CMPT 365 Project – Written Report

Group: Camille Janicki

1. For the CMPT 365 project, I implemented two different ways to create STIs from an mp4 with a video transition, STI by Copying Pixels and STI by Histogram Differences. The project was written in python. All the libraries needed can be downloaded by running requirements.txt. The *open-cv* library was the most useful library for this project and used the most throughout the project. I was able to solve the problems of the project with the *open-cv* library in conjunction with the *numpy* library.

*Open-cv i*s an open-source computer vision library. I used this library to parse through the video and manipulate each individual frame.

Numpy is the fundamental package for scientific computing with Python. I used this library to convert regular arrays into ndarrays that were necessary for many open-cv functions.

GUI

The *Tkinter* package was used to create the GUI for this project. The main GUI window (root) was initialized as a global variable so that the different parts of the code could append to it directly (either appending text directly onto the root GUI or appending a pop-up GUI). The confGUI function sets the general configurations for the main GUI window. The "Choose Video" button goes to chooseVideo which uses the *askopenfilename* function from the *tkFileDialogue* package which allows the user to choose a file in their filesystem and returns the path as a string so the program can use the requested file. All four buttons for the STI choices lead to a function called parseVideo that catches if no file has been chosen. Finally the quit button closes the instance of the main GUI.

Part 1

Part 1 was implemented by parsing through each frame of the video chosen by the user and concatenating the middle column (or row) of each frame to an image to create an STI.

Part 1 starts by ensuring that a proper mp4 file was selected. With the *open-cv* library, I found the height and width of each frame and the number of frames in the video. I then used the *open-cv* library to capture the video frame by frame. While I was at a certain frame, I collected the middle column (or row) and appended it to a blank image which was actually a numpy ndarray. Then using the *Pillow* library, I converted the numpy ndarray to an image and saved it as a png as well as displayed in a new window. I had trouble with the *open-cv* library to keep the pop-up GUI open before calling to close a window. I could only find a way to wait on the user for a key signal (any key pressed) to signal closing the window. Therefore, the user must press any key to close the pop-up window.

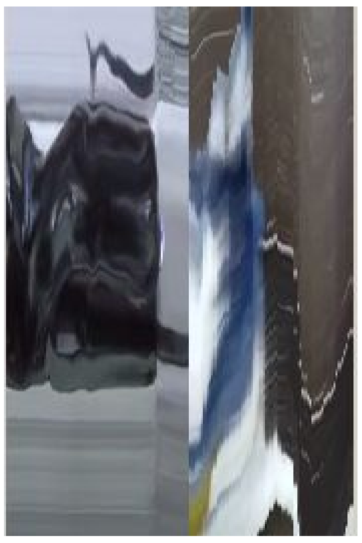
Figure 1. Result of Copying Columns

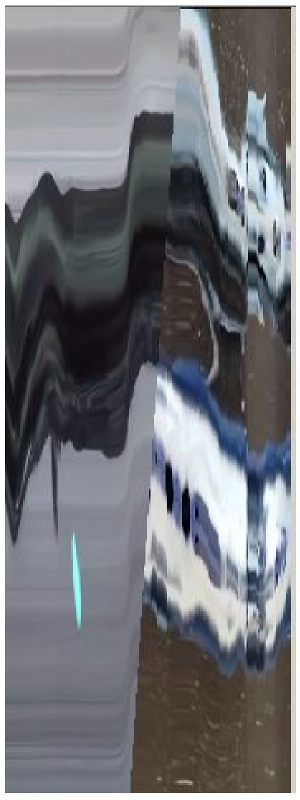
Figure 2. Result of Copying Rows

Part 2

Part 2 was implemented by parsing through each frame of the video chosen by the user, converting the pixel values of each frame from RGB to rg values, creating a histogram with the rg pixel values and comparing histograms to create the STI.

Part 2 starts as well by ensuring that a proper mp4 file was selected. With the *open-cv* library, the width and height of the frame were resized for quicker computation. Resizing the frames was essential because without resizing, the computation time was far too large, especially with uncompressed videos. I then created an array to store all the frames. Once I was finished collecting the frames, I looped through the array of frames and converted each frame from RGB values to rg values.

> chromaticity
>
> This was done by grabbing the RGB values of each pixel of each frame which was implemented by getting the pixel values in a nested loop, for ex. R = frame[j][i][0]. Each of the pixel values were divided by R+G+B. I had trouble with this section because python automatically stores everything as floats. Therefore, I was getting decimal numbers as pixel values which doesn't make sense. I used the round() function to remove the decimal values and then turned the float into an integer. Please note that I did this for all three pixel values, creating rgb instead of just rg, however, when creating my histogram later, I only used the rg values.

Once the chromaticity computation was complete, I looped through the array of frames (which was stored as a regular array) and converted each frame into an numpy ndarray which was necessary to make a histogram, then made a histogram per frame and finally appended each frame to a histogram list.

histList

This function takes in a list of frames and returns a list of histograms. This function loops through the list of frames and creates a histogram per frame (with makeHist) and returns the list of histograms.

makeHist

This function takes in a frame and converts it into a histogram. This is done by looping through the frame and finding the r and g values (not using the b value) and appending them to a new array. The array in converted into a numpy ndarray to be compatible with the *open-cv* library. Bins for the histogram were created as suggested in the project outline: N = 1 + log2(n), where n=size of data which was the length of the frame. I once again had to convert the bins into integers as python stores everything as floats. I was then able to use the *open-cv* function called *calcHist*. *calcHist* calculates a histogram from a set of arrays. I wanted to quickly display each histogram during computation, however, I had difficulties doing this with *open-cv*. I tried to use the *matplotlib* library to plot each histogram. I was successful in plotting the first one but it took quite a few seconds to render each histogram. If I took a couple seconds to demonstrate each histogram, the computation time would be huge (number of frames *x* couple seconds). Therefore, I decided to simply return the histogram.

Once I returned with the list of histograms, I created a new list called histogram_diff that stores the differences of the histograms. I used the *compareHist* function from the *open-cv* library and appended the result to histogram_diff. *compareHist* simply compares two histograms and returns a float. The list of histogram differences is then converted to a numpy ndarray to be able to be converted to an image using the *Pillow* library. I saved the image as a png and displayed it in a new gui window. I once again had the same issue with waiting to close the GUI as the first part, therefore, the code once again just waits for a key signal (any key pressed) to close the pop-up window.

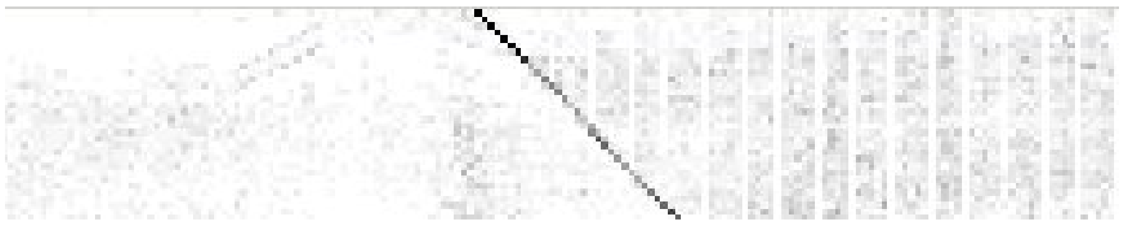

Figure 3. Result of using Columns

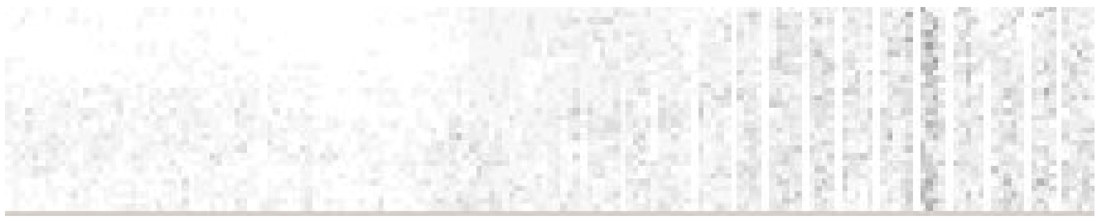

Figure 4. Result of using Rows

2. The project was quite challenging, especially as a group of 1. The project could have been improved by covering more of the technical details for the second part a little bit more in class, for example histogram differences. A different project could be something to do with JPEG compression as the second half of the course focuses on JPEG or to implement a different kind of compression. I think the project and assignments should all be done individually because there are many students in this program that take advantage of the work of their peers. Therefore, the project should be less difficult as it should equate to the amount of work that would be appropriate for one student only.