Image and film recolorization

Talha LAIQUE & Cédric JAVAULT – Master AI Year 1

INF573 – January 2020

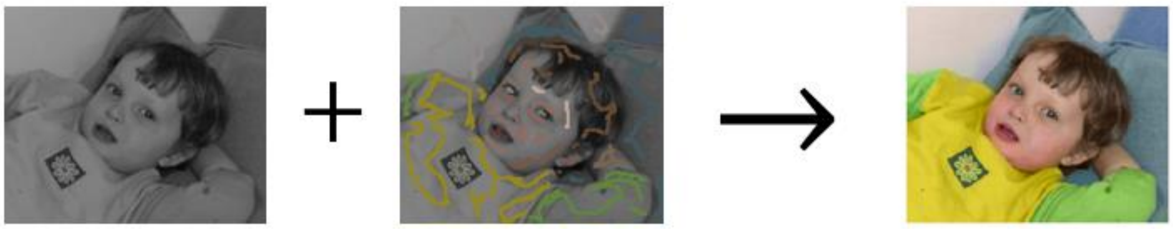
Introduction

For this project, we wanted to work on **recolorization algorithms**, which take a black and white image (more precisely an image with its grey levels) as an input and give a colored image as the output. We also wanted to expand it from images to films.

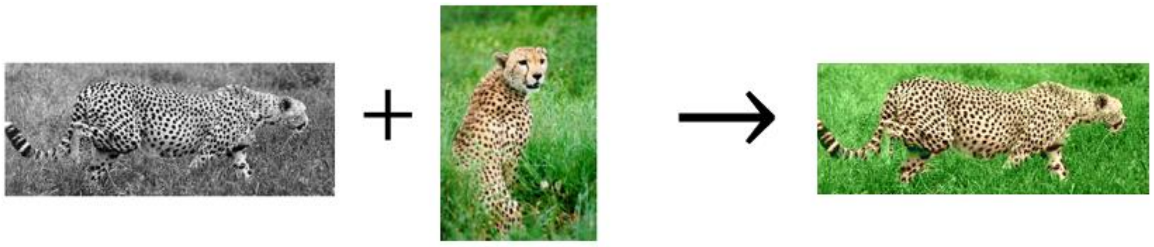
This is obviously a little bit tricky as the computer has somehow to start from a 1-byte information (the grey level) to give a 3-bytes information (the RGB colors). This tricky thing is more obvious if you see it that way: there is a bijective transformation between the RGB color space and the YUV color space. In the later, Y alone stands for the grey level; in other words, the computer has all to "find" U and V.

Actually, we found three completely different methods to address our problem:

- The first one was developed by Anat Levin, Dani Lischinski and Yair Weiss in 2004 in a paper called "**Colorization using optimization**". The idea is to help the computer giving it the colors (ie U and V) for only some points and to assume that "*Nearby Pixels in space-time that have similar gray levels should also have similar colors*". We made different versions of this algorithm and had pretty good result as shown in part I.



- The second one was developed by R. Irony, D. Cohen-Or and D. Lischinski the year after in a paper called "**Colorization by Example".** The idea is to help the computer by giving a similar image with witch it can learn the textures and the colors of the image. We worked quite hard (and used many tools seen in the course or found in openCv) but due to lack of details in the paper, we could not get a final result.



- The third one is based on deep learning, the computer being trained with millions of colored pictures. This gives astonishing results (although something not really accurate) but had nothing really to do with the content of INF573 so we did into look into it.

I. Colorization using Optimization

a) The mathematical part

Let us summarize the main ideas of the article:

- We work in the YUV color space, Y having all the grey level information (it is also possible to work in YIQ color space; the important thing is to isolate the Y).

- We assume that nearby Pixels that have similar gray levels should also have similar color. This is done by **minimizing** J(U) (and J(V) the same way):

  $$J(U) = \sum_{\mathbf{r}} \left(U(\mathbf{r}) - \sum_{\mathbf{s} \in N(\mathbf{r})} w_{\mathbf{rs}} U(\mathbf{s}) \right)^2$$

  N(r) are the neighbors of r
  w_{rs} should be large iif Y(s) is close to Y(r) with $\sum_{\mathbf{s} \in N(\mathbf{r})} w_{\mathbf{rs}} = 1$

- There are different possibilities to choose w_{rs}. For example: $w_{\mathbf{rs}} \propto e^{-(Y(\mathbf{r})-Y(\mathbf{s}))^2/2\sigma_{\mathbf{r}}^2}$ where σ_r is the variance of Y around r.

- The solution is the U and the V minimizing J(U) and J(V) under the constrains that U and V are know for some points (the "labelled" points).

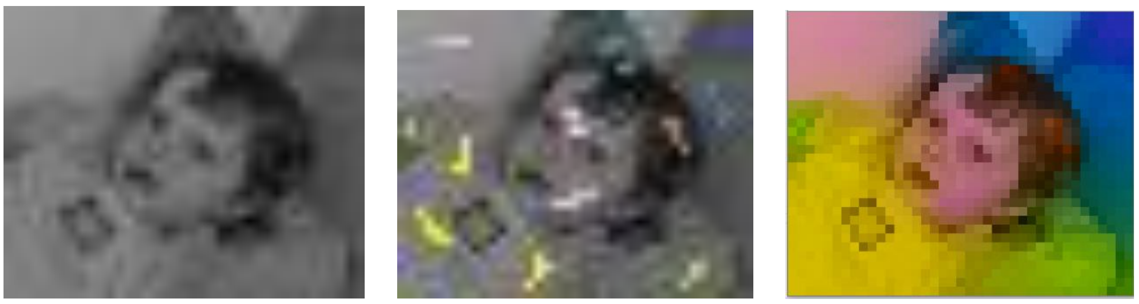
- The method to finally solve the problem is not fully explained in the paper. We build a sparse square matrix A whose size is the number of pixels (if n=200 and m=300, the matrix is a 60 000 x 60 000 sparse matrix) and solved Ax = u where u=U for the labelled points and 0 if else (and then Ax=V with v=V for the labelled points and 0 if else). The lines of A are built that way:
  - If the point r is labelled, only keep a 1 on the diagonal so that $x_r=U_r$ for a labelled point
  - If the point r is not labelled, keep the same one on the diagonal and put - w_{rs} for all the neighbors of the pixel r so that x_r = Sum (w_{rs} x_s) for a non labelled point.
  - (Do the same for V)

- Writing A = I – W (I identity matrix and W the w_{rs} matrix), Ax=b becomes x = b + Wx. It is therefore possible to iterate computing $x^{n+1} = b + Wx^n$ until convergence.

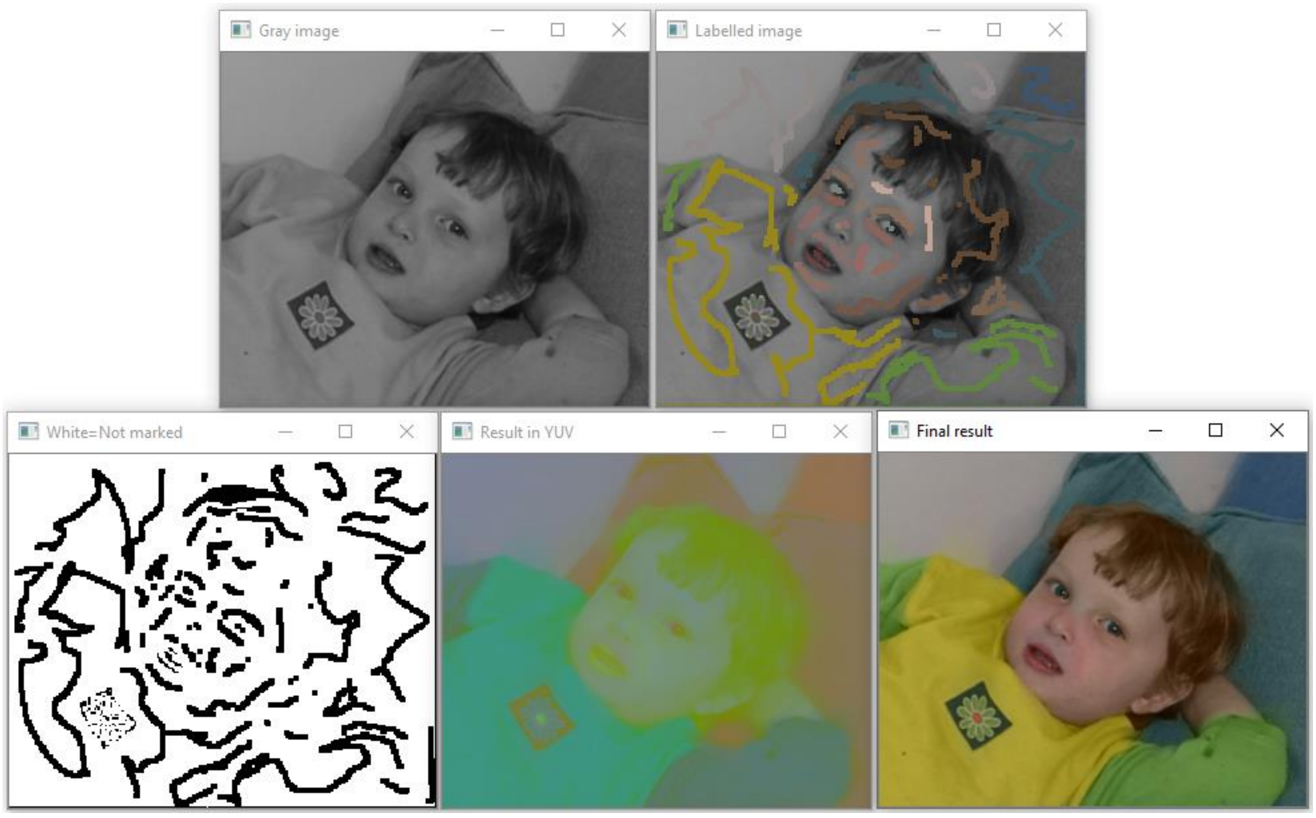
b) Implementation details and results for an image

We first saw that the authors of the paper published a C++ code, together with their article. But it's an "old" (2004) code, they are over 1200 lines that are not very well commented, and it is supposed to be run from Mathlab, witch we did not know.

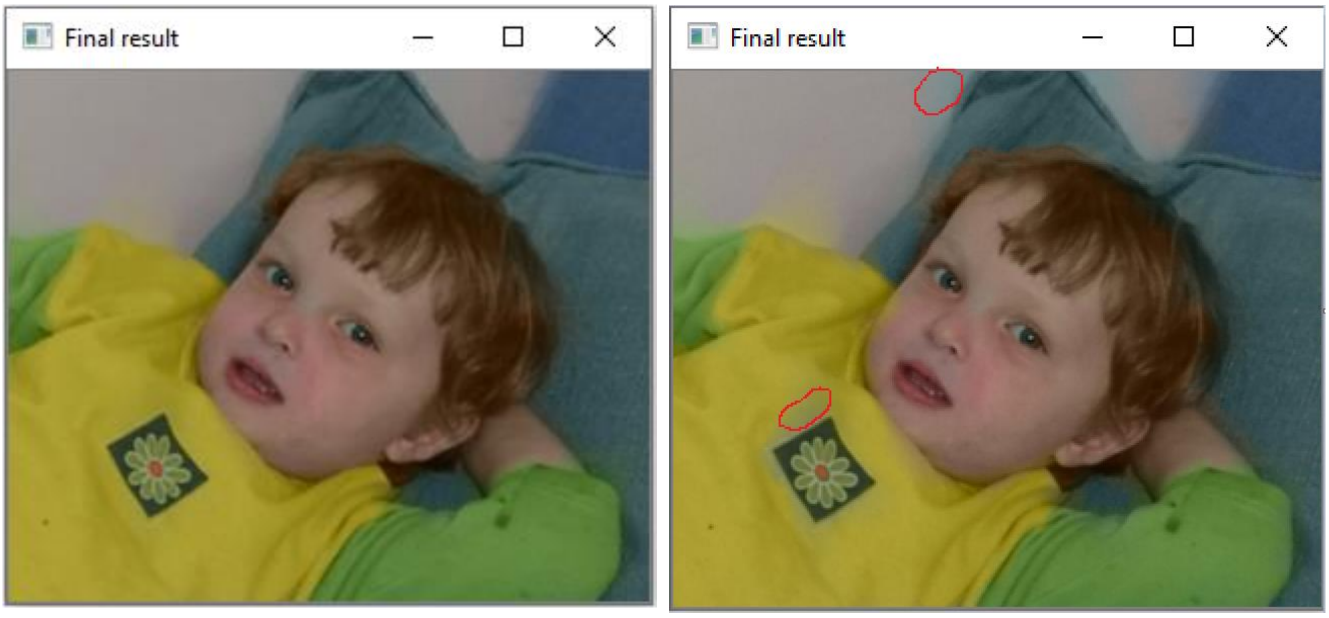
We decided to write our own algorithm and ended up with 180 lines (colorization_images_only.ccp) in C++ using opencv and Eigen/Sparse. To debug the code efficiently, we started with a very small image (39 * 32 pixels) which was a reduced version of the article's picture; we labeled it ourselves. We first used the exponential weights ($w_{\mathbf{rs}} \propto e^{-(Y(\mathbf{r})-Y(\mathbf{s}))^2/2\sigma_{\mathbf{r}}^2}$), struggled a little bit with Eigen because the SparseCholesky solver didn't work properly and finally got what we wanted with the SparseLU solver:



Then, we worked with bigger images and confirmed our results. This was achieved in less than 2 seconds (it is the main example from the article):

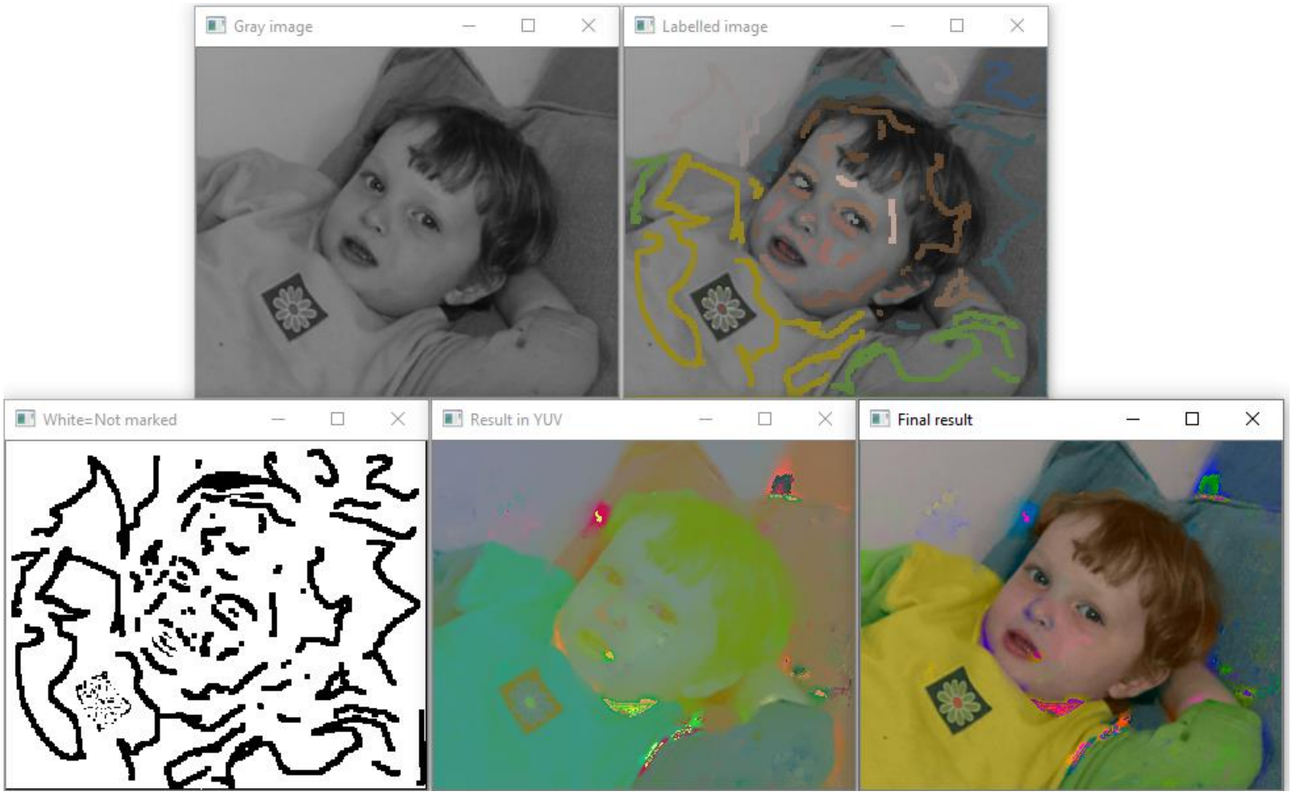


We tested another weigh function $w_{rs} = 1 + (\sigma_r)^2$ / abs((Y(r)-mean) * Y(s) – mean) +1e^{-6}) with good results (left : first weight function – right: second weight function). In both cases, the neighbors of a pixel are the 8 pixels surrounding it.

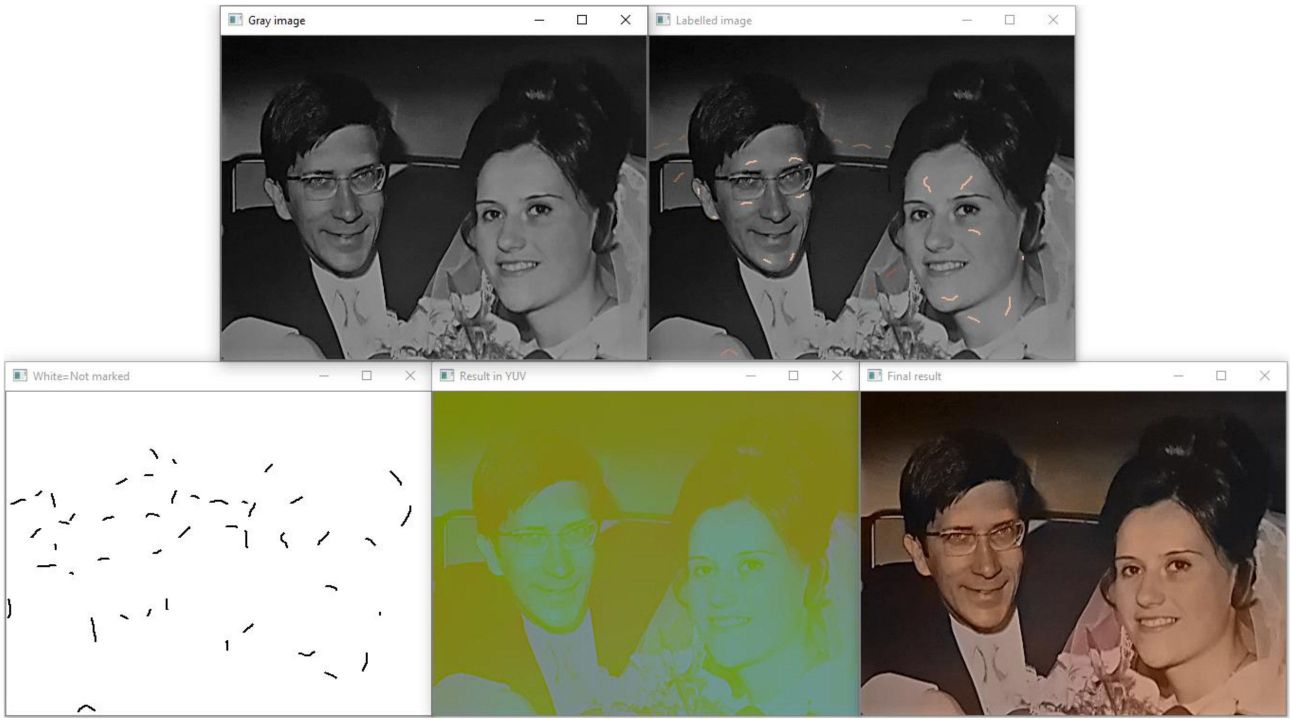


(Nevertheless, first weigh function tends to better segment the colors ; see for example the yellow of the tee shirt in the middle).

We were surprised to discover what seems to be a mistake in the article as the second weight function they suggested ($w_{\mathbf{rs}} \propto 1+\frac{1}{\sigma_{\mathbf{r}}^2}(Y(\mathbf{r})-\mu_{\mathbf{r}})(Y(\mathbf{s})-\mu_{\mathbf{r}})$) is obviously wrong because w_{rs} is not always big when Y(r) and Y(s). Anyway, the results are poor:



At last, we tested our own images and got for example the following (with the first weight function):



c) From images to films

One could think that expanding the image recolorization algorithm to a film recolorization algorithm is quite simple: you just move from 8 (3*3-1) neighbors to 26 (3*3*3-1) considering time as a third dimension. But in fact, there are different problems that have to be taken into account.

The first one is very simple but had still to be addressed: contrary to the image examples, the authors provided their film example in a compressed format. Due to the loss of information during compression, none of the 7 labelled images corresponded to any of the 84 images of the film. We had to compute a "distance" to re-assign the labelled images to the closest image from the film.

Once that was done, we had to deal with the real issues: the optical flow and the algorithmic complexity.

1. The optical flow

In a film, the camera moves, and the people and the object move with respect to each other. For example, those are 3 images from the example film we worked on (it is the example given and labelled by the authors of the article):

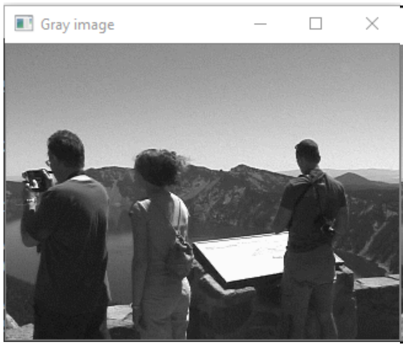


Image #2

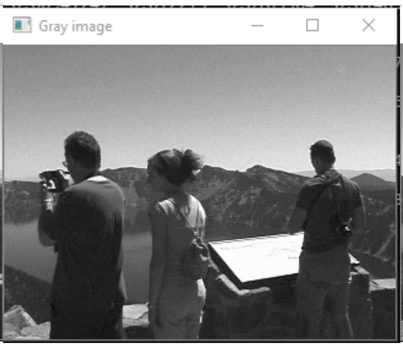


Image #6

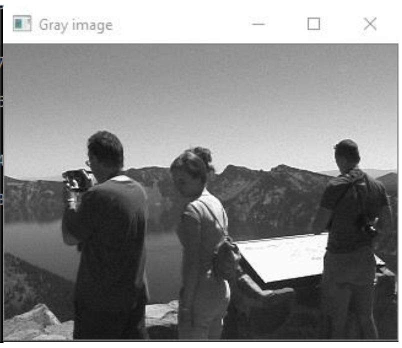
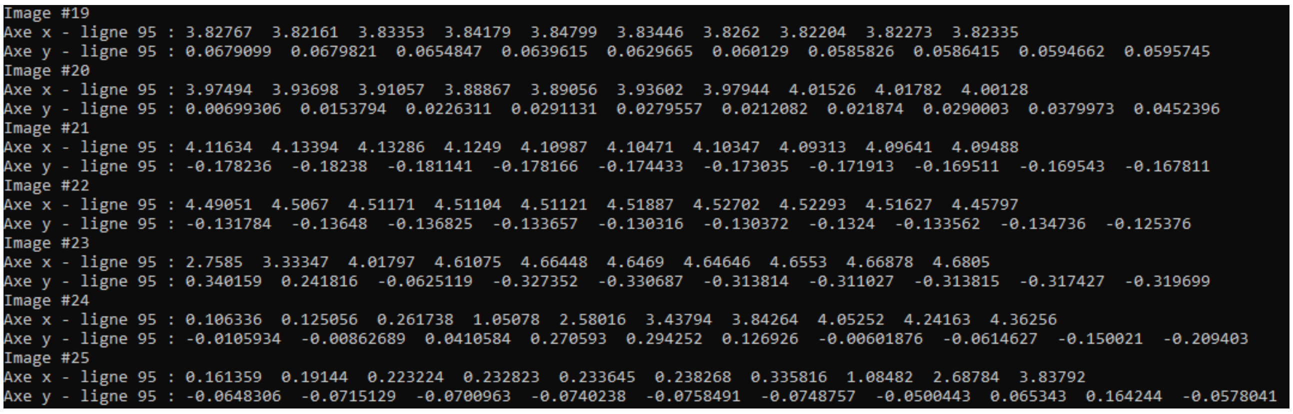


Image #10

The camera is obviously moving on the x axis (and the lady is also about to move toward x too). Therefore, you cannot just compare pixel (x, y, t) to pixel (x, y, t+1). You have to compute that motion and take into account. We used calcOpticalFlowFarneback from opencv to compute the flow (the flow is a couple of floats, one along x and the other along y, that is computed for every pixel of every image). Just to show the importance of the phenomenon, this is a screenshot that we took while debugging. It shows, image par image, the optical flow for the pixels (95,90) to (95,100). We can see that the movement (which was more or less +4 on x and nothing on y) for image #19 to #23 suddenly stops at image #25…



```
Image #19
Axe x - ligne 95 : 3.82767  3.82161  3.83353  3.84179  3.84799  3.83446  3.8262  3.82204  3.82273  3.82335
Axe y - ligne 95 : 0.0679099  0.0679821  0.0654847  0.0639615  0.0629665  0.060129  0.0585826  0.0586415  0.0594662  0.0595745
Image #20
Axe x - ligne 95 : 3.97494  3.93698  3.91057  3.88867  3.89056  3.93602  3.97944  4.01526  4.01782  4.00128
Axe y - ligne 95 : 0.00699306  0.0153794  0.0226311  0.0291131  0.0279557  0.0212082  0.021874  0.0290003  0.0379973  0.0452396
Image #21
Axe x - ligne 95 : 4.11634  4.13394  4.13286  4.1249  4.10987  4.10471  4.10347  4.09313  4.09641  4.09488
Axe y - ligne 95 : -0.178236  -0.18238  -0.181141  -0.178166  -0.174433  -0.173035  -0.171913  -0.169511  -0.169543  -0.167811
Image #22
Axe x - ligne 95 : 4.49051  4.5067  4.51171  4.51104  4.51121  4.51887  4.52702  4.52293  4.51627  4.45797
Axe y - ligne 95 : -0.131784  -0.13648  -0.136825  -0.133657  -0.130316  -0.130372  -0.1324  -0.133562  -0.134736  -0.125376
Image #23
Axe x - ligne 95 : 2.7585  3.33347  4.01797  4.61075  4.66448  4.6469  4.64646  4.6553  4.66878  4.6805
Axe y - ligne 95 : 0.340159  0.241816  -0.0625119  -0.327352  -0.330687  -0.313814  -0.311027  -0.313815  -0.317427  -0.319699
Image #24
Axe x - ligne 95 : 0.106336  0.125056  0.261738  1.05078  2.58016  3.43794  3.84264  4.05252  4.24163  4.36256
Axe y - ligne 95 : -0.0105934  -0.00862689  0.0410584  0.270593  0.294252  0.126926  -0.00601876  -0.0614627  -0.150021  -0.209403
Image #25
Axe x - ligne 95 : 0.161359  0.19144  0.223224  0.232823  0.233645  0.238268  0.335816  1.08482  2.68784  3.83792
Axe y - ligne 95 : -0.0648306  -0.0715129  -0.0700963  -0.0740238  -0.0758491  -0.0748757  -0.0500443  0.065343  0.164244  -0.0578041
```

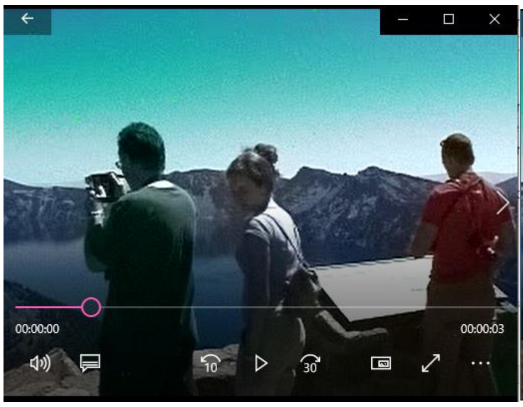
2. <u>Algorithm complexity</u>

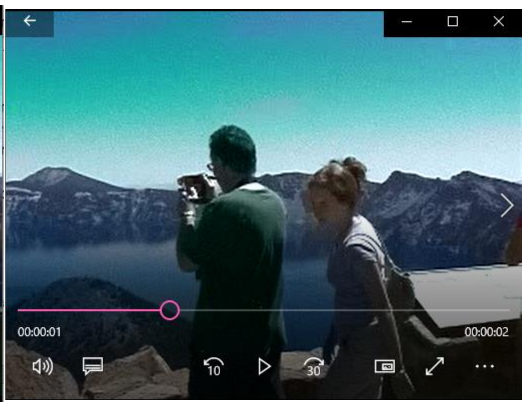
The size of our matrix is multiplied by the number of images. In our example, we had 84 images in the film. This means that the matrix had a size multiplied by 84^2… and solving Ax=b has probably as complexity between n^2 and n^3 (84^3 = 592 704… we're in trouble!).

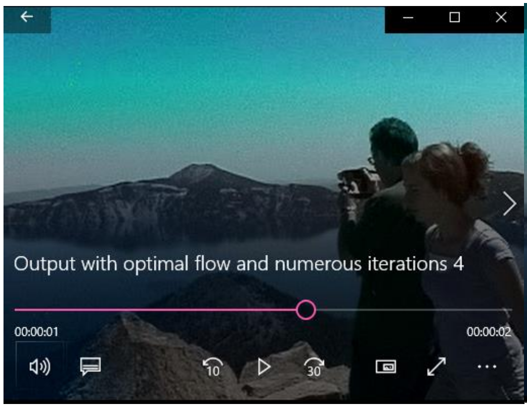
But the good news what that Levin and all were able to do it with the computation power they had in 2004... We decided to go back to their code, to understand those 1200 lines, and to plug in with the data structures supported by opencv. If fact, they are two ideas in the code that are not in the article:

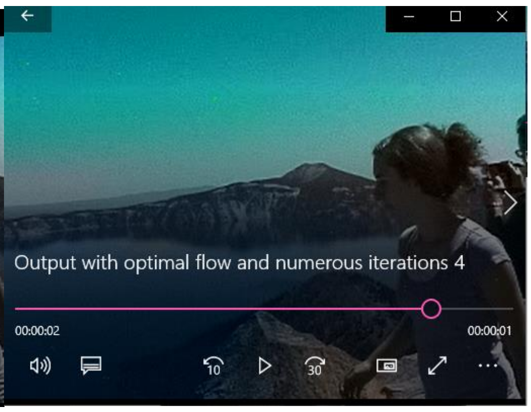
- The algorithm doesn't solve the problem the way we did it (Ax=b) but by iterating $x^{n+1} = b + Wx^n$ (see below).
- The algorithm works and optimizes first very small images (up to 32x32 times smaller in our case) and then grows step by step, optimizing the same image in a 2x2 bigger format starting with the result from the previous phase.

After several hours of computation, we finally got a very good result for our film (especially having just 7 images which were really very partially labelled). Here are a few screenshots (the complete film is with our data and code)









II. Colorization by example

Colorization by example involves coloring a grayscale input image with the help of a reference image of same context. As per the research paper, there are several steps to follow in order to complete the aforementioned task but due to lack of certain set of technical and implementation related details, we had to modify certain steps and attempted to implement it. The steps of our algo are as below:

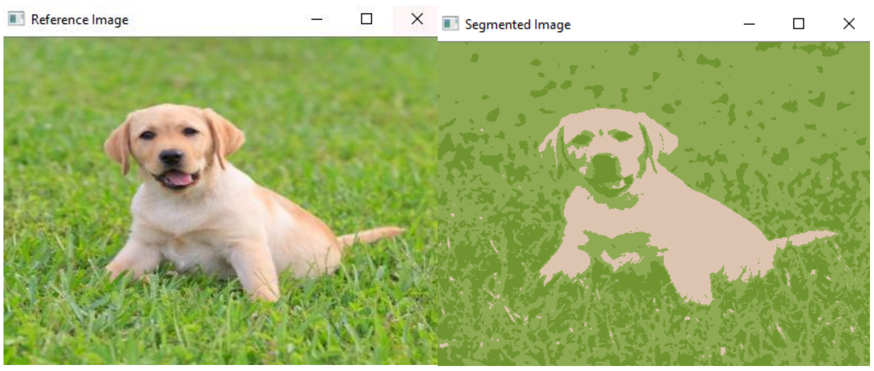
- Generating colored segmentation of the reference image
- Computing the features set of reference image luminance channel and the grayscale input image by applying discrete cosine transform
- Extracting nxn neighborhood of features from the transformed reference image and associating them with labels
- Applying PCA and LDA to the extracted nxn neighborhood of features and forming a new subspace (Features + Labels).
- Classification of input image features labels by training Knn-classifier with previously created features subspace (We divided labels into three main categories as we used k=3 where k is the no. of segmentation color for segmentation of the reference image and made the following assumptions,
  2: Label in Majority belongs to the background
  1: Label in 2nd Majority belongs to the foreground
  0. label in Minority belongs to the foreground certain regions i.e eyes, nose, mouth etc).
- Extracting foreground from the input gray scale image with GrabCut and detecting edges with opencv's canny algo, finding and drawing contours and bounding them in rectangles.
- Finding pixels inside the bounding rectangles which strictly belong to the foreground and coloring them as per their label category

a) Generating colored segmentation of the reference image:

We used K-means clustering for generating the colored segmentation of the reference image for three regions; background, foreground and sub-foreground features. We projected the reference image onto matrix(rows*cols,3) where each row represents the pixel of the reference image and rows=300, columns=400. We computed the labels for the projected reference image and listed three distinct labels counts;

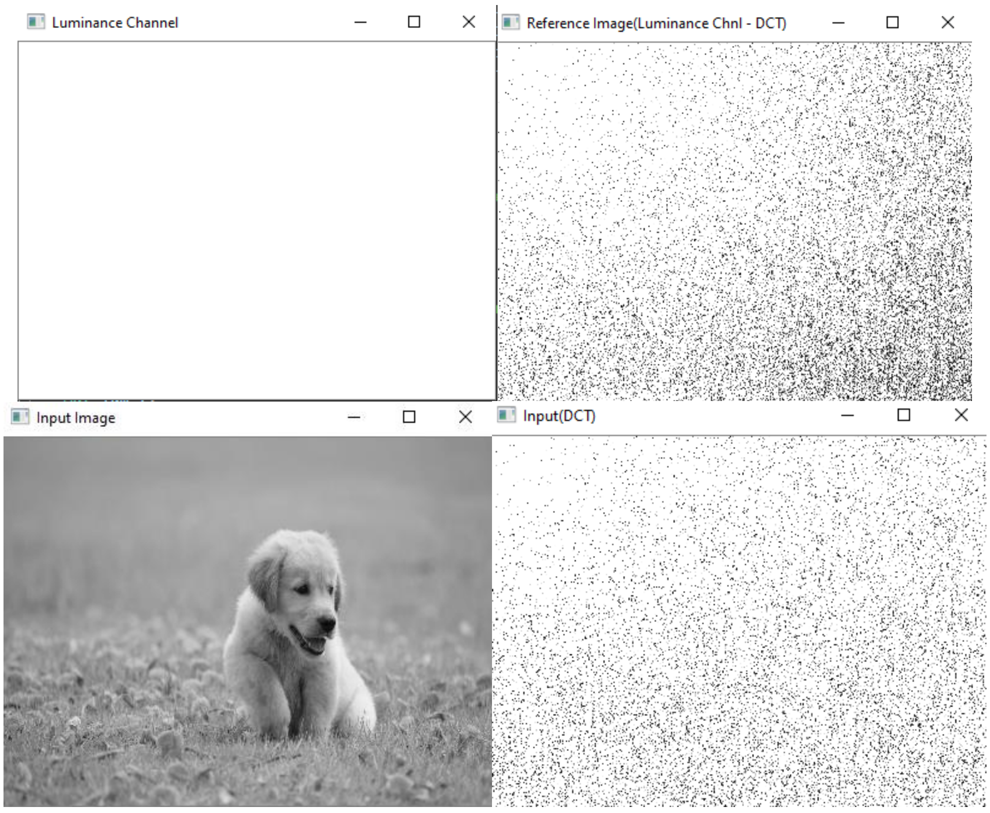
- Label 0: 31 203 (Sub-foreground features such as mouth, nose etc. and some misclassifications)
- Label 1: 17 206 (Foreground i.e. puppy's body)
- Label 2: 71 771 (Background i.e. grass)

Please note that each label corresponds to a unique color returned by the K-means clustering.



b) Computing the features set of reference image and the grayscale input image:

We only took the luminance channel of the reference image and applied discrete cosine transform (DCT) to it to get texture-based features. The reason being that we will be classifying the features of the grayscale input image so it is suggested to use the discrete cosine transform of the reference image's luminance channel so we get classification based on texture. Please note that the current DCT implementation only supports even number of rows and columns so we had to keep the rows and columns even. We computed the features set of grayscale input image (300x400) the same way and later own, we projected both of the transforms onto (rows*cols,1) where each row represents a feature for each pixel corresponding to the reference and grayscale input images. These projections are our training sample and testing sample data.



c) Extracting nxn neighborhood of features:

We created a (600x100) neighborhood of features from the previously projected sample training data and associated a label with each row so in total of 600 labels. We computed the mean value of each column of the sampled training data as for projection of these features to a new subspace mean value of each column is required by the LDA.

d) Applying PCA and LDA:

We applied principle component analysis to our previously sampled features set in order to compute the eigen values (100x1) and vectors(100x100) and features vectors which in our case is (600x100) vectors, we then applied the Linear Discriminant analysis to those vectors and their respective labels with *number_of_components=1* as our testing sample for later on classification has only one column, and computed the LDA eigen values(1x1) and vectors(100x1). Now due to the misclassifications associated with our features set labels, we decided to use the product of PCA and LDA eigen vectors for generating a new subspace for our features where misclassifications would be low. We used *gemm()* function for computing the product of both of the techniques eigen vectors and then used the LDA *subspaceproject()* function for creating a new features subspace as per the product eigen vectors. This resulted in the dimensionality reduction of previously computed features (600x100) to (600x1), in other words we have 600 rows that we assumed are correctly associated with their labels. Please note that such huge dimensionality reduction can still have misclassifications but it wouldn't be significant in number as compared to (600x100) features.

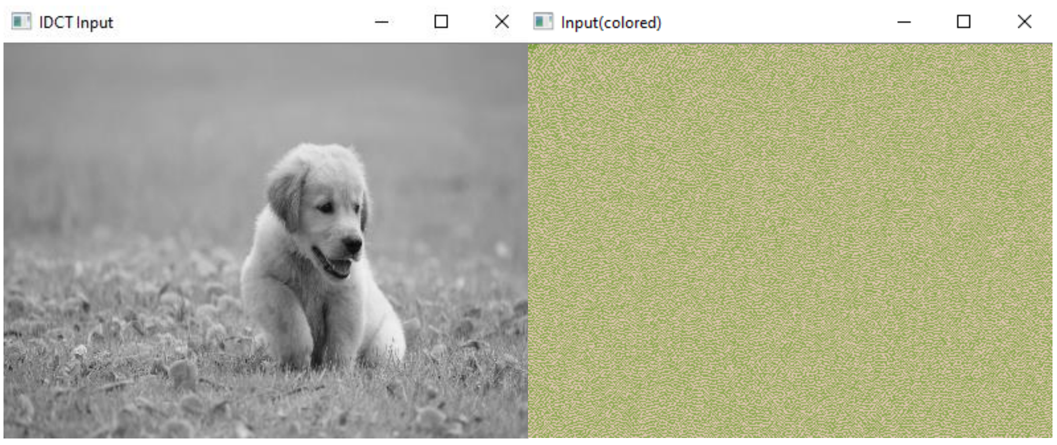
e) **Classification of sampled testing features:**

We used Knn-classification with brute force and 5 neighbors' configurations. First, we trained the classifier with previously computed features subspace (600x100) and their respective labels (600x1) then we tested it for our previously sampled testing features(120000x1) and generated 120000 labels. The counts of labels are:

- Label 0: 18
- Label 1: 58 663
- Label 2: 61 319

There is a huge difference between Label 0 and 1 if we compared them with labels returned by the kmean clustering during reference image segmentation and we think the reason might be the difference between the contents of our grayscale input image and colored reference image. As label zero corresponds to the sub-foreground features and it is evident from the input image that they are not as prominent as in the reference image. Please note that we have used different (nxn) neighborhood of feature pixels but the above labels were the best of all. We noticed that for every other neighborhood Label 0 was either not classified or was very less so we have to choose (600x100) as the dimensions of our features set.

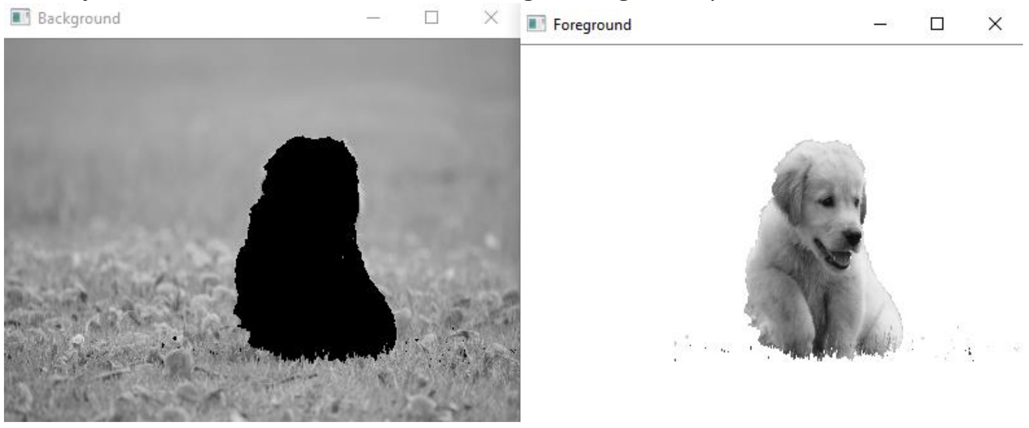
We then projected the (120000x1) testing sample features to (300x400) and computed the inverse discrete cosine transform and filled with the color values as per the predicted labels to check the color distribution of our input image.



As we can see that, there is no layout of the foreground in the colored version of our input image, we decided to use the opencv's grabCut algorithm and to color the foreground and background correctly.

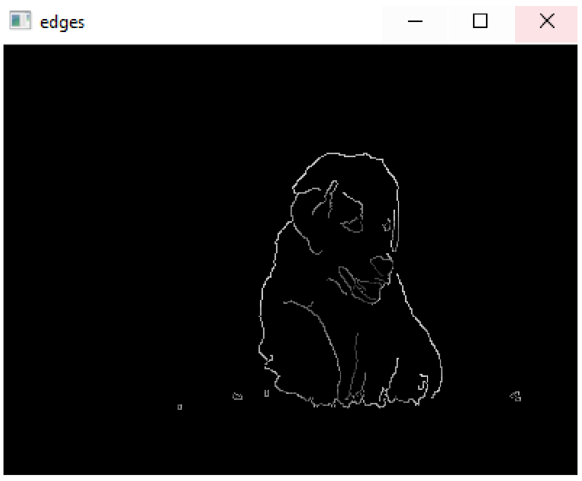
f) Extracting foreground with grabcut, edges detection with canny, contour drawing and bounding rectangles:

We used a rectangle of size(120x60) for capturing pixels (coloredInput.columns – 20, coloredInput.columns-110) with GC_INIT_WITH_RECT so we have the foreground pixels inside a rectangle and further used the compare method just to make sure that we have the right foreground pixels.

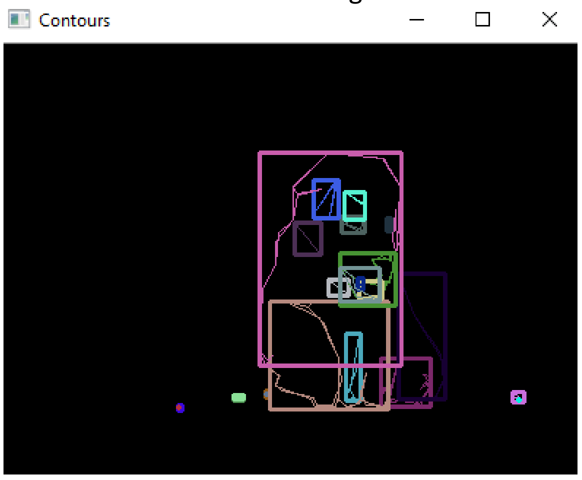


Then, we used the opencv's canny implementation for detecting edges with the following parameters which resulted in correct detection of foreground edges;

- lowThreshold=75
- ratio = 3
- kernel_size = 3



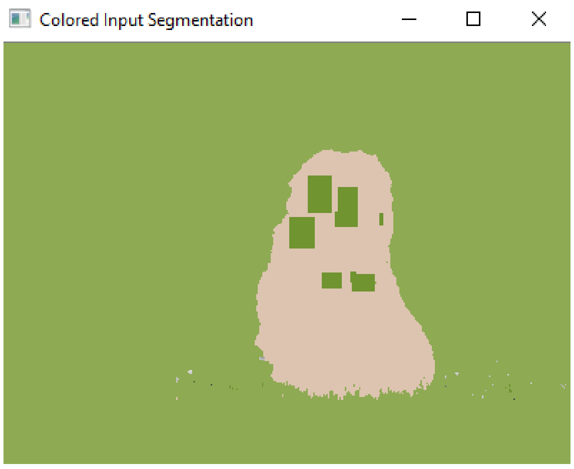
We then found and drew the contours corresponding to each edge and bounded them in their respective rectangles. The reason for bounding the contours with rectangles was to make sure that we have correctly identified the regions which we will use later on for coloring.



For extracting the pixels from previously identified regions, we used *multimap* datastructure, and listed a pair of (Rectangle area, Points), the main idea was to distinctively specify all of the pixels in their correct regions for later on coloring.

g) Coloring foreground, sub-foreground and Background:

Our aim is to color only the foreground pixels strictly belonging to the puppy. We computed the pixel values of the foreground image for each point which we listed previously with *multimap* datastructure from our bounding rectangles. If the value is not equal to 255, it belongs to foreground and hence we colored it with 2nd label in majority (Foreground color) else we colored it with 1st label in majority (Background color), furthermore, we specified the areas of the rectangles which belonged to the sub-foreground regions and colored them with label in minority. The final input image colored segmentation;



Of course, we are far from the complete expected result, but this still is a (very rough) estimation of the recolored image.

Conclusion

We tried two completely different ways to recolor images and films.

We were quite successful with the "*Colorization by optimization*" approach and implemented a totally homemade algorithm for the images and another one, adapted from the old mathlab-compatible code provided by the authors, for the films. In both cases, the results are comparable to those achieved by the authors.

We Also attempted to implement the algorithm for *"colorization by example"* as described in the research paper but due to the lack of clarity and implementation related details, we had to modify certain steps and made certain assumptions… got some results but did not really succeed.