**Project 2 – Final Program Proposal**

**1) Synopsis**

This document serves as an initial proposal for organizing all the operations our team performed to predict cilia regions based on frames given. Being a proposal, this document only serves as an initial idea for final organization, which should address the intrinsic goals for software: clear usage, reasonable options, examples of use, and the ability to generate the predictions we made. By formalizing what such goals entail in this circumstance should lead us toward clear, concise code for sharing.

**2) Outline**

From reading through the GitHub history of our team, I see many different files and their history. Together, there were two main branches for processing Optical Flow and Tiramisu. Both of these approaches underwent variations in their use: ranging from fluctuation in parameters used to different input/output format (from blurring results to changing CNN input size). In addition, there were a range of image processing options tested. The two sets together (image processing and cilia predations) could serve as the basis for what we provide users:

Results

* Optical Flow
* Tiramisu
  + Bilateral Filter
  + Dice Loss
  + Focal Loss

…

Predict Cilia with Process Options

* Download Examples
* CLAHE
* Bilateral Filter
* Sharpen image

…

Image Download/Format Options

The image above is a very brute illustration of a user’s flow through our software. My idea is to have a range of image processing options packaged together, everything from just downloading the images supplied by Quinn to apply any (to all) image format techniques we have tested with. In the general readme file, we will give examples for what parameters we used with these techniques along with clear command line executions that would generate the same results. These image processing options appear to be sequential in nature, meaning we would need to formalize the order by which we have applied them so users won’t be surprised (thereby allowing them to consider reordering along with further options before moving to the prediction process.

Once the images have gone through the download and/or formatting operations and users have directories containing frames and masks for the training and testing sets, we provide users with options for executing the different prediction operations we used. As opposed to the image processing, selection between Optical Flow and Tiramisu are disjoint, but a number of the different Tiramisu options tested should be given to the user as an option.

I (Jonathan Myers) believe these are reasonable goals that will make software that looks simple and is simple to use, but that is one perspective. Please take a moment before going into the following planning sections to determine if this feel right for you or if you have another organization idea that would be more appropriate.

**3) Python Scripts**

As discussed in the outline section, our project will supply three main programs: one for downloading and formatting files, one for Optical Flow, and one for Tiramisu. Hopefully, by providing examples of how to use these programs, users won’t have difficulty using them.

\*\*\* ONE SIDE NOTE: The plan for this project is to build software that will run on OS X or Linux. Not going to worry about Windows \*\*\*

**3.1) Download/Format Images**

The main function for this python script (with name say DownloadFormatImageData.py, feel free to throw out another name) provides the following options (NOTE: The image changes are applied in the order listed below, we could discuss allowing for order change later…):

1. –o <output directory>: This is a required field used to specify the directory that will store all processed files. This process won’t execute if there are any files already there; if the user select “-over <output directory>” then the script will delete everything already there before placing new files there.
2. –s <source directory> **OR** –g <git source address>: The user must select either of these options. If they already have frame and mask data in subdirectories of the source directory they define with –s, the script will check to make sure files agree with expectations and then copy them to the output directory for later processing. If the –g option was selected, the program will try to download all files (following the preconceived file organization expectations) and place them into the output directory. In general, the format expects the following (and we will show this to users):

<root dir> + - test.txt

+ - train.txt

+ - masks + - <name>.png

| ...

+ - data + - <name> + - frame0000.png

| + - frame0001.png

| ...

+ - <name> + - frame0001.png

...

1. -clahe (opt. –clipLimit <clip limit> -tileGridSize <n> <m>): Just selecting -clahe will aplly the CLAHE (Contrast Limited Adaptive Histogram Equalization) to the image with default settings 2 and (8,8). The user may override these default settings by using the optional parameters, but if they use those parameters without the option -clahe, the program will give them a warning. (<https://docs.opencv.org/3.1.0/d5/daf/tutorial_py_histogram_equalization.html>)
2. –bf (opt. –bf\_d <diameter of pixel neighborhood> -bf\_sc <width of color neighborhood> -bf\_ss <factor by which distance influences>): Selection of –bf alone comes with default settings 7, 30, and 30 to the optional parameters listed. User may specify these values, and the code will check for integer and double values.
3. –gb (opt. –gb\_x <sigma x> -gb\_y <sigma y> -gb\_border\_type <border type>): Selection of –gb alone comes with default settings 5, 5, and 0. User may specify these values and the code will check for two doubles and an integer.

Once the program checks all of the command line variables, checks to make sure all expected files are accounted for, and places the files in a target directory, it will process the files using any methods and parameters defined by the user. Once this process completes, without any errors, it will notify the user everything is ready for the next process (and it may show them a reminder of how to execute the next process.)

**3.2) Predict Cilia with Process Options**

Now we move into the main processing area of our software. Here the user specifies which of the two main branches they would like to use: Optical Flow or Tiramisu. For simplicity, I believe that having two programs (say one called RunOpticalFlow.py and one called RunTiramisu.py) would make things easy for the user to know what they chose and makes it easier for us to parse their parameters. Following this idea (and let me know if that feels wrong), we would have two main Python scripts here that parse different parameters.

**3.2.1) RunOpticalFlow.py**

I see now that Jiahao Xu genereated \_\_main\_\_.py script for both OpticalFlow and PixelVariance directories. These scripts are virtually identical, Optical Flow performs the same operations and Pixel Variance, but it performs one additional action calcOpticalFlowFarneback. I intend to merge both of these programs together into the RunOpticalFlow.py program and allow users to specify if and how they would like to use calcOpticalFlowFrarneback.

The RunOpticalFlow.py script will allow users to simply specify a root directory, that contains files and subdirectories that fill our expectations (i.e. they match the data format provided by Quinn), or the script will allow users to specify any of the individual items. The following list goes through some of the command line argument details.

1. –r <root data directory>: A users who supplies this value asserts all the files we need (train, test, masks, …) are contained in this directory following Quinn format. The script will check to make sure everything is there and then define the parameters before calling the OpticalFlow script.
2. –tr OR –th <threshold number>: A user may allow the script to derive a threshold from the train data or specify a threshold double (exclusive options)
3. –d <path>, -m <path>, -trf <train file>, -tsf <test file>: Allows the user to specify the directories and files. This will override the values derived from a root value if defined.
4. –OpFlow (opt. –pyr\_scale <pyr\_scale value> -levels <levels> -winsize <averaging window size>

-iter <num of iterations> -poly\_n <size of pixel neighborhood>): Selection of –OpFlow uses default values 0.5, 3, 15, 3, 5, and 1.2. If any other values are provided, the script will check for double, int, int, int, int, and double values respectively. If this option isn’t set, the process won’t use the OpFlow function when processing.

1. –o <output directory>: Basically the output directory, the program will try to generate the directory if it does not already exist.

The required fields include:

* -r OR ALL -d, -m, -trf, -tsf
* -tr OR -th
* -o

The script ends with the generation of prediction png files in the output directory.

**3.2.1) RunTiramisu.py**

This program carries three different steps that are necessary:

1. Organization of data into three sets: Training, Validate, and Testing
2. Build a Tiramisu CNNs with the Training and Validate sets
3. Apply on Tiramisu CNN to the Testing set

We could place all of these functions into a single script file, but the way python argument package handles command line arguments would demand the generation of redundant lines of code (which received criticism in a previous project). To avoid that potential problem, we will break this process into three different scripts: BuildTiramisuData.py, TrainTiramisu.py, and TestTiramisu.py. The three scripts will take responsibility for the three phases listed above, as their names suggest. To get around some intrinsic confusion these scripts will likely generate, we will certainly show examples on the team website illustrating how to use these scripts.

**3.2.1) BuildTiramisuData.py**

As opposed to the operations performed in the RunOpticalFlow.py process, the Tiramisu process requires dividing the training set into training and validate subsets. The program looks for the following in the target root directory:

<root dir> + train + masks + <name>.png

| | + ...

| + data + <name> + frame0000.png

| | + frame0001.png

| | ...

| + ...

+ validate + masks + <name>.png

| | + ...

| + data + <name> + frame0000.png

| | + frame0001.png

| | ...

| + ...

+ test + data + <name> + frame0000.png

| + frame0001.png

| ...

+ ...

We give users two options to work with these expectations:

1. Build a set of directories with this file structure
2. Allow use to build such a file structure for them

If the user’s data already conforms with these expectations, there is no need to generate these sets; they could simply move to the training stage. If they don’t have this setup correctly, they could execute this program using a target directory that contains data conforming with Quinn organization (as illustrated in section (3.1) earlier. Here are the arguments that BuildTiramisuData.py takes:

1. -r <root directory>: A path to the root directory which contains all the file shows in sections (3.1) earlier
2. -v <validate set size>: An integer signifying the size of the validate set this process will create
3. -s <random seed>: An integer to serve as the random seed for creating the validate set
4. -o <output root directory>: The root directory for building all the new directories and folders

The BuildTiramisuData.py script requires all four parameters. Once it has them, it checks to make sure everything exists in the source, generates the new folders in the output, and copies appropriate files over to the new directories. Once this is done, users may use the new location as a source for the training and testing steps, discussed in the following sections.