Automated Image Analysis of Neurological Tissue Targeted by Gene Therapy

William Burr
wb2117

Tiger Cross
tc2017

Rasika Navarange
rn3717

Arjun Singh
as15217

Joon-Ho Son
js6317

Kelvin Zhang
kcz17

January 2020

Contents

1 Executive Summary

ImageJ is an open-source image analysis tool used for processing scientific images, such as multi-dimensional microscope images. Its graphical user interface is similar to popular image editing software, such as Photoshop or GIMP, but tailored to microscopy and biomedical imaging (Schindelin et al., 2015). Many scientific image processing workflows are enabled by third party plugins built upon ImageJ's highly extensible Java plugin API and batch scripts built upon its internal scripting language. The objective of this project is to create a suite of ImageJ plugins which will automate repetitive ImageJ workflows mostly done manually by our clients, researchers at the University of Cambridge Department of Clinical Neurosciences. The plugins will be used by the clients to investigate new methods in gene therapy with potential clinical applications, such as preventing loss of vision.

Gene therapy is a treatment that can be used to treat genetic disorders and some acquired disorders by delivering DNA into targeted cells via a *vector* carrying the DNA or RNA, typically (and in our specific case) a viral vector. Once the virus enters the cells, the genes carried by the virus are released and can change these cells' behaviour (ideally towards a desired outcome).

A time-consuming process in research evaluating the efficacy of viral vectors is image analysis: from microscope images, researchers have to determine how many cells receive the gene therapy, how much therapy each cell receives, and also whether the correct type of cells have been targeted by the gene therapy. In an experiment, cells which should be targeted by the gene therapy (what we call *targeted* cells) are stained with one colour (e.g. red) and cells which have uptaken the viral vector (what we call *transduced* cells) become stained with another colour (e.g. green). These stains fluoresce under a microscope and researchers can import microscope images of these cells into ImageJ to perform analysis with the help of ImageJ's image processing tools.

However, current automated analysis methods to detect and annotate cells are not tailored towards our clients' experimental needs (e.g. the particular cell staining techniques used and circular shape of the cells). Additionally, current ImageJ analysis plugins require the user to manually pre-process images as plugins do not account for an image having dust/speckles or being too sharp or too blurry. These limitations lead to inaccurate analyses, so each image must instead be manually analysed: researchers must manually identify each red and green cell by clicking on them with a computer mouse. With each image containing hundreds of cells, manual analysis can take tens of minutes per image, with tens to hundreds of images analysed over the course of a research project, incurring a substantial time drain.

Many of ImageJ's analysis plugins have not greatly changed since the software's inception in 1997 (Schneider, W. S. Rasband, and Eliceiri, 2012), hence with advances in computer vision research over the past two decades, especially in the field of biomedical imaging, our clients have identified an opportunity to accurately automate their manual processes. Our project is to develop a suite of ImageJ plugins which apply classical computer vision techniques to automate the following two workflows that our clients frequently use:

- identifying individual cells of round shape and particular size, allowing for cell populations to be quantified; and
- determining whether transduced cells overlap with targeted cells, which directly quantifies the efficacy of a gene therapy experiment.

Our plugins are designed for minimal user interaction, including automation of pre-processing stages described previously. As a result, the plugins reduce the time spent evaluating researchers' experiments from tens of minutes to no more than fifteen seconds per image while matching the level of accuracy of manual human evaluation. This will allow our clients at the University of Cambridge to analyse results of their gene therapy experiments on brain and eye tissue much more easily and efficiently. The developed plugins will contribute to the image analysis process of a research project that aims to determine the best viral vectors to target certain cell types in the eye.

2 Introduction

2.1 Background

Fluorescence microscopy is one important part of gene therapy research which enables researchers to evaluate whether their experiments are effective by ingeniously exploiting microbiological processes: specific proteins can be chosen which fluoresce under different wavelengths of light and target specific types of cells or parts of cells. As described in the prior section, by choosing different proteins which fluoresce under different wavelengths, researchers can infer multiple data points to evaluate their experiments.

Our clients at the University of Cambridge have been performing gene therapy experiments on retinal ganglion cells (RGCs, the target cells) in mice with the aim of determining the optimal viral vector to transduce target cells, where viral vectors differ in their ability to be uptaken by cells and efficiency to activate packaged genes. Our clients are using markers for proteins called RBPMS (Rodriguez, de Sevilla Müller, and Brecha, 2014) and Brn3a (Dunn, Kamocka, and McDonald, 2011) to identify retinal ganglion cells, which causes these cells to fluoresce and are shown in images under the red colour channel. The cell body of retinal ganglion cells contains cytoplasm surrounding a small nucleus; this structure is important for identification of these cells: the marker for RBPMS stains the entire cell body, including the cytoplasm and nucleus, whereas the marker for Brn3a only stains the nucleus. A separate protein, green fluorescent protein (GFP), is introduced by the viral vector, so when a cell is transduced by the vector, the cell also fluoresces, this time under the green colour channel.

In our client's experiments, the red channel of an RGB image taken by a microscope shows retinal ganglion cells, the green channel shows green fluorescent protein which has been uptaken by a cell and occasionally, a third blue marker (DAPI nucleic marker) is used in some experiments which indiscriminately highlights all cells in the image. In general, Brn3a, RBPMS and DAPI are examples of visualising *cell morphology* and GFP is an example of visualising transduction.

Shape is also a factor in identifying retinal ganglion cells in images: images from experiments often have overlapping morphologies such as axons and dendrites, which appear as bright streaks in an image. Retinal ganglion cells appear in our clients' images from experiments with near-circular shape and similar size, making them easy to distinguish from axons and dendrites.

Figure 1: The left two images show the red and green channels of the same image of retinal ganglion cells, the image on the right shows both channels together. The yellow circles identify cells that are both red and green, hence they are retinal ganglion cells that have been transduced as they are also expressing the GFP.

The above images can be analysed using ImageJ by: counting the number of cells in the red, green and (if applicable) blue channels; comparing the number of cells which overlap between the red and green channels; and quantifying the brightness of each cell in the green channel. The following inferences can be made from this data:

- the number of retinal ganglion cells which assimilated the vector is determined by the number of overlapping red and green cells;

- effectiveness of the viral vector targeting depends on comparing the number from the prior bullet point against the total number of retinal ganglion cells from the red channel;
- functional outcome of the viral vector depends on the fluorescent intensity of the green channel; and
- *cell specificity*, where cells which have not been targeted assimilate the vector, are determined by counting the number of green-stained cells which do not overlap red-stained cells.

Third party ImageJ plugins have attempted to achieve the above analyses with minimal manual user interaction, such as ITCN (Image-Based Tool for Counting Nuclei) developed in UC Santa Barbara, and EzColocalization (Stauffer, Sheng, and Lim, 2018). However, these plugins are only semi-automated, requiring the user to manually pre-process the image beforehand (e.g., adjusting brightness and contrast, correcting for uneven background illumination, etc). For ITCN in particular, our clients have reported inaccurate results, where ITCN overcounts the number of cells.

Our clients have also previously used an ImageJ macro to determine colocalization between two colours (Nieuwenhuis, 2019) with brain tissue images, with NeuN-stained neurons' fluorescence appearing under the red channel and eGFP (a similar protein to GFP) (Zhang, Gurtu, and Kain, 1996) fluorescence appearing under the green channel. However, this macro is still semi-automated, still requiring manual measurements to be adjusted for each experiment, such as image processing (i.e. thresholding) parameters and the pixel dimensions of the area of interest. Hence, batch processing a set of images is not possible.

2.2 Objectives

The overall objective of our project is to develop a suite of ImageJ plugins which allow automated analysis of microscope images using classical computer vision techniques for a series of aims given by our client. The plugins must take in either a single image and display results within ImageJ, or batch process a selection of images and save output to a format which can be read by spreadsheet software (such as CSV). A test dataset of input images, microscope images of tissue, is provided for all aims.

The project consists of two main aims. Aim 1 is to create a plugin which automates counting and highlighting of individual cells in an image in order to quantify cell populations. Aim 2 is to measure the colocalization between a target cell colour channel and transduced cell channel in order to make the inferences detailed in the previous section. User experience is a focus for both aims: the plugins must require the minimal user interaction for our clients' experiments on retinal ganglion cells but still maintain configurability for general use cases.

2.2.1 Aim 1 (Simple Cell Counter)

The objective of this aim is to count and highlight individual cells, where an image needs to be pre-processed with minimal human interaction (e.g, removing artefacts such as dust and bubbles) so that computer vision techniques can be accurately applied, and key features such as shape of the cells in our clients' images need to be taken into account. The dataset provided by the client for this aim uses RBPMS staining; with greater surface area of cells stained, there is higher chance of overlap between cells. This aim exists partly to onboard us onto later aims: by overcoming cell overlap in this aim, greater accuracy is expected when dealing with Brn3a-stained images in later aims, which would have lower overlap rates as nuclei have smaller surface area.

2.2.2 Aim 2 (Simple Colocalization)

This aim forms the main goal of this project, measuring colocalization between multiple channels, tailored to the clients' use case of their experiments on retinal ganglion cells. The dataset for this aim is taken from experiments with Brn3a-stained retinal ganglion cells. The required outputs in order to quantify the colocalization are as follows:

- the total number of target cells;
- the total number of target cells overlapping transduced cells;
- (if a third channel is used) the number of cells which overlap all three channels; and
- quantification (median intensity) of each transduced cell.

2.3 Contributions

We have been able to familiarise ourselves with ImageJ and developed a suite of plugins which achieve the above aims: *Simple Cell Counter*, *Simple Colocalization*, and *Simple Batch*. In creating these plugins, we have overcome both technical and user-facing challenges.

Overall, we achieved accurate and reliable results in both aims and have created a significant reduction in analysis time for our clients. Our Evaluation section discusses our results in further detail. Additionally, we meet the objective of providing useful, detailed output in both ImageJ's GUI and spreadsheet-compatible format (CSV).

Simple Cell Counter achieves the goals of Aim 1 described in the prior section. On a technical level, we used this first aim to familiarise ourselves with the strengths and limitations of ImageJ's API, and to investigate several methods of applying computer vision techniques (i.e. image segmentation) which would come in use in all aims. From our technical investigations, we then designed a user-friendly interface and automated image analysis pipeline around the constraints of ImageJ's GUI API. Both user experience and accuracy received positive client feedback, with the plugin automating the time-consuming task of counting individual cells into one which takes less than a second.

Simple Colocalization achieves the goals of Aim 2, identifying and quantifying the amount of overlap of cells fluorescing in different colour channels. This was a more involved task: using our implementation of the first aim and undertaking significant research, we implemented a pipeline which gave fast, accurate analyses, providing data both summarising the overall image and per cell. We also integrated extensive user feedback for this plugin, given the focus of ensuring output data would be useful for evaluating gene therapy experiments.

Simple Batch achieves the user requirement of running the above two plugins on a large quantity of images, where the plugin can batch process a folder containing more than one input images. This supports a wide variety of image formats used by our clients, including LIF and TIFF formats. The output file is simple to read and allows our clients to easily distinguish the output for each file.

Across these plugins, we overcame several other user experience challenges, which had either technical or design complexities, including:

- requiring minimal manual tuning while maintaining configurability, where we performed significant research to distill each stage of our image analysis pipelines into the least number of parameters, and provided sensible defaults to reduce the situations where manual tuning would be necessary;
- taking advantage of ImageJ's plugin distribution system to make the plugin easy to install (discussed further in the Deployment section).

We expect that our application will be useful to our clients in their upcoming research and hope that they will find good use in the wider biological community.

2.3.1 Extensions

There are some elements of our work that were not included in the initial specification provided by the client, but we chose to complete these additional tasks in order to improve the quality of our product. The following are extensions to the brief which we completed:

- Accuracy in non-RGC use cases - For Aim 1 in particular, we have ensured our plugin runs accurately on images provided for other use cases. We were able to tweak our image analysis pipeline so that the plugin performed well on both images of retinal ganglion cells and NeuN-stained brain tissue, which was outside the scope of our project.
- Detailed colocalization output - In addition to displaying the required outputs in our original aims, we worked with the client to display more data which would be useful to their experiments, such as a calculation of the *integrated density* value of every transduced cell.
- XML output - Although the client only requested a CSV output format, we decided to also provide XML output in order to widen the potential usage of our plugin as providing this additional format could be beneficial to other researchers. We chose XML for this purpose due to its ability to openly represent structured data.

3 Design and Implementation

3.1 Overview

We have created a suite of three ImageJ plugins: one to fulfill Aim 1 (Simple Cell Counter), one for Aim 2 (Simple Colocalization) and one for batch processing images on either of the two prior plugins (Simple Batch). The division of our aims into these three separate plugins is intentional: many of the key design choices in the project are based on the constraints and conventions of the ImageJ plugin API. In this particular case, we followed convention to create separate plugins due to how distinct each plugin's inputs and outputs are.

ImageJ supports a wide variety of languages for its API, including Java and Python (Jython), all of which would produce a JAR plugin file for users to load into ImageJ. We decided to use Kotlin for our project, a statically-typed programming language designed to fully inter-operate with Java: with ImageJ being developed in Java, its Java API documentation is much more thorough compared to its other supported languages, and with Kotlin's static typing, conciseness and addition of powerful functional programming constructs (such as higher order functions) would increase our developer productivity.

We also focused on good software engineering practices, particularly structuring our codebase around ImageJ's API. Our code is split into an abstraction provided by the ImageJ API: services and commands. Services allow us to write methods intended to be re-used across images, while commands are designed to be used for one-off operations. As a result, our image segmentation steps for both plugins are encapsulated within services, while distinct code to handle the plugins' graphical user interfaces are provided by commands. This pattern reduces the amount of duplicated code in our codebase and increases its maintainability.

3.1.1 User Experience

The overall challenge on the user-facing side of all of our plugins was to provide an easy to use plugin for our clients which required minimum interaction for their specific use case, yet still allowed them to change parameters to suit other use cases.

For our UI we chose to use the ImageJ GUI API as opposed to implementing our own with, for example, Swing GUI. This was to ensure we developed a UI with a similar style and convention to other ImageJ plugins, reducing the learning curve for ImageJ users. However, the ImageJ API has limited functionality in its straight-forward declarative way of displaying windows with input boxes and buttons. Despite this constraint, we were able to develop user interfaces that were simple, consistent with one another, and consistent with other ImageJ plugins. This allows for a straightforward and friendly user experience.

In order to avoid an overcomplicated UI, we had to decide which image processing parameters would be configurable by the user: as discussed in later sections, achieving our tasks require many image processing steps, each of which require several numeric parameters. We spent a considerable amount of research time deciding: which parameters could be constant and hidden as they would work for most, if not all, expected input images; which parameters could be set to a default setting, optimised for our client's specific ongoing research images; and which parameters needed to be adjusted every time due to variations in input images.

We were able to make the advanced parameters optional due to our extensive research in finding default parameter values for our test images, leaving one numeric parameter for Simple Cell Counter and two numeric parameters for Simple Colocalization for the user to specify or leave as default. Six additional, advanced parameters are configurable by the user in a dialog (Figure 2b) shown only if the user checks a 'Manually Tune Pre-processing Parameters?' checkbox, used for images which differ from the research images we have targeted.

Specific user interfaces, including descriptions of parameters for each plugin, are explained in detail in their respective sections, 3.2.1 (Simple Cell Counter), 3.3.1 (Simple Colocalization) and 3.4.1 (Simple Batch). Overall, we have aimed for uniformity between each plugin, with Simple Cell Counter and Simple Colocalization having the following overall steps within ImageJ:

1. Open an image
2. Open the plugin by selecting `Plugins` → `Simple Cells` in the ImageJ toolbar

3. Configure essential parameters (output method, basic image processing parameters, etc.)
4. (Optional) Configure advanced parameters (primarily for image pre-processing, listed in Appendix A.6)
5. Display output in ImageJ or save to output file (CSV/XML)

We also aim for uniformity and conventional behaviour in line with other ImageJ plugins. For example, both, Simple Cell Counter and Simple Colocalization follow convention by only operating on images which are already open in ImageJ instead of manually opening a file window.

Different user interface approaches and flows were also considered: one alternative would have been to have all of the parameters in the single menu, but this would over-complicated our clients' typical usage and overwhelm them with a steep learning curve. Likewise, we considered hiding advanced parameters under an expandable accordion menu as opposed to opening a new dialog, but this would not have been possible due to limitations in the ImageJ GUI API. Given these considerations, our solution balances well the need for minimal user interaction with high configurability.

3.1.2 Backend Image Processing

Implementing the functionality behind the user interface requires explicit handling of the ImageJ API's representation of images, along with the application of classical computer vision techniques.

In fluorescence microscopy, microscope images contain many characteristics not present in standard image file formats (such as PNGs and TIFFs). Although ImageJ does support standard image formats with RGB colour channels and these formats are commonly used by our clients, our clients also frequently export microscope images in the Leica Image File (LIF) container format. As a result, our plugins support all of these file formats. LIF files add complexity to our backend image processing stages as LIF files contain features such as the following when imported into ImageJ:

- images can have multiple channels, such as a fourth colour channel for far-red wavelengths;
- the container contains one or more *series* of images (as opposed to PNGs and TIFFs which can only contain one series), where each series represents a disjoint section of the cell plate under the microscope, and consecutive series taken can usually be stitched into a larger, single image at the expense of processing time; and
- every series can contain several images which are collectively known as a *Z-stack*, representing the microscope taking images (*slices*) at different depths of the cell plate, therefore capturing image data in three dimensions.

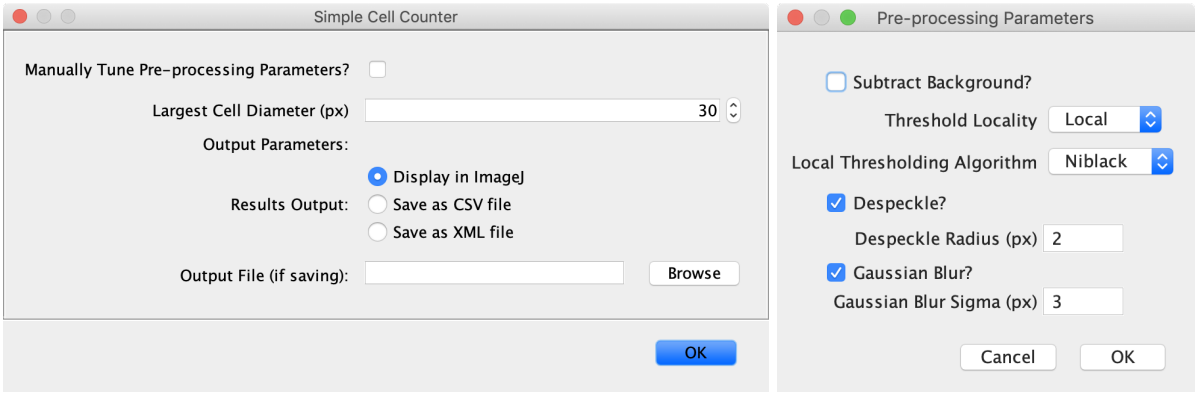
With our clients' input file formats in mind, backend image processing for our plugins consists of the following stages:

1. image extraction, retrieving the necessary channel(s), series and slice from the input image;
2. image pre-processing, cleaning the targeted image channel(s) to improve the accuracy of image segmentation;
3. image segmentation, separating overlapping cells and identifying individual cells from a pre-processed image; and
4. analysis, processing each individual cell and returning the required output(s).

In both aims, the pre-processing and segmentation stages presented the greatest technical challenges: the accuracy of plugin analysis and output heavily depends on accurate cell identification. The following sections cover the different techniques we have investigated in order to arrive at our current solution.

3.2 Simple Cell Counter

3.2.1 User Experience



(a) Initial menu (b) Pre-processing Parameter Menu

Figure 2: Menu screens for Simple Cell Counter

Simple Cell Counter has a simple user interface, hiding the complexity of the backend image processing stages. The initial menu, shown in Figure 2a consists of only the essential parameters. In this case this includes the 'Largest Cell Diameter' specified in pixels. This is clear to the user both what the parameter means and how it is measured. The other essential parameter(s) are for output where each option is presented clearly, and it is emphasised that if saving as a file then an output file location can be specified. Additional user flows for Simple Cell Counter are presented in B.1.

We provide the option to manually tune the other parameters with a clear checkbox at the top. If the user does specify to manually tune parameters then the pre-processing menu, shown in Figure 2b, appears.

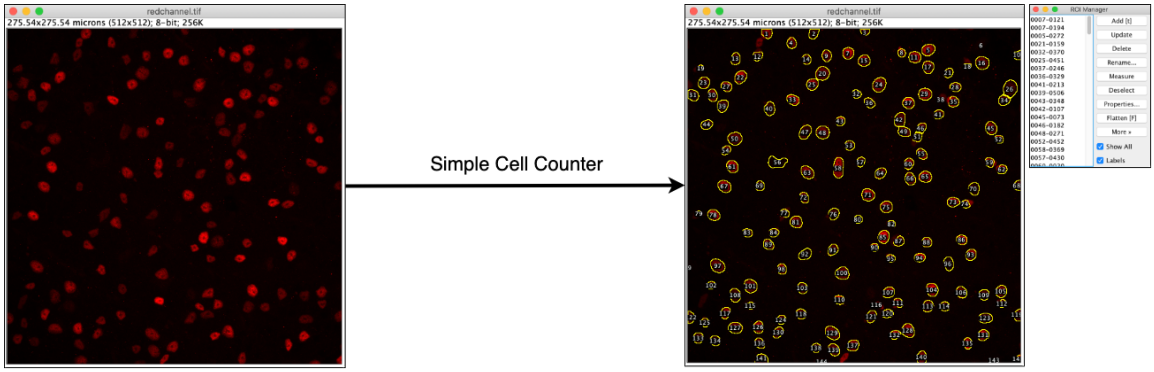


Figure 3: Output by Simple Cell Counter of ROIs on the original image

For output, we provide the option of an ImageJ table for convenience as shown in Figure 4a, as well as the option to save as CSV/XML files which enables further data processing. Figure 4b shows the result of opening our resulting CSV in Excel. Note that regardless of the output format specified, the identified cells will be added as ROIs to the ROI manager of ImageJ and will appear as overlays on the image, as shown in Figure 3.

3.2.2 Image Extraction

Before any image operations can take place, the opened ImageJ image may contain multiple slices. If this is the case, the only image extraction step required in this plugin is to flatten the three-dimensional Z-stack into a standard two dimension image. To do so, we use ImageJ's internal *Z-projection* functionality and following

(a) ImageJ table (b) CSV output opened in Excel

Figure 4: Output for Simple Cell Counter

our client's specification from their manual workflow, we take the *maximum intensity projection* of each pixel, which sets the pixel at a coordinate to the maximum pixel value across each slice in the Z-stack.

3.2.3 Pre-Processing: Thresholding

After the extraction phase, the plugin performs several pre-processing steps on the image to prepare it for segmentation. Pre-processing refers to the transformation of an extracted image into an image consisting of white (non-segmented) cells on a black background. This then allows the software to more accurately segment and identify individual cells. Pre-processing and segmentation are very closely related and a flowchart of the entire pipeline is shown in Figure 5.

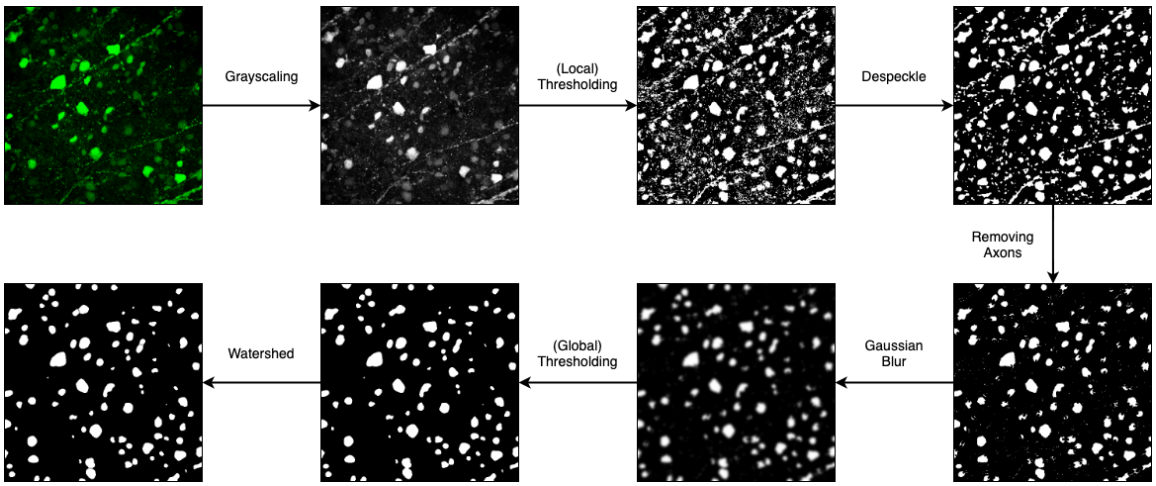


Figure 5: Default segmentation pipeline for the transduction channel with axons

From our client's suggestions and ImageJ image segmentation instructions for general use cases, it was clear that the first pre-processing step needed was to *threshold* the image. Thresholding converts an image to a binarised (black and white) image, where white represents parts of the image that are considered to be in the foreground (i.e. cells), and black represents the background. This categorisation is based on the intensity of the pixels: bright pixels are likely to be cells; dark pixels are likely to be the background. The ImageJ API has several thresholding methods available, hence we opted to use the API. As the API only takes in greyscale images, we convert the input image to greyscale before thresholding it at no cost to accuracy or performance.

At a high level, we had to choose between *local* and *global* thresholding algorithms. Thresholding algorithms work by determining a threshold value at which a pixel would be considered either in the foreground or background based on its value by evaluating the pixel intensities across an area of an image. Global algorithms evaluate the threshold based on intensities across the entire image, while local algorithms evaluate the threshold based on the area surrounding each pixel, set by a radius parameter.

Our plugin initially used global thresholding, however this did not perform well with our test images as more faint cells and clustered groups of brighter cells were not being placed in the foreground layer. This was a result of the background being unevenly illuminated, so the global threshold was too high for darker sections of the image and too low for brighter ones. As a result, we settled for local thresholding, which was designed to support changes in image brightness.

At a low-level, we had to choose the specific local thresholding algorithm out of the many offered by ImageJ's API (*ImageJ Auto Local Threshold Documentation* 2019). We performed qualitative assessments based upon our visual expectations for a well-thresholded image on our clients' test images to determine the optimal algorithm, and found that the Otsu, Bernsen and Niblack algorithms produced near-equally-optimal results. Since our goal is to require the least amount of manual configuration from our clients, we determined that Otsu was the best local thresholding algorithm to use as it did not require any additional parameters, unlike Niblack which takes a k-value and and offset parameter, and Bernsen which takes a contrast threshold as a parameter. Our clients reinforced our decision, expressing a preference for Otsu's algorithm due to positive results using it in the past.

3.2.4 Pre-Processing: Anomaly Removal

Early investigation showed that thresholding by itself was insufficient: certain anomalies were present which were causing inaccuracies, which we resolved by implementing techniques such as background subtraction, median filtering and ridge detection.

The first issue we encountered was that some pixels were being classified as a cell due to uneven background illumination causing speckles to appear in the image after thresholding. We found that our clients overcame this by manually performing multiple background intensity measurements across the image and reducing the brightness of the entire image by the average of these measurements before thresholding (Nieuwenhuis, 2019). However, we needed to automate the process, settling on a combination of two techniques. Firstly, before the thresholding stage, we use a *rolling ball background subtraction* filter (Sternberg, 1983) in ImageJ's API, which reduces background intensity locally over a radius which we set to the expected radius of the largest cell. Secondly, after the thresholding stage, any remaining speckles are removed by applying a *median filter* from ImageJ's despeckling tool (Ferreira and W. Rasband, 2012). This combination heavily reduced the number of false positives with our test images.

Another issue we encountered was that many of our input images contain axons and dendrites which become erroneously counted as cells in the segmentation stage. To prevent this, following the median filter stage, we added a stage responsible for detecting and removing axons and dendrites. As these features are typically linear in nature, we were able to use a *ridge detection* plugin (Thorsten Wagner, 2017), which implements and extends a ridge detection algorithm (Steger, 1998), to select the axons and dendrites. As the image at this stage is black and white, we are able to remove the selected features by painting over them in black, thus preventing them from being counted as cells.

One last issue we noticed was that the the earlier local thresholding stage was overly sensitive, separating individual cell bodies into smaller ones if the pixel intensities within their boundaries had a large variance. By drawing inspiration from (Korath, Abbas, and Romagnoli, 2008) and (Bankhead, 2020), we merge erroneously separated cells by applying a *gaussian blur* (pythonvision.org, 2019), achieving the effect of merging these cells back together, followed by performing a global threshold to remove the blur and recover distinct outlines of cells.

To ensure that these pre-processing steps consistently improved our count when included in our pre-processing pipeline, we performed an extensive investigation which allowed us to evaluate these pre-processing steps and obtain default parameter values our client's research images, which can be found in Appendix D.

3.2.5 Image Segmentation

Once image extraction and pre-processing is complete, the image is at a stage where segmentation can accurately be performed, where operations are performed on the image to separate overlapping cells into cells with distinct outlines. We investigated several segmentation methods, settling on the classical watershed segmentation algorithm (Beucher and Lantuéjoul, 1979, K V et al., 2016), widely used in both generic and fluorescence

microscopy computer vision workflows, which performed well given the effectiveness of our prior pre-processing stages in clearly separating individual cells.

Prior to choosing watershed as our segmentation algorithm, we spent a considerable amount of time investigating methods tailored more towards our image features. One example which closely aligned with our technical specifications was Molnar et al., 2016, tailored to fluorescent, stained cell nuclei, using an active contour model which favours our circular shapes while accounting for slight variations. However, the paper required the pixel intensity of regions to increase proportional to density so that overlapping regions are brighter; this essential feature was not present in our input images, hence this paper's technique could not be used. Likewise, Korath, Abbas, and Romagnoli, 2008's 'dip lining' technique claimed to be more effective than watershed segmentation, however, this technique depended on data being collected before our pre-processing and anomaly removal stages, and the amount of cell overlap and variation in background intensity rendered the dip lining technique to be ineffective.

3.2.6 Analysis

With cells separated in the image by the prior pages, the analysis stage involves retrieving cells from the segmented image and taking the count of cells. Here, we use a tool already provided in ImageJ, the Analyze Particles tool (*ImageJ Analyze Particles Documentation* 2019), used commonly in ImageJ cell analysis workflows. This tool identifies *regions of interest* (ROIs) based on radius and circularity parameters and saves each region of interest in ImageJ's *ROI Manager*. The effectiveness of the tool in identifying cells in the image depends on the quality of segmentation: since we have achieved a well-segmented image, we were able to use this tool for efficient cell detection.

For output, the cell count output is simply obtained from the ROI Manager class, either displaying the count in the user interface or saving to a file.

3.3 Simple Colocalization

3.3.1 User Experience

We intentionally designed Simple Colocalization to be clear and consistent with Simple Cell Counter. The initial configuration menu shown in Figure 6 contains fields to select the target, transduced channels and (if appropriate) the channel for all cells. By our client's suggestion, the user interface calls the target layer 'Cell Morphology Channel 1' and the all cells layer 'Cell Morphology Channel 2' in order to make the terminology more generic for wider use cases.

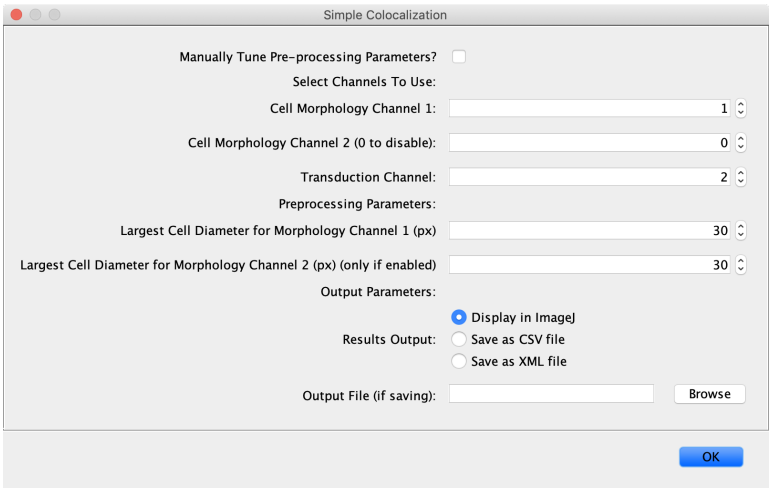


Figure 6: Initial menu screen for Simple Colocalization

As is the case with Simple Cell Counter, we provide the same checkbox offering the tuning of pre-processing parameters which, if checked, leads to the menu shown in Figure 2b. Using the same menu reinforces the consistency within our set of plugins and makes it easier for users to become familiar.

	Label	Count	Area	Median	Mean	Integrated Density	Raw Integrated Density
1	--- Summary ---	134	0	0	0	0	0
2	Total number of cells in cell morphology channel 1	134	0	0	0	0	0
3	Transduced cells in channel 1	10	0	0	0	0	0
4	--- Transduced Channel Analysis, Colocalized Cells ---	0	0	0	0	0	0
5	Cell 1	1	226	11	12	2712	2763
6	Cell 2	1	256	13	13	3328	3489
7	Cell 3	1	206	11	10	2060	2247
8	Cell 4	1	433	9	9	3897	4169
9	Cell 5	1	323	10	9	2907	3145

Figure 7: ImageJ table

The table output (Figure 7) provides both a summary and analysis of each colocalized cell. Regardless of which output type is chosen, the regions of interests representing identified transduced cells (that were targeted) are added to the ROI manager and visible. This is shown in Figure 8.

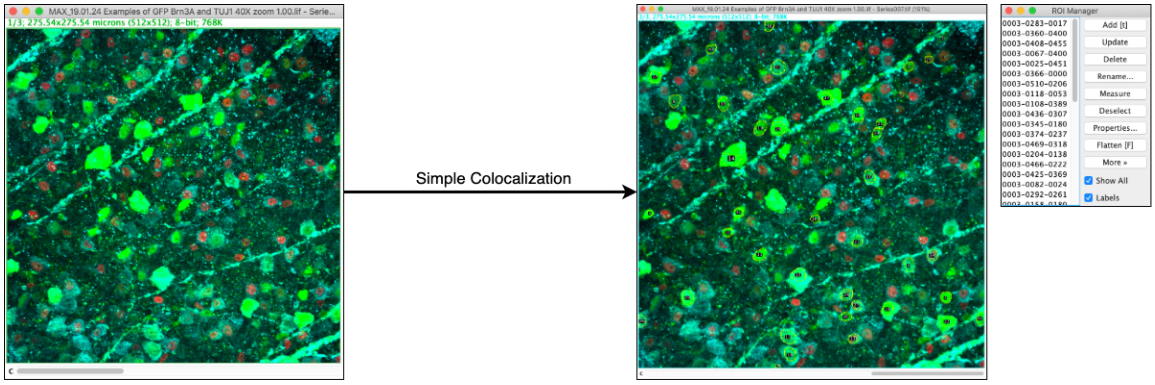


Figure 8: Output by Simple Colocalization of ROIs on the (Z-projected) original image

3.3.2 Image Extraction

As with Simple Cell Counter, input colocalization images may also have multiple slices, hence we again take the maximum intensity projection of each pixel to flatten the Z-stack. Additionally, Simple Colocalization involves the processing of multiple channels, hence the image is split into colour channels and the relevant ones are retrieved based on the channel numbers specified by the user in the configuration menu.

3.3.3 Image Pre-Processing and Segmentation

Simple Colocalization involves the same aim of Simple Cell Counter of identifying individual cells from a channel. With the need to identify individual cells across the target, transduced and all cells channels, the same pre-processing and segmentation pipeline as Simple Cell Counter is separately applied on each of these three channels.

There is one additional stage after the cell identification stage for the transduced channel which is called *intensity filtering*. This was added because few cells may fluoresce in the transduced layer due to ambient light without actually being transduced. Here, the median intensity for each cell is retrieved and only cells with the top specified percentage of values is kept, reducing the number of false positives in the later colocalization analysis stage. This parameter is selected so that very few cells are removed from the list of transduced cells as prior pre-processing steps also reduce the occurrences of these false positives.

3.3.4 Analysis

In order to meet the goals of Aim 2 (Simple Colocalization), once individual cells across channels are accurately identified, we first apply an algorithm to identify cell colocalization across two channels (such as the target and transduced channels); the overarching algorithm is presented in Algorithm 1.

Algorithm 1: Naive Colocalization

```
// threshold is the minimum percentage of overlapping pixels for a cell to be
   considered colocalized.
threshold ← 0.95
colocalizedCells ← initialize empty list
for transducedCell in transducedCells do
    for targetedCell in targetedCells do
        overlap ← size of intersection of pixel coordinates for transducedCell and targetedCell
        total ← size of pixel coordinates for transducedCell
        if (overlap / total) > threshold then
            append(colocalizedCells, transducedCell)
        end
    end
end
return colocalizedCells
```

ImageJ's API allows the required data for this algorithm to be retrieved easily: each cell is stored in an instance of ImageJ's `ROI` (region of interest) class, which contains a set of `Point`s storing the coordinates of each pixel belonging to the region of interest. In order to reduce the inter-dependence of our codebase on ImageJ's API and to increase testability, we use the adapter pattern, mapping each region of interest into our own lightweight data class representing a set of pixel coordinates, and perform the algorithm on our own internal class.

During the implementation stage, we discovered a challenge with the above algorithm which had to be overcome: the time complexity of this algorithm is cubic, with polynomial time set difference having to be performed on the pixels of every pair of transduced and targeted cells. This caused the naive algorithm to take up to thirty minutes to run on our images, as our images have dimensions in the thousands of pixels and contain hundreds of cells, with each cell consisting of tens to hundreds of pixels.

Our initial attempt at increasing the performance of this algorithm was to replace the polynomial time technique of comparing the amount of pixel overlap between two cells with a constant time technique: instead, the coordinate centres of every cell would be pre-processed and a pair of cells would be considered as colocalised if the distances between the centres fell below a particular threshold distance. However, this solution did not work well for our test images: the threshold was either too small, causing too many false negatives (i.e., cells which should have been reported as colocalised were not), or the threshold was either too large, causing too many false positives given the presence of smaller than average cells which were close to another cell but not overlapping.

Instead, the optimisation we made was to keep the technique of comparing pixel overlap between a pair of cells and to reduce the search space of targeted cells. Only target cells neighbouring a transduced cell would be compared against, as opposed to unnecessarily comparing against target cells which would be nowhere near the transduced cell in the image. The heuristic we used to achieve this was to divide the image into squares of fifty by fifty pixels (*buckets*) and only consider target cells which share buckets with the transduced cell. By implementing functions which map between the coordinates of a pixel and the indices of the bucket which contains the pixel, we can create a map from each bucket to a list of target cells which contain at least one pixel which falls under the bucket. Then, instead of looping through all target cells in the algorithm above, only target cells whose pixels fall within the transduced cell's buckets need to be compared against. This is shown in Algorithm 2.

With this improved algorithm, the pre-processing performed means that for every transduced cell, comparisons would only need to be made against at most ten targeted cells, as opposed to hundreds of targeted cells. This led

to the time taken for colocalization to reduce from fifteen minutes to less than five seconds on our test images, meeting the requirement that colocalization had to be performed quickly without loss of accuracy.

We use this algorithm for analysis and output in two ways: firstly, once the colocalised target and transduced cells are identified, the pixel colour values for each colocalised transduced cell is processed and the relevant pixel values (as shown above in the user flow); secondly, if the user states that the image has a channel which marks all cells on a plate, the colocalization between this channel and the prior two channels is analysed using the same algorithm and the total number of cells overlapping across all three channels is also returned to the user.

Algorithm 2: Bucketed Colocalization

```
// buckets is a two-dimensional array, where buckets[0][3] corresponds to the
   top-left coordinates of the bucket in the input image, (0 * 50, 3 * 50).
   buckets[0][3] contains a hash set of all targeted cells which has a pixel in this
   square.
buckets ← initialise two-dimensional array of hash sets corresponding to fifty-by-fifty pixel squares of
 the input image
for targetedCell in targetedCells do
    for pixel in targetedCell.pixels do
        add targetedCell to the bucket corresponding to the pixel's coordinates
    end
end

threshold ← 0.95
colocalizedCells ← initialize empty list
for transducedCell in transducedCells do
    transducedBuckets ← get list of bucket indices which transducedCell's pixels fall within
    neighbouringTargetedCells ← initialize list
    for transducedBucket in transducedBuckets do
        append(neighbouringTargetCells, bucket[transducedBucket.i][transducedBucket.j])
    end
    for targetedCell in neighbouringTargetedCells do
        ... // for loop unchanged from previous algorithm

    end
end
return colocalizedCells
```

3.4 Simple Batch

3.4.1 User Experience

Simple Batch meets the user need of running Simple Cell Counter and Simple Colocalization either on multiple files, or files with multiple series, with minimal user interaction. In general, ImageJ convention is to implement file batching as a *macro*, a script that uses ImageJ's own macro language (*ImageJ Macro Language* 2019), however, since a macro does not support user interaction with a GUI or the ability to interface with our existing codebase, we decided to implement this as a plugin, with user experience in mind. We kept the user interface, shown in Figure 9, for the Simple Batch plugin consistent with the previous two plugins so that users would be familiar with the interface, thus improving the user experience.

The output for Simple Batch running Simple Cell Counter (Figure 10a) is consistent in structure with the output for Simple Cell Counter (Figure 4b, enforcing consistency and familiarity for the user. However, the output from running Simple Batch on Simple Colocalization (Figure 10b) is intentionally not consistent with the output of Simple Colocalization on one image. This is because the output for one image includes per-cell quantification of intensities but this is omitted from the batch output as each row has to instead represent a file and it is not possible to include a third dimension in CSVs.

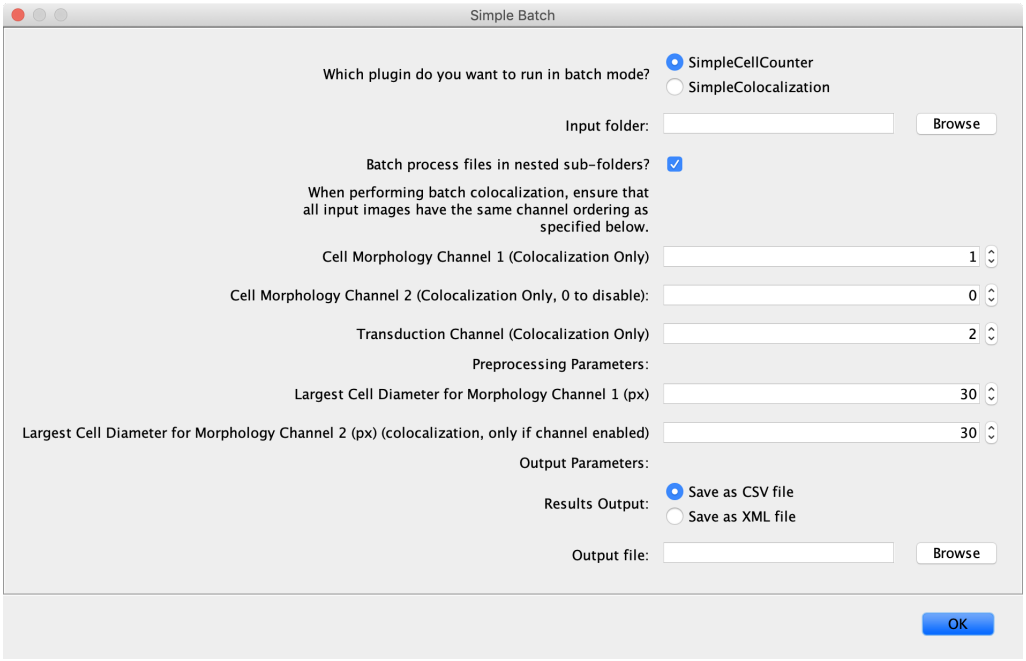
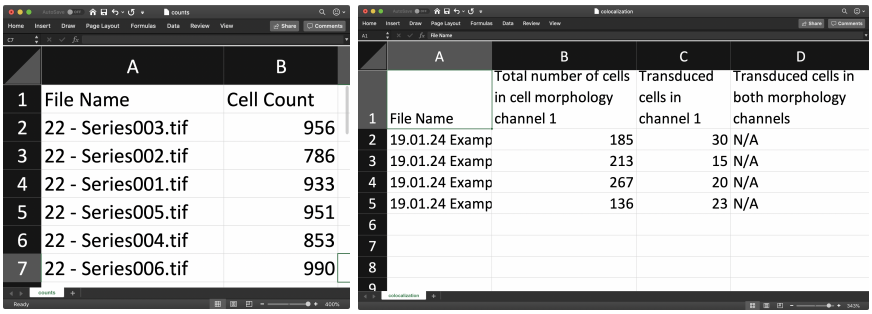


Figure 9: Menu for Simple Batch



	A	B
1	File Name	Cell Count
2	22 - Series003.tif	956
3	22 - Series002.tif	786
4	22 - Series001.tif	933
5	22 - Series005.tif	951
6	22 - Series004.tif	853
7	22 - Series006.tif	990

(a) Batch Cell Counter CSV output opened in Excel

	A	B	C	D
1	File Name	Total number of cells in cell morphology channel 1	Transduced cells in channel 1	Transduced cells in both morphology channels
2	19.01.24 Examp	185	30	N/A
3	19.01.24 Examp	213	15	N/A
4	19.01.24 Examp	267	20	N/A
5	19.01.24 Examp	136	23	N/A
6				
7				
8				
9				

(b) Batch Colocalization CSV output opened in Excel

Figure 10: Output for Simple Colocalization

3.4.2 Implementation

When planning the implementation of this plugin there were several core challenges we faced which influenced the decision between plugin and macro briefly discussed above. The main challenges we had to overcome were:

- handling multiple file-types, including some which ImageJ alone can't open;
- how to make effective use of the Simple Cell Counter and Simple Colocalization service plugins, applying good software engineering principles; and
- the best way to collate the input from the plugins after running multiple times.

Unlike prior plugins, input images are not already opened, hence file handling needed to be implemented. In particular, ImageJ's original distribution does not handle proprietary image formats such as LIF files. In order to deal with this caveat, we pass the responsibility of file handling to the third party *Bio-Formats* plugin instead of re-implementing its complex file handling functionality ourselves and require the user to already have this plugin installed. File handling then becomes simple, looping through the contents of a folder recursively.

In order to batch process images using plugins we had already implemented, we wrapped the existing Simple Cell Counter and Simple Colocalization plugins in new `BatchableCellCounter` and `BatchableColocalizer` classes (see UML diagram in Appendix C). We then created a command plugin class, `SimpleBatch`, as the interface to this functionality. This structure means that we do not need to duplicate code, especially given we implemented ImageJ's command and service plugin abstraction which encourages code re-use.

Our clients requested that we provide the output of the batch plugins into a single CSV file, hence we had to collate the results for each run of the single-run plugins into a single file. We had already created output classes for the previous two plugins, so with the flexibility we had from developing Simple Batch as a plugin instead of a macro, we simply modified these output classes to be compatible with both single-run and batch contexts in order to achieve our goal.

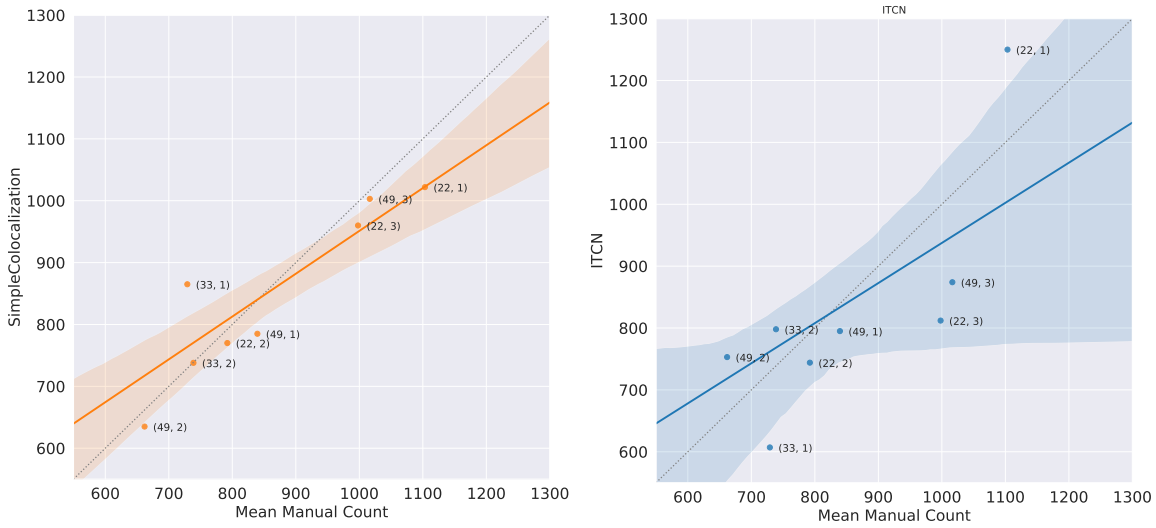
3.5 Deployment

Plugins in ImageJ are distributed as JAR files. While it would minimally suffice to provide a simple download link to this file, or require the user to compile it from source, ImageJ introduces the concept of 'update sites' as the recommended way for ImageJ plugins to be distributed - streamlining the process of plugin installation and usage. The latest version of a plugin, or package of plugins, are uploaded to the developer's update site. End-users can choose to enable specific update sites for plugins that they wish to use, which are managed by ImageJ's updater which manages updates and resolves dependencies. Our suite of plugins are distributed on the `https://sites.imagej.net/Sonjoonho/` update site. The latest version of our `master` branch is built, tested, and uploaded to the update site on each new release, forming our continuous integration and deployment pipeline - ensuring that our users will always have the latest version of the plugin.

4 Evaluation

4.1 Performance

4.1.1 Aim 1 (Simple Cell Counter)



(a) Simple Cell Counter

(b) ITCN

Figure 11: Linear regression plots of manual counts vs. plugin counts

In order to quantitatively assess the performance of our cell counting plugin, we benchmarked its performance against manual counts, collected by members of the team (non-experts). For comparison, the same tests were run using a third-party plugin called ITCN (see Background). By doing this, we are able to statistically demonstrate that our work improves on previous methods. The full analysis, including methodology and

exploratory data analysis can be found in the project repository under `notebooks/simple_cell_counter.ipynb`. The dataset used in this analysis consists of nine images of RBPMS stained brain tissue. These comprise the first three series of image sets `22.lif`, `33.lif`, `49.lif`. We will use the notation `(Image Set, Series)` to refer to these images.

From the manual and automated counts, we can perform a simple linear regression analysis to quantify the performance of each of the plugins. If a plugin works perfectly, we would expect the computed counts to match the manual counts exactly, and for the data points to lie on a straight line $y = x$. Therefore, we can measure the deviation from this line to assess accuracy. Visually, we can see the differences between the two models by plotting them and comparing their errors against a reference line, as shown in Figure 11. Each point represents an image, the solid line is the regression fit, and the band lines show the 95% confidence interval generated for the estimate. Immediately, we can see that Simple Cell Counter has a better fit to the reference line, and has a consistently smaller error. Some notable points are `(22, 1)` and `(33, 1)` for which ITCN performs poorly, while Simple Cell Counter gives a remarkably good result for `(22, 1)`.

To analyse these numerically, we first need to determine whether or not the difference between the two linear regression models is statistically significant. This can be done by performing a t-test on the regression of Simple Cell Counter and ITCN. The regression line can be expressed as

$$Y = \beta_0 + \beta_1 X$$

where B_0 is a constant and B_1 is the slope. From this, we can formulate the hypotheses

$$H_0 : B_1 = 0$$
$$H_1 : B_1 \neq 0$$

The null hypothesis states that the slope is equal to zero, and the alternative hypothesis states that the slope is not equal to zero. We take the significance level $\alpha = 0.05$. This yields a p-value of 0.117. Since $p > 0.05$ we do not have enough evidence at 5% significance level to reject the null hypothesis. Hence, we can say that there is a statistically significant difference between the performances of Simple Cell Counter and ITCN.

Plugin	Regression	Standard Error	p-value
Simple Cell Counter	$y = 0.692x + 259.067$	0.12	0.00068
ITCN	$y = 0.648x + 289.15$	0.263	0.04323

Table 1: Results of linear regression analysis for each plugin

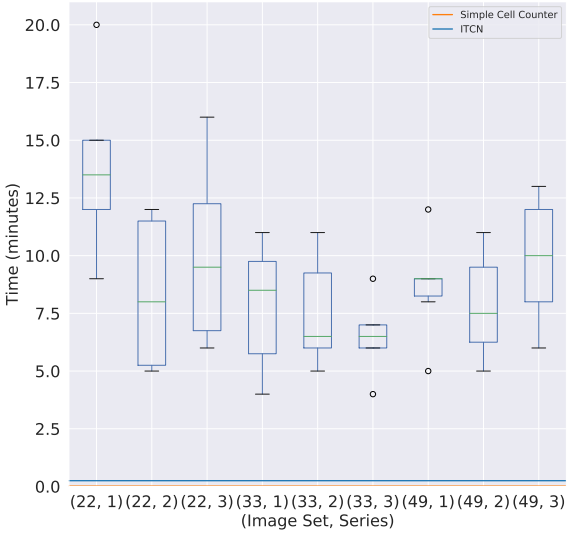


Figure 12: Box plot of time taken for manual vs. automated counts

An important aspect to the usability and convenience of using an ImageJ plugin is the time it takes to run. Cell counting is a tedious process so reducing this time was one of our core goals. In Figure 12, the rectangle represents the second and third quartiles, the middle line is the median, and the whiskers show 1.5 IQR. Outlier points are plotted individually. By plotting the time to take manual counts, we can see that it took as long as 20 minutes to count a single image. ITCN runs in less than 15 seconds, and Simple Cell Counter in under a single second, opening the door to processing large batches of images in a short amount of time.

4.1.2 Aim 2 (Simple Colocalization)

We followed a similar approach of comparing manually collected counts against those computed by the plugin when evaluating the colocalization component. In addition to just looking at the counts, we also evaluated how accurately the plugin measured GFP intensity in these cells. The full analysis, including methodology, can be found in the project repository under `notebooks/simple_colocalization.ipynb`. The dataset used in this analysis consists of six images of retinal tissue stained with Brn3A and GFP. These differ to the images used in the analysis of Simple Cell Counter in that they consist of multiple colour channels and Z-slices. They comprise the first two series of image sets `19.01.15.lif`, `19.01.24.lif`, `19.04.09.lif`, and we will use the same `(Image Set, Series)` notation to refer to them.

Unlike the analysis for Simple Cell Counter, we opted not to do a direct comparison of counts to another plugin. The primary reason for this is that it would be difficult to determine the exact combination of parameters which would provide an optimal result for that plugin, so it may not be a fair test. Moreover, our clients have now provided a quantitative target to assess the outcome of the project. Specifically, they recommend setting 10% as the error margin within which we should be satisfied with the accuracy of the plugin.

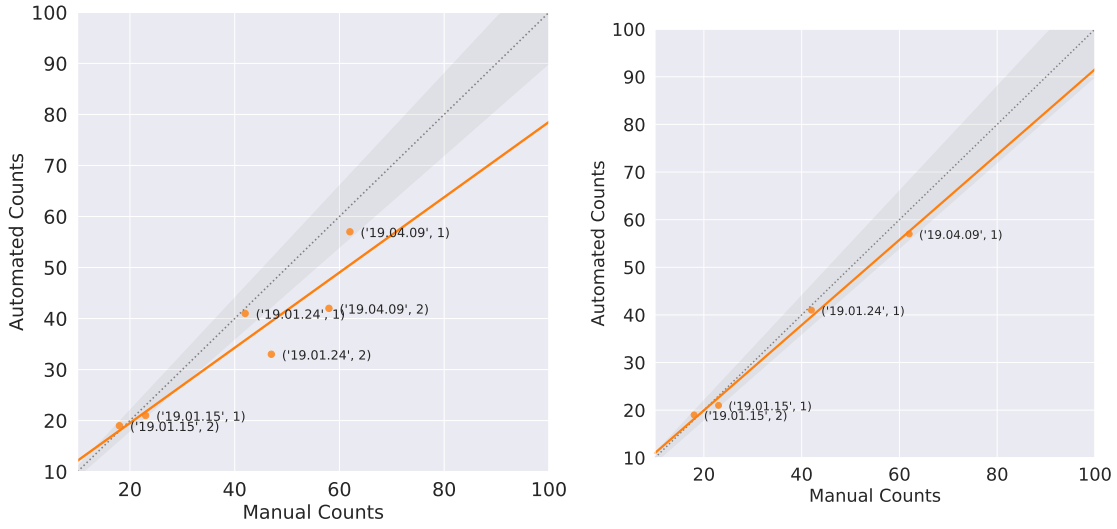
Figure 13a shows the linear regression plot for Simple Colocalization on all images. The translucent bands depict the 10% error margin that we are aiming for. Four out of six counts lie within the desired error range, but two outlier data points lie far outside the range which affects the fit greatly - namely `(19.04.09, 2)` and `(19.01.24, 2)`. From examining these specific inputs, we can see that they have an unusually high incidence of axons and dendrites in the green layer that is not present in the rest of the images. These anomalous structures should not be considered in the colocalization analysis as we are only interested in the transduction of retinal ganglion cells. In our implementation, this is handled by using ridge detection to remove them (described in the Pre-Processing: Anomaly Removal section). However, as a side-effect of this, any transduced cell that coincides directly with one of them will also be removed. This explains the under-count yielded by the plugin on these images. We can consider these images 'edge-cases', and the regression on a subset of images that excludes these can be seen in Figure 13b. This is not to say that these do not need to be handled, but there is value in comparing them to the general case performance.

With the removal of the edge-case images, repeating linear regression analysis gives significantly better performance, with all points lying within the error range. Note also that in order for the regression line to always stay within those bounds, the gradient must be $0.9 < x < 1.1$, which it falls slightly outside of. So the (predicted) relative accuracy will slowly deteriorate with the number of cells in the image.

Image Type	Regression	Standard Error	p-value
All images	$y = 0.738x + 4.765$	0.149	0.00775
Non-edge case images	$y = 0.895x + 2.049$	0.048	0.00285

Table 2: Results of linear regression analysis for each image category

The other metric of interest is GFP intensity. We quantify this by taking the median pixel intensity of each cell in the transduced channel. In theory, if the segmentation is perfect the GFP measurements will also be perfect. Figure 14 shows the mean of the median GFP measurements for each image. The edge case images show poor performance, as is to be expected as the segmentation and GFP are strongly related. `(19.01.15, 2)` is interesting however, as the counting performance was good but the automated intensity measurement remains much higher than the manual measurement. If we look at the histogram of intensities, we can see that the manual counts are more left-skewed, suggesting that the intensity filtering (described in Section 3.3.3) is overly sensitive.



(a) On all images

(b) On non-edge case images

Figure 13: Linear regression plots of manual counts vs. Simple Colocalization counts

The results of the time analysis are very similar to that of Simple Cell Counter, with the manual counting taking several minutes for each image and the automated cell count finishing in under a second.

4.2 User Feedback

Along with software performance, user feedback forms a key component in a successful product development cycle - even more so when building a product with such a niche use-case. During this project, we endeavoured to continuously iterate upon user feedback, collected via regular emails, Skype calls, and an in-person visit to Cambridge. This allowed us to be certain that we were fulfilling the project brief, as well as optimising specific pain-points, ensuring the best possible experience for our clients.

Possibly the greatest change to the users' workflow, and the feature they responded most positively to was the batch processing feature. The ability to process hundreds of images at once is a huge benefit to anyone performing these kind of analyses. As long as performance is consistent, there is potential here to increase the productivity of many researchers. Similarly, the flexibility in output formats was well-received.

Users said that they liked the option to manually tune parameters. This was implemented as a way to provide a high degree of customisability, without overwhelming first-time users, which was something we noticed many ImageJ plugins are guilty of. It was pointed out that these parameters would benefit from explanation, and in response to this we extended our documentation to accommodate this (Appendix A). To facilitate this feature, we needed to provide sensible defaults that would work well straight 'out of the box'. The client praised our choice of values, saying that they worked well with minimal to no tuning required.

Something that we noticed from testing other third-party plugins available for ImageJ was that installation was often tedious, requiring users to manually drag-and-drop JAR files into the appropriate installation folder. To avoid this, we took advantage of the update site feature (Section 3.5) to streamline the process. Users responded very positively to the ease of installation.

A major challenge when developing the plugin was working around limitations of the ImageJ software. For instance, during initial testing users said they found the ImageJ table output confusing. This was due to the fact that ImageJ provides very little flexibility when working with tables which resulted in outputs that could be difficult to interpret. Moreover, columns do not automatically resize to fit their contents, causing text to be difficult to read. Unfortunately, there was nothing we could do about this short of rewriting much of ImageJ's table functionality.

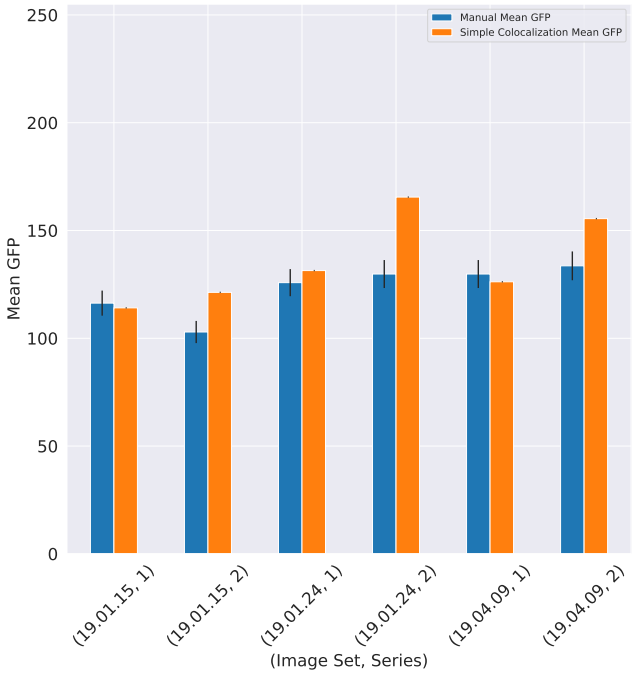


Figure 14: Mean GFPs of manual vs. automated measurements

5 Conclusion

From our evaluation, we can see that the project achieves the quantitative requirements of the aims accurately, with some edge cases in Aim 2, and meets the requirements of user experience and time improvements.

In Aim 1, we have created a plugin which default configuration parameters set such that our clients' images can be processed with: a significant increase in accuracy compared to human counts and existing plugins such as ITCN; and a reduction of counting time from tens of minutes for a human and fifteen seconds for ITCN to under a single second. At most one parameter needs to be configured in a normal use case leading to minimal user interaction required, although a dialog window can be opened for further parameter tuning for different use cases, described in the user guide (Appendix A.6).

In Aim 2, we have created a plugin consistent with the user interface, ease of use and configurability of Aim 1. We have shown that this plugin produces accurate results within an expected margin of error for clients' images where axons do not cover cells; for those which have many axons, our plugin undercounts slightly, although we propose methods of overcoming this in Future Work. We have also shown an improvement in time taken to analyse an image from several minutes for a manual human count to under a second using our plugin.

Finally, we have created a plugin which meets our batch processing requirement, receiving positive feedback from our clients.

In cases where results contain inaccuracies such as undercounts, our plugins are deterministic, hence our results are reliable and reproducible, whereas manual human analyses are subjective and prone to inconsistency.

6 Ethical Considerations

We have undertaken our project in an ethical manner, with a brief analysis of our ethical considerations given as follows.

We provide this plugin without warranty under the MIT License (Choose a License, 2020), which is justifiable as we have taken our best effort to ensure accurate output, having quantitatively measured the accuracy of our plugin against manual counts and other plugins (e.g. ITCN) used in industry, and having performed

statistical evaluation to find that our plugin performs within the expected range of accuracy requested by our clients.

There were also some ethical considerations to be made with regards to the image data sets we were provided by the clients. These images are the property of the client so we were careful to not make these public on our GitHub repository and obtained clear, written permission to safely store these images on our laptops. We were also mindful that we used these images only for the purposes of improving the quality of our plugins.

We are aware that the ethics of gene therapy and animal testing for medical research are debatable, but we believe this to be beyond the scope of the ethical considerations of our project. The animal research by our clients has been regulated under the Animals (Scientific Procedures) Act 1986 Amendment Regulations 2012 following ethical review by the University of Cambridge Animal Welfare and Ethical Review Body (AWERB).

7 Future Work

Further work can be undertaken in our project to improve the accuracy further of our plugins, both for our client's experiments on retinal ganglion cells and general use cases.

Firstly, further parameters can be investigated which could directly improve the accuracy of our current work. One example which our clients suggested in their feedback is to remove erroneously selected small particles by allowing the user to specify a smallest cell diameter parameter in both plugins.

For Simple Colocalization in particular, one method to improve its general accuracy is by calculating more data points for use during colocalization analysis. For example, we can use algorithms which quantify the exact amount of colocalization, Pearson's Correlation Coefficient (PCC) and Mander's Overlap Coefficient (MOC) (Kenneth W. Dunn, 2011) in order to exclude some cells which colocalise weakly due to ambient fluorescence but are still counted in are current plugin.

Additionally, Simple Colocalization could be improved to improve its accuracy when dealing with axons and dendrites. One hypothesis is that a more advanced intensity filtering algorithm would be able to overcome the overall increase in the image caused by bright axons and allow our plugin to detect cells underneath axons better. In the worst case, our plugin could add output to clearly highlight areas with axons which were not processed well for further consideration by the user.

Some more future work possible for this project stems from additional aims provided to us by the client; we agreed with the client that our first two aims would be priority and these additional aims would be out-of-scope. In this case, the plugin would be extended to detect a variety of different cell types for more general research. For example, our clients described to us that in the future, their research they may move on to looking at how their gene therapy affects astrocytes and microglia which do not have a circular shape. This would involve extending the plugin to allow for the user to sample the shape(s) they are looking to detect.

Finally, in the future it would be possible to try an alternative approach to cell counting by using an artificial neural network (ANN) (Weidi Xie, 2016). We can construct a set of 'ground truth' counts for images by asking experts to count cells manually; this image set can be used to train the ANN. The approach using an ANN can be used in more general cases than our current approach, as ANNs are not limited by the constraints of thresholding and watershed algorithms, which rely on cells having clear borders and having a distinct colour to the background. This is not a problem for the images for which we have designed our solution, as per the client's requirements; however, it could have limited effectiveness for other images.

References

Bankhead, Peter (2020). *Introduction · Analyzing fluorescence microscopy images with ImageJ*. URL: https://sydney.edu.au/medicine/bosch/facilities/advanced-microscopy/user-support/ImageJ_FL_Image_Analysis.pdf (visited on 01/03/2020).

Beucher, Serge and Christian Lantuéjoul (Jan. 1979). "Use of Watersheds in Contour Detection". In: vol. 132.

Choose a License (2020). *MIT License — Choose a License*. URL: https://choosealicense.com/licenses/mit/ (visited on 01/02/2020).

Dunn, Kenneth W., Malgorzata M. Kamocka, and John H. McDonald (Apr. 2011). *A practical guide to evaluating colocalization in biological microscopy*. DOI: 10.1152/ajpcell.00462.2010.

Ferreira, Tiago and Wayne Rasband (Oct. 2012). *ImageJ User Guide*. URL: https://imagej.nih.gov/ij/docs/guide/user-guide.pdf.

ImageJ Analyze Particles Documentation (Oct. 2019). URL: https://imagej.nih.gov/ij/developer/api/ij/plugin/filter/ParticleAnalyzer.html.

ImageJ Auto Local Threshold Documentation (Oct. 2019). URL: https://imagej.net/Auto_Local_Threshold.

ImageJ Macro Language (2019). URL: https://imagej.nih.gov/ij/developer/macro/macros.html (visited on 12/22/2019).

K V, Lalitha et al. (Dec. 2016). "Implementation of Watershed Segmentation". In: *IJARCCE* 5, pp. 196–199. DOI: 10.17148/IJARCCE.2016.51243.

Kenneth W. Dunn Malgorzata M. Kamocka, John H. McDonald (Jan. 2011). "A practical guide to evaluating colocalization in biological microscopy". In: DOI: 10.1152/ajpcell.00462.2010. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3074624/.

Korath, Jose, Ali Abbas, and Jose Romagnoli (Apr. 2008). "A Clustering Approach for the Separation of Touching Edges in Particle Images". In: *Particle & Particle Systems Characterization* 25, pp. 143–152. DOI: 10.1002/ppsc.200701107.

Molnar, Csaba et al. (2016). "Accurate Morphology Preserving Segmentation of Overlapping Cells based on Active Contours". In: *Scientific Reports* 6.1. DOI: 10.1038/srep32412.

Nieuwenhuis, B. (2019). "Gene therapy for axon regeneration". PhD thesis. DOI: 10.17863/CAM.41004.

pythonvision.org (2019). *Tutorial*. URL: http://pythonvision.org/basic-tutorial/ (visited on 12/29/2019).

Ridler, T. W. and S. Calvard (n.d.). "Picture Thresholding Using An Iterative Selection Method." In: *IEEE Transactions on Systems, Man and Cybernetics* 8 (), pp. 630–632. ISSN: 00189472. DOI: 10.1109/tsmc.1978.4310039.

Rodriguez, Allen R., Luis Pérez de Sevilla Müller, and Nicholas C. Brecha (Apr. 2014). "The RNA binding protein RBPMS is a selective marker of ganglion cells in the mammalian retina". In: *Journal of Comparative Neurology* 522.6, pp. 1411–1443. ISSN: 00219967. DOI: 10.1002/cne.23521. URL: http://doi.wiley.com/10.1002/cne.23521.

Schindelin, Johannes et al. (July 2015). "The ImageJ ecosystem: An open platform for biomedical image analysis". In: *Molecular Reproduction and Development* 82.7-8, pp. 518–529. ISSN: 1040452X. DOI: 10.1002/mrd.22489. URL: http://doi.wiley.com/10.1002/mrd.22489.

Schneider, Caroline A, Wayne S Rasband, and Kevin W Eliceiri (2012). *NIH Image to ImageJ: 25 years of image analysis*. Tech. rep. DOI: 10.1038/nmeth.2089.

Stauffer, Weston, Huanjie Sheng, and Han N. Lim (Dec. 2018). "EzColocalization: An ImageJ plugin for visualizing and measuring colocalization in cells and organisms". In: *Scientific Reports* 8.1. ISSN: 20452322. DOI: 10.1038/s41598-018-33592-8.

Steger, Carsten (1998). "An Unbiased Detector of Curvilinear Structures. IEEE Transactions on Pattern Analysis and Machine Intelligence". In: 20.2, pp. 113–125.

Sternberg (1983). "Biomedical Image Processing". In: *Computer* 16.1, pp. 22–34. DOI: 10.1109/MC.1983.1654163.

Thorsten Wagner, Mark Hiner (2017). *thorstenwagner/ij-ridgedetection: Ridge Detection 1.4.0 (Version v1.4.0). Zenodo*. DOI: 10.5281/zenodo.845874.

Weidi Xie J. Alison Noble, Andrew Zisserman (Jan. 2016). "Microscopy Cell Counting with Fully Convolutional Regression Networks". In: DOI: https://doi.org/10.1080/21681163.2016.1149104. URL: https://www.robots.ox.ac.uk/~vgg/publications/2015/Xie15/weidi15.pdf.

Zhang, Guohong, Vanessa Gurtu, and Steven R. Kain (Sept. 1996). "An enhanced green fluorescent protein allows sensitive detection of gene transfer in mammalian cells". In: *Biochemical and Biophysical Research Communications* 227.3, pp. 707–711. ISSN: 0006291X. DOI: 10.1006/bbrc.1996.1573.

Appendices

A User Guide

A.1 Installation

The primary and recommended method of installing the plugin is via an update site: `https://sites.imagej.net/Sonjoonho/`. The tutorial linked here (`https://imagej.net/Following_an_update_site`) describes how to use an update site to add plugins to your installation of ImageJ or Fiji. You must have the *Fiji* and *Java 8* update sites enabled in order for this plugin to work. In the Fiji distribution these should be enabled by default, hence Fiji is the recommended ImageJ distribution to use.

Alternatively, the JAR can be compiled using Maven and installed in the `jars/` directory under your ImageJ installation. However, this requires manual installation of dependencies.

A.2 Overview

The plugins can be found in the ImageJ menu under `Plugins` → `Simple Cells`. The three plugins should appear under a sub-menu as `Simple Cell Counter`, `Simple Colocalization` and `Simple Batch`.

Simple Cell Counter and Single Colocalization each process the current image in focus, hence it is required to open the input image in ImageJ before opening the plugin. Results can either be displayed within ImageJ or output to a single file.

In order to batch process a directory containing at least one image, the user may use the `Simple Batch` plugin which allows them to run either the `Simple Cell Counter` or `Simple Colocalization` plugins on several images in batch mode. Results will be output to a single file.

A.3 Simple Cell Counter

To run Simple Cell Counter, first ensure you have an image open and navigate to the ImageJ menu and select `Plugins` → `Simple Cells` → `Simple Cell Counter`.

Simple Cell Counter identifies all the cells in an image, adding them to ImageJ's ROI manager, hence you can use this plugin to automatically mark cells for further analysis by choosing `Display in ImageJ` when selecting the results output. If the input image has more than one channel, all channels will be taken into account during processing. In order to select a specific channel, close the plugin, navigate to `Image` → `Color` → `Split Channels` and bring the desired channel into focus before re-opening the plugin.

It is recommended to adjust the `Largest Cell Diameter (px)` parameter to match the diameter of the largest cell in your image for the most accurate results. If results are not as expected, further parameter tuning will be required (see the bottom of the user guide for advanced tuning of pre-processing parameters).

A.4 Simple Colocalization

To run Simple Colocalization, first ensure you have an image open and navigate to the ImageJ menu and select `Plugins` → `Simple Cells` → `Simple Colocalization`.

Simple Colocalization identifies colocalization between multiple colour channels. Three channels are supported: a first cell morphology channel, an optional, second cell morphology channel, and a transduction channel. The plugin primarily identifies colocalization between the first cell morphology channel and the transduced channel, analysing each colocalised cell and presenting cell intensity data from the *transduction* channel and (if displaying results in ImageJ) adding the colocalised cells from the *first cell morphology channel* to the ROI manager for further analysis. This allows the user to separately analyse the level of transduction on colocalised cells and inspect which of the cells in the first morphology channel have been transduced.

The user can optionally specify a second morphology channel, which could, for example, represent staining of all cells irrespective of whether they are part of the first morphology channel or transduction. If this is specified, the plugin will display the number of transduced cells in both morphology channels.

It is recommended to adjust the `Largest Cell Diameter (px)` parameters for each morphology channels in use to match the diameter sof the largest cell in your image for the most accurate results. If results are not as expected, further parameter tuning will be required (see the bottom of the user guide for advanced tuning of pre-processing parameters).

A.5 Simple Batch

To run Simple Batch, navigate to the ImageJ menu and select `Plugins` → `Simple Cells` → `Simple Batch`.

Simple Batch allows Simple Cell Counter and Simple Colocalization to be run on a folder containing one or files to be processed and outputs a set of results to a single file. If the file contains multiple series, each series will be separately processed. If input images are organised into sub-folders, the plugin will be able to process these nested images as well and explicitly display the image's file path in the image's associated output.

Note: if running Simple Colocalization, ensure that all images have the same channel ordering as the one specified in the plugin options.

A.6 Advanced: Manual Tuning of Pre-processing Parameters

The below table describes the parameters that can be adjusted. ImageJ's documentation, such its overview of segmentation techniques (`http://imagej.github.io/presentations/fiji-segmentation/`), provides useful context to the below parameters.

Parameter	Type	Explanation
Largest Cell Diameter (px)	Number	The approximate diameter of the largest cell in the image. This helps distinguish cell from the image during thresholding.
Subtract Background?	Checkbox	Background subtraction improves the accuracy of the plugin when the input image has unevenly illuminated backgrounds.
Threshold Locality	Dropdown	Choice between global and local thresholding algorithms. Global is faster, but only works well on images with a small variation in cell intensities. Local is slower but more accurate.
Local Thresholding Algorithm	Dropdown	If local thresholding is selected, this gives a choice between different algorithms. Defaults to Otsu's thresholding algorithm.
Despeckle?	Checkbox	Applies a median filter to remove noise and speckles from the image.
Despeckle Radius (px)	Number	Determines the size of the neighbourhood for median filtering. Higher means larger artefacts are removed.
Gaussian Blur?	Checkbox	Applies a Gaussian blur to remove small cell artefacts and merge individual cells which have been erroneously split apart during thresholding back together.
Gaussian Blur Sigma (px)	Number	Radius of the Gaussian blur. Higher means individual cells are less likely to become erroneously split up, but overlapping cells are more likely to be counted as one.

B User Flows

B.1 Simple Cell Counter

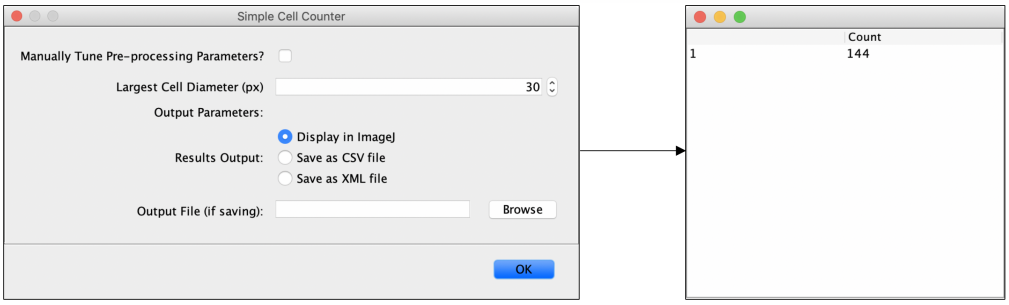


Figure 15: Flow of Simple Cell Counter with output in ImageJ

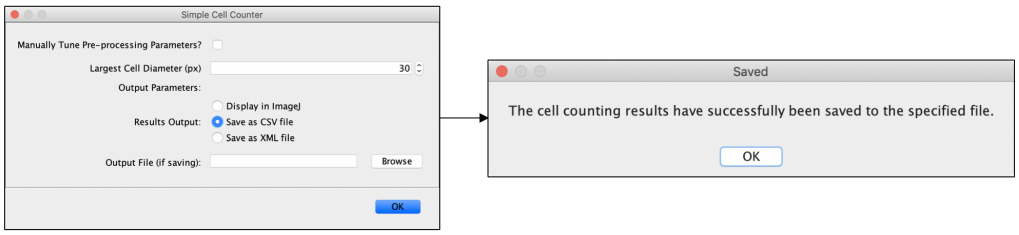


Figure 16: Flow of Simple Cell Counter with saving result as CSV

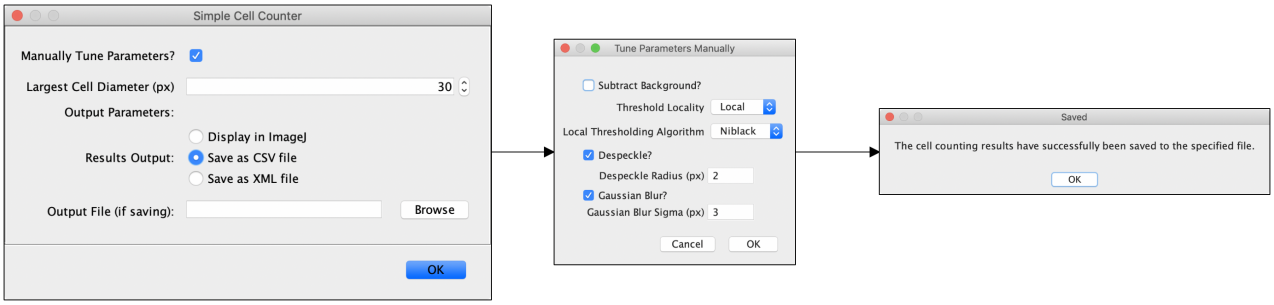


Figure 17: Flow of Simple Cell Counter with manual parameter tuning and saving result as CSV

B.2 Simple Colocalization

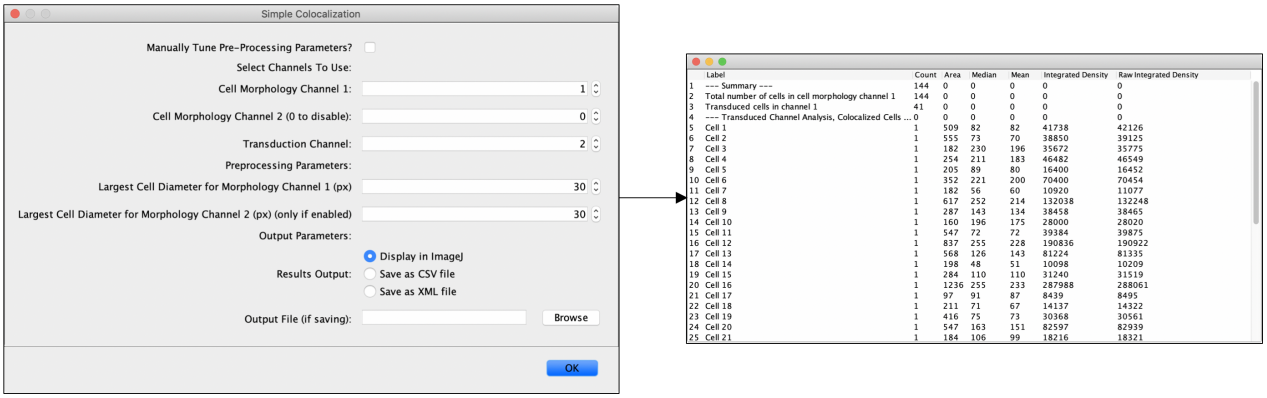


Figure 18: Flow of Simple Colocalization with output in ImageJ

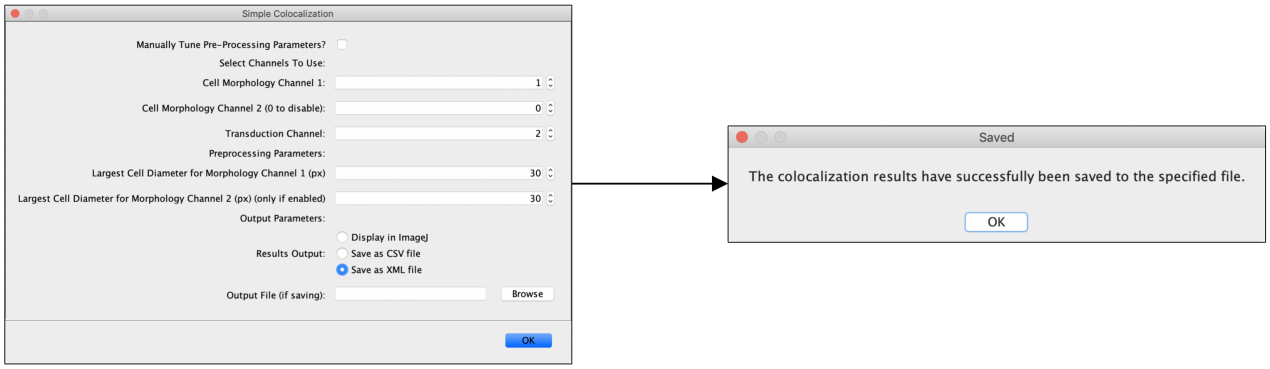


Figure 19: Flow of Simple Colocalization with saving result as XML

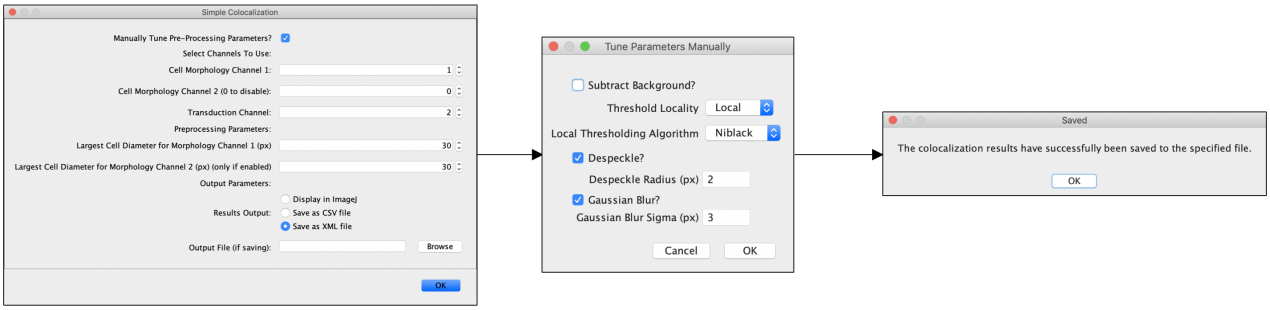


Figure 20: Flow of Simple Colocalization with manual parameter tuning with output as XML

B.3 Simple Batch

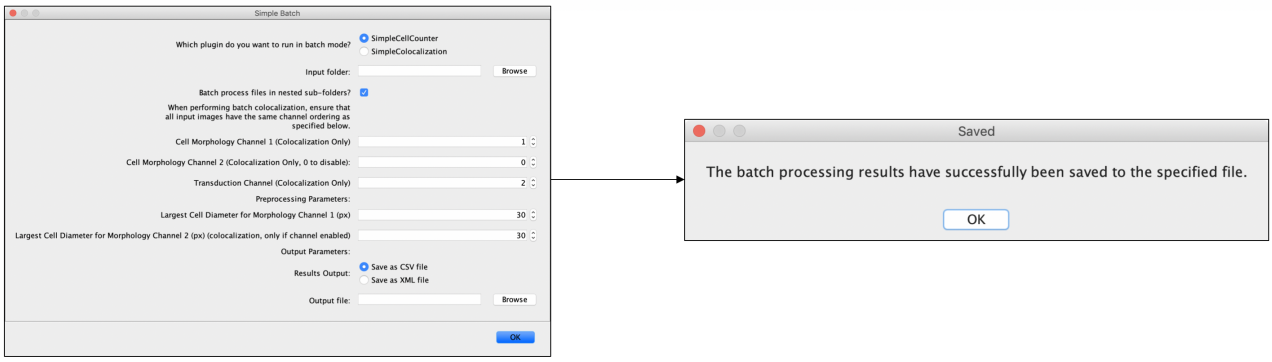


Figure 21: Flow of Simple Batch

C UML Diagram

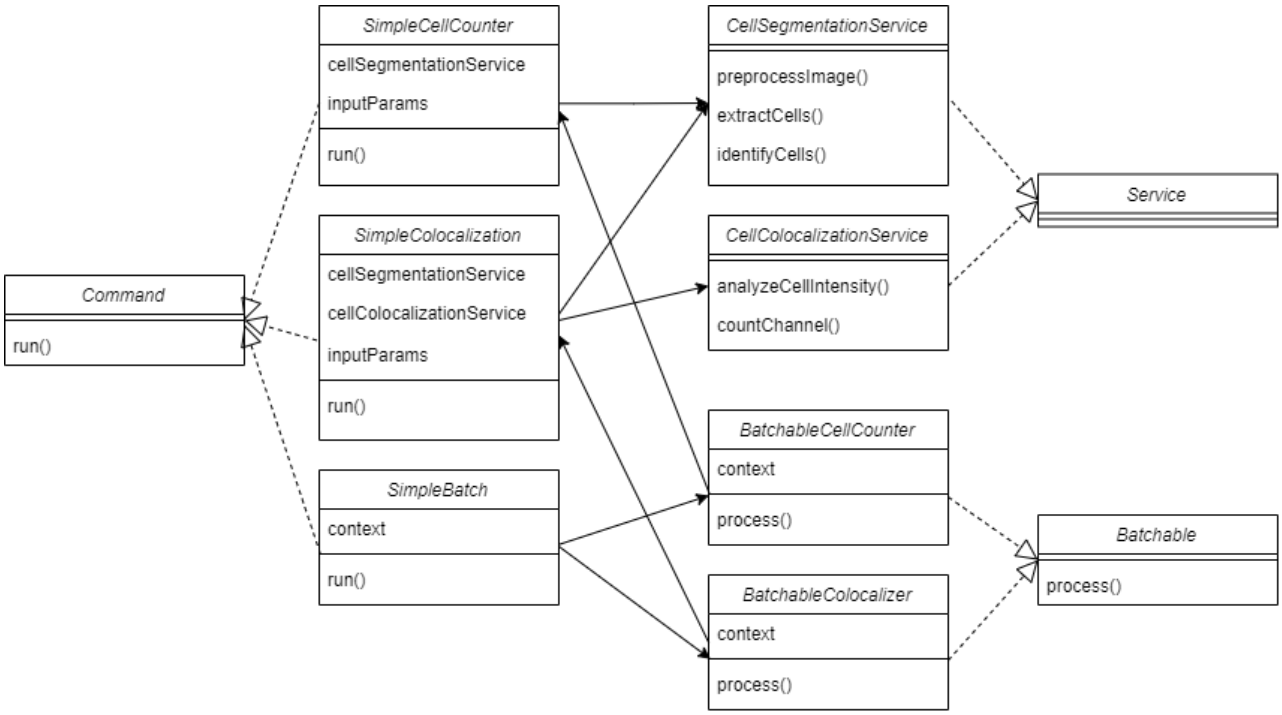


Figure 22: Simplified UML diagram of the class structure of the plugins

The dotted arrows indicate implementation and solid arrows indicate a 'has a' relationship. ImageJ provides interfaces for the two possible types of plugin: Service and Command. Service plugins can be used to implement internal functionality and can be used as utlity classes. Command plugins, however, are used for user interaction in order to create some desired result. Hence, in our design, we have used service plugins to implement the segmentation and colocalization processing. So, our command plugins use these services to process the images input by the user and give the appropriate output.

In order to implement batch processing, we created a `Batchable` interface for the two batchable functionalities of cell counting and colocalization. The Simple Batch plugin is the command plugin which the user interacts with in order to choose which plugin they want to run in batch mode.

D Investigation into Pre-Processing Parameters

D.1 Introduction

The aim of this investigation is to determine the level of abstraction of the parameters from the user, whether they should be immediately available as the parameter is highly image-dependent, able to tweak via an additional dialog (defaults are still efficient for most or all images) or whether they can be made into constants that the user doesn't need to see or change. We have identified a number of pre-processing parameters to find optimal level of abstraction and/or for; these are as follows:

Parameter	Explanation
Background Subtraction Radius (px)	Background subtraction improves the accuracy of the plugin when the input image has unevenly illuminated backgrounds. The radius specified is the neighbourhood around a pixel which should be considered when calculating the local background intensity value.
Threshold Locality	Choice between global and local thresholding algorithms. Global is faster, but only works well on images with a small variation in cell intensities. Local is slower but more accurate.
Thresholding Algorithm	Our client reccomended the use of Otsu, Moments and Shanbhag as candidate algorithms for global thresholding, and we identified Otsu, Niblack and Bernsen as candidate algorithms for local thresholding.
Local Thresholding Radius (px)	If a local thresholding algorithm is specified, the radius specifies the local neighbourhood over which a threshold should be computed.
Despeckle Radius (px)	Despeckling applies a median filter to remove noise and speckles from the image. Radius determines the size of the neighbourhood for median filtering. Higher means larger artefacts are removed.
Gaussian Blur Sigma (px)	Applies a Gaussian blur to remove small cell artefacts and merge individual cells which have been erroneously split apart during thresholding back together. Sigma describes radius of the Gaussian blur. Higher means individual cells are less likely to become erroneously split up, but overlapping cells are more likely to be counted as one.

This investigation is expected to give a range of potential values to use for each pre-processing parameter and further qualitative experimentation will then be needed to settle on exact values.

D.2 Methodology

We separately perform four variations of experiments (background subtraction, thresholding, despeckling and gaussian blurring), each on nine images of NeuN stained brain tissue, compromising the first three series of image sets `22.lif`, `33.lif` and `49.lif`. We use the notation `(Image Set, Series)` when referring to these image sets.

The independent variables for each experiment are identified as follows:

- **Subtract background radius** - background subtraction disabled, followed by radius values from 10px to 60px in increments of 10px.
- **Thresholding** -
  - Global thresholding with Otsu, Moments and Shanbhag.
  - Local thresholding with Otsu, Bernsen and Niblack, each with thresholding radii of 15px, 30px and 45 px.
- **Despeckling** - despeckling (median filtering) disabled, followed by radius values of 1px, 2px, 3px, 5px and 10px.
- **Gaussian Blurring** - gaussian blur disabled, followed by radius values of 1px, 2px, 3px, 5px and 10px.

For each of these independent variables, we fix a set of control variables (of course, excluding the independent variable) obtained from prior qualitative experimentation on the combination of parameters which would identify cells best: background subtraction is set to a radius of 30px; automatic thresholding form ImageJ's *Auto Threshold* plugin, based on the IsoData thresholding algorithm (Ridler and Calvard, n.d.); despeckling with a median filter with radius 2.0px; gaussian blurring with a sigma radius value of 3.0px.

The method for each experiment is as follows:

1. Record the ground truth value for each image series' cell count, taken from an average of six manual human measurements.
2. For each radius/thresholding value, run Simple Cell Count and record the actual cell count.
3. Calculate and record the percentage error compared to the ground truth value.

D.3 Results

D.3.1 Background Subtraction

Investigation of Background Subtraction Radius on 22.lif

	Image Series								
	Series 001			Series 002			Series 003		
Subtract Backround Radius (px)	Actual Cell Count	Expected (Manual) Cell Count	% Error	Actual Cell Count	Expected (Manual) Cell Count	% Error	Actual Cell Count	Expected (Manual) Cell Count	% Error
Disabled	852	1103	22.76%	736	792	7.07%	869	998	12.93%
10	1583		-43.52%	1150		-45.20%	1446		-44.89%
20	1075		2.54%	830		-4.80%	1057		-5.91%
30	1022		7.34%	784		1.01%	1018		-2.00%
40	996		9.70%	770		2.78%	960		3.81%
50	967		12.33%	771		2.65%	946		5.21%
60	962		12.78%	768		3.03%	939		5.91%

Investigation of Background Subtraction Radius on 33.lif

	Image Series								
	Series 001			Series 002			Series 003		
Subtract Backround Radius (px)	Actual Cell Count	Expected (Manual) Cell Count	% Error	Actual Cell Count	Expected (Manual) Cell Count	% Error	Actual Cell Count	Expected (Manual) Cell Count	% Error
Disabled	736	729	-0.96%	582	739	21.24%	566	536	-5.60%
10	1664		-128.26%	1149		-55.48%	1475		-175.19%
20	977		-34.02%	818		-10.69%	751		-40.11%
30	893		-22.50%	780		-5.55%	692		-29.10%
40	865		-18.66%	738		0.14%	686		-27.99%
50	843		-15.64%	735		0.54%	654		-22.01%
60	798		-9.47%	733		0.81%	648		-20.90%

Investigation of Background Subtraction Radius on 49.lif

	Image Series								
	Series 001			Series 002			Series 003		
Subtract Backround Radius (px)	Actual Cell Count	Expected (Manual) Cell Count	% Error	Actual Cell Count	Expected (Manual) Cell Count	% Error	Actual Cell Count	Expected (Manual) Cell Count	% Error
Disabled	660	840	35.10%	591	662	10.73%	932	1017	8.36%
10	1616		-58.90%	996		-50.45%	1421		-39.72%
20	881		13.37%	674		-1.81%	1077		-5.90%
30	832		18.19%	641		3.17%	1032		-1.47%
40	785		22.81%	635		4.08%	1003		1.38%
50	789		22.42%	621		6.19%	1020		-0.29%
60	760		25.27%	628		5.14%	999		1.77%

Figure 23: Investigation into parameters for background subtraction

D.3.2 Thresholding

Investigation of Thresholding Radius and Type on 22.lif

			Image Series								
			Series 001			Series 002			Series 003		
Threshold Locality	Threshold Type	Threshold Radius (px)	Actual Cell Count	Expected (Manual) Cell Count	% Error	Actual Cell Count	Expected (Manual) Cell Count	% Error	Actual Cell Count	Expected (Manual) Cell Count	% Error
Global	Otsu	N/A (Global)	1022	1103	7.34%	779	792	1.64%	1014	998	-1.60%
	Moments		1022		7.34%	779		1.64%	1014		-1.60%
	Shanbhag		1022		7.34%	779		1.64%	1014		-1.60%
Local	Otsu	15	1714		-55.39%	1326		-67.42%	1399		-40.18%
		30	1106		-0.27%	820		-3.54%	979		1.90%
		45	1034		6.26%	783		1.14%	905		9.32%
	Bernsen	15	1395		-26.47%	1011		-27.65%	1231		-23.35%
		30	952		13.69%	739		6.69%	924		7.41%
		45	744		32.55%	629		20.58%	776		22.24%
	Niblack	15	1974		-78.97%	1531		-93.31%	1565		-56.81%
		30	1199		-8.70%	878		-10.86%	1052		-5.41%
		45	1103		0.00%	798		-0.76%	931		6.71%

Investigation of Thresholding Radius and Type on 33.lif

			Image Series								
			Series 001			Series 002			Series 003		
Threshold Locality	Threshold Type	Threshold Radius (px)	Actual Cell Count	Expected (Manual) Cell Count	% Error	Actual Cell Count	Expected (Manual) Cell Count	% Error	Actual Cell Count	Expected (Manual) Cell Count	% Error
Global	Otsu	N/A (Global)	892	729	-22.36%	781	739	-5.68%	682	536	-27.24%
	Moments		892		-22.36%	781		-5.68%	682		-27.24%
	Shanbhag		892		-22.36%	781		-5.68%	682		-27.24%
Local	Otsu	15	1834		-151.58%	1431		-93.64%	1661		-209.89%
		30	952		-30.59%	894		-20.97%	837		-56.16%
		45	871		-19.48%	793		-7.31%	709		-32.28%
	Bernsen	15	1219		-67.22%	1203		-62.79%	1055		-96.83%
		30	872		-19.62%	847		-14.61%	724		-35.07%
		45	666		8.64%	713		3.52%	596		-11.19%
	Niblack	15	1830		-151.03%	1663		-125.03%	1877		-250.19%
		30	1063		-45.82%	985		-33.29%	967		-80.41%
		45	936		-28.40%	855		-15.70%	770		-43.66%

Investigation of Thresholding Radius and Type on 49.lif

			Image Series								
			Series 001			Series 002			Series 003		
Threshold Locality	Threshold Type	Threshold Radius (px)	Actual Cell Count	Expected (Manual) Cell Count	% Error	Actual Cell Count	Expected (Manual) Cell Count	% Error	Actual Cell Count	Expected (Manual) Cell Count	% Error
Global	Otsu	N/A (Global)	820	840	2.38%	635	662	4.08%	1026	1017	-0.88%
	Moments		820		2.38%	635		4.08%	1026		-0.88%
	Shanbhag		820		2.38%	635		4.08%	1026		-0.88%
Local	Otsu	15	1422		-69.29%	1411		-113.14%	1391		-36.77%
		30	901		-7.26%	742		-12.08%	1087		-6.88%
		45	836		0.48%	646		2.42%	1006		1.08%
	Bernsen	15	1159		-37.98%	867		-30.97%	1068		-5.01%
		30	830		1.19%	632		4.53%	603		40.71%
		45	684		18.57%	539		18.58%	349		65.68%
	Niblack	15	1632		-94.29%	1638		-147.43%	1581		-55.46%
		30	998		-18.81%	814		-22.96%	1125		-10.62%
		45	861		-2.50%	679		-2.57%	1044		-2.65%

Figure 24: Investigation into parameters for thresholding

D.3.3 Despeckling

Investigation of Despeckle Radius on 22.lif

	Image Series								
	Series 001			Series 002			Series 003		
Despeckle radius (px)	Actual Cell Count	Expected (Manual) Cell Count	% Error	Actual Cell Count	Expected (Manual) Cell Count	% Error	Actual Cell Count	Expected (Manual) Cell Count	% Error
Disabled	1017	1103	7.80%	791	792	0.13%	1018	998	-2.00%
1	1022		7.34%	784		1.01%	1018		-2.00%
2	986		10.61%	774		2.27%	1008		-1.00%
3	955		13.42%	757		4.42%	977		2.10%
5	858		22.21%	698		11.87%	891		10.72%
10	583		47.14%	555		29.92%	651		34.77%

Investigation of Despeckle Radius on 33.lif

	Image Series								
	Series 001			Series 002			Series 003		
Despeckle radius (px)	Actual Cell Count	Expected (Manual) Cell Count	% Error	Actual Cell Count	Expected (Manual) Cell Count	% Error	Actual Cell Count	Expected (Manual) Cell Count	% Error
Disabled	909	729	-24.69%	793	739	-7.31%	706	536	-31.72%
1	893		-22.50%	780		-5.55%	692		-29.10%
2	874		-19.89%	770		-4.19%	663		-23.69%
3	839		-15.09%	738		0.14%	639		-19.22%
5	728		0.14%	672		9.07%	565		-5.41%
10	498		31.69%	460		37.75%	407		24.07%

Investigation of Despeckle Radius on 49.lif

	Image Series								
	Series 001			Series 002			Series 003		
Despeckle radius (px)	Actual Cell Count	Expected (Manual) Cell Count	% Error	Actual Cell Count	Expected (Manual) Cell Count	% Error	Actual Cell Count	Expected (Manual) Cell Count	% Error
Disabled	840	840	0.00%	645	662	2.57%	1042	1017	-2.46%
1	832		0.95%	641		3.17%	1032		-1.47%
2	816		2.86%	631		4.68%	1017		0.00%
3	791		5.83%	625		5.59%	996		2.06%
5	718		14.52%	598		9.67%	932		8.36%
10	531		36.79%	468		29.31%	661		35.00%

Figure 25: Investigation into parameters for despeckling

D.3.4 Gaussian Blurring

Investigation of Gaussian Blur Sigma on 22.lif

	Image Series								
	Series 001			Series 002			Series 003		
Gaussian Blur Sigma (px)	Actual Cell Count	Expected (Manual) Cell Count	% Error	Actual Cell Count	Expected (Manual) Cell Count	% Error	Actual Cell Count	Expected (Manual) Cell Count	% Error
Disabled	1721	1103	-56.03%	1109	792	-40.03%	1345	998	-34.77%
1	1559		-41.34%	1036		-30.81%	1262		-26.45%
2	1209		-9.61%	869		-9.72%	1116		-11.82%
3	1002		9.16%	779		1.64%	1014		-1.60%
5	759		31.19%	650		17.93%	813		18.54%
10	362		67.18%	392		50.51%	431		56.81%

Investigation of Gaussian Blur Sigma on 33.lif

	Image Series								
	Series 001			Series 002			Series 003		
Gaussian Blur Sigma (px)	Actual Cell Count	Expected (Manual) Cell Count	% Error	Actual Cell Count	Expected (Manual) Cell Count	% Error	Actual Cell Count	Expected (Manual) Cell Count	% Error
Disabled	1835	729	-151.71%	1268	739	-71.58%	1359	536	-153.54%
1	1581		-113.85%	1156		-56.43%	1173		-118.84%
2	1120		-65.84%	903		-22.19%	853		-59.14%
3	892		-37.45%	781		-5.68%	682		-27.24%
5	647		-4.12%	587		20.57%	519		3.17%
10	325		50.34%	319		56.83%	297		44.59%

Investigation of Gaussian Blur Sigma on 49.lif

	Image Series								
	Series 001			Series 002			Series 003		
Gaussian Blur Sigma (px)	Actual Cell Count	Expected (Manual) Cell Count	% Error	Actual Cell Count	Expected (Manual) Cell Count	% Error	Actual Cell Count	Expected (Manual) Cell Count	% Error
Disabled	1631	840	-94.17%	934	662	-41.09%	1598	1017	-57.13%
1	1387		-65.12%	850		-28.40%	1431		-40.71%
2	993		-18.21%	715		-8.01%	1150		-13.08%
3	820		2.38%	635		4.08%	1026		-0.88%
5	663		21.07%	541		18.28%	846		16.81%
10	353		57.98%	331		50.00%	415		59.19%

Figure 26: Investigation into parameters for gaussian blurring

D.4 Conclusion

From the above results, we have made the following observations:

- One of the most useful observations was that the radius for background subtraction and local thresholding both followed a very similar correlation. Following some research into the algorithms for both methods, it became clear that they both implement a form of the 'rolling ball' algorithm discussed in 3.2.4. This means that both values can be represented using a single **largest cell diameter** parameter.
- In all images, setting the **background subtraction radius** to a low value (lower than 20px) causes large percentage errors. There are two instances where percentage error is most reduced in many (but not all) images: turning background subtraction off leads to low percentage error for all images except for (22, 1), (33, 2) and (49, 1); and a value of 30px is associated with one of the lower percentage errors compared to other values in all images except for (33, 1). Setting the radius at a value of 30px or higher leads fairly consistent percentage errors for all images.
- In **global thresholding**, the choice of algorithm does not change the cell count. Hence, the most accurate and fastest global thresholding algorithm would be appropriate if global thresholding is used (With accuracy taking precedence over speed).
- With **local thresholding**, we find that using Bernsen leads to poor segmentation, with no correlation between radius and percentage error (hence the accuracy simply depends on the image). Niblack and Otsu have the lowest percentage error with the larger radius of 45px; Niblack yields marginally better results and Otsu is much slower to run for larger radii, hence Niblack is preferred.
- The percentage error between global thresholding and local thresholding with Niblack at a 45px radius is similar in most images, although there are cases where the global thresholding algorithm noticeably performs better ((33, 2) and (33, 3)).
- In all images, larger **despeckling** radii lead to large percentage errors. In some images (e.g. (22, 2), (49, 2)), the percentage error is lowest when despeckling is off and gently increases with radius; in others (e.g. (33, 1), (33, 2), (49, 3)), the error reduces as the despeckling radius is increased to the 2-5px range, then greatly increases after. A despeckling radius of 1-2px leads to consistently low percentage errors across all images.
- In all images, the percentage error is greatest when **gaussian blurring** is disabled. Error decreases as the sigma value is increased to 3-5px, then increases largely as the sigma value is increased past 5px.

With the above observations, we settle on the following parameters:

1. **Largest Cell Diameter** - This could not be abstracted to have a default value as it is highly dependent upon the image (as different images may have different sized cells). Therefore it is necessary to ask the user to specify this parameter upon each run of the plugin.
2. **Background Subtraction** - None. We discovered that the local thresholding algorithms performed the same 'rolling ball' method to take into account varied background intensity and thus this was only required in unison with a global thresholding algorithm.
3. **Thresholding** - Local thresholding with Otsu's algorithm. Our main reasoning behind this was that this method allowed for the most accurate segmentation which was vital for both aims. Additionally, the percentage errors across thee board were relatively low for this.
4. **Despeckling** - Median filtering with a radius of 1.0px. Increasing the radius to much larger than 2px removed morphologies that could be large enough to be interpreted as a cell. Setting to this value decreases percentage error and ikelihood of
5. **Gaussian Blurring** - Sigma valuel set to 3.0px. We identified this as the ideal value to ensure that we don't blur together overlapping cells but also that we merge separate segments of a single cell together to be identified as just one cell. Further investigations may be required to determine if this value is related to the largest cell diameter parameter