Relevant concepts

- **Stack allocation** There are two types of allocated memory: Stack and heap allocated memory. Memory allocated on the stack is limited in size, but tends to be faster. Moreover, stack allocated memory is thread local and therefore provides an opportunity to untangle the mutual impact of parallel threads on one another. Thus, it is an important consideration to employ stack allocated memory in the innermost iteration steps, i.e., the Newton iteration for an individual point on the complex plane.
- **POSIX Threads** These allow us to parallelize the code by performing simultaneous computations resulting in faster processing. Various number of threads work independently sharing the same memory while being in the same process. By implementing POSIX Threads, we run Newtons iterations on various initializations in parallel, which otherwise would have taken significant amount of time.

Intended program layout

Per instructions we split the program into two broad subtasks: 1. The computation of the Newton iteration 2. The writing of results to the two output files.

Each of them will be implemented in a separate function, that are intended to be run as the main function of corresponding POSIX threads.

1. Computation of Newton Iteration

The computation of the Newton iteration can be further split up into : - Initializing the starting points of Newtons Iteration - Pre-computing the roots for all degree of the polynomial - Computation of of x_n in the iteration step - Implementing Iteration Checks

Initializing the starting points of Newtons Iteration

Based on the arguments passed through the command line, we get values for number of rows and columns for the output picture (-l), the number of threads to be used for computation (-t) and the degree of the polynomial. Example:

```
 ./newton -t5 -l1000 7
./newton -l1000 -t5 7
```

Based on the -l argument, we divide the real and imaginary plane between +2 and -2, into a grid of `lxl` lines, with each intersection representing an iteration point. These points are loaded into the program in order to iterate across each of these points and calculate the attractor and convergence for the same.

Pre-computing the roots for all degree of the polynomial

For a given polynomial of the form `x^d-1`, where d is the degree of the polynomial, we can pre-calculate all the roots of the polynomial and store it in a complex array of size `[degree-1]`. With the help of a switch statement, depending on the degree of the polynomial, we can load all the roots of the polynomial into the array.

```
     switch(degree)
    {
        case 1:
            //STATEMENTS FOR DEGREE 1
            roots[0] = <Pre-calculated root value a + ib>;
            break;
        case 2:
            //STATEMENTS FOR DEGREE 2
            roots[0] = <Pre-calculated root value a + ib>;
            roots[1] = <Pre-calculated root value a + ib>;
            break;
      //and so on for all possible degrees
      default:
          fprintf(stderr, "unexpected degree\n");
          exit(1);
    }
```

Computation of of x_n in the iteration step

Next, we need to implement the computation by finding an expression for the iteration step that is efficient. Since using `cpow()` is not an option, we naively implement the Newton iteration step in the form `x - (x^ d-1)/(d*x^(d-1))`, with repeated multiplication.

With the help of a switch statement, depending on the degree of the polynomial, we can calculate the value of the next iteration as follows :

```
switch ( degree )
    {
    case 1:
      //STATEMENTS FOR DEGREE 1
      x_k1 = xk - (xk-1);
      break;
    case 2:
      //STATEMENTS FOR DEGREE 2
      x_k1 = xk - (((xk*xk)-1)/(2*xk));
      break;
    //Similarly hardocded formulas for all degrees less than 10
    default:
      fprintf(stderr, "unexpected degree\n");
      exit(1);
    }
```

Implementing Iteration Checks

In addition, we need to implement iteration checks that need to be run for each point as long as the conditions are met. Else, the iteration needs to be aborted : - **If the iteration step is closer than 10^-3 to one of the roots of f(x), then abort the iteration.** - We need to loop through all the roots and verify if the absolute value of `iteration step value - roots` is within the required limit.

- **If x_i is closer than 10^-3 to the origin, then abort the iteration.**
    - We need to loop through all the roots and verify if the absolute value o `iteration step value` is within the required limit.
- **If the absolute value of its real or imaginary part is bigger than 10 ^10, then abort the iteration**
    - We access the real and imaginary parts of the complex number via `creal()` and `cimag()`

2. Writing of results to output files

As for the writing to file, we have identified 3 independent subtasks : - Configuring the ppm files (newton_attractors_xD.ppm and newton_convergence_xD.ppm) - Opening ppm file to write the colors and grayscales

Configuring the ppm files (newton_attractors_xD.ppm and newton_convergence_xD.ppm)

As mentioned in Assignment 2, in order to produce recognizable quality pictures, the number of iterations we run for the newtons method should be `>= 50`. Additionally, we must use one main thread to write to disc.

Based on the degree of polynomial and number of roots, we assign RGB Values of different colors that correspond to each of those roots. Also, based on the cap on the number of iterations chosen, we assign grayscale equivalent RGB Values for each iteration.

Opening ppm file to write the colors and grayscales

Once the newton iterations are performed and we have the corresponding attractor and convergence values, we write every row calculated into the ppm file. The different colors of the roots would yield a newtons fractal pattern in newton_attractors_xD.ppm, denoting the root which it converges to. For the grayscale ppm, depending on the number of iterations for which the roots converge, we will get a corresponding grayscale value in the Newton_convergence_xD.ppm file.