3D Quantitative Colocalisation Analysis

Bioimage Analysis Series

Fabrice P. Cordelières and Chong Zhang

3.1	Introduction – 34
3.1.1	What Is Colocalisation? – 34
3.1.2	Which Colocalisation Methods Are There? – 34
3.1.3	Some Image Preprocessing Tips You Should Keep in Mind – 36
3.2	Datasets – 37
3.3	Tools – 39
3.4	Workflow 1: Objects Overlap Volume Quantification – 39
3.4.1	Step 0: Building a Strategy – 40
3.4.2	Step 1: Normalize the Image Names – 41
3.4.3	Step 2: Tag the Objects – 43
3.4.4	Step 3: Isolating the Overlapping Parts – 47
3.4.5	Step 4: Retrieve Volumes – 49
3.4.6	Step 5: Generate Outputs – 53
3.4.7	Step 6: Make the Macro User Friendly – 57
3.4.8	What Then? – 58
3.5	Workflow 2: Objects Overlap Intensity Quantification – 59
3.5.1	What Should We Do? – 59
3.5.2	New Step 4: Retrieve Intensities – 60
3.5.3	Adapted Step 6: Make the Macro User Friendly – 62
	Bibliography – 66

What You Learn from This Chapter

In this module we will first build a 3D object based colocalisation macro step by step. Then we will practice to adapt and extend the current macro such that it can also work with intensity-based colocalisation methods.

3.1 Introduction

3.1.1 What Is Colocalisation?

Subcellular structures interact in numerous ways, which depend on spatial proximity or spatial correlations between the interacting structures. Colocalisation analysis aims at finding such correlations, providing hints of potential interactions. If the structures only have simple spatial overlap with one another, it is called *co-occurrence*; If they not only overlap but also co-distribute in proportion, it is then *correlation*.

Two proteins are said to be colocalised when the locations of their associated signals (fluorescence) are indistinguishable by the imaging system used, i.e. the distance between signal is below the resolution of the imaging system. For example, in [Fig. 3.1](#), the seemingly one pair of colocalised objects in low resolution images are actually several pairs of smaller objects in close proximity. Therefore, the diagnosis placed for colocalisation should always be stated relative to a particular resolution and sampling rate. In other words, conclusions that could be drawn from a colocalisation study are:

- **In cell biology:** the two proteins are at the same location;
- **In statistics:** considering the current resolution, it might not be excluded that the two proteins are indeed at the same location.

3.1.2 Which Colocalisation Methods Are There?

In general, when we have a specific application for colocalisation analysis, a few questions should be asked first, and depending on the answers to them, one or more methods should be applied. Here in [Fig. 3.2](#) is our recipe for which method(s) are appropriate. Specifically, colocalisation may be evaluated visually, quantitatively, and statistically:

- It may be identified by superimposing two images and inspecting the appearance of the combined color. For example, colocalisation of red and green structures can appear yellow. However, this intuitive method can work only when the intensity levels of the two images are similar (see a detailed example in Dunn et al. (2011)). Scatter

Fig. 3.1 (left to right) The same objects imaged with varying resolutions, expressed as a fraction of the highest resolution: 1/16, 1/8, 1/2, 1, and a zoom-in view (white frame). The white pixel in the bottom right corner depicts the sampling rate (pixel size) adapted to each resolution. The seemingly one pair of colocalised objects are actually several pairs, as shown in images with higher resolution

plot of pixel intensities from the two images also qualitatively indicates colocalisation, e.g. the points form a straight line if the two structures correlate. But visual evaluation does not tell the degree of colocalisation, nor if it is true colocalisation at all.

- In general, two categories of quantitative approaches to colocalisation analysis can be found: intensity based correlation methods and object based methods. Intensity based methods compute global measures about colocalisation, using the correlation information of intensities of two channels. Several review papers have been published during the last decade, where coefficients' meaning, interpretation, guide of use, and examples for colocalisation are given (Bolte and Cordelières 2006; Cordelières and Bolte 2014; Dunn et al. 2011; Zinchuk and Zinchuk 2008). Tools for quantifying these measures can be found in many image analysis open-source and commercial software packages, to name just a few: Fiji's JACoP plugin and Coloc 2, CellProfiler, BioImageXD, Huygens, Imaris, Metamorph, Volocity. Most object-based colocalisation methods first segment and identify objects, and then account for objects' inter-distances to analyze possible colocalisation. Usually, two objects are considered colocalised, if the centroids of the objects are within certain distance (Bolte and Cordelières 2006; Cordelières and Bolte 2014; Obara et al. 2013), or if two objects with certain percentage of area/volume overlap (Rizk et al. 2014; Wörz et al. 2010). We will implement some specific methods for both categories in two case studies described in ► Sects. 3.4 and 3.5.
- Colocalisation studies generally should perform some statistical analysis, in order to interpret whether the found co-occurrence or correlation is just a random coincidence or a true colocalisation. A common method is Monte-Carlo simulations Fletcher et al. (2010) but it is computationally expensive. Recently a new analytical statistics method based on Ripley's K function is proposed and included as an Icy plugin, Colocalisation Studio (Lagache et al. 2013).

3.1.3 Some Image Preprocessing Tips You Should Keep in Mind

Talking about colocalisation, we often also think about deconvolution. Careful image restoration by deconvolution removes noise and increases contrast in images, improving the quality of colocalisation analysis results. Noisy images may generate unwanted “matching pixels”: it should therefore be handled with great care. High contrast is always a plus, especially when trying to delineate structures for individual object's colocalisation determination. In Fiji, you can find several plugins for these tasks to try on your images, such as:

- Parallel Iterative Deconvolution (fiji.sc/Parallel_Iterative_Deconvolution), where the point spread function (PSF) can be estimated using the Diffraction PSF 3D plugin (fiji.sc/Diffraction_PSF_3D). An example can be found in ■ Fig. 3.3.
- DeconvolutionLab and DeconvolutionLab2 (Sage et al. 2017).
- To further remove background noise, you can try `[Process -> Subtract Background]` (for our images, the rolling ball radius can be set to 10 pixels)

However, deconvolution is not the focus of this module. Therefore, we would assume that the images to be processed during this module are either already deconvolved or are acquired with high image quality without the need of deconvolution. Other issues that may need to be dealt with during the preprocessing include: illumination correction, noise removal, background or artifacts disturbance. Since we have already been practicing techniques to handle these situations, here we would as well rather not to discuss them.

■ **Fig. 3.3** Example images before (*left*) and after (*middle/right*) two deconvolution algorithms (Richardson & Lucy, 200 iterations, middle; Meinel, 10 iterations, right). Lower row presents a magnification of upper figures, centered on the upper mitotic spindle pole

It is worth noting briefly here that before this step, several points should be taken care of during the image acquisition and collection part.

- To have the imaging hardware set up appropriately. That is, to adjust the exposure time, detector gain and offset so as to be able to detect the dimmest structures without saturating the brightest structures.
- To check for chromatic aberrations one uses small beads that are fluorescent in many colors and thus should 100% co-localize with themselves. If they appear shifted you have to realign your microscope or account for the shift during the analysis.
- To appropriately control bleed-through.

3.2 Datasets

Let's first have a look at the data. The images display a large number of dash-like structures. The life scientist informs us they result from the labeling of two proteins which both locate at the most dynamic end of microtubules: they belong to the +TiPs family (plus-end tracking proteins). When overlaying both images (EB1 on channel 1 and CLIP-170 on channel 2, available in this module's folder: Images/+TiPs.zip and as a smaller version Images/+TiPs_mini.zip), it seems that the overlap is only partial, which justifies the scientist's first impression. Now we have to "put numbers" and try to evaluate this partial colocalisation to either demonstrate it or prove it to be a simple visual artifact (■ Fig. 3.4).

■ **Fig. 3.4** Dataset on which colocalisation will be evaluated (Left: EB1, Middle: CLIP-170, Right: Overlay, Right-most: magnified sub-images)

■ **Fig. 3.5** Don't trust your eyes and brain on colocalisation!

Of course, most of us would rule out this latter hypothesis (■ Fig. 3.5). So let's take an example from [Fiji's website](#): do you think some of the red and green pixels colocalise on this image?

You may try zooming at the image and realize that yellow pixels are in fact resulting from the close proximity between green and red pixels: our brain simply blends one tone into the other. In case you do not believe this is an optical trick, try opening it under ImageJ then moving the cursor over the “yellowish” pixels: the status bar will display the values of the green and red components of the image. As a result, you'll figure out that red pixels are not green and the reverse way round.

Word of advise: in colocalisation studies, as you can't trust your eyes and the brain that lays (or lies) behind, better be ready to build a proper quantification strategy!

In addition to the requirement of a proper quantification strategy, this example also points out the need for a well characterized dataset. When building a workflow,

■ Fig. 3.6 Synthetic 3D dataset from two views

the image analyst might benefit from use of a synthetic dataset. This somewhat barbarian terms designate a computer-generated dataset where all parameters are human controlled. In microscopy, the images result from single captions of a scene, which are impaired by the optics and corrupted by noise. To simplify the prototyping process, we could generate two images, one per channel, containing simple shapes (circles, rectangles etc...) which sizes, and degree of overlap are controlled. Such a dataset, extended to the 3D case is provided. ■ Figure 3.6 shows two 3D views of the synthetic dataset with two channels, where channel 1 (*red*) has six objects and channel 2 (*blue*) seven. Each object in channel 1 has different level of spatial overlap with one of the objects in channel 2. The synthetic dataset can be found in this module's folder (Images/Synthetic.zip).

3.3 Tools

- Fiji
 - Download URL: ► <https://imagej.net/Fiji/Downloads>

3.4 Workflow 1: Objects Overlap Volume Quantification

Let's imagine a typical conversation between an image analyst and a life scientist who has a colocalisation analysis request for the dataset to be reviewed under ► Sect. 3.2:

- » I've got a set of two-channel 3D images where objects are overlapping. I think the overlap might not be the same from object to object.

Therefore, I would like to quantify the overlap and get a map of quantification.

A user comes to the Facility, asking for help

3.4.1 Step 0: Building a Strategy

So how would we tackle the problem?

The key words here are: objects, overlapping, map of quantification. So the first task is to identify objects of interest in the image, i.e. segmentation. Then, overlapping regions between objects in the two channels should be extracted. In order to obtain a map of quantification on overlap, one of the quantification metrics could be calculating the volume of objects and overlapping regions. Figure 3.7 and Table 3.1 summarizes the steps that need to be implemented in the macro.

Fig. 3.7 A schematic drawing of the full workflow: the first task is to identify objects of interest in the image, i.e. segmentation. Then, overlapping regions between objects in the two channels should be extracted. In order to obtain a map of quantification on overlap, one of the quantification metrics could be calculating the volume of objects and overlapping regions

Table 3.1 What needs to be done?		
Step	What needs to be done?	What for?
1	Normalize the data name	Split & rename channels
2	Tag the objects	Segment & label 3D objects
3	Isolate the overlapping parts	Segment overlapping regions
4	Retrieve volumes	Calculate & store object volumes
5	Generate outputs	Display & visualize volume ratio
6	Make the macro user friendly	Create user interface

Since most of the operations will be applied the same way on both channels, or on different objects/regions, the most efficient scripting strategy would be to define `functions` that consist of several macro commands such that a specific operation is performed. And these functions could be simply called by the function name plus arguments as many times as needed in the macro, instead of repeating the macro command lines. This way improves the readability of the macro thus the re-usability as well.

3.4.2 Step 1: Normalize the Image Names

Aim:

- The channels should be split so that images are processed independently.
- The first channel should be named as “Channel1”, the second “Channel2”.
- Generate a user-defined function to perform the task: `'normaliseNames'` with proper arguments, if needed.

When automating an image processing workflow, a major challenge is to make the macro re-usable and independent about the input images' naming convention. We will be facing an additional complication as the image we are working on is a composite, made of several channels. Processing will be applied to each single channel independently, once they will have been splitted. A proper strategy should be designed to keep track of the original image's title (ie, its name) and the subsequent channels' image obtained after separating them. The following section will handle this step in a simple way by renaming the original and the subsequent images with pre-defined names. This step could be seen as “normalizing the images' names” for better handling.

We will use the function [Plugins > Macros > Record] to track the sequential operations that have been applied; clean up and convert the recorded macro commands to our own defined function in the macro file. Some of the functions, for example to retrieve an image's title, are not recordable. In such case, making a simple word-based search on [ImageJ's Built-in Macro Functions page might help](#). `getTitle( )` function returns the title of the active image as a String that can be stored into a variable. Let's call it `'ori'` as it is used to store the name of the “original” image.

The original image is a composite. For the kind of processing we are planning to do, each channel should be handled separately. Channels are splitted by using the function [Image > Color > Split channels] that is macro recordable. ImageJ/Fiji applies specific naming rules to each image when splitting the channels: the resulting titles are built by concatenating the letter “C” (for channel), the number of the channel (starting at 1), a dash, and finally the original image's title. We therefore have a way to precisely activate the first or the second channel. However, to make the process a bit easier, we rename each channel in the form of “ChannelXX”.

Finally, once all operations have been recorded, and the code cleaned up, the corresponding lines can be encapsulated into a function. To generate a function, you simply need to append the keyword “function” with a proper name, a pair of empty parentheses (or contain required parameters) and a pair of curly brackets: the few lines of code we've just written should be copied/pasted in between.

To summarize, here is your working plan to implement the code:

Working Plan:

1. Split the image into its channels
2. Select the first channel
3. Rename the image "Channel1"
4. Select the second channel
5. Rename the image "Channel2"
6. Pack everything into a function, thinking about the proper arguments, if any, that should be entered for the process to be run

The correct code of the function is:

```

1 //Split the channels and rename them in a normalised way
2 function normaliseNames() {
3   ori= getTitle ();
4   run ("Split Channels");
5   selectWindow ("C1-"+ori);
6   rename ("Channel1");
7   selectWindow ("C2-"+ori);
8   rename ("Channel2");
9 }
```

Now, in order to run the macro properly with this function, we still need a line in the main code body of the macro to call it. For this particular step, it is straightforward to call the function, as follows:

■ ■ The main macro till Step 1

```

1 //-----
2 // Main macro part
3
4 //Step 1
5 normaliseNames();
6
7 //-----
8 }
```

Something that is as important as writing the macro itself is to comment in detail each step you do. This way, it not only helps to remind what the macro does, but also helps to reuse and adapt existing macros. The latter is much better practice than always writing things from scratch whenever you need to do some analysis. You will see later why it is so.

3.4.3 Step 2: Tag the Objects

Aim:

- For each channel, have a tagged map: all pixels from an object should carry a same tag/intensity.
- The tagging should be made in 3D.
- Generate a user-defined function to perform the task: `''isolateObjects''` with proper arguments, if needed.

Hints:

- We need to differentiate background pixels from objects' pixels: thresholding is one way to achieve this.
- ImageJ built-in functions are not the only ones being recordable.

Well, the aim being explicit, one can already see this part of the workflow is not specific to a single channels: both channels will go through this process. This is therefore a good candidate for building a function, that will be called twice to process each of them, and name as “isolateObjects”. The remaining question is how the process would differ from one image to another?

First, we have to define what an object is. Based on the image, we could define an object as a group of contiguous pixels, for which each pixel has an intensity above a certain threshold value.

The latter step is easy to achieve, as long as the background (i.e. the non-object voxels) is distributed within a well defined range of intensities. The function to be used lies in the [Image > Adjust > Threshold...] menu. This function is macro recordable: the recorder should contain the `run('Threshold...')` instruction. Note, however, that this function is commented. Un-commenting and running it won't do much: this function is only aimed at displaying the dialog box, not to perform any processing. However, it is useful, for instance, to pop-up the window before asking a user to set the threshold values. Once more, we will have to refer to [ImageJ's Built-in Macro Functions page](#) and look at threshold-related instructions. For any kind of process, two types of functions might be found: setters, to apply parameters and getters to retrieve them. In the specific case of thresholding, the two types exist, in the form of `setThreshold(lower, upper)` and `getThreshold(lower, upper)`. This information will show its use later on in this module.

The first step, grouping adjacent voxels into objects might seem to be a bit tricky. This is however the kind of processing that goes behind the scenes when using the [Process > Analyze > Particles...] function. But how does this work ? First, a destination image is created: we name it as “tagged map” (see [Fig. 3.8](#)). Each thresholded pixel from the original image is screened in turn, from the top-left corner to the bottom-right corner, line by line. The first thresholded pixel is attributed a tag, i.e. the corresponding pixel

Fig. 3.8 A schematic drawing of converting the segmented object mask into a “tagged map” of these objects, i.e. all pixels that are connected are labeled as the same color or pixel intensity; and those not connected are distinguished by a different color or pixel intensity. There are different configurations of “connected”, e.g. 4 or 8 connected in 2D

on the “tagged map” will be given a value of “1”. Next pixel (to the right) is examined: if thresholded, a tag will be given. As its direct, left neighbor, as already been given a tag of “1”, it receives the same tag: both are part of the same object. This process goes on for the entire image. In case a new thresholded pixel is found, that doesn’t have a tagged neighbor, a new tag is created: this pixel is a seed for a new object. Please note that several rules might be applied to define pixels’ vicinity: a pixel can be connected only through its sides (4-connected in 2D) or one might also consider its diagonal corners as contact points (8-connected in 2D). Depending on which rule is used, the number of objects might not be the same.

Although being convenient as an ImageJ built-in tool, the “Analyze Particles” function only works in 2D: an object spanning over several slices might end-up being counted several times. In 3D, another tool should be used: 3D-Object counter (aka **3D-OC**), which works with the 26-connected voxels rule. As it is fully macro-recordable, its usage from within a macro is straightforward. If using Fiji, the plugin is already installed, if not, you’ll have to download it from [this link](#). Note that when using it from a macro: the Fiji team has changed the name of its menu entries as compared to the original author’s version: a macro recorded under Fiji might need few changes to run under ImageJ and vice-versa.

We are now familiar with all the required components for isolating objects. Let’s do manually all the steps of the workflow, keeping an eye on the macro-recorder. We first need to select the proper window, then launch the threshold adapter and feed the lower threshold to the 3D-OC plugin, and finally ask it to display the objects map (i.e. tagged map).¹

The implementation is straightforward: as user input is required, the threshold box is first activated. To preset a threshold, assuming fluorescence images, one could press the Auto button: the following instruction will be recorded: (`setAutoThreshold ("Default dark") ;`). Now, we must find a way to display a message box inviting the user to

¹ In case you see numbers overlaid onto the tagged map, go to the “Set 3D Measurements” menu and uncheck the “Show numbers” option.

finely tune the threshold. In the meantime, the macro's execution should be idled until proper values are set. Such an instruction exists, once more, please refer to [ImageJ's Built-in Macro Functions page](#) to find the proper one. This is the purpose of the `waitForUser("text")` function. Now the values have been set, we should capture them into variables: this way, user-entered values could be recalled for subsequent steps of the workflow. In the final step of this instructions block, the threshold values are retrieved using the `getThreshold(lower, upper)` function. Note that this function expects two variables to be provided as arguments. The two variables will be filled with the content of the adjustment box, the former with lower bound value, the latter with the upper bound value.

Often the segmentation contains objects that are not interesting for us such as noise or other structures. Since object-based methods concern individual objects, then we should apply some filtering criteria in order to discard them for further analysis. Such criteria could be, for example:

1. (3D/2D/1D) size range of the object-of-interest (in each channel)
2. object shape, e.g. circularity,² compactness³
3. object location

It should be mentioned that this step greatly influences the colocalisation measurements. We will discuss only size related filtering here. `3D Objects Counter` is able to do this. Then let's select one channel, e.g. Channel 1, to record all the needed operations to convert into the second function, `isolateObjects`. When running [`Analyze > 3D Object Counter`], in the pop-up window there are three parameters of our interest: `Min` and `Max` in the `Size filter` field, and the `threshold`. We could pass along the variable that stores the user specified threshold value in Step 1, i.e. `lower`. Let's suppose that the object of interest should have a size of minimum 5 voxels and maximum 100,000 voxels. This filtering step removes the smallest object from the image. Although they may seem to overlap with objects in the other channel, they are likely to be e.g. noise and their spatial co-occurrence could be coming from randomly distributed particles/noises that are close to each other by chance. Before applying the `3D Objects Counter`, we should check the measurements setting in the [`Analyze -> 3D OC Options`] options window. This is similar to [`Analyze -> Set Measurements`] for the `Analyze Particles` function.

After the `3D Objects Counter`, we will also obtain a “tagged map” of objects in the image (■ Fig. 3.8). This “tagged map” is an image with different objects labeled with different intensity values (1 for the first object, 2 for the second, etc...). When thinking about it, you will find that this nice image contains a lot of information! How could we take advantage of it so as to isolate a particular object? How could we extract the number of voxels per object? Take time to think about it, before reading further. Isolating an object from a tagged map is quite simple as all its voxels have the same intensity: simply threshold the image, using its tag value as the lower and upper thresholds. As for each object's number of voxels, a histogram operation [`Analyze -> Histogram`] should do the trick! Let's keep this in mind for the next step.

-
2. Circularity measures how round, or circular-shape like, the object is. In Fiji, the range of this parameter is between 0 and 1. The more roundish the object, the closer to 1 the circularity.
 3. Compactness is a property that measures how bounded all points in the object are, e.g. within some fixed distance of each other, surface-area to volume ratio. In Fiji, we can find such measurements options from the downloadable plugin in [`Plugins > 3D > 3D Manager Options`].

Now that you have all the elements needed, here is your working plan to implement the function:

Working Plan:

1. Select the proper image
2. Display the threshold box
3. Pause the execution of the macro by displaying a dialog box asking the user to tune the threshold
4. Retrieve the threshold values
5. Make sure the proper options for 3D measurements are set
6. Run 3D-OC, using the input threshold and some size limits
7. Pack everything into a function, thinking about the proper arguments, if any, that should be entered for the process to be run

The correct code of the function is:

```

1 //Isolate the objects and get the characteristics on each image
2 function isolateObjects(minSize, image){
3   selectWindow (image);
4   run ("Threshold...");
5   waitForUser ("Adjust the threshold then press Ok");
6   getThreshold (lower, upper);
7   run ("Set 3D Measurements", "dots_size=5 font_size=10 redirect_
   to=none");
8   run ("3D object counter...", "threshold="+lower+ "slice=5 min.=
   "+minSize+ " max.=100000 objects");
9 }

```

Again, in order to run the macro properly with this function, we need to call it in the main code body of the macro. For this step, since we need to tag objects in both channels, thus the function will be called twice. The advantage of creating user-defined function is nicely visible here: we won't have to re-type all the code from channel 1 to use it on channel 2. Functions should be called after Step 1 (3.4.2) is done, as shown in Code 3.2.

■ ■ The main macro till Step 2

```

1 //-----
2 // Main macro part
3
4 //Step 1
5 normaliseNames();
6
7 //Step 2
8 isolateObjects(10, "Channel1");
9 isolateObjects(10, "Channel2");
10
11 //-----
12 }

```

3.4.4 Step 3: Isolating the Overlapping Parts

Aim:

- Isolate object parts that are overlapping.
- Generate user-defined functions to perform the task: `getCommonParts` and `maskFromObjects` with proper arguments, if needed.

Hints:

- We have already defined objects in the previous step.
- On the tagged maps, background is tagged as 0.
- On the tagged maps, the first object is tagged with an intensity of 1, the second of 2, and so on.
- Logical operations could be applied to binary masks.

Since the `3D Object Counter` gives a tagged map, a simple thresholding could convert it to a binary image with background being 0, since object tags start at 1. As we will still need the tagged maps later on in the process, we will first duplicate them, and work on copies (`[Image > Duplicate]`). In `ImageJ`, the binary image after thresholding has value 0 (non-object pixels) and 255 (object pixels). Sometimes, a binary image of value 0 and 1 makes further analysis easier. Let's see an example: you want to measure a "positive" volume within a certain region of interest. Instead of thresholding and measuring the area within this ROI, in case the image has intensities being either 0 or 1, you can simply measure the sum of intensities within the ROI to obtain its area! To perform this, we could divide every pixel by this image's non-zero value, i.e. 255, using `[Process > Math > Divide]`. These would be the steps needed in the function `maskFromObjects`.

Once these steps have been applied to both tagged map, we end up with two new masks displaying pixel objects from `channel1` and `channel2`. How to isolate the overlapping parts of objects between both channels? One should generate a new image, where pixels are turned "ON" when both the corresponding pixels on the two channels are also "ON": this is a job for logical operators that we will now review.

In Fig. 3.9, there are two binary images A and B. We could see that in both channels, the overlapping regions have values higher than zero in both channels; while the rest of the two images have either one or both channels with zero background. Therefore if we multiply the two images, only the overlapping regions will show values higher than zero. Alternatively, we could also apply logic operations, which can be calculated faster than multiplication for computers. From the three logic operations: AND, OR and XOR, which is the one that we need? It is AND. So we can run `[Process > Image Calculator]`, set the object maps of from the two channels as `Mask_Channel1` and `Mask_Channel2`, and set AND as Operation. We then rename the image with overlapping regions as "Common_volumes". These steps would go to the function `getCommonParts`. Think about where we should call `maskFromObjects`?

Fig. 3.9 Illustrations of applying logic operations on two binary images A and B

Working Plan:

- *Part 1: Convert an object map into a mask scaled between 0 and 1*
 1. Select the tagged map
 2. Duplicate it, giving it a name in the form "Mask_" + original name (be careful: we want the full stack, not just the current slice)
 3. Set the threshold between 1 and the maximum (65535)
 4. Convert the thresholded image to a mask
 5. Normalize the intensities between 0 and 1 (divide by 255)
 6. Pack everything into a function, thinking about the proper arguments, if any, that should be entered for the process to be run
- *Part 2: Isolate common parts from both images*
 1. Generate the normalized mask for channel 1 (i.e. 0–1 scaled)
 2. Generate the normalized mask for channel 2 (i.e. 0–1 scaled)
 3. Use logical operators between both mask to retrieve the overlapping voxels
 4. Pack everything into a function, thinking about the proper arguments, if any, that should be entered for the process to be run

The correct code of the two functions are:

```

1 //Generate an image of the overlapped parts from channel 1 and 2
2 function getCommonParts(){
3 //Generate the mask for channel 1
4 maskFromObjects("Channel1");

```

```

5 //Generate the mask for channel 2
6 maskFromObjects("Channel2");
7
8 //Combine the two masks
9 imageCalculator ("AND create stack", "Mask_Channel1", "Mask_Channel2");
10 rename ("Common_volumes");
11 }

```

```

1 //Generate a mask from objects map
2 function maskFromObjects(image){
3 selectWindow ("Tagged_map_"+image);
4 run ("Duplicate...", "title=Mask_"+image+ " duplicate");
5 setThreshold (1, 65535);
6 run ("Convert to Mask", "method=Default background=Dark");
7 run ("Divide...", "value=255 stack");
8 resetMinAndMax ();
9 }

```

And, the main code body of the macro is:

■ ■ The main macro till Step 3

```

1 //-----
2 // Main macro part
3
4 //Step 1
5 normaliseNames();
6
7 //Step 2
8 isolateObjects(10, "Channel1");
9 isolateObjects(10, "Channel2");
10
11 //Step 3
12 getCommonParts();
13 //-----
14 }

```

3.4.5 Step 4: Retrieve Volumes

Aim:

- Measure the volumes, object by object on: Mask_Channel1, Mask_Channel2 and Common_volumes.
- Store the volumes into arrays.
- Generate a user-defined function to perform the task: `getValues` with proper arguments, if needed.

Hints:

- On the tagged map, the first object is tagged with an intensity of 1, the second of 2...: the maximum intensity therefore corresponds to the number of objects in the tagged map.
- `Thresholding with 1-1` and then `Analyze Particles` allow sending the outlines of object 1 in the `ROI Manager`.
- A non-recordable macro function exists to retrieve basic image statistics: maximum intensity, the number of pixels or the area of a ROI.

In order to find and quantify colocalised objects, we have found the overlapping (or shared) parts of the two filtered channels. We now need to identify the corresponding objects in each channel that contain these overlapping regions. To achieve this, what do we need to do? There are multiple ways, all of which involve:

1. In each channel, calculate the volume (in voxels) of each object;
2. Retrieve the volume of each overlapping region;
3. Find the labels of objects in each channel that overlap with some object in the other channel.

We will have to store multiple values related to each individual object for each channel. That won't be feasible using regular variables. Therefore we have to switch to a different structure that allows to store several values—an array. Here are some tips about how they work and how they should be used.

Technical Points: Using arrays to store multiple values

- An array is like a box, with a tag and multiple compartments
- An array should be initialized using the `newArray` keyword: either with no content but a size, e.g. `myArray=newArray(3)` ; or with a content, e.g. `myArray=newArray(1, 50, 3)` ;
- To attribute a content to an array, the "=" sign should be used between the compartment address between "[" and the content, e.g. `myArray[1]=25` ;
- The size of an array can be retrieved by concatenating `.length` to the name of the array, e.g. `myVariable=myArray.length` ;

Now we know what is it to store data, we should think about how to retrieve them. What we want to do is to define for each object and for each channel the total volume of the object, and the volume of the object that is involved in the colocalisation process. How should we proceed? Let's think about a strategy, keeping in mind the following four important points:

1. On the tagged map image, each object is identified by its intensity (i.e. the tag): a simple thresholding from tag to tag as upper and lower threshold values allows isolating it.
2. `Analyze particles` allows exporting all object's outline to the `ROI Manager`. NB: as this function processes stacks slice-per-slice, ROIs are generated per z-plane? Depending on its spread along the z-axis, a 3D object might therefore be described by multiple 2D ROIs.

3. On the mask, each object pixel has an intensity of 1.
4. Measuring the integrated intensity within a ROI on the mask is the same as measuring the “positive” area.

Why do we decide to design such a procedure ? Let's think about what we need to retrieve. We will have to get and store the volume of all objects for channel 1 and channel 2, and the volumes involved in colocalisation for objects of both channels. All four relies on a common principle: defining the objects on a tagged map, and estimating the volume from a mask where positive pixels are labeled with a value of 1. Therefore, all four measurements can be retrieved by applying the same procedure to four different combinations of images: as the procedure is generic, building a single function is appropriate. We will call it four times, with different arguments.

Here are some technical points which should be useful for the implementation of the workflow.

Technical Points: ROI Manager-related functions

Common structure: `roiManager("function", "argument1", "argument2");`

- Some functions are recordable:
 - Add: push the active ROI to the ROI Manager
 - Select: select the i-th ROI (numbered from 0)
 - Rename: select the i-th ROI (numbered from 0)
- Some are not:
 - Reset: empty the ROI Manager
 - Count: returns the number of ROI within the ROI Manager
 - Set color: self-explanatory

To help you in this task, here is the working plan you should have come up with. Afterwards, it is always good to start writing the planned steps as comments in the macro file, and then fill with corresponding real code that you recorded and modified.

Working Plan:

1. Select the tagged map
2. Retrieve the total number of objects. Think: what is the tag of the last detected object? How to retrieve statistics from a stack ?
3. Create an array to store the volume of objects. Think: what should be the size of this array ?
4. Loop the following for each object:
 - (a) Select the tagged map
 - (b) Set the threshold to highlight only one object
 - (c) Empty the ROI Manager
 - (d) Run the `Analyze Particles` function
 - (e) Initialize a temporary variable to store current object's volume
 - (f) Loop for every found ROI:

- i. Activate the image where the quantification will be done
 - ii. Activate the proper ROI
 - iii. Retrieve the region's statistics
 - iv. Modify the temporary variable accordingly
- (g) Push the temporary variable's content to the corresponding array compartment
- 5. Pack everything into a function, thinking about the proper arguments, if any, that should be entered for the process to be run, and the output that should be made by the "return" statement

Here is the correct code for the function `getValues`:

```

1  //Retrieve volumes object per object
2  function getValues(objectsMap, imageToQuantify){
3  //Activate objects' map
4  selectWindow (objectsMap);
5
6  //Get and store the number of objects
7  getStatistics (area, mean, min, nObjects, std, histogram);
8
9  //Create an output array, properly dimensioned
10 measures= newArray (nObjects);
11
12 //For each object
13 for (i=1; i<=nObjects; i++){
14     //Activate the objects' map
15     selectWindow (objectsMap);
16
17     //Set the threshold to select the current object
18     setThreshold (i, i);
19
20     //Empty the ROI Manager
21     roiManager ("Reset");
22
23     //Run analyze particles, add outlines to ROI Manager
24     run ("Analyze Particles...", "add stack");
25
26     //Create a variable to store the volume and initialise it to zero
27     singleMeasure=0;
28
29     //For each outline
30     for (j=0; j<roiManager("Count"); j++){
31         //Activate the image on which to measure
32         selectWindow (imageToQuantify);
33
34         //Select the ROI
35         roiManager ("Select", j);
36
37         //Measure the volume
38         getStatistics (area, mean, min, max, std, histogram);

```

```

39
40     //Add the volume to the variable
41     singleMeasure+=area*mean;
42 } //End for each outline
43
44 //Push the measure to the output array
45 measures[i-1]=singleMeasure;
46
47 } //End for each object
48
49 //Return the output array
50 return measures;
51 }

```

And it will be called four times in the main macro code to calculate values for both channels:

■ ■ The main macro till Step 4

```

1  //-----
2  // Main macro part
3
4  //...
5
6  //Step 4
7  objectsVolume1=getValues("Tagged_map_Channel1", "Mask_Channel1");
8  commonVolume1=getValues("Tagged_map_Channel1", "Common_volumes");
9
10 objectsVolume2=getValues("Tagged_map_Channel2", "Mask_Channel2");
11 commonVolume2=getValues("Tagged_map_Channel2", "Common_volumes");
12 //-----
13 }

```

To avoid duplicates of same code, in code 3.4 line 4 represents the part in Code 3.3.

3.4.6 Step 5: Generate Outputs

Aim:

- With the measure for each channel, calculate the ratio of volume involved in colocalisation.
- Display the results in a `resultsTable`, one row per object.
- Build a map where the intensity of object corresponds to the ratio of overlap.
- Generate a user-defined function to perform the task: `generateOutputs` with proper arguments, if needed.

Hints:

- On the tagged map, the first object is tagged with an intensity of 1, the second of 2... the maximum intensity therefore corresponds to the number of objects.
- Caution: the ratios are decimal values, intensities on all images we are working on so far are integers.
- In ImageJ, [Process/Macro], there is a way to replace an intensity by another value.

Numbers have been extracted: well done! But still, the user can't see that: we need to create some outputs. Two ideas come up: should I display a table with all numbers? Or is the user rather of a visual kind and would require to have the values mapped to an image? Ok, let's not make a decision, but do both!

We will first generate a table per channel, with one row per object and three columns containing the volume of the object, the volume of this object involved in colocalisation (i.e. overlapping) and the ratio. Before starting coding this output, we should have a look at the possibilities of table output offered by ImageJ. This is the topic we will cover in the next technical point.

Technical Points: Using a ResultsTable to output data

- [Analyze/Clear Results]: empties any existing ResultsTable. This function is macro-recordable.
- nResults: predefined variable to retrieve the number of rows.
- setResult("Column", row, value): Adds an entry to the ImageJ results table or modifies an existing entry. The first argument specifies a column in the table. If the specified column does not exist, it is then added. The second argument specifies the row, where $0 \leq \text{row} \leq \text{nResults}$.
- To add a row in the table, simply address the last row+1: as the rows are numbered 0 to nResults-1, address row nResults.
- To add a column, one simply needs to give it a name.
- The rows might be labeled using "Label" as column name.

When reading the technical points, it seems that the output as a table is not that complicated. Do not forget to tag each line with a note about which object it is about! One additional trick: as two channels are analyzed, each of which should reside in a table, but only one table would be manipulated at a time. To solve this, once more, [ImageJ's Built-in Macro Functions page might help](#): try to find a function that could rename a ResultsTable.

Let's deal with the output as a colocalisation map. First, we need a container to push the ratios into. The simplest way is to duplicate an already existing image. Let's take the tagged map and make a duplication. The resulting image is a 16-bits type image. This implies that only integer intensities can be stored in it. As you may guess, ratios are decimal values:

if we push them directly, the values will be clipped and we are running into the risk of obtaining a black image! Image type conversion is therefore required and should be performed using the [Image > Type > 32-bit] menu. Finally, how and where to put the ratio? The easiest way would be to identify the object by its tag, then replace all the values of its pixels by the ratio. This is a straightforward process when knowing about the [Process > Batch > Macro] function. This menu allows applying some logic to images. In our case, we will have to enter something like `if (v=tag) v=ratio;` (to be adapted with the proper values).

We now have all the elements to build the function for this part of the workflow, let's have a look at the working plan.

Working Plan:

1. Make some clean-up! Empty any existing `ResultsTable`
2. Activate the tagged map
3. Remove any existing ROI
4. Duplicate this image to generate the future colocalisation map
5. Properly convert the image
6. Loop for all objects:
 - (a) Calculate the volume ratio
 - (b) Push a proper label to the `ResultsTable`
 - (c) Push the object's volume to the `ResultsTable`
 - (d) Push the object's volume involved in the colocalisation to the `ResultsTable`
 - (e) Push colocalisation ratio to the `ResultsTable`
 - (f) Activate the colocalisation map
 - (g) Replace the the intensity of any pixel carrying the current object's tag by the ratio
7. Pack everything into a function, thinking about the proper arguments, if any, that should be entered for the process to be run

Now that you have implemented your own version of the code, you may compare it to the functions we have implemented.

```

1  //Generates two types of outputs: a results table and 2 co-
   localisation maps
2  function generateOutputs(objectsMeasures, commonMeasures, objects-
   Map){
3  //Empties any pre-existing results table
4  run ("Clear Results");
5
6  //Duplicate the objects map
7  selectWindow (objectsMap);
8  run ("Select None"); //Needed to remove any ROI from the image
9  run ("Duplicate...", "title=Coloc_Map duplicate");
10 run ("32-bit"); //Needed to accomodate decimal intensities
11

```

```

12 for (i=0; i<objectsMeasures.length; i++){
13     //Calculate the ratio
14     ratio=commonMeasures[i]/objectsMeasures[i];
15
16     //Fill the results table with data
17     setResult ("Label", nResults, "Object_"+(i+1));
18     setResult ("Full object", nResults-1, objectsMeasures[i]);
19     setResult ("Common part", nResults-1, commonMeasures[i]);
20     setResult ("Ratio", nResults-1, ratio);
21
22     //Replace each object's tag by the corresponding colocalisation ratio
23     selectWindow ("Coloc_Map");
24     run ("Macro...", "code=[if (v==" + (i+1) + ") v=" + ratio + "] stack");
25 }
26 resetMinAndMax();
27 }

```

Once more, to test this new function, some lines should be added to the main body of our macro. An example is given hereafter.

■ ■ My main macro till Step 5

```

1  //-----
2  // Main macro part
3
4  //...
5
6  //Step 4
7  objectsVolume1=getValues("Tagged_map_Channel1", "Mask_Channel1");
8  commonVolume1=getValues("Tagged_map_Channel1", "Common_volumes");
9
10 objectsVolume2=getValues("Tagged_map_Channel2", "Mask_Channel2");
11 commonVolume2=getValues("Tagged_map_Channel2", "Common_volumes");
12
13 //Step 5
14 generateOutputs(objectsVolume1, commonVolume1, "Tagged_map_Channel1");
15 IJ.renameResults("Volume_colocalisation_Channel1");
16 selectWindow ("Coloc_Map");
17 rename ("Volume_colocalisation_Channel1");
18
19 generateOutputs(objectsVolume2, commonVolume2, "Tagged_map_Channel2");
20 IJ.renameResults("Volume_colocalisation_Channel2");
21 selectWindow ("Coloc_Map");
22 rename ("Volume_colocalisation_Channel2");
23 //-----

```

3.4.7 Step 6: Make the Macro User Friendly

Aim:

- A graphical user interface should be displayed first, to ask the user for the parameters to use for analysis.
- This step should be implemented as a function: GUI with the proper set of argument(s), if needed.
- The function should return the entered values as an array.

Hints:

- Identify which parameters are there
- Use previous technical points.

During step 1 (► Sect. 3.4.2), we have had a glimpse at user interactions: we used the `wait-ForUser` statement to pause the execution of the macro, and to ask for a user's intervention. There is another possible interaction: Graphical User Interface (GUI). GUI are dialog boxes, which can be fed with parameters. Our first task is to review all parameters that are user-defined, then to build a proper GUI. When looking at the workflow, we can identify two such parameters: the minimum and maximum expected sizes of objects to be isolated from both channels' images. We will build a basic GUI, asking for those two parameters. We will need to learn about the instructions to be used, detailed in the next technical points.

Technical Points: Generating Graphical User Interface

- Initialise a new GUI: use `Dialog.create('Title')`
- Add content to the GUI, where content could be number, string, checkboxes... e.g.: `Dialog.addNumber(label, default)` adds a numerical field to the dialog, using the specified label and default value.
- Display the GUI: `Dialog.show()`
- Retrieve the values in order: one instruction that retrieves the first number, then with a new call, the second...e.g.: `Dialog.getNumber()` returns the contents of the next numeric field.

Based on the above information, the building steps are quite simple:

Working Plan:

1. Create a new Dialog Box
2. Add a numeric field to retrieve the minimum expected size of objects
3. Add a numeric field to retrieve the maximum expected size of objects

4. Display the Dialog Box
5. Create a properly sized array to store the input data
6. Fill the array with retrieved data
7. Pack everything into a function, thinking about the proper arguments, if any, that should be entered for the process to be run, and to output that should be made by the “return” statement

Now it is your turn to do some coding! Try to go ahead yourself first, before looking at our version of the implementation below!

```

1 //Display the graphical user interface
2 function GUI(){
3   Dialog.create ("colocalisation");
4   Dialog.addNumber ("Minimum size of objects on channel1 (in vox-
   els)", 10);
5   Dialog.addNumber ("Minimum size of objects on channel2 (in vox-
   els)", 10);
6   Dialog.show ();
7
8   out= newArray (2);
9   out[0]= Dialog.getNumber ();
10  out[1]= Dialog.getNumber ();
11
12  return out;
13 }
```

We will revisit Code 3.1: as parameters are now stored in an array after applying our own defined function GUI, we need to use them in the function calls.

```

1  //-----
2  // Main macro part
3
4  //Step 6
5  parameters=GUI();
6
7  //Step 1
8  normaliseNames();
9
10 isolateObjects(parameters[0], "Channel1");
11 isolateObjects(parameters[1], "Channel2");
```

3.4.8 What Then?

What then? First, lay done, have a nice cup of tea, and get ready for a review of what you’ve achieved so far! The macro works nicely now and you’ve achieved all aims we’ve fixed on [Table 3.1](#). Here is an update ([Table 3.2](#)):

Table 3.2 What have we learned so far?

Step	What is it	What have learned
✓ 1	Normalize the data name (► Section 3.5.2)	Re-use & adapt existing code easily: thanks to user-defined functions
✓ 2	Tag the objects (► Section 3.5.3)	Waiting for an action from the user Using a plugin from a macro
✓ 3	Isolate the overlapping parts (► Section 3.5.4)	Manipulating binary masks
✓ 4	Retrieve volumes (► Section 3.5.5)	Exploiting the plugin's outputs, Using the ROI Manager (repeat)
✓ 5	Generate outputs (► Section 3.5.6)	Using the Results Table to output data (repeat), Generating output images
✓ 6	Make the macro user friendly (► Section 3.5.7)	Generating Graphical User Interfaces

So, are we done? Since users' requests always evolve, we won't have to wait long till the user comes back, with his mind changed or asking for more...Get ready for next challenges in ► Sect. 3.5!

3.5 Workflow 2: Objects Overlap Intensity Quantification

Scenario

» It works well...but...

I have now the impression that the overlap might not be the main parameter. I think what matters is the amount of protein engaged in the colocalisation process.

Therefore, I would like to quantify object per object, channel per channel the percentage of protein involved in the process and get a map of quantification.

The user comes back to the Facility...but he has changed his mind

3.5.1 What Should We Do?

The answer is quite simple. First, we will go for a loud primal scream: we spent 3 h coding a macro that most probably won't be used!?! Second, we will be tempted to trash all what we've done, and heavily swear at the user who has no clue about what he really needs. Finally, we will think again at the beauty of our code, and try to find a way to re-use it.

Lucky enough, we have been structuring our code into functional blocks. Several of them can be re-used, as we only need to adapt the analysis part and the GUI. In study case 1, ► Sect. 3.4.5, we used a normalized mask and determined the intensities on it, using the 0–1 scaling

■ Table 3.3 What more do we need to do?		
Step	What is it	What to implement
✓ 1	Normalize the data name (Same as in ► Section 3.5.2)	Re-use & adapt existing code easily: thanks to user-defined functions DONE!
✓ 2	Tag the objects (Same as in ► Section 3.5.3)	Waiting for an action from the user Using a plugin from a macro DONE!
✓ 3	Isolate overlapping parts (Same as in ► Section 3.5.4)	Manipulating binary masks DONE!
4	Retrieve intensities (Todo in ► Section 3.6.2)	Exploiting the plugin's outputs, Using the ROI Manager (repeat)
✓ 5	Generate outputs (Same as in ► Section 3.5.6)	Using the ResultsTable to output data (repeat); Generating output images DONE!
6	Make it user friendly (To adapt in ► Section 3.6.3)	Generating Graphical User Interfaces

to report for volumes. Instead of using a scaled map directly, we could try to retrieve the original intensities. This strategy needs a bit of thinking, which we'll do in ► Sect. 3.5.2. Next table gives an overview of our workflow and the steps to be adapted (■ Table 3.3).

3.5.2 New Step 4: Retrieve Intensities

- Aim:**
- We need to generate new inputs for `getValues` function.
 - This step should be implemented as a function: `getMaskedIntensities` with the proper set of argument(s), if needed.

- Hints:**
- We already have generated functions to get masks within intensity of 0 for background and 1 for objects.
 - Arithmetics is possible between images via `[Process >Image Calculator]`.

Now we have settled down on user's request, we realize that the workflow is actually not too much work to implement and adapt. We have already been able for each channel to determine for each object its volume, and the part of its volume involved in the co-localization process, allowing to generate a physical co-localization percentage map.

What we now need to do is exactly the same process replacing the volume per object by the total fluorescence intensity associated to it. In the previous case study, we have fed the `getValues` function with the 0–1 scaled mask and the tagged map. We now should feed it with the original image intensities, together with the tagged map. We already have the normalized mask. It can be used together with the original image to get a filtered-intensities map. The operation to be performed is a simple mathematical multiplication between both images. This is the purpose of the `[Process > Image Calculator]` function from ImageJ. This operation should be performed using both the objects' maps for a channel and for the overlaps' mask. With all these in mind, the plan is easy to form:

Working Plan:

1. Use `[Process > Image Calculator]` function between the objects' mask and the original image
2. Rename the resultant image as "Masked_Intensities"
3. Use `[Process > Image Calculator]` function between the overlaps' mask and the original image
4. Rename the resultant image as "Masked_Intensities_Common"
5. Pack everything into a function, thinking about the proper arguments, if any, that should be entered for the process to be run

And its implementation takes the following form:

```
1 //Generate masked intensities images for the input image
2 function getMaskedIntensities(mask, intensities){
3   imageCalculator ("Multiply create stack", mask,intensities);
4   rename ("Masked_intensities_"+intensities);
5   imageCalculator ("Multiply create stack", "Common_volumes",
6     intensities);
7   rename ("Masked_intensities_Common_"+intensities);
8 }
```

To be able to launch and test the new version of the macro, we will replace Code 3.5 by:

```
1 //-----
2 // Main macro part
3
4 //...
5
6 //Step 4
7 getMaskedIntensities("Mask_Channel1", "Channel1");
8 getMaskedIntensities("Mask_Channel2", "Channel2");
9
10 objectsIntensity1=getValues("Tagged_map_Channel1", "Masked_inten-
11   sities_Channel1");
12 commonIntensity1=getValues("Tagged_map_Channel1", "Masked_inten-
13   sities_Common_Channel1");
```



```

13 objectsIntensity2=getValues("Tagged_map_Channel2", "Masked_inten-
14 sities_Channel2");
15
16 //Step 5
17 generateOutputs(objectsIntensity1, commonIntensity1, "Tagged_map_
18 Channel1");
19 IJ.renameResults("Intensity_colocalisation_Channel1");
20 selectWindow ("Coloc_Map");
21 rename ("Intensity_colocalisation_Channel1");
22
23 generateOutputs(objectsIntensity2, commonIntensity2, "Tagged_map_
24 Channel2");
25 IJ.renameResults("Intensity_colocalisation_Channel2");
26 selectWindow ("Coloc_Map");
27 rename ("Intensity_colocalisation_Channel2");
28 //-----

```

3.5.3 Adapted Step 6: Make the Macro User Friendly

Aim:

- Modify the GUI function to add choice between the two types of analysis.
- The GUI function should return an array containing the 2 former parameters, plus the choice made by the user.

Hints:

- We already have all the required code.
- We previously have seen some function exist to create GUIs: have a look at IJ build-in macro functions webpage.
- We previously have seen that structures exist to branch actions on users' input.

During the previous step (► Sect. 3.5.2, we have introduced a new way to process the images. This is a second option to interpret the colocalisation: rather than being based on the physical overlap, it deals with the distribution of molecules, as reported by the associated signal. Both options could be used in colocalisation studies, and it would be ideal to be able to switch from one option to another, e.g. by choosing it from a drop-down menu.

We already have seen how to create a GUI. The remaining step consists of customizing the existing one so that it accommodates the list and behaves according to the user choices. Therefore, surely the required instructions would start with something like `Dialog.addSomething` and `Dialog.getSomething`. Refer to [ImageJ's Built-in Macro Functions page](#) to find the proper instructions.

Once the parameters are set, the behavior of the macro should adapt. We should use a conditional operation of the different parts of the macro, with a proper structure that is described in the next technical point.

Technical Points: Generating Graphical User Interface

- Several types of instructions exist to branch the execution of certain part of the code according to a parameter' value. For example, a parameter can be tested against one reference value, against two or more.
- `if(condition) {} statement;` a boolean argument is provided to the `if` statement. If `true`, the block of instructions reside between the curly brackets are executed. If not, the macro will resume its execution, starting with the first line after the closing bracket. The argument generally takes the form of two members, separated by a comparison operator. Amongst operators, greater than sign, `>`, or lower than sign, `<`, could be used. Greater/lower than or equal to statement could be generated by addition the equal sign to the formers: `=`. As for equality, the comparison statement is formed by doubling the equal sign `==`, to be not confused with the attribution statement (simple sign). Finally, the non equality might be checked either by `<>` or by constructing a "not" in front of the "equal" statement: `!=`.
- `if(condition) {} else{} statement;` same as above, except this structure also specifies a way to handle the alternative behavior, in case the argument is false.
- `if(condition1) {} elseif(condition2) {} ...elseif(conditionN) {} statement;` this structure allows testing several conditions and adapt accordingly. This is an alternative way to achieve same result as the `switch/case` statement, which is not handled by the ImageJ macro language.

And here is the working plan of the final building block:

Working Plan:

1. Create a new array containing the possible options for colocalisation analysis methods
2. Modify the Dialog Box to include a drop-down list allowing selection of colocalisation methods
3. Retrieve the colocalisation method chosen by the user
4. Modify the return statement to take into account the new parameter
5. Adapt the behavior of the main part of the macro, depending on the user's choice

The adapted GUI function now looks like this:

```
1 //Display the graphical user interface
2 function GUI(){
3 items= newArray ("Intensity", "Volume");
4
5 Dialog.create("colocalisation");
6 Dialog.addNumber ("Minimum size of objects on channell (in vox-
  els)", 10);
```

```

7 Dialog.addNumber ("Minimum size of objects on channel2 (in vox-
  els)", 10);
8 Dialog.addChoice ("Analysis based on", items);
9 Dialog.show ();
10
11 out= newArray (3);
12 out[0]= Dialog.getNumber ();
13 out[1]= Dialog.getNumber ();
14 //Same kind of elements should be stored in an array
15 out[2]=0;
16 //The returned string is encoded as a number
17 if (Dialog.getChoice ()== "Volume") out[2]=1;
18 return out;
19 }

```

A possible implementation of our macro's main body could be implemented as follows:

```

1  if (parameters[2]==0){
2  //Performs intensity-based analysis
3
4  getMaskedIntensities("Mask_Channel1", "Channel1");
5  getMaskedIntensities("Mask_Channel2", "Channel2");
6
7  objectsIntensity1=getValues("Tagged_map_Channel1", "Masked_inten-
  sities_Channel1");
8  commonIntensity1=getValues("Tagged_map_Channel1", "Masked_inten-
  sities_Common_Channel1");
9
10 objectsIntensity2=getValues("Tagged_map_Channel2", "Masked_intensi-
  ties_Channel2");
11 commonIntensity2=getValues("Tagged_map_Channel2", "Masked_inten-
  sities_Common_Channel2");
12
13 generateOutputs(objectsIntensity1, commonIntensity1, "Tagged_
  map_Channel1");
14 IJ.renameResults("Intensity_Colocalisation_Channel1");
15 selectWindow ("Coloc_Map");
16 rename ("Intensity_Colocalisation_Channel1");
17
18 generateOutputs(objectsIntensity2, commonIntensity2, "Tagged_map_
  Channel2");
19 IJ.renameResults("Intensity_Colocalisation_Channel2");
20 selectWindow ("Coloc_Map");
21 rename ("Intensity_Colocalisation_Channel2");
22} else {
23 //Performs volume-based analysis
24 objectsVolume1=getValues("Tagged_map_Channel1", "Mask_Channel1");
25 commonVolume1=getValues("Tagged_map_Channel1", "Common_volumes");
26
27 objectsVolume2=getValues("Tagged_map_Channel2", "Mask_Channel2");
28 commonVolume2=getValues("Tagged_map_Channel2", "Common_volumes");
29

```

```
30 generateOutputs(objectsVolume1, commonVolume1, "Tagged_map_Channel1");
31 IJ.renameResults("Volume_Colocalisation_Channel1");
32 selectWindow ("Coloc_Map");
33 rename ("Volume_Colocalisation_Channel1");
34
35 generateOutputs(objectsVolume2, commonVolume2, "Tagged_map_Channel2");
36 IJ.renameResults("Volume_Colocalisation_Channel2");
37 selectWindow ("Coloc_Map");
38 rename ("Volume_Colocalisation_Channel2");
39 }
```

Take Home Message

Thanks to our user’s uncertainty (and to our patience), we have come up with a flexible macro that performs both object-based and intensity-based colocalisation analysis. During the process of implementation, we have applied strategies that allow us to format the code in such a way that is easy to adapt and modify the overall workflow. The use of functional blocks is the key element in adapting the behavior of our macro.

For both methods, we have generated quality control images in the forms of colocalisation maps. A simple table as an output might be difficult to visually interpret or to link to positional clues on the image. With this type of map, the user can visually inspect the output of our macro, and adapt its parameters to make the analysis even more accurate (► Table 3.4).

► Table 3.4 Overall, what have we learned?

Step	What is it	What have learned
1	Normalize the data name (► Section 3.5.2)	Re-use & adapt existing code easily: thanks to user-defined functions
2	Tag the objects (► Section 3.5.3)	Waiting for an action from the user Usinga plugin from a macro
3	Isolate the overlapping parts (► Section 3.5.4)	Manipulating binary masks
4	Retrieve volumes (► Section 3.5.5)	Exploiting the plugin’s outputs, Using the ROI Manager (repeat), Adapt existing code
5	Generate outputs (► Section 3.5.6)	Using the ResultsTable to output data (repeat), Generating output images
6	Make the macro user friendly (► Section 3.5.7)	Generating Graphical User Interfaces Branch on user’s input

Acknowledgements We are very grateful to Anna Klemm (BioImage Informatics Facility, SciLifeLab, Uppsala University) for reviewing our chapter and suggesting extremely relevant enhancements to the original manuscript. In particular, the workflow exposition and clarification made through ■ Fig. 3.7 results from one of her suggestions.

3

Bibliography

- Bolte S, Cordelières FP (2006) A guided tour into subcellular colocalization analysis in light microscopy. *J Microsc* 224:213–232
- Cordelières FP, Bolte S (2014) Experimenters' guide to colocalization studies: finding a way through indicators and quantifiers, in practice. *Methods Cell Biol* 123:395–408
- Dunn KW, Kamocka MM, McDonald JH (2011) A practical guide to evaluating colocalization in biological microscopy. *Am J Physiol Cell Physiol* 300:C723–C742
- Fletcher PA, Scriven DR, Schulson MN, Moore DW (2010) Multi-image colocalization and its statistical significance. *Biophys J* 99:1996–2005
- Lagache T, Meas-Yedid V, Olivo-Marin JC (2013) A statistical analysis of spatial colocalization using Ripley's K function. In: ISBI, pp 896–899
- Obara B, Jabeen A, Fernandez N, Laissue PP (2013) A novel method for quantified, superresolved, three-dimensional colocalisation of isotropic, fluorescent particles. *Histochem Cell Biol* 139(3):391–402
- Rizk A, Paul G, Incardona P, Bugarski M, Mansouri M, Niemann A, Ziegler U, Berger P, Sbalzarini IF (2014) Segmentation and quantification of subcellular structures in fluorescence microscopy images using Squassh. *Nat Protoc* 9(3):586–596
- Sage D, Donati L, Soulez F, Fortun D, Schmit G, Seitz A, Guet R, Vonesch C, Unser M (2017) Deconvolutionlab2: an open-source software for deconvolution microscopy. *Methods* 115:28–41
- Wörz S, Sander P, Pfannmöller M, Rieker RJ, Joos S, Mechttersheimer G, Boukamp P, Lichter P, Rohr K (2010) 3D geometry-based quantification of colocalizations in multichannel 3D microscopy images of human soft tissue tumors. *IEEE Trans Med Imaging* 29(8):1474–1484
- Zinchuk V, Zinchuk O (2008) Quantitative colocalization analysis of confocal fluorescence microscopy images. *Curr Protoc Cell Biol* U4:19

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (► <http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.