David Pearson
1725412

5CS021

Numerical Methods and Concurrency

Assignment

Contents

Part 1: Numerical Methods in C

1.1 y = mx + c

Filename: `Task1-1.c`

Compile command: `cc Task1-1.c -o Task1-1`

Execute command: `./Task1-1`

Requirements:

Your function will take in 4 parameters; x1, y1, x2 and y2. The 1's represent the first x and y coordinate of where the line begins, and the 2's are the coordinates where the line ends. You will then need to work out m (gradient) and then "c." Your output should be in the form of "y = mx +c".

Functions:

Function Name	Arguments	Purpose	Logic Description	Return
Header	none	This function generates a header at the top of screen. Showing the user information about the program and its creator.	A single printf statement, spread over three lines and tabbed in.	none
ReadIn	none	This function prompts the user to enter 4 values, one at a time. To be used to calculate the required equation	Four printf statements each followed by a scanf to capture the typed integer then assign it to the corresponding global variable	none
Clear	none	This function simply clears the screen, to aid in reducing any confusion	This passes the required ASCII characters in via octal values to clear the screen	none
CoordinateOut	none	This function repeats back to the user the co-ordinates that have been passed in.	Two printf statements using the assigned integer variables. The variables are printed to 1 decimal place.	none
findValues	none	This is the primary solve function, additionally it will output the solution to the user	The variable m is calculated as y2-y1/x2-x1 The variable c is calculated by y1-(m*x1) The variables are printed to 1 decimal place.	none
formulaOut	none	This function displays to the user the solution in the form of y=mx+c	A single prinft statement using the calculated integer variables. The variables are printed to 1 decimal place.	none

Notes:

I did all of the working for this question via functions, my main for the program containing nothing but void function calls that manipulated global variables. If I were to revisit this, I would include various elements of user input validation, such as an option when the CoordinateOut function is called, the user could be given asked to verify the input and given the opportunity to enter new values.

Mark expectation:
3/3

1.2 Quadratic Equation

Filename: `Task1-2.c`

Compile command: `cc Task1-2.c -lm -o Task1-2`

Execute command: `./Task1-2`

Requirements:

The program compiles and the right output is printed depending on input. The function should take in 3 parameters (a, b and c). The program will then give the 2 x values where the curve meets the x axis.

Functions:

Function Name	Arguments	Purpose	Logic Description	Return
Header	none	This function generates a header at the top of screen. Showing the user information about the program and its creator.	A single printf statement, spread over three lines and tabbed in.	none
ReadIn	none	This function prompts the user to enter 3 values, one at a time. To be used to calculate the required equation	Three printf statements each followed by a scanf to capture the typed integer then assign it to the corresponding global variable	none
Clear	none	This function simply clears the screen, to aid in reducing any confusion	This passes the required ASCII characters in via octal values to clear the screen	none
solve	none	This is the primary solve function. For the program	The function uses the global variables A,B&C and then using the quadratic formula calculates and assigns x1 & x2. The calculations make use of the math.h library	none
printX	none	This function displays on the screen the two solutions for x to the user	A single printf statement is used to show the values for x1 and x2. The variables are printed to 1 decimal place.	none

Notes:

I did all of the working for this question via functions, my main for the program containing nothing but void function calls that manipulated global variables. If I were to revisit this, I would include a further set of print statements that would verify to the user the values entered as well as possibly a print statement that shows the values in the quadratic formula.

Mark expectation:
3/3

1.3 Solving cubic polynomials

Filename: `Task1-3.c`

Compile command : `cc Task1-3.c -lm -o Task1-3`

Execute command : `./Task1-3`

Requirements:

The program compiles and the right output is printed depending on input. The function should take in 4 parameters (a, b, c and d). The program will then give the 3 x values where the curve meets the x axis.

Functions:

Function Name	Arguments	Purpose	Logic Description	Return
Header	none	This function generates a header at the top of screen. Showing the user information about the program and its creator.	A single printf statement, spread over three lines and tabbed in.	none
ReadIn	none	This function prompts the user to enter 4 values, one at a time. To be used to calculate the required equation	Four printf statements each followed by a scanf to capture the typed double then assign it to the corresponding global variable	none
Clear	none	This function simply clears the screen, to aid in reducing any confusion	This passes the required ASCII characters in via octal values to clear the screen	none
FirstRoot	none	This function attempts to find the first root of the cubic polynomial	The logic used here is, while the lastguess variable does not equal G (the guess variable) and G is not a root, change G using the following formula G = cbrt(-((b*G*G + c*G + d) / a))	none
solve	none	This function finds the two remaining roots	The function uses the first root and substitutes vales into the quadratic equation to generate the remaining 2 roots.	none
printRoots	none	This function displays on the screen the two solutions for x to the user	A single printf statement is used to show the values for r1,r2 and r3. The variables are printed to 2 decimal places.	none

Notes:

My first attempt at a solution for this problem was to brute-force the first route. Set my guess variable to be negative ten thousand and increment to positive ten thousand. Though my solution works for the majority of values entered it will not work for imaginary or some complex routes.

Mark expectation:
3/4

1.4 Linear Regression

Filename: `Task1-4.c`

Compile command: `cc Task1-4.c -o Task1-4`

Execute command: `./Task1-4 datasetLR1.txt` *(1 through 4 may be used)*

Requirements:

The program compiles and the right output is printed depending on input. Your program will initially load in a text file of data. You will then generate a y=mx+c (linear model) to determine different groups of data (there will only be two types). Your program will then ask the user to input a new value (an x value) which will then print out which group that coordinate belongs to using your y = mx + c formula.

Functions:

Function Name	Arguments	Purpose	Logic Description	Return
Header	none	This function generates a header at the top of screen. Showing the user information about the program and its creator.	A single printf statement, spread over three lines and tabbed in.	none
Clear	none	This function simply clears the screen, to aid in reducing any confusion	This passes the required ASCII characters in via octal values to clear the screen	none
fileSize	Argv[1]	This function counts the number of lines in the file passed in and returns an integer with the count	The function uses the second argument passed in on the command line. It then opens the file using fopen and increments a count value each time fgetc reads in a new line character. The function then returns the count variable.	integer
LinearFunction	Argv[1]	This is the main function for this program. It will read the file passed in then using the linear regression formula create a y = mx +c equation. Finally asking the user to enter an X value and show the user the corresponding Y value.	The function uses the second argument passed in on the command line. It then opens the file passed in as an argument. It then populates two integer arrays, storing x values in the first and y vales in the second. The function then generates the variables needed for the linear regression equation , using a for loop to cycle through the elements. A printf statement will output to the user the equation in the form of y=mx+c Then a printf statement coupled with a scanf, will use the read input value to calculate the Y value and output this in a final printf statement.	none

Notes:

I was very happy with this program, though again I feel if I had to refine the program further. My refinements would take the form of input validation, covering all aspects from the file passed in on the command line through the requested x value.

Mark expectation:
7/7

Part 2: Concurrency

2.1 Prime Numbers

Filename: `Task2-1.c`

Compile command: `cc Task2-1.c -pthread -o Task2-1`

Execute command: `./Task2-1 'input filename' 'output filename'`

Requirements:

You will be given a file containing a list of numbers. These numbers will be random. You will create a C program which counts the number of prime numbers there are within the file and output to a file the amount of prime numbers along with the prime numbers themselves. The aim of this task is to use POSIX threads to parallelize the task to take advantage of the multicore processor within your machine to speed up the task. The threads you spawn within the program must compute an equal or close to an equal amount of computations to make the program more efficient in relation to speed. For this section, as you will only be reading one file and splitting it across many threads (determined by argv[1]), you should load in the file and split the file into equal parts, then process each slice within your threads. NOTE – this program should work with any amount of threads.

Functions:

Function Name	Arguments	Purpose	Logic Description	Return
Header	none	This function generates a header at the top of screen. Showing the user information about the program and its creator.	A single printf statement, spread over three lines and tabbed in.	none
Clear	none	This function simply clears the screen, to aid in reducing any confusion	This passes the required ASCII characters in via octal values to clear the screen	none
fileLength	char *filename	This function counts the number of lines in the file passed in and returns an integer with the count	The function opens the file using fopen and increments a count value each time fgetc reads in a new line character. The function then returns the count variable.	integer
Check_prime	long	This function looks at the long value passed in and checks to see if it is a prime number	The function first looks to see if the number passed in is less than 2, if not then it uses a for loop, the counter starting at 2 and increasing, incrementally to the value passed in -1. An if statement, using modulas evaluates if there is any remainder when divided by the count variable. The function will return a 0 if not prime and a 1 if prime.	integer
CreateNumberList	char *filename long *list int length	This function populates the *list passed in with the integers within the filename passed in.	The logic for this, is that it will open the file passed in using fopen with a read argument. Then within a for loop that uses the length variable passed in as a counter limit, it will use fscanf read each number and place them into the array at the index provided by the count variable.	none
outputNumbers	char *filename long *list int length	This function populates the *list passed in with the integers within the filename passed in.	The logic for this, is that it will open/create a file using the name passed in with fopen with the write argument. Then using a FOR loop that uses the length variable as a counter, it uses fprintf to place each number from the array that is not equal to -1 , into the file using the index provided by the count variable.	none

*tSolve	void *arg	This is the function that is used by each of the threads created.	The function takes in a struct that is then used by the algorithm to manipulate global variables. The struct passed in contains an end value and a start value, used as indexes for a FOR loop. The function checks each element at the counter location by passing it into the check_prime function. If it is found to be prime then it is placed into the outlist array, if it is not prime then the value of -1 is placed in the out array.	none

Notes:

This is the first program that I have created where work has been done in the main. The program also makes use of time.h to show if there is any increase based upon the number of threads requested. This is the flow of the program:

- The clock timer is started
- The screen is cleared
- The header is shown
- The user is asked to enter the desired number of threads
- The length of the file passed in is assigned to a variable using the fileLength function
- The user is shown how many numbers are in the file
- The two global pointers are allocated memory using malloc, dynamically, based on the size of the file
- The inList is populated using the CreateNumberList function
- An array of structs is created based on the number of threads
- The structs are dynamically, equally, filled based upon number of threads and number of values
- The user is shown the block sizes
- The threads are then initialised, set away, then re-joined
- The output text file is then created using the outputNumbers function
- The length of the newly created file is obtained using the fileLength function
- The timer is stopped
- The user is shown how long the file is and how many prime numbers were found

Mark expectation:
7/7

2.2 Word count (not threaded)

Filename: `Task2-2.c`

Compile command: `cc Task2-2.c -pthread -o Task2-2`

Execute command: `./Task2-2 'file1' 'file2' 'file3' 'file4'`

Requirements:

You will be given multiple files containing words. Your task is to count the frequencies of each distinct word across the files, and to produce a file at the end with all the distinct words and the number of times they each appear across all the input files. You will be given 4 text files containing a list of words. Your program will ONLY be tested using 4 threads or less. This means each thread should process at least one file. This section primarily tests on your method of returning the values from multi-threaded function, and then storing the data in an appropriate variable to printed to a text file.

Functions:

Function Name	Arguments	Purpose	Logic Description	Return
Header	none	This function generates a header at the top of screen. Showing the user information about the program and its creator.	A single printf statement, spread over three lines and tabbed in.	none
Clear	none	This function simply clears the screen, to aid in reducing any confusion	This passes the required ASCII characters in via octal values to clear the screen	none
fileLength	char *filename	This function counts the number of lines in the file passed in and returns an integer with the count	The function opens the file using fopen and increments a count value each time fgetc reads in a new line character. The function then returns the count variable.	integer
Addwords	char *filename int start int end	This function reads the words in the files and builds a struct for each one. It then populates all of these into one large global list	The function starts by opening the file with the name passed in. then using the start and end values passed in within a FOR loop, it calls fscanf reading each word as a string and then placing it in the struct array at the corresponding index position.	none

Notes:

For me this has been one of the most difficult programs in the entire assignment. This first version does not use multi-threading at all. However, it shows that the algorithm works. My logic for solving this was as follows:

- Created a struct which contained a string for the word and a count variable set to 0
- Create and malloc an array of my structs using the combined total number of words across all 4 files as the upper limit
- Populate the struct array using all words so that I have one element to manipulate
- Pass each struct in the array through the algorithm
    - The algorithm uses an outer FOR loop and looks firstly at the count variable, if it is -1 then it will move onto the next element. If the count is 0 then this means it is the first time the word has been seen, the count is set to 1 and this specific struct will be the unique word. If the count is > 0 then the algorithm will drop into a second FOR loop. This loop, again, checks the elements count variable and if is -1, moves onto the next element. The count variable is then checked to see if it is equal to 0, this symbolises a word that has not yet been counted. When this occurs the outer FOR loop elements word variable is checked against the inner FOR loop elements word variable, if they are the same then the outer FOR loop elements count is increased and the matching inner FOR loop element has its count set to -1. The word having been counted.
- The program then loops through the amended struct array and passes the word and count to a file for any with a count that is not -1 showing the user the number of unique words found.

Mark expectation:

10/17

2.2 Word count (threaded)

Filename: `Task2-2MT.c`

Compile command: `cc Task2-2MT.c -pthread -o Task2-2MT`

Execute command: `./Task2-2MT 'file1' 'file2' 'file3' 'file4'`

Requirements:

You will be given multiple files containing words. Your task is to count the frequencies of each distinct word across the files, and to produce a file at the end with all the distinct words and the number of times they each appear across all the input files. You will be given 4 text files containing a list of words. Your program will ONLY be tested using 4 threads or less. This means each thread should process at least one file. This section primarily tests on your method of returning the values from multi-threaded function, and then storing the data in an appropriate variable to printed to a text file.

Functions:

Function Name	Arguments	Purpose	Logic Description	Return
Header	none	This function generates a header at the top of screen. Showing the user information about the program and its creator.	A single printf statement, spread over three lines and tabbed in.	none
Clear	none	This function simply clears the screen, to aid in reducing any confusion	This passes the required ASCII characters in via octal values to clear the screen	none
fileLength	char *filename	This function counts the number of lines in the file passed in and returns an integer with the count	The function opens the file using fopen and increments a count value each time fgetc reads in a new line character. The function then returns the count variable.	integer
Addwords	char *filename int start int end	This function reads the words in the files and builds a struct for each one. It then populates all of these into one large global list	The function starts by opening the file with the name passed in. then using the start and end values passed in within a FOR loop, it calls fscanf reading each word as a string and then placing it in the struct array at the corresponding index position.	none
*tSolve	Void *arg	This is the function that is used by each of the threads created.	The function uses the algorithm described in detail in the last section, however it uses the index locations passed in as start and end points for the loop.	none

Notes:

For me this has been one of the most difficult programs in the entire assignment. This second version does use multi-threading. However, the threads manipulate the struct array and do not actually return a value from the thread. This program asks the user how many threads they would like to use, and dynamically allocated elements based upon this variable.

The chunks are used as array indexes for the thread-based solve function.

Mark expectation:
10/17

Part 3: Mini Project

3.1 Negative Filter Single pixel processing

Filename: `Task3-1.c`

Compile command: `cc Task3-1.c -pthread lodepng.c -o Task3-1`

Execute command: `./Task3-1 'original-image.png' 'output-image.png'`

Requirements:

Your program will decode a PNG file into an array and apply a "negative" filter. To do this, you must reverse the RGB values respectively. RGB values range from 0 (min) to 255 (max). For example, if the pixel has the following attributes:
R = 50, G = 100, B = 150, T = 255 (transparency remains the same)
The processed pixel will have the following values:
R = 205, G = 155, B = 105, T = 255 (transparency remains the same)
To gain the full amount of marks, you must allocate memory for the processed image and use POSIX threads to share the computational load.

Functions:

Function Name	Arguments	Purpose	Logic Description	Return
Header	none	This function generates a header at the top of screen. Showing the user information about the program and its creator.	A single printf statement, spread over three lines and tabbed in.	none
Clear	none	This function simply clears the screen, to aid in reducing any confusion	This passes the required ASCII characters in via octal values to clear the screen	none
*tFilterNeg	Void *arg	This function applies a negative filter to the image by reversing the values for RGB	The function takes in a stuct argument that has two variables a start and end integer. These represent the index within the 1D struct pixel array. The function then applies a negative filter to the R,G,B&T values within each struct. The maths used is: new value = 255 – old value	none
*tFilterRed	Void *arg	This function applies a red filter to the image by increasing the red value for each pixel by 50% of the total possible increase remaining	The function takes in a stuct argument that has two variables a start and end integer. These represent the index within the 1D struct pixel array. The function then applies a red filter to the R,G,B&T values within each struct. The maths used is: new value = ((255 – old value)/2)+old value	none
*tFilterGreen	Void *arg	This function applies a green filter to the image by increasing the green value for each pixel by 50% of the total possible increase remaining	The function takes in a stuct argument that has two variables a start and end integer. These represent the index within the 1D struct pixel array. The function then applies a green filter to the R,G,B&T values within each struct. The maths used is: new value = ((255 – old value)/2)+old value	none
*tFilterBlue	Void *arg	This function applies a blue filter to the image by increasing the blue value for each pixel by 50% of the total possible increase remaining	The function takes in a stuct argument that has two variables a start and end integer. These represent the index within the 1D struct pixel array. The function then applies a blue filter to the R,G,B&T values within each struct. The maths used is: new value = ((255 – old value)/2)+old value	none

Notes:

My approach to creating this program involved me creating a Pixel struct that contained the R,G,B&T values read in from the lodepng array. I was able to slice the input array and allocate the correct values into each struct. My logic flow was as follows;

- Define a Pixel Struct
- Define a Block Struct
- Ask the user how many threads they would like to use
- Ask the user the type of filter to apply
- Read in the image passed in as the second argument from the command line
- Calculate the total number of pixels (width*height)
- Dynamically allocate memory for my 1D pixel array using malloc
- Create block structs based upon how many threads were selected
- Slice the image array into the structs inside the 1D Pixel array
- Initialise and set the threads away based upon the selection of filter to apply
- Re-join the threads
- Re-build the image array based on the new pixel values
- Re-encode the png file using the third argument passed in on the command line

Mark expectation

25/25

3.2 Gaussian Blur multi-pixel processing

Filename: `Task3-2.c`

Compile command: `cc Task3-2.c -pthread lodepng.c -o Task3-2`

Execute command: `./Task3-2 'original-image.png' 'output-image.png'`

Requirements:

Your program will decode a PNG file into an array and apply the gaussian blur filter. Blurring an image reduces noise by taking the average RGB values around a specific pixel and setting it's RGB to the mean values you've just calculated. This smoothens the colour across a matrix of pixels. For this assessment, you will use a 3x3 matrix. For example, if you have a 5x5 image such as the following (be aware that the coordinate values will depend on how you format your 2D array):

0,4	1,4	2,4	3,4	4,4
0,3	1,3	2,3	3,3	4,3
0,2	1,2	2,2	3,2	4,2
0,1	1,1	2,1	3,1	4,1
0,0	1,0	2,0	3,0	4,0

The shaded region above represents the pixel we want to blur, in this case, we are focusing on pixel 1,2 (x,y) (Centre of the matrix). to apply the blur for this pixel, you would sum all the **Red** values from the surrounding coordinates including 1,2 (total of 9 R values) and find the average (divide by 9). This is now the new Red value for coordinate 1,2. You must then repeat this for Green and Blue values. This must be repeated throughout the image. If you are working on a pixel which is not fully surrounded by pixels (8 pixels), you must take the average of however many neighbouring pixels there are. To gain the full amount of marks, you must allocate memory for the processed image and use POSIX threads to share the computational load.

Functions:

Function Name	Arguments	Purpose	Logic Description	Return
Header	none	This function generates a header at the top of screen. Showing the user information about the program and its creator.	A single printf statement, spread over three lines and tabbed in.	none
Clear	none	This function simply clears the screen, to aid in reducing any confusion	This passes the required ASCII characters in via octal values to clear the screen	none
*tBlur	Void *arg	This function applies a gaussian blur to the image.	This particular function manipulates all pixels that have at least eight immediately adjacent pixels. The function looks at each of the adjacent pixels and resets the relevant value to the average.	none

Notes:

I was able to reuse a lot of my code from the single pixel filtering in section 3.1. Though the main difference being a created and dynamically allocated memory for two 2D pixel arrays. I then approached this in the same way that I would a jigsaw I was able to first apply the blur to the four corner pixels, then using two loops apply the filter to the top and bottom edges and left and right edges. Leaving the middle to be passed to my threaded function, I passed whole rows to the threads rather than divide them up.

Mark expectation

21/21

Appendix

1.1 Code - Y= MX + C

```
#include <stdio.h>
#include <math.h>

//function declarations
void header ();
void ReadIn ();
void CoordinateOut();
void clear ();
void findValues ();
void formulaOut();

//Instance Variables
float xone, xtwo, yone, ytwo, m, c;

//main method
int main(){
    Header();
    ReadIn();
    clear();
    header();
    CoordinateOut();
    findValues();
    formulaOut();
}

//Function returns header at the top of the page
void header (){
        printf("\n\tDavid Pearson 1725412\n\t5CS021\n\tNumerical Methods in C\n\t1.1 Y=MX+C\n\n");
}

//Function reads in the two sets of co-ordinates
void ReadIn (){
     printf("\tEnter the first value of X = ");
     scanf("%f", &xone);
     printf("\tEnter the first value of Y = ");
     scanf("%f", &yone);
     printf("\tEnter the first value of X = ");
     scanf("%f", &xtwo);
     printf("\tEnter the first value of Y = ");
     scanf("%f", &ytwo);
}

//Function returns co-ordinates entered on two lines
void CoordinateOut(){
     printf("\n\tYou have entered the co-ordinates");
     printf("\n\tFirst Set = (X %.1f, Y %.1f)\n\tSecond Set = (X %.1f, Y %.1f)\n\n", xone, yone, xtwo, ytwo);
}

//Function to clear the screen
void clear(){
     printf("\033[H\033[J");
}

//Function to find the slope or gradient M
void findValues(){
     m = (ytwo-yone)/(xtwo-xone);
     printf("\n\t the slope (M) is %.1f\n" , m);
     c = yone-(m*xone);
     printf("\n\t the constant (C) is %.1f\n" , c);
}

//Function to print out formula
void formulaOut(){
     printf("\n\t Y = %.1fX + %.1f\n", m, c);
}
```

1.2 Code - Quadratic Equation

```
#include <stdio.h>
#include <math.h>

//function declarations
void header ();
void ReadIn ();
void clear ();
void solve ();
void printX ();

//Instance Variables
float a,b,c,xone,xtwo;

//main method
int main(){
        header();
        ReadIn();
        clear();
        header();
        solve();
        printX();
}

//Function returns header at the top of the page
void header (){
        printf("\n\tDavid Pearson 1725412\n\t5CS021\n\tNumerical Methods in C\n\t1.2 Quadratic Equation\n\n");
}

//Function reads in the two sets of co-ordinates
void ReadIn (){
        printf("\tEnter the value for A = ");
        scanf("%f", &a);
        printf("\tEnter the value for  B = ");
        scanf("%f", &b);
        printf("\tEnter the  value for C = ");
        scanf("%f", &c);
}

//Function to clear the screen
void clear(){
    printf("\033[H\033[J");
}

//Function to solve x
void solve(){
        xone = ((b*-1) + sqrt((b*b)-(4*a*c)))/(2*a);
        xtwo = ((b*-1) - sqrt((b*b)-(4*a*c)))/(2*a);
}

//function to print the values of X
void printX(){
        printf("\tX = %.1f  &  %.1f\n", xone,xtwo);
}
```

1.3 Code - Solving Cubic Polynomials

```
#include <stdio.h>
#include <math.h>

//function declarations
void Header ();
void ReadIn ();
void Clear ();
void FirstRoot();
void Solve();
void PrintRoots();

//Instance Variables
double a,b,c,d,R1,R2,R3,n,p;
double lastGuess = 1;
double G = 0;

//creating a boolean type
typedef int bool;
#define true 1
#define false 0

//main method
int main(){
    Header();
    ReadIn();
    Clear();
    Header();
    FirstRoot();
    Solve();
    PrintRoots();
}

//Function returns header at the top of the page
void Header (){
     printf("\n\tDavid Pearson 1725412\n\t5CS021\n\tNumerical Methods in C\n\t1.3 Cubic Polynomials\n\n");
}

//Function reads in the two sets of co-ordinates
void ReadIn (){
     printf("\tEnter the value for A = ");
     scanf("%lf", &a);
     printf("\tEnter the value for  B = ");
     scanf("%lf", &b);
     printf("\tEnter the  value for C = ");
     scanf("%lf", &c);
     printf("\tEnter the  value for D = ");
     scanf("%lf", &d);
}

//Function to clear the screen
void Clear(){
     printf("\033[H\033[J");
}

//Better Function to find the first root
void FirstRoot(){
     while ((float)lastGuess != (float)G && (((a*(G*G*G))+(b*(G*G))+(c*G)+d) != 0)){
          lastGuess = G;
          G = cbrt(-((b*G*G + c*G + d) / a));
     }
     R1 = G;
}

//Function to solve remaining two roots, using quadratic formula
void Solve(){
     p=((d*-1)/R1);
     n=(b+(R1*a));
     R2 = ((n*-1) + sqrt((n*n)-(4*a*p)))/(2*a);
     R3 = ((n*-1) - sqrt((n*n)-(4*a*p)))/(2*a);
}

//function to print the roots
void PrintRoots(){
     printf("\n\tRoot 1 = %.2lf\n\tRoot 2 = %.2lf\n\tRoot 3 = %.2lf\n",R1,R2,R3);
}
```

1.4 Code - Linear Regression

```c
#include <stdio.h>
#include <stdlib.h>

//function declarations
void Header ();
int fileSize();
void LinearFunction();
void Clear ();

//Instance Variables
int    size = 0;
int    ch = 0;
double a,b ;
double xsum = 0;
double ysum = 0;
double xsqsum = 0;
double xysum = 0;
int c1,c2,c3;

//Main Program
int main(int argc, char *argv[] ){
    Clear();
    Header();
    fileSize(argv[1]);
    LinearFunction(argv[1]);
}

//Function returns header at the top of the page
void Header (){
     printf("\n\tDavid Pearson 1725412\n\t5CS021\n\tNumerical Methods in C\n\t1.4 Linear Regression\n\n");
}

//Function to get the size of the file
int fileSize(char *filename){
    FILE *myFile;
    myFile = fopen(filename, "r");
    while(!feof(myFile)){
          ch = fgetc(myFile);
               if(ch == '\n') {
                    size++;
               }
    }
    rewind(myFile);
    return size;
}

//Function to create and populate the arrays and Solve
void LinearFunction(char *filename){
     FILE *myFile;
     myFile = fopen(filename, "r");1
     int xArray[size+1];
     int yArray[size+1];
     for (c1 = 0; c1 <size; c1++) {
          fscanf(myFile, "%d,%d\n", &xArray[c1], &yArray[c1]);
    }
    fclose(myFile);
    for (c2 = 0; c2 <size; c2++) {
          xsum = xsum + xArray[c2];
          ysum = ysum + yArray[c2];
          xsqsum = xsqsum + (xArray[c2] * xArray[c2]);
          xysum = xysum + (xArray[c2] * yArray[c2]);
    }
    int n = size;
    a = ((ysum*xsqsum)-(xsum*xysum)) / ((n*xsqsum)-(xsum*xsum));
    b = ((n*xysum)-(xsum*ysum)) / ((n*xsqsum)-(xsum*xsum));
    double m,c;
    m = b;
    c = a;
    printf("\ty = mx + c\n\ty = %.2lfx + %.2lf\n",m,c);
    double newX,newY;
    printf("\tPlease enter a value for X = ");
    scanf("%lf", &newX);
    newY = (m*newX)+c;
    printf("\ty = %.2f\n",newY);
}

//Function to clear the screen
void Clear(){
     printf("\033[H\033[J");
}
```

2.1 Code - Prime Numbers (part 1)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>

//function declarations
void header ();
void clear ();
int fileLength(char *filename);
int check_prime(long a);
void createNumberList(char *filename, long *list, int length);
void outputNumbers(char *filename,long *list, int length);
void *tSolve (void *arg);

//Structure definition
struct block {
         long start;
         long end;
};

//Variable Declaration
int threads = 0;
long *inList = NULL;
long *outList = NULL;

int main(int argc, char *argv[] ){
         clock_t begin = clock();
         clear ();
         header ();
         printf("\n\tEnter number of threads = ");
         scanf("%d",&threads);
         int A = fileLength(argv[1]);
         printf("\n\tThe length of the file is %d\n", A);
         inList = malloc(sizeof(long)*A);
         outList = malloc(sizeof(long)*A);
         createNumberList(argv[1],inList, A);
         struct block tpoints[threads];
         long chunk = A / threads;
         long start = 0;
         for (int c = 0; c<(threads-1); c++) {
                  tpoints[c].start = start;
                  tpoints[c].end = start + chunk;
                  start += chunk;
         }
         tpoints[threads-1].start = (tpoints[threads-2].end);
         tpoints[threads-1].end = A;
         for (int c = 0; c <threads; c++){
         printf("\n\tthread %d starts at %ld ends at %ld",c,tpoints[c].start,tpoints[c].end);
         }
         pthread_t tIDs[threads];
         pthread_attr_t attr[threads];
         for (int c = 0; c < threads; c++){
                  pthread_attr_init(&attr[c]);
         }
         for (int c = 0; c < threads; c++){
                  pthread_create(&tIDs[c],&attr[c],tSolve,&tpoints[c]);
         }
         for (int c = 0; c < threads; c++){
                  pthread_join(tIDs[c],NULL);
         }
         outputNumbers(argv[2],outList,A);
         int B = fileLength(argv[2]);
         printf("\n\tThe length of the prime number file is %d\n", B);
         clock_t end = clock();
         double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
         printf("\n\ttime taken = %0.4f seconds\n",time_spent);
}

//Function returns header at the top of the page
void header () {
         printf("\n\tDavid Pearson 1725412\n\t5CS021\n\tNumerical Methods in C\n\t2.1 Prime Numbers\n\n");
}
```

2.1 Code - Prime Numbers (part 2)

```
//Function returns header at the top of the page
void header () {
        printf("\n\tDavid Pearson 1725412\n\t5CS021\n\tNumerical Methods in C\n\t2.1 Prime Numbers\n\n");
}

//Function to clear the screen
void clear() {
    printf("\033[H\033[J");
}


//Function to get the length of the file
int fileLength(char *filename){
        int length = 0;
        FILE *myFile;
        myFile = fopen(filename, "r");
        while(!feof(myFile)) {
                int ch = fgetc(myFile);
                        if(ch == '\n'){
                        length++;
                        }
        }
        fclose(myFile);
        return length;
}

//Function to check if input is prime
int check_prime(long a){
        int c;
        if (a < 2) {
                return 0;
        }
        for ( c = 2 ; c <= a - 1 ; c++ ) {
                if ( a%c == 0 ){
                        return 0;
                }
        }
        return 1;
}

//Function to create a list of numbers from the file
void createNumberList(char *filename, long *list, int length) {
        FILE *myFile;
        myFile = fopen(filename, "r");
        for (int c = 0; c < length; c++) {
                fscanf(myFile, "%ld\n", &list[c]);
        }
        fclose(myFile);
}


//Function to write out the file
void outputNumbers(char *filename,long *list, int length){
        FILE *myFile;
        myFile = fopen(filename, "w");
        for (int c = 0; c < length; c++) {
                        if (list[c] != -1) {
                        fprintf(myFile,"%ld\n",list[c]);
                }
        }
        fclose(myFile);
}

//Thread based solve function
void *tSolve (void *arg){
        struct block *tpoints = (struct block*) arg;
        for (long s = tpoints->start; s < tpoints->end; s++) {
                if (check_prime(inList[s]) == 1){
                        outList[s] = inList[s];
                        //printf("\t%ld was prime \n",inList[s]);
                } else{
                outList[s] = -1;
                //printf("\t%ld was not prime \n",inList[s]);
                }
        }
        pthread_exit(0);
}
```

2.2 Code - Word Count not threaded (part 1)

```
// Libraries that need to be loaded
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>

//function declarations
void header ();
void clear ();
int fileLength(char *filename);
void addwords(char* filename, int start, int end);

struct word {
        char word[255];
        int count;
};

struct word *wordStructList;

int main(int argc, char *argv[] ){
        clear();
        header();
        if (argc != 5){
                printf("\tYou have not entered enough arguments\n\tThis requires you to enter 4 file names");
        }
        int A = fileLength(argv[1]);
        printf("\n\t%s contains %d words\n",argv[1], A);
        int B = fileLength(argv[2]);
        printf("\n\t%s contains %d words\n",argv[2], B);
        int C = fileLength(argv[3]);
        printf("\n\t%s contains %d words\n",argv[3], C);
        int D = fileLength(argv[4]);
        printf("\n\t%s contains %d words\n",argv[4], D);

        long totalWords = A+B+C+D;
        printf("\n\tThe total number of words read in is %ld\n", totalWords);

        wordStructList = malloc(sizeof(struct word)*totalWords);

        addwords(argv[1],0,A);
        addwords(argv[2],A,(A+B));
        addwords(argv[3],(A+B),(A+B+C));
        addwords(argv[4],(A+B+C),(A+B+C+D));

        /*method to count the words, logic is:
        * (1) for each word, first look at the count if its the
        *          first time you have seen the word i.e. the count is 0
        *     set the count to 1
        * (2) if the count is -1 then the word has been counted
        *     move onto the next word
        * (3) if the count is greater than 0 this means this is the
        *     word we are incrementing count on
        * (4) now compare this word to every other word in the list
        * (5) If the word you are comparing it to has count of -1
        *     move onto the next word to compare to
        * (6) if the comparing word count is 0 then we want to
        *     compare them
        * (7) Compare the first word with a count of 1 to all
        *      words with a count of 0
        * (8) if they are the same then increment the count by 1
        *     and set the compared words count to -1
        */
        for(int i = 0; i < totalWords; i++){
                if(wordStructList[i].count == -1){
                }
                if(wordStructList[i].count == 0){
                        wordStructList[i].count = 1;
                }
                if(wordStructList[i].count > 0){
                        for(int x = 1; x < totalWords; x++){
                                if(wordStructList[x].count == -1){
                                }
                                if(wordStructList[x].count == 0){
                                        int check = strcmp(wordStructList[i].word,wordStructList[x].word);
                                                if (check==0){
                                                        ++wordStructList[i].count;
                                                        wordStructList[x].count = -1;
                                                }
                                }
                        }
                }
        }
```

2.2 Code - Word Count not threaded (part 2)

```
        int uwords = 0;
        FILE *outFile;
        outFile = fopen("Task2-2Output.txt", "w");
        for (int c = 0; c < totalWords; c++) {
                if (wordStructList[c].count > 0){
                        fprintf(outFile,"%s x %d\n",wordStructList[c].word,wordStructList[c].count);
                        uwords++;
                }
        }
        fclose(outFile);

        printf("\n\tThe number of unique words are %d\n", uwords);
        printf("\n\tResults have been saved in Task2-2Output.txt\n");
}


//Function returns header at the top of the page
void header () {
        printf("\n\tDavid Pearson 1725412");
        printf("\n\t5CS021\n\tNumerical Methods in C");
        printf("\n\t2.2 Word Count\n");
}

//Function to clear the screen
void clear() {
    printf("\033[H\033[J");
}

/*
Function to get the length of the file, by counting
the new line characters
*/
int fileLength(char *filename){
        int length = 0;
        FILE *myFile;
        myFile = fopen(filename, "r");
        while(!feof(myFile)) {
                int ch = fgetc(myFile);
                        if(ch == '\n'){
                        length++;
                        }
        }
        fclose(myFile);
        return length;
}


//Function to add words from the file to the struct word list
void addwords(char* filename, int start, int end){
        FILE *myFile;
        myFile = fopen(filename,"r");
        for(int i = start; i < end; i++){
                fscanf(myFile, "%s", wordStructList[i].word);
                wordStructList[i].count = 0;
        }
        fclose(myFile);
}
```

2.2 Code – Word Count Threaded (part 1)

```
// Libraries that need to be loaded
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>

//function declarations
void header ();
void clear ();
int fileLength(char *filename);
void addwords(char* filename, int start, int end);
void *tSolve (void *arg);

//Struct declarations
struct word {
        char word[255];
        int count;
};

struct block {
        long start;
        long end;
};

//Variable declarations
struct word *wordStructList;
int threads = 0;
long totalWords = 0;

//Main
int main(int argc, char *argv[] ){
        clear();
        header();
        //check correct number of arguments
        if (argc != 5){
                printf("\tYou have not entered enough arguments\n\tThis requires you to enter 4 file names");
        }

        //count each files words and output to user
        int A = fileLength(argv[1]);
        printf("\n\t%s contains %d words\n",argv[1], A);
        int B = fileLength(argv[2]);
        printf("\n\t%s contains %d words\n",argv[2], B);
        int C = fileLength(argv[3]);
        printf("\n\t%s contains %d words\n",argv[3], C);
        int D = fileLength(argv[4]);
        printf("\n\t%s contains %d words\n",argv[4], D);
        totalWords = A+B+C+D;
        printf("\n\tThe total number of words read in is %ld\n", totalWords);

        //ask the user how many threads
        printf("\n\tEnter number of threads = ");
        scanf("%d",&threads);

        //dynamically allocate memory based on how many words
        wordStructList = malloc(sizeof(struct word)*totalWords);

        //add the words from each file to the word list
        addwords(argv[1],0,A);
        addwords(argv[2],A,(A+B));
        addwords(argv[3],(A+B),(A+B+C));
        addwords(argv[4],(A+B+C),(A+B+C+D));
        /*
        create chunks based on number of words and number of threads
        save them as block structs
        */
        struct block tpoints[threads];
        long chunk = totalWords / threads;
        long start = 0;
        for (int c = 0; c<(threads-1); c++) {
                tpoints[c].start = start;
                tpoints[c].end = start + chunk;
                start += chunk;
        }
        tpoints[threads-1].start = (tpoints[threads-2].end);
        tpoints[threads-1].end = totalWords;
        for (int c = 0; c <threads; c++){
        printf("\n\tthread %d starts at %ld ends at %ld",c,tpoints[c].start,tpoints[c].end);
        }
        //thread stuff
        pthread t tIDs[threads];
        pthread_attr_t attr[threads];
        for (int c = 0; c < threads; c++){
                pthread_attr_init(&attr[c]);
        }
        for (int c = 0; c < threads; c++){
                pthread_create(&tIDs[c],&attr[c],tSolve,&tpoints[c]);
        }
        for (int c = 0; c < threads; c++){
                pthread_join(tIDs[c],NULL);
        }
```

2.2 Code – Word Count Threaded (part 2)

```
        int uwords = 0;
        FILE *outFile;
        outFile = fopen("Task2-2Output.txt", "w");
        for (int c = 0; c < totalWords; c++) {
                if (wordStructList[c].count > 0){
                        fprintf(outFile,"%s x %d\n",wordStructList[c].word,wordStructList[c].count);
                        uwords++;
                }
        }
        fclose(outFile);

        printf("\n\tThe number of unique words are %d\n", uwords);
        printf("\n\tResults have been saved in Task2-2Output.txt\n");
}


//Function returns header at the top of the page
void header () {
        printf("\n\tDavid Pearson 1725412");
        printf("\n\t5CS021\n\tNumerical Methods in C");
        printf("\n\t2.2 Word Count\n");
}

//Function to clear the screen
void clear() {
    printf("\033[H\033[J");
}

/*
Function to get the length of the file, by counting
the new line characters
*/
int fileLength(char *filename){
        int length = 0;
        FILE *myFile;
        myFile = fopen(filename, "r");
        while(!feof(myFile)) {
                int ch = fgetc(myFile);
                        if(ch == '\n'){
                        length++;
                        }
        }
        fclose(myFile);
        return length;
}


//Function to add words from the file to the struct word list
void addwords(char* filename, int start, int end){
        FILE *myFile;
        myFile = fopen(filename,"r");
        for(int i = start; i < end; i++){
                fscanf(myFile, "%s", wordStructList[i].word);
                wordStructList[i].count = 0;
        }
        fclose(myFile);
}

//Thread based solve function
void *tSolve (void *arg){
        struct block *tpoints = (struct block*) arg;
        for (long i = tpoints->start; i < tpoints->end; i++) {
                if(wordStructList[i].count == -1){
                }
                if(wordStructList[i].count == 0){
                        wordStructList[i].count = 1;
                }
                if(wordStructList[i].count > 0){
                        for(int x = 1; x < totalWords; x++){
                                if(wordStructList[x].count == -1){
                                }
                                if(wordStructList[x].count == 0){
                                        int check = strcmp(wordStructList[i].word,wordStructList[x].word);
                                                if (check==0){
                                                        ++wordStructList[i].count;
                                                        wordStructList[x].count = -1;
                                                }
                                }
                        }
                }
        }
        pthread_exit(0);
}
```

3.1 Code – Negative Filter Single Pixel Processing (part 1)

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "lodepng.h"

//function declarations
void header ();
void clear ();
void *tFilterNeg (void *arg);
void *tFilterRed (void *arg);
void *tFilterGreen (void *arg);
void *tFilterBlue (void *arg);

//Structure definition
struct pixel {
        unsigned char r; //red value
        unsigned char g; //green value
        unsigned char b; //blue value
        unsigned char t; //transparency value
};
struct block {
        long start;
        long end;
};

//Variables
        unsigned int error;
        unsigned int encError;
        unsigned char* image;
        unsigned int width;
        unsigned int height;
        int numThreads = 0;
        struct pixel *allPixels;
        int filter = 0;

int main(int argc, char **argv ){
        //clear the screen and add the header
        clear ();
        header ();

        //ask the user how many threads
        printf("\n\tEnter number of threads = ");
        scanf("%d",&numThreads);

        clear ();
        header ();
        //ask the user which filter they would like
        printf("\n\tEnter select the filter you would like to apply = ");
        printf("\n\t(1) = negative  : This will reverse the RGB values for each pixel");
        printf("\n\t(2) = Red       : This will increase the R value for each pixel by 50%%");
        printf("\n\t(3) = Green     : This will increase the G value for each pixel by 50%%");
        printf("\n\t(4) = Blue      : This will increase the B value for each pixel by 50%%");
        printf("\n\tEnter a number = ");
        scanf("%d",&filter);

        //read in the file
        error = lodepng_decode32_file(&image, &width, &height, argv[1]);
        if(error){
                printf("error %u: %s\n", error, lodepng_error_text(error));
        }

        //find out how many pixels we are dealing with
        long pixelCount = width*height;

        //create a list of pixel type structs, using malloc to assign the size
        //struct pixel *allPixels=malloc(pixelCount*sizeof(struct pixel));
        allPixels=malloc(pixelCount*sizeof(struct pixel));

        //create threat start and end points based on num of pixels and threads
        struct block tpoints[numThreads];
        long chunk = pixelCount / numThreads;
        long start = 0;
        for (int c = 0; c<(numThreads-1); c++) {
                tpoints[c].start = start;
                tpoints[c].end = start + chunk;
                start += chunk;
        }
        tpoints[numThreads-1].start = (tpoints[numThreads-2].end);
        tpoints[numThreads-1].end = pixelCount;


        //populate the pixel structs from the image list
        for(long i = 0; i < pixelCount; i++){
                allPixels[i].r = image[0+(i*4)];
                allPixels[i].g = image[1+(i*4)];
                allPixels[i].b = image[2+(i*4)];
                allPixels[i].t = image[3+(i*4)];
        }
```

3.1 Code - Negative Filter Single Pixel Processing (part 2)

```
        //thread stuff
        pthread_t tIDs[numThreads];

        //create an array of thread attr's
        pthread_attr_t attr[numThreads];

        //initialise each thread
        for (int c = 0; c < numThreads; c++){
                pthread_attr_init(&attr[c]);
        }
        //set each thread going
        for (int c = 0; c < numThreads; c++){
                if (filter == 1){
                pthread_create(&tIDs[c],&attr[c],tFilterNeg,&tpoints[c]);
                }
                if (filter == 2){
                pthread_create(&tIDs[c],&attr[c],tFilterRed,&tpoints[c]);
                }
                if (filter == 3){
                pthread_create(&tIDs[c],&attr[c],tFilterGreen,&tpoints[c]);
                }
                if (filter == 4){
                pthread_create(&tIDs[c],&attr[c],tFilterBlue,&tpoints[c]);
                }
        }
        //re join all threads
        for (int c = 0; c < numThreads; c++){
                pthread_join(tIDs[c],NULL);
        }
        //re-build the image list
                for(long i = 0; i <pixelCount; i++){
                image[0+(i*4)] = allPixels[i].r;
                image[1+(i*4)] = allPixels[i].g;
                image[2+(i*4)] = allPixels[i].g;
                image[3+(i*4)] = allPixels[i].t;
        }
        //re encode the png file
        encError = lodepng_encode32_file(argv[2], image, width, height);
        if(encError){
                printf("error %u: %s\n", error, lodepng_error_text(encError));
        }
        free(image);
        return 0;
}
//Function returns header at the top of the page
void header () {
        printf("\n\tDavid Pearson 1725412");
        printf("\n\t5CS021\n\tNumerical Methods in C");
        printf("\n\t3.1 Single Pixel\n\n");
}
//Function to clear the screen
void clear() {
    printf("\033[H\033[J");
}
//Thread based solve function
void *tFilterNeg (void *arg){
        struct block *tpoints = (struct block*) arg;
        for (long s = tpoints->start; s < tpoints->end; s++){

                allPixels[s].r = 255-allPixels[s].r;
                allPixels[s].g = 255-allPixels[s].g;
                allPixels[s].b = 255-allPixels[s].b;
        }
        pthread_exit(0);
}
//Thread based solve function
void *tFilterRed (void *arg){
        struct block *tpoints = (struct block*) arg;
        for (long s = tpoints->start; s < tpoints->end; s++){

                allPixels[s].r = (((255-allPixels[s].r)/2)+allPixels[s].r);
        }
        pthread_exit(0);
}
//Thread based solve function
void *tFilterGreen (void *arg){
        struct block *tpoints = (struct block*) arg;
        for (long s = tpoints->start; s < tpoints->end; s++){

                allPixels[s].g = (((255-allPixels[s].g)/2)+allPixels[s].g);
        }
        pthread_exit(0);
}
//Thread based solve function
void *tFilterBlue (void *arg){
        struct block *tpoints = (struct block*) arg;
        for (long s = tpoints->start; s < tpoints->end; s++){

                allPixels[s].b = (((255-allPixels[s].b)/2)+allPixels[s].b);
        }
        pthread_exit(0);
}
```

3.2 Code – Gaussian Blur Multi-Pixel Processing (part 1)

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "lodepng.h"

//function declarations
void header ();
void clear ();
void *tBlur(void *arg);

//Structure definition
struct pixel {
          unsigned char r; //red value
          unsigned char g; //green value
          unsigned char b; //blue value
          unsigned char t; //transparency value
};
struct block {
          long start;
          long end;
};
//Variables
          unsigned int error;
          unsigned int encError;
          unsigned char* image;
          unsigned int width;
          unsigned int height;
          int numThreads = 0;
          struct pixel **OI;
          struct pixel **MI;
int main(int argc, char **argv ){
          //clear the screen and add the header
          clear ();
          header ();
          //ask the user how many threads
          printf("\n\tEnter number of threads = ");
          scanf("%d",&numThreads);
          clear ();
          header ();
          //read in the file
          error = lodepng_decode32_file(&image, &width, &height, argv[1]);
          if(error){
                    printf("error %u: %s\n", error, lodepng_error_text(error));
          }
          //find out how many pixels we are dealing with
          long pixelCount = width*height;
          OI=malloc((sizeof(struct pixel)*height)*width);
                    for (int i = 0; i < height; i++){
                              OI[i] = malloc(sizeof(struct pixel)*width);
                    }
          MI=malloc((sizeof(struct pixel)*height)*width);
                    for (int i = 0; i < height; i++){
                              MI[i] = malloc(sizeof(struct pixel)*width);
                    }
          //populate the pixel structs from the image list
          for(long row=0; row < height; row++){
                    for(long col=0; col < width; col++){
                              OI[row][col].r = image[0+(col*4)+(4*(width*row))];
                              OI[row][col].g = image[1+(col*4)+(4*(width*row))];
                              OI[row][col].b = image[2+(col*4)+(4*(width*row))];
                              OI[row][col].t = image[3+(col*4)+(4*(width*row))];
                    }
          }
//BLUR
//top left
MI[0][0].r = ((OI[0][0].r + OI[0][1].r + OI[1][0].r + OI[1][1].r)/4);
MI[0][0].g = ((OI[0][0].g + OI[0][1].g + OI[1][0].g + OI[1][1].g)/4);
MI[0][0].b = ((OI[0][0].b + OI[0][1].b + OI[1][0].b + OI[1][1].b)/4);
MI[0][0].t = ((OI[0][0].t + OI[0][1].t + OI[1][0].t + OI[1][1].t)/4);
//bottom left
MI[height-1][0].r = ((OI[height-1][0].r + OI[height-1][1].r + OI[height-2][0].r + OI[height-2][1].r)/4);
MI[height-1][0].g = ((OI[height-1][0].g + OI[height-1][1].g + OI[height-2][0].g + OI[height-2][1].g)/4);
MI[height-1][0].b = ((OI[height-1][0].b + OI[height-1][1].b + OI[height-2][0].b + OI[height-2][1].b)/4);
MI[height-1][0].t = ((OI[height-1][0].t + OI[height-1][1].t + OI[height-2][0].t + OI[height-2][1].t)/4);
//top right
MI[0][width-1].r = ((OI[0][width-1].r + OI[0][width-2].r + OI[1][width-1].r + OI[1][width-2].r)/4);
MI[0][width-1].g = ((OI[0][width-1].g + OI[0][width-2].g + OI[1][width-1].g + OI[1][width-2].g)/4);
MI[0][width-1].b = ((OI[0][width-1].b + OI[0][width-2].b + OI[1][width-1].b + OI[1][width-2].b)/4);
MI[0][width-1].t = ((OI[0][width-1].t + OI[0][width-2].t + OI[1][width-1].t + OI[1][width-2].t)/4);
//bottom right
MI[height-1][width-1].r = ((OI[height-1][width-1].r + OI[height-1][width-2].r + OI[height-2][width-1].r + OI[height-2][width-2].r)/4);
MI[height-1][width-1].g = ((OI[height-1][width-1].g + OI[height-1][width-2].g + OI[height-2][width-1].g + OI[height-2][width-2].g)/4);
MI[height-1][width-1].b = ((OI[height-1][width-1].b + OI[height-1][width-2].b + OI[height-2][width-1].b + OI[height-2][width-2].b)/4);
MI[height-1][width-1].t = ((OI[height-1][width-1].t + OI[height-1][width-2].t + OI[height-2][width-1].t + OI[height-2][width-2].t)/4);

//top and bottom edge
for (long c = 1; c < width-1; c++){
//top
MI[0][c].r = ((OI[0][c-1].r + OI[0][c].r + OI[0][c+1].r + OI[1][c-1].r + OI[1][c].r + OI[1][c+1].r)/6);
MI[0][c].g = ((OI[0][c-1].g + OI[0][c].g + OI[0][c+1].g + OI[1][c-1].g + OI[1][c].g + OI[1][c+1].g)/6);
MI[0][c].b = ((OI[0][c-1].b + OI[0][c].b + OI[0][c+1].b + OI[1][c-1].b + OI[1][c].b + OI[1][c+1].b)/6);
MI[0][c].t = ((OI[0][c-1].t + OI[0][c].t + OI[0][c+1].t + OI[1][c-1].t + OI[1][c].t + OI[1][c+1].t)/6);
//bottom
MI[height-1][c].r = ((OI[height-1][c-1].r + OI[height-1][c].r + OI[height-1][c+1].r + OI[height-2][c-1].r + OI[height-2][c].r + OI[height-2][c+1].r)/6);
MI[height-1][c].g = ((OI[height-1][c-1].g + OI[height-1][c].g + OI[height-1][c+1].g + OI[height-2][c-1].g + OI[height-2][c].g + OI[height-2][c+1].g)/6);
MI[height-1][c].b = ((OI[height-1][c-1].b + OI[height-1][c].b + OI[height-1][c+1].b + OI[height-2][c-1].b + OI[height-2][c].b + OI[height-2][c+1].b)/6);
MI[height-1][c].t = ((OI[height-1][c-1].t + OI[height-1][c].t + OI[height-1][c+1].t + OI[height-2][c-1].t + OI[height-2][c].t + OI[height-2][c+1].t)/6);
}
```

3.2 Code – Gaussian Blur Multi-Pixel Processing (part 2)

```
//left and right edge
          for (long c = 1; c < height-1; c++) {
          //left
          MI[c][0].r = ((OI[c-1][0].r + OI[c][0].r + OI[c+1][0].r + OI[c-1][1].r + OI[c][1].r + OI[c+1][1].r)/6);
          MI[c][0].g = ((OI[c-1][0].g + OI[c][0].g + OI[c+1][0].g + OI[c-1][1].g + OI[c][1].g + OI[c+1][1].g)/6);
          MI[c][0].b = ((OI[c-1][0].b + OI[c][0].b + OI[c+1][0].b + OI[c-1][1].b + OI[c][1].b + OI[c+1][1].b)/6);
          MI[c][0].t = ((OI[c-1][0].t + OI[c][0].t + OI[c+1][0].t + OI[c-1][1].t + OI[c][1].t + OI[c+1][1].t)/6);
          //right
          MI[c][width-1].r = ((OI[c-1][width-1].r + OI[c][width-1].r + OI[c+1][width-1].r + OI[c-1][width-2].r + OI[c][1].r + OI[c+1][width-2].r)/6);
          MI[c][width-1].g = ((OI[c-1][width-1].g + OI[c][width-1].g + OI[c+1][width-1].g + OI[c-1][width-2].g + OI[c][1].g + OI[c+1][width-2].g)/6);
          MI[c][width-1].b = ((OI[c-1][width-1].b + OI[c][width-1].b + OI[c+1][width-1].b + OI[c-1][width-2].b + OI[c][1].b + OI[c+1][width-2].b)/6);
          MI[c][width-1].t = ((OI[c-1][width-1].t + OI[c][width-1].t + OI[c+1][width-1].t + OI[c-1][width-2].t + OI[c][1].t + OI[c+1][width-2].t)/6);
          }

          //create threat start and end points based on num of pixels and threads
          struct block tpoints[numThreads];
          long chunk = height / numThreads;
          long start = 1;
          for (int c = 0; c<(numThreads-1); c++) {
                    tpoints[c].start = start;
                    tpoints[c].end = start + chunk;
                    start += chunk;
          }
          tpoints[numThreads-1].start = (tpoints[numThreads-2].end);
          tpoints[numThreads-1].end = height-1;

          //create an array of thread id's
          pthread_t tIDs[numThreads];
          //create an array of thread attr's
          pthread_attr_t attr[numThreads];
          //initialise each thread
          for (int c = 0; c < numThreads; c++){
                    pthread_attr_init(&attr[c]);
          }
          //set each thread going
          for (int c = 0; c < numThreads; c++){
                    pthread_create(&tIDs[c],&attr[c],tBlur,&tpoints[c]);
                    }
          //re join all threads
          for (int c = 0; c < numThreads; c++){
                    pthread_join(tIDs[c],NULL);
          }
          for(long row=0; row < height; row++){
                    for(long col=0; col < width; col++){
                              image[0+(col*4)+(4*(width*row))] = MI[row][col].r;
                              image[1+(col*4)+(4*(width*row))] = MI[row][col].g;
                              image[2+(col*4)+(4*(width*row))] = MI[row][col].b;
                              image[3+(col*4)+(4*(width*row))] = MI[row][col].t;
                    }
          }
          //re encode the png file
          encError = lodepng_encode32_file(argv[2], image, width, height);
          if(encError){
                    printf("error %u: %s\n", error, lodepng_error_text(encError));
          }
          free(image);
          return 0;

}

//Function returns header at the top of the page
void header () {
          printf("\n\tDavid Pearson 1725412");
          printf("\n\t5CS021\n\tNumerical Methods in C");
          printf("\n\t3.2 Multi Pixel\n\n");
}

//Function to clear the screen
void clear() {
    printf("\033[H\033[J");
}

//Thread based solve function
void *tBlur(void *arg){
          struct block *tpoints = (struct block*) arg;
          for (long i = tpoints->start; i < tpoints->end; i++){

for (long c = 1; c < width-1; c++){
                              MI[i][c].r = ((OI[i-1][c-1].r + OI[i-1][c].r + OI[i-1][c+1].r + OI[i][c-1].r + OI[i][c].r + OI[i][c+1].r + OI[i+1][c-1].r
+ OI[i+1][c].r + OI[i+1][c+1].r)/9);
                              MI[i][c].g = ((OI[i-1][c-1].g + OI[i-1][c].g + OI[i-1][c+1].g + OI[i][c-1].g + OI[i][c].g + OI[i][c+1].g + OI[i+1][c-1].g
+ OI[i+1][c].g + OI[i+1][c+1].g)/9);
                              MI[i][c].b = ((OI[i-1][c-1].b + OI[i-1][c].b + OI[i-1][c+1].b + OI[i][c-1].b + OI[i][c].b + OI[i][c+1].b + OI[i+1][c-1].b
+ OI[i+1][c].b + OI[i+1][c+1].b)/9);
                              MI[i][c].t = ((OI[i-1][c-1].t + OI[i-1][c].t + OI[i-1][c+1].t + OI[i][c-1].t + OI[i][c].t + OI[i][c+1].t + OI[i+1][c-1].t
+ OI[i+1][c].t + OI[i+1][c+1].t)/9);
                    }
          }
          pthread_exit(0);
}
```