Karl Erlandsen c1198379 04/03/06

Innovations Project Report

Introduction

For my innovations project, I decided to look at the issues surrounding image recolouring, including current techniques, their drawbacks, and possible solutions. I then wanted to try to implement a new image recolouring algorithm to attempt to address the limitations inherent in these current methods. This report documents my efforts in this area; my successes, my failures, and what I learnt along the way.

The problem

Colorization is defined as the process of adding colour to black and white images, or changing the colours already present in colour images. The term was originally coined in the 1970s to refer to the process of recolouring black and white film footage, although it has since also expanded to include techniques related to recolouring colour images for stylistic effect (such as colour grading, a technique famously used to give images a mood in films such as the Lord of the Rings trilogy, Amélie, and Sin City).

Although the idea of using colorization to affect images has been around for over 30 years now (longer, if you include similar efforts in photography), there are still significant difficulties associated with the technique. It's an extremely artist-intensive process that requires a long time and significant experience on the part of the artist to be able to achieve reasonable results. Currently, the artist must use a tool such as Adobe Photoshop or Gimp to select areas of the image and fill them with the same colour. Even with advanced selection tools like the “magic wand” this is still a time consuming process, especially if the the borders between coloured areas aren't clear. This is often the case if the image is blurred, for example, or contains items with complicated edges such as hair or fur. The process is often complicated further in areas which feature large numbers of colours or subtle changes of colour (such as skin or clouds).

Whilst more advanced selection tools and techniques have become available in recent years (such as *normalised cuts*, or the *SIOX Simple Interactive Object Extraction* algorithm), they offer only a partial solution to the problem. They will certainly increase the speed of segmenting an image up into areas of similar colours, but they still leave the significant task of actually recolouring the image to the artist.

A potential solution

In 2004 Anat Levin (a PhD student), Yair Weiss (Levin's PhD supervisor) and Dani Lischinski (Assistant Professor/Senior Lecturer) of the School of Computer Science and Engineering at the Hebrew University of Jerusalem released a SIGGRAPH paper entitled “Colorization Using Optimization”. In the paper, they described a new artist-friendly technique to recolourise images using just a few simple colour scribbles.

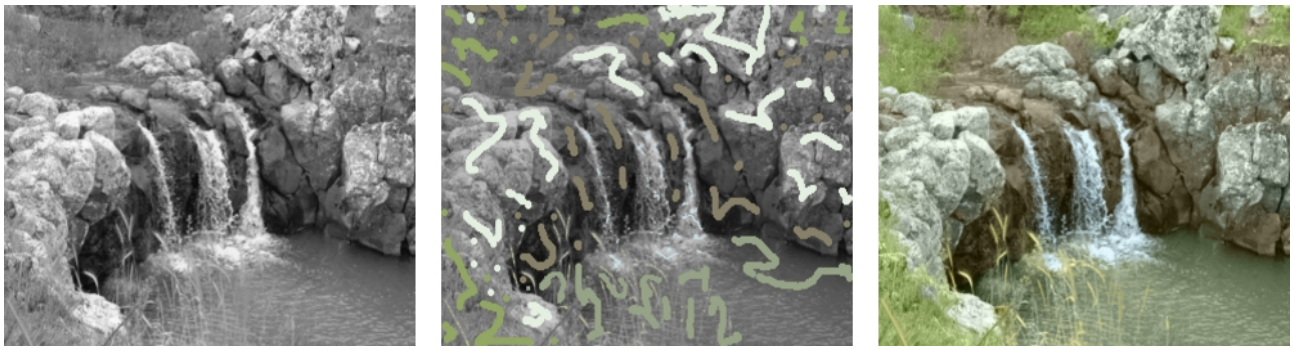

Figure 1.: Example images showing the capabilities of the algorithm described in the 2004 SIGGRAPH paper "Colorization Using Optimization". On the left, the original black and white image. In the centre, the original image with some quick colour scribbles added by the artist. On the right, the result of running the algorithm. More examples may be found at http://www.cs.huji.ac.il/~yweiss/Colorization/ and also in the SIGGRAPH paper itself.

The results from this algorithm are impressive, to say the least. The artist only needs to provide a few rough colour scribbles on the original black and white image, and the algorithm automatically calculates both image segmentation and colour spreading with visually convincing results. I was so impressed with this algorithm, that I decided to try to implement it.

The Algorithm

The algorithm described operates in YUV colour space. YUV colour space is conceptually similar to the YIQ colour space, and is mostly used for TV broadcast signals, since it allows backwards compatibility with black and white TVs. YUV encodes colour with 3 colour channels, where the first channel (Y) encodes the intensity of the given pixel, and the other 2 colour channels (U and V) encode the colour of the pixel.

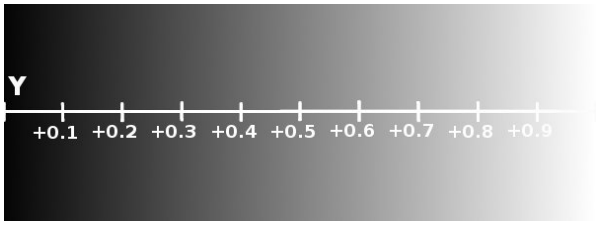

Figure 3.: The Y intensity scale

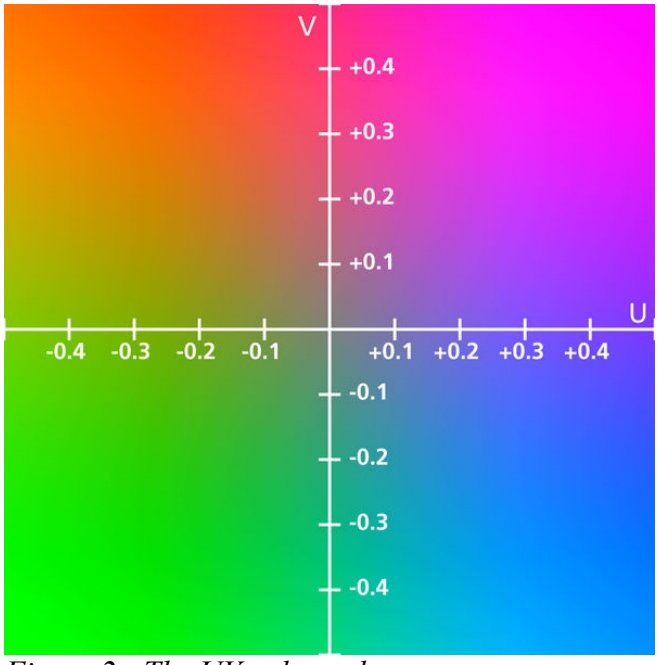

Figure 2.: The UV colour plane

Obviously, YUV colour space is ideal for an algorithm such as this one, since colorization seeks to keep the intensity of the image the same, but derive or change colour information. This means we can leave the Y channel alone, and simply focus on generating U and V channel values.

The algorithm itself works on the simple principle that if two given pixels are near each other and have a similar intensity, then they should also have a similar colour value. This is expressed in formulaic form in the following way:

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

Figure 4.: formula expressing the constraint that pixels with similar intensities should have similar colours

where:

- J(U) is our function, which takes a colour volume U as input
- **r** is a given pixel within the input colour volume
- U(**r**) is the value of the pixel **r** in the colour volume U
- **s** is also a given pixel within the input colour volume
- **s**∈ N(**r**) denotes the fact that pixel **s** is a neighbouring pixel of **r**
- U(**s**) is the value of the pixel **s** in the colour volume U
- w_{rs} is a weighting function, large when **r** and **s** have similar intensities, small when **r** and **s** have different intensities

The aim of this algorithm is to minimise J(U) for each pixel of the input colour volumes U and V – that is, the squared difference between a pixel's intensity and the weighted intensities of the neighbouring pixels should be as small as possible.

The obvious question from this is what the weighting function for this could be. The paper gives two examples that they've experimented with, which is says give roughly equal results:

$$w_{\mathbf{rs}} \propto e^{\frac{-(Y(\mathbf{r}) - Y(\mathbf{s}))^2}{2\sigma_{\mathbf{r}}^2}}$$

Figure 5.: first formula for the weighting function

$$w_{\mathbf{rs}} \propto 1 + \frac{1}{\sigma_{\mathbf{r}}^2} - (Y(\mathbf{r}) - \mu_{\mathbf{r}}) - (Y(\mathbf{s}) - \mu_{\mathbf{r}})$$

Figure 6.: second formula for the weighting function

where:

- σ_r^2 is the variance of the intensities of all pixels neighbouring **r**
- Y(**r**) is the intensity of pixel **r**

- μ_r is the mean of the intensities of all the pixels neighbouring **r**
- Y(**s**) is the intensity of pixel **s**

The paper states that given a series of pixels which have had their colours marked by an artist, and by assuming that each pixel's colour is a linear function of its intensity (where this function will be different for each "pixel window" that we sample in), we can derive a sparse linear system that can be solved using a number of standard methods.

Technical Research and Framework Implementation

My first step in attempting to implement this paper was to see if this algorithm had ever been implemented before, and if so to see what I could learn from them. My search came up nearly empty – whilst a search on Google will reveal many people expressing their admiration of the results this algorithm generates, there was almost nothing of any practical use. I found a closed-source time-limited demo of a Windows program called *Recolored* that appeared to use a similar algorithm, but this was obviously of very limited use to me. Whilst re-reading the paper's home page, I stumbled upon a link to some proof-of-concept code written by the authors of the SIGGRAPH paper. Unfortunately for me, it was written in Matlab. Even worse, it wasn't commented at all, and featured variable names with descriptive titles, such as "wd", "vals", "gvals".

Matlab is a commercial mathematical programming language and interpreter/runtime environment. It has found itself popular in mathematical, engineering and educational circles, particularly those dealing with numerical analysis, matrices, and linear algebra. Since the commercial program costs several thousand US dollars for a licence, I decided to use a free program named GNU Octave, which is an attempt by the free software community to produce an open source Matlab clone.

Whilst I knew OPL, Pascal, bash script, C and C++ to various degrees, I was unfamiliar with Matlab. Given a distinct lack of alternatives, I decided to teach myself Matlab using GNU Octave, with the aim of understanding enough to be able to interpret the proof-of-concept code. I downloaded, compiled and installed GNU Octave into my home area, and then I went through the proof-of-concept code one line at a time. Whenever I came across a command or syntax that wasn't immediately obvious, I went to GNU Octave and made simple tests with small matrices or integers to try to understand the command. I then went through the code and commented every line.

```
t_val=ntscIm(i,j,1);
gvals(tlen+1)=t_val;
c_var=mean((gvals(1:tlen+1)-mean(gvals(1:tlen+1))).^2);
csig=c_var*0.6;
mgv=min((gvals(1:tlen)-t_val).^2);
if (csig<(-mgv/log(0.01)))
  csig=-mgv/log(0.01);
end
```

Figure 7.: A snippet of the Matlab proof-of-concept, syntax coloured for legibility. Here the variance of the intensity of a window of pixels is being calculated, and clamped to stop it getting too small.

Whilst this was certainly useful, I think it's fair to say that at this point I still wasn't sure how to implement the algorithm myself. The reason for this was that whilst I understood what the Matlab code was doing, I didn't understand why. The code seemed to bear little relation to my understanding of the algorithm as described in the paper, so whilst I understood the computing theory of what the code was doing, the mathematical theory underpinning it was lost on me. Things were made more difficult by the fact that a key part of the code that appeared to deal with linear system solving was handled by the "sparse" function, a function that GNU Octave hadn't yet implemented. I'd hit my limit with the Matlab code, and decided to move on.

By this point I was conscious that before I could start coding the filter itself, I'd need a simple environment that I could test it on. I decided to start making a framework that would allow me to load in images, run a filter on them and display the result, so I left the complex Matlab code alone and instead focused on getting the framework this project needed up and running.

For my choice of programming language, I decided to use C++. I chose C++ because we'd only had a brief introduction to it in lectures, and whilst I did have some prior C++ experience from several years ago I was eager to brush my knowledge up and get some good, practical experience with the language. For my choice of IDE, I decided to go with KDevelop for similar reasons – I wanted to branch out and try an IDE I hadn't really used properly before.

I started by making a simple window with SDL that could respond to keyboard input. Most of the code to do with initialising SDL in the function *initSDL()* was copied from library documentation at the SDL website, but everything else was my own work. From here I made the window contents drawn by OpenGL, and created a the function *draw()* to draw a textured plane across the whole window, with the intention of having the resulting image from the filter operation displayed here. In retrospect, it would have perhaps been more sensible to have designed the program so that rather than displaying the resulting image in a window on screen, it instead wrote the resulting image to a new file on disk which could then have been displayed using any standard image viewer. However, at the time I didn't know what my choice of image library would be and how easy it would be to use. I also wanted to have the ability to quickly compare filtered images with the originals by flipping back and forth between them, and OpenGL seemed the quickest and easiest way of achieving this.

For my choice of imaging library, I was torn between SDL_Image, DevIL and ImageMagick. SDL_Image boasted excellent integration with SDL, DevIL boasted superb OpenGL integration, and I had had experience with both of them before. However, in the interests of learning and broadening my horizons, I chose to experiment with ImageMagick first. All doubts very quickly disappeared – ImageMagick was a genuine joy to use. Thanks to some excellent documentation on its website and a clean, simple C++ interface through libMagick++, I managed to compile, install, learn, and integrate ImageMagick into my code in under half an hour. The only (tiny) limitation that I ran into was that ImageMagick didn't support reading image data directly as YUV – I needed to read it in as RGB floating point data first, and then convert it to YUV myself.

I realised at this point that throughout the program, I'd be dealing with several independent images, all of which would need to be able to convert from RGB to YUV and back again. All of them would need a GLuint texture reference number to be able to have their contents drawn with OpenGL, and all of them would need to be able to provide me with per-pixel access to RGB/YUV channels at any given time. This seemed like the perfect sort of functionality to wrap up into a class, so I created a class called "*yuvimage*" to provide me with exactly that. Each image was given pointers to image data, a GLuint OpenGL texture reference number, constructors and destructors that handled memory allocation automatically, and the methods "*yuvimage::RGBtoYUV()"* and "*yuvimage::YUVtoRGB()"*. I also gave the class a *"yuvimage:: makeDrawData()"* method which would convert the image data into an OpenGL texture, storing the texture reference in the

appropriate member. To implement the YUV/RGB conversion routines, I looked up details of the YUV colour space on the Internet, and quickly found the following conversion matrices:

$$\begin{pmatrix} Y \\ U \\ V \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.147 & -0.289 & 0.436 \\ 0.615 & -0.515 & -0.100 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

Figure 8.: matrix to convert RGB values to their YUV equivalent

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1.140 \\ 1 & -0.395 & -0.581 \\ 1 & 2.032 & 0 \end{pmatrix} \begin{pmatrix} Y \\ U \\ V \end{pmatrix}$$

Figure 9.: matrix to convert YUV values to their RGB equivalent

Implementing these was very straightforward, and simply required me to loop through each pixel in the image and calculate the necessary coefficients. To test the algorithm, I made a test program to load a sample image and convert it from YUV to RGB and back again 100 times, and then numerically compared the results with the original. I was pleased to see that even after 100 conversions back and forth, the largest value deviation was 0.001. I considered this numerically acceptable for my purposes, and moved on. It may be worth noting that if this algorithm were to be used to generate HDR images, these coefficients would perhaps have to be updated with more accurate values.

I now had a simple C++ class that would allow me to load an arbitrary number of images, convert them almost perfectly between RGB and YUV, and access individual sample values directly as floating point values between 0 and 1. I set up a program that would use this class to load in 3 images and allow the user to choose which one to display (the intention being that these images would be the original black and white image, the user marked image, and a colour image for comparison).The next stage was to start coding the algorithm itself.

At this stage, I had a stroke of luck. Whilst researching the algorithm further, I stumbled upon an implementation of the algorithm that had been written as a Gimp plugin. Best of all, the source code was freely available for download. The quality of the code itself wasn't much better than the Matlab code – the only meaningful comment in the code ominously read '*Yes, it's ugly. I'm lazy*'. Still, I was optimistic that I now had a second reference implementation that I could look at if I got stuck.

```
#define M_ALLOC(type,i,j) malloc((i)*(j)*sizeof(type))
#define V_ALLOC(type,d) malloc((d)*sizeof(type))
#define M_IDX(i,j)        ((i)*w+(j))
#define M_V(m,i,j)        m[M_IDX((i),(j))]
#define V_V(v,d)          v[d]
h = src_rgn.h;
w = src_rgn.w;
A = M_ALLOC(*A, WINDOW_PIXELS, h*w);
AI = V_ALLOC(*AI, WINDOW_PIXELS*h*w);
AJ = V_ALLOC(*AJ, WINDOW_PIXELS*h*w);
```

Figure 10.: A snippet of the Gimp colorization code, syntax coloured for legibility. Here, memory is being allocated for a sparse matrix.

I noted straight away that the Gimp code used a library called *UMFPack* to handle its linear system solving, which had been the piece of the puzzle I'd been missing. UMFPack is an open source library which contains routines to solve asymmetric sparse linear systems (and coincidentally is also used by Matlab to implement the "sparse" command). I downloaded and installed UMFPack into my home area, and started reading the user documentation. Use of the library seemed to involve generating a matrix of values, and then using UMFPack to factorize this matrix in 3 stages:

1. Perform a symbolic analysis of the matrix, and generate a "symbolic object" with results of the analysis in it.
2. Use this symbolic object to perform a numeric analysis of the matrix, and generate a "numeric object" with results of the analysis in it.
3. Use this numeric object to factorize the matrix.

I also started looking at how Gimp used UMFPack, and at this point I had an epiphany. I noticed that the Gimp code called a UMFPack function called "*umfpack_di_triplet_to_col()*", which didn't seem to have anything to do with linear system solving. Investigating this function forced me to look up linear system solving in much more depth than I had previously, and provided the missing piece of the puzzle.

Linear Systems

A system of linear equations is a set of linear equations of the following form:

$$\begin{aligned} a_{11}x_1+a_{21}x_2+a_{31}x_3+\ldots+a_{n1}x_n &= b_1 \\ a_{21}x_1+a_{22}x_2+a_{23}x_3+\ldots+a_{n2}x_n &= b_2 \\ \vdots \qquad\qquad & \vdots \\ a_{1m}x_1+a_{2m}x_2+a_{3m}x_3+\ldots+a_{nm}x_n &= b_n \end{aligned}$$

Figure 11.: a generalised linear system in algebraic form

where:

- a_{ij} are all constants, and are called the coefficients of the system
- $b_{1..n}$ are all constants, and are called the results of the system
- $x_{1..n}$ are constants, but their values aren't known. The problem is to find values for x that satisfy all the equations in the linear system

Such a linear system can also be expressed in matrix form:

$$\begin{pmatrix} a_{11} & a_{21} & \cdots & a_{n1} \\ a_{21} & a_{22} & \cdots & a_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1m} & a_{2m} & \cdots & a_{nm} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

Figure 12.: a generalised linear system in matrix form

So, this problem can be represented as A**x**=**b** , where A is a matrix of coefficients, **x** is a vector of unknowns, and **b** is a vector of results. Such a representation would obviously make sense for a problem such as this one described in the SIGGRAPH paper, where each pixel in our image is affected by all other pixels around it.

Let A be a matrix of weight values generated by the weighting function mentioned in the SIGGRAPH paper, and let **b** be a vector of colour values from our artist-marked image. If we factorised our equation and calculated the values in vector **x**, the resulting vector would contain new colour values for our image, where each pixel's colour had been colorized according to the weightings specified in matrix A. In other words, we would have successfully colorized the image in the manner described in the SIGGRAPH paper.

Sparse Matrices

Unfortunately, representing matrix A just as 2D array of floating point numbers would be very wasteful, and exceedingly memory intensive. Imagine that we were colorizing an image of size 100 pixels x 200 pixels. Our matrix A for such an image would require one column for each sample per pixel, and one row for each pixel in the image. For our example image, this would mean a matrix with ((100 x 200) x (100x200x9)) elements, or 3,600,000,000 elements. The vast majority of these elements would be zero, however – only (100x200x9m or 18,000) of those elements would even have the potential to be non-zero. Such a matrix is referred to as a *sparse matrix* – a matrix populated primarily with zeros. There are two common ways of representing such a matrix in memory more efficiently. The first is a method called "*triplet form*":

$$M=\begin{pmatrix} 0 & 0 & 0 & 0.1 & 0 \\ 0.2 & 0 & 0.3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.4 & 0 \\ 0 & 0 & 0 & 0 & 0.5 \end{pmatrix} \qquad Ax=\begin{pmatrix} 0.1 \\ 0.2 \\ 0.3 \\ 0.4 \\ 0.5 \end{pmatrix} Ai=\begin{pmatrix} 4 \\ 1 \\ 3 \\ 4 \\ 5 \end{pmatrix} Aj=\begin{pmatrix} 1 \\ 2 \\ 2 \\ 4 \\ 5 \end{pmatrix}$$

Figure 13.: An example sparse matrix M on the left, represented more efficiently in triplet form with 3 column vectors on the right

Triplet form represents any matrix as three column vectors "Ax", "Ai" and "Aj" where:

- Ax[i] contains the data stored in the 'i'th non-zero element of A (running across the matrix rows from top to bottom)
- Ai[i] refers to which column the 'i'th non-zero element of A is in (where a value of '1' means the first column)
- Aj[i] refers to which row the 'i'th non-zero element of A is in (where a value of '1' means the first row)

As you can see, this is much more efficient than storing the whole matrix, as triplet form here only requires 15 elements of data compared to the full 25 required by matrix A. The gains in larger sparse matrices are even more pronounced.

A second even more efficient way of representing sparse matrices is with a method called "*compressed column notation*":

$$M = \begin{pmatrix} 0 & 0 & 0 & 0.1 & 0 \\ 0.2 & 0 & 0.3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.4 & 0 \\ 0 & 0 & 0 & 0 & 0.5 \end{pmatrix} \quad Ax = \begin{pmatrix} 0.2 \\ 0.3 \\ 0.1 \\ 0.4 \\ 0.5 \end{pmatrix} Ap = \begin{pmatrix} 0 \\ 1 \\ 2 \\ 5 \end{pmatrix} Ai = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 3 \\ 4 \end{pmatrix}$$

Figure 14.: An example sparse matrix M on the left, represented even more efficiently in compressed column notation with 3 column vectors on the right

Compressed column notation again represents any matrix as three column vectors where:

- Ax[i] contains the data stored in the 'i'th non-zero element of A (running down the matrix columns from left to right)
- Ap[i] is the index of those data elements in Ax that start new columns (where a value of '0' means the first element of Ax)
- Ai[i] refers to which column the 'i'th non-zero element of A is in (where a value of '0' means the first column)

UMFPack takes its input matrix in compressed column notation, but also provides the function "*umfpack_di_triplet_to_col()*" to convert a triplet form matrix into compressed column notation. It was the reference to this function in the Gimp plugin code that put me on the trail of sparse matrices, and finally helped me understand how this algorithm could be implemented. I now understood the theory that underpinned how both the Matlab reference implementation and the Gimp plugin code worked, and I found the code much more accessible because of this.

Implementation

I could see now that the implementation of the sparse matrix would be the key to the implementation of the algorithm. I had a choice between either implementing a triplet form matrix and then converting it to compressed column notation with UMFPack before feeding it to UMFPack's solver, or implementing a compressed column notation matrix directly. Triplet form is conceptually simpler to program (since it essentially only consists of data and coordinate triplets), whilst compressed column notation offers memory savings at the expense of complexity and processing time. After a lot of thought, I decided to implement a triplet form matrix, since memory wasn't particularly at a premium and I was far more concerned with getting a working implementation at this stage than making it as efficient as possible.

I created a simple class called "*tripletMatrix*" to hold all the matrix functionality. I set the matrix up so that it would start its existence as a triplet matrix. Triplets could be added to the matrix one at a time through the "*tripletMatrix::addTriplet()*" function call, with memory for them allocated at construction time and freed at destruction time. The matrix could then be converted to compressed column notation through a call to "*tripletMatrix::convertToCompressedColumn()*" (which in fact simply called *umfpack_di_triplet_to_col()*). From this point on triplets could no longer be added to the matrix. I also didn't provide functionality to convert back from compressed column notation to triplet form, as it simply wasn't needed and would have complicated matters unnecessarily.

I understood now that I would need to be able to access each image colour channel as a

separate array within my *yuvimage* class in order to be able to pass it into UMFPack as a column vector (which would be necessary at the solving stage when UMFPack would need access to individual channels of the marked image). Up until now I had been interleaving channel data within *yuvimage* – that is, all image data had been stored in a single array rather than separated into individual channels. This meant I had to rewrite large portions of the *yuvimage* class as well.

With the groundwork in place, I started to write a *filter()* function that took a black and white *yuvimage*, a colour-marked *yuvimage*, and an empty *yuvimage* to write its results to. I broke down what needed to be done into a series of steps:

1. For each pixel, if it hasn't had its colour marked by the user:
   i. Collect intensity samples in a window around the pixel
   ii. Calculate the variance of those samples (necessary for step 3)
   iii. Generate a weighting value for each sample around the pixel, and store it in the *tripletMatrix*
2. Convert the matrix to compressed column notation
3. Perform symbolic and numeric analysis on the matrix with UMFPack
4. Factorize the matrix against both marked U and V channels
5. Copy the resulting output U and V colour values into the result *yuvimage* along with the Y channel intensity values from the original black and white *yuvimage*

This was relatively straightforward, and I managed to implement this without too much trouble. The problem was when I ran it, it didn't work. FIXME – mention float rounding problems here.

Bug Fixing

It took me a long time and a lot of effort to get my algorithm implementation debugged, and I learnt a lot from the process, so I feel that it's worth documenting some of the stages I went through. The first time I ran the filter, it looked like nothing was happening. The top half of the filtered image looked almost identical to the marked input image. However, the bottom half of the filtered image was completely black. After reading my code over a few times to check for obvious logic bugs and not finding any, I started debugging my image by printing out colour and weight values at various points of the process. This quickly revealed the problem – on some of the problematic pixels, some pixels were generating ridiculous weight values (infinity, as opposed to the range 0 <= x <= 1 where they should have been). The cause of this problem turned out to be the code responsible for calculating colour weighting. Since I was using the weighting function described in figure 5, at one stage this involved doing a division by the variance. However, when the variance tended towards 0 (for example when sampling pixels which all had nearly the same intensity), this division operation would generate results that would tend towards infinity, and that would in turn generate ridiculous weighting values. This was easily fixed by clamping the variance so that it was never smaller than 0.000002. This fixed the problem where half the image would appear in black – now the whole image simply looked unchanged.

The next problem was to work out why the filter didn't seem to be doing anything. I checked all key parts of my code individually by printing out weighting information, and everything seemed to be working the way it should. Eventually I noticed that UMFPack's numeric analysis and solving functions were returning 1, as opposed to 0. A check in UMFPack's header file found that I was getting a warning that the input matrix I was giving it was singular and non-invertible, and therefore

any solutions generated by UMFPack would probably contain invalid numbers like infinity or NAN. This was deeply confusing, and very difficult to debug. Since the matrix I was giving it was obviously very large, manually inspection was impractical. However, there was no information that I could get from UMFPack that would help me determine why the matrix I was giving it was non-singular. The matrix as a whole was either singular, or it wasn't – if there was any one value that was causing the operation to fail, I had no way of knowing which it was. I had no way to narrow the problem down so I could identify its source. Furthermore, if I tinkered with the code and changed what values were being added to the matrix or in what order, UMFPack would occasionally solve it and generate new colour values (although the colours were often appeared to be little more than random noise). This confused me even further. I ended up having to compare my code almost line for line against the reference implementations to try to find where it was going wrong.

I eventually identified two problems rather than just one (hence why simply tinkering with the code didn't really work, as there were two factors at play in causing the bug). The first problem was that non-marked pixels in the marked image needed to have their U and V values explicitly set to 0. This was causing invalid matrix input. Fixing this caused UMFPack to stop returning 1 and actually start solving the linear system (although it still appeared to be generating random noise for colour values). The second problem was simply a typing error on my part in two locations – when calculating array offsets from x and y coordinates, I had written:

```
int sampleOffset = x + (y * input.height);
```

instead of :

```
int sampleOffset = x + (y * input.width);
```

The filter now generated mostly good colour values. You could already see the potential results of the filter. Unfortunately in dark areas of the image, the filter sometimes generated very bright and obvious splodges of pure blue. My first instinct was to search for the pixels in question and then trace back to try until I found why they weren't working. Oddly enough, a search of the output image for any pixels with high values of blue came up empty, even though they were plain to see on screen. It turned out that the YUV to RGB colour conversion would occasionally generate values of blue that were less than zero. Inspection of the YUV to RGB conversion matrix (figure 9) showed that this was certainly possible – if a pixel had the YUV value of 0, -0.5, 0 (a very dark blue) this would generate a blue value of (2.032 x -0.5), or -1.016. When the texture was being converted to an OpenGL texture, OpenGL must therefore have been wrapping negative numbers to the positive end of the scale, hence why dark colours would suddenly have large obvious patches of pure blue in the middle of them. By clamping the results of RGB/YUV conversion to the range $0 <= x <= 1$, the blue patches went away and the filter now worked perfectly.

Example images

All of the images here were generated using my code. Whilst filter time did vary between images (largely depending on image resolution) most images were solved on a P4 3GHz processor in under 15 seconds. All images were created with a sampling radius of 1 – values greater than this would lead to better results, but at vastly increased calculation time.

Conclusion

I have learnt a lot from this project and the practical experience that it has given me. This is the first time that I've ever had to implement a largely theoretical concept, and I've learnt that you need to approach such problems very differently from normal computing problems. My normal working process when programming is to start with a small base and then slowly build features up on top of this base, one little step at a time. This technique didn't work here, however - I made very little progress implementing this algorithm until I started exploring the mathematical theories underpinning it such as YUV colour space, linear systems of equations, sparse matrices, and various notations for them. It was only once I had a solid grip of all of these foundation topics that I was able to find a way to approach the problem at all and implement a solution for it.

The code itself is now ready to be used as a plugin for any image manipulation package that supports a plugin mechanism. It would be a trivial exercise to take this code and integrate it with any paint package that can supply the necessary inputs as an array of RGB values. It would've been nice to have done this myself as part of this project, but ultimately I had to put a limit on what I wanted to do for the sake of my other two projects this year..

Throughout this project I have only focused on colorization of single images. The SIGGRAPH paper actually goes further than this and suggests that by changing the definition of "the neighbours of a pixel" to also include those one frame ahead and one frame behind the current frame (after accounting for motion between frames) a similar colorization can be achieved on video without the need for marking every frame. Indeed, the home page of one of the paper's authors gives some very impressive technical demonstrations of this concept. In one video example that they give, an excellent colorization has been achieved despite the fact that only 7 frames out of an 83 frame video have been marked by the artist. It would also be an interesting expansion of this project to take the code here and attempt to apply it to image sequences, particularly in conjunction with video post-processing tools such as Shake or After Effects.

Another interesting avenue for exploration would be speeding the technique up. At the moment the calculation times are respectable, but certainly could be improved. It would be an interesting exercise to attempt to implement this algorithm as a GPU fragment shader, and see if the specialised hardware of the GPU could be leveraged to deliver speed increases. There are also many techniques for solving linear systems – Gauss-Jordan elimination, Cholesky decomposition and Levinson recursion to name a few. A different choice of solver could yield large changed in algorithm performance.

Overall, I feel that this project has been a success. I set out to learn about image colorization, and was successful in that respect. I have implemented an algorithm which addresses the weaknesses of previous techniques and allows an artist to colorize single images intuitively without the need for manual image segmentation or manual colour variation. The result is a noticeable improvement on previous efforts, and one that I'm proud of.