



ImageJ for Partially and Fully Automated Analysis of Trypanosome Micrographs

Richard J. Wheeler

Abstract

Trypanosomes and related parasites such as *Leishmania* are unicellular parasites with a precise internal structure. This makes light microscopy a powerful tool for interrogating their biology—whether considering advance techniques for visualizing the precise localization of proteins within the cell or simply measuring parasite cell shape. Methods to partially or fully automate analysis and interpretation are extremely powerful and provide easier access to microscope images as a source of quantitative data. This chapter provides an introduction to these methods using ImageJ/FIJI, free and open source software for scientific image analysis. It provides an overview of how ImageJ handles images and introduces the ImageJ macro/scripting language for automated images, starting at a basic level and assuming no previous programming/scripting experience. It then outlines three methods using ImageJ for automated analysis of trypanosome micrographs: Semiautomated cropping and setting image contrast for presentation, automated analysis of cell properties from a light micrograph field of view, and example semiautomated tools for quantitative analysis of protein localization. These are not presented as strict methods, but are instead described in detail with the intention of furnishing the reader with the ability to “hack” the scripts for their own needs or write their own scripts for partially and fully automated quantitation of trypanosomes from light micrographs. Most of the methods described here are transferrable to other types of microscope image and other cell types.

Key words Trypanosome, Microscopy, Image analysis, Automation, High content analysis

1 Introduction

Microscope images are an incredibly rich data source. Light microscopy is undergoing an ongoing revolution, with the development of genetically encoded fluorescent markers [1], the refinement of digital cameras [2], accessible super resolution light microscopy [3] and most recently the ongoing developments in selective plane illumination (SPIM) methods [4]. Computational analysis of microscope images is a necessity for much modern research which

Electronic supplementary material: The online version of this chapter (https://doi.org/10.1007/978-1-0716-0294-2_24) contains supplementary material, which is available to authorized users.

may involve an extremely large number of images [5]. For example, microscopy-based screens and the nascent field of genome-wide sub-cellular protein localization [6–9]. Alternatively, it may involve three dimensional imaging of large volumes (such as SPIM or electron tomography) or precise image analysis (such as super resolution image analysis). Automation makes these analyses more accessible, scalable, and repeatable.

This chapter focuses on accessible methods for partial or full automation of image analysis of trypanosomes, particularly *Trypanosoma brucei*, *T. cruzi*, and *Leishmania* spp. Automated image analysis is a flexible tool, therefore these methods have been designed to both be methods and accessible introductions to fundamentals of automated image analysis. The methods cover principals which are applicable to many pieces of image analysis software but the specific examples are written in the ImageJ macro language. Similarly, the image analysis principles are explained using trypanosomatids, but are transferrable to many nontrypanosomatid systems.

ImageJ (<https://imagej.nih.gov/ij/>) is the leading free and open source scientific image analysis software and includes a powerful and accessible scripting language [10, 11]. ImageJ has recently ceased active development, with a complete reimplementa-tion (called ImageJ2) now spearheading development [12, 13]. A distribution of ImageJ2 packaged with community-derived plugins is called FIJI (FIJI is just ImageJ) and is very popular (<https://fiji.sc/>) [14]. ImageJ2 and FIJI also use the ImageJ macro language, but also have several other scripting languages. For these methods the use of ImageJ is described, as the ImageJ macro language is simple and ImageJ has debugging tools which help learning. However FIJI and ImageJ2 are better suited for more complex applications.

The methods here can be followed using only a copy of ImageJ or FIJI along with example phase contrast and fluorescence micrographs. To this end, example micrographs of a procyclic form *T. brucei* cell line expressing Histone H3 (Tb927.1.2430) N terminally tagged with mNeonGreen (mNG) [15] are included as supplemental material for this chapter. These images are taken from the TrypTag project, the ongoing project on track to determine the subcellular localization of every protein encoded in the trypanosome genome [6]. These are micrographs typical of a modern epifluorescence microscope, and include phase contrast, green fluorescence (mNG) and blue fluorescence (Hoechst 33342, a DNA stain) channels.

This chapter is not an overview of how to use ImageJ; there are many tutorials for basic image handling in ImageJ (for example, <https://imagej.nih.gov/ij/docs/guide/user-guide.pdf>). Instead it focuses on scripting for full and partial automation. This is of particular advantage for trypanosomatids, as existing automated

tools are typically optimized for use with mammalian cells, restricting the options for using existing scripts and plugins. Relative to mammalian cells, *Trypanosoma* and *Leishmania* pose particular challenges. This is particularly because of their vermiform shape and the two DNA-containing organelles, the kinetoplast and nucleus—both of these features are fundamentally unlike typical mammalian cells. However, the precisely defined morphology of these parasites means micrographs are particularly data rich; cell-to-cell variation of an asynchronous population encodes cell cycle information [16] and subcellular localization of a protein is particularly informative for function through the precise positioning of many organelles [17]. The methods are built on concepts from several pieces of work using automated analysis. However, the concepts here are related and transferrable to automated high-content image based screening—an important topic for trypanosomatid drug development [18–24].

There are a great range of computational image analysis methods of value for trypanosomatid biology, many more that can be summarized in this chapter. Valuable methods include image corrections (camera readout noise correction, correction of illumination unevenness and chromatic aberration correction), video corrections (photobleaching correction, image stabilization), *z* stack corrections (chromatic offset, post-capture autofocus), particle tracking (cell swimming behaviors, subcellular particles), and quantitative analysis of microscopy techniques (fluorescence recover after photobleaching (FRAP), ratiometry). I hope that the methods described here give a useful introduction to the toolkit available for these approaches.

These methods are written assuming no programming experience. It is possible to perform automated image analysis by only making use of previously published tools, however the ultimate value of automated analysis is customizability—these methods aim to confer the reader some scripting ability to achieve this. The chapter begins with a quick introduction of how image data is handled by computers and an overview of the ImageJ macro language syntax (which falls in the “curly-bracket” group along with languages like C and javascript). This is followed with an explanation of how ImageJ handles multidimensional image stacks. Three experimental methods are then covered: Firstly, partial automation of the preparation of images for figures, secondly, methods for automating the analysis of protein expression level and cell cycle stage of trypanosomes, then, finally, two methods for quantitative image analysis. The intention is to introduce sufficient concepts, starting with simple examples, so that automated image analysis methods published with primary research papers will be accessible to people who have followed the examples in this chapter. Readers with some programming experience may want to jump straight to the later examples.

When using the scripts in the methods it is heavily recommended to type them out, rather than copying and pasting. Please also be aware that typesetting can wrap long lines which may introduce incorrect new lines when copying and pasting. Recognizing errors in syntax in handwritten code and interpreting the errors reported when trying to execute invalid code are important for understanding how to write scripts.

2 Materials

2.1 Software

1. ImageJ (<https://imagej.nih.gov/ij/>) or FIJI (<https://fiji.sc/>): Both are free and open source, and can be downloaded and run on Windows, Mac, or Linux.
2. ImageJ reference guides: The ImageJ User Guide (for new ImageJ users) <https://imagej.nih.gov/ij/docs/guide/user-guide.pdf> and the Macro Reference Guide https://imagej.nih.gov/ij/docs/macro_reference_guide.pdf.
3. LOCI Bio-Formats plugin: If using ImageJ with images in proprietary formats, for example saved from light microscope software (*see Note 1*), this plugin is required. Download the jar file (<http://www.openmicroscopy.org/bio-formats/downloads/>) and copy it to the plugins folder of your ImageJ directory. You will have to restart ImageJ for it to be recognized.

2.2 Images

High bit-depth images with separate images for each imaging channel: merged images are not suitable and compressed image formats (e.g., jpg/jpeg) or image formats for day-to-day use (e.g., png) should be avoided wherever possible (*see Notes 2 and 3*). Similarly, for videos, compressed formats (avi, mp4, etc.) should be avoided wherever possible—a series of uncompressed still images is more suitable. For any quantitative analysis images must be captured with great care (*see Note 4*). Problems with the images will cause systematic errors in any downstream analysis and can prevent automated analysis entirely. Example images are provided in the supplemental material. These are good examples of the quality of image suitable for quantitative image analysis.

3 Methods

3.1 Loading and Running ImageJ Macros

ImageJ scripts are called macros and are simply text files. They can be saved as a text (file extension txt) but are often saved with the file extension ijm. To open a macro simply use File>Open... in ImageJ and it will open as a new window showing the macro text. The macro can also be opened by dragging the file from a file explorer window to the ImageJ window.

Once open, the macro can be run using `Macros>Run Macro` in the window showing the macro text, or by selecting the macro window and using the keyboard shortcut `Ctrl+R`. Macros can also be run in debug mode (*see Note 5*).

3.1.1 ImageJ Macro Syntax

At the basic level, an ImageJ script/macro is a set of commands to run in order. Almost every menu option can be used as a command in a macro and ImageJ provides the means to record the commands you enter using the Macro Recorder (`Plugins>Macros>Record...`). There are also many functions specific to the scripting language, these can be found in the macro reference documentation (https://imagej.nih.gov/ij/docs/macro_reference_guide.pdf) or by the search box in the main window (only in ImageJ2/FIJI).

```
//The command recorded for File>New>Image... then entering some values
newImage("Untitled", "16-bit black", 256, 256, 1);
```

This function takes 5 parameters: image name (a string), image type (one of several options) then three integers: width, height, and number of slices. It generates a new blank image with these properties. This particular function can be found by using the macro recorder and creating a new image or by looking up the function in the reference material.

The semicolon at the end of the line indicates that that is the end of this line of code; new lines in this scripting language are cosmetic, ImageJ looks for the semicolon. Comments can be included in the code: Any text on a line following `//` is a comment is not interpreted as code. Similarly, blocks of text between `/*` and `*/` are comments.

The simplest scripts are simply a series of ImageJ commands or functions.

```
//Makes a new image with a white rectangle in the centre
newImage("Untitled", "16-bit black", 256, 256, 1);
makeRectangle(64, 64, 128, 128);
setColor(65535);
fill();
run("Select None");
```

This script first makes a new image and makes a rectangular selection in the centre of the image. These first two functions are “recordable.” It then sets the current working color (an internal variable in ImageJ) and fills the selection with that color. These functions are not recordable and have to be looked up in the macro guide. Finally, it unselects all current selections.

Variables are script-defined names for values which can be used at multiple places in the script. The values of these variables can have different types: a Boolean (true or false), a number (an integer

or real number) or a string (a line of text). These variables can be handled by standard string and mathematical functions.

```
//Starting variables
//The base and channel names are strings
baseName="NewImage";
channelName="Red";
//The base dimension is an integer
baseDimension=128;
//Variables derived from the starting variables
//Concatenating strings (image name)
finalName=baseName+"_"+channelName;
//Mathematics using numbers (image dimensions)
width=baseDimension;
height=baseDimension*2;
newImage(finalName, "16-bit black", width, height, 1);
```

This will generate a new image called `NewImage_Red` with a width of 128 pixels and a height of 256 pixels. If you are new to programming, mathematical functions (*see Note 6*) will be quite intuitive. String functions (*see Note 7*), Boolean algebra functions (*see Note 8*), and Boolean comparisons (*see Note 9*) will likely be new but are important to understand.

Variables can also be an array, which is a list of Booleans, numbers, and/or strings.

```
//Make a new array
values=newArray(10, "red", true);
print(values[0]);
print(values[1]);
print(values[2]);
```

Here, the array `values` has three values within it. These values can be accessed by specifying the index within the array, specified using a value in square brackets immediately following the array variable name: `values[n]` means the *n*th entry in the array (*see Note 10*).

Commands/functions in ImageJ may have one of several effects: Immediately running a command, doing a calculation and returning a single value, or setting multiple variables. This can be a little confusing.

```
//This immediately makes a new image using these parameters
newImage("Image1", "16-bit black", 100, 150, 1);

//This runs the ImageJ pow() exponent function
//It returns a number, and the variable a is set to the result
a=pow(2, 16);

//This runs the ImageJ getSelectionCoordinates() function
//The variables x and y are both set to equal two different parts of
```

```
the result
getSelectionCoordinates(x, y);
```

Here, `newImage()` immediately runs an ImageJ command, `pow` is a script function which returns a value (a number in this case) and `getSelectionCoordinates()` sets two variables.

Within a script the basic script flow controls are available: Conditional code statements (if, else if, and else) and looped statements (for and while). Custom functions can also be defined which take parameters and can be called within a script. Combining these with recorded commands provides an easy route to powerful image handling.

The syntax for any of these tools for controlling the flow of the program use braces (curly brackets: `{}`) to delineate the chunks of code which are within the function, loop, conditional statement, and so on. For ease of reading a script, it is recommended to indent the contents of braces using tab or a consistent number of spaces. However, like new lines, this is purely cosmetic.

The syntax of each of these flow control tools is different. if, while and for use the contents of the brackets immediately following the command to define their function. Functions differ, with the parameters in the brackets representing variables to be specified when the function is called.

```
//Conditional statements
a=10;
b=4;
if (a>b) {
    //Code to run if the comparison a>b is true
    print("a is larger than b");
} else if (b>a) {
    //Code to run if the comparison b>a is true and a>b was not true
    print("b is larger than a");
} else {
    //Code to run if neither b>a or a>b were true
    print("a is equal to b");
}

//While statements
a=0;
b=10;
while (a<b) {
    //Code to loop while the comparison a>b is true
    print("This block of code has run "+(a+1)+" times");
    //a++ is the same as writing a=a+1
    a++;
}

//For loops
```

```

//Three statements are needed for a for loop:
// 1) A start condition (here, set i=0)
// 2) A continuation test (here, loop again if i<10 is true)
// 3) A command to run every loop (here, increment i by 1, ie. i=i+1)
for (i=0; i<10; i++) {
    print("This block of code has run "+(i+1)+" times");
}

//Functions
customFunction("Hello", "world!");
//The function is not called by itself, only when called, as above.
function customFunction(parameter1, parameter2) {
    print("First parameter is: "+parameter1);
    print("Second parameter is: "+parameter2);
}

```

Finally, sub-macros can be defined. This essentially allows for multiple macros within a single file, and they can appear as menu or toolbar entries. This format allows for a second method of running the script; they can be installed to the Plugins menu in ImageJ using **Plugins>Macros>Install....**

```

//This script can be installed using Plugins>Macros>Install...
//When installed it will appear under Plugins>Macros>Save and
close all images
macro "Save and close all images" {
    //Display a directory select dialog and record the path in the
variable path
    path=getDirectory("");
    //Call the custom function saveImagesToPath
    saveImagesToPath(path);
}

//A custom function which could also be used elsewhere in the script
function saveImagesToPath(path) {
    //A while loop which continues until no images are open
while (nImages()>0) {
    //Select the first image, save it and close it
    selectImage(1);
    saveAs("TIFF", path+getTitle());
    close();
}
}

```

When installed, this macro appears under the **Plugins>Macros** menu in ImageJ as an entry called **Save and close all images**. The macro calls a custom function (called `saveImagesToPath`), which itself uses a while loop to continue looping while the logical test `nImages()>0` (there is still at least one image open) gives the value true.

3.1.2 Image Handling

ImageJ best handles multichannel, multi-focal plane and multi-timepoint images as a “hyperstack”—a multislice image stack where the individual images are mapped to a channel, focal plane, and timepoint. This is then displayed one of three ways: Greyscale (where only the selected image is shown), Color (where only the selected image is shown, but pseudocolored), and Composite (where all the channels for the current focal plane and time point are combined with their pseudocolors to a single display image). In general, the Color and Composite modes are for peoples’ convenience viewing images, while image processing commands may only function reliably in Greyscale mode. To make sure ImageJ understands the “structure” of the image, the number of channels, focal planes, and timepoints this can be set via Image>Hyperstack>Stack to Hyperstack.... New images can also be created as hyperstacks using the newImage function. When loaded correctly it is easy to access the different images in the hyperstack for analysis.

```
//Create a new hyperstack
newImage("HyperStack", "16-bit composite-mode label", 400, 300,
3, 1, 5);

//Set the display mode to greyscale
Stack.setDisplayMode("grayscale");
//Set the displayed image to channel 2 of the 5th timepoint
Stack.setChannel(2);
Stack.setFrame(5);
```

3.2 Semiautomated Image Preparation for Presentation or Publication

When displaying micrographs it is important to recognize that they are primary data and treated as such [25]. Useful guidelines are: Firstly, to aid interpretation, showing cells in images cropped to a consistent size and in a consistent orientation helps side-by-side comparison. Secondly, to give a fair representation of the data, the mapping of the raw data to image intensity in the final image (i.e., the brightness and contrast) should be selected which do not saturate detail or clip the background. Finally, to avoid artificial enhancement of structures, no filtering should be used which alter some areas of the image but not others.

This method outlines three ImageJ macros which help follow these guidelines and streamline image preparation for presentation. Firstly cell cropping, secondly automatic setting of contrast, and finally saving of raw and composite images. These examples assume a composite image with three channels (phase contrast, a green fluorescent protein and a DNA stain) and just one focal plane and time point. This is an excellent example of how simple sequential commands can do very useful functions.

3.2.1 *Semiautomated Cell Rotation and Cropping*

Trypanosomes (and related organisms like *Leishmania*) are highly polarized, so it is simple to identify a preferred orientation. This script rotates and crops a cell in a larger field of view to a standard orientation and image size. This is semiautomated, and the cell orientation must be specified manually. In this case, the user must draw a line, using the line selection tool in ImageJ, along the cell anterior to posterior axis (*see Note 11*).

```
//Two variables to define the output image size in pixels
width=323;
height=162;

//Check if the current selection type is 5 (a line selection)
if (selectionType()!=5) {
//Exit and display an error if the selection is missing or is
not a line
exit("Error: No line selection found!");
}
//Get the coordinates of the user-specified line
//This sets x and y to arrays which describe the x and y coordinates of
the selection
getSelectionCoordinates(x, y);
//Calculate the orientation and center of the line
angle=-180*atan2(y[1]-y[0], x[1]-x[0])/PI;
centerX=(x[1]+x[0])/2;
centerY=(y[1]+y[0])/2;
print("Center: "+centerX+"px, "+centerY+"px Orientation: "+angle+"
degrees");

//Start the cropping process
//First, calculate a temporary image width
//This will allow subsequent rotation of the image without clipping
tempWidth=pow((width*width/4+height*height/4), 0.5);
//Make a rectangle of twice the temporary width around the cell
makeRectangle(centerX-tempWidth, centerY-tempWidth, temp-
Width*2, tempWidth*2);
//Duplicate this region of the image to a new image
run("Duplicate...", "duplicate");
//Rotate by the angle
run("Rotate... ", "angle="+angle+" grid=1 interpolation=Bicu-
bic stack");
//Crop to the output image size
makeRectangle(tempWidth-width/2, tempWidth-height/2, width,
height);
run("Crop");
```

3.2.2 Automated Contrast

The cropped image can then be prepared for display. ImageJ has a useful built-in function for automatically setting image brightness and contrast to clip/saturate a specified percentage of pixels. This script uses this function to quickly set the contrast and display color of the phase contrast, green, and DNA stain channels.

```
//The phase contrast and DNA stain images are set to have 0.1% clipped
pixels
//This gives good contrast for easy viewing
//The Green fluorescence image is set to have 0% clipped pixels
//This ensures no image data is lost in saturated bright or clipped
dark pixels
//Phase contrast
Stack.setChannel(1);
run("Grays");
run("Enhance Contrast", "saturated=0.1");
//Green fluorescence
Stack.setChannel(2);
run("Green");
run("Enhance Contrast", "saturated=0");
//DNA stain
Stack.setChannel(3);
run("Magenta");
run("Enhance Contrast", "saturated=0.1");
```

For some applications it is more important to show images with the same contrast for direct comparison, for example green signal intensity before and after induction of an inducible cell line. In these cases, it is likely better to specify the specific contrast to use, for example replacing line 12 with:

```
setMinAndMax(1000, 15000);
```

This function sets the display of the image such that values of 1000 map to black (0 on your 8-bit screen) and 15,000 maps to white (255 on your 8-bit screen). The particular values suitable for any set of images require careful consideration.

3.2.3 Automated Saving of Raw and Composite Images

Finally, it is convenient to automate the saving of the image with the raw data (as a tiff) and composite images for display (as png) (*see Note 12*). This script asks the user for a directory, then saves the raw data as a tiff and png images of the overlay and the green fluorescent channel alone.

```
//Bring up a window for the user to select a directory
path=getDirectory("");
//Save the raw data as a tiff
name=getTitle();
saveAs("TIFF", path+name+".tif");
```

```
//Set the display to all three channels in composite mode and save as a
png
Stack.setDisplayMode("composite");
Stack.setActiveChannels("111");
saveAs("PNG", path+name+"_overlay.png");
//Set the display to the green channel only in greyscale mode and save
as a png
Stack.setDisplayMode("grayscale");
Stack.setActiveChannels("010");
saveAs("PNG", path+name+"_gfp.png");
//Rename the image back to its starting name (it gets modified when
saved)
rename("name");
```

These are deliberately written to be minimal scripts, specific to this particular combination of image channels. For example, these do not check that the image has the expected number of channels or that they are in the correct order. Adjusting these to a particular data set is intentionally left as a challenge for the reader.

3.3 Automated Cell Cycle Analysis

Micrographs of cells at random cell cycle stages encode information about the cell cycle of the cell [16]. This is particularly true in *T. brucei* where the precise series of cell cycle events can be analyzed [26]. This is also readily accessible by automated analysis, by mixing ImageJ commands with basic flow control. Automated analysis ensures unbiased repeatable analysis, important for any cell profiling [5].

This method shows a basic measurement of green fluorescence signal intensity for a field of view of multiple trypanosomes. If using the example images this is a measurement of Histone H3 (green fluorescence signal intensity) in trypanosomes as a proxy for cell cycle stage. There are two steps to analyzing the particles (the cells) in the image: Firstly, identifying them and isolating them from the background. Secondly, measuring the properties of interest. In this method, this is extended to a third step, which outlines a method for counting the number of kinetoplasts and nuclei in cells.

3.3.1 Thresholding Trypanosomes from Phase Contrast

Phase contrast images (*see Note 13*) can be prefiltered to allow intensity thresholding to generate a simplified image where particles (assuming it has all worked correctly, cells) are white (255) and the background is black (0). However, depending on the precise sample preparation method (*see Note 14*) and the particular microscope (*see Note 15*) these will need some adaptation, particularly of the radii of the unsharp masks and the background subtraction.

```
//Select the phase contrast image
Stack.setChannel(1);
run("Select None");
```

```
//Switch to grayscale display mode, as thresholding only works in
this mode
Stack.setDisplayMode("grayscale");

//Filter the phase image to highlight particles the size of
trypanosomes
run("Unsharp Mask...", "radius=10 mask=0.60 slice");
run("Unsharp Mask...", "radius=20 mask=0.60 slice");
run("Subtract Background...", "rolling=25 light slice");

//Threshold the image
setOption("BlackBackground", true);
setAutoThreshold("Default");
//Next we manually convert this image to a binary image (either
black 0 or white 255)
//This is a little more manual than the built-in ImageJ tools
to handle the 16-bit image
getThreshold(min, max);
changeValues(0, max, 0);
changeValues(max, pow(2, 16)-1, 255);
run("Macro...", "code=v=255-v slice");

//Filter and re-threshold the binary image
//This helps separate closely-spaced cells
run("Gaussian Blur...", "sigma=3 slice");
changeValues(0, 192, 0);
changeValues(192, 255, 255);
```

In this example the process of converting the phase contrast image to a binary thresholded image is a little complex. When working with a single image the ImageJ command run(“Convert to Mask”) can be used to generate the binary image. However, this command is not compatible with processing only a single slice of a hyperstack made of 16-bit images, forcing the more convoluted approach here. Making the thresholded image as part of the stack simplifies the following steps.

3.3.2 Quantitation of Fluorescence Intensity

To analyze these particles a list of coordinates of particles can be retrieved then looped through, selecting the particle at each coordinate with the “wand” tool for automated analysis. At this point, we can check the area of the particle to determine if it is likely to be a cell and analyze the integrated signal intensity in the green fluorescence image to get a measure of expression of Histone H3 tagged with mNG.

```
//Variables setting the minimum and maximum area that looks like a
single cell
minimumArea=2500;
maximumArea=8000;
```

```

//Use a built-in tool to make a point selection of all the thresholded
particles
Stack.setChannel(1);
run("Select None");
run("Find Maxima...", "noise=10 output=[Point Selection]");
//Record the list of coordinates of the particles as two arrays, x and y
getSelectionCoordinates(x, y);
//Loop through the list of coordinates to analyse the particles
for (i=0; i<lengthOf(x); i++) {
//Select the thresholded phase contrast image
Stack.setChannel(1);
//Use the wand selector to select the particle
doWand(x[i], y[i]);
//Get the particle area (in pixels)
getRawStatistics(area);
if (area>minimumArea && area<maximumArea) {
//Only analyse particles large enough to be a cell
//Check the bounds of the selection
getSelectionBounds(sx, sy, sw, sh);
if (sx>0 && sy>0 && sx+sw<getWidth()-1 &&
sy+sh<getHeight()-1) {
//Only analyse if the particle is not touching the image edge
//Get the image intensity from the DNA stain channel
Stack.setChannel(3);
getRawStatistics(area, mean);
//Output the result
print(x[i], y[i], area, mean*area);
} else {
//'Delete' particles which should not be analysed
setColor(0);
fill();
}
} else {
//'Delete' particles which should not be analysed
setColor(0);
fill();
}
}
}

```

This script will output a tab-delimited data table in the log window with four columns: *X* coordinate, *Y* coordinate, area (in pixels), and sum signal intensity in the green fluorescence channel.

3.3.3 *Kinetoplast and Nucleus Counts*

This approach can be extended to threshold both the phase contrast and DNA stain image, then loop through the particles in the phase contrast image and count the number of DNA particles within the bounds of the phase contrast particle. This is simplified

by the behavior of the run("Find Maxima...") tool which only identifies particles within the area of the current selection—in this case finding thresholded particles in the DNA stain image within the bounds of the cell as identified in the phase contrast image. These particles can then be classified based on area, yielding a count of the number of kinetoplasts and nuclei in the cell. The following script takes the analyzed image resulting from the previous script (i.e., with cells already identified in a binary image), and reanalyzes it to look at the number of kinetoplasts and nuclei per cell.

```
//The area threshold to count as a kinetoplast or nucleus
knAreaCutoff=100;

//Set the DNA stain channel
Stack.setChannel(3);
run("Select None");
//Do a small blur then thresholding, similar to the phase
contrast image
run("Gaussian Blur...", "sigma=1 slice");
setAutoThreshold("Otsu dark");
getThreshold(min, max);
changeValues(0, min, 0);
changeValues(min, pow(2, 16)-1, 255);

//Repeat the cell detection as before
//The cells have already been filtered to be within the necessary
size range
Stack.setChannel(1);
run("Find Maxima...", "noise=10 output=[Point Selection]");
getSelectionCoordinates(x, y);
for (i=0; i<lengthOf(x); i++) {
    Stack.setChannel(1);
    doWand(x[i], y[i]);
    //Do the kinetoplast and nucleus count
    countKN(knAreaCutoff);
}

function countKN(knAreaCutoff) {
    //Setup variables to record the count of Ks and Ns
    kCount=0;
    nCount=0;
    Stack.setChannel(3);
    //Analyse the particles using the same strategy as for the cells
    run("Find Maxima...", "noise=10 output=[Point Selection]");
    getSelectionCoordinates(x2, y2);
    for (i=0; i<lengthOf(x2); i++) {
        //Select the current DNA particle
        doWand(x2[i], y2[i]);
```

```

getRawStatistics(area);
//If the area is larger than the cutoff count as N, otherwise K
if (area>knAreaCutoff) {
nCount++;
} else {
kCount++;
}
}
//Print the resulting K and N count
print(kCount+"K"+nCount+"N");
}

```

The limit in the quality of an approach such as this is the accuracy of thresholding the phase contrast and DAPI images and the accuracy of classifying phase contrast particles as cells and DAPI particles as kinetoplasts or nuclei. Careful analysis of the output will reveal a tendency to split some nuclei into multiple particles and miss some kinetoplasts. More advanced thresholding methods can be used, and more classifiers just area can be used (circularity, maximum length dimension, signal properties), but ultimately more information may be required.

Use of additional DNA stains, with a differing preference for AT or GC rich DNA is one such approach [27]. This is a powerful approach, allowing for unambiguous analysis of the kinetoplasts and nuclei of candidate *T. brucei* gametes [28] and diverse kinetoplastids and related organisms [29]. The example images with this chapter have an mNeonGreen fusion of Histone H3 along with the DNA stain; it is a challenge to the reader to adapt this approach to use that additional information for a more accurate kinetoplast/nucleus count.

3.4 Automated Quantitative Analyses

Computational image analysis has access to the raw numerical data which defines the image and can therefore carry out quantitative analyses that are not possible by eye. Two example methods selected here are quantitative colocalization and very precise localization of the center of signal from a diffraction limited particle, both of which exploit the built-in curve fitting functions in ImageJ.

3.4.1 Quantitative Colocalization

Quantitative measures of colocalization are valuable in many biological contexts. Using the example images provided it can be used to quantitatively compare Histone H3 and DNA stain signal and compare the result for nuclei and kinetoplasts.

Every pixel in a fluorescence microscope image is a proxy for the local concentration of the fluorescent marker, therefore colocalization of two molecules visualized in two different fluorescence channels should manifest as correlation of the pixel values in one channel versus the other. When plotted as a scatter plot a strong

colocalization is manifested as a linear positive correlation in pixel values with a small Pearson's correlation coefficient (R^2) [30].

```

if (selectionType!=0) {
    exit("Error: Needs a rectangular selection!");
}
//Get the bounds of the current selection
getSelectionBounds(x, y, w, h);
//Sets up two arrays to contain green (v1) and blue (v2) pixel values
v1=newArray(w*h);

v2=newArray(w*h);
//Loop through x and y coordinates to record values from the green
channel
Stack.setChannel(2);
//For values of a between 0 and the width of the selection
for (a=0; a<w; a++) {
    //And values of b between 0 and the height of the selection
    for (b=0; b<h; b++) {
        //Record the value of the pixel in the corresponding array entry
        //The pixel value at (x+a, y+b) is recorded as array entry [a+b*w]
        v1[a+b*w]=getPixel(x+a, y+b);
    }
}
//Loop through x and y coordinates to record values from the blue
channel
Stack.setChannel(3);
for (a=0; a<w; a++) {
    for (b=0; b<h; b++) {
        v2[a+b*w]=getPixel(x+a, y+b);
    }
}
//Do a linear regression analysis
//A positive linear correlation indicates co-localisation
//Carries out the actual fit
Fit.doFit("Straight Line", v1, v2);
//Plots the fit as a graph
Fit.plot();
//Prints the return values
print("R squared: "+Fit.rSquared()+" Gradient: "+Fit.p(1));

```

This script uses several approaches. Firstly, it reads image information in the form of pixel values (using `getPixel()`) and records it as entries in an array, mapping the 2D pixel data (relative coordinates a and b) into a 1D array (index $a + b \times w$). Secondly, it analyzes this data by automated use of the built-in graph plotting and curve fitting functions in ImageJ.

Using the example images, if this is run with a rectangular selection covering a nucleus then it will return a high R^2 , a positive gradient and a plot showing a positive linear correlation of nuclear

DNA stain signal with Histone H3 signal. If the selection covers a kinetoplast then it will show a very low R^2 , near-zero gradient, and no correlation of the two signals.

3.4.2 Point Spread Function Fitting for Superresolved Analysis

Precise determination of the center of a signal is a powerful tool whenever considering structures which are very small. Its particular power is in measuring distances between small structures. If the structures are sparse or in separate fluorescent channels then this is a non-diffraction-limited measurement. Repeated imaging of sparse single fluorophores is the basis for the PALM/STORM class of superresolution microscopy, while comparison between two channels can allow the reconstruction of molecule position in very small structures [31]. This is normally achieved by 2D Gaussian (a good approximation for the Airy disk point spread function) fitting of near-point sources [32, 33].

This method uses an example macro to take a user-selected point, then measures the signal distribution at that point to refine that point to the precise center of the signal. Specifically, it fits a Gaussian in X and then Y and the center of those Gaussian distributions are used to shift the initially selected coordinate to the signal center.

```
//Distance in pixels to use for Gaussian fitting
r=6;

//Call the custom refinePoints function
//This refines every point in a user-specified selection
refinePoints(r);

function refinePoints(r) {
  //Get the coordinates of the point selection
  if (selectionType()!=10) {
    exit("Error: Point selection needed!");
  }
  getSelectionCoordinates(x, y);
  //Setup output arrays for the corrected x and y coordinates
  ox=newArray(); oy=newArray();
  for (i=0; i<lengthOf(x); i++) {
    //For every point, make a point selection
    //Round the coordinates to the nearest pixel
    makePoint(round(x[i]), round(y[i]));
    //Call the custom refinePoint function to do the Gaussian fitting
    refinePoint(r);
    //Check if a selection exists
    //(if the refinePoint function can't fit the Gaussian)
    if (selectionType!=-1) {
      //Record the refined location for output
      getSelectionCoordinates(cx, cy);
    }
  }
}
```

```

    ox=Array.concat(ox, cx); oy=Array.concat(oy, cy);
  }
}
//Make a point selection of the refined points
makeSelection("points", ox, oy);
}

function refinePoint(r) {
  //Record the ID of the original image
  src=getImageID();
  //Get the coordinates of the point to refine
  getSelectionCoordinates(x, y);
  x=x[0]; y=y[0];
  if (x>r && y>r && x<getWidth()-r && y<getHeight()-r) {
    //Only continue if the point is not too close to the image
    edge
    xo=x; yo=y;
    //Duplicate a rectangle around the point to refine
    makeRectangle(x-r, y-r, r*2, r*2);
    run("Duplicate...", " ");
    tmp=getImageID();
    //Do the Gaussian fitting
    //This part of the code needs to be run in the X and Y
    direction
    //By using a for loop the code can avoid unnecessary
    duplication
    for (o=0; o<2; o++) {
      //Select the image and get the signal intensity profile
      makeRectangle(0, 0, r*2, r*2);
      vy=getProfile();
      //Setup an array of horizontal pixel positions
      vx=newArray(lengthOf(vy));
      for (i=0; i<lengthOf(vx); i++) {
        //The offset of 0.5 is due to the handling of pixel data
        //The distances are from the top left corner of the pixel
        vx[i]=i-r+0.5;
      }
      //Do the gaussian fit
      Fit.doFit("Gaussian", vx, vy);
      if (Fit.rSquared())>0.9) {
        //If a good fit is achieved
        if (Fit.p(2)>-r/2 && Fit.p(2)<r/2) {
          //And the centre of the Gaussian reasonable
          //Correct either the X or Y coordinate
          if (o==0) {
            x+=Fit.p(2);
          } else if (o==1) {
            y+=Fit.p(2);
          }
        }
      }
    }
  }
}

```

```

    }
    } else {
        //Failure to fit returns an invalid number
        if (o==0) {
            x=0/0;
        } else if (o==1) {
            y=0/0;
        }
    }
    } else {
        //Failure to fit returns an invalid number
        if (o==0) {
            x=0/0;
        } else if (o==1) {
            y=0/0;
        }
    }
    }
    if (o==0) {
        //After the first (X) loop rotate the image for Y
        run("Rotate 90 Degrees Left");
    } else if (o==1) {
        //After the second loop close the temporary image
        selectImage(tmp);
        close();
    }
    }
} else {
    //Return an invalid number if too close to the image edge
    x=0/0; y=0/0;
}
//Make a selection of the corrected point location in
the source image
selectImage(src);
if (!isNaN(x) && !isNaN(y)) {
    makePoint(x, y);
} else {
    run("Select None");
}
setBatchMode(false);
}

```

This script uses two custom functions, `refinePoints()` and `refinePoint()`, to carry out the analysis. This modularity in the design of the script is deliberate, and allows easy reuse or adaptation of all or part of the script in a future analysis. It also makes use of setting a variable to equal `0/0` (which has no defined value and recorded in ImageJ as “NaN” or not a number) when an analysis is not possible, then later checking for this internal error using the function `isNaN()`.

The example images do not provide a valuable use case for this method, but the function of the script can be readily tested on kinetoplasts. *T. brucei* kinetoplasts are very small and, although they are not a diffraction limited point, their position can be analyzed by this method. Running this script with a point selection in the DNA stain channel close to the center of a kinetoplast will refine the point location to the precise fitted center of signal. In the case of kinetoplasts this allows, for example, precise measurement of kinetoplast separation in 2 K cells.

4 Notes

1. Microscopes may save images in a proprietary format, for example, czi files (Zeiss), lif files (Leica), nic files (Nikon), and oir files (Olympus). These normally record the image data in a similar way to high bit-depth TIFF files (*see Note 3*), but potentially pack multiple images or more metadata into the file. Most can be opened using the LOCI Bio-Formats plugin.
2. It is important to understand the different types of image data. Most images in day-to-day life are RGB images, where each pixel has a red, green and blue value between 0 and 255 (2^8-1 , 8-bit), and are typically compressed, where the image data has been compressed (with some loss of information) to reduce the file size. Example formats are jpg, png, and gif. Scientific images are often not colored (although may be artificially pseudocolored), are either uncompressed or losslessly compressed and often have a higher bit-depth (*see Note 3*). Scientific images with multiple imaging modes of colors of fluorescence normally exist as a set of independent images. “Merged” images (combining multiple fluorescence or other images using pseudocolors) are useful for our interpretation, but unusable for a computational analysis.
3. Bit-depth refers to the number of bits used to store intensity values per pixel, and a higher bit depth corresponds to the ability to store a larger range of values. Scientific images often have pixel values from 0 to 65,535 ($2^{16}-1$, 16-bit). While our eyes cannot easily perceive this additional graduation of intensity it is important data.
4. Microscope images must be high quality and captured with care. If an image is in any way tricky to analyze by eye then it is not suitable for automated image analysis: Your eye and brain represent hundreds of millions of years of evolution of image analysis, a computer will not be as good. Key things to consider during image capture are: eliminate dirt on the slide, contamination of the sample and damage or death of any cells. Optimize sample preparation to minimize background fluorescence. Do not over-expose the image (saturate pixels)

but also pick a suitably long exposure time to avoid image noise in balance with avoiding photobleaching. Ensure the illumination is uniform, realign the Kohler illumination and fluorescence light sources if necessary, and critically assess any offset between channels (e.g., chromatic aberration or stage instability).

5. The macro can also be run line by line by starting it in debug mode: `Debug>Debug Macro` or the keyboard shortcut `Ctrl+D`. Once in debug mode it can be run line by line (`Debug>Step` or `Ctrl+E`), and the current line being executed is highlighted. When running in debug mode an additional window showing the values of each variable is also shown. This is extremely useful for understanding how the macro runs.
6. The normal functions of `+` addition, `-` subtraction or negation, `/` division, and `*` multiplication. Exponents must be written using the `pow()` function.
7. Two strings can be joined together (concatenated) using `+`. The opposite of concatenation is extracting a subportion of a string using `substring()`.
8. Boolean algebra functions allow the combination of true and false values using logic functions (sometimes called logic gates). The three common functions are: `!` not, `||` or, and `&&` and. For example `!true` equals false, `true || false` equals true.
9. Logical comparators give a value of true or false depending on the values being compared. These include the `>` greater than, `<` less than, `>=` greater or equal to and `<=` less or equal to. For example `1>2` equals false, `3<=3` equals true. Equality is tested using `==` (double equals sign). Using a single equals sign in a logical comparison will generally be reported as a syntax error, but can give unexpected behaviors.
10. Entries in arrays are numbered from zero, which is a common programming convention.
11. The line selection tool is on the normal ImageJ toolbar—simply select the tool then click on the start and endpoint for the line in the image. Note there are several line selection tools available (straight line, polyline, etc.) which can be switched between by right clicking on the line selection tool.
12. All multichannel high bit-depth information can be preserved by saving images as tiff files (the default ImageJ format). These are noncompressed and preserve all the image data. Here, we will be analyzing images of this format: A collection of 16-bit images of different channels aligned with each other in a multislice image stack. To save an image for presentation (in a figure or talk), it will normally need to be converted to an 8-bit or RGB color image and saved in a more widely used format—png is widely compatible and has lossless compression.

13. Phase contrast is preferred to either bright field or differential interference contrast (DIC) for the automated identification of cells. Phase contrast provides much higher image contrast than bright field illumination and the directional pseudo-shadow of DIC leads to minimal image contrast perpendicular to the interference contrast axis.
14. The recommended sample preparations for high quality phase contrast images of trypanosomes are either live cells or cells fixed with formaldehyde. Drying, methanol fixation and/or detergent treatment can reduce contrast (by extracting material) or alter cell morphology. For detailed protocols concerning sample preparation for light microscopy see Chapter 23 by Dean and Sunter.
15. The microscopes used for optimization of this approach were Leica DM550b (upright), Zeiss Axio Scope.A1 (upright) and Zeiss Axio Observer.A1 (inverted) widefield epifluorescence microscopes, using standard halogen or LED transillumination and 40 \times , 63 \times , or 100 \times phase-contrast objectives.

Acknowledgments

I would like to thank the members of the Gull, Gluenz, Vaughan, and Sunter labs, interaction with whom have helped select the contents of this chapter. I would also like to thank the TrypTag project team; the TrypTag data set has demanded the refinement of many of the approaches outlined here. Research underlying the methods described here was supported by the Wellcome Trust [103261/Z/13/Z, 104627/Z/14/Z, 108445/Z/15/Z, 211075/Z/18/Z].

References

1. Cranfill PJ et al (2016) Quantitative assessment of fluorescent proteins. *Nat Methods* 13 (7):557–562
2. Stuurman N, Vale RD (2016) Impact of new camera technologies on discoveries in cell biology. *Biol Bull* 231(1):5–13
3. Sahl SJ, Hell SW, Jakobs S (2017) Fluorescence nanoscopy in cell biology. *Nat Rev Mol Cell Biol* 18(11):685–701
4. Power RM, Huisken J (2017) A guide to light-sheet fluorescence microscopy for multiscale imaging. *Nat Methods* 14(4):360–373
5. Caicedo JC et al (2017) Data-analysis strategies for image-based cell profiling. *Nat Methods* 14 (9):849–863
6. Dean S, Sunter JD, Wheeler RJ (2017) Tryp-Tag.Org: a trypanosome genome-wide protein localisation resource. *Trends Parasitol* 33 (2):80–82
7. Huh W-K et al (2003) Global analysis of protein localization in budding yeast. *Nature* 425 (6959):686–691
8. Thul PJ et al (2017) A subcellular map of the human proteome. *Science* 356(6340): eaal3321
9. Werner JN et al (2009) Quantitative genome-scale analysis of protein localization in an asymmetric bacterium. *Proc Natl Acad Sci* 106 (19):7858–7863
10. Collins TJ (2007) ImageJ for microscopy. *Bio-Techniques* 43(1 Suppl):25–30
11. Schneider CA, Rasband WS, Eliceiri KW (2012) NIH image to ImageJ: 25 years of image analysis. *Nat Methods* 9:671–675

12. Rueden CT et al (2017) ImageJ2: ImageJ for the next generation of scientific image data. *BMC Bioinformatics* 18(1):529
13. Schindelin J, Rueden CT, Hiner MC, Eliceiri KW (2015) The ImageJ ecosystem: an open platform for biomedical image analysis. *Mol Reprod Dev* 82(7–8):518–529
14. Schindelin J et al (2012) Fiji: an open-source platform for biological-image analysis. *Nat Methods* 9(7):676–682
15. Shaner NC et al (2013) A bright monomeric green fluorescent protein derived from *Branchiostoma lanceolatum*. *Nat Methods* 10(5):407–409
16. Wheeler RJ (2015) Analyzing the dynamics of cell cycle processes from fixed samples through ergodic principles. *Mol Biol Cell* 26(22):3898–3903
17. Halliday C et al (2018) Cellular landmarks of *Trypanosoma brucei* and *Leishmania mexicana*. *Mol Biochem Parasitol*. <https://doi.org/10.1016/j.molbiopara.2018.12.003>
18. Alonso-Padilla J et al (2015) Automated high-content assay for compounds selectively toxic to *trypanosoma cruzi* in a myoblastic cell line. *PLoS Negl Trop Dis* 9(1):e0003493
19. Dagley MJ, Saunders EC, Simpson KJ, McConville MJ (2015) High-content assay for measuring intracellular growth of leishmania in human macrophages. *Assay Drug Dev Technol* 13(7):389–401
20. Engel JC et al (2010) Image-based high-throughput drug screening targeting the intracellular stage of *Trypanosoma cruzi*, the agent of Chagas' disease. *Antimicrob Agents Chemother* 54(8):3326–3334
21. Eren RO et al (2018) Development of a semi-automated image-based high-throughput drug screening system. *Front Biosci Elite Ed* 10:242–253
22. Moon S et al (2014) An image-based algorithm for precise and accurate high throughput assessment of drug activity against the human parasite *trypanosoma cruzi*. *PLoS One* 9(2). <https://doi.org/10.1371/journal.pone.0087188>
23. Siqueira-Neto JL et al (2012) An image-based high-content screening assay for compounds targeting intracellular *Leishmania donovani* amastigotes in human macrophages. *PLoS Negl Trop Dis* 6(6):e1671
24. Sykes ML, Avery VM (2015) Development and application of a sensitive, phenotypic, high-throughput image-based assay to identify compound activity against *Trypanosoma cruzi* amastigotes. *Int J Parasitol Drugs Drug Resist* 5(3):215–228
25. Cromey DW (2013) Digital images are data: and should be treated as such. *Methods Mol Biol* 931:1–27
26. Woodward R, Gull K (1990) Timing of nuclear and kinetoplast DNA replication and early morphological events in the cell cycle of *Trypanosoma brucei*. *J Cell Sci* 95(1):49–57
27. Wheeler RJ, Gull K, Gluenz E (2012) Detailed interrogation of trypanosome cell biology via differential organelle staining and automated image analysis. *BMC Biol* 10(1):1
28. Peacock L, Bailey M, Carrington M, Gibson W (2014) Meiosis and haploid gametes in the pathogen *Trypanosoma brucei*. *Curr Biol* 24(2):181–186
29. Lukeš J, Wheeler R, Jirsová D, David V, Archibald JM (2018) Massive mitochondrial DNA content in diplomemid and kinetoplastid protists. *IUBMB Life* 70(12):1267–1274
30. Zinchuk V, Zinchuk O (2008) Quantitative colocalization analysis of confocal fluorescence microscopy images. *Curr Protoc Cell Biol* 4:19
31. Haase J et al (2013) A 3D map of the yeast kinetochore reveals the presence of core and accessory centromere specific histone. *Curr Biol* 23(19):1939–1944
32. Deschout H et al (2014) Precisely and accurately localizing single emitters in fluorescence microscopy. *Nat Methods* 11(3):253–266
33. Small A, Stahlheber S (2014) Fluorophore localization algorithms for super-resolution microscopy. *Nat Methods* 11(3):267–279

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.

