RAWFlash: A cloud based RAW image editor

Student Name: Ryan Collins

Supervisor Name: Dr Tom Friedetzky

Submitted as part of the degree of [MEng Computer Science] to the

Board of Examiners in the Department of Computer Sciences, Durham University

April 16, 2018

Abstract —

Context/Background RAW images are used by photographers to improve the quality of their images, specifically when editing. With JPEG files, the loss of data from compression leads to poorer quality edits, while using RAW results in a higher dynamic range. The editing of these files is traditionally done on a high-spec machine, capable of processing images with large resolutions. In the field, this can be difficult. When used for websites, the standard process involves editing a RAW file locally and exporting it, uploading the exported image to a Content Management System (CMS). Here, the CMS will then edit the file again, resizing the image to ensure it displays properly on the browser.

Aims The main aim of this project is to test the feasibility of a Cloud-based RAW image editor, allowing image adjustments such as exposure adjustment, colour adjustment, and overall improvements to the quality of the image, all within a cloud application rather than on the desktop.

Method A render server back-end will first be implemented as an API, taking in an input as a JSON object, and then processing the image, and then a JavaScript client shall be created to interface with this API.

Results

Keywords — RAW image editing, dcraw, cloud image editing, docker, image processing

I INTRODUCTION

The project itself concerns itself with editing RAW files through a system deployable to the cloud so that it can both be used by people editing photos, as well as by extension by other systems. While RAW editing isn't new, the creation of a system to edit RAW files would give users more control in content management, and produce better quality images.

A RAW Files Explained

RAW isn't a file format in itself, but rather an umbrella term for a set of many different formats that store the raw camera sensor data. There are a set of different proprietary file formats, that all store similar information, but in different ways.

When an image is captured, the camera sensor uses charge buildup to represent the amount of light that is incident on the sensor (using a phenomenon known as the photoelectric effect). Colour data itself isn't stored at all in the RAW image. Colour is achieved by putting a filter over the sensor, such that each individual part of the sensor captures only red, green or blue light, which then allows one to build up a full colour image. The process of creating a full colour image from the camera sensor is called demosaicing.

There are several different steps to decoding RAW images, in addition to demosaicing which was explained above. These are:

White Balance

Gamma Correction Cameras typically represent colour changes linearly, with a gamma of 1.0. **TODO FIND SOURCE OTHER THAN CURRENt**. However, this isn't necessarily pleasing to the eye, so this can be changed to change how tones are reflected. For example, this can

Sharpening/Noise Reduction Sometimes, noise can creep into the image, particularly if the image itself is shot with large ISO levels (ISO is a sensitivity setting that can be set when taking a photo, to allow the camera sensor to be more sensitive to light **TODO FIND SOURCE FOR ISO DEFINITION**). This can be corrected during the editing process.

Also, sometimes edges might need to be enhanced to bring out the detail in the image. This can again be used for stylistic purposes, but this is ultimately the decision of the image editor.

A.1 Comparison to JPEG

JPEG, a common format in the consumer photography market, often has smaller files due to JPEG compression. While this can be good to minimize the amount of memory used, the lossy compression is problematic as adjusting tone and colour information as JPEG heavily compresses colours (so minor adjustments in colour will appear as the same colour in JPEG). When shooting an image with JPEG, the camera uses a built-in RAW processor to automatically process the RAW data, perhaps using some settings that can be adjusted in the camera, and compresses the image down into a JPEG format.

On the other hand, RAW allows one to have far more control over the final image, by adjusting various settings with a far greater freedom and by using the camera's built in processing.

(Fraser 2004)

B Applications of RAW Image Editing

B.1 Content Management

When using Content Management Systems (CMS) with websites, some element of image processing is usually carried out.

When working with better quality RAW images, usually these RAW files are edited locally, exported in some format, and then uploaded to the Content Management System. From here, this exported image is then processed and various images are produced, typically with different sizes and sometimes different focal points (with the image cropped around certain objects/locations).

However, a RAW based image editor would allow one to make this process more concise by uploading the RAW file, editing it from within the Content Management System, and using the RAW file and defined operations (along with the size instructions supplied from the various templates/webpages) to produce individual images for each page on the website. This removes the need for purchasing expensive RAW editing software, allowing the user to focus on creating excellent content, rather than spending time producing outstanding images with potentially complex software.

C Problems with Current Implementations

Most current implementations exist as native applications, designed specifically for one or several different platforms. With this, a specific machine shall have the program installed, and shall contain all the features from managing files, to rendering images, to displaying the output to the user as required.

However, larger images can require a larger amount of resources to render properly, and furthermore if one intends on editing RAW images on the go, transporting a larger machine isn't convenient. Furthermore, the RAW editing software might not necessarily be available on the platform that is nearest to the user.

Instead of this approach, this project is taking an approach of having a Cloud system that can render images, which can scale if necessary by adding more servers, so whatever client device is used, raw images can be edited providing the user has an internet connection (which with the recent presence of 3G and 4G, they are likely to have).

Furthermore, due to the proprietary nature of most RAW processing applications, using them as a component in another system becomes very difficult, and by creating a system that can be controlled by other external systems can provide RAW image editing as a service.

D Project Objectives

The project objectives can be expressed as Minimum, Intermediate and Advanced Objectives

Minimum Objectives

- Exposure adjustment
- Noise reduction methods (Gaussian, mean)
- Web Interface interacting with an image processing server
- Non-destructive image adjustment (i.e. no reduction in quality over time)

Intermediate Objectives

- Load DNG RAW files by upload
- White Balance Adjustment
- Gamma Correction
- Modern, user friendly User Experience
- Cropping, Rotating and Exporting to other formats

Advanced Objectives

- Addressing potential scalability issues
- Haze removal
- Sharpening

II RELATED WORK

A RAW files, and parsing

As explained in the introduction, there are many different types of proprietary RAW file formats, such as Canon's CRW and CR2, and Nikon's NEF among others. Each of these formats stores similar information, but in a different way.

Adobe created and made public their standard RAW file format called DNG. It is defined as "a non-proprietary file format for storing camera raw files that can be used by a wide range of hardware and software vendors" by Adobe in the specification. (Ado 2012)

TODO: ADD DNG SPEC TO BIBTEX FILE

B Existing Solutions

Adobe Lightroom Adobe Lightroom is a native and proprietary tool, developed by Adobe, for editing images, both RAW images, and otherwise. RAW processing in Lightroom is done by Adobe Camera RAW (as Lightroom builds on this useful tool),

RawTherapee **TODO: Complete this section**

Darktable **TODO: Complete this section**

C

III SOLUTION

A Client-Server Communication Protocol

In order to communicate between the client side interface (where the user provides inputs and instructions for rendering images) to the server side (the entrance of the rendering system, which manages jobs), we need a method for transmitting the correct information in an appropriate manner.

A.1 Communication Methods

Representational State Transfer (REST) The REST method uses URL routes to send/retrieve information, using HTTP methods such as GET, POST, PUT, DELETE.

Web Socket Web Socket is a new technology that allows easy two-way communication between client and server, rather like how sockets work within normal network programming. The client would send a message to the server with the required render instructions, and when the job is complete, the server would return the result to the user. Listeners on both sides wait for messages to be received, and process the messages in the appropriate manner.

This lends itself quite well to our method, as with the user constantly changing settings and requesting a render (to view the preview), and repreating the process as settings are adjusted, two way communication seems far more approriate for this application.

However, Web Socket does have downsides. As a fairly new technology, it's not universally supported, and only modern browsers support the protocol, and potentially some firewalls might block ports required, and some hosting providers might not support it either. Therefore, while websocket is ideal, compatibility might be a problem.

Socket.IO Socket.io itself isn't really a protocol, but an implementation of both REST and Websocket, designed for event driven applications. Where websocket is supported, it is used, but where websocket isn't supported, the system falls back to REST to transmit and receive data through polling. Implementation wise, the system is built in an event driven way, with message handlers being written, and the framework deals with the edge cases with compatibility.

However, a problem arises with this, as this is only fully supported in a limited number of languages, namely JavaScript (both client and server side), and unofficial support is available for Java. Therefore, JavaScript shall be used for the client side (which is really the only option anyway), and Java for the server side, due to the Image library provided.

Futhermore, the added benefit is one of state: in the event the server goes down temporarily, providing the session is still open (i.e. the user hasn't closed their browser window), the system will reconnect when the server comes back online. Providing we implement some stateless system (where all the data needed is provided within the request), then this will help improve the reliability of our system, not requiring users to repeatedly restart their session if some error occurs in the dispatcher.

Unlike REST, and much like websocket, we have the added complexity of identifying users, so that data is sent back to the correct user. This can easily be done by finding a client unique ID, and sending a message to the connection with that ID. This is an additional piece of information that needs to be included in the request between the dispatcher and the render server (to ensure that when the response is received, the response itself is forwarded to the correct person).

Therefore, this is the solution I've decided to use.

B Rendering Server Structure

The final structure of the render server can be seen in Figure B.

When designing this section, several different structural methods were considered. Initially, a local cache wasn't implemented, but this was added to improve performance (as discussed in Part D).

Initially, the aim was just to have one single monolithic structure to the image processing, with the same structure dealing with the entire process, from reading in RAW to returning the result. However, the decision was taken to implement these seperately, as it would allow for the ability to swap the RAW processor used. This could be useful if for example support is dropped for dcraw, or if a better RAW processing library is created (perhaps one that doesn't require file writes). The DCRAW Manager therefore deals with the Dcraw processing, by interfacing with the custom JDCRAW library that was created (but open sourced as a separate module). The creation of this module further abstracts DCRAW from our system.

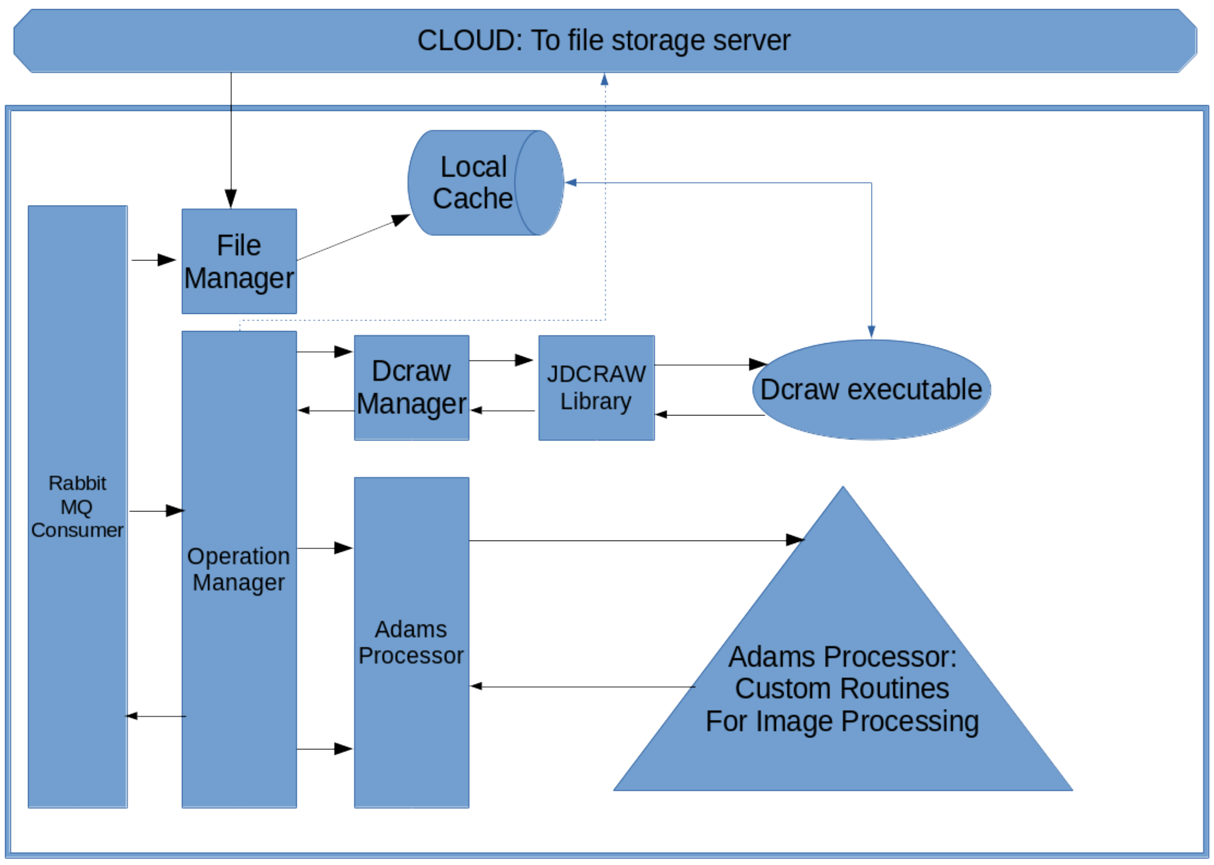

Figure 1: The structure of the render server. The local cache is simply stored on the server local filesystem, while the RabbitMQ Consumer takes requests from the message queue and processes them, returning the result.

The entire image editing process however is managed by one class, the Operation Manager. This is responsible for taking the user specified instructions and passing them to all the possible processors (currently only DCRAW and Adams are implemented, but potentially many more processors could easily be added, perhaps using class loading in the future).

More detail about the Adams Processor, whcih is the name for all the custom image processing implementation can be found in Part C.

C Adams Processor

C.1 Gamma Correction

The process of gamma correction involves taking the RAW image, which by default has a linear tone curve (gamma of 1.0), and mapping the colours to a different tone curve.

The function f is used as a transformation from the image, to the gamma correction image. f is defined as:

$$f(\gamma,v)=255\cdot(\frac{v}{255})^{\gamma}$$

In the above definition, v is the brightness value of one cell of the image, where $0<=v<=255$.

On the implementation level, the best approach is to use a Lookup Table, with an input of the RAW image components (in our case, Red, Green and Blue), and an output of the adjusted component values in accordance with the gamma correction function. The benefit of using a lookup table, rather than calculating the gamma value for every single possible value, is that one doesn't need to calculate the same piece of data more than once (as bright regions, or dark regions might potentially have the same input values).

TODO: DIAGRAM SHOWING WORKFLOW OF THE LUT VS CALCUATING FUNCTION

C.2 Noise Reduction

Within an image, noise can be introduced by various means, including excessive ISO settings. Therefore are several different types of noise, including Amplifier Noise, Salt-and-pepper noise, Shot noise, Quantization noise, on-isotropic noise, speckle noise and periodic noise. (Hambal et al. 2017)

Amplifier Noise This noise typically occurs with different sensitivities for different channels (e.g. red, green and blue detectors have different sensitivities). (Hambal et al. 2017) This can be removed by applying a Gaussian Blur. (Gonzalez & Woods 2009)

Salt-and-Pepper Noise This noise is random noise, where random pixels might differ greatly from their neighbours. This noise can be caused by dead pixels on the camera sensor. (Hambal et al. 2017)

Additive Noise Spatial Filtering (Gonzalez & Woods 2009, p. 231)

C.3 Unsharp Image Adjustment

Unsharp Image Adjustment (masking), is one way of sharpening images

However, doing this alone yields images that are sharp, but also lose information (due to clipping). Furthermore, the colours and brightness are just too high.

The solution is to normalize the image, by reducing the luminance down to values that can be represented in the image).

Using standard Contrast Stretching does work, but at times tonal curves can look weird. By manually applying gamma though, this problem can be solved.

One major problem regarding the Gaussian filter is that the user can potentially choose a large Kernel size, which results in increasingly lengthy periods to process. The alternative is to associate the SIGMA value of the Gaussian with the radius, detailing the standard deviation (spread) of the Gaussian over a fixed kernel. The Mean blur filter remains as before (though this feature is disabled through the web interface, as Gaussian is primarily used).

One major problem by using RGB images only for the Unsharp Mask is that colours can be warped, as well as the requirement to carry out the entire process on each individual channel. The alternative is to convert the entire image to HSV and carry out Gaussian on the Lightness channel.

C.4 White Balance Adjustment

Each colour image is made up of a variety of individual images, which are placed together to produce the colour images. These are the Red, Green and Blue masks.

When one adjusts colours, our aim is to mix these three images together with a variety of different amounts, so that the overall image produced blends them together to create white (rather than an orange). The white balance control here can be adjusted further to tint the entire image.

D Improving performance of fetching RAW files

One of the biggest sources of delay in our system is the necessity to obtain RAW files. The user provides a URL pointing to the RAW file in their render request, and then the system downloads the file from that URL to use as the RAW file. Doing this for the same image constantly is very time consuming, and as the user will likely be making many edits on the same RAW file, many downloads will be needed.

Therefore, implementing some form of caching is needed, to ensure that this download step only occurs once, and then afterwards the cached RAW file will be used. Caching works in this situation, as when rendering, the RAW file will not change at all (it's read only from our point of view), meaning that we obtain the maximum quality through edits.

In order to do this, we need to save the image to the render server local disk, without any filename collisions (where two different RAW URLs are cached to the same filename on disk).

The solution to this is to use a Universally Unique Identifier (UUID).

Our method uses the RAW file URL as the basis for the UUID generation, generating a Type 3 UUID based on the full path specified. When downloading a file, the system first generates this UUID filename, and checks the local filesystem. If the file is present, then it is already cached, and therefore the cached version is used. If the file isn't present, then it isn't cached, and a file is downloaded, and written to a filename based on the UUID (in this case, the format will be UUID followed by the extention). Type 3 UUID generation is based on the MD5 hashing algorithm, by generating a 128 bit UUID. The hexadecimal representation (ignoring the dashes), is used as the replacement filename.

For a large number of RAW files being processed, using Type 5 UUIDs might be more suitable (as the algorithm used is SHA1 with a larger entropy), but for a prototype use with few image URLs, this is more suitable.

One other concern is the possibility of deleting files. While this is a grave concern typically, the current system doesn't have a time decay of caching, simply because RAW files might be worked on over a period of days, and furhtermore as the system is stateless, all files stored on the local system shall be deleted regularly (configured inside Docker). Restarting render servers regularly is ideal, as this way we prevent memory leaks from the Java BufferedImage implementation which have been minimized in our implementation but not completely eradicated.

E Web Editor Functionality

The web editor itself provides an interface to the RAW Editing Service. Rather than manually adjusting the settings and sending requests to the service, the parameters shall be controllable through a Graphical User Interface, with the image result also being displayed on the screen.

Furthermore, one must be able to undo and redo particular tasks. This is because the RAW editing process typically requires refining parameter values, testing, and then opting to use the previously set value. Having an undo/redo system would make this easier.

E.1 Undo/Redo Functionality

Due to our implementation, coupling that with the nature of our system, no current undo/redo queue library seemed to be quite an ideal fit for our system. Therefore, it's necessary to build our own.

The system should, on modification of any parameters, store the change in some data structure, so that it can be reverted.

This entire system relies on using a stack. When a change is made (one change shall be made at a time), both the name of the parameter edited shall be stored, along with the old and new values. This shall be stored together in an object, and pushed to the stack. To undo, one simply needs to pop the stack, and set the specified parameter (given by the parameter name) value to the old value specified.

In order to enable redo functionality, we create an empty stack, called the redo stack. When the undo function is called, the object popped from the stack is also pushed to the redo stack. If the redo queue is not empty, and a normal change in parameter has occurred, then the redo queue is emptied (one can no longer redo).

There are also a few edge cases. If someone presses the undo button when there are no actions to undo, then nothing can happen, as there is nothing to undo. The same is true for the redo button.

F Storing User Images

While one can edit individual images by simply supplying them to the editor, they need to be accessed somehow, preferably through supplying some URL, where the RAW image can be downloaded, and then used to edit an image. The system shall be built to be as customizable as possible, by downloading images from a supplied URL in the request, and this RAW image is downloaded (or cached to the render server), wherever the file itself is hosted. This way, we can either supply a system for uploading/storing images, or the user can choose to use an image store elsewhere such as Amazon S3, OpenStack Swift, or other storage backend. Our system simply uses the Django file system, which stores the file in an uploads folder on disk.

G Managing a big system

As our system design calls for a large number of individual components, managing all of these together can be quite complex.

One option is to write a script that deals with the setup of each individual component: web server, web services, render dispatcher, render servers (however many we need), and message queue. However, if a component of the system stops working, that'll bring down the system entirely, and writing a script won't necessarily allow us to deploy the system across several different machines.

Our solution to this problem is to use Docker. Each component in the system shall be given its own individual container (essentially a virtual machine), where it is sandboxed to ensure that systems don't interfere with one another, and each container is networked together. Rather than manually setting up the networking, one can use Docker Compose to automatically deal with the links between different containers, to ensure that private links between containers are set up, but these links can't be accessed from outside the system. This method is preferable, as then one cannot access the message queue/other sensitive components directly. That way, only by sending the proper request, will the system respond with an output.

TODO: INSERT IMAGE OF ARCHITECTURE HERE TODO: DOCKER SOURCE

H Unit Testing Issues with Image Processing

For one to compare the output of the system with other RAW editors is not necessarily as easy as bitwise ANDing the files, and finding the percentage differences. While this might work if all systems use the same algorithms to do the same tasks, due to the proprietary nature of RAW image editing, different implementations of demosaicing exist, yielding slightly different images, along with potential different ways of adopting equalization, gamma correction and various other colour adjustments (for example, we use RGB gain to adjust the colours, while LightRoom instead uses Chromacity). It'll be difficult to use the same settings for every single system. Instead, it might be better to edit a variety of different images, and compare them by eye, to determine whether the algorithm itself is working correctly, and using base cases to test the output of a system by eye.

Base cases include:

Using a gamma of zero will yield a completely white image. Uisng a gamma of 1 will yield the input image, for gamma correction. Setting red gain to 1, and all others to zero will yield an image with only the red channel. Doing the same for green and blue should reveal the same. This will ensure the algoritm is working correctly.

Exposure zero should yield a completely black image. Using a very high exposure should yield a completely, or nearly white image.

We can test undo and redo trivially.

IV RESULTS

A Comparison With Other Editors

Testing the system against Lightroom, Darktable and RawTherapee, testing the image output compared to these, and also performance based tests compared with traditional native software.

TODO: Mention specs of the machine.

B Functionality Testing

As explained previously, the functionality of the system can be tested by carrying out a set of boundary tests.

C User Testing

Get User Comments on system

D

V EVALUATION

A Application to Web Content Management Systems

B Benefits of Portable Image Editing

C Performance Issues and Improvements

C.1 Problems with Java Image library

As a basis for the Adams Processor image processing, the Java built-in Image library was utilised, by using ImageIO to load an image into a BufferedImage object, and then applying various BufferedImage operations on the image.

Java has a built in library which automatically implements Convolution, resizing, and various other routines. While this is useful, several problems were encounted when using this routines.

High Memory Usage/Memory Leaks When processing images, high memory usage was often encounted, at once point using 6GB of RAM for processing one RAW image taken from a Nikon D7100. After the image had finished being processed, this memory appeared not to have been deallocated, which led to high memory usage for our entire system. In Java, Garbage Collection should automatically deallocate this, but for some reason it didn't function. As a fix, garbage collection was called manually after the response had been sent, which improved performance.

Non-functioning Operations While implementing Gamma Correction, the plan was to utilize Java's LUTOp, to create a lookup table (based on the gamma value), and apply it to the image. When implementing this however, the image itself would not output any applied lookup, as the original image would be output. The LUT generated the correct LUT, and this was passed to the correct routine, but for some reason it was never applied to the image. Looking through documentation and examples didn't assist with this, as examples that were supposed to work didn't.

Eventually, this implementation approach was abandoned, and replaced with a custom approach of iterating through the image and applying the LUT manually. While slower (not relying on Java's faster implementation of yielding RGB values), it solved the problem and only appeared marginally slower.

Lack of support for Multithreading Initially, the plan was to apply multithreading to Convolution operations, but this didn't end up getting implemented due to the overhead of BufferedImage. As BufferedImage allows for multiple representations of images, it does not allow access directly to a primitive datastructure storing the image, or some way of yielding one directly. The naive way by passing Bufferedimage as a pointer didn't work either, as the overhead caused the system to run much slower than without parallellization.

While multithreading itself isn't always wanted on image processing (with web servers, each request is assigned a process, and therefore if each process generated several threads it could cause problems), for our architecture, multithreaded processing is suitable and encouraged, as there would be a performance increase.

In the future, building a custom Image datastructure would be ideal, as this would allow for multithreaded processing of images.

VI CONCLUSIONS

This section summarises the main points of this paper. Do not replicate the abstract as the conclusion. A conclusion might elaborate on the importance of the work or suggest applications and extensions. This section should be no more than 1 page in length.

The page lengths given for each section are indicative and will vary from project to project but should not exceed the upper limit. A summary is shown in Table 1.

References

Ado (2012), *Digital Negative (DNG) Specification.* Version 1.4.0.0.

Fraser, B. (2004), White paper: Understanding digital raw capture, Technical report, Adobe Systems Incorporated, 345 Park Avenue, San Jose, CA, United States. Accessed 10th April 2018.

Gonzalez, R. C. & Woods, R. E. (2009), *Digital Image Processing*, 2 edn, Prentice Hall.

Table 1: SUMMARY OF PAGE LENGTHS FOR SECTIONS

	Section	**Number of Pages**
I.	Introduction	2–3
II.	Related Work	2–3
III.	Solution	4–7
IV.	Results	2–3
V.	Evaluation	1-2
VI.	Conclusions	1

Hambal, A., Pei, Z. & Ishabailu, F. L. (2017), 'Image noise reduction and filtering techniques', *International Journal of Science and Research* **6**(2), 201–213.