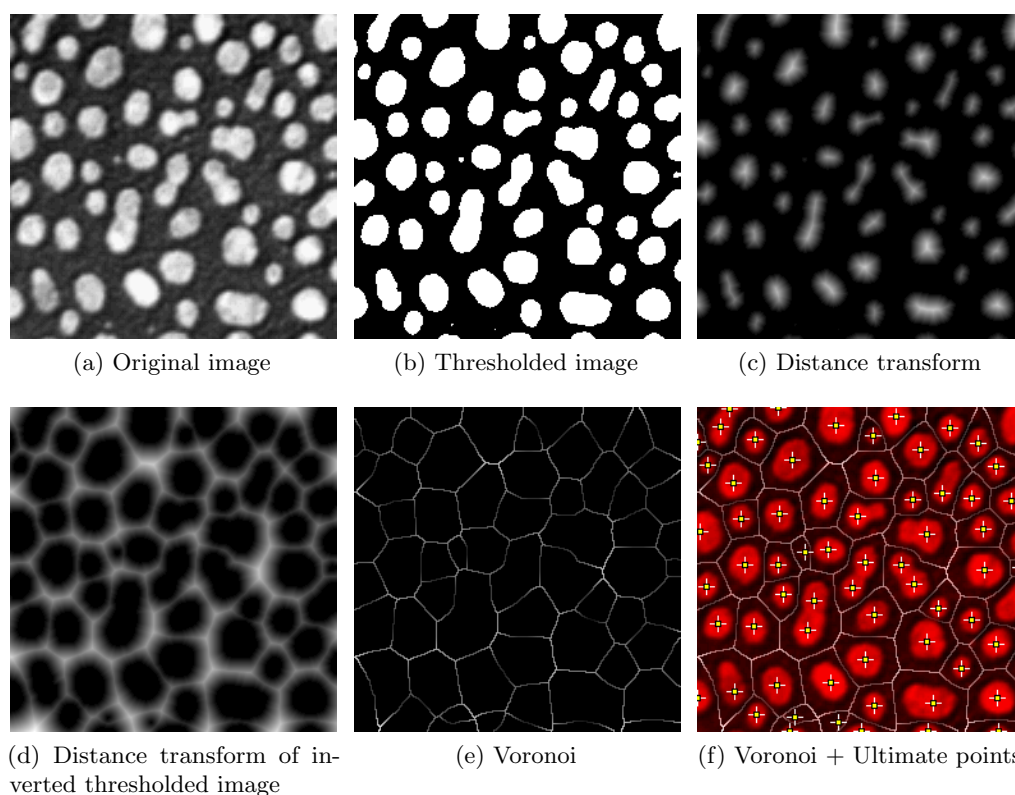


Figure 11.3: Applying the distance transform and related commands to a segmented (using Otsu’s threshold) version of the sample image `Blobs`. In (f), the original image is shown in the red channel, the Voronoi lines shown in gray and the ultimate points added as a `Point selection`.

the distance transform to identify the last points that would be removed if the objects would be eroded until they disappear. In other words, it identifies centres. But these are not simply single centre points for each object; rather, they are maximum points in the distance map, and therefore the pixels furthest away from the boundary. This means that if a structure has several ‘bulges’, then an ultimate point exists at the centre of each of them. If segmentation has resulted in structures being merged together, then each distinct bulge could actually correspond to something interesting – and the number of bulges actually means more than the number of separated objects in the binary image (Figure 11.3f).

Alternatively, if the distance transform is applied to an *inverted* binary image, the pixel values give the distance to the closest foreground object (Figure 11.3d). With this, the `Voronoi` command partitions an image into different regions so that the separation lines have an equal distance to the nearest foreground objects. Now suppose we detect objects in two separate channels of an image, and we want to associate those in the second channel with the closest objects in the first. We

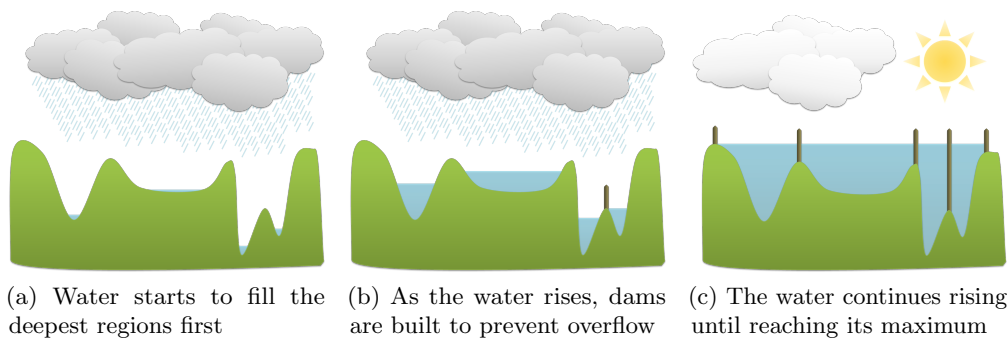


Figure 11.4: A schematic diagram showing the operation of the watershed transform in 1 dimension. If you imagine rain falling on a hilly surface, the deepest regions fill up with water first. As the water rises further, soon the water from one of these regions would overflow into another region – in which case a dam is built to prevent this. The water continues to rise and dams added as necessary, until finally when viewed from above every location is either covered by water or belongs to a dam.

could potentially calculate the distance of all the objects from one another, but this would be slow and complicated. Simply applying *Voronoi* to the first channel gives us different partitions, and then we only need to see which partition each object in the second channel falls into (see Figure 11.3f). This will automatically be the same partition as the nearest first-channel object.

Question 11.2

Imagine you have created a binary image containing detected cells, but you are only interested in the region inside the cell that is close to the membrane, i.e. within 5 pixels of the edge of each detected object. Any pixels outside the objects or closer to their centres do not matter. How would you go about finding these regions?

Note: There are a few ways using the techniques in this chapter, although these do not necessarily give identical results. *Solution*

11.4.2 The watershed transform

The watershed transform provides an alternative to straightforward thresholding if you need to partition an image into many different objects. To understand how it works, you should imagine the image as an uneven surface in which the value of each pixel corresponds to a height. Now imagine water falling evenly upon this surface and slowly flooding it. The water gathers first in the deepest parts; that is, in the places where pixels have values lower than all their neighbours. Each of these we can call a water basin.

As the water level rises across the image, occasionally it will reach a ridge

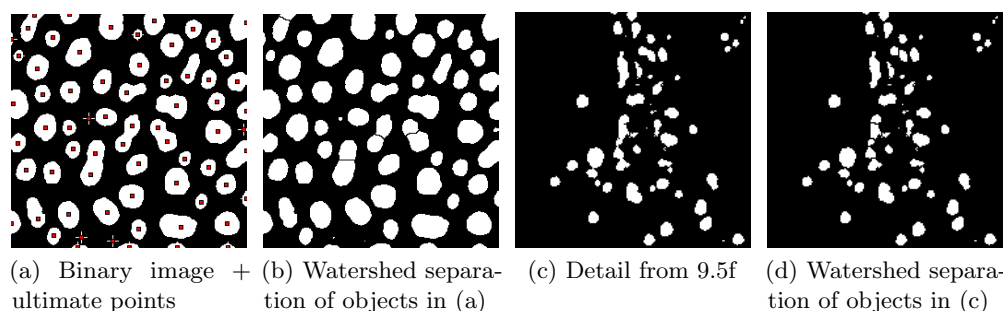


Figure 11.5: Applying the watershed transform to the binary images from Figures 11.3 and 9.5 is able to separate some of the merged spots based upon their shape.

between two basins – and in reality water could spill from one basin into the other. However, in the watershed transform this is not permitted; rather a dam is constructed at such ridges. The water then continues to rise, with dams being built as needed, until in the end every pixel is either part of a basin or a ridge, and there are exactly the same number of basins afterwards as there were at first.

This is the general principle of the watershed transform, and is illustrated in Figure 11.4. ImageJ’s `Watershed` command is a further development of this, in which the *watershed transform is applied to the distance transform* of a binary image, where the distance transform has also been inverted so that the centres of objects (which, remember, are just the *ultimate points* mentioned above) now become the deepest parts of the basins that will fill with water. The end result is that any object that contains multiple ultimate points has ridges built inside it, separating it into different objects. If we happen to have wanted to detect roundish structures that were unfortunately merged together in our binary image, then this may well be enough to un-merge them (Figure 11.5).

Intensity-based watershed

To apply the watershed transform to the original data, see `Process` → `Find Maxima...` with the output type `Segmented Particles`

Visualizing the watershed transform

If you turn on `Debug mode` under `Edit` → `Options` → `Misc...` and then run the `Watershed` command on an image, a stack is created in which you can see the individual steps of the transform – and perhaps get a better idea of what it is doing.

Solutions

Question 11.1 To outline the objects in a binary image, you can simply calculate the difference between the original image and an eroded (or dilated, if you want the pixels just beyond the objects) version.

Question 11.2 Two ways to find the region close to the boundaries of detected cells:

1. Compute the distance transform of the image. Run `Threshold...`, choose `Set` and enter `Lower Threshold Level: 1` and `Higher Threshold Level: 5`.
2. Duplicate the original binary image, then erode the duplicated version using a `Minimum` filter, `Radius = 5 pixels`. Compute the difference between this and the original using the `Image Calculator...`

There are more possible ways, such as applying `Outline`, dilating the result and then excluding pixels that fall outside the original cell regions... but that is a bit more work.

Processing data with higher dimensions

Chapter outline

- *Many processing operations can be extended into more than 2 dimensions*
- *Adding extra dimensions can greatly increase the processing requirements*
- *Objects can be detected & measured in 3D*

12.1 Introduction

So far, we have concentrated on only 2D images. Most of the operations we have considered can also be applied to 3D data – and sometimes data with more dimensions, in cases where this is meaningful.

12.2 Point operations, contrast & conversion

Point operations are straightforward: they depend only on individual pixels, so the number of dimensions is unimportant. Image arithmetic involving a 3D stack and a 2D image can also be carried out in ImageJ using the **Image Calculator**, where the operation involving the 2D image is applied to each slice of the 3D stack in turn. Other options, such as filtering and thresholding, are possible, but bring with them extra considerations – and often significantly higher computational costs.

Setting the LUT of a 3D image requires particular care. The normal **Brightness/Contrast...** tool only takes the currently-displayed slice into consideration when pressing **Reset** or **Auto**. Optimizing the display for a single slice does not necessarily mean the rest of the stack will look reasonable if the brightness changes much. **Process** → **Enhance Contrast...** is a better choice, since here you can specify that the information in the entire stack should be used. You can also specify the percentage of pixels that should be saturated (clipped) *for display*, i.e. those that should be shown with the first or last colours in the LUT. So long as **Normalize** and **Equalize histogram** are not selected, the pixel values will not be changed.

Question 12.1

By default, the percentage of saturated pixels in `Enhance Contrast...` is set to 0.4. Why might this be chosen instead of 0?

Solution

Converting bit-depths of multidimensional images

As described in Section 3.4, the minimum and maximum display range values are used by default when reducing the bit-depth of an image. To minimize the information lost, these should be set to the minimum and maximum pixel values within the image – otherwise values will be clipped. In 2D it is enough to press **Reset** in the **Brightness/Contrast** window, *but in 3D this will only work if the minimum and maximum values from the entire stack happen to appear on the current slice!* Therefore it is good practice to run `Enhance Contrast...` prior to reducing bit-depths of stacks, setting the saturation to 0 and using the entire stack. This means no pixels will be clipped in the output (although rescaling and rounding will still occur).

12.3 3D filtering

Many filters naturally lend themselves to being applied to as many dimensions as are required. For example, a 3×3 mean filter can easily become a $3 \times 3 \times 3$ filter if averaging across slices is allowed. Significantly, it then replaces each pixel by the average of 27 values, rather than 9. This implies the reduction in noise is similar to that of applying a 5×5 filter (25 values), but with a little less blurring in 2D and a little more along the third dimension instead. Several 3D filters are available under the `Plugins` → `Process` → submenu.

12.3.1 Fast, separable filters

The fact that 3D filters inherently involve more pixels is one reason that they tend to be slow. However, if a filter happens to have the very useful property of *separability*, then its speed can be greatly improved. Mean and Gaussian filters have this property, as do minimum and maximum filters (of certain shapes) – but not median.

The idea is that instead of applying a single $n \times n \times n$ filter, three different 1D filters of length n can be applied instead – rotated so that they are directed along each dimension in turn. Therefore rather than n^3 multiplications and additions being required to calculate each pixel in the linear case, as in Figure 10.4, only $3n$ multiplications and additions are required. With millions of pixels involved, even when n is small the saving can be huge. Figure 12.1 shows the basic idea in 2D, but it can be extended to as many dimensions as needed. `Process` → `Filters` → `Gaussian Blur 3D...` uses this approach.

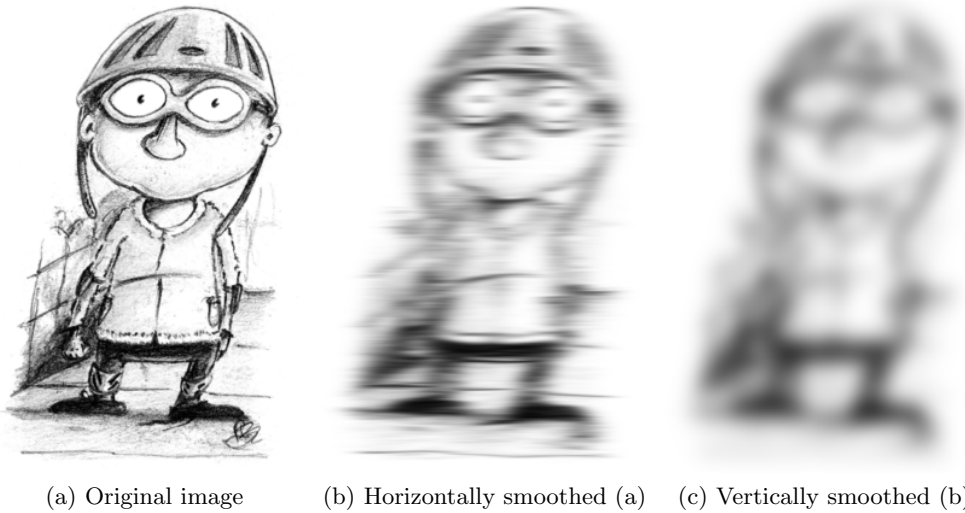


Figure 12.1: Smoothing applied separably using a 1D Gaussian filter (first aligned horizontally, then vertically) to an image of a rather well-protected young man seen playing safely on the streets of Heidelberg in 2010. The end result (c) is the same as what would be obtained by applying a single, 2D Gaussian filter to (a).

Fast filters & the Fourier transform

Not all linear filters are separable, and applying a large, non-separable linear filter can also be extremely time-consuming. However, when this is the case a whole other method can be used to get the same result using the Fourier transform – where the speed no longer has the same dependence upon the filter size. Unfortunately, the Fourier method cannot be used for non-linear filters such as the median filter.

12.3.2 Dimensions & isotropy

If applying a filter in 3D instead of 2D, it may seem natural to define it as having the same size in the third dimension as in the original two. But for a z -stack, the spacing between slices is usually larger than the width and height of a pixel. And if the third dimension is time, then it uses another scale entirely. Therefore more thought usually needs to be given to what sizes make most sense.

In some commands (e.g. `Plugins → Processor → Smooth (3D)`), there is a `Use calibration` option to determine whether the values you enter are defined in terms of the units found in the `Properties...` and therefore corrected for the stored pixel dimensions. Elsewhere (e.g. `Gaussian Blur 3D...`) the units are pixels, slices and time points – and so you are responsible for figuring out how to compensate for different scales and units.

Isotropic filters

A filter is said to be *isotropic* if it has the same effect along all dimensions.

12.4 Thresholding multidimensional data

When thresholding an image with more than 2 dimensions using the `Threshold...` command, it is necessary to choose whether the threshold should be determined from the histogram of the entire stack, or from the currently-visible 2D slice only. If the latter, you will also be asked whether the same threshold should be used for every slice, or if it should be calculated anew for each slice based upon the slice histogram. In some circumstances, these choices can have a very large impact upon the result.

Question 12.2

When you threshold a stack, you have an option to choose `Stack Histogram`. Then, when you choose `Apply` you are asked if you want to `Calculate Threshold for Each Image`. What difference do you expect these two options to make, and what combinations would you use for:

1. a stack consisting of 2D images from different colour channels
2. a *z*-stack
3. a time series

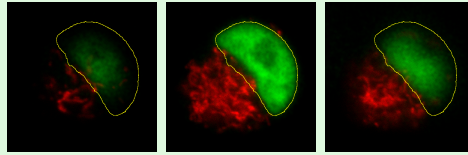
Note: Have a look at what happens when you click `Auto` while scrolling through one channel of the stack `File` → `Mitosis` (26 MB, 5D Stack), with and without `Stack Histogram` selected. You will have to split the channels for this because ImageJ currently refuses to threshold multichannel images with extra dimensions (which helps avoid some confusion). `Dark Background` should always be selected here. *Solution*

12.5 Measurements in 3D data

ImageJ has good support for making measurements in 2D, particularly the `Measure` and `Analyze Particles...` commands. The latter can happily handle 3D images, but only by creating and measuring 2D ROIs independently on each slice. Alternatively, `Image` → `Stacks` → `Plot Z-axis Profile` is like applying `Measure` to each slice independently, making measurements either over the entire image or any ROI. It will also plot the mean pixel values, but even if you do not particularly want this the command can still be useful. However, if single measurements should be made for individual objects that extend across multiple slices, neither of these options would be enough.

Practical 12.1

Suppose you have a cell, nucleus or some other large 3D structure in a z -stack, and you want to draw the smallest 2D ROI that completely contains it on every slice. An example is shown on the right for the green structure in the **Confocal Series** sample image.

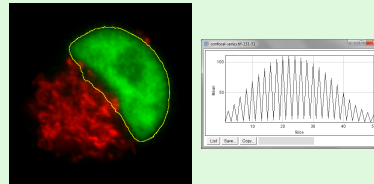


How would you create such a ROI, and be confident that it is large enough for all slices?

Solution

Practical 12.2

When I create a ROI on the sample image **Confocal Series** and run **Plot Z-axis Profile**, I get a strangely spikey result (shown right). How could this be explained?



(I used Fiji/ImageJ 1.46k. This behaviour has been corrected since then.)

Solution

12.5.1 Histograms & threshold clipping

One way to measure in 3D is to use the **Histogram** command and specify that the entire stack should be included – this provides some basic statistics, including the total number, mean, minimum, maximum and standard deviation of the pixels¹. This will respect the boundaries of a 2D ROI if one has been drawn.

This is a start, but it will not adjust to changes in the object boundary on each 2D plane. A better approach could be to use **Image** → **Adjust** → **Threshold...** to set a threshold that identifies the object – but do *not* press **Apply** to generate a binary image. Rather, under **Analyze** → **Set Measurements...** select **Limit to threshold**. Then when you compute the stack histogram (or press **Measure** for 2D) only above-threshold pixels will be included in the statistics. Just be sure to reset **Limit to threshold** later.

Question 12.3

How can you translate the total number of pixels in an object into its volume, e.g. in μm^3 ? Give some thought to how accurate your method will be.

Solution

¹Be careful! If you have multiple channels, these should be split first.

Using NaNs

I am not a fan of **Limit to threshold**, because I am likely to forget to reset it afterwards and may subsequently measure the wrong things for days thereafter.

An alternative that I prefer is to set my threshold on a 32-bit copy of the image I am working with, and then **Apply** the threshold using the **Set Background Pixels to NaN** option. Then all below-threshold pixels will automatically be excluded from any measurements I make on the result, since they are ‘no longer numbers’ (see Section 9.4.1).

12.5.2 The 3D Objects Counter

Currently, the closest thing to **Analyze Particles...** for measuring connected objects in 3D built-in to Fiji is the 3D Objects Counter (**Analyze** → **3D Objects Counter**)². Its settings (analogous to **Set Measurements...**) are under **Analyze** → **3D OC Options**. In addition to various measurements, it provides labelled images as output, either of the entire objects or only their central pixels – optionally with labels, or expanded to be more visible.

Find Connected Regions

Plugins → **Process** → **Find Connected Regions** is a command primarily for creating labelled images from thresholded 3D data, which can also give the total number of pixels per object. If the main thing you want is the labelled image without many more results, it may be faster than **3D Objects Counter**.

Additional 3D tools

For working with 3D data, it may be very useful to download the ‘3D ImageJ Suite’ from http://imagejdocu.tudor.lu/doku.php?id=plugin:stacks:3d_ij_suite:start. This not only includes a range of fast filters and commands for segmentation, but also a 3D version of the ROI Manager.

While created for bone image analysis, *BoneJ* (<http://bonej.org/>) also includes some components that are useful for general applications – including a fast 3D Particle Analyser (another alternative to the 3D Objects Counter) and a tool to interpolate ROIs across image slices.

²See S Bolte and F P Cordelières. “A guided tour into subcellular colocalization analysis in light microscopy.” In: *Journal of microscopy* 224.Pt 3 (Dec. 2006), pp. 213–32. ISSN: 0022-2720. DOI: 10.1111/j.1365-2818.2006.01706.x. URL: <http://www.ncbi.nlm.nih.gov/pubmed/17210054>

Solutions

Question 12.1 If the percentage of saturated pixels is 0, then the minimum and maximum pixel values throughout the image will be given the first and last LUT colours respectively, with all other colours devoted to values in between. This is sensitive to outliers, and often results in images that have poor contrast. It is usually better to accept some small amount of visual saturation, in order to give more LUT colours to the pixel values between the extremes.

Question 12.2 If **Stack Histogram** is checked, the thresholds are computed from a histogram of all the pixels in the entire image stack; otherwise, the histogram of only the currently-displayed image slice is used. BUT! If **Calculate Threshold for Each Image** is chosen, then this is ignored: the threshold is always determined by the selected automatic method using the histogram of the corresponding slice only.

Therefore, the most sensible combinations of thresholding options to use depend upon the type of data.

1. *Colour channels* – There is often no good reason to suppose the amount of fluorescence in different colour channels will be similar, and so thresholds should be calculated from each channel independently.
2. *z-Stacks* – It is normally a good idea to use the stack histogram with *z*-stacks. If you do not, then your threshold will be affected by whatever slice you happen to be viewing at the time of thresholding – introducing a potentially weird source of variability in the results. It is probably *not* a good idea to calculate a new threshold for each slice, because this would lead to at least *something* being detected on every slice. But in the outer slices there may well only be blur and noise – in which case nothing *should* be detected!
3. *Time series* – In a time series, bleaching can sometimes cause the image to darken over time. In such a case, using the stack histogram might cause fewer pixels to exceed the threshold at later time points simply for this reason, and recalculating the threshold for each image may be better. On the other hand, if images were previously normalized somehow to compensate for bleaching³, then the stack threshold might be preferable again. It's tricky.

There is one other implementation issue that needs attention. When **Dark Background** is checked and an automated threshold is computed, then it is only really the low threshold that matters – the high threshold is always set to the maximum in the histogram to ensure that all brighter pixels are designated 'foreground' in the result. However, if not using **Stack Histogram**, then for

³See http://fiji.sc/wiki/index.php/Bleach_Correction

non-8-bit images the histograms are calculated using the minimum and maximum pixels on the slice, and consequently the high threshold cannot be higher than this maximum value (look at how the high threshold value changes in `Mitosis` as you compute auto thresholds for different slices). This means that any brighter pixels will be *outside* the threshold range (and therefore ‘background’) if they occur on a different slice. This can cause holes to appear in the brightest parts of structures, and is probably not what you want. A similar situation occurs with the low threshold when `Dark Background` is unchecked.

Practical 12.1 My strategy would be to create a z -projection (max intensity) and then draw the ROI on this – or, preferably, create the ROI by thresholding using `Image` → `Adjust` → `Threshold` and the `Wand` tool. This ROI can then be transferred over to the original stack, either via the ROI Manager or `Edit` → `Selection` → `Restore Selection`.

Practical 12.2 `Confocal Series` has two channels: it is a hyperstack (4D). But `Plot Z-axis Profile` ignored this previously, and treated it like a stack (3D). Therefore measurements from each channel were interleaved with one another. Splitting the channels first, then calculating the profiles separately would overcome this.

Question 12.3 You could treat each pixel as a rectangular cuboid, with a volume equal to $pixel\ width \times pixel\ height \times voxel\ depth$ (as given in `Image` → `Properties...`). Then multiply this by the number of pixels within the object. This is what the `3D Objects Counter` plugin does when measuring volumes (Section 12.5.2).

Whenever you want to compare object sizes across images acquired with different pixel sizes, this is certainly better than just taking the raw pixel counts as measures of volume. However, it is unlikely to be very accurate – and volume measurements obtained this way should not be trusted too much, especially when dealing with very small sizes. They are also likely to be quite sensitive to z -stack spacing.

Writing macros

Chapter outline

- *Processing & analysis steps can be automated by writing macros*
- *Straightforward macros can be produced without any programming using the Macro Recorder*
- *Recorded macros can be modified to make them more robust & suitable for a wider range of images*

13.1 Introduction

It is one thing to figure out steps that enable you to analyze an image, it is quite another to implement these steps for several – and perhaps many – different images. Without automation, the analysis might never happen; all the mouse-moving and clicking would just be too time-consuming, error-prone or boring, and momentarily lapses in concentration could require starting over again.

Even a brief effort to understand how to automate analysis can produce vast, long-lasting improvements in personal productivity and sanity by reducing the time spent on mind-numbingly repetitive tasks. In some straightforward cases (e.g. converting file formats, applying projections or filters, or making measurements across entire images), this can already be done in ImageJ using the commands in the **Process** → **Batch** → submenu and no programming whatsoever. But it is also very worthwhile to get some experience in producing macros, scripts or plugins, after which you can add your own new commands to the menus and carry out customized algorithms with a single click of a button or press of a key.

Macros are basically sequences of commands, written in some programming language (here ImageJ's own macro language), which can be run automatically to make processing faster and easier. This chapter is far from an extensive introduction to macro-writing, but rather aims to introduce the main ideas quickly using a worked example. Should you wish to delve deeper into the subject, there is an introduction to the language on the ImageJ website¹, and a very helpful tutorial on the Fiji wiki², while the list of built-in macro functions is an indispensable reference³. Once confident with macros, the next step would be to enter the

¹<http://imagej.net/developer/macro/macros.html>

²http://fiji.sc/wiki/index.php/Introduction_into_Macro_Programming

³<http://imagej.net/developer/macro/functions.html>

world of scripts and plugins. These can be somewhat more difficult to learn, but reward the effort with the ability to do more complicated things. Links to help with this are available on the developer section of the ImageJ website at <http://imagej.net/developer>.

Finally, although it is possible to use ImageJ rather than Fiji to create macros, Fiji's script editor makes the process much easier by colouring text according to what it does, so I will assume you are using this.

13.2 A Difference of Gaussians filter

Difference of Gaussians (DoG) filtering was introduced in Chapter 10.4.3 as a technique to enhance the appearance of small spots and edges in an image. It is quite straightforward, but time consuming to apply manually very often – and you might need to experiment with the filter sizes a bit to get good results. This makes it an excellent candidate for a macro.

Recording a macro

Rather than diving into writing the code, the fastest way to get started is to have ImageJ do most of the hard work itself. Then you only need to fix up the result. The procedure is as follows:

- Open up an example (2D, non-colour) image to use, ideally one including small spot-like or otherwise round objects. I am using the image found under **File** → **Open Samples** → **HeLa Cells**, after extracting the red channel only.
- Start the *Macro Recorder* by choosing **Plugins** → **Macros** → **Record**. Make sure that **Record: Macro** appears at the top of this window (see the drop-down list). Every subsequent click you make that has a corresponding macro command will result in the command being added to the window.
- Convert your image to 32-bit. This will reduce inaccuracies due to rounding whenever the filtering is applied.
- Duplicate the image.
- Apply **Process** → **Filters...** → **Gaussian Blur...** to one of the images (it does not matter if it is the original or the duplicate), using a small sigma (e.g. 1) for noise suppression.
- Apply **Gaussian Blur...** to the other image, using a larger sigma (e.g. 2).
- Run **Process** → **Image Calculator...** and subtract the second filtered image from the first. This produces the 'difference of Gaussians' filtered image, in which small features should appear prominently and the background is removed.

- Press the **Create** button on the macro recorder. This should cause a text file containing the recorded macro to be opened in Fiji's **Script Editor** (which you can find under **Plugins** → **Scripting** → **Script Editor**).
- Save the text file in the plugins folder of Fiji. The file name should end with the extension `.ijm` (for 'ImageJ Macro'), and include an underscore character somewhere within it.

Now you have a macro! To try it out, close Fiji completely, then start it again and reopen the original image you used. There should be a new command in the **Plugins** menu for the macro you have just created⁴. Running this new command on your example image should give you the same result as when you applied the commands manually. (If not, keep reading anyway and the following steps should fix it.)

Cleaning up

Now reopen your macro in the Script Editor. It should look something like mine:

```
run("Find Commands...");
run("32-bit");
//run("Brightness/Contrast...");
run("Enhance Contrast", "saturated=0.35");
run("Duplicate...", "title=C1-hela-cells-1.tif");
run("Find Commands...");
run("Gaussian Blur...", "sigma=1");
selectWindow("C1-hela-cells.tif");
run("Find Commands...");
run("Gaussian Blur...", "sigma=2");
run("Find Commands...");
imageCalculator("Subtract create", "C1-hela-cells-1.tif",
    "C1-hela-cells.tif");
selectWindow("Result of C1-hela-cells-1.tif");
```

Your code is probably not identical, and may well be better. One problem with automatically generated macros is that they contain (almost) *everything* – often including a lot of errant clicking, or other non-essential steps. For example, I am particularly fond of pressing **L** to bring up the **Find Commands** box, but these references should be removed from the macro. I also changed the contrast of an image, but this was only to look at it – and it does not need to be included in the macro. After deleting the unnecessary lines, I get:

```
run("32-bit");
run("Duplicate...", "title=C1-hela-cells-1.tif");
run("Gaussian Blur...", "sigma=1");
```

⁴Without an underscore in the file name, the command will not be added to the menu.

```
selectWindow("C1-hela-cells.tif");
run("Gaussian Blur...", "sigma=2");
imageCalculator("Subtract create", "C1-hela-cells-1.tif",
               "C1-hela-cells.tif");
```

Understanding the code

You can most likely work out what the macro is doing, if not necessarily the terminology, just by looking at it. Taking the first line, `run` is a *function* that tells ImageJ to execute a command, while `"32-bit"` is a piece of text (called a *string*) that tells it which command. Functions always tell ImageJ to do something or give you information, and can be recognized because they are normally followed by parentheses. Strings are recognizable both because they are inside double inverted commas and the script editor shows them in a different colour. Notice also that each line needs to end with a semicolon so that the macro interpreter knows the line is over.

Functions can require different numbers of pieces of information to do their work. At a minimum, `run` needs to know the name of the command and the image to which it should be applied – which here is taken to be whichever image is currently active, i.e. the one that was selected most recently. But if the command being used by `run` requires extra information of its own, then this is included as an extra string. Therefore

```
run("Duplicate...", "title=C1-hela-cells-1.tif");
```

informs the `Duplicate` command that the image it creates should be called `C1-hela-cells-1.tif`, and

```
run("Gaussian Blur...", "sigma=1");}
```

ensures that `Gaussian Blur...` is executed with a sigma value of 1.

`selectWindow` is another function, added to the macro whenever you click on a particular window to activate it, and which requires the name of the image window to make active. From this you can see that my example file name was `C1-hela-cells.tif`. Without this line, the duplicated image would be filtered twice – and the original not at all.

Finally, the `Image Calculator` command is special enough to get its own function in the macro language, `imageCalculator`. The first string it is given tells it both what sort of calculation to do, and that it should **create** a new image for the result – rather than replacing one of the existing images. The next two strings give it the titles of the images needed for the calculation.

Removing title dependencies

The fact that the original image title appears in the above macro is a problem: if you try to run it on another image, you are likely to find that it does not work

because `selectWindow` cannot find what it is looking for. So the next step is to remove this title dependency so that the macro can be applied to any (2D) image.

There are two ways to go about this. One is to insert a line that tells the macro the title of the image being processed at the start, e.g.

```
titleOrig = getTitle();
```

where `getTitle()` is an example of a function that asks for information. The result is then stored as a *variable*, so that any time we type `titleOrig` later this will be replaced by the string corresponding to the original title⁵. Then we just find anywhere the title appears and replace the text with our new variable name, i.e. in this case by writing

```
selectWindow(titleOrig);
```

If we do this, the window we want will *probably* be activated as required. However, it is possible that we have two images open at the same time with identical titles – in which case it is not clear which window should be selected, and so the results could be unpredictable. A safer approach is to get a reference to the *image ID* rather than its title. The ID is a number that should be unique for each image, which is useful for ImageJ internally but which we do not normally care about unless we are programming. Using IDs, the updated macro code then becomes:

```
idOrig = getImageID();
run("32-bit");
run("Duplicate...", "title=[My duplicated image]");
idDuplicate = getImageID();
run("Gaussian Blur...", "sigma=1");
selectImage(idOrig);
run("Gaussian Blur...", "sigma=2");
imageCalculator("Subtract create", idDuplicate, idOrig);
```

We had to change `selectWindow` to `selectImage` for the IDs to work. I also changed the title of the duplicated image to something more meaninglessly general – which required square brackets, because it includes spaces that would otherwise mess things up⁶. Also, because the duplicated image will be active immediately after it was created, I ask ImageJ for its ID at that point. This lets me then pass the two IDs (rather than titles) to the `imageCalculator` command when necessary.

⁵There is nothing special about `titleOrig` – this text can be changed to any variable name you like, so long as it is one word and does not contain special characters.

⁶In ImageJ's macro language, spaces in the string telling a command what to do are used to indicate that a separate piece of information is being given. So titles or file names that require spaces need to be put inside square brackets.

Adding comments

Whenever macros become more complicated, it can be hard to remember exactly what all the parts do and why. It is then a *very* good idea to add in some extra notes and explanations. This is done by prefixing a line with `//`, after which we can write whatever we like because the macro interpreter will ignore it. These extra notes are called *comments*, and I will add them from now on.

Customizing sigma values

By changing the size of the Gaussian filters, the macro can be tailored to detecting structures of different sizes. It would be relatively easy to find the `Gaussian Blur` lines and change the sigma values accordingly here, but adjusting settings like this in longer, more complex macros can be awkward. In such cases, it is helpful to extract the settings you might wish to change and include them at the start of the macro.

To do this here, insert the following lines at the very beginning:

```
// Store the Gaussian sigma values -
// sigma1 should be less than sigma2
sigma1 = 1.5;
sigma2 = 2;
```

Then, update the later commands to:

```
run("Gaussian Blur...", "sigma="+sigma1);
selectImage(idOrig);
run("Gaussian Blur...", "sigma="+sigma2);
```

This creates two new variables, which represent the sigma values to use. Now any time you want to change `sigma1` or `sigma2` you do not need to hunt through the macro for the correct lines: you can just update the lines at the top⁷.

Adding interactivity

Usually I would stop at this point. Still, you might wish to share your macro with someone lacking your macro modification skills, in which case it would be useful to give this person a dialog box into which they could type the Gaussian sigma values that they wanted. An easy way to do this is to remove the sigma value information from the `run` command lines, giving

```
run("Gaussian Blur...");
```

Since `Gaussian Blur` will not then know what size of filters to use, it will ask. The disadvantage of this is that the user is prompted to enter sigma values at two different times as the macro runs, which is slightly more annoying than necessary.

⁷Note that `+` is used to join multiple strings into one, converting numbers into strings as needed. Therefore in this case the lines `"sigma="+2` and `"sigma="+sigma2` would each give us the same result: one longer string with the extra part appended at the end, i.e. `"sigma=2"`.

The alternative is to create a dialog box that asks for all the required settings in one go. To do this, update the beginning of your macro to include something like the following:

```
Dialog.create("Choose filter sizes for DoG filtering");
Dialog.addNumber("Gaussian sigma 1", 1);
Dialog.addNumber("Gaussian sigma 2", 2);
Dialog.show();
sigma1 = Dialog.getNumber();
sigma2 = Dialog.getNumber();
```

The first line generates a dialog box with the title you specify. Each of the next two lines state that the required user input should be a number with the specified prompts and default values. The other lines simply show the dialog box and then read out whatever the user typed and puts it into variables. This is documented in ImageJ's list of built-in macro functions.

Suggested improvements

You should now have a macro that does something vaguely useful, and which will work on most 2D images. It could nevertheless still be enhanced in many ways. For example,

- You could close any unwanted images (e.g. the original and its duplicate) by selecting their IDs, and then inserting `close();` commands afterwards.
- You could make the macro work on entire image stacks. If you want it to process each plane separately, this involves only inserting the words `stack` and `duplicate` in several places – by recording a new macro in the same way, but using a stack as your example image, you can see where to do this. If you want the filtering to be applied in 3D, you can use the `Gaussian Blur 3D...` command instead of `Gaussian Blur...`
- You could create a log of which images you have processed, possibly including the settings used. The log is output by including a `log(text);` line, where `text` is some string you have created, e.g. `text = "Image name: "+ getTitle()`.
- More impressively, you could turn the macro into a full spot-detector by thresholding the DoG filtered image, and then running the `Analyze → Analyze Particles...` command. If you want to measure original spot intensities, you should remember to go to `Analyze → Set Measurements...` to make sure the measurements are redirected to the original image – which you should possibly have duplicated at the beginning, since otherwise it will have been Gaussian filtered by the time your macro reaches the measurement stage.

In any case, the process of developing a macro is usually the same:

1. Record a macro that does basically the right thing
2. Remove all the superfluous lines (contrast adjustment, errant clicking etc.)
3. Replace the image titles with image ID references
4. Add comments to describe what the macro is doing
5. Track down bugs and make improvements

Part III

Fluorescence images

From photons to pixels

Chapter outline

- *The pixel values in a fluorescence image depend upon numbers of detected photons*
- *Blur & noise are inevitable*

14.1 The big picture of fluorescence imaging

Images in fluorescence microscopy are formed by detecting light – and such small amounts of light that it can be thought of in terms of individual photons. The photons are emitted from fluorescent molecules within the sample being imaged. Sometimes these photon-emitting molecules may be the very things we are interested in studying, but often they have only been introduced to the sample because they have the helpful property of fluorescing when in the presence of the (otherwise non-fluorescent) molecules or structures we would *really* like to see.

Either way, the most that the image can tell us is how much light was emitted from any particular point. From this information we make our interpretations, such as about the presence or absence of some feature, about the size and shape of a structure, or about the relative concentration of a molecule. But in no case are we seeing the feature, structure or molecule directly in the recorded images: we only have measurements of numbers of photons we could detect.

We will not give much attention here to what any particular number of photons emanating from a sample really indicates from a biological point of view – this would depend too much upon the design and details of the experiment, i.e. on the cells, stains and other substances involved. We can, however, often make general assumptions, such as that if we were to see (on average) twice as many photons originating from one region as from another, the number of fluorescing molecules must be around twice as high in the first region¹. But before we can worry about such things, we first need to concentrate upon how accurately we can even determine the number and origins of photons being emitted from the sample, given the limited quality of the images we can actually record.

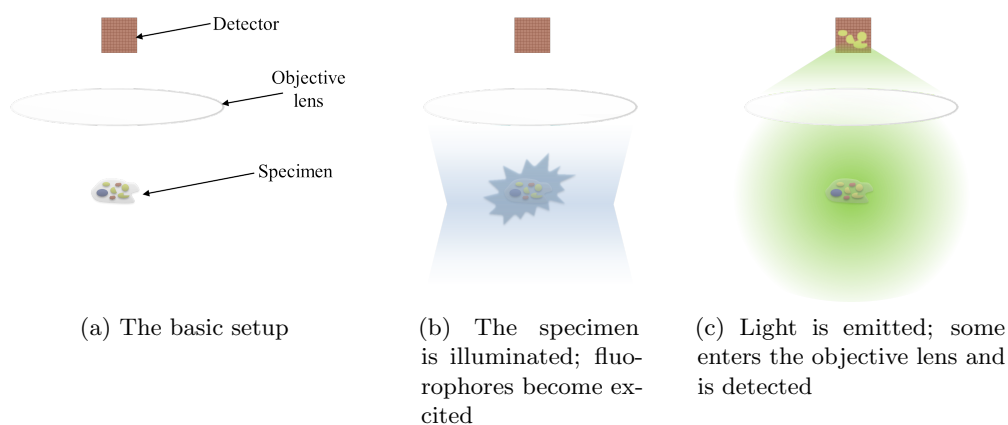


Figure 14.1: A (very) simplified diagram showing the steps of image formation in fluorescence microscopy.

14.1.1 Recording images

Once a specimen has been prepared and is waiting underneath the microscope, the basic process of recording a fluorescence image comprises four main steps (summarized in Figure 14.1):

1. *Fluorophore excitation.* The fluorescent molecules (fluorophores) first need to be raised into an excited state. This happens upon the absorption of a photon, the energy (i.e. wavelength) of which should fall into a range specific to the fluorophore itself. This is carried out by illuminating the specimen, e.g. with a lamp or laser.
2. *Photon emission.* When returning to its ground state, each fluorophore may emit a photon – this time with a lower energy (i.e. longer wavelength) than the photon previously absorbed.
3. *Photon detection.* Most emitted photons can be thought of, rather informally, as ‘shooting off in the wrong direction’, in which case we have no hope of detecting them. But a proportion of the light should enter the objective lens of the microscope and be focussed towards a detector. When a photon strikes the detector, it is registered as a ‘hit’ by the release of an electron (when we are lucky; detectors are imperfect so this might not occur, meaning the photon is effectively lost).
4. *Quantification & storage.* After fixed time intervals, the charges of the electrons produced by photons striking the detector are quantified, and from

¹This assumes a linear relationship, which does not always hold (e.g. if there is dye saturation, or high laser powers are used for illumination).

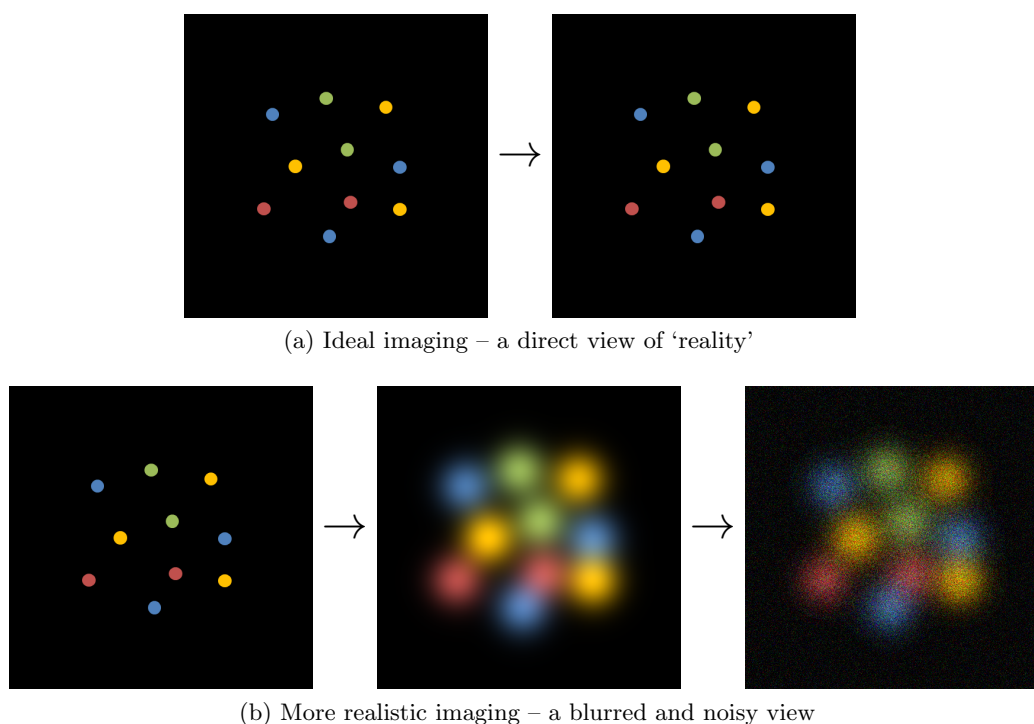


Figure 14.2: The difference between what we might *wish* to image, and what we actually *can* image. In both cases (a and b), the ‘real’ scene is shown on the left. Ideally, the small coloured spots in reality would directly map to coloured spots of a related size and separation in the image (a). However, the light emitted from these spots would actually end up producing larger objects in the image, which can then blur together (b, centre). Noise is further added to this blurriness to give us the best image we can really record (b, right).

these quantifications pixel values are determined. A larger charge indicates more photons, which translates into a higher pixel value.

14.1.2 Errors and imprecisions

From the above summary, it is clear that we are quite some distance away from knowing exactly how much light is emitted from the specimen: most photons do not reach the detector, and many that do are still not registered. But since we can expect to always lose track of a similar proportion of emitted light – perhaps 90% – this does not matter much: we can expect all parts of the image to be similarly affected, so relative differences in brightness would still be reflected in our pixel values. However, there are two more critical ways in which the images we can record are less good than the images we would like:

1. *Uncertainty in space.* Ideally, all light originating from a particular point in the specimen would strike the detector in exactly the same place, and

therefore end up contributing to exactly the same pixel. In practice, however, *the light beginning from one point cannot be focused back to a single point on the detector*. Consequently, it can end up being spread over several pixels. The end result is that the image is *blurred*.

2. *Uncertainty in brightness*. When an image is blurry we would also expect it to be smooth, but this is not usually what we get in fluorescence microscopy. Rather, there are seemingly random variations in brightness everywhere throughout the image: the *noise*. Some noise can come from imprecisions when determining the charge of small clouds of electrons quickly. But, more curiously, *the emission of the photons is itself random*, so that even if we detected every photon perfectly we would still get noisy images.

The twin issues of blur and noise do not affect all images equally. For example, blur can cause us to misjudge the size of something by several hundred nanometres, but if the thing we are measuring is much larger than this then the error may be trivial. Also, if we are detecting many thousands of photons then the uncertainty due to noise may be extremely small relative to the numbers involved. But very often we are interested in measuring tiny structures in images containing only tens of photons at their brightest points. In these cases, the effects of blur and noise cannot be ignored.

Blur & the PSF

Chapter outline

- *Measurements in fluorescence microscopy are affected by blur*
- *Blur acts as a convolution with the microscope's PSF*
- *The size of the PSF depends on the microscope type, light wavelength \mathcal{E} objective lens NA, \mathcal{E} is on the order of hundreds of nm*
- *In the focal plane, the PSF is an Airy pattern*
- *Spatial resolution is a measure of how close structures can be distinguished. It is better in xy than along the z dimension.*

15.1 Introduction

Microscopy images normally look blurry because light originating from one point in the sample is not all detected at a single pixel: usually it is detected over several pixels and z -slices. This is *not* simply because we cannot use perfect lenses; rather, it is caused by a fundamental limit imposed by the nature of light. The end result is as if the light that we detect is redistributed slightly throughout our data (Figure 15.1).

This is important for three reasons:

1. Blur affects the apparent *size* of structures
2. Blur affects the apparent *intensities* (i.e. brightnesses) of structures
3. Blur (sometimes) affects the apparent *number* of structures

Therefore, *almost every measurement* we might want to make can be affected by blurring to some degree.

That is the bad news about blur. The good news is that it is rather well understood, and we can take some comfort that it is not random. In fact, the main ideas have already been described in Chapter 10, because blurring in fluorescence microscopy is mathematically described by a *convolution* involving the microscope's *Point Spread Function* (PSF). In other words, the PSF acts like a linear filter applied to the perfect, sharp data we would like but can never directly acquire.



Figure 15.1: Schematic diagram showing the effects of blur. Think of the sand as photons, and the height of the sandcastle as the intensity values of pixels (a greater height indicates more photons, and thus a brighter pixel). The ideal data would be sharp and could contain fine details (a), but after blurring it is not only harder to discriminate details, but intensities in the brighter regions have been reduced and sizes increased (b). If we then wish to determine the size or height of one of the sandcastle's towers, for example, we need to remember that any results we get by measuring (b) will differ from those we would have got if we could have measured (a) itself. Note, however, that approximately the same *amount* of signal (sand or photons) is present in both cases – only its arrangement is different.

Previously, we saw how smoothing (e.g. mean or Gaussian) filters could helpfully reduce noise, but as the filter size increased we would lose more and more detail (Figure 10.14). At that time, we could choose the size and shapes of filters ourselves, changing them arbitrarily by modifying coefficients to get the noise-reduction vs. lost-detail balance we liked best. But the microscope's blurring differs in at least two important ways. Firstly, it is effectively applied to our data *before* noise gets involved, so offers no noise-reduction benefits. Secondly, because it occurs before we ever set our eyes on our images, the size and shape of the filter (i.e. the PSF) used are only indirectly (and in a very limited way) under our control. It would therefore be much nicer just to dispense with the blurring completely since it offers no real help, but unfortunately light conspires to make this not an option and we just need to cope with it.

The purpose of this chapter is to offer a practical introduction to why the blurring occurs, what a widefield microscope's PSF looks like, and why all this matters. Detailed optics and threatening integrals are not included, although several equations make appearances. Fortunately for the mathematically unenthusiastic, these are both short and useful.

15.2 Blur & convolution

As stated above, the fundamental cause of blur is that light originating from an infinitesimally small point cannot then be detected at a similar point, no matter

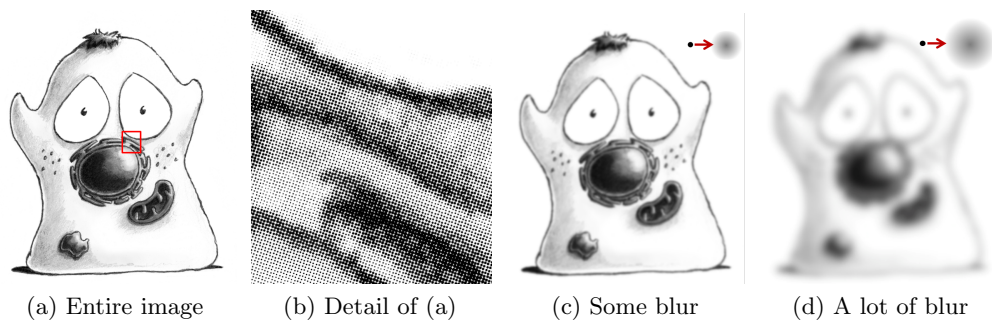


Figure 15.2: Images can be viewed as composed of small points (b), even if these points are not visible without high magnification (a). This gives us a useful way to understand what has happened in a blurred image: each point has simply been replaced by a more diffuse blob, the PSF. Images appear more or less blurred depending upon how large the blobby PSFs are (c) and (d).

how great our lenses are. Rather, it ends up being focused to some larger volume known as the PSF, which has a minimum size dependent upon both the light's wavelength and the lens being used (Section 15.4.1).

This becomes more practically relevant if we consider that *any* fluorescing sample can be viewed as composed of many similar, exceedingly small light-emitting points – you may think of the fluorophores. Our image would ideally then include individual points too, digitized into pixels with values proportional to the emitted light. But what we get instead is an image in which every point has been replaced by its PSF, scaled according to the point's brightness. Where these PSFs overlap, the detected light intensities are simply added together. Exactly how bad this looks depends upon the size of the PSF (Figure 15.2).

Section 10.2.5 gave one description of convolution as replacing each pixel in an image with a scaled filter – which is just the same process. Therefore it is no coincidence that applying a Gaussian filter to an image makes it look similarly blurry. Because every point is blurred in the same way (at least in the ideal case; extra aberrations can cause some variations), if we know the PSF we can characterize the blur throughout the entire image – and thereby make inferences about how blurring will impact upon anything we measure.

15.3 The shape of the PSF

We can gain an initial impression of a microscope's PSF by recording a z -stack of a small, fluorescent bead, which represents an ideal light-emitting point. Figure 15.3a shows that, for a widefield microscope, the bead appears like a bright blob when it is in focus. More curiously, when viewed from the side (xz or yz), it has a somewhat hourglass-like appearance – albeit with some extra patterns. This exact shape is well enough understood that PSFs can also be generated theoretically

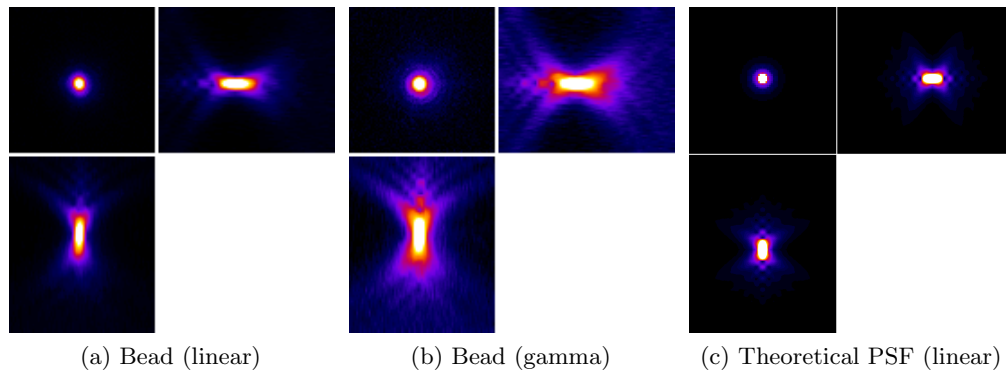


Figure 15.3: PSFs for a widefield microscope. (a) and (b) are from z -stacks acquired of a small fluorescent bead, displayed using linear contrast and after applying a gamma transform to make fainter details easier to discern (see Section 8.2.3). (c) shows a theoretical PSF for a similar microscope. It differs in appearance partly because the bead is not really an infinitesimally small point, and partly because the real microscope's objective lens is less than perfect. Nevertheless, the overall shapes are similar.

based upon the type of microscope and objective lenses used (c).

Generating PSFs

Plugins to create theoretical PSFs are available from

- <http://www.optinav.com/Diffraction-PSF-3D.htm>
- <http://bigwww.epfl.ch/algorithms/psfgenerator/>

In & out of focus

Figure 15.4 attempts to show that the hourglass aspect of the PSF is really perfectly intuitive. When recording a z -stack of a light-emitting point, we would *prefer* that the light ended up at a single pixel in a single slice. But the light itself is oblivious to our wishes, and will cheerfully be detected if it happens to strike a detector, no matter where that is. Therefore we should expect the light to be detected across a small region only if the image is in-focus; otherwise it will be more spread out to an extent that depends upon how far from the focal point it is detected. From the side (xz or yz), this leads to an hourglass shape.

Question 15.1

In focus, a light-emitting point looks like a small, bright blob. Out of focus, it is much less bright and extends over a wider area. However, how would you expect the *total amount* of light to differ in a widefield image depending upon

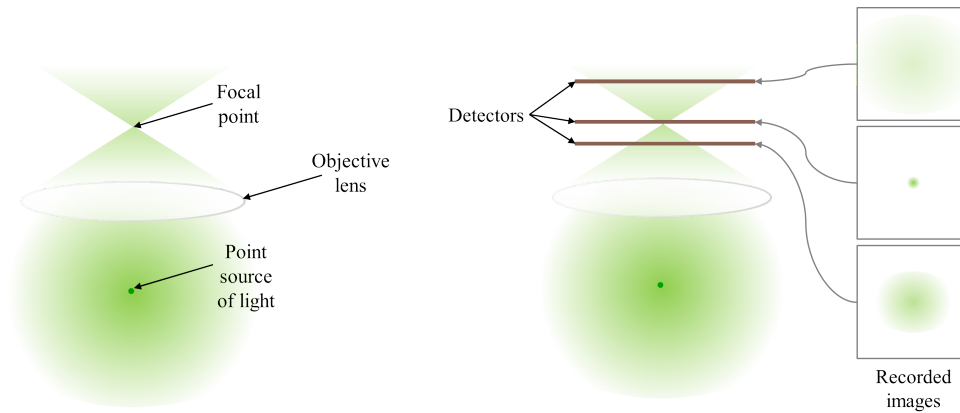


Figure 15.4: Simplified diagram to help visualize how a light-emitting point would be imaged using a widefield microscope. Some of the light originating from the point is captured by a lens. If you imagine the light then being directed towards a focal point, this leads to an hourglass shape. If a detector is placed close to the focal point, the spot-like image formed by the light striking the detector would be small and bright. However, if the detector were positioned above or below this focal plane, the intensity of the spot would decrease and its size would increase.

whether a plane is in-focus or not? In other words, would you expect more or less light in the focal plane than in other planes above or below it? *Solution*

The appearance of interference

Figure 15.4 is pretty limited in what it shows: it does not begin to explain the extra patterns of the PSF, which appear on each 2D plane as concentric rings (Figure 15.5), nor why the PSF does not shrink to a single point in the focal plane. These factors relate to the interference of light waves. While it is important to know that the rings occur – if only to avoid ever misinterpreting them as extra ring-like structures being really present in a sample – they have limited influence upon any analysis because the central region of the PSF is overwhelmingly brighter. Therefore for our purposes they can mostly be disregarded.

The Airy disk

Finally, the PSF in the focal plane is important enough to deserve some attention, since we tend to want to measure things where they are most in-focus. This entire xy plane, including its interfering ripples, is called an *Airy pattern*, while the bright central part alone is the *Airy disk* (Figure 15.6). In the best possible case, when all the light in a 2D image comes from in-focus structures, it would already have been blurred by a filter that looks like this.

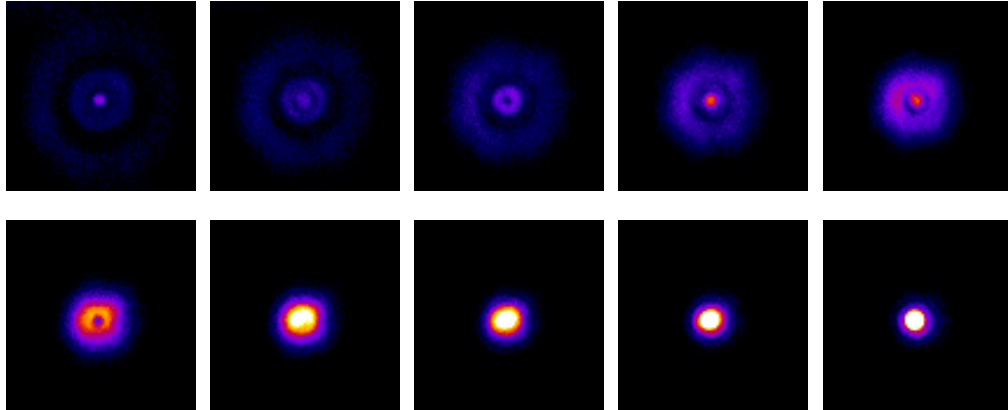
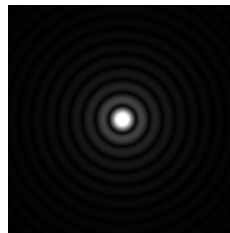


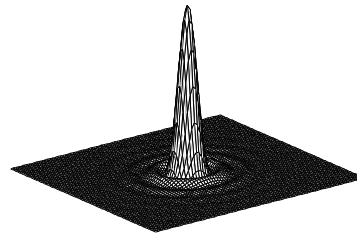
Figure 15.5: Ten slices from a z -stack acquired of a fluorescent bead, starting from above and moving down to the focal plane. The same linear contrast settings have been applied to each slice for easy comparison, although this causes the in-focus bead to *appear* saturated since otherwise the rings would not be visible at all. Because the image is (approximately) symmetrical along the z -axis, additional slices moving below the focal plane would appear similar.



(a) George Biddell Airy
(1801–1892)



(b) Airy pattern



(c) Surface plot of Airy pattern

Figure 15.6: George Biddell Airy and the Airy pattern. (a) During his schooldays, Airy had been renowned for being skilled ‘*in the construction of peashooters and other such devices*’ (see <http://www-history.mcs.st-and.ac.uk/Biographies/Airy.html>). The rings surrounding the Airy disk have been likened to the ripples on a pond. Although the rings phenomenon was already known, Airy wrote the first theoretical treatment of it in 1835 (http://en.wikipedia.org/wiki/Airy_disk). (b) An Airy pattern, viewed as an image in which the contrast has been set to enhance the appearance of the outer rings surrounding the Airy disk. (c) A surface plot of an Airy pattern, which shows that the brightness is much higher within the central region when compared to the rings.

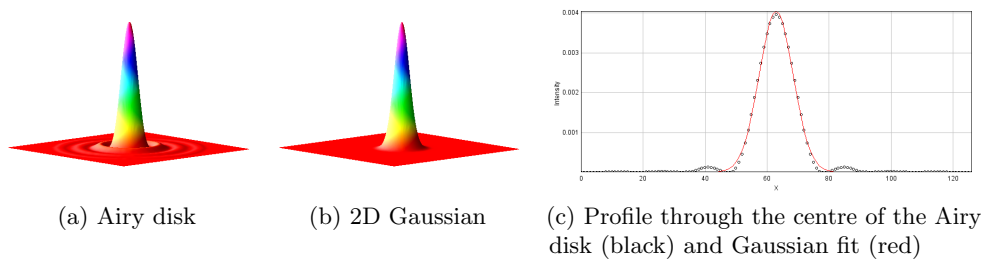


Figure 15.7: Comparison of an Airy disk (taken from a theoretical PSF) and a Gaussian of a similar size, using two psychedelic surface plots and a 1D cross-section. The Gaussian is a very close match to the Airy disk.

The Airy disk should look familiar. If we ignore the little interfering ripples around its edges, it can be very well approximated by a Gaussian function (Figure 15.7). Therefore *the blur of a microscope in 2D is similar to applying a Gaussian filter*, at least in the focal plane.

15.4 The size of the PSF

So much for appearances. To judge how the blurring will affect what we can see and measure, we need to know the *size* of the PSF – where smaller would be preferable.

The size requires some defining: the PSF actually continues indefinitely, but has extremely low values when far from its centre. One approach for characterizing the Airy disk size is to consider its radius r_{airy} as the distance from the centre to the first *minimum*: the lowest point before the first of the outer ripples begins. This is given by:

$$r_{airy} = \frac{0.61\lambda}{NA} \quad (15.1)$$

where λ is the light wavelength and NA is the numerical aperture of the objective lens¹.

Question 15.2

According to Equation 15.1, what are the two variables we *may* be able to control that influence the amount of blur in our images, and how must they be changed (increased or decreased) for the images to have less blur? *Solution*

¹Note that this is the *limit* of the Airy disk size, and assumes that the system is free of any aberrations. In other words, this is the best that we can hope for: the Airy disk cannot be made smaller simply by better focusing, although it could easily be made worse by a less-than-perfect objective lens.

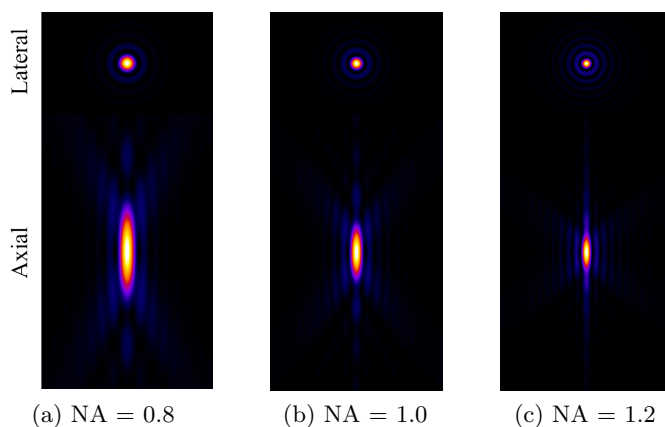


Figure 15.8: Examples of theoretical PSFs generated with different Numerical Apertures.

A comparable measurement to r_{airy} between the centre and first minimum along the z axis is:

$$z_{min} = \frac{2\lambda \times \eta}{NA^2} \quad (15.2)$$

where η is the refractive index of the objective lens immersion medium (which is a value related to the speed of light through that medium).

Question 15.3

Does the NA have more influence on blur in the xy plane, or along the z axis?

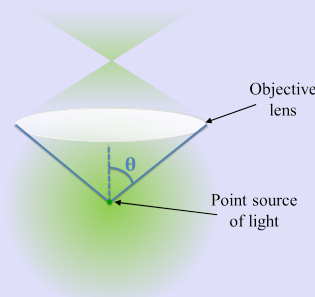
Solution

Numerical Aperture

The equations for the PSF size show that if you can use an objective lens with a higher NA, you can potentially reduce blur in an image – especially along the z axis (Figure 15.8). Unfortunately, one soon reaches another limit in terms of what increasing the NA can achieve. This can be seen from the equation used to define it:

$$NA = \eta \sin \theta \quad (15.3)$$

where η is again the refractive index of the immersion medium and θ is the half-angle of the cone of light accepted by the objective (*above*). Because $\sin \theta$ can never exceed 1, the NA can never exceed η , which itself has fixed values (e.g. around 1.0 for air, 1.34 for water, or 1.5 for oil). High NA lenses can therefore reduce blur only to a limited degree.



An important additional consideration is that the highest NAs are possible when the immersion refractive index is high, but if this does not match the refractive index of the medium surrounding the sample we get *spherical aberration*. This is a phenomenon whereby the PSF becomes asymmetrical at increasing depth and the blur becomes weirder. Therefore, matching the refractive indices of the immersion and embedding media is often *strongly* preferable to using the highest NA objective available: it is usually better to have a larger PSF than a highly irregular one.

For an interactive tutorial on the effect of using different NAs, see <http://www.microscopyu.com/tutorials/java/imageformation/airyna/index.html>

Question 15.4

Convince yourself that z_{min} will be considerably higher than r_{airy} using one of the following methods:

- Put an example refractive index (e.g. $\eta = 1.34$ for water), and some reasonable values of λ and the NA into Equations 15.1 and 15.2, and compare the results
- Calculate the ratio z_{min}/r_{airy} and substitute in the equation for the NA. This should reveal that the ratio is bigger than 1, i.e. that z_{min} is larger.

What is the main implication of this observation, in terms of how separated structures need to be along different dimensions for them still to be distinguishable?

Solution

15.4.1 Spatial resolution

Spatial resolution is concerned with how close two structures can be while they are still distinguishable. This is a somewhat subjective and fuzzy idea, but one way to define it is by the *Rayleigh Criterion*, according to which two equally bright spots are said to be resolved (i.e. distinguishable) if they are separated by the distances calculated in Equations 15.1 and 15.2. If the spots are closer than this, they are likely to be seen as one. In the in-focus plane, this is illustrated in Figure 15.9.

It should be kept in mind that the use of r_{airy} and z_{min} in the Rayleigh criterion is somewhat arbitrary – and the effects of brightness differences, finite pixel sizes and noise further complicate the situation, so that in practice a greater distance may well be required for us to confidently distinguish structures. Nevertheless, the Rayleigh criterion is helpful to give some idea of the scale of distances involved, i.e. hundreds of nanometres when using visible light.

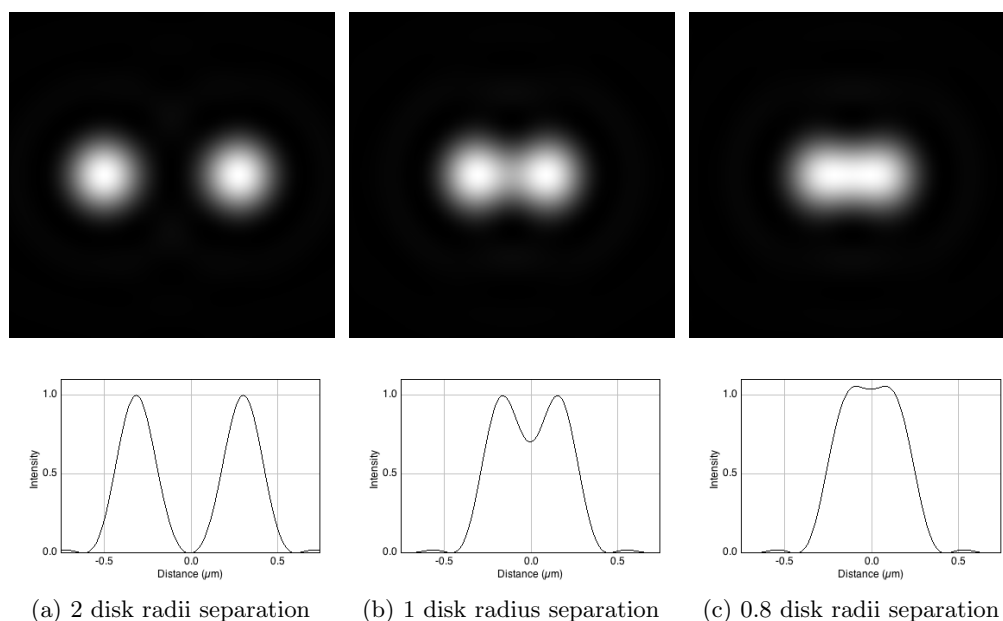


Figure 15.9: Airy patterns separated by different distances, defined in terms of Airy disk radii. The top row contains the patterns themselves, while the bottom row shows fluorescence intensity profiles computed across the centres of the patterns. Two distinct spots are clearly visible whenever separated by at least one disk radius, and there is a dip apparent in the profile. However, if the separation is less than one radius, the contrast rapidly decreases until only one structure is apparent.

Question 15.5

Suppose the diameter of the Airy disk is around 500 nm, and you are looking at an image containing separate, well-spaced structures that are 2 nm, 20 nm and 200 nm in size. Assuming that you have imaged all of these exactly in focus (after all, you are a brilliant microscopist), how will these structures appear in the image?

Note: This is a particularly important question! Think of both the size and brightness.

Solution

15.4.2 Measuring PSFs & small structures

Knowing that the Airy disk resembles a Gaussian function is extremely useful, because any time we see an Airy disk we can fit a 2D Gaussian to it. The parameters of the function will then tell us the Gaussian's centre exactly, which corresponds to where the fluorescing structure really is – admittedly not with complete accuracy, but potentially still beyond the accuracy of even the pixel size (noise is the real limitation). This idea is fundamental to single-molecule

localization techniques, including those in super-resolution microscopes like STORM and PALM, but requires that PSFs are sufficiently well-spaced that they do not interfere with one another and thereby ruin the fitting.

In ImageJ, we can somewhat approximate this localization by drawing a line profile across the peak of a PSF and then running **Analyze** → **Tools** → **Curve Fitting**.... There we can fit a 1D Gaussian function, for which the equation used is

$$y = a + (b - a) \exp[-(x - c)^2 / 2d^2] \quad (15.4)$$

a is simply a background constant, b tells you the peak amplitude (i.e. the maximum value of the Gaussian with the background subtracted), and c gives the location of the peak along the profile line. But potentially the most useful parameter here is d , which corresponds to the σ value of a Gaussian filter. So if you know this value for a PSF, you can approximate the same amount of blurring with a Gaussian filter. This may come in useful in Chapter 18.

Solutions

Question 15.1 In a widefield image, every plane we can record contains in-focus light along with *all* the detectable light from *all* other planes added together. Therefore we should expect approximately *the same total amount of light* within each plane of a z -stack – just differently distributed. That is potentially a lot of light in the ‘wrong’ place, especially if looking at a thick sample.

At least, this would be so for an infinitely-large detector, or a small, centred sample. In practice, if the light originates from a location so out of focus that its light spills over the side of the detector then this plane would contain less light.

Question 15.2 If the wavelength λ is *lower* or the objective NA is *higher*, r_{airy} decreases and we have less blur.

Question 15.3 Because of the squaring, the NA has a much greater influence on blur along the z axis than in xy .

Question 15.4 The ratio is

$$\frac{z_{min}}{r_{airy}} = \frac{2\lambda \times \eta}{NA^2} \times \frac{NA}{0.61\lambda} = \frac{3.28\eta}{NA} = \frac{3.28}{\sin \theta} \quad (15.5)$$

Therefore, even as $\sin \theta$ becomes close to 1 (i.e. a very high NA objective is used), the value of z_{min} remains over 3 times larger than r_{airy} – the z resolution is much worse. When the NA is lower, the difference is even more.

The main practical implication is that it is more likely you will be able to distinguish structures that are separated from one another by a short distance in xy than similarly separated in z . If you really need information along the z -dimension more than anywhere else, maybe rotating your sample could help?

Question 15.5 Because even an infinitesimally small point cannot appear smaller than the Airy disk in the recorded image, *potentially all 3 of these structures look the same!* There may be *some* increase in size visible with the 200 nm structure (because it is larger than a single point, this makes it like many different, slightly-shifted-but-mostly-overlapping Airy disks added together), but it will certainly not appear 10 or 100 times larger than the others.

However, because smaller objects typically emit fewer photons, the smaller structures may well appear less bright – if they are bright enough to be visible at all. Therefore, at this scale accurate measurements of size are impossible from (conventional, non-super-resolution) fluorescence microscopy images, but the actual size may have some relationship with brightness.

Noise

Chapter outline

- *There are two main types of noise in fluorescence microscopy: photon noise & read noise*
- *Photon noise is signal-dependent, varying throughout an image*
- *Read noise is signal-independent, & depends upon the detector*
- *Detecting more photons reduces the impact of both noise types*

16.1 Introduction

We could reasonably expect that a noise-free microscopy image should look pleasantly smooth, not least because the convolution with the PSF has a blurring effect that softens any sharp transitions. Yet in practice raw fluorescence microscopy images are not smooth. They are always, to a greater or lesser extent, corrupted by noise. This appears as a random ‘graininess’ throughout the image, which is often so strong as to obscure details.

This chapter considers the nature of the noisiness, where it comes from and what can be done about it. Before starting, it may be helpful to know the one major lesson of this chapter for the working microscopist is simply:

If you want to reduce noise, you need to detect more photons

This general guidance applies in the overwhelming majority of cases when a good quality microscope is functioning properly. Nevertheless, it may be helpful to know a bit more detail about why – and what you might do if detecting more photons is not feasible.

16.1.1 Background

In general, we can assume that noise in fluorescence microscopy images has the following three characteristics, illustrated in Figure 16.1:

1. *Noise is random* – For any pixel, the noise is a random positive or negative number added to the ‘true value’ the pixel should have.

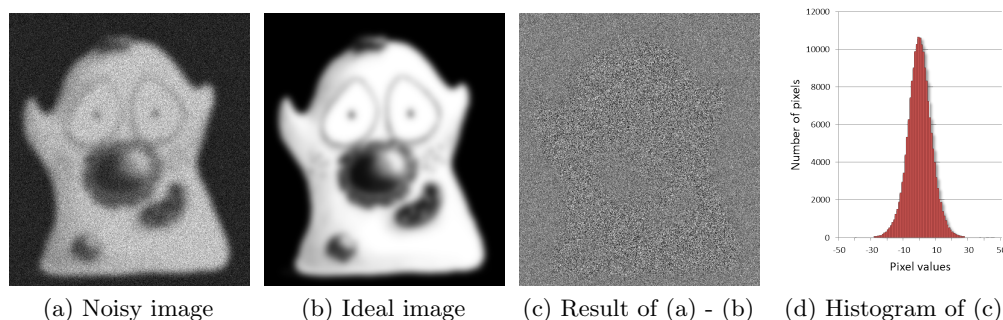


Figure 16.1: Illustration of the difference between a noisy image that we can record (a), and the noise-free (but still blurred) image we would prefer (b). The ‘noise’ itself is what would be left over if we subtracted one from the other (c). The histogram in (d) resembles a normal (i.e. Gaussian) distribution and shows that the noise consists of positive and negative values, with a mean of 0.

2. *Noise is independent at each pixel* – The value of the noise at any pixel does not depend upon where the pixel is, or what the noise is at any other pixel.
3. *Noise follows a particular distribution* – Each noise value can be seen as a *random variable* drawn from a particular distribution. If we have enough noise values, their histogram would resemble a plot of the distribution¹.

There are many different possible noise distributions, but we only need to consider the *Poisson* and *Gaussian* cases. No matter which of these we have, the most interesting distribution parameter for us is the *standard deviation*. Assuming everything else stays the same, if the standard deviation of the noise is higher then the image looks worse (Figure 16.2).

The reason we will consider two distributions is that there are two main types of noise for us to worry about:

1. *Photon noise*, from the emission (and detection) of the light itself. This follows a Poisson distribution, for which *the standard deviation changes with the local image brightness*.
2. *Read noise*, arising from inaccuracies in quantifying numbers of detected photons. This follows a Gaussian distribution, for which *the standard deviation stays the same throughout the image*.

Therefore the noise in the image is really the result of adding two² separate random components together. In other words, to get the value of any pixel P

¹Specifically its probability density or mass function – which for a Gaussian distribution is the familiar bell curve.

²Actually more. But the two mentioned here are usually by far the most significant, and it does not matter to our model at all if they contain various other sub-components. The important fact remains that there is some noise that varies throughout the image, and some that does not.

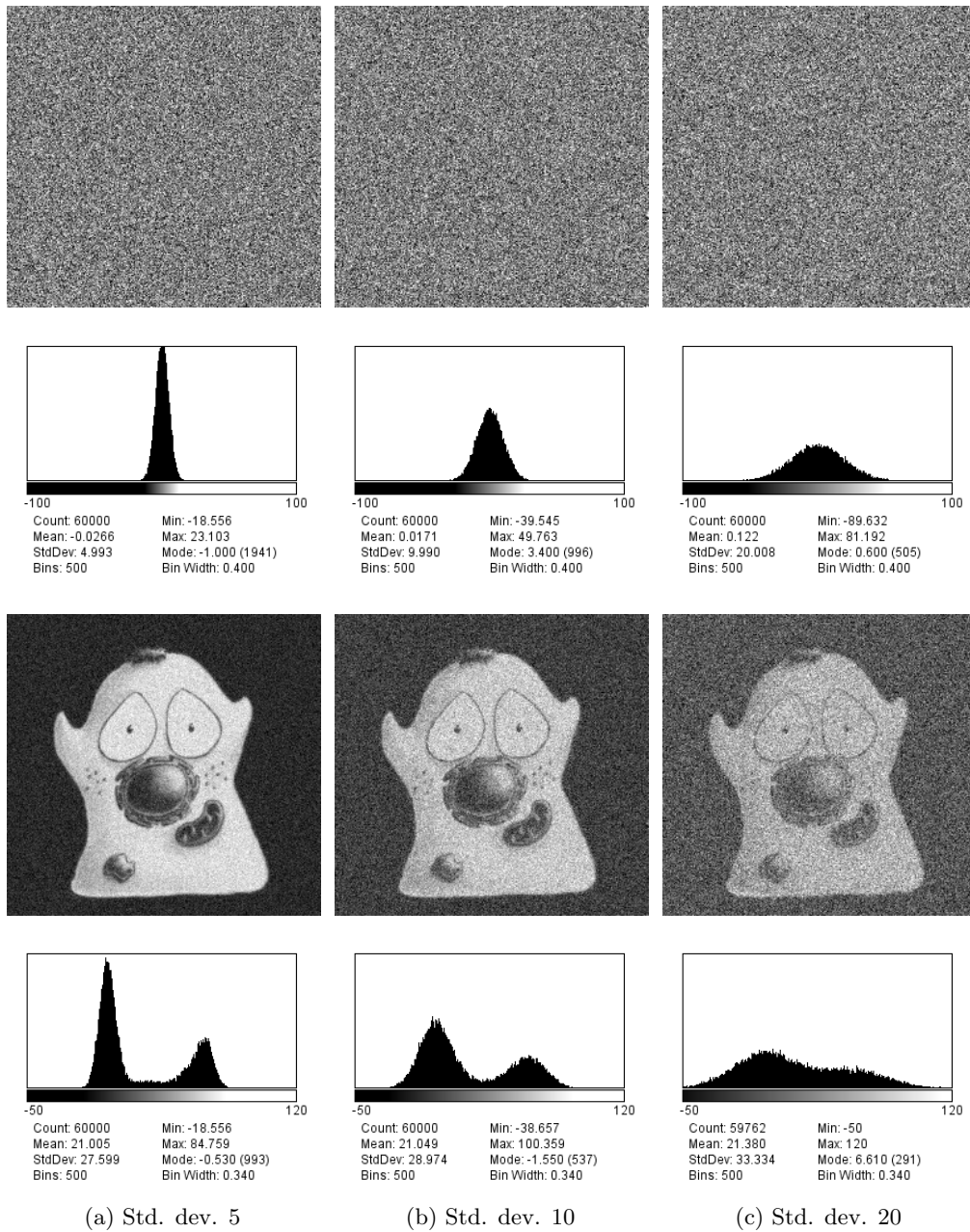


Figure 16.2: Gaussian noise with different standard deviations. (*Top*) Noise values only, shown as images (with contrast adjusted) and histograms. (*Bottom*) Noise values added to an otherwise noise-free image, along with the resulting histograms. Noise with a higher standard deviation has a worse effect when added to an image.

you need to calculate the sum of the ‘true’ (noise-free) value T , a random photon noise value N_p , and a random read noise value N_r , i.e.

$$P = T + N_p + N_r \quad (16.1)$$

Finally, some useful maths: suppose we add two random noisy values together. Both are independent and drawn from distributions (Gaussian or Poisson) with standard deviations σ_1 and σ_2 . The result is a third random value, drawn from a distribution with a standard deviation $\sqrt{\sigma_1 + \sigma_2}$. On the other hand, if we multiply a noisy value from a distribution with a standard deviation σ_1 by k , the result is noise from a distribution with a standard deviation $k\sigma_1$.

These are all my most important noise facts, upon which the rest of this chapter is built. We will begin with Gaussian noise because it is easier to work with, found in many applications, and widely studied in the image processing literature. However, in *most* fluorescence images photon noise is the more important factor.

16.2 Gaussian noise

Gaussian noise is a common problem in fluorescence images acquired using a CCD camera (see Chapter 17). It arises at the stage of quantifying the number of photons detected for each pixel. Quantifying photons is hard to do with complete precision, and the result is likely to be wrong by few photons. The error is the read noise.

Read noise typically follows a Gaussian distribution and has a mean of zero: this implies there is an equal likelihood of over or underestimating the number of photons. Furthermore, according to the properties of Gaussian distributions, we should expect around 68% of measurements to be ± 1 standard deviation from the true, read-noise-free value. If a detector has a low read noise standard deviation this is then a good thing.

16.2.1 Signal-to-Noise Ratio (SNR)

Read noise is said to be *signal independent*: its standard deviation is constant, and does not depend upon how many photons are being quantified. However, the extent to which read noise is a problem probably *does* depend upon the number of photons. For example, if we have detected 20 photons, a noise standard deviation of 10 photons is huge; if we have detected 10 000 photons, it is likely not so important.

A better way to assess the noisiness of an image is then the ratio of the interesting part of each pixel (called the *signal*, which is here what we would

ideally detect in terms of photons) to the noise standard deviation, which together is known as the *Signal-to-Noise Ratio*³:

$$\text{SNR} = \frac{\text{Signal}}{\text{Noise standard deviation}} \quad (16.2)$$

Question 16.1

Calculate the SNR in the following cases:

- We detect 10 photons, read noise standard deviation 1 photon
- We detect 100 photons, read noise standard deviation 10 photons
- We detect 1000 photons, read noise standard deviation 10 photons

For the purposes of this question, you should assume that read noise is the only noise present (ignore photon noise).

Solution

16.2.2 Gaussian noise simulations

I find the best way to learn about noise is by creating simulation images, and exploring their properties through making and testing predictions. **Process** → **Noise** → **Add Specified Noise...** will add Gaussian noise with a standard deviation of your choosing to any image. If you apply this to an empty 32-bit image created using **File** → **New** → **Image...** you can see noise on its own.

Practical 16.1

Create an image containing only simulated Gaussian noise with a standard deviation of approximately 3. Confirm using its histogram or the **Measure** command that this standard deviation is correct.

Now add to this *more* Gaussian noise with a standard deviation of 4. Keeping in mind the end of Section 16.1.1, what do you expect the standard deviation of the result to be?

Solution

Question 16.2

`gauss_noise_1.tif` and `gauss_noise_2.tif` are two images containing Gaus-

³This is one definition of SNR. Many other definitions appear in the literature, leading to different values. The fact that any interesting image will vary in brightness in different places, the SNR is not necessarily the same at all pixels – therefore computing it in practice involves coming up with some summary measurement for the whole image. This can be approached differently, but the general principle is always to compare how much noise we have relative to interesting things: higher is better.

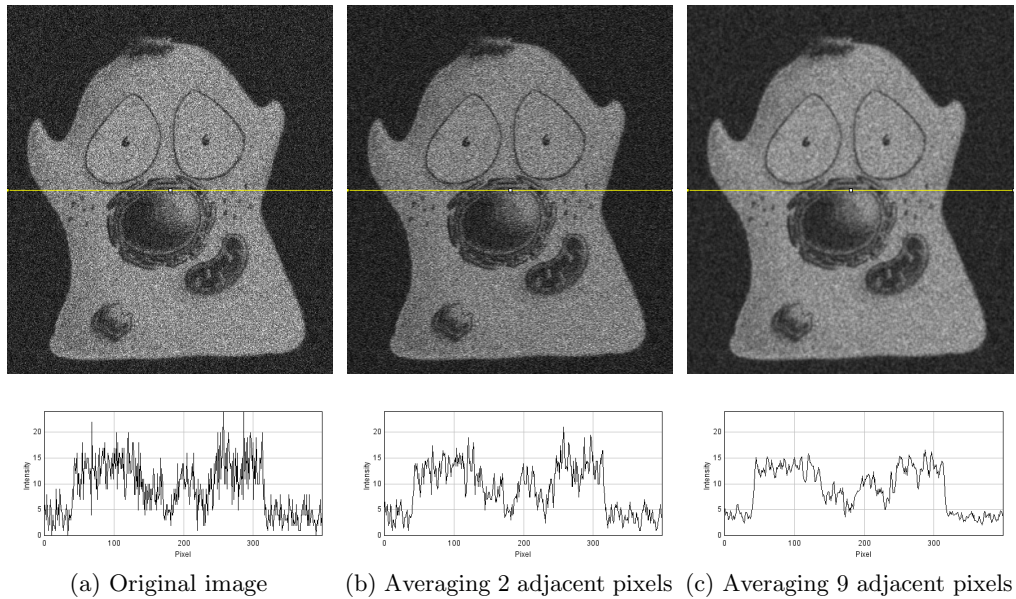


Figure 16.3: Noise reduction by averaging adjacent pixels.

sian noise with standard deviations of 5. Estimate (and optionally test with `Image Calculator...` and `Measure`), what the standard deviations will be:

1. When you add both images together
2. When you subtract `gauss_noise_2.tif` from `gauss_noise_1.tif`
3. When you subtract `gauss_noise_1.tif` from `gauss_noise_2.tif`
4. When you average both images together
5. When you add an image to itself

Solution

16.2.3 Averaging noise

Section 16.1.1 stated how to calculate the new standard deviation if noisy pixels are added together (i.e. it is the square root of the sum of the original variances). If the original standard deviations are the same, the result is always something *higher*. But if the pixels are *averaged*, then the resulting noise standard deviation is *lower*. This implies that if we were to average two independent noisy images of the same scene with similar SNRs, we would get a result that contains *less* noise, i.e. a higher SNR. This is the idea underlying our use of linear filters to reduce noise in Chapter 10, except that rather than using two images we computed our

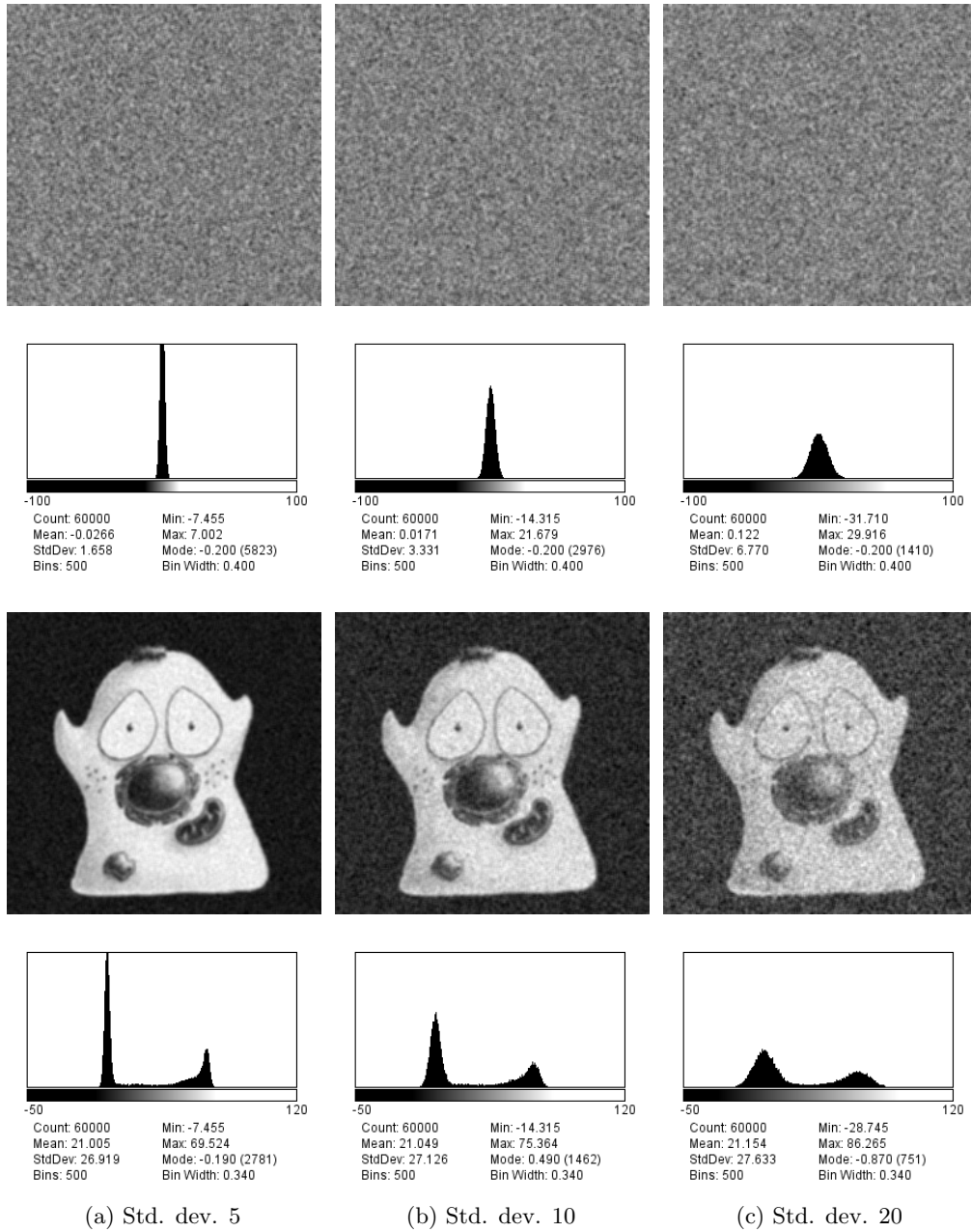


Figure 16.4: Images and histograms from Figure 16.2 after replacing each pixel with the mean of it and its immediate neighbours (a 3×3 mean filter). The standard deviation of the noise has decreased in all cases. In the noisiest example (c) the final image may not look brilliant, but the peaks in its histogram are clearly more separated when compared to Figure 16.2c, suggesting it could be thresholded more effectively (see Figure 9.4). It is not always the most aesthetically pleasing image that is the best for analysis.

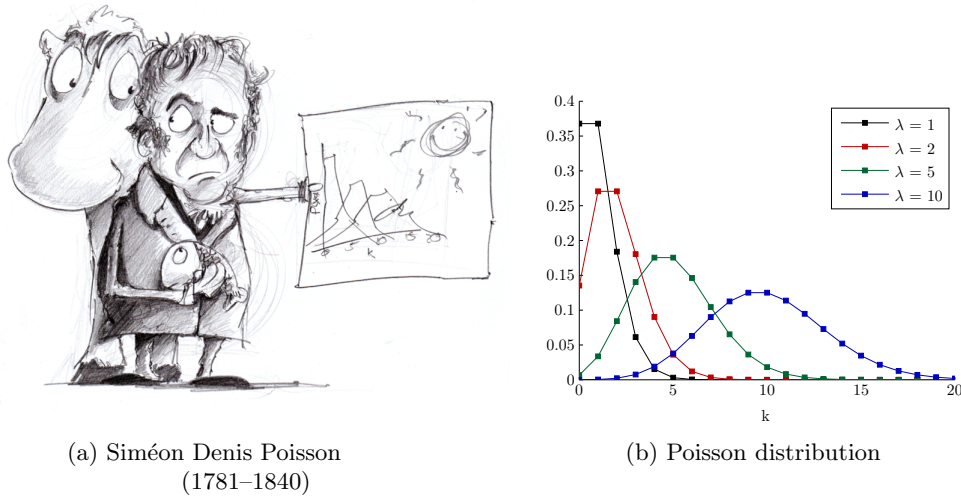


Figure 16.5: Siméon Denis Poisson and his distribution. (a) Poisson is said to have been extremely clumsy and uncoordinated with his hands. This contributed to him giving up an apprenticeship as a surgeon and entering mathematics, where the problem was less debilitating – although apparently this meant his diagrams tended not to very well drawn (see <http://www-history.mcs.st-andrews.ac.uk/history/Biographies/Poisson.html>). (b) The ‘Probability Mass Function’ of the Poisson distribution for several different values of λ . This allows one to see for any ‘true signal’ λ the probability of actually counting any actual value k . Although it is more likely that one will count exactly $k = \lambda$ than any other possible k , as λ increases the probability of getting precisely this value becomes smaller and smaller.

averages by taking other pixels from within the same image (Figures 16.3 and 16.4).

Practical 16.2

Create another image containing simulated Gaussian noise with a standard deviation of 30. What do you expect the standard deviation to be after applying a 3×3 mean filter (e.g. `Process` \rightarrow `Smooth`)? The calculation you need is much the same as in the last practical, but with some extra scaling involved.

Now apply the filter to the same image a second time. Is the noise reduced by a similar amount? How do you explain your answer? *Solution*

16.3 Poisson noise

In 1898, Ladislaus Bortkiewicz published a book entitled *The Law of Small Numbers*. Among other things, it included a now-famous analysis of the number of soldiers in different corps of the Prussian cavalry who were killed by being kicked by a horse, measured over a 20-year period. Specifically, he showed that these

numbers follows a *Poisson distribution*. This distribution, introduced by Siméon Denis Poisson in 1838, gives the probability of an event happening a certain number of times, given that we know (1) the average rate at which it occurs, and (2) that all of its occurrences are independent. However, the usefulness of the Poisson distribution extends far beyond gruesome military analysis to many, quite different applications – including the probability of photon emission, which is itself inherently random.

Suppose that, on average, a single photon will be emitted from some part of a fluorescing sample within a particular time interval. The randomness entails that we cannot say for sure what will happen on any one occasion when we look; sometimes one photon will be emitted, sometimes none, sometimes two, occasionally even more. What we are really interested in, therefore, is not precisely *how many* photons are emitted, which varies, but rather the *rate* at which they would be emitted under fixed conditions, which is a constant. The difference between the number of photons actually emitted and the true rate of emission is the *photon noise*. The trouble is that keeping the conditions fixed might not be possible: leaving us with the problem of trying to figure out rates from single, noisy measurements.

16.3.1 Signal-dependent noise

Clearly, since it is a rate that we want, we could get that with more accuracy if we averaged many observations – just like with Gaussian noise, averaging reduces photon noise, so we can expect smoothing filters to work similarly for both noise types.

The primary distinction between the noise types, however, is that Poisson noise is *signal-dependent*, and *does* change according to the number of emitted (or detected) photons. Fortunately, the relationship is simple: if the rate of photon emission is λ , the noise variance is also λ , and the noise standard deviation is $\sqrt{\lambda}$.

This is not really as unexpected as it might first seem (see Figure 16.6). It can even be observed from a very close inspection of Figure 16.7, in which the increased variability in the yeast cells causes their ghostly appearance even in an image that ought to consist only of noise.

Confusing λ
 λ often represents the mean of a Poisson distribution – it has nothing to do with wavelengths here

Question 16.3

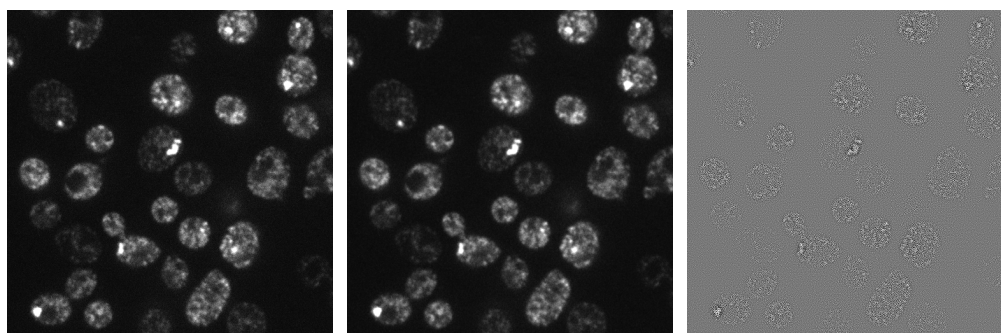
The equation for the probability mass function of the Poisson distribution is:

$$\mathcal{P}(k) \sim \frac{e^{-\lambda} \lambda^k}{k!} \quad (16.3)$$

where

- λ is the mean rate of occurrence of the event (i.e. the noise-free photon emission rate we want)

Figure 16.6: ‘The standard deviation of photon noise is equal to the square root of the expected value’. To understand this better, it may help to imagine a fisherman, fishing many times at the same location and under the same conditions. If he catches 10 fish on average, it would be quite reasonable to catch 7 or 13 on any one day – while 20 would be exceptional. If, however, he caught 100 on average, then it would be unexceptional if he caught 90 or 110 on a particular day, although catching only 10 would be strange (and presumably disappointing). Intuitively, the range of values that would be considered likely is related to the expected value. If nothing else, this imperfect analogy may at least help remember the name of the distribution that photon noise follows.



(a) Original image

(b) Gaussian filtered image

(c) Result of (a) - (b)

Figure 16.7: A demonstration that Poisson noise changes throughout an image. (a) Part of a spinning disk microscopy image of yeast cells. (b) A Gaussian filtered version of (a). Gaussian filtering reduces the noise in an image by replacing each pixel with a weighted average of neighbouring pixels (Section 10.4). (c) The difference between the original and filtered image contains the noise that the filtering removed. However, the locations of the yeast cells are still visible in this ‘noise image’ as regions of increased variability. This is partly an effect of Poisson noise having made the noise standard deviation larger in the brighter parts of the acquired image.

- k is an actual number of occurrences for which we want to compute the probability
- $k!$ is the *factorial* of k (i.e. $k \times (k - 1) \times (k - 2) \times \dots \times 1$)

So if you know that the rate of photon emission is 0.5, for example, you can put $\lambda = 0.5$ into the equation and determine the probability of getting any particular (integer) value of k photons. Applying this, the probability of not detecting any photons ($k = 0$) is 0.6065, while the probability of detecting a single photon ($k = 1$) is 0.3033.

Assuming the mean rate of photon emission is 1, use Equation 16.3 to calculate the probability of actually detecting 5 (which, at 5 times the true rate, would be an extremely inaccurate result). How common do you suppose it is to find pixels that are so noisy in the background region of a dark image? *Solution*

16.3.2 The SNR for Poisson noise

If the standard deviation of noise was the only thing that mattered, this would suggest that we are better not detecting much light: then photon noise is lower. But the SNR is a much more reliable guide. For noise that follows a Poisson distribution this is particularly easy to calculate. Substituting into Equation 16.2

$$\text{SNR}_{\text{Pois}} = \frac{\lambda}{\sqrt{\lambda}} = \sqrt{\lambda} \quad (16.4)$$

Therefore *the SNR of photon noise is equal to the square root of the signal!* This means that as the average number of emitted (and thus detected) photons increases, so too does the SNR. More photons \rightarrow a better SNR, directly leading to the assertion

*If you want to reduce photon noise,
you need to detect more photons*

Why relativity matters: a simple example

The SNR increases with the number of photons, even though the noise standard deviation increases too, because it is really *relative* differences in the brightness in parts of the image that we are interested in. Absolute numbers usually are of very little importance – which is fortunate, since not all photons are detected.

Yet if you remain unconvinced that the noise variability can get bigger while the situation gets better, the following specific example might help. Suppose the true signal for a pixel is 4 photons. Assuming the actual measured value is within one noise standard deviation of the proper result (which it will be,

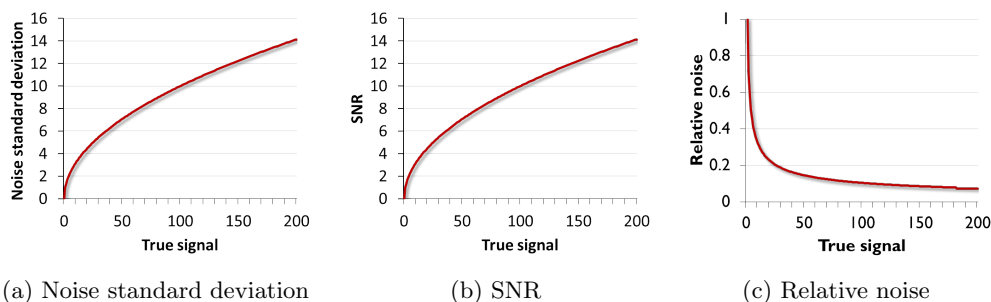


Figure 16.8: For Poisson noise, the standard deviation increases with the square root of the signal. So does the SNR, with the result that plots (a) and (b) look identical. This improvement in SNR despite the growing noise occurs because the signal is increasing faster than the noise, and so the noise is *relatively* smaller. Plotting the relative noise ($1/\text{SNR}$) shows this effect (c).

about 68% of the time), one expects it to be in the range 2–6. The true signal at another pixel is twice as strong – 8 photons – and, by the same argument, one expects to measure a value in the range 5–11. *The ranges for both pixels overlap!* With photon counts this low, even if the signal doubles in brightness, we often cannot discern with confidence that the two pixels are even different at all. In fact, it is quite possible that the second pixel gives a *lower* measurement than the first.

On the other hand, suppose the true signal for the first pixel is 100 photons, so we measure something in the range of 90–110. The second pixel, still twice as bright, gives a measurement in the range 186–214. These ranges are larger, but crucially they are not even close to overlapping, so it is very easy to tell the pixels apart. Thus the noise standard deviation alone is not a very good measure of how noisy an image is. The SNR is much more informative: the simple rule is that higher is better. Or, if that still does not feel right, you can turn it upside down and consider the noise-to-signal ratio (the *relative noise*), in which case lower is better (Figure 16.8).

16.3.3 Poisson noise & detection

So why should you care that photon noise is signal-dependent?

One reason is that it can make features of identical sizes and brightnesses easier or harder to detect in an image purely because of the local background. This is illustrated in Figure 16.9. In general, if we want to see a fluorescence increase of a fixed number of photons, this is easier to do if the background is very dark. But if the fluorescence increase is defined *relative* to the background, it will be much easier to identify if the background is high. Either way, when attempting

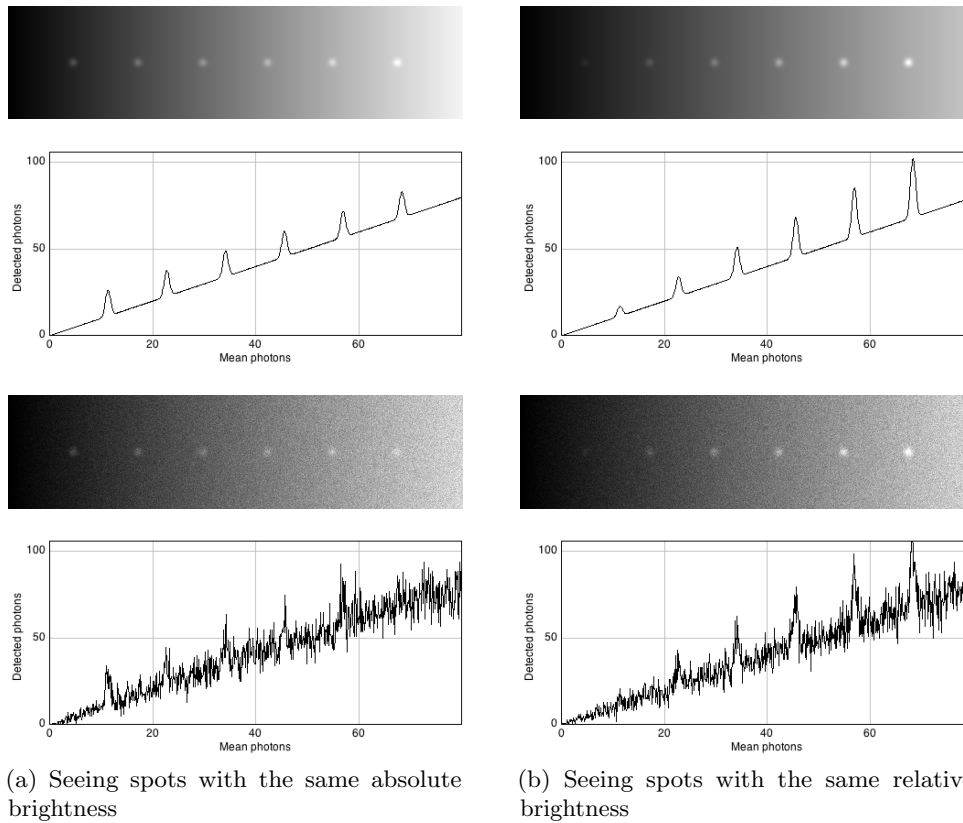


Figure 16.9: The signal-dependence of Poisson noise affects how visible (and therefore detectable) structures are in an image. (a) Six spots of the same *absolute* brightness are added to an image with a linearly increasing background (top) and Poisson noise is added (bottom). Because the noise variability becomes higher as the background increases, only the spots in the darkest part of the image can be clearly seen in the profile. (b) Spots of the same brightness *relative* to the background are added, along with Poisson noise. Because the noise is *relatively* lower as the brightness increases, now only the spots in the brightest part of the image can be seen.

An instance in which we could increase background without adding to the absolute brightness of the meaningful signal (and thereby make detection more difficult) would be if we were to open the pinhole of a confocal microscope very widely, and thereby detect a large amount of out-of-focus light but very little extra in-focus light. By contrast, increasing exposure times would lead to detecting more light from both background and structures of interest, potentially aiding detection by causing the same relative increase in brightness everywhere and improving image quality (at least providing the additional exposure is not too detrimental to the sample).

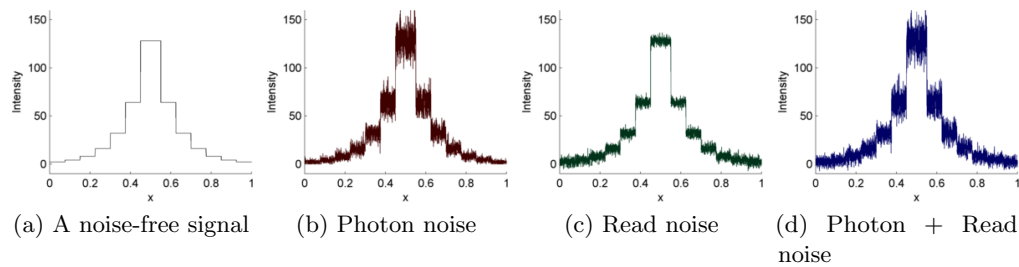
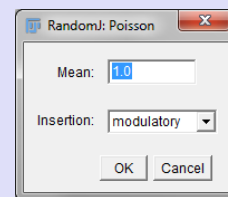


Figure 16.10: An illustration of how photon noise differs from read noise. When *both* are added to a signal (here, a series of steps in which the value doubles at each higher step), the relative importance of each depends upon the value of the signal. At low signal levels this doubling is very difficult to discern amidst either type of noise, and even more so when both noise components are present.

to determine the number of any small structures in an image, for example, we need to remember that the numbers we will be able to detect will be affected by the background nearby. Therefore results obtained from bright and dark regions might not be directly comparable.

Simulating photon noise

You can add simulated Poisson noise to an image using `Process` → `Noise` → `RandomJ` → `RandomJ Poisson`. The pixel values before adding the noise will be treated as the true signal if you select `Insertion: modularity`, in which case the choice of `Mean` setting does not matter.



Practical 16.3

Open the images `mystery_noise_1.tif` and `mystery_noise_2.tif`. Both are noisy, but in one the noise follows a Gaussian distribution (like read noise) and in the other it follows a Poisson distribution (like photon noise). Which is which?

Solution

16.4 Combining noise sources

Combining our noise sources then, we can imagine an actual pixel value as being the sum of three values: the true rate of photon emission, the photon noise component, and the read noise component⁴. The first of these is what we want, while the latter two are random numbers that may be positive or negative.

⁴For a fuller picture, gain and offset also need to be taken into consideration, see Chapter 17.

This is illustrated in Figure 16.10 using a simple 1D signal consisting of a series of steps. Random values are added to this to simulate photon and read noise. Whenever the signal is very low (indicating few photons), the variability in the photon noise is very low (but high *relative* to the signal! (b)). This variability increases when the signal increases. However, in the read noise case (c), the variability is similar everywhere. When both noise types are combined in (d), the read noise dominates completely when there are few photons, but has very little impact whenever the signal increases. Photon noise has already made detecting relative differences in brightness difficult when there are few photons; with read noise, it can become hopeless.

Therefore overcoming read noise is critical for low-light imaging, and the choice of detector is extremely important (see Chapter 17.3). But, where possible, detecting more photons is an *extremely* good thing anyway, since it helps to overcome *both* types of noise.

Other noise sources

Photon and read noise are the main sources of noise that need to be considered when designing and carrying out an experiment. One other source often mentioned in the literature is *dark noise*, which arises when a wayward electron causes the detector to register a photon even when there was not actually one there. In very low-light images, this lead to spurious bright pixels. However, dark noise is less likely to cause problems if many true photons are detected, and many detectors reduce its occurrence by cooling the sensor.

If the equipment is functioning properly, other noise sources could probably not be distinguished from these three. Nevertheless, brave souls who wish to know more may find a concise, highly informative, list of more than 40 sources of imprecision in *The 39 steps: a cautionary tale of quantitative 3-D fluorescence microscopy* by James Pawley:

http://www.zoology.wisc.edu/faculty/Paw/pdfs/The_39_Steps_corrected.pdf

Question 16.4

Suppose you have an image that does not contain much light, but has some isolated bright pixels. Which ImageJ command could you use to remove them? And is it safe to assume they are due to dark noise or something similar, or might the pixels correspond to actual bright structures?



Solution

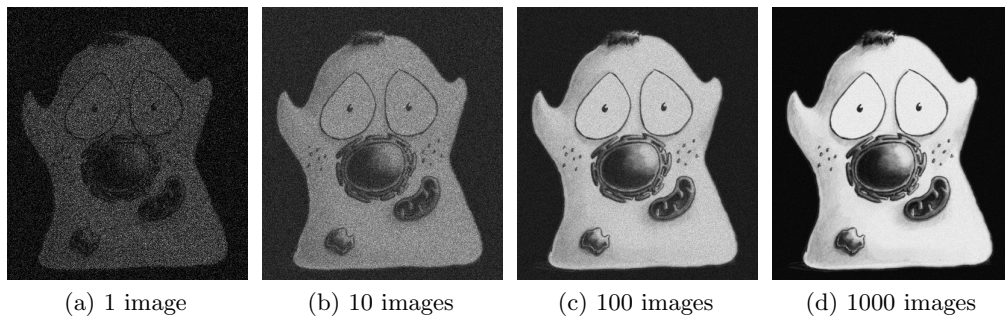


Figure 16.11: The effect of adding (or averaging) multiple noisy images, each independent with a similar SNR.

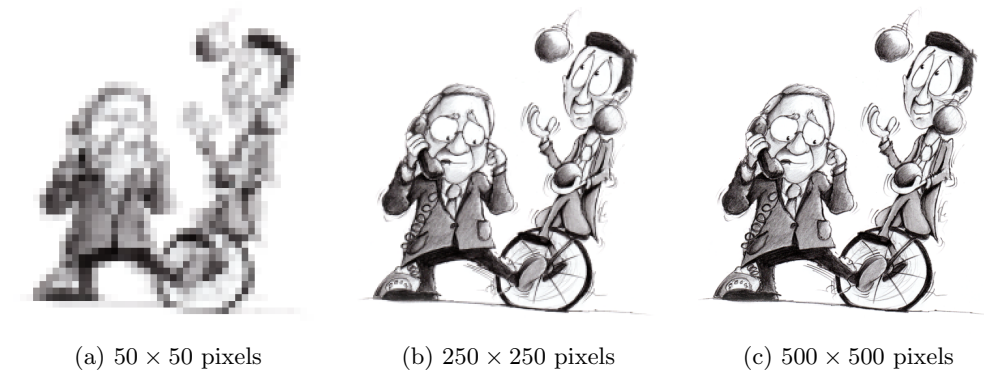


Figure 16.12: Harry Nyquist (1889-1975) and Claude Shannon (1916-2001), sampled using different pixel sizes. Their work is used when determining the pixel sizes needed to maximize the available information when acquiring images, which depends upon the size of the PSF.

16.5 Finding photons

There are various places from which the extra photons required to overcome noise might come. One is to simply acquire images more slowly, spending more time detecting light. If this is too harsh on the sample, it may be possible to record multiple images quickly. If there is little movement between exposures, these images could be added or averaged to get a similar effect (Figure 16.11). An alternative would be to increase the pixel size, so that each pixel incorporates photons from larger regions – although clearly this comes at a cost in spatial information (see Section 17.3.3).

Nyquist sampling

Small pixels are needed to see detail, but also reduce the number of photons

per pixel and thereby increase noise. However, Chapter 15 has already argued that ultimately it is not pixel size, but rather the PSF that limits spatial resolution – which suggests that there is a minimum pixel size below which nothing is gained, and the only result is that more noise is added.

This size can be determined based upon knowledge of the PSF and the *Nyquist-Shannon sampling theorem* (Figure 16.12). Images acquired with this pixel size are said to be *Nyquist sampled*. The easiest way to determine the corresponding pixel size for a given experiment is to use the online calculator provided by Scientific Volume Imaging at <http://www.svi.nl/NyquistCalculator>. You may need larger pixels to reduce noise or see a wider field of view, but you do not get anything extra by using smaller pixels.

Solutions

Question 16.1

- We detect 10 photons, read noise std. dev. 1 photon: $SNR = 10$
- We detect 100 photons, read noise std. dev. 10 photons: $SNR = 10$
- We detect 1000 photons, read noise std. dev. 10 photons: $SNR = 100$ The noise causes us a similar degree of uncertainty in the first two cases. In the third case, the noise is likely to be less problematic: higher SNRs are good.

Practical 16.1 The standard deviation of the result should be close to $\sqrt{9 + 16} = 5$

Question 16.2 When operating on the noise images with standard deviations of 5, the (approximate) standard deviations you should get are:

1. $\sqrt{50} = 7.1$ – add the variances, then take the square root
2. $\sqrt{50} = 7.1$ – same as addition; switching the sign (positive or negative) of a Gaussian noise image with zero mean just gives another Gaussian noise image with zero mean
3. $\sqrt{50} = 7.1$ – the order of subtraction doesn't matter
4. $\sqrt{50}/2 = 3.5$ – same as addition, but divide the result by 2
5. 10 – equivalent to multiplying the image by 2, so multiply the standard deviation by 2

Practical 16.2 After applying the filter once, the standard deviation should be around 10. Using a 3×3 mean filter, the noise standard deviation should be reduced to around 1/3 of its original value.

You can break down the problem this way:

- Let the original noise standard deviation be σ , the variance is σ^2
- The filter first replaces each pixel with the sum of 9 independent values. The variance becomes $9\sigma^2$, the standard deviation $\sqrt{9\sigma^2} = 3\sigma$
- The filter divides the result by 9 to give means. This also divides the standard deviation by 9, giving $3\sigma/9 = \sigma/3$

However, when I apply the filter a second time, the standard deviation is 7. It has decreased by much less, despite using the same filter. This is because *after the first filtering, the noise is no longer independent at each pixel.*

Question 16.3 The probability of detecting 5 photons is approximately 0.0031.

$$\frac{e^{-1}}{5!} = \frac{1}{120e} = 0.0031 \quad (16.5)$$

Although this is a very low probability, images contain so many pixels that one should expect to see such noisy values often. For example, in a rather dark and dull 512×512 pixel image in which the average photon emission rate is 1, we would expect 800 pixels to have a value of 5 – and two pixels even to have a value of 8. The presence of isolated bright or dark pixels therefore usually tells us very little indeed, and it is only by processing the image more carefully and looking at surrounding values that we can (sometimes) discount the possibility these are simply the result of noise.

Practical 16.3 The noise in `mystery_noise_1.tif` is Gaussian, it is Poisson in `mystery_noise_2.tif`. Since there are reasonably flat regions within the cell and background, I would test this by drawing a ROI within each and measuring the standard deviations. Where these are similar, the noise is Gaussian; if there is a big difference, the noise is likely to be Poisson.

If no flat regions were available, I would try applying a gradient filter with the coefficients `-1 1 0`, and inspecting the results. Alternatively, I might try plotting a fluorescence profile or subtracting a very slightly smoothed version of each image.

Question 16.4 A median filter is a popular choice for removing isolated bright pixels, although I sometimes prefer `Process` \rightarrow `Noise` \rightarrow `Remove Outliers...` because this only puts the median-filtered output in the image if the original value was really extreme (according to some user-defined threshold). This then preserves the independence of the noise at all other pixels – so it still behaves reliably and predictably like Poisson + Gaussian noise. We can reduce the remaining noise with a Gaussian filter if necessary.

Assuming that the size of a pixel is smaller than the PSF (which is usually the case in microscopy), it is a good idea to remove these outliers. They *cannot* be real structures, because any real structure would have to extend over a region at least as large as the PSF. However if the pixel size is very large, then we may not be able to rule out that the ‘outliers’ are caused by some real, bright structures.

Microscopes & detectors

Chapter outline

- *The choice of microscope influences the blur, noise & temporal resolution of images*
- *An ideal detector would have a high Quantum Efficiency & low read noise*

17.1 Introduction

Successfully analyzing an image requires that it actually contains the necessary information in the first place. There are various practical issues related to the biology (e.g. not interfering too much with processes, such as by inadvertently killing things) and data handling (bit-depths, file formats, anything else in Part I). However, assuming that these are in order, there are three other main factors to consider connected to the image contents:

1. *Spatial information*, dependent on the size and shape of the PSF,
2. *Noise*, dependent on the number of photons detected,
3. *Temporal resolution*, dependent on the speed at which the microscope can record images.

No one type of microscope is currently able to optimize all of these simultaneously, and so decisions and compromises need to be made. Temporal resolution and noise have an obvious relationship, in that a high temporal resolution means that less time is spent detecting photons in the image, leading to more photon noise. However, improving spatial information is also often related to one or both of the other two factors.

This chapter gives a very brief overview of the main features of several fluorescence microscopes from the point of view of how they balance the tradeoffs mentioned. Be aware that it does *not* do justice to all the complexities, variations and features of the microscopes it describes! But it is included nonetheless in case it might be useful to anyone as a starting point.

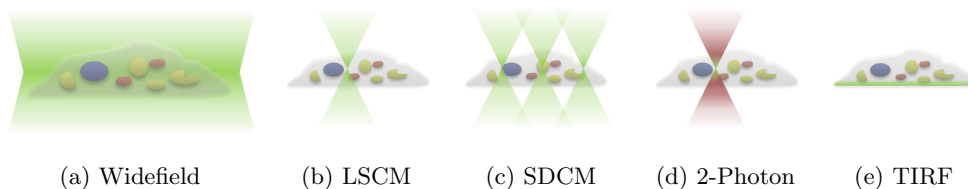


Figure 17.1: Schematic diagrams to show the differences in excitation patterns. Here, an inverted microscope is assumed (the cell is resting on a coverslip, and illuminated from below). During the recording of a pixel, light can potentially be detected if it arises from any part of the green-illuminated region, although in the laser scanning and spinning disk confocal cases the pinhole will only permit a fraction of this light to pass through. Note that, for the 2-photon microscope, the excitation is confined to only the small, central region, while the red light above and below is not capable of exciting the fluorophores that have been used.

17.2 Types of microscope

17.2.1 Widefield

So far, our schematic diagrams (e.g. Figure 14.1) and PSF discussion (Chapter 15) have concentrated on *widefield microscopes*. In widefield microscopy, the entire sample is bathed in light, so that many fluorophores throughout the sample can be excited and emit photons simultaneously. All of the light that enters the objective may then be detected and contribute to any image being recorded. Because photons are potentially emitted from everywhere in the sample, there will be a lot of blur present for a thick specimen (for a thin specimen there is not much light from out-of-focus planes because the planes simply do not contain anything that can fluoresce). What we get in any single recorded image is effectively the sum of the light from the in-focus plane, and every other out-of-focus plane. Viewed in terms of PSFs, we capture part of the hourglass shape produced by every light-emitting point depending upon how in-focus it is in every image (Figure 15.4).

This is bad news for spatial information and also for detecting small structures in thick samples, as the out-of-focus light is essentially background (see Figure 16.9a). On the other hand, widefield images tend to include a lot of photons, which overcomes read noise, even when they are recorded fast.

17.2.2 Laser scanning confocal

Optical sectioning is the ability to detect photons only from the plane of focus, rejecting those from elsewhere. This is necessary to look into thick samples without getting lost in the haze of other planes. *Laser Scanning Confocal Microscopy (LSCM)* achieves this by only concentrating on detecting light for one pixel at any

time, and using a pinhole to try to only allow light from a corresponding point in the sample to be detected.

Because at most only one pixel is being recorded at any given moment, it does not make sense to illuminate the entire sample at once, which could do unnecessary damage. Rather, a laser illuminates only a small volume. If this would be made small enough, then a pinhole would not actually be needed because we would know that all emitted light *must* be coming from the illuminated spot; however, the illumination itself is like an hour-glass PSF in the specimen and so the pinhole is necessary to make sure only the light from the central part is detected. This then causes the final PSF, as it appears *in the image*, to take on more of a rugby-ball (or American football) appearance.

The end result is an image that has relatively little out-of-focus light. This causes a significant improvement in what can be seen along the z -dimension, although the xy resolution is not very different to the widefield case. Also, because single pixels are recorded separately, the image can (potentially) be more or less any size and shape – rather than limited by the pixel count on a camera (see Section 17.3.2).

However, these advantages comes with a major drawback. Images usually contain thousands to millions of pixels, and so only a tiny fraction of the time required to record an image is spent detecting photons for any one pixel in a LSCM image – unlike in the widefield case, where photons can be detected for all pixels over the entire image acquisition time. This can cause LSCM image acquisition to be comparatively slow, noisy (in terms of few detected photons) or both. Spatial information is therefore gained at a cost in noise and/or temporal resolution.

Question 17.1

Why is it often recommended that the pinhole in LSCM be set to have the same size as one Airy disk? And why is it sometimes better not to strictly follow this recommendation?

Solution

17.2.3 Spinning disk confocal

So a widefield system records all pixels at once without blocking the light from reaching the detector anywhere, whereas a LSCM can block out-of-focus light by only recording a single pixel at a time. Both options sound extreme: could there not be a compromise?

Spinning disk confocal microscopy (SDCM) implements one such compromise by using a large number of pinholes punched into a disk, so that the fluorophores can be excited and light detected for many different *non-adjacent* pixels simultaneously. By not trying to record adjacent pixels at the same time, pinholes can be used to block *most* of the out-of-focus light for each pixel, since this originates close to the region of interest for that pixel (i.e. the PSF becomes very

dim at locations far away from its centre) – but sufficiently-spaced pixels can still be recorded simultaneously with little influence on one another.

In practice, spinning disk confocal microscopy images are likely to be less sharp than laser scanning confocal images because some light does scatter through other pinholes. Also, the pinhole sizes cannot be adjusted to fine-tune the balance between noise and optical sectioning. However, the optical sectioning offered by SDCM is at least a considerable improvement over the widefield case, and an acceptable SNR can be achieved with much faster frame-rates using SDCM as opposed to LSCM.

17.2.4 Multiphoton microscopy

As previously mentioned, if the size of excitation volume could be reduced enough, then we would know where the photons originated even without a pinhole being required – but normally focused excitation is still subject to PSF-related issues, so that fluorophores throughout a whole hourglass-shaped volume can end up being excited. The main idea of multiphoton microscopy is that fluorophore excitation requires the *simultaneous absorption of multiple photons*, rather than only a single photon. The excitation light has a longer wavelength than would otherwise be required to cause molecular excitation with a single photon, but the energies of the multiple photons combine to cause the excitation.

The benefit of this is that the multiphoton excitation only occurs at the focal region of the excitation – elsewhere within the ‘specimen PSF’ the light intensities are insufficient to produce the ‘multiphoton effect’ and raise the fluorophores into an excited state. This means that the region of the specimen emitting photons at any one time is much smaller than when using single photon excitation (as in LSCM), and also less damage is being caused to the sample. Furthermore, multiphoton microscopy is able to penetrate deeper into a specimen – up to several hundred μm .

17.2.5 Total Internal Reflection Fluorescence

As previously mentioned, widefield images of very thin specimens do not suffer much from out-of-focus blur because light is not emitted from many other planes. *Total Internal Reflection Fluorescence* (TIRF) microscopy makes use of this by stimulating fluorescence only in a very thin section of the sample close to the objective. Very briefly, TIRF microscopy involves using an illumination angled so that the change in refractive index encountered by the light as it approaches the specimen causes a further change in angle sufficient to prevent the light from directly entering the specimen (i.e. it is ‘totally internally reflected’). Nevertheless, fluorophores can still be excited by an evanescent wave that is produced when this occurs. This wave decays exponentially, so that only fluorophores right at the surface are excited – meaning fluorophores deeper within the specimen do not interfere with the recording.

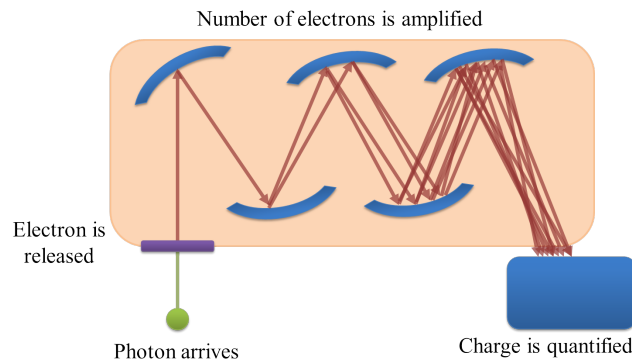


Figure 17.2: Diagram showing the detection of a photon by a PMT. Each photon can be ‘multiplied’ to produce many electrons by accelerating the first produced electron towards a dynode, and repeating the process for all electrons produced along a succession of dynodes. The charge of the electrons reaching the end of the process can then be quantified, and ought to be proportional to the number of photons that arrived at the PMT.

Importantly, because the subsequent recording is essentially similar to that used when recording widefield images, photons are detected at all pixels in parallel and fast recording-rates are possible. Therefore if it is only necessary to see to a depth of about 100 nm, TIRF microscopy may be a good choice.

17.3 Photon detectors

Certain detectors are associated with certain types of microscopy, and differ according to the level and type of noise you can expect. Understanding the basic principles greatly helps when choosing sensible values for parameters such as gain, pixel size and binning during acquisition to optimize the useful information in the image. The following is an introduction to three common detectors.

17.3.1 PMTs: one pixel at a time

If you only need to record one pixel at a time, a *photomultiplier tube* (PMT) might be what you need. The basic principle is this: a photon strikes a photocathode, hopefully producing an electron. When this occurs, the electron is accelerated towards a dynode, and any produced electrons accelerated towards further dynodes. By adjusting the ‘gain’, the acceleration of the electrons towards successive dynodes can be varied; higher accelerations mean there is an increased likelihood that the collision of the electrons with the dynode will produce a higher number of electrons moving into the next stage. The charge of the electrons is then quantified at the end (Figure 17.2), but because the (possibly very small) number of original detected photons have now been amplified to a (possibly much) larger number

electrons by the successive collisions with dynodes, the effect of read noise is usually minor for a PMT.

More problematically, PMTs generally suffer from the problem of having low *quantum efficiencies* (QEs). The QE is a measure of the proportion of photons striking the detector which then produce electrons, and typical values for a conventional PMT may be only 10–15%: the majority of the photons reaching the PMT are simply wasted. Thus photon noise can be a major issue, especially if there is a low amount of light available to detect in the first place.

Converting electrons to pixels

It is important to note that the final pixel values are not *equal* to numbers of detected photons – nor even numbers of counted electrons. They are rather proportional, often with an offset added. It is essential to estimate this offset (perhaps from a background region or an image acquired without using any excitation light) and subtract it if comparing pixel values in different images, as well as to use identical acquisition settings.

17.3.2 CCDs: fast imaging when there are a lot of photons

A *Charged Coupled Device* (CCD) is a detector with a region devoted to sensing photons, and which is subdivided into different ‘physical pixels’ that correspond to pixels in the final image. Thus the image size cannot be changed arbitrarily, but it is possible to record photons for many pixels in parallel.

When a photon strikes a pixel in the sensing part of the CCD, this often releases an electron – the QE is typically high (perhaps 90%). After a certain exposure time, different ‘electron clouds’ have then formed at each physical pixel on the sensor, each of which has a charge related to the number of colliding photons. This charge is then measured by passing the electrons through a charge amplifier, and the results used to assign an intensity value to the pixel in the final image (Figure 17.3).

The electron clouds for each pixel might be larger than in the PMT case, both because of the higher QE and because more time can be spent detecting photons (since this is carried out for all pixels simultaneously). However, the step of amplifying the numbers of electrons safely above the read noise before quantification is missing. Consequently, read noise is potentially more problematic, and in some cases can even dominate the result.

17.3.3 Pixel binning

One way to address the issue of CCD read noise is to use *pixel binning*. In this case, the electrons from 4 (i.e. 2×2) pixels of the CCD are added together before being quantified: the electron clouds are approximately 4 times bigger relative to the read noise (Figure 17.4), and readout can be faster because fewer electron

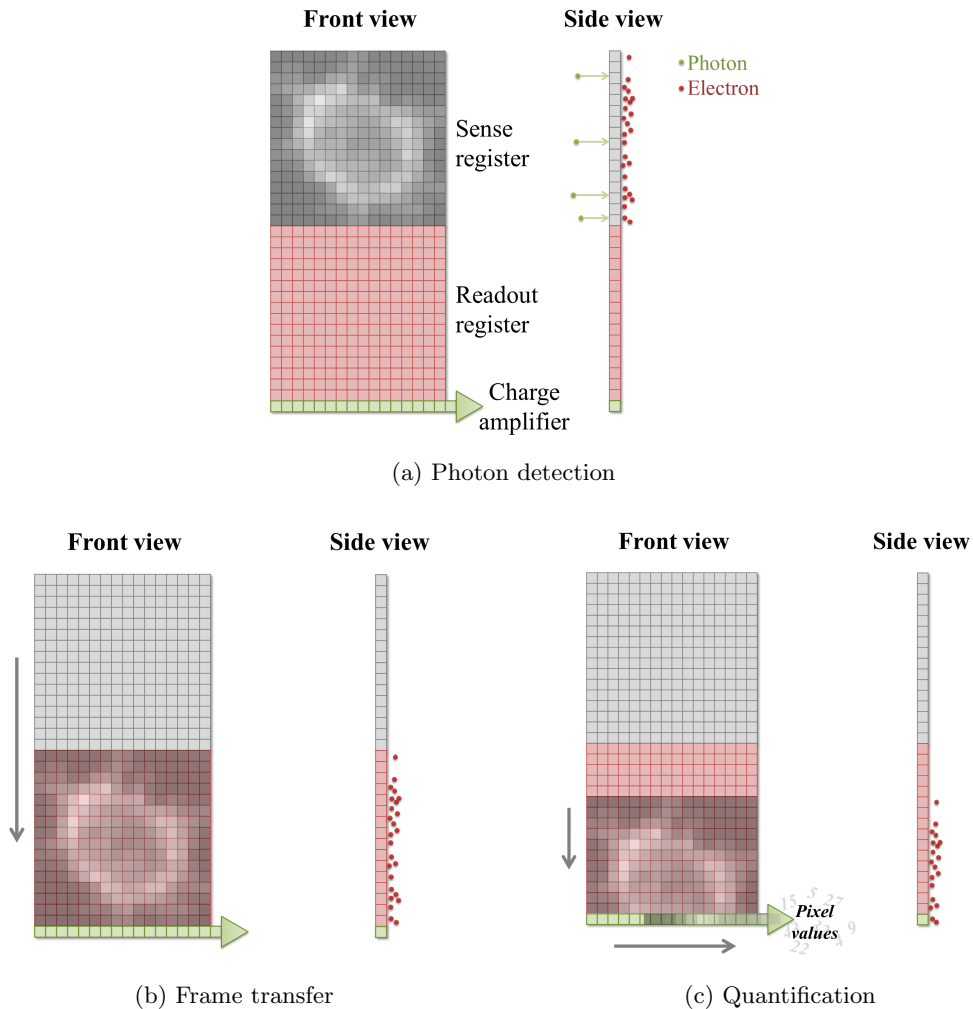


Figure 17.3: An illustration of the basic operation of a CCD camera (using frame transfer). First, photons strike a sense register, which is divided into pixels. This causes small clouds of electrons to be released and gather behind the pixels (a). These are then rapidly shifted downwards into another register of the same size, thereby freeing the sense register to continue detecting photons (b). The electron clouds are then shifted downwards again, one row at a time, with each row finally being shifted sequentially through a charge amplifier (c). This quantifies the charge of the electron clouds, from which pixel values for the final image are determined.

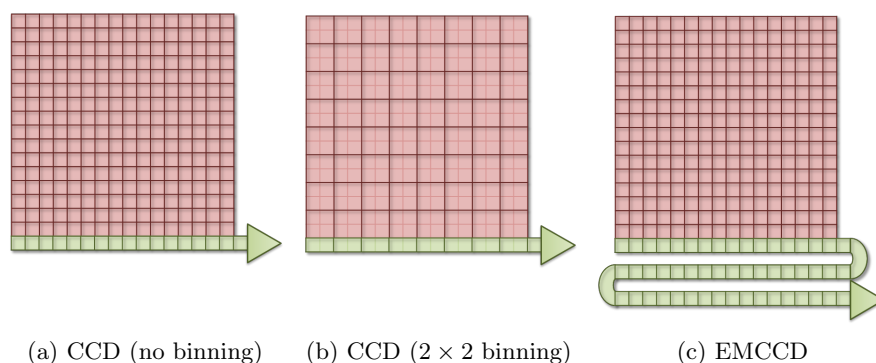


Figure 17.4: A simplified diagram comparing a conventional CCD (with and without binning) and an EMCCD. While each has the same physical number of pixels, when binning is used electrons from several pixels are combined before readout – thereby making the ‘logical’ pixels in the final image bigger. For the EMCCD, the electrons are shifted through a ‘gain register’ prior to quantification. See Figure 17.3 for additional labels; the sense register has been omitted for simplicity.

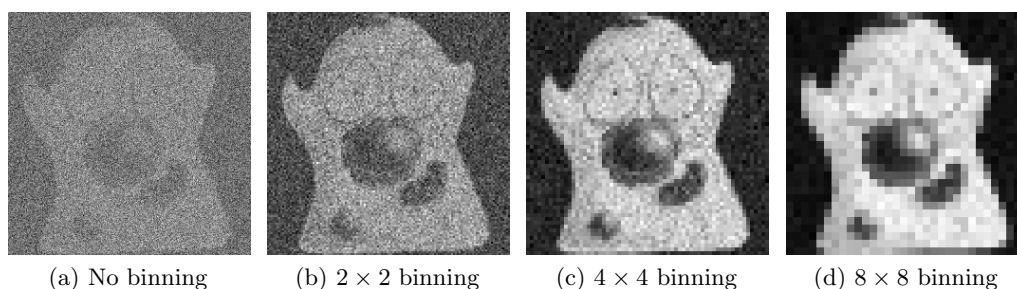


Figure 17.5: Illustration of the effect of binning applied to an image suffering from photon and read noise. As the bin size increases, the photons from neighbouring pixels are combined into a single (larger) pixel before the read noise is added. As a consequence, the image becomes brighter relative to the read noise – but at a cost of spatial information.

clouds need to be quantified. The obvious disadvantage of this is that one cannot then put the electrons from the 4 pixels ‘back where they belong’. As a result, the binned measurement is simply treated as a single (bigger) pixel. The recorded image contains 25% of the pixels in the unbinned image, while still covering the same field of view, so spatial information is lost. Larger bins may also be used, with a correspondingly more dramatic impact upon image size (Figure 17.5).

17.3.4 EMCCDs: fast imaging with low light levels

And so you might wonder whether it is possible to increase the electron clouds (like with the PMT) with a CCD, and so get its advantages without the major inconvenience of read noise. *Electron Multiplying CCDs* (EMCCDs) achieve this

to some extent. Here, the electrons are first passed through an additional ‘gain register’ before quantification. At every stage of this gain register, each electron has a small probability – perhaps only 1% – of being amplified (through ‘impact ionisation’) and giving rise to two electrons entering the next stage. Despite the small probability, by incorporating > 500 such stages, the size of the electron cloud arising from even a single photon may be amplified safely above the read noise.

However, the randomness of the amplification process itself introduces a new source of uncertainty, so that the final outcome can be thought of as having the same precision as if perhaps only around half as many photons were detected (see Section 16.3 for the relationship between noise and the number of photons). Therefore read noise is effectively overcome at the cost of more photon noise.

Question 17.2

From a practical point of view, an EMCCD is rather like having a CCD with no read noise, but with half the QE. Under what circumstances (i.e. high or low numbers of photons) is an EMCCD preferable to a CCD? *Solution*

Question 17.3

Based upon the above descriptions, which detectors seem most appropriate (generally!) for (a) widefield microscopy, (b) SDCM and (c) LSCM? *Solution*

Question 17.4

Section 17.3.3 described how CCDs can use 2×2 binning to combine the electrons corresponding to multiple pixels together into a single pixel, which is then ‘less noisy’. A similar effect can be achieved by just acquiring an image without binning and applying a 2×2 filter, in which all the coefficients have values of one. Both techniques result in images that have roughly four times as many photons contributing to each pixel, and thus a better SNR.

Think of one major advantage and one disadvantage of using filtering *after* acquisition, rather than binning *during* acquisition. *Solution*

Solutions

Question 17.1 As described in Chapter 15, the vast majority (approximately 80%) of the light from the in-focus plane falls within the Airy disk (see Figure 15.6c). Using a pinhole smaller than this can result in so little light being detected that the image becomes too noisy to be useful. On the other hand, increasing the size of the pinhole will result in some more of the remaining 20% located in the outer rings of the Airy pattern, but *most* extra photons will come from out-of-focus planes. This causes a reduction in the effectiveness of the optical sectioning. Therefore, a pinhole diameter of approximately 1 Airy disk provides a reasonable balance between detecting most of the in-focus light and achieving good optical sectioning.

Nevertheless, sometimes a reduction in optical sectioning is a worthwhile cost – such as when photons are very scarce, and some extra background is tolerable. Also, when recording a multichannel image then you may well have to set the pinhole size according to the Airy disk for one channel. But because the size of the disk depends upon the light wavelength (Equation 15.1), the pinhole diameter will differ in terms of Airy disk sizes for other channels.

Question 17.2 The gain register of EMCCDs offers benefits primarily when few photons are available (i.e. when read-noise is the main problem, such as in SDCM). CCDs are preferable when many photons are available (e.g. in widefield).

If you are skeptical about this, consider an image in which your read noise standard deviation is 10 electrons and you detect on average 9 electrons (originally photons). The photon noise then has a standard deviation of 3. The read noise is much larger and will completely dominate the image: nothing interesting will be visible. It is then worth the cost of even more photon noise to be able to eliminate the read noise. The final image will still look pretty bad, but at least interpreting it is not hopeless.

But suppose you happen to have 90000 detected photons instead, in which case the standard deviation of the photon noise is now 300. The read noise of 10 is comparatively insignificant, and there is nothing to gain from electron multiplication and making the photon noise situation worse.

Question 17.3 The following are reasonable rules of thumb:

- *Widefield microscopy:* A CCD is suitable because of its ability to record many pixels simultaneously. The large number of photons normally detected means that read noise is not usually a big issue, and an EMCCD can make the problem of noise worse instead of better.
- *Spinning Disk Confocal Microscopy:* A CCD may be used, but an EMCCD is often preferable. This is because SDCM usually gives lower photon counts

(certainly lower than in a comparable widefield image), which can mean that read noise would dominate the result unless the photons are somehow amplified.

- *Laser Scanning Confocal Microscopy*: PMTs are suitable, since the image is built up one pixel at a time.

Question 17.4 A 2×2 binned image contains $1/4$ the number of pixels of the original image. This represents a considerable loss of spatial information, and you would get a different result if you were to start binning at the first or second row or column, since different pixels would be combined in each case. On the other hand, filtering has the advantage of giving you an image that is exactly the same size as the original. This is like getting all four possible binned images for the price of one (i.e. all four different ways to split the image into 2×2 pixel blocks, instead of just one way), so less spatial information is lost. With filtering you also have much more flexibility: you might choose a 3×3 filter instead, or a Gaussian filter, or a range of different filtering options to see which is best. With binning, you need to choose one option during acquisition and stick with it.

However, if read noise is a major problem then filtering might not be such a good choice. This is because read noise is added to every acquired pixel once, and it does not matter if you have a few photons or many – its standard deviation remains the same. Therefore, if the electrons from four pixels are combined by binning during acquisition, then only one ‘unit’ of read noise appears in the image corresponding to those pixels. But an unbinned image would have read noise added four times, and even after 2×2 filtering this is still more read noise than in a comparable binned image. Sometimes this extra noise is too high a cost for the flexibility of filtering, and binning is better. (In this regard, remember that read noise is typically worse for CCD cameras, but not such a problem for EMCCDs or PMTs.)

Simulating image formation

18.1 Introduction

The Difference of Gaussians macro developed in Chapter 13 was useful, but quite simple. This chapter contains an extended practical, the goal of which is to develop a somewhat more sophisticated macro that takes an ‘ideal’ image, and then simulates how it would look after being recorded by a fluorescence microscope. It can be used not only to get a better understanding of the image formation process, but also to generate test data for analysis algorithms. By creating simulations with different settings, we can investigate how our results might be affected by changes in image acquisition and quality.

18.1.1 Image formation summary

The following is a summary of the aspects of image formation discussed so far:

- Images are composed of pixels, each of which has a *single numeric value* (not a colour!).
- The value of a pixel in fluorescence microscopy relates to a number of detected *photons* – or, more technically, the charge of the electrons produced by the photons striking a detector.
- Images can have many *dimensions*. The number of dimensions is essentially the number of things you need to know to identify each pixel (e.g. time point, channel number, *x* coordinate, *y* coordinate, *z* slice).
- The two main factors that limit image quality are *blur* and *noise*. Both are inevitable, and neither can be completely overcome.
- Blur is characterized by the *point spread function (PSF)* of the microscope, which is the 3D volume that would be the result of imaging a single light-emitting point. It acts as a *convolution*.
- In the focal plane, the image of a point is an *Airy pattern*. Most of the light is contained within a central region, the *Airy disk*.

- The *spatial resolution* is a measure of the separation that must exist between structures before they can adequately be distinguished as separate, and relates to the size of the PSF (or Airy disk in 2D).
- The two main types of noise are *photon noise* and *read noise*. The former is caused by the randomness of photon emission, while the latter arises from imprecisions in quantifying numbers of detected photons.
- Detecting more photons helps to overcome the problems caused by *both* types of noise.
- Different types of microscope have different advantages and costs in terms of *spatial information*, *temporal resolution* and *noise*.
- *PMTs* are used to detect photons for single pixels, while *CCDs* and *EMCCDs* are used to detect photons for many pixels in parallel.

The macro in this chapter will work for 2D images, and simulate the three main components:

1. the blur of the PSF
2. the inclusion of photon noise
3. the addition of read noise

Furthermore, the macro will ultimately be written in such a way that allows us to investigate the effects of changing some additional parameters:

- the size of the PSF (related to the objective lens NA and microscope type)
- the amount of fluorescence being emitted from the brightest region
- the amount of background (from stray light and other sources)
- the exposure time (and therefore number of detected photons)
- the detector's offset
- the detector's gain
- the detector's read noise
- camera binning
- bit-depth

18.2 Recording the main steps

It does not really matter which image you use for this, but I recommend a single-channel 2D image that starts out without any obvious noise (e.g. a single channel from `HeLa Cells`). After starting the macro recorder, completing the following steps will create the main structure for the macro:

- Ensure the starting image is 32-bit.
- Run `Gaussian Blur...` using a sigma value of 2 to simulate the convolution with the PSF.
- Here, we will assume that there are some background photons from other sources, but around the same number at every pixel in the image. So we can simply add a constant to this image using `Add...`. The value should be small, perhaps 10.
- The image now contains the ‘average rates of photon emission’ that we would normally like to have for one particular exposure time (i.e. it is noise-free). If we change the exposure time, we should change the pixel values similarly so that the rates remain the same. Because adjusting the exposure works like a simple multiplication, we can use the `Multiply...` command. Set it to a ‘default’ value of 1 for now.
- To convert the photon emission rates into actual photon counts that we could potentially detect, we need to simulate photon noise by replacing each pixel by a random value from a Poisson distribution that has the same λ as the rate itself. Apply this using `RandomJ Poisson`, making sure to set the `Insertion:` value to `Modulatory`. The `Mean` value will be ignored, so its setting does not matter. Notice that all the pixels should now have integer values: you cannot detect parts of photons.
- The detector gain scales up the number of electrons produced by detected photons. Make room for it by including another multiplication, although for now set the value to 1 (i.e. no extra gain).
- Simulate the detector offset by adding another constant, e.g. 100.
- Add read noise with `Process → Noise → Add Specified Noise...`, setting the standard deviation to 5.
- Clip any negative values, by running `Min...` and setting the value to 0.
- Clip any positive values that exceed the bit-depth, by running `Max...`. To assume an 8-bit image, set the value to 255.

Now is a good time to clean up the code by removing any unnecessary lines, adding suitable comments, and bringing any interesting variables up to the top of the macro so that they can be easily modified later (as in Section 13.2). The end result should look something like this:

```
// Variables to change
psfSigma = 2;
backgroundPhotons = 10;
exposureTime = 1;
readStdDev = 5;
detectorGain = 1;
detectorOffset = 100;
maxVal = 255;

// Ensure image is 32-bit
run("32-bit");

// Simulate PSF blurring
run("Gaussian Blur...", "sigma="+psfSigma);

// Add background photons
run("Add...", "value="+backgroundPhotons);

// Multiply by the exposure time
run("Multiply...", "value="+exposureTime);

// Simulate photon noise
run("RandomJ Poisson", "mean=1.0 insertion=modulatory");

// Simulate the detector gain
run("Multiply...", "value="+detectorGain);

// Simulate the detector offset
run("Add...", "value="+detectorOffset);

// Simulate read noise
run("Add Specified Noise...", "standard="+readStdDev);

// Clip any negative values
run("Min...", "value=0");

// Clip the maximum values based on the bit-depth
run("Max...", "value="+maxVal);
```


You would have a perfectly respectable macro if you stopped now, but the following section contains some ways in which it may be improved.

18.3 Making improvements

Normalizing the image

The results you get from running the above macro will change depending upon the original range of the image that you use: that is, an image that starts off with high-valued pixels will end up having much less noise. To compensate for this somewhat, we can first normalize the image so that all pixels fall into the range 0–1. To do this, we need to determine the current range of pixel values, which can be found out using the macro function:

```
getStatistics(area, mean, min, max);
```

After running this, four variables are created giving the `mean`, `minimum` and `maximum` pixel values in the image, along with the total image `area`. Normalization is now possible using `Subtract` and `Divide` commands, and adjusting their values. In the end this gives us

```
getStatistics(area, mean, min, max);
run("Subtract...", "value="+min);
divisor = max - min;
run("Divide...", "value="+divisor);
```

Varying the fluorescence emission rate

The new problem we will have after normalization is that there will be a maximum photon emission rate of 1 in the brightest part of the image, which will give us a image dominated completely by noise. We can change this by multiplying the pixels again, and so define what we want the emission rate to be in the brightest part of the image. I suggest creating a variable for this, and setting its value to 10. Then add the following line immediately after normalization:

```
run("Multiply...", "value="+maxPhotonEmission);
```

Modifying this value allows you to change between looking at samples that are fluorescing more or less brightly. For a less bright sample, you will most likely need to increase the exposure time to get a similar amount of signal – but beware that increasing the exposure time also involves collecting more unhelpful background photons, so is not quite so good as having a sample where the important parts are intrinsically brighter.

Simulating binning

The main idea of binning is that the electrons from multiple pixels are added together prior to readout, so that the number of electrons being quantified is

bigger relative to the read noise. For 2×2 binning this involves splitting the image into distinct 2×2 pixel blocks, and creating another image in which the value of each pixel is the sum of the values within the corresponding block.

This could be done using the `Image` \rightarrow `Transform` \rightarrow `Bin` command, with `shrink factors` of 2 and the `Sum` bin method. The macro recorder can again be used to get the main code that is needed. After some modification, this becomes

```
if (doBinning) {
  run("Bin...", "x=2 y=2 bin=Sum");
}
```

By enclosing the line within a *code block* (limit by the curly brackets) and beginning the block with `if (doBinning)`, it is easy to control whether binning is applied or not. You simply add an extra variable to your list at the start of the macro

```
doBinning = true;
```

to turn binning on, or

```
doBinning = false;
```

to turn it off. These lines performing the binning should be inserted *before* the addition of read noise.

Varying bit-depths

Varying the simulated bit-depths by changing the maximum value allowed in the image takes a little work: you need to know that the maximum value in an 8-bit image is 255, while for a 12-bit image it is 4095 and so on. It is more intuitive to just change the image bit-depth and have the macro do the calculation for you. To do this, you can replace the `maxVal = 255;` variable at the start of the macro with `nBits = 8;` and then update the later clipping code to become

```
maxVal = pow(2, nBits) - 1;
run("Max...", "value="+maxVal);
```

Here, `pow(2, nBits)` is a function that gives you the value of 2^{nBits} . Now it is easier to explore the difference between 8-bit, 12-bit and 14-bit images (which are the main bit-depths normally associated with microscope detectors, even if the resulting image is stored as 16-bit).

Rounding to integer values

The macro has already clipped the image to a specified bit-depth, but it still contains 32-bit data and so potentially has non-integer values that could not be stored in the 8 or 16-bit images a microscope typically provides as output. Therefore it remains to round the values to the nearest integer.

There are a few ways to do this: we can convert the image using `Image` \rightarrow `Type` \rightarrow commands, though then we need to be careful about whether there

will be any scaling applied. However, we can avoid thinking about this if we just apply the rounding ourselves. To do it we need to visit each pixel, extract its value, round the value to the nearest whole number, and put it back in the image. This requires using *loops*. The code, which should be added at the end of the macro, looks like this:

```
// Get the image dimensions
width = getWidth();
height = getHeight();

// Loop through all the rows of pixels
for (y = 0; y < height; y++) {
  // Loop through all the columns of pixels
  for (x = 0; x < width; x++) {
    // Extract the pixel value at coordinate (x, y)
    value = getPixel(x, y);
    // Round the pixel value to the nearest integer
    value = round(value);
    // Replace the pixel value in the image
    setPixel(x, y, value);
  }
}
```

This creates two variables, *x* and *y*, which are used to store the horizontal and vertical coordinates of a pixel. Each starts off set to 0 (so we begin with the pixel at 0,0, i.e. in the top left of the image). The code in the middle is run to set the first pixel value, then the variable *x* is incremented to become 1 (because *x++* means ‘add 1 to *x*’). This process is repeated so long as *x* is less than the image width, *x < width*. When *x* then equals the width, it means that all pixel values on the first row of the image have been rounded. Then *y* is incremented and *x* is reset to zero, before the process repeats and the next row is rounded as well. This continues until *y* is equal to the image height – at which point the processing is complete¹.

18.3.1 Final code

The final code of my version of the macro is given below:

```
// Variables to change
psfSigma = 2;
backgroundPhotons = 10;
exposureTime = 10;
readStdDev = 5;
```

¹If you are unfamiliar with programming, the syntax of loops may look quite strange. Reading through some online tutorials for the ImageJ macro language or for-loops in Java should help demystify what is happening here.

```
detectorGain = 1;
detectorOffset = 100;
nBits = 8;
maxPhotonEmission = 10;
doBinning = false;

// Ensure image is 32-bit
run("32-bit");

// Normalize the image to the range 0-1
getStatistics(area, mean, min, max);
run("Subtract...", "value="+min);
divisor = max - min;
run("Divide...", "value="+divisor);

// Define the photon emission at the brightest point
run("Multiply...", "value="+maxPhotonEmission);

// Simulate PSF blurring
run("Gaussian Blur...", "sigma="+psfSigma);

// Add background photons
run("Add...", "value="+backgroundPhotons);

// Multiply by the exposure time
run("Multiply...", "value="+exposureTime);

// Simulate photon noise
run("RandomJ Poisson", "mean=1.0 insertion=modulatory");

// Simulate the detector gain
// (note this should really add Poisson noise too!)
run("Multiply...", "value="+detectorGain);

// Simulate binning (optional)
if (doBinning) {
  run("Bin...", "x=2 y=2 bin=Sum");
}

// Simulate the detector offset
run("Add...", "value="+detectorOffset);

// Simulate read noise
run("Add Specified Noise...", "standard="+readStdDev);
```

```

// Clip any negative values
run("Min...", "value=0");

// Clip the maximum values based on the bit-depth
maxVal = pow(2, nBits) - 1;
run("Max...", "value="+maxVal);

// Get the image dimensions
width = getWidth();
height = getHeight();

// Round the pixels to integer values
for (y = 0; y < height; y++) {
  // Loop through all the columns of pixels
  for (x = 0; x < width; x++) {
    // Extract the pixel value at coordinate (x, y)
    value = getPixel(x, y);
    // Round the pixel value to the nearest integer
    value = round(value);
    // Replace the pixel value in the image
    setPixel(x, y, value);
  }
}

// Reset the display range (i.e. image contrast)
resetMinAndMax();

```

18.4 Limitations and uses

Of course, the above macro is based on some assumptions and simplifications. For example, it treats gain as a simple multiplication of the photon counts – but the gain amplification process also involves some randomness, which introduces extra noise. Because this noise behaves statistically quite like photon noise, the effect can be thought of as decreasing the number of photons that were detected. Also, we have treated the background as a constant that is the same everywhere in an image. In practice, the background usually consists primarily of out-of-focus light from other image planes, and so really should change in different parts of the image, particularly in the widefield case.

Nevertheless, quite a lot of factors have been taken into consideration. By exploring different combinations of settings, you can get a feeling for how they affect overall image quality. For example, you could try:

- Increasing the background, while keeping the maximum photon emission the same
- Removing the detector offset, or setting it to a negative value
- Comparing the effects of binning for images with low and high photon counts
- Creating multiple images from the same source data, and then averaging them together to see how the noise is changed

When planning to implement some analysis strategy – particularly if fluorescence intensity measurements are being made – it may also be useful to test its effectiveness using this macro. To do so, you would need to somehow create a ‘perfect’, noise and blur-free example image, either manually or by deconvolving a suitably similar sample image. You can then apply your algorithm to this perfect image to find out what it detects and what conclusions you could draw. Then apply the exact same algorithm to a version of the image that has passed through the simulator, and see how different your measurements and conclusions would be. Ideally, the results should be the same in both cases. If they are different, the comparison gives you some idea of how affected by the imaging process your measurements are, and therefore how reliably they relate to the ‘real’ underlying sample.

Index

- 3D images, *see* multidimensional data
- Airy disk, 135–137
- background
 - estimation, 72–73
 - subtraction, 61, 63, 72–73
- binary images, 101–106
 - connectivity, 75
 - converting to ROIs, 74–76
 - creating by thresholding, 67, 68
 - definition, 67
 - distance transform, 103–106
 - masking regions, 63
 - refining, 102–106
 - representation in ImageJ, 68
 - skeletonization, 103
 - Voronoi, 104
 - watershed, *see* watershed transform
- bit depth, 23–30, 180
 - converting, 28, 41
 - stacks, 110
 - practical issues, 73–74, 84
- blur, *see* point spread function (PSF)
 - by filtering, 81, 92
 - in fluorescence imaging, 86, 129–141
- brightness & contrast, 9–11
 - applying to stacks, 109–110
 - nonlinear, 61
- cameras, *see* detectors
- clipping, 25–27, 70
- colour
 - composite / multichannel, 34–35
 - converting, 36, 37, 41
 - RGB, 35–36, 41
- detectors
 - charged coupled devices (CCDs), 146, 168–171
 - overview, 167
 - photomultiplier tubes (PMTs), 167–168
- distance transform, *see* binary images
- file formats, 39–47
 - bitmap / vector, 43, 46
 - choosing, 43–46
 - compression, 41–43
 - for figures, 44
- filters, 79–96
 - at boundaries, 87
 - edge detection, 84–85, 94, 95
 - for 3D stacks, 110–111
 - Gaussian, 90–96, 137
 - difference of Gaussian, 93–94, 118–123
 - Laplacian of Gaussian, 94–95
 - gradient, 84–85
 - kernels / masks, 81–82
 - linear, 79–87
 - defining, 82–84
 - mean, 80–81, 90, 91
 - median, 88–89, 91

- minimum / closing, 102
 - minimum / maximum, 88–89, 102
 - nonlinear, 88–89
 - opening / closing, 102
 - to remove outliers, 89, 157
 - unsharp mask, 95
- flattening, *see* regions of interest (ROIs)
- fluorescence imaging
 - imprecisions, *see* blur, noise, pixel size
 - overview, 127–130
 - simulating, 175–184
- histograms, 7, 69–70, 74, 113
- hyperstacks, *see* multidimensional data
- image formation, *see* fluorescence imaging
- ImageJ/Fiji
 - introducing, 4
 - referencing, 6
 - tips & tricks, 5
- labeled images, 67, 75, 114
- lookup tables (LUTs), 8–10, 35
- macros
 - writing, 117–124, 175–184
- measurements, 53–57, 74–76
 - choosing, 53, 54
 - effects of blur, 131, 140–141
 - for 3D stacks, 112–114
 - of specified regions, *see* regions of interest (ROIs)
 - redirecting to another image, 76
- microscope settings
 - gain & offset, 26
- microscope types
 - laser scanning confocal, 164–165
 - multiphoton, 166
 - spinning disk confocal, 165–166
 - super-resolution, 141
 - Total Internal Reflection Fluorescence (TIRF), 166–167
 - widefield, 164
- morphological operations, *see* binary images, filters
- multichannel images, *see* color
- multidimensional data, 15–20, 109–114
 - correcting dimensions, 16
 - filtering, 110–111
 - measurements, 112–114
 - thresholding, 112
 - viewing, 17–20
 - virtual stacks, 45
- noise, 130, 143–159
 - and thresholding, 71–72, 154–156
 - distributions
 - Gaussian, 147–150
 - Poisson, 150–157
 - reducing
 - binning, 168–171, 179–180
 - by more photons, 157–159
 - filtering, 80, 88–89, 148–150, 171
 - salt and pepper, 88
 - signal-to-noise ratio (SNR), 146–147, 153–154
 - simulating, 63–64, 147–148, 156, 177
 - types
 - dark, 157
 - photon, 144, 150–157, 176
 - read, 144, 146–147, 156–157, 176
- Numerical Aperture (NA), 138–139
- Nyquist sampling, 158
- overlays, *see* regions of interest (ROIs)
- pixel size, 11–13, 54
 - and noise, 158
 - binning, 168–171, 179–180
 - choosing, 158
- pixels
 - definition, 3
- point operations, 59–64

- arithmetic, 59
 - for contrast enhancement, 61
 - image calculator, 63–64, 94
 - inverting, 60
- point spread function (PSF), *see* blur
 - affect on resolution, 139–140
 - as filter, 132–133, 177
 - measuring, 140–141
 - shape, 133–139
- Rayleigh criterion, 139
- regions of interest (ROIs), 54–57
 - drawing, 54
 - flattening, 8, 57
 - modifying, 55
 - overlays, 56
 - ROI manager, 55, 75
 - saving, 56
 - transferring between images, 56
- resolution
 - microscopes & detectors, 163–171
 - spatial, 12, 139–140, 158, 163
 - temporal, 163
- ROI Manager, *see* regions of interest (ROIs)
- saturation, *see* clipping
- stacks, *see* multidimensional data
- thresholding, *see* binary images, labeled images, 67–76
 - and background, 72–73
 - and noise, 71–72
 - automatic, 68–70
 - for 3D stacks, 112
 - manual, 68
 - using histogram, 69–70, 74
- type
 - of an image, *see* bit depth
 - floating point, 24
 - unsigned integer, 23
- watershed transform, 105–106
- z-projections, 18, 113