# Case Study in Scientific Computing: Image Colourisation

Zella Baig

*HT23*

# 1 Introduction

Image colourisation is concerned with trying to recolour a greyscale image given a small amount of information of the true colour values at some points within the image. In practical contexts, this problem may arise out of wanting to recolourise historic photographs, or simply to restore information which has been lost (such as e.g. destroyed frescos of which there exist black-and-white photography).

## 1.1 Images

In the context of this problem we consider only rectangular images. We consider images of two different formats: JPEG and PNG, chosen for their ubiquity on the web[1]. The main difference between the two formats which is of concern to our project is that while JPEG images store three colour channels containing information on red, green, and blue intensity (which we refer to as the "RGB values"), PNG images contain an optional fourth "alpha" channel which contains information on transparency [9] and hence must be accounted for.

Before proceeding, we take a small detour to discuss "floats" and "integers" in the context of computing. An integer can be thought of as a data type representing a value in the set $\mathbb{Z}$. *Unsigned integers* are then simply all non-negative integers: unsigned integers contain no information of whether they are positive or negative. *Floats* or floating-point values are then to integers in computing what the reals are to integers in mathematics, with the caveat that floats are never stored exactly. For example, in Python, floats are stored by default to "double precision", or 53 bits worth of precision [1].

We now refer to $n$-bit values, a general concept within computing. This concept refers to data types that can store any value using up to $n$ bits of their associated type (either integers or floats). For example, an 8-bit unsigned integer may store any integer $n$ which lies $0 \leq n \leq 255 \, (= 2^8 - 1)$, while a *signed* 8-bit integer may store any value integer between $-128$ and 127. Related to this concept is the idea of "integer overflows", which is what happens when a value is written in a format which cannot support it, such as trying to store 256 in an 8-bit unsigned integer format [15]. In this case, the value stored would "wrap around" and be stored as 0.

Proceeding and ignoring the existence of the alpha channel for the moment, an image of size $N \times M$ pixels can be thought of as an $N \times M$ array, with each entry of this array containing RGB values for a single pixel. For a given channel, the intensity

[1] For an unrigorous illustration via a webscraper, see [18].

of the colour can range from 0 to a maximum value (either 1.0 as a float, or 255 as an unsigned 8-bit integer).

## 2 Formulation of Problem

We wish to mathematically represent our problem. To this end, consider a domain $\Omega$ comprising of discrete points $\{z_i\}, 1 \leq i \leq m$. Here each $z_i$ represents a pixel within our domain $\Omega$ which corresponds to the image itself. Also define a subset of $\Omega$, $D := \{x_i\}, 1 \leq i \leq n$, which represents the pixels within $\Omega$ which have associated colour information given. Importantly we place the bound that $n < m$ and that $x_i = z_j$ for some $z_j$, for all $i$. This serves to impose the condition that we have fewer coloured pixels than there are pixels in the image, and that each coloured point does indeed correspond to some pixel in the image.

We also define the greyscale map $\gamma : \Omega \to I^3_\gamma$; $I^3_\gamma := \left\{(i,i,i) \in \mathbb{Z}^3 | 0 \leq i \leq 255\right\}$ at all $\Omega$. In other words, $\gamma$ represents a map from a given point $z_k$ to a tuple of three values which we shall use to represent the RGB values $(r_k, g_k, b_k)$ at the pixel associated with $z_k$. Further, we impose the condition that $r_k = g_k = b_k$ to ensure that the resulting tuple is a shade of grey (with white represented by $(255, 255, 255)$ and black by $(0,0,0)$). We formalise knowing the colour information in $D$ by defining the map $f : D \to I^3 := \left\{(i,j,k) \in \mathbb{Z}^3 | 0 \leq i,j,k \leq 255\right\}$ where $I^3$ relaxes the condition that the RGB values must be the same. The goal of the colourisation is to find a map $F : \Omega \to I^3$ such that $F|_D \approx f|_D$, that is to say that we expect the RGB values mapped by $F$ to approximately equal the true values known by $f$ within $D$. We also introduce the notation $F^c_j$ to represent the RGB information of a point $z_j$ in a single colour channel, i.e. $c \in \{r, g, b\}$. Similarly, let $\gamma_j$ be shorthand for representing the greyscale information $(\gamma_j, \gamma_j, \gamma_j)$ at point $z_j$.

In order to obtain the greyscale information at a given point we define the *monochrome luminance signal* [19] as

$$\gamma_j = 0.3r_j + 0.59g_j + 0.11b_j. \tag{1}$$

This linear combination of the RGB values at a given point then allows us to construct an artificially coloured grey image, which we may then subsequently add colour information to.

## 2.1 Kernels

Before further exploration of the mathematical formulation of the problem let us take a moment to examine *kernels*. Kernels allow us to represent comparisons between different abstract objects $x_i$, which lie in some set $\mathbb{X}$, as matrices $K$ with entries $K_{i,j} = k(x_i, x_j)$ for some function $k(x, y) : \mathbb{X} \times \mathbb{X} \to \mathbb{R}$. Importantly we are able to do this whilst remaining agnostic to the actual form of the objects themselves provided we can define an appropriate $k$ [14]. For a more concrete formulation, we desire $k$ to be some function which computes the *similarity* of the elements $x, y$. If we consider $x, y$ to just be cartesian coordinates, a simple example might be $k(x, y) := \|x - y\|$. Here, $k$ is large when $x, y$ are very far apart from one another, and small when $x, y$ are close.

Let us bring this idea back to image recolourisation. We define the *radial basis function kernel* $K$ using

$$k(z_i, z_j) := \phi\left(\frac{\left\|\xi_i - \xi_j\right\|_2}{\sigma_1}\right)\phi\left(\frac{\left|\gamma_i - \gamma_j\right|^\rho}{\sigma_2}\right), \tag{2}$$

for some parameters $\{\rho, \sigma_1, \sigma_2\}$ and function $\phi(r) := \exp\left(-r^2\right)$, where $\xi_i$ is the *euclidean position* of point $z_i$ within our rectangular image. Let us examine this kernel in more detail.

$\phi(r)$ is a positive function with a maximal value of 1 at the origin, and which tends to 0 as its argument grows. Furthermore, $\left\|\xi_i - \xi_j\right\|_2$ is just the metric on $L_2$ space of two given points within the image, and $\left|\gamma_i - \gamma_j\right|$ is a measure of the difference of greyscale values between those two points as well. Thus, we can say that (up to scaling by $\rho, \sigma_1, \sigma_2$) this kernel represents a measure of similarity of *pixel location* and *greyscale value*, approaching a maximum of 1 when the two points are both close and of similar greyscale values, and approaching a minimum of 0 when these two points are very far, different in greyscale value, or satisfy both these criteria. In the section that follows let us build up a method of colourising an image with the information that we have, in a very similar procedure to that in [5].

For $z \in \mathbb{R}^m$, $a^c \in \mathbb{R}^n$, let us consider the map $F^c(\cdot\,; a^c)$ of the form

$$F^c(\cdot\,; a^c) := \sum_{j=1}^{n} k\left(\cdot, z_j\right) a_j^c. \tag{3}$$

We claim that such a map serves as a "recolourisation" map for a given colour channel $c$, with the justification for doing so being that (3) maps similarity in both position and greyscale[2] between a given pixel in $\Omega$ and the colourised points, with weightings $a_j^c$.

[2]In other words all the information we have for a given pixel in the greyscale image.

Now consider the functional

$$L(F^c) = \frac{1}{n}\sum_{i=1}^{n}(F^c(z_i;a^c) - f^c(z_i))^2 + \delta||F^c(\cdot;a^c)||^2 \quad (4)$$

which may be thought of as a least-squares-esque loss function between $F^c$ and $f^c$ with an additional "smoothing" term associated with the parameter $\delta$, for a colour channel $c \in \{r,g,b\}$. Importantly, we know the map $f^c$ is just the specific colour information for a given channel $c$ for all the coloured pixels in our otherwise greyscale image. We seek to minimise this functional with respect to $a^c$, which corresponds to seeking the weights in $F^c$ that lead to the "best-possible" recolourisation of our image when measuring against how much recolourised RGB values differ for points that we have true RGB information for.

To proceed, let us first define the matrix $K_D \in \mathbb{R}^{n\times n}$ via $(K_D)_{i,j} := k(z_i,z_j)$, and subsequently note that

$$\sum_{i=1}^{n} F^c(z_i;a^c) = K_D a^c, \quad (5)$$

and

$$||F^c(\cdot;a^c)||^2 = (a^c)^T K_D a^c. \quad (6)$$

Using (5) - (6), and noting that

$$\frac{\partial \sum_{i=1}^{n}(F^c(z_i;a^c) - f^c(z_i))^2}{\partial a^c} = 2\sum_{i=1}^{n}(F^c(z_i;a^c) - f^s(z_i))^T K_D, \quad (7)$$

we take $\partial_{a^c}$ of (4) to see that

$$\partial_{a^c} L(F^c) = \frac{2}{n} K_D \left(K_D a^c - \bar{f}^c + \delta n a^c\right), \quad (8)$$

where $\bar{f}^c := (f(z_1), f(z_2), \cdots, f(z_n))^T$. Setting the left hand side in (8) to 0 we see that, provided $K_D$ is non-singular, the unique root of this expression occurs when

$$a^c = (K_D + \delta n I)^{-1}\bar{f}^c. \quad (9)$$

Using this result, we may insert the values $a_j^c$ into (3) in order to construct our map $F^c$; importantly, we have shown that $F^c$ as defined by $a^c$ in (9) is the *best possible* map of the form specified.

Further, we wish to implement this colourisation routine in a program with an associated graphical user interface (GUI). The goal with such an implementation is that an end user should be able to select a given image, have it be converted to greyscale, have some colour added back in, and then finally recoloured.

# 3 Computational Solution

Now that we have established the mathematical overview of the problem, let us discuss the broad principles which will underline our computational approach towards this problem.

Upon loading the image, we desire the greyscale image to be represented in an $N \times M$ array. Note here we derive $m$, the number of elements in $\Omega$, via the relationship $m := MN$. Further, we assume that we already have $n$ pixels within this greyscale image array to have colour information inserted at the points $\{x_i\}$, and let us also assume that we know the indices of the array at which these points reside.

As an aside, note that we also discard any transparency information by enforcing a pure white background where there is any transparency information. In this manner we do not need to consider this aspect of PNGs.

We also note that as as $F^c$ acts on every pixel within $\Omega$, it would be computationally ideal to try to create vectorised functions which bypass the need for lengthy for-loops over each pixel. Using this as motivation, we seek methods to allow us to compute the resulting $N \times M$ matrix via a series of matrix operations. Consider the matrix $K_D$, which is simply the $n \times n$ kernel matrix evaluated at each of the points where we have colour information and is the first thing we require to construct $a^c$, whose values are then used when constructing our map $F^c$. We note, by considering (3), that we can construct this matrix map without looping over indices if, for example, we generate an $m \times n$ matrix $T$ (where $T_{i,j} = k(z_i, x_j)$), right multiply by the $n \times 1$ vector $a^c$, and reshape.

Crucially, this allows us to generate $T$ *without* consideration of which colour channel we are currently filling in; in (3) it is only the $a_j^c$ values which depend on the channel and as such our matrix $T$ may be thought of as a "template" matrix, depending only on the kernel with the kernel functions $k(z_i, x_j)$ taking all $z_i \in \Omega$ and all $x_j \in D$. Perhaps the most appealing aspect of such an approach however, is the fact that we expect in the colourisation that the construction of this matrix $T$ to be the most computationally intensive step. In general, the number of total pixels is greater than the number of coloured pixels input, and for large images this can be true by several orders of magnitude. Thus, by constructing $T$ *before* we solve for $a^c$, we potentially reduce computation time by a factor of three (via not needing to repeat the calculation for each of the three colour channels).

Using these steps, we can now construct a code template of how we might colourise an image which we present as Code 1, which while seemingly simple actually presents a fair amount of complexity hidden in the steps to construct $K_D$, $T$, and $Ta^c$ when we

consider practical limitations. We present further discussion of these in Section 4.1.

**Code 1:** Pseudocode for main colourisation function

```
def colourise:
    Construct K_D;
    Construct T;
    for c = {r, g, b} do
        Compute a^c = (K_D + δnI)^{-1} f̄^c;
        Compute Ta^c;
        Reshape Ta^c to an N × M matrix L^c;
        Set the image colour channel in c to L^c;
    end
```

# 4 Practical Implementation

For the practical implementation of this project, we chose to use Python for several reasons, with the two primary reasons as follows.

1. Access to a wide library of both scientific, diagnostic, and GUI packages to application with aesthetically pleasing results.

2. Familiarity with the language over alternatives such as e.g. MATLAB.

With this decision in mind, one of the first choices to make was to decide on the GUI framework to utilise. The choice of these frameworks was in some way guided by the decision to work with Matplotlib [6], a Python visualisation library, for the displaying of the images generated. Some of the choices considered for the GUI were Matplotlib itself (given the capability for interactive figures), PyQT [11] (a Python toolkit incorporating the popular Qt framework, featuring extensive customisability), and Tk [17], a standard Python interface.

The decision was made to go with Tk given the somewhat simpler learning curve, and native support of Matplotlib interactive figures within Tk windows. However, one of the key drawbacks of going with Tk was the somewhat dated UI; this issue was rectified by instead switching to go with CustomTkinter [12], a drop-in replacement for Tk which comes stylised with “modern” presents for the appearance of objects within the framework, such as buttons.

The next major decision to enact was the structure of the overall project, in terms of whether to approach the problem via an *object-oriented-programming* (OOP) approach, in which we consider "objects" which have "attributes" which may be altered via "methods" [21], compared to a *procedural programming* approach [22] in which we have "procedures" (or functions) which linearly act upon some object to alter it.

The decision was made to go with the OOP approach given that the structure of the project lent itself far more readily to such an implementation, for several reasons. Firstly, GUIs lend themselves incredibly well to implementations in OOP paradigms given the structure of the actual "windows" which themselves can be split into objects such as buttons, displays, and the like - each of which may be altered in predefined manners. Furthermore, in OOP methods different "classes" of objects may "inherit" from other classes; this idea can be extended quite readily to modularise the codebase such that we would be able to completely detach the main GUI from any function which would colourise images. Not only would this lend itself to rapid extension and reformation of the codebase without harming functionality of aspects which do not relate to one another[3], but it also ensures a cleaner codebase since methods and attributes could be constrained to their own class. An example of this may be to have a method to construct buttons, which would be limited to the GUI class. We present a pseudocode outline of the structure of the project in Code 2, given in Appendix B.

Another practical decision made was to employ version control via Git [4]. This would enable the project to have a synced workflow, and also enable concurrent development of new features to be implemented in the codebase (which are then "merged" when finished) without risking the main trunk.

For the computational aspect of the implementation, we work extensively with NumPy, a library with a plethora of in-built matrix manipulation functions which importantly also offers *vectorisation*, in which many operations take place in pre-compiled C code (allowing for much faster performance, with the main drawback being the requirement for typed data within *ndarrays*, the NumPy equivalent of a Python array) [20].

## 4.1 Numerical Constraints

Several issues presented themselves when commencing work on the project, primarily concerning memory allocation. Let us take a moment to explore this in more detail, within the context of NumPy, and our matrix $T$. We also note that when loaded, JPEG

[3] Concretely, if the colourisation class is separated from the GUI class, neither of the two would rely upon the other such that they may be developed concurrently.

and PNG files are converted to arrays containing tuples of RGB values in each of the $m$ entries. However, these values may either be stored as unsigned 8-bit integers, or as floats. As we perform matrix operations on this array which leads to arithmetic which is not constrained in any way, we must immediately convert these values to floats before further processing within our colourisation routine. This prevents, for example, incurring overflows performing a multiplication on an unsigned integer which leads to a value greater than 255.

Consider now 64-bit floating point values (the "standard" for non-integer values in NumPy), which each occupy 64 bits in memory [3]. Now consider loading a $1000 \times 1000$ image, coloured with $n = 1000$ pixels (which amounts to 0.1% of the image being coloured), which when loaded via Matplotlib into a NumPy array takes the form of a $1000 \times 1000 \times 3$ array. The matrix $T$ is then a $1000^2 \times 1000 \times 3$ array of 64-bit values, which amounts to $\approx 7.5$ gibibytes (GiB) - approaching the 8 GiB available to most consumer-grade laptops *in sum*. Despite it being possible to construct some arrays, such as if no intermediate arrays were created (which *are* created by default in NumPy expressions), but we sought to protect against memory loss issues as even just the creation of the GUI entails further memory usage.

A reasonable, and straightforward solution at this point is to instead perform any construction of large matrices columnally. However, this involves lengthy for-loops, which add significant timing costs for our program. We explored different avenues to speed this process up. One of the initial avenues we explored was to employ just-in-time compilation with Numba [7], a Python library which allows certain functions (largely limited to NumPy and pure Python functions) to be compiled to C code at execution, potentially allowing for significant boosts in execution times for functions where there does not exist a NumPy replacement.

However, this approach did not yield any notable benefit as the benefit of faster processing times for images came with the drawback of a notable time to compile the requisite binaries; on the iteration of the program tested Numba was *slower* when running on a single small ($\approx 250 \times 250$ pixel) image. Further, casting the code in a format which allowed for Numba compliance (incorporating loops going over array indices as opposed to array slicing) further slowed code down, but did indeed solve the issue of running out of memory. Nevertheless, given the performance issues for smaller images we sought another solution.

The next library we tried was NumExpr [8], a limited library compatible with a select few NumPy and Python functions, which nevertheless offers speeds up to an order of magnitude faster for certain operations on large arrays. NumExpr achieves

this performance boost through automatic parallelisation of the parsed code, which is "chunked" (or split into smaller elements) before being run on a virtual machine (written in C) on the CPU. Further, NumExpr avoids any allocation of memory for steps between input and output leading to further memory access reduction. This enormous performance increase (resulting in a real time boost of several seconds for images $\sim 700 \times 700$ pixels in size, which initially took $\sim 10$ seconds) came with another drawback alongside the aforementioned non-support of most NumPy functions, namely that array slicing is not supported; this ultimately did not present too much issue as predefining the slices of arrays to be evaluated and then passing this new variable to NumExpr was still faster than either NumPy, or via trying this approach in Numba-based methods. Thus, the decision was made to, wherever possible, utilise the NumExpr method.

At this point the only further memory-related issue to consider was consideration of if the matrix $T$ was small enough to load into memory entirely or not; in the former case the operation $Ta^c$ could be done in one step as outlined in the Code 1, else this operation would need to be done columnally as discussed in this section. For a comparison we ran benchmarking on an artificial scenario where we constructed an image of $m$ pixels, with $m$ varying from 25 to 800. We assumed $n = 500$ (leading to the unrealistic scenario where $m < n$; this is not a real concern in this section as we seek simply to profile our methods) and ran two different tests comparing the construction of $Ta^c$ outside the colour channel loop (the "fast method"), and another bypassing the construction of $T$ in its entirety and instead performing the construction $Ta^c = \sum_{j=1}^{n} T_j a_j^c$ (the "efficient[4] method"), thereby only needing to construct a given column $T_j$ at a time. The results are shown in Figure 1, where it is clearly seen that the "fast" method utilises exponentially more memory as $m$ increases. On the other hand, whilst both methods display an exponentially increasing time to generate the colourised image, the "fast" method rises at a much slower rate against $m$.

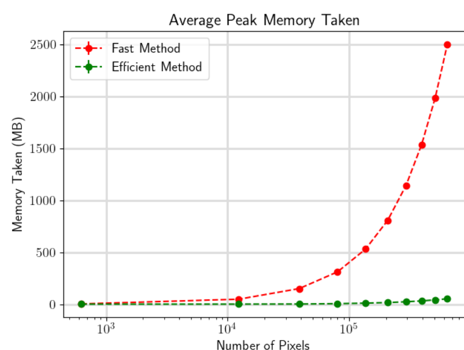
# 5 Results

In our final program note that we have, beyond the allocation of $n$ randomly placed coloured pixels, incorporated two further methods to add colour information into the greyscale image:

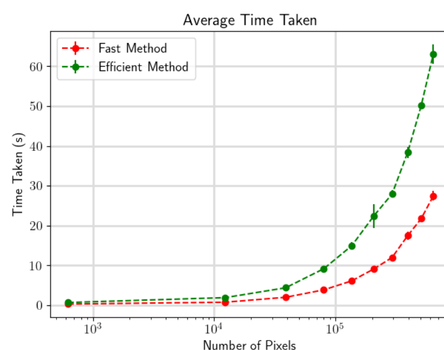
1. Via the "Color by Pixel" method[5] the user is able to select a colour from a palette,

[4]Efficiency here means memory-related efficiency.

[5]The decision was made to keep all in-program spellings to the American versions as to not clash with




(a) Average peak memory usage for the "efficient" and "fast" methods as a function of $m$.




(b) Average time usage for the "efficient" and "fast" methods as a function of $m$.

Figure 1: Comparison of averaged performance metrics of the "efficient" and "fast" methods for construction of $Ta^c$ when colourising an image as a function of the number of pixels on input image ($m$), sampling over 10 values of $m$ and repeated thrice per $m$.

and apply a "paintbrush" type effect where a brush of varying size may be used to colour the image.

2. Via the "Color by Grid" method the user may manually place a $2 \times 2$ block of correctly coloured pixels onto the greyscale image by selecting with the mouse.

Further, we allow for on-the-fly updating of the parameters $\delta, \rho, \sigma_1$, and $\sigma_2$, as well resetting of the colourised image such that an alternative colourisation base may be set. A screenshot[6] of the final GUI is presented in Figure 2.

## 5.1 Dataset

Prior to further discussion of results, note that all training and analysis is done on a set of 60 images split into two subcategories: "cartoon" and "real life" style images. Each of these subcategories is then split into two with half the images being $256 \times 256$ pixels, and the other half being $512 \times 512$ pixels.

NumPy spellings, which use American versions.

[6]Note that part of the filepaths in the screenshot have been obscured to hide identifying information.

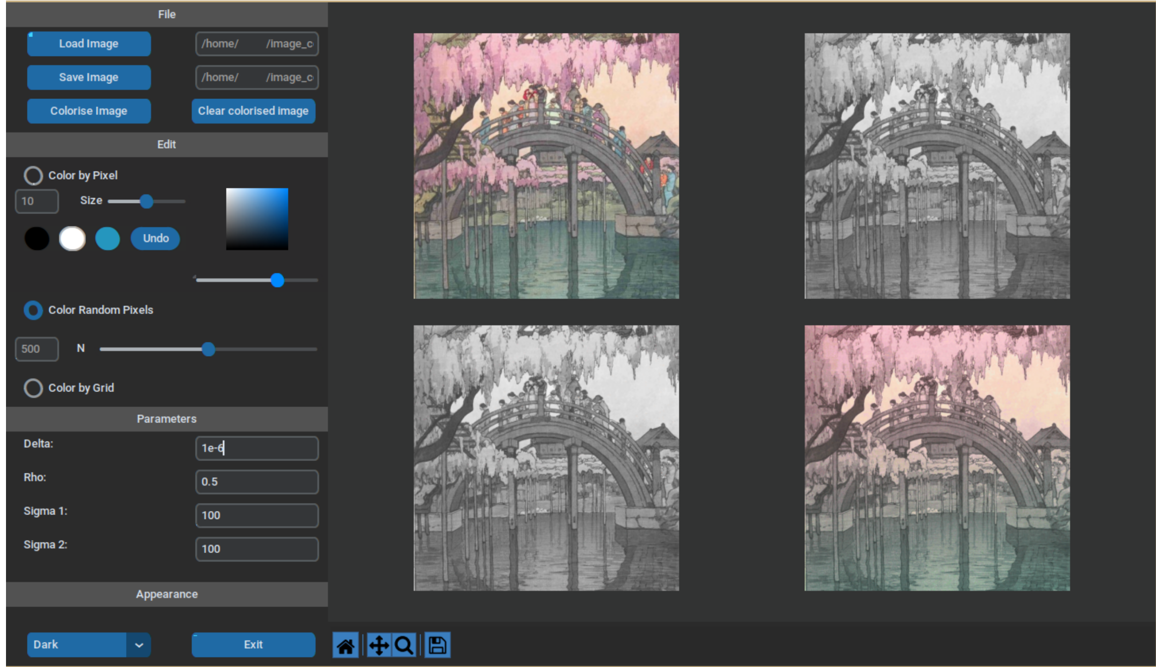



Figure 2: Screenshot of the GUI, with the original $512 \times 512$ image ("bridge-512") on the top left, the greyscale image on the top right, the greyscale image with $n = 500$ randomly placed pixels with the correct colour on the bottom left, and the colourised result on the bottom right.

## 5.2 Optimisation

As is seen qualitatively from Figure 2, the colouriser works to a reasonable degree: with only $\approx 0.008\%$ of the pixels initially coloured in, we get an image which appears broadly correctly coloured to the eye. In order to determine a more quantitative measure for the success of our method, we must define some cost function $C$. In order to do so, let us first define by $L_0^c$ as the colour information for all $z \in \Omega$ in channel $c$ of the original image, i.e. the "true colour" in channel $c$. Then we define

$$C := \frac{1}{m}\left(0.3\left|L^r - L_0^r\right| + 0.59\left|L^g - L_0^g\right| + 0.11\left|L^b - L_0^b\right|\right), \tag{10}$$

corresponding to (roughly) a weighted measure of how incorrect each pixel in the colourised image is, on average. Note that we weight each colour channel using the same weights as in the monochrome luminance signal, in the hopes that these weights give some impression of how "important" each colour channel is; alternatively this may just be thought of as comparing the grey images (by channel) generated by $L^c$ and $L_0^c$.

With this cost function one of the first things we examined was normalising distances when constructing either $T$ or $K_D$. To elaborate, the first $\phi$ term in (2) takes as an input

the euclidean norm of the two points given to the kernel. In this sense the parameter $\sigma_1$ is some measure of both the *scale* of the image, as well as of how similar the distances between those two points are. We examine if we achieve better results using $C$ if we normalise this distance by, for example, dividing the norm by $m$. We expect once normalised, $\sigma_1$ should only account for the similarity of distances and so for any given parameter set, provide better values of $C$. In order to test this hypothesis, we run both versions on all 60 images in our dataset, and present the results in Table 1.

| Method: | Normalised | Un-normalised |
|---|---|---|
| Mean $C$: | 6.4 | 8.0 |
| Variance $C$: | 27.04 | 18.49 |

Table 1: Comparison of normalised and un-normalised kernel on all 60 images in dataset, for $\delta = 1 \times 10^{-4}, \rho = 0.2, \sigma_1 = \sigma_2 = 100$, and $n = 0.001m$ coloured pixels for each image.

Thus, while the normalised method seems to present a lower mean value for our cost function, we cannot say conclusively if it provides any significant benefit given the large variances. Examining the histogram of losses for the two methods (presented in Figure 9 in Appendix A) further supports the idea that it is potentially more beneficial to use the normalised kernel, given the distribution of losses appears to be concentrated at lower values of $C$.

Investigating now the effect of the parameters input into the model, we ran an optimisation routine on different images using BenderOpt [16], a hyperparameter optimisation library which utilises Tree-Structured Parzen Estimators [2], a standard method for black-box hyperparameter optimisation when the function being evaluated either does not have a gradient or is difficult to compute.

The optimisation routine was ran for 250 iterations on each image in the $256 \times 256$ pixel image set with a seeded 66 pixels chosen to be coloured in[7]. The losses (from our cost function $C$), and parameter values associated with the run which led to the lowest cost were recorded.

Looking at only "real life" images we examine the parameters that lead to the minimum value of $C$ in our optimisation run for the 15 images we test, shown in Table 2. We note that whilst we have been able to ascertain order-of-magnitude results for all the parameters, there is a very large amount of variance within $\sigma_1$ and $\delta$, as well as a significant amount within $\sigma_2$.

[7] 66 being $\approx 0.001m$ for $m = 256^2$.

| Parameter: | $C$ | $\delta$ | $\sigma_1$ | $\sigma_2$ | $\rho$ |
|---|---|---|---|---|---|
| Mean: | 5.4 | $2.6 \times 10^{-3}$ | 62.7 | 101.3 | 0.9 |
| Variance: | 11.56 | $1.7 \times 10^{-5}$ | $1.9 \times 10^{3}$ | $7.40 \times 10^{2}$ | $4.0 \times 10^{-2}$ |

Table 2: "Best loss" parameter mean values and variances ran on 15× "real life" images of 256 × 256 pixels in size.

Similarly, when ran on "cartoon" images we present present results in Table 3. Immediately, we can see that both $\sigma_1$ and $\sigma_2$ seem to be "type"-agnostic, displaying similar means and variances in either case. However, $\rho$ falls by a third as we go from "real life" to "cartoon" images. Similarly, the mean of $C$ over the datasets appears to fall as well. This latter result is somewhat as expected; "cartoon" images have sharper edges and less colour gradients and so need fewer $n$ to build up an "idea" of what other colours in the image should look like.

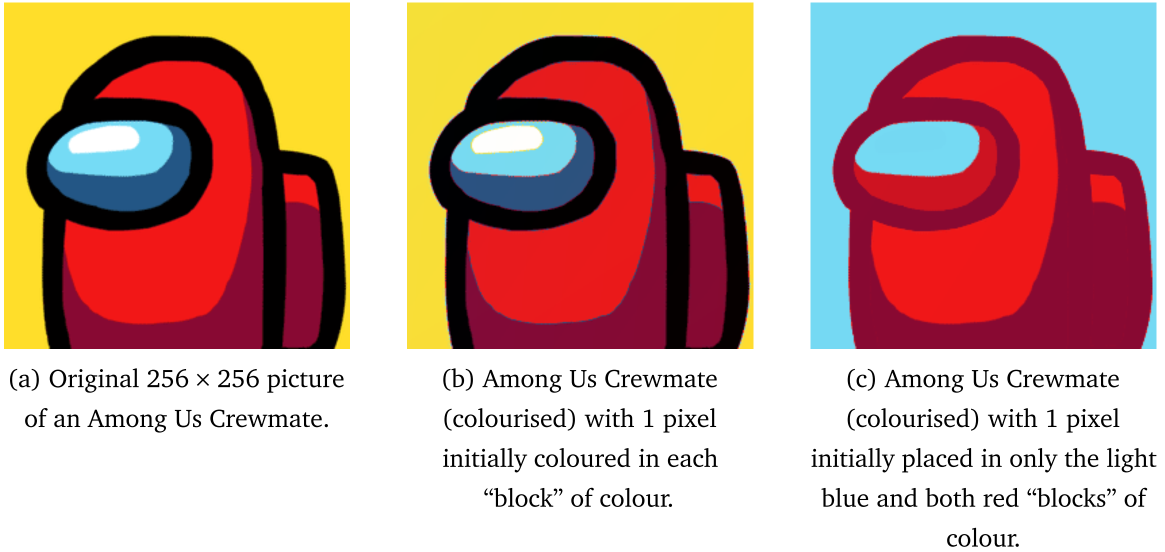
| Parameter: | $C$ | $\delta$ | $\sigma_1$ | $\sigma_2$ | $\rho$ |
|---|---|---|---|---|---|
| Mean: | 2.3 | $2.7 \times 10^{-3}$ | 60.9 | 93.5 | 0.6 |
| Variance: | 6.3 | $6.4 \times 10^{-5}$ | $1.6 \times 10^{3}$ | $1.2 \times 10^{3}$ | $4 \times 10^{-2}$ |

Table 3: "Best loss" parameter mean values and variances ran on 15× "cartoon" images of 256 × 256 pixels in size.

Further, at first glance the $\delta$ values also do not seem to change, which seems a surprising result as $\delta$ pertains to the smoothing we apply within the image. However, inspecting the dataset, we see that if we exclude the results for "green-circle-256.png", which has an optimal delta parameter of $3.2 \times 10^{-3}$, the mean for $\delta$ falls to $6.2 \times 10^{-5}$ and the variance falls to $2.9 \times 10^{-6}$, clearly straying significantly from the values as for "real life" images. Nevertheless, let us set the "best-loss" values for the "real life" and "cartoon" datasets as $P_r$ and $P_c$ respectively, to be used in future reference when appropriate.

When looking at the parameters, we expect information (and location) to play a significant role, given the "decay" of the radial basis functions the further away we attempt to colourise on the image. To illustrate, consider Figure 3; giving no information on the yellow background leads to a colourisation of the closest value in greyscale (i.e. the light blue of the eyes) as shown in 3(c).

We also expect that as we increase the number of pixels initially supplied with colour information, we achieve lower $C$ values. We check for this behaviour this by using $P_r$



(a) Original 256 × 256 picture of an Among Us Crewmate.

(b) Among Us Crewmate (colourised) with 1 pixel initially coloured in each "block" of colour.

(c) Among Us Crewmate (colourised) with 1 pixel initially placed in only the light blue and both red "blocks" of colour.

Figure 3: Comparison of final results when colourising a cartoon image with 1 pixel in a given colour "block", performed with $P_c$.

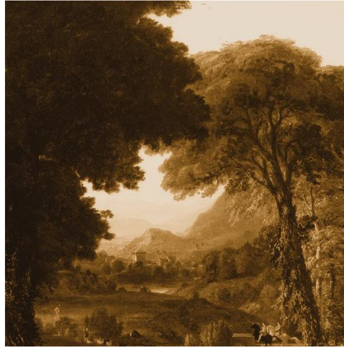
to test the 256 × 256 pixel image dataset, and record losses as a function of increasing $n$. The results are as expected, showing a clear decrease in $C$ as $n$ rises from 50 pixels to 200 pixels; this information is not surprising given that a greater $n$ leads to more points information being given from which to define the kernel, which then acts on the rest of the pixels in the image. We run the test five times at each given value of $n$ to calculate the costs associated with that $n$. A chart of the results for varying $n$ can be seen in Figure 10 in Appendix A.

We can also investigate the effect of the parameter $\delta$, as done in Figure 4. Note how $\delta$ visually acts to "smooth" the image; higher values of $\delta$ result in a more washed out colourised image.

Looking at $\sigma_1$ and $\sigma_2$, we expect these to relate to how "far" the colourisation method can reach other pixels. Figure 11 (shown in Appendix A) illustrates how $\sigma_1$ represents a parameter relating to the closeness of colour; when $\sigma_1$ is small (in Figure 11(a)) we lose information outside the points where the image has colour information, and so end up with dark "shadows". Similarly as we increase $\sigma_1$ we weight local information less and so we lose e.g. small blotches of colour that are locally confined; this is shown as we lose the blues of the sky as we increase $\sigma_1$ from 1 to 10 in Figure 11(b) to Figure 11(c).

Similarly, we expect $\sigma_2$ to play a role in differentiating greyscale information, and indeed with the image "dore-512.jpg" we get a particularly illustrative result, as shown in Figure 5. This example features only shades of grey and red, and by setting $\sigma_1$ low

(a) Original $512 \times 512$ landscape painting.



(b) Painting (colourised) with $\delta = 1$. Note the "washing out" of the yellow tint over the entire image, noticeable particularly on the mountains in the back.

(c) Painting (colourised) with $\delta = 1 \times 10^{-8}$. Note the distinct edges on colours visible on the mountains in the background.

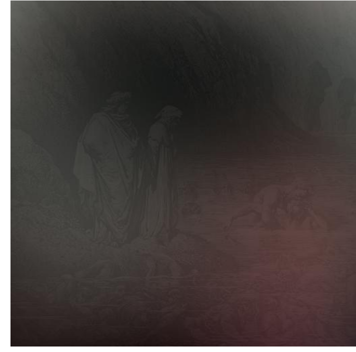
Figure 4: Comparison of effects of $\delta$ on a $512 \times 512$ image coloured with 500 randomly placed pixels at $P_r$, but with varied $\delta$.

we can increase the "resolution" at which we examine these colour differences; in other words a low $\sigma_1$ ensures we focus on information which is physically close to a given pixel. Then looking at $\sigma_2$, we see clearly how as $\sigma_2$ increases, the kernel incorporates similarities between further shades of grey. This washes out the image as we can see in Figure 5(c) at very large $\sigma_2$; note in particular how the dark shadows of the cave represent themselves as a dark-grey cloud in the upper half of the image, while the foreground takes on a different shade.

The parameter $\rho$ pertains to the greyscale information; its action is to pre-scale the difference in greyscale values by some power (reducing the difference if less than 1 or increasing if greater than 1. In essence, as $\rho$ increases from 0 to 1 we expect less emphasis on greyscale differences, acting in some measure as a smoothing parameter akin in greyscale space. The behaviour $\rho$ exhibits is as expected with our results which show a lower value of $\rho$ for "cartoon" images; the solid colour divisions naturally have different greyscale values which thus give more useful information than on a "real life" image. However, we note that the colourisation breaks down if we increase $\rho$ past 1; this is again as expected in some sense as it can be shown the kernel $K_2(x, y) := \exp\{-\|x - y\|_2^{2\rho} / \sigma^2\}$ is not positive definite for $\rho \geq 1$, as shown in e.g. [13]. This result extends to our case given our $K$ is a composite of $K_2$; colourisation in our model with $\rho \geq 2$ results in randomly coloured artefacts present in the image.

(a) Original artwork of cave.

(b) Artwork of cave (colourised) with $\sigma_2 = 101.3$.



(c) Artwork of cave (colourised) with $\sigma_2 = 1 \times 10^4$.

Figure 5: Comparison of effects of $\sigma_2$ on a $512 \times 512$ image coloured with 500 randomly placed pixels at $\delta = 2.6 \times 10^{-3}, \sigma_1 = 1, \rho = 0.9$, but $\sigma_2$ is varied. Note that $\sigma_1$ is set low to enhance colour discrimination from the reds of the lake with the otherwise greyscale image.

#### 5.2.1 Personal Extension

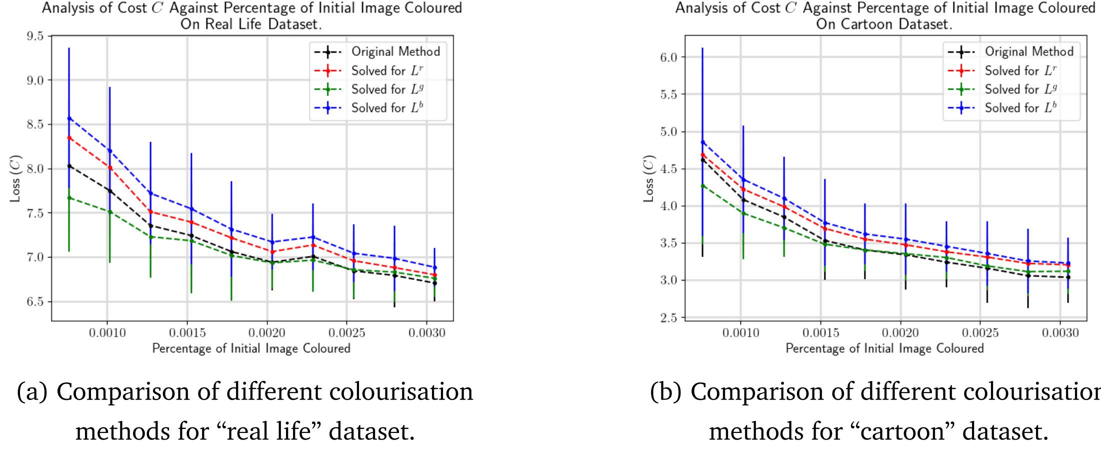
An alternative method to colourisation might be to utilise the greyscale information which we already know at any given point $z_i$, from (1). Recall that in our Code 1 we construct the final image one layer at a time. Instead, it is possible to use the information known from the relationship

$$\Gamma = 0.3L^r + 0.59L^g + 0.11L^b, \tag{11}$$

defining $\Gamma$ as the $M \times N$ matrix of greyscale values. We test whether this improves our results by fixing the parameters $\delta, \sigma_1, \sigma_2$, and $\rho$ for a given dataset[8] (using the entire $256 \times 256$ pixel image dataset), and comparing the $C$ values generated via the "normal" way (i.e. via generating each $L^c$) against utilising (11) for each of $L^c$ for $c \in \{r, g, b\}$. We run ten different tests for varying numbers $n$ of randomly coloured initial pixels given, using an unseeded generator for the random pixels coloured. For each $n$, we calculate the mean of the losses over each image five times across the dataset for the images of the specified type, and present the results as well as the standard deviations for each data point. We set the parameters to somewhat arbitrary values to simulate not knowing *a-priori* what the best parameters would be for a given image; within this extension we would like to explore whether our knowledge of $\Gamma$ helps suppress $C$ values regardless of any other information we may have on the image besides $n$. The results

[8] In other words, we set these parameters to the values that led to "best-loss" results in Section 5.2.

are given in Figure 6.




(a) Comparison of different colourisation methods for "real life" dataset.

(b) Comparison of different colourisation methods for "cartoon" dataset.

Figure 6: Comparison of averaged $C$ values for different $256 \times 256$ pixel images recolourised via different methods. All images were run with $\delta = 1 \times 10^{-4}$, $\sigma_1 = \sigma_2 = 100$, and $\rho = 0.5$. $n$ was varied between 50 and 200 over 10 sample points, and each run at a given $n$ was repeated 5 times.

Immediately note the decrease of loss $C$ for both datasets as $n$ increases, showing the behaviour we expect. As a broad trend, we note that solving for $L^g$ from (11) appears to give better results than either the original method or for solving for any of the other two colours, while solving for $L^b$ gives worse results. Further note that at these differences are more pronounced at lower values of $n$. This behaviour is somewhat as expected: our method for calculating each image layer is bound to produce errors in the actual colourisation of any given layer, and therefore using information which *know* to be correct (via our knowledge of $\Gamma$), we can attempt to minimise losses by inferring the relationship between the RGB and greyscale values, as opposed to introducing *further* errors via trying to construct another $L^c$ layer.

When performed on the green channel, which has the highest weighting in the monochrome luminance signal, we note that by only calculating the red and blue channels via $L^c$, we construct a much smaller "proportion" of a pixel's RGB value (assuming $0.3R + 0.59G + 0.11B = 1$), leading to a greater "proportion" being inferred via (11). Similarly, when constructing $L^b$, the opposite argument holds as we have a greater proportion of the RBG value based off of "bad" data. In the original method the blue channel would introduce errors solely caused by its own construction, but by solving for $L^b$ we compound the errors introduced via $L^g$ and $L^r$ by assuming some fixed relationship between these three values.

We note the large errors on our chart. This again, is expected, as we test over the entire dataset with the same parameters, and also due to the random nature of pixel placement. To try and examine this behaviour in closer detail, we sample a single image for 20 values of $n$ from 50 to 200, and rerun this test twenty times for a given value of $n$, and present the results in Figure 7.

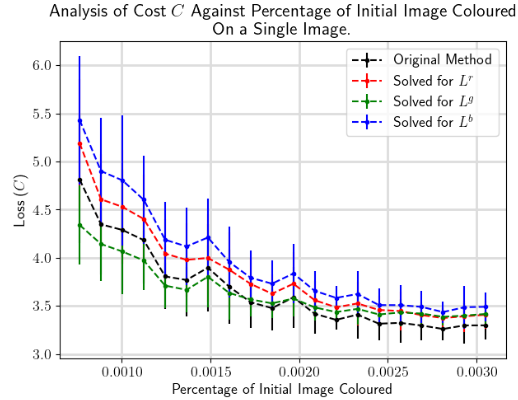
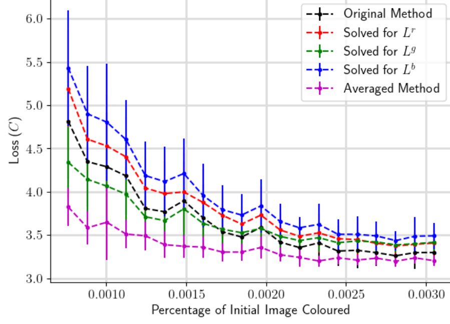



Figure 7: Analysis of cost against percentage of initial image coloured, on “knights-256”. Run with $\delta = 1 \times 10^{-4}, \sigma_1 = \sigma_2 = 100$, and $\rho = 0.5$. $n$ was varied between 50 and 200 over 20 sample points, and each run at a given $n$ repeated 20 times.

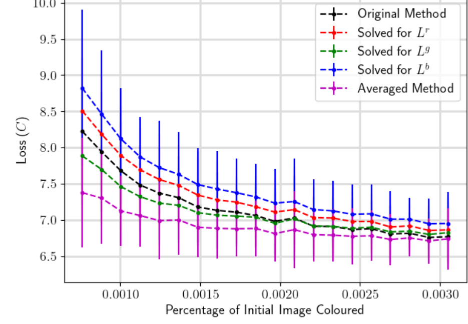
In order to extend this idea, let us introduce some notation. Let $L_1^c$ be a colour channel computed via the original method, that is via solving for $Ta^c$. Subsequently, let $L_2^c$ be a layer solved for via (11). We can thus try computing the *averaged* method, in which we define $L_3^c := \frac{1}{2}(L_1^c + L_2^c)$. While not requiring much more computational power due to simply needing three sets of matrix operations after each $L_1^c$ has been generated (each $L_i^c$ being the size of the raw image and thus not unreasonably large), this method proves (as shown in Figure 8) to provide good results.

In both of these tests ran, the “averaged” method seems to produce better results than the other methods discussed, which is particularly clear in Figure 8(a) which is ran on a single image. This method seems particularly attractive given the relatively significant decreases in cost at the expense of a comparatively insignificant amount of computational power, though further examination might also present us with different weightings to apply to the parameters $L_1^c$ and $L_2^c$, rather than the equal weights which are currently applied. However, note that if computational cost is at a premium then perhaps solving for $L^g$ might be a better suggestion than the “averaged” method.

We can attempt to quantify the upper bound of the error for this method by first defining the error in a colour channel for a method by $e_i^c := \left|L_i^c - L_0^c\right|$, and the scalar




(a) Analysis of cost on a single image from the "real life" dataset using the "averaged" method.




(b) Analysis of average cost on all images from the "real life" dataset using the "averaged" method.

Figure 8: Comparison of "averaged" method to other colourisation methods, sampled for twenty values of $n$ between 50 and 200. Parameters were set to $\delta = 1 \times 10^{-4}, \sigma_1 = \sigma_2 = 100$, and $\rho = 0.5$.

factor associated with the channel in the monochrome luminance signal as $\alpha_c$. Then, for example in the red channel,

$$e_3^r := \alpha_r \left| \frac{L_1^c + L_2^c}{2} - L_0^c \right| \tag{12}$$

$$\leq \frac{\alpha_r}{2} \left| L_1^r - L_0^r \right| + \frac{\alpha_r}{2} \left| L_2^r - L_0^r \right|. \tag{13}$$

We can extend this calculation to each of the channels and sum to get

$$e_3 \leq \frac{e_1}{2} + \sum_{c=r,g,b} \frac{\alpha_c}{2} \left| L_2^c - L_0^c \right|. \tag{14}$$

To quantify the sum in (14) we expand and therefore have

$$\sum_{c=r,g,b} \frac{\alpha_c}{2} \left| L_2^c - L_0^c \right| = \frac{\alpha_r}{2} \left| L_2^r - L_0^r \right| + \frac{\alpha_g}{2} \left| L_2^g - L_0^g \right| + \frac{\alpha_b}{2} \left| L_2^b - L_0^b \right| \tag{15}$$

$$= \frac{1}{2} \left( \left| \Gamma - \alpha_g L_1^g - \alpha_b L_1^b - \alpha_r L_0^r \right| + \ldots \right) \tag{16}$$

$$= \frac{1}{2} \left( \left| \alpha_g \left( L_0^g - L_1^g \right) + \alpha_b \left( L_0^b - L_1^b \right) \right| + \ldots \right) \tag{17}$$

$$\leq \frac{1}{2} \left( (e_1^g + e_1^b) + (e_1^r + e_1^b) + (e_1^r + e_1^g) \right) = e_1, \tag{18}$$

where we have thus placed an upper bound of $\frac{3}{2} e_1$ on the "averaged" method. Nevertheless, it is possible to perform *better* than $e_1$ in the case that the sum in (14) is less than $e_1$ itself; and potential further exploration of the behaviour of this method might be illuminating.

# 6 Conclusions

Using kernel methods to recolourise images has been shown to be a viable approach, up to computational constraints in handling large matrices. We have explored several design decisions in the creation of a Python recolourisation program to ensure rapid processing of computationally intensive calculations. Furthermore, we have examined the effect parameter variation has in recolourisation of images. We have shown that the number of pixels initially coloured in and then passed to the colouriser is the strongest indicator of final colourisation performance, and also explored the role that various parameters play within the radial basis function kernels which we have defined within the kernel that we utilise to measure similarity of pixels.

We have further shown that recolourisation results generally vary both with image type (between "cartoon" and "real life" images), and even between images of the same type, with optimal values of parameters lying within a sizeable range relative to the magnitude of the parameters themselves when applied to a dataset. In future work the exploration of automatic optimisation of these parameters might be in order.

Lastly, we have explored the effect of utilising greyscale information when recolouring, and have demonstrated it to generally improve results. Further we posit an alternative method to incorporating this greyscale information which we dub the "averaged" method, which we have also shown to improve colouriser performance even further. Exploration of the mechanism behind why such behaviour occurs in a more rigorous context might be of use for further applications.

# 7 References


[1] *15. Floating Point Arithmetic: Issues and Limitations*. Python documentation. URL: https://docs.python.org/3/tutorial/floatingpoint.html (visited on 03/04/2023).

[2] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. “Algorithms for hyper-parameter optimization”. In: *Proceedings of the 24th International Conference on Neural Information Processing Systems*. NIPS’11. Red Hook, NY, USA: Curran Associates Inc., Dec. 12, 2011, pp. 2546–2554. ISBN: 978-1-61839-599-3. (Visited on 02/25/2023).

[3] *Double Precision Floating Point - IBM Documentation*. Jan. 20, 2023. URL: https://ibm.com/docs/en/aix/7.2?topic=types-double-precision-floating-point (visited on 02/22/2023).

[4] *Git*. URL: https://git-scm.com/ (visited on 02/22/2023).

[5] Minh Ha Quang, Sung Ha Kang, and Triet M. Le. “Image and Video Colorization Using Vector-Valued Reproducing Kernel Hilbert Spaces”. In: *Journal of Mathematical Imaging and Vision* 37.1 (May 1, 2010), pp. 49–65. ISSN: 1573-7683. DOI: 10.1007/s10851-010-0192-8. URL: https://doi.org/10.1007/s10851-010-0192-8 (visited on 02/20/2023).

[6] *Matplotlib — Visualization with Python*. URL: https://matplotlib.org/ (visited on 02/21/2023).

[7] *Numba: A High Performance Python Compiler*. URL: https://numba.pydata.org/ (visited on 02/25/2023).

[8] *NumExpr: Fast numerical expression evaluator for NumPy*. original-date: 2013-11-30T22:33:48Z. Feb. 24, 2023. URL: https://github.com/pydata/numexpr (visited on 02/25/2023).

[9] *PNG Specification: Data Representation*. URL: https://www.w3.org/TR/PNG-DataRep.html (visited on 02/19/2023).

[10] *Pygame*. URL: https://www.pygame.org/news (visited on 02/21/2023).

[11] *Riverbank Computing | Introduction*. URL: https://riverbankcomputing.com/software/pyqt/ (visited on 02/21/2023).

[12] Tom Schimansky. *TomSchimansky/CustomTkinter*. original-date: 2021-03-04T17:24:00Z. Feb. 22, 2023. URL: https://github.com/TomSchimansky/CustomTkinter (visited on 02/22/2023).

[13] I. J. Schoenberg. "Metric spaces and positive definite functions". In: *Transactions of the American Mathematical Society* 44.3 (1938), pp. 522–536. ISSN: 0002-9947, 1088-6850. DOI: 10.1090/S0002-9947-1938-1501980-0. URL: https://www.ams.org/tran/1938-044-03/S0002-9947-1938-1501980-0/ (visited on 02/21/2023).

[14] Bernhard Schölkopf, Jean-Phillipe Vert, and Koji Tsuda. *Kernel Methods in Computational Biology*. July 16, 2004. DOI: 10.7551/mitpress/4057.001.0001. URL: https://direct.mit.edu/books/book/3898/Kernel-Methods-in-Computational-Biology (visited on 02/20/2023).

[15] *The Web Application Security Consortium / Integer Overflows*. URL: http://projects.webappsec.org/w/page/13246946/Integer%20Overflows (visited on 02/19/2023).

[16] Valentin Thorey. *benderopt*. original-date: 2017-03-13T15:05:14Z. Dec. 23, 2022. URL: https://github.com/vthorey/benderopt (visited on 02/25/2023).

[17] *tkinter — Python interface to Tcl/Tk*. Python documentation. URL: https://docs.python.org/3/library/tkinter.html (visited on 02/21/2023).

[18] *Usage Statistics of Image File Formats for Websites, February 2023*. URL: https://w3techs.com/technologies/overview/image_format (visited on 02/19/2023).

[19] John Watkinson. *The MPEG handbook: MPEG-1, MPEG-2, MPEG-4, second edition*. 2nd ed. Amsterdam, Boston: Elsevier/Focal Press, 2004. ISBN: 978-0-240-80578-8.

[20] *What is NumPy? — NumPy v1.24 Manual*. URL: https://numpy.org/doc/stable/user/whatisnumpy.html (visited on 02/22/2023).

[21] *What is Object-Oriented Programming (OOP)? - Definition from Techopedia*. Techopedia.com. URL: http://www.techopedia.com/definition/3235/object-oriented-programming-oop (visited on 02/22/2023).

[22] *What is Procedural Programming? - Definition from Techopedia*. Techopedia.com. URL: http://www.techopedia.com/definition/21481/procedural-programming (visited on 02/22/2023).

# Appendices

## A Further Results

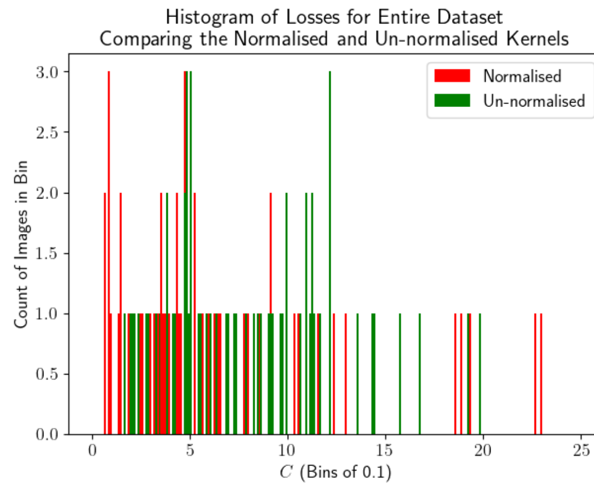



Figure 9: Histogram of losses for the normalised and un-normalised methods, with bins of 0.1 for 60 total images of varying size with 0.1% of pixels coloured.

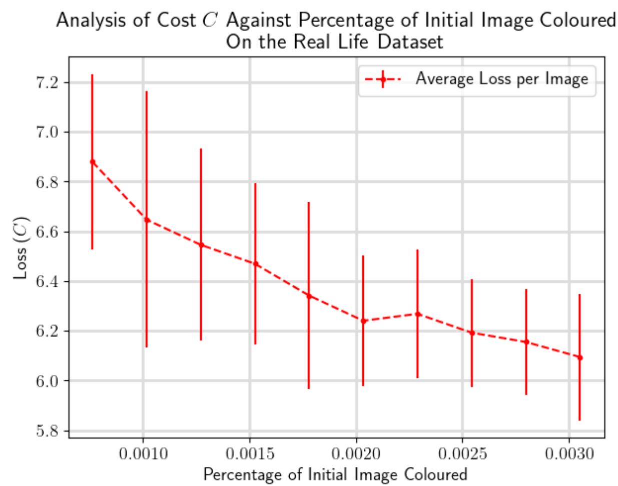
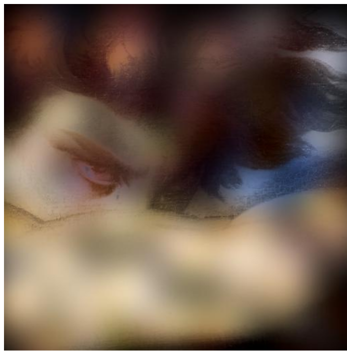
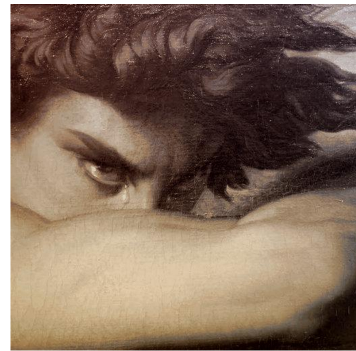



Figure 10: Examination of average $C$ varying with $n$ as a percentage of the total number of pixels for the "real life" dataset, tested with parameters $P_r$, for 10 values of $n$.



(a) Artwork (colourised) with $\sigma_1 = 0.1$.

(b) Artwork (colourised) with $\sigma_1 = 1$.



(c) Artwork (colourised) with $\sigma_1 = 10$.

Figure 11: Comparison of effects of $\sigma_1$ on a $512 \times 512$ image coloured with 500 randomly placed pixels at $P_r$, but with varied $\sigma_1$.

# B Code

**Code 2:** Outline of main program

```
class GUI:
    method constructGUI:
        Define button1;
        ...;
    end
    method loadImage:
        initialImage = code to load image here;
    end
    method generateImageWithSomeColour:
        imageWithSomeColour = code to add some colour to image;
    end
    method colouriseImage:
        colouriserClassInstance = Colouriser(imageWithSomeColour);
        imageThatIsColourised = colouriserClassInstance.colouriseImage();
    end
    ...;
class Colouriser:
    method colouriseImage:
        Construct K_D;
        Construct T;
        for c = {r, g, b} do
            Compute a^c = (K_D + δnI)^{-1} f̄^c;
            Compute Ta^c;
            Reshape Ta^c to an N × M matrix L^c;
            finalImage^c = L^c;
        end
        return finalImage;
    end
    ...;
```