School of Computer Science and Statistics
CS1022 – Computing II Assignment
Conor Gildea - [redacted]
Computing II Assignment

Assignment #1

2-Dimensional Arrays – Image Manipulation

Introduction

Each problem in this assignment was based on working with 2-Dimenasional Arrays for an image and every problem benefited from correct use of subroutines and the system stack. A brief summary of my assignment is as follows:

Stage One – Console Inputted Values from the user, of contrast and brightness, are used to adjust the contrast and brightness of an image.

Extra Miles:

- Console Input to allow the user to enter contrast and brightness values.
- Efficient run time.
- Error Handling for all cases, using console input.
- Ensuring colour components are within the correct range (0-255).

Stage Two – Console Inputted Value from the user, of radius, is used to apply a motion blur filter to an image, the size of the radius dictates the severity of the blur.

Extra Miles:

- Console Input to allow the user to enter their desired radius of the blur.
- Efficient run time.
- Error Handling for all cases, using console input.
- Ensuring colour components are within the correct range (0-255).

Stage Three – By the central limit theorem, repeated iterations of box blur to resemble a Gaussian blur affect.

Extra Miles:

- Prevented making a second copy of the image.
- Efficient run time.
- Ensuring colour components are within the correct range (0-255).
- As box blur was used a kernel to form the correct value of each pixel based on the surrounding pixels.
- Performed many iterations of box blur on the sample image in order to resemble Gaussian blur, by the central limit theorem.

Global Subroutine – Getting Colour Values

Before I began working on any stage of this assignment I had to develop how I can get each separate colour value of a pixel, from this I was able to program subroutines to use throughout this assignment.

VIVID CERULEAN – 0x0000A1E4 – RGB(0, 161, 228)
MALACHITE – 0x0004E762 – RGB(4,231,98)
SELECTIVE YELLOW – 0x00F5B700 – RGB(245,183,0)

Fig 1. Selection of colours with their hexadecimal and respective RGB values.

Getting Colour Values

For each pixel it consists of three different value components, red, green and blue (RGB). These component values are saved together in the same word sized value. Blue saved in Byte 0, Green being saved in Byte 1, Red being saved in Byte 2 whilst Byte 3 remains null. From this information we are able to work out the RGB values of any colour.

Colour Component Separation Example

Malachite (Fig.1) = 0x0004E762

Using the above information we can now split this pixel's colours, into RGB values.

Byte 3 (NULL)	Byte 2 (Red)	Byte 1(Green)	Byte 0(Blue)
0x00	0x04	0xE7	0x62

After we convert these values to decimal it yields:

Byte 3 (NULL)	Byte 2 (Red)	Byte 1(Green)	Byte 0(Blue)
0	4	231	98

From Fig.1 we are able to see we have be able to calculate the correct RGB values for the colour Malachite.

After I had completed a working example of separating the colour components I then needed to transfer this to pseudocode and ARM assembly code. There was two ways I could have tackled this problem by either using an operator such as AND to separate the values separately or by logically shifting left and right until the value left is the necessary RGB component value.

I personally decided to logically shift the binary pixel values until the correct RGB component value was found. Although logically shifting seems like a low level ability, this can still be found and is used in higher level languages such as Java E.G. (Integer.toBinaryString(2 << 24);

Psuedocode to find the Green Component of a Colour Value

```
Load Pixel Value into pixelValue;
Logically Shift pixelValue Left by 16;      //Removes the Red Component
Logically Shift pixelValue Right by 24;     //Removes the Blue Component
```

From this psuedcode I programmed a different subroutine for finding Red, Green and Blue component values from a pixel. (Subroutines: getPixelR, getPixelG, getPixelB)

Global Subroutine – Console Input - getUserValue

Due to being recently introduced to subroutines in ARM Assembly in CS1022, I decided I wanted to incorporate previous code I had written to gather console input, into a subroutine. The problem with inputted values into ARM assembly programs is that the values are kept in the ASCII character values form. Another problem with inputting numbers into ARM assembly is the fact, it doesn't process numbers by the same means humans do, "4" being entered before a "3", isn't interpreted as having the decimal value of 43 in ARM assembly. Due to this I needed a consoleInput subroutine capable of converting user input into the correct values. This is very useful for when I wanted to get the brightness, contrast and radius blur in this assignment.

Attached below is a previous worked example, I created, to explain how my console input code works.

"*For "426" each digit must first be treated individually as their hexadecimal values.*

"4" = 0x34 *"2" = 0x32* *"6"=0x36*

The numbers currently are equal to their ASCII symbol values. We must now minus 0x30 from each value to convert it from ASCII symbol values to the actual value the user intended.

0x34 – 0x30 = 0x04 = 4 *0x32 – 0x30 = 0x02 = 2* *0x36 – 0x30 = 0x06 = 6*

We must now develop a system to make: 4, 2, 6 be equal to 426. To do this we must first introduce a running total variable (runningTotal) with a value of zero and a constant with a value of 10 (which we shall refer to as simply "TEN"). After the first digit is entered we then let that be added on to the running total variable.

runningTotal = runningTotal + 4

After the second digit is entered, we must now multiply the previous running total by the second variable we declared with the value of 10.

*runningTotal = runningTotal * TEN //runningTotal = 40*

At the end of the reading loop we then must again add on the new digit to the running total.

runningTotal = runningTotal + 2 //runningTotal = 42= 0x2A

After this process is completed another time, after the third digit "6" is entered, the value should be correctly processed as "4", "2", "6" = 426 = 0x1AA."

- **Conor Gildea-CS1021–Assignment #1**

The changes I made to this subroutine specifically for this assignment was to have a boolean negativeAllowed as a parameter for this subroutine. This allows me to prevent the user inputting negative values.

Stage One – Console Inputted Brightness and Contrast

Producing a program to adjust the brightness and contrast of an image in ARM assembly was quite non-trivial. Changing the brightness and contrast was quite a simple task based on the formula's given as part of the specification for this assignment (see below), where in beta is the brightness and alpha is the contrast.

$$R'_{i,j} = \left(\frac{R_{i,j} \times \alpha}{16}\right) + \beta$$

$$G'_{i,j} = \left(\frac{G_{i,j} \times \alpha}{16}\right) + \beta$$

$$B'_{i,j} = \left(\frac{B_{i,j} \times \alpha}{16}\right) + \beta$$

To complete this task I built a subroutine to multiply the RGB components values by the contrast and then be divided by 16, to get the correct contrast value. Later on in this subroutine I had the program add/subtract the brightness/darkness (depending on the user input). This subroutine was called **contrastAndBrightness**. An important factor to remember when building this subroutine was to complete it in the correct order, the order of precedence matters significantly. We must complete the contrast calculations then adjust the brightness.

Extra Mile – Console Input and Error Handling

To make my program more user friendly and to prevent the user entering in invalid numbers for the contrast, I used console input. By using console input, I was able to restrict what values were entered. This was increasingly important for entering the contrast values when negative numbers should not be allowed. This was a simple error to prevent happening, simply by only allowing the user to enter whole number values.

Main Psuedocode for Stage One

After the user input has been collected and using the subroutines created, the main pseudocode for stage one, looked like this:

```
for(int row = 0; row<image length; row++)
{
    for(int column= 0; column<image width; column++)
        {
                Calculate address for current pixel
                get red pixel value
                change contrast and brightness
                get green pixel value
                change contrast and brightness
                get blue pixel value
                change contrast and brightness
                combine new red pixel, green pixel and blue pixel values and save
        }
}
```

The main pseudocode for stage one is extremely easy as it is simply going to every pixel in the image and adjusting the values to match the user defined contrast and brightness values.

Extra Mile – Error Handling

After making the relative adjustments to the colour values in a RGB component it was important to check to see was this value within the RGB range for each component (0-255). If the value of the component happened to be less than 0, I had to make my program set it to 0 and if the value happened to be above 255, I had to set my program to set it to 255. This ensured my program behaved correctly.

Subroutines for Stage One

contrastAndBrightness

Function:

Changed the contrast and brightness of a RGB component value

Parameters:

componentValue contrastValue brightnessValue

Return value:

adjustedComponentValue

setPixel

Function:

Saves a pixel value in an address

Parameters:

pixelValue addressOfPixel

Return value:

Pixel Saved in Memory

getPixelR/getPixelG/getPixelB

Function:

Gets the corresponding RGB component colour of a colour (depending on subroutine called)

Parameters:

addressOfPixel

Return value:

correspondingRGBComponentColourValue

getUserValue

Function:

Converts ASCII number input into a hexadecimal number value, and allows/prevents (depending on parameters) negative numbers.

Parameters:

negativeNumbersAllowedBoolean

Return value:

userEnteredValue negativeNumberEnteredBoolean

negate

Function:

Converts a number into 2's Compliment Form – Turning it negative

Parameters:

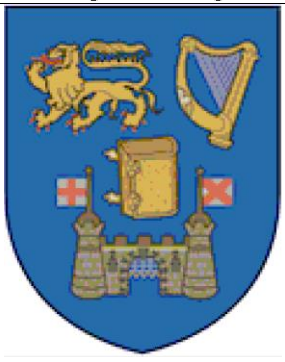
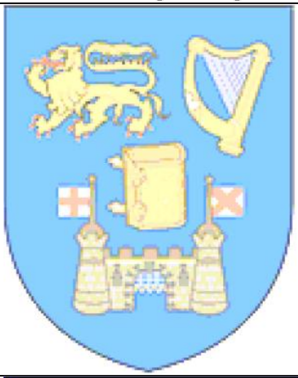
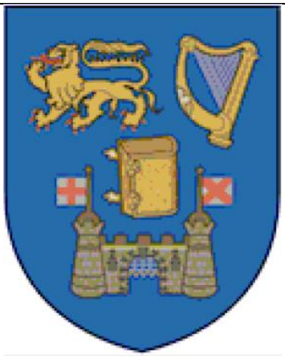
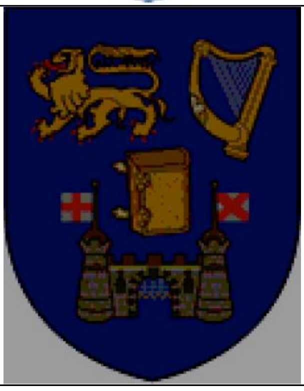
valueToConvert convertToNegativeBoolean

Return value:

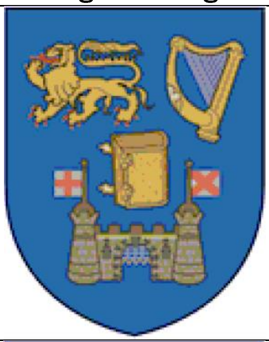
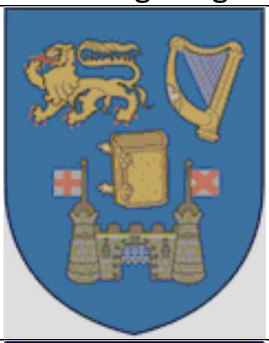
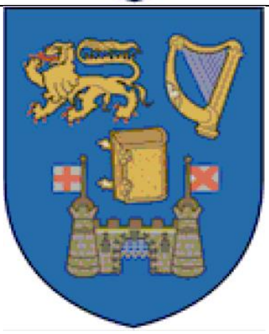
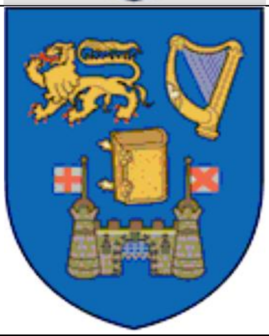
valueIn2ComplimentForm

Methodology for Stage One

Brightness Test Results:

Original Image	Resulting Image	Test Case – Correct/Incorrect? Why?
		Brightness set to 100 Contrast set to 16 (default – No change) **Correct:** The brightness of the resulting image has increased as expected.
		Brightness set to -100 Contrast set to 16 (default – No change) **Correct:** The brightness of the resulting image has decreased as expected.

Contrast Test Results:

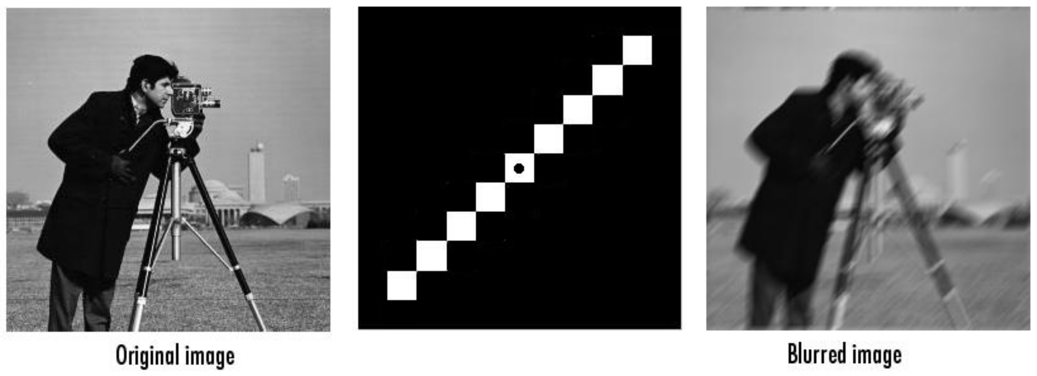
Original Image	Resulting Image	Test Case – Correct/Incorrect? Why?
		Brightness set to 0 Contrast set to 10 **Correct:** The contrast has decreased as expected due to the inputted contrast value being less than 16.
		Brightness set to 0 Contrast set to 20 **Correct:** The contrast has increased as expected due to the inputted contrast value being greater than 16.

	Brightness set to 0
	Contrast set to -2
	`Calling start() ...`
	
	`Please enter the negative or positive value you want to`
	`increase the brightness by:`
	`-10`
	`Please enter the value you want to change the contrast by,`
	`if the value is above 16 the contrast will increase,`
	`if it is below 16 the contrast will decrease and if the value is 16,`
	`the contrast will remain unchanged.The value must be positive.`
	`2`
	
	Correct: Console input prevents negative contrast numbers being entered, as contrast must be positive following this formula.

<u>Conclusion for Stage One:</u> From all of the previous testing, we are able to see my program functions perfectly for adjusting the values of brightness and the contrast whilst error handling.

Stage Two – Motion Blur

In Stage Two, I had to apply a motion blur to an image. I believe the following image is a great example of how this motion blur filter can be achieved. First you add up all the separate RGB components of all the pixels in white (the pixels in the diagonal radius of the pixel being processed), the program must then divided each RGB component value by the number of pixels that have been sampled. The final values then left after this division are the RGB component values for the pixel being tested (the pixel with a black dot in it). After this process has been completed, we move onto the pixel beside the previous pixel being sampled and repeat the process.



Source image: http://www-rohan.sdsu.edu/doc/matlab/toolbox/images/deblurr9.html

For this stage, I also had to implement a div subroutine, to be able to divide two numbers and find the quotient. As I would be dividing the total for each of the separate RGB total component values by the numbers of pixels sampled to get the value of the pixel currently having the motion blur applied.

In stage two, I implemented my **getUserInput** subroutine, to allow the user to enter a value for the radius of the motion blur they would desire, using this subroutine I also prevented a negative radius from being entered. I was able to reuse a lot of the subroutines I made for stage one, which was extremely convenient and shows the value of producing general propose subroutines.

Extra Mile – Error Handling

Just like in my previous stage, after making the relative adjustments to the colour values in a RGB component it was important to check to see was this value within the RGB range for each component (0-255). If the value of the component happened to be less than 0, I had to make my program set it to 0 and if the value happened to be above 255, I had to set my program to set it to 255. This ensured my program behaved correctly.

Main Pseudocode for Stage Two – Motion Blur

Below is the main pseudocode, I used, based on the explanation above, to program my motion blur.

```
for(int row = 0; row<image length; row++)
{
    for(int column = 0; column<image width; column++)
        {
                column – radius;
                row – radius;
                redTotal = 0;
                greenTotal = 0;
                blueTotal = 0;
                numberOfPixelsCompleted= 0;
                while(!maxPixelInDiagonalRadius)
                {
                        if(validPixel)
                        {
                                Load pixel values;
                                Find and save RGB component values separately into redTotal,
                                greenTotal and blueTotal;
                                numberOfPixelsCompleted++:
                        }
                        Column++;
                        Row++;
                }
                red = redTotal/numberOfPixelsCompelted;
                green = greenTotal/numberOfPixelsCompleted;
                blue = blueTotal/numberOfPixelsCompleted;
                Combine these values of red, green and blue to find the correct values for the
                pixel currently having the motion blur applied.
                SAVE RGB value in the pixel of the corresponding row and column.
            }
}
```

Subroutines for Stage Two

motionBlur

Function:

Applies a motion blur to an image of a previously defined radius.

Parameters:

blurRadius

Return value:

A saved image with a motion blur filter, of the previously defined radius, applied.

div

Function:

Divides two numbers firstNumber/secondNumber

Parameters:

numberToBeDivided divisor

Return value:

quotient

getPixelR/getPixelG/getPixelB

Function:

Gets the corresponding RGB component colour of a colour (depending on subroutine called)

Parameters:

addressOfPixel

Return value:

correspondingRGBComponentColourValue

getUserValue

Function:

Converts ASCII number input into a hexadecimal number value, and allows/prevents (depending on parameters) negative numbers.

Parameters:

negativeNumbersAllowedBoolean

Return value:

userEnteredValue negativeNumberEnteredBoolean

negate

Function:

Converts a number into 2's Compliment Form – Turning it negative

Parameters:

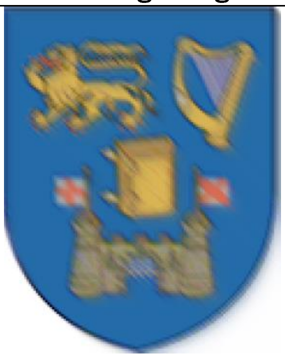
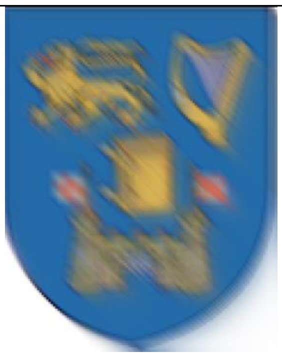
valueToConvert convertToNegativeBoolean

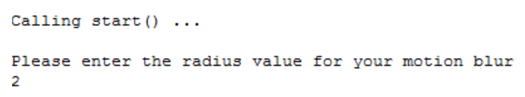
Return value:

valueIn2ComplimentForm

Methodology for Stage Two

Motion Blur Test Results:

Original Image	Resulting Image	Test Case – Correct/Incorrect? Why?
		Radius set to 2 **Correct:** The applied motion blur, would correspond with a motion blur of radius two.
		Radius set to 4 **Correct:** The applied motion blur, would correspond with a motion blur of radius four.

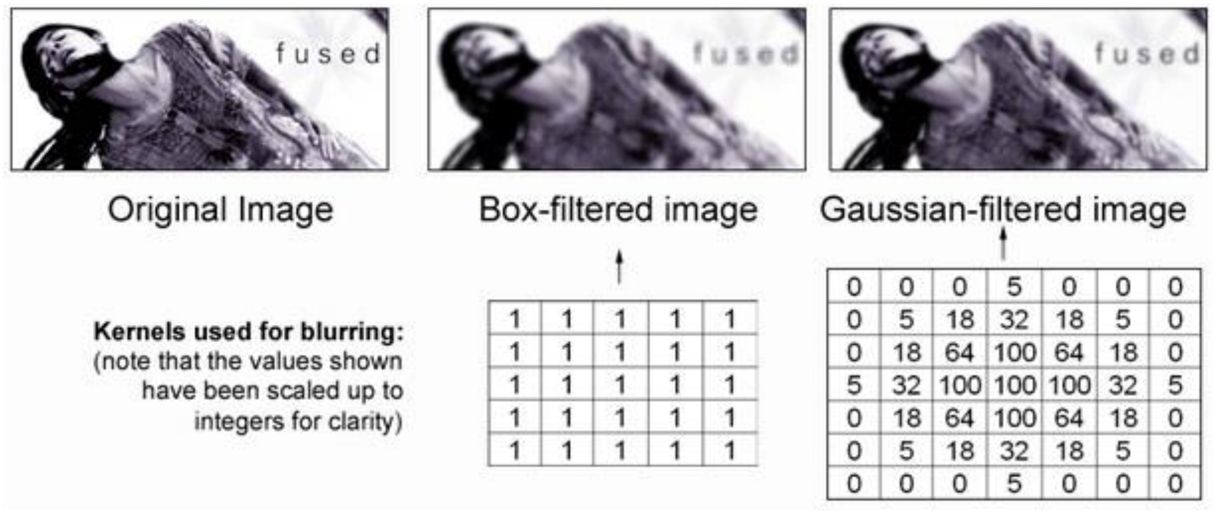
	Radius set to -2
	
	
	`Calling start() ...`
	
	`Please enter the radius value for your motion blur`
	`2`
	
	Correct: Console Input Prevents the user from entering negative numbers as their blur radius.	

Conclusion for Stage Two: From all of the previous testing, we are able to see my program functions perfectly for applying a user defined motion blur to an image, while error handling.

Stage Three – Bonus Effect – Box Blur Iterated Repeatedly in Order to Resemble Gaussian Blur

For Stage Three, I had considered many different effects I could implement however I was eventually torn between Gaussian Blur and Box Blur. I felt many effects such as edge detection were too easy and I wanted to challenge myself with something more sophisticated.

I believe the diagram below is extremely useful for comparing the differences between Gaussian blur and box blur. From this diagram we can see both blurs use a kernel however the kernel for a Gaussian Blur priorities the pixels closest to and the exact pixel being tested.


fused
fused
fused
Original Image
Box-filtered image
Gaussian-filtered image
Kernels used for blurring:
(note that the values shown have been scaled up to integers for clarity)

1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

0	0	0	5	0	0	0
0	5	18	32	18	5	0
0	18	64	100	64	18	0
5	32	100	100	100	32	5
0	18	64	100	64	18	0
0	5	18	32	18	5	0
0	0	0	5	0	0	0

Source: http://www.gamasutra.com/view/feature/131511/four_tricks_for_fast_blurring_in_.php

I had thought though how I could implement both Gaussian Blur and Box Blur before I had programmed stage three. One of the hardest parts of attempting a Gaussian blur would have been the implementation of floating point numbers, originally I had planned to implement this by saving the integer part in a register and have the decimal part in a separate register, however I soon realised how implausible this would be given the number of calculations needed to take place and the amount of registers/memory needed.

During my research I had considered the possibility of using IEEE-754-1985 standard, the method by which floating point numbers can be saved in a single register. I had even gone as far to research all the separate mathematical operations and how they can be performed when numbers are in IEE-754-1985 standard. This included the finding the square root approximation of a floating point number, thanks to the help of John Carmack's fast inverse square root algorithm (also called "0x5f3759df").However I eventually decided to implement box blur over Gaussian blur, given the discovery of the central limit theorem.

Gaussian Blur vs Box Blur – Central limit Theorem

"Box blurs are frequently used to approximate a Gaussian blur. By the central limit theorem, if applied 3 times on the same image, a box blur approximates the Gaussian kernel to within about 3%, yielding the same result as a quadratic convolution kernel."

Source: http://www.gutenberg.us/Article.aspx?Title=box_blur

From discovering the implications of the central limit theorem, by having three iterations of box blur, I could produce a Gaussian blur within approximately 3%. From this information I decided to attempt to prove central limit theorem to be true, by producing a program that applies a box blur to an image three times. After I had my program complete, I wanted to be able to test it against a Gaussian blur and see the differences or similarities.

Main Psuedocode for Stage Three – Gaussian Blur Approximation using Box Blur

```
for(int row = 0; row<image length; row++)

{
    for(int column = 0; column<image width; column++)
        {
                column – 1;
                row – 1;
                boxIndex = 0;
                redTotal = 0;
                greenTotal = 0;
                blueTotal = 0;
                numberOfPixelsCompleted= 0;
                while(boxIndex!=10//When boxIndex==9 3x3 Box Blur is complete on the pixel
                {
                        if(validPixel)
                        {
                                Load pixel values;
                                Find and save RGB component values separately into redTotal,
                                greenTotal and blueTotal;
                                numberOfPixelsCompleted++:
                        }
                        boxIndex++;
                        if((boxIndex==4)||(boxIndex==7))
                        {
                                column-3;
                                row++;
                        }
                        else if(boxIndex!=10)
                        {
                                column++;
                        }
                        else if(boxIndex==10)
                        {
                                column--;
                                row--;//Return column and row values back to pixel being blurred
                        }
                }
                red = redTotal/numberOfPixelsCompelted;
                green = greenTotal/numberOfPixelsCompleted;
                blue = blueTotal/numberOfPixelsCompleted;
                Combine these values of red, green and blue to find the correct values for the
                pixel currently having the motion blur applied.
                SAVE RGB value in the pixel of the corresponding row and column.
        }
}
```

Subroutines for Stage Three

boxBlur

Function:

Applies a box blur to an image multiple times to resemble Gaussian Blur by the central limit theorem.

Parameters:

None

Return value:

A saved image which has had a box blur applied multiple times to approximate a Gaussian blur.

div

Function:

Divides two numbers firstNumber/secondNumber

Parameters:

numberToBeDivided divisor

Return value:

quotient

getPixelR/getPixelG/getPixelB

Function:

Gets the corresponding RGB component colour of a colour (depending on subroutine called)

Parameters:

addressOfPixel

Return value:

correspondingRGBComponentColourValue

getUserValue

Function:

Converts ASCII number input into a hexadecimal number value, and allows/prevents (depending on parameters) negative numbers.

Parameters:

negativeNumbersAllowedBoolean

Return value:

userEnteredValue negativeNumberEnteredBoolean

negate

Function:

Converts a number into 2's Compliment Form – Turning it negative

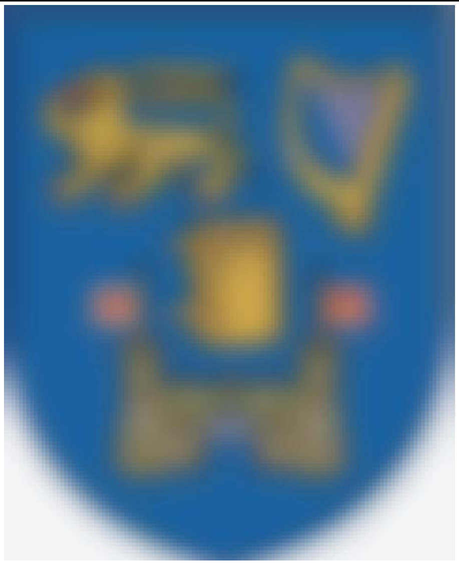
Parameters:

valueToConvert convertToNegativeBoolean

Return value:

valueIn2ComplimentForm

Methodology for Stage Three

Original Image	Resulting Image
	

Conclusion for Stage Three: From all of the previous testing, we are able to see that the central limit theorem is proven true, that multiple iterations of box blur produces an approximation of Gaussian blur.

Gaussian Blurred Image (Using Gimp) (Based on Original Image)	Resulting Image (Based on Original Image)