Assignment 2 report

Relevant concepts

- **Argument parsing**. In Assignment 0 we learnt how to parse arguments. In particular, it was mentioned the standard solution POSIX getopt, and we will use this solution rather than our own one to parse the arguments.

- **Memory types**. In Assignment 0 we were introduced to two types of allocated memory: stack and heap memory. Allocated stack memory is acquired by using array declarations inside functions, and is limited in size, but tends to be faster than heap allocated memory, which can be allocated with global declaration of arrays and its size is not as limited as the arrays allocated in the stack. Besides, stack allocated memory is thread local, i.e. it is not shared between threads, therefore we plan to use stack allocated memory in the most repetitive and less memory required operations within a given thread, such as the Newton iteration for an individual point.

- **Contiguous and non-contiguous memory, allocation and freeing** In Assignment 0 we learnt that when we allocate memory on the heap there is no guarantee about where it is positioned, and a common strategy to this is to allocate the memory required in large blocks of contiguous memory. Contiguous memory is typically faster, but less flexible as it is required to allocate all the memory at once. Non-contiguous memory can be allocated and freed whenever it is required. We plan to allocate the memory in the heap only before using the required array in the computation and freeing it as soon as we are not using it anymore to avoid memory leaks and have free memory for other computations in case we have a large image size, hence we will use non-contiguous memory. The choice of non-contiguous memory can be reviewed if the performance is not as good as expected.

- **Writing to file are expensive operations**. In the lectures, we have discussed the memory model of modern computers. In particular, we it has been mentioned that the there is more latency when accessing data, in ascending order, for the following memory levels: L1, L2, L3 caches, RAM and storage disks. The order is the opposite when the capacity of each level is considered. Therefore, accessing the storage disks should be minimised if we want to achieve the best performance, i.e. writing to file operations should be minimised.

  In Assignment 0 it was suggesteed that we can prepare a string in memory, for example using `sprintf`, and write it to file using `fwrite`, which is said to be faster than `fprintf`, even if we provide the formatted string to the latter. Hence, we plan to format each row of the image to be written in a string, and once we have the complete string, write it to file with `fwrite`, therefore only running `fwrite` as many times as rows of the image.

  For each result, whether it is the attractor for each point in the plane or the iterations used, we will represent it with a colour. These colours can be precomputed and stored, as will be finite and known, already as strings. Therefore, if we have an array of results and the string to be written to file defined, for each result we only need to access its corresponding colour string and append it to the string to be written to file, thus we plan to use this technique for a good performance.

- **Threads, mutexes and thread synchronisation**. In the lectures we have been introduced to threads and mutexes, and how heap memory allocated can be accessed from different threads. Our approach to store the results is to allocate a matrix in the heap for each type of result (attractor and convergence), by defining an array of pointers, where each pointer will point to an array representing one row of pixels in the image to be produced. We will also create a status array to indicate whether a row of the results matrix has been computed or not. We will have the main thread, a thread dedicated to writing and threads dedicated to computing the Newton method for each row. Each computing thread will be instructed to compute one row of the results matrix (different for each thread) before it starts computing another row. We will use mutexes to safekeeping the status array, acquiring the lock before changing the status and releasing it after, as well as acquiring a lock before writing the results to file, freeing the results row written and releasing the lock after to avoid freeing memory that has to be accessed later.

  Since the results matrices are allocated in the heap, we can access it from different threads. We will take advantage of this by accessing the result matrices computed with the compute threads from the write thread when it is indicated by the status array. Hence, we will write to file the the rows of the results matrices one at a time, which will be accomplished by acquiring a lock before the write to only access the file pointer sequentially and releasing it after having written to file.

  It can happen that the results rows are not completed in a friendly order for writing: for example, if we have 3 threads and 3 rows, ideally the first row will be computed first and written to file, followed by the second and the third, but in reality the first row might be slower than the second, and the third, so if the write is naively implemented it will write to file as a first row the actual second row, and so on. We can make use of `fseek` to correctly position the file pointer to the appropiate position in the file after computing the correct offset. In this way, we can have independence between the computation threads and the write thread.

The above list concepts that are intended to be used in the program. Tools discussed in previous assignments, such as `valgrind --tool=memcheck` and the `gdb` debugger will be used to check for memory leaks and bugs if required, and also the flag `-Wall` will be used while developing to catch all warnings.

Intended program layout

The program naturally splits in the following tasks:

- Reading the arguments
- Calculating the roots of the function
- Preparing the colours for the PPM files
- Finding the attractor roots
  - Calculations for Newton's method
- Splitting up computation between threads
- Write the results to PPM files

The program will use dedicated threads. There will be one thread dedicated to writing to file, threads for computing the attractors, whose amount will be specified as an input argument, and the main thread.

Reading the arguments

To read the arguments we will implement a function that makes use of `getopt`, so that it automatically accepts the two following situations:

- Independency of whitespace: `-l100` and `-l 100` are the same
- Independency of argument placing: `-l100 -t5 5`, `-t5 -l10 5`, `-l100 5 -t5`, `5 -l100 -t5`, `5 -t5 -l100` result the same

This function will assign to global variables the values of length of each row (and column) `l` representing the complex plane `[-2,2]x[-2i,2i]`, the number of computation threads `t`, and the degree `d` of the function `x^d - 1`.

This task of the program will be executed in the main thread before any other task is run.

Calculating the roots of the function

The roots of the function `x^d - 1`, where `d > 0`, are the roots of unity (also called de Moivre numbers). If we allow `x` to be complex, we have that there are `d` roots given by the formula

```
cos(2*k*pi/d) + i*sin(2*k*pi/d), k = 0, 1, ..., d-1.
```

We will implement a function that takes the degree `d`, computes all the roots of unity for that degree, and stores them in a global variable array of length `d`. This array can be used in the compute threads to check if the convergence threshold is greater than the absolute difference between the iteration point and any of the roots.

This task of the program will be executed in the main thread after parsing the arguments and before creating any of the computation threads.

Prepare colours for the PPM files

The expected output are two PPM files, one image showing the attractor root for each point in the complex plane, referred as the "attractor image", and one image showing the iterations used, referred as the "convergence image". The former will be in RGB, and the latter in grayscale, therefore this task has two subtasks.

Before explaining the two subtasks, let's have a look at the PPM files. Our PPM files will have the following header:

```
P3
Lc Lr
C
```

where both `Lc` and `Lr` are integers and will be replaced by the value of the argument `l`, and `C` is an integer and will be different for the RGB and grayscale images.

In a PPM file we define each pixel as the RGB triplet `r g b`, being `r`, `g`, `b` the values of reed, green, and blue for each pixel, respectively, where `0 <= r < C`, and analogously for `g` and `b`. Note that if `C` is of length 2, i.e. if `C` is between 10 and 99. This is of importance because the numbers `r`, `g`, and `b` must have the same length as `C`. For instance, if `C` is 10, and we want to have a value of 0 or 1 for red, these values must be written as `00` and `01`, respectively, in the PPM file. The length of an integer can be found by different methods, for instance using math functions or recursive functions.

For the attractor image we will need `d+1` colours, being `d` the degree of the function `x^d - 1`. The reason behind this is that each root is represented with a different colour. As Newton iteration does not converge for all points in our complex plane, we will assign the same additional root to all the points that do not converge.

To obtain `d+1` different colours we will use an algorithm such as this one (source):

```
b = a % C
g = ((a-b)/C) % C
r = ((a-b)/C**2) - g/C
```

where `r`, `g`, `b` are the red, green and blue values and `a` is the root index number, i.e. `a = 0,..., d`.

Combining this concepts we will implement an algorithm that stores in a global array for each image the colours to be used in the write to file task. This algorithm will take as inputs the number of colours to produce (`d+1`) and whether we want to obtain grayscale colours or not, as the attractors image is coloured and the convergence is in grayscale. For the attractors image, `C` will be computed as the smallest integer satisfying `d+1 <= C^3` for the attractors cas, as it is permutations with repetitions from combinatorics. In the grayscale case, i.e. in the convergence image, `r = g = b`, and `C = MAX iterations+1`. Then, for each colour we will compute the corresponding RGB string, printing leading 0's if imposed by the `C` value, and save it to the array of colours. This task will be completed before any writing to file takes place.

Until now, all the tasks are planned to be executed only once and therefore we do not expect to have a big impact on the performance of the program

Find attractors

For a given point in complex plane we are supposed to find out to which root it converges, i.e. the attractor. To do this we need to apply Newton's method and check if the new point we obtain is one of the roots or if it is close enough to it. If the new point fulfils the convergence criteria for a root then we set that root as the attractor for that point. Otherwise we repeat the previous step with each new point until an attractor is found.

The intuative way to find out if a point is close enough to a root is calculating the difference between the point and the root, take the absolute value of the difference and see if it is below a threshold. This is how it currently implemented in our code. However, calculating the absolute value of a complex number requires a square root calculation which is slow. Consequently, instead of `cabs(a+bi) < c` we will use `a^2 + b^2 < c^2`.

Here is a pseudocode:

```
 attractor = No root
convergence_counter = 0
point = a given point in complex plane

while(iteration < max_iterations)
{
    iteration++
    point = Newtons_method_calculation(point)

    for( each root of function f(x) )
    {

        if( abs( point - root ) < tolerance )
        {
            attractor = root
            convergence_counter = iteration

            return attractor, iteration
        }

        if( abs( point ) < tolerance      ||
            abs( point.real ) > maxNumber ||
            abs( point.imag ) > maxNumber   )
        {
            attractor = No root
            convergence_counter = iteration

            return attractor, iteration
        }
    }
}
```

It is similar to the Wikipedia pseudocode which was used as a basis.

Calculations for Newton's method

We plan to separate the main calculation for Newton's method into its own function for better readability and to make it simpler to optimise. Before we explain the implementation we need to state the equation for one iteration of Newton's method.

```
new x = x - f(x) / f'(x)
```

The functions we are exploring are `f(x) = x^d - 1` with varying `d` and the iteration will look like the following:

```
new x = x - ( x^d - 1 ) / ( dx^(d-1) )
```

Since we are going to be running this part very often in our program we want it to be as efficent as possible. To calculate powers of `x` it possible to use `cpow` but it can be slow. We will start with simply hardcoding the multiplication for degrees from 1 to 10. An example is shown below.

```
// If we want to calculate x^9
double complex x = given complex number;
double complex temp = 0;

temp = x * x;       // temp = x^2
temp = temp * temp; // temp = x^4
temp = temp * temp; // temp = x^8
temp = temp * x;    // temp = x^9
```

Calculating a power of complex number often is something we want to avoid if possible. There are two ways to go about this: rewriting the equation or calculating `x^(d-1)` and storing it memory for calculating `x^d`. The equation can be rewritten as: `new x = x - x/d + 1/(dx^(d-1))`. Rewriting the equation like this saves on calculating `x` in multiple powers. Storing `x^(d-1)` in memory and using it to calculate `x^d` is a simple optimisation method that may be overlooked. We will implement both methods and pick the one that performs best.

Splitting up computation between threads

We are working with the complex plane bounded from -2 to 2 on the real axis and -2i to 2i on the imaginary axis. We want to generate an image of attractors and convergence based on this square with granularity specified by the user with input `l`. From the input `l` we would like to create a `l * l` matrix and calculate the attractor and convergence for element (complex number) in the matrix. Since each element is independant we do not need to worry about race conditions if we divide the work among threads.

We create two dynamically allocated pointer arrays with global reference in the main thread, one for attractors and one for convergences. These arrays will point to arrays which will represent rows of the matrix. See example below.

```
int** matrix = malloc(sizeof(*int) * l);

for( int row = 0; row < l ; row++ )
{
    matrix[row] = malloc(sizeof(int) * l);
}
```

However unlike the example we will not allocate the row arrays in the main threads. We will leave that to the calculation threads for reasons that we will explain later. We will also allocate a status array and create a mutex for its safekeeping. This array will keep track of which rows in the attractor matrix and convergence matrix have been computed.

Afterwards, we create the calculation threads in the main thread and give them each an identifier from `0` to `t-1` (where `t` is number of calculation threads). Each thread will create an array of length `l` for each of the

matrices and insert into a row corresponding to their identifier. Next, the thread will populate the arrays with the values obtained from Newton's method. When that is finished it will lock the mutex, set corresponding entry in the status array to ready and unlock the mutex. Then the thread will jump `t` rows and repeat the sequence of actions for that row and so on.

This spreads the load roughly equally among threads and ensures that all rows have an assigned thread. The reason why we decided on non-contigous memory for the matrices is that the write thread will free the rows when it has finished writing it to file. That recently freed memory in heap can then be used when the a new row is allocated. We expect this to use less heap memory.

```
for( int row = thread_id; row < l; row += t)
{
    int* array = malloc(sizeof(int) * l);         // Simplified with only one matrix

    for (int col = 0; col < l; col++)
    {
        calculate attractor and convergence for entry (row, col)
        insert attractor and convergence into array[col]
    }

    matrix[row] = array;

    grab mutex

    status_array[row] = ready;

    release mutex
}
```

Writing the results to file

The structure of the PPM files to be written has been discussed before. Since the header of the PPM files is small and only has to be written once for each file, we will use `fprintf` to write it to file to avoid preparing a short string which will be used only once, as we think that it will not impact performance.

Previously, we have defined the colours to be used for each root, therefore once the status array signals that a row is ready, we can produce its colour string, for example for the attractors matrix, following this C pseudocode:

```
// Allocate enough space for the string
char attractor_row_str[10*l];

// Place the first pixel manually
int root_i = attractor_matrix[row][0];
strcpy(attractor_row_str, attractor_colours[root_i]);
// Separate RGB values from next pixel
strcat(attractor_row_str, " ");
for ( j = 1; j < l-1; ++j)
{
    // Repeat previous string concatenation
    root_i = attractor_matrix[row][j];
    strcat(attractor_row_str, attractor_colours[root_i]);
    strcat(attractor_row_str, " ");
}
// Place last pixel manually and a new line
root_i = attractor_matrix[row][l-1];
strcat(attractor_row_str, attractor_colours[root_i]);
strcat(attractor_row_str, "\n");
```

Therefore, the task to write to file can be implemented following this C pseudocode:

```
 // Acquire the lock
pthread_mutex_lock(&mutex);
if (status_array[row] == 1)
{
    ////////////////////////////////////////
    // Row string production
    ////////////////////////////////////////

    ////////////////////////////////////////
    // If required, position the file_pointer
    // here according to row with using
    // fseek and ftell if needed
    ////////////////////////////////////////

    // Write to file
    fwrite(attractor_row_str, sizeof(char),
           strlen(attractor_row_str), file_pointer);

    // Free the attractor_matrix row
    // as it will not be used anymore
    free(attractor_matrix[row]);

    // Prevent writing the same row again
    status_array[row] = -1
}

// Release the lock
pthread_mutex_unlock(&mutex_ready);
```

Note that we are freeing the attractor matrix row written to file here as previously explained. The writing to the convergence image will be done in the same fashion, and actually implemented in the same place.