Applied Math 205: Introduction to Compiled Languages - Exercise

Michael S. Emanuel

Assigned: 09-Sep-2021

Instructions: Please complete ONE of the following two exercises, either **Calculating π** or **Newton Fractal**. Please make sure to submit each requested item as an individual file (eg .pdf, .txt, .hh or .cc) on Canvas. This allows us to grade the submissions natively on Canvas and provide feedback easily. But if you submit a zip archive, we can't communicate our feedback using Canvas.

Calculating π

1. In Lecture 1, we saw the Machin formula for π, [1]

$$\frac{\pi}{4} = 4\tan^{-1}\left(\frac{1}{5}\right) - \tan^{-1}\left(\frac{1}{239}\right) \tag{1}$$

In 1706, Machin used this formula to correctly calculate the first 100 digits of pi. He evaluated the arctangent using the series published by James Gregory,

$$\tan^{-1} x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \dots \tag{2}$$

In this exercise, you will recreate Machin's calculation with a modern C++ program. As a warm-up, calculate π in double precision using the Machin formula (1) and the C++ library function `atan`, which is declared in the `<cmath>` header. Compare this result to the "official answer" in C++, which is available as `std::numbers::pi`.

Does your calculation match to full double precision? If not, what is the absolute error?

Once you have completed the warm-up, you are ready for the main event. Use the Machin formula to calculate the first 100 digits of pi, following in Machin's footsteps, only faster. A significant challenge is that double precision arithmetic only has about 16 significant digits. so you will probably need some kind of extended precision data type. One accessible approach is to use the GNU Mutiple Precision Library GMP.[2] You can read the documentation online at https://gmplib.org/. On a Linux system using the apt package manager such as Ubuntu, you can install this library in one line by typing

```
$ sudo apt install libgmp-dev
```

in the terminal. To use the library in your program, you will need to include the header file like this: `#include <gmp.h>` [3] Once you get the compiler to find the GMP headers, you will also need to tell `g++`

[1] Here is a derivation for the Machin formula using complex numbers that I find more intuitive than the traditional approach with trigonometric identities:

$$(5+i)^4 \cdot (239-i) = (476+480i)(239-i) = 2^2 \cdot 13^4 \cdot (1+i)$$

Now take logs of both sides, and equate the imaginary parts. $\mathrm{Im}\left(\log(5+i)\right) = \tan^{-1}\left(\frac{1}{5}\right)$ and $\mathrm{Im}\left(\log(239-i)\right) = -\tan^{-1}\left(\frac{1}{239}\right)$. Of course $\mathrm{Im}\left(\log(1+i)\right) = \frac{\pi}{4}$. Making these three substitutions, the Machin formula follows.

[2] Using GMP is not required for this problem. You can use any library you like, or code everything yourself from scratch.

[3] Remember, when you issue an include preprocessor directive, enclosing the name of the header file in angle brackets tells the compiler to search for it on an implementation-defined search path that will include the standard library files. On my Ubuntu Linux system, the gmp headers are installed in `/usr/include/gmpxx.h` and `g++` is smart enough to find it there when I use the angle-bracket syntax. If you have trouble getting this to work, you can manually drop the header file somewhere, e.g. in a directory under your home directory, and pass an -I flag to the compiler. An even simpler expedient is to put a copy of the header file in the same folder as your C++ source, and include it using double quotes `''gmpxx.h''` rather than angle brackets.

to link to this library by including the argument `-lgmp` in the linkage command. Because compiling and linking the program may be the most painful part of the assignment for students new to C++, I include below example calls that work on my vanilla Ubuntu installation:

```
g++ -c src/pi.cc -o obj/pi.o -std=c++20 -Wall -Wextra -Werror -O3
g++ -o exec/pi.x obj/pi.o -lfmt -lgmp
```

The first command *compiles* the source file `src/pi.cc` into an object file `obj/pi.o`. The second command *links* the object file `obj/pi.o` into an executable `exec/pi.x`; the last two arguments tell the linker to include the format library `fmt` (for the print function) and `gmp`(for multilple precision math).

2. Now that you have selected an extended precision library and figured out how to compile and link against it, you are ready to start on the substantive part. Do a simple pencil and paper calculation to get a lower bound on the number of terms you need to evaluate in the series for both arctangent terms for 100 digits of precision. This bound doesn't need to be particularly sharp; it's more important to get the digits right than to get the lowest possible bound.
How many terms in the series for $\tan^{-1}\left(\frac{1}{5}\right)$ and $\tan^{-1}\left(\frac{1}{239}\right)$ are required for 100 decimal places of accuracy?

3. **Calculate the first 100 digits of pi using the required number of terms.**
You can find someone else's calculation of the first 100 digits at math.com.
Compare your answer to the published one. How many digits do you agree on?

Report for Calculating π

1. A short report (about one page) as a PDF docuement with your answers to questions (1), (2) and (3) above. Include in your report a brief discussion of how you organized your program and any notable implementation decisions.

2. A copy of all the source code files (both headers `file.hh` and implementations `file.cc`). If Canvas complains about accepting files with .cc or .hh suffixes, you can name them e.g. file.cc.txt.

3. A file named `pi.txt` which is output by your program. This file should contain the decimal expansion you calculated, in the format "3.141592654..." and have at least 100 digits after the decimal point. (You can include as many as you like, up to a million.)

Newton Fractal[4]

As we will cover soon in the course, Newton's method, also known as the Newton-Raphson method, is widely used in finding the roots of a function. Assume we have an initial guess, x_0, and it's close to a root, then function $f(x)$ can be approximated by its first order Taylor expansion,

$$f(x_0 + \Delta x) \approx f(x_0) + f'(x_0)\Delta x$$

If this approximation is good, and we set $f(x_0 + \Delta x) = 0$, we can solve for the location of the root,

$$\Delta x = -\frac{f(x_0)}{f'(x_0)};\ x_r = x_0 + \Delta x$$

However, in actuality, the approximation may not be good enough, but the above reasoning gives us an iteration rule, or a recurrence relation,

$$x_1 = x_0 + \Delta x = x_0 - \frac{f(x_0)}{f'(x_0)}$$

or in greater generality,

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{3}$$

[4]Problem created by Yuexia Luna Lin, Applied Math 205, Fall 2020

There are some caveats to this rule, especially in tricky cases where the assumptions of (1) initial guess being close to a root (2) the function having a continuous, non-zero first derivative, are violated. Usually, if a division by zero happens, we terminate the iterations, change the initial guess and try again. Sometimes, the algorithm can get stuck in a cycle.[5] In practice, we set a threshold of error tolerance, ϵ, such as the algorithm is terminated if

$$|x_{n+1} - x_n| < \epsilon \tag{4}$$

is reached. Besides that, we also impose a limit on the number of maximum iterations the algorithm can take, so that we can terminate the algorithm gracefully when it has failed to converge.

A good initial guess can help Newton's method converge very rapidly, a bad one would make it stuck in a cycle forever or diverge. But there isn't a simple rule to discern which initial guess would converge to which root. This is where the plot thickens.

We know that polynomials can have complex roots. Newton's method can be adapted to find the roots of polynomial $p(z)$ of complex argument z in the complex plane. The regions that converge to a particular root are called the *basins of attraction* for that root. One might expect there to be a clean cut boundary dividing these regions, because we may think that a point closer to a root would converge to that root. Not so! The boundaries between basins of attractions, in polynomials of degree > 2, are **fractal**.

If we color the basin of attraction for a root in the complex plane by the same color, we will see some amazing patterns. Figure 1 shows such an example from *Wikipedia* for polynomial $p(z) = z^3 - 1$, whose roots are $1, -\frac{1}{2} \pm \frac{\sqrt{3}}{2}i$. The basins of attraction for 1 are colored shades of yellow, those for $-\frac{1}{2} + \frac{\sqrt{3}}{2}$ are colored shades of magenta, and those for $-\frac{1}{2} - \frac{\sqrt{3}}{2}$ are colored shade of cyan. We indeed see that in the boundaries between the large basins of attraction have fractal structure[6]. Patterns like these are called the Newton fractals.

Report for Newton Fractal

1. Your first exercise is to reproduce Figure 1 using the complex version of Newton's method[7],

$$z_{n+1} = z_n - \frac{p(z_n)}{p'(z_n)} \tag{5}$$

for polynomial $p(z) = z^3 - 1$. The first derivative[8] , $p'(z) = 3z^2$. To compute these quantities in the complex plane, use the `complex<double>` type in the STL library. (You will find example `complex_num.cc` in the GitHub repo[9] on how to use this data type.)

Discretize a part of the complex plane, $[-L, L] \times [-iL, iL]$, with $N \times N$ grid points; the grid spacing is $\Delta h = \frac{2L}{N-1}$. Define $z_{00} = -L - iL$, so the complex number of a grid point indexed with (m, n) is $z_{mn} = z_{00} + m\Delta h + in\Delta h$, for $m, n \in [0, N-1]$.

Perform iterations defined by Eqation 5 using each grid point as an initial guess, and keep track which root this guess converges to, e.g. you can use 0 to indicate failure to converge, 1, 2, and 3 as root markers. Make sure to set two termination conditions, one checking that the iteration has converged (Equation 4) and the other limit the maximum number of iteration allowed.

To make sure the resolution of your image is good, you should use $dh < 0.005$. Use your preferred data plotting software to visualize the end results. For example, in python, you can use `matplotlib.pyplot.imshow`. A command line plotting tool called `Gnuplot` is also very useful. You can beautify the figure by using some nice palettes. (You will find `gnuplot_example.gp` on the GitHub repo with codes to plot a sample dataset with a nice purple palette in `Gnuplot`.)

2. Now that you have the code working to create a Newton fractal for $p(z) = z^3 - 1$, choose another polynomial, and see what Newton fractal you can create!

[5] Just try apply Newton's method to $f(x) = x^3 - 2x + 2$, with initial guess $x_1 = 1$.
[6] Visit http://usefuljs.net/fractals/docs/newtonian_fractals.html for a fun read, and an online fractal generator!
[7] See https://en.wikipedia.org/wiki/Newton_fractal for details.
[8] This function is analytic in the context of complex analysis. Its derivative can be written as $p(z) = \frac{dp(z)}{dz}$.
[9] https://github.com/chr1shr/am205_g_activities/compiled_lang.git

3. Submit a one-page write-up along with your **code**. The write-up should

    - Include a plot of your reproduction of Figure 1 in a palette different from the original. Report your parameters, L and N, and the total time of computation.
    - Present the polynomial of degree > 2 of your choosing. Include a plot of your very own Newton fractal. Report your parameters, L and N, and the total time of computation.

Good luck and have fun!

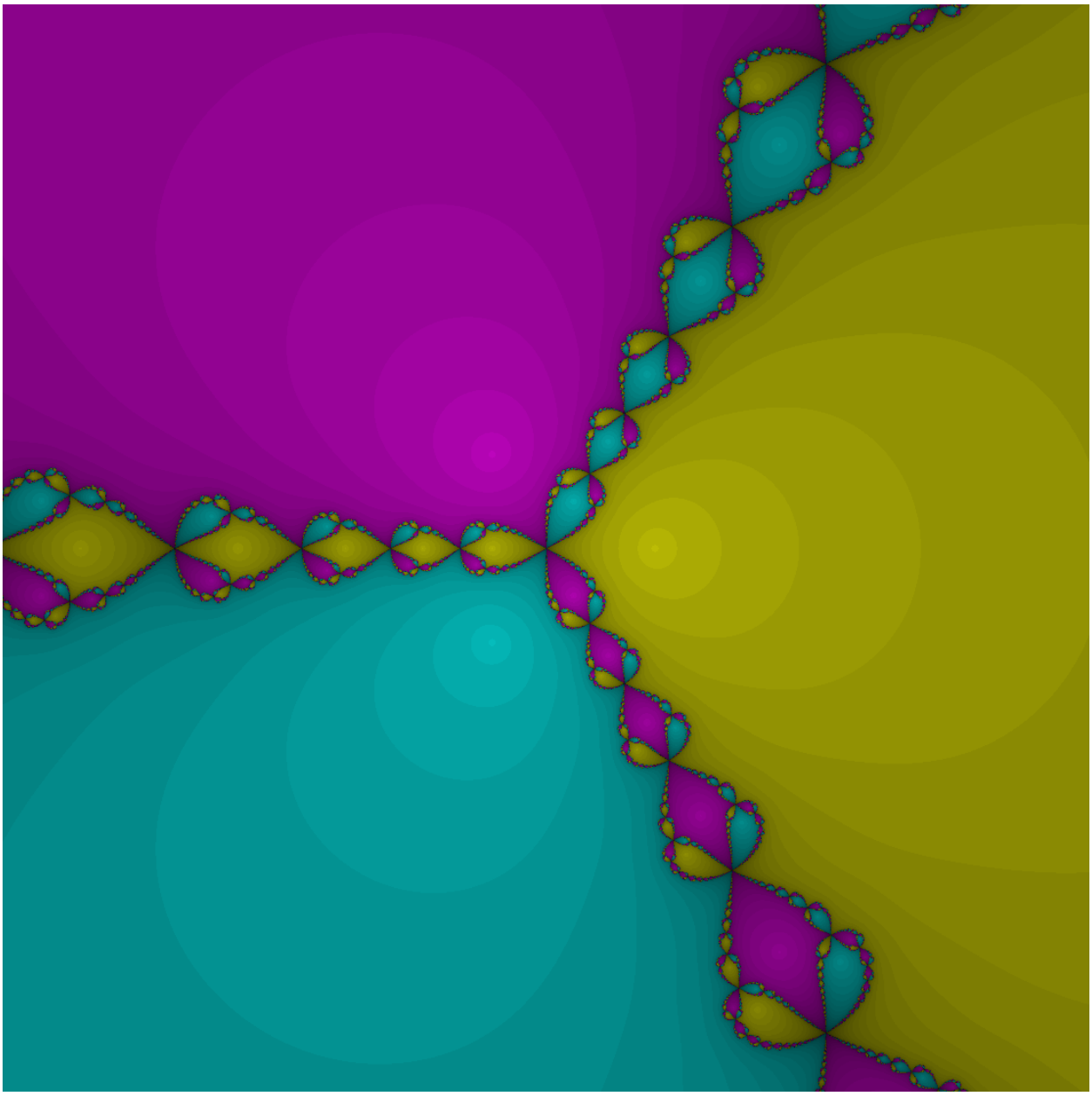

Figure 1: Newton fractal for complex polynomial $p(z) = z^3 - 1$. Credit: Wikipedia user LutzL.