MA117 Project 2: Root Finding

Administrative Details

- This project is the second of the three assignments required for the assessment in this course. It is to be submitted by **Noon, Monday 16th March 2020**. Details of the method of the submission via the Tabula system have been described in the lecture notes and are also available on the course web page.
- This assignment will count for **40%** of your total grade in the course.
- The automated submission system requires that you closely follow instructions about the format of certain files; failure to do so will result in the severe loss of points in this assessment.
- You may work on the assignment during the lab session, provided you have completed the other tasks that have been set. You can use the work areas at all times when they are not booked for teaching, 7 days per week. If you are working on the assignment on your home system you are advised to make regular back-up copies (for example by transferring the files to the University systems). You should note that **no** allowance will be made for domestic disasters involving your own computer system. You should make sure well ahead of the deadline that you are able to transfer all necessary files to the University system and that it works there as well.
- The Tabula system will be open for the submission of this assignment starting from **1st March 2018**. You will not be able to test your code for correctness using Tabula but you can resubmit your work several times, until the deadline, if you find a mistake after your submission. A later submission always replaces the older one, but you have to re-submit **all** files.
- Remember that all work you submit should be **your own work**. Do not be tempted to copy work; this assignment is not meant to be a team exercise. There are both human and automated techniques to detect pieces of the code which have been copied from others. If you are stuck then ask for assistance in the lab sessions. TAs will not complete the exercise for you but they will help if you do not understand the problem, are confused by an error message, need advice on how to debug the code, require further explanation of a feature of Java or similar matters.
- If you have more general or administrative problems e-mail me immediately. Always include the course number (MA117) in the subject of your e-mail.

1 Formulation of the Problem

Finding the roots of a function is a classical and extremely well-known problem which is important in many branches of mathematics. In Analysis II, you have probably seen that it is often easy to prove results about the existence of roots. For example, using the Intermediate Value Theorem, you should easily be able to prove that the function $f(x) = x - \cos x$ has a root in the interval $[0, 1]$. On the other hand, calculating an exact value for this root is impossible, since the equation is transcendental.

Root finding is a classic computational mathematical problem, and as such there are many algorithms which one may use to approximate the roots of a function. In this project, you will write a program which uses the *Newton-Raphson* algorithm. Let $f : \mathbb{C} \to \mathbb{C}$ be continuously differentiable, and pick a point $z_0 \in \mathbb{C}$. Consider the sequence of complex numbers $\{z_n\}_{n=0}^{\infty}$ generated by the difference relation

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

Typically, if z_n converges and $\lim_{n\to\infty} z_n =: z_*$, then $f(z_*) = 0$.

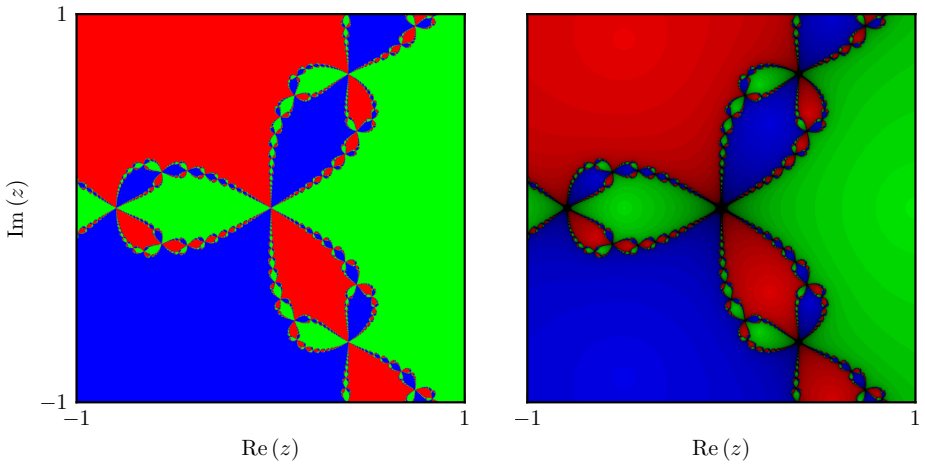

Figure 1: An example of Newton fractals for the function $f(z) = z^3 - 1$ in the square with bottom left-corner $-1 - i$ and width 2.

In general then, to apply Newton-Raphson, one must know the derivative of f. Whilst there are numerical tricks to accomplish this, this problem is somewhat beyond the scope of this course. Instead then, you will consider a polynomial $P \in \mathbb{C}[z]$; i.e.

$$P(z) = a_0 + a_1 z + a_2 z^2 + \cdots + a_n z^n$$

where $a_k \in \mathbb{C}$. P has a (hopefully obvious!) exact derivative.

1.1 Newton Fractals

One of the most fascinating aspects of this problem arises from a very simple question: given a starting position $z_0 \in \mathbb{C}$, which root does the sequence produced by Newton-Raphson converge towards? It turns out that the answer to this question is very hard!

Figure 1 shows two examples of how we might visualise this for the polynomial $f(z) = z^3 - 1$. Recall that the roots of this polynomial are $\alpha_k = e^{2\pi i k/3}$ for $k = 1, 2, 3$ (i.e. the third roots of unity). Each of the three colours represents one of these roots. In the left-hand figure, we colour each point depending on which root the method converges to. The right-hand figure is the same, asides from the fact that we make the colour darker as the number of iterations it takes to get to the root within a tolerance ε increases. The resulting images are examples of *fractals*, which you have undoubtedly seen before.

Don't worry if all of this seems quite difficult – the main aim of the assignment is for you to successfully implement the Newton-Raphson scheme. Most of the code to deal with drawing and writing the images will be given to you.

1.2 Summary

Your task then involves several distinct elements. You will:

- write a class to represent complex numbers;
- write a class to represent a polynomial in $\mathbb{C}[z]$;
- implement the Newton-Raphson method to find the roots of the polynomial;
- investigate some interesting fractals and draw some pictures!

2 Programming instructions, classes in your code and hints

On the course web page for the project you will find the following files, which should serve as templates and should help you to start with the project. As with the previous projects, the files have some predefined methods that are either complete or come with predefined names and parameters. You **must** keep all names of `public` objects and methods as they are in the templates. Other methods have to be filled in and it is up to you to design them properly. The files defines three basic classes for your project:

- `Complex.java`: represents points $z \in \mathbb{C}$;
- `Polynomial.java`: represents polynomials in $\mathbb{C}[z]$;
- `Newton.java`: given an initial `Complex` point z_0 calculates the corresponding root of `Polynomial` by Newton-Raphson, if possible;
- `NewtonFractal.java`: will generate a fractal similar to the one pictured above in a square.

These classes are documented in more detail in the following sections. You should complete them in the order of the following sections, making sure to carefully test each one with a `main` function.

2.1 `Complex`

`Complex` is the simplest of the classes you will need to implement, and will represent complex numbers. In fact, it bears a striking resemblence to the `CmplxNum` class you (hopefully) implemented in week 14. They are **not identical** however, so you should carefully copy and paste your code into this new class.

2.2 `Polynomial`

The `Polynomial` class is designed to represent a polynomial $P(z) = \sum_{n=0}^{N} a_n z^n$. As such, it contains `coeff`, an array of `Complex` coefficients which define it. It is assumed that `coeff[0]` corresponds to a_0, `coeff[1]` to a_1 and soforth. To complete this class, you will have to:

1. Define appropriate constructors. There are two that need implementation; a default constructor which initialises the polynomial to the zero polynomial (i.e. $a_0 = 0$), and a more general constructor which is passed an array of `Complex` numbers $\{a_0, a_1, \ldots, a_N\}$ which should be copied into `coeff`. In addition, you should ensure that if any of the leading co-efficients are zero then they are *not* copied. For example, if the constructor is passed the complex numbers $\{a_0, a_1, 0, 0\}$ then it should copy $\{a_0, a_1\}$ to `coeff`. (When testing for equality to zero, do not use any tolerances.)
2. Return the degree of the polynomial. Recall that $\deg f = N$.
3. Evaluate the polynomial at any given point $z \in \mathbb{C}$. Note that you **should not** implement a `pow` function inside `Complex` as it is unnecessary and inefficient. Instead, notice that (for example)

$$P(z) = a_0 + a_1 z + a_2 z^2 + a_3 z^3 = a_0 + z(a_1 + z(a_2 + z a_3)).$$

2.3 `Newton`

This class will perform the Newton-Raphson algorithm. There are two constants defined in this class:

- `MAXITER`: the maximum number of iterations to make; that is, you should generate the sequence z_n for $0 \leq n \leq$ `MAXITER` and no more.
- `TOL`: At each stage of the Newton-Raphson algorithm, we must test whether a sequence converges to a limit. In this project, we will say that z_M approximates this limit if, at any stage of the

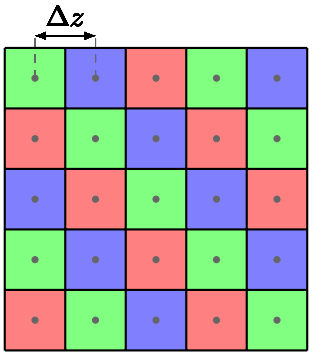

Figure 2: A 5x5 pixel image representing the square with top left corner $-1+i$ and bottom right corner $1-i$. The center of each pixel represents a complex number on the plane.

algorithm, $|z_M - z_{M-1}| <$ `TOL`. We then say that the starting point z_0 required M iterations to converge to the root.

Additionally, you will need to define the `iterate` function. This accepts a single parameter, z_0, which defines the initial condition of the Newton-Raphson difference relation, and performs the root finding algorithm. There are three things that can occur during this process:

- everything is fine and we converge to a root;
- the derivative $f'(z_k)$ goes to zero during the algorithm;
- we reach `MAXITER` iterations.

If any of the last two cases occur, then you set the error flag `err` to be -1 and -2 respectively; otherwise, `err` is set to zero. Here is a quick example of how `Newton` should be used:

```
Complex[] coeff = new Complex[] { new Complex(-1.0,0.0), new Complex(),
                                  new Complex(), new Complex(1.0,0.0) };
Polynomial p    = new Polynomial(coeff);
Newton     n    = new Newton(p);
n.iterate(new Complex(1.0, 1.0));
System.out.println(n.getRoot());
```

This will print out the root of $f(z) = z^3 - 1$ obtained with the starting point $z_0 = 1 + i$.

2.4 `NewtonFractal`

`NewtonFractal` will be responsible for drawing images of the fractals we saw in figure 1. However, let us briefly consider how images are represented on computer first. A two-dimensional image is, in general, broken down into small squares called *pixels*. Each of these is given a colour, and there are generally many hundreds of pixels comprising the width and height of the image.

An example of this can be seen in figure 2. This image (badly) represents a square in the complex plane with top-left corner $-1+i$ and bottom-right corner $1-i$. In `NewtonFractal` you will generalise this concept to visualise squares with a top-left corner `origin` and width `width`, stored as instance variables inside `NewtonFractal`. The image will be of size `NUMPIXELS` by `NUMPIXELS`. Each pixel can be accessed by using an ordered pair (j, k) where j is the row number, k the column number and $(0, 0)$ is the top left pixel, with $0 \leq j, k <$ `NUMPIXELS`. The image itself will be generated using `createFractal`, which accepts a single argument `colorIterations`. When `true`, the function generates a figure like the right hand side of figure 1.

To complete the class, first ensure that you call the `setupFractal` function at the end of your constructor. This will initialise the more complex drawing objects. It also checks that the polynomial you have given it has $3 \leq \deg p \leq 5$. **You will not need to consider any other polynomials in this class.** Then inside `createFractal`, use the following logic:

1. Copy `colorIterations` to the instance variable.
2. Iterate over each pixel at position (j, k). Then translate this position to a complex number using `pixelToComplex`, which uses the simple mapping $(j, k) \mapsto$ `origin` $+ \Delta z(j - ik)$.
3. Run this complex number through the Newton-Raphson algorithm.
4. Check to see whether you've found this root already. You will store the list of already found roots inside the `ArrayList roots`. This is the purpose of the `findRoot` function. In this formulation, two complex numbers z_1 and z_2 are equal if $|z_1 - z_2| <$ `Newton.TOL`.
5. Finally, colour the pixel using the `colorPixel` function.

After you are done, you can save the image using `saveFractal`. Here is an example from start to finish, which creates the two images of figure 1. Note that your filename should end with `.png`:

```
NewtonFractal f = new NewtonFractal(p, new Complex(-1.0,1.0), 2.0);
f.createFractal(false);
f.saveFractal("fractal-light.png");
f.createFractal(true);
f.saveFractal("fractal-dark.png");
```

You should then create a document which is precisely one page long. In this document, pick a polynomial $P(z)$, a square in the complex plane and use your program to generate the two plots. You should call this file `Fractal.pdf` and **ensure it is saved as a PDF file**.

3 Submission

You should submit, using the Tabula system, the following four files: `Complex.java`, `Polynomial.java`, `Newton.java`, `NewtonFractal.java` and `Fractal.pdf` which contains your plots.

There will be a large number of tests performed on your classes. This should allow for some partial credit even if you don't manage to finish all tasks by the deadline. Each class will be tested individually so you may want to submit even a partially finished project. In each case, however, be certain that you submit Java files that compile without syntax error. Submissions that do not compile will be marked down.

If you submit a code and later (before the deadline) you realise that something is wrong and you want to correct it, you may do so. You can submit as many versions as you wish until the deadline. However, only the last submission will be tested. Each new submission replaces the previous one, so in particular, you **MUST** submit all files required for the project even if you corrected only a single file.

Keep back-up copies of your work. Lost data are not a valid excuse for missing the deadline.