# Assignment 2: Newton

Names redacted for the sake of collaborator

November 24, 2022

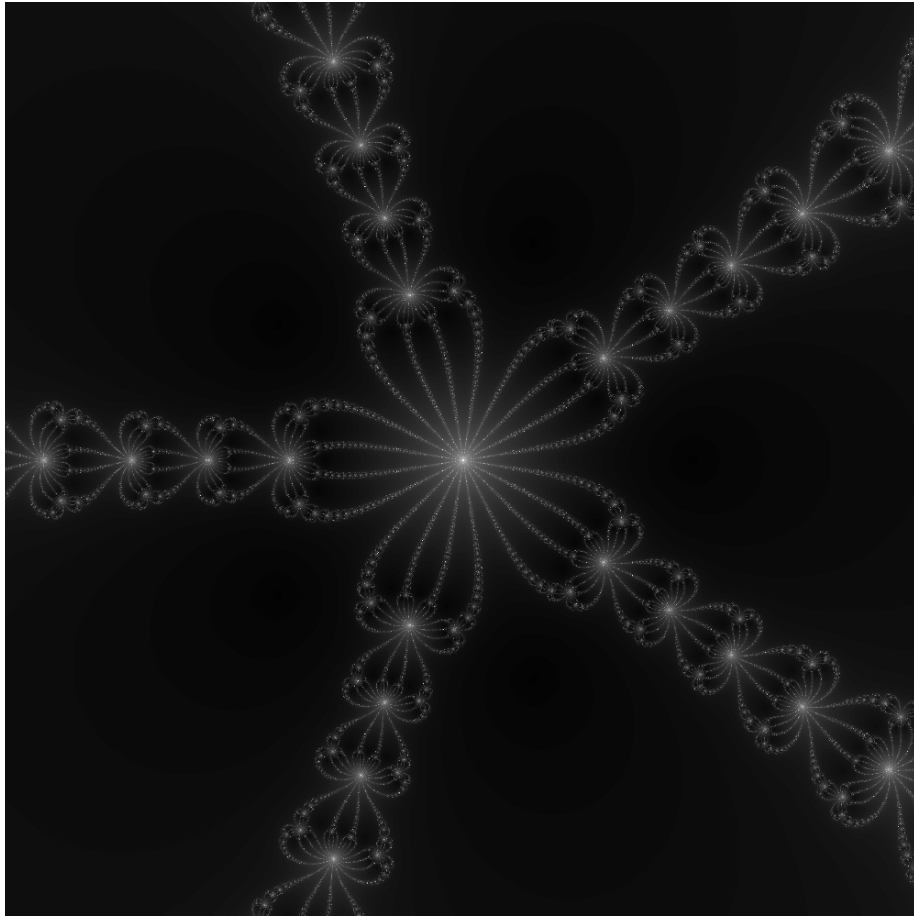

Figure 1: Newton fractal showing convergence for $z^5 = 1$.

# Introduction

Generating a Newton fractal is a massively parallel endeavor. Through cleverness, mathematical cunningness and laziness we are able to beat the example program by a large margin and beat all time limits with *only one thread*. This report will first describe the overall structure of the program and then focus on the most important optimizations and mathematical insights.

# Program layout

The main thread will start up one printer thread and $n$ worker threads. Once the threads are started the main thread does nothing, no coordination of workers is done at all.

In rough terms,

- the printer will print row by row as soon as there is data to print,
- every worker will process one row at a time (evaluate Newton's method for all points in that row) and upon completion will pick up the next unprocessed row.

## Threading and shared data

There are several pieces of shared data used in the program, most importantly

**`**iter` and `**root_id`** will be filled with row by row results from the computation

**`*row_done`** marks a row as processed and ready for printing, this is set by workers and read by the printer

**`next_y`** used to denote the next row to be processed by the next free worker, incremented by all workers.

Clearly, `*row_done` and `next_y` are prone to race conditions. This is resolved by using a mutex. Since both varibles are modified together when a worker is done with a row, a single common mutex is sufficient. This mutex is also used by the printer to ensure that no updates are done on `*row_done` while rows are checked for completion.

## A printer's daily routine

The printer goes through rows from top to bottom. Once going the printer will print row by row until an unfinished row is encountered, then using a condition variable it will wait on a signal. Every time a worker is done with any row, it will send a signal to wake up the printer. Using this strategy the printer is almost entirely separated from the workers.

### Worker works workingly

The work for the worker is made up of four distict tasks.

1. Pick first unprocessed row
2. For every complex number in the row
    a. Run Newton's method
    b. If converged to a root, store number of iterations, identify and store the root
    c. Else, store special values for both the number of iterations and the root
3. Mark row as done and wake up the printer
4. If there are unprocessed rows, return to 1.

## Newton's method

Our goal is to find the roots of

$$f(z) = z^d - 1$$

using Newton's method from a given starting point $z_0 \in C$, where a step is given by

$$z_{n+1} = z_n - \frac{f(z)}{f'(z)}.$$

However as written this is numerically unstable because both $f$ and $f'$ are powers of $\sim d$ and can become extremely large for large $d$. Because of this we use the following simplification

$$z_{n+1} = \frac{z_n}{d}(d - 1 - \frac{1}{z_n^d}).$$

### Test for convergence

For this task, $z$ is said to have converged if $\exists\, z^* : f[z^*] = 0$ and

$$|z - z^*| \leq \epsilon.$$

Applying this criterion directly is slow and requires knowledge of the roots of the function. A more natural and general way is to check if $z^*$ solves $f(z) = 0$ (within some margin). Assuming $z$ is close to a root, e.g. $z = z^* + \Delta$, $|\Delta| < \epsilon$, we can make a Taylor expansion centered at the root

$$\begin{aligned} f(z) &= f(z^* + \Delta) \\ &= f(z^*) + \Delta f'(z^*) + ... \\ &= 0 + \Delta d[z^*]^{d-1} + O((d\epsilon)^2) \end{aligned}$$

Considering the modulus of this expression we get

$$\begin{aligned} |f(z)| &= d|\Delta||z^*|^{d-1} + O((d\epsilon)^2) \\ \{|z^*| = 1\} &= d|\Delta| + O((d\epsilon)^2), \end{aligned}$$

which means that for any $z$ at most $\epsilon$ from a root $z^*$ we have

$$|z - z^*| = \frac{|f(z)|}{d} + O(\epsilon^2 d).$$

The important thing to notice here is that $f(z)$ is both simple to calculate and does not depend on explicit knowledge of $z^*$. Because of this it is natural to define the approximate criterion, where $z$ is said to have converged if

$$\frac{|f(z)|}{d} < \epsilon.$$

For this criterion to be equivalent to the naive criterion we need $d\epsilon^2 \ll \epsilon$. For the cases considered here $d\epsilon \sim 0.01$, so this approximate criterion is quite accurate. Considering that Newton's method has quadratic convergence and converges very quickly close to roots, this criterion is at most one iteration off. This error is very rare and the speedup from using the modified criterion is huge.

# Optimizations

Here we will cover the most important optimizations and cleverness used for this task. Our general approach is to first make a plan, write a *working program* and *then* optimize. Getting the structure correct avoids sad surprises along the way, getting the correct output is essential so that we know that further optimizations actually work.

As an example of this is this use of the complex type introduced in C99. Writing the code with the complex type was straightforward and not very prone to errors. When everything worked we started optimizing with a custom exponentialtion algorithm and even dropping the complex type altogether, all while continouosly checking that the output was still correct. Doing optimization along the way also means that it's easy to see if an optimization is actually worth doing.

## Complex calculations

Initial calculations were done entirely with the complex type introduced in C99. Moving away from this and doing calculations manually on the real and imaginary part reduces runtime significantly ($\sim 50\%$).

### Exponentiation

Exponentiation using `cpow` in `complex.h` is very general, and while technically $O(1)$ the MinGW implementation requires conversion to and from polar form (`sqrt`, `sin`, `cos`), an `atan` a `log` and an `exp`. In short very expensive operations.

Since we're restricted to integer exponents we can improve on this. In this case exponentiation by squaring is used, which scales as $O(\log d)$, where $d$ is the exponent. While worse asymptotically, this method uses only additions, bitshifts and multiplications which makes it much faster for our range of exponents.

## Lazy root identification

Because of the modified convergence criterion we don't know which root we converge to and must check this after applying Newton's method. Finding the exact root is simple, but uses `atan2` which is rather slow. Luckily, through some cleverness this can often be avoided.

Looking at the attractor plots it's clear that adjacent points often converge to the same root. So, if we recall the last root found we can make a quick check if we converged to the same root again. This can be tested by $|z - z_{\text{last}}| < 2\epsilon \because |z - z_{\text{last}}| \leq |z^* - z| + |z^* - z_{\text{last}}| < \epsilon + \epsilon$, where $z^*$ is the exact root.

Because of this it's reasonable to make a quick test to determine if we converged to the same root as the last evaluated point. If this is true the `atan2` call is not needed. Using our techniques we go from `atan2` having a large computational cost to being negligible (confirmed by profiling).

## Using symmetry

The given problem is almost perfectly symmetric. The input range is centered at the origin, the calculations made for $z$ is the same as for $\bar{z}$, only conjugated. Using this we only need to perform Newton's method on the upper half-plane. Then for the lower half-plane the same results can be reused, with the only difference that the roots must be conjugated.

An issue with this approach is that the approach of freeing memory as rows are printed does not work well here because we don't reach the symmetry rows until much later on. Without using symmetry the memory usage is negligible assuming the printer isn't the bottleneck (true unless very many worker threads). This is a typical time/space trade-off.

## Fast output

Using plain `printf` is convenient but slow for large volumes of data, using 10 threads for computations the `printf` version couldn't keep up. Manually writing integers to a buffer and then writing the buffer to file speeds of write speeds up things significantly, to the point where the printing is quicker than computations even for a large number of threads.

This improvement is even more important after using symmetry. Raw write speed for this method is about 500 Mib/s, and during calculations with 10 threads just over 200 MiB/s is written, i.e. the I/O is not the bottleneck during calculations.

The most significant improvement on top of this would be to do away with this ungodly verbose format, just writing a 16 bit block of values would be a huge improvement.

## Possible further optimizations

Note that we have opted to not hardcode any cases, although this would have been very easy for low exponents. We opted for writing code for the general case, with good numerical stability (both in choice of data type and computations) and with good structure. More care was put into algorithm choices and the mathematical approach than low level optimizations (other than data locality and avoiding branching when possible).

From what we can see there are no obvious optimizations that would give huge benefits unless we approach large $d$. Our exponentiation code has time complexity $O(log(d))$ but doesn't use the (very expensive, but constant time) complex logarithm. For large $d$ the constant time approach might be preferable, but in such cases the numerical stability is a larger problem than time complexity.