Project 2: The Inverse Iteration Method for drawing Julia sets

Due: 11:59pm, Tuesday, April 28

Collaboration Policy: You can discuss topics related to the project with your classmates or others. You may also do research on the internet or from books (this is certainly not required). But you must design and write your own code and report. In particular, you should understand how your code works, and be able to justify your design choices.

Background

In this project, you will draw the Julia set of polynomials using a different algorithm than the one we used in class. Recall that the Julia set for a complex polynomial f is defined as

$$J(f) = \text{boundary of } \{z \in \mathbb{C} : \lim_{n\to\infty} f^{\circ n}(z) = \infty\}.$$

The algorithm we used in class (for both colored and uncolored versions) tried to directly draw the set $\{z \in \mathbb{C} : \lim_{n\to\infty} f^{\circ n}(z) = \infty\}$ by testing grid points z in a fine grid and computing higher and higher iterates of them. We will call this the *Boundary Scanning Method* (BSM). This worked fairly well, but it was hard to say for sure whether any given point z is in $J(f)$, since it might be the case that z does eventually escape to infinity, but only after wandering around for a high number of iterates. Occasionally the pictures produced by BSM can be very misleading.

The following gives an alternative description of the Julia that seems quite different:

Theorem 1. *Let f be a polynomial, and $z_0 \in J(f)$. Then*

$$J(f) = \overline{\{w : f^{\circ n}(w) = z_0 \text{ for some non-negative integer } n\}}.$$

In other words, if we already know some z_0 in the Julia set, we can find the whole Julia set by finding all points w that eventually get mapped to z_0 under some iterate of f. The line above the set in the theorem means that we take the *closure* of the set, i.e. we add in all points that can be approximated arbitrarily well by points in the set. For the purposes of drawing, taking the closure doesn't change anything. The proof of the above theorem is beyond the scope of this course, but we can nevertheless use it to get an algorithm for drawing.

The set of all inverse images in the above theorem is still (usually) infinite, but we can approximate them by finite sets. For N a large integer (and any $z_0 \in J(f)$), we have the approximation

$$J(f) \approx J_N(f) = \{w : f^{\circ n}(w) = z_0 \text{ for some non-negative integer } n \le N\}.$$

The set $J_N(f)$ on the right is always finite. The *Inverse Iteration Method* (IIM), which you will implement, involves drawing this set $J_N(f)$. One advantage of IIM is that the points that it draws

are definitely in the Julia set (on the other hand, a very large N will be needed to find some points). Compare this to BSM, which will sometimes draw points that are not actually in the Julia set.

Programming tasks [70 pts]

(1) `[15 pts]` To implement IIM, we must first find a point $z_0 \in J(f)$. You will use the following facts:

(i) For any polynomial f, if $f(p) = p$ and either $|f'(p)| > 1$ or $f'(p) = 1$, then p is in $J(f)$. Here $f'(p)$ is the derivative of f evaluated at p (a complex number).

(ii) For a polynomial $f_c(z) = z^2 + c$ from the quadratic family, such a point p satisfying the conditions in (i) always exists.

Find, by hand, such a p for the polynomial $f(z) = z^2 + 1/4$. Then write a Sage function to find such a p for any $f_c(z) = z^2 + c$. The function should take in c as an argument. (HINT: first use the quadratic formula to find p with $f_c(p) = p$; then test the condition on the derivative).

(2) `[20 pts]` Using the $z_0 = p \in J(f_c)$ you found in the previous part, implement IIM for the quadratic family by computing the set $J_N(f_c)$ defined above as a list of complex numbers. As the name Inverse Iteration Method suggests, you will find all inverse images of z_0 (i.e. all w with $f(w) = z_0$), and then all inverse images of those points, etc. Then plot this list using Sage's `list_plot` function or something similar.

(Note that f_c, being a quadratic polynomial, is not invertible, and z_0 will usually have two inverse images, i.e. two values of w with $f_c(w) = z_0$. Given z_0, one can compute the inverse images using a little algebra. Once you have done this, recursion is a good way to iterate the process of taking inverse images.)

(3) `[5 pts]` Draw a plot of $J(f_c)$ for $c = 0.25, i, -0.12 + 0.74i$ and one other c of your choice using your code from (2).

(4) `[10 pts]` Modify your IIM algorithm from (2) so that instead of producing a list, it produces a grid represented as an array (as we did for `julia` in class), where a grid square is assigned the value 1 if $J_N(f)$ contains some point in that grid square, and 0 otherwise. Then plot this grid using `matrix_plot` or something similar. This will produce an image with fewer "bunched together points".

(5) `[5 pts]` Draw a plot of $J(f_c)$ for $c = 0.25, i, -0.12 + 0.74i$ and one other c of your choice using your code from (4).

(6) `[15 pts]` Make one of the modifications/generalizations to your IIM code from the list below:

(a) Make your algorithm work for any *polynomial* f (not just those of the form $z^2 + c$). You will still use Theorem 1 and the sets $J_N(f)$, but the technique you used in (1) to find z_0 will need to be modified. You can use the fact that for any polynomial f, if there is some integer n with $f^{\circ n}(p) = p$ and $|(f^{\circ n})'(p)| > 1$, then p is in $J(f)$. It is also true that any polynomial has such a p with a fairly small n.

(b) Modify IIM so that it stops taking inverse images early for certain points. One of the issues with IIM is that, although Theorem 1 guarantees that the sets $J_N(f)$ will eventually get close to all points of $J(f)$, certain points will be "popular" (i.e. many points of $J_N(f)$ will come very close, even for small N), while other points of $J(f)$ will be "unpopular" (i.e it may take a large N for $J_N(f)$ to come close). The idea of this improvement is to stop taking iterated inverse images of some popular points, which will make the algorithm faster thus allowing you to use a higher N. One way to do this is to use your grid from (4) to keep track of the number of points among those from $J_N(f)$ that you have computed so far that land in that grid square. Then stop taking inverse images of a point that lands in a grid square that has already been hit too many times. You will have to decide how to define "too many times"; experiment with different choices.

(c) Use *random inverse images* to produce a better image in less time. This is meant to address the issue that some points in $J(f)$ will not be close to any points of $J_N(f)$ unless N is large. To implement this, in (2) above, choose one of the inverse images randomly, and ignore the other one. This will allow you to use a much larger N.

Written report [30 pts]

The report should include:

- A comparison of the various algorithms you implemented in (2), (4), (6), as well as the method we used in class. You should discuss qualitative differences in the images generated. You should also discuss differences in efficiency/run-time.
- A detailed written description of the modification/generalization you made for task (6).
- Any unexpected programming or math challenges you encountered doing the assignment.

What to turn in

You will submit everything on Blackboard. Include the following:

- A `.ipynb` notebook file named `code_firstname_lastname.ipynb` with all your code from the **Programming tasks** section. Make sure to clearly separate the various parts. Also, test this notebook by restarting it and running from the beginning; this should produce no errors.

- A `pdf` file named `report_firstname_lastname.pdf` with your **Written report**. The exact format of this report is up to you. It can be hand-written (legibly) and then scanned (legibly) into a `pdf` document. Or it can be be made on a computer. It must be in `pdf` form (it is generally easy to convert other forms such as .doc to `pdf`).

Hints/advice

- Recursion will often result in simple, elegant code. But for certain parts of the above, it may be more suitable to use a loop (particularly (6) part (c)).
- Particularly for (6) part (a), you may want to use some of the functions for dealing with polynomials included in the package `numpy`. For instance, `polyval` is useful for composing polynomials and `polyder` is good for taking derivatives. You can find good documentation online. You may also want to use the `roots` function, which numerically finds roots of polynomials.
- You can time a command by placing `%time` at the beginning of the line.