Counting Non-Uniform Cell Sets

1.0 Project Overview

Cellular imaging of neurons is an intensive and time consuming process, but also one that requires a non-trivial amount of skill and training. Typically, a researcher goes through pictures of multiple consecutive layers, referred to as Z-stacks, of the brain to identify cells that have been stained with a variety of fluorescence. The goal of this project is to create an interface that leverages existing open source cell analysis programs in order to rapid prototype a tool that can be customized for Dr. Burke's Lab's needs. The project should be easily maintainable and well documented as well as user friendly.

The original scope of the project revolved around an existing, open-source program called Farsight which had the ability to automatically label and keep track of neurons that, at the time of slicing, were exhibiting a range of behaviors. Additionally, The development of a GUI for lab members to run who may not be as comfortable running the code in its present state was desired. Due to advanced legacy code, that lacked maintenance and little support/documentation available from the original developers, the project was lead to seek out the possibility of implementation through other existing software.

Market research lead to the discovery of another open source image analysis toolkit by the Name of CellProfiler, which is an actively developed suite by the Carpenter Lab at the Broad Institute of Harvard and MIT. CellProfiler has a strong, active community that provides a stable support for developers as well as quick responses from official developers about technical design and specifications. Although CellProfiler is not yet as advanced as Farsight, active, rapid development shows potential to quickly surpass the capabilities of the existing brittle system.

2.0 Acceptance Criteria

Because the program is meant to replace an existing procedure of hand counted data sets, I was able to gain a large set of baseline counts in order to compare my output to. These served as the Baseline Acceptance Criteria for a successful implementation. The primary data set I was comparing against had fourteen unique Z-stacks, the counts of which are shown in Table 2.1. The goal of the program is to be with a 15% error of the below counts. The raw count, under the sum column, is the initial number that is to be met, due to the relative complexity required to calculate and count all variables.

Image set	Sum	Foci	Cyto	Both	Neg
Christian	215	7	12	1	195
Buyout	125	9	1	1	114
Combo	77	13	2	0	62
Phillip	129	2	1	0	126
Problem dog	89	3	1	0	85
Pseudophedrine	134	3	5	0	126
rhee	70	0	9	0	61
Ricen	155	4	0	0	151
Rick	270	10	12	6	242

Table 2.1: Initial Baseline Comparison used to determine accuracy of final program

3.0 Cellprofiler Wrapper

The CellProfiler is an open source software tool for high-throughput experiences with dense data sets. Tools for data visualization and raw data handling allow for it to not only produce useful results immediately but also provide a useful starting point for machine learning. The CellProfiler program functions through the use of pipelines with distinct modules that each provide a functionality to the overall program. These modules act like large scale methods or functions in object oriented programming.

3.1 Building the Cellprofiler Wrapper

The cell profiler wrapper is built through a maven build script. Maven is an apache made java packager. This can be setup through the following link: https://www.mkyong.com/maven/how-to-install-maven-in-windows/.
Once maven is installed and working on your machine, the next step is to pull the git repository. Currently, the git repo is owned by Nick Dicola and can be found at https://github.com/StevenRemington/CellProfilerWrapper. Once the repository is downloaded, the program can be built with the command “mvn install”. This command cleans, compiles, packages, then tests the program. In order to run the cell profiler wrapper, run “mvn exec:java” in a bash terminal from within the git repo.

3.1 The Java Phase Handler

The cell profiler wrapper is a program that orchestrates multiple different phases of the cell counting processes across two different CellProfiler pipelines. The wrapper can be split into five phases: Data Set Collection, Configuration, Final Pipeline, Data Evaluation, and Report Generation. The flow of the wrapper program is depicted in figure 3.0. The Data Set Collection section prompts the user to select whether or not they will be batch processing or processing a singleton then collects the requisite files. The Configuration phase looks at a sampling of the current data set with default settings and attempts to formulate specific settings to apply to the data set. The Final pipeline phase attempts to use the configuration settings to calculate the final counts and output metadata and images of the sets. The Data Evaluation Stage is relatively short and attempts to filter and count any unique cells within the data set. The final stage is report generation where the values of all sets are collected and placed in single report file.

The User, in the beginning, is asked to provide some information about the processing through java GUI prompts. This information includes whether or not the instance will be done in a singleton fashion or a batch process. In addition to this the user is asked if any size modifier values will be used to adjust for similarities within the cell sizes. There is a minimum and maximum modifier that allows you to increase and decrease the bounds of cell size proportionally.

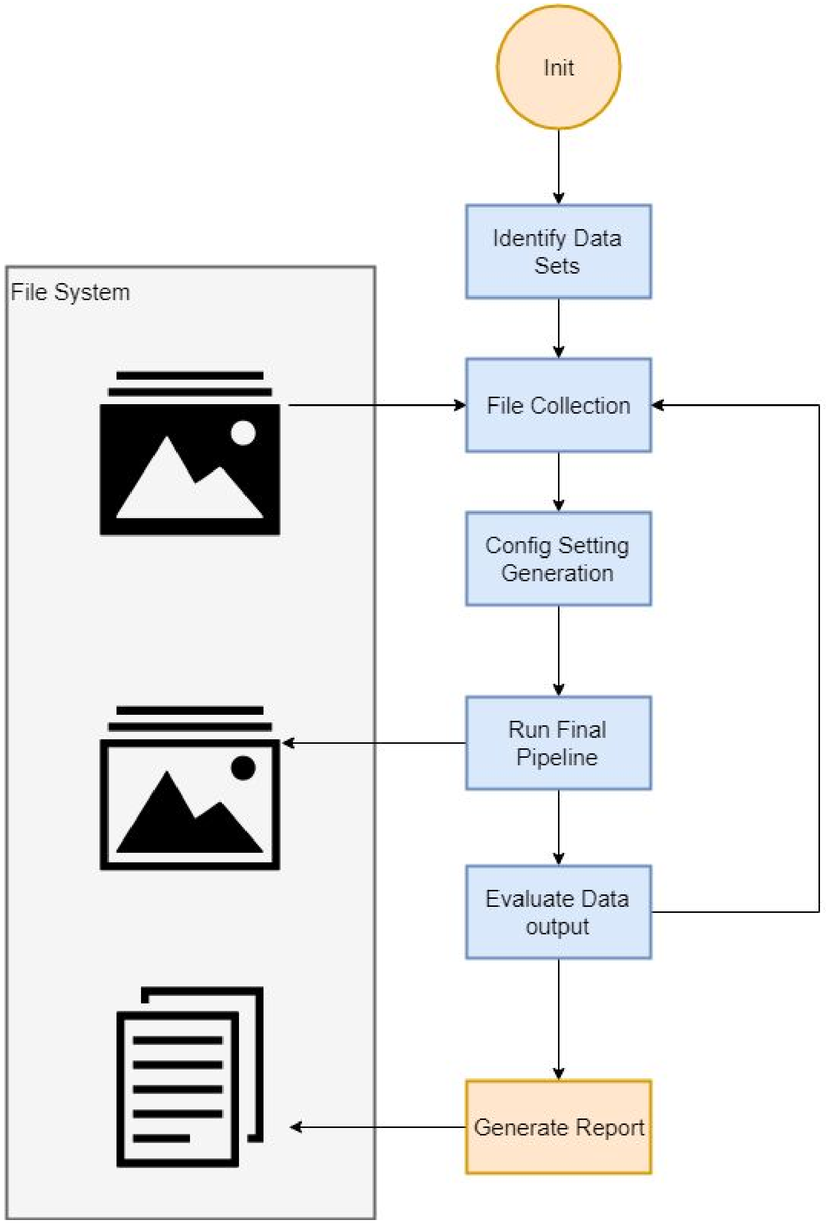


Figure 3.0: The general flow of the cell profiler wrapper.

3.1.1 Data Set Collection

The wrapper program launches with a pop-up window asking users if the job they want to run is a batch job. This lets the wrapper know whether we intend to run CellProfiler on multiple sets of images (batch job), or run on a single image set. With either selection, the user will then see a file selection window appear, where they can navigate to the desired folder for the job. If it is a batch job, the user will select a folder containing subfolders which contain the images. Otherwise, they select a folder which contains the image sets directly. Given the correct directory, the Java wrapper then runs CellProfiler on one image set at a time. The images must conform to the naming convention compatible with parsing settings specified in the CellProfiler configuration pipeline file.

3.1.2 Analysis of the Stack

The analysis of each individual stack is done through the CountingEngine class which manages all the stacks for both singleton and batch processing. Each stack is processed and interpreted within the CountingEngine.analyzeStack(...) method whose flow is shown in figure 3.1. From a high level, each stack creates an instance of a Cell profiler class which is then used to execute the configuration and final pipelines. Once the final pipeline is full completed, the final counts are logged in the cell profiler results object. Currently the infrastructure supports additional types of cell counts, but the pipelines have not yet been updated to generate and process that information.

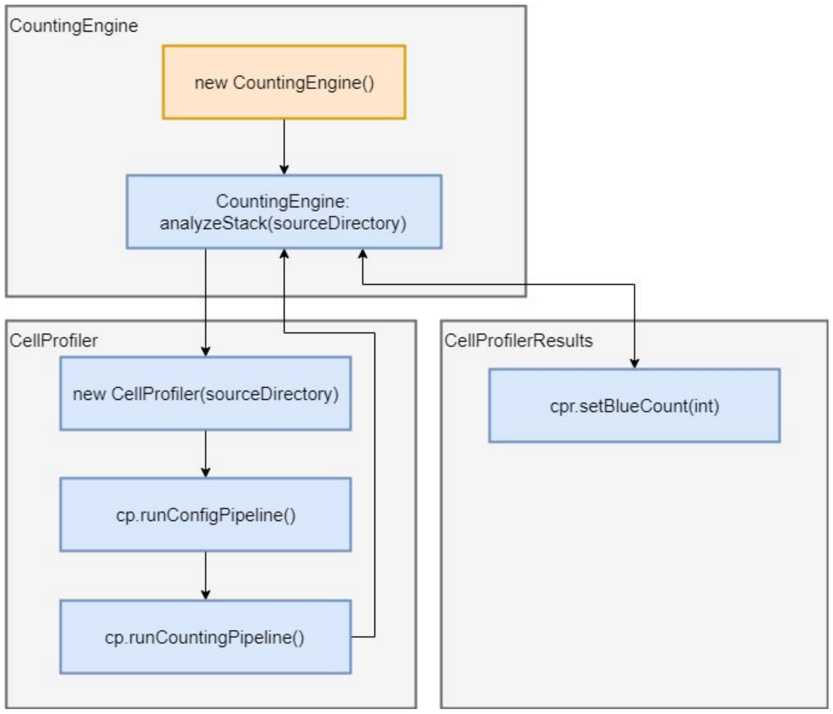


Figure 3.1: An overview of the analyzeStack method flow

The configuration section generates a set of xls files that generate basic contextual information about the data set. Information such as approximate size of cells, average brightness of objects and anticipated lifetime of cells throughout the stack can then be used to generate a specially tailored pipeline for that stack in order to improve the process. Unlike the final pipeline, the configuration pipeline is the same for every stack because a consistent control is needed to evaluate the trends. The current pipeline is shown in figure 3.2. The Identify Primary Objects module attempts to count the main cells within an image by separating intensity channels into bins and performing a watershed algorithm. From this module the objects are measured and tracked through the stacks and the raw information is sent to an excel spreadsheet that will be used to construct a final pipeline.

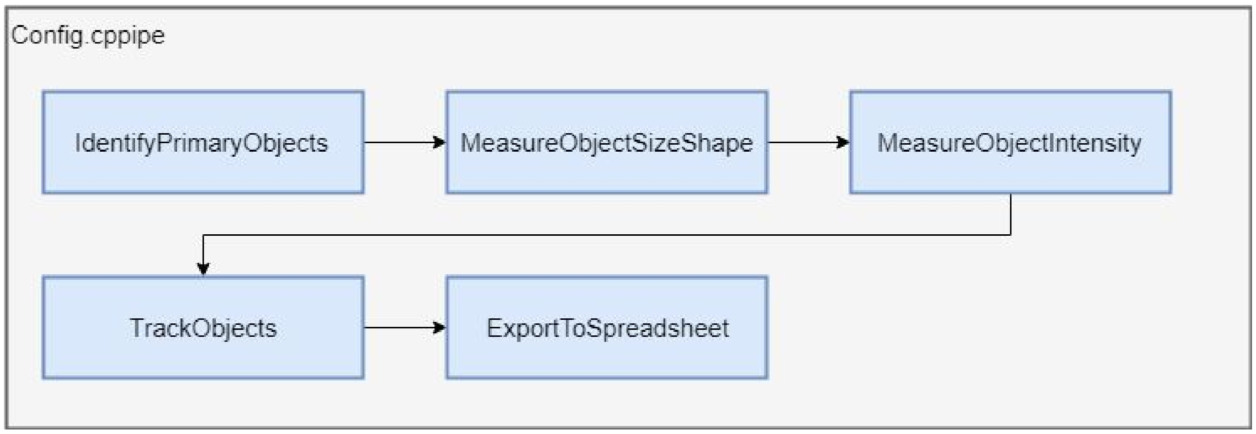


Figure 3.2: The sequence of modules within the configuration pipeline.

After the configuration pipeline, the counting pipeline is constructed then executed in a similar fashion of the configuration pipeline. FIgure 3.3 demonstrates the java portion of the final pipeline counting process. After receiving the results from the configuration pipeline, the PipelineFactory is used to substitute the calculated values with those of the template (marked by @variableName).

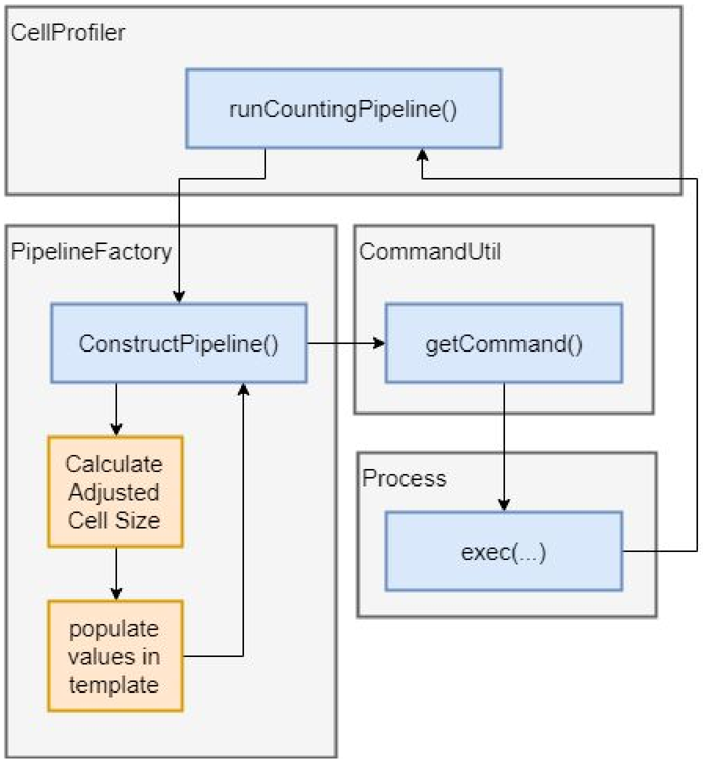


Figure 3.3: The programmatic flow of the counting pipeline.

Once the final pipeline is executed, the path to which is shown in figure 3.3, the Template.cppipe file is generated into the local temporary folder for this execution. The pipeline makes the assumption that the red channel ends with “CH2.tif” and that the blue channel ends with “CH3.tif”, if this is not the case the pipeline will not produce an output. If the assumptions are met then the pipeline starts. The first two modules are heavily related; CorrectIlluminationCalculate generates an illumination function for correcting uneven lighting and shadows which is then applied to the blue image in the following CorrectIlluminationApply Module. The Median filter then attempts to smooth out the image to account for incorrect assumptions from within the illumination application.

The next step is to identify cells from within the bluestack through the IdentifyPrimaryObjects module which contains the main replacement function calculated from within the config pipeline. The IdentifyPrimaryObjects module first places intensity levels in seperate bins in order to identify through global thresholding utilizing the Otsu Method. This allows the program to recognize what is background and foreground. From here a watershed algorithm is used to to identify individual structures from within the photo; in this case the algorithm roughly detects cell structures.

Once the initial cell structures are outlined, various measurements are taken of them, including the size and shape of the objects as well as their intensities. With these measurements the Cellprofiler Wrapper attempts to filter out objects that are not real cells but speckling in the frame and objects that are too bright, which are glial cells that are not factored into the final counts.

The next stage of the program is arguably the most important. The TrackObjects module identifies objects through multiple frames within the stack, attaches a unique identifier to each cell and tracks the lifetime, movement, and changes of that cell as it exists within the stack. Because the program processes Z-Stacks, this module can be used to find the overall volume of cells and the number of distinct unique cells that exist throughout the provided Z-Stack.

The final set of commands relates to serializing the information, which is exported in the form of excel spreadsheets and images. The excel sheets include all the unique identifiers and any relevant meta information relating to the cells. The images show the original cell image layered with the unique identifier and an outline of what is considered part of the cell. This information is then sent to the report log so be easily consumed by the user.

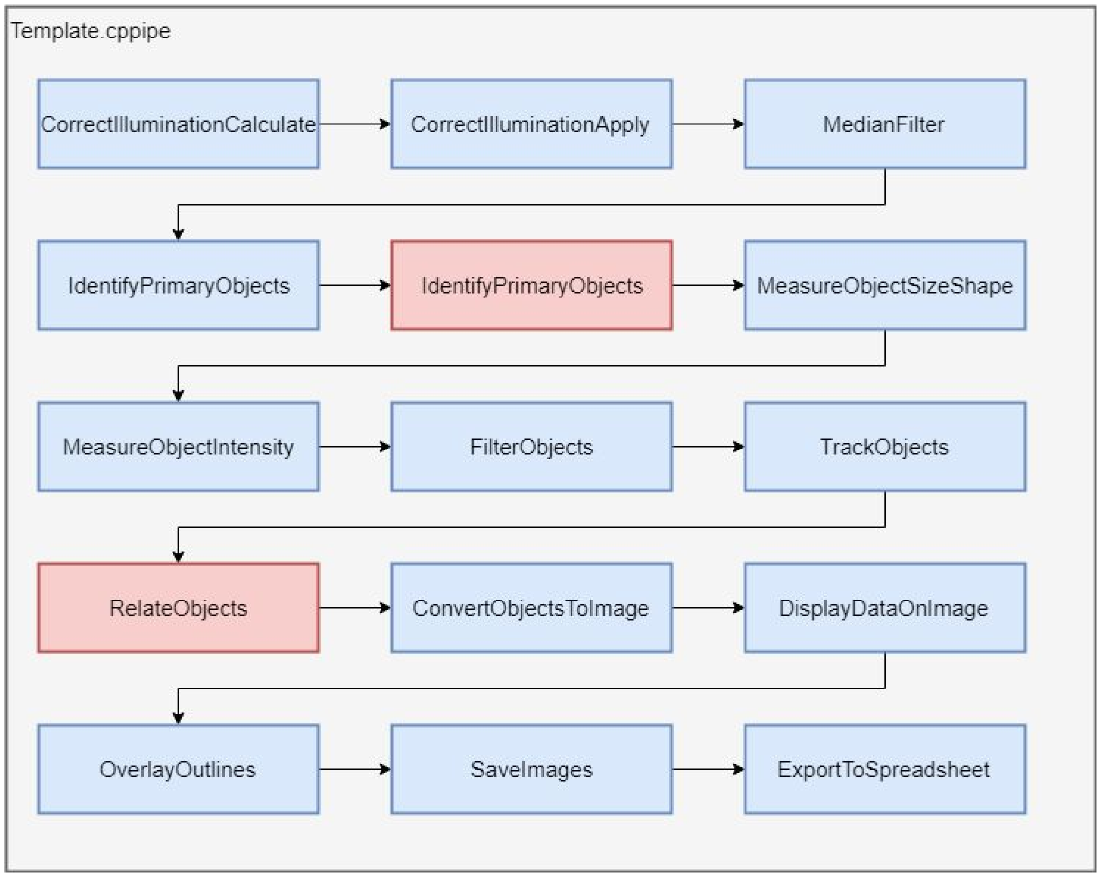
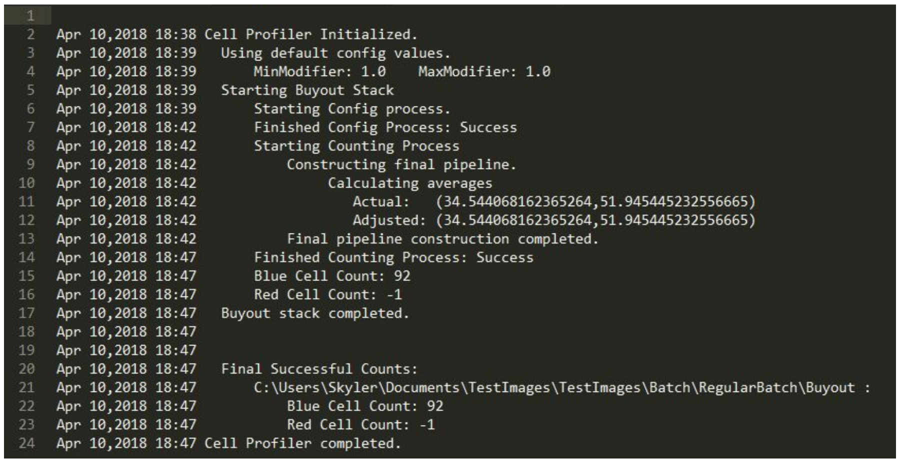


Figure 3.4: The pipeline that results in the final counts for cells. Note that this includes the inactive red cell modules marked in red.

3.1.3 Report Generation and Logging

CellProfiler outputs several Excel sheets with information about the counts. Also outputted is a logging file, shown in Figure which provides a record of how the program ran. It provides timestamps for each step of the process, along with information like the Min-modifier used and cell counts. It first gives the Min-modifier values, then marks the start and end of configuration process, where CellProfiler is sun with the Config pipeline. Next, the file marks the start of the counting process, gives the averages obtained from running the config process, then the final counts for red and blue cells.



```

Apr 10,2018 18:38 Cell Profiler Initialized.
Apr 10,2018 18:39  Using default config values.
Apr 10,2018 18:39      MinModifier: 1.0    MaxModifier: 1.0
Apr 10,2018 18:39  Starting Buyout Stack
Apr 10,2018 18:39      Starting Config process.
Apr 10,2018 18:42      Finished Config Process: Success
Apr 10,2018 18:42      Starting Counting Process
Apr 10,2018 18:42          Constructing final pipeline.
Apr 10,2018 18:42               Calculating averages
Apr 10,2018 18:42                  Actual:   (34.544068162365264,51.945445232556665)
Apr 10,2018 18:42                  Adjusted: (34.544068162365264,51.945445232556665)
Apr 10,2018 18:42          Final pipeline construction completed.
Apr 10,2018 18:47      Finished Counting Process: Success
Apr 10,2018 18:47      Blue Cell Count: 92
Apr 10,2018 18:47      Red Cell Count: -1
Apr 10,2018 18:47  Buyout stack completed.
Apr 10,2018 18:47
Apr 10,2018 18:47
Apr 10,2018 18:47  Final Successful Counts:
Apr 10,2018 18:47      C:\Users\Skyler\Documents\TestImages\TestImages\Batch\RegularBatch\Buyout :
Apr 10,2018 18:47          Blue Cell Count: 92
Apr 10,2018 18:47          Red Cell Count: -1
Apr 10,2018 18:47 Cell Profiler completed.
```

Figure 3.5: Logging file

3.3 Cell Profiler GUI

The GUI is a series of dialog boxes created using the Java Swing library. The first to appear is a JOptionPane asking if the job is a batch job, and it is shown in Figure 3.6

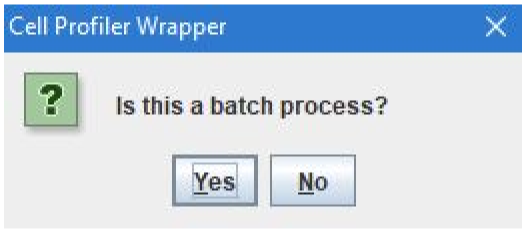
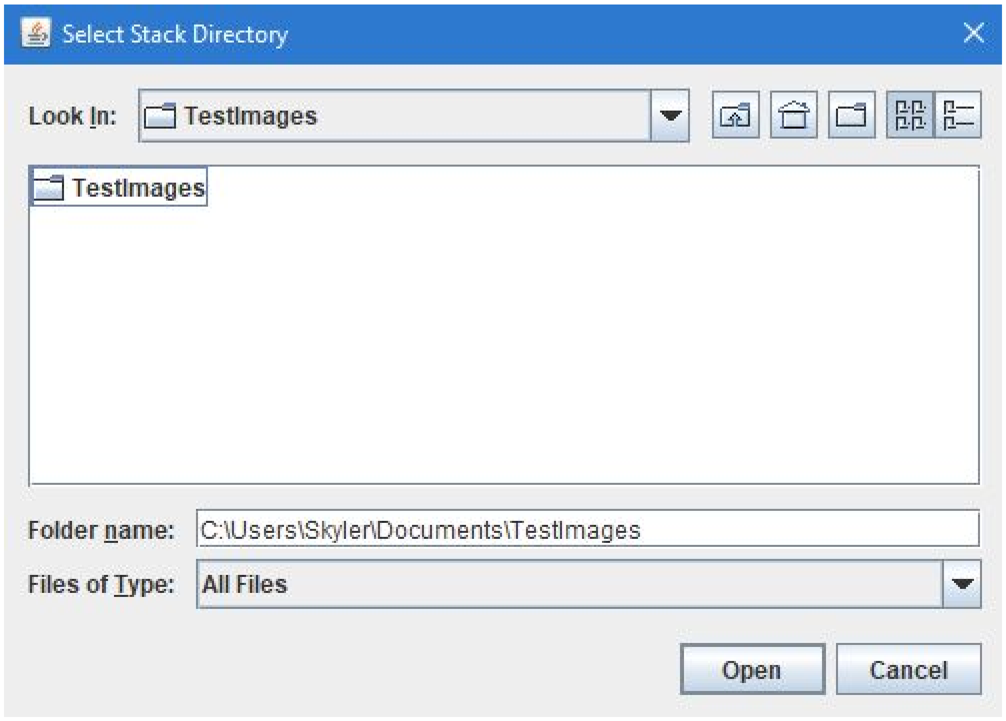


Figure 3.6: First dialog box to appear. Determines whether we are running multiple Z-stacks (yes) or just one (no).

The next dialog box is a JFileChooser, and it allows users to navigate to the folder containing the Z-stack(s) they wish to run CellProfiler on, shown in Figure 3.7. Users simply click on folders and their subfolders until the desired path is selected.

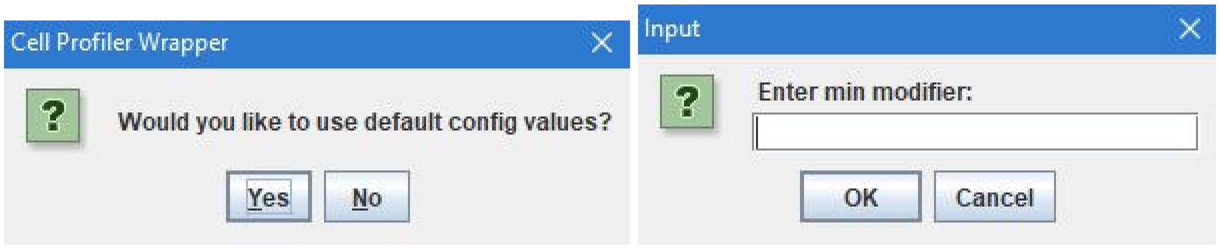


Folder name: C:\Users\Skyler\Documents\TestImages

Files of Type: All Files

Figure 3.7: File selection tool used to find path to Z-stack

Next, a dialog box appears which asks if the user would like to use the default config values (Figure 3.8a). If the user selects "no", they can then input a min-modifier in a new dialog box which appears (Figure 3.8b).



Would you like to use default config values?

Enter min modifier:

(a) (b)

Figure 3.8: (a) Option to use default config values. (b) Dialog box for user input of a min modifier value.

4.0 Cell Profiler Testing and Results

Testing from within the cell profiler is in two separate stages. The first stage of the testing is through unit testing, whereas the second is more holistic in order to validate the cell counts. Unit testing is the testing of individual features with the goal of ensuring that any future changes will not change the actions of other features. Within the wrapper there are three main categories of testing: CSV utilities, Pipeline Authoring, and Pipeline Factories.

The comma separated value (CSV) utilities contain testing functions relating to the retrieval of data and performing calculations on this data. The Pipeline author tests functionality relating to generating a templated pipeline through replacement of variables. Pipeline author ONLY builds the pipeline with pre given values and does not do any calculations. The pipeline factory, on the other hand, is given the config file output as a seed for the pipeline factory and calculating everything that needs to be replaced. These tests are described in the below table. These tests all currently pass.

#	Test Group	Test Name	Test Description	Pass/ Fail
1	CSVUtil	testLoadCSV	Tests to ensure that the CSV util can correctly load a file that exists.	pass
2	CSVUtil	testGetSingleValue WithColIndex	Tests the ability to get a specific cell within a preloaded CSV file based off an Index	pass
3	CSVUtil	testGetSingleValue WithColName	Tests the ability to get a specific cell within a preloaded CSV file based off a column name and row value	pass
4	CSVUtil	testColumnAverag e	Tests the calculation of the average of values within a column given a column name. Note that this method may be off slightly due to Floating Point calculation	pass

5	CSVUtil	testNthPercentile	Given a column name and a percentile, calculates the percentile of that column. This is normally used to evaluate the typical cell size modification.	pass
6	CSVUtil	testUniqueValues	Tests to ensure that we can evaluate the number of unique values in a column. This is used for finding the number of unique identifiers for cells within a column.	pass
7	PipelineAuthor	testTemplateReplacement	Creates a replacement rule and attempts to populate a templated pipeline with the replacement values.	pass
8	PipelineFactory	testFactory	This test provides a data file and attempts to construct a pipeline based off that information.	pass

The second set of testing is far more holistic and is done through manually running the program and varying the modification values until the output resembles the desired system. These values can be increased if the cells are larger than average or decreased if they are smaller. We have identified three different sets within the test values in order to reach the desired error rates and outputs. Table XX contains these three sets categorized by (minModifier, maxModifier) sets.

	(1, 1.25)		(0.5, 1.5)		(1.25, 1.75)	
Image Set	Count	Error	Count	Error	Count	Error
Christian	204	-5.39%				
Buyout			132	5.3%		
Combo					88	12.5%
Phillip			132	2.27%		
Problem Dog	100	11%			89	0%
Pseudophedrine	148	9.46%			133	-0.75%
Rhee						

Ricen	157	1.27%				
Rick			250	-8%		

Note that Rhee is the only that is currently not within the error bounds. After heavily reviewing the images, I believe that this is due to the large variety of light intensity in the stain. This causes difficulty in the binning with image and possible miscalculation of what is “foreground”. The below figures show the final test image and the large scale of intensities. On the converse side, a data set like Christian is optimal for a program like this because the cells are distinct and separate.