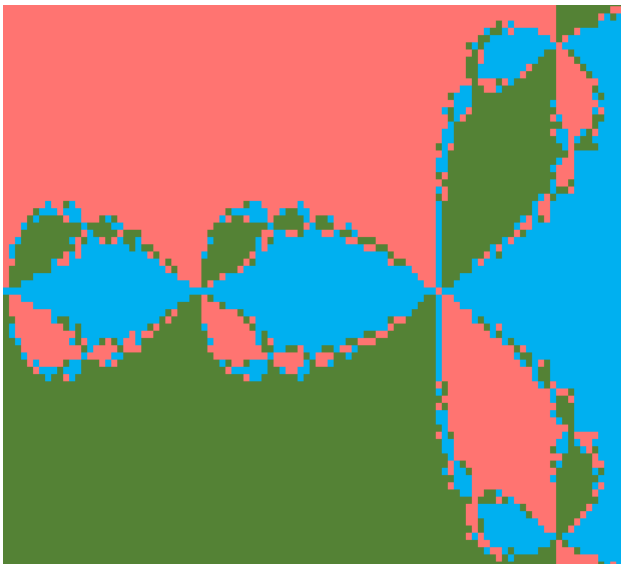
Project 3: Basins of Attraction in Complex Plane
Write original code to solve each the following programming project. Be sure to use proper style and to document

You are welcome to work on this assignment in pairs.
Due Wednesday, April 10th at the start of class.

1 Babylonian Basins of Attraction for $Z^n = 1$



The Babylonian algorithm iterates

$$x_{n+1} \leftarrow \frac{x_n + A/x_n}{2}$$

for convergence to $\sqrt{A}$. This can be generalized to cube roots and so on using the iterative formula

$$x_{n+1} \leftarrow \frac{(k-1)x_n + A/x_n^{k-1}}{k}$$

for convergence to a kth root of A.

In the complex plane, the kth roots unity ($\sqrt[k]{1}$) are evenly distributed around the unit circle. For example, if z is a complex number then $z^3 = 1$ has three different solutions which can be found algebraically by solving

$$\begin{aligned} z^3 &= 1 \\ z^3 - 1 &= 0 \\ (z-1)(z^2+z+1) &= 0 \end{aligned}$$

So either $z = 1$ or

$$\begin{aligned} z^2 + z + 1 &= 0 \\ \left(z + \frac{1}{2}\right)^2 + \frac{3}{4} &= 0 \\ z &= -\frac{1}{2} \pm \frac{\sqrt{3}}{2} i \end{aligned}$$

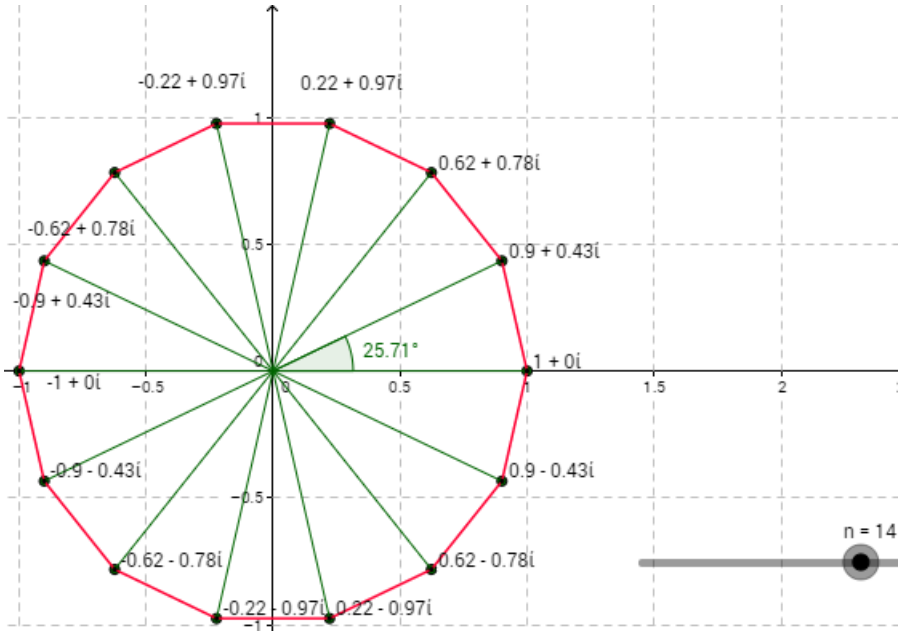
More generally, we take the polar form of the complex number, $z = r \cdot e^{i\theta}$ and since we're on the unit circle, $r = 1$ and our equation becomes

$$z^n = 1 \Leftrightarrow \left(e^{i\theta}\right)^n = 1$$

and using de Moivre's formula,

$$\left(e^{i\theta}\right)^n = (\cos\theta + i\sin\theta)^n = \cos(n\theta) + i\sin(n\theta) = 1$$

Thus we have $n\theta = 0 + 2\pi j, j = 0, 1, \ldots, n-1$, evenly spaced angles around the unit circle.
At the web site https://www.geogebra.org/m/67393 : Roots of Unity there's a geogebra utility for showing how the roots of unity a distributed around the unit circle for various roots. For fourteenth roots the picture looks like this:



The following code associates a number 1, 2 or 3 with each point in a grid of points from the rectangle $[-2,-2]\times[2,2]$ in the complex plane, depending on which cube root of i: $\frac{\sqrt{3}}{2}i + \frac{1}{2}$, $\frac{-\sqrt{3}}{2} + \frac{1}{2}$, or $-i$ (you can verify these are cube roots of i by cubing them. Note that the are evenly distributed around the unit circle in the complex plane.

```
// GH: babylonian basins of attraction

#include <iostream>
#include <complex>
#include <vector>
#include <cmath>
#include <fstream>
using namespace std;

complex<double> cubeRoot(complex<double>, complex<double>);
void seeOut(vector<vector<unsigned> >, ofstream&);

int main() {
    /// set up file for writing
    ofstream out("roots.txt");
    if(!out) cerr << "too bad, no file.";

    ///roots numbered 1,2,3 go here
    vector<vector<unsigned> > rootGrid;
    vector<unsigned> column; ///a member of rootGrid

    complex<double> initial{0,0}, converge, A{0,1};

    /// The three complex cube roots of i:
```

```
    const complex<double> root1{sqrt(3.)/2,0.5};
    const complex<double> root2{-sqrt(3.)/2,0.5};
    const complex<double> root3{0,-1};
    ///for each column (starting at upper left corner)
    for(double x = -2; x <= 2; x += 0.02) {
        /// for each row
        column.clear();
        for(double y = 2; y >= -2; y -= 0.02) {
            initial.real(x); initial.imag(y);
            converge = cubeRoot(A,initial);
            if(abs(converge-root1)<0.1) column.push_back(1);
            else if(abs(converge-root2)<0.1) column.push_back(2);
            else column.push_back(3);
        }
        rootGrid.push_back(column);
    }
    seeOut(rootGrid, out);
}

complex<double> cubeRoot(complex<double> A, complex<double> temp) {
    complex<double> two{2,0};
    complex<double> three{3,0};
    ///complex<double> temp{A.real()/3,A.imag()/3};
    complex<double> next{0,0};
    next = (two*temp+A/(temp*temp))/three;
    while(abs(next-temp)>abs(A)*1e-10) {
        temp = next;
        next = (two*temp+A/(temp*temp))/three;
    }
    return next;
}

void seeOut(vector<vector<unsigned> > v, ofstream& out) {
    //vector<unsigned> col;
    for(int i = 0; i < v.size(); ++i) {
        //col = v[i];
        for(int j = 0; j < v[i].size(); ++j) {
            //cout << v[i][j];
            out << v[i][j] << "␣";
        }
        //cout << endl;
        out << endl;
    }
}
```

1. Write your own version of the class for complex numbers and call it Complx and put the interface and implementation for the class in a separate header file, `Complx.h`. It should provide at least the following features:

    1 A constructor and a default constructor.

    2 Overloaded `+,-,*,/` operators that will take as a `rhs` operand either a `Complx` or a `double`.

    3 Get and set functions overloaded with the same name: `real()` or `imag()` where if a double is passed it will return void and set the real or imaginary part, and is no argument is passed it will return the real or imaginary part.

    4 An `abs()` function which will return the modulus (distance from the origin) of a complex number.

5 Helper functions to overload the input and output stream operators: `<<` and `>>`.

Also, any other functions that you may think relevant and useful, like, say, the `conjugate()` function, or whatever. Get the above code to work with your `Complx` class.

Write at least one test program to verify your class is correct. If you just answer the questions below with one or more source files (.cpp) files giving the implementation of the interface in the the header file and the required functionality for the question. Ideally, you could put them all together in one menu-driven program, but that's not required.

Submit your `Complx.h` file and one or more source files for the questions below with the format `<your initials>_fracta` where `i=1,2,3,4` for the various questions. Or use a menu-driven approach to answer all the questions with a single, multi-featured program.

2. Modify the code to allow the user to choose a different rectangle and the number of points to sample. Say, allow the user to input `xmin, xmax, ymin, ymax` of a rectangular region in the complex plane with corners `[xmin, ymin]`, and `[xmax, ymax]` and the number of rows and columns the grid should be divided into. This will give the user the ability to "zoom" in on regions of the plane, and provide a level of resolution in sampling precision.

3. Implement your code in SFML to visualize the basins of attrations.

4. Change the `cuberoot()` function to allow the user to choose k for the kth root of a user-supplied number. It would be appropriate to change the name of the function to something, "`kthRoot()`".

5. Change the iterative scheme from the Babylonian algorithm to an algorithm that iterates $z_{n+1} \leftarrow z_n^2 + c$ for some initial complex constant c. So the iterates will be $c, c^2 + c, (c^2 + c)^2 + c, ((c^2 + c)^2 + c)^2 + c$ and so on. Check for divergence or convergence. If the iterations diverge to infinity label the point 8. If the iterates converge, label the point 0. If they converge to some non-zero value, label them something else.