Final Report
Image Forgery Detection

CM3203 Large One Term Individual Project
40 Credits

By Joshua Head
Supervised by Dr Yu-Kun Lai

May 5, 2015

Abstract

This report examines the results of three image forgery detection algorithms, Error Level Analysis, Copy-Paste Cloning Detection and Fourier based Resampling Detection. Each algorithm was implemented within MATLAB and run on a sample library of forged and unmodified images, including a selection of images from an image manipulation dataset. Each method was found to have its own set of advantages and limitations, however with all three methods combined the overall detection rate was an impressive 80%. Error Level Analysis provided decent results on previously compressed, high-quality JPEG files, but struggled with newly compressed images or low quality samples. Copy-Paste Cloning Detection was highly successful on images forged using cloning methods, however the overall runtime was much higher than the other methods, and due to the nature of the algorithm false positives were routinely detected. Image Resampling Detection operated on a wide variety of images, provided good overall results on each dataset, and the rate of false positives was low. The algorithm was also highly efficient, however resampling must have occurred in order for any forgery to be detected, and it was therefore unsuitable for direct copy-paste forgeries. Metadata Tag Detection was also run on each image, however it was found to be too rudimentary to be considered a method in its own right, as tags within files can be cleared or removed without much effort. This project therefore provides an ideal base for a user to determine the most applicable image forgery detection method for their use, depending on the types of images that they routinely deal with.

Acknowledgements

I would like to express my very great appreciation towards my research supervisor, Dr. Yu-Kun Lai, for his professional guidance and valued support throughout this project.

I would also like to thank my parents for their support and encouragement throughout my academic studies.

Contents

List of Figures

1 Introduction

Since the invention of photography, individuals and organisations have often sought ways to manipulate and modify images in order to deceive the viewer. Whilst originally a fairly difficult task requiring many hours of work by a professional technician, with the advent of digital photography it is now possible and fairly trivial for anyone to easily modify images, and even easier to achieve professional looking results. This has resulted in wide reaching social issues, ranging from the reliability of the images reported by the media to the doctoring of photographs of models in order to improve their looks or body image. With the sheer amount of methods available in which to manipulate an image, image forgery detection has become a growing area of research in both academia and the professional world alike.

Many methods exist in order to detect forgery within digital images, however it is difficult to find which are the most efficient and practical to implement and run. Whilst one algorithm may have a good detection rate, it could also have a large rate of false positives. In addition, runtime is a major factor that contributes to the efficiency and overall usability of an algorithm, but tends to only be mentioned academically as opposed to in real world terms.

The aim of this project is to research and investigate in to the many methods surrounding image forgery detection. In order to reduce the complexity of this task, as set out in the initial report, algorithms will be grouped in to five distinct algorithm types. These are JPEG Compression Quantization, Edge Detection, Clone Detection, Resampling Detection and Light & Colour Anomaly Detection. More specific research will then be concluded on these different groups, determining the efficiency of the described algorithm type in general. If the method is found to be reliable, then an algorithm from within this group is implemented. These groups have been chosen as their detection methods are entirely different from each other, and therefore should achieve very different results depending on the image forgery type.

Extensive testing using a library of images will then be performed on the implemented algorithms in order to determine their success rate. Other general properties of the algorithm, such as its false positive rate and runtime will also be reported. Additionally, more specific tests on variants of the same algorithm will also be performed. For example, an algorithm may have parameters that can dramatically alter its performance and detection rate on certain classes of images, and so by testing these values we're able to comprehensively determine an algorithms performance on a variety of different image types. This ensures that more advanced algorithms are not unfairly discarded simply because their internal parameters needed tweaking.

The results of this research will be of great use in order to improve the credibility of images used within the media. Image forgery is an ever increasing issue in modern society, and there have been instances where forged images have been used by mistake, or when images have specifically doctored in order to be misleading. Despite the importance of the issue, there is still no widely recognised method in order to detect image forgeries, and certainly no industry

standard. This represents an opportunity to provide an insight that will benefit one of the largest industries in the world, and potentially improve the reliability and credibility of the images presented by the media. This also allows individuals the opportunity to determine the credibility of the images provided to them, either through official, credible sources or elsewhere, such as on an internet message board or shared by a friend on social media.

Whilst the results of the project aim to be both comprehensive and clear, its important to note the scope of the research involved. Research will be undertaken on algorithms and methods that have been discussed in existing academic papers. Improvements and changes can and will be made to these methods, however no new algorithms will be created or tested as part of this project. The aim is to implement pre-existing algorithms, improving and comparing them, not to conduct research on generating potentially new algorithms. This would require much more time than is feasible, impacting the testing stage, and ultimately reducing the effectiveness of the research. Determining new algorithms are therefore beyond the scope of this project.

Whilst conclusions will be able to be made at the end of the project, the outcome will be rather open ended as the effectiveness of an algorithm will entirely depend on the type of image that you're attempting to detect forgery within. However, using sample libraries we will be able to confidently assess which algorithm type has the overall advantage; that is it is able to detect the most amount of forgeries within a sample library, has the lowest false positive rate and runs in a reasonable amount of time. We will also be able to determine the ideal conditions of the algorithm, using additional parameters and settings, in order to understand the specifics of the method in question. This can also include running the algorithm on the same image but at different compression levels or in saved as different file formats.

However, whilst we can come up with conclusions based upon the results of the project, it's important to note that there will never be a perfect algorithm for every situation. The best algorithm for an individuals needs will vary depending on the type of images that they are routinely dealing with. This very much leads the results open to interpretation. In addition, some images may require multiple tests run on them in order to detect all forgeries, and it is inevitable that some image forgeries will evade every detection algorithm. Therefore, whilst the results of the project will show the competency and efficiency of different algorithms on a variety of forgeries, the best algorithm will be subjective and will vary between each user. This project aims to provide evidence and research results that allow the user to achieve their own conclusion.

2 Background

Image forgery has been an issue since the advent of traditional photography in the 19th century, however it is a much more prevalent problem in the digital age. The primary issue is that photographs are often used as concrete evidence of an event, and are generally seen by the public as truthful and trustworthy. Images that are forged, therefore abusing this trust, can have many wide-reaching social impacts. For example, CCTV images are often used in a court of law in order to provide solid evidence either by the defence or by the prosecution. If the confidence in these images are put in to doubt and the jury is unable to put their utmost trust in them, then the trial is put in to repute. Detecting manipulation and forgery within these images is therefore of the utmost importance.

Similarly, forged images are extensively used within the media, either deliberately or accidentally. Tabloid newspapers, magazines and marketing campaigns routinely modify images of models or famous figures in order to make them look more aesthetically pleasing to the viewer. This can be a simple case of adding a filter or modifying the contrast of the image, but it is often much more extreme; improved muscle definition, more toned body parts and wrinkle removal are examples of commonly achieved results. The issue has become so prevalent and well known that the verb "photoshopped", referring to the popular image editing application Adobe Photoshop, has become a neologism for manipulating and modifying digital images.

One of the most pressing issues is that there are many different ways of modifying an image, and due to a digital images' complex nature it's impossible to have an algorithm that detects every type of image forgery. Because of this, image forgery detection isn't widely used in the professional world. The underlying concept would be highly useful in the majority of professional fields that deal with images on a day to day basis, where the reliability and credibility of these images is crucial. In addition, with the large increase in the use of social media, individuals would also benefit greatly from being able to detect forgeries within images. Convincingly manipulated images are widely circulated on social media platforms [17], and are able to be spread rapidly within communities who believe them to be true. In order to detect these image forgeries, it is required that we understand some typical methods used in order to manipulate images. These include:

- Copy-paste Cloning - This is where existing areas within an image are cloned, allowing regions to be covered or objects to be duplicated. This is a commonly used method as the forgeries have the potential to look very convincing, due to the fact that they have come from the source image to begin with.

- Image Splicing - Whereby objects from another image are spliced together with the source image, adding objects that weren't present in the original image. Various blending techniques exist, such blurring edges, reducing the contrast and utilising cloning to help disguise the new object in with the surrounding area.

- Modification of existing regions - This is similar to copy-paste cloning, but instead of being an exact duplication, existing regions are modified in order to suit the needs of the forgery. This can include simply resizing the object, mirroring or skewing it, or splicing two existing objects together. In all of these cases however, the duplicated region has been resampled, meaning that it has been modified enough not to be recognised by any clone detection algorithm.

Whereas existing projects have worked on the comparison of image forgery detection methods, these are often limited in scope and only compare variants of the same algorithm on images that are specifically created for that type of method. For example, JPEG Analysis and Edge Detection have been compared [1], however no reason is given as to why these specific implementations were chosen over others, as both tend to detect similar kind of forgeries. In addition, no detail of the images that were used in the research is provided; for example it is unknown if they are standard library images or images tailored for this kind of forgery detection algorithm.

In addition, pre-existing image forgery detection applications are often of an academic nature (proof of concept or of prototype quality), or very simple. Searching for forgery detection mainly brings up academic papers on the subject, however the most downloaded results on the popular open-source site SourceForge return fairly trivial applications that only detect metadata tags embedded within images [2][3][4]. Whilst this is a useful measure, and something which will also be tested in addition to the main algorithms within our implementation, metadata tags are easily removed or manipulated and it is therefore not an accurate measure of whether an image has been forged or not. Although the implementation within this project is mainly a proof of concept and used purely for research purposes, it is a starting point that could be developed in to a fully fledged application. Creating a polished, user friendly interface for the chosen algorithm is then fairly trivial, bringing that type of forgery detection to the mass market.

As mentioned, one of the largest issues surrounding image forgery detection is the sheer number of options; many different types of algorithms that detect forgeries in very different ways, and each with hundreds of variants based upon that specific implementation. This project aims to simplify the process of detection, condensing the types of algorithms into distinctive groups [7], evaluating the general effectiveness of each method, and if appropriate implement and test an algorithm from that group. Due to the ever changing methods used in image forgery, no algorithm is perfect, and detecting every type of forgery with one algorithm is an impossible task. However this research will allow the user to not only conclude which algorithm is best for them, but also provide an insight in to the most efficient algorithm overall; that is the one that has the greatest detection rate when run on the group of sample images.

The purpose of researching into distinct algorithm groups is to ensure that different types of forgeries are able to be discovered. Whilst in depth research will be concluded on each type of algorithm, the main purpose is to discover

which algorithms detect different forgery types. Research into very similar types of algorithms is largely unhelpful as it doesn't provide this useful insight in to different types of forgeries. The algorithm groups researched operate in very different ways; JPEG Compression Quantization detection exploits differences within the 8 x 8 macroblocks used within JPEG compression [6], whereas clone detection finds similarities within sections of the image to detect forgeries. Edge Detection searches for large differences in the frequency of regions of interest [1], and resampling detection operates on the differences in resampling artefacts that occur when sections of the image are modified [9]. These different algorithm types operate in entirely contrasting ways, and each have strengths in detecting particular image forgeries. For example, it's fair to assume that a clone detection algorithm is not going to detect a forgery that has been created by splicing two images together, but will be efficient in detecting areas that have been duplicated and moved around. Each algorithm has its advantaged and disadvantages, and this project aims to investigate these.

The implementation will be created within MATLAB, due to its efficiency in dealing with images and its support for more complex mathematical functions. Version R2014a on Windows will be used for development and testing, however as no new version specific features are being utilised it should be backwards compatible with previous versions of MATLAB.

Aim: The aim of this project is to investigate different types of image forgery detection algorithms that currently exist, allowing the efficiency and competency of each to be evaluated on a variety of sample images. This will ultimately allow the user to decide which algorithm is best suited for the type of images that they routinely deal with.

Research question(s): In order to demonstrate the achievement of the stated aim, this project will identify existing methods for image forgery detection, research their overall effectiveness and if suitable implement the most appropriate algorithm from that group. Each algorithm will then be run on a series of sample images, resulting in both success and false positive rates, and have its runtime performance measured. In addition, each algorithm will be extensively tested based on adjusting its own unique parameters or variables in order to find the ideal forgery detection type for that algorithm.

3 Specification & Design

The design of the system is heavily based around the groups of algorithms described. Research in to each of these distinct groups has produced the following system specification:

- JPEG Compression Quantization
  This method was found to be effective in detecting a variety of image forgeries, as it relies on exploiting the quantization process of JPEG compression and not on detecting any particular forgery method imposed by the user. JPEG is also the most widely used image format on the internet for photographs and true colour images [16], due to its large, lossy compression ratios and generally high image quality. In order for a forgery to be detected by the algorithm, it is required that the original image was previously compressed. Uncompressed, forged images which are then compressed for the first time will not produce any results as both the forged area and the original area have only been compressed once.

- Edge Detection using Standard Deviation
  This type of algorithm, utilising Standard Deviation or any other technique (such as a Sobel Edge Detector) can be used in order to detect splicing forgeries within images. Legitimate images will tend to have softly blurred edges due to camera lens imperfections (such as Chromatic Aberration [15]). Cloned regions will retain these features, whereas spliced areas won't exhibit the same behaviour, and will tend to have harsher edges that are much more visible when highlighted using an edge detection technique.

  However, it is possible to utilise blurring or edge filters in order to convincingly blend forged areas in with the original image. Whilst edge detection could potentially aid discovery in very particular cases, these generally rely on having harsh, unrefined edges visible within the forgery areas. In these instances, the forgeries are usually visible to the naked eye, as more advanced and believable methods will tend to blend the forgeries in with the surrounding, original image. In addition, many natural images contain frequent and sharp edges, for example images of architecture, and this method isn't suitable for use in such images. As such, it was found that this method wasn't a particularly effective solution for detecting image forgeries overall, as it is only applicable in cases consisting of rough, poorly blended splices.

- Clone Detection
  Many genuine images contain repeating patterns, such as trees and walls. Whilst these areas appear to be indistinguishable to the human eye, they often contain minute differences and are simply very similar as opposed to being fundamentally identical. Copy-paste cloning forgeries can be very difficult to spot, in many cases appearing to look the same as legitimate

areas. Cloning is therefore a popular forgery method as duplicating existing areas of an image tends to produce much more believable results than splicing an image from elsewhere. Copy-paste cloning detection algorithms operate by scanning the image for regions of matching, identical pixels. By its nature, this type of algorithm will only detect cloning forgeries, it is not a general use algorithm and will therefore not detect other types of forgeries. In addition, re-compression or resampling can slightly modify clusters of pixels, enough to reduce the likelihood of the algorithm detecting duplicates. However, clone detection algorithms are still very useful, as it is generally difficult to otherwise differentiate between similarly looking patterns and forged, duplicated regions.

- Image Resampling Detection
  Whenever a digital image is extensively modified, it is resampled. A genuine image will tend to have any resampling artefacts spread consistently throughout, that is each region will look similar when broken down to its underlying frequencies. Whether a forgery is created by splicing two images together or modifying existing sections of an image, more advanced forgeries will tend to have to resize, rotate or modify that forged area in some form. This leads to the resampling of that newly forged area, which now contains differing artefacts to the rest of the image. Breaking down specific regions of interest and computing their underlying frequencies allows us to potentially detect both image modification and image splicing forgeries. Detecting resampled areas was found to have a good success rate on a variety of different image types, although the process worked the best on images with no or little compression. Reducing the quality of an image too much caused additional compression artefacts to appear, limiting the detection rates of the algorithm.

- Colour and Light Anomalies
  A more emerging forgery detection method that was looked in to was detecting forgeries through changes in light and colour throughout the image. Forged areas are often composed of different images, which have inconsistent lighting qualities compared to the source image. The ability to detect this would allow forgery detection within a seemingly large variety of images. Unfortunately, research and initial testing found this method to be inconsistent, many legitimate images contain large variants in colour, and calculating the lighting of an image is rather computationally intense [19] due to the number of calculations required. In addition, forgeries that mismatch the surrounding area enough for an the algorithm to detect it could be picked out fairly easily by the naked eye.

  Out of the five algorithm groups, three proved reliable and consistent enough in order to be included as part of the implementation. These are Clone Detection algorithms, Image Resamping Detection and JPEG Compression Quantization. In addition, Metadata Tag Detection is to be included as part of the implementation. Image metadata contains basic

information regarding the image, such as the resolution, bit-depth and size. When an image is processed within image editing software and re-saved, the software will generally modify the image's metadata by creating additional tags, including the software used to modify it and the modified date and time. Whilst these additional tags by no means guarantee forgery; it is possible that the image was accidentally re-saved without any changes for example, it's a useful indicator when combined with other forgery detection methods.

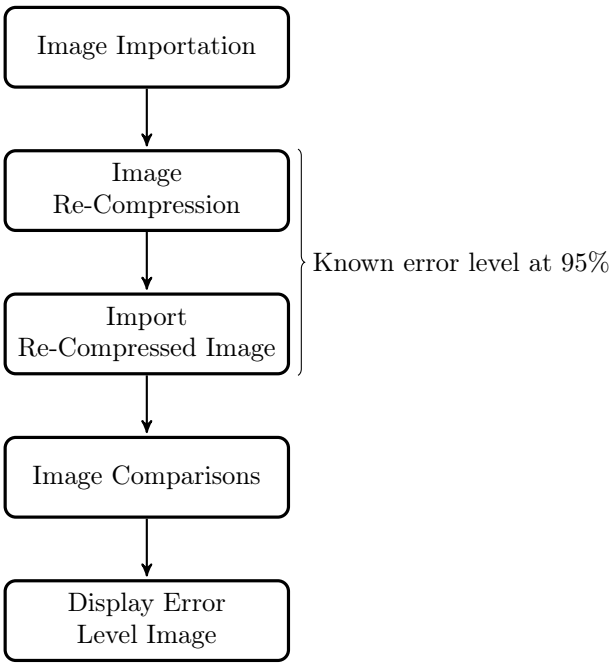
3.1 JPEG Compression Quantization

One technique that can be used for detecting JPEG Compression artefacts is Error Level Analysis. This operates by re-compressing an already compressed JPEG image at a known error level, and comparing the two, pixel by pixel. Legitimate images generally show an equal error level throughout the entire image, creating one shade without much variance. As forged areas have been re-compressed at a different level, their quantization levels are unlike the rest of the image; creating an area that has a much larger margin of error in comparison to any legitimate sections.

Whilst quantization differences can also be shown by scanning an 8 x 8 window across the entire image, Error Level analysis only requires a pixel by pixel comparison, dramatically reducing the runtime complexity of the algorithm. In a real world scenario, a 500 x 500 pixel image only requires 500 direct comparisons; whereby sliding an 8 x 8 window along the image requires a minimum of 400^2 comparisons.

As MATLAB handles images as matrices, direct dot comparisons of two matrices (that is, comparing each element in matrix A with the equivalent element in matrix B) is quick and computationally insignificant, compared to looping through each overlapping 8 x 8 block.

The general architecture of this algorithm is shown below:



Once an image is imported, it is re-saved using MATLAB's built in JPEG encoder, at a known compression level of 95%. This level was chosen as it creates a large enough margin of error in order to generate a perceivable difference, without reducing the overall quality of the image to such an extent that the margin of error is far too high. The re-compressed image is then imported, and dot comparisons are carried out between the original image matrix and the newly imported one, creating a error level image. From this image we are able to not only visibly detect areas that have potentially been re-compressed, highlighting forgeries, but also generate an overall error margin for the entire image. This error level margin is calculated as follows:

$$E_A = \frac{(\frac{\sum_{i=1}^{n-1} A_i}{n})}{255} \times 100$$

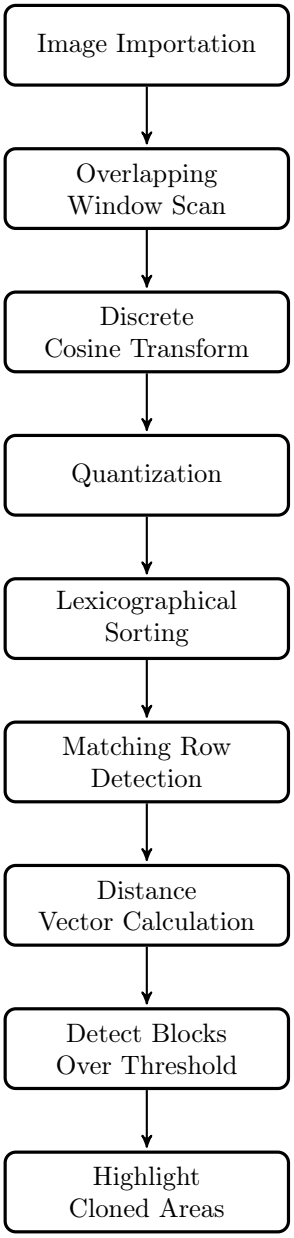
where A is our error level matrix and n is the total number of elements within the matrix.

The higher the error margin, the larger the overall difference between the two images. However, it's important to note that this value cannot be used to fundamentally determine the forgery level of an image; newly compressed images will tend to have a much larger error level overall, compared to an image that has been re-compressed many times. Instead, this value should be used in conjunction with any visual indicators within the image of potentially forged areas.

3.2 Clone Detection

Clone detection, by its nature, is a computationally expensive algorithm. Whilst in principle it's possible to match each pixel with another to create a match, not only would this become far too complex on any standard image (the same 500 x 500px image discussed above would require 500! comparisons, roughly $1.22\text{x}10^{1134}$), but it would also produce far too many matches to produce a meaningful result. A grayscale image contains pixel values between 0 and 255, meaning that in our example image, due to the pigeonhole principle almost half of our pixels are guaranteed to be duplicates of already used pixel values [5].

A compromise to this pixel by pixel method is to use an overlapping window instead, however caution must be used in order not to utilise a window too large that potential forgeries are missed. JPEG compression, as mentioned above, operates by utilising a quantization matrix on each 8 x 8 block within the image. Because of this, using an 8 x 8 window would run the risk of being affected by the quantization process, and potential duplications could be recorded as being different purely because of the quantization used. A 16 x 16 window is therefore optimal, as being double the size of the traditional quantization block allows JPEG compression errors to be included within the comparison. Whilst this could potentially allow small enough forgeries to remain undetected, any notable modifications must be larger than that size in order to be perceived by the untrained eye. Whilst still computationally expensive, a 16 x 16 window helps cut down the number of comparisons required to 235,226 for our example image. The basic architecture of the algorithm can be shown:



Once each 16 x 16 window is read, a discrete cosine transform is performed on each block, and then quantized using an expanded JPEG quantization matrix. This expanded matrix is calculated utilising the following formula [8]:

$$Q_{16} = \begin{pmatrix} Q'_8 & 2.5q_{18}I \\ 2.6q_{81}I & 2.5q_{88}I \end{pmatrix}, where\ Q'8 = \begin{pmatrix} 2q_{00} & 2.5q_{12} & \cdots & 2.5q_{18} \\ 2.5q_{21} & 2.5q_{22} & \cdots & 2.5q_{28} \\ \cdots & \cdots & \cdots & \cdots \\ 2.5q_{81} & 2.5q_{82} & \cdots & 2.5q_{88} \end{pmatrix}$$

where I is an 8 x 8 unit matrix, comprising of all 1s, and Q is the quality factor determining the amount of quantization that occurs. At high values ($>$ 1), more and more matrices will result in a match, both increasing the likelihood of detecting false positives and also raising the runtime of the algorithm.

The resulting matrices are converted into single lined vectors, sorted lexicographically, and then each matching row is recorded. However, counting the number of block matches would still record a large number of false positives. Many images contain repeating information that is in no way duplicated, for example a brick wall is bound to contain a few 16 x 16 block matches. A simple solution to this problem is to calculate the distance vector of the two matching blocks, as computed by:

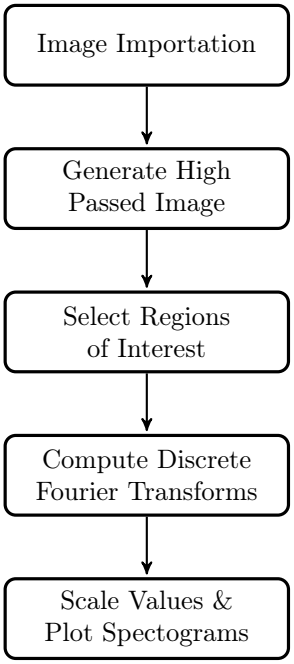
$$|(x2 - x1), (y2 - y1)|$$

whereby (x1, y1) and (x2, y2) are the coordinates of the first and second blocks, respectively.

Computing the distance vector and counting the occurrence of each allows us to determine matching regions, as opposed to simply matching blocks. A large region consisting of different, but consecutively matching blocks is a strong indicator of forgery. Distance vectors that have enough matches to push them over a pre-determined threshold are flagged as potentially duplicated areas, and the coordinates of the 16 x 16 blocks that made up those distance vectors are highlighted within the original image. The value of the threshold will vary depending on the complexity of the image, and its tenancy to already include repeating patterns. Such an image will require a higher threshold value than that of a simpler image.

3.3 Image Resampling Detection

Resampling detection aims to be both computationally quick and robust in detecting forgeries in a variety of situations. A major advantage is that it is both scale and rotation independent, which is simply not possible with traditional pixel block comparison methods. Whilst there are a few ways to detect resampling, our chosen method operates as follows:



We firstly generate a high-passed version of the image; this is achieved through utilising a modified ideal low pass filter [11] and computing the difference between the source image and the filtered image. This produces an image with all high frequency areas (regions of change, such as edges) intact, whilst dramatically reducing other, less noticeable areas. By working on this generated, high-passed image, we are able to concentrate on areas of change, reducing the possibility of minute changes, within low frequency areas, of adversely affecting the results.

Once the high-passed image variant is generated, three n x n blocks, where n is a positive integer, can be selected by the user. One of these areas should be a suspected forged area, whereas the other two should be areas that appear to be legitimate. Performing a Fast Fourier Transform (FFT) on the three blocks gives us the underlying frequencies of the highlighted regions, which when scaled to meaningful values and plotted accordingly, allows us to view the spectrogram of the frequencies of each area. Blocks that have been resampled will tend to appear differently, generally looking distorted compared to the other two legitimate samples.

Computing the Fast Fourier Transform of an n x n matrix within MATLAB is both quick and computationally negligible, due to its efficient native implementation of both matrices and of the FFT method itself. As only three regions of the image are selected, as long as n is chosen to be a reasonable number ($<$ 500), the algorithm is extremely efficient and has a low runtime complexity.

3.4 User Interface

Whilst the application itself is more of a proof of concept testing and evaluating the chosen algorithms, a graphical user interface has been developed in order to improve the application's ease of use. Importing an image is achieved using an OS native file-picking interface; once an image is selected it is loaded in to the application and displayed. Each algorithm has a graphical button that can be selected in order to run it, and any algorithm specific parameters are changeable within the interface itself. The resulting image of each algorithm, in addition to any figures or percentages, are displayed in its own region of the application. It's therefore possible to run each algorithm on the same image without having to close the application or reset it; the user is able to easily compare the results of each algorithm within the same window.

Whilst the user interface is still rather rudimentary at this stage, it is simple, easy to use and allows direct comparisons of the results of different algorithms. As we are dealing with images, graphical user interfaces assist in displaying information and highlighting differences where a simple command line interface would undoubtedly fail. Developing the UI into a fully featured, consumer friendly and polished interface would be entirely possible; however in its current form the aim of the application is to simply convey information effectively and concisely to the user.

3.5 Data Flows & Structures

One of the largest benefits of utilising MATLAB is the way that it natively handles image files, an essential aspect of this project. Whereas third party libraries are available for other programming languages that allow the importation and manipulation of images, these tend to be handled through the use of custom, specific image classes. In comparison, MATLAB imports images as matrices, which are an entirely native and primitive data structure within the application. This not only has the advantage of being much faster than pre-defined, object orientated classes, but also allows the use of MATLAB's pre-built functions, such as the Fast Fourier Transform and Discrete Cosine Transform. These native methods run far more efficiently than attempting to run a third party library or by creating our own implementations, in addition to guaranteeing reliability when used correctly.

Unfortunately, whilst the creation of a matrix is fairly simple, initial tests showed that the update time of matrices were far slower than anticipated. This posed a problem with the algorithms on test; especially clone detection, as it relies on continuously updating matrices based on the results of any methods applied to them. Further research concluded that the memory allocation for matrices operates by allocating a large enough chunk to contain each element. Even a 16 x 16 block contains 256 elements; working with thousands of blocks and re-arranging these constantly proved to take far too much time even for a relatively small 100 x 100 px image.

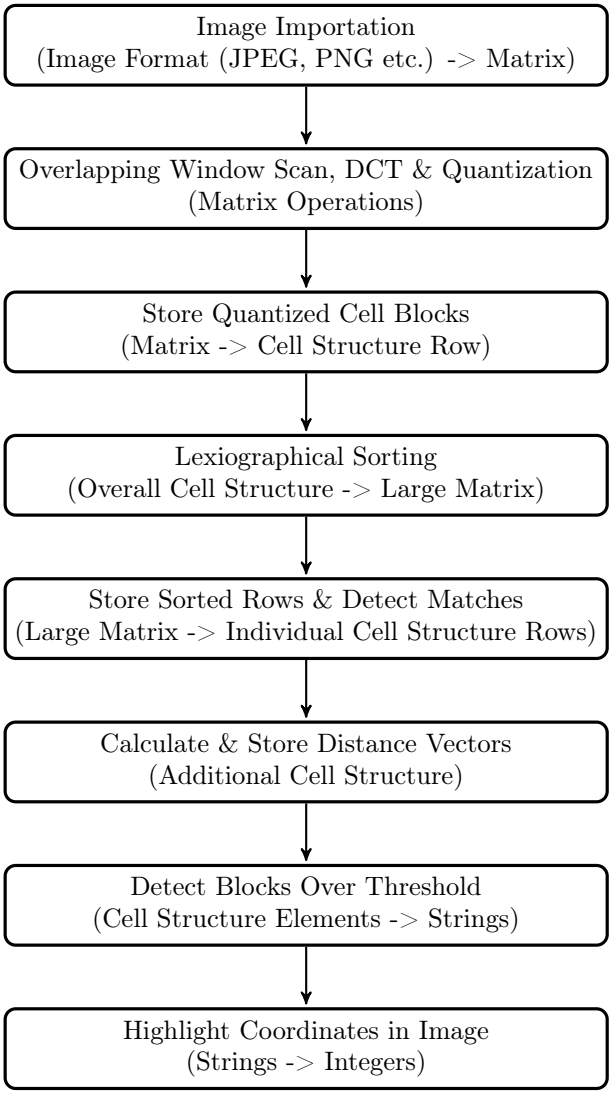
MATLAB also supports cell structures, which unlike standard matrices in-

stead stores data in available memory blocks, utilising pointers to create a meaningful connection between the data. Whilst slightly more complex, the advantage of utilising this data structure is that when a cell array is updated, the list of pointers are also updated. Computationally, this provides a huge advantage when compared to having to update the actual list of elements each time. Testing found that converting to cell arrays before the sorting process dramatically cut runtime by an average of a third, which in some cases was enough to differentiate between the algorithm completing successfully, or simply failing and running out of memory. However, as more specific operations such as the Fast Fourier Transform can only be performed on standard matrices, conversion between the two data structures is necessary, depending on the nature of the algorithm.

JPEG Error Level Analysis utilises MATLAB's native matrix implementation in order to run efficiently. Additionally, the algorithm requires no major sorting or constant updating of different matrices. Due to these properties, cell structures are not used, as converting between a cell structure and a matrix would require more computation than would be saved. An image is imported as a matrix, re-compressed, and both matrices are then compared in order to provide an error level image.

Similarly, Image Resampling Detection operates entirely on matrices, due to the fact that the native implementation of the Fast Fourier Transform only operates on that data structure. Similarly to JPEG Error Level Analysis, converting to a cell array would not provide any benefit to stability or to runtime performance. The imported image is filtered to generate a high-passed variation of the original image. We are then able to compute the 2D Fast Fourier Transform on this image, plotting the results to produce meaningful spectrograms of the underlying frequencies of each region of interest.

The nature of the Clone Detection algorithm means that we are required to convert between data structures in order to provide an optimal runtime and ensure efficiency. An overall view of the data structure conversions required for each step in the system architecture is detailed below:



Whilst the number of data structure exchanges may appear to be counter productive, each data structure is used to provide the greatest benefit regarding efficiency and runtime. Converting between two data structures requires only two computationally expensive efforts, and allows much faster operations overall. In comparison, utilising a non-optimal data structure means that we run inefficiently run operations on it thousands of times. It is therefore much more beneficial to take a couple of larger performance hits, in comparison to continuously utilising inefficient data structures.

4 Implementation

The final MATLAB implementation consists of one application, *ImageForgery.m*, split into five distinct modules:

1. Image Importation Module
2. Metadata Tag Detection
3. JPEG Error Level Analysis
4. Copy-Paste Clone Detection
5. Image Resampling Detection

A single graphical user interface is used in order to run each module, and provides a hub-like interface that works to connect each algorithm together. Once the user has imported an image, each method can be run without reloading the image or restarting the application. The results of each algorithm are each viewable in their own windows, side by side for ease of comparison.

The graphical user interface is generated using MATLAB's inbuilt GUIDE library. This provides several advantages, including a variety of default UI elements and native support for OS specific features (such as file-picking windows and button styles). This allows us to write and generate GUI code only once, whilst still providing cross platform support and reliability. GUIDE also natively supports callback functions for all interactable UI elements. All application code is encased within these specific callback functions. For example, JPEG Error Analysis code is contained within the specific errorAnalysisButton callback:

```
function errorAnalysisButton_Callback(hObject, eventdata, handles)
```

This ensures that each section of code is only run upon the users request, there is no background activity unless the user specifically chose to run an algorithm.

4.1 Image Importation Module

The image importation module allows the user to specify an image file from a local disk drive, using the standard OS file-picking interface generated from MATLAB's *uigetfile* function. The image is imported into a w x h x c matrix, whereby w and h are the image width and height, and c is the number of channels within the image. Black & white and grayscale images contain one channel, whereas colour images contain three channels for red, green and blue colours respectively.

In order to ensure that the image matrix is persistent throughout the application, we assign it the global keyboard in order to ensure it's accessible outside of the method it was created within. Whilst global variables tend to be discouraged in other languages, designating the image matrix as global minimises the

number of read-write operations performed. This can have a dramatic impact on the overall runtime and efficiency of the application.

Once the imported image is made global, it is then able to be utilised by each module in turn. In the majority of cases, all three channels of the image aren't needed and would only increase runtime. Therefore in some cases (Clone Detection & Resampling Detection), if the imported image is colour then we also convert it to grayscale in order to improve the performance of the algorithm:

```
if size(importedImage, 3) == 3
    gimportedImage = rgb2gray(importedImage);
end
```

4.2 Metadata Tag Detection

Although the chosen image has been converted into a matrix, Metadata information is lost when this image importation has taken place. This means that detection of any potential forgery flags has to come from reading the image directly. Fortunately, MATLAB provides a native function, *iminfo*, for gathering the raw data of an image. This in turn allows us to import image metadata. The *iminfo* function returns a structure containing all of the raw data associated with the image file. A typical structure as returned by this method is illustrated below:

Field	Value	Min	Max
Filename	'D:\Users\Josh\Google Drive\Docu...		
FileModDate	'04-Feb-2015 16:26:14'		
FileSize	2075891	2075891	2075891
Format	'jpg'		
FormatVersion	''		
Width	2048	2048	2048
Height	1363	1363	1363
BitDepth	24	24	24
ColorType	'truecolor'		
FormatSignature	''		
NumberOfSamples	3	3	3
CodingMethod	'Huffman'		
CodingProcess	'Sequential'		
Comment	*0x0 cell*		
BitsPerSample	[8,8,8]	8	8
PhotometricInterp...	'RGB'		
Orientation	1	1	1
SamplesPerPixel	3	3	3
XResolution	72	72	72
YResolution	72	72	72
ResolutionUnit	'Inch'		
Software	'Adobe Photoshop CS6 (Windows)'		
DateTime	'2015:02:04 16:26:13'		
DigitalCamera	*1x1 struct*		
ExifThumbnail	*1x1 struct*		

We are particularly interested in the Software field, as this suggests that the image has been opened and manipulated within an application after its original capture. The DateTime field can then be used in conjunction with the Software field to suggest the last modified date and time. However, as these fields are only included within the metadata if a piece of software specifically adds them, we're required to use an evaluation function in order to test whether they exist:

```
eval('softwareFieldExists=1;info.Software;','softwareFieldExists=0;');
```

If the evaluation is true, then it is simply a case of displaying the contents of the fields to the user, indicating modification and potential forgeries.

4.3 JPEG Error Level Analysis

Error Level Analysis operates by calculating the error level between both the original image, and the same image but saved at a known error level. MATLAB's native handling of matrices and support for difference functions allows us to implement this algorithm in a very compact, concise and efficient way. This is achieved through re-compressing the image via the native *imwrite* JPEG encoder, which allows us to choose a specific error level via a compression ratio percentage:

```
imwrite(importedImage,tempFileName,'Quality',95);
```

As previously mentioned, 95% was chosen as a good balance between generating a clear error level image, without degrading the quality of the image to such an extent that too much detail is lost. Once the newly re-compressed image is imported, we delete the temporary file.

MATLAB provides a simple solution to generating the difference between the two images. As both are now stored in memory as matrices, we are able to easily find the difference between both by utilising the *imabsdiff* function, which calculates the absolute difference between matching elements in the original image and the error induced image:

```
imageDifference = imabsdiff(importedImage,importedLowerQualityImage) * 30;
```

The resulting error level image is increased by a factor of thirty in order to highlight any mismatching areas. A standard error level image will appear purple in colour; where substantial differences in the JPEG compression quantization exists the region will tend to be of a much lighter colour than the surrounding area. Conversely, equally distributed coloured areas indicate a low margin of error. The overall error level between both images is then calculated by determining the mean error value within the image, before converting that from a scaled value to a percentage.

4.4 Copy-Paste Clone Detection

The nature of this algorithm requires many iterations and comparisons. MATLAB excels in working with matrices, however is less optimised when faced with multiple loops. It was therefore highly important that loops were kept to a minimum, instead opting to use more efficient methods such as the colon operator on matrices. Modifying a row of a matrix using the colon operator is almost instantaneous regardless of the matrix size, whereas a for loop takes far longer to complete. Taking advantage of MATLAB's matrix orientated nature and implementing the algorithm in a more unique way allows us to ensure optimal performance in these situations.

In addition, part of the algorithm requires a 16 x 16 extended quantization matrix to be generated. A simple MATLAB implementation was created based upon a standard calculation formula [8] in order to populate this extended matrix:

```
% Standard JPEG quantization 8x8 matrix
quMatrix = [4 4 6 11 24 24 24 24
  4 5 6 16 24 24 24 24
  6 6 14 24 24 24 24 24
  11 16 24 24 24 24 24 24
  24 24 24 24 24 24 24 24
  24 24 24 24 24 24 24 24
  24 24 24 24 24 24 24 24
  24 24 24 24 24 24 24 24];

quMatrix16 = zeros(16,16);

% Calculate 16 x 16 matrix
for i = 1:8
    for j = 1:8
      quMatrix16(i,j) = quMatrix(i,j) * 2.5;
    end
end

quMatrix16(1,1) = 2.0 * quMatrix(1,1);

for i = 9:16
    for j = 1:8
        quMatrix16(i,j) = quMatrix(1,8) * 2.5;
    end
end

for i = 9:16
    for j = 9:16
        quMatrix16(i,j) = quMatrix(8,8) * 2.5;
    end
end

for i = 1:8
    for j = 9:16
        quMatrix16(i,j) = quMatrix(8,1) * 2.5;
    end
end
```

The standard 8 x 8 quantization matrix is firstly loaded; this is used as the basis for the conversion to a 16 x 16 matrix. As this matrix has a fixed value that remains unchanged, the final implementation simply assigns the generated values to an empty matrix, instead of computing the same values each time. This ensures that the impact on overall performance is computationally insignificant. For the purpose of our testing, we will only operate with a 16 x 16 sliding window, however this formula could also be adapted to both smaller or larger window sizes.

MATLAB has no direct support for sliding an overlapping window across an image. The *blockproc()* function can be used in order to process chunks of an image, however it only supports simple methods such as calculating the mean of the block. As we need to extensively process each block and then store it, this method is not applicable to us and looping through the image is instead preferable. Splitting the image in to chunks is t achieved through use of the colon operator:

```
subImage = gimportedImage(y:y + (height), x:x + (width));
```

Performing the 2D Discrete Cosine Transform is possible via the native *dct2()* method. We are then able to quantize this matrix using the extended quantization matrix that we previously calculated. The *quality* factor is used in increasing or dampening the effect of the quantization, making quantized blocks more or less similar to each other.

Once the current block has been quantized, it is converted into a single vector, which allows us to store each block as a row within a new matrix. However as previously stated, constant updating of matrix records is slow, and therefore we instead store the block as a single row within a cell structure in order to improve efficiency:

```
qSubImageArray = reshape(quSubImage',1,[]);
quantisedValuesCell{counter,1} = qSubImageArray;
```

At this stage, we also store the x and y coordinates of the top left pixel. This allows us to reduce the number of comparisons needed later as we aren't then required to match each row to its original coordinates. Lexicographical sorting is possible by utilising MATLAB's *sortrows()* function, however this can only be run on standard matrices. A conversion of the overall cell structure to a standard matrix allows us to sort the matrix in its entirety, before converting it back to a cell structure for the comparison stage of the algorithm.

Our cell structure is then looped over, comparing each row to the following row and checking their equality. As cell structures simply contain pointers to elements within memory, it's important that we utilise the *isequal()* method instead of simply using a standard equality comparison of ==, as this could incorrectly determine that the elements are different when they are in fact matching.

When equal pairs are found, the shift vector of each matching pair is calculated and stored within a cell structure; if that shift vector already exists then

we increase the count by one, if not then we create a new row. Each x and y coordinate is appended to an existing list of coordinates that contribute to the appropriate shift vector. A counter is used to indicate how many rows already exist within the shift vector cell structure.

```
% Find if this shift vector exists
doesExist = strcmp(localShift, shiftVectors);
doesExist = any(doesExist(:));

% Does exist, so find index and increase count by 1
if doesExist
localIndex = find(strcmp(localShift, shiftVectors));
shiftVectors{localIndex,2} = shiftVectors{localIndex,2} + 1;
shiftVectors{localIndex,3} = strcat(shiftVectors{localIndex,3}, {' '}, x12);
shiftVectors{localIndex,4} = strcat(shiftVectors{localIndex,4}, {' '}, y12);

% Doesn't exist, so we need to find the current new row and append
 else
shiftVectors{currentRow,1} = localShift;
shiftVectors{currentRow,2} = 1;

shiftVectors{currentRow,3} = strcat(shiftVectors{currentRow,3}, {' '}, x12);
shiftVectors{currentRow,4} = strcat(shiftVectors{currentRow,4}, {' '}, y12);

currentRow = currentRow + 1;
end
```

Finally, we approach the end of the algorithm with a cell structure containing each shift vector, its number of occurrences and the coordinates that contributed towards that particular shift vector. It is then a case of simply detecting which shift vectors occur more times than noted in our chosen threshold value, which gives us the regions to be highlighted as potential forgeries. Each coordinate pair is then plotted on the source image, with the 16 x 16 pixel region around each pair highlighted in order to match the blocks processed within the algorithm.

4.5 Image Resampling Detection

Whilst the concept of resampling detection itself is fairly complicated, MATLAB's native support for filters and the Fast Fourier Transform allows us to implement the algorithm very efficiently. The source image is firstly converted into the frequency domain using the *fft2* function. This allows us to then filter the image as required. An ideal low pass filter is implemented, as adapted from [11]; here all frequencies above our cut off threshold, u0 are removed. In our implementation, this threshold is set as 50 by default:

```
[M N] = size(gimportedImage)

% Ideal low pass filter
u = 0:(M-1);
v = 0:(N-1);

idx = find(u > M/2);
u(idx) = u(idx)-M;

idy = find(v > N/2);
v(idy) = v(idy)-N;

[V,U] = meshgrid(v,u);
D = sqrt(U.^2+V.^2);
H = double(D <= u0);
```

Passing the result through an Inverse Fourier Transform and taking the real values gives us a resulting low-passed image. In order to generate a high-passed image from this, we simply calculate the difference between the original image and the low passed image:

```
highPassedImage = minus(double(gimportedImage),lowPassedImage);
```

The algorithm then picks out three regions of interest within the high passed image, as specified by the user. Calculating the Fast Fourier Transform of these regions and plotting the resulting spectrogram gives us the ability to view potential re-sampling artefacts. However as the result of the Fourier Transform returns both real and imaginary numbers, it's important that we scale the result correctly:

```
F = fft2(vhighPassedImage);
F = fftshift(F);

F = abs(F);
F = log(F+1); % Use log scaling
F = mat2gray(F);
```

Firstly, we utilise *fftshift()* in order to re-arrange the frequencies of the transform. By default, these frequencies are shifted towards the four quadrants of the transform; re-arranging these so that the zero component is at the centre of the array provides a much better visualisation. We then remove all negative or imaginary components in order to improve the clarity of the spectrogram, scaling the results on a logarithmic scale. We use $\log(F + 1)$ for our scale, as $\log(0)$ is undefined and would cause the algorithm to fail. Finally, the *mat2gray()* method is used in order to convert each logarithmic value onto a scale of between 0 and 1, allowing a grayscale visualisation of the data. The resulting matrices are plotted, producing three distinct spectrograms of the chosen regions of interest.

5 Results & Evaluation

In order to test the efficiency of each algorithm, a sample set of 40 images has been created. The breakdown of these images are as follows:

- 20 Unique Forged Images - These have been created by manipulating existing background images. A variety of forgery methods have been utilised, including copy-paste forgeries, splicing of two images and modifying existing sections of an image.
- 20 Image Manipulation Dataset Images - A sub-selection of images included as part of the Image Manipulation Dataset [13].
- 10 Unmodified Images - Original images that lack any kind of forgery have also been included in order to provide a benchmark test for false positives.

Each unique image has a width of 500px; with the height varying slightly depending on the aspect ratio of the source image. This provides an equal footing to each image, and ensures that any differences in runtime are down to the complexity of the image as opposed to differences in image resolution. The majority of images are saved as JPEG images, with compression quality set to either High (75%) or Maximum (100%). Where a source image existed as a PNG, the forged variant was also saved as a lossy PNG. Any image manipulation was carried out within Adobe Photoshop CS6; however forged images weren't saved in a lossy format until all modifications were complete. This ensured that the images were only re-compressed once, as repeat compression would degrade the quality of the images and possibly have an impact on the test results.

Images from within the Image Manipulation Dataset were chosen in order to demonstrate the competency of the algorithms on standard library images. As the dataset contains a large number of images in total, 20 of these were chosen at random in order to provide a large enough sample to compliment the pre-existing test images. Unfortunately, by default the resolution of each image varies greatly, which would not provide a fair estimate of runtime. Regardless, as the runtime complexity of the Copy-Paste Clone Detection algorithm is fairly large compared to other algorithms, image resolution was reduced for the purpose of testing. Where possible, images were cropped to 500px x 500px in order to reduce the possibility of downsampling affecting the forgeries within the images. However on occasions, where this wasn't possible, for example when multiple forgeries spanned the entire image, the image was downsampled using the Bicubic Sharper resampling method.

All images used within the testing phase of the application are free of copyright restrictions, with a licence to modify and distribute without issue [10]. Full test results of each image are included as an appendix to this report.

In order to test the runtime of each algorithm, MATLAB's native *tic()* and *toc* functions were used, as they present an accurate reading in elapsed seconds.

Timing of each algorithm starts at the first line of the appropriate button's callback code, and finishes once the resulting image has been displayed and all processing has completed. As this time is affected by outside factors such as CPU usage and available RAM, each result is timed three times and the average time is computed.

All testing was completed within MATLAB R2014a, running on a quad core Intel i5 CPU and an NVIDIA GTX 970 GPU. The operating system environment was Windows 10 Developer Preview at the time of testing.

5.1 JPEG Error Analysis

Error Analysis operates by calculating the error factor between different compressions of the same image. The more an image has been compressed, the less the error level will be between re-compressions. This allows us to highlight forged regions that have been re-compressed fewer times than the remainder of the image, providing a larger margin of error. Due to the nature of the algorithm, only JPEG images can be used as the effects of lossless compression do not compound each time the image is saved. In fact, re-saving a PNG file will keep the integrity of the image intact, it will not cause further quality loss. Whilst PNG images could be converted to the JPEG file format, this would simply cause the whole image, forgery included, to be compressed at the same level. There would then be no perceivable difference within the quantization process of the legitimate and forged areas, as this would have been the first time JPEG compression occurred. As a result of this, PNG images have been excluded from the JPEG Error Analysis results set as the algorithm is not appropriate for detecting forgeries within these images.

5.1.1 Detection Rates & Quality

The quality of the JPEG image greatly affects the overall success rate of the algorithm. Here we see a comparison of the results when running on source image number 1 at Maximum (100%), High (75%), Medium (50%) and Low (25%) quality levels:

Figure 1: Forged JPEG Image. Notice the addition of the robotic dog on the hillside.

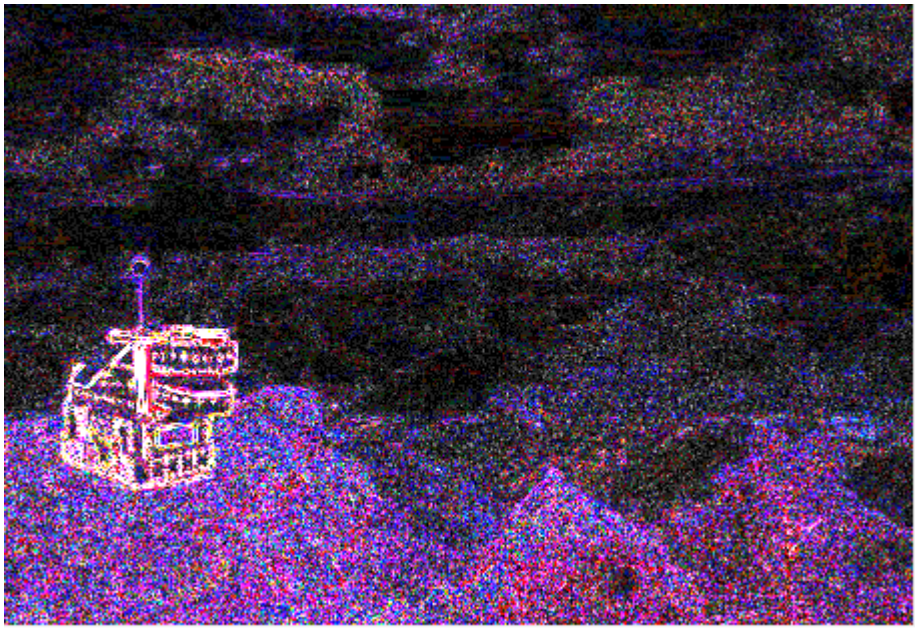


Figure 2: Maximum, 100% Quality Error Level Image

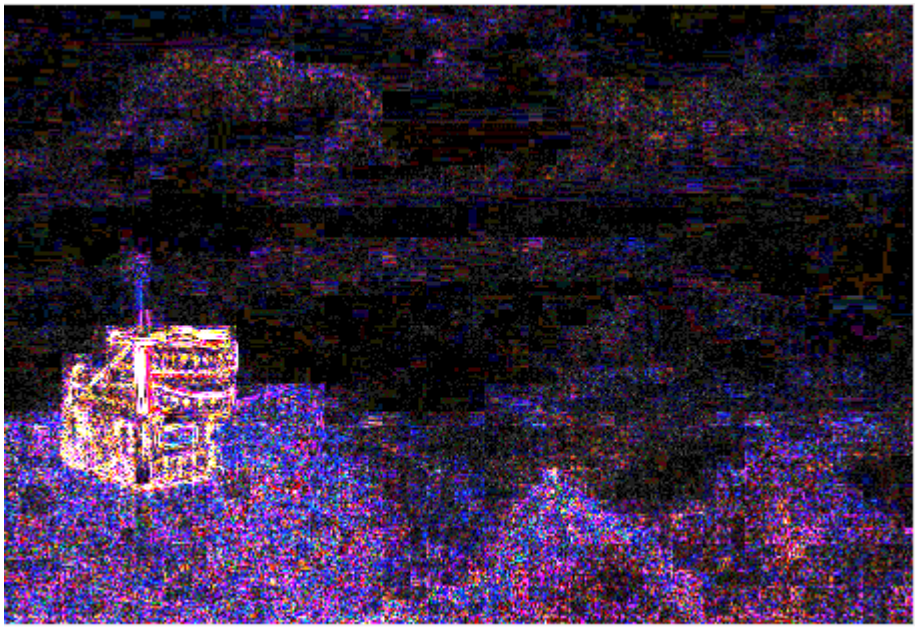


Figure 3: High, 75% Quality Error Level Image

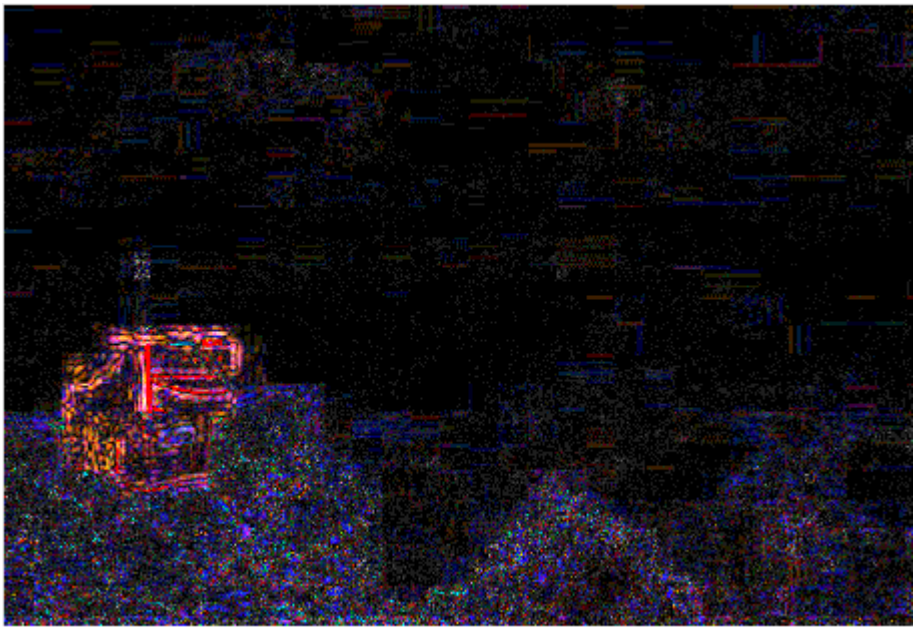


Figure 4: Medium, 50% Quality Error Level Image

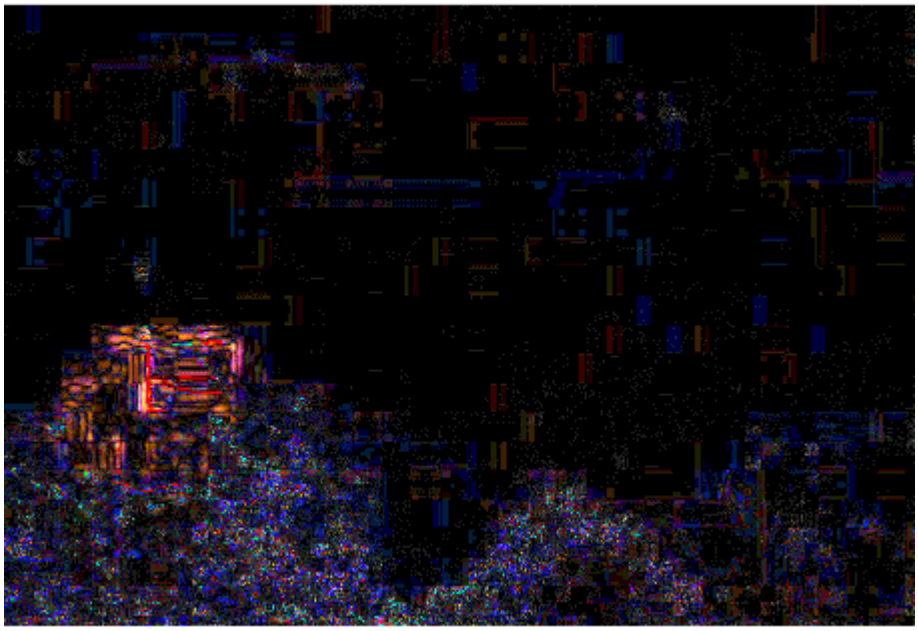


Figure 5: Low, 25% Quality Error Level Image

The above example is the ideal scenario for the algorithm, whereby the background image has been compressed (possibly at a medium or high quality level) and re-sampled, diminishing detail within the error level image. As the robot has only been compressed once, and at a different quality level to the background image, we are able to easily perceive the difference between the forged region and the remaining, legitimate areas. Depending on the quality of the image, the error level ranges between 5% and 25%, suggesting that the vast majority of the image has already been re-compressed.

Once we reduce the overall quality of the image, although the outline of the re-compressed object is still visible, it becomes much more difficult to distinguish detail, as the whole image is further compressed to a much lower level than the source. This becomes particularly troublesome when the forged images themselves have been re-compressed numerous times, diminishing the overall detail of the image to such an extent that highlighting newly forged areas utilising Error Level analysis isn't possible.

Newly compressed images can also pose an issue for the algorithm. In this example, the original source image was a PNG file, and therefore no lossy compression had been performed. Once the forgery was completed, the image was saved as a JPEG file at a high quality level, meaning that the JPEG Quantization step was only performed once on the entire image:

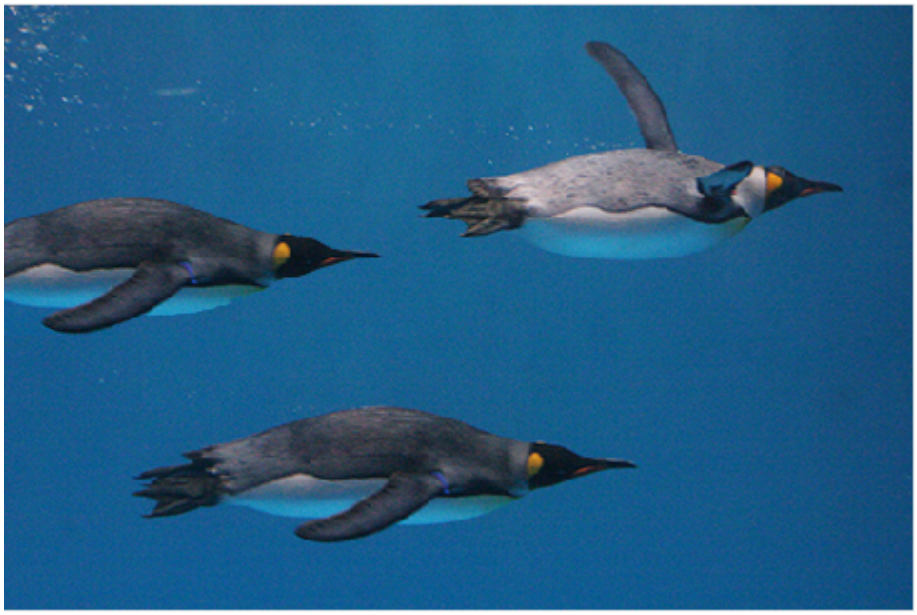


Figure 6: High, 75% Quality JPEG Image. Notice the duplication of the dark coloured penguin.

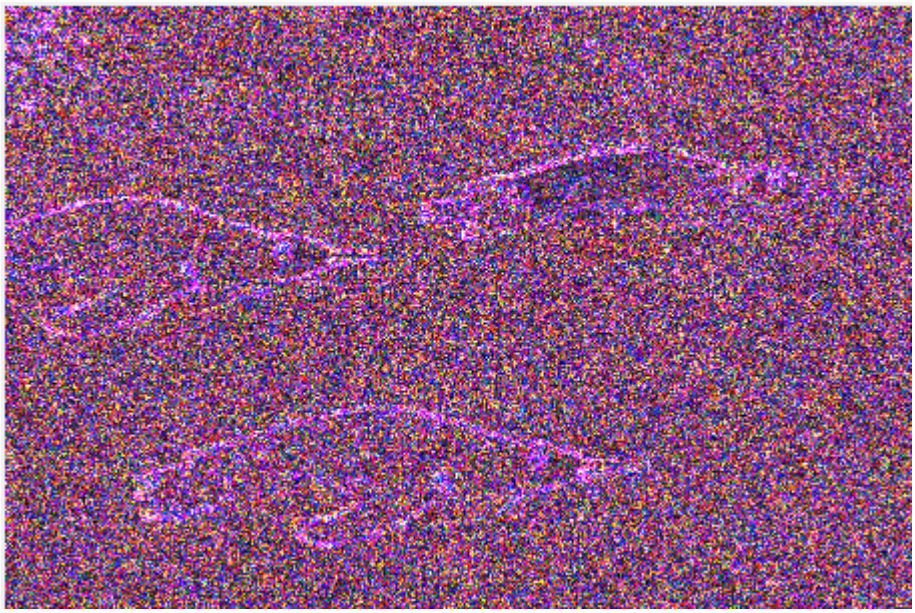


Figure 7: Error Level Image, lacking any obvious detail.

The overall error level of this image is 45%, meaning that there was a large margin of error between the original and re-compressed image. As the background had no previous compression artefacts, there is no noticeable difference between the legitimate and forged areas of the image, the overall error level is far too high.

5.1.2 Sample Set Results

As the forged images from the Image Manipulation Dataset are overwhelmingly saved as uncompressed PNG files, Error Level Analysis is not suitable for detecting forgeries within those samples. Overall, the algorithm was run able to be run on all 13 JPEG images out of the 20 original forgery images. 6 of these images were clearly detected as forgeries from the error level image produced, producing a 46% average detection rate for the samples used. Whilst this seems disproportionately low, it's important to note that Error Level Analysis operates on a specific forgery type; that is compressed, JPEG images that have been modified and then further compressed. When images have been forged in this manner, the results of the algorithm are both accurate and clearly visible within the error level images:

Figure 8: Forged Image (15). Overall error level is 14%.

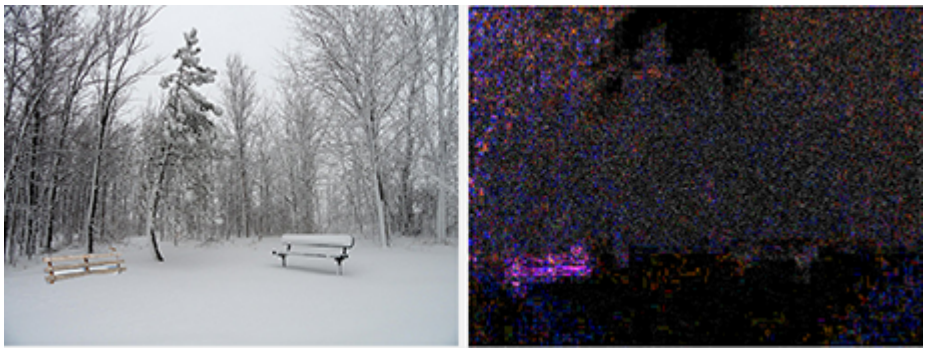


Figure 9: Forged Image (20). Overall error level is 12%.

The algorithm was also run on the 10 untouched, benchmark images in order to test the rate of false positives. A false positive within this algorithm can be defined as a single, highlighted region that looks to be a potentially forged area. Images that contain false regions but also have a high error level ($> 30\%$) aren't included as false positives.

Within the unique sample set, of the 13 JPEG images two were determined to produce potentially misleading results; a respectable false positive rate of 15%. For example, here a section of the sea has been duplicated and rotated, however the algorithm gives the indication that the boat has been modified:

Figure 10: Forged Image (18).

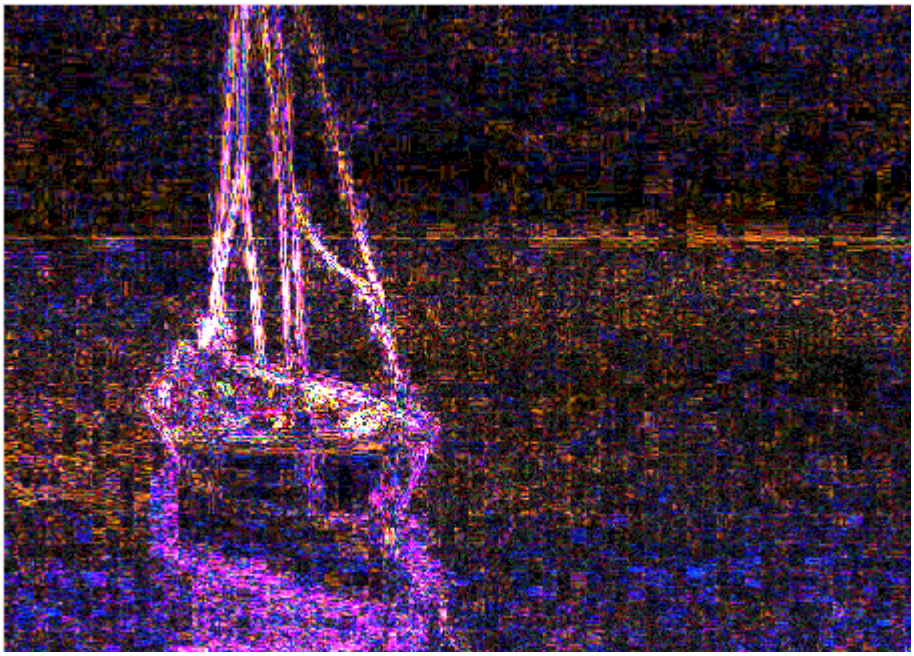


Figure 11: Forged Image (28). Overall error level is 20%.

Within the untouched benchmark set, 2 out of 10 (20%) of images were falsely identified as potential forgeries. Here, the barn looks out of place and is being flagged as an area of error, despite being perfectly legitimate:

Figure 12: Forged Image (f3).

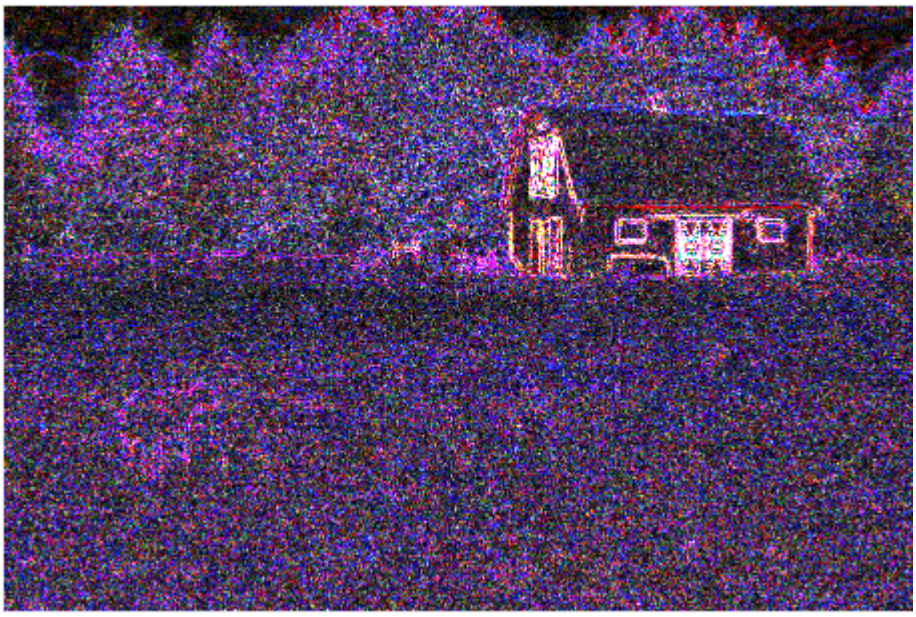


Figure 13: Forged Image (f3). Overall error level is 30%, at the very end of the acceptable error range.

5.1.3 Error Level Correlation

As expected, a correlation was found between the overall error level of an image and its ability to be detected as a forgery. Images of a high error level (around 30% or higher) were found to contain too much variance in order to pinpoint an exact forgery location. These images tended to be newly compressed, or previously converted from a lossless image format. On the contrary, a large enough error level (over 10%) is required in order to detect any kind of noticeable forgery area within the image. Images with a very low error level tended to have been saved at a very low JPEG quality level, or had been re-compressed too many times so that a substantial amount of detail was lost. Our tests have therefore shown that the ideal range for the image error level is between 10 - 30%.

5.1.4 Algorithm Performance

Runtime performance was extremely positive, averaging a mere 0.03 seconds per image comparison. Due to MATLAB's native implementations of both matrix comparisons and JPEG encoder, each comparison is achieved extremely quickly. Whilst, as expected, the runtime does degrade as image resolution increases, a 5,000px x 5,000px sample image completed in an average of 1.55 seconds, and a 10,000px x 10,000px averaging only 6.93 seconds. As a whole, the algorithm is extremely efficient on all image sizes, as long as the user has the required free memory in order to store both the original and error level images in RAM.

Overall, whilst the detection rate of the algorithm is fairly low on samples of randomly forged images, its very efficient nature and low runtime ensure that it has a place supplementing additional forgery detection algorithms. Despite not being robust enough to detect multiple forgery types, when operating on re-compressed JPEG images, the results are clear and allow the user to easily detect potentially forged areas.

5.2 Copy-Paste Clone Detection

The copy-paste detection algorithm operates on multiple image formats, as it does not rely on any inaccuracies or changes due to quantization or compression techniques. However, whilst the algorithm was run on all 45 image samples, it is important to note that by design it will only detect a specific type of image forgery; that is where regions have been copied and pasted from existing areas within the original image. It would therefore be unfair to assume that forgeries containing new objects would also be detected by the algorithm.

Despite this, a large percentage of forgeries contain elements from the existing image, as it is an easy way to ensure that the forgery blends in well with the original image. As such, whilst the algorithm will only detect a specific type of forgery, it's such a commonly used technique that a dedicated algorithm is required.

5.2.1 Detection Rates & Unique Parameters

The algorithm contains two unique parameters that can be controlled within the UI; the threshold value and the quantization quality factor. These parameters can have a dramatic result on both the detection rate, and the number of false positives determined by the algorithm.

Threshold Value - This parameter determines the minimum number of matches required for each distance vector to be detected as a potential forgery and highlighted. As we lower this value, more distance vectors will be included in our final forgery set, and as a result their corresponding 16 x 16 blocks will also be highlighted within the original image. Conversely, increasing the threshold value too much will reduce the overall detection rate of the algorithm, as seen here:

Figure 14: Forged JPEG Image (3).

Figure 15: Clone detection result at a threshold of 10 and a quality value of 0.5. Notice the false positives in the sky and clouds.

Figure 16: Clone detection result at a threshold of 300. Whilst we have reduced some false positives, we've also stopped detection of the cloned chimneys.

Modifying the threshold level has little performance impact, as the cell structure containing the list of distance vectors is still populated as usual. The only difference is the number of pixel blocks that are highlighted on the original image; and as the image matrix is being modified regardless, the addition of a few hundred more modifications isn't going to have a dramatic impact on runtime performance. In the above example, the average runtime utilising a threshold value of 10 is 43.89 seconds. Increasing the threshold value to 300 gives us an average runtime of 45.14 seconds, a minor increase of less than two seconds compared to the original runtime.

Quantization Quality Value - The second user-modifiable parameter is the quality level of the quantization. The larger this value, the more the effect of the quantization on each 16 x 16 block, making different blocks appear more and more similar to each other. This is useful as often an image has been re-compressed and re-sampled, slightly modifying the surrounding pixels enough to have an effect on the results of the algorithm. Here we can see the dramatic difference that minor quality level changes can have on the results of the algorithm:

Figure 17: Forged JPEG Image (5).

Figure 18: Clone detection as a result of a quality factor of 0.2 and a threshold value of 10. Notice the lack of detection.

Figure 19: Clone detection as a result increasing the quality factor to 0.5. The cloned bikes have been detected with a few false positives.

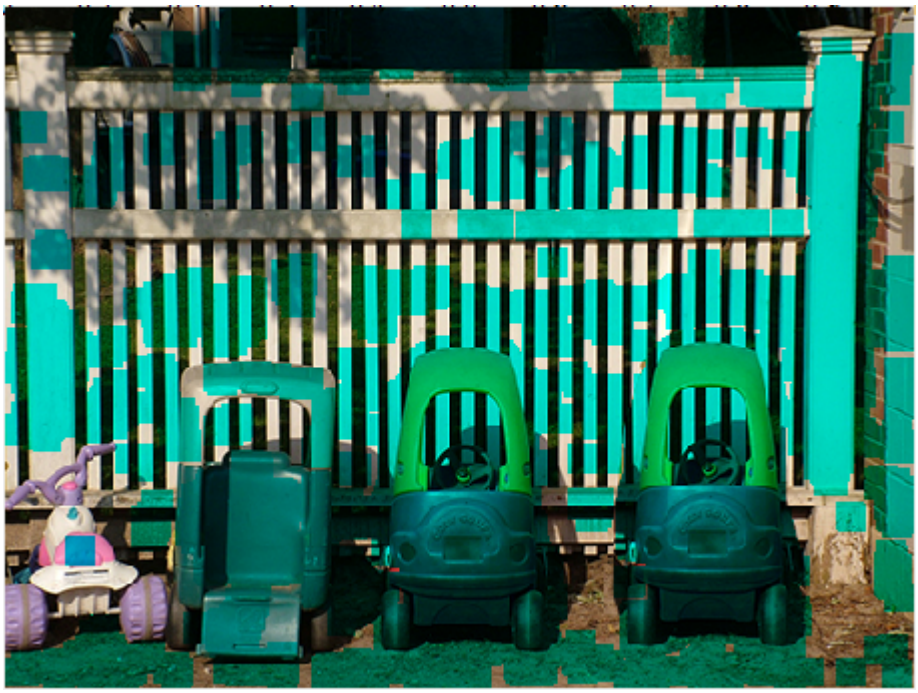


Figure 20: Clone detection as a result of further increasing the quality factor to 2. There are too many false positives to indicate a result.

Changing the quantization quality factor has the potential to have a dramatic impact on the algorithm's performance. As the quality factor adjusts the amount of quantization, as it increases more and more 16 x 16 blocks get flagged as being identical to each other. This in turn causes far more comparisons within the algorithm, increasing the amount of time spent modifying each matrix and cell structure. The final list of distance vectors is therefore much larger the higher the quality factor is set. In the above examples, the average runtime for each quality factor was 14.62 seconds, 26.10 seconds and 104.00 seconds respectively. Increasing the quality factor therefore has a dramatic impact on the overall performance of the algorithm. Ideally, it is optimal to keep the quality factor as low as possible, whilst retaining enough quantization to overcome any potential compression artefacts.

5.2.2 Sample Set Results

The copy-paste clone detection algorithm is file format independent, and whilst it is most effective on uncompressed or high quality, lossy images, it will run on any image regardless of its compression ratio. Despite the fact that some images produced better results when the algorithms parameters were tailored to that specific image, default values of 10 for the threshold and 0.5 for the quantization quality factor were used throughout in order to provide a fair comparison of the general detection rate of the algorithm.

Out of all 50 images, the majority returned some form of result, generating the following breakdown:

- **Unique Forged Images** - Of the 20 unique forged images, 17 ran successfully, with 3 failing to produce an output within a reasonable time ($<$ 5 minutes). Overall forgery detection rate was 29%, however this includes all forged images; clone detection only successfully operates on a specific type of forgery. Of the 20 images, 5 were forged using copy-paste duplication methods, and the detection rate of the algorithm on these images was an impressive 80%. This shows that the algorithm is extremely effective in detecting copy-paste forgeries.
- **Image Manipulation Dataset** - Results on the dataset varied somewhat, as the original size of the images was too large in order to successfully run through the copy-paste clone detection algorithm. Where possible, the region containing the forgery was cropped in order to reduce the overall resolution of the image without any resampling, however this wasn't always possible. In addition, some of the more elaborate image forgeries weren't detected as any rotation or re-sampling of the forgery selection severely limits the ability of the algorithm to detect the sections as similar. However, considering the nature of the forgeries themselves are unknown, overall results were respectable, with an average detection rate of 50% for the tested images.

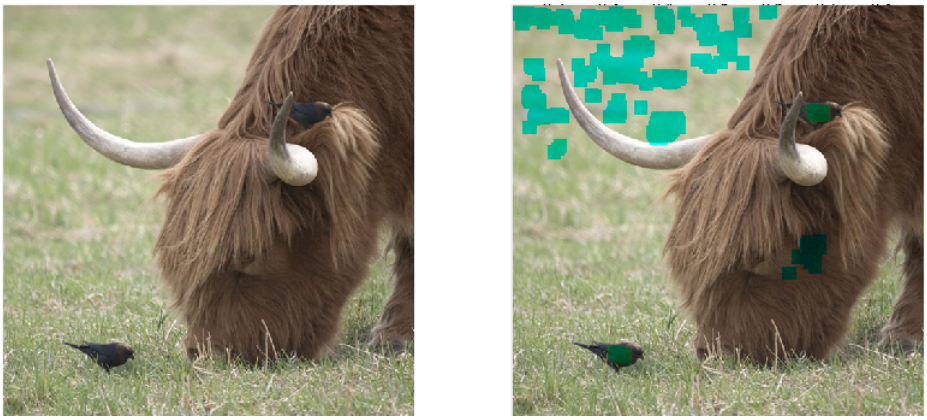


Figure 21: Forged Image (r3) Here the crow has been correctly highlighted, along with some false positives. The contrast of the crow block has been increased for clarity.

Figure 22: Forged Image (r11) The background has been highlighted as it is all one shade of blue. However the cloned sections of the horse statue are correctly identified.

Whilst adjusting the quality factor can have a large impact on how similarly looking each 16 x 16 block appears, even minor variations within image regions can have an impact on the detection results of the algorithm. For example, in the dataset image below, both trees are clearly duplicates, and look very similar at a 100% view level:

Figure 23: Forged Image (r7). Notice the two duplicated trees on either side of the frame.

Displaying both trees at a very high zoom level shows that whilst they are indeed similar, re-sampling has occurred which has modified clusters of pixels slightly:

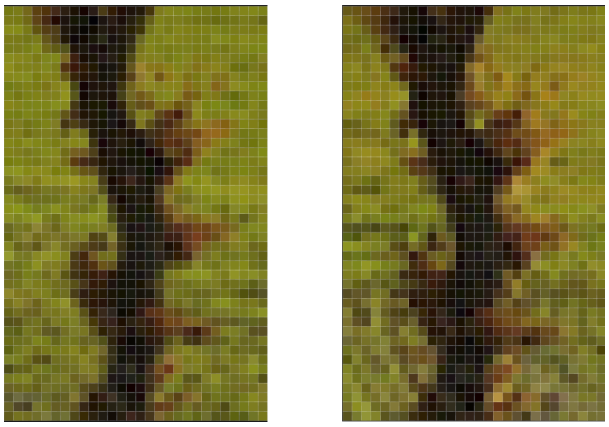


Figure 24: The right and left hand side trees, respectively. Slight variations exist between both images, despite aligning up exactly.

The algorithm would therefore detect each of these 16 x 16 blocks as different, causing the forgery to remain undetected. Whilst the quality factor could be increased, this would also have create too many false positives. Many images contain naturally occurring patterns, and false positives are inevitable with an algorithm of this detail. Out of the images successfully tested, each image contained at least one false positive block. Using the penguin forgery image as shown in the Error Level Analysis section, here we see here that even though both penguins have been flagged as duplicated, the majority of the sea has also been flagged as a cloned region:

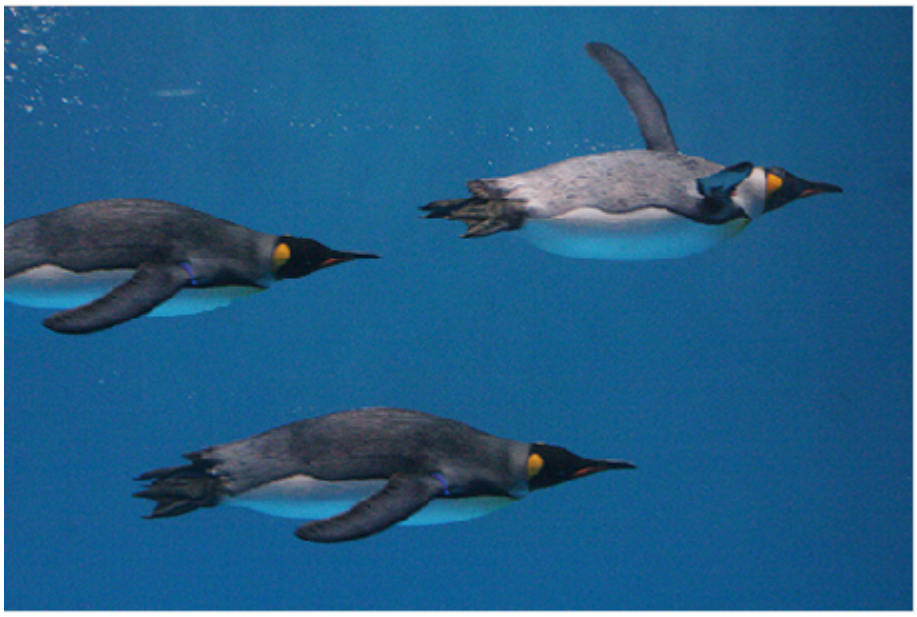


Figure 25: Forged Image (8). The two darker penguins have been cloned.

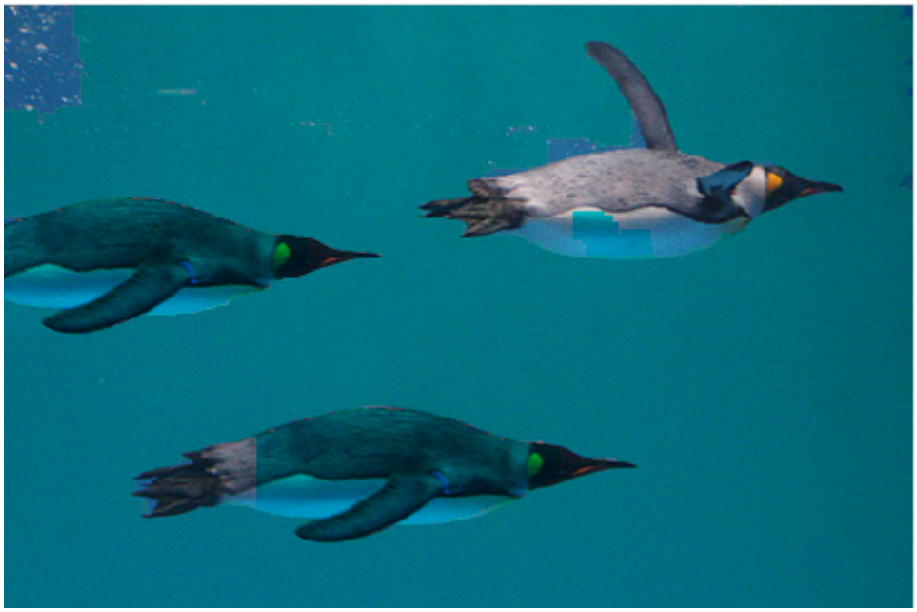


Figure 26: Resulting clone image, with the background contrast increased in order to better show the false positives.

This also has the effect of dramatically increasing the runtime of the algorithm, taking 242 seconds to complete. Images that failed to produce any results within a reasonable time frame generally had large amounts of natural patterns, such as this example image:

Figure 27: Forged Image (9). Here the sky is largely one shade of blue, which produces too many results.

5.2.3 Algorithm Performance

The performance of the algorithm varies significantly depending on the nature of the image used. As mentioned above, more complex images with naturally occurring patterns increase the runtime substantially, sometimes to the point where the algorithm no longer completes successfully. The smallest computation time on a successfully detected forged image was 24.95 seconds, whereas the largest recorded time was 242.65 seconds, taking almost 10x longer to compute.

In addition, the number of operations is highly dependant on the total amount of pixels within the image. Each image used as part of the sample data had a width of 500px, and reducing the size of an image had a dramatic effect on the computation time of the algorithm. Image sample 5, is a 500px x 375px image that took an average of 24.95 seconds to compute. Halving this image resolution meant that the algorithm took a mere 3.32 seconds to complete. Each pixel is read by the 16 x 16 moving window, and so increasing the resolution increases the number of comparisons required when working through the entire algorithm.

However, it's important to note that MATLAB is an interpreted language, and that performance would vary substantially depending on the language used for the implementation. Whilst the overall performance of MATLAB's image

processing is impressive, it tends to falter slightly when large operations are required within different looping sections.

Overall, whilst the general detection rate of the algorithm is acceptable, it excels when used on known copy-paste cloning forgeries, resulting in a high detection rate. Many images are manipulated in this way, and whilst re-compression and resampling can negatively affect the ability of the algorithm, it is able to detect these specific forgeries in the majority of cases. Performance could be improved if required by utilising multi-threading, which in turn would allow images of a larger resolution to also be used.

5.3 Image Resampling Detection

Similarly to Copy-Paste Clone Detection, the Image Resampling algorithm accepts all image formats, it is however a far more generalised algorithm. It aims to detect resampling artefacts by computing the underlying frequencies of three regions of interest and plotting them as a visible spectrogram. Here, potential anomalies caused by methods such as rotation, compression or scaling may be viewed. As such, in a similar way to JPEG Error Level Analysis, it focuses mainly on a side effect of saving an image forgery, as opposed to attempting to detect a specific type of forgery method.

The ideal conditions for the algorithm are images that are either uncompressed, or use a low compression ratio. This ensures that anomalies within the spectrogram aren't simply all compression artefacts; the lower the quality of the image the harder it becomes to detect these minute differences.

5.3.1 Detection Rates & Quality

Whilst there are no direct parameters, test results varied considerably based on whether the image was saved as a lossless PNG, or highly compressed. Here, utilising the same three coordinates, we see the difference between the quality of the spectrograms produced:

Figure 28: Forged Image (r2). The three points listed are ROI 1, ROI 2 & ROI 3 respectively.

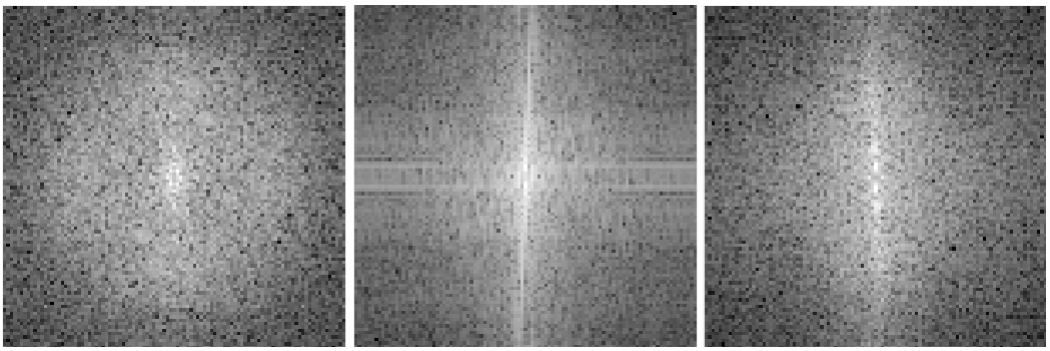


Figure 29: The resulting PNG spectrograms. We can see that whilst ROI 1 & 3 are fairly consistent, ROI 2 is drastically distorted, suggesting resampling.

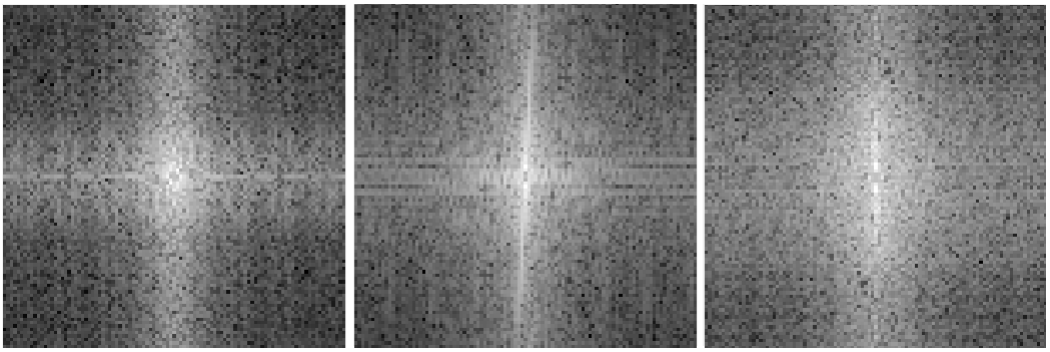


Figure 30: Using a low quality JPEG version of the image, whilst ROI 2 is still distorted, ROI 1 has been affected by compression, causing the result to be less clear.

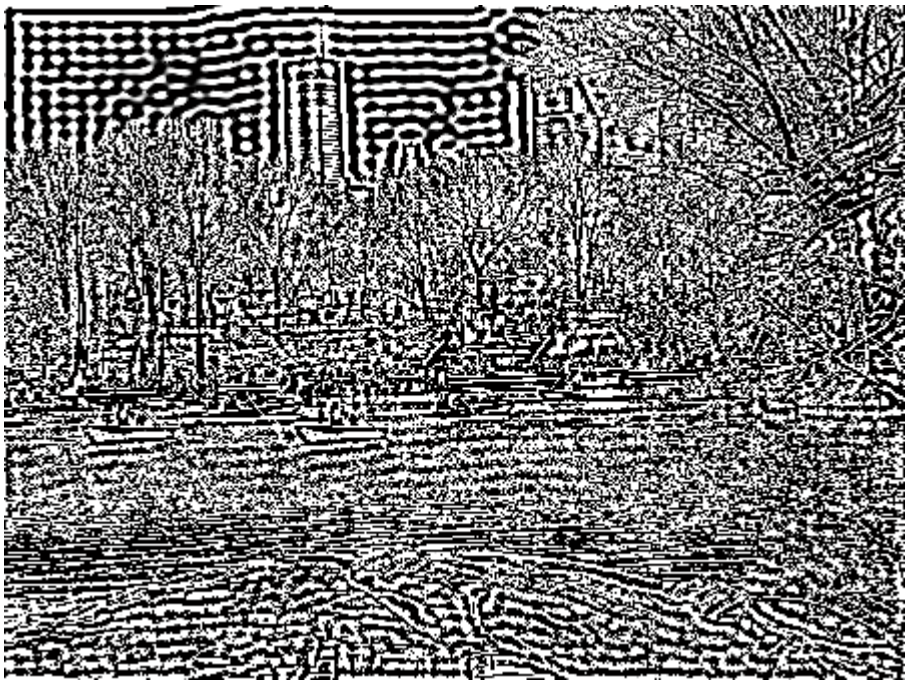


Figure 31: The high pass image used when calculating the above spectrograms. With the exception of the sky (due to overexposure in the source image), the image is fairly uniform.

Utilising this high pass image allows us to focus solely on edges and important details, and reduces the effect that small, minor blemishes may have on the overall result. As shown above, this becomes particularly important when we consider how much of a difference the quality of compression (or better still, lack of) has on our resulting spectrograms.

5.3.2 Sample Set Results

As resampling occurs regardless of whether an image is compressed or not, the Image Resampling detection algorithm operates on all image formats. We were therefore able to successfully test all 50 sample images, resulting in the following breakdown:

- **Unique Forged Images** - Out of a total of 20 unique forged images, 12 contained areas that were correctly identified by the algorithm as having noticeably different spectrograms to the rest of the image. This provides us with a successful detection rate of 60%.
- **Image Manipulation Dataset** - The results of the Image Manipulation Dataset faired even better, with 13 out of 20 images correctly identified by the algorithm, giving us a detection rate of 65%.

Both sets of images faired well, and our average detection rate for the Image Resampling algorithm sits at a respectable 62.5%. However, it's important to note that unlike the other two algorithms discussed, specific regions of interest must be chosen by the user. They therefore must already have a suspicion, or at the very least a general idea of where a forged area is potentially located. Whilst it's correct to state that the algorithm isn't able to search the image for forgeries, it is a very useful tool to have in order to confirm any existing suspicions.

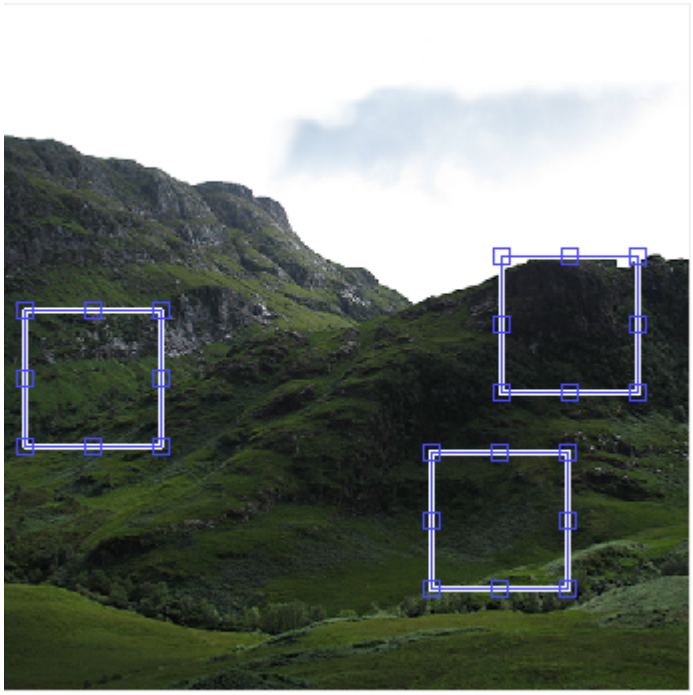


Figure 32: Forged Image (r18) - The hillside on the right is a new addition.

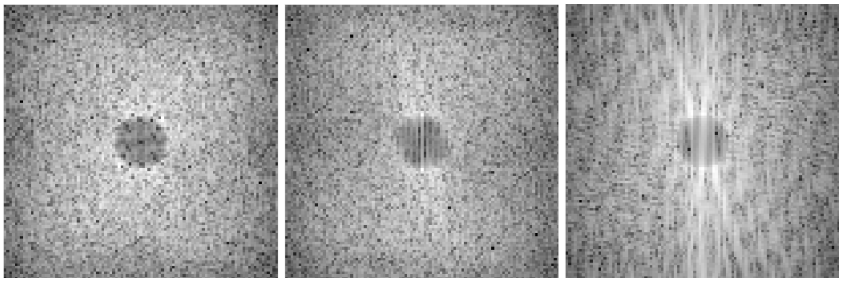


Figure 33: Here we see that the generated spectrogram for the forged ROI (right) is substantially different to the remainder of the image.

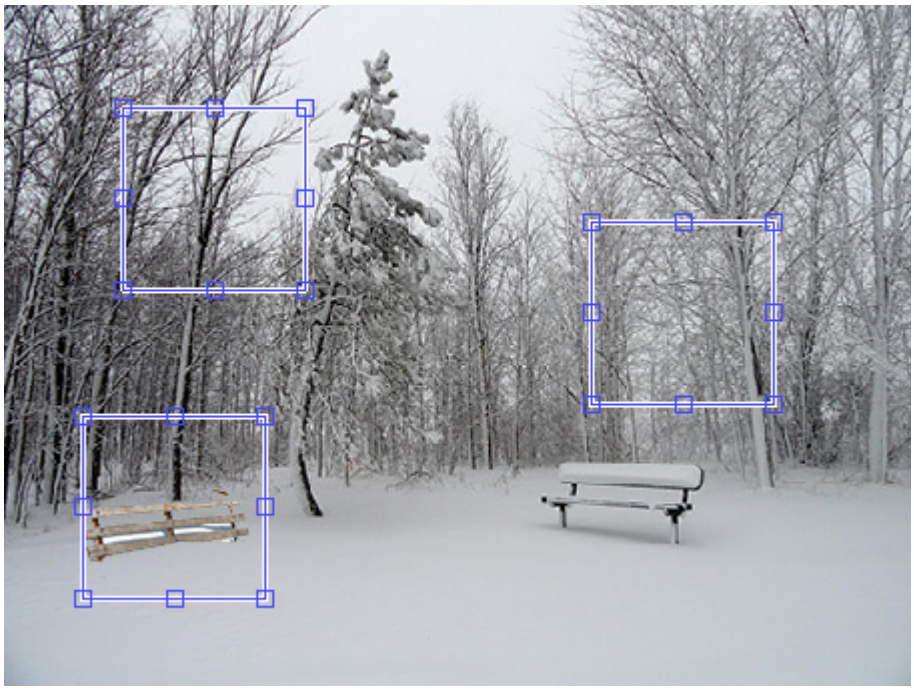


Figure 34: Forged Image (20) - The bench on the left is not part of the original image.

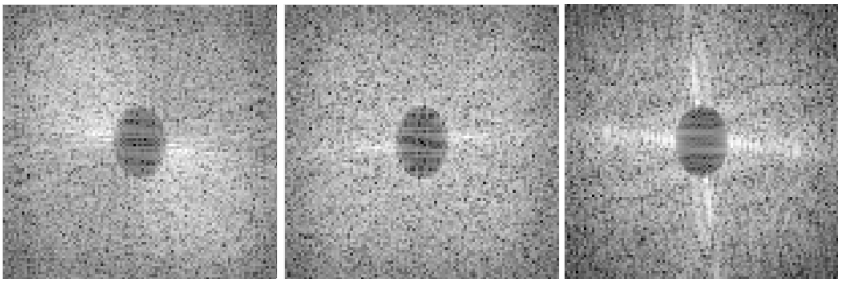


Figure 35: Again, the spectrogram for the ROI containing the bench (right) is substantially different to the rest of the image.

The rate of resampling within the forged image has a large impact on whether it is detected or not. Some images, despite highlighting a forged area as a region of interest, provided inconclusive results:

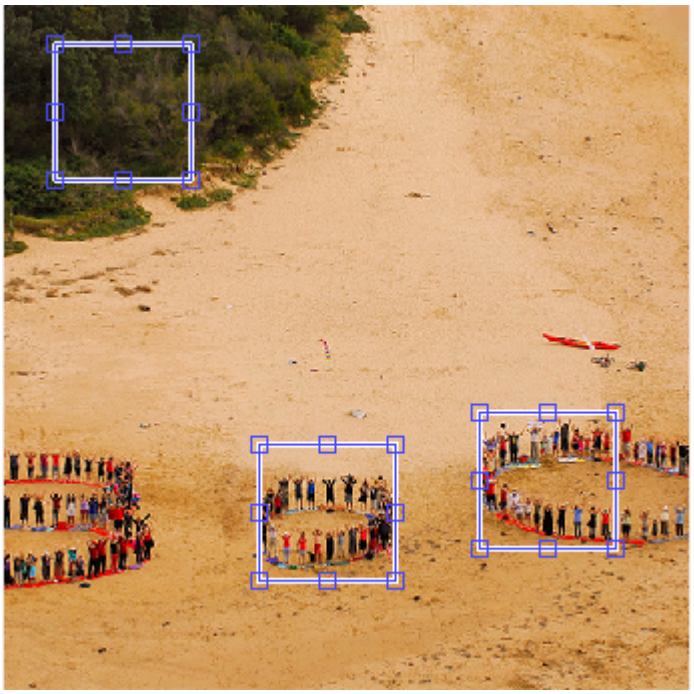


Figure 36: Forged Image (r6) - The centre group of people have been modified.

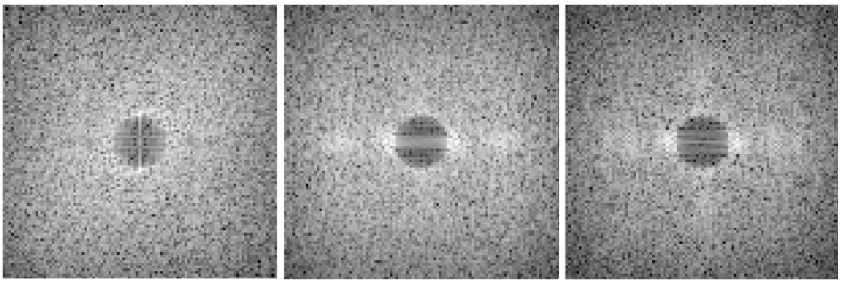


Figure 37: None of the spectrograms show any noticeable discrepancies, with the forged area looking identical to the legitimate circle of people.

False positives were difficult to calculate, however a result was deemed to be a false positive if a legitimate area looked substantially different to the two other regions of interest. The false positive rate within the set of unique forged images was 25%, and only 10% within the Image Manipulation Dataset images. The algorithm was also run on the set of 10 unmodified images, choosing three random areas as our points of interest. 33% of the images had one substantially different region of interest, giving us an overall false positive rate of 20%.

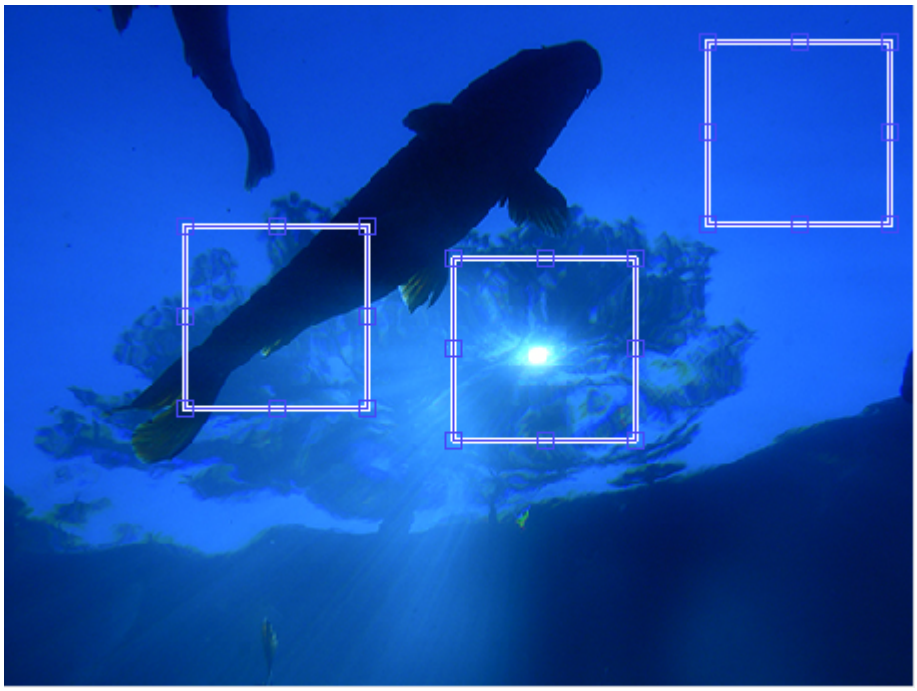


Figure 38: Forged Image (f2) - No part of this image has been modified.

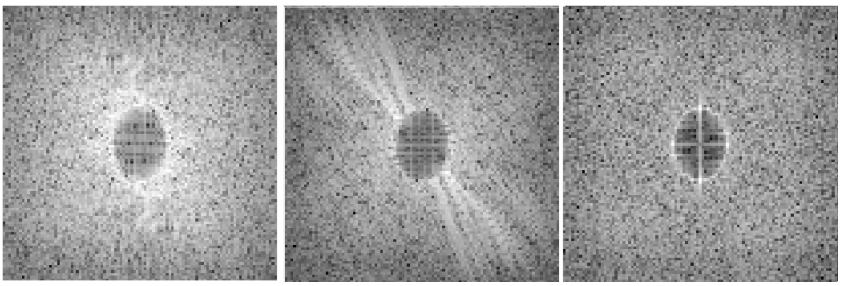


Figure 39: However, ROI 2 produces a spectrogram that is vastly different to the other two, indicating a false positive due to the drastic lighting change in the centre of the image.

5.3.3 Algorithm Performance

Due to MATLAB's extremely efficient implementation of the Fast Fourier Transform, the algorithm completes in a very short amount of time. The average runtime for the whole set of 50 images was 0.12 seconds, with the longest time being 0.17 seconds. As we can see, this is an extremely efficient algorithm, however time factored into selecting the three regions of interest must be considered

in practical use tests. Increasing the overall image resolution has a small impact on performance, a 5,000px x 5,000px image is completed in an average of 3.15 seconds. Doubling the image resolution to 10,000px x 10,000px unfortunately caused MATLAB to run out of available memory, however as this image size is fairly unrealistic this does not cause a problem in practical terms.

Another possible factor that can negatively impact the performance of the algorithm is the size of the regions of interest selected. By default, the UI selects a 100px x 100px block, however this can be adjusted by the user to be larger or smaller. Doubling the size of the region of interest to 200px x 200px increases the runtime to an average of 1.2 seconds, as doubling the size of the image squares the total number of pixels that the Fourier Transform is being computed upon.

Overall, the algorithm has high detection rates and low false positive rates. Whilst there are specific conditions, such as when an image is overly compressed, in which the algorithm doesn't perform correctly, it is able to be run on any image format as it doesn't rely on detecting specific types of compression artefacts. It also has the advantage on working on a wide range of forgery types, the more a region has been modified, the more likely that the area in question has been resampled; therefore containing traces detectable by the algorithm.

Conversely, it is unable to detect forged areas by its self; the user must either already have a general idea of where they'd like to choose as a potentially forged region, otherwise trial and error must be used in order to find these regions. However, this inconvenience is offset by the fact that the algorithm is extremely efficient with a very low runtime and computational cost. Combined with its high detection rates, this makes it an ideal algorithm to use when specific areas need to be tested for forgeries, as opposed to when a whole image needs to be scanned in order to ensure its legitimacy.

5.4 Metadata Tag Detection

Metadata tag detection was also performed on each image as a way of complimenting the results of the above algorithms. In order to ensure that the images from the Image Manipulation Dataset weren't inadvertently modified whilst cropping or re-sizing, tag detection was performed on the original resolution images as provided.

The result of each Metadata analysis, along with any relevant strings and timestamps, are included with the full list of results. However, as a rule each uniquely forged image was detected as being processed within Adobe Photoshop CS6. As expected, the cropped images from the Image Manipulation Dataset were also flagged with the same modified metadata tag. However, the original dataset images also contained a Software field, but the field was void of any information. This is unusual, as per the EXIF specification the Software field is optional and isn't usually created by default [12]. Whilst this field is also used to record camera software in addition to any editing software, the fact that the field has been created but left blank suggests that metadata has been

cleared from the images. None of the unmodified images had the Software field embedded within them and were free from all potential forgery tags.

Whilst this approach is indeed fairly rudimentary, it can serve as a useful indicator when combined with the main forgery detection methods illustrated above.

6 Future Work

This project compares the overall performance of three distinct image forgery detection algorithms. However, the scope for further research within this field is huge, as there are numerous new and emerging forgery detection techniques that can also be researched and tested. For example, work was put in to researching methods that operate on detecting anomalies within the lighting and colour levels of the image, which appeared to be an emerging yet promising new technique. However, it was found rather early on that these techniques would have taken far too much time to implement, and it was unfortunately decided that they were beyond the scope of this project in its current form. Future work would primarily be based around implementing this kind of cutting edge technology and evaluating its performance compared to the more grounded methods demonstrated within this project.

Due to time constraints one algorithm was chosen to be implemented from each group. Whilst this gives a good overview of the chosen technique, it would also be beneficial to be able to test variants, as different variations of the same technique could have potentially large differences in results and performance. For example, whilst the implemented Copy-Paste Clone Detection algorithm is widely used, many variants of the same algorithm exist, and it would be useful to be able to compare these differing algorithms. Additionally, the aim of this project was always to compare existing algorithms and techniques, however a potential future goal would also be to work on developing new, unique algorithms that could have the ability to detect image forgeries much more efficiently than existing methods.

A problem that was identified early on was that there were very few user friendly solutions to image forgery detection. A quick search mostly reveals academic papers, and as mentioned at the beginning of this report, most pieces of consumer software rely on primitive methods such as tag reading in order to "detect" forgeries. Whilst developing a completely polished, user friendly system was beyond the scope of this research project, there is a definite need for such an application in the professional world. The user interface of the implementation, whilst unpolished, is fully functional and simple enough for the average user to use without too much of a learning curve. It would be more than possible to work on a more refined UI that works to present these different image forgery methods in a polished, finished product that is both accessible and functional to a standard end user.

Additionally, whilst MATLAB proved to perform exceptionally when dealing with images, its performance suffered when faced with multiple loops and repeating operations. This is seen rather clearly in the difference in runtime performance between the Copy-Paste Clone Detection algorithm and the Image Resampling Detection algorithm. Overall however, the use of MATLAB was a huge assistance, it substantially reduced development time due to its native support for GUIs, matrices, and more complex operations such as the Fast Fourier Transform & the Discrete Cosine Transform. However now that the algorithms have been implemented successfully, it would be beneficial to investigate any

possible performance improvements from switching to a more traditional, complied language. There is also the possibility of utilising the MATLAB Compiler [14], and analyse any performance benefits.

Overall, this project has provided a good, solid framework for research involved in image forgery detection methods. However, not only is this an emerging research field, but it is increasing in importance and relevance day by day as images become more and more important in our daily lives. Building on this foundation, expanding on research of image forgery detection methods and potentially releasing a polished, user friendly application shows that this project has limitless potential, and contains very exciting prospects for the future.

7 Conclusions

This project has successfully demonstrated the strengths and weaknesses of three distinct image forgery detection algorithms, and their ability to perform on a large sample set of both unique sample sets and dataset image libraries. It was clear from the beginning of the project that no algorithm would work flawlessly in every situation, however the variety of images used has allowed the user to decide on the algorithm best suited for their needs. Out of the 40 total forged images, 32 were successfully detected as containing forgeries by at least one of the three detailed methods, providing us with an average success rate of 80% for the entire sample data. Therefore, whilst the detection rate of each algorithm varies widely depending on the specific type of forgery contained within the image, utilising all three methods provides a robust detection rate on a wide variety of forged images.

Error Level Analysis was found to be efficient on detecting additions to JPEG images, where the newly forged area has a different compression ratio to the original background. The algorithm worked best on images that were saved as high quality JPEG images; further compression reduced the ability of the algorithm to detect newly compressed areas, hence its overall detection rate of 46%. In addition, by design the algorithm will only run on JPEG images, meaning that not all sample images were suitable to be tested. Its performance was hugely promising however, averaging 0.03 seconds per image, and its false positive rate was entirely respectable at 20%.

Copy-Paste Clone Detection was successful in detecting copy-paste cloning within a variety of images. Whilst the detection rate on the unique forged image set was 29%, as previously mentioned this includes forged images that contain no duplicated regions, and therefore this is unrepresentative of the actual results of the algorithm. When only images with cloned areas are included, the success rate jumps to 80%. Results on the image manipulation dataset were slightly lower at 50%, due to the fact that re-compression and re-sampling occurred on many of the images, causing differences within the cloned areas. Each image showed at least one false positive block, however this is to be expected due to the natural occurrence of patterns in genuine images. The pixel by pixel nature of the algorithm ensured that this was the slowest of the three algorithms, however with an average runtime of 71 seconds for the entire sample data, overall runtime is still perfectly reasonable. It's also important to note that this includes a few results that took an abnormally long amount of time to compute, slightly skewing the results.

Image Resampling Detection provided a robust solution for detecting resampling distortions within the underlying frequencies of the image. Success rates were similar between both the unique forged image set and the image manipulation dataset, averaging at a successful detection rate of 62.5%. The algorithm had a low false positive rate of 20%, and performance was extremely promising, with an average runtime of only 0.12 seconds. Whilst this proved to be an efficient algorithm with good detection rates, it's important to note that three regions of interest must be chosen by the user, with one of these being a region

of potential forgery. Unlike the other two algorithms, the user must therefore have an idea of a potentially forged region, or simply use trial and error to find these potential regions.

Metadata tag detection was also utilised as a complimentary algorithm. The method was not expected to perform admirably due to its rudimentary nature, however it did correctly identify all of the unique forged images as having been modified in Photoshop. The collection of untouched images were also correctly identified as legitimate, as they contained no potential forgery flags. However, the Image Manipulation Dataset images contained a blank Software field, suggesting that EXIF tag removal had taken place at some point in an attempt to remove any potentially revealing flags.

Whilst the overall success rate of the chosen algorithms were very promising, we've also learnt that new methods are constantly being developed in order to provide better results and improved performance. It is therefore appreciated that further work will be required in order to fully investigate a wider range of forgery detection methods. Research in to this new and exciting field is becoming more and more important, as determining the trustworthiness of images becomes a wider issue in modern society. This project provides a sound framework for additional tests to be carried out on an even wider range of algorithms in the future.

8 Reflection

This project has played a huge role in developing my progress not only academically, but also having an impact on the way that I plan and structure large tasks that I am faced with. Admittedly, the sheer scale of this project was not only daunting at first, but the thought of independently working in order to achieve something both unique and genuinely insightful was unlike anything that I had encountered throughout my academic studies. Whilst our second year contained a large group project, much more guidance was given, time constraints were far less and tasks were delegated between a large group of people. Whilst I've had the terrific support of my supervisor throughout this project, the onus has very much been on me from start to finish to not only achieve my goals, but also to come up with the goals themselves to begin with. Beyond being given the title of the project and a short description, a blank canvas was presented; and I am extremely proud of how I have seen the project through from a simple concept to a fully fledged research study.

I feel one of the most important skills that I have developed throughout this project is a sense of timekeeping. Whilst I have had to deal with deadlines and general timekeeping throughout my academic career, being sorely responsible for a project this size was a completely new experience to me. A general plan was outlined within the initial report, and whilst I followed this schedule as much as possible, it was inevitable that certain aspects took much longer to complete than first thought. I've learnt that even the best plans will falter at some point, and that the most important aspect is learning how to deal with these unexpected turns. For example, my original plan was to implement one algorithm from each of the five different groups, as detailed in the original report. However it became fairly obvious that there simply wouldn't be enough time to implement and comprehensively test five different algorithms. Whilst I could have stuck rigorously to the original plan, this would have had a detrimental effect on the testing and results phase of the project. I therefore decided that it would be much more beneficial to comprehensively focus on three algorithms instead, given the time constraints given. What's important is that I understand that this is no failure on my part, these types of scenarios happen regularly when dealing with large projects. Plans will never be perfect, and being able to foresee potential problems and re-structure the project accordingly is a very important skill to have.

In hindsight, I feel that I could have managed my time with the report better. Whilst I was sure to give myself ample time and completed the report well within the time frame specified, I waited until after the all implementations were complete before starting the write-up. Many of the sections could have been completed as the implementation was being developed; in fact this would have been to my advantage as I often found myself having to look back through my code in order to explain some concepts, especially for the implementation sections. It would have made a lot more sense for me to write the design & implementation sections whilst the actual system was being designed and implemented. The most important lesson that I have learnt in regards to

timekeeping is to never put off until tomorrow what can be done today; working at a constant pace is far more beneficial in the long term than completing large chunks of tasks in a short time frame.

Another completely new concept has been the process of professionally referencing academic work. Within my studies, coursework has been an independent assessment, generally no outside work is permitted, whether referenced or not. Whilst this indeed makes sense in terms of assessing ones understanding of concepts, it is not how the further academic field or the professional world operates. It is far more efficient, both in time and manpower, to re-use code where appropriate and to base ideas off of already existing concepts. This also holds true for statements made within the report, anything suggested as a fact that isn't common knowledge must be referenced appropriately in order to maintain academic integrity within the report. This makes a lot of sense, however I found myself having to go back over sections in order to attempt to find the article or paper that I referenced that statement or concept from. Getting used to academic writing is all about changing how you think, once you get in to the mindset of justifying and proving everything, it becomes a lot easier and almost second nature to do so. In the future, I'll definitely reference as I write, as I feel this ensures that all work is credited where due and reduces the workload required at the end of the write-up. Professionally referencing within this academic report has certainly changed the way that I think when writing, and has allowed me to appreciate the importance of citing relevant work.

9 References

[1] Murali, S, Chittapur, GB, Prabhakara, SH, Anami, BS. 2012. Comparison and Analysis of Photo Image Forgery Detection Techniques, http://airccse.org/journal/ijcsa/papers/2612ijcsa05.pdf.

[2] 2014. JPEGSnoop Application, http://sourceforge.net/projects/jpegsnoop/.

[3] 2013. Image Forgery Detector Application, http://sourceforge.net/projects/ifdetector/.

[4] 2013. Digital Image Forgery Detector Application, http://sourceforge.net/projects/difdetector/.

[5] The Pigeonhole Principle, https://www.math.ust.hk/ mabfchen/Math391I/Pigeonhole.pdf.

[6] Belkasoft Research. 2013. Detecting Forged (Altered) Images, http://articles.forensicfocus.com/2013/08/22/detecting-forged-altered-images/.

[7] Ansari, MD, Ghrera, SP, Tyag, V. 2014. Pixel-Based Image Forgery Detection: A Review, http://www.tandfonline.com/doi/pdf/10.1080/09747338.2014.921415.

[8] Fridrich, J, Soukal, D, Lukas, J. Detection of Copy-Move Forgery in Digital Images, http://www.ws.binghamton.edu/fridrich/Research/copymove.pdf.

[9] Sturak, JR. 2004. Forensic Analysis of Digital Image Tampering, http://www.dtic.mil/dtic/tr/fulltext/u2/a430512.pdf.

[10] morgueFile Image Licensing, http://www.morguefile.com/license/full

[11] Marshall, D. 2014. CM3106 Multimedia Lecture Notes, http://www.cs.cf.ac.uk/Dave/Multimedia/PDF/02_Filters_and_Fourier_Transforms.pdf

[12] 2002. EXIF Specification. http://www.exiv2.org/Exif2-2.PDF

[13] Riess, C, Jordan, J. Image Manipulation Dataset. https://www5.cs.fau.de/research/data/image-manipulation/

[14] MathWorks. MATLAB Compiler. http://uk.mathworks.com/products/compiler/

[15] Pontinen, P. Study on Chromatic Aberration of Two Fisheye Lenses.

http://www.isprs.org/proceedings/XXXVII/congress/3_pdf/05.pdf

[16] W3Techs. 2015. Usage of image file formats for websites.
http://w3techs.com/technologies/overview/image_format/all

[17] BBC News. 2015. Fake Germanwings pictures circulate online.
http://www.bbc.co.uk/news/blogs-trending-32037008

10 Bibliography

Wandji1, DW, Xingming, X, Kue, MF. Detection of copy-move forgery in digital images based on DCT,
http://arxiv.org/ftp/arxiv/papers/1308/1308.5661.pdf.

Paganini, P. 2013. Photo Forensics: Detect Photoshop Manipulation with Error Level Analysis,
http://resources.infosecinstitute.com/error-level-analysis-detect-image-manipulation/.

Farid, H. 2009. Image Forgery Detection - A survey
http://www.cs.dartmouth.edu/farid/downloads/publications/spm09.pdf