University of the Witwatersrand

# Parallel Computing: Image noise filtering project

Liam Pulles - 855442
Jadon Manilall - 815050
Nivek Ranjith - 802119
Tumbone Asukile - 499426

October 2, 2016

## 1 Introduction

Visual Information transmitted in the form of digital images is a major method of communication in the modern age, but the image is often corrupted with noise. The image that is received needs to be restored to remove the noise before it can be used in applications. The method of restoration involves the processing of the image data to produce an image that is more discernible and of a perceptively higher quality.

### 1.1 Background Theory

Digital images are of vital importance in the fields of research, technology, information systems and medical science. When an Image is captured various forms of degradation may occur such as blurring or noise due to factors such as electronic and photometric conditions. Noise in an image is caused by unwanted signal that interferes with the original scene or by an optical system that is out of focus. Image denoising refers to the pre-processing step that aims to restore much of the degradation in an image before any further image processing is carried out.


Figure 1.1: An example of an image that gone through noise addition and then denoising.

Types of Noise

1. Gaussian Noise

   Gaussian noise is evenly distributed over the signal. It is statistical noise having a probability density function equal to the normal distribution. This means that each pixel in the noisy image is the sum of the true pixel value and a random Gaussian distributed noise value.

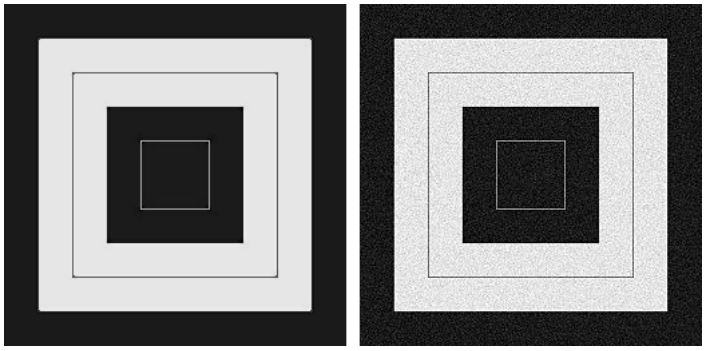


Figure 1.2: An example of Gaussian noise.

2. Salt and Pepper Noise

   Salt and Pepper is an impulse type of noise and is also referred to as intensity spikes. It presents itself as sparsely occurring white and black pixels. It is generally caused due to errors in data transmission. The corrupted pixels are set alternatively to either a minimum or to a maximum value.

Figure 1.3: An example of Salt and Pepper noise.

3. Speckle Noise

   Speckle noise is multiplicative noise. The vast majority of surfaces, synthetic or natural, are extremely rough on the scale of the wavelength. This type of noise occurs in images obtained from these surfaces by coherent imaging systems such as laser, synthetic aperture radar (SAR), and ultrasound imagery.

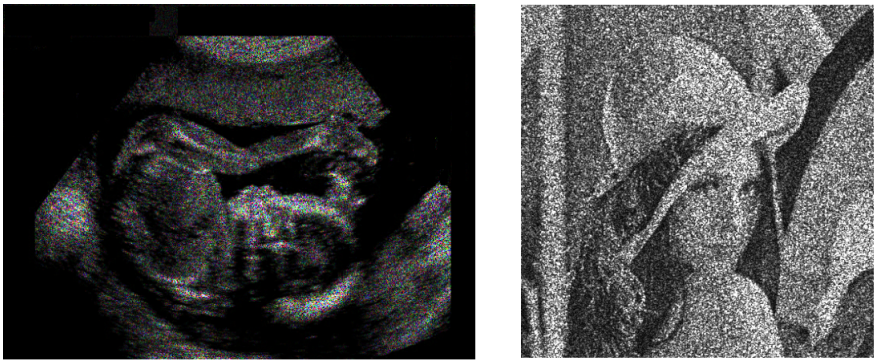


Figure 1.4: An example of Speckle noise.

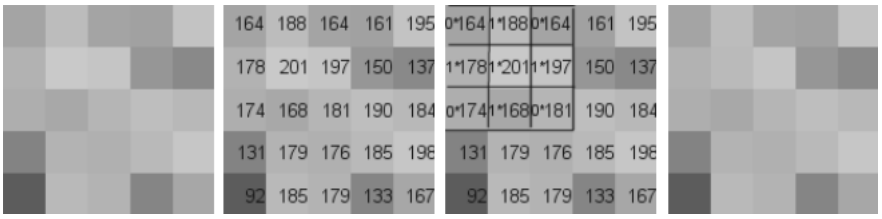
## 1.2 Problem statement

We have set out to restore an image with Gaussian Noise.

A method called Image Convolution is used to de-noise the image. It involves applying a Gaussian filter on an image to smoothen it, and this reduces the intensity variation between adjacent pixels.

This is accomplished by means of convolution between a kernel and an image to achieve the effects such as blurring, sharpening, embossing and edge detection. Convolution works by determining the value of a central pixel by adding the weighted values of all its neighbors together. This process is applied to all pixels in the image and the result obtained is the new modified filtered image.

For example:




In this example we apply a convolution to the pixel value (201). This is shown below:

$$\begin{bmatrix} 164 & 188 & 164 \\ 178 & 201 & 197 \\ 174 & 168 & 181 \end{bmatrix} * \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \frac{932}{5}$$

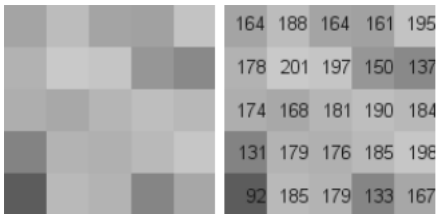
New pixel value.

Divide by the sum of the kernel elements.

With a growing demand for digital images with higher resolution this method of filtering becomes extremely expensive with regards to the time taken to execute the convolutions. This report presents comparative analysis solutions of both serial and parallel implementations of restoring an image containing Gaussian Noise.

## 1.3 Proposed solution overview

The first step will be to convert the image into a matrix representation of its pixel color values.




During the calculations the edge extrapolation function will be used where a pixel outside the range of the image is needed. This process will add a border of extended pixels to the edge of our image and thus enable us to be able to apply our kernel matrix to the edge pixels of our original image.

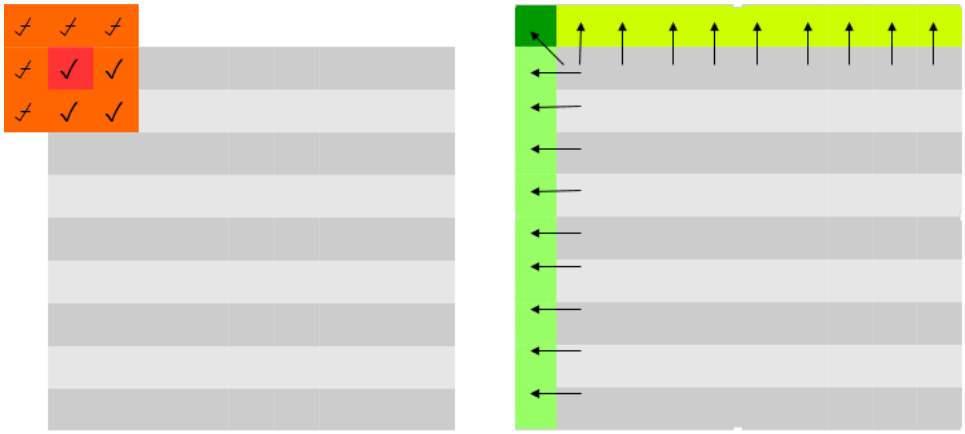


Figure 1.5: Edge extrapolation demonstration.

Two versions of implementing Spatial Domain Filtering will be carried out for analysis. A serial implementation and a parallel implementation.

The solution will explore different kernel settings within the two methods of implementation in an attempt to improve computation time and resulting image quality.

# 2 Solution technicalities

## 2.1 Serial Implementation

The serial implementation works as follows:

1. An image is loaded into memory (a pointer to BMP).
2. The matrix structure is created (we've made a few standard 5x5 and 3x3 matrices).
3. The convolution function is called.
4. The convolution function makes a blank output BMP* structure with the same dimensions as the source image for storing the calculated pixels.
5. The convolution function calls the kernel function for each pixel, using two nested for loops.
6. The kernel function calculates the kernel for a pixel using the edge extrapolation function, the matrix structure, and the source image. The final calculation for the pixel is stored in the output BMP* structure directly for speed.
7. Once the whole convolution is complete, the program writes a new file with the filtered image.

## 2.2 Parallel Implementation

The parallel implementation differs from the serial implementation in that instead of the master thread (the serial program) iterating through each pixel, the parallel implementation has several threads going through different pixels at the same time.

This is achieved by use of an omp for loop structure. We use dynamic scheduling to accommodate for potential differences in complexity in regions of the image. Each iteration of the outer for loop calculates the kernel for a (image height)x([matrix width]x2) sized chunk of the image (See figure 2.1) (Note that the final chunk is truncated to fit with the width of the image). There are a few reasons for this choice of task decomposition:

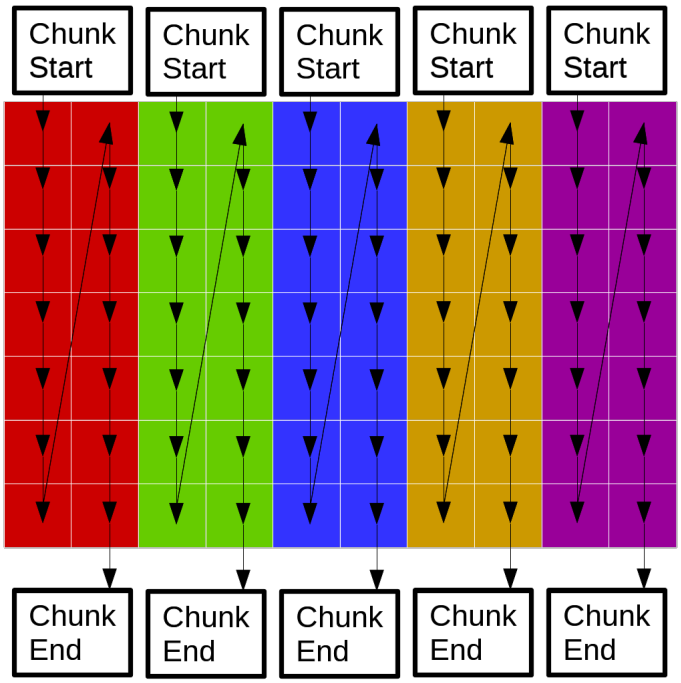



Figure 2.1: The decomposition of the workload of the image into chunks.

- We want to to keep the number of chunks for threads to process low so as to reduce unproductive inter-thread communication time, but we also want there to be enough chunks so that threads that execute their chunks quickly can still have something useful to do while waiting for other threads.
- We want the dimensions of the chunks to minimize potential contention between threads, specifically when we are calculating the kernel for two separate pixels which are in kernel dimension range (this would result in both the kernel calculations needing to contend for the same pixels for their own calculation, see figure 3.2). Assuming the image is reasonably sized, using (matrix width)*2 width chunks will nearly always keep the kernel calculations far enough away from each other at each moment of execution, even when one thread is running faster than the others. For

a 5x5 kernel, a thread would have to have done more than 5 columns of the image before one thread had done a single column for there to be contention issues in this regard.

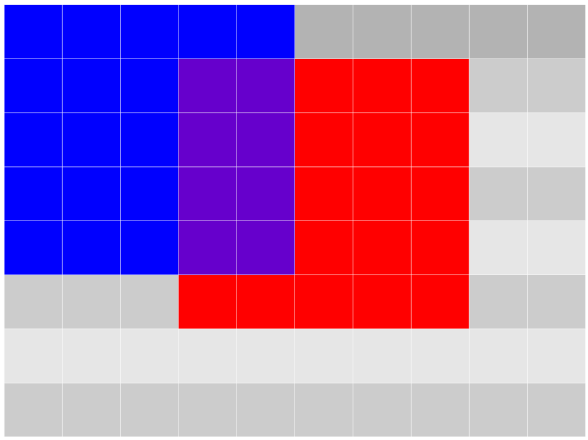


Figure 2.2: An example of contention with nearby simultaneous kernel calculations. The purple region represents potentially contested pixels.

Each thread works column by column. We do not create separate sub images for each thread to use for source and output because it would require splitting overhead at the beginning and joining overhead at the end. It is mentioned in the EasyBMP documentation (EasyBMP Manual 1.06, pg. 16) that the RGBApixel pointers are stored internally as a 2d array. We also know that this must be stored on the heap, since it has to do deal with varying sized images (and thus, varying sized 2d arrays). Finally, we can deduce that since the size of the array is width x column that these 2d arrays store arrays of rows (the x component of the image).

So what about the contention that arises when threads read from the same source image in memory at approximately the same time, and write to the same output image at approximately the same time. Will contention on the cache line arise from reading/writing the same row?

We compared two versions of the parallel implementation - one which sequentially accessed by row and one which sequentially accessed by column. The difference in execution time was negligible.

Why is this? We believe it is because the 2d array stores pixel POINTERS.

Suppose there is contention between 2 threads trying to access far away pixels on the same row, and - we shall presume for argument's sake - the same cache line. Then one thread will have to wait while the other gets their pixel pointer (which is of size int). But after that contention, modifying or reading to that RGBApixel - which forms the vast bulk of the process of modifying a pixel - will be independent of modifying or reading any other RGBApixel, since it is a pointer and can exist anywhere in the heap and thus on

separate cache lines. (Note that this process includes getting the actual separate R,G,B components of the RGBApixel from memory)

So since the delay caused by the contention is so small, it has an insignificant effect on the total execution time. The number of contentions also becomes less over time, as once a contention does arise, one of the threads will move forward while another has to wait and the threads will then both move forward in this synchronization. So ultimately, the threads will often be working on both different rows and different columns anyway.

# 3 Results analysis

## 3.1 Data Decomposition Graph

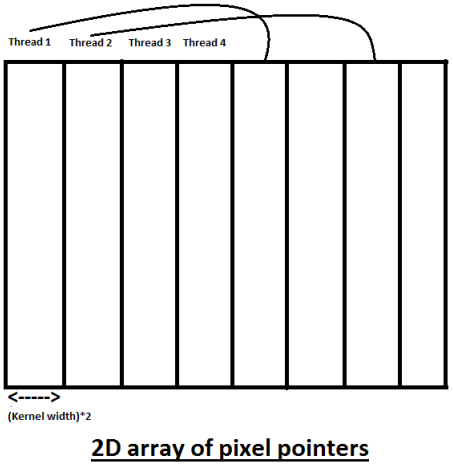



Figure 3.1: Splitting of the data

For more explanation see the chunks image above.

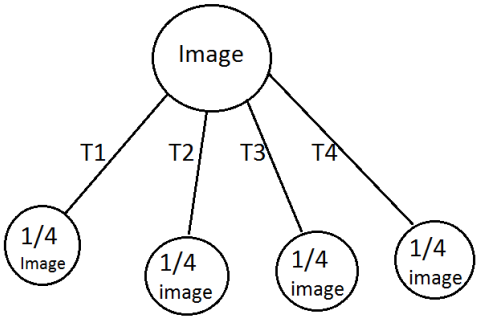



Figure 3.2: Data decomposition graph

## 3.2 Task Dependency Graph

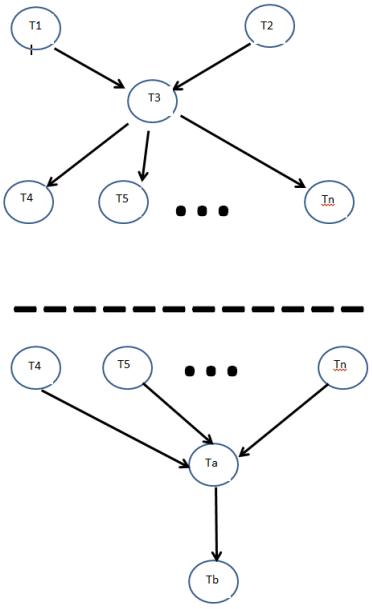



Figure 3.3: Data decomposition graph

- T1 = Create Kernel Matrix
- T2 = Load Image
- T3 = Image Covolution (start)
- T4 - Tn = Kernel Multiplication
- Ta = Image Convolution Return
- Tb = Save Image

## 3.3 About the code

- **To compile** : make
- **To Run**

  ./main source.bmp output.bmp < s | p > < simple | thread | images >**s**

S or p determines whether the parallel or serial version is used (not used for thread test). simple, thread, image determines whether the program is simply run (simple) - thread makes the program run the thread test, images makes the program run the image test.

## 3.4 About the Hardware

Our 4 core machine:

- 3.8 GHz
- Quad Core

Our 8 core machine:

- 3.8 GHz
- 8 Cores

## 3.5 Analysis

The following results were obtained by using a 5x5 Gaussian blur kernel matrix on the image set stored in TestBMP. Each .bmp image has a resolution of 2560x1440 px resulting in a size of 11.1Mb per image. Now, Because of this large resolution each image was processed 10 times and the average of that processing time was recorded to promote more accurate results.

We should also note that the time taken to read from and write to files have been excluded.

### 3.5.1 Serial vs. Parallel (4 threads) on 10 images

The following table illustrates the parallel and serial processing time for each image.

| **Image Used** | **Serial Time(S)** | **Parallel Time(S)** | **Difference(S)** |
|---|---|---|---|
| 1.bmp | 3.69834 | 1.03163 | 2.66671 |
| 2.bmp | 3.67825 | 1.02016 | 2.65809 |
| 3.bmp | 3.68039 | 1.02509 | 2.6553 |
| 4.bmp | 3.68526 | 1.03562 | 2.64964 |
| 5.bmp | 3.66079 | 1.03453 | 2.62626 |
| 6.bmp | 3.67016 | 1.0301 | 2.64006 |
| 7.bmp | 3.678 | 1.03763 | 2.64037 |
| 8.bmp | 3.68696 | 1.03739 | 2.64957 |
| 9.bmp | 3.67905 | 1.03905 | 2.64 |
| 10.bmp | 3.67905 | 1.09719 | 2.58186 |

Using the above table we can see that on average our parallel implementation lowers our processing time by 2.640786 seconds.

Moreover, If we denote the average serial time by **Ts** and denote the average parallel time by **Tp**, where
**Ts** = 3.679625 and

**Tp** = 1.038839
We can calculate the Average Speed Up($S_a$) for our solution by using the formula

$$\begin{aligned} S_a &= \frac{Ts}{Tp} \\ &= \frac{3.679625}{1.038839} \\ &= 3.5420551 \end{aligned} \tag{3.1}$$

More over, We can calculate our efficiency(E) using

$$E = \frac{S_a}{p} \tag{3.2}$$

Here, p represents the number of processing elements involved. In our case, We've used p = 4, Since 4 processors is optimal for our current hardware situation. Hence the Above becomes,

$$\begin{aligned} E &= \frac{3.5420551}{4} \\ &= 0.885514 \end{aligned} \tag{3.3}$$

The reason for this increase in performance is explained in section 2.2
The following diagram is a graph which represents the above table in a visual way...

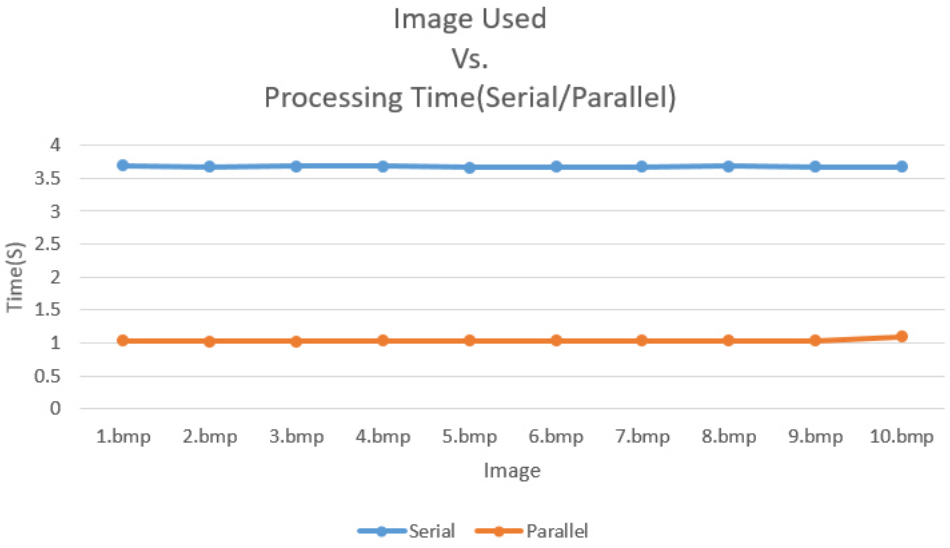



Figure 3.4: Serial Vs Parallel (Lower is better)

### 3.5.2 Serial vs. Parallel with 1-23 threads on 4 core and 8 core computers.

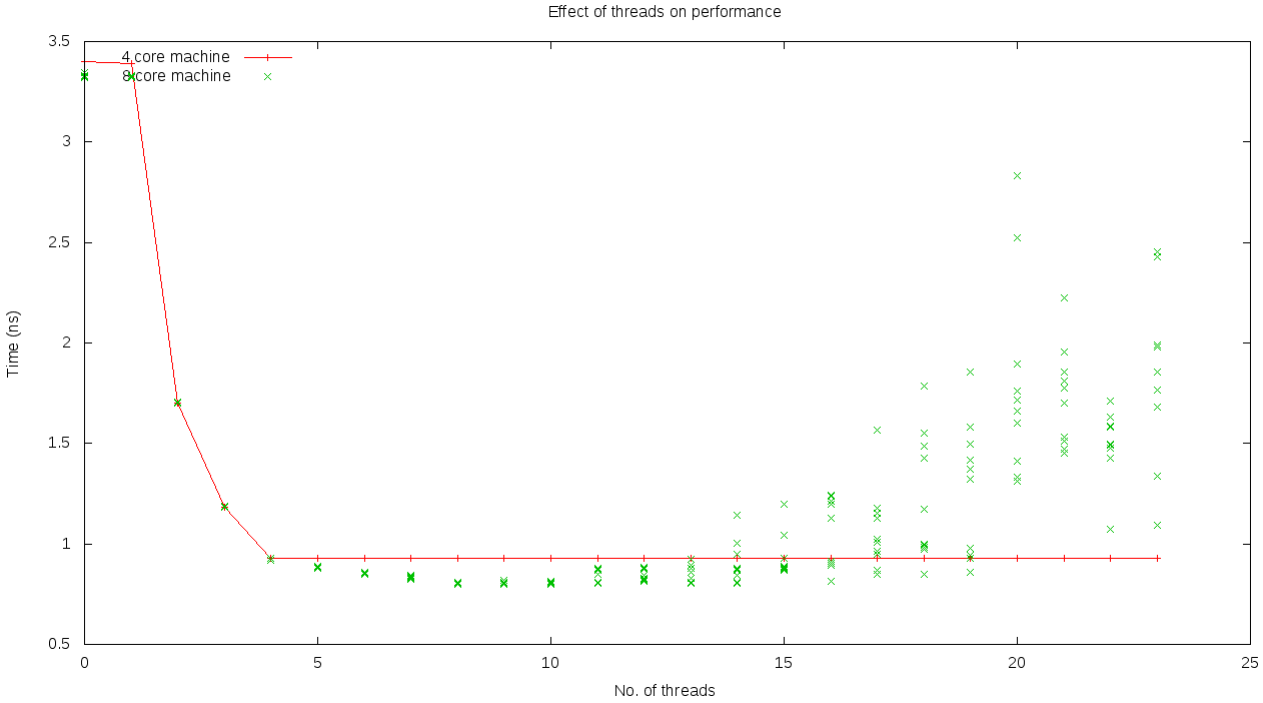



Figure 3.5: Graph displaying the effect of various amounts of threads. Note: 0 threads represents the serial implementation.

Here, using the above graph it becomes easy to see that our parallel implementation greatly out performs our serial implementation every time.

## 4 Conclusion

From the analysis it is clear that the parallel version of this particular image processing project is much more efficient than that of the serial version. This is due to the parallel version iterating through multiple pixels at the same time as opposed to one pixel at a time.

The implementation of the 2 parallel versions (accessing the 2d array row-wise and accessing the 2d array column-wise) has no significant difference in the performance. It is concluded that this is due to the fact that the 2D array stores pixel pointers and not the actual pixel value, and any contention between threads in accessing a pointer results in minor idling.

The 1st problem encountered was finding a means of inputting an image in terms of its pixel values. This was quickly resolved by the EasyBMP library. A very interesting

problem was encountered when multiplying the kernel matrix to pixels on the edges of the 2D array. It was then resolved by means of edge extrapolation, which ensures that when using the pixels at the edges that there are still values outside the 2D array for computation.

One of the main issues encountered was deciding which kernel matrix would output the best de-noised image, i.e. changing the values as well as the dimensions of the kernel matrix to create the best output image. In that case we have devised multiple examples illustrating the effects of different kernel matrices on a noisy image. We have concluded that every picture (having its own unique noise) will need a different kernel matrix for optimum de-noising.

As a whole, this project does what it was designed to do and that is to input a noisy image and change each pixel of that image by means of accurate calculations to output an image more friendly to our eyes.